



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

LETICIA TRINIDAD VALDERAS RODRÍGUEZ
**A COMPILER FRAMEWORK FOR A COARSE-GRAINED RE-
CONFIGURABLE ARRAY**

Master of Science Thesis

Examiners: Prof. Jari Nurmi
Dr. Waqar Hussain
Examiners and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 6th May 2015

ABSTRACT

LETICIA TRINIDAD VALDERAS RODRÍGUEZ: A Compiler Framework for a Coarse-Grained Reconfigurable Array

Tampere University of Technology

Master of Science Thesis, 55 pages

June 2015

Master's Degree Programme in Information Technology

Major: Electronics Engineering

Examiner: Prof. Jari Nurmi

Dr. Waqar Hussain

Keywords: Compiler, CGRA, MVM, RPN, Place and Route, SCREMA, COFFEE RISC, CREMA

The number of transistors on a chip is increasing with time giving rise to multiple design challenges. In this context, reconfigurable architectures have emerged to provide high flexibility, less power/energy consumption yet while delivering high performance levels. The success of an embedded architecture depends on powerful compiler support. Current studies are focused on developing compilers to reduce the designer's effort by introducing many automation related features. In this thesis work, a compiler framework is presented for a scalable Coarse-Grained Reconfigurable Array (CGRA) called SCREMA.

The compiler framework presented in this thesis replaces the existing GUI compiler with an added feature of automatic placement and routing. The compiler receives a Reverse Polish Notation (RPN) description of the target algorithm by the user. It extracts the computational information from the RPN description and performs placement and routing over the CGRA template. The first configuration stream generated by the compiler is the main processing context. Furthermore, if additional configuration patterns have to be designed, the compiler framework gives the possibility to implement them in two different design paradigms: a preprocessing context and a canonical context. Preprocessing context is used to align the data into a CGRA to facilitate post-processing. Canonical context allows the user to perform additions in sum-of-products related algorithms.

The compiler framework has been tested by implementing real integer Matrix-Vector Multiplication (MVM) algorithms. Specifically, the tested MVM orders are 4th, 8th, 16th and 32nd on the CGRA sizes of 4x4, 4x8, 4x16 and 4x32 PEs, respectively. All the implementations are based on the RPN description of 4th-order MVM. Other than implementing 4th-order MVM, the rest of tested MVM algorithms need preprocessing and canonical contexts to be designed and implemented. The user effort which was needed to Place and Route (P&R) an algorithm manually on SCREMA is now reduced by using this compiler framework as it provides an automatic P&R mechanism.

PREFACE

The work presented in this thesis is conducted at the Department of Electronics and Communications Engineering, Tampere University of Technology, Finland.

I am deeply grateful to Dr. Waqar Hussain for proposing this interesting subject and for his consistent support, guidance and motivation during the implementation of this work. Sincere acknowledgement goes also to Prof. Jari Nurmi for accepting me and supervising this thesis. I thank M. Sc. Sajjad Nouri for helping me from the beginning to the end of this project.

I would like to express my deepest gratitude to my parents, Maria Josefa and Asterio for their constant support, patience and love. Without their strength and effort, I had not come until here. Special thanks to my sister and friend, Rebeca, for listening to me and providing me her advices. I thank to the rest of my family for believing in my possibilities and encouraging me in my whole life.

I would like to thank my best friends, Saez, Laura, Dakas, Castro, Andy y Pedro for their time, friendship, advices and support. I am also grateful to my Rolix girls, Conchi and Maria for all the relaxed days that have helped me to continue fighting to be here.

I am also thankful to my high school friends, Leti, Mire and Sara, for still remembering all the good days we had at Lázaro Cardenas.

I cannot forget to Noemí, Alex, Inés, Fer, Lamas, Lallana, Merino, Ana, Olalla, Sandra, Dani Martinez, Serrano, Lucas, Sergio, Cristobal, Pablo, Velasco, Carazov, Nacho, Ramirez, Lalo, Antonio, Maria, Marta, Ainara, Carlos Lopez, Carlos Martinez and Luis for their company during the last six years at Universidad Politécnica de Madrid, Spain.

Finally, I thank to my new family and friends, Tano, Diane, Victor, David, Pedro, Sergi, Charlotte, Sebastian, Alejandro, Alba, Romain, Ursules, Pedro Juan, Edu, Carmen, Laura, Enrico Manuzzato, Enrico Mosconi, Maxime and Pedro portugués.

Tampere, June 2015
Leticia Trinidad Valderas
Rodríguez

TABLE OF CONTENTS

1.	INTRODUCTION	1
2.	LITERATURE REVIEW	3
2.1	CoDe-X Compiler	3
2.2	DIL compiler	3
2.3	DRESC Compiler	4
2.4	RaPiD-C Compiler	5
2.5	XPP-VC Compiler.....	5
3.	THE COFFEE RISC PROCESSOR	7
4.	THE SCALABLE CGRA TEMPLATE	11
4.1	The existing GUI-based Compiler	12
4.1.1	CGRA Parameter File	13
4.1.2	Configuration Header Files	18
5.	DESIGN AND IMPLEMENTATION OF AUTOMATIC COMPILER	20
5.1	Reverse Polish Notation	20
5.2	Matrix-Vector Multiplication.....	20
5.3	Top-level module	21
5.4	Information Processing	24
5.5	Context Implementation	29
5.6	Mask Delivery	38
6.	TESTING AND EVALUATION	43
6.1	4 th -Order MVM	43
6.2	8 th -Order MVM	46
6.3	16 th -Order MVM	51
6.4	32 nd -Order MVM.....	51
7.	CONCLUSIONS.....	53
7.1	Future work	54

LIST OF FIGURES

Figure 3.1 Interfacing the COFFEE RISC core [12].....	8
Figure 3.2 COFFEE RISC Core pipeline [12]	9
Figure 4.1 Design blocks defined at compile-time using a set of parameters [6].....	12
Figure 4.2 PE Core Template [6]	13
Figure 4.3 Graphical User Interface to design new context.....	14
Figure 4.4 Binary fields in the configuration words	19
Figure 5.1 Compiler Dataflow	21
Figure 5.2 Elements of SCREMA template.....	22
Figure 5.3 Example of the information process for MVM algorithm.....	24
Figure 5.4 Enumeration of the memory banks and the PE of the first row.....	26
Figure 5.5 First step of information function	27
Figure 5.6 Different arrays which save information related to the operations and operands.....	27
Figure 5.7 Third step of schedule function where the operands are placed in the local input memory.....	27
Figure 5.8 First step in the processing function for MVM algorithm.....	29
Figure 5.9 Second step in the processing function for MVM algorithm	29
Figure 5.10 Implementation of second addition when MVM is processed	29
Figure 5.11 Explanation of the calculation of the best PE.....	31
Figure 5.12 The identifier number for each PE.....	33
Figure 5.13 Implementation of the mainContext function using MVM for a 4x8 PE CGRA	36
Figure 5.14 Implementation of canonical context in a 4x8 PE CGRA using a 4x4 PE CGRA template.....	37
Figure 5.15 Example of pre-processing context	38
Figure 5.16 Example of a matrix generated with parameters function and the different kinds of data.....	39
Figure 5.17 Example of a matrix generated with mask function and the different kinds of data	40
Figure 6.1 The two context of 4 th –Order MVM in 4x4 PE CGRA.....	44
Figure 6.2 Terminal results of 4 th Order MVM in 4x4 CGRA	45
Figure 6.3 Terminal results of 4 th Order MVM in 4x8 CGRA	45
Figure 6.4 The contexts of 8 th –Order MVM in 4x4 PE CGRA	47
Figure 6.5 Terminal results of 8 th – Order MVM in 4x4 PE CGRA.....	48
Figure 6.6 The context of 8 th -order MVM in 4x8 PE CGRA	49
Figure 6.7 The context of 16 th -order MVM in 4x8 PE CGRA	50

LIST OF TABLES

Table 4.1 Interconnections between PEs in CREMA [6]	15
Table 4.2 Possible functionalities of each PE with their related identifier	17
Table 5.1 Different connection cases considering one operand is in the memory and the PE is in the first row	34
Table 5.2 Different connections considering that one operand is not in the memory and the PE is in the third row	34
Table 5.3 Different data saved into contextPE array	41

LIST OF SYMBOLS AND ABBREVIATIONS

ADRES	Architecture for Dynamically Reconfigurable Embedded System
ALE-X	Arithmetic and Logic Expressions for Xputers
ASIC	Application-Specific Integrated Circuit
BP	Bank Position
CCB	Core Configuration Block
CGRA	Coarse-Grained reconfigurable architectures
CISC	Complex Instruction Set Computer
DIL	Data-flow Intermediate Language
DPSS	Data-Path Synthesis
DRESC	Dynamically Reconfigurable Embedded System Compiler
DSP	Digital Signal Processing
FireTool	Field programming and REconfiguration management Tool
FPGA	Field Programmable Gate Array
FU	Functional Unit
GP-REG	General Purpose REGisters
MOVTI	MOVE To Immediate
MVM	Matrix-Vector Multiplication
NML	Native Mapping Language
NPL	Native Mapping Language
NRE	Non-Recurring Engineering
NRE	Non-Recurring Engineering
PCB	Peripheral Control Block1
PE	Processing Element
PEP	PE Position
rDPA	Data-Path Array
RF	Register Files
RISC	Reduced Instruction Set Computer
RPN	Reverse Polish Notation
SCREMA	Scalable Coarse-Grain Reconfigurable Array with Mapping Adaptiveness
VLIW	Very Long Instruction Word
XPP	eXtreme Processing Platform

1. INTRODUCTION

Currently, the use of embedded systems and applications has been continuously increasing at the exponential pace. Applications such as streaming audio/video, image processing and interactive services demand high performance and use of sophisticated algorithms. This demand is shadowed by the tradeoff between constraints and performance. Constraints include design costs, time-to-market, non-recurring engineering (NRE) costs, etc., whereas efficiency can be related to performance, power dissipation and some other high level metrics, e.g., W/Hz. [1]. The two most used approaches for implementation are software running on a general purpose processor and hardware in the form of Application-Specific Integrated Circuit (ASIC) [2]. General Purpose Processors (GPPs) are characterized to be flexible enough to perform various applications. However, it does not provide fast execution time and high performance. ASICs are designed for a specific application, consuming less power and providing a high performance as they are custom-tailored. Reconfigurable architectures take the advantages of the two mentioned approaches and provide high performance for various applications along with flexibility. They can be classified in three groups according to their level of granularity: Fine-grained, Middle-grained and Coarse-grained. Fine-Grained reconfigurable architectures consist of functional units interconnected by a programmable network. These functional units implement low-level bit-oriented logic functions [3]. Field Programmable Gate Array (FPGA) belongs to such reconfigurable systems. Middle-Grained consists of configurable cells with a granularity level of equal to or less than 8-bits. They are mostly found integrated with microprocessor cores [3]. Coarse-Grained Reconfigurable Arrays (CGRAs) are composed of an array of functional units and storage resources. The functional units are designed to execute word-level or subword-level operations like additions and subtractions [1]. These reconfigurable architectures provide a drastic reduction of configuration memory and configuration time, as well as a diminution of the placement and routing problem. For that reason, various platforms using template-based CGRA have been designed; very popular since the last decade [1]. It should be noticed that the architectural specifications, computational model and designed tools diverge from one CGRA to another.

One of the first CGRA was Xputer architecture [3]. It consists of a reconfigurable Data-Path Array (rDPA) organized as a uniform two-dimensional array of ALUs. Another CGRA is RaPiD composed of a three integer ALUs, multipliers, six general-purpose datapath registers and three RAM blocks [4], whereas PiPeRench [3] is composed of reconfigurable pipelines stages called hardware stripes. Each of these hardware stripes has an array of Processing Elements (PEs) with registers and ALUs. Architecture for Dynamically Reconfigurable Embedded System (ADRES) is another CGRA. It consists of two major components, a Very Long Instruction Word (VLIW) processor and a re-

configurable array [1]. Another CGRA is BUTTER [5], which is 4x8 matrix of PEs with a 32-bit datapath. It works with general purpose Reduced Instruction Set Computer (RISC) processor. Since CGRAs contain high computational parallelism and throughput, they occupy an area of few million gates. In this context, a Scalable Coarse-Grained Reconfigurable Array with Mapping Adaptiveness (SCREMA) template was developed to generate applications-specific accelerators for optimal resource utilization was providing scalability [6]. Using SCREMA, the user can instantiate only those resources that are required by a specific set of applications.

One of the most important challenges of designing a reconfigurable architecture is related to its programming and compilation. Programming CGRAs are highly dependent on the structure and the granularity. In an ideal CGRA compiler, the user does not program in complex low-level programming. Compilation environments for reconfigurable systems span from manual placement and routing (P&R) tools to automatic design flow from high-level programming languages, such as C.

The target of this research work is to design and implement a compiler framework for SCREMA, to replace the existing manual P&R Graphical User Interface (GUI) tool. Such tool would be time consuming for the user if the size of the CGRA is extra large and there is large amount of computational information. The implemented compiler will automatically place and route the operations over the matrix of PEs and generate output configuration packages needed to create the SCREMA template. The input source is a Reverse Polish Notation (RPN) file with the description of the target algorithm. From this file, the main processing context will be created by the compiler. The compiler framework implements two important dataflow graphs to perform a broad range of sum-of-products related algorithms. The compiler framework has been tested and evaluated specifically for integer Matrix-Vector Multiplication (MVM) algorithms.

This thesis work is organized as follows. Chapter 2 discusses some of the related CGRA compiler frameworks. The architecture of platform which is composed of a RISC core and a CGRA is explained in Chapters 3 and 4, respectively. The design and implementation of the compiler is explained in Chapter 5 and the results from the Matrix Vector Multiplication are discussed in Chapter 6. In Chapter 7, conclusions and future work are presented.

2. LITERATURE REVIEW

Compiler environments of CGARs differ by the approach used for technology mapping, placement and routing. The characteristics of each compiler depend on the constraints and specifications of the reconfigurable architecture. The primary works regarding compiler design can be found for FPGA. In this case, compilers analyze a hardware descriptive text, i.e, VHDL or Verilog, and synthesize it for the FPGA device. In case of CGRAs, the compilers use only high-level programming languages, e.g., C. The following examples are the most important and relevant academic work for CGRA compilers.

2.1 CoDe-X Compiler

CoDe-X compiler is used to map a C-like code, called X-C, into Xputer hardware [7]. Xputer is one of the first CGRA. It consists of two-dimensional array of arithmetic and logic reconfigurable units. There are three kinds of interconnections between them [3]: nearest neighbors, row/column back-buses and one global bus. The architecture is characterized for providing high area effectiveness. The input source for the compiler is X-C. The input program is divided by using the compiler which applies a number of loop parallelization transformations. In the next step, the compiler creates a description for each division using Arithmetic and Logic Expressions for Xputers (ALE-X) which will be synthesized using a Data-Path SyntheSis (DPSS). ALE-X can be generated automatically from the input code or it can be made manually. DPSS generates an assembler file for the configuration of the target architecture. Moreover, DPSS has a data scheduler which is responsible for organizing the operation to be mapped in the architecture. Finally, after scheduling and with the help of a mapper, placement and routing is performed. Furthermore, the array configurations and corresponding controllers are also generated.

2.2 DIL compiler

DIL compiler has been designed to map the input source to PipeRench architecture [8]. In this case, the compiler uses a Data-flow Intermediate Language (DIL) as an input source. The compiler is responsible for scheduling, placement and routing.

PipeRench is a template-based, pipelined reconfigurable architecture since it consists of virtual pipeline stages implemented as hardware stripes [9]. There can be several programmable PEs that can be found in each stripe. Each PE is composed of a number of ALUs, pass registers and several control logic units. Each ALU has three inputs; two

inputs for data and the other one is used as control input. Three types of connections can be found inside this reconfigurable architecture. One of them is between two stripes, the other one is between a stripe and some of their PEs, and the last connection is pass-register interconnect, where each register can transfer information to the other registers [9].

DIL [3] is a hardware independent high-level language which is also similar to Silage [10] and behavioral Verilog. It can be used to describe the algorithms for reconfigurable architectures. DIL has the C-operators and allows the programmer to manipulate arbitrary the width of integer values to guarantee that no information loss happens due to the overflow or type-conversions.

The first step is reading the specification of the target architecture. Then, the function and module techniques are performed in the input source to facilitate further analysis. During the synthesis process, the compiler constructs a graph whose structure is hierarchical acyclic and the nodes represents operations, I/O ports and delay-registers. After generating the graph, the compiler performs placement and routing over the graph, to generate the virtual hardware stripes [3]. In those processes, the compiler uses a deterministic linear-time greedy algorithm based on list scheduling [3]. This compiler has excellent compilation speeds and produces optimal hardware utilization.

2.3 DRESC Compiler

Dynamically Reconfigurable Embedded System Compiler (DRESC) [1] is a compiler designed for ADRES. This compiler maps the intensive loops parts of the C program to the reconfigurable architecture while the rest of the code is executed by VLIW processor. An intermediate representation is used to implement the scheduling process, and then, a novel modulo scheduling algorithm is employed to perform a combination of scheduling, placement and routing.

ADRES is composed of a VLIW processor and a reconfigurable array. The reconfigurable array consists of two dimensional array of functional units (FUs) which are responsible for executing a set of operation. The storage elements are Register Files (RFs) and memories blocks. Each FU can perform fixed-point operations. ADRES template is defined using an XML-based architecture specification language which is generated by DRESC compiler [3].

DRESC compiler accepts a C language program as input source. The first step is identifying the loops which will be mapped into the two dimensional array of FUs. Then, an intermediate representation is generated using a VLIW compiler framework called IMPACT. The representation is called *Lcode* and it is used to execute the scheduling process. A novel modulo scheduling algorithm is developed by the program and the architecture representation as input sources. It is a software pipelining technique which at-

tempts to schedule an iteration of a loop and determine which of the FUs will perform each operation. Finally, the placed and scheduled operations are implemented [1]. The objective of this compiler is to provide high-performance using all the family of ADRES architectures

2.4 RaPiD-C Compiler

RaPiD [4] architecture is a reconfigurable pipelined datapath architecture that uses RaPiD-C compiler to map a high-level hardware description. This compiler needs RaPiD-C, as a C software language to implement an application over the reconfigurable system. It also requires the user collaboration to specific parallelism, data movement and data partitioning across the multiple elements of the RaPiD architecture [3]. The compiler process is composed of four steps [4]: netlist generation, dynamic control extraction, instruction stream/decoder generation and I/O address generation.

