# Description and Specialization of Coarse-grained Reconfigurable Architectures

**Dissertation**
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
**M.Sc. Julio Alexandrino de Oliveira Filho**
aus Recife, Brasilien

**Tübingen**
**2010**

|  |  |
|---|---|
| Tag de mündlichen Qualifikation: | 10.02.2010 |
| Dekan: | Prof. Dr.-Ing. Oliver Kohlbacher |
| 1. Berichterstatter: | Prof. Dr. Wolfgang Rosenstiel |
| 2. Berichterstatter: | Prof. Dr. Edna Natividade da Silva Barros, UFPE, Brazil |

# Danksagung

Ich möchte an dieser Stelle allen herzlich danken, die mich während dieser Zeit auf vielfältiger Weise unterstützt und die zum Gelingen dieser Arbeit beigetragen haben.

Mein besonderer Dank gilt Herrn **Prof. Dr. Wolfgang Rosenstiel** für die wertvolle wissenschaftliche Betreuung, die Bereitstellung einer exzellenten Arbeitsumgebung, sowie die Übernahme der Betreuung dieser Dissertation. Mit seiner Aufgeschlossenheit gegenüber neuen Ideen, seiner Bereitschaft für internationale Kooperationen und seiner wertvollen Kritik hat zu dieser Arbeit wesentlich beigetragen.

Frau **Prof. Dr. Edna Natividade da Silva Barros** danke ich für die Übernahme des Referats dieser Dissertation und ihre sorgfältige Begutachtung der Arbeit.

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter im Arbeitsbereich "Technische Informatik" am Wilhelm-Schickard-Institut für Informatik an der Universität Tübingen und ich möchte mich zunächst bei allen Kolleginnen und Kollegen an dieser Universität bedanken. Ganz besonders Dank vor allem dem CRC-Team: **Thomas Schweizer**, **Sven Eisenhardt** und **Tobias Oppold**, die durch zahlreiche Gespräche und Diskussionen hilfreiche Hinweise und Anregungen beigesteuert haben. Für die fruchtbare Zusammenarbeit danke ich noch den Studenten **Stephan Masekowsky** und **Anja Küster**. Für die aktive Revision und Korrektur dieses Dokuments danke ich noch Herrn **Dr. Tommy Kuhn** und **Andreas Bernauer**.

Für die finanzielle Unterstützung, bedanke ich mich bei dem **Alßan Program** - European Union Programme of High Level Scholarships for Latin America (E04D045457BR) und der **Deutschen Forschungsgemeinschaft** - Priority Programm 1148 on Reconfigurable Computing Systems (RO 1030/13-X). Ein weiterer Dank gilt Frau **Margot Reimold** für die administrative Unterstützung hinter den Kulissen.

Mein herzlichster Dank gilt meiner Frau **Suzanne Oliveira-Martens**, die durch sorgfältige Korrekturen und wertvolle Hinweise die Qualität dieses Dokuments deutlich erhöht hat. Ihr noch herzlichen Dank für die ausdauernde Unterstützung und das Verständnis während der arbeitsintensiven Phasen meiner Promotion.

Schließlich danke ich meinen Eltern, **Julio Oliveira** und **Selma Tavares**, die mich während meiner gesamten Studienzeit stets darin bestärkt und unterstützt haben, meinen Weg zu gehen.

Tübingen, January 2009

Julio A. Oliveira Filho

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# 1. Introduction

Computer systems left the realm of desktop PCs and became part of almost every day-to-day device, such as cell phones, personal assistants, microwave ovens, automobiles, airplanes, medical devices, musical instruments, and children toys, just to name a few. These so-called *embedded systems* affect every aspect of modern life, from communication to transport, from entertainment to surveillance. The broad applicability and the high-end nature of embedded systems make their design an ever changing challenge, a struggle for portability, low power consumption, low heat dissipation, low production cost, high performance, and flexibility after production.

To deal with these challenges, designers of embedded systems have available a large variety of computing elements from general-purpose microprocessors to application-specific integrated circuits (ASICs). Microprocessors have been the heart of personal computers and workstations for decades, and their use in embedded systems increases everyday. They offer flexibility through their versatile instruction sets, which allow users to ”program” the implementation of any computational task. ASICs, on the other hand, are dedicated hardware circuits tailored to a specific task. For a given task, dedicated circuits execute faster, are more reliable, require less area and costs, and are more power efficient than general-purpose microprocessors.

In the last two decades, a new class of computing elements has emerged, which aims at combining the flexibility of microprocessors and the efficiency of ASICs. These type of elements are referred to as *reconfigurable computing systems*. The hardware of reconfigurable devices can be adapted to an individual application (*configured*) after production. Reconfigurable devices are per definition more flexible than ASICs, and at the same time, they can achieve a higher efficiency than microprocessors [107][115]. The first technology for building reconfigurable systems was the field-programmable gate array (FPGA) [17]. FPGAs consist of an array of logic and I/O blocks surrounded by a mesh of routing channels that interconnects these blocks. Typically, FPGAs are fine-grained architectures that operate on bit-wide data types and use look-up tables as computing elements. Along the time, the family of reconfigurable computing systems diversified in terms of granularity, execution models, and mechanisms to implement reconfiguration. Each member of the family addresses specific needs in the trade-off between performance and flexibility.

Recent advances on reconfigurable computing systems led to the development of *coarse-grained dynamically reconfigurable architectures* (CGRAs). CGRAs employ an array of processing elements that, contrary to the look-up table based logic blocks of FPGAs, operate on word-wide data types. These devices have the following advantages in comparison to their fine-grained counterparts [55]:

- Coarse grained processing elements reduce the amount of configuration data. That allows devices to reconfigure faster and reduces area and power consumption of circuits that control reconfiguration.

- Routing channels transfer words, instead of bits. That reduces the need for control signals and corresponding configuration storage.
- Computing elements and routing channels in CGRAs are more area-efficient than those in FPGAs. Complex operators, such as ALUs, can be directly implemented in silicon, instead of constructed with several logic blocks and look-up tables.
- Coarse granularity is better suited for application mapping from a high-level programming language, such as `C`, since its computing elements correspond more closely to operations in these languages.

CGRAs also allow an extremely fast, cycle-by-cycle reconfiguration mechanism. In CGRAs, configuration data is stored in several *contexts*, and at any given time exactly one context is active. During reconfiguration, contexts are switched, i.e., a previously inactive, stored context becomes active. In CGRAs, frequent and fast reconfiguration is part of the regular execution: at each clock cycle, the system instantiates and executes only that part of the circuit that is needed.

This work deals with the design of coarse grained reconfigurable architectures; in particular, with the specialization of CGRAs towards a set of applications and with the description of CGRAs during the design phase.

## 1.1. Motivation

After the underlying technology is fixed, designers of embedded systems must optimize their designs; for example by specializing the underlying processor architecture. This approach applies if the designer knows *a priori* the set of applications that will run in the system. Specialization consists of modifying a generic instance of the architecture by:

- removing components that are not necessary for the target applications;
- changing the data type used in the system to provide more efficient or accurate computation;
- rerouting the communication channels between components to reflect the way the target applications access, store, and exchange data;
- incorporating computing units, such as custom operations or dedicated hardware, that are tailored for the execution of specific tasks.

This specialization retains the flexibility of the system for the considered set of applications while improving the system performance, implementation area, and power consumpution.

During the last decade, incorporating custom operations and dedicated hardware became an increasing trend in the design of several architecture types, such as FPGAs [56] and application-specific instruction processors (ASIPs) [60]. FPGA systems were tightly coupled to instruction set processors and gained fully dedicated circuits, such as hard-wired floating point multipliers. Concurrently, designers of ASIPs started integrating dedicated operations, called *custom instructions*, in the instruction sets of general-purpose processors [36][23][63]. This technique led to average speed ups of 3.4× in the execution time, and 3.2× less power consumption [110]. The specialization of FPGAs and ASIPs is nowadays an established technique, investigated by academical research, and supported by design methodologies and software tools. That is not the case for the design of coarse grained reconfigurable architectures.

There is a lack of techniques, methodologies, algorithms, and tools to conduct the specialization of CGRAs. Up to date, academic and commercial coarse grained architectures use a regular and repetitive arrays of elements of the same type. The reason is that these simple arrays can be easily scaled, and their functionality and costs can be simulated and estimated by existing commercial tools. The few specialization approaches in the literature consist of an exploration of the design space to find the correct size of the array, topology of the interconnection network, size of internal memory, width of the data type, external memory architecture, and connection to the environment around the array.

In particular, no academic or commercial CGRA platforms were proposed that consider custom instructions in the instruction set of their processing elements. Also, evaluations to determine the costs and benefits of applying custom instructions to the design of CGRAs are lacking.

Two factors can explain the hesitance of designers to specialize CGRAs with custom instructions: the need to modify the application mapping and the extra complexity introduced in the design phase. Custom instructions modify the way the architecture will be used to execute applications. Application mapping tools, their techniques, and algorithms have to be modified to recognize parts of the application that can benefit from the specialized hardware, and to map them accordingly. In this sense, any proposal to specialize CGRAs, including those that use custom instructions, must consider the concurrent development of architecture and compiler.

The specialization of CGRAs increases the complexity of the design phase because designers have to frequently rewrite, verify, and evaluate processing elements and FUs. Moreover, dedicated operations potentially induce arrays with different types of processing elements and irregular interconnection networks. Describing these arrays and modifying their elements with description languages such as VHDL, Verilog, and SystemC are difficult and error prone tasks. These languages were designed to describe ASICs and systems-on-chip, and cannot express specific concepts used for the design of coarse grained arrays, such as instruction set, reconfigurability and spatial arrangement of components. As CGRAs grow in complexity, their description demands the use of higher abstraction levels.

## 1.2. Objectives and proposed solutions

This work aims at developing methods, techniques, algorithms, and tools to describe and specialize coarse grained reconfigurable architectures during their design phase.

### 1.2.1. Description of coarse grained reconfigurable architectures

> This work proposes a new approach to describe coarse grained architectures at a higher level of abstraction.

A higher level of abstraction means the designer can describe concepts that are specific for the design of CGRAs, such as coarse granularity, instruction-set based functional units, context-based reconfiguration, spatial distribution of components, and scalability of the array and of the interconnection network. The ability to express efficiently these concepts makes the description of CGRAs easier, clearer, and more concise. Moreover, the implementation of more efficient software tools, such as simulators, estimators, and compilers, become easier. As a result, the designer can describe, modify, verify, and evaluate the design in a fast and efficient way.

By adopting higher level of abstraction in the description, the designer prepares the grounds for specialization. Higher level descriptions yields clearness and eases automation, and therefore, deal with the increasing complexity introduced by specialization tasks.

As a solution, this work introduces the *coarse grained architecture description language: CGADL.* CGADL has a clear semantic. Its innovative key features addresses specific concepts in the design of CGRAs:

**Coarse granularity of processing elements** CGADL's has keywords to describe explicitly coarse grained components, such as multiplexers, register banks, memory modules, etc.

**Instruction-set based functional units** CGADL's functional units are components with designer-defined functionality. The designer describes functions as instructions that are controlled and executed inside functional units. Complex behavior can therefore be described by assembling coarse grained components as well as by instantiating functional units.

**Spatial distribution of processing elements** With CGADL, the designer can attribute relative positioning information to the components in the architecture. So, it is possible to describe spatial relations, such as "the processing element to the right", or "the processing element at position (1,2) in the array". This simplifies the design of arrays with different types of processing elements, and the description of the interconnection network.

**Scalability** CGADL allows the description of parameterizable architecture templates, in which aspects, such as the number of registers in the processing element, and the array geometry, can be altered by changing a parameter setting.

**Network interconnection** Together with the spatial positioning system, this feature allows the designer to describe in a fast and concise way complex network interconnections.

**Multi-context reconfigurability** CGADL has constructs to describe context memories: units used to store the configuration information. The concept of reconfiguration is intrinsic to the semantic of these constructs.

CGADL is suitable for the development of software tools. As an example, this works builds a hardware complexity estimation tool that works with CGADL descriptions. This estimation tool fulfills two goals: first, it demonstrates that CGADL-based software tools, such as estimators, simulators, and compilers, can be implemented; second, it provides a fast and automated evaluation of the architecture template, early at the design phase.

### 1.2.2. Specialization of coarse grained reconfigurable architectures

This work uses custom instructions to specialize coarse grained architectures towards a set of applications.

The specialization process consists of a refinement of the architecture design by using the knowledge about a set of applications. So, for example, one may tailor functional units to execute application-specific operations. As a result, specialized arrays are smaller, faster, and less power consuming, when running applications from the targeted domain.

The specialization task requires some knowledge about the set of target applications. However, that doesn't mean that only specialists in the application field can carry out this task. Any methodology to specialize the architecture should state how this knowledge is extracted from the target applications, and how it is used in the design.

The specialization task affects both the architecture and the application mapping phase. Accordingly, the specialization methodology introduced in this work considers the co-development of architecture and compiler.

To achieve these objectives, this work proposes a design framework to integrate custom instructions in the instruction set of CGRAs's functional units. This framework defines:

- Methods and algorithms to extract, evaluate, and select clusters of operations that emerge regularly within the set of target applications. These clusters are selected based on the frequency with which they can be found in the applications. These clusters correspond to the knowledge acquired from the application domain.
- Software tools to automate the previously mentioned methods and algorithms.
- Techniques to transform theses clusters of operations in a datapath description of custom instructions.
- Methods that modify the application mapping tools (compiler) to identify, transform, and map clusters of operations to corresponding custom instructions.
- Techniques to modify the compiler process, such that, during the application mapping, clusters of operations corresponding to custom instructions are correctly transformed and mapped.

In this work, the design of a custom instruction corresponds to groups of operations that appear frequently in the set of applications in the same kind of execution pattern. If a custom instruction exists that implements this pattern, it can be used to map any group of operations with the same structure. The design of custom instructions produces a hardware datapath and annotations for the application mapping tool (compiler). The datapath description corresponds to the design of one or more custom instructions. The annotations allow the compiler to identify, in any application, groups of operations that correspond to these custom instructions.

## 1.3. Workflow and organization of this document

This dissertation starts, in Chapter 2, with a discussion about the established praxis for the design of coarse grained reconfigurable architectures. This discussion details the structure of CGRAs and their functionality, as well as that of their components. Basic concepts and terminology are presented, that will be used along this document. The final part of Chapter 2 is dedicated to the description and mapping of applications onto coarse grained reconfigurable architectures, since these activities are also affected by the proposals in this work.

Chapter 3 lists the most representative coarse grained architectures presented in the literature, and some industrial instances that made their way to the market. The focus of this chapter, however, is a review of the languages used to describe CGRAs, and of techniques to specialize these arrays during the design phase. This review delineates the limitations of actual design methodologies in these two aspects and points out the contributions in this work to overcome them.

The architecture description language CGADL and the design of custom instructions are discussed in the remaining chapters according to the role they assume in the design flow of coarse grained architectures, as depicted in Figure 1.1. The design of CGRAs starts with a generic model of the architecture and a set of target applications.

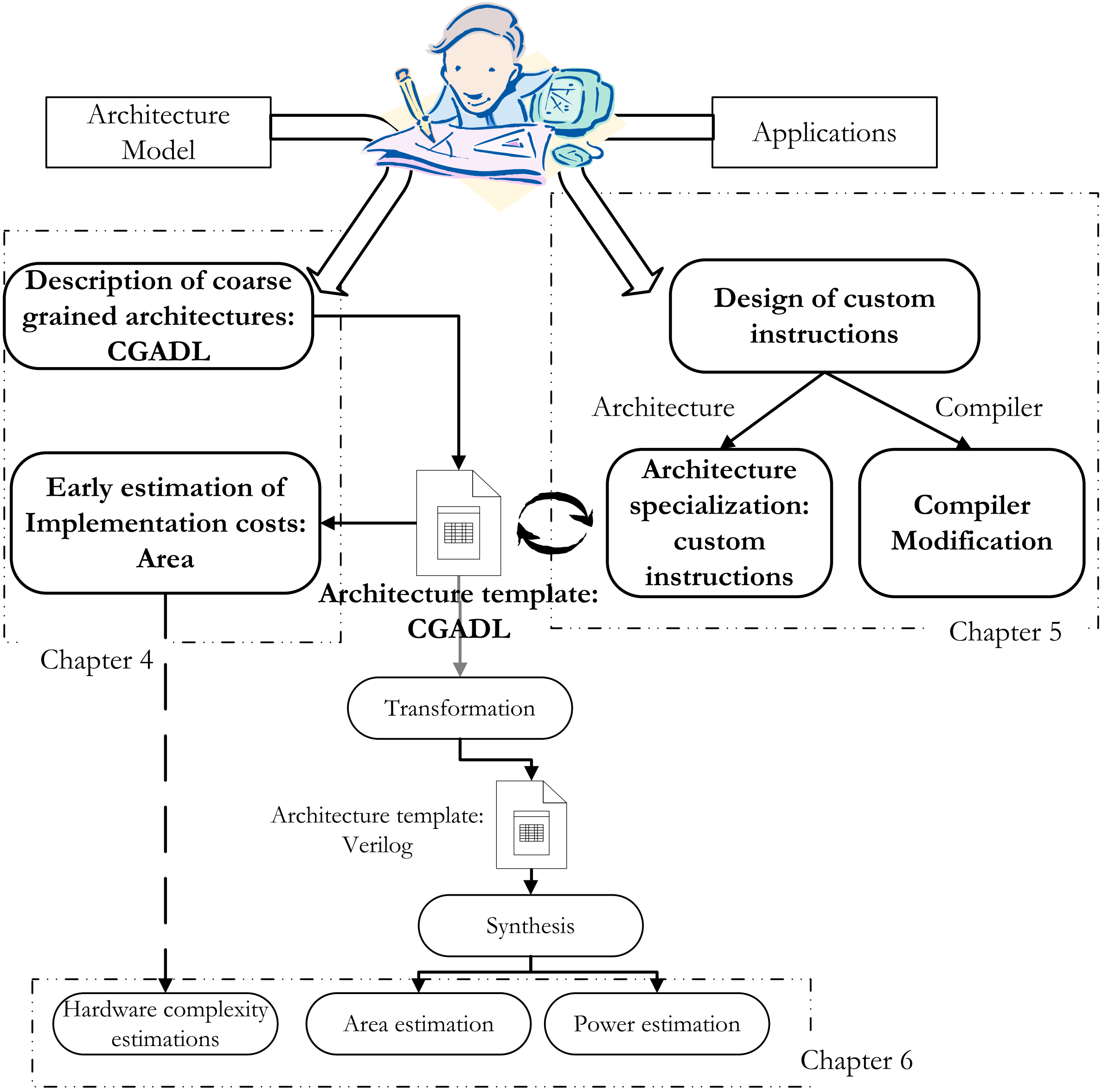

Architecture Model
Applications
Description of coarse grained architectures: CGADL
Design of custom instructions
Architecture
Compiler
Early estimation of Implementation costs: Area
Architecture specialization: custom instructions
Compiler Modification
Architecture template: CGADL
Chapter 4
Chapter 5
Transformation
Architecture template: Verilog
Synthesis
Hardware complexity estimations
Area estimation
Power estimation
Chapter 6

Figure 1.1.: Design flow proposed in this work for the development of CGRAs.

The generic model determines the basic structure of the architecture: an array of coarse grained processing elements surrounded by an interconnection network and input/output resources. From this model, the designer derives an architecture template: a detailed but flexible description of the array and its processing elements. The set of applications is used to design custom instructions. The designer uses these custom instructions to specialize the architecture and modify the compiler. At this point, the hardware complexity of a CGADL template can be estimated by using the software tool developed during this work.

The coarse grained architecture language CGADL, and the hardware complexity estimation tool are introduced and discussed in Chapter4.

The design of custom instructions for coarse grained reconfigurable architectures is discussed in Chapter5. The final parts of Chapters 4 and 5 present software tools developed with the ideas introduced in this research work. The first software tool implements the estimation method proposed in Chapter 4. The second tool captures parts of the specialization methodology presented in Chapter 5. These tools demonstrate the feasibility of techniques, methods and algorithms proposed here, as well as allowed automated experiments and generation of results.

The (CGADL) architecture template is the central point of the workflow, where the architecture specialization and the hardware cost estimation take place. After the template is described, dimensioned, and specialized with custom instructions, it is transformed in an equivalent Verilog description, which allows synthesis with commercial tools. This transformation is out of the scope of this thesis, and will not be discussed here. Synthesis transforms the Verilog description into a circuit netlist that implements the architecture array. Based on this netlist, the synthesis tool provides detailed and accurate estimates for silicon area and power consumption of the array and its components.

Chapter 6 presents, in two sections, the experiments and results obtained in this work. Section 6.1 compares the estimates produced by the CGADL-based estimation tool with those produced by a commercial synthesis tool. This comparison demonstrates that the evaluation of hardware costs can be done directly from a CGADL description, much earlier in the design phase, and without the need of synthesis. Section 6.2 uses the design flow in Figure 1.1 to specialize coarse grained arrays towards two sets of applications: the scalable OFDMA modulation scheme of the WiMax standard; and a set of applications used in automobile driving assistance systems. These experiments demonstrate the feasibility and effectiveness of the specialization method proposed here. Results provide a detailed evaluation for the impact of using custom instructions during the specialization of CGRAs.

# 2. Basics

Design teams, from academy and industry, started to develop coarse grained reconfigurable architectures about 15 years ago. Along this time, they produced different instances of these architectures, but their design workflow converged to a somehow similar methodology that starts with a model of the architecture and refines it down to a concrete instance. This chapter discusses the general lines of this methodology, and introduces, under this discussion, the basic concepts and terminology used in this dissertation. Examples for coarse grained architectures and particularities about their design phase will follow in Chapter 3.

This chapter is divided in two parts corresponding to the entry points in the design of coarse grained architectures (see Figure 1.1): the architecture model and the set of target applications. Section 2.1 explains the design of CGRAs, their structure, and functionality. Section 2.2 discusses how appications are represented and how they are mapped into the architecture.

## 2.1. Architecture

The development of CGRAs follows three steps, as depicted in Figure 2.1. At the first step, the designer conceives a generic model: an array of coarse grained *processing elements* (PEs) surrounded by a network interconnection, input and output resources, and memory blocks. This model is called *architecture model*. At the second step, the designer writes down the architecture model as a parameterizable description. This description is called *architecture template*. The template outlines the granularity, type and disposition of PEs, the possible network interconnections, and the organization of the memory components. Templates are flexible descriptions because they can be modified by adjusting the value of parameters. Parameters regulate certain characteristics of the architecture, such as the number of lines and columns (width and height) of the array, the number of available reconfiguration contexts, the number of internal registers within the PE, and the interconnection network. At the third step, an *architecture instance* is generated by fixing the value of each template parameter. The architecture instance is a well defined description of an architecture, which may be synthesized, evaluated, and/or simulated.

Following this design flow, Oppold presented a generic model for coarse grained arrays [94][96]. He wrote this model as a template in the hardware description language Verilog [97]. According to this model, the template can be *configured* at design time by modifying the parameters of its Verilog description, and processing elements can be *reconfigured* at execution time to perform different tasks. This feature of the model to be configured at design time and reconfigured at execution time accounts for its name: ***Configurable Reconfigurable Core*** (**CRC**). The CRC model enables simulation and synthesis; and thus, the evaluation of functionality, performance and implementation costs. The next subsections discuss the CRC model, the CRC template, and CRC instances in more detail.

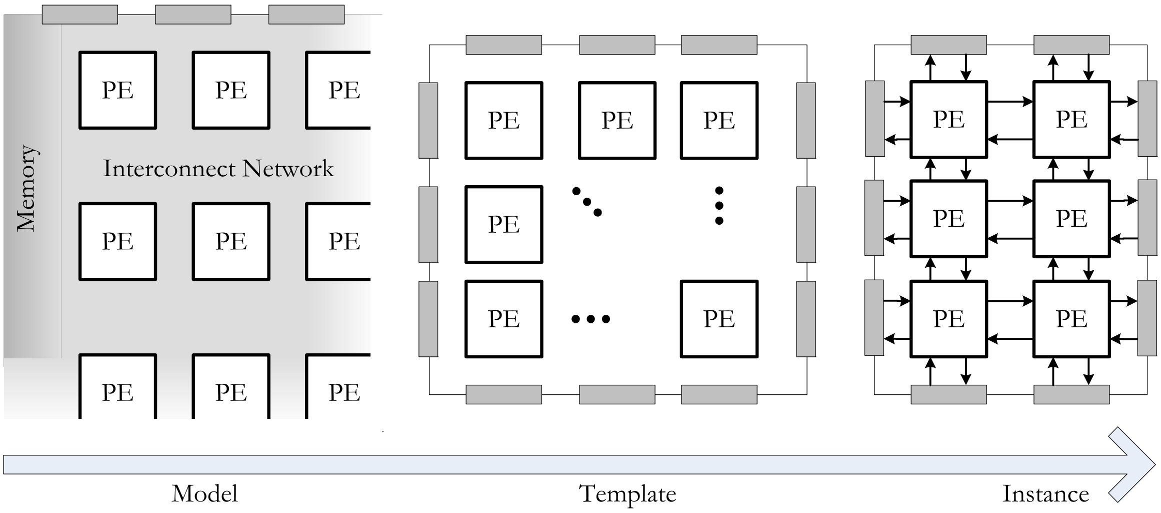


Figure 2.1.: General design flow for coarse-grained reconfigurable architectures.

### 2.1.1. The CRC Model

The CRC model is a generic base for the design and development of coarse grained arrays. It fixes only a few characteristics common to the major part of the actual CGRAs: the architecture is composed of processing elements (PEs) surrounded by an interconnection network, as depicted in Figure 2.2. The spatial organization of these PEs follows an array-like structure. The interconnection network defines how data transfer between PEs occurs. The CRC model also establishes the existence of input and output (I/O) resources in the array. Input resources transport data from the external environment to some or all processing elements in the array. Output resources make the data produced by PEs available to the external environment.

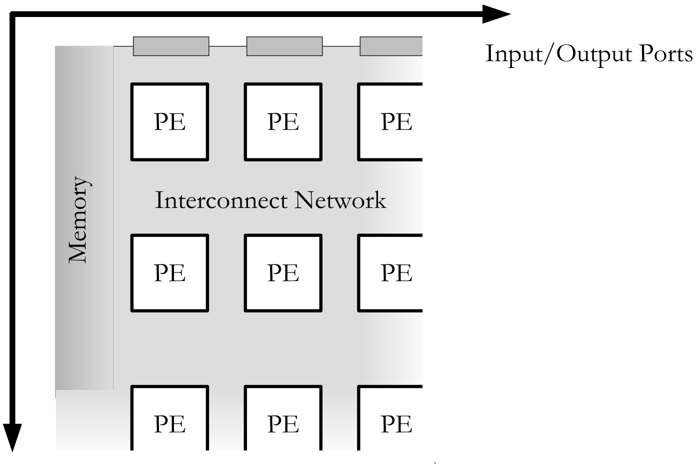


Figure 2.2.: CRC architecture model - basic concept.

The CRC model fixes the array-like organization of the PEs, and the existence of interconnection and I/O resources, but not their implementation. Instead, the designer *configures* the architecture at design time by choosing one realization for each one of these components (PEs, interconnection network, etc.). For example, the designer determines the number of columns and lines in the array, the type of each processing element, and how each PE is connected in the array. An array is said to be *homogeneous*, if all PEs in the are of the same type. And it is said to be *heterogeneous* if different types of PEs are used.

Beside the architecture structure, the CRC model defines one functional aspect for the architecture: the reconfiguration. In the CRC model, *reconfiguration* refers to the ability to change the context of the architecture at execution time. The CRC model requires a context information at any time point during the execution of an application. This *context* information describes which operation is carried out by each PE and how the data flows between PEs. Figure 2.3 helps understanding this principle. Context 1 describes that $PE_{11}$ and $PE_{12}$ will read input ports and perform additions. The data produced in both PEs is transferred to $PE_{22}$, which subtracts the two values and makes the result available at an output port. Context 2 determines that all PEs will perform multiplications with locally stored data and store the result at internal registers. No data is transferred between PEs.

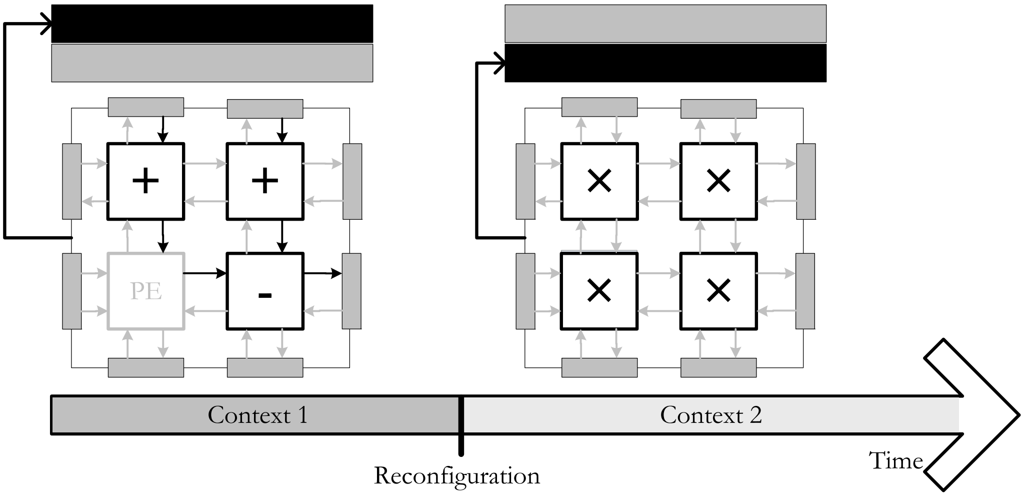


Figure 2.3.: *Reconfiguration* in the CRC model. A *context* defines the functionality of each PE and the flow of data between PEs. Reconfiguration takes place by choosing another context in the context memory.

Nothing is fixed in the CRC model about the time when the reconfiguration takes place. Several options are available to the designer: the reconfiguration can take place at the beginning of the execution and retain the context during the rest of the time (*static reconfiguration*), or it may occur one or more times during the execution (*dynamic reconfiguration*). Reconfiguration can affect all elements in the architecture (*total reconfiguration*) or only some of them (*partial reconfiguration*). At the remainder of this work, a dynamic and total reconfiguration may take place cycle-by-cycle; that is, a new context may be selected at each clock cycle. Many newly developed CGRA devices such as ADRES [77], DRP [84], and DPGA [27] use this extremely fast reconfiguration scheme. Frequent reconfiguration as part of the regular execution constitutes a new principle and is referred to as *processor-like reconfiguration* [93][96]. Processor-like reconfiguration allows it to instantiate and to execute, within one clock cycle, exactly that part of a circuit that is needed in this cycle. Such fast reconfiguration scheme was shown to increase performance when the architecture is targeted to an application domain such as computer vision [92], and to introduce new power optimization possibilities [104].

In the CRC model, contexts are stored in a memory element called *context memory*. Each entry in the context memory contains the information that completely describes one context. When one entry is selected, the context memory outputs a control signal that activates processing elements and the interconnection network. In Figure 2.3, the entry corresponding to context 1 is selected initially. At a given time point, a reconfiguration takes place by selecting the entry in the context memory that corresponds to context 2.

We say the context memory is *centralized* if each entry contains the context information for the whole array. It is also possible to use several context memory units; each unit stores the context information for a group of PEs and part of the interconnection network. In this case, it is possible to reconfigure only part of the architecture. At the remainder of this work, one context memory unit is available in each processing element and the part of the interconnection network related to it. Also, the term *context* is used interchangeably to designate the information stored in one entry of context memory and the context information for all the array. Individual context memories in each PE and the processor-like reconfiguration principle allows to reconfigure each PE individually at each clock cycle.

A Verilog description of the CRC model, called *CRC Template*, captures the array-like structure and the processor-like reconfiguration mechanism discussed previously.

### 2.1.2. The CRC Template

The CRC template is a base description for a family of coarse grained reconfigurable architectures that abides by the CRC model. Its development was based on large number of commercial devices such as NEC's DRP [85], PicoChip's PicoArray [30][99], PACT XPP [98], and Quicksilver's ACM [100], and in academic research work such as DPGA [27], MATRIX [81], and Morphosys [71]. For example, DRP, DPGA and MATRIX strongly influenced how context based reconfiguration is implemented in the CRC template. The CRC template is not the only possible description of the model, but it is representative to the actual state of the art.

The CRC template is a structural description at register transfer level using Verilog. At its higher structural level, the template describes an array of processing elements and a point to point connection between the input and output ports of distinct PEs. It also describes modules to load context information in the context memories; however, these modules are not important for this work and will be ignored here. At a lower structural level, the template details the internal composition of processing elements. Each PE comprises one or more functional units, data register banks (used as local memory), I/O ports, control units, and a context memory. Multiplexers implement flexible interconnection mechanisms between these components. At its lowest structural level, the template describes the internal composition of each PE component. A detailed discussion for the composition of functional units and context memories is presented in sections 2.1.4 and 2.1.5.

The CRC template was made flexible and easy to modify, such that it represents a large number of different coarse grained architectures. Flexibility is achieved by means of parameterization. Parameters are variables in the Verilog description to which the designer assigns a specific value at design time. Changing the value of a parameter modifies one specific structural or functional aspect of the architecture, as depicted in Figure 2.4. Values are assigned using a definition file, and this procedure does not require modifying the Verilog description. The CRC template foresees that the following characteristics may be changed using parameters:

**Geometry of the array ($\mathbf{n}, \mathbf{m}$):** PEs in the CRC template are spatially distributed in an array-like formation; $n$ is the number of lines, and $m$ the number of columns of this array. Both $n$ and $m$ are parameters in the CRC template.

**Datapath width ($\mathbf{d}$):** The datapath width determines the number of signal lines for one data word. It directly affects size of the interconnection network and of the PE internal routing, the size of each register in the register bank, and the size of multiplexers used for routing data. Typical datapaths use 8, 12, 16, 24, 32, 64, or 128 bits.

**Number of PE registers ($\mathbf{r}$):** The parameter $r$ adjusts the number of registers available in the local register bank of each PE.

**Number of entries in the context memory ($\mathbf{c}$):** The parameter $c$ determines how many contexts may be stored locally in the context memory of each PE. The number of contexts for the complete array is $c^{(n \times m)}$, which is obtained combining all possible local contexts.

**Type of the control unit ($\mathbf{t}$):** The control unit within each PE is modeled like a finite state machine (FSM). The parameter $t$ defines whether the FSM is of Mealy [74], Moore [83], or Medvedev type.

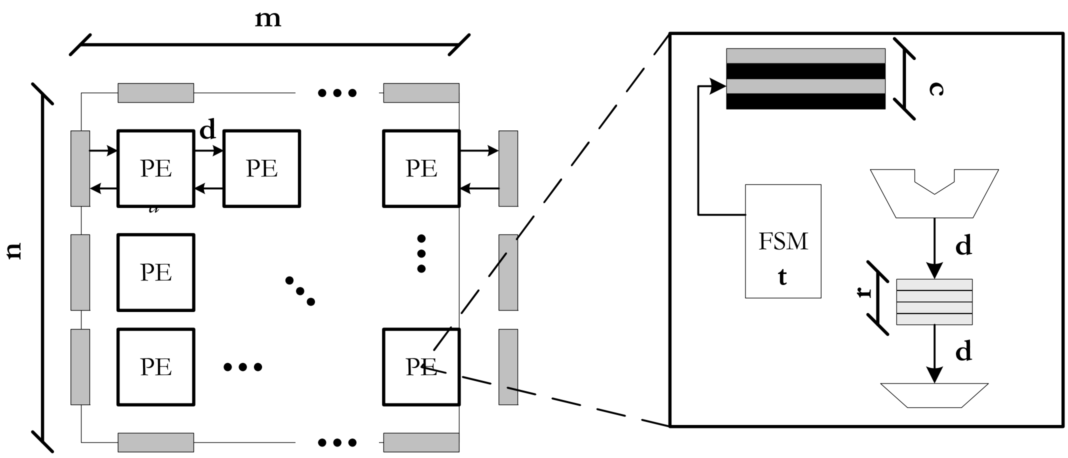


Figure 2.4.: Parameterization in the CRC model. $\mathbf{n}$ and $\mathbf{m}$ determine the number of rows and the number of columns in the array; $\mathbf{d}$ determines the datapath width; $\mathbf{c}$, the number of entries in the context memory; $\mathbf{r}$ corresponds to the number of registers in the register bank; and $\mathbf{t}$ indicates the number of states available in the finite state machine.

Parameters are allowed to vary within predefined values; for instance, the number of columns or lines in an array must be an integer larger than 1. The use of parameters is limited to the characteristics listed previously; for other cases, parameters cannot be used and a direct modification of the Verilog description is necessary. Two of the architecture characteristics that cannot be regulated by using parameters are of special interest in this work: new connection lines between PEs and new functional elements describing custom instructions.

New connection lines are often necessary when modifying the interconnection network topology. It requires the designer to explicitly declare new multiplexers and I/O ports inside the PE description, and to declare explicitly the point to point connections at array level. Both tasks are cumbersome and error prone, and may slow down the design phase. In Chapter 4, we argue that the problem relies on the description language and propose solutions for that.

New custom instructions requires PEs and their functional units to be modified. The designer must insert the datapath that executes the custom instruction in the datapath of one functional unit. Then, the designer must integrate the modified component in the PE description, adjusting the interconnection among elements within the PE. The context memory structure must be modified to allocate the signals that control the newly inserted custom instruction. The procedure to incorporate custom instructions in the PEs of the CRC template is discussed in details in Section 5.2.

### 2.1.3. The CRC Instance

If values are assigned to each template parameter, the resulting description corresponds to a fully defined architecture, and is called an *architecture instance* or *CRC instance*. Figure 2.5 depicts examples of architecture instances that can be configured using the CRC template. The CRC instance is valuable because it allows synthesis and simulation. Synthesis transforms the instance into a circuit netlist at gate level. This netlist allows an accurate estimation of implementation costs, such as the necessary silicon area, static power consumption, and maximal circuit operation frequency. Additionally, the designer can map applications onto the CRC instance by programming their context memory. Simulation of the application-instance pair allows to verify the functionality and to evaluate performance criteria such as throughput, latency, and execution delay.

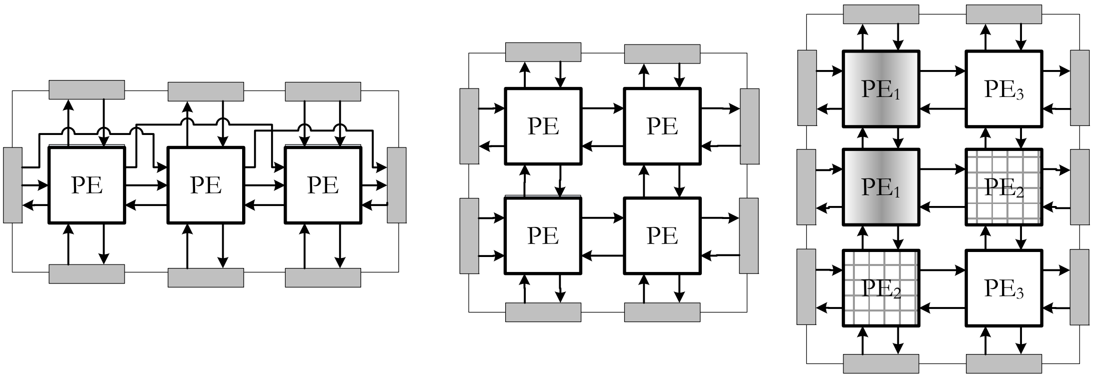


Figure 2.5.: Examples of architecture instances for the CRC model: (a) Line architecture with 3 PEs connected with a 0-1-hop network; (b) 2x2 array of PEs connected with the nearest neighbor; (c) 3x2 array with different PE types.

In the next two sections, the PE and the functional unit internal structure are discussed.

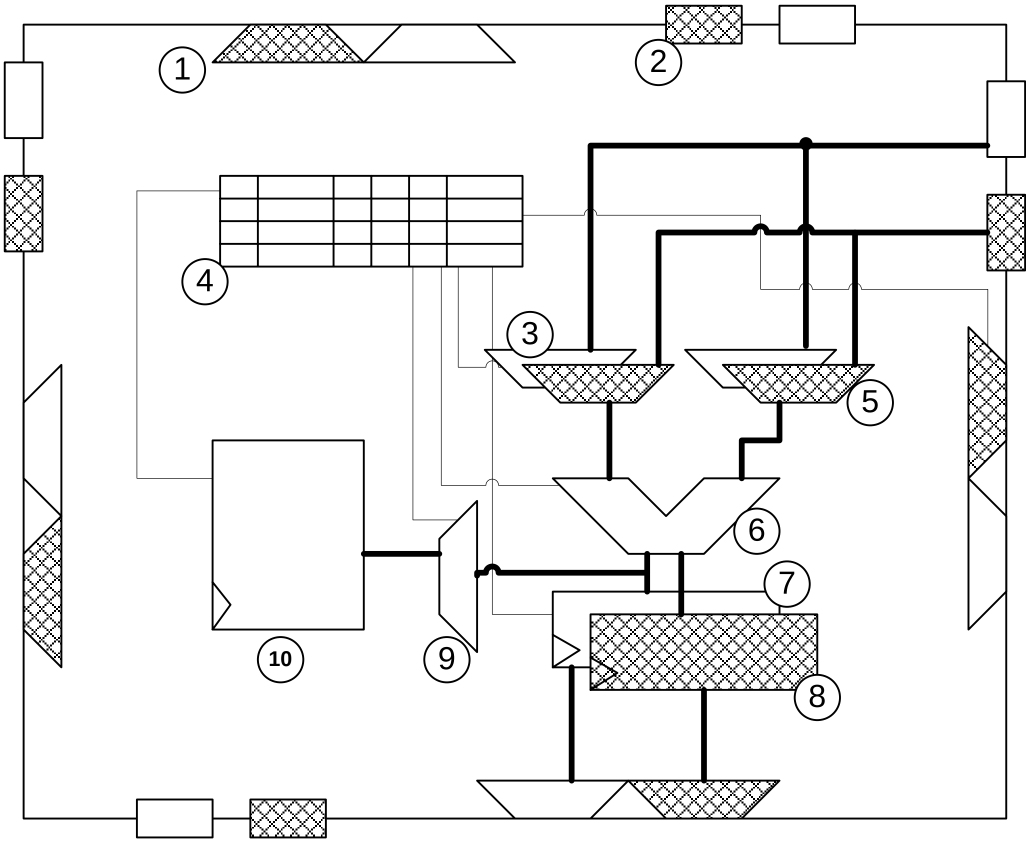


Figure 2.6.: The basic CRC processing element. (1) output ports; (2) input ports; (3,5,9) multiplexers; (4) context memory; (6) functional unit; (7,8) registers; (10) finite state machine;

## 2.1.4. The Processing Element

The base processing element considered in this work is depicted in Figure 2.6. For sake of simplicity, not all connection channels are depicted. Modules that process datawords are depicted by dashed areas, whereas modules that process flag information are depicted by blank areas. The PE consists of two main parts: a control path and a data processing path. The control path part provides hardware circuits that implement the processor-like reconfiguration model. It controls the sequence in which the entries in a context memory are executed, and activates, for each execution cycle, the signals that control modules in the data processing path. According to these control signals, the data processing path retrieves data from input ports and/or local memory, processes it in functional units, and writes back the result into output ports and/or local memory.

The control path comprises a programmable finite state machine ⑩[1] (FSM) and a context memory ④. The finite state machine is a programmable hardware module that stores states and rules for transitions among these states. Such rules are based in input signals that come from the data processing path or from the context memory. For example, one possible rule is: if the FSM's actual state is 1 and a divide-by-zero flag becomes is active, the FSM changes to state 4 — otherwise, it remains in state 1. In the programming phase, each state is associated with one entry in the context

[1] Circled numbers correspond to units depicted in Figure 2.6.

memory. When one state is active, it selects this associated entry. The context memory is also a programmable module. Its program describes, for each entry, the signaling information to control the modules in the data processing path. Additional modules to program the FSM and the context memory are also part of the PE; however, they are not relevant in this work and details about their construction will not be discussed here.

The data processing path comprises input ports ②, multiplexers ③ ⑤, functional units ⑥, internal memory blocks ⑦ ⑧ ⑨, and ouput ports ①. Input ports transfer data from interconnection network to modules inside the PE. Multiplexers implement the internal connection among PE modules. They select where the data to be processed come from and to where the results go. Functional units are compound hardware blocks containing modules to process data. Local memory blocks are register banks that provide fast acess to temporary data and constants. Finally, output ports transfer data stored or produced within one PE to the interconnection network (array). Each output multiplexer has as input the signals coming from the remaining three (input) ports and the signals coming from the register set (one per register in the set). Data from other ports may be selected to route information from one PE to another. Data routed through the PE does not occupy the functional unit, so it is possible to execute an operation in the FU and route data between ports simultaneously

A complete execution cycle for the PE can be described by the following steps:

1. At each positive transition of the clock signal, the finite state machine ⑩ and the register set ⑦ ⑧ are activated. The finite state machine goes to its next state according to flag multiplexer signals and its internal program. This new state selects a new entry in the context memory ④. Simultaneously, the register set stores the result of the last operation.

2. The information for the selected context is passed to the context memory, which outputs accordingly all the control signals to each element in the PE datapath. These control signals reconfigure the datapath.

3. After this control information reaches each PE element, the datapath is executed as follows:
   - operand multiplexers ③ ⑤ select the data/flag lines that will be processed by the FU ⑥. These lines come from the register set (one per register in the set) and from the input ports ② (one per input port);
   - the FU receives an instruction code signal that determines which operation is to be executed;
   - the data/flag register set is addressed to the place where the result should be stored (see step 1);
   - flag lines (from the FU output and input ports) are sent to the flag multiplexer ⑨. One of these signals will be used to select the next state of the finite state machine;
   - and output multiplexers ① select the data/flag lines that go to the array interconnection network.

4. When the computation within one PE is finished, the system waits for the next clock cycle and restarts the process.

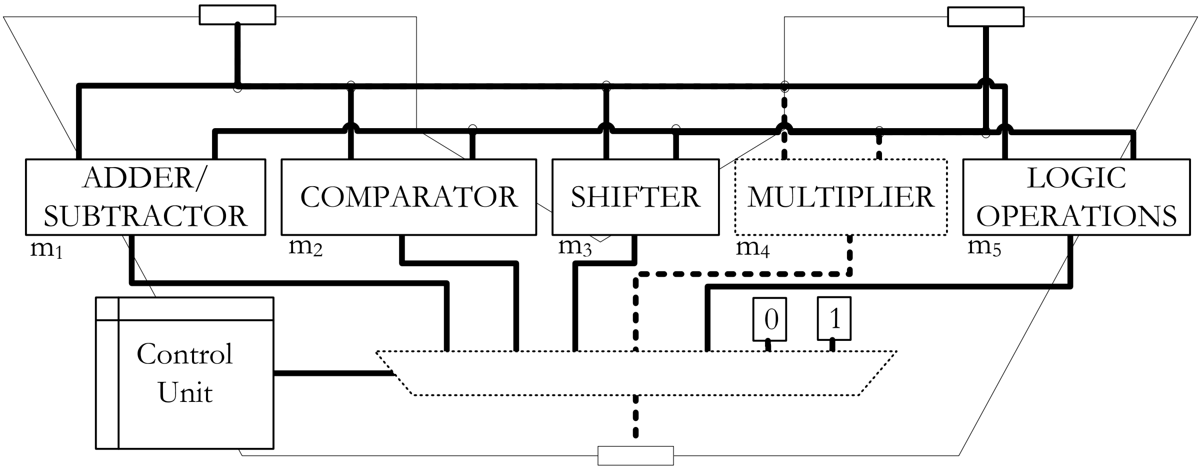


Figure 2.7.: A basic functional unit. During the execution of an *instruction*, the control unit activates exactly the necessary operation module to carry out the desired data transformation. Control units also control the multiplexers to select the result(s) that will be presented in the output port(s).

Each one of these modules — input and output multiplexers, finite state machine, register sets, and context memory — were individually planned and made available as library components in the Verilog description of the CRC template. They are assembled together to compose an structural description of the PE. The only exception is the functional unit. The functional unit is a complex module composed of other several operation modules. In the next subsection, the FU is outlined in detail.

## 2.1.5. The Functional Units

Functional units (FUs) are modules in the CRC template that effectively transform or process data. The structure of a functional unit determines a datapath between their input and output ports passing through *operation modules*. Operation modules are hardware units responsible to carry out a specific transformation of the data; accordingly, we say each operation module carries out one specific *operation*. Examples of operation modules are adders/subtractors, shifters, multipliers, logic arithmetic units, and bit-manipulation units.

In this work, an *instruction* is a chain of operation modules in the datapath of a functional unit that performs a specific data processing task. When the datapath processes the data in a given way, we say the FU executes one instruction. The set of all possible instructions that can be executed by an FU is its *instruction set*.

A base model for the functional unit is depicted in Figure 2.7. The datapath in this FU can execute instructions corresponding to the operators of the programming language `C`, except for the modulo (/) and division (%) operations. This functional unit has two input ports and one output port.

In general, functional units have an internal control unit. The control unit is responsible for the activation and coordination of the operation modules that are necessary for one instruction. Suppose one multiplication is to be executed with the FU in Figure 2.7. The control unit activates the multiplier, deactivates other operation modules, and controls the ouput multiplexer to forward its input port corresponding to the multiplier output data. The control unit receives an instruction code from the context memory and modifies the datapath accordingly.

The datapath of an FU can be modified to include new operation modules or to enable other combinations of the existing operation modules. In this case, new instructions are added to the instruction set of the FU. This set of new instructions is referred to as an *instruction set expansion*. The term *custom instruction* refers to one instruction that was designed to meet specific demands or to provide advantage in the execution of a specific application or application group. A *custom instruction set expansion* is a set of new custom instructions.

## 2.2. Application

According to the WebOpedia: ”A computer program (software) is an organized list of instructions that, when executed, causes the computer to behave in a predetermined manner [*WebOpedia*, 2009, s.v. ’program’, online, [122]]”. Programs can be simple, written on a few instructions, or very complex, composed of thousands of intercommunicating parts. *Applications* are computer programs that execute a specific subtask within a larger system. Applications support their users, or their container system, to get part of the job done. For example, a noise filter is an application in a telecommunication system. A group of different systems that solve similar or related problems often reuse the same applications; they are called an *application domain*. The more an application can be reused inside an application domain, the more representative it is to this domain.

The instructions inside a program must not necessarily be executed sequentially. Some instructions are independent from each other; they can be executed in any order or even in parallel. Modern computer architectures use this parallelism to improve performance. Coarse grained reconfigurable arrays may execute tens to hundreds of instructions at a time; therefore, this work focus on applications with a high instruction parallelism degree. Examples of these applications are vectorizable algorithms, computing intensive internal loops, and programs to process streaming data.

There are several ways to represent an application; for instance, using a programming language, an algorithm description, or a mathematical formulation. In some representations, such as *data flow graphs* (DFGs), instruction parallelism is expressed more clearly than in others, which eases its exploitation. DFGs are discussed in Section 2.2.1.

Whatever the application representation is, it must be transformed to a machine-near format, such that a processor can understand and execute it. This transformation process is called *application mapping* and frequently comprises several complex and architecture dependent tasks. A software that carries out the application mapping process is called *compiler*; in this work, the term *compiler* and *application mapping* are used interchangeably. Section 2.2.2 discusses the tasks and techniques for mapping applications onto coarse-grained reconfigurable architectures.

### 2.2.1. Application representation

Data flow graphs (DFGs) are a formal, graph based representation used to capture programs in an application set. This section introduces the notation and several basic graph theory concepts related to DFGs and used throughout this work. The definitions following are based on Golumbic [43] and Valiente [117].

**Definition 2.1** *A* ***directed graph*** $G(V, E)$ *consists of a finite non-empty set* $V$ *of* ***vertices*** *and an irreflexive binary relation between two elements of* $V$*. The binary relation is represented by a collection of ordered pairs* $E \subseteq V \times V$*. The element* $(v_i, v_j) \in E$ *is called an* ***edge*** *from* $v_i$ *to* $v_j$*. Additionally, if* $V'$ *is a subset of* $V$*,* $G(V')$ *denotes the subgraph* $G(V', E')$ *such that* $E' = \{(v_i, v_j) \in E | v_i, v_j \in V'\}$*.*

In the remainder of this work, the term *graph* is used interchangeably with *directed graph*. A graph $G(V, E)$ has a cycle if there is a sequence of vertices $[v_0, v_1, v_2, \ldots, v_l, v_0]$ such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, l$, and $(v_l, v_0) \in E$. A *directed acyclic graph* is a directed graph without cycles. Directed acyclic graphs are extensively used in computer science, as for example, to represent parse trees or dependencies between instructions of a program.

It is common to assign labels to the vertices of the DFG. These labels indicate attributes or information represented by the vertices, such as their color or weight. A *labeling function* is a function that maps each vertex in graph $G(V, E)$ to an element of a given set $A$. The *label* of $v \in V$ is the outcome of a labeling function applied to the vertex $v$. A graph with one or more associated labeling functions is referred as a *labeled graph*.

In order to represent one application, a common type of directed acyclic graph is used: the data flow graph (DFG) [28][112].

**Definition 2.2** *A data flow graph (DFG) is a labeled directed acyclic graph* $G(V, E)$ *where the set of vertices* $V$ *represents operations and the set of edges* $E$ *denotes the set of data dependencies (precedence constraint) between these operations. A labeling function* $Op$ *maps each vertex to an operation type, which is available in a predefined set;* $Op(v)$ *denotes the operation type represented by* $v$*.*

DFGs define a partial order to the execution of the operations of a program, based on the data dependency. If there are data dependencies between any two operations $v_i, v_j \in V$, these operations must be executed sequentially because $v_j$ needs data that are produced by executing the operation $v_i$. Operations that do not have any data dependency may be executed in any order or in parallel. DFGs are a base for scheduling and parallelism extraction methods. They are widely applied in the area of high level synthesis because they expose the parallelism between operations, which is a potential source of optimizations.

We assume that any operation can be classified into one of the following categories:

**Atomic operation** The term *atomic* refers to the fact that, after an operation starts its execution, it will continue without interference until it produces the output data. Examples of atomic operations are logic or arithmetic operations like addition, subtraction, arithmetic shift, and store/load operations. Atomic operations must have at least one incoming edge and at least one outcoming edge. That means atomic operations always depend on another operation that provides their input data, and they always produce a data that is used later by another operation. The label $Op(v)$ indicates the specific type of atomic operation. Atomic operations can be complex, and may consume and produce several data values each time they are executed.

**Input operation** Corresponds to data inputs ports and constants of the data flow graph. These operations do not have any incoming edge because they do not consume data. If $v \in V$ is an input operation, the label $Op(v)$ can be *Input port*, for data input ports, or *Constant*, for constant values.

**Ouput operation** Corresponds to data outputs ports of the flow. These operations do not have any outcome edge because they only consume data. If $v \in V$ is an output operation, the label $Op(v)$ is always *Output port*.

To illustrate how DFGs represent one application, consider the trilinear interpolation algorithm [29], depicted in Figure 2.8. The operations and their data dependency are obtained from a `C` code and modeled as the data flow graph . Twenty eight atomic operations are used to compute the algorithm, but they can be grouped in 3 types: subtraction, multiplication and arithmetic shift. These operations consume data produced by other atomic operations, input ports or constants. In all these cases, an edge is used to connect the producer and the consumer of the data.

Vertices representing input variables are depicted as triangles with outgoing edges, whereas constants are represented by rectangles. This eases the understanding of the elements in the graph. Both vertex types are input operations. Atomic operations are depicted as circles. The type of a given atomic operation is indicated inside the circle. Output operations are depicted as triangles with incoming edges. When necessary, an index (e.g., $v_i$) is placed beside the circle to indicate that specific operation node.

The definition presented for a DFG does not consider the representation of control flow within the application. It represents the program's behavior only within a basic block. A *basic block* is a sequence of program code with only one entry and only one exit point. If the application contains a control flow, we use techniques to transform the control flow into a pure data flow. The technique to be applied depends on if the control flow is a loop or a conditional branch.

To eliminate `for` and `while` loops that have a small number of operations, or to which the number of iterations is known a priori, *Loop unrolling* [86] [1] is applied. Loop unrolling replaces the body of the loop by several copies of the body and adjusts the loop control code accordingly. Compilers use loop unrolling to reduce the overhead of executing the indexed loop and to improve the effectiveness of other optimizations such as common-subexpression elimination, instruction scheduling, and software pipelining. Loop unrolling exposes parallelism at instruction level, and thus it increases the number of operations that can be combined to form new custom instructions. If loop unrolling cannot be applied, `for` and `while` loops are represented only by their internal basic block; the control information is simply ignored. This information does not affect the extraction of instruction patterns or the composition of custom instruction. The control information, however, is again considered when mapping the application in the architecture.