RaPiD [4] is a coarse-grained field-programmable architecture that enables pipelined computational structures to be built from an array of arithmetic units, registers and memories. It consists of functional units such as ALUs, multipliers, General Purpose REGisters (GP-REGs) and RAMs. The large numbers of functional units are spread across a field-programmable segmented bus structure.

The application description is given in RaPiD-C language, which consists of multiple loops, one for each stage of the application. The programmer has to design the code paying attention to the architecture characteristics, for instance, to the number of functional units and the available memory. From the RaPiD-C language, specific operations are assigned to a specific stage at a specific time. One operation is performed by each pipeline stage in each cycle [4].

During the netlist generation process, the compiler instantiates registers for variables, ALUs for adds (and also other operations), multipliers for multiplication and multiplexers for *if-then-else* statements [4]. Once the netlist is generated, the compiler executes dynamic control extraction process. All the multiplexers are gathered in a larger multiplexer, depending on the size of the reconfigurable architecture. The compiler also realizes the same operation for the functional units. The compiler then extracts the address and instructions generated to control transfers values between stages [3]. The compiler is specifically designed for RaPiD, however it can be used in different technologies such as FPGAs or ASICs.

2.5 XPP-VC Compiler

The eXtreme Processing Platform (XPP) [10] uses XPP-VC compiler to map a C code into the reconfigurable architecture. The compiler flow consists of a preprocessing and a

dependence analysis of the input code followed by a temporal partitioning of each task, Native Mapping Language (NPL) generation and the final placement and routing.

The XPP technology is provided with development tool suite consisting of a placer, a router, a simulator and a visualizer [10]. The architecture is characterized by two-dimensional array of PEs, internal memories and interconnection resources. The PEs executes arithmetic and logic operations, comparisons and special operations as counters. It should be noted that the output of each PE can be considered as the input source of another PE.

The input source of the compiler is a C code without structures, floating-point data, pointers, irregular control flow, recursive and system calls [10]. The user can provide a file where the parameters of the architecture and the external memories are defined. A preprocessing step is required by the compiler to transform the code into a program which facilitates the next steps. In the temporal partitioning, the code is divided into tasks that will be transformed to a Native Mapping Language (NML) reconfiguration code. After that, the NML module is placed and routed over the architecture. Finally, the configuration data and the binary codes are generated.

3. THE COFFEE RISC PROCESSOR

The compiler framework presented in this thesis work generates CGRA accelerators using SCREMA template in order to work with COFFEE RISC processor in a processor/co-processor model. COFFEE and SCREMA generated accelerators interact with each other using a network of switched interconnections that provide dedicated connections for fast communication [11].

COFFEE RISC Core [12] is a general-purpose processor which was designed for setting up embedded systems with a motivation to accelerate the tasks related to telecommunication and multimedia applications. There are different types of processor cores which can be classified depending on the instruction types: Reduced Instruction Set Computer (RISC), Complex Instruction Set Computer (CISC) and Digital Signal Processing (DSP). RISC and CISC are used to implement general purpose operations. However, DSP is a core to execute digital signal processing kernels where large numbers of mathematical operations have to be performed quickly. The difference between RISC and CISC is basically the type of instructions. CISC executes complex instructions such as an arithmetic operation load from memory. Nevertheless, RISC only accesses to the memory if load or store operations are called. The target of RISC is to perform the instructions using fewer cycles per instruction. COFFEE RISC Core was designed by following the RISC philosophy to executing one instruction per cycle. It uses delay branching with efficient hardware which consists of finding instructions to be placed in delay slots [12]. The length of a RISC instruction is fixed. RISC instructions contain all needed information without requiring any access to the memory. Only load and store instructions access the memory.

The COFFEE RISC Core instruction set architecture incorporates the most common instructions of a typical RISC core. It has flexibility in its instruction-set which allows coprocessor support. A hardware support is also placed to give the opportunity to execute DSP instructions. There are 14 arithmetic instructions, 10-bit field manipulation instructions, six boolean operation, eight conditional branches, four other jumps and six shift instructions [12]. In addition, COFFEE RISC is characterized by implementing real-time operating systems which can be achieved by dividing its mode into user register set and supervisor register set.

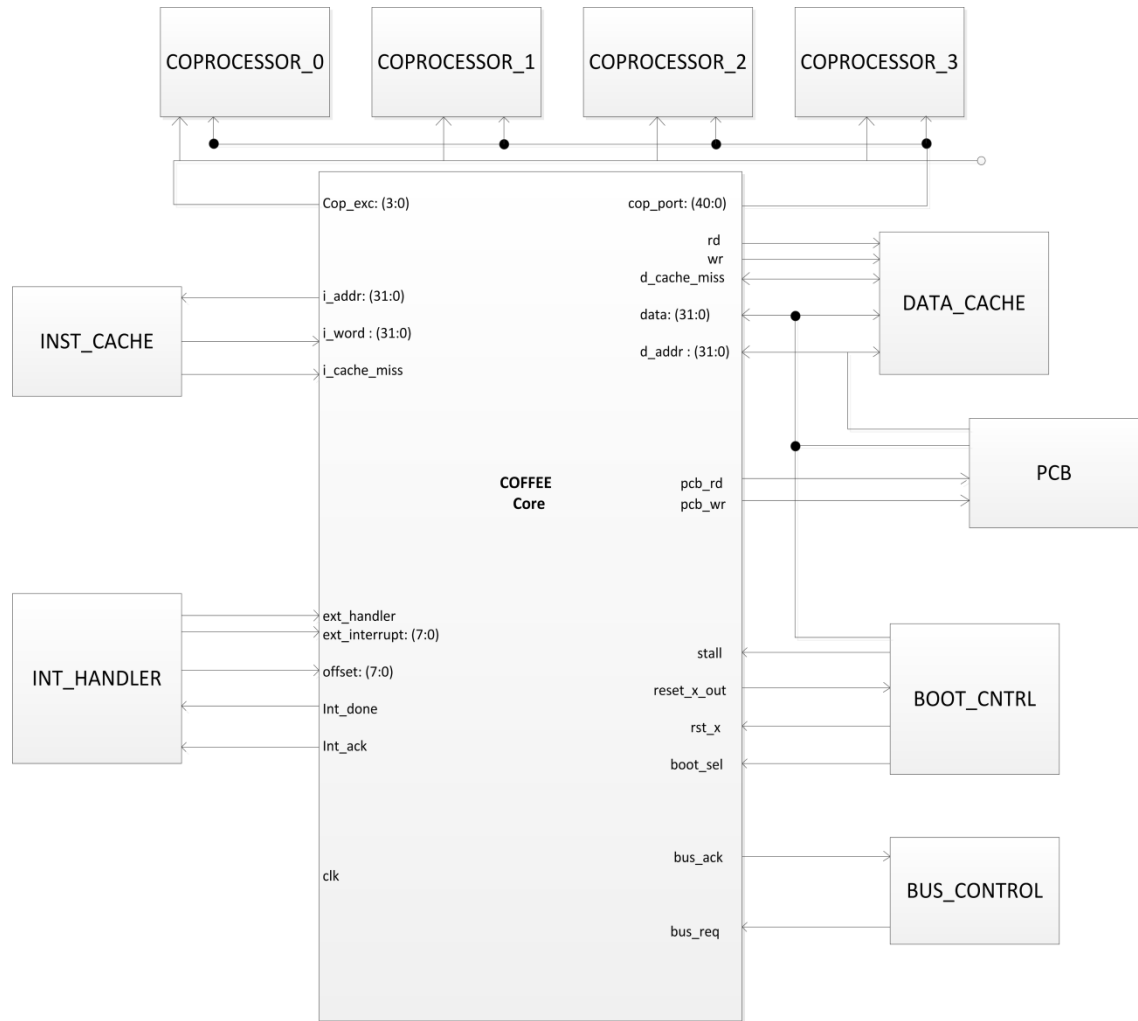


Figure 3.1 Interfacing the COFFEE RISC core [12]

The COFFEE RISC Core can be called load-store machine since memory operands are saved into registers before any operation [12]. The same occurs with the result of each operation where the value is saved into a register. To avoid excessive memory traffic, a large general-purpose register bank is used. This register is divided in two parts, one part for the user mode and the other for the supervisor mode. Moreover, COFFEE RISC offers the opportunity to configure it. For that reason, an internal memory mapped configuration register bank called Core Configuration Block (CCB) is provided. A Peripheral Control Block (PCB) was designed to configure and communicate with the peripheral devices.

The COFFEE RISC core is 32-bit Harvard architecture, where data and instruction memory can be distinguished from their interfaces. As it can be seen from the interface of COFFEE RISC Core in Figure 3.1, the memories are interconnected and they allow accessing them with different memory addressing schemes. The access time to the

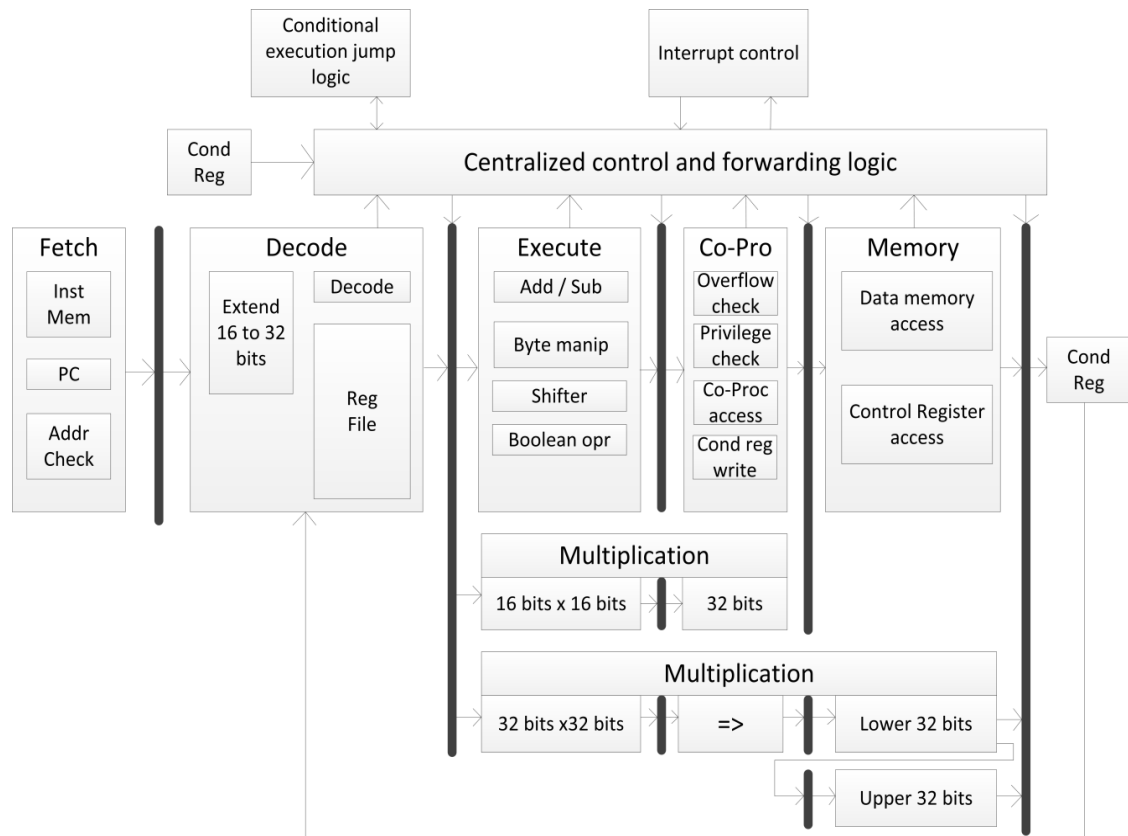


Figure 3.2 COFFEE RISC Core pipeline [12]

memory may be reconfigurable using CCB. The PCB module is included to the COFFEE RISC Core's data bus interface.

The COFFEE RISC Core can support four coprocessor interfaces that are attached as memory interface along with the maximum register bank size of 32 bits [12]. Such coprocessors can interrupt the core by an exception signal. COFFEE RISC core also provides an internal interrupt and eight different kinds of external interrupts. Those external interruptions can be from twelve sources if the coprocessor interfaces are not used.

As it can be seen in Figure 3.2, the COFFEE RISC Core has six stage pipelines with a maximum throughput of one instruction per cycle. The first pipeline stage is called fetch, where one instruction is fetched from the instruction memory depending on the pointer of the program counter. The address is checked by the *Addr Chek* block. During the decode stage, the instruction is identified and processed for the following stages. In the third pipeline staged, called execute, the arithmetic and logic operations are performed. Some operations like multiplication need to be executed in two pipeline stages. For that reason, co-processor stage is provided. After that, if load or store operations are required, they can be executed in the data memory stage. Finally, in the last stage, the final data will be written in the destination register of the register file.

The COFFEE RISC Core is a VHDL description which can be portable between technologies. For this reason, it has been synthesized for 90 nm low-power ASIC technologies, considering low supply voltage of 0.95V, high temperature of 125 degree centigrade and variations to guarantee high manufacturing processes. The highest frequency found was 150 MHz [12].

In addition, it was synthesized and P&R on ALTERA EP2S130F1020C4 and Xilinx XC4VLX160FF1148-11 devices, with different timing constraints. The results show the difference between the two technologies, in such a way that Altera targets higher density and higher operating frequencies, while Xilinx has less power dissipation [12].

The software development tools are designed at Tampere University of Technology, Finland for COFFEE RISC Core. Several template-based CGRAs have been developed to work with COFFEE RISC Core, e.g., CREMA, AVATAR and SCREMA. CREMA is a 4x8 matrix of PEs. AVATAR consists of 4x16 matrix of PEs while SCREMA is a scalable CGRA to instantiate different sizes of CGRAs chosen by the user. SCREMA platform will be explained in details in the next chapter.

4. THE SCALABLE CGRA TEMPLATE

SCREMA is a Scalable Coarse-Grained REconfigurable Array with Mapping Adaptiveness [6]. SCREMA was designed to be scalable in both the number of PE rows and columns to meet the specific performance requirements of an algorithm. SCREMA is equipped with 32-bit local memories. The mapping adaptiveness consists of tailoring the array to specific application requirements [6]. A CGRA is composed of a matrix of interconnected PEs. The functionality and the routing among PEs can be modified by using the runtime reconfigurability feature of CGRAs. Every PE supports a fixed set of operations and different interconnections. Specifically, each PE can perform 32-bit integer or floating point operation in IEEE-754 format [11]. In general, interconnections can be classified into local, interleaved and global interconnection. A reconfiguration pattern, called context, specifies the operation to be implemented in each PE, as well as the interconnection. Each application consists of one or more reconfiguration patterns. For each context, several reconfiguration words are generated. They are stored into the reconfigurable memory of each PE that enables fast switching between contexts. Such mechanism permits the modification of the functionality and routing which can be performed in one Clock-Cycle (CC) [6].

The CGRA design is realized using a VHDL template which can be generated using Field programming and REconfiguration management Tool (FireTool). Using Firetool, the user can instantiate manually the functionality and the routing among PEs for each context. It generates two different files. The first one is a VHDL package based on the template size. The second one is a set of C header files that contain the operations and interconnections need to be selected for each context. The content of this thesis is to provide an automatic approach to generate both files, without using Firetool.

The elements that can be found inside the VHDL package are related to the routing of PEs, the PE functionality and the reconfiguration infrastructure. Figure 4.1 shows the last three components.

Each PE has two inputs and two outputs. The data of each input is taken from a multiplexer. Each multiplexer is connected to the possible data sources. The selection bits, defined in the VHDL file will indicate which connection has to be performed by the PE for each context. Moreover, in the VHDL package, the size of such multiplexer is defined which depends on the number of input sources. For example, if in one application, one PE is always connected to the same input source, then the multiplexer must not be used. However, if one PE needs four different connections, a 4-to-2 multiplexer has to be instantiated.

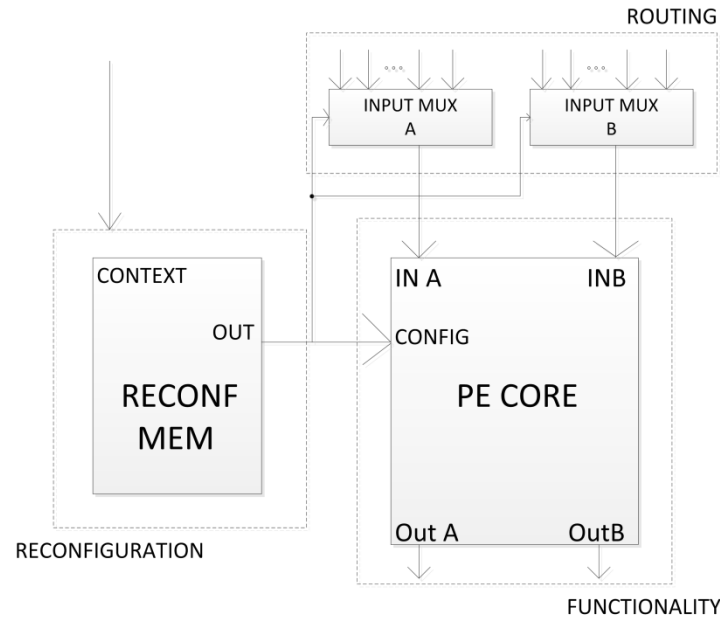


Figure 4.1 Design blocks defined at compile-time using a set of parameters [6]

The architecture of the PE is in most part compile-time configurable [6]. The different elements of the PE core can be seen in Figure 4.2. The configuration of each element can be found in the VHDL package. The components can be divided in two groups: functional blocks and reconfigurable control blocks. Functional blocks comprise integer functional units, such as adder, multiplier, shifter and a memory for LUT logic function implementation, immediate register and a floating point block. The VHDL file indicates which blocks have to be instantiated for each context. Reconfigurable control block consists of a decoder and the output multiplexer. The size of these two elements depends on the number of operation to be performed by each PE.

The configuration words are inserted into the array of PEs using pipelined infrastructure [13]. Such process consists of injecting the reconfigurable patterns into the array to be propagated along the horizontal and vertical directions.

4.1 The existing GUI-based Compiler

FireTool is the manual graphical tool used to design the different reconfiguration patterns for the CGRA. This tool consists of several windows. In the first one, the user is asked to introduce information about the size of the CGRA, the project name and the directory where the information will be saved. In the second window, the user is allowed to design a new context or to modify an existing one. The buttons to generate the VHDL package and the C header files are also found in the second window.

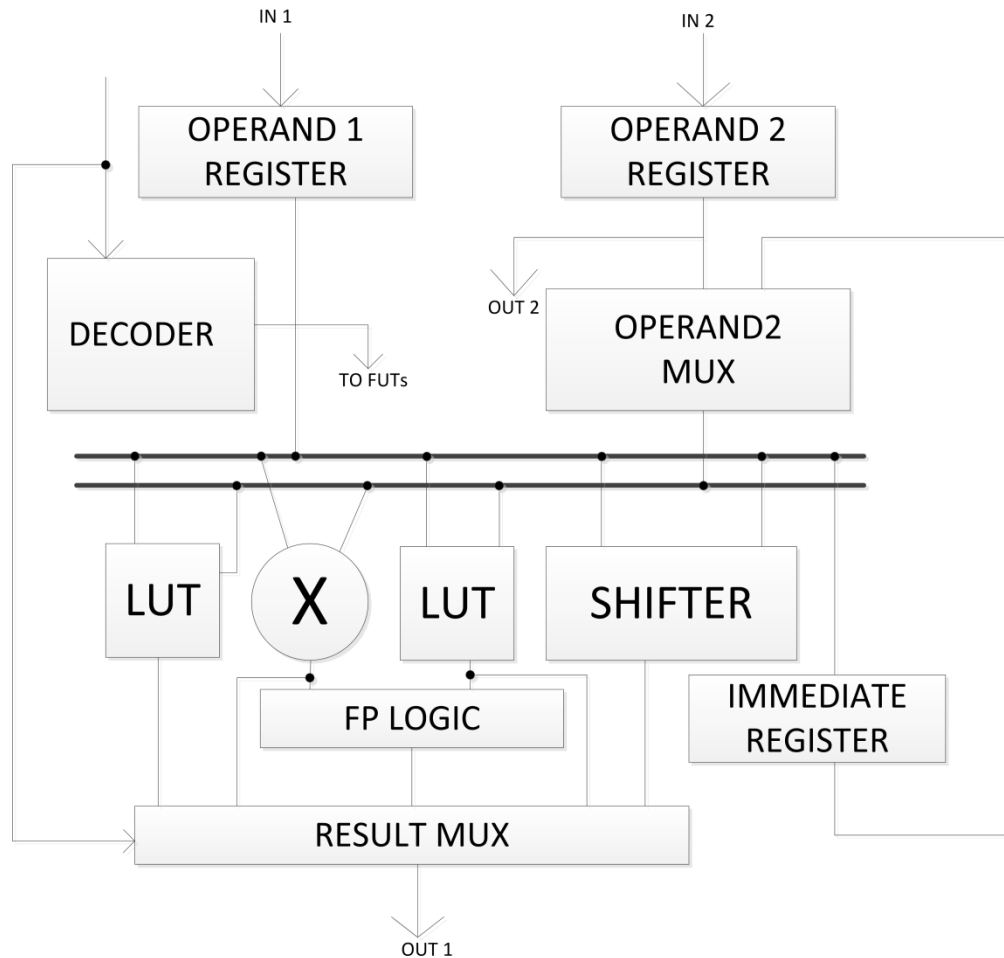


Figure 4.2 PE Core Template [6]

Regarding the design of a context, a user must perform the placement and routing manually by using the graphical interface shown in Figure 4.3. It consists of an array of PEs where the users are allowed to select the functionality and the interconnection of each PE input. The user can also activate the use of the immediate register for shift operation.

After configuring one context, the VHDL package and the C header files can be generated. Once the new reconfiguration pattern is designed, both of the files (VHDL, C headers) will be updated. The characteristics of each file are explained in detail in the following subsections.

4.1.1 CGRA Parameter File

It is a VHDL file which is based on the CGRA template configuration. It defines the architecture of each configuration pattern, i.e., the routing among PEs, the PEs functionality and the reconfiguration infrastructure. The file consists of several parameters. The description of the most important parameters is presented below:



Figure 4.3 Graphical User Interface to design a context

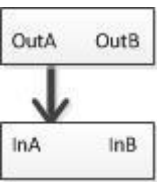
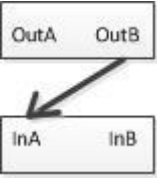
- *crema_cfg_ic_depth_A* → This parameter represent a matrix of the same size as the size of the CGRA. Each position represents one PE. As it was mentioned in the Section 4, a PE has two inputs and in each of them, one multiplexer is implemented to select the input source. In this matrix, the select line information of the multiplexer of the input A is saved.
- *crema_cfg_ic_depth_B* → Its description is same as *crema_cfg_ic_depth_A* except that each position of this matrix saves the selection bit of the multiplexer of the input B.
- *crema_cfg_width* → The size of this matrix is the same as the size of the CGRA. This parameter is a matrix of integers that represents the selection bit of the output multiplexer. In this case, the size of the multiplexer depends on the number of operations that have to be performed by a PE.
- *crema_sel_width* → Each integer in the matrix is related to the output of the PE. The integers select the multiplexed output of the PEs. The specific value of the integer depends on the number of functions that a PE processes.
- *crema_mux_mask_A* → This large constant matrix is composed of several smaller matrices. The value of these smaller matrices is equal to the number of PE rows in the CGRA that means if there are four PE rows in the CGRA there will be four groups of small matrices. Every small matrix has to be of the same size. A non-zero column of these small matrices represent an interconnection with another PE.

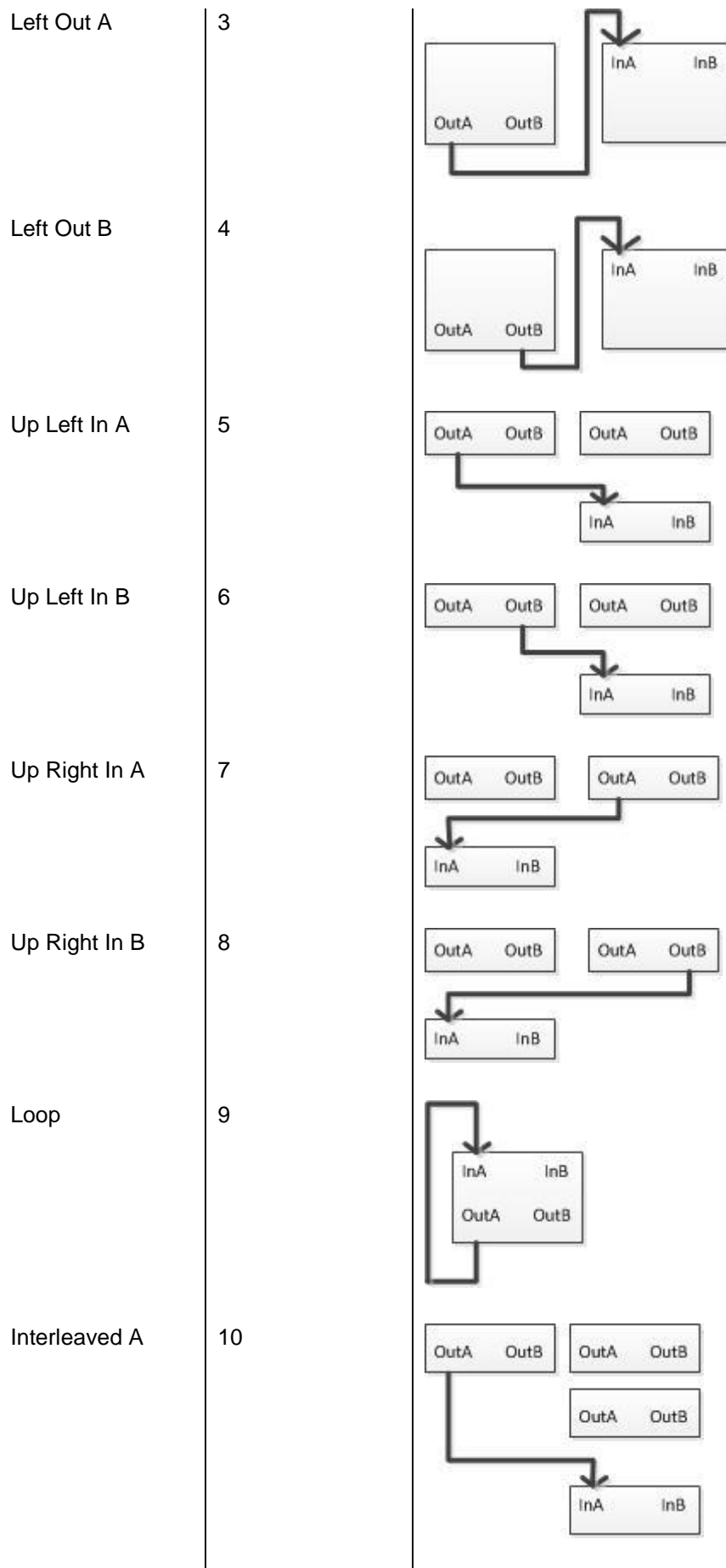
All stored information in these small matrices corresponds with the connection identifier (See Table 4.1) for the input A of the PE. As it can be seen in Table

4.1, each connection is represented by a number. This means that if the input source of one PE is in the output A of another PE, the connection would be Up In A and is represented by '1'.

- *crema_mux_mask_B* → This large matrix also consists of several smaller matrices, as in the previous case of *crema_mux_mask_A*. The unique difference is that the connection identifier for the input B can be stored in each position.
- *crema_cell_mask* → The dimension of this group of matrices is the same as in the last two cases. As it can be seen in Table 4.2 that an operation to be performed by a PE has an identifier. The operation identifier can be stored in each position of this group of matrices.
- *crema_add* → The dimension of this matrix is equal to the size of the CGRA. The binary information saved in each position will indicate if the PE has to perform an addition operation.
- *crema_sub* → The characteristics of this matrix are the same as in the last case. However, the only difference is binary information in the matrix which indicates if a subtraction has to be performed by each PE.
- *crema_mul* → Each position of this matrix indicates if the PE has performed a multiplication. The size of the matrix is the same as the size of the CGRA.
- *crema_shifter* → In case, if a shift operation has to be performed by a PE, a binary parameter 'true' will be stored in each position of the matrix.
- *crema_ram* → In this case, each position of the matrix will represent a RAM related operation.
- *crema_imm* → The characteristics of this matrix are the same as the others, but with a difference that it will indicate if the immediate register has been used by the PE.
- *crema_confup* → This matrix indicates if the configuration of the above PE has been used for the reference PE. The dimension is the same as the CGRA.
- *crema_word_width_int* → It indicates the word width of each PE which is equal to 32 bits.

Table 4.1 Interconnections between PEs in CREMA [6]

Interconnections	Interconnections identifier	Representation
Disconnected	0	
Up In A	1	
Un In B	2	



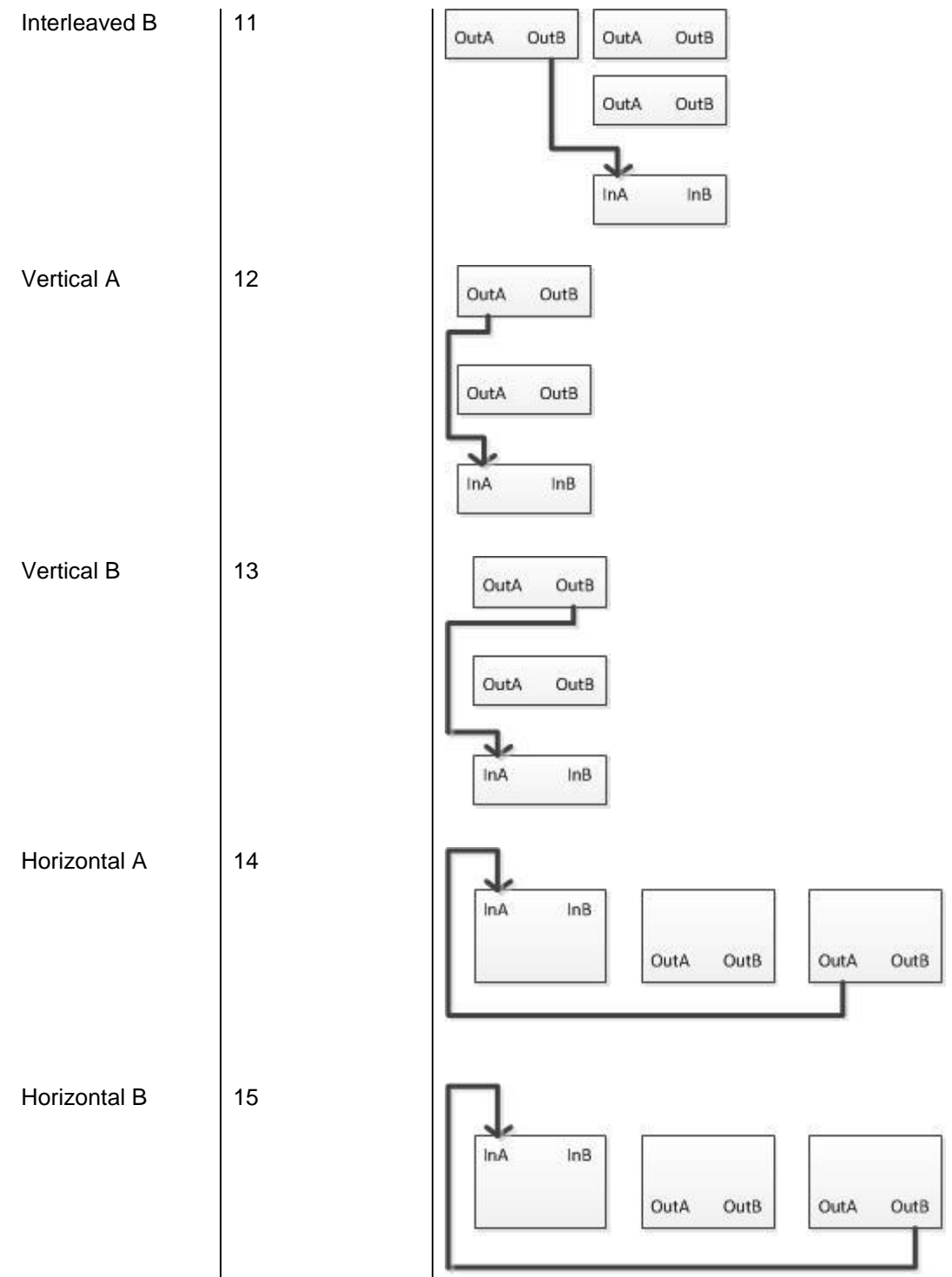


Table 4.2 Possible functionalities of each PE with their related identifier

Functionality	Functionality identifier
NOP	0
ADD	1
SUB	2
MUL	3
ADDS	4

SUBS	5
MULS	6
SHIFT	7
LOAD	8
STORE	9
MOVTI	10
DELAY	11
UNREGFT	12

4.1.2 Configuration Header Files

Header files contain different configuration words for each context. Each configuration file consists of several configuration arrays. The number of configuration arrays is the same as the total number of contexts needed to be implemented for an application.

The configuration words are represented by eight hexadecimal digits and each number keeps the information related to the routing and the functionality of each PE. The first two digits save the PE identifier. Starting from the last significant bit side, the first hexadecimal digit saves information about the interconnection of the input A. The information related to the interconnection of the input B is saved in the next digit. The next hexadecimal number is reserved to store the information about the functionality. If the PE uses the immediate register, one digit is needed to show in which context the immediate register has been used. Generating such configuration words is explained in the following example.

Figure 4.4 depicts the example, where the configuration words can be seen and are represented in binary numbers for each context. It can be seen that each group of bits represent several other routing and operation parameters, as it was mentioned earlier. The number of bits for the groups called inA, inB and Oper depends on amount of interconnections and different functions to be performed by each PE, respectively.

Suppose that first PE will receive the input sources from three different PEs, both for input A and input B. To represent three options, two bits are needed. Hence, the selection bits for each multiplexer will be equal to two. If a PE is supposed to receive data from only one input source then only one bit would be required. Two bits representing the routing field inA and inB are shown on the right side of Figure 4.4 as the selection bits of the multiplexer for each input. Each input source is listed depending on the order it was used. If one input source is used again in another context, it is codified with the same number. If we look at Figure 4.4, it is supposed that input A uses the input source numbered as zero both in the first and fourth context. The same occurs for input B in the third and fourth context, but in this case the input source is numbered as three.

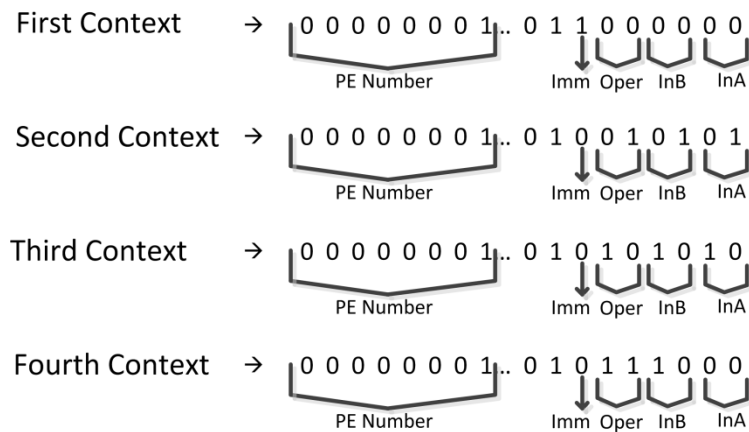


Figure 4.4 Binary fields in the configuration words

Then, assume that PE performs four different operations. In this case, the number of selection bits for the decoder is equal to two since there are four different possibilities. Each operation will also be represented by two bits called Oper (See Figure 4.4). The bits for the decoder are also listed depending on the order they are used.