To eliminate conditional branches, such as `if-else` control structures, a similar approach to *speculative execution* [86] is used. During an speculative execution, all the branches are executed, but the results of only one of them is used later depending on the evaluation of a condition. Figure 2.9 exemplifies how to describe the conditional branch as a pure data flow. All branches are executed, as well as the computation of the conditional value. The data produced in each branch is used to feed a select operation. The select operation uses the conditional value to choose which incoming data is propagated. This procedure transforms the control dependency between the

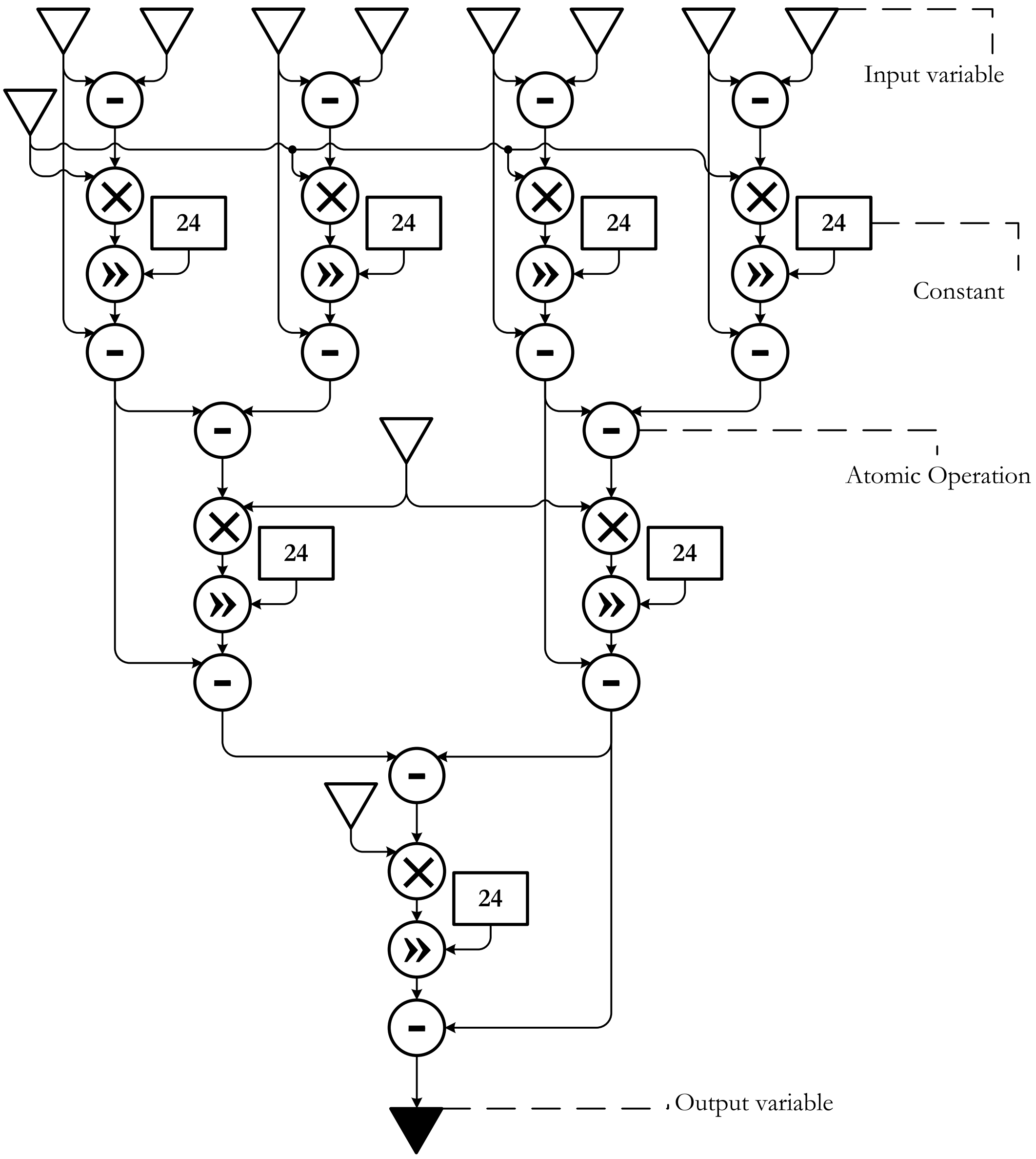


Figure 2.8.: Data flow graph example.

branches into data dependency. This technique increases parallelism at instruction level because both branches may be executed at the same time.

### 2.2.2. Application mapping

*Application mapping* refers to a sequence of tasks that determines how one application runs within a target system. The application mapping process receives the following three inputs:

**The application description** lists the operations to be executed and their dependencies, which defines the operation execution order. This description is a data flow graph, discussed previously in this section.

```
if (c < 10) {
    b = a + 1;
} else {
    b = a - 1;
}
```

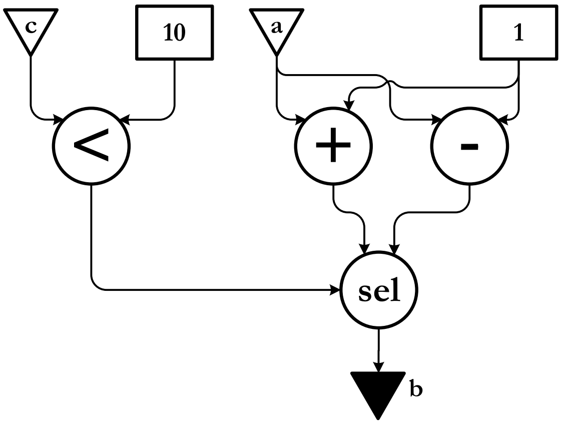


Figure 2.9.: Transformation of `if-else` structures in a pure data flow. Both branches are executed producing data. Then a select operation chooses the data to be used further as `b` considering the computed condition.

**The architecture of the target system** describes which resources (FUs, memory, etc.) are available to accomplish the application task, and how these resources are interconnected. In this work, an architecture is completely defined by a CRC instance, as discussed in Section 2.1.1. Additional information about the architecture includes usage costs for each resource, such as execution delay and power consumption. This information is obtained through synthesis and evaluation of the CRC instance.

**Constraints** establish requirements for the execution of the application, such as minimal throughput, maximal power consumption, and maximal area or resource usage. The type of requirement to be used highly depends on the application.

The application mapping phase is usually an automatic process carried on by a *compiler*. In this work, both compiler and application mapping refer to a process that transforms the application description into an executable format, which can be processed by the target architecture. The sequence of tasks that compose the application mapping phase resembles that of the behavioral or high-level synthesis [35]. It comprises three central tasks: *scheduling*, functional unit *binding*, and *routing* or interconnection binding.

### Scheduling

Scheduling is a central task in behavioral synthesis. The scheduling task partitions the design behavior into time steps (also called control step) such that all operations in a time step execute in the same clock cycle [121]. There are several scheduling strategies, but in the remainder of this work, only the *resource constrained scheduling* variant is used. The goal of resource constrained scheduling is to minimize the number of time steps given the type and the maximal number for each functional unit available.

During the scheduling phase, each operation in the DFG can be executed at a certain time step only if all its predecessors have been executed at previous time steps. Thus, an operation executes only after all its dependencies are resolved. Additionally, operations can be added to a time step as long as there are processing elements available to execute them. The schedule for the DFGs, depicted in Figure 2.10, may clarify this. In the DFG on the left, only two PEs are available in the architecture. If operations $v_1$ and $v_2$ are executed at time step 1, operation $v_3$ can be executed only at time step 2; there are no other PEs available to execute it. Consequently, $v_4$ must be executed at time step 3 because it must come in a later control step as its predecessors ($v_1$, $v_2$ and $v_3$). In the DFG on the right, 4 PEs are available in the architecture. As a result, operations $v_1$, $v_2$ and $v_3$ can be executed simultaneously in time step 1; the execution of $v_4$ follows at time step 2.

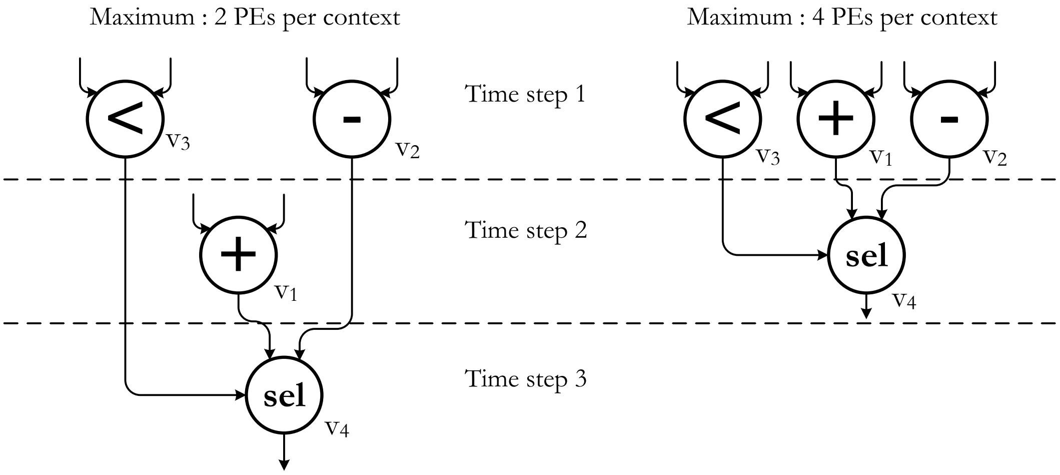


Figure 2.10.: Scheduling examples.

### Binding

The binding task assigns each operation within a given time step to the processing element (functional unit) that will execute the operation. A feasible mapping between operations and PEs is called a *binding state*. Figure 2.11 depicts examples of some possible binding states for the DFG of Figure 2.9. The quality of a binding state can be evaluated according to different criteria; for example, the area usage, the total data transfer delay, or the network distance between communicating operations. The network distance takes into account the data dependency between operations: the shorter the distance between two communicating operations, the better is the binding quality. One way to define *network distance* it to consider the semi-perimeter of the smallest rectangle that inscribe all network nodes. The network distance measured for a binding state is called *binding state energy*.

It is possible to achieve a good quality binding state (low binding state energy) through successive improvements of an initial binding state. For example, moving the node $v_4$ in Figure 2.11a to the central PE leads to the configuration in Figure 2.11b, which has a smaller overall network distance. The binding state depicted in Figure 2.11c has an optimal low binding state energy; it can be reached through successive improvements of the initial configuration. This successive improvement may be realized by using techniques such *simulated annealing* [66][118]: a probabilistic heuristic to solve global optimization problems.

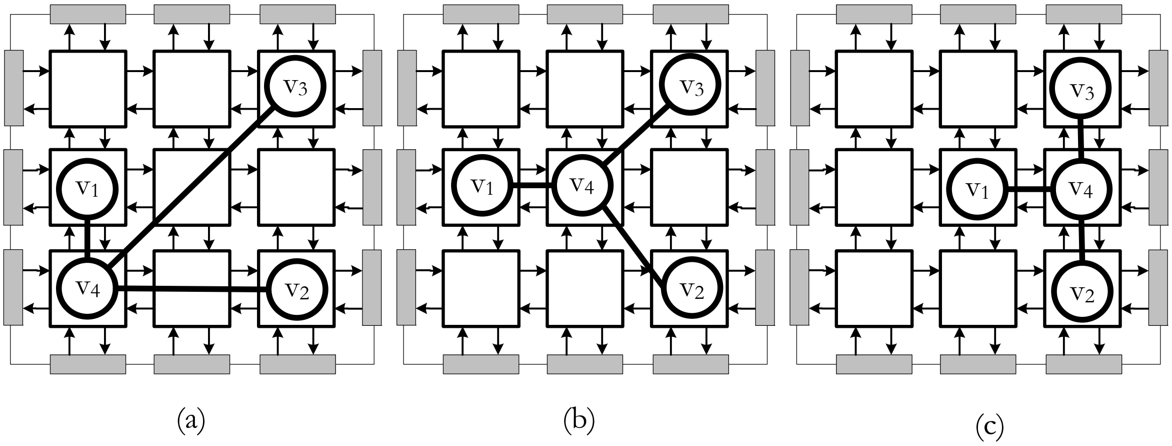


Figure 2.11.: Examples of binding states: (a) long network distance between nodes $v_4$ and $v_3$; (b) shorter network distance after displacement of $v_4$ to the middle PE; (c) binding state with minimal network obtained after successive improvements.

### Routing

The third task of the application mapping phase is to bind data transfers, represented by edges in the DFG, to interconnection resources, such as input/output ports and PE interconnection lines. To distinguish this from the binding of operations to functional units, this task will be called *routing* in the remainder of this work. The routing phase goal is to minimize the overall communication between producer and consumer of the data.

### Reconfiguration

Processor-like reconfiguration is another important issue during the application mapping considered here. The scheduling, binding and routing tasks must consider the partition of the application in contexts. Each context contains exactly that part of the application that will be executed at a given clock cycle. Processor-like reconfiguration enables to map the application using three different strategies: *multi-context*, *pipeline* or *multi-context pipeline*. Examples for these strategies are depicted in Figure 2.12. A pure multi-context mapping assigns each scheduled time step to one different context. During execution, the sequence of context reconfigurations follows exactly the sequence of time steps determined by the scheduler. As depicted in Figure 2.10, operations $v_1$, $v_2$ are executed at the first context (time step 1), operation $v_3$ at the second context (time step 2) and $v_4$ at the third.

In a pure pipelined mapping all time steps are executed at the same context. This strategy does not use reconfiguration or resource sharing. All operations execute at the same time, but each one of them works on data that was produced by its predecessors in the previous clock cycle. At a certain clock cycle $i$, the operation $v_3$ (PE$_3$) in Figure 2.12 processes the data produced by $v_1$ and $v_2$ in clock cycle $i - 1$. $v_1$ and $v_2$ also execute, but they process already new data. Pipeline mapping can only be applied if there are enough PEs available in the architecture to accommodate all operations.

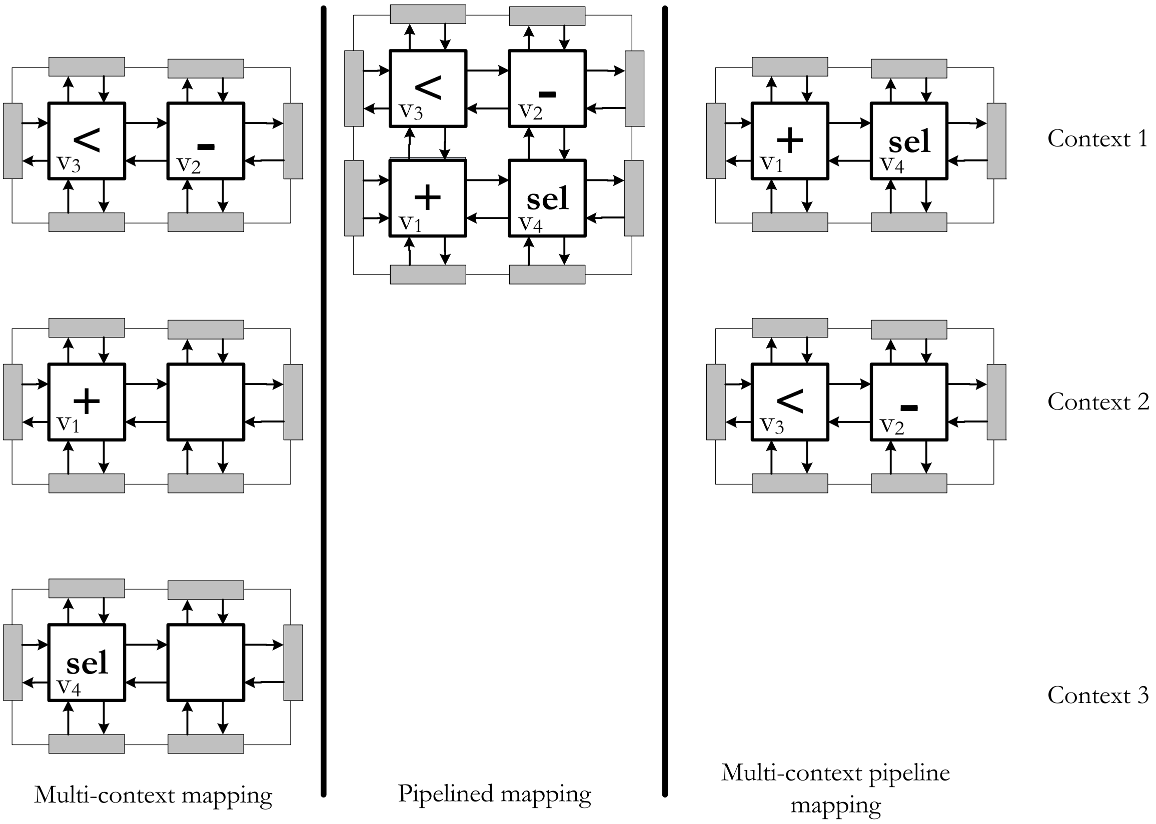


Figure 2.12.: Multi-context, pipeline, and multi-context pipeline

A combination of both strategies, called multi-context pipeline, is possible. In this case, at least two different (but not all) time steps share the same context. The time diagram in Figure 2.13 helps understanding how the DFG in Figure 2.12 is executed. Consider context 1 is executed at clock cycle 1 and 3, and context 2 is executed at clock cycle 2. At clock cycle 1, context 1 executes: operation $v_1$ produces valid data, but operation $v_4$ produces dummy data because there are some non-resolved data dependencies. At clock cycle 2, context 2 executes and operations $v_2$ and $v_3$ produce their results. The system reconfigures back to context 1 at clock cycle 3; now, $v_4$ processes the data produced in clock cycles 1 and 2, while $v_1$ processes data from a new input set.

Table 2.1 summarizes the resource requirements and the performance for the previous example. The pure pipelined mapping requires 4 PEs. The pure pipelined and the multi-context pipelined mappings require two PEs because they use reconfiguration to share resources among different operations. The multi-context strategy needs three clock cycles to execute the application, where the other two strategies require only 2. Finally, pure multi-context solution achieves the smallest throughput (1/3) because it has to execute the application before it can consume the next input data set. The best solutions are achieved by the pipelined (1) and multi-context pipelined mapping (1/2), as they simultaneously process data from different data sets. This example shows a common trend between these strategies: multi-context leads to less resource usage and slower designs; pipeline requires a great amount of processing elements but offer the best performance marks; and multi-context pipeline combines the advantages of both approaches to achieve a middle point trade off.

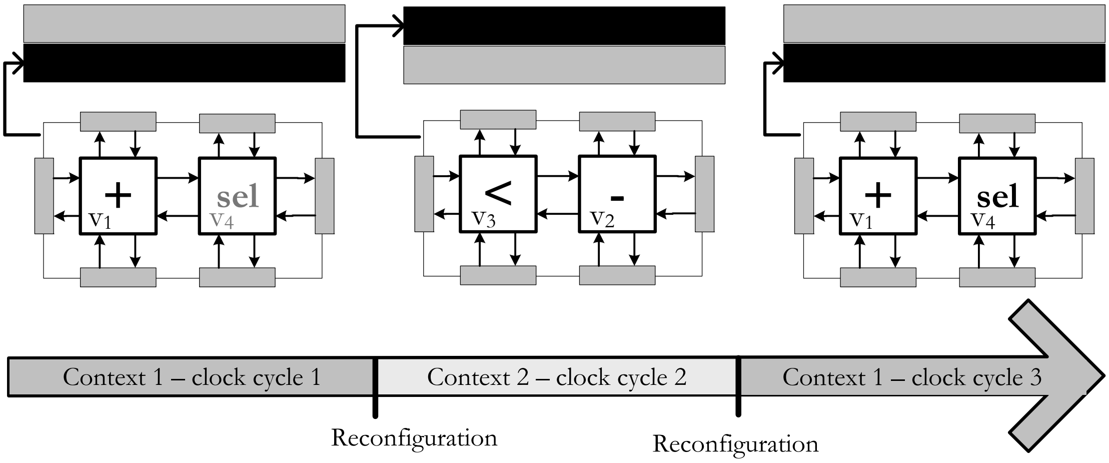

+
v1
sel
v4
<
v3
-
v2
+
v1
sel
v4
Context 1 – clock cycle 1
Context 2 – clock cycle 2
Context 1 – clock cycle 3
Reconfiguration
Reconfiguration

Figure 2.13.: Example for the execution of the multi-context pipeline.

Table 2.1.: Resource usage and performance for the example in Figure 2.12.

| | Multi-context | Pipeline | Multi-context pipeline |
|---|---|---|---|
| PEs | 2 | 4 | 2 |
| Latency (clock cycles) | 3 | 2 | 2 |
| Throughput (input set per cycle) | 1/3 | 1 | 1/2 |

We showed that processor-like reconfiguration capability affects simultaneously application mapping tasks. It is then reasonable to formalize and solve these tasks together. The author presented a method to solve simultaneously the scheduling, binding, and routing problems in [16] using an integer linear programming approach; the details for this method are out of the scope of this thesis, and will not be discussed here.

# 3. State of the Art

During the last two decades, a large number of electronic systems based on coarse grained architectures have been proposed in academic research works, and introduced in the market as commercial devices. They yield a better performance and smaller power consumption than general purpose processors and FPGAs, and more flexibility than ASICs. This chapter presents an overview on actual coarse grained reconfigurable architectures from academy and industry.

Four of the, actually, most representative CGRA instances (NEC's DRP, Silicon Hive's ULIW, IMEC's ADRES, and the Weakly Programmable Processor Array (WPPA)) were seleced to be discussed here (Section 3.1). NEC's DRP and the Silicon Hive's ULIW architectures are commercial instances that demonstrate the applicability and feasibility of these devices. IMEC's ADRES, and the Weakly Programmable Processor Array (WPPA) are academic proposals that allow a deeper insight on the development phase. Historically important proposals are also briefly discussed in Section 3.2. Each section discusses the array structure and functionality of the instance, lists some of its use cases, and compares it with other approaches. However, the focus of the discussion will rely on the description and specialization of CGRAs during the design phase (specially on the academic instances).

Section 3.3 evaluates the design of CGRAs and points out two productivity bottlenecks: the innadequacy of description languages and a complex, trial-based exploration of the design possibilities. This section also clarifies the points in which this thesis improves the state of the art.

## 3.1. Coarse grained reconfigurable architectures

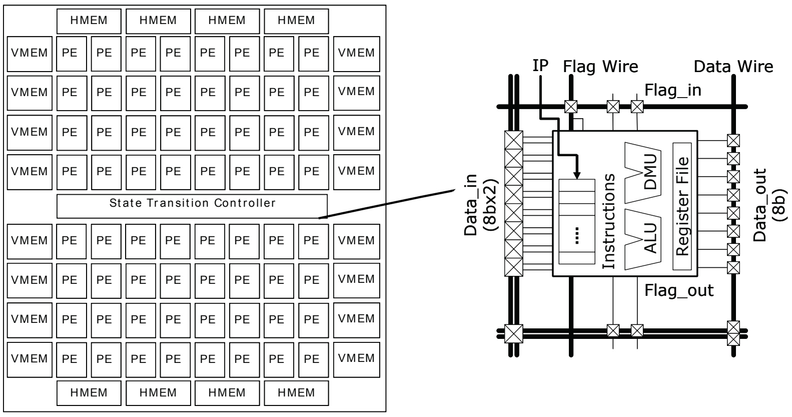
### 3.1.1. NEC — DRP

The *Dynamically Reconfigurable Processor* (DRP) was introduced by the japanese company NEC in the year 2002 [84]. According to their designers, the DRP offers an ASIC-like performance and a software-like scalability (flexibility). ASIC-like performance is obtained by configuring custom datapaths in an array of PEs, whereas a very fast dynamic reconfiguration of these datapath configurations build up to the software-like flexibility.

#### Architecture

NEC's DRP uses an homogene array with 512 processing elements divided in 8 tiles (see Figure 3.1a). PEs have a set of internal registers, but memory blocks are also spread out along the array to store larger amount of data, if necessary. The context memory, denoted STC in the DRP, is centralized in each tile: that means, each STC contains the configuration data for a complete tile. Sixteen contexts are available in each STC. Also inside each tile, the interconnection network between PEs is claimed to be fully programmable, and can deal with up to 2 input 8-bit words.

The DRP device has yet 1 SDRAM/CAM controller, 1 PCI controller, and 8 32-bits multipliers, which are tightly coupled to the array through buses. These components, however, are not part of the array; their usage incurs on extra delay for the execution because data have to be routed outside the array.



(a) (b)

Figure 3.1.: Array organization and PE circuit in NEC's DRP. Source [84].

The DRP's processing element, depicted in Figure 3.1b, contains an arithmetic-logic unit (ALU), which performs 8 bits operations, and a data management unit (DMU), that handles data load and storage, constant generation, and bit manipulations. An internal instruction memory determines (per instruction) ALU and DMU operations as well as the source and destiny of data for each cycle. Source and destination operands can come either from the internal register file or other PEs (using operation chaining).

The functionality of the DRP resembles that of the CRC model, explained in Section 2.1.1: at each clock cycle, the STC excites the instruction memory of PEs within the tile and configurates a custom datapath. The difference to the CRC model is on the fact that each context in the STC affects all the tile, whereas in the CRC each PE is controlled independently. According to NEC, the DRP architecture was designed for stram data processing, such as network packet routing, motion or still picture processing, and wireless data streams.

The first DRP version (DRP-1) was implemented in a $0.15\mu m$ CMOS technology, and could run at a maximal speed of 133Mhz. The most recent version, the DRP-2, uses a $90\eta m$ process technology and runs up to 250Mhz.

### Description and design

Not much can be said about the description and design phase of the DRP. Motomura, one of the main designers of the DRP, explains the basic idea behind the development as:

> DRP is architected based on a clear picture of how C code is compiled into hardware. [*Microprocessor Forum – A Dynamically Reconfigurable Processor Architecture*, 2002, [84]]

However, there are no further publications from the company NEC or their academic partners that enlight how the design decisions were taken during the development of this device. For example, it is not clear which language was used to describe the processing elements and network interconnection, or how designers decided for the division of the array in tiles.

### 3.1.2. Silicon Hive — ULIW

Silicon Hive is a Netherlands-based startup funded by the Philips Electronics. Silicon Hive presented in 2003 the *Ultra Long Instruction-Word* architecture (ULIW) [105]: a mixture of coarse grained reconfigurable array and very long instruction-word (VLIW) processor. Since then, several variants of this device appeared, such as the Avispa and Avispa+, designed for signal processing in OFDM radio systems; the HiveGo-CSS series, devoted to CCD cameras; and the HiveGo VSS, focused on video streaming applications.

The most remarkable feature of the Silicon Hive's platform is the configurability during the design phase. The ULIW architecture is commercialized as synthesizable intellectual properties (IP Cores) that can be tailored to achieve the costumer's requirements. Despite its commercial character, this platform allow a deep insight in the description and specialization of the architecture.

#### Architecture

The foundation of the Silicon Hive's ULIW architecture is the component called *processing and storage element* (PSE). The basic structure of the ULIW architecture can be seen in Figure 3.2. The PSE is a processing element with own functional units, register set, and local memory I/O; similar (but more complex) to the PEs of the CRC model described in Section 2.1.4. The 'network' circles in Figure 3.2 represent the local interconnects between a PSE's register sets and functional units, or between PSEs. This network is also configurable during design time. Like the DRP and the CRC model, PSEs are designed to work in parallel, but as a single datapath.

The characteristics of PSEs change depending on the device instance. For example, the Avispa+ has four identical PSEs, each with up to 12 functional units, including 16-bits ALUs, 16-bits multipliers, a 40-bit adder/accumulator, a 16-bit barrel shifter, two 16-bits load/store units, and a four-way SIMD add-compare-select unit for acceleration of Viterbi algorithms. In resume, PSEs can be a mix of general purpose computing elements with dedicated ones. The performance reported by Silicon Hive is impressive (see Table 3.1 for a comparison between the Avispa and Avispa+): in its peak operation point, the architecture can execute up to 60 operations per clock cycle at a 150Mhz frequency. That corresponds to 9 billion operations per second.

#### Description and design

To support the development of ULIW-based architectures, Silicon Hive has created a whole software tool chain composed of specialized ULIW cores, a library of function units for designers to choose from, and adaptive software-development tools. In terms of the terminology introduced in Chapter 2, Silicon Hive provides the designer with basic components of an architecture template, that can be assembled into concrete architecture instances [18]. The design methodology, depicted in Figure 3.3, follows the same flow as proposed by the designers of the CRC model.

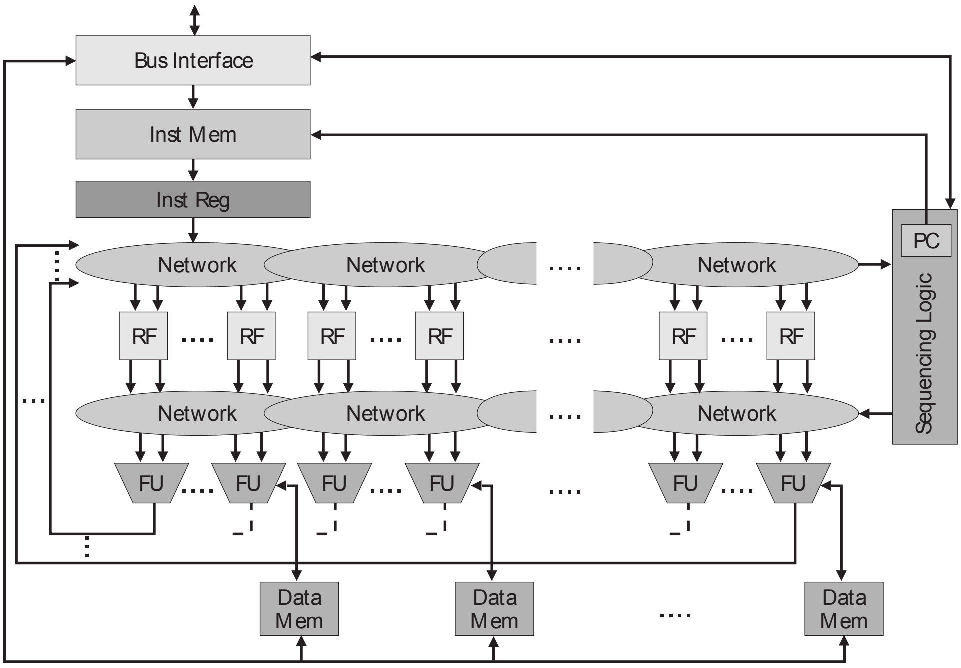


Figure 3.2.: Block diagram of the ULIW architecture from Silicon Hive. The number of processing and storage elements(PSEs), registers, FUs, and data memory is configurable during the design phase. Source [51].

Table 3.1.: Configuration of two commercial versions of the ULIW architecture: the Avispa and the Avispa+ devices. Source [51].

| | Silicon Hive's Avispa | Silicon Hive's Avispa+ |
|---|---|---|
| Appication Domain | OFDM Radio | OFDM Radio |
| Instruction-Word Width | 486 bits | 768 bits |
| Issue Slots Per Word | 41 operations | 60 operations |
| Instruction Memory | 32K | 48K |
| Arithmetic PSEs | 4 | 4 |
| Functional Units | 75 | 103 |
| Register sets | 95 | 130 |
| Clock Frequency (0.13$\mu m$) | 150Mhz | 150Mhz |
| Core Area (0.13$\mu m$) | 6.5$mm^2$ | 4$mm^2$ |
| Power | 127.5mW | 150mW |
| Peak performance | 6.15 GOPS | 9 GOPS |

The starting point for the design of an instance is the proprietary hardware design language *The Incredible Machine* (TIM). Silicon Hive claims TIM is a higher-level language than VHDL, Verilog, or Tensilica Instruction Extension (TIE) languages because it allows designers to configure the architecture template by specifying parameters, such as the number of function units, register

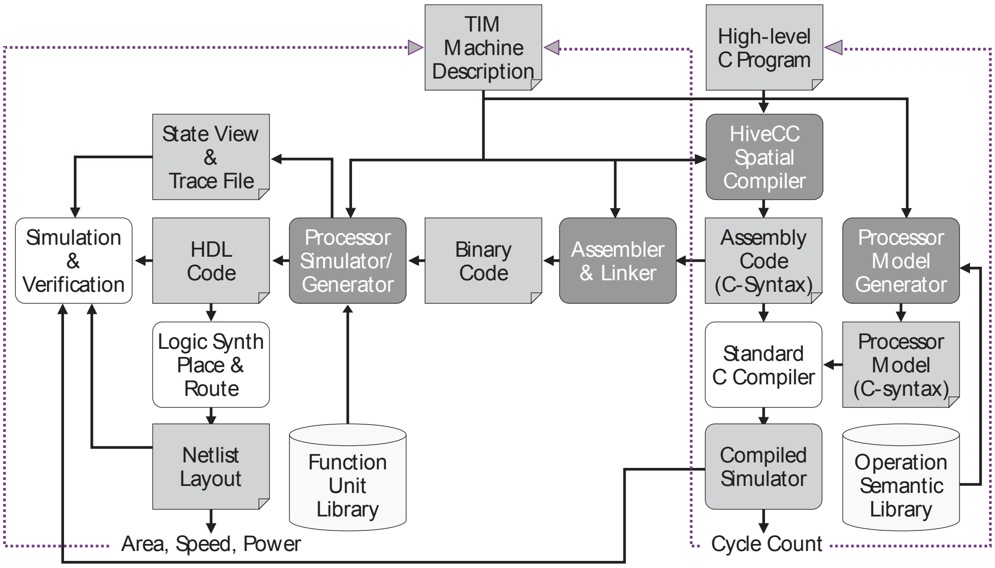


Figure 3.3.: Design flow used by costumers of Silicon Hive. The starting point of the design flow is the proprietary language TIM. From this description a Processor simulator, VHDL (or Verilog) code, and a compiler are automatically generated. Source [51].

files, interconnects, and the list of instructions each function unit can execute [51]. Based on the TIM description of the architecture, prewritten blocks of VHDL (or Verilog) code are instantiated together forming a synthesizable description. TIM also drives the development-tool generator that creates a matching assembler-linker, C compiler, instruction-set simulator, and cycle-accurate simulator.

Specialization of the architecture consists of adjusting the number of registers, number of functional units, memory size, and interconnects to the requirements of a set of applications. This is a refinement process based on an exploration of the design space; that is, designers generate and evaluate several instances with different configurations, and choose the one that demonstrated the best performance-power-area trade off for the target applications. It is also possible to include instructions that are tailored to specific tasks, such as multiply-and-accumulate units or the SIMD add-compare-select units. These special instructions, however, must be part of a Silicon Hive's set of predefined cores.

### 3.1.3. IMEC — ADRES

The *Architecture for Dynamically Reconfigurable Embedded Systems*, in short ADRES, was developed by Mei et al. [76] at the Inter-University Microelectronics Center (IMEC), Belgium. According to their designers, the ADRES is " a flexible architecture template that includes a tightly coupled very long instruction word (VLIW) processor and a CGRA[*Architecture Exploration for a Reconfigurable Architecture Template*, 2005, page 90, [75]]". During the last seven years, ADRES became one of the most representative CGRA proposals in the academic community. The large number of publications involving the ADRES architecture provides a thorough view for the organization and connection of CGRAs [76][75], their design phase [77] [75] [15], and their specialization to application domains [119] [14] [37] [4].

### Architecture

The ADRES architecture is divided in two parts (see Figure 3.4) a VLIW processor and a reconfigurable array. The VLIW processor executes parts of the code that cannot be mapped to the array. The way it is connected to the array, allows the compiler to see the array as an extension of the instruction set in the VLIW processor. The reconfigurable array executes only computationally intensive kernels of applications.

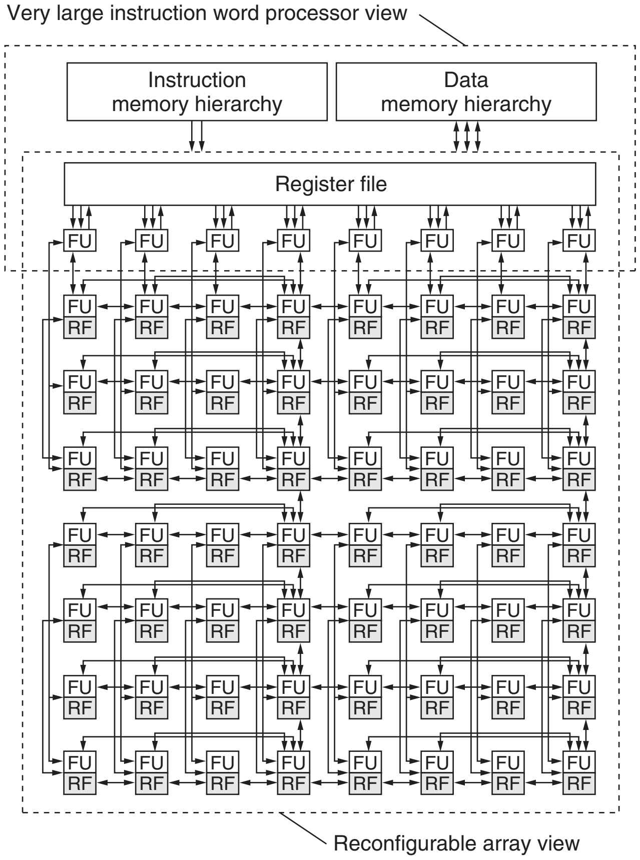

Very large instruction word processor view
Instruction memory hierarchy
Data memory hierarchy
Register file
FU
RF
Reconfigurable array view

Figure 3.4.: Instance of the ADRES architecture template. The ADRES couples a VLIW processor with a reconfigurable array. Source [75].

Like the CRC model, discussed in Chapter 2, the ADRES is described as a flexible template consisting of functional units, storage blocks and routing resources. Basically, the routing resources connect the computational resources to form a certain array topology. Communication between the VLIW and the reconfigurable array takes place over a register file. The functional units of ADRES implement a *predicated execution* that helps to transform the program control flow into a pure data flow [86]. Predicated execution refers to the conditional execution of an instruction based on the value of a boolean source operand: the predicate. Local registers coupled to the FUs avoid long communication delays by buffering intermediary data.

### Description and design

The development team of the ADRES architecture uses the same base methodology as the CRC, DRP, and Silicon Hive groups[1]. However, ADRES became more representative than other architectures and made important contributions for the understanding of CGRA's design phase. The reason is that ADRES's team has at hand a very flexible template and an extensive tool flow that comprises the retargetable compiler DRESC [78], instruction set simulator, RTL simulator, and synthesis tools. Like the CRC template, the ADRES template accepts parameters to adjust the number of registers, number of functional units, and so on. However, this template goes beyond and makes available pre-configured network topologies, functional units, and memory modules, that can be combined with each other arbitrarily. This flexibility combined with the retargetable software tool flow allow designers of the ADRES architecture to generate and evaluate a great number of distinct instances.

Rather than a gradual refinement of the architecture, the designers of ADRES generate a large number of instances and evaluate them: a try-and-error design space exploration (see Figure 3.5). During the evaluation of each instance, the complete set of applications is mapped and simulated to obtain cycle-accurate performance estimates. RTL simulation provides dynamic power consumption and precise propagation delay results. Synthesis provides area and implementation cost estimates.

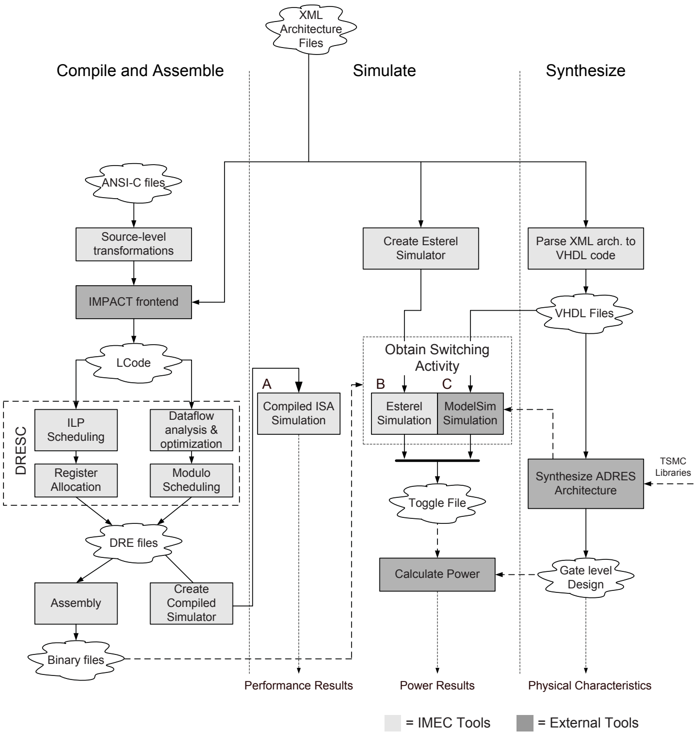


Figure 3.5.: ADRES software-tool flow: an extensive CAD support for the design phase, which includes compiler, simulator, and synthesis. Source [15].

The development of an ADRES instance starts with a description of the template in an eXtended

[1] Other examples will follow in Section 3.2.

Markup Language (XML) format [70]. This XML notation is mentioned in several publications as a ”high-level parameterized description that lets the designer quickly specify different architecture variations [*Architecture Exploration for a Reconfigurable Architecture Template*, 2005, page 93, [75]]” [15] [77][119]. However, there are no explanations on how this notation looks like, or what it can really describe. This XML-base description has to be considered as a proprietary language.

For the ADRES architecture, specialization is a product obtained from the design space exploration. The set of target applications is mapped and evaluated in several instances. Each instance combines a type of interconnect, has a different distribution of functional units and register files, such as the ones depicted in Figure 3.6. The instance that provides the best cost/benefit rate is considered to be specialized to the application domain [119] [37] [4] [75]. More recently, some specific modifications, such the use of clock-gated functional units and register files, were included to improve the architecture power consumption [14]. This is a different approach to specialization, which is independent of the exploration in the design phase.

### 3.1.4. Weakly Programmable Processor Arrays

In 2005, Hannig et. al. proposed the *weakly programmable processor arrays* (WPPA) [52] [67]. In many aspects, WPPAs are other than CGRAs. First, WPPAs have resembles more a network of instruction-set processors than typical reconfigurable arrays: their processing elements are complex processors, with full control structures, fetch and decode stages, instruction pipelining, and instruction memory. Second, their interconnection network resemble more communication buses, where strings of data (messages) are changed between PEs. Third, for the application mapping, the compiler does not divide the application in contexts neither combine the processors to form a datapath to be executed at each clock cycle. Instead, programs are divided in independent threads (with several instructions), which are mapped to individual processors. Other similar architectures are the MIT RAW [111], picoArray [30], and Ambric [62].

Because of all this, WPPAs do not exert the same influence over the research community as the ADRES, nor is this architecture a typical representative for CGRAs. However, the design of WPPAs is important for this work because of its description language: the *MAchine Markup Language* (MAML) [34]. The designers of the WPPA architecture developed an extension to MAML, that deals with description of coarse grained elements, scalability, and the spatial organization of the architecture. These are also common challenges during the description of CGRAs. Since their approach is in many aspects comparable to the CGADL language proposed here, a more detailed revision of their design phase will follow, and in particular, its description with MAML.

#### Description of WPPAs

Beside all the differences pointed before, WPPAs and CGRAs have also several similarities: they use PEs organized in an (scalable) array, their interconnection is a type of mesh-like network, and computing elements are coarse-grained. These characteristics make the description and development of WPPAs similar to that of CGRAs. The development of WPPAs starts with a description of a parameterizable template [67], whose parameters are configured along the design phase and according to a set of target applications.

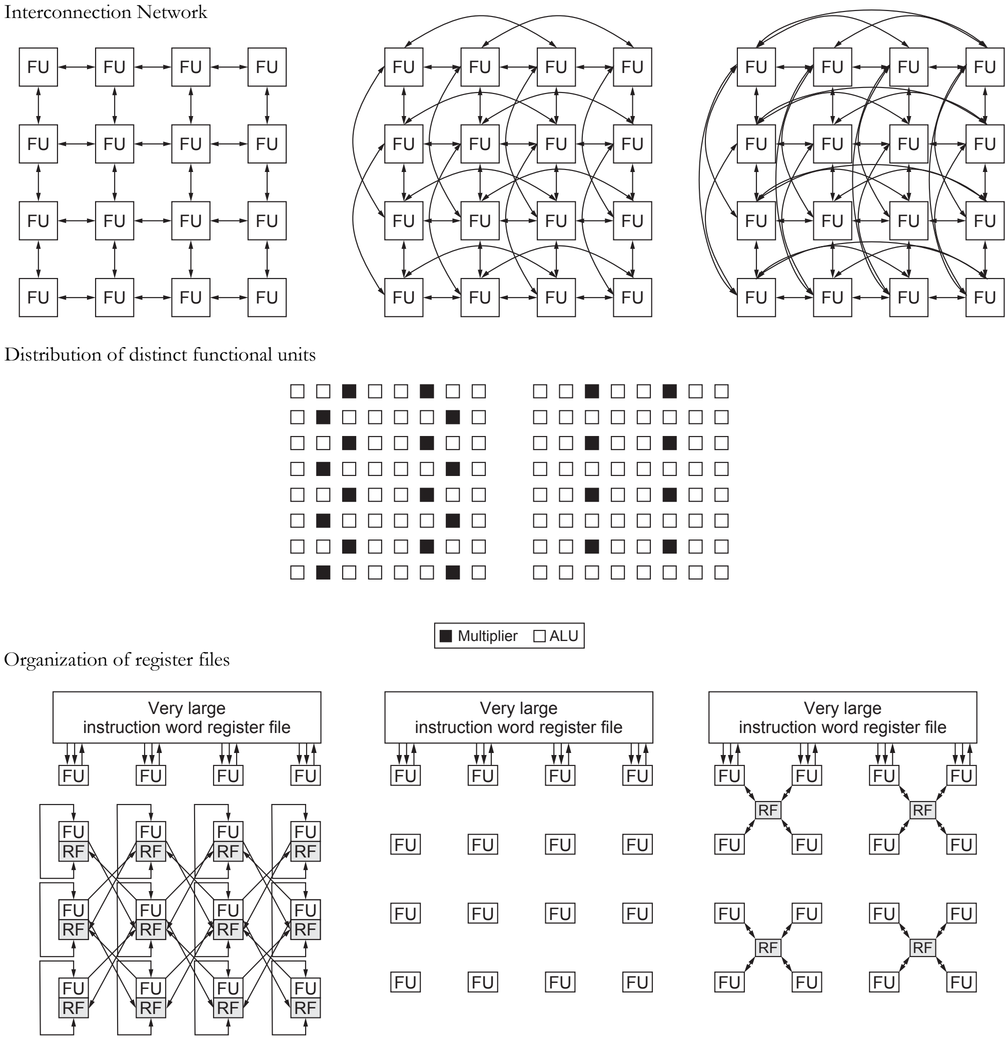

Interconnection Network
FU
Distribution of distinct functional units
■ Multiplier □ ALU
Organization of register files
Very large instruction word register file
FU
RF

Figure 3.6.: Possible configurations for the ADRES interconnection network, disposition of different functional units, and organization of register files. Many more are available in the ADRES template library. Source [75].

Designers of WPPAs use the *MAchine Markup Language* (MAML) to describe their templates. MAML is an XML-based notation and was used initially for describing architecture parameters, such as number of FUs and register allocation, required for the application mapping [34]. In 2006, Kupriyanov et. al. proposed an extension to MAML that was aimed at the design of massively parallel processor architectures [68]. This extension divides the architecture description in two views: the processor view and the array view. The processor view has XML constructs (tags) to describe two aspects: coarse grained components, such as multiplexers and register files; and the instruction-set of functional units (and even their binary code representation). The array view has XML constructs (tags) to describe the array geometry and interconnection network. This view uses a coordinate system to identify and interconnect PEs.

### The processor view

MAML describes a PE type as a structural datapath by listing its storage components (register sets, instruction memory, etc.), functional units, and the interconnection between them. This is a difference, for example, to hardware description languages, where these components have to be first assembled using elementary logic blocks. For example, the tag

```
<RegisterBank name = 'myBank' num = 16>
```

specifies a register file with 16 registers.

In the processor view, the designer can also describe the instruction-set of functional units in a PE. For that, the designer lists the operation name, its opcode (binary code used by the compiler), and characteristics of the execution, such as number of cycles and resources usage. There is however, no behavioral description. The description of instruction sets is another feature that eases the description of CGRAs. This feature is normally present in architecture description languages, such as LISA and ArchC, but not in hardware description languages, such as VHDL and Verilog.

### The array view

MAML allows the designer to assign positions of a 2D plane to the processing elements in an array. Describing the spatial distribution of PEs within an array is of great help for the designers of CGRAs for many reasons: it eases the description of the network interconnection because spatial concepts such as 'above' and 'neighbors' can be used to connect the PEs; it brings valuable locality information to the compiler, which may try to map dependent operations near to each other; and it allows to describe the geometry of the array and how it is scaled.

Based on that positioning system, MAML designers can efficiently describe the array geometry and the interconnection network. The array geometry is described by listing the PE type used in the array and its size (number of rows and columns). So for example, the tag:

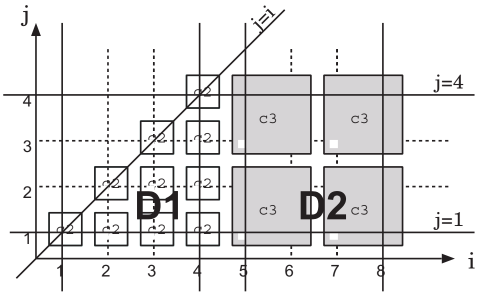
```
<PELements name='pe' rows='4' columns='12'>
```

describes an array `pe` with possibly up to 48 PEs (4 × 12 array). Each element can be referred by the name of the set and teh 2D indexes. In the example, each PE in the array can be designated as follows: `pe[1,1]....pe[4,12].`

The interconnection network description uses a mathematical notation, called *polytopes*, to designate sets of positions (domains) in the space, as the example depicted in Figure 3.7. Every PE inside one domain is connected with the same connection pattern. With this technique, the description of the interconnection network can be done based in rules: first, the designer describe sets of PEs that are connected in a similar way; then, one rule is written that describes how the connection is carried out [69]. This principle is also used in this work for the proposed description language CGADL, and will be discussed in detail in Chapter 4.

## 3.2. Other Work

This section lists several other academic and commercial platforms that fairly represent coarse grained reconfigurable architectures. Details about their architecture will not be discussed, since all of them use some instance of a spatially distributted array of processing elements.



**Domain D1:**

$$\begin{pmatrix} i \\ j \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} 1 & 0 \\ 0 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leqslant \begin{pmatrix} 4 \\ -1 \\ 0 \end{pmatrix} \right\}$$

**Domain D2:**

$$\begin{pmatrix} i \\ j \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 5 \\ 1 \end{pmatrix} \wedge \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leqslant \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \right\}$$

Figure 3.7.: MAML's polytope domains representation. In MAML, the designer can assign positions in a 2D space to the PEs. Later, PEs can be referred to by using this positioning system. Source [68].

### 3.2.1. KressArray

One of the pioneer work in the area of coarse grained reconfigurable arrays was the KressArray, proposed by Hartenstein and Kress in 1994 [53]. The KressArray focused on executing the application as a stream of operations formed by a chain of PEs in the array. This chain of operations would not change during run-time, and thus, this architecture did not support dynamic reconfiguration. The commercial architecture *Extreme Processing Platform* (XPP) from the company PACT was inspired on the KressArray architecture [11]. An important contribution of the KressArray team was the *KressArray Explorer* [55]: a tool-supported design methodology that helped to evaluate and choose between possible instances of the architecture. The KressArray Explorer can be considered a prototype of the actual template-based development methodology. It was less flexible though.

### 3.2.2. Morphosys

Morphosys was another CGRA proposal, which had large acceptance in the academy[71]. It was an CGRA tightly coupled to a RISC processor. The idea behind Morphosys was to use the reconfigurable array to implement the datapath of custom instructions for the processor. Each column of the array receives the same configuration data, but each PE processes different data: this makes the execution model similar to that of a *single instruction multiple data* (SIMD) machine. This idea inspired the design of several other CGRAs, such as the Garp [19], the Chameleon's Montium [108], and the Silicon Hive.

The Morphosys architecture was described in VHDL and implemented in a 0.35 micron technology in 2000. Morphosys's architecture components, such as the context memory and functional units, and its array were custom designed to optimize delay and area [106] . There was no possibility to configure the architecture to obtain different instances. The authors claim that the interconnection network, the coupling to a RISC processor, and the memory interface were targeted to multimedia applications. However, it is not clear how design decisions were taken.

### 3.2.3. Rapport's Kilocore (Piperench)

The *Piperench* architecture was developed in the Carnegie Mellon University [42], and brought out to the market by the company Rapport under the name *Kilocore*. Rows of PEs in the Piperench are called *stripes*, and were designed to work as stages of a pipeline. Piperench allows different architecture instances with different numbers of PEs per row, but has limited flexibility in its interconnection network, which primarily aims the transfer of data between rows in a pipelined way. Verilog was used to design the Piperench architecture [42].

### 3.2.4. Summary

Many other examples for coarse grained architectures could be discussed here. However, an exhaustive discussion of all the members in the family of coarse grained arrays is a hard task, which is out of the scope of this thesis: in the last 10 years, more than 30 different instances were introduced [54], from which at least 11 became commercial devices [3][51]. The presented architectures are fair representatives of the advances achieved in academy and industry.

A close observation in the design phase of coarse grained architectures discloses two facts: first, most of CGRAs are designed with basis at a configurable template, and specialization consists of finding the best configuration for this template (given a set of target applications); second, designers use hardware description languages, such as Verilog and VHDL, to describe CGRAs. The use of higher abstraction levels is restricted to proprietary languages, whose details are not available to the community.

## 3.3. Evaluation of the state of the art

This section provides a more detailed view about the actual design of CGRAs. Emphasis lays on the challenges for the description of flexible architecture templates and the possible approaches to specialize the architecture towards an application domain.

### 3.3.1. Description of CGRAs

The design of coarse grained reconfigurable architectures must deal with several aspects at the same time: the distribution and type allocation of PEs, the granularity of the datapath, structure of the network interconnection, and memory architecture. The result is a very large and complex design. As CGRAs grow in complexity, designers need new ways to describe their design: the description of individual circuits composed of thousands of logic circuits makes the design impossible to be carried out.

Nowadays, the most common approach to design CGRAs is to describe an architecture template. The template fixes the background structure of the array but some aspects, such as the number of PEs, can still be configured. Designers of CGRAs have to face the following specific challenges when describing architecture templates:

**Multi-context Reconfigurability** Hardware reconfiguration provides after-production flexibility to CGRAs, and constitutes one important aspect for their design. Every modern coarse grained architecture provides some kind of partial or total reconfigurability carried out in some static or dynamic way (see Section 2.1.1). Moreover, there is a growing interest on CGRAs as reconfigurable platforms because, due to the granularity of their components, they allow extremely fast context-based reconfiguration mechanisms, such processor-like reconfiguration.

**Spatial distribution of PEs** Part of the high performance presented by CGRAs is due to their ability on distributing the computation spatially. The PEs of coarse grained architectures, commercial and academic ones, are spatially distributed in some kind of array or grid. This organization of elements is intentional and partially determines how applications are mapped and executed in the architecture. Therefore, the design of CGRAs must consider, from the beginning, some positioning system for the processing elements.

**Homogeneous and heterogeneous arrays** The simplest array configuration uses only one PE type for the whole array. It is called an *homogeneous* array. During a long time, the design of CGRAs used exclusively homogeneous arrays, due to their simplicity and predictability. Nowadays, however, the design of arrays with different PE types organized in a somehow irregular pattern is a growing practice.

**Scalability of the array** The number of necessary PEs, their type, and distribution are mostly undefined at the initial description of an architecture. Often, after refinement or design space exploration, the array must be scaled by inserting new PEs, or by redistributing and removing old ones. When using actual HDLs, the designer has to reorganize the array manually and reconnect communication lines in a point-to-point basis. That makes scaling the array a complex and error-prone task.

**Interconnection Network** The interconnection network is one of the most variable and complex aspect of CGRAs. Nevertheless, they are critical to the overall architecture performance, area, and power consumption. The topology of the interconnection network is mostly undefined at the initial description of an architecture. After refinement or design space exploration, this topology is often altered by scaling the array, inserting new communication lines, or redirecting old ones.

Actual hardware, system, or architecture description languages can deal with these challenges only partially, or not at all. The limitations of these description languages are pointed in the following.

### Hardware Description Languages

Most of the actual methodologies for the design of CGRA architectures use some kind of hardware description language(HDL), such as Verilog [6] [97] or VHDL [5] [116], as depicted in Table 3.2. However, the use of HDLs for the design of CGRAs have the following drawbacks:

- The low level of abstraction of HDLs results in architecture descriptions that are hard to write and modify. For example, including a new component requires several modifications in the description of the interconnection network.

- The description is hard to scale, even if parameters are used; HDL generative functions, such as the VHDL statement *generate*, are not suitable for this purpose, as the designer cannot foresee the implications for all possible parameter combinations in the description.
- It is difficult to extract information from the code in order to perform analysis, formal verification, or derive software tools, such as a compiler or simulator.
- They do not provide features to describe hardware reconfiguration, particularly context based reconfiguration, which are commonly used in the design of coarse grained architectures. This implies that designers must describe the mechanism of reconfiguration indirectly.
- They do not capture the spatial distribution of modules in the architecture; there is no way to add positioning information for modules when using these languages. However, the design of CGRAs usually incorporate some spatial arrangement, such as an array or row.
- They cannot adequately describe the topology of networks because only point-to-point communication channels are allowed. For example, it is not possible to declare that a module is connected to its neighbors, since there is no notion of neighborhood.

### System description languages

Among all investigated works, only the CRC project uses SystemC in its design framework, as depicted in Table 3.2. System description languages, such as SystemC [21] or SystemVerilog [6], support higher abstraction level concepts. For example, communication channels may be modeled using message passing mechanisms, which do not require the declaration of implementation details. This might ease the functional description of intercommunicating modules, such as switch boxes and buses, but they do not solve the problems with scaling hundreds of point-to-point connections. Furthermore, system level languages are too generic and do not provide specific features to describe CGRAs. For example, neither SystemC or Verilog can describe spatial positioning or reconfigurability.

### Proprietary languages

Some successful CGRA design teams, such as the Silicon Hive and ADRES, use proprietary languages to describe their templates (see Table 3.2. The problem with these languages is that they are not available to the community. For example, designers of the ULIW architecture (Silicon Hive) use the language TIM, discussed in Section 3.1.2. Instances of the ULIW architectures are designed inside the company and delivered to the clients in VHDL or Verilog. Based on the features list of the TIM language [51], it seems to be an excellent example that higher-level of abstractions may have a positive impact on the description of CGRAs. However, it is not possible to discuss and compare details of TIM in this work, only its purposes.

### MAML

The extension of MAML, proposed by Kupryianov [68], can deal with all the challenges listed in the beginning of this section. It was developed with the same purpose as the language proposed in this work: CGADL. However, this approach has the following drawbacks:

- MAML is based at XML, and thus, it requires the design to deal with long and illegible XML tags, instead of a clean, concise, and readable description;

- The positioning system in MAML is done through mathematical definition of regions, called polytopes. Polytopes are way too complex (see, for example Figure 3.7) to write and understand. This complexity has two serious consequences: first, it makes the composition of heterogeneous arrays difficult; second, it makes the description of interconnection networks unclear and difficult to follow.

- MAML allows the description of regions in the space, for example, where all PEs are of the same type or have the same connection pattern. However, these regions cannot be resized (scaled).

### Summary

Table 3.2 depicts a comparison among all the languages discussed here. Most part of the development teams use a hardware description language (VHDL or Verilog), which fully supports synthesis and simulation. However, these languages make the description of scalable and heterogeneous arrays, as well as the description of scalable interconnection networks difficult and error prone. Moreover, these languages lack features to describe reconfigurability and spatial distribution of elements.

Table 3.2.: Description languages used in the development of CGRAs.

| Language | Used by | Synthesis | Simulation | Reconfigurability | Spatial distribution | Heterogeneous arrays | Scalability | Interconnection Network |
|---|---|---|---|---|---|---|---|---|
| Verilog [6] | KressArray<br>CRC<br>Piperench | √ | √ | | | | | |
| VHDL [116] | ADRES<br>Morphosys | √ | √ | | | | | |
| SystemC [21] | CRC | √ | √ | | | | | |
| Proprietary Languages | ULIW (TIM) | | √ | n.a. | n.a. | n.a. | n.a. | n.a. |
| | ADRES (XML) | | √ | n.a. | n.a. | n.a. | n.a. | n.a. |
| MAML | WPPAs | | √ | √ | √ | complex | complex | complex |
| **CGADL** | This work (CRC) | | √ | √ | √ | simple | simple | simple |

Proprietary languages, like Silicon Hive's TIM and the XML notation from ADRES, are said to have the necessary features to describe CGRAs at high abstraction level [51] [75]. It is impossible, however, to know it for sure, as the details about these languages are under concealment policy, and were never made publicly available.

MAML has an extension, whose features deal with the same challenges this work aims at. However, the description of heterogeneous arrays and scalable complex network still complex in this notation.

The coarse grained architecture description language, proposed in this work, can deal with all this challenges in a simple and concise way. A detailed discussion about CGADL will follow in Chapter 4.

### 3.3.2. Design and specialization of coarse grained architectures

This section discusses several works that deal with the design and specialization of coarse grained reconfigurable architectures.

#### Design space exploration for CGRAs

Most CGRA's design teams use parameterizable templates, for example: Silicon Hive [18], ADRES [75], WPPAs [67], CRC [96], KressArray [55], and Piperench [42]. For these teams, a large part of the design task consists in finding the parameterization that optimizes the architecture for a given set of applications. Each combination of parameter values (design decisions) generates an architecture instance that vary, for example in the granularity of their processing elements, in the structure of their network interconnection, or in the instruction set of their FUs. And each of these design decisions may lead to architecture instances that strongly differ in performance, area and power consumption. This design process is called *design space exploration*.

The problem is that each architecture instance has to go through a full evaluation cycle, including benchmark simulations, verification, synthesis, and estimations of area, performance and power consumption. Typical templates have an extremely large parameterization space, which make a trial-and-error approach unfeasible. This problem was stated by Mey et. al. during the development of the ADRES architecture as follows:

> Our retargetable architecture template and its compiler provide a solid base for architecture exploration of CGRAs. However, this framework only lets us quickly check different architecture options. **Finding a good architecture instance for a given application domain remains a big challenge**[emphasis added]. Exhaustive search is impossible because of the huge search space and the time required for each step. [*IEEE Design & Test of Computers*, May 2005, page 100 [75]].

Several works try to improve this design approach by focusing on a methodology to prune the design space and conduct the exploration:

- **Bossuet et. al.** proposes the use of a completely functional model of the architecture, so that architecture instances do not need to be synthesized to a netlist [13]. To evaluate performance and power consumption, Bossuet proposes two metrics: the architectural processing rate and the communication distribution. The architectural processing rate indicates the average number of operations per cycle achieved by the architecture instance. The communication distribution points out which components of the architecture have more activity, and therefore, consume more power. This approach aims to find a power-efficient architecture for the application domain.

The problem with this approach is that it does not consider the implementation area. The factor ”implementation cost” is therefore excluded.

- **Miramond and Delosme** presents a design exploration tool to map a DFG (application) onto an heterogeneous architecture composed of a general purpose processor and a reconfigurable array [80]. Thus, their proposal is focused on the application mapping. However, their method can deal with different architecture instances and different compilation methods, and can be used during the design phase too. Their advantage is that application mapping techniques are also considered as part of the exploration.

  The author of this thesis also tried a similar approach as Miramond, but with focus on the design space exploration [91]. This approach would consider simultaneously different instances and mapping strategies. The most appropriate instance is selected by using a multi-criteria optimality concept (Pareto optimality).

  Including mapping strategies in the design space exploration helps to co-develop architecture and compiler, but it also makes these two approaches extremely complex.

- **Chattopadhyay et. al.** argue that the exploration is difficult because the modelling level and the description languages, used in actual projects, are inappropriate[2] [22]. Their proposal is then a high-level modelling methodology which includes also a description notation. The search for the best instance, however, retains the same trial-and-error approach, and does not introduce any technique to prune the design space.

Summarily, current methodologies for the design of CGRAs fail because there are no known methods to prune the design space based on information about the set of applications the architecture aims for. Methods for design space exploration consist mostly of a combinatorial search, rather than a constructive method that gradually refines and specializes the architecture during its development.

### Other approaches to specialization

Some proposals to specialize CGRAs exist, which do not necessarily use the traditional design space exploration. To deal with this problem, several research groups propose general design guidelines, as depicted in Table 3.3, based on experimental results.

Kim et. al. proposes to share critical resources, such as multipliers, among several PEs [65][64]. These resources are used alternately by different PEs. Complex, pipelined interconnection networks are used to guarantee performance and a fair resource sharing. This centralization of critical resources can indeed reduce the implementation area, as not every PE have to include its own unit,

[2] This corresponds to the experience obtained during the development of this thesis.

Table 3.3.: Design guidelines for coarse grained reconfigurable architecture.

| Architecture | Guideline | Discussed in Section |
|---|---|---|
| NEC's DRP | Use a central state transition controller. | |
| IMEC's ADRES | Tightly couple the array to a VLIW processor. Use heterogene arrays. | |
| Kim et. al. [65] | Share critical resources and pipeline slow operations. | |
| Bansal et. al. [10] | Use multiple and distinct FUs inside each PE. | |

however, it augments the complexity of the application mapping and the power consumed in the interconnection.

In 2003, Bansal et. al. suggested that coarse grained arrays should be heterogeneous, and use PEs with more than one functional unit. They conclude that ”better performance for these (CGRA) designs are achieved by increasing the number of functionl units in individual PEs as compared to architecture in which each PE has only one functional unit [*Analysis of the Performance of Coarse-Grain Reconfigurable Architectures with different Processing Element Configurations*, Workshop on Application Specific Processors, 2003, [10]]. Since this work, an increasing number of CGRAs adopt this strategy, for example, Silicon Hives's ULIW, and WPPAs.

### Summary

This work introduces a completely innovative approach to the specialization of CGRAs: the inclusion of custom instructions in the functional units. The present work proposes more than design directives or guidelines. It establishes also a framework, composed of methods, tools, and algorithms, to evaluate the set of target applications and design the custom instructions.

## 3.4. Summary: analogy with the design of application specific processors

Around the middle of the last decade, designers of application specific processors (ASIPs) realised the importance of using description languages with higher abstraction levels in their design flows. Therefore, new description languages were developed that can easily describe particular features of the processor architecture. About this demand, the designer of the description language LISA [58], Andreas Hoffman, wrote: ”The research of the machine description language LISA and the associated tooling was originally motivated by the tedious and error prone task to write instruction-set simulators manually[*Architecture Exploration for Embedded Processors with Lisa*, 2003, page vii - preface, [58]]”. Between 1992 and 2003, several architecture description languages, such as nML [33], ISDL [49], EXPRESSION [50], LISA [58], and ArchC [8][101], were proposed in academia and industry. Most of them focused on the description of instruction set processors.

Nowadays, commercial platforms, such as Tensilica Xtensa [113] and Coware's LisaTek [26], employ these languages for the design of microprocessors.

During the same period, designers of ASIPs adopted design space exploration methodologies, which drifted from a combinatorial search to the specialization of the processor's instruction set. Instead of trying out several architecture instances, their approach consisted of profiling a set of applications to extract groups of instructions that are executed regularly and using the same execution pattern. A custom module would then be inserted in the processor's datapath to carry out this pattern as a single instruction. This methodology has been shown to speed up the performance and increase the number of instructions per unit of area of the processor.

These two approaches, the use of an architecture targeted description language and specialization by custom instructions, motivated the contributions in this work. **In this thesis, we apply these approaches to the design of coarse grained reconfigurable architectures: we develop a coarse grained-targeted architecture description language and we use custom instructions to improve and specialize the design of this architecture type.**

# 4. Description of Coarse Grained Arrays

In this chapter, a new architecture description language is proposed, which was tailored for the design of CGRAs: CGADL. CGADL was designed to describe CGRA templates. It has technical innovations that deal with specific challenges during the description of CGRAs, such as reconfiguration models, spatial distribution of PEs, scalability, interconnection network, and instruction-set based functional units. These technical innovations are key features of CGADL, that are not present in any other hardware description language.

This chapter starts with a discussion about several concepts used during the design of CGRAs and a brief view on how CGADL deals with them. The key features of CGADL, its semantics, and syntax, are presented in Section 4.2. Section 4.3 presents a method to estimate the hardware complexity of CGADL descriptions.

## 4.1. Motivation and contributions of CGADL

The design of the architecture description language CGADL was motivated by the specific challenges that appear during the design of spatially distributed coarse grained architectures (discussed in Section 3.3.1). These challenges are listed here together with a short overview of the CGADL feature that deals with them.

**Multi-context reconfigurability: context memories and finite state machines** CGADL defines models of context memories and finite state machines. Context memories, specially when used together with finite state machines, allow the designer to describe context-based reconfigurable processing elements and architectures. CGADL is the only description language that provides explicit model and language features for the description of context memories. That makes the design of reconfigurable components easier, since the details for implementation of the reconfiguration mechanism are embedded in the language. Other languages, such as the ones discussed in Section 3.3.1, cannot directly describe context-based reconfigurability. In these languages, an explicit composition of the circuit that implements the reconfiguration, with basic elements such as multiplexers, latches, memory blocks, and dedicated glue logic, is necessary.

**Spatial distribution of PEs: array-like positioning system** In a CGADL description, each element is logically bound to a position within a bi-dimensional grid. CGADL's semantics and syntax define a matrix notation that allows the identification, the placement, and the connection of elements using its position, instead of identifiers. When combined with other

CGADL features (discussed in the following), this positioning system eases the description of scalable arrays and interconnection networks.

**Homogeneous and heterogeneous arrays: composition and replication** CGADL describes the distribution of processing elements in the array by composing and combining them in blocks. That allows the designer to express homogeneous and heterogeneous arrays in an equally easy way.

**Scalability of the array: parameterization** In CGADL, blocks of processing elements can be arbitrarily composed using a matrix notation. This notation allows to quickly insert, remove, or reorganize elements of the array. Parameters can be used to control the number of PEs and geometry of the array, such that scaling the array may be as simple as attributing a new value to the parameter.

**Interconnection network: connection rules** CGADL introduces an innovative concept to describe the interconnection network at array level: the *connection rules*. A connection rule partitions the array into *regions*, such that all PEs within one region have their input ports connected the same way. Regions may express some positional characteristic, such as *all PEs in the first row*, *all PEs at the borders*, etc. Beside the partition, connection rules define how the PEs inside a region are connected. Connection rules express a set of connection statements: for example, *all PEs at the borders*(region) are connected *to an input port of the array*. Connection rules describe the interconnection network based on positions, and therefore independent of a specific distribution of PEs. As a consequence, the interconnection network scales with the array.

CGADL incorporates several concepts that are specific for the design of coarse-grained array-based architectures, such as an array-positioning system and a context-based reconfiguration. Nevertheless, the design of CGADL is generic and describes a broad palette of CGRAs.

## 4.2. CGADL - Semantics and Syntax

CGADL is a simple and intuitive notation language. It uses a PE centric approach: every module in the architecture is a processing element. Processing elements may process, store, and route data using functional units, register files, memory modules, shared buses, and switching boxes. This PE centric approach differentiates between pure interconnection lines and processing/storage/routing modules of the architecture. In this section, the CGADL's semantics and syntactic model are presented.

In CGADL, an *architecture* is a set of processing elements which communicate by *connections*, as depicted in the Unified Modeling Language (UML) diagram (figure 4.1). Every processing element in the architecture can be uniquely designated by a pair $(w, h) \in \mathbb{N} \times \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. The architecture has a so-called bi-dimensional *array* structure, because it may be graphically represented using a bi-dimensional grid, in which each processing element is placed in one node of the grid. Here, this spatial analogy will be frequently used to explain the concepts in CGADL.

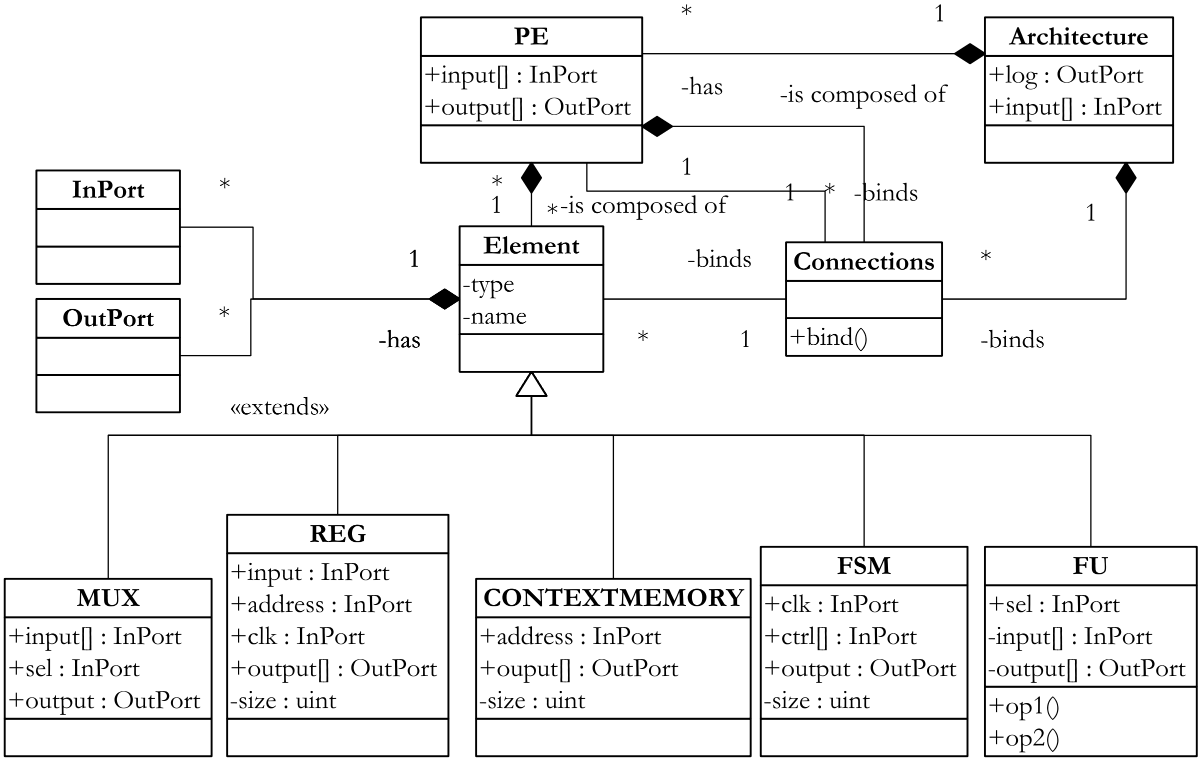


Figure 4.1.: UML diagram for the semantic model of CGADL.

A processing element is a structural composition of *elements* and its functionality is completely determined by its internal structure. An *element* is an abstract and generic concept: it represents a hardware module that encapsulates a behavior, but no internal structure. An element may have any number of input ports, from where data is read, and output ports, to where data is written. The specific behavior, as well as the exact number of output and input ports is not defined yet in the element; these are defined using *extensions*. An extension defines a behavior, which in turn determine how input data is processed to generate output data. CGADL has a set of basic hardware modules, which are extensions of an element, as depicted in Figure 4.1. CGADL modules are *MUX*, *REG*, *CONTEXTMEMORY*, *FSM* and *FU*.

In CGADL, *connections* encapsulate the concept of a communication channel: each connection transfers data from one point to another. At array level, connections bind PE ports; whereas within one PE, connections bind element ports. CGADL has one restriction: each connection must bind one output port to one or more input ports, but each input port cannot bound to more than one output port. The datatype and the wordlength is not specified explicitly; they are adopted only when necessary using annotations.

Further details to the semantics of CGADL are introduced in the following sections along with its syntax. In each case, the syntax will be first explained using syntax diagrams: a graphical alternative to Backus-Naur Form (EBNF). Following a path on the diagram from its starting point to the terminal symbol `EOF` (end-of-file) produces a syntactic valid description of an architecture in CGADL. The concise syntax diagram for a CGADL description can be seen in figure 4.2. The structure of aCGADL description file comprises three sections: PARAMETER, PE, and ARCH(see

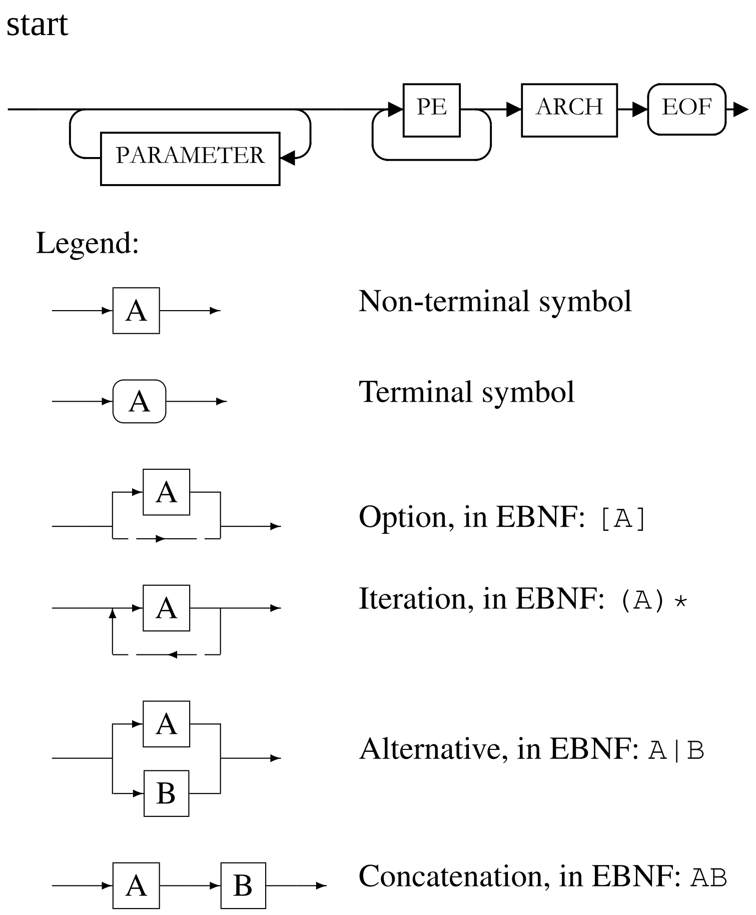


Figure 4.2.: Starting syntax diagram for a CGADL description. This diagram corresponds to the production of the start symbol in a EBNF notation. To read syntax diagrams, follow the arrows from left to right. Branches are possible divergent paths and only one of them should be followed at a time. A non-terminal symbol (rectangular box) represents a production rule by itself. A terminal symbol (rounded box) represents a language keyword oder a symbol.

figure 4.3). A CGADL description may have none, one, or several PARAMETER sections, but they always precede the other sections. At least one PE section must exist, whereas typically several are described. Finally, exactly one ARCH section completes the description. Each section has its own production rule, which will be discussed in the following. A full description of the language keywords, language symbols, and the grammar production rules in EBNF format is presented in Appendix A.2.

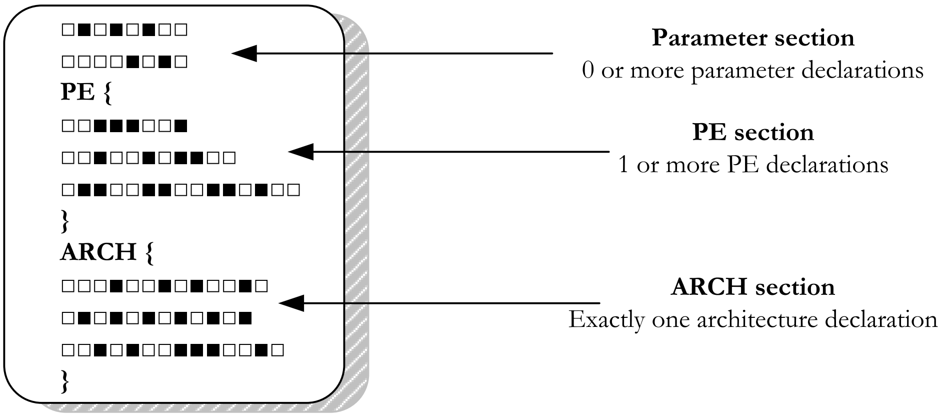

PE {
}
ARCH {
}
Parameter section
0 or more parameter declarations
PE section
1 or more PE declarations
ARCH section
Exactly one architecture declaration

Figure 4.3.: Structure of a CGADL description file.

## 4.2.1. The PARAMETER section

A CGADL file describes the architecture as a parameterized model, the architecture template. As discussed in Section 2.1.2, many characteristics of the architecture are not yet fixed in a template; parameters are used to describe which of them may vary. Further in the design process, an architecture instance is obtained by assigning a value to each parameter. To allow flexible and scalable templates, CGADL includes mechanisms to parameterize the following architecture characteristics:

- number of registers in a register bank;
- type of state machine;
- number of states in a finite state machine;
- number of contexts in a context memory;
- number of input ports of multiplexers and functional units;
- number of output ports of the context memory and functional units;
- array geometry: number of lines and columns; and
- interconnection network type and complexity.

The PARAMETER section consists of a list with an arbitrary number of parameters. A parameter declaration starts with the keyword PARAMETER followed by an identifier (a non-keyword name) and a set of integer numbers (see syntax diagram in figure 4.4). This set of numbers corresponds to the possible values that can be assigned to the parameter when defining an arquitecture instance. This values are declared, separated by commas, between square brackets. For example, the parameter `REGSize`, in figure 4.5, can have values 8, 16, or 32. A scope of integer values may be declared using the two-points scope operator (`..`). In this case, all integers inside the scope (including its limits) are possible values for the parameter. For example, the parameter `ArrayWidth`, in figure 4.5, can have values 1, 2, 3,. . . , 10, 50, or 20; and the parameter `CMSize` can assume the values 4, 5, 6,. . . , 15, and 16.

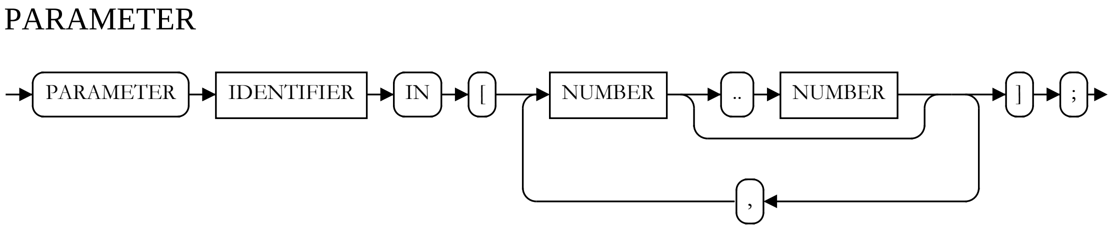


Figure 4.4.: Syntax diagram for a PARAMETER section.

Parameters are variables visible and valid in the subsequent PE and ARCH sections. Declared parameters are used as arguments when declaring CGADL elements, communication channels, and the geometry of the array.

## 4.2.2. The PE section

In CGADL, a processing element (PE) is a structural composition of *elements* (see Section 4), and the PE section is where these elements and their interconnections are declared. The PE section is composed of two subsections: the declaration and the connection sections (see syntax diagram in figure 4.6).

### The declaration section

The first part of the PE section consists of a list of one or more statements; each of them declares one element. The syntax diagram of one declaration statement is depicted in figure 4.7. The elements that can be declared are multiplexers (MUX), register banks (REG), finite state machine (FSM), context memories (CONTEXTMEMORY), functional units (FU), input ports (INPORT),

```
PARAMETER REGSize IN [8 ,16 ,32];
PARAMETER CMSize IN [4..16];
PARAMETER ArrayWidth IN [1..10 ,50 ,20];
```

Figure 4.5.: Example of a PARAMETER section.

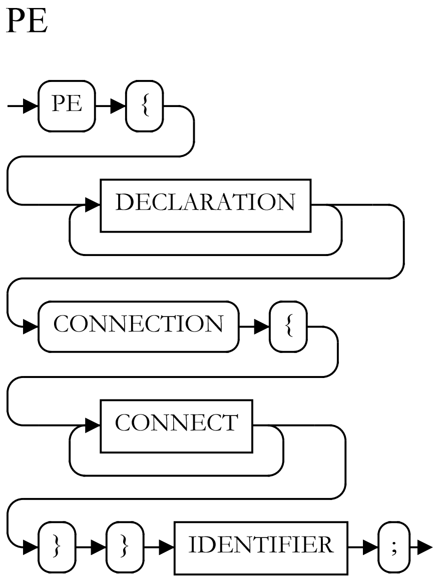


Figure 4.6.: Syntax diagram of the PE section.

and output ports (OUTPORT). These elements are classified in simple types (TYPE), parameter types (PTYPE), and data types (DTYPE), as follows:

**TYPE** The simplest type of declaration. It is used for elements that do not need arguments in their declaration. Multiplexers are in this class; they are declared with the keyword MUX and a name, which is used as identifier for this specific multiplexer, for example:

```
MUX dinMux1;
```

**PTYPE** This type of declaration is reserved for elements that can receive parameters as arguments. Arguments for this type must be a value or a parameter, which was declared earlier in the PARAMETER section. Register banks, finite state machines and context memories are in this class; they are declared with a keyword (REG, FSM, or CONTEXTMEMORY), a name (used as identifier), and the identifier of a parameter or a value, for example:

```
REG dataRegisterSet(REGSize);
FSM myFSM(FSMSize);
CONTEXTMEMORY myContextMem(12);
```

**DTYPE** This type is used specifically for the declaration of functional units. An FU is declared with the keyword FU, a name (used as identifier for the FU), and a list of operations that can be executed with this FU, for example:

```
FU myALU(uadd, usub, mul, div);
```

**PORT** This type of declaration is used for the input and output ports. The input ports of a PE are declared together in a single statement composed of the keyword INPORT and a number,

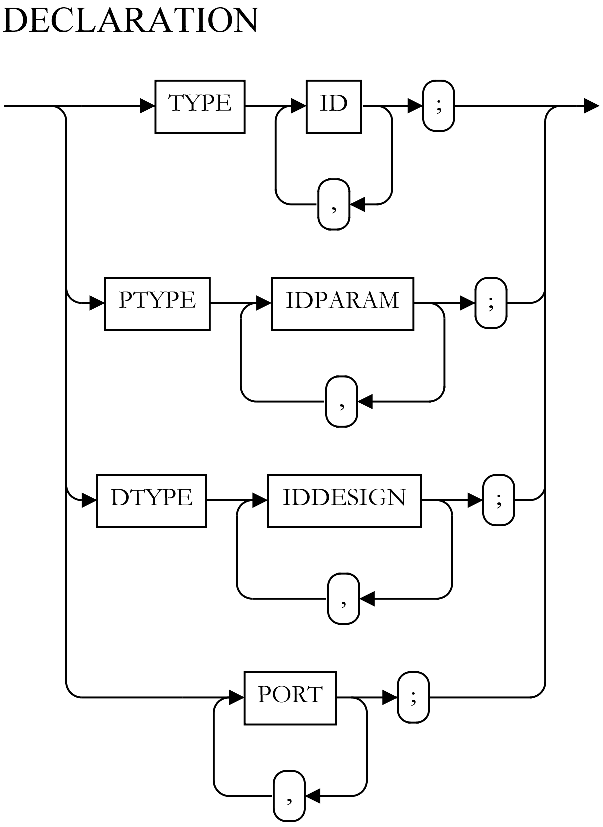


Figure 4.7.: Syntax diagram of the declaration section.

which indicates the number of input ports of the PE. Similarly, the output ports of a PE are declared together in a single statement composed of the keyword OUTPORT and a number, which indicates the number of output ports of the PE. Parameters cannot be used in place of the number, for example:

```
INPORT ( 8 ) ;
OUTPORT ( 4 ) ;
```

Declarations of elements can appear in any order, and the name of declared elements cannot be repeated. Details for the declaration and semantics of each element will be discussed later in this section. The next section discusses how the interconnection between these elements is described in CGADL.

### The connection section

The interconnection between elements of one PE is described in the connection section. Each declared element receives one ordered list of connections, named after that element. Each entry in the list corresponds to an input port of this element in the same order. The input ports of an element are ordered, as depicted in figure 4.8. The first entry in the list declares a connection that terminates in the input port 0, the second entry declares a connection to the input port 1, etc. The content of each entry determines the output port which initiates the connection. For example, the line

```
myMux(dataRegister[0], INPORT[0], myContextMem[4]);
```

declares a connection between the output port 0 of dataRegister and the input port 0 of myMux, the PE's input port 0 and the input port 1 of myMux, and the output port 4 of myContextMem and the input port 2 of myMux.

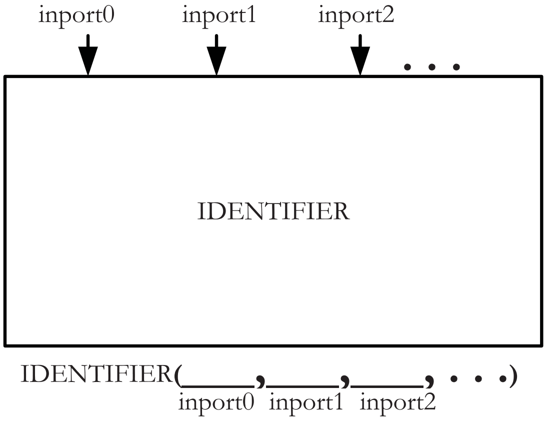


Figure 4.8.: Generic schema for the connections of a CGADL element.

The connection section has the following properties:

- Each element listed in the declaration section of a PE must receive one connection list; otherwise, its input ports are considered to be unconnected. The assignment between elements and their connection lists can happen in any order.
- The number of input ports of an element is determined in the connection section; it corresponds to the number of entries in the connection list.
- The number of output ports of an element is determined in the connection section; it corresponds to the highest index declared to an output port of this element.
- An entry in the connection list can be declared as an ordered set of output ports. For example, instead of declaring

  ```
  myMux(dReg[0], dReg[1], dReg[2]);
  ```

  it is possible to write

  ```
  myMux(dReg[0..2]);
  ```

  both of which indicate the connection of output ports 0, 1 and 2 of dReg, to the input ports 0, 1, and 2 of myMux.

- Parameters, declared in the PARAMETER section, can be used instead of an output port number. Moreover, the value of parameters can also be decreased up to a non-negative number. For example, the line

  ```
  myMux(dReg[0..REGSize−1], myContextMem[4]);
  ```

  connects the output ports $0, 1, \ldots, k$ of dReg to the input ports $0, 1, \ldots, k$ of myMux, where REGSize is a parameter and $k =$ REGSize $- 1$. The total number of input ports of myMux, in this case, is REGSize $+ 1$ (including the input port connected to myContextMem).

The several elements that can be used to compose one PE, their semantics, and their syntax are explained in more details in the following. A complete example for the declaration of elements within a PE will be provided later (see Figure 4.16).

### Multiplexers - MUX

The *MUX* element represents generic multiplexers. Multiplexers are common components in reconfigurable datapaths. They are used for routing data, flags, or control signals between different elements. In CGADL, a MUX element has two or more input ports, one output port, and one selection control port, as depicted in figure 4.9. The selection port indicates which of the input ports will be forwarded to the output port. The connection with other elements is defined in the connection section; the last entry in the input port connection list of a multiplexer is reserved for the selection control port. The number of input ports of a multiplexer is defined by the number of entries in its connection list. The input ports of the multiplexer dinMux1, in figure 4.10, are connected to the input port 0 of pe1, the output ports of the register set. The selection control port is connected to the fourth output of the context memory. Notice that the number of input ports of dinMux1 depends on the parameter REGSize.

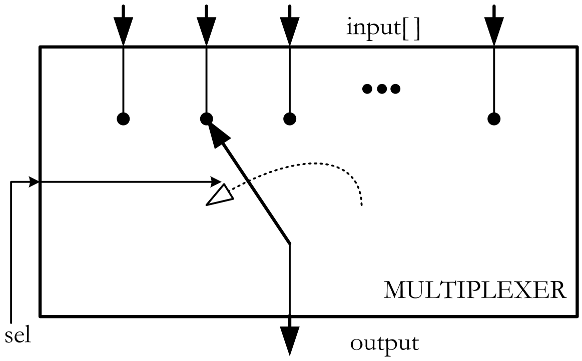


Figure 4.9.: Model of a multiplexer (MUX) in CGADL.

### Register Set - REG

The *REG* element provides the behavior of a hardware register set (see figure 4.11). A register set element has exactly one input data port, exactly one input address port, and *size* output ports, where *size* is also to the number of registers in the set. At a clock transition, the data present at the

```
PARAMETER REGSize IN [4,8];
PARAMETER CMSize IN [4..12];
PE {
  //Declaration section
  MUX dinMux1;
  FU myFU(addSub, umult, shift);
  REG regSet(REGSize);
  CONTEXTMEMORY myCM(CMSize);
  CONNECTION {
    //Connection section
    dinMux1(INPORT[0], regSet[0..REGSize-1], myCM[4]);
    myFU(myCM[3], dinMux[0]);
    regSet(myCM[2],myFU[0]);
  }
} pe1;
```

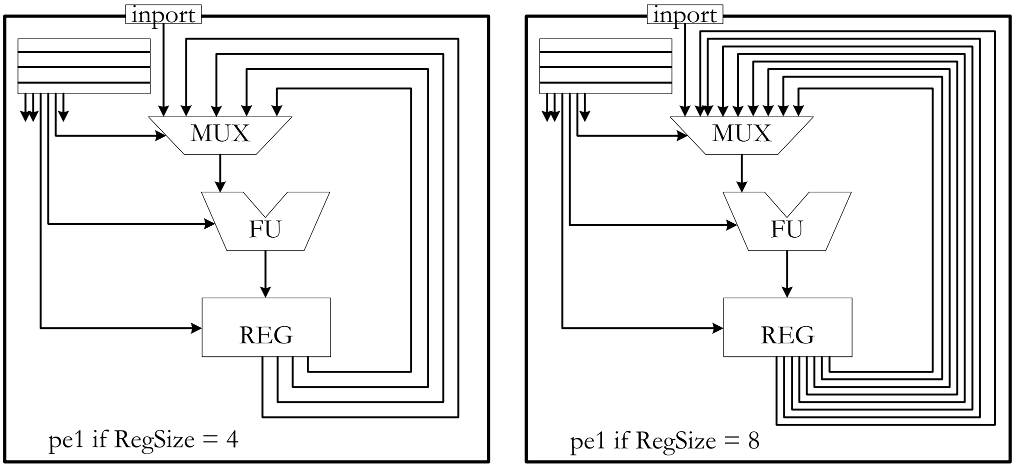


Figure 4.10.: Example of a multiplexer declaration and connection.

input port is stored in the position indicated by the address port. Any entry can be read directly, at any time, using the respective output port.

Register sets are declared with the keyword REG, a name (used as identifier), and a parameter or value indicating the *size* of the set. The declaration of the register set `regSet`, in figure 4.10, is an example. A register set has only two input ports; and therefore its connection list is composed of two entries. The first input port is reserved for the address signal, whereas the second is reserved for the data input port. In the example, regSet is addressed by the context memory (output port 2) and stores the data present at the output port 0 of the functional unit.

### Finite state machine - FSM

The *FSM* element provides the behavior of a finite state machine. The FSM is a clocked hardware block with one input *condition* port and one output port, as depicted in figure 4.12. At each clock transition, a new state for the machine is chosen depending on the signal at the *condition* port. This

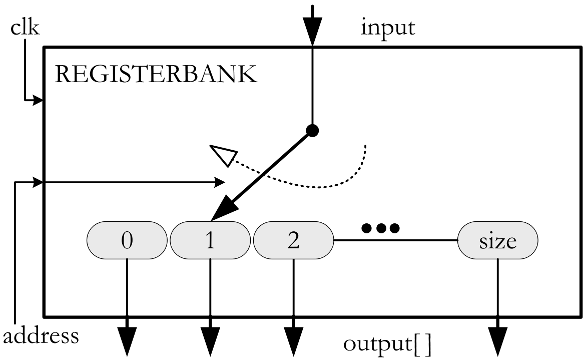


Figure 4.11.: Semantic model of a register set (REG) in CGADL.

new state determines the data to be presented at the output port. The *size* parameter controls the maximal number of internal states used in the FSM.

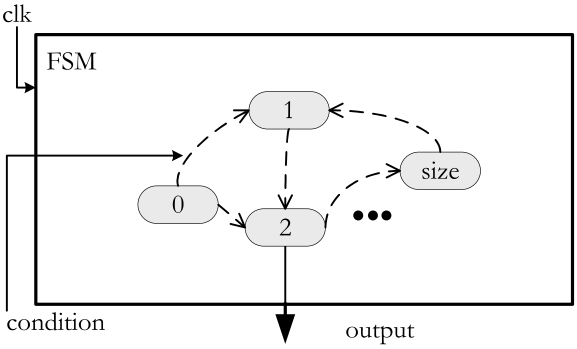


Figure 4.12.: Semantic model of a finite state machine (FSM) in CGADL.

Finite state machines are declared with the keyword FSM, a name (used as identifier), and a parameter or value indicating the number of states in the machine; that is, its *size*. The finite state machine is a programmable element, which receives an additional description, a *program*, that describes behavior. This program must abide by the following characteristics:

- The program describes states and transitions between these states. Transitions occur on a discrete-time basis, as for example, at positive transitions of a clock signal.
- The program has at most *size* states.
- Transitions between states are only dependent on the actual state and the data at the *condition* port.

CGADL makes no assumptions about how the FSM program is written; its style may be a table, a comma separated value file, XML, etc.

The connection list of a finite state machine contains only one entry, which corresponds to its input *condition* port.

### Context memory - CONTEXTMEMORY

The context memory element is a latch memory which implements context-based reconfiguration. The context memory element has one input address port and $n$ output ports, as depicted in figure 4.13. Each entry in this memory model stores one context with $n$ fields. Each field corresponds to one output port. A field stores the signal to control another hardware module such as multiplexer select signals, register addresses, and FU operation codes. At any time, the context memory writes to its outputs the entry selected by the address port. The *size* parameter controls the number of entries in the context memory.

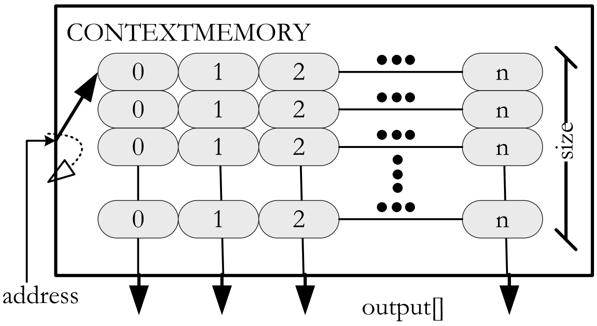


Figure 4.13.: Semantic model of a context memory (CONTEXTMEMORY) in CGADL.

Context memories are declared with the keyword CONTEXTMEMORY, a name (used as identifier), and a parameter or value indicating the number of entries (contexts) in the memory; that is, its *size*. The context memory is a programmable element, which receives an additional description, a *program*, that describes its behavior. This program must abide by the following characteristics:

- The context memory program describes contexts. Each context is an ordered list with $n$ data fields, which are presented at output ports if the context is active; the first field describes the data to be presented at output port 0 of the context memory, the second field corresponds to output port 1, etc.
- The context memory program describes at most *size* contexts.
- Each context is identified with an unique information tag, for example, an index number or identifier. If this tag is presented to the address port, the context is made immediately active.

CGADL makes no further about how this program is written; for example, it may be a table, a comma separated value file, XML, etc.

The connection list of a context memory requires only one entry, which corresponds to its input address port. The number of output ports of a context memory is equal to the largest output port index, used to connect the context memory to other elements, plus one (as the CGADL indexing system starts at 0). That determines also the number of fields in each context.

### Functional Unit - FU

The FU is the most flexible element in CGADL: its behavior is defined by the designer. FUs transform, route and store data, or replace and extend the set of elements discussed previously.

The FU models a hardware module with an operation select port and an arbitrary number of data input and output ports, as depicted in figure 4.14. During the FU operation, one specific behavior is chosen using the *operation* port. The corresponding processing algorithm (function) uses the data present at input ports, process them, and presents the result at output ports. Therefore, the designer may model any other custom element by declaring an FU and providing a behavior for it.

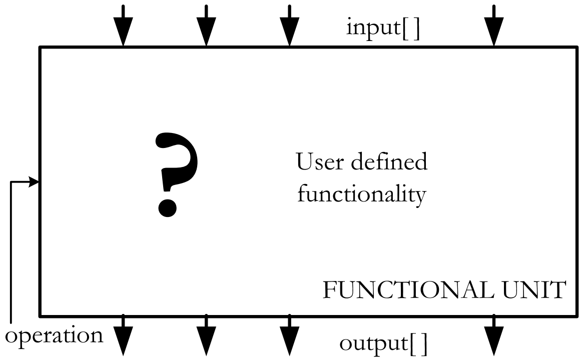


Figure 4.14.: Semantic model of a functional unit (FU) in CGADL.

Functional units are declared with the keyword FU, a name (used as identifier), and an ordered list of operation identifiers. When an operation identifier is presented to the operation port, the FU executes the corresponding operation using the data from input ports as arguments, and transfers the results to output ports. As an example, a functional unit with two input and one output ports is considered. This functional unit comprises one unsigned addition (called *uadd*) and one unsigned multiplication (called *umult*), and is declared as follows:

```
FU myFunctionalUnit(uadd, umult);
```

The *uadd* operation receives the index with value 0, whereas the *umult* receives the index with value 1: according to the order in which they are listed. If the operation port is excited the value 1 (index of *umult*), the FU multiplies the values at the input ports and transfers the data to the output port.

The connection with other elements defines the number of input and output ports of the functional unit. The number of input ports is equal to the number of entries in the connection list. The first entry in the list is reserved to the operation port. The number of output ports is equal to the largest index, which is used when connecting the FU to other elements, plus one (since the CGADL indexing system starts at 0).

The behavior for each operation of the FU is defined by the designer in a separate description, which must abide by the following characteristics:

- Each operation must have its behavior defined separately. A specific format for the behavior description is not fixed by CGADL. It can be, for example, a C function, an algorithm, or a data flow graph.

- Each operation processes at most $n$ arguments, $n$ being the number of input ports of the FU. Similarly, each operation produces at most $o$ output values, $o$ being the number of output ports of the FU. The number of input ports and output ports of a functional unit is defined in the connection section, as discussed previously.

- The description of an operation is triggered only when the corresponding index is presented at the operation port.

The description of an operation's behavior will be particularly important in this work when integrating custom instructions (Chapter 5). In this sense, some guidelines will be included in the following example, that explain how the behavior of the operations were described in C for the purposes of this work. Consider a functional unit myFU, with three input ports and one output port, that was declared in a CGADL description as follows:

**FU** myFU(uAdd, uMAC);

The behavior of uAdd and uMAC can be described in the programming language C with the code depicted in figure 4.15. Note that each declared operation has its own function, and each operation processes at most 3 inputs (args) and produces at most 1 output.

```
unsigned int uAdd(unsigned int args[3]) {
    return args[0] + args[1];
}
unsigned int uMAC(unsigned int args[3]) {
    return args[0] * args[1] + args[2];
}
```

Figure 4.15.: An example for the description of operation's behavior. In this work, the programming language C is used to describe the behavior of FU's operations.

### Input and output ports for the PE - INPORT and OUTPORT

The input ports of a PE transfer data from the exterior of the PE to its internal elements. Each input port corresponds to one communication line. The output ports transfer data, which are produced, stored, or routed within one PE to its external environment. They correspond to multiplexers whose output ports are visible only outside the PE. Like a multiplexer, an output port has several input ports and one selection control port. The selection control port indicates which input signal is to be transfered to the PE's external environment.

Input ports are declared with the keyword INPORT, and a value corresponding to the total number of input ports in the PE. Output ports are declared with the keyword OUTPORT, and a value corresponding to the total number of output ports in the PE. If multiple declarations for the input or output ports are present, only the first one is valid. There are no parameters to control the number of input ports or the number of output ports of the PE.

The incoming data of input ports are not accessible inside the PE; therefore, there is no connection list for the input ports in the connection section. The connection list for output ports is similar to that of multiplexers. The number of input signals entering an output port is equal to the number of entries in its connection list; the last entry is reserved for the selection control port. The output signal of the output port cannot appear in the connection section, since it is visible only outside the PE.

### Summary of the PE section

A summary for the semantics of CGADL elements is depicted in Table 4.1.

Table 4.1.: Summary of CGADL elements.

| | Number of input ports | Number of output ports | Parameters | Comments |
|---|---|---|---|---|
| MUX (TYPE) | defined in the connection section | 1 | input ports | last input port is reserved for the selection control port |
| REG (PTYPE) | 2 | defined in the declaration section | number of registers, number of output ports | first input port is reserved for the address port , clock |
| FSM (PTYPE) | 1 | 1 | number of states | clock |
| CONTEXT MEMORY (PTYPE) | 1 | defined in the connection section | number of contexts, number of output ports | |
| FU (DTYPE) | defined in the connection section | defined in the connection section | operations, number of input/output ports | first input ports is reserved for operation selection |
| INPORT | 0 | defined in the declaration section | | input port is not visible inside the PE |
| OUTPORT | defined in the declaration section | | | output port is not visible inside the PE |

An example of a complete description for a processing element is depicted in figure 4.16. This description corresponds to the basis PE, discussed in Section 2.1.4. Since this basis PE will be used in several discussions along this document, it will be referred to as the PE *Bianca*.

## 4.2.3. The ARCH section

The ARCH section describes the distribution and the interconnection of PEs in the architecture; that is, it describes the structure of the architecture rather than its functionality. As discussed before, every architecture in CGADL follows an array-like structure; each PE is uniquely identified by a pair $(w, h) \in \mathbb{N} \times \mathbb{N}$, which is said to be its *position*. An architecture instance has a defined geometry; that is, there exist $W \in \mathbb{N}^*$ and $H \in \mathbb{N}^*$, such that $w < W$, and $h < H$. Moreover,

```
PARAMETER REGSize IN [2,4,8,12,16,24,32];
PARAMETER FSMSize IN [12,16,24];
PARAMETER CMSize IN [4..12,16,24,32];
PE {
  //Declaration section
  MUX dinMux1, dinMux2;
  MUX finMux1, finMux2;
  REG dRegSet(REGSize);
  REG fRegSet(REGSize);
  OUTPORT(8);
  FSM myFSM(FSMSize);
  CONTEXTMEMORY myCM(CMSize);
  FU myFU(addSub, mult, and, or, xor,
          not, shift, sel, compare,
          bitAnd, bitOr, bitXor, bitInv);
  INPORT(4);
  CONNECTION {
    ...
  }
} Bianca;
```

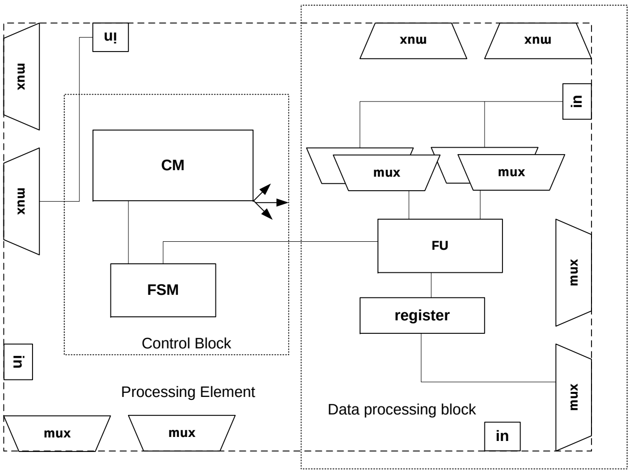


Figure 4.16.: CGADL description and schematic diagram of the PE *Bianca*.

there exist only one PE for every pair $(w, h)$ where $w < W$ and $h < H$. This model is analogous to a bi-dimensional grid, as depicted in figure 4.17.

In the ARCH section, PEs are like *black-box* components: only their input and output ports are visible, and elements inside a PE cannot be accessed directly. The ARCH comprises three main parts, as depicted in its syntax diagram (figure 4.18):

**ARRAY** The ARRAY section describes the distribution of PEs within an array; that is, their *placement*. Any of the PE types declared in the PE section may be used. The notation used in CGADL to describe the array is an innovative feature, which is not present in any other architecture description languages. It allows to describe the placement of PEs in an array using

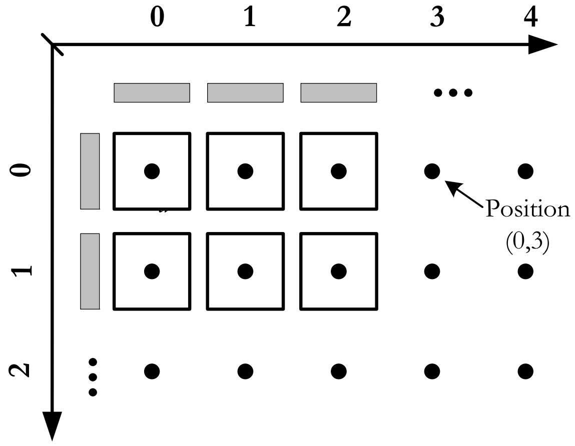


Figure 4.17.: Positional model for the distribution of PEs in the architecture: an array-like structure. A $2 \times 3$ architecture instance is depicted on the grid; every position has exactly one PE, which can be uniquely identified using the positioning system.

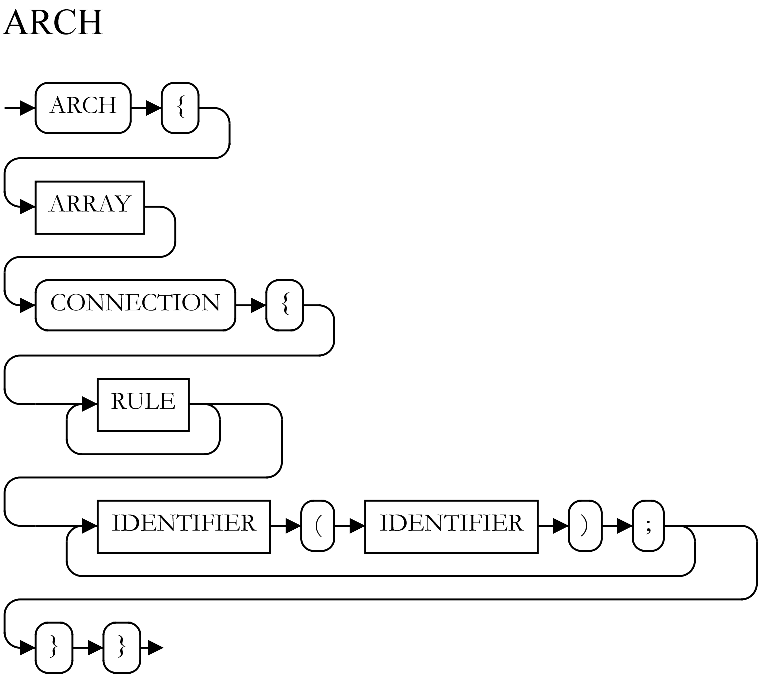


Figure 4.18.: Syntax diagram of the ARCH section.

parameters, such that the array can be scaled easily.

**CONNECTION-RULE** The CONNECTION-RULE section describes the interconnection network between PEs within an array. In CGADL, the description of an interconnection network is made by using *connection rules*. Connection rules describe regular interconnection structures that apply for a coordinate (or group of coordinates) in the array. For example, a rule may state that every PE of the array, except for the ones at the borders, is connected to its neighbor PEs. The concept of connection rules is an innovative feature, which is not present in any other architecture description languages. It allows a much simpler and scalable description of regular interconnection networks.

**BINDING** The binding section assigns a connection rule to a specific array (placement). This binding requires the connection rule and the array to be mutually compatible; that is, every connection must start at an input port, and terminate at an output port.

In the following, each one of these sections is discussed in detail.

### The array description

The ARRAY section describes uniquely the distribution of PEs within an array. The interconnection between PEs is not fixed at this point. The ARRAY section comprises two parts: the first part *concatenates* PEs to create composite, arbitrarily irregular blocks; the second part defines an architecture array by replicating a given block in the grid structure (see figure 4.19).

This way to describe the array, composition and replication of PE blocks, is an innovative feature introduced by this work. Other architecture description languages, such as ArchC, Verilog, and VHDL, force the designer to compose the array manually or using a generative procedure. For example, in ArchC [8][101], every PE must be individually instantiated. In Verilog[6] or VHDL[5], a *generate* function can be used to automate the declaration of homogeneous blocks; however, the description of heterogeneous blocks is difficult and error prone.

In the first part of the array declaration, PEs are grouped in blocks composed with PEs of the same type (homogeneous) or different types (heterogeneous). These blocks can be further concatenated to describe larger and more complex blocks. Finally, an array is built by replicating previously declared blocks in a grid structure.

The semantics and syntax of CGADL expresses the array-like arrangement used for the design of coarse grained reconfigurable architectures. CGADL uses a matrix notation to describe an horizontal or vertical concatenation of processor elements or other blocks, as indicated by the syntax diagram in figure 4.20. The items (PEs and other blocks) within a block are organized in lines and columns, as follows:

- Lines correspond to an horizontal concatenation of items. Lines can be declared as a list of items separated by a space(`" "`) or comma(`","`), for example,

```
myLineBlock = [pe1, pe2, pe2, pe1];
aLineBlock  = [pe3 myBlock];
```

- Columns correspond to a vertical concatenation of items. Columns can be declared as a list of items separated by a newline or a semicolon(`";"`). ,for example,

```
myColumnBlock = [pe1; pe2; pe2; pe1];
aColumnBlock  = [pe3
                  myBlock];
```

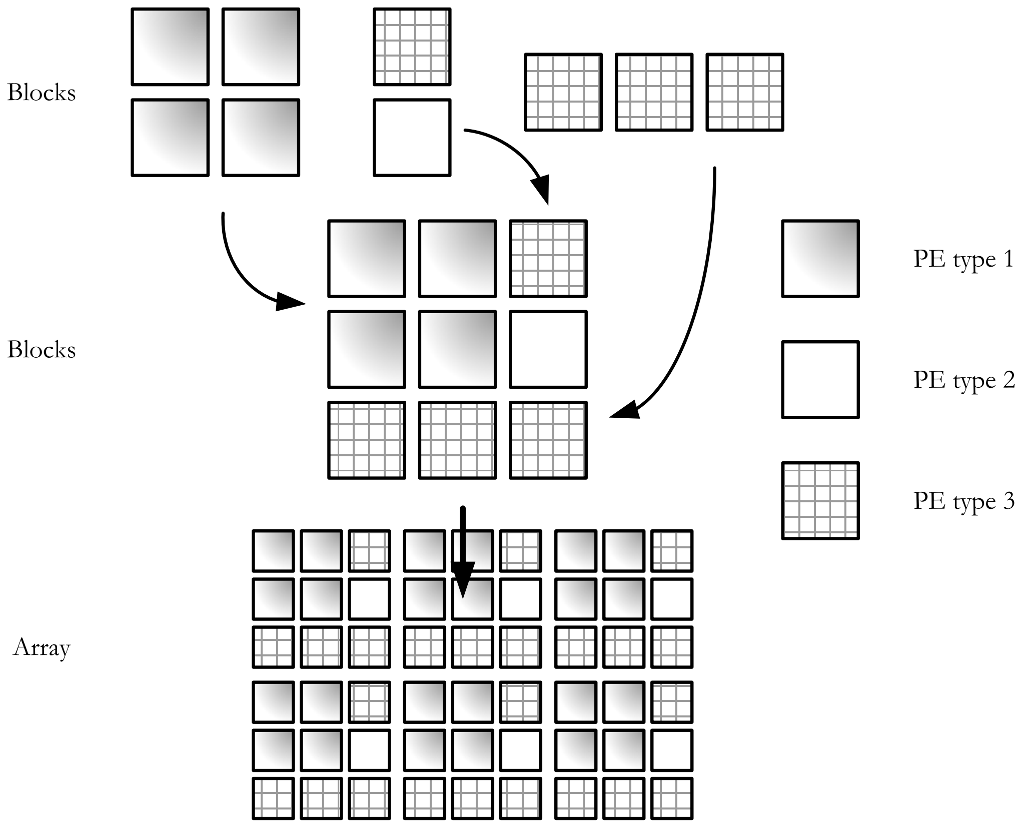


Figure 4.19.: Composition of a regular array based on blocks.

- Horizontal and vertical concatenations may be combined, if the resulting block is convex and dense; that is, if the block constitutes a rectangular matrix without empty positions (holes) inside. Two conditions must hold simultaneously: all rows must have the same number of PEs, and all columns must have the same number of PEs.
- Each block receives an unique identifier which cannot be equal to the identifier of any other block or PE.

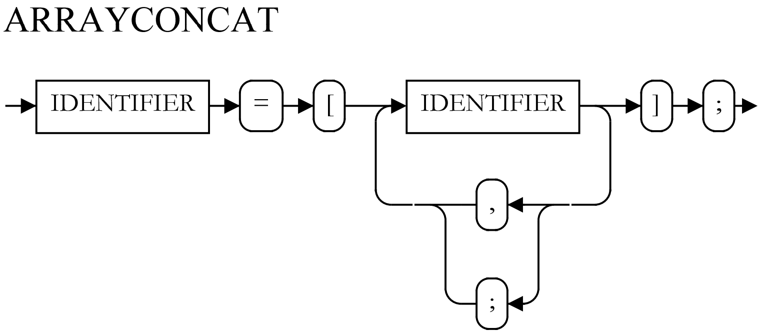


Figure 4.20.: Syntax diagram for the array declaration.

Examples for the composition of blocks are depicted in figure 4.21. *aBlock* and *cBlock* are valid blocks, whereas *bBlock* and *dBlock* have lines with different number of PEs; and thus constitute non valid blocks.

In the second, and last, part of the ARRAY section, one or more arrays may be declared. An array determines the placement of PEs in the architecture. It is composed with a regular repetition, in lines and columns, of exactly one block. The declaration of one array is done with the keyword ARRAY followed by three parameters: the number of vertical repetitions, the number of horizontal repetitions, and the block to be repeated. The number of vertical and horizontal repetitions may be parameterized, which allows an easily scaling of the architecture.

Examples are depicted in figure 4.21. The array *aBlockArray* describes an architecture template, whose geometry is defined by the parameters `width_param` and `height_param`. This template has no fixed geometry, and all possible value combination for these parameters generate a different architecture instance. If, for example, `width_param` and `height_param` are both equal to 2, then `aBlockArray` corresponds to an architecture instance with $4 \times 4$ PEs.

### The array interconnection network

The CONNECTION-RULE section comprises three parts: the first part declares connections between PEs, or between input ports of the array and input ports of PEs; the second part declares connections to the outports of the array, that is, communication lines that transfer data to the outside of the array; and the third part, declares connections that for some reason will not be used.

CGADL introduces an innovative concept to describe the interconnection network at array level. Instead of describing explicit connections between PEs ports, the array is partitioned into *regions*, such that all PEs within one region have their input ports connected the same way. Thus, for each region, there is only one description for how the elements are to be connected. The figure 4.22 illustrates this idea. All internal PEs of this instance have their input ports connected directly to their adjacent neighbors. Hence, only that region (internal PEs) and its connection pattern (adjacent neighbors) need to be described.