Finally, if a PE needs the use of the immediate register then the bit called imm is used. If the immediate register is not instantiated then the imm bit is not written in the configuration word. To know in which context the immediate register is needed, imm bit will be 1. As it can be seen in Figure 4.4, the immediate register was only needed in the first context.

5. DESIGN AND IMPLEMENTATION OF AUTOMATIC COMPILER

The compiler framework replaces the existing GUI tool with an added feature of automatic placement and routing. The input source of the compiler is a Reverse Polish Notation (RPN) file. Such file is analyzed to extract the computational information and to generate the main processing context. The user has the possibility of implementing additional configuration patterns. Two different design paradigms are provided: pre-processing context and canonical context.

5.1 Reverse Polish Notation

The input source for the compiler is a RPN sequence related to the equations to be mapped on the CGRA. RPN is a mathematical notation where each operator is placed after their operands. This means that instead of writing $a+b$ in general, the RPN is written as $ab+$. Such notation was invented by Burks, Warren and Wright [14]. At the beginning of 1960 it was reinvented by Friedrich L. Bauer and Edsger Dijkstra [14]. They reinvented it to reduce the memory access time in the computers and to provide higher performance. RPN is frequently used in computer science, e.g., in Unix pipelines and in concatenative programming languages based on stacks. A Last-In-First-Out (LIFO) is needed to process the RPN information. The algorithm followed to process the RPN information consists of distinguishing between operands and operators. The operands are store into one LIFO. If one operator is identified, the program pulls the last two stored operands from the LIFO, produces the operation and pushes the result into the same LIFO. This algorithm is followed by the compiler as it will be explained later.

5.2 Matrix-Vector Multiplication

The compiler has been tested by Matrix Vector Multiplication (MVM). An N^{th} order matrix consists of N scalar products between each row of an $N \times N$ matrix and a vector. This process is just the multiplication between one element in the row of the matrix and another element of the vector which are in the same position. Once all multiplications have been made, the addition of all results is required. Eq. 5.1 represents an example of 4^{th} order MVM. The result of each row of 4×4 matrix and the vector can be observed in Eq. 5.2.

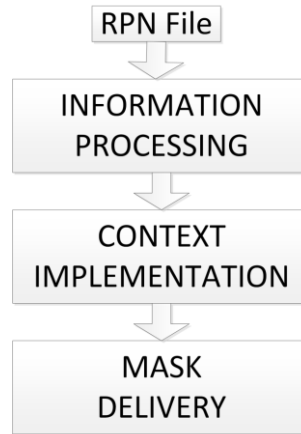


Figure 5.1 Compiler Dataflow

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} \quad (5.1)$$

$$p_1 = (a_{11} \times b_1) + (a_{12} \times b_2) + (a_{13} \times b_3) + (a_{14} \times b_4)$$

$$p_2 = (a_{21} \times b_1) + (a_{22} \times b_2) + (a_{23} \times b_3) + (a_{24} \times b_4)$$

$$p_3 = (a_{31} \times b_1) + (a_{32} \times b_2) + (a_{33} \times b_3) + (a_{34} \times b_4)$$

$$p_4 = (a_{41} \times b_1) + (a_{42} \times b_2) + (a_{43} \times b_3) + (a_{44} \times b_4)$$

(5.2)

In order to implement this algorithm on SCREMA template, the RPN sequence of only p1 is sufficient as p1, p2, p3 and p4 are all identical.

5.3 Top-level module

Compiler design consists of three main steps: information processing, context implementation and mask delivery. The Figure 5.1 depicts the compiler dataflow. During the information processing, the RPN file is analyzed to place the operands inside the local input memory of the CGRA optimizing resources. Within the context implementation stage, the different contexts are designed. The main processing context is designed from the RPN file. The rest of contexts are implemented by two different design paradigms (pre-processing and canonical context). Finally, the VHDL package and C header files are generated in the mask delivery step.

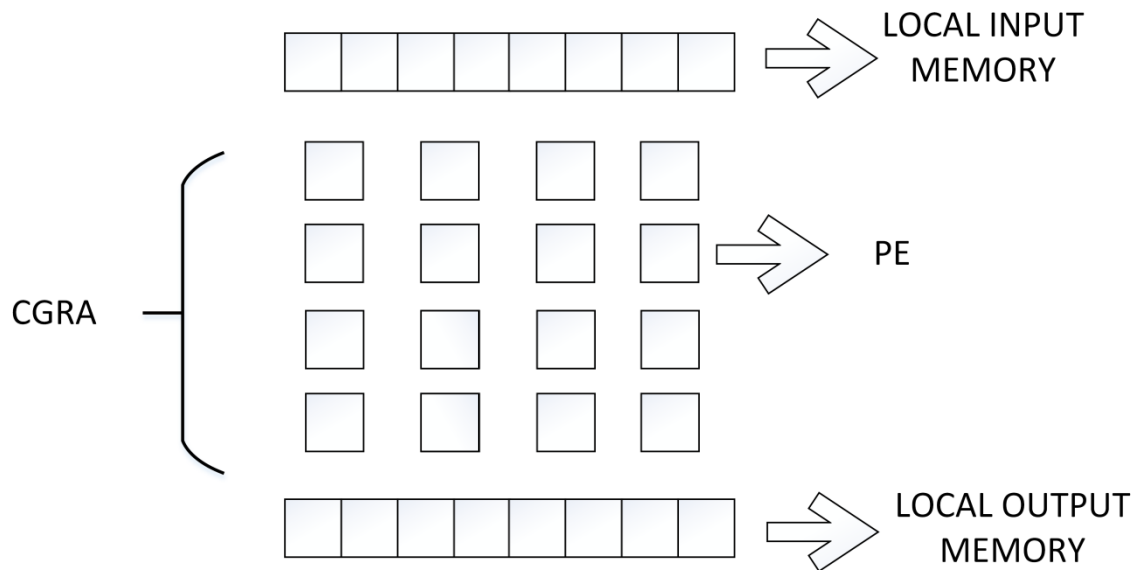


Figure 5.2 Elements of SCREMA template

To design the compiler, the SCREMA template has been virtualized. In Figure 5.2, it can be seen the virtualized elements of SCREMA template: the local input memory, the local output memory and the PEs.

The design of the compiler consists of several functions which will be explained below. The main function is responsible for saving the initial information provided by the user which is in the RPN file, i.e., the project name and the size of the desired CGRA. The main program also initializes the global variables, executes the rest of the functions and interacts with the user to design new contexts.

As it was mentioned before, the main function needs to declare all the global variables. Such variables define the virtualized parts of the CGRA (PEs and local memories), the VHDL matrices (See Section 4.1.1), LIFOs and their pointers. The VHDL matrices are represented as arrays. Two CGRA are created: a CGRA template and the CGRA to be instantiated. The CGRA template is generated from the RPN file. The template design might be replicated to obtain the CGRA to be instantiated. The template CGRA can instantiate at maximum sixteen columns of PEs and any number of PE rows based on the user input. In the design of compiler, a PE is defined as structure, as it can be seen in the description below. It consists of two strings, *inA* and *inB*. The connection identifier for each input is saved in these strings, respectively. The operation identifier is saved in the string called *operation*. The information about the equation executed by the PE is saved in the *info* string. *ConnectedPE* indicates if that PE has been connected to other PEs or to the output memory. The PE information for each context is stored in the *con-*

text array. This array is needed to generate the C header file. Finally, *pointerContext* is defined to indicate in how many contexts the PE have been used.

```

1 typedef struct PE {
2     char inA[3];           //save the connection identifier for InA
3     char inB[3];           //save the connection identifier for InB
4     char operation[3];     //save the operation identifier
5     char info[100];        //store information about the operation
6     int connectedPE;
7     int context[8][9];    //save the values of this PE
8     int pointerContext;
9 }PE;
```

Program 5.1 PE structure declaration

Both of the input and output local memories of the CGRA are defined in the main function. The information that will be used by the PEs is saved into the input memory while the results of the operations to be performed by the CGRA are saved into the output memory. The size of both memories is two times the number of PE columns of the CGRA that is being targeted by the compiler.

As it was discussed in section 5.1, the RPN algorithm is based on the use of LIFOs. The compiler needs five of them to process the information. The compiler reads the RPN file character by character. When one operand is identified, it is saved into a LIFO called LIFO1. The compiler repeats the same process until it reads an operator. In that case, the compiler extracts the two last operands from LIFO1 and it searches for the ideal PE to place the operation to be performed. If the PE is in the first row of the CGRA, the result is saved into the LIFO called ROW0. The same occurs with the rest of the results of the other rows. They will be saved inside of their respective LIFOs. They are called ROW1, ROW2 and ROW3, respectively. Every LIFO has its own pointer to know the position for pushing or pulling the data, and also indicates how many values can be stored in them.

Finally, the number of context is initialized in the main function. This parameter is needed to know which context is being designed at each moment since that information is required once the C header file is generated by the compiler.

Inside the main function, several files can be seen. They are required to read the RPN file and to generate the VHDL package and the also C header file. In particular, *file1* is for saving the RPN file, *file2* for creating the VHDL file and *file3* for generating the C header file.

Some functions are executed by the main program: *initialization*, *information*, *processing*, *mainContext*, *draw*, *header*, *preprocessing* and *canonical*. The VHDL matrices are initialized by the *initialization* function. This function is needed to be able to modify those matrices in other functions. The required information for that function is the size

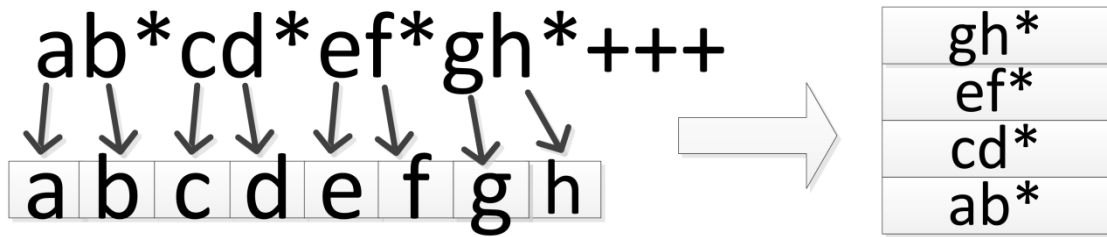


Figure 5.3 Example of the information processing for MVM algorithm.

of the CGRA and the VHDL file since the dimensions of the matrices depend on the size of the CGRA. *Information* function is used to extract the relevant information of RPN file and to place the operands inside the input memory. *Information* function needs the CGRA template dimensions, the RPN file and the number of operations to be performed by the CGRA. *Processing* function is responsible for reading the RPN file and performing the placement and routing in the CGRA template where the size of the CGRA template and the RPN file will be required. *MainContext* function replicates the CGRA designed in *processing* function depending on the size of the desired CGRA. As it can be understood, this function requires the dimension of both CGRA template and the CGRA to be instantiated. *Draw* is the function for generating the VHDL package. It reads the data from the VHDL matrices and creates the corresponding file. *Header* function is responsible for generating the C header file. This function will read the data from the *context* array of each PE and design the configuration stream. The input parameters of this function are the size of the final CGRA, the C header file and the project name. Two design paradigms are provided: pre-processing and canonical contexts. *Preprocessing* and *canonical* function will design these two contexts, respectively.

5.4 Information Processing

In information processing module, the compiler places automatically the pair of operands in the input memory. It also performs the placement and routing for the first row of the CGRA template. Information processing module consists of *information* and *schedule* functions

Information function collects the important data from the RPN file to optimize the process of placing the operands in the local input memory of the CGRA. In this case, the compiler keeps an image of the local input and output memory to perform placement and routing. The compiler needs to know which operations will require the two operands from the local input memory of the CGRA. Within this step, the compiler will

know if there is enough space in the local input memory to place the operands. Otherwise, the compiler will indicate a constraint.

First, the compiler places the operands in the local input memory of the CGRA. If two operations use the same operand, it is placed in the local input memory only once. The order to place the operands is the order in which they are written in the RPN sequence. The compiler then reads again the RPN file and searches for the operations which are required to take both operands from the input memory.

In Figure 5.3, the mentioned process can be seen. The RPN sequence is related to the MVM algorithm. The sequence consists of a few addition and multiplication operations. First, the compiler places all the operands in the input local memory of the CGRA. The operations that have both operands in the input memory are saved into the *equation* array which can be seen on the right side of Figure 5.3. As it can be noticed, this function would not be needed in the case of MVM since the compiler might just read character by character and place the operands in the input memory. However, this solution was implemented to design a more efficient and general compiler.

This function needs the size of the CGRA, the number of total operations and the RPN file. The dimensions of the CGRA are required to know if the operands would be placed inside the local input memory and are also used for executing the next *schedule* function. The number of total operations is needed to create the mentioned *equation* array. Finally, the RPN file is required to read each character and to identify which of those are operands or operators.

Schedule is another function inside the information processing module. Basically, this function calculates which operand is the most used in the global equation. Such operand is placed inside the local input memory of the CGRA depending on how many times it is required in the global equation. The compiler then searches for the other operands (called neighbors) needed to perform the operation. The compiler places them together in the local input memory of the CGRA. *Route* function is executed to make a connection between each PE and each memory bank of the local input memory of the CGRA.

The parameters inside this function represent the dimension of the CGRA template, the number of operations which require both operands in the memory, the number of operands which are needed to implement the last operations and the mentioned *equation* array.

Inside *schedule* two different structures can be found. The first one, shown in Program 5.2 and is called *data*. It saves information about each operand, i.e., how many times it is required in the equation, the number of the other operands that are used with it (called neighbors) and the name of these operands. The operations and information that says if that operation has been analyzed are saved in the second structure (See Program 5.3).

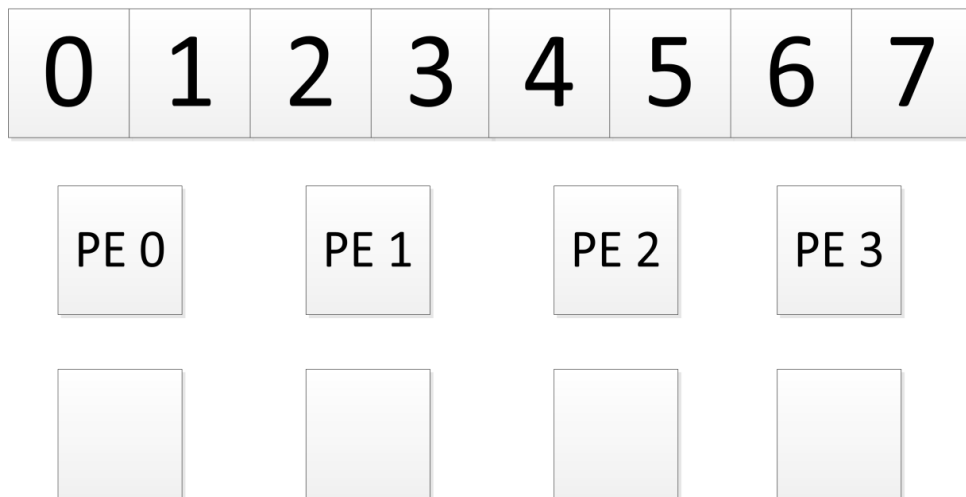


Figure 5.4 Enumeration of the memory banks and the PE of the first row

Once these structures are defined, the *schedule* function starts to fill them with *equation* array that is provided by *information* function.

```

1 typedef struct data{
2     char name;                //name of the operand
3     int times;                //how many times is used
4     int neighbour;           //how many neighbors has
5     char neighborName[numCharacter]; //saved the name of the neigh-
6     }data;

```

Program 5.2 Definition of data structure

```

1 typedef struct operations{
2     char e[1][4]; //save the equations
3     int used; //indicates if that operation has been used by PE
4 }operations;

```

Program 5.3 Definition of operation structure

Before going in more detail about how this function works, it is needed to explain the considerations that have been taken in account during the design. Due to the constraints in the hardware structure provided by SCREMA, each PE of the first row can only take the information from Left Up In A, Left Up In B, Up In A, Up In B, Right Up In A, Right Up In B (See Table 4.1). Hence, this fact limits the space where to put the operands inside the local input memory of the CGRA.

If we look at the numeration provided in Figure 5.4, it can be seen that if one operand is placed in the memory bank number one, it can be used only for PE 0 and PE 1. However, if the same operand is placed in the position number three of the local input memory of the CGRA, it may be used for more number of PEs. The operand which is most used for operation is placed by the compiler, so that the maximum number of PEs can use it.

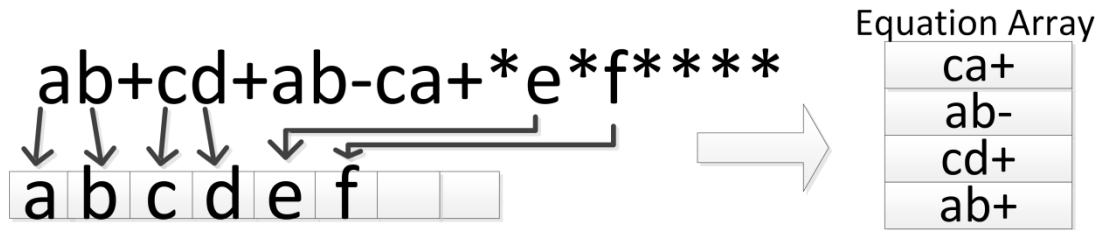


Figure 5.5 First step of information function

Opr	Times	N.Neib	Neighbor
a	3	2	b c
b	2	1	a
c	2	2	a d
d	1	1	c

Equation Array	
Equations	use
ca+	0
ab-	0
cd+	0
ab+	0

Figure 5.6 Different arrays which save information related to the operations and operands

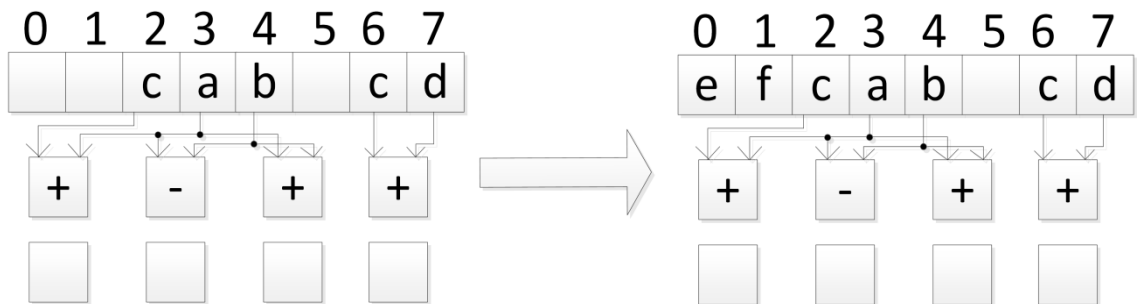


Figure 5.7 Third step of schedule function where the operands are placed in the local input memory.