Most of the existing architecture description languages(ADLs), force the designer to connect every port individually. In CGRAs, this can turn out to be an exhaustive and error prone task due to size of the array. Individual connections also hinders the scalability of the array, because the insertion of new elements may imply rewriting several existing connections. Some ADLs use generative procedures, as the *generate* function used in Verilog and VHDL [116]. These languages are still closely dependent on the specific placement of the array.

To describe regions in an array, the coordinate system discussed in section 4.2.3 is used. Assume an array of arbitrary width $W$ and height $H$. One array position can be designated absolutely, using a coordinate $(w, h)$, from (0,0), in the upper left corner, to ($W-1$,$H-1$) in the lower right corner of the array. A position can also be designated relative to the array limits using the keyword END, which describes the last position in a row or column; for example, the coordinate (END$-$1,END) indicates the PE in the next-to-last row and the last column. To describe a set of different positions, a colon operator is used, as follows:

```
(starting row:step:end row, starting column:step:end column)
```

```
PARAMETER width_param = [1 ,2];
PARAMETER height_param = [1 ,2];
...
ARCH {

  aBlock = [pe1, pe2; pe3, pe4];
  bBlock = [pe1, pe1, pe1
            pe2, pe2];            // Invalid
  cBlock = [pe1, pe1, pe1];
  dBlock = [cBlock ; aBlock];               // Invalid

  ARRAY(3 ,3 ,pe1) homogeneousArray;
  ARRAY(1 ,2 ,cBlock)   cBlockArray;
  ARRAY(width_param, height_param, aBlock) aBlockArray;

} CONNECTION {  ...
```

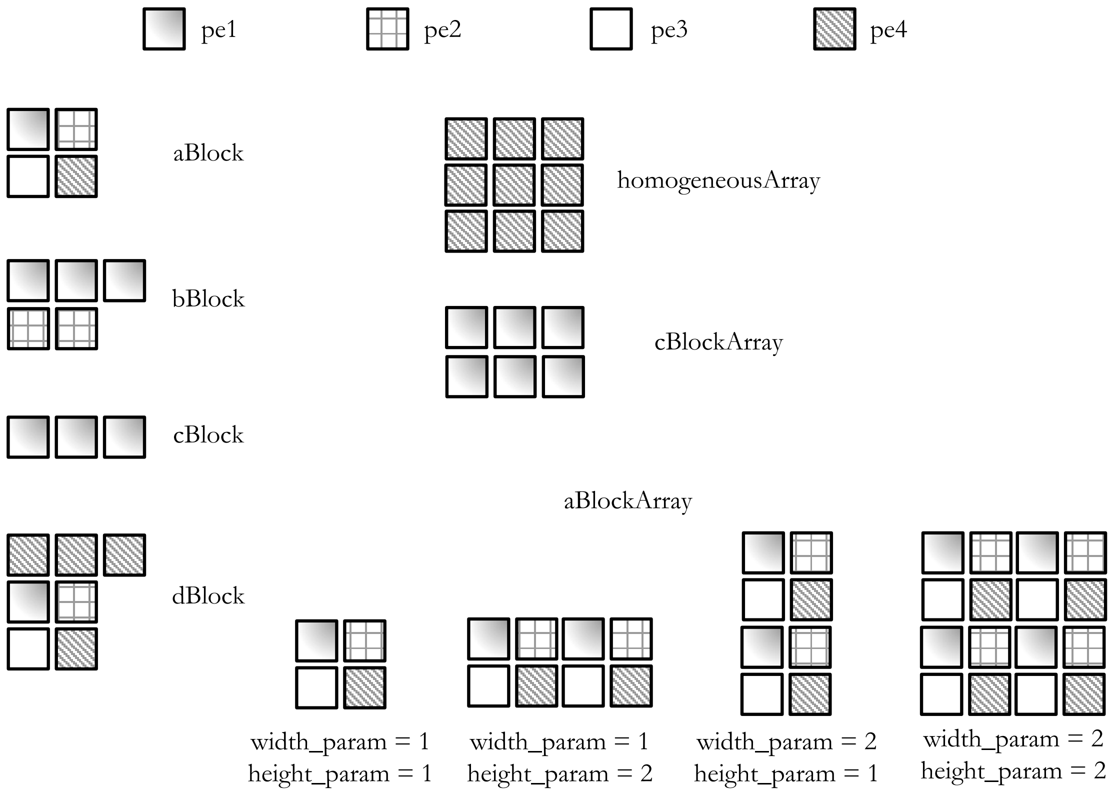


Figure 4.21.: Declaration of blocks and arrays in the ARCH section: examples. The *aBlockArray* is a template, which can generate up to 4 different instances depending on the values of *width_param* and *height_param*.

There are two parts divided by a comma. The first part declares a set of rows and the second a set of columns. The region comprises all positions $(w, h)$ corresponding to the intersection of these two sets. Examples on how to choose (ir)regular sets of PEs are depicted in figure 4.23.

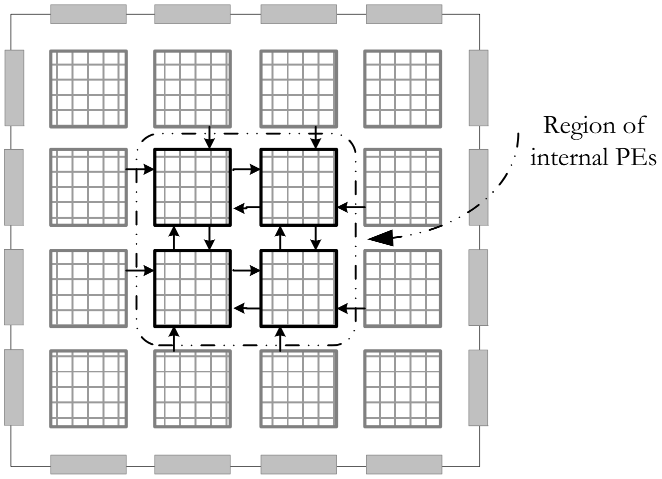


Figure 4.22.: Description of the interconnection network topology based on a region. Regions are groups of PEs that are connected in the same way. For example, all internal PEs in this array are connected in the same way: internal PEs build up a *region* that abide by the same interconnection *rules*. CGADL uses this concept to describe the interconnection network. The design of an interconnection topology based on regions and rules is an innovative feature introduced by this work.

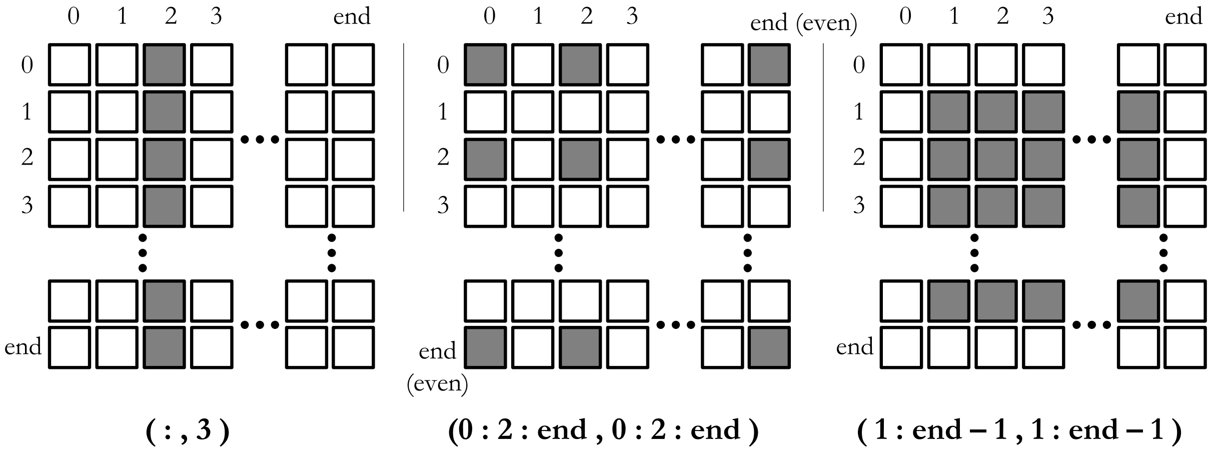


Figure 4.23.: Regions representing subsets of PEs in the array. The first example selects all PEs in the fourth column. The second selects all PEs that are simultaneously in even columns and even rows. Finally, the internal PEs of an array may be selected as shown in the third example.

A set of regions that partition the architecture array is called a *connection rule*. All PEs within a region receive the same connection pattern to their input ports. Therefore, two regions within a connection rule cannot have any common position. If that happens, one PE follows two different connection patterns, which produces an error. Moreover, a connection rule is complete only if all positions in the array is part of one region. The syntax diagram for declaring a connection rule is depicted in figure 4.24. Each rule is introduced with the keyword RULE, which is followed by the description of the rule and a name for later identification.

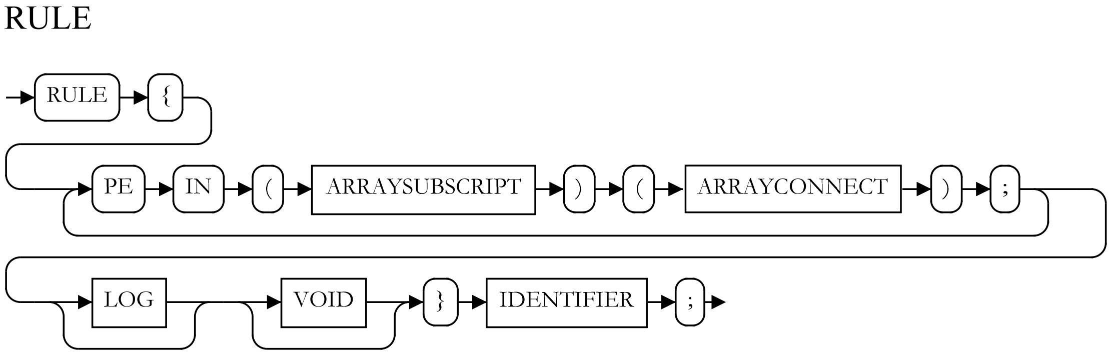


Figure 4.24.: Syntax diagram of a connection rule.

Every PE in a region is connected using a connect-by-position mechanism similar to the one used in the PE section: a list represents input ports in increasing index order for a PE of the region, and each entry in this list declares the source port where the connection comes from. As an example, the code depicted in figure 4.25 connects all internal PEs in an array in the following way: input port 0 is connected to output port 2 of the adjacent PE in north; input port 1 is connected to output port 3 of the adjacent PE in east; input port 2 is connected to output port 0 of the adjacent PE in south; and finally, input port 3 is connected to output port 1 of the adjacent PE in west. That connection is known as *nearest-neighbors*, and is depicted in Figure 4.22.

```
ARCH {
  B=[ pe1 pe2 ; pe2 pe1];
   ARRAY(width,height,B) myArray;
   CONNECTION {
   RULE {
    PE IN (1:END−1,1:END−1)
    (REL_COORD( −1 ,0)[2] ,REL_COORD(0 ,1)[3] ,
    REL_COORD(1 ,0)[0] ,REL_COORD(0 , −1)[1]);
      ...
   LOG {
    PE IN (END,:)[0];
   } nearest_neighbor;
    RULE {...}myRule;
    myArray(nearest_neighbor);
   }
}
```

Figure 4.25.: Example of a CONNECTION-RULE section.

The second part of a connection rule, introduced by the keyword LOG, is a list of output ports of PEs in the array that will be connected to output ports of the array. Each statement of this part declares a region of the array, and an index, such that every PE in the region has the indexed output port *logged*. The code depicted in figure 4.25 designates that all PEs in the last array row have their output port 0 connected to output ports of the array.

After describing parts 1 and 2, there may remain PEs in the array whose output ports have not been connected; they are ports that are intentionally not used. CGADL does not allow unused ports to remain unconnected; they need to be explicitly terminated. The third part of a connection rule is a list, like the one in the LOG part, of output ports (of PEs) that have not been connected. This part starts with the keyword VOID to indicate that those ports are explicitly terminated. The VOID part allows to discern between an intentionally terminated communication line and a design error, a forgotten connection between two points.

### The binding section

The last step in the declaration of architecture is the binding section, where declared arrays and connection rules are assigned to each other. To apply one connection rule to one array, the identifier of the array is used followed by identifier of the connection rule (in parenthesis). For example, the declarations

```
myArray(myRule);
arrayA(myRule);
arrayB(anotherRule);
```

assign the rule named `myRule` to the arrays `myArray` and `arrayA`; and the `anotherRule` to the `arrayB`.

Connection rules and arrays can be combined arbitrarily if the following conditions hold:

- Each position in the array is declared in exactly one of the regions defined in the connection rule. This guarantees that all input ports, at all PEs, will have a connection.
- The number of input ports of each PE in the array must be equal to the number of connections that is declared in the connection rule of the region that contains this PE.
- Every connection declared in the connection rule must start at the output port of a PE, or at an input port of the array (INPORT), and must finish at the input port of a PE, at an output port of the array (LOG), or at a dead-end termination (VOID).

Several concurrent bindings between arrays and connection rules may exist; each one of them represent an architecture template, or instance (if parameter values are assigned).

## 4.3. Estimation of hardware costs

The success of a description language depends on the support of software tools, such as compilers, simulators, verifier, and estimators. It should, therefore, be possible to implement software tools that 'understand' the designer's description and transform, or use it to ease the design task. So, for example, simulators can be written that proof the functionality of the described architecture, or softwares to formally verify if the design meet some requirement. When no tools are available that work with a certain language, the designer is obliged to rewrite the description into another language to, for example, simulate, verify, or synthesize the design.

In this section, a method is proposed to estimate the hardware complexity of processing elements and architecture templates described in CGADL. Hardware complexity is a measure for the number of basic logic gates, such as inverters and logic OR, necessary to implement an electronic circuit, and therefore it is directly related to the implementation area of this circuit. The proposed method allows to evaluate the following aspects:

**Area composition of processing elements and architecture instances** The area composition indicates how the implementation area is distributed among the components of the architecture or PEs. This distribution is dependent on the values attributed to the parameters of the template. Consider , for example, processing elements differing only in the width of their datapath. As the datapath width increases, functional units, registers, and multiplexers build up a larger portion of the circuit area in comparison to context memories and finite state machines, whose size do not increase with this parameter.

**Scalability of the implementation area** The scalability analysis indicates how the area of components scale as a function of parameters of the description. The scalability analysis can indicate, for example, how the complexity (and thus area) of multiplexers and output ports is affected by parameters such as the number of registers.

**Comparison between architecture instances** The hardware complexity analysis allows to compare indirectly the implementation area of architecture (or PE) instances with distinct parameterizations. This comparison is important for the design space exploration, as it indicates which parameter configuration leads to smaller hardware complexity, and thus requires less implementation area.

The estimation method proposed here increases the productivity of the development phase and anticipates the design space exploration because:

- it does not require synthesis of the architecture model. All the analysis is based exclusively on the CGADL description of the architecture.
- it is technology independent. The estimated complexity of a circuit is presented as the number of gates, each of which equivalent to an inverter-gate, necessary to implement the circuit. It is, therefore, independent of the fabrication technology that will be used to implement the array.
- it must run only once for all the architecture instances. The estimation method outputs a set of estimation functions, which have a similar parameterization as the architecture model. To evaluate new instances, it is not necessary to run the estimation method again. Instead, the same set of functions is solved considering new parameter values.

### 4.3.1. Estimation flow

To implement the estimation method, a circuit model of each native element of the CGADL language was created, composing a library as depicted in figure 4.26. From each circuit model a function was derived, which estimates the hardware complexity as a function of the circuit parameters, such as width of the datapath and number of input ports. The estimation method itself consists in a composition analysis which receives as input a CGADL description file and the library of circuit models with their respective estimation functions. Using the CGADL description file, the estimation method analyzes the list of elements in the declaration section (see section 4.2.2) of each PE to compose a set of estimation functions. In the next two sections, the library of analytic models and the composition analysis from a CGADL description are discussed in detail.

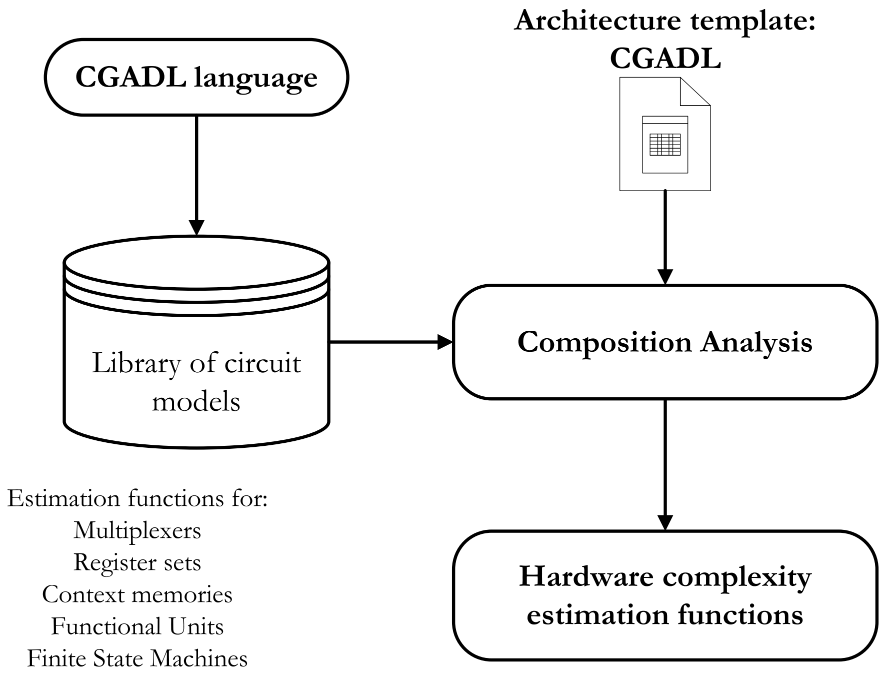

CGADL language
Architecture template: CGADL
Library of circuit models
Composition Analysis
Estimation functions for:
Multiplexers
Register sets
Context memories
Functional Units
Finite State Machines
Hardware complexity estimation functions

Figure 4.26.: Workflow for the hardware complexity analysis method.

## 4.3.2. Library of circuit models

By using the methodology presented in [82], an electronic circuit model was elaborated for each of the following elements: multiplexer, register set, output ports, FSM, FU, and context memory. Each circuit model is an hierarchical composition of smaller circuits, such that its complexity may be calculated by summing up the complexity of its components. The hardware costs for each circuit are expressed in *inverter gates equivalence* considering the costs of basic gates as in Table 4.2.

Consider, for example, the model of an $r$-registers bank with $r$ $n$-bits registers, as depicted in figure 4.27. This bank receives a $n$-bits signal as data input and an $a$-bits signal as address input. At the clock rising edge, the address is decoded and the value present in the input is stored in the addressed register. The number of necessary address signals is normally equal to $\lceil \log_2(r + 1)\rceil$, because one register address is used to indicate that there will be no storage. Each cell in the register set model comprises a D flip-flop, a 2-input 1-bit multiplexer, and an inverter as additional

Table 4.2.: Hardware costs of basic logic gates.

| | Costs (1bit) |
|---|---|
| $C_{\text{inv}}$ | 1 |
| $C_{\text{nand}}$, $C_{\text{nor}}$ | 2 |
| $C_{\text{and}}$, $C_{\text{or}}$ | 2 |
| $C_{\text{xor}}$, $C_{\text{xnor}}$ | 4 |
| $C_{\text{mux}}$ | 3 |
| $C_{\text{ff}}$ | 11 |

glue logic to reset and load enable circuits. These are necessary for the initialization of the register bank and its addressing control.

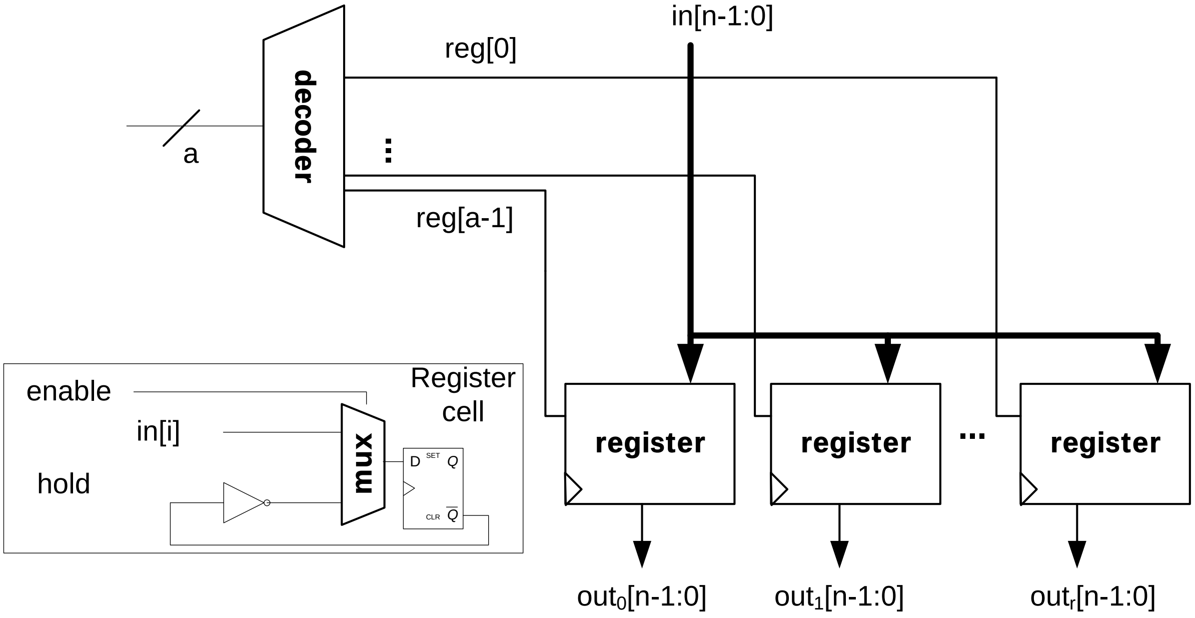


Figure 4.27.: Circuit model for hardware cost estimation of a register set.

The complexity of the register bank can be calculated by summing the complexity cost of $r$ registers and one $a$-address decoder. The complexity of $r$ registers is given by

$$C_{\text{regSet}}(r, n) = r \times C_{\text{reg}}(n) = r \times n \times (C_{\text{ff}} + C_{\text{mux}} + C_{\text{inv}})$$

and the costs for an $a$-address decoder[82] is given by

$$\begin{aligned} C_{\text{regDec}}(1) &= C_{\text{inv}} \\ C_{\text{regDec}}(a) &= C_{\text{regDec}}(\left\lceil \frac{a}{2} \right\rceil) + C_{\text{regDec}}(\left\lfloor \frac{a}{2} \right\rfloor) + 2^a C_{\text{and}} \end{aligned}$$

Composing these elements, the total cost for the register bank is

$$C_{\text{REG}}(r,n) = n \times r \times (C_{\text{ff}} + C_{\text{mux}} + C_{\text{inv}}) + C_{\text{regDec}}(\lceil \log_2(r+1) \rceil)$$

The estimation functions for the hardware complexity of each basic CGADL element are presented in Table 4.3. The assembly of these estimation functions and their background circuit models are detailed in Appendix A.3. The library of estimation functions needs to be assembled only once. However, modifications are possible, for example, to adequate the model of the FU to other instruction sets, or to adjust one of the circuit models to better reflect synthesis results.

Table 4.3.: Hardware costs of CGADL basic elements relative to an inverter gate.
$r$ is the number of registers.
$n$ is the width of the datapath (number of bits in the data word).
$c$ is the number of contexts.
$s$ and $b$ are the number of states and branches per state in the FSM, respectively.
$f$ is the number of decision flags in the FSM.
Detailed explanation for each equation term can be found in Appendix A.3.

| CGADL element | Hardware complexity estimation function |
|---|---|
| Register set | $C_{\text{REG}}(r,n) = r \times C_{\text{reg}}(n) + C_{\text{regDec}}(\lceil \log_2(r+1) \rceil)$ |
| Multiplexers<br>Output ports | $C_{\text{mux}^i}(n) = 3n(i-1)$ |
| FSM | $C_{\text{FSM}}(s,c,b,f) = C\text{stateMem}(s,b,c) + C_{\text{mux}^f}(1) + C_{\text{mux}^b}(\log_2 c) + C_{\text{ff}}(\log_2 c)$ |
| Context memory | $C_{\text{CM}}(a_w, a_r, c, n) = C_{\text{cmWrDec}}(a_w) + C_{\text{latchMem}}(c,n) + C_{\text{mux}^{a_r}}(n)$ |
| FU | $C_{\text{FU}}(n) = C_{\text{AddSub}}(n) + C_{\text{mult}}(\frac{n}{2}) + C_{\text{and}}(n) + C_{\text{or}}(n) +$<br>$C_{\text{xor}}(n) + C_{\text{not}}(n) + C_{\text{shift}}(n, \log_2 n) + C_{\text{sel}}(n) +$<br>$C_{\text{comp}}(n) + C_{\text{and}} + C_{\text{or}} + C_{\text{xor}} + C_{\text{inv}} + C_{\text{mux}^{16}}(1) + C_{\text{mux}^{12}}(n)$ |

### 4.3.3. Composition analysis

The second phase of the hardware complexity estimation is the analysis of a CGADL description. The analysis evaluates the composition of each PE in the architecture description and determines the hardware complexity estimation functions for the set of output ports, the composition of the FU datapath, register sets, PE datapath, PE control path, and the PE general composition. As an example, consider the PE *Bianca* described previously in the figure 4.16. The PE *Bianca* is composed of:

- Two $n$-bits multiplexers for the input of data in the functional unit. Each multiplexer should be able to switch the content of $r_d$ registers plus 4 input ports. The number of input ports of the multiplexer is extracted from the connection section (not shown).

- Two 1-bit multiplexers for the input of flags in the functional unit. Each multiplexer should be able to switch the content of $r_f$ registers plus 4 input ports. The number of input ports of the multiplexer is extracted from the connection section (not shown).

- One $r_d$-registers, $n$-bits register set, used for data storage. $r_d$ is given by the parameter `REGSize`.

- One $r_f$-registers, 1-bit register set, used for flag storage. $r_f$ is given by the parameter `REGSize`.

- Four n-bits multiplexers for the output of data from the PE. Each multiplexer should be able to switch the content of $r_d$ registers, 3 other input ports, and the direct output from the FU.

- Four 1-bit multiplexers for the output of flag signals from the PE. Each multiplexer should be able to switch the content of $r_f$ registers and 3 input ports. Together with the last item, it make a total of 8 output ports.

- One $s$-state finite state machine, with 2-branch possibilities and $\log_2 c$-bits output, where $c$ is the number of contexts in the context memory. $s$ is equivalent to the description parameter `FSMSize`, and $c$ is equivalent to the parameter `CMSize`. Additionally, the FSM becomes $(r_f + 5)$ 1-bit flag signals as input, which come from the $r_f$ flag registers, 4 flag input ports and one flag directly from the functional unit. The later information is extracted from the connection section (not shown).

- One context memory with $c$ contexts.

- One $n$-bits functional unit composed of the following operations: $n$-bits addition/subtraction, $\frac{n}{2}$-bits multiplication, $n$-bits logic operations (AND, OR, XOR, NOT), shift, 2 $n$-bits word selection, a $n$-bits comparator, and a 1-bit flag operation unit(with AND, OR, XOR, NOT).

The complexity cost of a PE may be resumed as:

$$\begin{aligned} C_{\text{PE}}(r_d, r_f, s, c, n) =& C_{\text{REG}}(r_d, n) + C_{\text{REG}}(r_f, 1) + \\ & 2C_{\text{mux}^{r_d+4}}(n) + 2C_{\text{mux}^{r_f+4}}(1) + 4C_{\text{mux}^{r_d+4}}(n) + 4C_{\text{mux}^{r_f+3}}(1) + \\ & C_{\text{FSM}}(s, c, 2, (r_f + 5)) + C_{\text{CM}}(a_w, a_r, c, n) + C_{\text{FU}}(n) \end{aligned} \tag{4.1}$$

where $r_d$, $r_f$ are given by the parameter `REGSize`, $s$ is given by the parameter `FSMSize`, $c$ is given by the parameter `CMSize`, and $n$ is the width of the datapath. The last parameter, $n$, is not explicitly declared in the CGADL description, and must be provided by the designer when evaluating the estimation functions.

### 4.3.4. The hardware complexity estimation tool

Within this research work a software tool was developed that implements the hardware complexity estimation method proposed in the previous section. This tool has two purposes in this work: first, it automates the tasks of the proposed estimtion methodology and provides an automatic, software-based production of results; second, it demonstrates that tools can be developed which directly use the proposed CGADL language. This section discusses the implementation details for this tool.

The estimation tool is a command-line based program, written in the programming language `Java` [45], which follows the flow depicted in Figure 4.28. The input for the tools is a CGADL description of the architecture template. This description is parsed inside the tool and transformed into an intermediate format, as explained in Section 4.3.4. The intermediate format contains (among other information) a list of all PEs in the CGADL description. Each entry in this list is a record of all components within the PE and its interconnections.

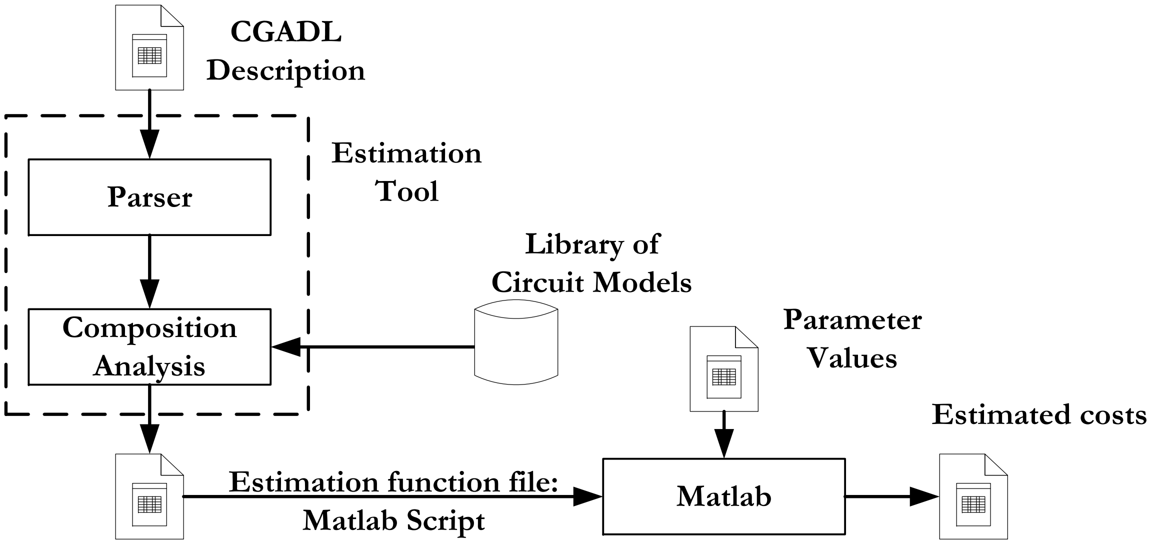


Figure 4.28.: Workflow of the hardware cost estimation software tool.

After the CGADL description is parsed, the estimation tool analyses which components are present in each PE. This phase is called *composition analysis*, and is discussed in Section 4.3.4. The composition analysis produces an estimation function by summing the costs of individual components within the PE. These costs are obtained from the library of circuit models (see 4.3.2) and consist of a pre-written estimation function dependent on the component type. The composition analysis phase outputs a Matlab [41] script file containing the cost estimation function.

The cost estimation function Matlab script can be evaluated inside the Matlab environment for a set of arbitrary parameters values. This evaluation outputs hardware complexity costs, expressed in gate equivalence, for the PE and its FU. This step is discussed in Section 4.3.4.

## The CGADL parser

The CGADL parser reads a CGADL description and generates the intermediary format depicted in Figure 4.29. The intermediary format is a data structure composed of several lists: a list of the template parameters, a list of PE types (their composition and internal connections), a list of concatenations, a list of described arrays, and a list for connection rules for these arrays. For the generation of cost estimation functions, only the list of template parameters and the list of PE types is used.

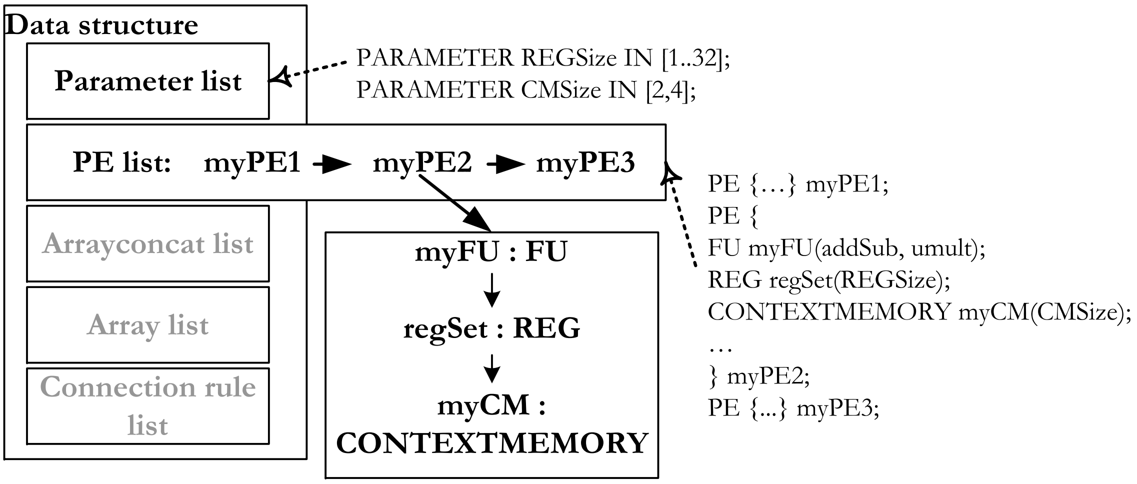


Figure 4.29.: Intermediate format generated by the CGADL parser.

The parameter list contains all the parameters declared in the CGADL description (see Section 4.2.1) and the possible values they may assume. These parameters are used when instantiating components of the PEs, and they have a direct correspondence to the estimation function parameters.

The list of PE types contains an entry for each PE section in the CGADL description (see Section 4.2.2). Each PE type entry contains a record of all basic CGADL elements that are used inside the PE. This record will be used by the composition analysis to build the total hardware cost estimation function.

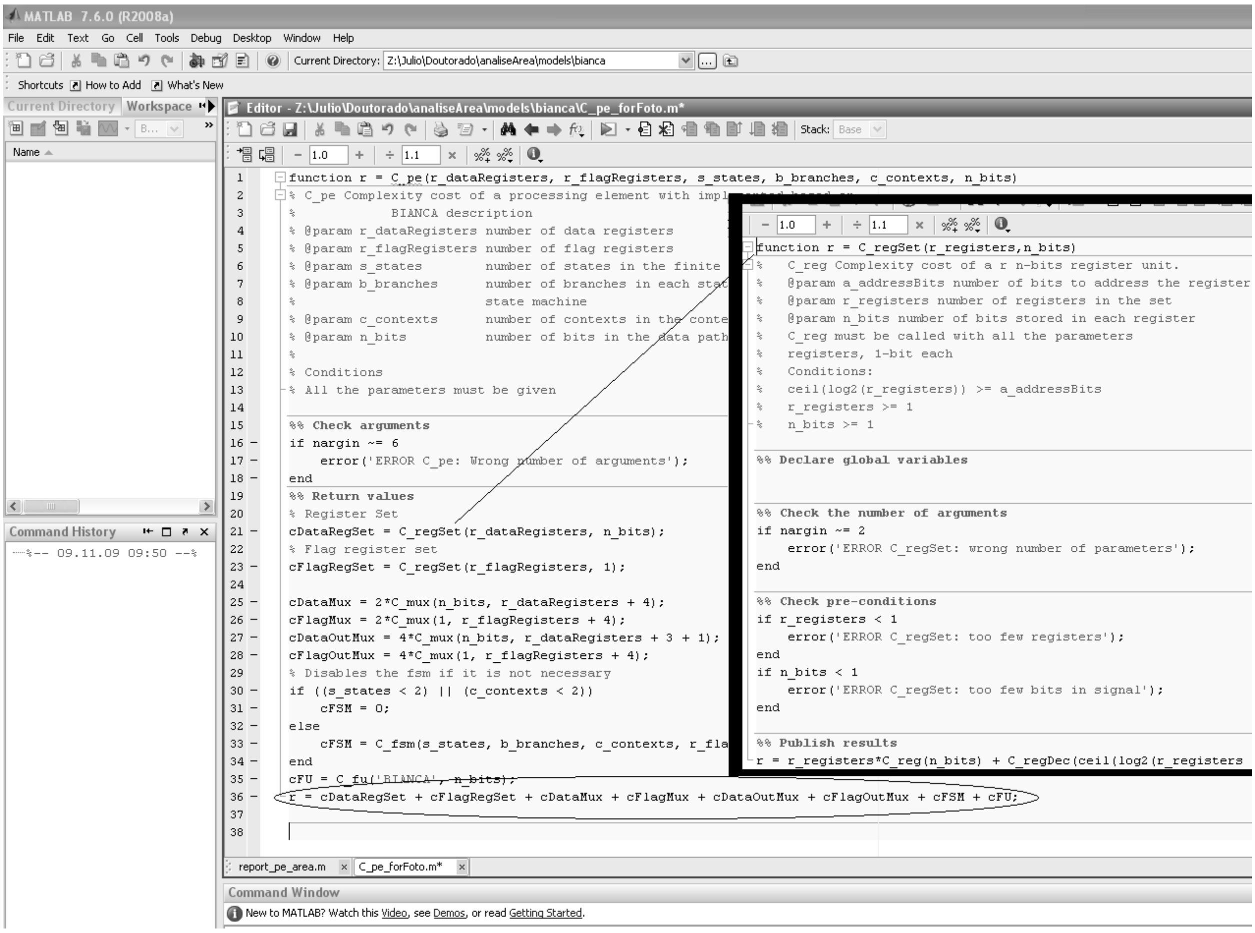
The parser also makes available other information, which is present in the ARRAY section, such as a list of blocks (concatenated arrays), a list of possible architecture arrays, and a list of connection rules. However, this information is not useful for the hardware cost estimation tool and will not be discussed here.

### Composition Analysis Module

The composition analysis module receives two inputs: the intermediate format produced by the CGADL parser and cost estimation functions corresponding to basic element of CGADL such as MUX, REG, etc. Estimation functions for each basic element are part of the library of circuit models; they were obtained by using the estimation method explained in Section 4.3 and Appendix A.3, and implemented as Matlab scripts. One example can be seen in the detail of Figure 4.30: the estimation function file for the register set (`RegSet`), discussed in Section 4.3.2.

During the composition analysis, the following procedure is made for each PE present in the intermediate format (list of PE types). First, the tool lists each one of the internal components of the PE and their quantities. Second, the tool retrieves from the library of circuit models estimation function files for the listed components. These files are copied to a separate directory. Third, the tool produces a Matlab script that evaluates the estimation function files with a correct parameterization and sums up their evaluations to build up the cost of a PE.

The main file, depicted in Figure 4.30, contains an example of the file generated for the PE


MATLAB 7.6.0 (R2008a)
File Edit Text Go Cell Tools Debug Desktop Window Help
Current Directory: Z:\Julio\Doutorado\analiseArea\models\bianca
Shortcuts How to Add What's New
Current Directory Workspace
Name
Command History
Editor - Z:\Julio\Doutorado\analiseArea\models\bianca\C_pe_forFoto.m*
Stack: Base
report_pe_area.m
C_pe_forFoto.m*
Command Window

```
function r = C_pe(r_dataRegisters, r_flagRegisters, s_states, b_branches, c_contexts, n_bits)
% C_pe Complexity cost of a processing element with impl
%             BIANCA description
% @param r_dataRegisters number of data registers
% @param r_flagRegisters number of flag registers
% @param s_states        number of states in the finite
% @param b_branches      number of branches in each stat
%                        state machine
% @param c_contexts      number of contexts in the conte
% @param n_bits          number of bits in the data path
%
% Conditions
% All the parameters must be given

%% Check arguments
if nargin ~= 6
    error('ERROR C_pe: Wrong number of arguments');
end
%% Return values
% Register Set
cDataRegSet = C_regSet(r_dataRegisters, n_bits);
% Flag register set
cFlagRegSet = C_regSet(r_flagRegisters, 1);

cDataMux = 2*C_mux(n_bits, r_dataRegisters + 4);
cFlagMux = 2*C_mux(1, r_flagRegisters + 4);
cDataOutMux = 4*C_mux(n_bits, r_dataRegisters + 3 + 1);
cFlagOutMux = 4*C_mux(1, r_flagRegisters + 4);
% Disables the fsm if it is not necessary
if ((s_states < 2) || (c_contexts < 2))
    cFSM = 0;
else
    cFSM = C_fsm(s_states, b_branches, c_contexts, r_fla
end
cFU = C_fu('BIANCA', n_bits);
r = cDataRegSet + cFlagRegSet + cDataMux + cFlagMux + cDataOutMux + cFlagOutMux + cFSM + cFU;

```

```
function r = C_regSet(r_registers,n_bits)
%   C_reg Complexity cost of a r n-bits register unit.
%   @param a_addressBits number of bits to address the register
%   @param r_registers number of registers in the set
%   @param n_bits number of bits stored in each register
%   C_reg must be called with all the parameters
%   registers, 1-bit each
%   Conditions:
%   ceil(log2(r_registers)) >= a_addressBits
%   r_registers >= 1
%   n_bits >= 1

%% Declare global variables


%% Check the number of arguments
if nargin ~= 2
    error('ERROR C_regSet: wrong number of parameters');
end

%% Check pre-conditions
if r_registers < 1
    error('ERROR C_regSet: too few registers');
end
if n_bits < 1
    error('ERROR C_regSet: too few bits in signal');
end

%% Publish results
r = r_registers*C_reg(n_bits) + C_regDec(ceil(log2(r_registers
```

New to MATLAB? Watch this Video, see Demos, or read Getting Started.

Figure 4.30.: Matlab scripts for hardware complexity estimation. The main figure shows the file produced by the estimation tool. In detail, a pre-written estimation function (C_RegSet) from the library of circuit models. Compare with value in table 4.3.

*Bianca*, discussed in Section 4.2.2, Figure 4.16. In this example, the costs of each individual component within the PE *Bianca* is evaluated by calling the pre-written functions `C_regSet`, and `C_mux`, and storing their results in intermediary values. Finally, the final estimation cost of the PE *Bianca* is evaluated as a sum of the costs of all its components.

## Estimation function evaluation

The hardware cost estimation function produced by the composition analysis module is file containing a Matlab function (see Figure 4.30). In order to obtain the estimate hardware complexity of a particular instance, it is necessary to invoke this file in the Matlab environment, passing to the file a set of parameter values corresponding to that of the instance to be evaluated.

# 5. Design of Custom Instructions for Coarse Grained Architectures

This chapter proposes methods, techniques, and algorithms to design and integrate custom instructions in coarse grained architectures. *Custom instructions* are designed to meet specific demands or to provide advantage in the execution of a specific application or application group (see discussion in Section 2.1.5). Therefore, custom instructions are the way by which this work specializes the architecture.

There are several ways to design custom instructions. For example, the designer can rewrite the datapath of an operation to reduce its implementation area execution delay [87] [124]. Another option is to use another gate technology in the instruction datapath; for example, Bouwens et al. implemented instructions of the ADRES architecture using a clock-gating library to reduce power consumption [14].

In this work, custom instructions are designed based on groups of operations found in the set of applications. It can be often observed that some groups of operations in the DFG of one application follow the same execution pattern. If a custom instruction is available in the architecture that implements this pattern, it can be used to map and execute any of those groups of operation. Also often, some clusters of operations are found in DFGs of different applications that follow the same execution pattern. In this case, different applications may reuse the same custom instruction. Therefore, the more likely it is to find groups of operations with the same execution pattern, the better this group is to guide the design of a custom instruction.

The design of custom instructions, in this work, specializes and improves the architecture design according to the following principle:

> *A custom instruction executes a group of operations that, otherwise, would only be accomplished by using several individual PEs. PEs embedded with such custom instruction can be used to replace groups of non-custom PEs, decreasing the total number of necessary PEs in the array.*

This principle can be explained by means of the example depicted in Figure 5.1.

Figure 5.1 depicts the DFG of the trilinear interpolation filter kernel, used in the resampling phase of the ray casting algorithm [29]. The ray casting is a real-life application for the visualization of 3D scientific and medical data. This application was mapped in a coarse grained array as a pipeline: each operation was assigned to a certain PE in the array, respecting constraints and execution order. That required a total of 28 PEs and 12 pipeline stages, so that each PE could process one operation per clock cycle.

A closer observation of the trilinear interpolation DFG reveals a very regular structure, where some subgraphs (corresponding to similar clusters of operations) appear very often. Three examples of such clusters – $IP_a$, $IP_b$, and $IP_c$ – are indicated in Figure 5.1. Custom instructions

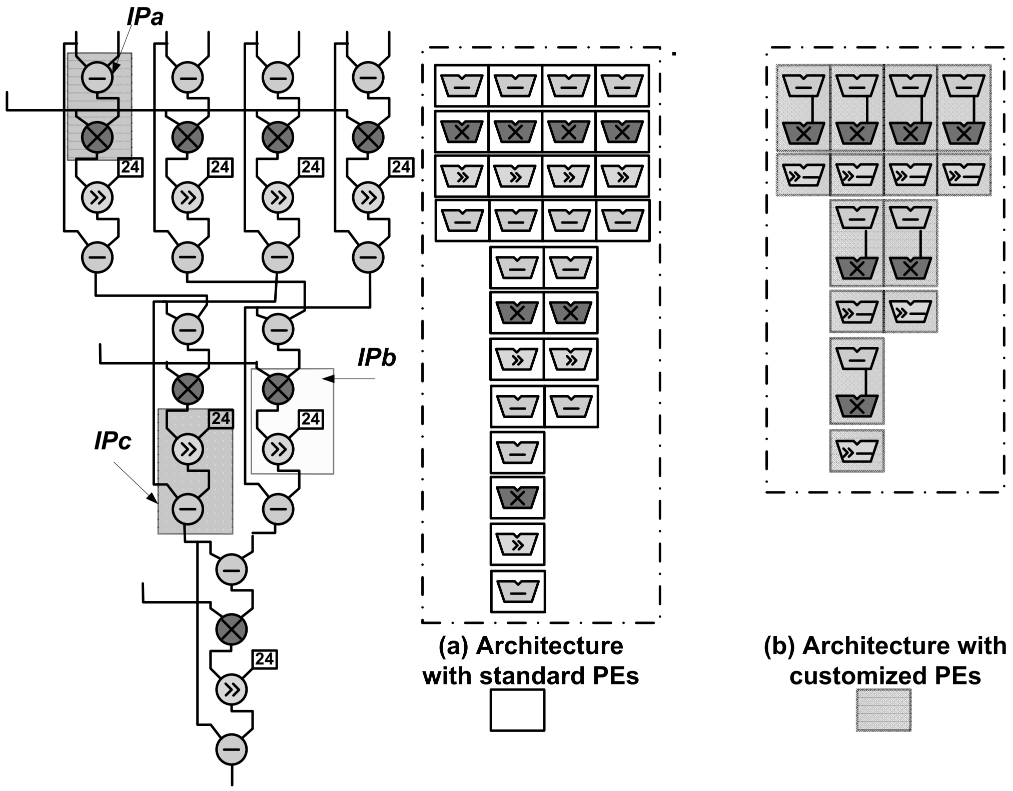

IPa
IPb
IPc
24
(a) Architecture with standard PEs
(b) Architecture with customized PEs

Figure 5.1.: Trilinear Interpolation mapping on architectures with (a) standard and (b) customized PEs.

were designed to execute the clusters $IP_a$ and $IP_c$ as a single operation (and one clock cycle). These custom instructions were integrated in the funcional units of the coarse grained array used before. Then, the application was mapped as a pipeline again, but now using the specialized array: groups of operations with the same execution pattern as $IP_a$, $IP_b$, and $IP_c$ could be mapped to one single PE, whereas in the non-custom array they required two PEs. The mapping of the trilinear interpolation in this specialized architecture (Figure 5.1(b)) required 14 PEs, and 6 pipeline stages.

Table 5.1.: Impact of instruction specialization

| | #PEs | PE Area | Total | Dynamic |
|---|---|---|---|---|
| | | $(\mu m^2)$ | Area | Power |
| | | $\times 10^3$ | $(mm^2)$ | $(mW)$ |
| Standard PEs | 28 | 96.42 | 2.70 | 177.64 |
| Customized PEs | 14 | 97.14 | 1.36 | 91.39 |

Table 5.1 depicts the costs for area and power of the two architectures after synthesis in a $130nm$ technology. The PEs of the specialized architecture are more complex because they incorporate two new instructions. Therefore, each PE requires more area. However, the overall necessary area and consumed dynamic power are 49% and 48% smaller, respectively. That is explained by

the fact that only half the amount of PEs is necessary. For these cases, we also have significant performance improvement. While the throughput is still the same, the number of necessary pipeline stages (latency) drops from 12 to 6.

## 5.1. Instruction Pattern Identification and Custom Instruction Composition

Basically, the design of a custom instruction consists of transforming parts of the application's dataflow into the description of a circuit datapath. There are several ways to do this, for example, one can use high level synthesis techniques to generate a scheduled description at register-transfer level and a controler circuit [79] [112] [35] [20]. In this work, the design of custom instructions generates a single cycle circuit having all the operators chained. That allows designers to easily estimate area and delay for the new instruction, and to merge two or more custom instructions into one datapath (further detailed in Section 5.1.3).

When using operation chaining to design the datapath of custom instructions, nodes and edges of the DFG are in direct analogy to elements in the datapath: nodes can be considered as operation modules and edges as interconnection lines. The example in Figure 5.2 clarifies this. Nodes in a DFG that represent input operations can be transformed into input ports of the datapath; and, nodes representing output operations can be transformed into output ports. Nodes representing atomic operations can be mapped to operation modules. Edges between two nodes can be seen as communication lines used to transfer data from one module (or port) to other.

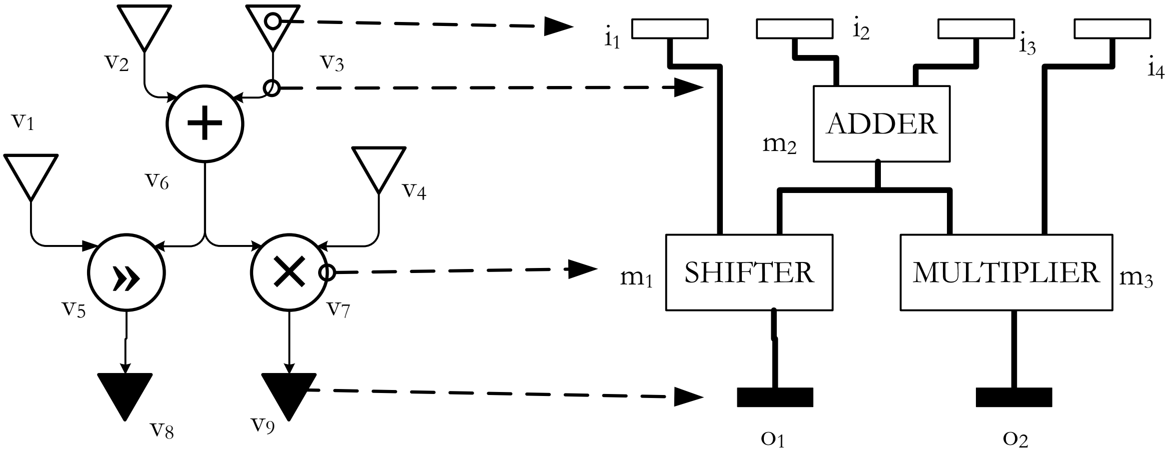


Figure 5.2.: DFGs can be seen as the description of a datapath: input operations ($\{v_1, v_2, v_3, v_4\}$) are mapped to input ports ($\{i_1, i_2, i_3, i_4\}$); output operations ($\{v_8, v_9\}$) correspond to output ports ($\{o_1, o_2\}$); atomic operations ($\{v_5, v_6, v_7\}$) represent operation modules ($\{m_1, m_2, m_3\}$); and edges correspond to data transfer lines between modules and/or ports.

This idea is used to rationalize and guide the design of custom instructions in this work.

### 5.1.1. Extraction of operation clusters

Consider the data flow graph $G(V, E)$ of an application, as defined in Section 2.2.1.

**Definition 5.1** ***Operation cluster*** *is a subgraph $S \subset G(V, E)$ induced by any non-empty subset of atomic operations $V_a \subset V$.*

In the DFG of an application, any subgraph induced by a set of atomic operations is an operation cluster, and may be used to design a custom instruction for this application.

However, not every operation cluster leads to a feasible custom instruction. The design of custom instructions must usually meet some constraints, such as a maximal implementation area, a maximal execution delay, or a maximal number of input ports. Some operation clusters may lead to instructions that violate these constraints. These clusters should not be considered to design custom instructions. The objective of the extraction phase is to list all operation clusters that can be used to design feasible, constraint-conform custom instructions.

The methodology presented here considers the following constraints for the design of custom instructions:

- A feasible instruction must be an *atomic* chaining of operations. *Atomic* means that after an instruction starts its execution, it will continue without interference until all the output data is produced (see definition in Section 2.2.1).

- A feasible instruction must use at most a predefined number of input ports; and, it must use at most a predefined number of output ports.

- A feasible instruction must have at most a predefined hardware implementation cost, usually measured in silicon area units or gate equivalence.

- And a feasible instruction must execute in no more than a predefined delay.

A property of the operation cluster graph may be formally described for each one of these constraints.

#### Atomic instruction – Convexity

**Definition 5.2** *An operation cluster $S$ is said to be* **convex** *if for all $v_i, v_j \in S$, all paths between $v_i$ and $v_j$ are within $S$.*

In this work, the datapath of a custom instruction is a chain of operations. Convexity guarantees that the data to be processed remains within this chain throughout the execution. If the cluster is not convex, its corresponding datapath produce data that must be processed outside the datapath, and thus the execution must be interrupted (non-atomic). Therefore, convexity is a necessary property for the design of atomic instructions. Convexity also ensures that all input values are available to the FU when it starts executing one instruction.

Examples of convex and non-convex clusters are depicted in Figure 5.3. $S_1$ is a convex operation cluster because there is no path between $v_2$ and $v_1$ that uses a node outside $S_1$. In contrast, $S_2$ is a non-convex operation cluster because the result of operation $v_4$ has to be processed by $v_3$, which is not in $S_2$, and returned to $v_2$ within this cluster.

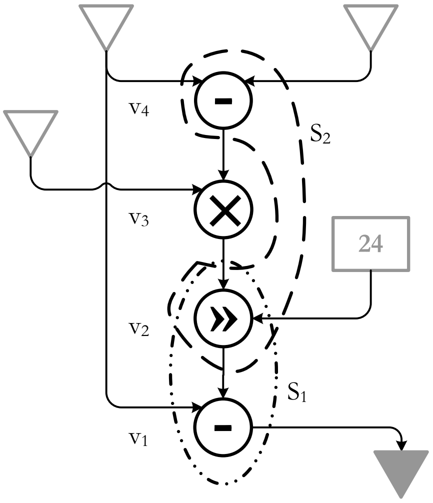


Figure 5.3.: Operation clusters in a DFG. $S_1$ is the subgraph induced by $\{v_1, v_2\}$. It is convex, consumes three input values ($\mathrm{IN}(S_1) = 3$) and produces one output value ($\mathrm{OUT}(S_1) = 1$). Its estimated implementation cost is $C(S_1) = C(v_1) + C(v_2)$, and its estimated execution delay is $\delta(S_1) = \delta(v_1) + \delta(v_2)$. $S_2$ is a non-convex cluster induced by $\{v_2, v_4\}$.

### Input and output ports – number of input and output values

Instructions must abide by constraints in the usage of input and output ports. Each input value consumed by an operation cluster corresponds to an input port. Each output value produced by an operation cluster corresponds to an output port. Formally, one can write:

**Definition 5.3** *For a given subgraph $S \subset G(V, E)$, the* ***input number*** *of $S$, denoted by $\mathrm{IN}(S)$, is the number of predecessor vertices of those edges that enter the operation cluster $S$ from any $v \in V$. Similarly, the* ***output number*** *of $S$, denoted by $\mathrm{OUT}(S)$, is the number of predecessor vertices $v \in S$ of edges leaving the operation cluster $S$.*

$\mathrm{IN}(S)$ matches the number of input values used by $S$, and thus the number of input ports necessary to implement the corresponding custom instruction. $\mathrm{OUT}(S)$ corresponds to the number of output values produced by $S$, and thus the number of output ports of the corresponding custom instruction. For example, in Figure 5.3, $\mathrm{IN}(S_1) = 3$ and $\mathrm{OUT}(S_2) = 2$. During the extraction of feasible operation clusters, a maximal number of input data values $N_{\text{in}}$ and a maximal number of output data values $N_{\text{out}}$ are allowed. $N_{\text{in}}$ and $N_{\text{out}}$ are defined at design time.

### Hardware implementation cost – estimated cost of an operation cluster

To keep the implementation area of the FU small, designers usually define constraints for the hardware costs implied by a custom instruction. When the custom instruction is based on an operation cluster, an estimate for this overhead can be defined as follows:

**Definition 5.4** *Given a subgraph $S \subset G(V, E)$, the* ***estimated cost*** *$C(v)$* ***of the vertex*** *$v \in S$, is the cost of implementing the atomic operation $Op(v)$*[1] *in hardware. The* ***estimated cost*** *$C(S)$* ***of an***

[1] *Op* is the labeling function defined in Section 2.2.1, definition 2.2.

***operation cluster*** *$S$ is the estimated cost of implementing $S$ as a custom instruction. It is given by $C(S) = \sum_{v \in S} C(v)$.*

Typically, the costs $C(v)$ and $C(S)$ are expressed in silicon area units or gate equivalent. $v$ usually represents well known basic operations, such as addition, arithmetic shift, or multiplications, and their costs are known at design time or can be easily obtained from synthesis. In the CRC template, these operations correspond to hardware modules of a design library. Each module has well specified characteristics in terms of usage area, execution delay, and static power consumption. These costs can also be obtained by using the estimation method proposed in Section 4.3. During the extraction of operation clusters, an upper bound $C_{\text{max}}$ is adopted for the estimated cost of an operation cluster.