If the operand is required three or more times, it will be placed in the memory bank number three since in that position more PEs may use it, as it was mentioned and considering the case if the size of the CGRA is 4x4 PEs. If that operand is just needed twice or once, it will be placed in the memory bank number one.

Once the most frequently used operand is placed, the compiler looks for the other operands to place the operator in the PEs. First of all, the compiler searches operation that includes the most frequently used operand. Then it looks for a place inside the local input memory of the CGRA to place the second operand of same equation. The compiler then places the operation in the nearest PE. *Route* function is executed to place the

equation in the PE and to implement the interconnections. If all the PEs are occupied, the compiler stops and indicates accordingly. Once that operation is placed, the PE is marked as used. When all the operands have been put inside the local input memory of the CGRA, the *schedule* function searches if there are other operations without having been placed in the CGRA. The compiler places the operands of these operations following the same steps. At the end it checks if all the operands are placed in the input local data memory of the CGRA.

In order to understand the mentioned explanation about the information module, one example is provided to show why this implementation has been designed. In Figure 5.5, shows the first step followed by the compiler. The compiler reads the RPN sequence character by character to identify which are the operands and which are the operators. The compiler then places all the operands in the local input memory of the CGRA. It saves the operations which take both of the operands from the memory, in the *equation* array which is shown on the right side of Figure 5.5. As it can be noticed, the operations which use ‘e’ and ‘f’ operands are not saved into the array, because one operand will be taken from the local input memory while the other one will be taken from the output of one PE.

The next step is shown in Figure 5.6 which consists of saving the information related to the number of times that one operand is used in the global equation. The names along with the number of neighbors are also saved into the array which is represented on the left side of Figure 5.6. In this case, the *equation* array is modified to indicate if one equation has been processed by the compiler or not, as it can be seen in Figure 5.6.

The compiler measures the number of times that each operand is used and then depending on this number the operand is placed in one or in another memory bank, as it was explained. As it can be seen in Figure 5.6, the operand which is used more times is “a” and the number of repetitions is equal to three. As it was mentioned, the operands will be placed depending on how many times operands are used. Since “a” operand is used three times, it means that it has to be used by three PEs, so it will be placed in the memory bank number three (See Figure 5.7).

After that, the compiler looks for the other operands (neighbors) are used with “a”. The first neighbor found is “c”. It is placed to the left of “a” operand in the input memory (See Figure 5.7). The compiler indicates that the equation “ca+” has been processed. Then, the compiler searches if there are other operations which use both operands. In this case “a” and “c” are used only once.

The next operand used with “a” is “b”. It is placed to the right of “a” in the local input memory of the CGRA (See Figure 5.7). The compiler searches if there are other operations which use both operands. In this example, it can be seen that there are two operations which use both “a” and “b” operands. Finally, when all the operations have been

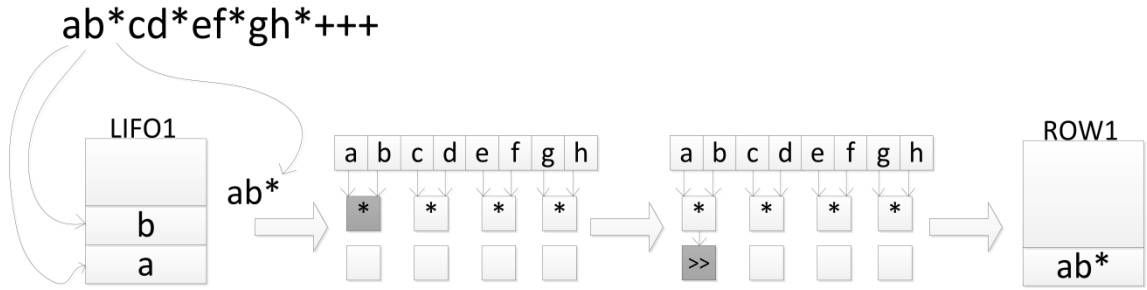


Figure 5.8 First step in the processing function for MVM algorithm

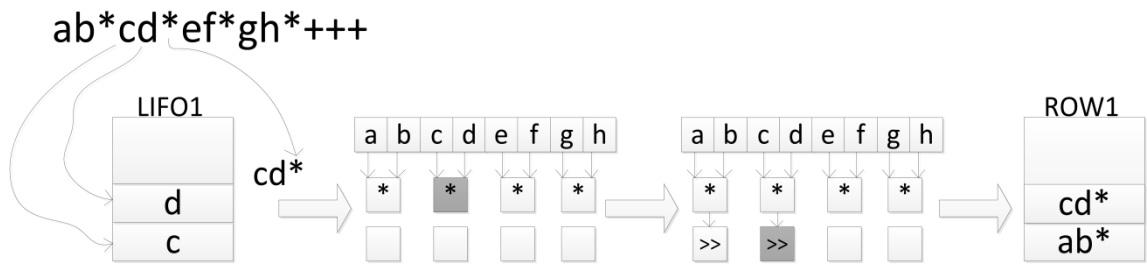


Figure 5.9 Second step in the processing function for MVM algorithm

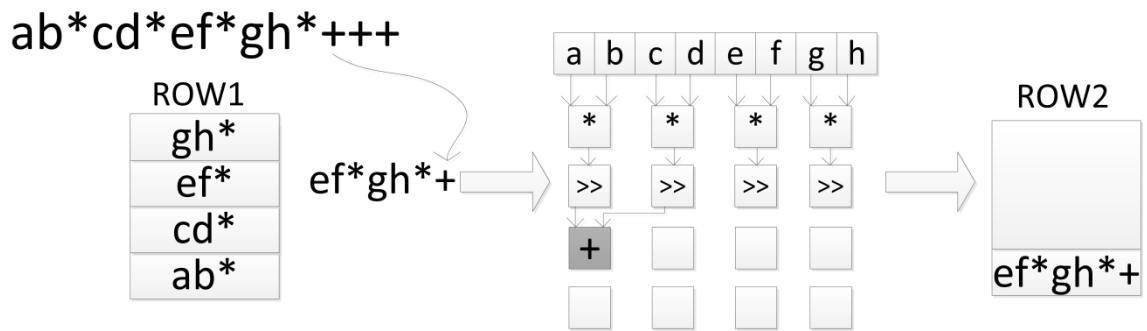


Figure 5.10 Implementation of second addition when MVM is processed

put in the CGRA, the compiler checks if all the operands have been placed in the local input memory, otherwise, it will place them inside the empty memory banks. In this case “e” and “f” are placed in the local input memory after the last step (See Figure 5.7).

5.5 Context Implementation

Context implementation module consists of functions, i.e., *processing*, *place*, *mainContext*, *preprocessing*, *canonical* and *finalPlace* functions.

The main processing configuration pattern is designed inside the CGRA template with *processing* and *place* functions. After all the operands have been placed in the input local memory of the CGRA, the compiler starts again reading the RPN sequence. It uses the RPN algorithm, explained in the section 5.1, to process the RPN information. The compiler then executes *place* function to make the P&R in the CGRA template. The result is replicated in the *mainContext* function to generate the CGRA to be instantiated. If the information from one PE is not used by another PE, it should be moved to the mentioned output memory, which represents the virtual local output memory of the CGRA. The data movement is performed by *finalPlace* function.

The parameters of processing function are the dimension of the CGRA template and the RPN file. The LIFOs (LIFO1, ROW0, ROW1, ROW2 and ROW3) and their pointers are used in this function.

Processing function implements the RPN algorithm (See section 5.1). The compiler reads the RPN sequence. If it reads an operator, it will look for the two operands of that operation. The operands will be saved into the LIFOs, starting by LIFO1 then by ROW0, ROW1 until ROW3. The compiler checks the position of each pointerLIFO to know where the operands are saved. If one LIFO does not have enough operands to implement the operation, the operands will be moved to other LIFO to be able to perform the operation.

Processing function starts reading the first character of the RPN file. If it is an operand, it will be saved into the LIFO1. The compiler continues doing this step until one operator is read. In that case, the compiler checks in which position is the pointerLIFO1. On the left of Figure 5.8, the implementation for MVM algorithm can be seen. If the pointer value is larger or equals to two, it means that there are enough operands to perform the operation. The compiler pulls the two last operands from LIFO1. It designs the operation and it verifies if that operation has already been placed in the CGRA. If the two operands have been pulled from LIFO1, it means that the operation was placed and routed in the information processing module (See section 0), because both operands are in the local input memory of the CGRA. If the compiler does not find the operation inside the CGRA, it will stop and it will give a comment to the user.

Once the compiler confirms that the operation is placed and routed in the CGRA, it checks if the operation is a multiplication. After one multiplication, a shift operation must to be executed in the CGRA. If the operation is a multiplication, a shift operation will be placed in the PE below (See Figure 5.8).

Finally, the output of the PE must to be pushed in a LIFO, such as the RPN algorithm was described in section 5.1. The result will be saved depending on the position of the PE. For example, if the PE is in the second row (See Figure 5.8), the result will be saved into ROW1 and if it is saved in the third row, the result will be saved into ROW2.

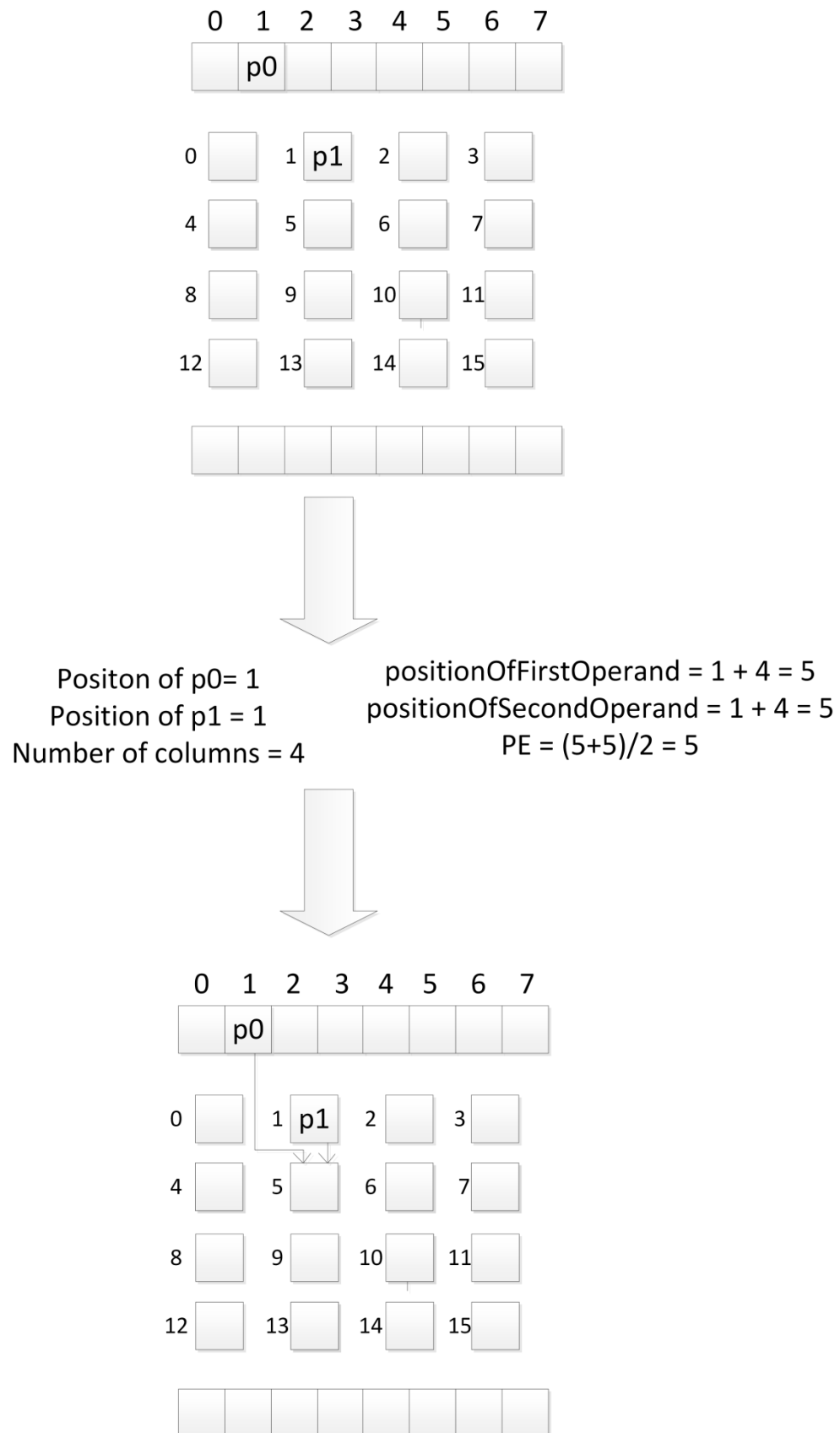


Figure 5.11 Explanation of the calculation of the best PE

The compiler then continues reading the RPN sequence and following the same steps as it can be seen in Figure 5.9. The result of this operation is also push into ROW1. The compiler will continue following the same steps until the addition is read.

In Figure 5.10, it can be seen that the compiler starts extracting the operands from ROW1 since in this case pointerLIFO1 and pointerROW0 are equals to zero.

When the two operands from one operation are not saved into *LIFO1*, the compiler executes *place* function to implement the placement and routing. *Place* function searches for the optimal location of the PE and makes the required connections.

Basically, *place* function calculates the position of each operand inside the CGRA and it calculates the optimal PE to perform the operation. If the ideal PE is occupied, *place* function searches for an empty PE. If all of them are used, the compiler stops and finishes the process.

The parameters of *place* function are the size of the CGRA template, the operation to be placed and routed, the LIFO in which *processing* function is working and the pointer of that LIFO.

Place function works in a direction depending on if the operands are in the input local memory or not. If both operands are in the input memory, the compiler calculates the ideal PE. All memory banks of the local input memory of the CGRA are assigned a number, as it was explained in section 0. The compiler performs the arithmetic mean of the two positions to calculate the ideal PE. Otherwise, the compiler performs the following Eq. 5.3

$$\begin{aligned}
 positionFirstOperand &= positionFirstOperand + col \\
 positionSecondOperand &= positionSecondOperand + col \\
 PE &= \frac{positionFirstOperand + positionSecondOperand}{2}
 \end{aligned} \tag{5.3}$$

Figure 5.11 represents one example to understand this implementation. The operand called p0 is placed in the memory bank numbered as one and the operand called p1 is placed in the PE numbered as one. If the Eq. 5.3 is executed by the compiler, the PE chosen is number five. On the bottom of the Figure 5.11 the routing is represented for this case.

The second step of *place* function is to make the interconnections. Each PE is identified by a number as it can be seen in Figure 5.12. The first option is to consider that the first operand is in the local input memory and the PE is in the first row. Table 5.1 shows different possible cases. The first column of Table 5.1 represents the number of the Bank

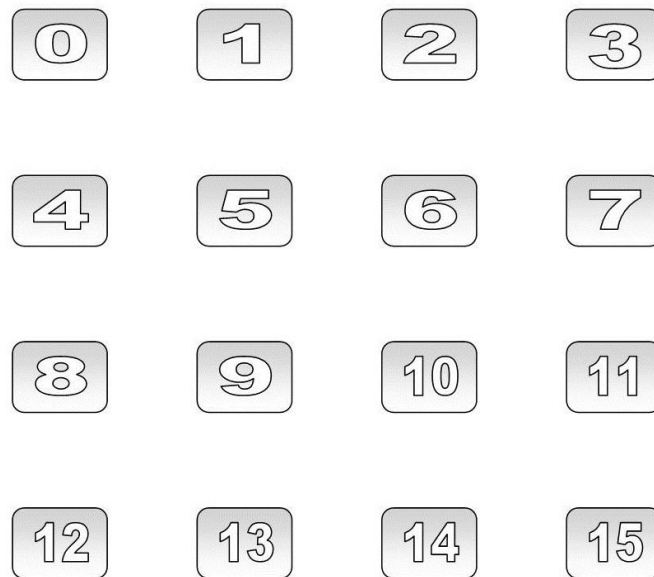


Figure 5.12 The identifier number for each PE

Position (BP), the second is the difference between the BP and the PE Position (PEP) and the third one represents the type of interconnection. If the table is analyzed, it can be seen that the connection *Up In A* meets in the same place with the result of *BP-PEP* when it is equal to the PEP. From that position, it can be seen that the rule would be the same for the rest of the connections. For instance, *Up Right In B* meets when the result of *BP-PEP* is equal to the number of the PE plus three.

In the PE numbered as one, the connections *Up Left In A* and *Up Left In B* must not be implemented in the CGRA because of the hardware constraints. When the PE is the last one in the row, *Up Right In A* and *Up Right In B* must not be implemented in the CGRA. Therefore, *place* function takes in account these considerations.

If the PE is in another row different from the first one and the connection has to be with the local input memory, *vertical A* and *vertical B* are the only possible connections. If *place* cannot realizes the connections, it will stop and finish the execution giving a comment to the user.

Suppose that the operand is not in the memory. Table 5.2 represents all the possibilities for the third row of PEs.