### Instruction execution delay – cluster execution delay

To keep the execution delay of the FU small, designers usually define constraints for the delay implied by the datapath of a custom instruction. When the custom instruction is based on an operation cluster, an estimate for this delay can be defined as follows:

**Definition 5.5** *The* ***execution delay*** *$\delta(v)$* ***of a vertex*** *$v \in S$ is the delay of the atomic operation $Op(v)$ when implemented in hardware. The* ***execution delay*** *$\delta(S)$* ***of an operation cluster*** *$S$ is the estimated delay for the the critical path of $S$, when $S$ is implemented as a custom instruction. This delay is $\delta(S) = \max_{p \subseteq S} \sum_{v \in p} \delta(v)$, $p$ being a path between an input and an output node.*

Typically, the delays $\delta(v)$ and $\delta(S)$ are expressed in nanoseconds. In Figure 5.3, $\delta(S_1) = \delta(v2) + \delta(v_1)$ and $\delta(S_2) = \max\{\delta(v_4), \delta(v_2)\}$. During the extraction of operation clusters, an upper bound $\delta_{\text{max}}$ is adopted for the execution delay of an operation cluster.

### Problem formulation

The extraction of operation clusters for the design of feasible custom instructions can be formulated as:

**Problem 1** *Given a graph $G(V, E)$, list all operation clusters $S$ that meet the following constraints:*

1. $\text{IN}(S) \leq N_{in}$ *and* $\text{OUT}(S) \leq N_{out}$
2. $C(S) \leq C_{max}$ *and* $\delta(S) \leq \delta_{max}$
3. *$S$ is convex*

An exhaustive approach to solve this problem is to list all possible operation clusters in $G(V, E)$ and then select those that meet the constraints. Assuming that $V'$ is the subset of all atomic operations of $V$, then there are $2^{|V'|}$ possible operation clusters to be listed. This approach may be computationally not feasible if the number of atomic operations in the DFG of an application is large.

Another approach is proposed here based on the work by Atasu [7]. Atasu proposes an algorithm that builds a binary search tree over the DFG of an application. Each node in the tree corresponds to

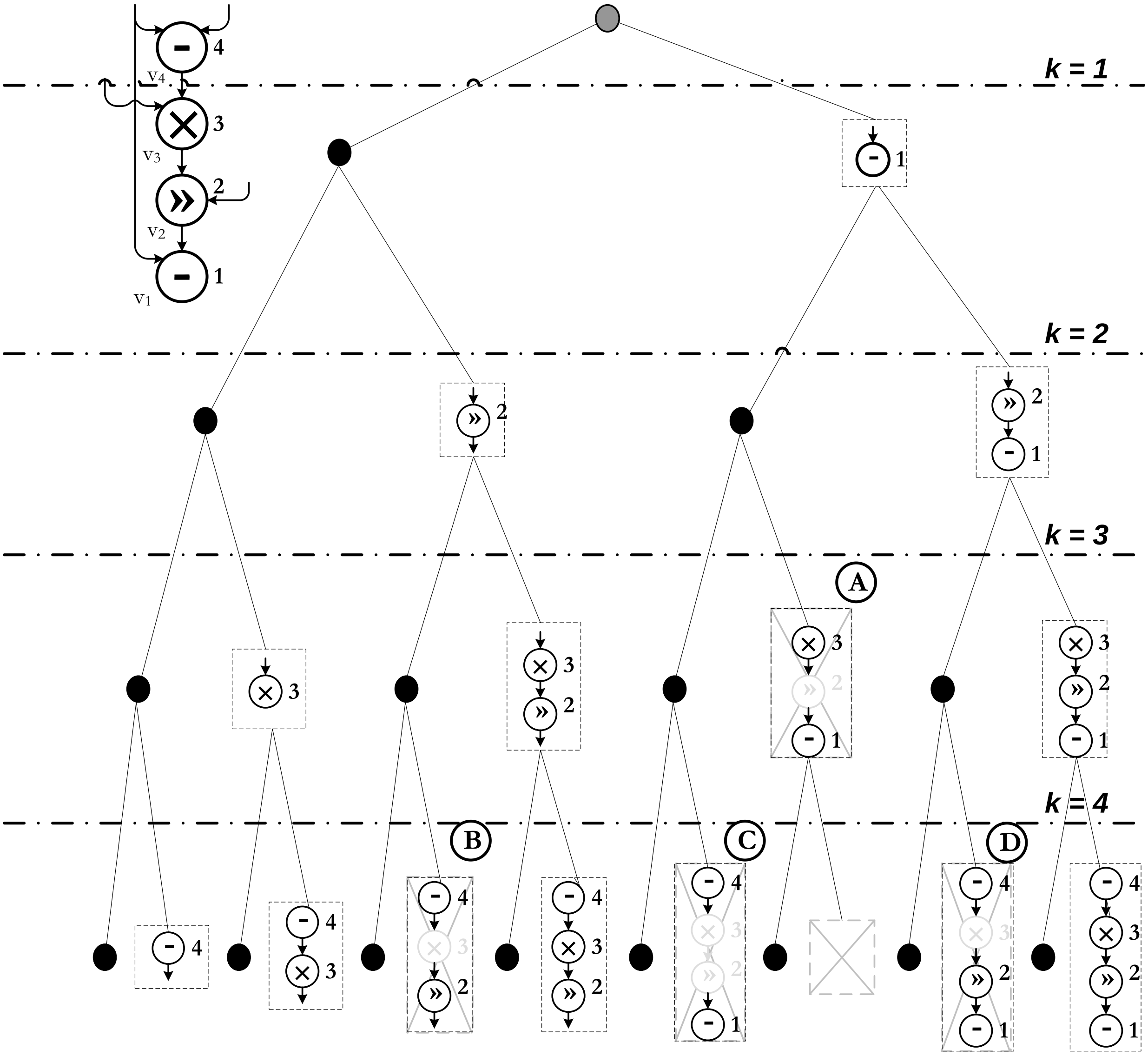


Figure 5.4.: Binary search tree for the extraction of feasible operation clusters of the depicted DFG. Constraints are $N_{\text{out}} = 1$, $N_{\text{in}} = 4$, $C_{\text{max}} = \infty$, and $\delta_{\text{max}} = \infty$. Each branching level $k$ considers the exclusion (left branch) or the inclusion (right branch) of the node numbered as $k$. Crossed boxes indicate operation clusters that violate some constraint: node Ⓐ produces more than 2 output values; and nodes Ⓐ, Ⓑ, Ⓒ, and Ⓓ are not convex.

a possible operation cluster. The search tree potentially spreads over all the search space, but during its construction, the algorithm detects and prunes branches that corresponds to unfeasible operation clusters. In this work, Atasu's algorithm is extended to consider the hardware implementation cost $C(S)$ and the execution delay $\delta(S)$. The base algorithm was kept the same. An example is illustrated in Figure 5.4.

The algorithm starts listing the atomic operations in $G(V, E)$ in a topological order. In a topological order, the operation $v_i$ always appears after the operation $v_j$, if $v_j$ is dependent on the datum produced by $v_i$. In Figure 5.4, the vertices of the example DFG are numbered in topological order. Based on this topological order, a binary search tree is built. The root node represents an empty operation cluster and at each branching level $k$ the node numbered as $k$ is considered. At each level, pairs of 1- and 0-branches represent the addition and non-addition, respectively, of the node to the operation cluster represented by the parent node. Following the example in Figure 5.4, after branching from the root node, all nodes at the left side of the tree represent operation clusters for which the operation 1 ($v_1$) is not present. Concurrently, the right side of the tree represents operation clusters for which the operation 1 ($v_1$) is present. The next branch considers then the inclusion or not of the operation 2 ($v_2$), and so on. Nodes of the search tree immediately following a 0-branch represent the same operation cluster as their parent node, and are not considered.

It can be shown that in some cases there is no need to build the branches under a certain node, and the corresponding search space can be pruned. There are four situations where the branches derived from a given operation cluster can be completely ignored:

1. *The number of output ports* $\mathrm{OUT}(S)$ *of a given operation cluster* $S$ *violates the constraint* $N_{out}$. Adding vertices that appear later in topological order cannot decrease the number of output ports. Therefore, all derived operation clusters will also violate the same constraint. In Figure 5.4, all branches under and including the node Ⓐ will produce 2 or more data values and can be ignored if $N_{\mathrm{out}} = 1$.

2. *The estimated cost* $C(S)$ *of a given operation cluster* $S$ *surpass the upper bound* $C_{max}$. Lower levels of the search tree implies that the derived operation clusters have equal or more operations than its parents. If $C(S)$ exceeds $C_{\mathrm{max}}$ at one node, there is no way to reduce this cost by adding new operations.

3. *The execution delay* $\delta(S)$ *of a given operation cluster* $S$ *surpass the upper bound* $\delta_{max}$. There is no way to decrease the critical path delay associated with an operation cluster. That is because adding new vertices in topological order can only increase or maintain the paths within the subgraph $S$.

4. *A given operation cluster* $S$ *is not convex.* If convexity is violated at a certain tree node, there is no way of regaining it by considering the insertion of vertices of $G(V, E)$ that appear later in the topological order. This is the case for nodes Ⓐ, Ⓑ, Ⓒ, and Ⓓ in Figure 5.4.

In Figure 5.4, all branches under and including the node A are not convex, and can be ignored.

In case the number of input ports $\mathrm{IN}(s)$ of a given operation cluster $S$ violates the constraint $N_{\mathrm{out}}$, the corresponding node is invalidated and will not be listed as a valid operation cluster. However, because nothing can be assumed about the operation clusters derived in lower levels, the search does not prune the tree and continues into further branches.

The complete search tree corresponds to all possible operation clusters that can be listed in the graph $G(V, E)$. The worst case complexity of this algorithm is therefore $2^{|V'|}$, where $V'$ is the subset of all atomic operations of $V$. Although still exponential, this algorithm significantly reduces the search in practical situations (see [7]).

Later in the development of this work, another algorithm was proposed by Bozini that lists feasible operation clusters in polynomial time with respect to the size of the graph [12]. This algorithm is based on properties of convex subgraphs and the concept of multiple-vertex dominators.

After the search is complete, a list is available containing all operation clusters of an application that meet the defined constraints. This phase is repeated for each application of the application set. The next section discusses how to group these operation clusters using their structural similarity and select the groups that will be used to compose custom instructions.

### 5.1.2. Instruction pattern selection

The set of operation clusters extracted in the previous phase is used as input for this selection. Designing one custom instruction for each different operation cluster may lead to a huge design and hardware overhead due to the large number of possible clusters. Often, a small subset of clusters is sufficient to *cover* the target applications. The question is then how to select a small set of operation clusters so that a large part of each target application can be mapped using custom instructions? One may look at how representative are the operation clusters in this set, and how large should be the set of generated custom instructions.

The representativeness of a given operation cluster is related to the chance of finding other clusters with the same structure. If two distinct operation clusters describe the same datapath, they correspond to the same custom instruction. This similarity can be described as follows.

**Definition 5.6** *Two graphs $G(V, E)$ and $G'(V', E')$ are called* ***isomorphic****, denoted $G \cong G'$, if there is a bijection $f : V \to V'$, that satisfies $(v_i, v_j) \in E \Leftrightarrow (f(v_i), f(v_j)) \in E'$, for all $v_i, v_j \in V$[43]. Two graphs are called* ***label isomorphic with respect to a labeling function*** *$L$, if they are isomorphic and $L(v_i) = L(f(v_i))$ for all $v_i \in V$.*

**Definition 5.7** *Two operation clusters $S$ and $S'$ have the same* ***instruction pattern*** *$IP$, if they are label isomorphic with respect to the labeling function Op, defined in Section 2.2.1.*

This work proposes the flow depicted in Figure 5.5 as an approach to select which custom instructions should be implemented. Initially, all operation clusters are partitioned into subsets, such that any two elements in a subset have the same instruction pattern. These subsets are called ***instruction pattern sets***. Each instruction pattern set is evaluated and ordered in a list according to the frequency their members can be found in the applications. The first instruction pattern in the list is selected to generate a custom instruction. It is then measured to which extent the target applications will benefit if the selected patterns are implemented as custom instructions. If the number of operations in the target application set that can be mapped into custom instructions is below a lower bound, another instruction pattern set is added to the selection list. Each one of these steps is discussed in more details in the following.

#### Partition of operation clusters into instruction pattern sets

Let $\Omega^a$ be the set of all feasible operation clusters from an application $a$ of the target application set $A$. Operation clusters with the same instruction pattern in $\Omega^a$ can be defined as follows:

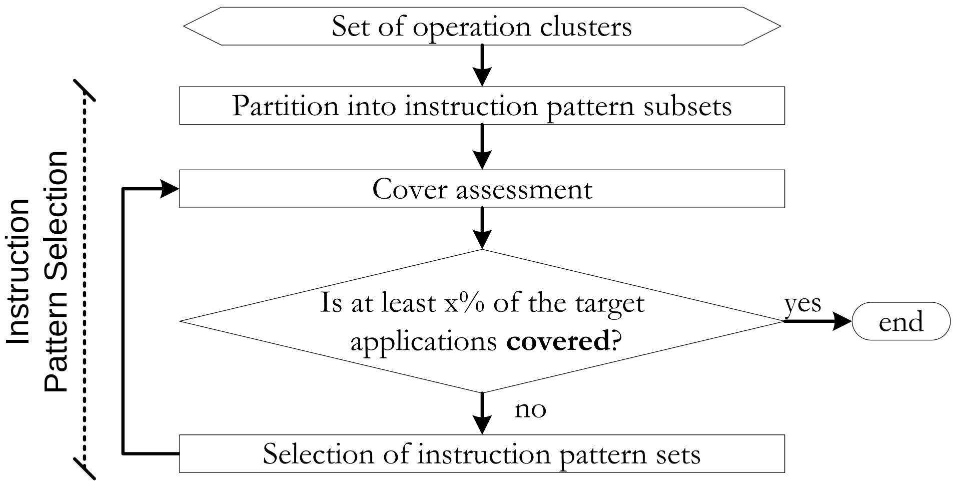

Set of operation clusters
Partition into instruction pattern subsets
Cover assessment
Is at least x% of the target applications **covered**?
yes
end
no
Selection of instruction pattern sets
Instruction Pattern Selection

Figure 5.5.: Activity flow proposed to select which instruction patterns should be implemented as custom instructions.

**Definition 5.8** *An **instruction pattern set**, denoted as* $\mathrm{set}(IP)$*, is a set of operation clusters such that for any two elements* $S_i, S_j \in \mathrm{set}(IP)$*,* $S_i$ *have the same instruction pattern as* $S_j$*.*

The procedure to partition a given set of operation clusters $\Omega$ into instruction pattern sets is depicted in lines 4 to 17 of Algorithm 1. One element $S'$ is removed from a given set of operation clusters $G$ and inserted into an empty instruction pattern set $\mathrm{set}(IP_i)$. Then, label isomorphism between $S'$ and every other element $S \in G, S \neq S'$ is checked. If the test is positive, $S$ is added to the $\mathrm{set}(IP_i)$ and eliminated from $G$. The procedure is repeated until there are no elements left in $G$.

Two potential problems may arise when applying this algorithm. First, the number of label isomorphism tests increases with the cardinality of $\Omega$. In the worst situation, no two graphs are isomorphic to each other, and the number of tests is $(|\Omega|^2 - |\Omega|)/2$. Second, the complexity of the graph isomorphism test increases with the number of vertices in the inspected graphs. For directed acyclic graphs, such as dataflow graphs, the complexity is isomorphism complete[9].

Usually, it is unneccessary to run an isomorphism test between all elements of $\Omega$. For two operation clusters $S_i(V_i, E_i)$ and $S_j(V_j, E_j)$ to be label isomorphic, the following conditions are necessary (but not sufficient):

- The number of vertices in both clusters is the same, that is $|V_{S_i}| = |V_{S_j}|$.
- The number of edges in both clusters is the same, that is $|E_{S_i}| = |E_{S_j}|$.
- There is a label matching function. That is, there exists a bijection $f : V_i \to V_j$ such that if $f(v) = u, v \in V_{S_i}, u \in V_{S_j}$, then $Op(v) = Op(u)$.

These conditions can be easily tested, so the isomorphism test can be performed in two parts. First, operation clusters in $\Omega$ are groupped, such that two elements in a group have the same number of vertices, the same number of edges and present at least one label matching function. This is done in the procedure PREPARTITION indicated in the line 1 of the algorithm 1. Second, the partition algorithm using the isomorphism test is applied only among the elements of each group.

**Algorithm 1** Partitionate $\Omega$ into instruction pattern sets

```
$G_\Omega \leftarrow \mathrm{PREPARTITION}(\Omega)$
$i \leftarrow 1$
for all $G \in G_\Omega$ do
    repeat
        $\mathrm{set}(IP_i) \leftarrow \emptyset$
        Get one element $S'$ of $G$
        $\mathrm{set}(IP_i) \leftarrow \mathrm{set}(IP_i) \cup S'$
        $G \leftarrow G - S'$
        for all $S \in G$ do
            if $S$ is label isomorph to $S'$ then
                $\mathrm{set}(IP_i) \leftarrow \mathrm{set}(IP_i) \cup S$
                $G \leftarrow G - S$
            end if
        end for
        $i \leftarrow i + 1$
    until $G = \emptyset$
end for
```

The output of algorithm 1 is a partition $\Pi^a = \{\mathrm{set}(IP_1), \mathrm{set}(IP_2), \ldots, \mathrm{set}(IP_m)\}$, for each application $a \in A$ such that

$$\Omega^a = \mathrm{set}(IP_1) \cup \mathrm{set}(IP_2) \cup \cdots \cup \mathrm{set}(IP_m)$$

and

$$\mathrm{set}(IP_i) \cap \mathrm{set}(IP_j) = \emptyset \text{ for } i \neq j,\ i = 1, 2, \ldots, m \text{ , and } j = 1, 2, \ldots, m. \tag{5.1}$$

### Cover assessment

The next step is to assess which instruction patterns are the most useful for, or *representative* to, the applications in the target set. If a custom instruction, based on a given instruction pattern $IP$, is available in an architecture, all operation clusters in the instruction pattern set $\mathrm{set}(IP)$can use it during the application mapping phase. The more *representative* one instruction pattern is, the larger is that part of one application (or application group) that can be executed using the custom instruction suggested by the pattern. It is necessary to define representativeness of an instruction pattern and how to measure it.

An approach is to use the cardinality of the instruction pattern set $|\mathrm{set}(IP)|$ as a metric for the representativeness of an instruction pattern. In that case, the more operation clusters follow a pattern, the more representative its set is. Another approach is to use the total number of operations in $\mathrm{set}(IP)$ as a metric. However, neither of these approaches account for the interference between operation clusters.

This is clarified by the example in Figure 5.6. Three operation clusters $S_1$, $S_2$ and $S_3$ were extracted with the same instruction pattern $IP_1$: two sequential additions. The instruction pattern

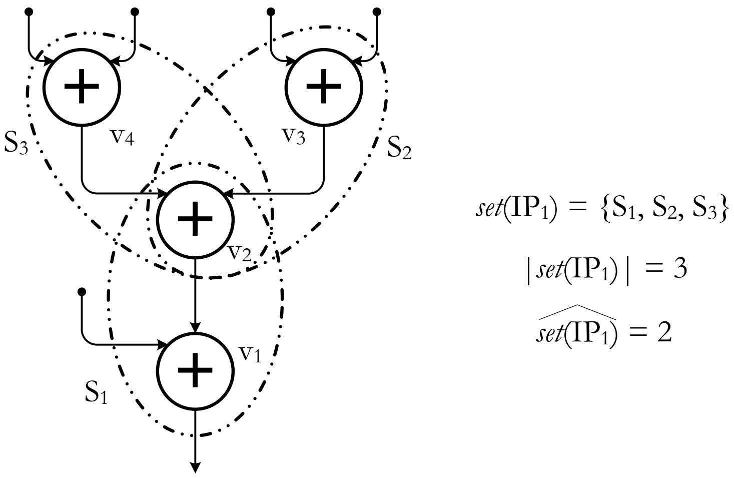


Figure 5.6.: Operation clusters with common operation vertices: only one cluster at a time may be mapped to a custom instruction.

set $\text{set}(IP_1) = \{S_1, S_2, S_3\}$ has cardinality 3 and consists of 4 operations. But these are not good measures of how representative this operation cluster is. If $S_1$ is mapped to a custom instruction, neither $S_2$ nor $S_3$ may be mapped without executing the operation of vertex $v_2$ again. The same happens when mapping $S_2$ or $S_3$ first. Only one of these operation clusters may be mapped to a custom instruction because all clusters share one vertex.

A new measure for the representativeness is proposed here:

**Definition 5.9** *The **cover set** of an operation cluster $S(V_S, E_S)$ is the set of operation vertices $V_S$. The **cover** of S, denoted $\widehat{S}$, is the cardinality of its cover set; that is $\widehat{S} = |V_S|$. Equivalently, one can say that S **covers** $|V_S|$ operations.*

Now, the representativeness of an instruction pattern set may be measured as follows:

**Definition 5.10** *The **cover of an instruction pattern set**, denoted as $\widehat{\text{set}(IP)}$, is the maximal number of operations that can be covered by any subset of non-overlapping operation clusters of* $\text{set}(IP)$*. Equivalently, let $\wp(\text{set}(IP))$ be the set of all possible subsets (*power set*) of* $\text{set}(IP)$*. Let $C = \{c \in \wp(\text{set}(IP)) | \text{for all } S_i, S_j \in c, \text{and } i \neq j \text{ then } S_i \cap S_j = \emptyset\}$ then:*

$$\widehat{\text{set}(IP)} = \max_{C} \sum_{S \in c} \widehat{S} \tag{5.2}$$

In Figure 5.6, $C = \{\emptyset, \{S_1\}, \{S_2\}, \{S_3\}\}$ for any other subset of $\wp(\text{set}(IP))$ has overlapping operation clusters. For example, $\{S_1, S_2\}$ cannot be in $C$ because $S_1 \cap S_2 = \{v_2\}$. The cover $\widehat{\text{set}(IP_1)} = 2$ is the effective number of operations that can be covered at a time by this instruction pattern.

Formally, one can obtain the cover of an instruction pattern set $\widehat{\text{set}(IP^a)}$ for a given application $a$ by means of an undirected graph $G_{IP^a}(V_{IP^a}, E_{IP^a})$. $G_{IP^a}$ is created as follows (see Figure 5.7). Each operation cluster $S \in \text{set}(IP^a)$ corresponds to a vertex $v_S$ in $V_{IP^a}$. There exists an edge $(v_{S_i}, v_{S_j})$ in $E_{IP^a}$ for each $S_i, S_j \in \text{set}(IP^a), i \neq j$, if $S_i \cap S_j \neq \emptyset$. $G_{IP^a}(V_{IP^a}, E_{IP^a})$ is called the *conflict graph* of $\text{set}(IP^a)$; two connected vertices represent operation clusters that have at least one operation vertex in common (a *conflict*). Additionally, one weight function $W : V_{IP^a} \rightarrow \mathbb{R}$ maps the cover of the operation cluster $S$ to each vertex $v_S \in V_{IP^a}$ ; that is $w(v_S) = \widehat{S}$.

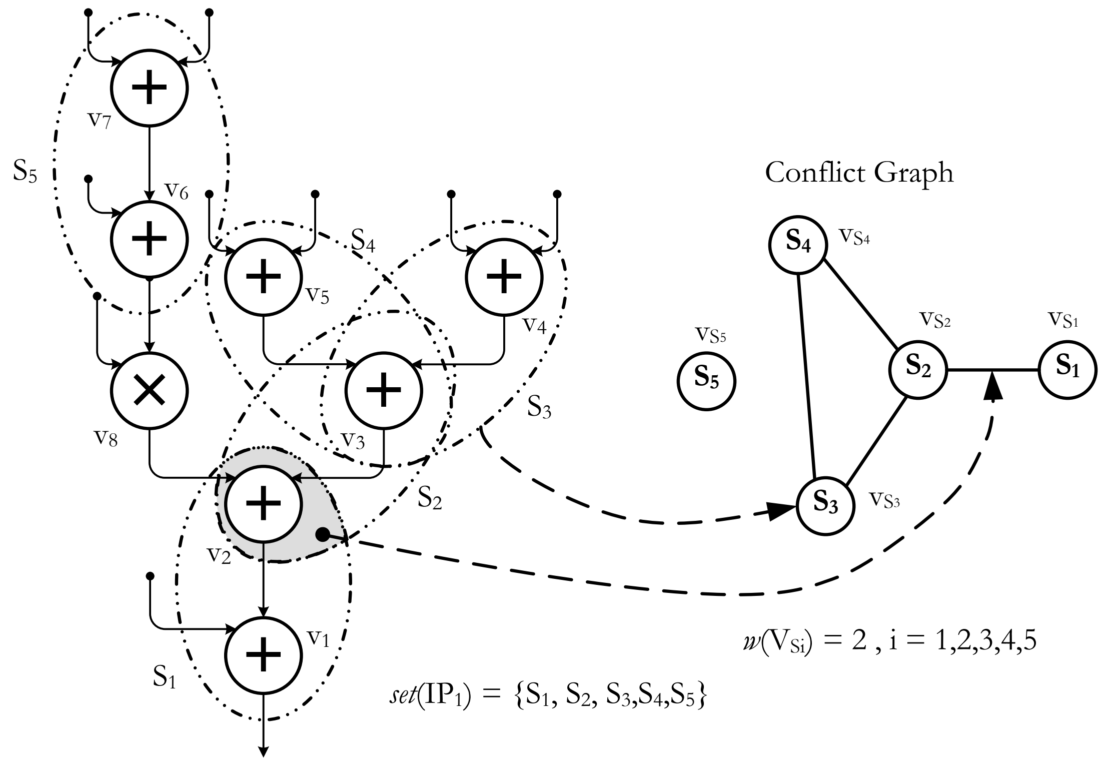


Figure 5.7.: Building the conflict graph for the instruction pattern set $\text{set}(IP) = \{S_1, S_2, S_3, S_4, S_5\}$: Vertices $v_{S_1}$, $v_{S_2}$, $v_{S_3}$, $v_{S_4}$, and $v_{S_5}$ represent the operation clusters $S_1$, $S_2$, $S_3$, $S_4$, and $S_5$, respectively. $v_{S_1}$ and $v_{S_2}$ are connected because they have $v_2$ as a common vertex. $v_{S_2}$, $v_{S_3}$ and $v_{S_4}$ are interconnected because there is a conflict in $v_3$. Operation cluster $S_5$ does not have a conflict with any other operation cluster, therefore $v_{S_5}$ has no edges.

In a conflict graph, non-connected vertices represent non-overlapping operation clusters. Each vertex is weighted by the number of operations covered by its matching operation cluster. Therefore, a subset of non-interconnected vertices with maximum sum of weights represents a group of non-overlapping operation clusters that covers a maximal number of operations, as defined in Definition 5.10. Finding this subset is known as *the maximum weight stable set problem*[38], which can be formally defined as follows.

**Definition 5.11** *Let $G(V, E)$ be an undirected graph, and $G(V')$ a subgraph induced by the set $V' \subseteq V$. If no two vertices in $G(V')$ are adjacent to each other, $V'$ is called an **stable set** in $G(V, E)$. Given a graph $G(V, E)$ and an weight function $W : V \rightarrow \mathbb{R}$, the **maximum weight stable set problem** is to find an stable set $V'$ in $G(V, E)$ of maximum weight $w(V') = \sum_{v \in V'} w(v)$ [43].*

The maximum weight stable set problem is a well-known NP-hard problem[38]. Polynomial solutions are possible for some restricted graph classes such as arc-circular[40] and overlap graphs[39].

Unfortunately, no assumptions can be made about the class of the conflict graph. Some exact algorithms for small graphs were proposed in the literature [125][9][114].

This work uses one branch-and-bound exact algorithm to solve this problem [117]. The algorithm builds a search tree where each node represents a set of vertices $V'$ not yet included in a stable set. Each node also records a variable $a_w$ for the weight of the actual node, and a variable $b_w$ for the best accumulated weight. For each node, the algorithm branches the search into $|V'|$ subtrees, each considering the inclusion of one vertex $v \in V'$ in the stable set. Each time a branch is executed, a bound is calculated for the total weight that can be achieved for any stable set obtained from this point. If this bound is less or equal than the best accumulated weight achieved at any other previously searched subtree, the search stops for the underlying subtree. If the bound is greater, there is still a chance to find an stable subset with larger weight. In this case, the actual and the best accumulated weight are updated and the search continues branching into deeper subtrees.

**Algorithm 2** The weight of a maximum weight stable set - source [117]

```
Require: Graph $G(V, E)$ and weight function $W : V \to \mathbb{R}$
procedure MAX_SS_WEIGHT:
return BRANCH($V, 0, 0$) {Initial node: $V' \leftarrow V, a_w \leftarrow 0, b_w \leftarrow 0$}
end
procedure BRANCH($V', a_w, b_w$):
for all $v \in V'$ do
    $V^* \leftarrow V' - \{v\} - adj(v)$
    $a_w^* \leftarrow a_w + w(v)$
    if $a_w^* + w(V^*) \leq b_w$ then
        return $\max\{b_w, a_w^*\}$
    else
        $b_w = \max\{b_w,$ BRANCH$(V^*, a_w^*, b_w)\}$
    end if
end for
return $b_w$
end
```

The algorithm is depicted in Algorithm 2. Notice that at each step, only the weight of the maximum weight stable set is calculated, as there is no need to record the stable set itself. The algorithm starts branching from a root node, where $V' = V$, such that the induced graph $G(V')$ is equivalent to the complete conflict graph. Additionally, $a_w$ and $b_w$ are set to 0. The branch procedure derives a new node for each $v \in V'$ using the following steps. It calculates the remaining vertex subset $V^*$ subtracting $v$ and the adjacency set $adj(v)$ from $V'$. It updates the actual accumulated weight adding to it the weight of the considered vertex $v$. Finally, it calculates an upper bound for the weight that can be achieved from this node: a maximum weight subset is achieved from this node if the remaining set $V^*$ induces a graph $G(V^*, E^*), E^* = \emptyset$. In this case, all the vertices in $V^*$ contribute to the weight of the stable set in $w(V^*)$. If the upper bound is less or equal than a previously obtained best accumulated weight, that is $a_w^* + w(V^*) \leq b_w$, the subtree under this node is ignored.

Figure 5.8 shows the application of Algorithm 2 to the example DFG given in Figure 5.7. The root node corresponds to all vertices in the conflict graph. The left subtree considers the inclusion of the vertex $v_{S_1}$ in the stable set. The first derived node represents the graph induced by $V^* = \{v_{S_3}, v_{S_4}, v_{S_5}\}$. The actual accumulated weight is $a_w = w(v_{S_1}) = 2$ because the only node in the stable set up to here is $v_{S_1}$. The calculated upper bound at this node assumes that $v_{S_3} \cap v_{S_4} \cap v_{S_5} = \emptyset$. If this is the case, the total weight under this subtree would be 8. The search proceeds until the leaves of this subtree, and a best accumulated weight of 6 is achieved. Later, another branch is created from the root node considering the inclusion of $v_{S_2}$. Here, the first derived node has a maximal weight of 4. This is because if $v_{S_2}$ is in the stable set, $v_{S_1}$, $v_{S_3}$ and $v_{S_4}$ cannot be included in the stable set anymore. The search does not need to continue in this subtree. The weight of the maximum weight stable set is 6. That corresponds to covering the data flow graph with $S_1, S_5$, and $S_3$ or $S_4$.

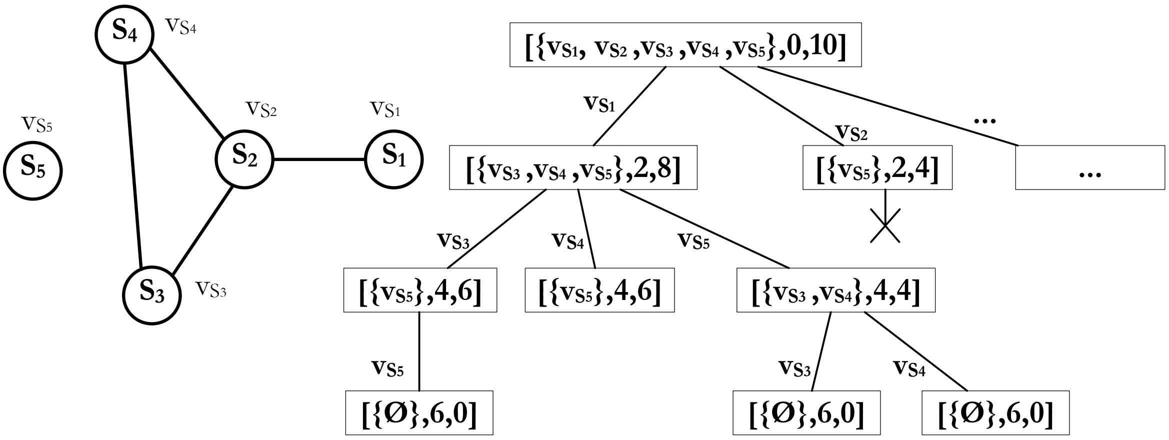


Figure 5.8.: Calculating the cover of $\mathrm{set}(IP) = \{S_1, S_2, S_3, S_4, S_5\}$, for the example DFG depicted in Figure 5.7. This search tree is generated executing the Algorithm 2 to the example conflict graph. The conflict graph is repeated here to ease the understanding. Each node is depicted as a tuple $[\{v_1, v_2, \ldots, v_k\}, a_w^*, a_w^* + w(V^*)]$, representing the nodes not yet considered in the stable set, the actual accumulated weight up to this node, and the upper bound to the weight that can be achieved by the underlying subtree.

One may also calculate the cover of several applications by building a conflict graph based on the set $\mathrm{set}(IP) = \bigcup_{a \in A} \mathrm{set}(IP^a)$. This is , however, not necessary because two applications have no operation clusters in common (and consequently no common operation vertices). Therefore, the cover $\widehat{\mathrm{set}(IP)}$ along all applications may be easily calculated by

$$\widehat{\mathrm{set}(IP)} = \sum_{a \in A} \widehat{\mathrm{set}(IP^a)}. \tag{5.3}$$

Now, $\widehat{\mathrm{set}(IP)}$ is a measure of how representative the instruction pattern $IP$ is to the target application set.

Finally, the concept of representativeness of an instruction pattern $IP$ can be extended to a group of instruction patterns $IP_1, IP_2, \ldots, IP_k$. In this case, operation clusters $S_i$ and $S_j$ may still have common vertices, even if they follow different instruction patterns $IP_i$ and $ip_j$, respectively. $S_i$ and $S_j$ are also in conflict, and only one of them will be mapped into a custom instruction. Similar to the definition 5.10:

**Definition 5.12** *Let $\bar{IP} = \text{set}(IP_1) \cup \text{set}(IP_2) \cup \ldots \text{set}(IP_k)$ be a set of $k$ instruction pattern sets. The* ***cover of an instruction pattern selection****, denoted by $\widehat{\bar{IP}}$, is the maximal number of operations that can be covered by any subset of non-overlapping operation clusters of $\bar{IP}$. Equivalently, let $\wp(\bar{IP})$ be the set of all possible subsets (power set) of $\bar{IP}$, and let $\bar{C} = \{\bar{c} \in \wp(\bar{IP}) |$ for all $S_i, S_j \in \bar{c}$, and $i \neq j$ then $S_i \cap S_j = \emptyset\}$. Then:*

$$\widehat{\bar{IP}} = \max_{\bar{C}} \sum_{S \in \bar{c}} \widehat{S} \tag{5.4}$$

Similar to equation 5.3, the cover $\widehat{\bar{IP}}$ along all applications is

$$\widehat{\bar{IP}} = \sum_{a \in A} \widehat{\bar{IP}^a}. \tag{5.5}$$

### Selection of instruction pattern sets

Up to this point, clusters of operations were extracted from the target applications, and grouped into sets using their similarity or instruction pattern. Also, the representativeness of an instruction pattern, i.e. to which extent one application can make use of a custom instruction based on a given instruction pattern, was evaluated. And this evaluation was expanded to consider several applications and several instruction patterns simultaneously. In this last step, one subset of high representative instruction patterns will be selected to guide the design of custom instructions. It is important to keep the number of custom instructions to be generated small. A large set of new instructions hinders the productivity of the design phase and increases the implementation costs.

The more patterns are included in the selected group, the larger is the number of custom instructions in the architecture, and the larger the chance to bind operations onto custom hardware during the application mapping phase. However, each new custom instruction produces an overhead in the design phase, as is must be integrated and tested. Each new instruction demands the addition of new hardware modules and modification of existing ones, such as I/O ports, multiplexers and register banks. This increases the complexity of the design, implementation area and static power consumption. Therefore, it is desirable to find a small number of instruction patterns that is extensively representative of the target applications.

## 5.1.3. Custom Instruction Composition

Custom instruction composition is the process that transforms an operation cluster into the description of an instruction datapath. The custom instruction composition is *simple* if it considers only one operation cluster. It is *compound* if it considers more than one operation cluster that do not follow the same instruction pattern. The compound process improves the design of custom instructions because it allows different instruction datapaths to reuse hardware modules.

The *simple custom instruction composition* is a direct mapping between vertex and edges of the operation cluster graph to modules and communication paths that describe a hardware datapath. One module is created for each vertex in the operation cluster. Modules are hardware units, like adders and multipliers; they execute the atomic operation indicated by the corresponding vertex. Input and output vertices in the operation cluster generate input and output ports at the datapath description, respectively. The elements in the datapath description are interconnected in correspondence to the edges of the operation cluster.

In the example depicted in Figure 5.9, the operation cluster $S$ composes a custom instruction. Two adders and a multiplier modules are created; they correspond to vertices $v_1, v_2$ and $v_3$, respectively. Vertices $v_6$, $v_7$, and $v_8$ describe three input ports, and the vertex $v_5$ is transformed into an output port. The modules in the datapath description are connected according to the edges in $S$; for example, the edge $(v_1, v_5)$ implies a connection between the multiplier and the output port module. Data is transferred between modules using these connection lines.

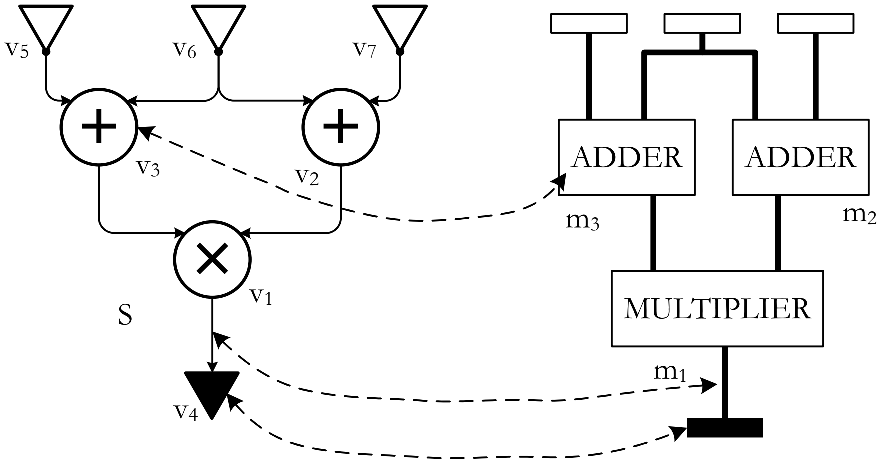


Figure 5.9.: Simple custom instruction composition. Only one instruction pattern is considered.

A simple custom instruction composition generates a feasible datapath if the operation cluster satisfies the properties discussed in Section 5.1.1. These properties guarantee that the datapath description satisfies the constraints for number of input and output ports, implementation cost, and execution delay. However, generating one datapath for each operation cluster is not efficient because it duplicates hardware modules that could be shared between two datapaths. Suppose, for example, that the two operation clusters depicted in Figure 5.10 are implemented as custom instruction in the same processing element. If they are generated using a simple composition, two multipliers are instantiated, one for each special instruction. These multipliers will never be used simultaneously, as a PE executes only one instruction at a time; a more efficient datapath could be generated where the multiplier is shared between both instructions.

The *compound custom instruction* composition merges two or more operation clusters and use the resulting graph to describe a reconfigurable instruction datapath. The goal is to design a datapath with the minimum number of functional units and interconnections. Reconfigurable means that

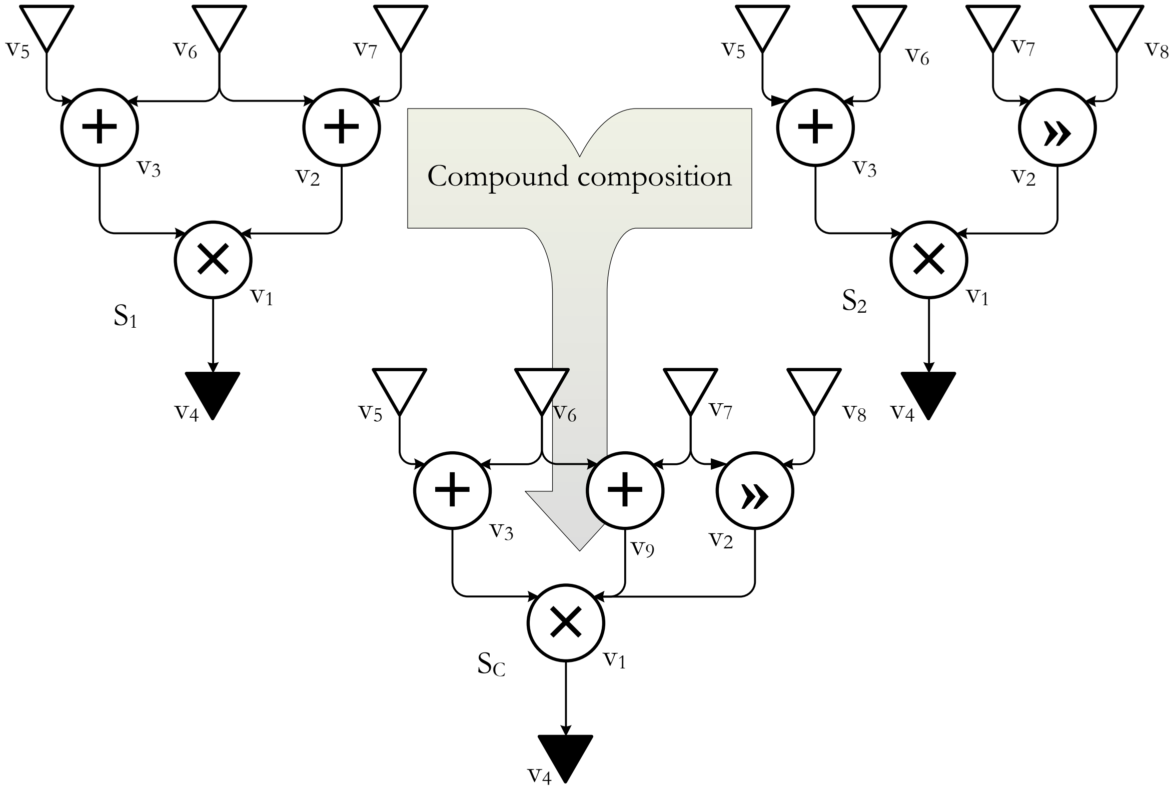


Figure 5.10.: Compound composition.

this datapath can be controlled at run time to execute any instruction corresponding to the merged operation clusters.

Consider the design of a compound custom instruction based on two operation clusters $S_1$ and $S_2$, in Figure 5.10. $S_1$ and $S_2$ can be merged into a graph $S_c$ as follows:

1. Create a graph $S_c(V_c, E_c)$ such that $S_1 \subseteq S_c$ and $S_2 \subseteq S_c$.

2. The number of input vertices in $S_c$ is equal to the maximum number of input vertices among all operation clusters. $S_1$ has 3 input vertices, whereas $S_2$ has 4. Consequently, $S_c$ has also 4 input vertices.

3. The number of output vertices in $S_c$ is equal to the maximum number of output vertices among all operation clusters. The number of output vertices of both $S_1$ and $S_2$ are equal to 1; $S_c$ has therefore 1 output vertex.

4. The number of vertices in $S_c$ representing an atomic operation $t$ is equal to the maximum number of vertices representing $t$ among all operation clusters. $S_1$ has two vertices representing an addition, whereas $S_2$ has one; thus, $S_c$ contains 2 vertices representing additions.

The number of vertices in $S_c$, when constructed in this way, is the minimal to meet the conditions $S_1 \subseteq S_c$ and $S_2 \subseteq S_c$. Adding another vertex to the merged graph does not increase parallelism, because each instruction is executed once at a time. The vertex set of the compound operation cluster $S_c$ is therefore simply the maximum number of each vertex type among all the involved operation clusters.

The next problem is to map the vertices of $S_1$ and $S_2$ onto vertices of $S_c$ such that the number of interconnections is minimal. This problem is solved using the concept of edge similarity. A vertex[2] $(v_i^1, v_j^1)$ of $S_1$ is called *similar* to the vertex $(v_l^2, v_m^2)$ of $S_2$, and denoted $(v_i^1, v_j^1) \sim (v_l^2, v_m^2)$ if $Op(v_i^1) = Op(v_l^2)$ and $Op(v_j^1) = Op(v_m^2)$. In other words, two vertices are similar if the following conditions hold:

- the sources of both vertices represent operations of the same type; and
- the sinks of both vertices represent operations of the same type.

For example, in Figure 5.10, the edge $(v_1^1, v_4^1)$ is similar to $(v_1^2, v_4^2)$ because both of them connect a vertex representing a multiplication to an output vertex. Figure 5.11 lists all possible similarities between the vertices of $S_1$ and $S_2$. Additionally, suppose that $(v_i, v_j)$ and $(v_o, v_p)$ are edges of $S$ and $(v_l, v_m)$ and $(v_r, v_s)$ are edges of $S'$. Two edge similarities $(v_i, v_j) \sim (v_l, v_m)$ and $(v_o, v_p) \sim (v_r, v_s)$ are *incompatible* if and only if they map the same vertex of $S$ to two different vertices of $S'$. For the example in Figure 5.10, the similarities $(v_3^1, v_1^1) \sim (v_3^2, v_1^2)$ and $(v_2^1, v_1^1) \sim (v_3^2, v_1^2)$ are incompatible because they map the vertex $v_3^2$ to two different vertices $v_2^1$ and $v_3^1$. Incompatible edge similarities may be found if at least one of the following conditions is true:

$$
\begin{array}{lll}
(v_1^1, v_4^1) \sim (v_1^2, v_4^2) & (v_3^1, v_1^1) \sim (v_3^2, v_1^2) & (v_2^1, v_1^1) \sim (v_3^2, v_1^2) \\
(v_5^1, v_3^1) \sim (v_5^2, v_3^2) & (v_5^1, v_3^1) \sim (v_6^2, v_3^2) & (v_6^1, v_3^1) \sim (v_5^2, v_3^2) \\
(v_6^1, v_3^1) \sim (v_6^2, v_3^2) & (v_6^1, v_2^1) \sim (v_5^2, v_3^2) & (v_6^1, v_2^1) \sim (v_6^2, v_3^2) \\
(v_7^1, v_2^1) \sim (v_5^2, v_3^2) & (v_7^1, v_2^1) \sim (v_6^2, v_3^2) &
\end{array}
$$

Figure 5.11.: All possible similarities between edges of $S_1$ and $S_2$ in Figure 5.10.

$$
\begin{aligned}
C_1 &: (v_i = v_o) \wedge (v_l \neq v_r) \\
C_2 &: (v_l = v_r) \wedge (v_i \neq v_o) \\
C_3 &: (v_j = v_p) \wedge (v_m \neq v_s) \\
C_4 &: (v_m = v_s) \wedge (v_j \neq v_p)
\end{aligned}
\tag{5.6}
$$

An *incompatibility graph* is a graph $H(V_H, E_H)$, such that each vertex of $H$ corresponds to a similarity between two edges: one from $S_1$ and other from $S_2$. There exists an edge between two vertices of $H$ if two similarities are incompatible. Sets of non-interconnected vertices (stable set) in $H$ correspond to edges that can be merged in the compound composition process because they are similar and *not* incompatible. In a maximal stable set in $H$, all the similar edges in the operation clusters, considered during the compound composition, may be merged. Therefore, the problem can be formulated as that of finding a maximal stable set in the incompatibility graph $H$. This can be solved using the Algorithm 2 in Section 5.1.2. The incompatibility graph for the similarities in Figure 5.11 is depicted in Figure 5.12.

[2] Here, the superscript in the vertex representation indicates to which subgraph (instruction pattern) belongs the vertex.

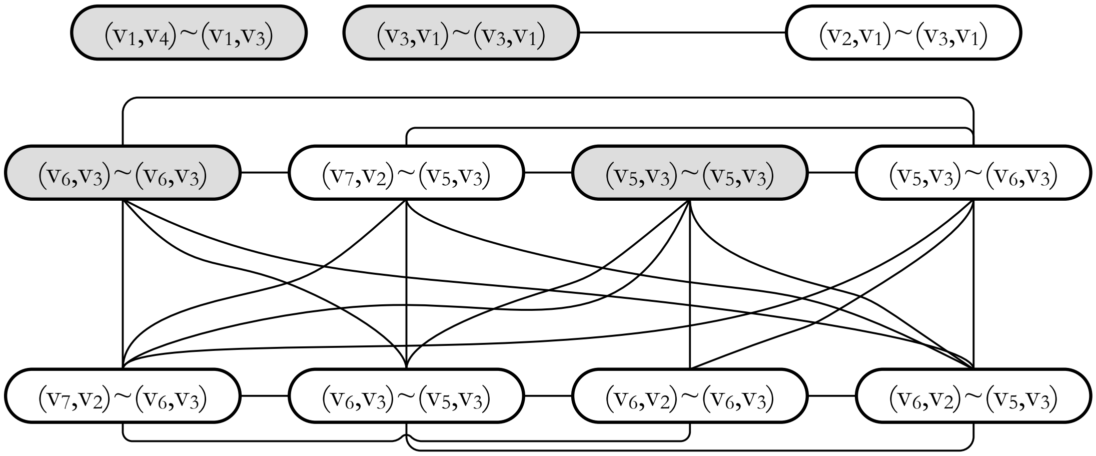


Figure 5.12.: Incompatibility graph for the edge mapping between $S_1$ and $S_2$ in the Figure 5.10. Vertices shaded in dark are part of the maximal stable set extracted.

The problem of merging two DFGs was formally described and solved in the work of Huang and Moreano[59][109]; they applied the discussed solution to merge the datapath of program loops during the design of coprocessors. They also show that any stable set found in the incompatibility graph leads to a merged graph with the minimal number of connections.

The explanation presented in this work considers only two operation clusters. In order to merge more than two operation clusters, this method is applied iteratively; at each iteration, a previously merged solution and a new operation cluster are considered. The algorithm to merge more than two graphs is presented in [59].

By merging all the operation clusters, an unique graph is obtained, which can be used to describe the datapath of merged custom instructions. This graph is transformed using the simple custom instruction composition method discussed previously. There is, however, one difference: when the input port of an operation module in the datapath has to be used by two or more connection lines, a multiplexer is inserted in the design. For example, consider the graph $S_c$ in Figure 5.13, obtained by merging the graphs $S_1$ and $S_2$ in Figure 5.10. The second operand of the multiplier module has to be shared by one of the adder modules and the shifter (edges $(v_9, v_1)$ and $(v_2, v_1)$). It is necessary to insert one multiplexer which selects which data are to be used, as a function of the instruction to be executed.

A control circuit to reconfigure this merged datapath must be designed manually. The control circuit receives as input the code for the instruction that should be executed. It can be any instruction corresponding to the considered operation clusters. The control circuit output signals to enable or disable operation modules, as well as controlling the multiplexers. When a certain instruction is chosen, the active part of the datapath must correspond to the datapath of the chosen instruction as if it was obtained using a simple composition process.

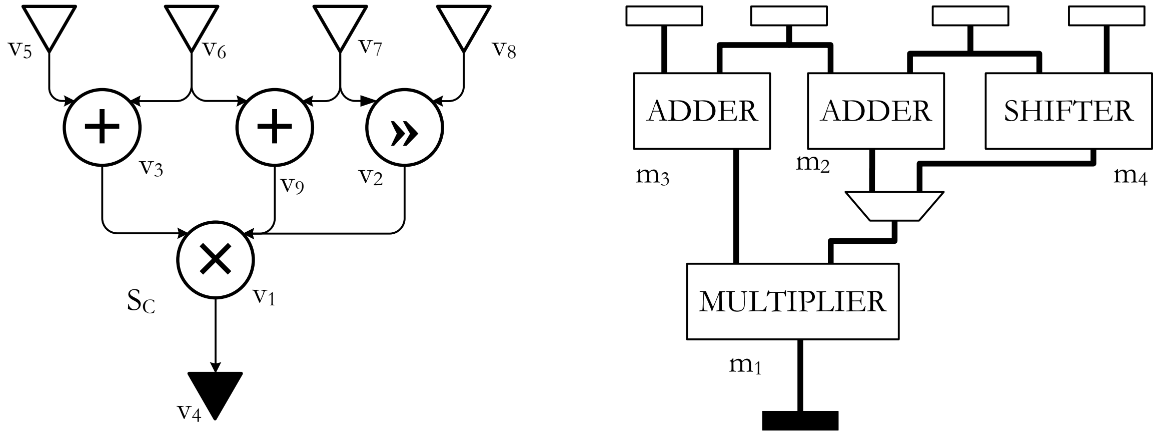


Figure 5.13.: The graph $S_c$ is used to describe the datapath of compound custom instructions based in the operation clusters $S_1$ and $S_2$ in Figure 5.10.

## 5.2. Custom Instructions for Coarse Grained Architectures

Integrating the designed custom instructions in the instruction set of coarse grained architectures involves three points, which includes:

- modifications that should be carried out in the description of the FU behavior;
- modifications that should be carried out in the compiler to adapt the application mapping phase to the new integrated custom instructions;
- the organization (distribution) and incorporation of processing elements with custom instructions in the architecture.

### 5.2.1. Description of custom instructions in CGADL

The first step is to describe the behavior of a newly designed custom instructions in the CGADL description. In this case, it is necessary to create a behavioral description by using, for example, the programming language `C`. This process was explained in Section 4.2.2, but for the sake of understanding, an example will be provided here.

The custom instruction depicted in the detail of Figure 5.14 receives three operands and outputs the sum of them: it will be then denoted as `Add3`. This new operation can be included in the set of operations of an FU, say `myCustomFU`, in a CGADL description, as follows:

```
FU myCustomFU(op1, op2, ... , Add3);
```

where, `op1` and `op2`, are operations already existent in the `myCustomFU` FU. The behavior of the `Add3` operation is then described in `C` as:

```
unsigned int Add3(int args[3]) {
    return args[0] + args[1] + args[2];
}
```

Custom instructions are represented, in each case, individually, even if their datapath is merged with those of other custom instructions (compound composition).

### 5.2.2. Integration of custom instructions in the application description

The use of custom instructions during the application mapping phase starts with the application description: the scheduling, binding and routing tasks must already know which parts of the application will be mapped into custom hardware. Accordingly, the compiler must be extended to recognize and transform these parts. There are two ways to integrate custom instructions in the application description: an explicit modification of the base description, or a modification of the intermediary representation. In the first case, the designer uses function calls or keywords to determine exactly that part of the application that will be mapped to a custom instruction. In the second case, the compiler identifies and transforms these parts of the intermediary description that can be mapped to a custom instruction.

This work modifies the application's DFG automatically, since the operation clusters used to generate custom instructions are known from the design phase. The compiler identifies the operation clusters that corresponding to one of the custom instructions available in the architecture, and replace them with one node that represents this new operation. Suppose that the target architecture supports a custom instruction that has two sequential additions as instruction pattern (see Figure 5.14). The compiler identifies the operation clusters within the DFG that match the instruction pattern ($S_1, S_2, S_3, S_4, S_5$), and replace each one of them by a single operation node representing the custom instruction (node marked with #). The resulting DFG contains atomic nodes representing both single operations and collapsed clusters.

Identifying certain operation clusters within a DFG is equivalent to finding subgraphs, that match a given pattern, within a graph. This problem is largely studied and is referred to as subgraph isomorphism [31][25], pattern matching [24], and subgraph covering [48]. Given an instruction pattern subgraph, the present work solves this problem using the following steps:

- The algorithm presented in Section 5.1.1 is used to list all convex operation clusters that have an equal or smaller amount of output ports, an equal or smaller amount of input ports, an equal or smaller implementation cost, and an equal or smaller execution delay as the matching instruction pattern;
- The list is filtered to remove all operation clusters that do not have the same number of both input and output ports as the matching instruction pattern;
- A graph isomorphism test (the same used in Section 5.1.2) is performed between each remaining operation cluster and the instruction pattern graph; a positive match indicates that the operation clusters under test follow the instruction pattern, otherwise the operation cluster is removed from the list.

The resulting list indicates all operation clusters within the DFG that match the desired instruction pattern.

In this final list, there may exist some overlapping operation clusters; for example, $S_2$, $S_3$, and $S_4$ in Figure 5.14 overlap, because they have at least one common operation ($v_3$). The same holds for

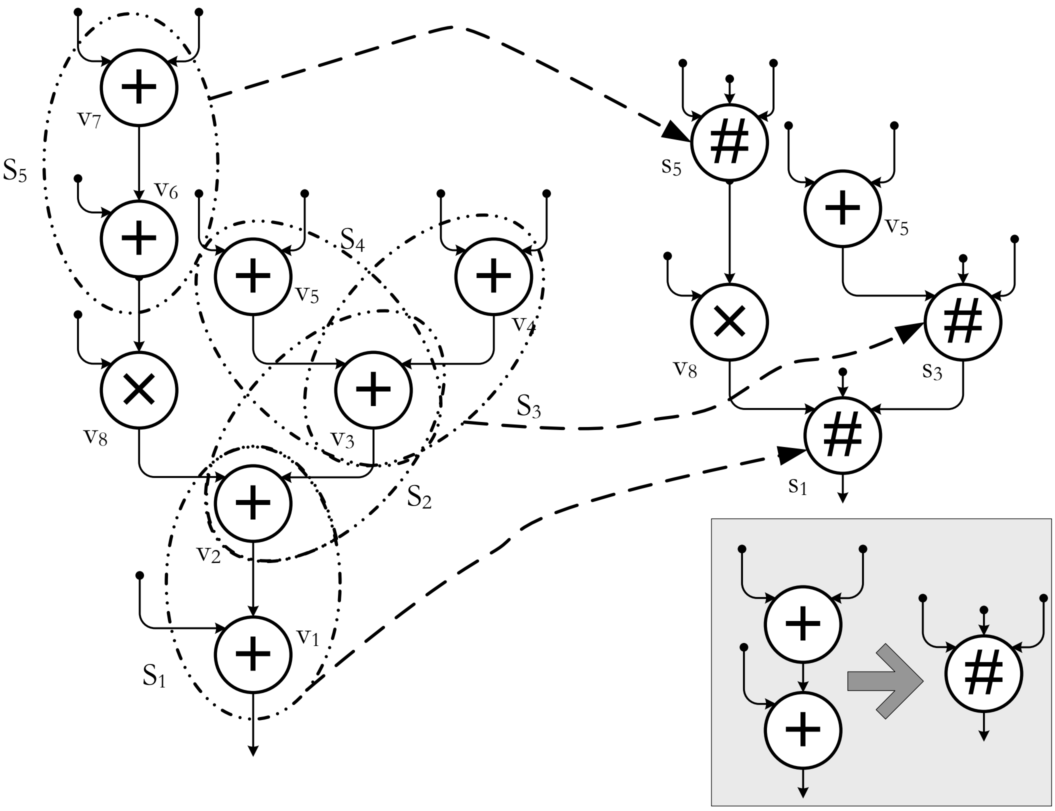


Figure 5.14.: Modifying the original DFG to incorporate nodes that represent a custom instruction: clusters of operations that match the instruction pattern of a custom instruction are identified and colapsed into one single atomic operation node.