Table 5.1 Different connection cases considering one operand is in the memory and the PE is in the first row

BP	BP-PEP	Connection	BP	BP-PEP	Connection
0	$0 - 0 = 0$	Up In A	0	$0 - 1 = -1$	Up Left In A
1	$1 - 0 = 1$	Up In B	1	$1 - 1 = 0$	Up Left In B
2	$2 - 0 = 2$	Up Right In A	2	$2 - 1 = 1$	Up In A
3	$3 - 0 = 3$	Up Right In B	3	$3 - 1 = 2$	Up In B
4	$4 - 0 = 4$		4	$4 - 1 = 3$	Up Right In A
5	$5 - 0 = 5$		5	$5 - 1 = 4$	Up Right In B
6	$6 - 0 = 6$		6	$6 - 1 = 5$	
7	$7 - 0 = 7$		7	$7 - 1 = 6$	
BP	BP-PEP	Connection	BP	BP-PEP	Connection
0	$0 - 2 = -2$		0	$0 - 3 = -3$	
1	$1 - 2 = -1$		1	$1 - 3 = -2$	
2	$2 - 2 = 0$		2	$2 - 3 = -1$	
3	$3 - 2 = 1$		3	$3 - 3 = 0$	
4	$4 - 2 = 2$	Up In A	4	$4 - 3 = 1$	Up Left In A
5	$5 - 2 = 3$	Up In B	5	$5 - 3 = 2$	Up Left In B
6	$6 - 2 = 4$	Up Right In A	6	$6 - 3 = 3$	Up In A
7	$7 - 2 = 5$	Up Right In B	7	$7 - 3 = 4$	Up In B

Table 5.2 Different connections considering that one operand is not in the memory and the PE is in the third row

PEP	PEP-Position	Connection	PEP	PEP-Position	Connection
0	$0 - 8 = -8$	Vertical A	0	$0 - 9 = -9$	
1	$1 - 8 = -7$		1	$1 - 9 = -8$	Vertical A
2	$2 - 8 = -6$		2	$2 - 9 = -7$	
3	$3 - 8 = -5$		3	$3 - 9 = -6$	
4	$4 - 8 = -4$	Up In A	4	$4 - 9 = -5$	
5	$5 - 8 = -3$	Up Right In A	5	$5 - 9 = -4$	Up In A
6	$6 - 8 = -2$		6	$6 - 9 = -3$	Up Right In A
7	$7 - 8 = -1$		7	$7 - 9 = -2$	
8	$8 - 8 = 0$	Loop	8	$8 - 9 = 1$	Left Out A
9	$9 - 8 = 1$	Horizontal A	9	$9 - 9 = 0$	Loop
10	$10 - 8 = 2$		10	$10 - 9 = 1$	Horizontal A
11	$11 - 8 = 3$		11	$11 - 9 = 2$	
12	$12 - 8 = 4$		12	$12 - 9 = 3$	
PEP	PEP-position	Connection	PEP	PEP-position	Connection
0	$0 - 10 = -10$		0	$0 - 11 = -11$	
1	$1 - 10 = -9$		1	$1 - 11 = -10$	
2	$2 - 10 = -8$	Vertical A	2	$2 - 11 = -9$	
3	$3 - 10 = -7$		3	$3 - 11 = -8$	Vertical A
4	$4 - 10 = -6$		4	$4 - 11 = -7$	
5	$5 - 10 = -5$		5	$5 - 11 = -6$	
6	$6 - 10 = -4$	Up In A	6	$6 - 11 = -5$	
7	$7 - 10 = -3$	Up Right In A	7	$7 - 11 = -4$	Up In A
8	$8 - 10 = -2$		8	$8 - 11 = -3$	Up Right In A
9	$9 - 10 = -1$	Left Out A	9	$9 - 11 = -2$	

10	$10 - 10 = 0$	Loop	10	$10 - 11 = -1$	Left Out A
11	$11 - 10 = 1$	Horizontal A	11	$11 - 11 = 0$	Loop
12	$12 - 10 = 2$		12	$12 - 11 = 1$	

The algorithm would be changed in the case if the operands are not in the input memory. The connection *Up In A* meets the result of the operation when it is equal to -4, which is equivalent to the number of columns. This pattern is also followed by other connections. When the PE is the first element of the row, *Left Out A* cannot be implemented in the CGRA. The same occurs in the last PE of the row. In this case, the connection *Horizontal A* cannot be implemented in the CGRA because of the hardware constraints.

There is a special case, in which both operands are in the input memory, and the connections cannot be made since there are not any PEs that can perform such connections. In that situation, the *place* function of the compiler uses PEs to move the operands closer and to be able to implement the connections.

At the end of *processing* function, the compiler executes *finalPlace* function. It searches for the PEs which have not been connected to the other PEs or to the local output memory of the CGRA. *FinalPlace* moves the result of these PEs to the local output memory of the CGRA.

As it was mentioned earlier, there is a variable called *connectedPE*. *FinalPlace* function uses *connectedPE* to know if one PE has been connected to other PEs or to the output memory. Therefore, *finalPlace* function verifies that in the PEs that have performed an operation the value of *connectedPE* is equal to zero. If *connectedPE* is equals to one, *finalPlace* will search for an empty PE and performs the connection. In the case that the *finalPlace* function cannot find one empty PE, the compiler will stop performing P&R while leaving a comment for the user.

Once the compiler implemented the RPN file inside the CGRA template, *mainContext* function can be executed. This function asks the user how many times the CGRA template has to be instantiated in the desired CGRA. *mainContext* function copies the CGRA template and instantiates it in parallel to the number of times the user has indicated. If the number of replications is larger or smaller than the quotient between the number of columns of the final CGRA and the number of columns of the CGRA template, the compiler comments to the user. After this check, the main processing context is designed by the compiler.

The implementation of this case by MVM algorithm is depicted in Figure 5.13. The user wants to implement MVM algorithm on a 4x8 PE CGRA. First, the user will give the RPN sequence of 4th order of MVM. Once the compiler finishes performing the P&R on the CGRA template (CGRA on the left in Figure 5.13), it asks the user the dimen-

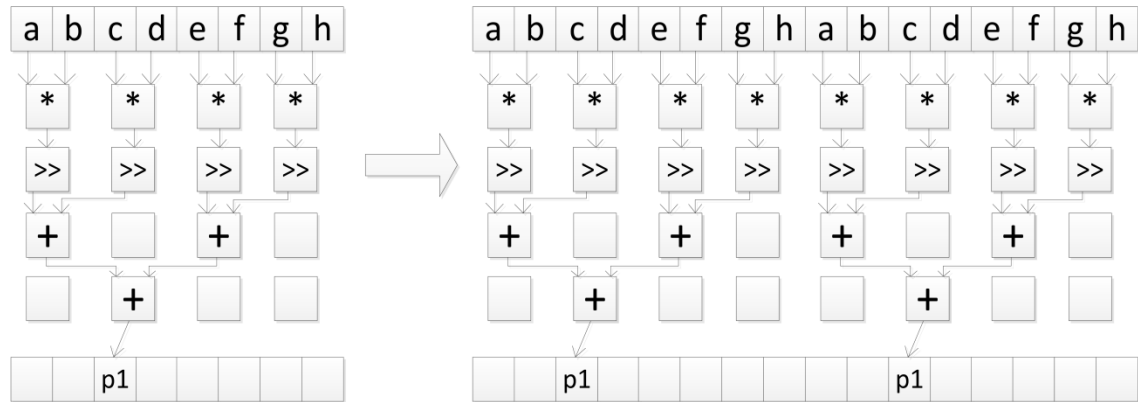


Figure 5.13 Implementation of the *mainContext* function using MVM for a 4x8 PE CGRA

sions of the CGRA to be instantiated. The user indicates 4x8 PE CGRA. The compiler instantiates the CGRA from the CGRA template which can be seen on the right side of Figure 5.13.

It was mentioned earlier that after one multiplication the compiler implements a shift operation which uses the shift amount from the immediate registers inside the PE. For that reason, one extra context is designed just to load the shift amount into the immediate registers. The compiler designs this context after the main context and before the pre-processing or canonical context. *immValues* function is responsible for generating this context. It checks the instantiated CGRA to know if there are PEs that have implemented a shift operation. When *immValues* finds one PE with a shift operation, it saves the number of such PEs and it continues searching for more PE. Once it finishes inspecting all the CGRA, it will implement MOVE To Immediate (*MOVTI*) operation in the PEs which have performed a shift operation. *MOVTI* moves the shift amount to the immediate registers. *immValue* function then implements a *Vertical A* connection to receive the shift amount data from the local input memory. This new context is called immediate values context.

Once the main processing context and the values context have been implemented by the compiler, it provides two design paradigms to design new contexts: canonical context or preprocessing context. Canonical context performs addition operations to add the partial products in the MVM algorithms. The canonical contexts are designed by *canonical* function. It also provides the possibility for the user to change the size of the CGRA template which allows a large range of design possibilities.

The user can see where the results will be saved in the local output memory of the CGRA. The user can decide the positions of the output memory that have to be added in the CGRA. The compiler generates the context and asks from the user if the CGRA template has to be replicated. . Suppose the user wants to implement an MVM algo-

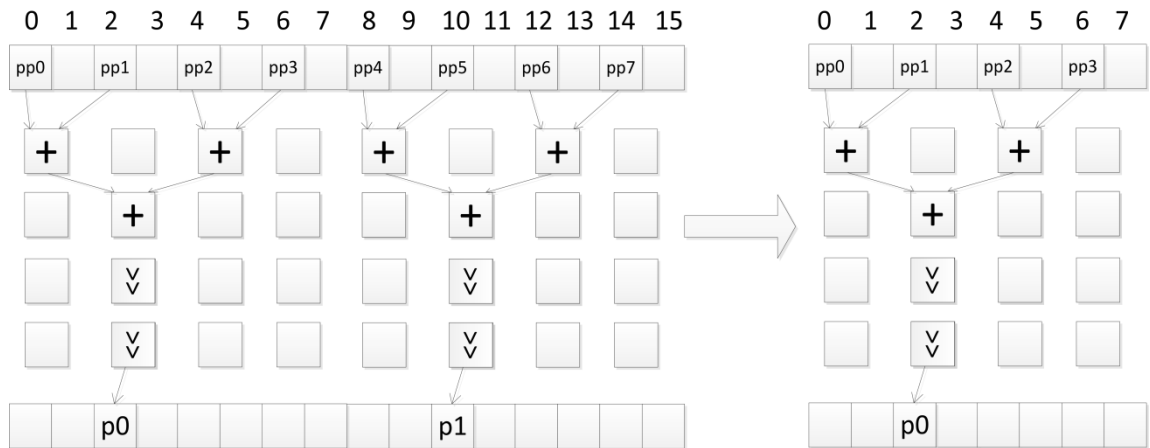


Figure 5.14 Implementation of canonical context in a 4x8 PE CGRA using a 4x4 PE CGRA template

rithm on a 4x32 PE CGRA. The user selects a *canonical* context. Since the size of the CGRA is 4x32 PEs, the number of memory banks in the local input memory of the CGRA has to be twice the number of PE columns in the CGRA. However, if the addition follows a symmetrical shape, it may be designed with this option.

Figure 5.14 depicts how *canonical* function works. Suppose that the instantiated CGRA is a 4x8 PE CGRA as it can be seen on the left part of Figure 5.14. The results of the first context are saved in the zeroth, second, fourth, sixth, eighth, tenth, twelfth and fourteenth memory banks and have to be added. The user will be asked for the size of the CGRA template. As a response, the user introduces 4x4 PE CGRA, because the result follows a symmetrical shape. Then, he/she introduces the number of memory banks to be added, and in this case, the user just introduces 0, 2, 4 and 6. Then, the CGRA template is replicated twice. In Figure 5.14 several PEs with delays are shown. They are implemented automatically by this function.

If a larger order MVM is processed over a smaller order (rows x columns) of PEs, then a preprocessing context is required between a processing context. After a processing context, the results are stacked on each other in the local output memory and another context is needed to place the result in an aligned order for further processing. For that reason a preprocessing context is designed by the compiler. It consists of chained delays depending on the amount of data to be processed.

Figure 5.15 illustrate a preprocessing context. The length of the delay chain depends of the amount of data to be processed by an already implemented context. The compiler asks the user for the length of the delay chain. Once the function knows the number of delays, it starts checking the positions where the results were saved in the output memory. For each position, *preprocessing* function designs the context. It inserts one

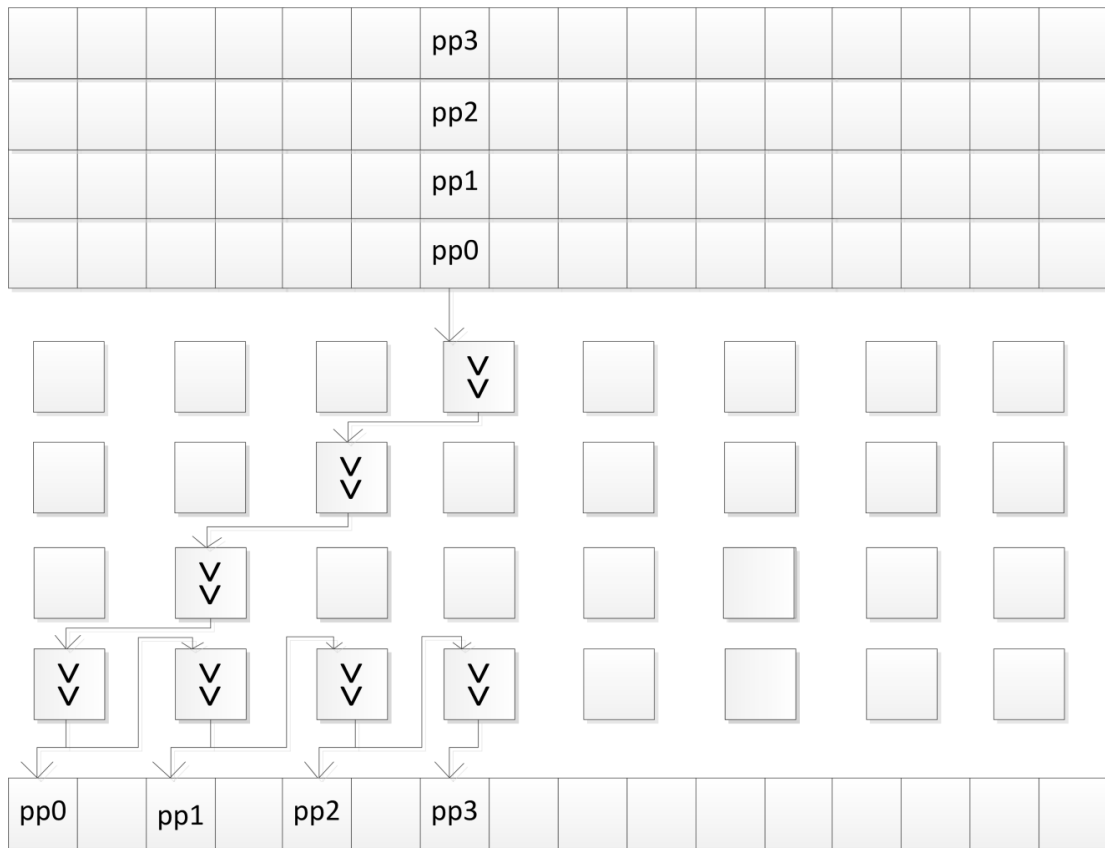


Figure 5.15 Example of pre-processing context

delay in each row to move the data until the last row. In the last row preprocessing function implements the number of delays the user indicated. Notice that the number of delays will be the same for each position of the local output memory.

5.6 Mask Delivery

Mask delivery is responsible for generating the two output files: VHDL file and C header file. It is composed of several functions, i.e., *initialization*, *parameters*, *masks*, *analyze*, *draw* and *header*.

Initialization is a function that is executed by the main program and is responsible for initializing the parameter-matrices of the CGRA template. This function needs the number of rows and columns of the CGRA to be instantiated and, also requires the VHDL file. It executes other functions. The first one is called *fputs*, and basically, it prints one value into the file. It writes the constant values of the VHDL file. *Parameters* designs the CGRA template parameter matrices (See section 4.1.1). *Parameters* function requires the name of the matrix, the number of rows and columns of the final CGRA, the data to be placed in the matrix and the VHDL file. The compiler designs the

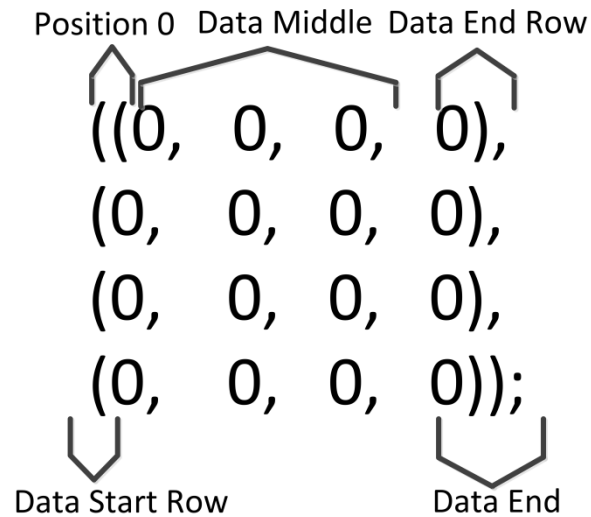


Figure 5.16 Example of a matrix generated with parameters function and the different kinds of data.

rest of matrices executing the *mask* function. Such function needs the same variables as *parameters* function. There are two functions (*parameters* and *mask*) to perform the same work. This is due to the different size of the matrices which means that the data saved inside them has to be treated in different conditions. In Figure 5.16 and Figure 5.17, they can be seen the small and large parameter-matrices, respectively.

The details related to the parameters and mask functions in the compiler construction are as follows. The parameter-matrices are represented as two dimensional array due to they save string values. A string in C language is an array of characters. Each position of the two dimensional array saves a character array. In parameters function there is a variable called *MaxPosition*. It defines the dimension of the small parameter-matrix. In

Figure 5.16, it can be seen that in each row there are the same number of columns as in the CGRA. If that value is multiplied by the number of rows, the *maxPosition* is established. Inside the function some variables called *dataMiddle*, *dataEndRow*, *dataStartRow* and *dataEnd* are declared and initiated. The value of each string is different depending on the position of the PE.