$S_2$ and $S_1$, which share $v_2$. When two or more clusters overlap, only one of them can be mapped to a custom instruction; otherwise, the shared operations will execute unnecessarily more then once. The problem to determine which clusters will be mapped to a custom instruction is known as *allocation*.

Allocation is a classical task within the high level synthesis, together with scheduling and binding (discussed in Section 2.2.2). The allocation task determines the type and quantity of resources to map the application. The goal of allocation is to make appropriate trade-offs between the mapping cost and performance. For example, when applied to the DFG in Figure 5.14, the allocation task determines if the operation cluster $S_3$ is to be implemented using one PE with custom instruction #, or using two PEs with additions. In this work, allocation is resolved such that the number of clusters that will be mapped into custom instructions is maximum.

### 5.2.3. Integration of custom processing elements in the architecture

When several types of processing elements are available to compose the architecture array, two important questions arise. How many PEs should be used from each type? And how should they be distributed within the array? The design methodology presented in previous sections may produce differentiation between PEs for several reasons: (1) custom instructions are designed using different operation clusters; (2) they are grouped for merging, such that non similar clusters are placed in different groups; and (3) merged custom instructions are embedded in the functional units of different PEs. If distinct PE types are available, how many of each type should be used to compose the array? This problem is called *the array allocation problem.* Moreover, assuming the necessary quantity is known, how should they be displaced in the array? Does the spatial distribution of PEs, defined at design time, has an influence for the application mapping phase? This problem is called *the array placement problem.*

This section proposes a method to answer these questions when PEs with different instruction sets are used to compose the array. Solutions for the array allocation and the array placement problems are usually grouped in the literature in two classes depending on the array compostion: homogeneous arrays or custom arrays. This work defines and destinguishes a third class called *pattern array*. These classes are explained in the following.

### Homogeneous arrays

*Homogeneous arrays* use only one type of processing element, which implies that differences between distinct PEs must be resolved before the array is composed, as depicted in Figure 5.15. Only one PE type is designed which supports all custom instructions. The number of PEs to be allocated is the minimal number of PEs necessary to run all the target applications under their performance constraints. The array placement problem is reduced to that of determining the array geometry.

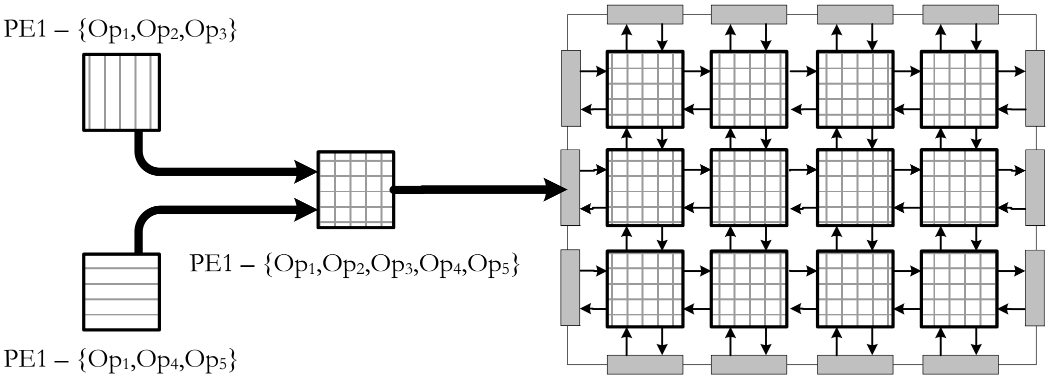


Figure 5.15.: Homogeneous arrays are composed with one PE type. If several PE types are available, they must be first combined into one PE type containing all the operations supported by them.

Homogeneous arrays are simple to design. They maximize the reusability of the PE and are easy to scale. They also simplify the application mapping phase, as scheduling and binding phases do not have to consider different types of elements and their position. However, this simplicity has a cost: PEs tend to have a larger silicon area and higher power consumption due to the hardware components that implement all custom instructions. Often, there will be more instructions available in the architecture than it is necessary to map the applications; that is, unnecessary hardware overhead.

The first coarse grained arrays were designed using homogeneous arrays [71] [11]. However, nowadays designers tend to use different types of PEs. Area and power consumption costs may be reduced if specific functional units, such multipliers, may be shared among PEs[65]. PEs with multiple functional units may be more efficient than their one-FU counterparts[10], as discussed in Chapter 3.

### Custom arrays

In *custom arrays*, the allocation and placement of PEs reflect specific requirements of the target applications. The array allocation problem is solved quite straightforward: the designer allocates the minimal amount of PEs (of each type) that is necessary to map the applications. The distribution of PEs in the array and the array geometry are tailored to the set of applications; and therefore highly dependent on this set.

Custom arrays are smaller, consume less energy and provide a better performance. However, their design is complex and expensive because no generic solution to this problem is available. Moreover, this hand-tailored array usually requires the compiler to be matched; the application mapping must be "aware" of specific issues, like the position of PEs and different intercommunication interfaces.

### Pattern arrays

This work introduces a third category to classify the composition of coarse grained arrays: *pattern arrays*. *Pattern arrays* allocate and distribute PEs according to some rules. For example, PEs with custom instructions for I/O access should be placed on the border, and general purpose PEs at the center (see Figure 5.16a). Rules may also simply describe an desirable array configuration, usually repetitive and scalable, for this distribution; for example, a *cluster* configuration joins PEs of the same type forming islands along the array (Figure 5.16b). This rule base placement makes it easier to implement and scale the array than at custom arrays. Note that an homogeneous array may be described according to the rule that the array has only one PE type, which supports the whole instruction set (also custom instructions).

The array allocation problem is solved in a similar way as for custom arrays: a minimal number of PEs (of each type) are allocated according to the demand of target applications. Like custom arrays, pattern arrays use a smaller silicon area and consume less energy than their homogeneous counterparts.

### Summary

Table 5.2 resumes the advantages and drawbacks of homogeneous, custom, and pattern arrays.

## 5.3. INSTPATT - A software tool for extraction of instruction patterns

During the development of this work, a software tool, called *InstPatt*, was developed to extract operation clusters out of an application's DFG, and group them into instruction pattern sets. This

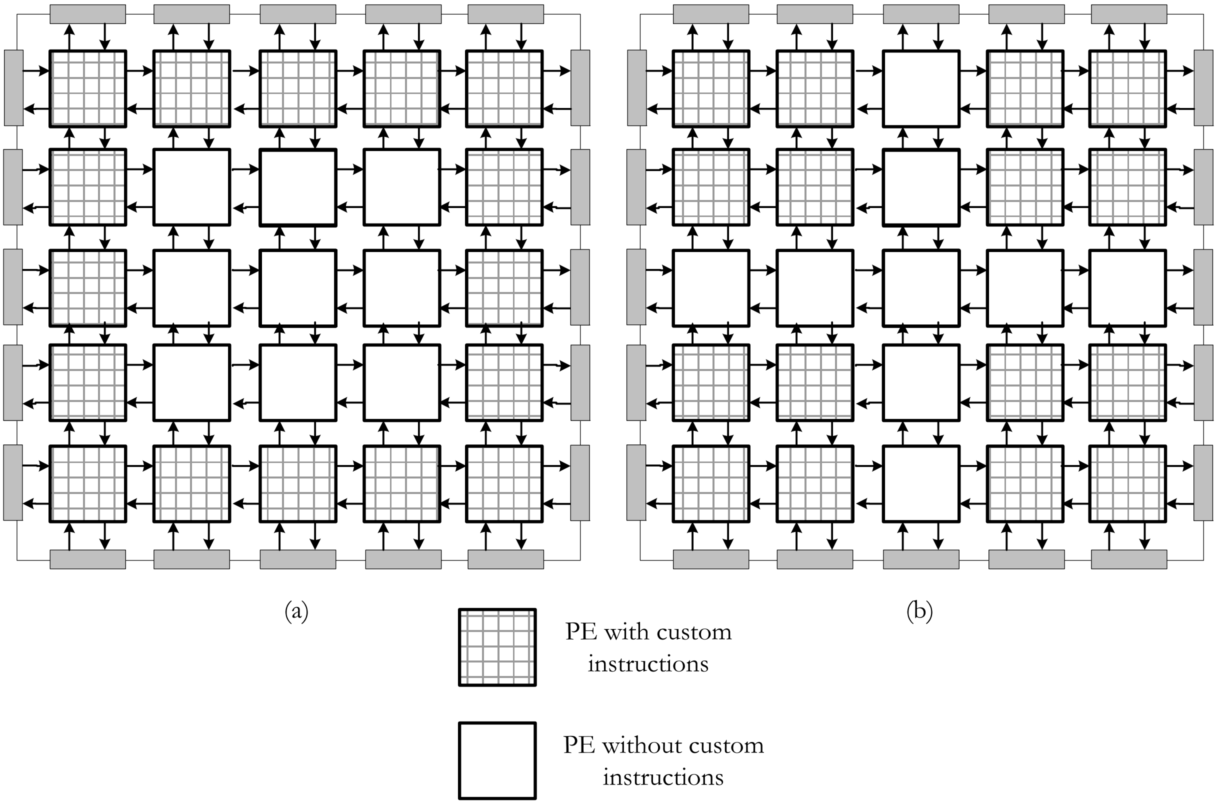


Figure 5.16.: Two different pattern arrays: *custom on borders* (a) and *cluster* (b).

tool automizes the methods and algorithms presented in Sections 5.1.1 and 5.1.2. Like the hardware complexity estimation tool, the *InstPatt* software demonstrates the feasibility and practical applicability of the algorithms introduced in this work. It also provides automation for some tasks of the proposed methodology and a software-based production of results;

### 5.3.1. The *InstPatt* flow

The *InstPatt* tool was implemented in the programming language `Java`. It receives as input the description of one application's DFG in an XML dialect called *GraphML* [46]. The tool parses the GraphML notation and creates an internal representation of the graph. This graph is available inside the tools for further processing as indicates the flow in Figure 5.17.

Basically, the *InstPatt* tool processes the applications DFG in three steps: an operation cluster extraction step; a pre-partitioning of clusters based on some characteristic, such as the number of nodes in the graph; and a label isomorphism partitioning. After the label isomorphism partitioning, instruction pattern sets are available for statistical and cover evaluation. These steps are explained in the following.

Table 5.2.: Comparison between *homogeneous*, *custom*, and *pattern* arrays: possible solutions to allocate and place PEs in the array during the design phase.

| | Homogeneous | Pattern | Custom |
|---|---|---|---|
| Silicon area usage | $--$ | $+$ | $++$ |
| Power consumption | $--$ | $+$ | $++$ |
| Complexity for the design | $++$ | $+$ | $--$ |
| Complexity for the application mapping | $++$ | $+$ | $--$ |
| Flexibility | $++$ | $+$ | $--$ |

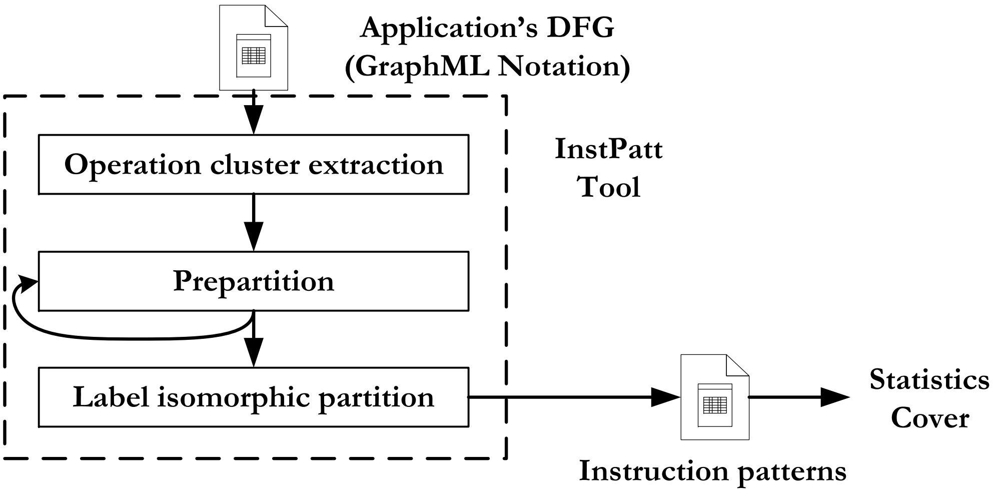


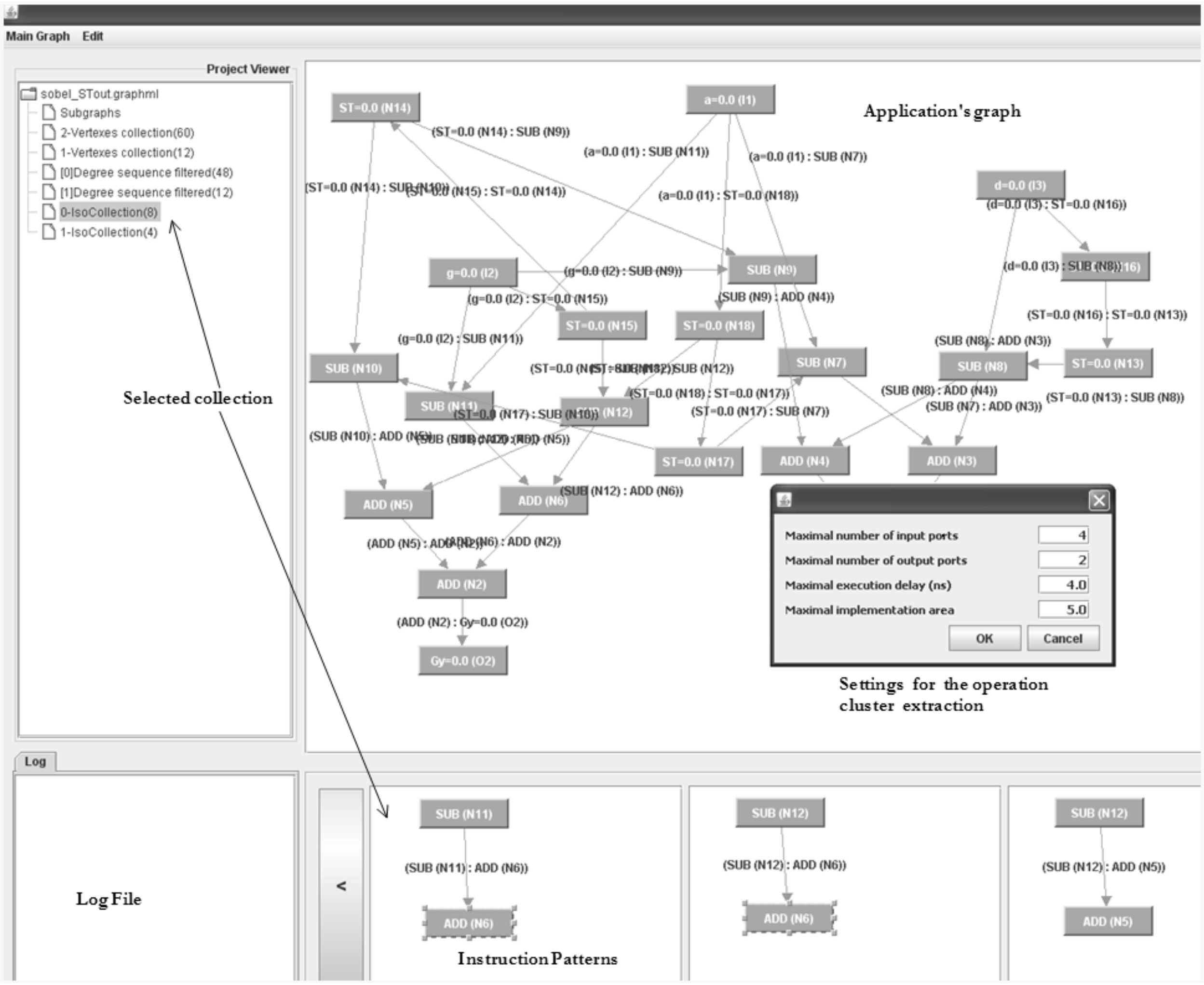
Figure 5.17.: Workflow of the software tool *InstPatt*.

### Extraction of operation clusters with the *InstPatt* tool

A screen shot of the *InstPatt* tool is depicted in Figure 5.18. Before operation clusters can be extracted, it is necessary to open an application's DFG. That is done through the menu item `Main Graph`. The most recent application's DFG is shown in the central window, and the application name is shown as a file in the `Project Viewer` area (left superior side).

To extract operation clusters from the application's DFG, the designer right-clicks in the application name (`Project Viewer` area) and chooses the option `Extract Subgraphs`. The extraction settings dialog appears, which is depicted in the detail of Figure 5.18. In this dialog, the designer can determine the maximal number of input ($N_{\text{in}}$ and output ($N_{\text{out}}$) data, the maximal execution delay ($\delta_{\text{max}}$), and the maximal estimated implementation area ($C_{\text{max}}$), as discussed in Section 5.1.1. When `Ok` is pressed in this dialog, the *InstPatt* tool runs the algorithm presented in Section 5.1.1 and extracts all operation clusters with the given characteristics.

The set of extracted operation clusters can be seen by selecting the item `Subgraphs` in the `Project Viewer` area. Moreover, this collection can now be further processed and splitted in smaller subsets.



Application's graph

Selected collection

Maximal number of input ports 4
Maximal number of output ports 2
Maximal execution delay (ns) 4.0
Maximal implementation area 5.0

Settings for the operation cluster extraction

Log File

Instruction Patterns

Figure 5.18.: Screenshot the software tool *InstPatt*. The project viewer on the left side lists collections (sets) of operation clusters extracted from the application's DFG. The selected collection can be seen separately in the windows at the lower part. A log viewer prints out error and warning messages. The area at the center depicts the application's DFG graph. In detail, one can see the dialog to adjust the settings for the operation cluster extraction.

## Prepartition and label isomorphic partition

The prepartition of the set of operation clusters is not obligatory. The set can be directly partitioned, after it is generated, into label isomorphic subsets. However, depending on the number of elements in this set, and the individual size of the operation clusters, this task can be computing intensive and require time. To overcome this problem, it is possible to previously partition the set of operation clusters according to the following criteria: number of nodes in the cluster, number of edges in the cluster, and input/ouput degree of the nodes in the cluster. All the operation clusters within a prepartition collection has the same characteristic. For example, the operation clusters set in Figure 5.18 (`Subgraphs`), was prepartitioned in two subsets with respectivelly one and two nodes in the cluster: the `2-Vertices Collection` and the `1-vertex Collection`.

Each subcollection can be further partitioned according to other criterium. The objective, however, is to obtain label isomorphic collections, that correspond to the instruction pattern sets. For that, the *InstPatt* tool applies label isomorphic partition algorithm introduced in Section 5.1.2. The output of this phase is a partition of the set of operation clusters into instruction pattern sets.

From this point on, the *InstPatt* tool helps the designer to visualize and identify where operation clusters corresponding to a given instruction set can be found in the application's DFG. Moreover, it allows the designer to evaluate the frequency with which each instruction pattern can be found in the application. And it helps the designer to calculate the cover of a given instruction pattern.

# 6. Experiments and Results

This chapter presents experiments and results involving the description of coarse grained arrays and PEs with CGADL, and the specilization of these arrays by using custom instructions. The starting point for all the experiments is the CGADL description of the PE *Bianca*, discussed in details along the Chapter 4.

Section 6.1 estimates the hardware (gate equivalency) costs directly from the CGADL description by using the hardware complexity estimation methodology and the software tool, proposed by the author in Section 4.3. These estimated costs are compared to the implementation costs obtained after using a commercial synthesis tool. Results show that CGADL can be used to estimate implementation costs without the need of synthesis, and therefore, earlier in the design phase. Moreover, these results demonstrate, through an example, that it is possible to derive software tools out of the description language proposed here.

Section 6.2 evaluates the impact of using custom instructions during the design (specialization) of coarse grained architectures. The starting point for these experiments are coarse grained arrays based on the PE *Bianca*, analysed in 6.1. The methodology, techniques, and algorithms introduced in Chapter 5 are used custom two arrays according to the demand of two real-world application domains: the scalable OFDMA modulation scheme, and computer-vision for driving-assistance systems.

Each one of these experiments was partially automated by using the software tools presented in Sections 4.3.4 and 5.3.

## 6.1. A CGADL-based software tool: estimation of hardware costs

Chapter 4 introduced the coarse grained architecture description language CGADL, and presented a method to estimate the hardware complexity of PEs and architecture instances described in this language. This section uses this estimation method as an example to demonstrate that software tools can be derived from CGADL, and that these tools may produce results which are comparable to results obtained from synthesis in a traditional design flow. The experiments in this section capture the tasks within this method in a software framework, and apply it to answer three usual questions about the design of coarse grained arrays: (1) how does the silicon area is distributed among the components of PEs and architecture instances; (2) how does the design area scales if architecture resources, such as registers, are inserted or removed; and (3) how can architecture instances be sorted according to their size or complexity. Experimental results demonstrate that, when analysing these three issues, the estimated hardware complexity can be used as a metric in place of area units, which is usually available only after synthesis. That implies that CGADL-based software tool can replace commercial synthesis tools earlier at the design phase.

### 6.1.1. Experimental set up

The experiments in this section concern the design of the PE *Bianca*, which was described in Sections 4.2.2 and 4.3.3. The PE *Bianca* was chosen because it represents a typical processing element, as usually found in commercial and academic CGRAs. This PE was described twice: the first description used CGADL and can be partially seen in Figure 4.16, section 4.2.2, along with a schematic diagram of the PE. The second description used the hardware description language Verilog. This description is not included in this thesis for sake of clarity and space. Both descriptions corresponded to exactly the same PE template, and its composition was discussed in details in Section 4.3.3.

Design costs for the PE *Bianca* were estimated using CGADL and Verilog software tool flows, as follows (depicted also as a flow in Figure 6.1):

1. The CGADL description was input to the software program presented in Section 4.3.4, which produces estimation functions for the hardware complexity. This program parses a CGADL description of the PE *Bianca*, analyzes the PE composition as well as its FUs, and outputs estimation functions for the hardware complexity in form of Matlab [41] scripts. Hardware complexity estimations for a particular PE instance, expressed in gate equivalence, are then obtained by evaluating these scripts for a corresponding set of parameter values within the Matlab environment.

2. The Verilog description was synthesized with a commercial tool. Synthesis results included a detailed estimation of the circuit implementation area, expressed in $\mu m^2$. During synthesis, the commercial software tool was configured to produce area optimal results under a time constraint of $5ns$ for all combinational datapaths within one PE.

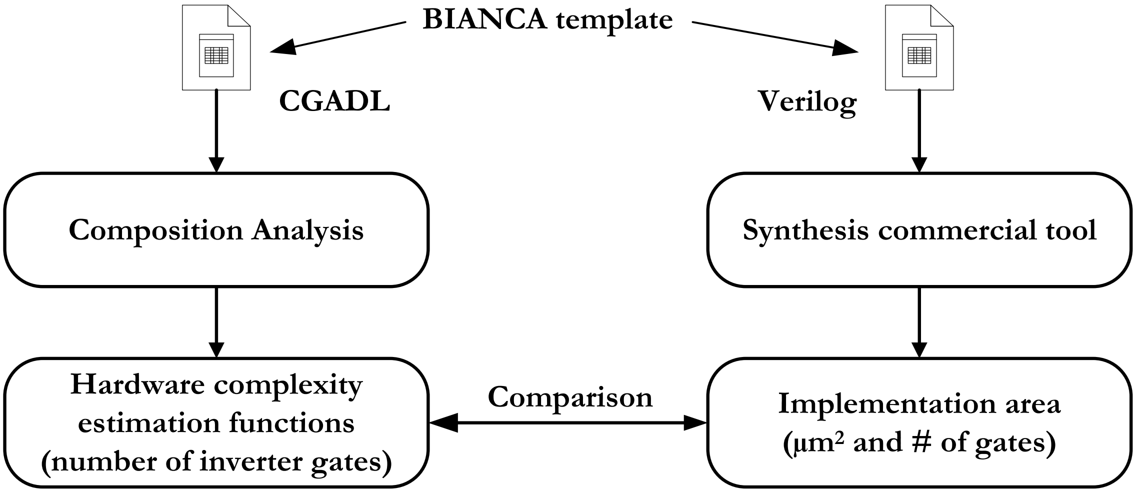

BIANCA template
CGADL
Verilog
Composition Analysis
Synthesis commercial tool
Hardware complexity estimation functions (number of inverter gates)
Comparison
Implementation area (µm² and # of gates)

Figure 6.1.: Generation flow of the data compared in this section.

Both software tools produce estimations for the implementation costs of the PE *Bianca*: the CGADL-based tool expresses these costs in gate equivalence, and the commercial tool in area units. The design of the PE *Bianca* is investigated by comparing the produced results in view of three questions:

**Composition** how is the design area distributed among the components of the PE?

**Scalability** how does this area scale as a function of the parameter values used to describe the PE instance?

**Ordering** how can PE instances be sorted based on their implementation costs?

Each one of these aspects is discussed in the following.

### 6.1.2. Analysis of PE datapath composition

Architecture designers need to determine which components of the datapath dominate the circuit area before they can improve the design. These components are prioritary targets for optimizations and refinements because reducing them may have a more significative impact on the total implementation area. To identify such components, the designer can use an analysis of the datapath composition.

The analysis of the datapath composition lists the portion of area occupied by each component relative to the total area of the design. Figure 6.2 depicts the datapath composition of the PE *Bianca* for datapath widths from 8 (upper row) to 32 bits (lower row). Each row compares the distribution of hardware costs obtained from estimation and from synthesis. Components of the architecture are grouped in data input and output multiplexers, flag input and output multiplexers, data and flag register banks and the FU. The third column depicts the error of the estimation method for each component group of the datapath. A register bank with 16 registers was used for all cases.

It can be observed that the FU, data register banks, and data output multiplexers sum up to more than 75% of the circuit area for all the considered datapath widths. The longer the datapath width, the smaller is the area portion used by flag components (registers and multiplexers) because their circuit is the same at all instances.

The composition analysis based on the CGADL description corresponded well to the analysis obtained with synthesis results of the Verilog model. The differences between estimation and synthesis results were typically under 2.5%, and remained under 5% for all the cases. The estimation tool overestimated the FU portion in about 2.5% in each case. This effect will be explained later in this section. This analysis is valid for all considered datapath widths. In addition to identifying area critical resources, the analysis of a PE's datapath composition helps to determine how hardware costs are redistributed if resources, such as registers or contexts are inserted or removed. For example, modifying the number of registers in a PE is likely to affect the size of the register bank and the size of multiplexers. Figure 6.3 depicts the composition of a 16-bit datapath of the PE *Bianca* with 12, 24 and 36 registers. Each row compares the distribution of hardware costs obtained by the CGADL-based estimation tool and from synthesis of the Verilog description. The third column depicts the error of the estimation method for each component group of the datapath.

The area portion of register banks increases for increasing number of registers, whereas the FU portion decreases. The datapath composition obtained by estimating the hardware complexity of the CGADL description corresponds well to that obtained by synthesis of the Verilog description: errors are typically under $2.5\%$.

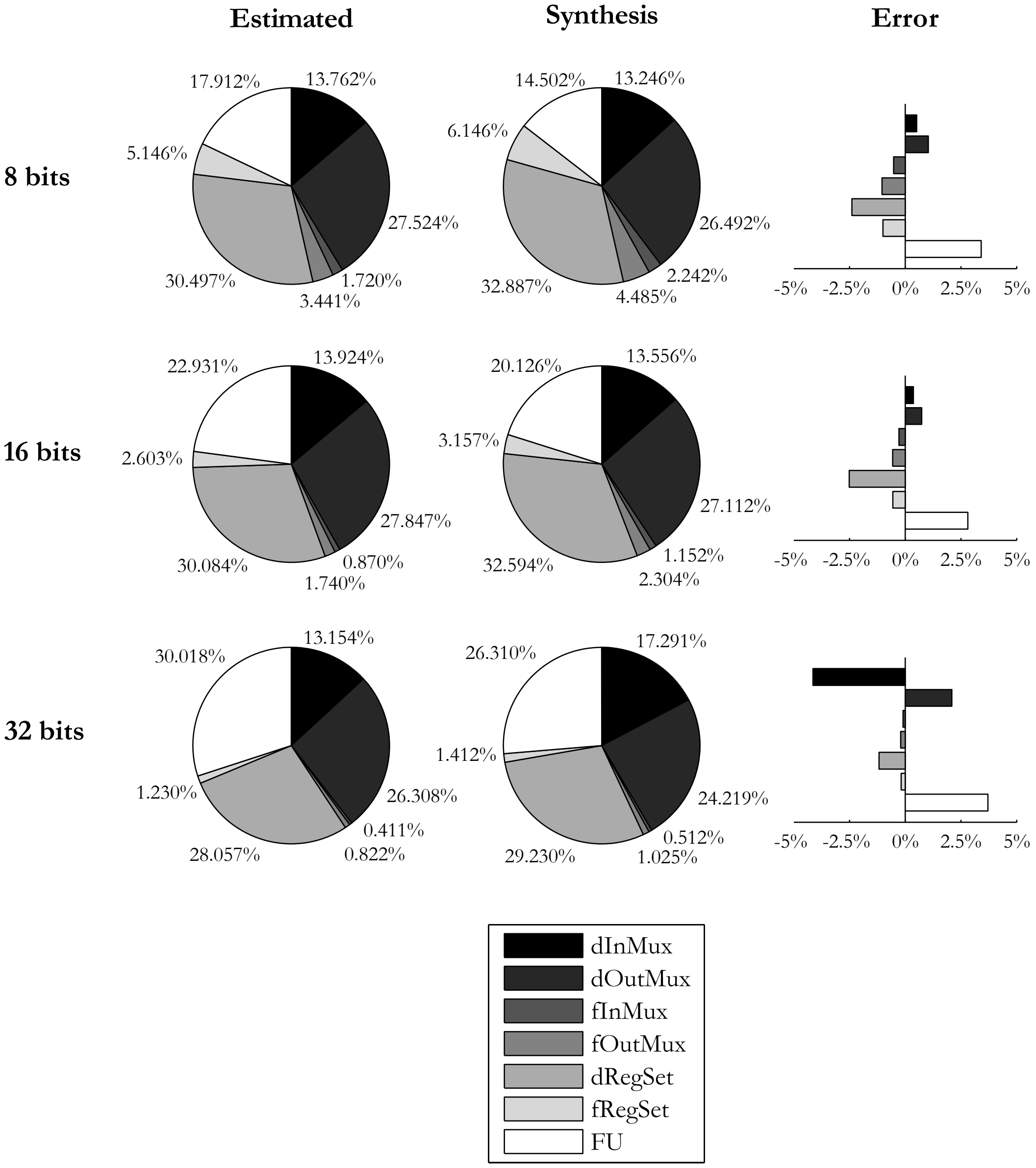


Figure 6.2.: Distribution of hardware costs for the PE *Bianca* when varying the datapath width. Results from the CGADL-based estimation tool and synthesis of the Verilog description. Rows compare PEs with datapath width of 8, 16 and 32 bits (top to bottom). Differences between the proposed estimation method and commercial synthesis tools remain under $5\%$.

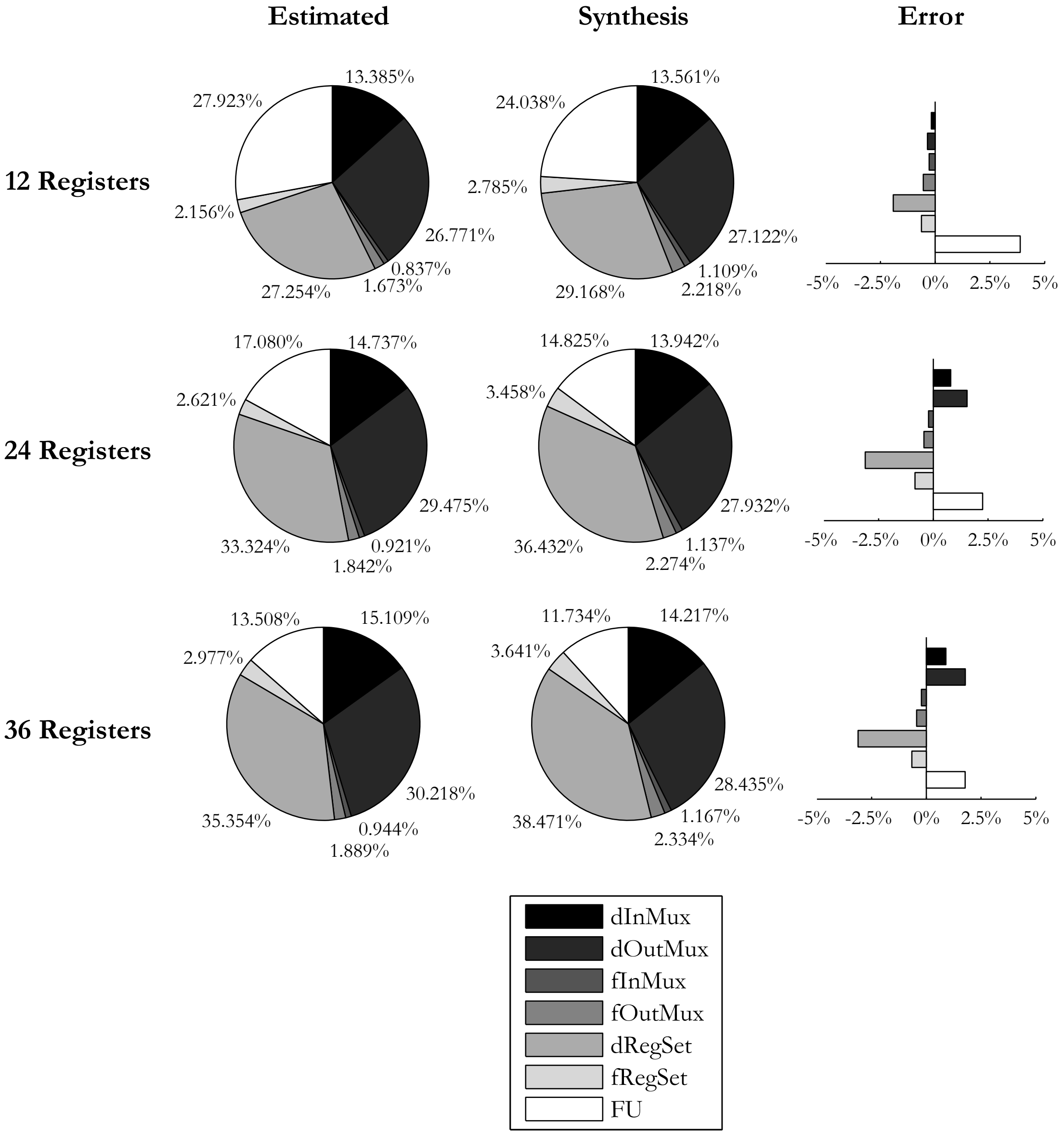


Figure 6.3.: Distribution of hardware costs of the PE *Bianca* when varying the number of registers in the register set. Results from the CGADL-based estimation tool and synthesis of the Verilog description. Rows compare PEs with 12, 24 and 36 registers in the register set (top to bottom). Differences between the proposed estimation method and commercial synthesis tools remain under $5\%$.

The analysis of the datapath composition also applies to sub-components of a PE, such as the FU. Figure 6.4 depicts the composition of the FU in the PE *Bianca* for datapath widths from 8 (upper row) to 32 bits (lower row). Each row compares the distribution of hardware costs obtained by the CGADL-based estimation tool and from synthesis of the Verilog description. The FU is composed of a selection multiplexer, adder/subtracter, a comparator, a multiplier, a shifter, and glue-logic, which is used for decoding the instruction and controlling the datapath (denoted in

Figure 6.4 as *Others*). The third column depicts the difference between area portions (relative to the total area) of each component of the FU.

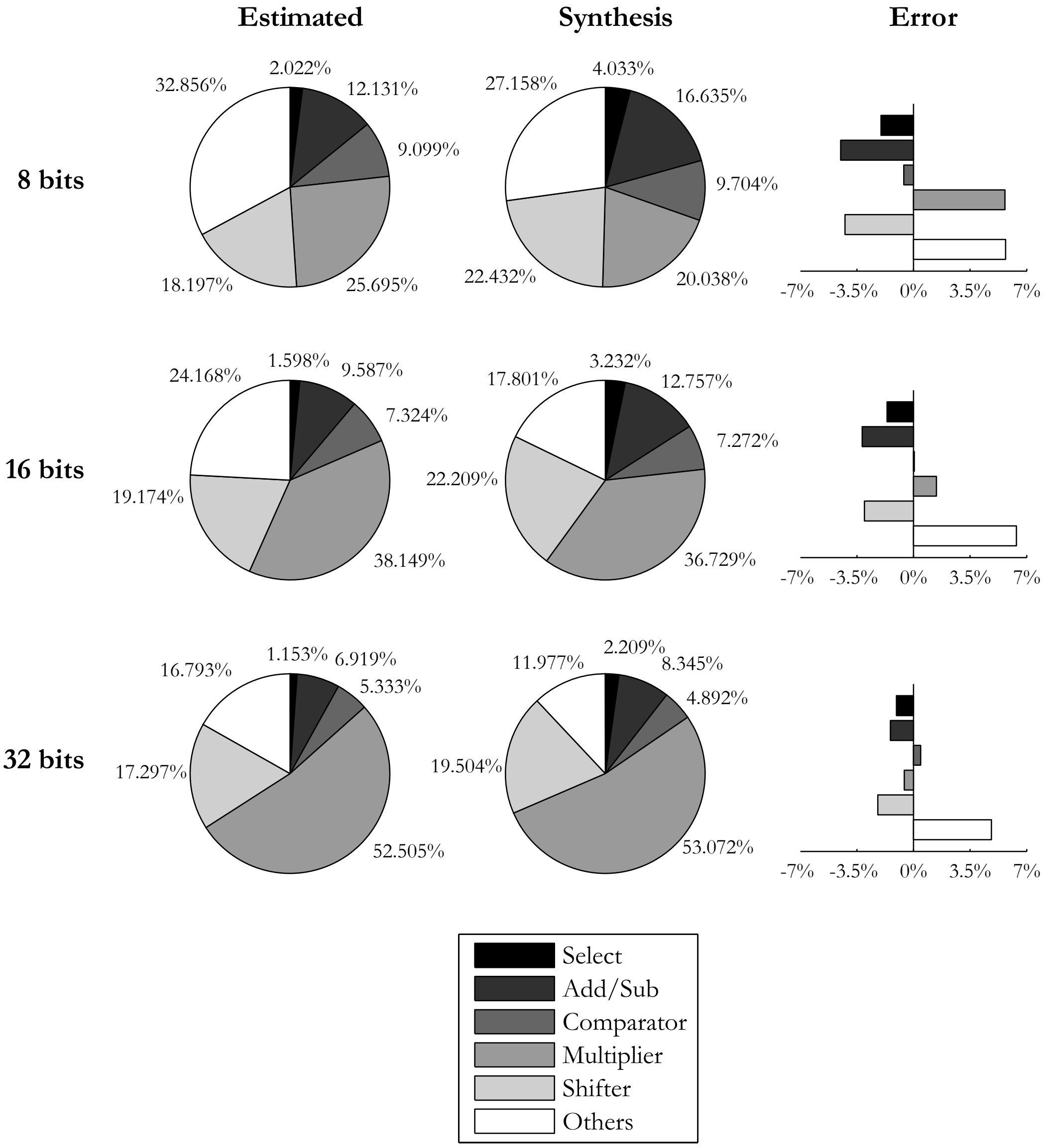


Figure 6.4.: Distribution of hardware costs for the FU when varying the datapath. Results from the CGADL-based estimation tool and synthesis of the Verilog description. Rows compare FUs with datapath width from 8 (upper row) to 32 bits(lower row). Differences between the proposed estimation method and commercial synthesis tools remain under 7%.

The multiplier uses about 25% of the implementation area on the 8-bits FU, whereas this portion increases to about 52% on the 32-bits FU. Concurrently, the area necessary for glue-logic circuits decreases as the width of the datapath increases because this circuit is not part of the FU datapath.

The difference between the area portion of each FU component obtained from the estimation results and that obtained from synthesis remains under 5%. The only exception is the area portion of glue logic circuits, which presented a difference of about $7\%$. Glue-logic circuits existent at the library of element models (see Section 4.3.1) are composed of decoders and multiplexers, whereas the description in Verilog uses a `switch-case` block. Apparently, the synthesis tool implements these `switch-case` blocks as lookup table based circuits, which have a lower cost than full decoders and multiplexers.

### 6.1.3. Scalability of the model

Designers of CGRAs need to determine how the complexity of PE's elements, of PEs, and of architecture templates scales as a function of the parameters of the model. For example, how does the area of the FU increases as bits are added to the datapath width? A CGADL description contains the necessary information to carry out this analysis early in the design phase. Experimental results demonstrate that the hardware complexity, estimated from the CGADL description, scales proportionaly to the implementation area obtained from synthesis of the Verilog description. This proportional relationship is shown in two examples: the hardware costs of the multiplexers and the hardware costs of the PE *Bianca*.

1. The size of output multiplexers (i.e. output ports) and input multiplexers depends on the number of registers in the PE and the datapath width. For each new register in the PE, one new input port is necessary in each multiplexer. The datapath width affects the number of bits in each input port of the multiplexer. Figure 6.5 depicts how the hardware complexity scales as new registers are inserted in the register set. Hardware complexity estimates, measured in inverter-equivalent gates, were obtained by using the CGADL-based tool and by synthezising of the Verilog description.

2. A similar comparison is shown in Figure 6.6, which depicts how the overall hardware complexity of the PE *Bianca* scale as a function of the number of registers used to build it. Results are depicted for PE instances with 8, 16 and 32-bits. Lines indicate results obtained from synthesis of the Verilog description, whereas markers indicate the costs obtained from the CGADL-based estimation tool.

The examples in Figure 6.5 and 6.6 demonstrate that the estimated hardware complexity obtained by the CGADL-based tool can be used in place of the results obtained from synthesis of the Verilog description.

### 6.1.4. Comparison of implementation area

It is not always clear which combination of parameter values leads to the smallest implementation area. For example, consider a PE instance with 16-bits datapath and 32 registers and a PE with 32-bits datapath and 8 registers. It is not trivial to decide which of these two instances has the smallest area because the width of the datapath and the number of registers affect the area in opposite trends. To solve this problem, it is necessary to directly compare the implementation area of the involved PE instances.

In the current experiment, instances of the PE *Bianca* with different datapath width and number of registers were sorted according to their implementation area, as depicted in Figure 6.7. Two area estimates were obtained per instance:

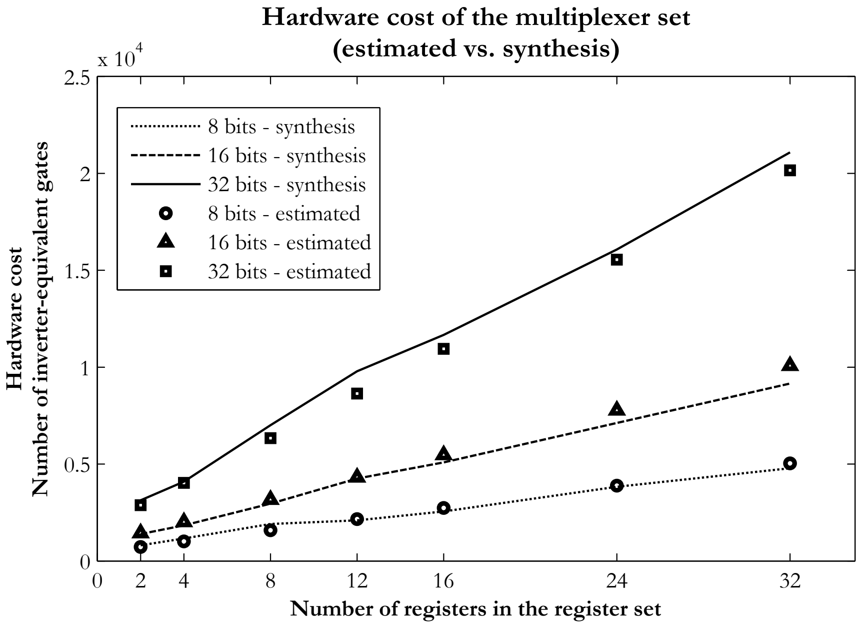


Figure 6.5.: Hardware complexity of the multiplexer set (PE *Bianca*) described as a function of the the register bank size. PE instances with 8, 16 and 32-bits datapath were considered. Discrete markers indicate results obtained by the CGADL-based estimation tool. Results from Verilog synthesis are presented as lines.

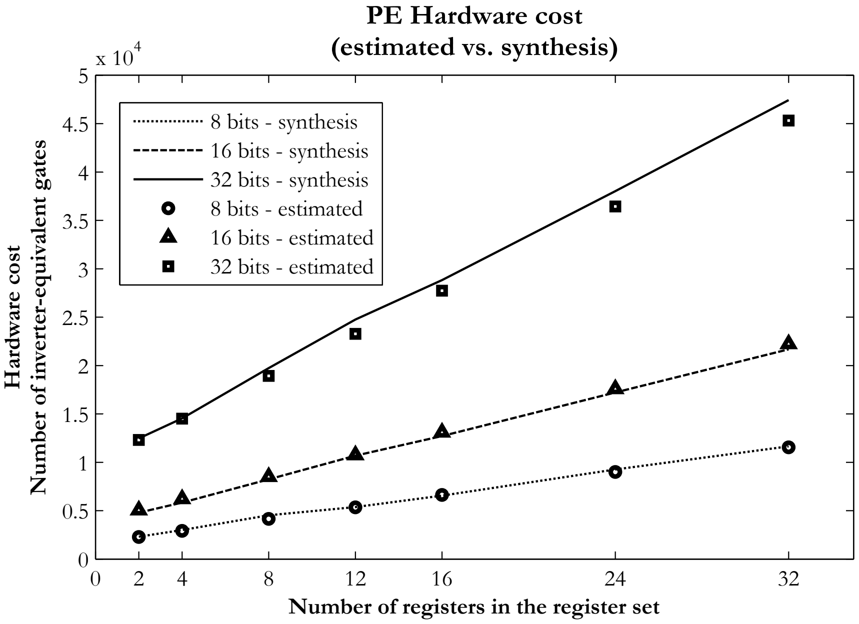


Figure 6.6.: Hardware complexity of the PE *Bianca* described as a function of the register-bank size. Results are depicted for PE instances with 8, 16 and 32-bits. Discrete markers indicate results obtained by the CGADL-based estimation tool. Results from Verilog synthesis are presented as lines.

- The first estimate was obtained by converting the hardware complexity results of the CGADL-based estimation tool, expressed in inverter-equivalent gates, to area units. During this conversion, the implementation area of one inverter gate was assumed to have an area of $2.54\mu\text{m}^2$. This value corresponds to the average area of the inverter gate in the $130n$m TSMC technology library used during synthesis.
- The second estimate was obtained directly from synthesis of the Verilog description.

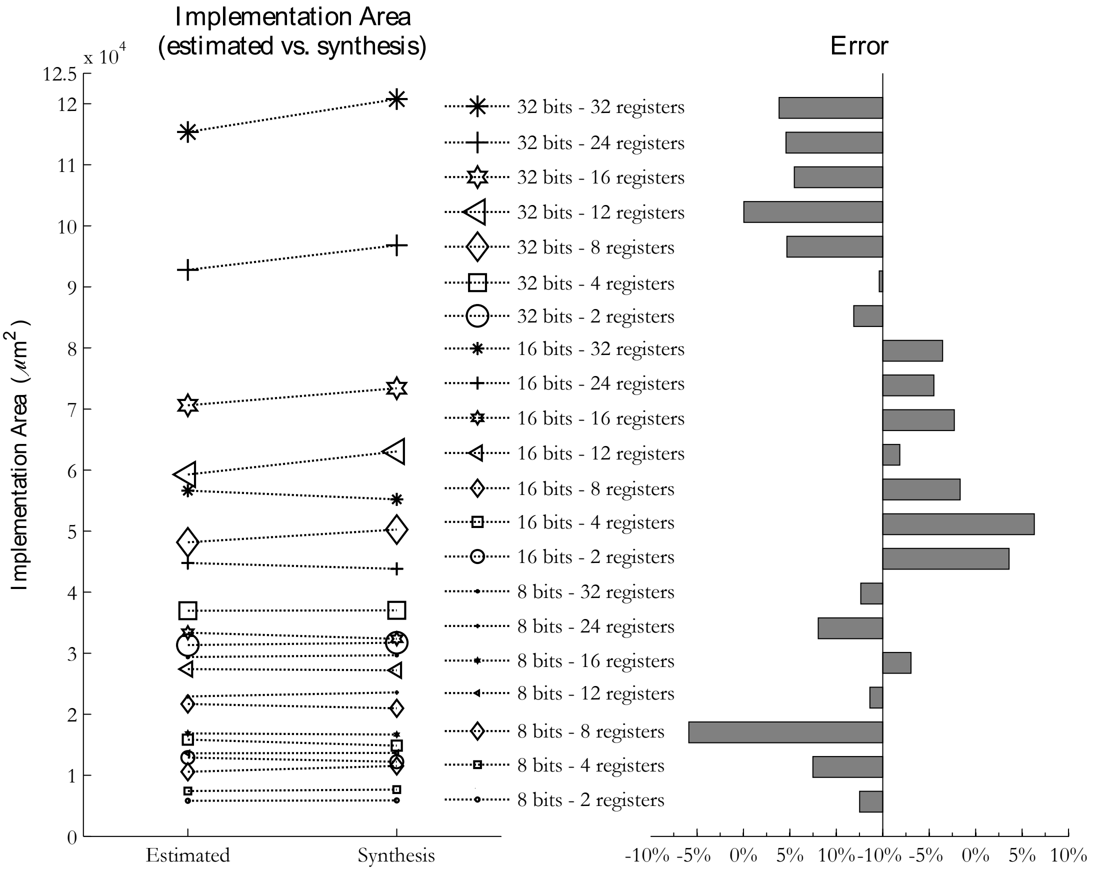


Figure 6.7.: Implementation area of different architecture instances (on the left): estimated and synthesis results agree upon the ordering of the instances. Relative error between estimation and synthesis results (on the right). Errors remained under 10%.

The ordering of architecture instances using results from the CGADL-based tool was the same as the one obtained using results from the Verilog synthesis. A similar ordering cannot be always guaranteed by the proposed estimation method, and inversions of order may happen. However, the presented results show that for typical cases the proposed method preserves the same order.

The graph on the right side of Figure 6.7 presents the relative difference between area values obtained by estimation and by synthesis for each of the architecture instances. The comparison between the area obtained by the CGADL tool and that obtained by synthesis showed a typical error between $5\%$ and $9\%$. This error can be larger if an element of the PE, such as the register set or FU, is considered because differences between the model of these elements and the synthesized circuit become more apparent. Therefore, a comparison of implementation area for internal elements of the PE is less reliable. Moreover, the conversion between gate-equivalence and area units may lead to different results depending on the process technology and gate library adopted.

### 6.1.5. Discussion on the hardware costs estimation

The results presented here show that:

1. The hardware complexity distribution among the components of FUs, PEs, and architecture instances corresponds well to the distribution of the implementation area, obtained from synthesis results. The estimation of hardware costs from a CGADL description can replace results of commercial synthesis tools for analysis of the area composition.

2. The hardware complexity of architecture templates, of PEs, and of PE's components varies as a function of parameters of the model. This variation was the same for results obtained from the CGADL estimation tool for results by synthesis of the Verilog description. The estimation of hardware costs from a CGADL description can therefore replace synthesis results when evaluating the impact of a parameter over the architecture area.

3. The ordering of PEs and architecture instances was preserved when sorting them by their hardware complexity, obtained from the CGADL-based tool, or by their implementation area, obtained by synthesis of the Verilog description. In all the investigated cases, hardware complexity estimatives preserved the same relation between instances, as when the comparison was done using area units. Therefore, the proposed method may replace synthesis results when comparing the size (or complexity) of architecture instances with different parameter settings.

The proposed estimation method can be generalized to all designs (PEs or arrays) which are based on the elements present in CGADL: multiplexers, register sets, logic-arithmetic units, finite state machines, and context memories. This generalization is possible because the complexity model used to evaluate each element is generic and scalable. If custom elements are used, which cannot be partially modeled or composed by CGADL's elements, an external complexity model must be elaborated.

Assessing hardware costs from a CGADL description has a number of advantages over other methodologies that extract results after synthesis. These advantages are listed in table 6.1. The proposed method applies directly to a CGADL description, which is much simpler than a fully synthesizable description in Verilog or VHDL. As a consequence, the evaluation cycle can begin earlier in the development phase. Gate equivalence results are also technology independent, and may be more adequate if the underlying implementation technology is not yet defined. The CGADL based estimation is also much faster: each instance could be completely evaluated in less than one second. On the other hand, a complete synthesis cycle took in average 660s (11 minutes) for each architecture instance. That can be explained by the fact that many other tasks are involved in the synthesis process, which delivers detailed information for the area usage, signal propagation timing, and power consumption, besides the netlist of the circuit.

Table 6.1.: Comparison between CGADL-based estimation and synthesis-based hardware cost analysis.

| CGADL-based estimation | Synthesis-based estimation |
|---|---|
| **Early in design phase** — only CGADL description necessary. | **Late in design phase** — full, synthesizable description necessary. |
| **Technology independent** | **Technology dependent** |
| **Fast estimation** — less than 1s per instance (Pentium 4, 2.6Ghz, 2Gb). | **Slow estimation** — about 660s per instance, for a complete synthesis cycle (Pentium 4, 2.6Ghz, 2Gb). |
| **No extra ouput** — hardware complexity estimates only. | **Extra output info** — area usage, timing, power consumption, netlist. |

## 6.2. The impact of custom instruction sets on CGRAs

The experiments in this section investigate the impact of custom instructions that are based on instruction patterns on the design of coarse grained architectures. The design methodology proposed in Chapter 5 is applied in two real-world applications and considers constraints. The first experiment concerns the design of a custom coarse grained array that supports the *Orthogonal frequency-division multiple access modulation scheme* (OFDMA) of the WiMax[123] standard. The second experiment concerns the design of a CGRA for a set of diverse computer vision applications, which are commonly used in driving assistance systems.

The main contributions of these experiments are:

- They provide a detailed evaluation on how the use of custom instructions affect the area, performance, power consumption, and composition of coarse grained architectures, their PEs and FUs. Both experiments show that embedding custom instructions in the PEs of coarse grained arrays
  - augments the complexity, area, and power consumption of individual PEs and FUs, but reduces the complexity, area, and power consumption of the complete array.
  - augments the efficiency of architecture resources, such as context memories, control units, and routing components during the execution of applications: custom arrays need less configuration bits to control operations and route data than CGRAs with non-custom PEs.
  - augments the efficiency of silicon area usage: custom arrays use less area per operation as their non-custom counterparts.
- They demonstrate that instruction patterns, extracted from the target applications, constitute an effective basis for the generation of custom instructions for coarse grained architectures.
- They show that the proposed design methodology keeps low the number of custom instructions, and maximizes their usage during the execution of the target applications.

- They assert the feasibility of this methodology: extraction, evaluation, and selection of instruction patterns, as well as their integration in the FUs as custom instructions is realized in a rational and effective way.

Both experiments follow the same general set up, as depicted in Figure 6.8. Their starting points were twofolds: (1) a set of applications, which characterized an application domain, and (2) an homogene, non-customized coarse grained array, called *initial array* and composed by PEs *Bianca* or a variant of it. The design and costs of the PE *Bianca* were analysed in the last section. Instruction patterns were extracted from the application domain and integrated to the instruction set of PEs in the *initial array* transforming it into into a *custom array*. Both, initial and custom arrays were, simulated, synthesized and evaluated regarding their composition, implementation area, performance, and power consumption.

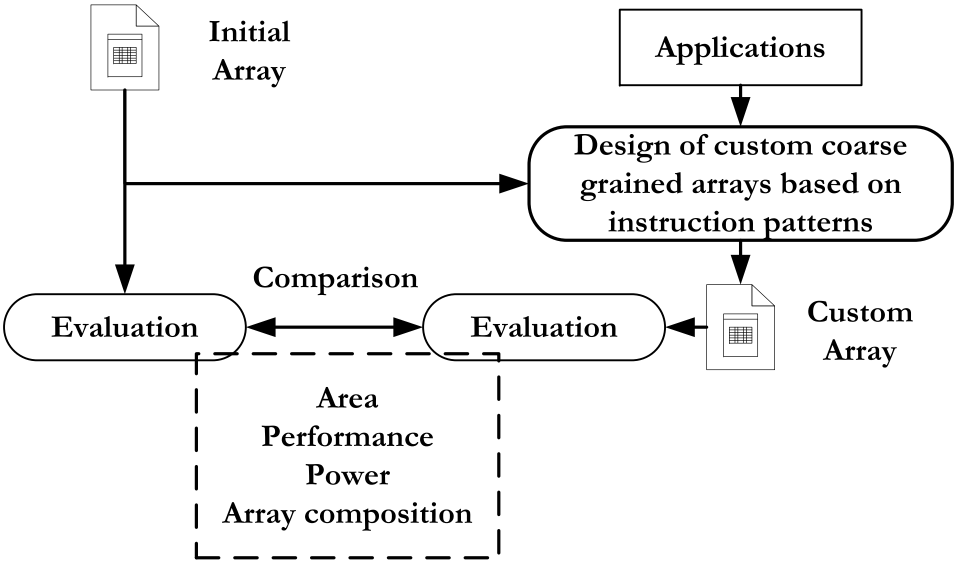


Figure 6.8.: Design flow for the experiments presented in this section. The design of custom coarse grained arrays is conducted according to the methodology proposed in Chapter 5.

An area analysis, similar to that carried out in the previous section (Section 6.1), could be used to evaluate the costs of both *initial* and *custom* arrays. However, evaluation results, used for comparison, were obtained using a commercial synthesis tool for two reasons:

- This allows a more accurate evaluation for the impact of using custom instructions to specialize CGRAs. Custom instructions mainly affect the functional unit complexity and area. Results discussed in Section 6.1 indicate that estimation method proposed in this work presents an error of 5%-7% when estimation the FU area. This error could be larger when considering custom instructions, because of the innacurate hardware model. To avoid this error, we use synthesis results.