Mask function is used to design the parameter-matrices with larger dimension than the ones explained before. The implementation works as *parameters* function. In this case, the matrices are also two dimensional string arrays. The unique difference is that the sizes of these matrices are bigger. In this case, the maximum number of contexts is taking in account since it will represent the columns of the matrices. Such value is equal to 16. The calculation of *maxPosition* can be seen in the Eq. 5.4 which is the result of multiplying the size of the CGRA by the maximum number of context.

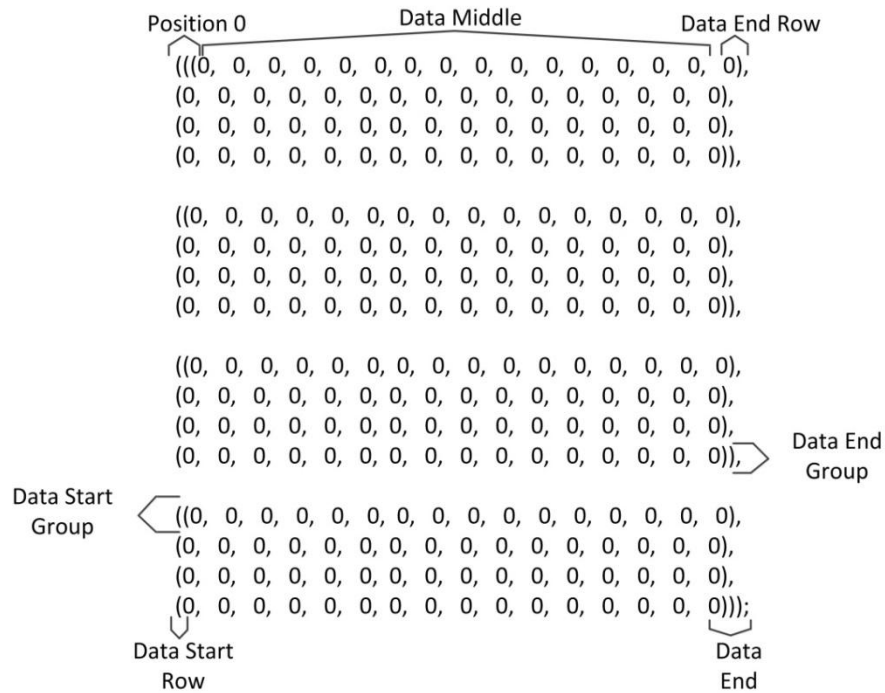


Figure 5.17 Example of a matrix generated with mask function and the different kinds of data

$$\text{maxPosition} = (\text{MaxNumContext} + 1) \times \text{row} \times \text{col} \quad (5.4)$$

The matrices which will be generated by *mask* function, have different shapes than the implemented by parameters function. In this case, they consist of a group of matrices. The number of matrices in each group depends on the number of rows of the CGRA. This aspect can be seen in 4.1.1.

Analyze function also belongs to the mask module. It is responsible for reading the data from the CGRA and modifying the matrices of the VHDL file (See Section 4.1.1) depending on the new and old values. This function is executed every time one context is designed by the compiler. It needs the dimension of the final CGRA to work correctly.

The implementation of *analyze* function is divided in four independent steps. The first one is related to the input A of the PE. *Analyze* function reads all the PE of the CGRA to be instantiated and it checks if the value of the input A is different from zero. If the value is different, it examines if that number is already placed in the *crema_mux_mask_A* matrix. In that case, the number is not saved in the matrix, it searches for an empty place, which means a zero inside the matrix. It then writes the new value in the matrix. Inside of this steps, *crema_cfg_ic_depth_A* matrix is also modified depending on the number of different connections are made for each PE, as it was explained in Section 4.1.1.

The second step is the same but with input B of the PE. Therefore, in this case *analyze* function checks all the input B of every PE. It modifies *crema_mux_mask_B* matrix and *crema_cfg_ic_depth_B* matrix by the same steps as it was described earlier.

Analyze function also examines the new values related to the operation for each PE. It searches for new types of connections and it writes them in the *crema_cell_mask* matrix. It also counts how many different connections are implemented by each PE. Depending on these values, *analyze* function modifies *crema_sel_width* and *crema_cfg_width* matrices. Finally, *analyze* function modifies *crema_mul*, *crema_add*, *crema_shifter*, *crema_imm* and *crema_sub* writing the binary parameter TRUE in the positions that represent PE which have performed one of the previous operations.

Draw function is responsible for writing the VHDL file with the data from the matrices. It needs the dimension of the CGRA to be instantiated. This function is executed just at the end when all the contexts are designed. *Draw* function writes the constant values of the VHDL file, it reads the values saved inside each matrix of the VHDL and it creates the file with these values.

The last function implemented was *header*. It consists of reading the configuration of each PE for each context and generating the C header file by the steps described in Section 4.1.2. It is executed by the compiler at the end of all processes. This function needs the size of the CGRA to be instantiated, the C header file and the project name.

As it was mentioned earlier, each PE has one array called *contextPE*. It saves the information related to each context. The dimensions of the arrays are equal to eight columns and the number of rows depends on the number of contexts that are implemented by that PE. The Table 5.3 represents the information saved in each column. The position number zero represents the number of context. The positions one, two and three represent the identifier of the connection for each input and the identifier of the operation, respectively. Then, the fourth place saves the code for the *input A*. The mentioned code depends on the number of different connections that are implemented by that PE. The sixth and seventh represent the code for *input B* and the code for the operation to be performed, respectively. Finally, the last position indicates if such PE in that context uses the immediate register.

Table 5.3 Different data saved into *contextPE* array

Position 0	Position 1	Position 2	Position 3	Position 4	Position 5	Position 6	Position 7
Number of context	Number of Input A	Number of Input B	Number of Operation	Code of Input A	Code of Input B	Code of Operation	Immediate Register Used

At first, the *header* function checks in which context the immediate registers are used and write one in the last column of the *contextPE* array. Then, depending on the different number of connections and operations, the function writes the code for each input and for each operation in the context array. Finally, it generates the configuration word by extracting the information from the *contextPE* array. The configuration word is a hexadecimal number but the information saved into the *contextPE* arrays is in decimal notation. Therefore, *header* function converts the decimal values to binary digits. When all the information has been transformed to binary, the function converts the binary digits to hexadecimal representation. Finally, *header* function will write every configuration word into the C header file.

6. TESTING AND EVALUATION

To verify the compiler behavior, different orders of Matrix-Vector Multiplication (MVM) were implemented on different sizes of CGRA. Specifically, the tested cases were 4th, 8th, 16th and 32nd order of MVM on CGRA sizes of 4x4, 4x8, 4x16 and 4x32 PEs. It chapter shows the designed steps and the results for these cases.

6.1 4th-Order MVM

The matrix equation for 4th order of MVM is represented in Eq. 6.1. The partial products p1, p2, p3 and p4 from Eq. 6.1 are shown in Eq. 6.2. To implement these partial products on SCREMA, the RPN sequence of only one partial product is sufficient. The RPN is showed in Eq. 6.3.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} \quad (6.1)$$

$$\begin{aligned} p_1 &= (a_{11} \times b_1) + (a_{12} \times b_2) + (a_{13} \times b_3) + (a_{14} \times b_4) \\ p_2 &= (a_{21} \times b_1) + (a_{22} \times b_2) + (a_{23} \times b_3) + (a_{24} \times b_4) \\ p_3 &= (a_{31} \times b_1) + (a_{32} \times b_2) + (a_{33} \times b_3) + (a_{34} \times b_4) \end{aligned} \quad (6.2)$$

$$\begin{aligned} p_4 &= (a_{41} \times b_1) + (a_{42} \times b_2) + (a_{43} \times b_3) + (a_{44} \times b_4) \\ ab \times cd \times ef \times gh \times + + + \end{aligned} \quad (6.3)$$

Two contexts are needed to implement 4th-order of MVM. The first one is the context which represents the RPN description. The second is the immediate value context. The approach can be seen Figure 6.1.

The compiler runs by using the following command:

```
./program 4mvm RPN.txt 4 4
```

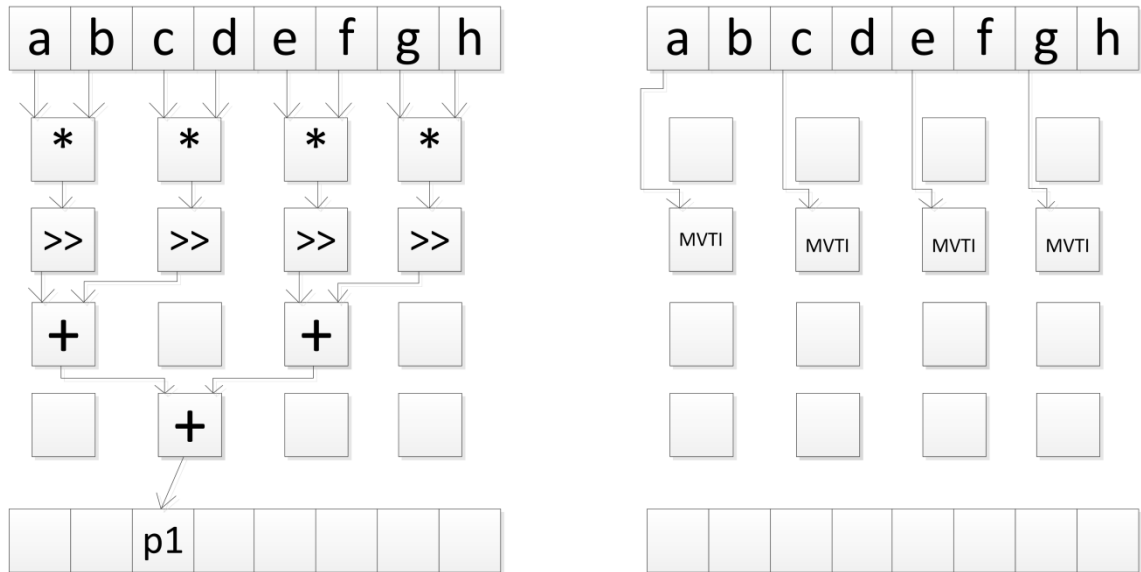


Figure 6.1 The two context of 4th-Order MVM in 4x4 PE CGRA

This command basically contains the project name, the RPN file and the size of the CGRA to be instantiated. The computer terminal shows the input memory and CGRA template. In this test case, the CGRA template coincides with the final CGRA. Then, the compiler asks the user the number of times to replicate the template. Then, the terminal shows the first main context and the value saved into the output memory. The compiler asks if the user wants to implement another context. The user must press the button 0 to finish the program. All the process can be seen in Figure 6.2. The left part of the figure shows the input memory where all the operands are saved. The right part shows the PE array. It also represents each PE with the inputs and the operations identifiers. The compiler requests user input from an interactive terminal. The input can be responded with numerics as option.

The previous example was for a 4x4 PE CGRA. If the same algorithm is implemented for 4x8 PE CGRA, then the interactive terminal can be seen as in Figure 6.3. As it can be noticed, the solution is the same but the only difference is that the result is on the half-left of the CGRA and the half-right is not used. The same occurs if 4th order of MVM is implemented in 4x16 and 4x32 CGRA.

6.2 8th-Order MVM

The matrix equation for this case is shown in the Eq. 6.4. The equations related to partial products are not represented because of their large length. For that reason, the compiler just needs the previous RPN (Eq. 6.3) to implement the main context. In this case the user will choose the replicate option as it will be shown below.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{68} & a_{69} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{pmatrix} \quad (6.4)$$

In Figure 6.5, the different contexts are shown that are needed to implement 8th-order of MVM on a 4x4 PE CGRA. This implementation consists of four contexts: the main context, the immediate context for shift operations, one preprocessing context with two delays and one canonical context which adds the memory bank number zero and two.

The terminal results are the same as in the previous approach. The only difference is that in the main menu, instead of selecting to end the implementation, the user selects a preprocessing context. When the compiler is designing this context, it asks the user to introduce the number of delays. To implement one equation of 8th-order of MVM on a 4x4 PE CGRA, that equation has to be divided in two partial products (p0 and p1 in Figure 6.6). For that reason, the user must choose two delays. Once the pre-processing context is generated, the compiler shows the main menu. In that case, the user must press 2 to select a canonical context. The canonical context adds the two partial products. During the process of designing a canonical context, the user introduces the size of the new CGRA instant. In this case, the same can be used, hence, the user will introduce 4 rows and 4 columns. Then, the compiler asks for the identifiers of memory banks that have to be added by the canonical context. The user introduces number 0 and number 2 because the two partial products are allocated in these memory banks. The user will press the button “s” to finish. In this test case, it is not needed to replicate the CGRA template. With these steps, the implementation of 8th-order of MVM completes.

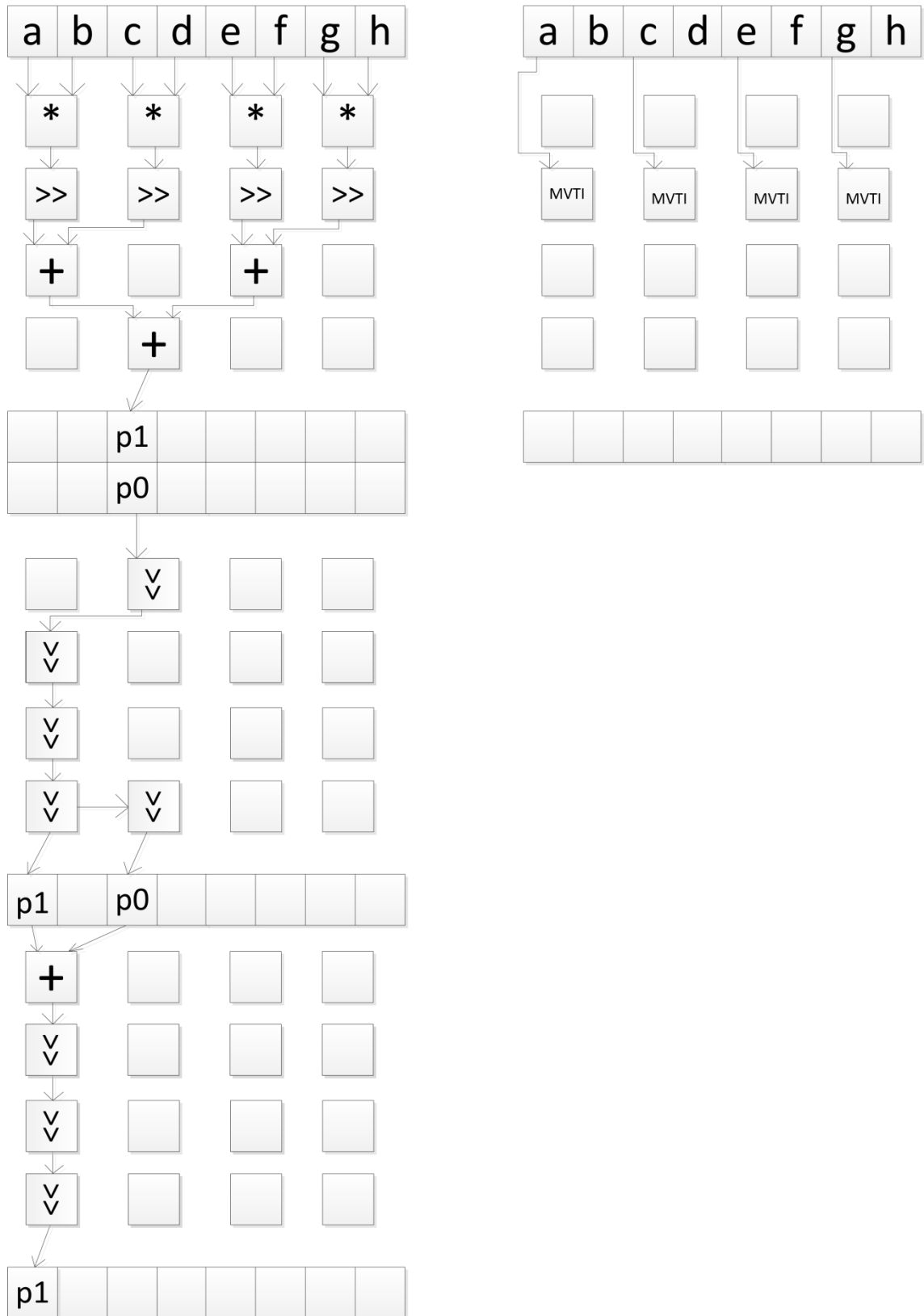


Figure 6.4 The contexts of 8th-Order MVM in 4x4 PE CGRA


```

What would you like to design now?
PRESS
1 Pre-processing
2 Canonical context
0 end
1
PRE-PROCESSING
How many delays would you like at the end?
2
PRE-PROCESSING
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
MEMORY OUT
0 pp1
1 0
2 pp0
3 0
4 0
5 0
6 0
7 0
PRE-PROCESSING
inA = 0 inA = 1 inA = 0 inA = 0
inB = 0 inB = 0 inB = 0 inB = 0
oper = 0 oper = 11 oper = 0 oper = 0
PRE-PROCESSING
inA = 7 inA = 0 inA = 0 inA = 0
inB = 0 inB = 0 inB = 0 inB = 0
oper = 11 oper = 0 oper = 0 oper = 0
PRE-PROCESSING
inA = 1 inA = 0 inA = 0 inA = 0
inB = 0 inB = 0 inB = 0 inB = 0
oper = 11 oper = 0 oper = 0 oper = 0
PRE-PROCESSING
inA = 1 inA = 3 inA = 0 inA = 0
inB = 0 inB = 0 inB = 0 inB = 0
oper = 11 oper = 11 oper = 0 oper = 0

What would you like to design now?
PRESS
1 Pre-processing
2 Canonical context
0 end
2
Could you introduce the size of the template would you like to use?
Row =
4
rowTemplate: 4
Col =
4
colTemplate: 4
Could you introduce the number of memory bank would you like to add?
0
Introduce another bank memory or press s to finish
2
Introduce another bank memory or press s to finish
s
CANONICAL CONTEXT
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
CANONICAL CONTEXT
inA = 1 inA = 0 inA = 0 inA = 0
inB = 7 inB = 0 inB = 0 inB = 0
oper = 1 oper = 0 oper = 0 oper = 0
CANONICAL CONTEXT
inA = 1 inA = 0 inA = 0 inA = 0
inB = 0 inB = 0 inB = 0 inB = 0
oper = 11 oper = 0 oper = 0 oper = 0
CANONICAL CONTEXT
inA = 1 inA = 0 inA = 0 inA = 0
inB = 0 inB = 0 inB = 0 inB = 0
oper = 11 oper = 0 oper = 0 oper = 0
CANONICAL CONTEXT
inA = 1 inA = 0 inA = 0 inA = 0
inB = 0 inB = 0 inB = 0 inB = 0
oper = 11 oper = 0 oper = 0 oper = 0

Would you like to replicate?
y/n
Please answer the question
Would you like to replicate?
y/n
n
MEMORY OUT
0 +pp1pp0
1 0
2 0
3 0
4 0
5 0
6 0
7 0
What would you like to design now?
PRESS
1 Pre-processing
2 Canonical context
0 end
0

```

Figure 6.5 Terminal results of 8th – Order MVM in 4x4 PE CGRA

The new terminal solutions are depicted in Figure 6.5. On the upper side, the results of implementing a pre-processing context can be seen. Below them, the canonical context is depicted, where different questions and the respective answers as shown. Finally, the final result is saved in the output memory. The implementation of the contexts changes if 8th-order of MVM is designed on a different CGRA size. Since the numbers of rows are not enough to represent all the operations of one equation, the main processing context must be split in two different contexts. This case occurs when 8th, 16th and 32nd orders of MVM are implemented in a 4x8 and 4x16 PE CGRA. The solution for 8th-order MVM in 4x8 PE CGRA is show in Figure 6.6.