- The time required to evaluate the results through synthesis did not constitute a hindrance in the following experiments. The advantage of the CGADL-based hardware cost estimation relies on avoiding the long synthesis cycles usually necessary during the design. However, these long cycles were not the case for the two experiments presented in this section.

## 6.2.1. Custom architecture for scalable OFDMA based systems

The scalable orthogonal frequency-division multiple access (OFDMA) is a digital modulation scheme used by leading wireless communication standards, such as the IEEE 802.16 Wireless MAN (also known as WiMax[123]) and the 3GPP Long Term Evolution standard (also known as LTE[47]). Scalable means that the bandwidth of each communication channel varies depending on traffic demand. For example, the physical layer of the WiMax standard allocates channels with bandwidths varying from 1.25 MHz to 20 MHz, depending on the number of users at a time point. The scalability in the OFDMA requires the underlying hardware architecture to support fast Fourier transforms (FFTs) with a high throughput rate (up to 480MSamples/s), implemented in a stream oriented way, and with a input vector size varying between 128 and 8196 points. In this first study case, the methodology proposed in Chapter 5 is used to obtain a custom architecture for this multipoint fast Fourier transform algorithm (FFT).

### The multipoint FFT algorithm

The equation for an $N$-points FFT is given by

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \text{ where } W_N = e^{-j2\pi/N}. \tag{6.1}$$

This equation can be factored by using a two dimensional map, as follows. Consider

$$\begin{aligned} N &= N_1 \times N_2, \\ n &= N_2 n_1 + n_2, \text{ where } \begin{cases} n_1 = 0, 1, 2, \ldots, N_1 - 1 \\ n_2 = 0, 1, 2, \ldots, N_2 - 1 \end{cases}, \text{ and} \\ k &= k_1 + N_1 k_2, \text{ where } \begin{cases} k_1 = 0, 1, 2, \ldots, N_1 - 1 \\ k_2 = 0, 1, 2, \ldots, N_2 - 1; \end{cases} \end{aligned}$$

then, equation 6.1 can be rewritten as

$$X(k_1 + N_1 k_2) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x(N_2 n_1 + n_2).W_N^{(N_2 n_1 + n_2)(k_1 + N_1 k_2)}$$

Expanding the twiddle factors ($W_N$) reduces the problem to

$$\begin{aligned} X(k_1 + N_1 k_2) &= \sum_{n_2=0}^{N_2-1} \{W_N^{n_2 k_1} [\sum_{n_1=0}^{N_1-1} x(N_2 n_1 + n_2) W_{N_1}^{n_1 k_1}]\} W_{N_2}^{n_2 k_2} \\ &= DFT_{n_2, N_2}[W_N^{n_2 k_1} DFT_{n_1, N_1}[x(N_2 n_1 + n_2)]]. \end{aligned} \tag{6.2}$$

For the results presented here, the equation 6.2 is considered for FFT problems with 128, 256, 512, 1024, 2048, 4096, and 8192 points, such that $N_1 = 8$ and $N_2 = \frac{N}{8}$. In these cases, the FFT problem can be partitioned in two parts or *Fields*, as follows:

$$X(k_1 + 8k_2) = \overbrace{DFT_{n_2,\frac{N}{8}}[W_N^{n_2k_1}\ \underbrace{DFT_{n_1,8}[x(\frac{N}{8}n_1 + n_2)]]}_{\text{Field A}}}^{\text{Field B}}$$

Field A diminishes the demand for data storage when $N$ is large. It solves $\frac{N}{8}$ 8-points FFTs, and divides the remaining computation task into 8 independent $\frac{N}{8}$-points FFTs, each of which is solved in one of the B fields.

The implementation of field A is a pipeline that solves the radix-8 FFT algorithm [72], whereas each field B implements the radix-$2^2$ Single-path Delay Feedback algorithm as proposed by [57]. The general FFT data flow graph (DFG) can be seen in Figure 6.9. At each clock cycle, the field A is fed with a set of 8 complex data values, corresponding to $\{x(n_2), x(\frac{N}{8} + n_2), x(\frac{1N}{8} + n_2), x(\frac{3N}{8} + n_2), x(\frac{4N}{8} + n_2), x(\frac{5N}{8} + n_2), x(\frac{6N}{8} + n_2), x(\frac{7N}{8} + n_2)\}$, where $n_2 = 1, 2, \ldots, \frac{N}{8} - 1$. Field A processes this data in a 6-stage pipeline and outputs 8 streams of data, each of which feeds a B field.

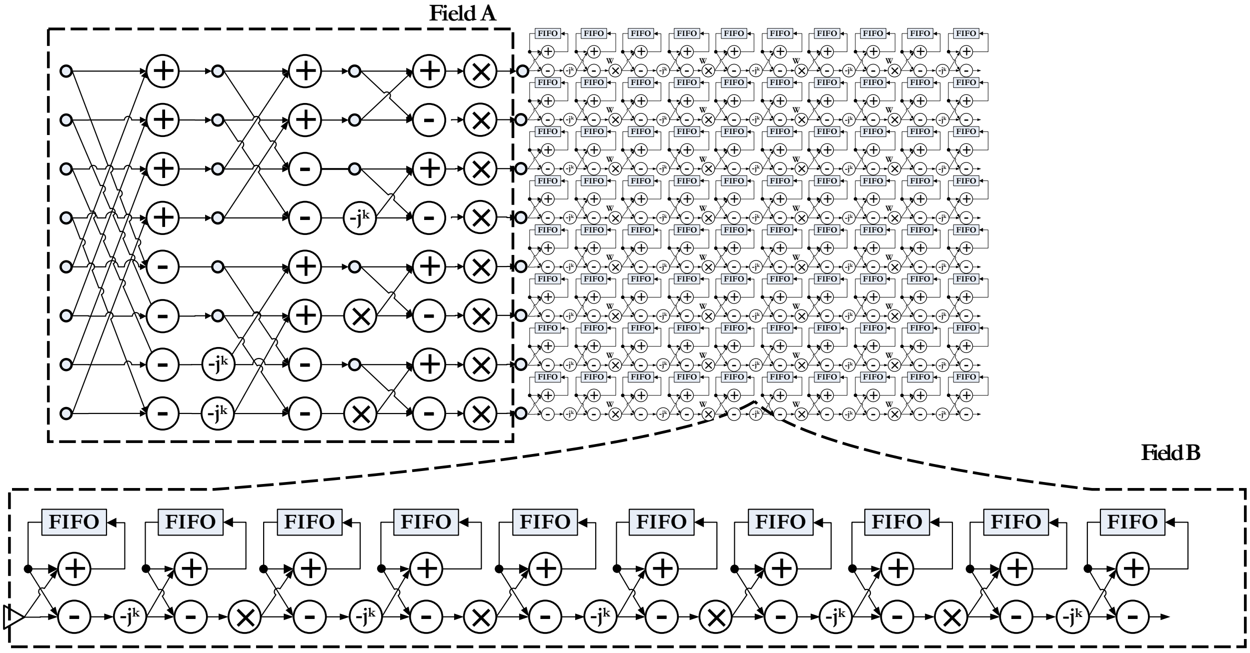


Figure 6.9.: Data flow graph of the multipoint FFT algorithm.

## Initial setup

The architecture instance in Figure 6.10 is the starting point for the experiments presented here. The number of PEs in this instance, their type distribution in the array, and the interconnect network correspond to operations and communication edges of the DFG in Figure 6.9. This architecture instance executes the algorithm in a pipelined way. Therefore, one PE is necessary for each operation in the DFG.

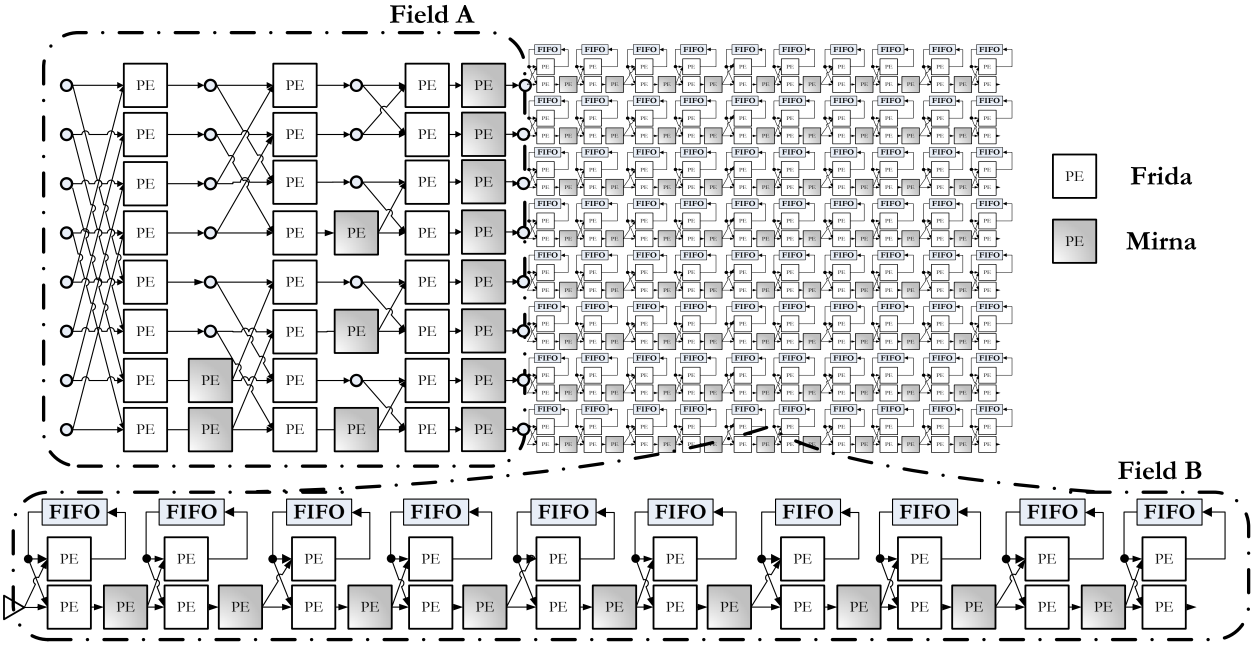


Figure 6.10.: Initial architecture instance – or *initial array* – adopted as reference for comparison results.

To compose the architecture array, two PE types, denoted as *Frida* and *Mirna*, were derived from the generic PE *Bianca*, detailed in Sections 4.2.2 and 4.3.3. The PEs *Frida* and *Mirna* have a similar composition as the PE *Bianca* except for their FU, which was simplified to contain only the necessary instructions for the FFT computation. The FU of PEs type *Frida* is composed of a 16-bits adder/subtracter unit with integer complex-arithmetic, as depicted in Figure 6.11a. Logical and comparison operations are not presents in the PEs because they are not necessary for the FFT. PEs of type *Mirna* have 4 simple 8-bits integer multipliers, which together with the adder/subtracter unit yields a complex-arithmetic multiplier, as depicted in Figure 6.11b. The design chosen for *Frida*'s and *Mirna*'s FUs allows this work to evaluate the area costs introduced by inserting custom instructions in a more fairly basis, because the FU excludes non-necessary instructions.

The B fields in the architecture instance are attached to a bank of FIFOs. These memory modules are not considered as part of the array because they are not altered in any way after the PEs's specialization. The context memory of each PE has 8 contexts, and their finite state machine 8 states. These contexts and states allow to implement different FFTs with input vector sizes (through reconfiguration) and are suitable for implementing the algorithm in field B. In the following, the architecture instance in Figure 6.10 will be referred to as the *initial array*.

### Running the experiment

Operation clusters were extracted from the FFT's DFG by applying the method introduced in Section 5.1.1 with the following set up constraints:

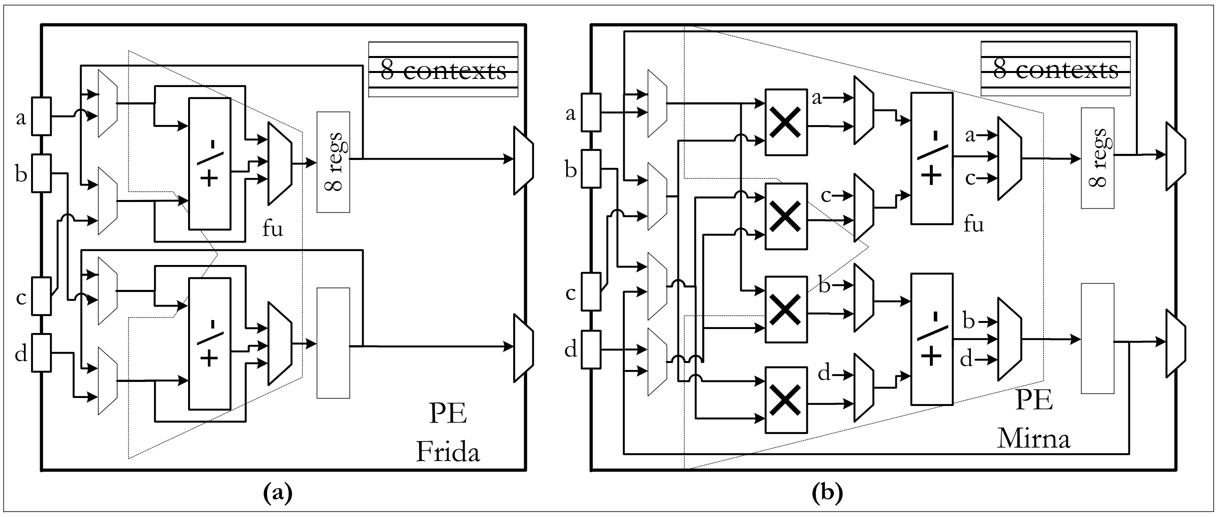


Figure 6.11.: PEs used in the initial architecture instance. Context memory, finite state machine and control signal lines are implemented as in the PE *Bianca*, presented in Figure 4.16.

- An operation cluster can process at most 2 complex-arithmetic values. That leads to custom instructions with up to 4 simple 16-bits input ports;
- An operation cluster can produce at most 2 complex-arithmetic values. That leads to custom instructions with up to 4 simple 16-bits output ports;
- An operation cluster can contain up to 3 atomic operations;
- The execution delay of an operation cluster must take no longer than 5ns. That leads to custom instructions that can be executed within this interval.

A total of 61 valid operation clusters were obtained from the DFG of field A, and 37 other operation clusters were obtained from the DFG of field B[1].

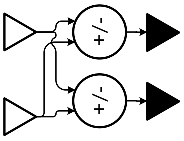
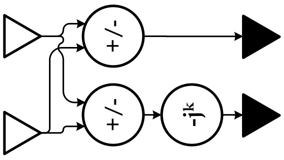
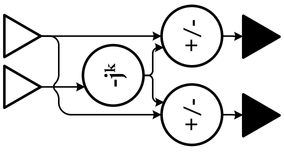
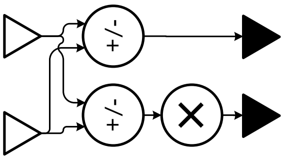
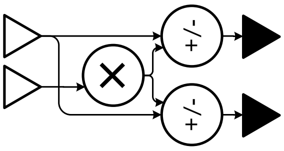
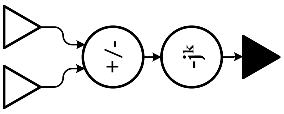
The extracted operation clusters were grouped in 11 instruction patterns by the procedure in Section 5.1.2. Table 6.2 presents six of these instruction patterns ordered according to their individual cover on the entire DFG (cover in field A plus 8 times the cover of field B).

The instruction pattern $IP_1$ corresponds to the so-called FFT *butterfly*. This pattern can be found frequently in the DFGs of fields A and B and covers up to $68\%$ of the whole application. Patterns $IP_2$ and $IP_3$ extend this *butterfly* by multiplying its output ($IP_2$) or its input ($IP_3$) by the term $((-j)^k)$. Each of these instruction patterns cover about $48\%$ of the application.

An instruction pattern set $IP_{FFT} = \{IP_1, IP_2\}$ has a small size, but covers up to $84.38\%$ of the FFT application due to the regular and repetitive structure of the FFT DFG. This instruction pattern set was selected to compose the custom instructions for this experiment. A common instruction datapath was composed by applying the composition method explained in Section 5.1.3. Multipliers were not used in this datapath because the multiplication term $((-j)^k)$ is always one of the complex numbers $1, -j, -1,$ or $j$. Instead, this multiplication term was implemented as a

[1] Operation clusters with only one operation node were not included in this total.

Table 6.2.: Feasible instruction patterns of the FFT's DFG. The number of operations clusters, represented by a given instruction pattern, and the cover of each instruction pattern are indicated separately for field A and field B. The instruction patterns were ordered according to their individual cover considering the entire FFT DFG(cover in field A plus 8 times the cover of field B)

| Pattern | | | Number of operation clusters | Cover | Total Cover (% of the application) |
|---|---|---|---|---|---|
| $IP_1$ |  | Field A | 12 | 24 | 184 (68.40%) |
| | | Field B | 10 | 20 | |
| $IP_2$ |  | Field A | 3 | 9 | 129 (47.95%) |
| | | Field B | 5 | 15 | |
| $IP_3$ |  | Field A | 3 | 9 | 129 (47.95%) |
| | | Field B | 5 | 15 | |
| $IP_4$ |  | Field A | 6 | 15 | 111 (41.30%) |
| | | Field B | 4 | 12 | |
| $IP_5$ |  | Field A | 2 | 6 | 102 (37.91%) |
| | | Field B | 4 | 12 | |
| $IP_6$ |  | Field A | 3 | 6 | 86 (31.90%) |
| | | Field B | 5 | 10 | |
| Others | | Field A | 72 | n.a. | Maximal 31.0% |
| | | Field B | 4 | n.a. | |

FFT *butterfly* where one of the output values shifts its real and imaginary terms and their signs depending on the value $k$.

By using the methodology introduced in Section 5.2, the new datapath was embedded as custom instructions in the FU of the PE *Frida*, explained earlier in this Section. This custom version of the PE *Frida* is called *Fiji*, and its layout can be seen depicted in Figure 6.12. The PE type *Fiji* have the same control path, routing resources, and register banks of the PE type *Frida*, but different FU and context memory. These last two elements were modified to support the custom instructions for patterns $IP_1$ and $IP_2$.

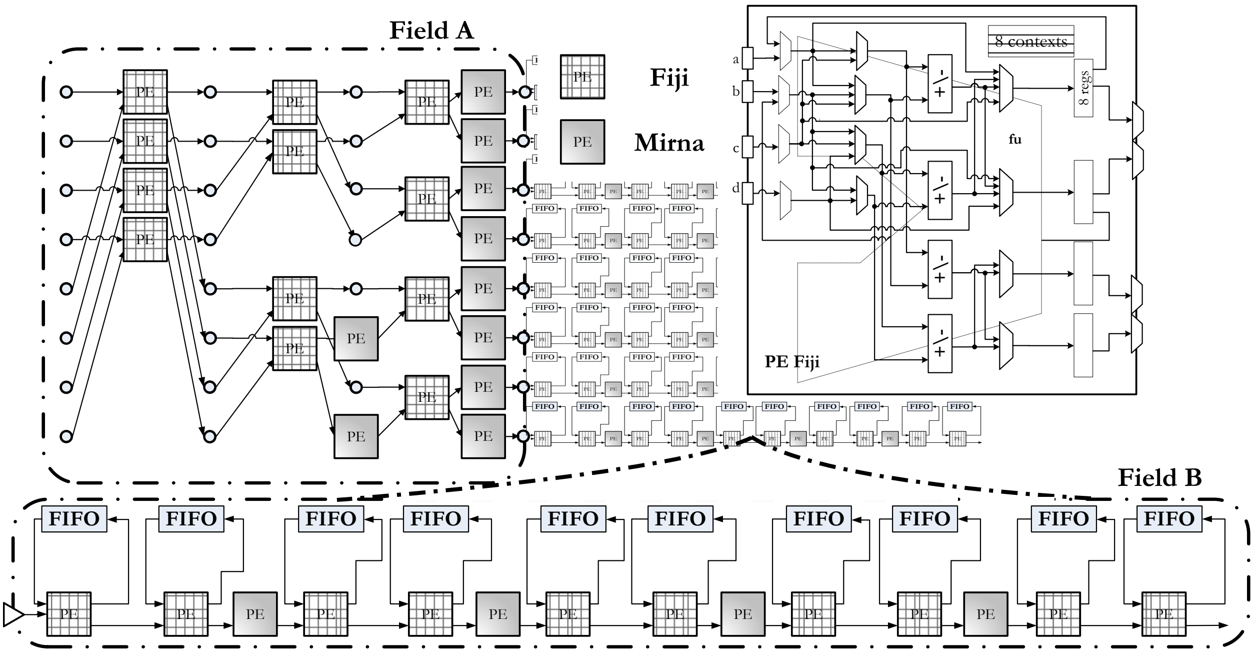


Figure 6.12.: Architecture instance composed with custom PEs *Fiji*. In detail: the PE type *Fiji*, which embeds custom instructions based on the instruction patterns $IP_1$ and $IP_2$.

The architecture presented in Figure 6.10 was refined in the following way. Every pair of PEs *Frida* that was connected in a *butterfly* structure was replaced by one PE of the type *Fiji*. Also, PEs carrying out the $(-j)^k$ multiplication term were suppressed. This new architecture instance has a total of 134 PEs organized as depicted in Figure 6.12. In the following, this architecture will be referred to as the *custom array*.

### Impact on composition and implementation area of singular PEs

The composition and implementation area of the PEs *Frida* and *Fiji* are compared in this section. The extra costs introduced by the custom instructions can be determined by direct comparison of these two PEs, as depicted in table 6.3.

*Fiji* needs to incorporate 2 extra adders/subtracters, 2 extra output multiplexers and 4 multiplexers for internal data routing. This overhead corresponds to the hardware of the custom instructions that were integrated. The resulting implementation area of *Fiji*'s FU is about 3 times that of *Frida*'s FU. *Frida* uses 15 configuration bits, whereas *Fiji* needs 23. The 8 extra bits are used for instruction selection, as well as activation of routing and output multiplexers. Because of this, the area of *Fiji*'s context memory is about 71% larger than area of the context memory in *Frida*. The integration of custom instructions in the PE *Frida* increased its total area in about 37%. Apparently, adding custom instructions (even a small set) to the instruction set can significantly increase the size of an individual PE.

Table 6.3.: FU composition and implementation area of the PEs *Frida* and *Fiji*. *Fiji* has the same design as *Frida*, except for its FU and context memory.

| | FU composition | FU Area ($\mu m^2$) | Configuration bits | Context Memory Area ($\mu m^2$) | Total PE area ($\mu m^2$) |
|---|---|---|---|---|---|
| *Frida* | 2 adders/subtracters<br>2 output multiplexers<br>1 decoder | 2279.60 | 15 | 3323.50 | 21796.31 |
| *Fiji* | 4 routing multiplexers<br>4 adders/subtracters<br>4 output multiplexers<br>1 decoder | 7093.43 | 23 | 5698.17 | 29896.30 |
| *Mirna* | 4 routing multiplexers<br>2 adders/subtracters<br>2 output multiplexers<br>4 multipliers<br>1 decoder | 17119.97 | 16 | 3505.13 | 36752.10 |

The PE *Mirna* is about twice as large the PE *Fiji* because of its 4 multipliers, so replacing PEs *Mirna* with PEs *Fiji* reduce the overall area of the architecture. This replacement took place during the architecture refinement, and followed the discussion presented in Section 5.2.3. Specially the `B` fields (see last section) were affected: 5 from the 9 PEs *Mirna* were replaced by PEs *Fiji* in each field `B`. On the other hand, PEs *Fiji* have more configuration bits and a larger context memory area than *Mirna* which indicates that its control path is more complex. The use of custom instructions explain this extra complexity.

### Impact on architecture composition, area, and power consumption

The impact of custom instructions must also be investigated at architectural level. A comparison is presented here for the composition, implementation area, and power consumption of two architecture instances: the *initial array* (see Figure 6.10) and the *custom array* (see Figure 6.12).

The composition of both architecture instances can be seen in table 6.4. The *initial array* required the use of 269 PEs (184 PEs *Frida* and 85 PEs *Mirna*) without custom instructions for a completely pipelined execution of the FFT algorithm. On the other hand, the *custom array* requires, for this same task, a total of 134 PEs (92 PEs *Fiji* and 42 PEs *Mirna*). This corresponds to a reduction of 50% in the number of PEs.

The *custom array* uses larger but less PEs in its composition. So, what was the real gain in terms of implementation area? A comparison of the area usage of initial and custom arrays is depicted in Figure 6.13a. The *initial array* uses 7.1344$mm^2$ of silicon area, whereas its custom counterpart occupies 4.2940$mm^2$. That corresponds to an effective reduction of 39.81% in the implementation area of the architecture instance.

Table 6.4.: Composition of the *initial* and *custom* arrays.

| Instance | | PEs type *Frida* | PEs type *Mirna* | PEs type *Fiji* |
|---|---|---|---|---|
| Initial array | Field A (1x) | 24 | 13 | 0 |
| | Field B (8x) | 20 | 9 | 0 |
| **Total** | 269 PEs | 184 | 85 | 0 |
| Custom array | Field A (1x) | 0 | 10 | 12 |
| | Field B (8x) | 0 | 4 | 10 |
| **Total** | 134 PEs | 0 | 42 | 92 |

The reduction in area usage also implies a reduction of the static power (leakage power) and dynamic power, as depicted in Figure 6.13b and 6.13c. The *initial array* consumes up to $1.44mW$ leakage power, whereas for the *custom array* the estimation is $0.84mW$. That corresponds to a reduction of $41\%$ in the leakage power consumption. For the dynamic power estimation, a toggling probability of 50% was assumed for input bits of PEs. The dynamic power was estimated in $750mW$ for the *initial array* and in $534.42mW$ for the *custom array*. That corresponds to a reduction of $28\%$ in the dynamic power consumption.

### Discussion of experimental results

In the present experiment, both *custom* and *initial* arrays execute the same number and sequence of operations during the execution of the multipoint FFT algorithm. Both arrays have similar throughput performance: they consume 8 samples per clock cycle. However, the *custom array* required only half the number of PEs. That is explained by the fact that custom instructions inside a single PE *Fiji* can compute groups of operations that, on the *initial array*, can only be executed using 2 (butterfly pattern) or even 3 (butterfly scaled by $-j^k$) distinct PEs. Due to the reduced number of PEs, the *custom array* uses about $39\%$ less silicon area, and consumes up to $40\%$ less leakage power than its non-custom version.

In Chapter 5, this work defined and used instruction patterns to design and integrate custom instructions to coarse grained arrays. This experiment demonstrates the principle by which the proposed methodology improves the design of CGRAs: **A custom instruction based on instruction patterns executes a group of operations that, otherwise, would only be accomplished by using several individual PEs; PEs embedded with such custom instruction can be used to replace groups of non-custom PEs, decreasing the total number of necessary PEs in the array.** The experimental results show that custom instructions based on instruction patterns improve the efficiency with which resources in the architecture are used.

To demonstrate that custom arrays use resources of the architecture more efficiently, two efficiency metrics will be considered (see table 6.5): the number of configuration bits per operation, and the area per operation. The number of configuration bits per operation is a metric, often proposed in the literature (e.g. [75] and [120]), to assess the complexity of control resources in CGRAs. The larger the number of configuration bits per operation, the more complex is the control circuitry.

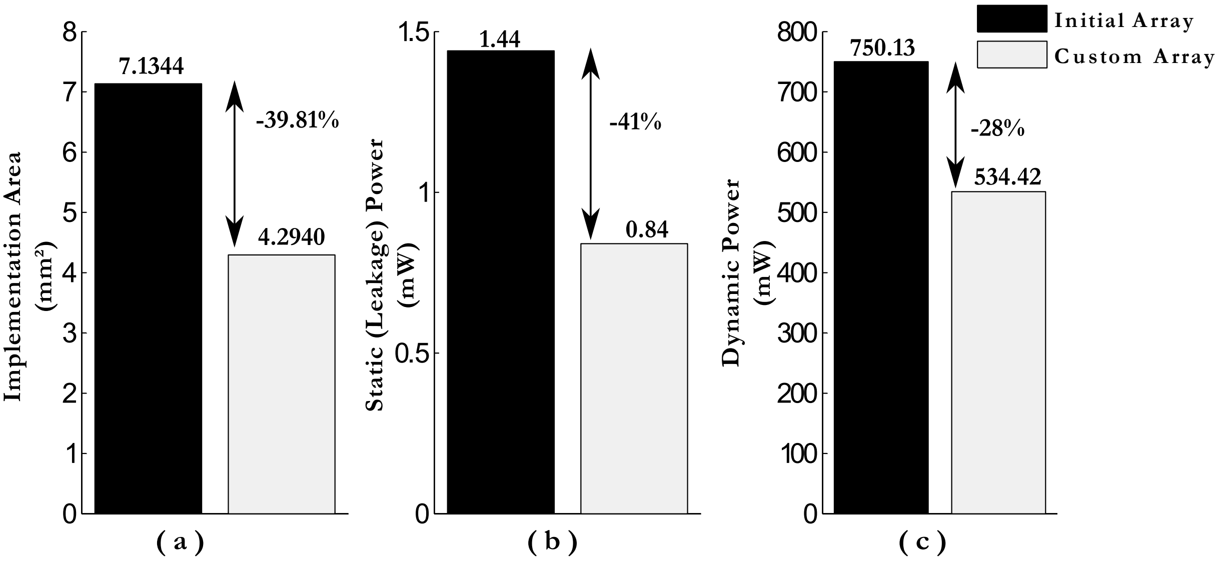


Figure 6.13.: Initial vs. Custom Array: estimations for silicon area (a), static power consumption (b), and dynamic power consumption (c). The array using custom PEs *Fiji* can be implemented in smaller area and consumes less power.

The *initial array* uses 4120 configurations bits per context (15 bits on each PE *Frida* and 16 bits on each PE *Mirna*.)[2], which corresponds to about 15.31 bits per operation. The *custom array* needs only 2788 bits per context (23 bits on each PE *Fiji* and 16 bits on each PE *Mirna*) [3], which leads to an efficiency of the control structures of 10.36 bits per operation. The *custom array* is more efficient because it uses fewer configuration bits to control the same amount of operations than the *initial array*.

Table 6.5.: Efficiency of the area usage for the *initial* and *custom* arrays. Both instances can execute up to 269 operations per context.

| Instance | Composition | Configuration bits per operation | Area per operation |
|---|---|---|---|
| Initial array | 184 PEs *Frida*<br>85 PEs *Mirna* | 15.31 | $26521\mu m^2$ |
| Custom array | 92 PEs *Fiji*<br>42 PEs *Mirna* | 10.36 | $15962\mu m^2$ |

Another efficiency measure is the silicon area per executed operation. The *initial array* uses $7.1344mm^2$ to execute 269 operations; that is, $26521\mu m^2$ per operation. This area usage is improved up to $15962\mu m^2$ per operation at the custom instance (269 operations within $4.2940mm^2$).

[2] See tables 6.3 and 6.4.

[3] See tables 6.3 and 6.4.

Table 6.6.: Comparison with state-of-art FFT designs

| | ASIC1<br>[73] | ASIC2<br>[72] | Altera<br>FPGAs [2] | Custom Array<br>(proposed) |
|---|---|---|---|---|
| Area ($mm^2$) | 1.25 | 3.09 | n.a. | 4.29 |
| Throughput (MSamples/s) | 160 | 1000 | 195 | 1600 |
| Power Consumption<br>($mW$/MSample) | 0.34 | 0.175 | n.a. | 0.33 @ 8196 points<br>0.26 @ 1024 points<br>0.16 @ 128 points |
| Clock Frequency (MHz) | 250 | 250 | 220 | 200 |
| Technology | 180$nm$ | 180$nm$ | Stratix III | 130$nm$ |
| Input Vector Size | 1024 | 128 | 256,1024,4096 | 256-8196 |

The *custom array* is more efficient because it uses a smaller implementation area than the *initial array* to realize the same work.

The FFT experiment presented in this section had as starting point an arbitrary coarse grained array (the *initial array*), which was refined up to a *custom array* by applying the method proposed in Chapter 5. When comparing initial and custom arrays, experimental results demonstrate a reduction in area and power consumption for the latter. However, the *custom array* must still be compared to similar state-of-art solutions. Therefore, a comparison between the *custom array* and state-of-art work will be presented in the following.

### Comparison with other state-of-art FFT solutions

Some works can be found in the literature that tackled the multi-point FFT problem recently. The design of Liu et. al. [73] proposes an array with few complex elements strongly connected among each other. It is optimized for a low memory fingerprint and requires a small area usage, but because of its sequential execution it only achieves 160 MSamples/s. Lin et. al. [72] uses the radix-$2^2$ Single-path Delay Feedback algorithm, which implements the B fields. They aim at a high throughput, stream-oriented design (up to 1 GSamples/s) for a 128-points FFT. These two designs, [73] and [72], have their input vector size fixed at design time. The third design, by Altera for FPGAs [2], is more flexible and can be customized for calculating 256-, 1024- or 4096-points FFTs. Altera's design achieves a throughput of around 200 MSamples/s. Unfortunately, no numbers for overall area and power consumption are available.

Table 6.6 depicts a comparison between the *custom array* and state-of-art FFT designs in terms of silicon area, throughput and power consumption. The coarse grained architecture proposed in this work has a larger silicon area usage, which provides a higher hardware parallelism. The size of this array is comparable to the size of the designs mentioned before[4]. The massive hardware parallelism of the *custom array* allows a throughput of 1.6 GSamples/s and the design accepts all input vectors between 128 and 8196 points.

Power consumption was obtained for the custom array from a commercial synthesis tool, and for the other platform from their respective literature sources. is equivalent to that of the state-of-art

[4] A comparison based on gate equivalence is not possible, because this data is not available in [73] and [72].

VLSI designs. The dynamic power consumption is dependent on the size of the FFT's input vector: $0.33mW$/MSample when 8196 points FFTs are calculated, and down to $0.16mW$/MSample when 128 points are considered. The small difference between these values and the ones obtained by [73] and [72] can be explained by the implementation technology rather than by the use of custom instructions.

### Critics to the FFT and motivation for a second experiment

The application set considered so far comprises FFTs with several input vector sizes. Operation clusters and instruction patterns extracted for this application originated from variations of the same DFG structure, which was scaled when necessary to accommodate new points in its input vector. This example particularly helped to investigate the impact of custom instructions when the target applications have regular and repetitive tasks.

However, the FFT application set could be considered atypical. The design of coarse grained architectures typically aims at several different applications, which do not necessarily use similar structures. The next example addresses an application set with several applications. This example will demonstrate that, even in such cases, embedding custom instructions in the PEs of a CGRA may reduce area and power consumption, and improve performance.

## 6.2.2. Computer vision for automotive applications

Reconfigurable architectures, including the coarse grained ones, are usually tailored to an application domain – a family of diverse programs which are used within a common context or to solve a more complex problem. The ability to address diverse applications should be considered and maintained by any methodology that aims at the design of coarse grained reconfigurable architectures. The second experiment presented in this work applies the design methodology proposed in Chapter 5 to coarse grained arrays that must address the execution of several applications.

The second experiment will deal with computer vision algorithms used in automotive driving assistance systems. Driving assistance systems integrate a growing number of functionalities, such as navigation, driving safety and visual recognition of traffic signs. Each one of these functions requires the system to make decisions within critical time constraints. For example, a traffic sign recognition system that captures images at 30 frames per second has to process and classify each video frame within 30ms. Typical image processing techniques in these systems are image filtering, edge detection and feature extraction. These techniques are computing intensive because driving assistance systems require the use of high resolution images.

### Application Domain – Computer vision for automotive driving assistance

Seven algorithms were selected as the application set of this experiment: general mask convolution, the Sobel and Prewitt gradient operators, the Hough transform for a circle, and the RGB-to-YIQ and the RGB-to-CMYQ conversion. These algorithms are used by several automotive driving assistance systems, such as traffic sign recognition softwares, person identification systems, and video based crash avoidance systems.

**Generic Mask Convolution [44]** This technique is employed when spatial filtering is needed (as opposed to frequency filtering using Fourier transform); for example Gaussian, high pass, and low pass filters. It consists in re-calculating the value of each pixel on the image as a weighted sum of its neighbor pixel values, that is:

$$p'(x,y) = \sum_{u=-m}^{m} \sum_{v=-m}^{m} M(u,v)p(x-u,y-v), \text{ where } M \text{ is a matrix of size } m \times m. \quad (6.3)$$

Spatial mask convolutions may turn into a intensive computing task, as the size $m$ of the mask grows. For the experiments presented here, a $m$ was set to 5.

**Sobel and Prewitt gradient operators [61]** Sobel and Prewitt are instances of edge detection filters, which are frequently used to identify strong local changes in an image. These operators calculate the horizontal and vertical gradients associated to a pixel based on its neighbors, as depicted in equations 6.4 and 6.5. In both cases the intensity of the target pixel is given by $\sqrt{G_h^2 + G_v^2}$.

$$\begin{aligned}
&\textbf{Sobel}\\
G_h =& p(x-1,y-1) + 2p(x-1,y) + p(x-1,y+1)\\
&- p(x+1,y-1) - p(x+1,y) - p(x+1,y+1)\\
G_v =& p(x-1,y-1) + 2p(x,y-1) + p(x+1,y-1)\\
&- p(x-1,y+1) - p(x,y+1) - p(x+1,y+1) \qquad (6.4)\\
&\textbf{Prewitt}\\
G_h =& - p(x-1,y-1) - p(x-1,y) - p(x-1,y+1)\\
&+ p(x+1,y-1) + p(x+1,y) + p(x+1,y+1)\\
G_v =& - p(x-1,y-1) - p(x,y-1) - p(x+1,y-1)\\
&+ p(x-1,y+1) + p(x,y+1) + p(x+1,y+1) \qquad (6.5)
\end{aligned}$$

**Hough transform for a circle** The Hough transform can be used to determine the parameters of a circle when a number of points that fall on the perimeter are known. A circle with radius $r$ and center $(a,b)$ can be described with the parametric equations $x = a + r\cos\theta$ and $y = b + r\sin\theta, 0^\circ \leq \theta \leq 360^\circ$. If an image contains many points, some of which fall on perimeters of circles, then the job of the search program is to find parameter triplets $(a,b,r)$ to describe each circle. The fact that the parameter space is 3D makes a direct implementation of the Hough technique expensive in terms of memory usage and time.

If the circles in an image are of known radius $R$, then the search can be reduced to 2D, as depicted in Figure 6.14. Most of the calculation work in this approach relies on drawing a circle in the parameter space for each non-blank pixel in the image. Because of this, the circle rasterization algorithm is used to customize the array in this experiment.

**RGB-to-YIQ conversion** During a RGB-to-YIQ conversion, the color information (red, green,

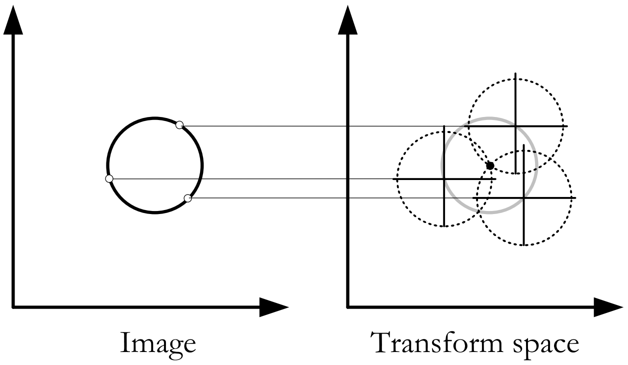


Figure 6.14.: Hough transformation for a circle with known radius $R$.The locus of $(a, b)$ points in the parameter space fall on a circle of radius $R$ centered at $(x, y)$. The true center point will be common to all parameter circles, and can be found with a Hough accumulation array.

and blue values) of each pixel in an image is transformed in a brightness value (Y) and two color difference values (I and Q), as depicted in equations 6.6. This conversion is used in for the transmission of digital color images and to keep compatibility between color and black-an-white images.

$$\begin{aligned} y =&((y_r r + y_g g + y_b b + (1 \ll (\text{scale} - 1))) \gg \text{scale} \\ i =&((i_r r + i_g g + i_b b + (1 \ll (\text{scale} - 1))) \gg \text{scale} \\ q =&((q_r r + q_g g + q_b b + (1 \ll (\text{scale} - 1))) \gg \text{scale} \end{aligned} \tag{6.6}$$

**RGB-to-CMYQ conversion** During a RGB-to-CMYQ conversion, the color information (red, green, and blue values) of each pixel in an image is transformed in values for cyan (C), magenta (M), and yellow (Y), as well as a key value for black: another color system. In contrast to the RGB-to-YIQ conversion, this algorithm requires also control structures to decide the key value for black.

### Initial Setup

The initial architecture instance for this experiment has 18 PEs type *Bianca* with 32-bits datapath, and organized in a 3×6 homogene array (see Figure 6.15). The PEs type *Bianca* were discussed in detail in Sections 4.2.2 (Figure 4.16) and 4.3.3. The number of PEs, the double connection nearest neighbor network, and the geometry of the initial architecture were found by exploring the design space according to [91]. For example, 17 PEs were necessary to map the mask convolution filter, which is the application with largest number of operations in one context. An extra PE (the 18th) was kept in the array to retain the rectangular geometry. Similarly, the double connection network (as opposed to one simple nearest neighbor network) is necessary to route data to/from the extra storage resources in the Prewitt application, which is the most communication intensive application. This architecture instance will be referenced as *initial array* in the remainder of this section.

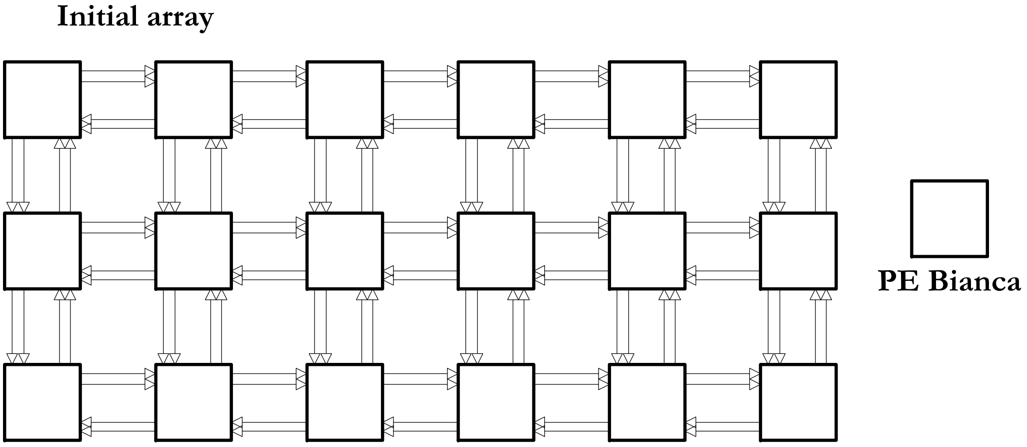


Figure 6.15.: Initial architecture, denoted *initial array*, for the computer vision experiment. This instance is composed of 18 (3×6) PEs of the type *Bianca*.

### Running the experiment

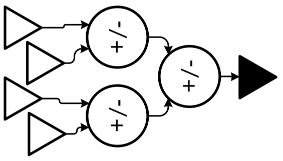
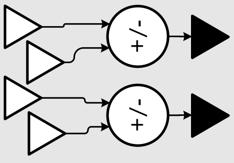
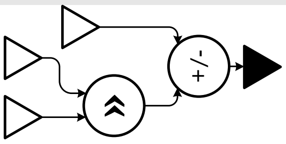
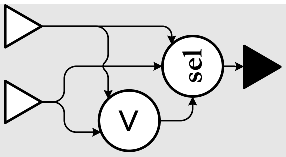
Operation clusters were extracted from the DFG of each application by applying the method introduced in Section 5.1.1. The following set up constraints were considered: clusters may process at most 4 32-bits input values, produce at most one output value, and use no more than 3 functional units. No timing constraints were imposed for the operation clusters or for the custom instructions derived from them. Storage operations, present in the Prewitt, Sobel, Mask, and RGB-to-CMYK applications, were not considered during the extraction. These operations use the register bank, instead of an FU, and cannot be transformed into a custom instruction by using the method proposed in this work.

The number of operation clusters varied depending on the application (see table 6.7). 39 operation clusters were extracted for the mask convolution, 14 for the Sobel operator, 33 for the Prewitt operator, 40 for the RGB-to-YIQ conversion, 11 for the circle rasterization of the Hough transform and 8 for the RGB-to-CMYK conversion algorithms.

In each case, the operation clusters could be grouped into a smaller number of instruction patterns (see table 6.7) by using the methodology introduced in Section 5.1.2. For example, the 39 operation clusters of the mask convolution were partitioned in only 7 different patterns. Similarly, the 40 operation clusters extracted from the RGB-to-YIQ correspond to only 8 different instruction patterns. That reinforces the idea that many application are mainly composed of groups of operations with the same execution pattern.

The cover of each instruction pattern was evaluated for each application and for the complete application set by using the method introduced in Section 5.1.2. Four patterns were selected to compose custom instruction datapaths according to the method proposed in Section 5.1.2: patterns $\text{IP}_A$, $\text{IP}_B$, $\text{IP}_C$, and $\text{IP}_D$, indicated in table 6.7. $\text{IP}_A$ and $\text{IP}_B$ have only adders in their structure, and this structure that can be found several times in almost all DFGs. For example, 60% of the operations in the Prewitt gradient calculation can be associated to independent clusters with the pattern

Table 6.7.: Instruction patterns for the computer vision application set. Extracted operation clusters could, in each case, be grouped in a much smaller number of instruction patterns. That reinforces the idea that, in many applications, it is possible to find several groups of operations that have the same execution pattern. The individual and combined cover of patterns IP$_A$, IP$_B$, IP$_C$, and IP$_D$ is depicted for each application. These patterns were chosen to compose custom instruction datapaths for this experiment.

| | | Mask convolution | Sobel gradient | Prewitt gradient | Hough transform | RGB-to-YIQ | RGB-to-CMYK |
|---|---|---|---|---|---|---|---|
| Extracted operation clusters | | 39 | 14 | 33 | 11 | 40 | 8 |
| Instruction patterns | | 7 | 4 | 10 | 8 | 8 | 3 |
| Pattern IP$_A$ |  | 41.17% | 50% | 60% | 16.6% | 45% | 0% |
| Pattern IP$_B$ |  | 35.28% | 100% | 80% | 55.5% | 45% | 33.3% |
| Pattern IP$_C$ |  | 0% | 0% | 0% | 22.22% | 40% | 0% |
| Pattern IP$_D$ |  | 0% | 0% | 0% | 0% | 0% | 50% |
| Patterns IP$_A$, IP$_B$, IP$_C$, and IP$_D$ | | 47.05% | 100% | 100% | 83.33% | 65% | 83.3% |

IP$_A$, and the complete Sobel DFG can be covered by clusters with the simpler pattern IP$_B$. Patterns IP$_C$ can only be found in the Hough transform, and RGB-to-YIQ applications. It covers 22.22% of the Hough transform and 40% of the RGB-to-YIQ: a small share for both cases. When combined with patterns IP$_A$ and IP$_B$, however, the pattern IP$_C$ leads the cover of the Hough transform DFG up to 83.33%, and the cover of the RGB-to-YIQ up to 65%. Pattern IP$_D$ corresponds to a *compare and select* cluster that covers up to 50% of the RGB-to-CMYK DFG.

The instruction pattern set composed of IP$_A$, IP$_B$, IP$_C$, and IP$_D$ covers more than 80% of each application DFG in almost all cases. The exceptions are the mask convolution (47.05%) and the RGB-to-YIQ (65%) applications. The remainder of these applications use multipliers, which cannot be found in any of the selected patterns. Clusters including multipliers were only found in these two applications, so they do not contribute to the cover in the other applications.

IP$_A$ and IP$_B$ were used to compose a custom instruction datapath, whereas IP$_C$ and IP$_D$ were used to compose a second one. There are two reasons to build two different custom instruction datapaths instead of one: the similarity between IP$_A$ and IP$_B$, and the more restricted usage of IP$_C$ and IP$_D$. IP$_A$ and IP$_B$ can be easily combined in one datapath, as IP$_B$ is a subgraph of IP$_A$, but their combination with IP$_C$ or IP$_D$ is more difficult, as they do not share common operations. IP$_C$ and IP$_D$ are exclusively used for the Hough transform, RGB-to-YIQ, and RGB-to-CMYK. Therefore, it makes sense to isolate the datapath for a custom instruction that is targeted for these three applications.

A new PE type, denoted PE$_I$, was obtained by expanding the FU of the PE *Bianca* with the custom instruction datapath formed by IP$_A$ and IP$_B$. Similarly, another PE type, denoted PE$_{II}$, was obtained by expanding the FU of the PE *Bianca* with the custom instruction datapath composed of IP$_C$ and IP$_D$. A new architecture instance, denoted *custom array*, was obtained from the *initial array* by substituting 5 PEs of type *Bianca* by 5 PEs of type PE$_I$; substituting 3 PEs of type *Bianca* by PEs of type PE$_I I$; suppressing the use of 6 redundant PEs of type *Bianca*; and organizing the PEs in a 4×3 array, as depicted in Figure 6.16. The composition of the custom array was chosen, so that the number of PEs of each type is the minimal necessary to run all applications with at least the same performance as the initial array.

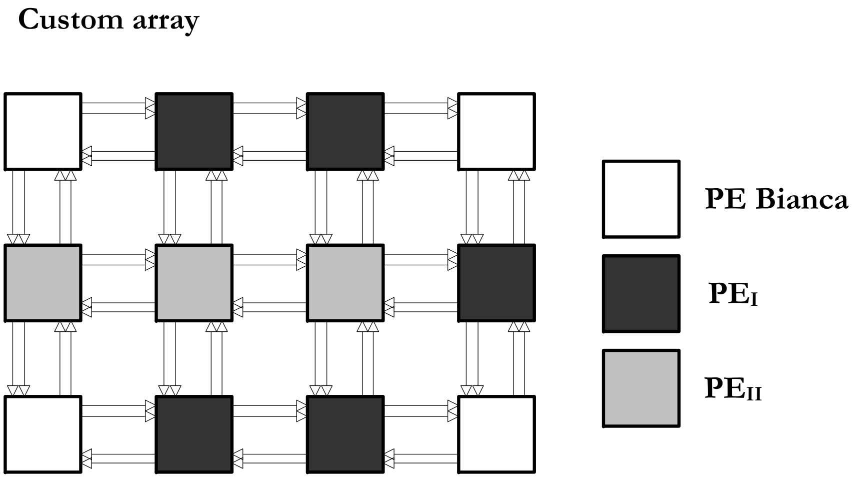


Figure 6.16.: Custom architecture, denoted *custom array*, for the computer vision experiment. This instance is composed of 12 (4×3) PEs of the type *Bianca*, PE$_I$, and PE$_{II}$.

### Impact on composition and implementation area of singular PEs

The composition of the custom PEs PE$_I$ and PE$_{II}$ resembles that of the PE *Bianca*, except for small details, as depicted in table 6.8. PE$_I$ uses 2 extra adders/subtracters to implement the datapath of instruction patterns IP$_A$ and IP$_B$. The original adder/subtracter of the PE *Bianca* was kept in the FU and merged together within this datapath. Custom instructions in FUs of the PE$_{II}$ type required only 3 internal routing multiplexers. The shifter, adder, comparator and selection units originally used in the PE *Bianca* were merged in a common datapath, that allows the execution of each operation, or their combination to execute custom instructions with patterns IP$_C$ and IP$_D$. The FU in PEs type PE$_I$ uses 24% more silicon area than FUs in PEs *Bianca* do. This overhead is of 3.81% for the FU in PEs type PE$_{II}$. This small area overhead [5] can be explained by the fact that custom instructions use units available in the base FU in their composition, as opposed to extra operational units.

Table 6.8.: FU composition and implementation area of the PEs *Bianca*, PE$_I$ and PE$_{II}$. Details about the implementation of PE *Bianca* can be found in Section 4.3.1, Figure 4.16. PE$_I$ and PE$_{II}$ have the same basic design as *Bianca*, except for its FU and context memory.

| | FU composition | FU area ($\mu m^2$) | Configuration Bits | Total PE area ($\mu m^2$) |
|---|---|---|---|---|
| *Bianca* | 1 adders/subtracters<br>1 shifter, 1 comparator,<br>1 multiplier,<br>1 logic unit, and 1 decoder | 22493.94 | 58 | 96597.3 |
| PE$_I$ | Equal to PE *Bianca* plus 2 extra adders/subtracters and 1 multiplexer | 28018.13(+24%) | 65 | 108607 |
| PE$_{II}$ | Equal to PE *Bianca* plus 3 multiplexers | 23352.82(+3.81%) | 61 | 97028.47 |

PEs of type *Bianca* need 58 configuration bits. This number increases to 65 in PE$_I$, and to 61 in PE$_{II}$. In both cases, extra configuration bits are necessary due to the new input ports of their FUs. For example, the FU in PEs type PE$_I$ uses 4 input ports because of the custom instructions with patterns IP$_A$ and IP$_B$.

The estimated area for PE$_I$ is 108607$\mu m^2$, about 12.43% larger than that of the PE *Bianca*. PE$_{II}$ occuppies 97028.47$\mu m^2$ (estimated), which corresponds to an increase of 0.44% in the area in comparison to the PE *Bianca*.

[5] Compare to the overhead incurred in the FUs of type *Fiji* in the multi-point FFT experiment!

### Impact on architecture composition, area, and power consumption

The *initial array* uses 18 PEs type *Bianca*, whereas the *custom array* uses a total of 12 PEs: 4 PEs *Bianca*, 5 PEs of type PE$_I$,and 3 PEs of type PE$_{II}$ (see Figure 6.16). This combination of PE types was determined according to the the mapping strategy and requirements of each application, which are depicted in table 6.9. The mask convolution, Sobel and Prewitt applications were completely pipelined, since they require a high throughput rate. In all of them, three pixel values have to be processed per clock cycle. The circle rasterization of the Hough transformation has a loop with data dependency between iterations, and therefore cannot be completely pipelined. It's mapping is a mixture of multi-context and pipeline. The 2 other remaining applications are mapped by using multi-contexts (explained in Section 2.2.2).

Table 6.9.: Number and type of PEs used during the mapping of each application of the autovision experiment: initial and custom arrays. PEs with custom instructions were eventually used to execute simple operations, such as multiplication, when necessary.

| Application | Mapping style | Initial Array | Custom Array | | |
|---|---|---|---|---|---|
| | | PE$_0$ | PE$_0$ | PE$_I$ | PE$_{II}$ |
| Mask Convolution | pipelined | 17 | 4 | 5 | 3 |
| Sobel Gradient | pipelined | 12 | 0 | 5 | 0 |
| Prewitt Gradient | pipelined | 10 | 0 | 4 | 0 |
| Hough transform (circle rasterization) | multi-context + pipeline | 2(field A) and 4(field B) | 4 | 5 | 2 |
| RGB-to-YIQ | multi-context | 10 | 4 | 3 | 3 |
| RGB-to-CMYK | multi-context | 3 | 0 | 1 | 3 |

The area estimated for the *initial array* is 1.73$mm^2$ (see table 6.10) ; the area estimated for the *custom array* is 1.22$mm^2$, which is about 29% smaller. The static power consumption (leakage power) estimated for the *initial array* corresponds to 114.19$\mu W$. The *custom array* consumes 31.36% less leakage power.

Table 6.10.: Impact of instruction specialization - autovision

| | Initial Array | Custom Array |
|---|---|---|
| Area ($\mu m^2$) | 1.73$mm^2$ | 1.22$mm^2$ |
| Power ($\mu W$) | 114.19 | 78.379 |

## Impact on performance

Custom instructions affected the execution performance of applications in different ways, depending on the mapping strategy used: pipeline or multi-context. When pipeline is used, custom instructions tend to reduce the number of pipeline stages, and thus, the pipeline latency. At multi-context mappings, custom instructions potentially reduce the number of necessary contexts. These two effects are explained here using two examples from the previous set of applications.

Consider the pipeline mapping used for the Prewitt gradient calculation, as depicted in Figure 6.17. All the operations are mapped in the same context, but on different pipeline stages depending on their depth at the DFG. In the *initial array*, three different pipeline stages are required: the first precalculates pixel differences, whereas the second and the third solve two 2-level adders/-subtracters trees. In the *custom array*, instructions exist (PE$_I$) that execute these trees as a single atomic operation (IP$_A$). This custom instruction only requires one cycle, so the whole pipeline can be resume to two stages instead of three. This implies a reduction of 33% in the pipeline latency.

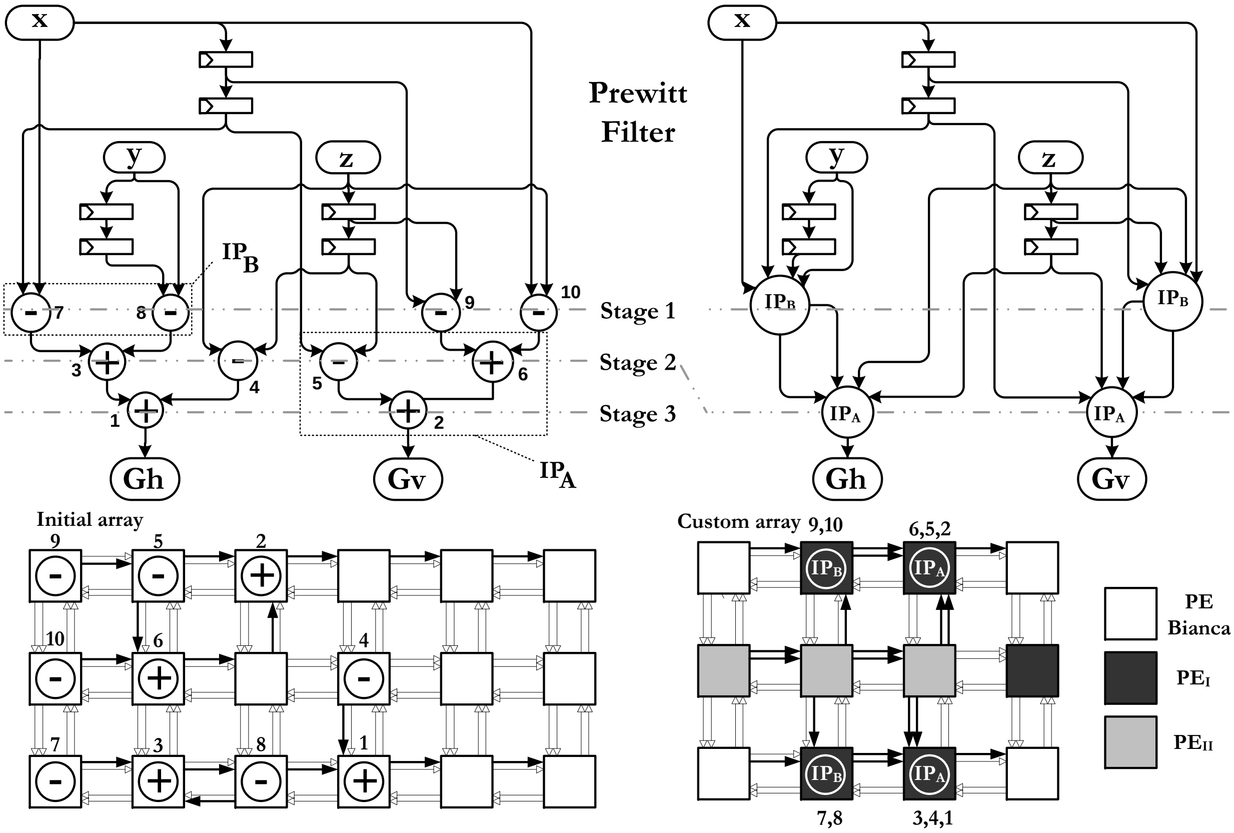


Figure 6.17.: Mapping of the Prewitt's gradient calculation on the *initial array* and on the *custom array*. This application uses 10 PEs of the initial array to implement a 3 stages pipeline. The use of custom instructions allow the same application to be mapped as a 2-stages pipeline that uses only 4 PEs(custom array).

When a multi-context mapping strategy is used, each operation is assigned to a different context depending on their depth at the DFG. An example is depicted in Figure 6.18 for the circle rasterization loop of the Hough transform. When mapping this application in the *initial array*, a total of 5 contexts is necessary. Intermediate results from the second context have to be first shifted (third stage) before they are incremented by one in the fourth context. In the *custom array*, this group of shift-and-add operations (IP$_C$) constitutes a single custom instruction of the PE$_{II}$, which executies in one context. The overall number of necessary contexts drops from 5 to 4, and thus the loop latency is reduced to 4 clock cycles. The contexts in the circle rasterization example are part of a loop, and is typically executed millions of times per processed picture. The use of IP$_C$ as a custom instruction reduces in 20% the whole task execution time, when compared to the *initial array*.

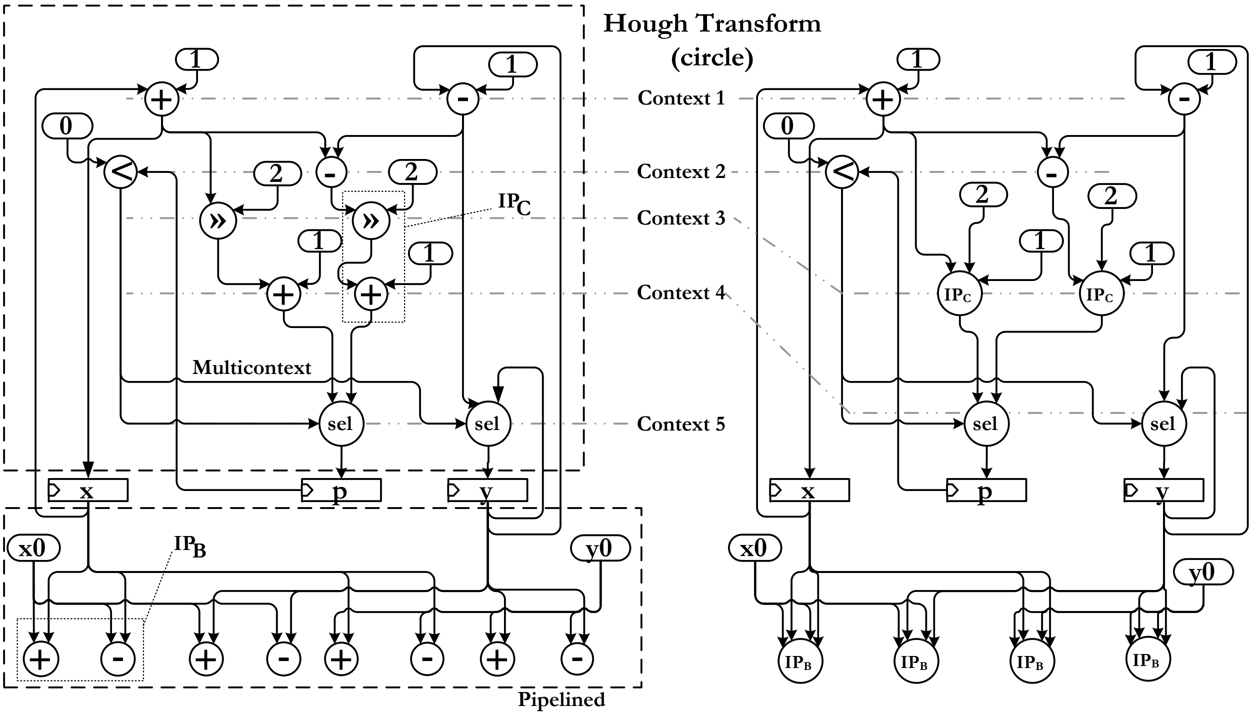


Figure 6.18.: The circle rasterization internal loop of the Hough transformation needs 5 contexts if there are no custom instructions in the architecture. With custom instructions based in the patterns IP$_A$, IP$_B$, and IP$_C$, this application can be mapped onto 4 contexts only.

### Discussion of experimental results

Like in the FFT experiment, the *initial* and *custom* arrays in this experiment execute the same number and sequence of operations during the execution of each application of the set. Both arrays have similar throughput performance: applications running at the initial array consume the same ammount of input data at each clock cycle as when running in the custom array. However, the *custom array* uses 12 PEs, whereas the *initial array* uses 18 PEs: a reduction of 29%. Like in the first experiment, the reason is the use of custom instructions which executes, in one PE, a group of operations that, otherwise, would be mapped onto several individual PEs. As a result, the total number of necessary PEs in the array decreases.

This experiment considers simultaneously several application. It could be observed that there was no instruction pattern which was frequently used by every applications. However, a considerably small set of them (in this case 4 patterns) would already cover a large part of each application. For example, the four extracted patterns cover all the Sobel and Prewitt gradient calculations, and up to 83% of the Hough transform and the RGB-to-CMYK applications. The only exception was the mask convolution, because the area constraints adopted did not allow the extraction of patterns with multipliers. To cover a large part of each application with a small set of instruction patterns is important to keep the number of custom instructions (and the extra costs introduced by them) small. *This experiment is an example that, even when the target set contains several applications, it is possible to design a small set of custom instructions that improves architecture area, power consumption, and architecture efficiency (area per operation).*

In this experiment, the patterns IP$_A$ and IP$_B$ were merged in one single datapath by using a compound composition (see Section 5.1.3). The same for patterns IP$_C$ and IP$_D$. The custom FU for the first two patterns is about 20% bigger than when no custom instructions are used, and the area increment is not significant for the custom FU with the other two patterns. These extra costs are much smaller than when compound composition is not used (compare with the FFT experiment).

One last aspect can be observed from this experiment: custom instructions also may improve the performance of applications running in coarse grained arrays. PEs with custom instructions can concentrate the execution of operations which would otherwise be scheduled to different pipeline stages or contexts. As a consequence, it may happen that the number of necessary pipeline stages, or the number of necessary contexts, decreases when mapping the applications. Two examples, the mapping for the circle rasterization of the Hough transform and the Prewitt filter, were presented, which show this effect. *This demonstrates that the use of instruction pattern based custom instructions may improve the application's performance, besides the architecture area.*

# 7. Conclusions

This research work investigated and presented new methods, techniques, algorithms, and tools to describe and specialize coarse grained reconfigurable architectures during their development phase. For the description of CGRAs, a new architecture description language, CGADL, was introduced that allows the design of these architectures at a high level of abstraction. For the specialization of CGRAs, this work presented methodology, algorithms, and tools that allow the designer to identify, design, and integrate custom instructions in the functional units of CGRAs's processing elements. This chapter summarizes the improvements proposed in this work to the design of CGRAs.

The first research goal in this work was the description of coarse grained architectures. This work showed that, currently, CGRAs are described in a poor and complex way. Actual description languages do not address some specific challenges for the design of CGRAs, such as configurability and spatial distribution of components. The approach introduced in this work was a new, high level, architecture description language: CGADL.

By using CGADL, the designer can describe coarse grained architecture templates in concise, definite, and easy way. In contrast to hardware/system description languages, such as Verilog, VHDL, and SystemC, the key features and technical innovations in CGADL allows the designer to:

- model explicitly how the architecture reconfigures, for example, by use of fast, dynamic, multi-context reconfigurability or by use of statically programmed memories;
- describe the architecture array by means of a spatial positioning system: PEs and interconnection network can be referred to by their position in the array;
- compose and scale the architecture array in an easy, concise, and parameterizable way;
- describe complex and scalable interconnection network based on connection rules, instead of point-to-point connections.

The use of descriptions with high level of abstractions, as the proposed CGADL, eases the generation of software tools, such as estimators, compilers, and simulators. This is an advantage because compilers and simulators can be co-generated, and the architecture templates can be evaluated much earlier in the development phase. This works demonstrates that CGADL allows the generation of software tools: methodology and tools were introduced here that estimate the hardware complexity (gate-count equivalence) of architecture templates described in CGADL. This estimation methodology does not require logic synthesis of the template, and therefore, it can be used early in the development phase. Experimental results show that estimates obtained from CGADL correspond well to those obtained after synthesis in at least three situations:

1. when determining the distribution of hardware costs (gates) among the components of FUs, PEs, and architecture arrays.

2. when determining how the hardware costs (implementation area) of PEs and of FUs vary as a function of parameters of the model, such as number of registers in the register bank and width of the datapath.
3. when sorting FUs, PEs, and architecture instances according to their hardware costs (implementation area).

The second research goal in this work was the specialization of coarse grained architectures towards a set of target applications. To carry out the specialization, this work investigated the design, integration, and use of *custom instructions*: instructions that meet specific demands of one application or application group. One decade ago, the use of custom instructions boosted the industrial development of application specific processors (ASIPs) and is up-to-date an established praxis for the design of these architectures. This work is the first one to use custom instructions for the design of coarse grained arrays. It investigates the principles by which custom instructions can be integrated in CGRAs's PEs, and it demonstrates that custom instructions may decrease the implementation area, decrease the power consumption, and improve performance of coarse grained arrays.

For the design of custom instructions, groups of operations were considered that appear frequently in a set of target applications and with the same kind of execution pattern. According to this design approach, each custom instruction executes a group of operations that, otherwise, would only be accomplished by using several individual PEs. As a consequence, PEs with custom instructions may replace groups of non-custom PEs, decreasing the total number of PEs in the array.

A complete design framework, including methodology, techniques, and algorithms were introduced that allows the designer to:

- identify clusters of operations which lead to feasible custom instructions;
- group clusters according to the similarity of their structure;
- evaluate these clusters according to the frequency they appear in the applications;
- design and implement custom instructions based in one or more clusters of operations;
- integrate the implemented custom instructions in the description of FUs, PEs, and architecture arrays;

Moreover, a tool was implemented that automates the first three tasks mentioned above.

To evaluate the impact (costs and benefits) of custom instructions, two representative, real-world oriented experiments were implemented. The first experiment carries out the specialization of an architecture template for the execution of scalable-OFDMA[1] based systems, such as WiMax [123] and LTE [47]. This target set of applications, composed of streaming FFT algorithms with different input-vector sizes, is very regular and repetitive. This corresponds to an ideal condition to apply the custom instruction design approach proposed here.

According to the experimental results, PEs embedded with custom instructions were about 37% larger than its non-custom counterparts. However, the overall number of PEs in the array could be

[1] Orthogonal frequency-division multiple access modulation scheme.

decreased in 50%, since each PE with custom instructions replaced groups of two or three non-custom PEs. As a consequence, the final, specialized array was about 40% smaller, consumed about 41% less static (leakage) power, and consumed 28% less dynamic power than the initial, non-custom array. This work also demonstrated that the use of custom instructions augments the area efficiency of the architecture: custom arrays use less configuration bits per operation and need less silicon area per operation than its non-custom counterparts.

The second experiment considered a set of seven applications used in computer vision applications for automotive systems. This set of applications was composed of different and irregular applications, and reflects better the specialization of the architecture when several applications are involved. In this situation, it is more difficult to find clusters of operations that are common to all applications, and correspondingly, to design domain-adequate custom instructions. Experimental results show that, even under these conditions, the use of custom instructions has a positive effect to the design of CGRAs.

In the second experiment, the use of custom instructions led to a reduction of 33% in the number of PEs in the array. The custom array used 29% less implementation area than the non-custom array. And the static (leakage) power consumption could be reduced in 31% after specialization. This experiment also demonstrated that the use of custom instructions may improve the execution performance in two situations. First, if the application is mapped in the architecture as a pipeline, the use of custom instructions may execute in one pipeline stage groups of operations that, otherwise, would be executed using several pipeline stages. As a consequence, custom instructions may reduce the number of pipeline stages and therefore the pipeline latency. Second, if the application is mapped in the architecture using a multi-context execution, the use of custom instructions may execute in one context groups of operations that, otherwise, would be executed using several contexts. As a consequence, custom instructions may reduce the number of contexts necessary to map one application.

During the development of this research work, the author published several articles that complement and reflects the content of this dissertation:

1. The description language CGADL was presented for the first time in the article **"*An Architecture Description Language for Coarse-Grained Reconfigurable Arrays*" [89], in the International Conference on Hardware/Software Codesign and System Synthesis (CODES) - Workshop on Application Specific Processors (WASP), in 2007, Salzburg, Austria**.

2. In addition to the estimation tool presented in this dissertation, two additional software tools based in CGADL, a validation tool and a simulator generator, were presented in the journal article **"*CGADL: An Architecture Description Language for Coarse-Grained Reconfigurable Arrays*" [90], IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pages 1233-1246, volume 17, number 9, in 2009**.

3. Experiments for the specialization and design space exploration of coarse grained architectures were published in the article **"*Tuning Coarse-Grained Reconfigurable Architectures towards an Application Domain*" [91], International Conference on Reconfigurable Computing and FPGAs (ReConfig), in 2006, San Luís Potosi, Mexico**.

4. Complementary results and discussions about the use of custom instructions to specialize coarse grained arrays were presented in **"*Evaluating the Impact of Customized Instruction Set on Coarse Grained Reconfigurable Arrays*" [88], International Conference on Field-Programmable Technology (ICFPT), in 2008, Taipei, Taiwan**.

5. A method for mapping applications in CGRAs considering simultaneously the scheduling, the binding, and the routing tasks was proposed in co-authorship with Brenner et al. in the article **"*Optimal Simultaneous Scheduling, Binding and Routing for Processor-Like Reconfigurable Architectures*" [16], International Conference on Field Programmable Logic and Applications (FPL), in 2006, Madrid, Spain.**

6. The mapping of Parallel Algorithms in specialized coarse-grained architectures was discussed in co-authorship with Rullmann et al. in the article **"*Efficient Mapping and Functional Verification of Parallel Algorithms on a Multi-Context Reconfigurable Architecture*" [102], International Conference in Architecture of Computing Systems (ARCS), in 2007, Zurich, Switzerland**.

During the development of this research work, the author contributed in several other research works involving the design of CGRAs. These works considered the use of processor-like reconfigurable arrays [95] [96], their specialization to decrease power consumption [104] [103], and their integration in system-on-chip environemnts [32].

# A. Appendix A

## A.1. CGADL keywords and symbols

Table A.1.: CGADL specific keywords.

| Keyword |
|---|
| MUX |
| REG |
| FSM |
| CONTEXTMEMORY |
| FU |
| INPORT |
| OUTPORT |
| IN |
| PARAMETER |
| PE |
| ARCH |
| RULE |
| CONNECTION |
| ABS_COORD |
| REL_COORD |
| END |
| CONST |
| LOG |
| VOID |

## A.2. CGADL grammar production rules - EBNF

In computer science, Extended Backus–Naur Form (EBNF) is a metasyntax notation used to express context-free grammars: that is, a formal way to describe computer programming languages and other formal languages. It is an extension of the basic Backus–Naur Form (BNF) metasyntax notation. In the following the grammar production rules for the CGADL language is presented.

```
start            ::=   ( PARAMETER )* ( PE )* ARCH <EOF>
PARAMETER        ::=   <PARAMETER> <IDENTIFIER> <IN> <LBRACKET>
```

```
                         <NUMBER> ( <RANGE> <NUMBER> )?
                         ( <COMMA> <NUMBER>( <RANGE> <NUMBER> )? )*
                         <RBRACKET> <SEMICOLON>
PE               ::=      <PE> <LBRACE> ( DECLARATION )*
                         <CONNECTION> <LBRACE> ( CONNECT )*
                         <RBRACE> <RBRACE> <IDENTIFIER><SEMICOLON>
DECLARATION      ::=      ( <TYPE> ID ( <COMMA> ID )* <SEMICOLON> )
                         | ( <PTYPE> IDPARAM ( <COMMA> IDPARAM )*
                         <SEMICOLON> )
                         | ( <DTYPE> IDDESIGN ( <COMMA> IDDESIGN )*
                         <SEMICOLON> )
                         | PORT ( <COMMA> PORT )? <SEMICOLON>
ID               ::=      <IDENTIFIER>
IDPARAM          ::=      <IDENTIFIER> <LPARENTHESES>
                         ( <NUMBER> | <IDENTIFIER> ) <RPARENTHESES>
IDDESIGN         ::=      <IDENTIFIER> <LPARENTHESES><IDENTIFIER>
                         ( <COMMA><IDENTIFIER> )* <RPARENTHESES>
PORT             ::=      ( <INPORT> | <OUTPORT> )
                         <LPARENTHESES> <NUMBER> <RPARENTHESES>
CONNECT          ::=      IDC <LPARENTHESES> IDCP ( <COMMA> IDCP )*
                         <RPARENTHESES> <SEMICOLON>
IDC              ::=      ( <IDENTIFIER> | <OUTPORT> )
IDCP             ::=      (<IDENTIFIER> | <INPORT> )
                         ( <LBRACKET> ( <NUMBER> | <IDENTIFIER> )
                         (<MINUS> <NUMBER> )?
                         ( <RANGE> ( <NUMBER> | <IDENTIFIER> )
                         ( <MINUS> <NUMBER> )? )? <RBRACKET> )?
                         | <VOID>
ARCH             ::=      <ARCH> <LBRACE> ARRAY <CONNECTION> <LBRACE>
                         ( RULE)* ( <IDENTIFIER> <LPARENTHESES>
                         <IDENTIFIER> <RPARENTHESES> <SEMICOLON> )*
                         <RBRACE> <RBRACE>
ARRAY            ::=      ( ARRAYCONCAT )* ( <ARRAY> <LPARENTHESES>
                         ( <IDENTIFIER> | <NUMBER> ) <COMMA>
                         ( <IDENTIFIER> | <NUMBER> )
                         <COMMA> <IDENTIFIER> <RPARENTHESES>
                         <IDENTIFIER> <SEMICOLON> )*
ARRAYCONCAT      ::=      <IDENTIFIER> <EQUALS> <LBRACKET> <IDENTIFIER>
                         ( ( ( <COMMA> | <SEMICOLON> ) )? <IDENTIFIER> )*
                         <RBRACKET> <SEMICOLON>
RULE             ::=      <RULE><LBRACE> ( <PE> <IN> <LPARENTHESES>
                         ARRAYSUBSCRIPT <RPARENTHESES>
                         <LPARENTHESES> ARRAYCONNECT <RPARENTHESES>
                         <SEMICOLON> )* ( <LOG> <LBRACE>
```

```
                    ARRSUBSCROUTPORTS <RBRACE> )?
                    ( <VOID> <LBRACE> ARRSUBSCROUTPORTS
                    <RBRACE> )? <RBRACE> <IDENTIFIER> <SEMICOLON>
ARRAYSUBSCRIPT  ::=  ( ( COLONOPERATOR | <COLON> ) | <LBRACKET>
                    (COLONOPERATOR )* <RBRACKET> ) <COMMA>
                    ( ( COLONOPERATOR | <COLON> ) | <LBRACKET>
                    ( COLONOPERATOR )* <RBRACKET> )
COLONOPERATOR   ::=  ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? ( <COLON>
                    ( <MINUS> ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? <COLON>
                    ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? |
                    ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? ( <COLON>
                    ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? )? ) )?
ARRAYCONNECT    ::=  ARRINCONNECT ( <COMMA>
                    ARRINCONNECT )*
ARRINCONNECT    ::=  ( ( ( <RELCOORD> <LPARENTHESES> ( <MINUS> )?
                    ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? <COMMA> ( <MINUS> )?
                    ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? <RPARENTHESES> )
                    |
                    ( <ABSCOORD> <LPARENTHESES>
                    ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? <COMMA>
                    ( <NUMBER> | <IDENTIFIER> | <END> )
                    ( <MINUS> <NUMBER> )? <RPARENTHESES> ) )
                    <LBRACKET> <NUMBER> ( <RANGE> <NUMBER> )?
                    <RBRACKET> )
                    | <CONST> <LPARENTHESES> <NUMBER> <RPARENTHESES>
                    | <INPORT>
ARROUTPORTSS    ::=  ( <PE> <IN> <LPARENTHESES> ARRAYSUBSCRIPT
                    <RPARENTHESES>
                    <LBRACKET> <NUMBER> ( <RANGE> <NUMBER> )?
                    <RBRACKET> <SEMICOLON> )*
```

# A.3. Circuit models and hardware complexity estimation costs

## A.3.1. Multiplexer Block

### Notation

The complexity of a multiplex block can be calculated using 2-input multiplexers as a base. Multiplexers may be parameterized with the number of inputs and the input wordlength (in bits).

- $C_{\text{mux}^2}$ : is the complexity cost of a 2 input multiplexer, where each input is an 1-bit signal.
- $C_{\text{mux}^2}(n)$ : is the complexity cost of a 2-input multiplexer, where each input is an n-bit signal.

### Schematic drawing

No schematic drawing.

### Complexity

For an $i$-input multiplexer, where $i > 2$, the following costs may be applied, because such multiplexer uses $(i-1)$ 2-input multiplexers in a tree-like structure:

$$\begin{aligned} C_{\text{mux}^i}(n) &= (i-1)C_{\text{mux}^2}(n) \qquad \text{(A.1)} \\ C_{\text{mux}^i}(n) &\leq 3n(i-1) \end{aligned}$$

### Observations for the CRC model

In the CRC Model, the instantiation of a multiplexer block requires the parameterization of the input bit-width and the select signal bit-widht. That gives margin to a second interpretation, where the generated multiplexer would have 2s inputs, where s is the number of bits in the select signal. That is, however, not always the case. For example, one can instantiate a multiplexer with a 3-bit select signal to control 5, 6 or 7 inputs. In that case, the synthesis is optimized to take out the, from 8 possible, not used inputs (and consequently internal gates).

## A.3.2. Finite state machine block

### Notation

An $s$-state finite state machine (FSM) is capable of codifying $s$ states. Each state contains the information for $b$-branch possibilities. Each branch determines to which state the machine will jump if the branch is taken. The output information is stored as an $n$-bit signal. There are $f$ flag signals used to take the branching decision. The complexity of a finite state machine block can be calculated using the complexity cost of its components. These are:

- $C_{\text{stateMem}}(s, b, c)$ : is the complexity cost of an $s$-word latch memory. Each word in this memory block contains $b$ branchwords, each one with $\log c$ bits. $c$ corresponds here to the number of contexts that should be addressed by the output value of the FSM. Modeling $c$, instead of directly indicating the number of bits, is more interesting in the CRC Model, because the number of contexts is a natural parameter of the system.
- $C_{\text{mux}^f}(1)$ : is the complexity cost of an $f$-input multiplexer, where each input is a 1-bit long signal. Such multiplexer is used to select which flags are going to be considered when deciding the next state of the finite state machine.
- $C_{\text{stateDec}}(b, c)$ : is the complexity cost for the state decoder. Each state in the finite state machine codifies the information about the next state to be used after branching and the output value associated to the state. Up to now, the CRC Model makes no explicit distinction between these two informations, although it is a possibility in future instances. Such information is stored in each one of the $b$-branch words of each state, and it is represented with a $(\log c)$-bit signal.

### Schematic drawing

The general model for the realization of the FSM in the CRC Model is depicted in figure A.1. Particularly for the model *Bianca*, which is analysed here, the output is used to select the context (output) as well as the next state of the machine. Such connection is shown with a dashed line.

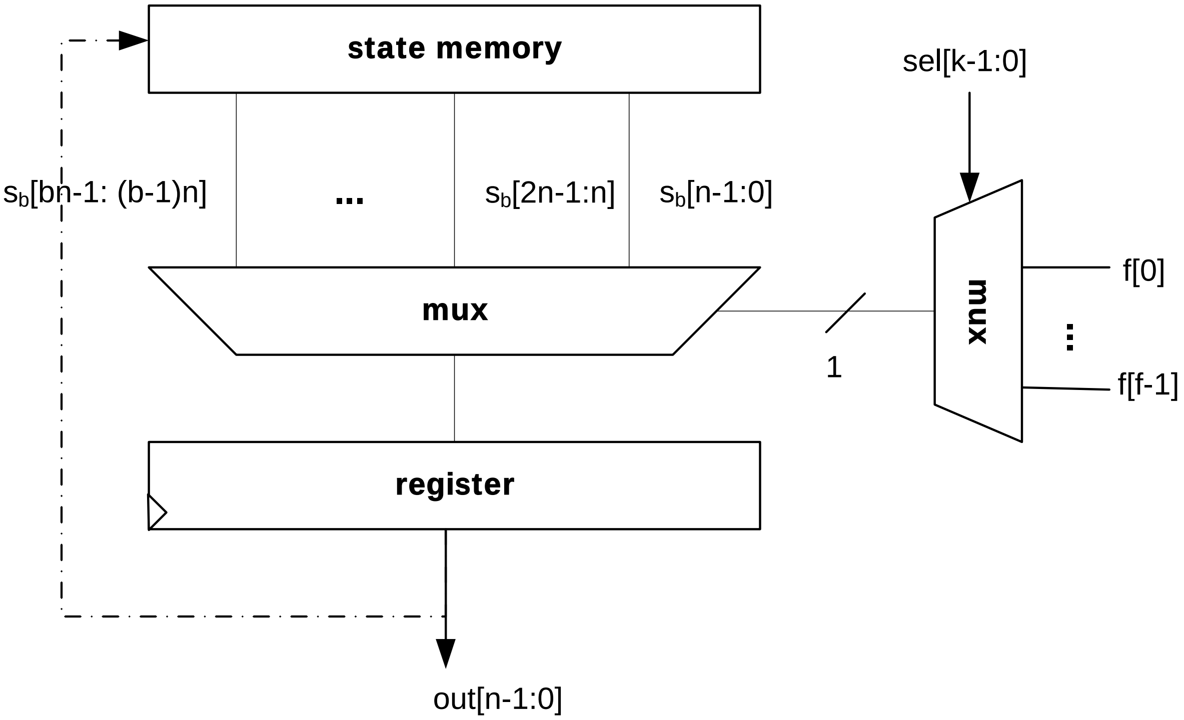


Figure A.1.: Schematic for the CRC-FSM block.

### Complexity

Before coming to the final cost equation of the finite state machine, we have to discuss the construction of its sub-elements. The state memory block is implemented as RAM latch cells, which take us to the following cost and delay:

$$C_{\text{stateMem}}(s, b, c) = C_{\text{RAMcell}}(s+3)(b \log c + \log\log(b \log c)) \tag{A.2}$$

If the number of addressed contexts is the same of addressed states, then $s = \log c$. The cost for the flag multiplexer follows the cost of a multiplexer block, as indicated in section A.3.1.

The finite state machine decoder may be decomposed in:

- one $b$-input multiplexer, where each input has $(\log c)$ bits, and therefore its costs is $C_{\text{mux}^b}(\log c)$; and
- one $(\log c)$-bit register with cost $C_{\text{ff}}(\log c)$.

Therefore, it is possible to write the costs of the finite state machine block as follows:

$$C_{\text{FSM}}(s, c, b, f) = C_{\text{stateMem}}(s, b, c) + C_{\text{mux}^f}(1) + C_{\text{mux}^b}(\log c) + C_{\text{ff}}(\log c)$$

## A.3.3. Context memory block

### Notation

A context memory is a latch based memory module. That is, the bits are stored in latches instead of registers. A $c$-contexts block is a set of $c$ $n$-bits latches. This module receives an $a_w$-bits write address, an $a_r$-bits read address and a $n$-bits vector. It outputs a $n$-bits dataword. When writing to this memory model, the input data is stored on the position indicated by the write address. When reading from this module, the content presente in the position indicated by the read address is output. The complexity of a context memory may be calculated using the complexity cost of its components. These are:

- $C_{\text{cmWrDec}}(a_w)$ : is the complexity cost of a $a_w$ bits decoder. The context memory decoder also includes a set of enabling gates to control the writing process on the latches.
- $C_{\text{latchMem}}(c, n)$ : is the complexity cost of the latch set forming $c$-contexts with $n$-bits each.
- $C_{\text{mux}^c}(n)$ : is a $n$-bits $c$-inputs multiplexer unit.

### Schematic drawing

The general circuit for the realization of the context memory in the CRC model is depicted in figure A.2.The number of necessary address signals is normaly equal to $\lceil \log(n) \rceil$. Each cell in the context memory model is based in simple tri-state buffers and consider additional glue logic to reset and load enable circuits. These are necessary for the initialization, writing and reading on the latch memory.

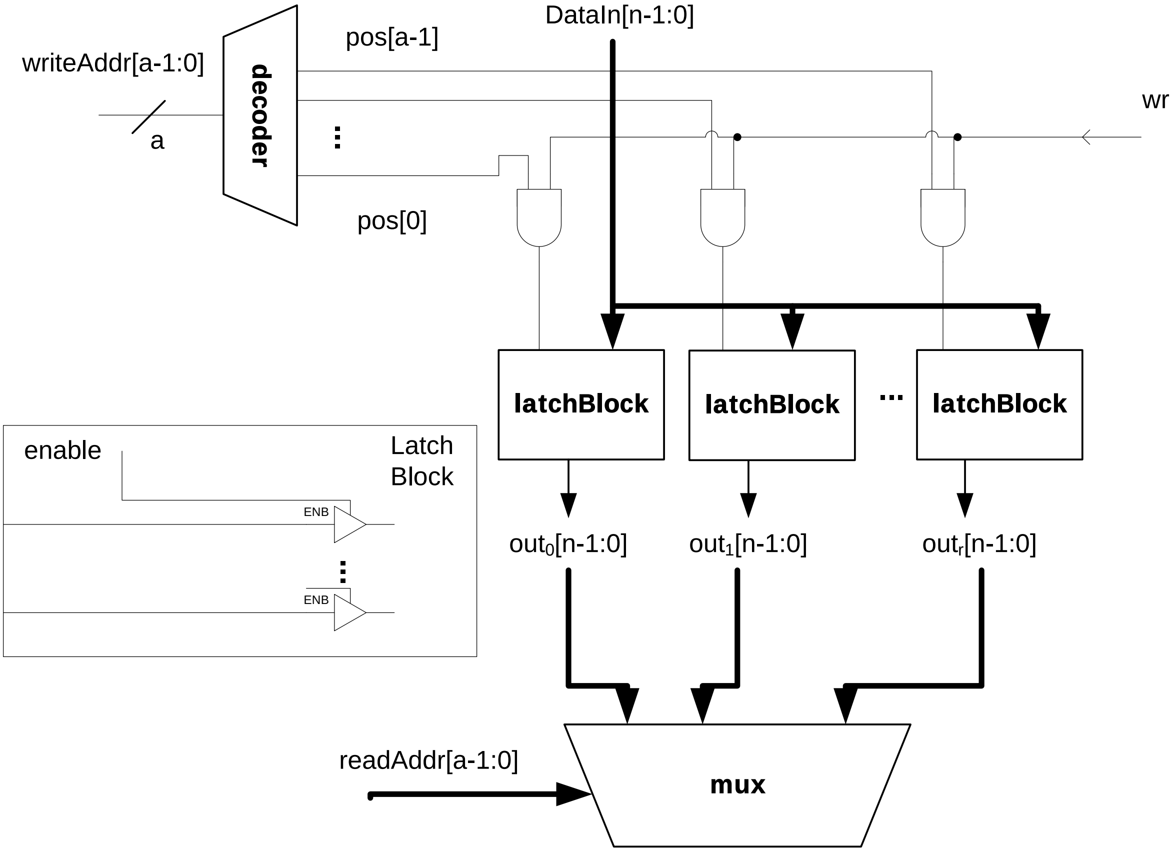


Figure A.2.: Schematic for the CRC context memory block.

### Complexity

Before coming to the final cost equation of the register bank, we have to discuss the construction of its sub-elements. The decoder cost is equivalent to the cost of a register set, as discussed in chapter 4. Additionally, each decoding line receives an AND gate as part of the writing mechanism control.

$$C_{\text{cmWrDec}}(a_w) = C_{\text{Dec}}(a_w) + a_w^2 C_{\text{and}}$$
$$C_{\text{latchMem}}(c,n) = c \times n \times C_{\text{3-stateDrive}}$$

It is then possible to write the costs of the context memory block as follows:

$$C_{\text{CM}}(a_w, a_r, c, n) = C_{\text{cmWrDec}}(a_w) + C_{\text{latchMem}}(c,n) + C_{\text{mux}^{a_r}}(n)$$

# Bibliography

All the Internet sources are from October 2009.

[1] AHO, A. V., R. SETHI and J. D. ULLMAN: *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1996.

[2] ALTERA CORPORATION: *FFT MegaCore Function User Guide - v6.1*. Online: http://www.altera.com, 2004.

[3] AMANO, H.: *A Survey on Dynamically Reconfigurable Processors*. IEICE Transactions on Communications, E89-B(12):3179–3187, 2006.

[4] ARBELO, C., A. KANSTEIN, S. LOPEZ, J. LOPEZ, M. BEREKOVIC, R. SARMIENTO and J.-Y. MIGNOLET: *Mapping Control-Intensive Video Kernels onto a Coarse-Grain Reconfigurable Architecture: the H.264/AVC Deblocking Filter*. In *Proceedings of the Design, Automation, and Test Conference in Europe (DATE)*, Nice, France, 2007.

[5] ASHENDEN, P. J.: *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[6] ASHENDEN, P. J.: *Digital Design : an Embedded Systems Approach using Verilog*. Morgan Kaufmann Publishers, Amsterdam, Boston, 2008.

[7] ATASU, K., L. POZZI and P. IENNE: *Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints*. In *Proceedings of the Design Automation Conference (DAC)*, New York, USA, 2003.

[8] AZEVEDO, R., S. RIGO, M. BARTHOLOMEU, G. ARAUJO, C. ARAUJO and E. BARROS: *The ArchC Architecture Description Language and Tools*. International Journal of Parallel Programming, 33(5):453–484, 2005.

[9] BABEL, L.: *A Fast Algorithm for the Maximum Weight Clique Problem*. Computing, 52:31–38, 1994.

[10] BANSAL, N., S. GUPTA, N. DUTT and A. NICOLAU: *Analysis of the Performance of Coarse-Grain Reconfigurable Architectures with Different Processing Element Configurations*. In *Workshop on Application Specific Processors (WASP)*, San Diego, USA, 2003.

[11] BAUMGARTE, V., G. EHLERS, F. MAY, A. NAECKEL, M. VORBACH and M. WEINHARDT: *PACT XPP—A Self-Reconfigurable Data Processing Architecture*. Journal of Supercomputing, 26(2):167–184, 2003.

[12] BONZINI, P. and L. POZZI: *Polynomial-time Subgraph Enumeration for Automated Instruction Set Extension.* In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Nice, France, 2007.

[13] BOSSUET, L., G. GOGNIAT and J.-L. PHILIPPE: *Generic Design Space Exploration for Reconfigurable Architectures.* In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Washington, USA, 2005.

[14] BOUWENS, F., M. BEREKOVIC, B. D. SUTTER and G. GAYDADJIEV: *Architecture Enhancements for the ADRES Coarse-Grained Reconfigurable Array.* In *HiPEAC*, Goteborg, Sweden, 2008.

[15] BOUWENS, F. J., M. BEREKOVIC, A. KANSTEIN and G. N. GAYDADJIEV: *Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array.* In *Proceedings of International Workshop on Applied Reconfigurable Computing (ARC)*, Zurich, Switzerland, 2007.

[16] BRENNER, J., J. VAN DER VEEN, S. FEKETE, J. **OLIVEIRA FILHO** and W. ROSENSTIEL: *Optimal Simultaneous Scheduling, Binding and Routing for Processor-Like Reconfigurable Architectures.* In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, 2006.

[17] BROWN, S. and J. ROSE: *FPGA and CPLD Architectures: A Tutorial.* IEEE Design and Test of Computers, 13(2):42–57, 1996.

[18] BURNS, G. F., M. JACOBS, M. LINDWER and B. VANDEWIELE: *Silicon Hive's Scalable and Modular Architecture Template for High-Performace Multi-Core Systems.* Online, http://www.siliconhive.com, Silicon Hive, 2005.

[19] CALLAHAN, T. J., J. R. HAUSER and J. WAWRZYNEK: *The Garp Architecture and C Compiler.* Computer, 33(4):62–69, 2000.

[20] CAMPOSANO, R. and W. ROSENSTIEL: *Synthesizing Circuits From Behavioral Descriptions.* IEEE Transaction on Computer-Aided Design, 8(2):171–180, 1989.

[21] CHAREST, L., E. M. ABOULHAMID and A. TSIKHANOVICH: *Designing with SystemC: Multi-Paradigm Modeling and Simulation Performance Evaluation.* In *Proceedings of the International Hardware Description Language Conference (HDL)*, San Jose, CA, 2002.

[22] CHATTOPADHYAY, A., X. CHEN, H. ISHEBABI, R. LEUPERS, G. ASCHEID and H. MEYR: *High-Level Modelling and Exploration of Coarse-Grained Re-configurable Architectures.* In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, New York, USA, 2008.

[23] CONG, J., Y. FAN, G. HAN and Z. ZHANG: *Application-Specific Instruction Generation for Configurable Processor Architectures.* In *Twelfth International Symposium on Field Programmable Gate Arrays*, Monterey, California, USA, 2004.

[24] CORDELLA, L. P., P. FOGGIA, C. SANSONE and M. VENTO: *Performance Evaluation of the VF Graph Matching Algorithm.* In *Proceedings of the International Conference on Image Analysis and Processing (ICIAP)*, Washington, DC, USA, 1999. IEEE Computer Society.

[25] CORDELLA, L. P., P. FOGGIA, C. SANSONE and M. VENTO: *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs.* IEEE Transactions on Pattern Analysis Machine Intelligence, 26(10):1367–1372, 2004.

[26] COWARE CORPORATION: *Lisatek.* Online, http://www.coware.com, 2009.

[27] DEHON, A.: *DPGA Utilization and Application.* In *Proceedings of the 1996 ACM Fourth International Symposium on Field-programmable Gate Arrays (FPGA)*, New York, USA, 1996.

[28] DEMICHELI, G.: *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill, New York, 1994.

[29] DREBIN, R. A., L. CARPENTER and P. HANRAHAN: *Volume rendering.* SIGGRAPH Computer Graphics, 22(4):65–74, 1988.

[30] DULLER, A., D. TOWNER, G. PANESAR, A. GRAY and W. ROBBINS: *picoArray Technology: The Tool's Story.* In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Washington, DC, USA, 2005.

[31] EBELING, C.: *GeminiII : a Second Generation Layout Validation Program.* In *Procceedings of the International Conference on Computer-Aided Design (ICCAD)*, Santa Clara, USA, 1988.

[32] EISENHARDT, S., J. **OLIVEIRA FILHO**, T. KUHN and W. ROSENSTIEL: *Speculative Configuration Prefetching for Multi-Context Architectures.* In *Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI)*, Okinawa, Japan, 03 2009.

[33] FAUTH, A., J. V. PRAET and M. FREERICKS: *Describing Instruction Set Processors Using nML.* In *Proceedings of the European Design and Test Conference (DATE)*, Munich, Germany, 1995.

[34] FISCHER, D., J. TEICH, R. WEPER, U. KASTENS and M. THIES: *Design space characterization for architecture/compiler co-exploration.* In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, New York, USA, 2001.

[35] GAJSKI, D. D. and L. RAMACHANDRAN: *Introduction to High-Level Synthesis.* IEEE Design and Test of Computers, 11(4):44–54, 1994.

[36] GALUZZI, C. and K. BERTELS: *The Instruction-Set Extension Problem: A Survey.* In *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC)*, London, UK, 2008.

[37] GARCIA, A., M. BEREKOVIC and T. V. AA: *Mapping of the AES Cryptographic Algorithm on a Coarse-Grain Reconfigurable Array Processor*. In *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, Montreal, Canada, 2008.

[38] GAREY, M. R. and D. S. JOHNSON: *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, California, 1978.

[39] GAVRIL, F.: *Algorithms for a Maximum Clique and a Minimum Independent Set of a Circle Graph*. Networks, 3:261–273, 1973.

[40] GAVRIL, F.: *Algorithms on Circular-arc Graphs*. Networks, 4:357–369, 1974.

[41] GILAT, A.: *Matlab – An Introduction With Applications*. John Wiley & Sons, Incorporated, 2008.

[42] GOLDSTEIN, S. C., H. SCHMIT, M. BUDIU, S. CADAMBI, M. MOE and R. TAYLOR: *PipeRench: A Reconfigurable Architecture and Compiler*. IEEE Computer, 4(33):70–77, 2000.

[43] GOLUMBIC, M. C.: *Algorithmic Graph Theory and Perfect Graphs*. Annals of discrete mathematics. Elsevier, Amsterdam, Netherlands, 2004.

[44] GONZALEZ, R. and R. WOODS: *Digital Image Processing*. Addison Wesley, New York, USA, 1992.

[45] GOSLING, J., B. JOY, G. STEELE and G. BRACHA: *The Java Language Specification*. Addison Wesley, 2005.

[46] GRAPHDRAWING.ORG: *The GraphML File Format*. Online, http://graphml.graphdrawing.org, 2009.

[47] GSA - THE GLOBAL MOBILE SUPPLIERS ASSOCIATION: *Evolution to LTE (GSA Information Paper) confirms 42 LTE network commitments in 21 countries*. Online, http://www.gsacom.com, 2009.

[48] GUO, Y., G. SMIT, P. HEYSTERS and H. BROERSMA: *A Graph Covering Algorithm for a Coarse Grain Reconfigurable System*. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, USA, 2003.

[49] HADJIYIANNIS, G., S. HANONO and S. DEVADAS: *ISDL: An Instruction Set Description Language for Retargetability*. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, California, USA, 1997.

[50] HALAMBI, A. and P. GRUN: *EXPRESSION: A language for architecture exploration through compiler/simulator retargetability*. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Munich, Germany, 1999.

[51] HALFHILL, T. R.: *Silicon Hive Breaks Out — Philips Startup Unveils Configurable Parallel-Processing Architecture*. Microprocessor, 12:1–7, 2003.

[52] HANNIG, F., H. DUTTA, A. KUPRYIANOV, J. TEICH, R. SCHAFFER, S. SIEGEL, R. MERKER, R. KERYELL, B. POTTIER, D. CHILLET, D. MENARD and O. SENTIEYS: *Co-Design of Masiively Parallel Embedded Processor Architectures*. In *Proceedings of the ReCoSoC Workshop*, Montpellier, France, 2005.

[53] HARTENSTEIN, R., R. KRESS and H. REINIG: *A New FPGA Architecture for Word-Oriented Datapaths*. In *Proceedings of the International Wrokshop on Field Programmable Logic and Applications (FPL)*, Prague, Czech Republic, 1994.

[54] HARTENSTEIN, R. W.: *A Decade of Reconfigurable Computing: a Visionary Retrospective.*. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Munich, Germany, 2001.

[55] HARTENSTEIN, R. W., M. HERZ, T. HOFFMANN and U. NAGELDINGER: *Generation of Design Suggestions for Coarse-Grain Reconfigurable Architectures*. In *Proceedings of the The Roadmap to Reconfigurable Computing, International Workshop on Field-Programmable Logic and Applications (FPLA)*, London, UK, 2000.

[56] HAUCK, S. and A. DEHON: *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.

[57] HE, S. and M. TORKELSON: *Designing pipeline FFT processor for OFDM (de)modulation*. In *Procceedings of the International Symposium on Signals, Systems, and Electronics (ISSSE)*, Pisa, Italy, 1998.

[58] HOFFMANN, A., H. MEYR and R. LEUPERS: *Architecture Exploration for Embedded Processors with LISA*. Kluwer Acaddemic Publishers, 2002.

[59] HUANG, Z., S. MALIK, N. MOREANO and G. ARAUJO: *The Design of Dynamically Reconfigurable Datapath Coprocessors*. ACM Transactions on Embedded Computing Systems, 3(2):361–384, 2004.

[60] IENNE, P. and R. LEUPERS: *Customizable Embedded Processors–Design Technologies and Applications*. Systems on Silicon Series. Morgan Kaufmann, San Mateo, California, USA, 2006.

[61] JAIN, R., R. KASTURI and B. G. SCHUNCK: *Machine Vision*. McGraw-Hill International Editions, 1995.

[62] JONES, A. M. and M. BUTTS: *TeraOPS Hardware: A New Massively-Parallel MIMD Computing Fabric IC*. In *Proceedings of the IEEE Hot Chips Symposium*, Stanford, USA, 2006.

[63] KASTNER, R., S. OGRENCI-MEMIK, E. BOZORGZADEH and M. SARRAFZADEH: *Instruction generation for hybrid reconfigurable systems*. In *Procceedings of IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, San Jose, California, 2001.

[64] KIM, Y., M. KIEMB and K. CHOI: *Efficient Design Space Exploration for Domain-Specific Optimization of Coarse-Grained Reconfigurable Architecture*. In *SoC Design Conference*, Korea, 2005.

[65] KIM, Y., M. KIEMB, C. PARK, J. JUNG and K. CHOI: *Resource Sharing and Pipelining in Coarse-Grained Reconfigurable Architecture for Domain-Specific Optimization*. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Washington, USA, 2005.

[66] KIRKPATRICK, S., C. D. GELATT and M. P. VECCHI: *Optimization by Simulated Annealing*. Science, 220(4598):671–680, 1983.

[67] KISSLER, D., F. HANNIG, A. KUPRIYANOV and J. TEICH: *A Dynamically Reconfigurable Weakly Programmable Processor Array Architecture Template.*. In *ReCoSoC*, Darmstadt, Germany, 2006.

[68] KUPRIYANOV, A., F. HANNIG, D. KISSLER, R. MERKER, R. SHAFFER and J. TEICH: *An Architecture Description Language for Massively Parallel Processor Architectures*. In *Proceedings of the GIITGGMM-Workshop - Methoden und Beschreibungssprachenzur Modellierung und Verifikation von Schaltungen und Systemen*, Dresden, Germany, 2006.

[69] KUPRIYANOV, A., F. HANNIG, D. KISSLER, J. TEICH, J. LALLET, O. SENTIEYS and S. PILLEMENT: *Modeling of Interconnection Networks in Massively Parallel Processor Architectures*. Lecture Notes in Computer Science - Architecture of Computing Systems (ARCS), 4415:268–282, 2007.

[70] LAURENT, S. S.: *XML: a Primer*. John Wiley & Sons, Inc., New York, USA, 2001.

[71] LEE, M.-H., H. SINGH, G. LU, N. BAGHERZADEH, F. J. KURDAHI, E. M. C., A. FILHO and V. CASTRO: *Design and Implementation of the MorphoSys Reconfigurable Computing Processor*. Journal of VLSI Signal Processing Systems, 24(2/3):147–164, 2000.

[72] LIN, Y.-W., H.-Y. LIU and C.-Y. LEE: *A 1-GS/s FFT/IFFT Processor for UWB Applications*. IEEE Journal of Solid-State Circuits, 40(8):1726–1735, 2005.

[73] LIU, Z., Y. SONG, T. IKENAGA and S. GOTO: *A VLSI Array Processing Oriented Fast Fourier Transform Algorithm and Hardware Implementation*. IEICE Transactions on Fundamentals, E88-A(12):3523–3530, 2005.

[74] MEALY, G. H.: *A Method for Synthesizing Sequential Circuits*. Bell System Technical Journal, 34(5):1045–1079, 1955.

[75] MEI, B., A. LAMBRECHTS, D. VERKEST, J.-Y. MIGNOLET and R. LAUWEREINS: *Architecture Exploration for a Reconfigurable Architecture Template*. IEEE Design and Test, 22(2):90–101, 2005.

[76] MEI, B., S. VERNALDE, D. VERKEST, H. DE MAN and R. LAUWEREINS: *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix.* In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, 2003.

[77] MEI, B., S. VERNALDE, D. VERKEST and R. LAUWEREINS: *Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study.* In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Washington, USA, 2004.

[78] MEI, B., S. VERNALDE, D. VERKEST, H. D. MAN, R. LAUWEREINS, B. MEI, S. VERNALDE, D. VERKEST, H. DE MAN and R. LAUWEREINS: *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures.* In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, Hong Kong, China, 2002.

[79] MERIBOUT, M. and M. MOTOMURA: *Efficient Metrics and High-Level Synthesis for Dynamically Reconfigurable Logic.* IEEE Transactions on Very Large Scalse Integration (VLSI) Systems, 12(6):603–621, 2004.

[80] MIRAMOND, B. and J.-M. DELOSME: *Design Space Exploration for Dynamically Reconfigurable Architectures.* In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Washington, USA, 2005.

[81] MIRSKY, E. and A. DEHON: *MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources.* In *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, 1996.

[82] MÜLLER, S. and W. PAUL: *Computer Architecture: Complexity and Correctness.* Springer, 2000.

[83] MOORE, E. F.: *Gedanken Experiments on Sequential Machines.* In *Automata Studies*, pp. 129–153. Princeton U., 1956.

[84] MOTOMURA, M.: *A dynamically Reconfigurable Processor Architecture.* In *Microprocessor Forum*, Tokio, Japan, 2002.

[85] MOTOMURA, M., T. FUJII, K. FURUTA, K. ANJO, Y. YABE, K. TOGWA, J. YAMADA, Y. IZAWA and R. SASAKI: *New Generation Microprocessor Architecture (2) Dynamically Reconfigurable Processor (DRP).* Joho Shori, 46(11):1259–1265, 2005.

[86] MUCHNICK, S. S.: *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[87] OH, J.-Y. and M.-S. LIM: *New Radix-2 to the 4th Power Pipeline FFT Processor.* IEICE Transactions on Electronics, 88(8):1740–1746, 2005.

[88] **OLIVEIRA FILHO**, J., T. KUHN and W. ROSENSTIEL: *Evaluating the Impact of Customized Instruction Set on Coarse Grained Reconfigurable Arrays.* In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*, Taipei, Taiwan, 2008.

[89] **OLIVEIRA FILHO**, J., S. MASEKOWSKY, T. SCHWEIZER and W. ROSENSTIEL: *An Architecture Description Language for Coarse-Grained Reconfigurable Arrays*. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES) - Workshop on Application Specific Processors (WASP)*, Salzburg, Austria, 10 2007.

[90] **OLIVEIRA FILHO**, J., S. MASEKOWSKY, T. SCHWEIZER and W. ROSENSTIEL: *CGADL: an Architecture Description Language for Coarse-Grained Reconfigurable Arrays*. IEEE Transactions in Very Large Scale Integration Systems, 17(09):1233–1246, September 2009.

[91] **OLIVEIRA FILHO**, J., T. SCHWEIZER, T. OPPOLD, T. KUHN and W. ROSENSTIEL: *Tuning Coarse-Grained Reconfigurable Architectures towards an Application Domain*. In *International Conference on Reconfigurable Computing and FPGAs (ReConfig)*, San Luis Potosi, Mexico, 2006.

[92] OPPOLD, T., U. KANUS, T. SCHWEIZER, T. KUHN, W. ROSENSTIEL and W. STRASSER: *Evaluation of Ray Casting on Processor-Like Reconfigurable Architectures*. In *International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finland, 2005.

[93] OPPOLD, T., T. SCHWEIZER, T. KUHN and W. ROSENSTIEL: *Cost Functions for the Design of Dynamically Reconfigurable Processor Architectures*. In *Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI)*, Kanazawa, Japan, 2004.

[94] OPPOLD, T., T. SCHWEIZER, T. KUHN and W. ROSENSTIEL: *A New Design Approach for Processor-Like Reconfigurable Hardware*. In *Proceedings of the Euro DesignCon*, Munich, Germany, 2004.

[95] OPPOLD, T., T. SCHWEIZER, J. **OLIVEIRA FILHO**, S. EISENHARDT, T. KUHN and W. ROSENSTIEL: *Execution Schemes for Dynamically Reconfigurable Architectures*. In *Procceedings of the Workshop on Synthesis And System Integration of Mixed Information (SASIMI)*, Taipei, Taiwan, 2006.

[96] OPPOLD, T., T. SCHWEIZER, J. **OLIVEIRA FILHO**, S. EISENHARDT and W. ROSENSTIEL: *CRC-Concepts and Evaluation of Processor-Like Reconfigurable Archtitectures*. it - Information Technology, 49(3):157–164, 2007.

[97] PALNITKAR, S.: *Verilog®Hdl: a guide to digital design and synthesis*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2003.

[98] PETROV, M., T. MURGAN, F. MAY, M. VORBACH, P. ZIPF and M. GLESNER: *The XPP Architecture and Its Co-Simulation Within the Simulink Environment*. Lecture Notes in Computer Science, 3203:761–770, 2004.

[99] PICOCHIP: *picoArray Architecture*. Online, http://www.picochip.com/, 2009.

[100] QUICKSILVER TECHNOLOGY: *Homepage*. Online : http://www.qstech.com/, 2009.

[101] RIGO, S., G. ARAUJO, M. BARTHOLOMEU and R. AZEVEDO: *ArchC: A SystemC-Based Architecture Description Language*. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Washington, DC, USA, 2004.

[102] RULLMANN, M., S. SIEGEL, R. MERKER, J. **OLIVEIRA FILHO**, T. SCHWEIZER, T. OPPOLD and W. ROSENSTIEL: *Efficient Mapping and Functional Verification of Parallel Algorithms on a Multi-Context Reconfigurable Architecture*. In *Proccedings of the International Conference in Architecture of Computing Systems (ARCS) – Workshop for Dynamically Reconfigurable Systems (DRS)*, Zurich, Switzerland, 2007.

[103] SCHWEIZER, T., J. **OLIVEIRA FILHO**, T. KUHN and W. ROSENSTIEL: *Low Energy Voltage Dithering in Dual VDD Circuits*. In *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Delft, Netherlands, 09 2009.

[104] SCHWEIZER, T., J. **OLIVEIRA FILHO**, T. OPPOLD, T. KUHN and W. ROSENSTIEL: *Evaluation of Temporal-Spatial Voltage Scaling for Processor-Like Reconfigurable Architectures*. In *Proceedings of the Euro DesignCon*, Munich, Germany, 2005.

[105] SILICON HIVE. Online, http://www.siliconhive.com, 2009.

[106] SINGH, H., G. LU, M.-H. LEE, N. BAGHERZADEH, R. MAESTRE, E. FILHO and F. KURDAHI: *MorphoSys: Case Study of a Reconfigurable Computing System Targeting Multimedia Applications*. In *Proceedings of the Design Automation Conference (DAC)*, Los Alamitos, CA, USA, 2000.

[107] SIPPER, M. and E. SANCHEZ: *Configurable Chips Meld Software and Hardware*. IEEE Computer, 33(1):120–121, 2000.

[108] SMIT, G. J. M., P. M. HEYSTERS, M. A. J. ROSIEN and E. MOLENKAMP: *Lessons Learned from Designing the Montium: a Coarse-Grained Reconfigurable Processing Tile*. In *Proceedings of the International Symposium on System-on-Chip (ISSC)*, Los Alamitos, California, 2004.

[109] SOUZA, C. C. DE, A. M. LIMA, N. MOREANO and G. ARAUJO: *The Datapath Merging Problem in Reconfigurable Systems: Lower Bounds and Heuristic Evaluation*. In *Workshop on Efficient and Experimental Algorithms*, Rio de Janeiro, Brazil, 2004.

[110] SUN, F., S. RAVI, A. RAGHUNATHAN and N. K. JHA: *Custom-Instruction Synthesis for Extensible-Processor Platforms*. IEEE Transactions on CAD of Integrated Circuits and Systems, 23(2):216–228, 2004.

[111] TAYLOR, M.: *The RAW Processor – A Scalable 32-bit Fabric for Embedded and General Purpose Computing*. In *Proceedings of IEEE Hot Chips Symposium*, Stanford, USA, 2001.

[112] TEICH, J.: *Digitale Hardware/Software Systeme – Synthese und Optimierung*. Springer Verlag, 1997.

[113] TENSILICA. Online, http://www.tensilica.com, 2009.

[114] TOMITA, E. and T. SEKI: *An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique*. Lecture Notes in Computer Science, 2731:278–289, 2003.

[115] TREDENNICK, N. and B. SHIMAMOTO: *Go Reconfigure*. IEEE Spectrum, 40(12):36–40, 2003.

[116] VAHID, F. and R. LYSECKY: *VHDL for Digital Design*. Wiley, John & Sons, Incorporated, 2007.

[117] VALIENTE, G.: *Algorithms on Trees and Graphs*. Springer, 2002.

[118] ČERNÝ, V.: *Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm*. Journal of Optimization Theory and Applications, 45(1):41–51, 1985.

[119] VEREDAS, F.-J., M. SCHEPPLER, W. MOFFAT and B. MEI: *Custom Implementation of the Coarse-grained Reconfigurable ADRES Architecture for Multimedia Purposes*. In *Proceeding of the International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finland, 2005.

[120] VEREDAS-RAMIREZ, F. J., M. SCHEPPLER and H. J. PFLEIDERER: *A Survey on Reconfigurable Computing Systems: Taxonomy and Metrics*. In *Workshop on Reconfigurable Computing and Applications (JCRA)*, Spain, 2004.

[121] WALKER, R. A. and S. CHAUDHURI: *Introduction to the Scheduling Problem*. IEEE Design and Test of Computers, 1:60–69, 1995.

[122] WEBOPEDIA: *s.v. program*. Online, http://www.webopedia.com, 2009.

[123] WIMAX FORUM: *Mobile WiMax*. Online: http://www.wimaxforum.org/, 2006.

[124] WU, J., K. LIU, B. SHEN and H. MIN: *A Hardware Efficient VLSI Architecture for FFT Processor in OFDM Systems*. In *International Conference on ASIC*, Shangai, China, 2005.

[125] YAMAGUCHI, K. and S. MASUDA: *A New Exact Algorithm for the Maximum Weight Clique Problem*. In *Proceedings fo the International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, Shimonoseki, Japan, 2008.