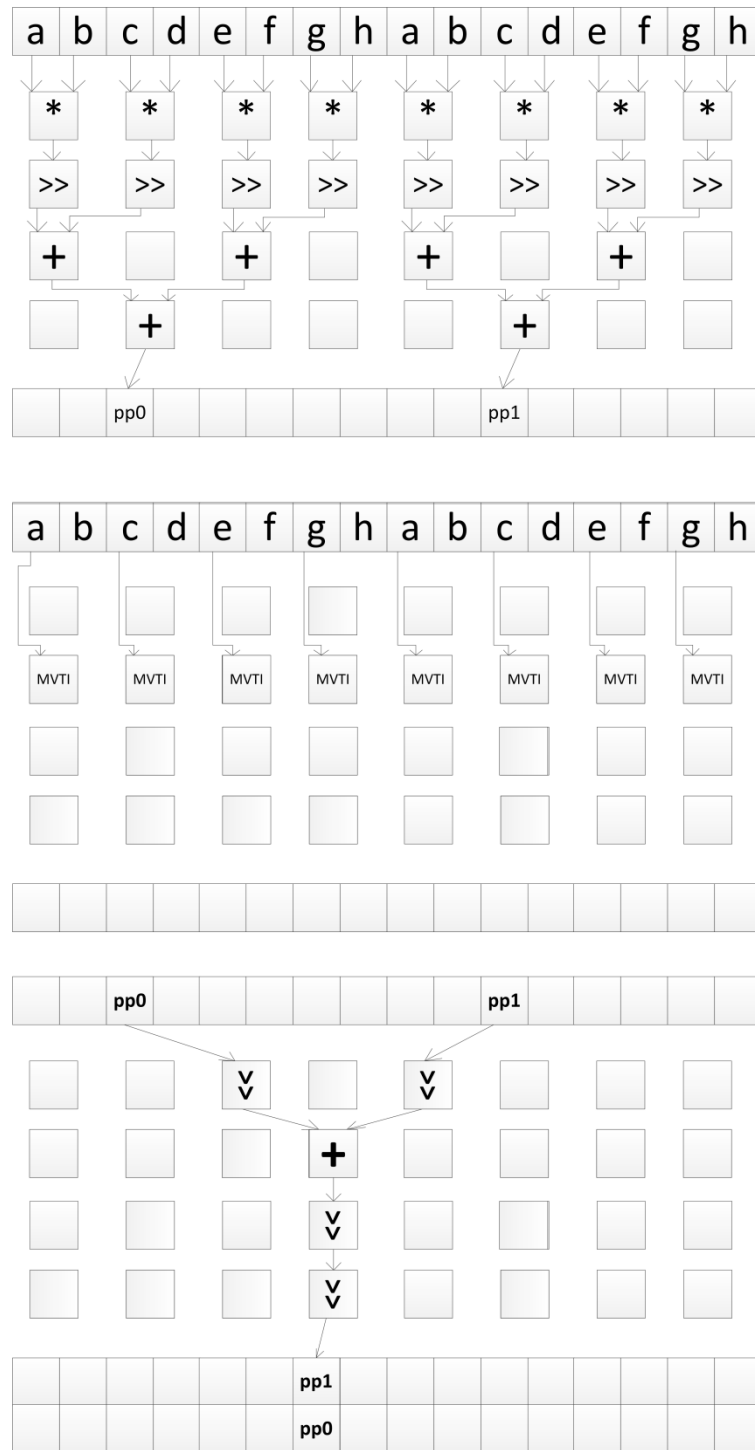


Figure 6.6 The main processing contexts of 8th-order MVM in 4x8 PE CGRA

From the figure, it can be observed that only three contexts are required for the implementation. The first context is generated from the RPN description of 4th-order of MVM. The solution is replicated to have the main processing context in a 4x8 PE

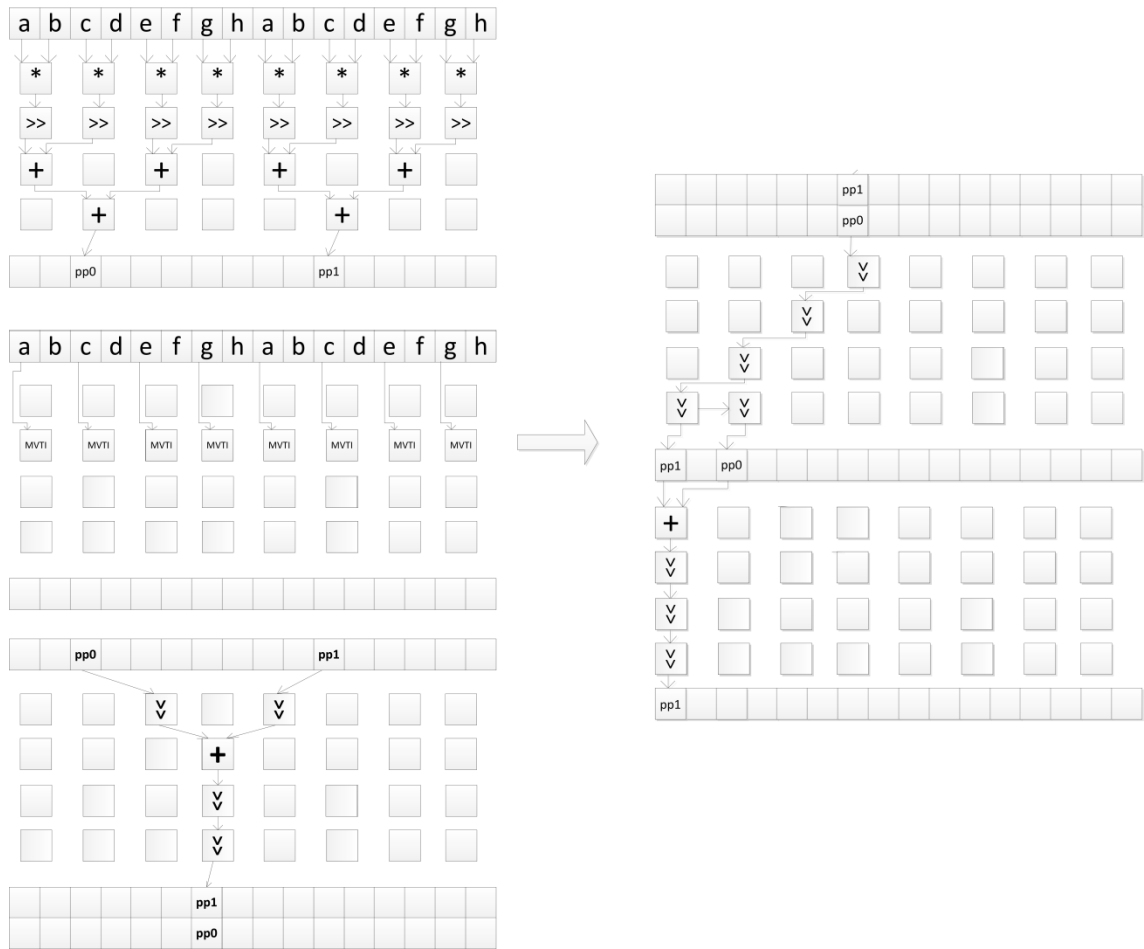


Figure 6.7 The context of 16th-order MVM in 4x8 PE CGRA

CGRA. From this context, the compiler designs the second context. It consists of moving the initial data to the immediate registers (immediate value context).

When one row of the first matrix in Eq.6.4 is multiplied with the vector, seven additions are needed to have one partial product. As it can be seen, those additions cannot be implemented in the first context. For that reason the third context is implemented. In this case, the user tells the compiler that the next context will be a canonical context. The canonical template may be for a 4x4 PE CGRA or 4x8 PE CGRA. If the first option is introduced, the solution must be replicated once.

When this 8th-order of MVM is implemented on a 4x16 or 4x32 PE CGRA, the solution is the same, the only difference is that the PE on the right part will not perform any operation.

6.3 16th-Order MVM

16th order of MVM was implemented for 4x4, 4x8, 4x16 and 4x32 PE CGRAs. All of the approaches start from a 4th-order of MVM description in a RPN file. If the CGRA sizes other than 4x4 PE CGRA is supposed to be generated from SCREMA template, the main processing context(s) of 4x4 PE CGRA needs to be replicated in parallel.

When 16th-order of MVM is implemented in 4x4 PE CGRA, four contexts are needed. These four contexts are the same as in the case of 8th-order of MVM. The differences are the number of delays in the preprocessing context and the number of additions. The number of delays is four because there are four partial products and since the number of PE columns is equal to four. These four partial products are added in the canonical context.

The other two contexts are one preprocessing and one canonical context. The equation obtained of multiplying one row of one matrix with the vector is divided into two parts (partial products), since in this case the number of PE columns is equal to eight. These two partial products have to be added in the canonical context. However, they are stacked on each other and hence they cannot be added together without a pre-processing context. The delay of this preprocessing context is equal to two since there are two partial products. After this context, the results are placed in the position zero and two of the memory output. These two positions are introduced to the compiler to design the canonical context.

When this order is implemented in 4x16 and 4x32 PE CGRA, only three contexts needed with the same shape as in 8th-order of MVM.

6.4 32nd-Order MVM

The implementation of 32nd-order of MVM in 4x4 PE CGRA consists of six contexts. The equation related to the multiplication of a row of matrix with a vector is divided into eight partial products since the number of columns of CGRA is equal to four. The addition of these eight partial products cannot be implemented in four contexts. After implementing the same four contexts as in 16th-order of MVM, the partial products are reduced to two. The compiler must add these two last partial products that are stacked in the output memory. For that reason, the compiler implements a pre-processing context and a canonical context.

When 32nd-order of MVM is implemented in 4x8 PE CGRA, it can be understood that the equation is divided into four parts. In this situation, five contexts are needed, the same as in 16th order of MVM in 4x8 PE CGRA. Four delays are introduced by the user in the preprocessing context. When the canonical context is being designed, there are

two options, to implement the additions directly in a 4x8 PE CGRA or to use a CGRA template of 4x4 PE and then replicate the solution.

Five contexts are needed when 32nd-order of MVM is implemented in 4x16 PE CGRA, because in this case the equation will be divided into two parts. The pre-processing context will have two delays and the canonical context will add these two parts.

In the situation of implementing 32nd-order of MVM in a 4x32 PE CGRA, the number of contexts needed are equal to three. This solution is the same as in the previous cases in which the order of MVM is the same as the number of PE columns.

7. CONCLUSIONS

In this thesis work, a compiler design for a template-based scalable Coarse-Grain Reconfigurable Array (CGRA) called SCREMA is presented. The SCREMA-template generates application-specific accelerators. This compiler framework replaces the existing GUI tool for a general case of Matrix-Vector Multiplication (MVM). The compiler performs automatic placement and routing of different orders of MVM on different sizes of the CGRA using the Reverse Polish Notation (RPN) description of the application. The compiler provides two configuration pattern templates to design other contexts.

The data flow graph of the compiler is composed of three different modules: Information Processing, Context Implementation and Mask Delivery. The first module of the compiler is used to extract the information related to the operations to be performed in the CGRA. In this step, the operands are placed into the input local data memory of the CGRA template in a read-frequency priority mechanism.

The configuration patterns which are part of an accelerator design are performed in context implementation module. There are three different contexts that the compiler can design. The first one is the main processing context designed from the RPN sequence of the target application. To implement this context a LIFO-algorithm is followed by the compiler which reads the RPN sequence character by character and stores them in a LIFO. In the case one operator is read, the compiler extracts the two last operands saved in the LIFO and generates the operation which is placed in the CGRA. The compiler finds the optimal PE where to place the operation. It connects automatically the operands with the PE.

The second context is a preprocessing context. Basically, the compiler designs delay-chains of different lengths to align the data in the local data memories of the CGRA for further processing..

The third context template is called canonical context which performs additions of the products computed by the main processing context.

Mask delivery is related to the generation of the output files. The output files are a VHDL file and a C header file. The VHDL file contains identifiers of the operations to be performed by each PE. The identifiers of the interconnection are also written in this file. The C header file contains the configuration stream related to each context.

The compiler framework has been tested for Matrix Vector Multiplication (MVM) algorithms of the orders of 4th, 8th, 16th and 32th by mapping them on the CGRA of sizes 4x4, 4x8, 4x16 and 4x32 PEs. These cases are designed from 4th-order MVM RPN description used as basis. The compiler generates the first CGRA instance which consists of 4th-Order MVM on 4x4 PE CGRA. The user may replicate such instances in parallel to make instances for other sizes. When the order of MVM is larger than the number of rows of the CGRA, extra contexts have to be defined because the partial product of MVM cannot be added by the main processing context. In such cases, preprocessing contexts are designed to align the partial product and a posterior canonical context which adds those products.

7.1 Future work

The large challenge to design a general compiler for SCREMA makes this thesis a first step towards finding an ideal and optimum compiler. For that reason, numerous projects can be followed after this work:

- Implement the placement of computational and routing resources using data structures and algorithms. Those algorithms will have to design one reconfiguration pattern and test if that pattern is the optimal.
- Implement the interaction between the user and the compiler through a graphical user interface instead of the terminal.
- To give the opportunity to the user to distribute the operands into the input local memories of the CGRA. To modify the canonical context in the way that it performs other operation instead of only additions.
- This work can be extended for other kernels like Fast-Fourier Transform (FFT).

BIBLIOGRAPHY

- [1] M. B. J.-Y. M. B. Mei, "ADRES & DRESC: Architecture and Compiler for a Coarse-Grain Reconfigurable Processors," in *Fine- and Coarse-Grain Reconfigurable Computing*, 2007, pp. pp 255-297.
- [2] Y. Kim, R. Mahapatra and K. Choi, "Design Space Exploration for Efficient Resource Utilization in Coarse-Grained Reconfigurable Architectures," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on (Volume:18 , Issue: 10)*, pp. 1471 - 1482, Oct. 2010.
- [3] J. M.P.Cardoso and P. C.Diniz, *Compilation Techniques for Reconfigurable Architectures*, Springer Science, 2009.
- [4] D. C. Cronquist, P. Franklin, S. G. Berg and C. Ebeling, "Specifying and Compiling Applications for RaPiD," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, Napa Valley, CA, 15-17 Apr 1998.
- [5] C. Brunelli, F. Garzia and J. Nurmi, "A coarse-grain reconfigurable architecture for multimedia applications featuring subword computation capabilities," *Journal of Real-Time Image Processing*, Vols. Volume 3 Issue 1-2, pp. 21-32, March 2008.
- [6] F. Garzia, W. Hussain and J. Nurmi, "CREMA: A COARSE-GRAIN RECONFIGURABLE ARRAY WITH MAPPING ADAPTIVENESS," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Prague, Czech Republic, Aug. 31 2009-Sept. 2 2009.
- [7] J.Becker, R. Hartenstein, M.Herz and U.Nageldinger, "Parallelization in Co-Copilation for Configurable Accelerators," in *Design Automation Conference 1998. Proceedings of the ASP-DAC '98. Asia and South Pacific*, Yokohama, 10-13 Feb 1998.
- [8] M. Budiu and S. C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics," in *FPGA '99 Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, New York, 1999.
- [9] S. C.Goldstein, H. Schmit, M. Budiu, S. Cadambi and M. Moe, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 3, no. 4, pp. 70-77, Apr 2000.

- [10] J. M.P. Cardoso and M. Weinhardt, "XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, vol. 2438, S. B. Heidelberg, Ed., Springer, 16 Aug 2002, pp. 864-874.
- [11] W. Hussain, T. Ahonen and J. Nurmi, "Effects of Scaling a Coarse-Grain Reconfigurable Array on Power and Energy Consumption," in *System on Chip (SoC), 2012 International Symposium on*, Tampere, 10-12 Oct. 2012.
- [12] J. P. Kylliäinen, T. Ahonen and J. Nurmi, "General-Purpose Embedded Processor," in *Processor Design*, S. Netherlands, Ed., Springer, 2007, pp. 83-100.
- [13] C. B. a. J. F. Garcia, "A pipelined infrastructure for the distribution of the configuration bitstream in a coarse-grain reconfigurable array," in *Proceedings of the 4th International Workshop on Reconfigurable Communication-centric System-on-Chip*, Univ Montpellier II, July 2008.
- [14] "www.wikipedia.com," [Online].