A Study on Representation Model for Coarse-Grained Dynamically Reconfigurable Systems

September 2008

Vasutan Tunbunheng

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank all the people who supported to accomplish this thesis.

First of all, thank you Japanese government for giving me a chance to get the Monbukagaku-Sho scholarship to pursue my Master Degree and Ph.D. Degree in computer science at Keio University, and the supported money from KLL. This research will never complete without this main source of money.

Thank you to Professor Hideharu Amano, he is my adviser of my Master Degree and Ph.D. Degree. I am very happy during researching in his laboratory and learn many things from him. With his help to organize the story of paper when submitted to SASIMI 2007, this research has been received outstanding reward, that is highly appreciated for me. I have discovered my new life style in the last two years before writing this thesis, that, I should stop working when I feel tried. I always tried to make everything perfectly in the pass and worked hard all the time. That is not so good because human did not born for just only working. I also found the fact that, the greatest wisdom is in simplicity. Everything is formed by simple in the natural. We can use simplex method to solve problem, but we will introdue more problem when using complex method. Thus, this thesis shows one of my simplex method to solve a problem in the computer science.

I would like to thank all of my doctoral committee members, Professor Motomichi Toyama, Professor Michita Imai, and Professor Tadahiro Kuroda for their careful reviews and comments to my thesis. Thank you to Dr. Yohei Hasegawa, Dr. Hiroki Matsutani and Dr. Satoshi Tsutsumi, they are very kind to me and help me to solve many problems. For example, translating and filling application forms in Japanaese language, going with me to many conferences, and explaining many things to me. Thank you to Dr. Masato Yoshimi, he helps me when I joined cell programing contest in 2008 and also helps me to setup PlayStation3 mechines using in my research. And, thank you to everyone who belongs in WASMII group, MuCCRA project, especially all users of my Black-Diamond compiler in this research.

Thank you to Dr. Itthisek Nilkhamhang and all Thai people who study in Keio-University at Yagami-Campus with me, they taught me about living in Japan. Finally, with the love of my parents, everything is going well. Thank you very much my father and my mother.

Best regards,
Vasutan Tunbunheng
August 2008

Abstract

Recently, dynamically reconfigurable architectures become popular and widely researched. It can achieve high computing performance as well as low power consumption, which is important in portable device such as MP3 players, mobile phones, portable game engines, and cameras for saving battery. Various architectures have been proposed and they require compilers to generate configuration data for evaluating the architectures at design time. Retargetability is important for a compiler or synthesis tool used for architecture exploration. If the compiler can be used for multiple target architectures, it can save the time to develop a compiler for every architecture. This kind of compiler is called "retargetable compiler". However, most of traditional retargetable compilers provide flexibility to customize the target architecture by selecting parameters and options. Thus, architecture designers still have to spend time to modify compiler code when the available options do not support the brand new architecture.

This research focuses on representation model which can be used to customize different target architectures without modifying the compiler itself. *Graph with Configuration Information (GCI)* is proposed to represent reconfigurable resources in Dynamically Reconfigurable Processor Arrays (DRPAs). The functional unit, constant unit, register, and routing resource can be represented in the graph as well as the configuration information.

A prototype compiler called Black-Diamond with GCI is now available for three different DRPAs. It translates data-flow graph from C-like front-end description, applies placement and routing by using the GCI, and generates configuration data for each element of the DRPA. Evaluation results of simple applications show that Black-Diamond can generate reasonable designs for all three different architectures. Other target architectures can be easily treated by representing many aspects of architectural property into a GCI.

Chapter 1

Introduction

In recent years, coarse grained Dynamically Reconfigurable Processor Arrays (DRPAs) have received attention as a flexible and efficient off-loading engine for various types of System-on-a-Chip (SoC). Some devices are commercially available [19, 27, 23, 25], and some of them have been integrated into digital appliances [15].

In order to achieve better area- and power-efficiency compared with traditional field-programmable devices such as FPGAs, they incorporate the following properties; (1) a simple coarse grained processor consisting of an ALU, a data manipulator, a register file and other functional modules is used as a primitive Processing Element (PE) of an array, and (2) dynamic reconfiguration of a PE array which enables time-multiplexed execution is introduced. Some of them use a multicontext facility which can change its configuration with a single clock cycle.

Unlike common FPGAs, in which the island-style structure using Look Up Tables (LUTs) with 4 or 5 inputs are commonly used, there are wide design choices in DRPAs, such as the PE granularity, the number of hardware contexts which can be switched dynamically, the total amount of wiring resource, and the size of PE array. Our performance evaluation results revealed that the optimal PE array size considering the area and power consumption varies in each application [8]. Thus, there is no all-around architecture in DRPAs, and the structure should be configurable or customizable for its main target application. Since DRPAs are embedded into an SoC, their architecture should be customized at the design time.

1.1 Motivation

For such customized DRPAs, the programming environment, especially a compiler, must also be customized. A retargetable compiler which generates configuration data from C-like description is the most important component for such customized DRPAs. Unlike compilers for common CPUs, the core of compilers for DRPAs is mapping functionality similar to the situation as placement and routing in common FPGA design tools. A directed graph which represents the target architecture is needed in such compilers. Unlike uniform FPGAs, DRPAs equip various types of component each of which has different restrictions. For

example, the interconnections between networks and components are often limited for reducing the hardware or corresponding configuration data. Even in the switching element, the flexibility is often limited. There are a lot of hardware restrictions on different architecture which must be supported or controlled by the compiler.

Although some retargetable compilers for DRPAs have been announced [18, 17, 10], it is difficult to treat brand new architectures. In those traditional retargetable compilers, the target architecture can be customized by changing parameters and options which controlled in program code. Thus, the compiler must be modified when the available options can not support the target architecture. It results to slow down the research on architecture exploration even using the retargetable compiler.

1.2 Objective and contribution

This research focuses on studying standard model which can be used to customize different target architectures without modifying the compiler itself. *Graph with Configuration Information (GCI)* is proposed to represent reconfigurable resources in dynamically reconfigurable architecture. A retargetable compiler can use the graph to represent various kinds of DRPA for mapping application into the different target architectures. By applying a technique to control selecting configuration at each node called *Disabling Configuration Testing (DisCounT)*, the hardware restriction in target architecture can be easily treated.

I have developed Black-Diamond compiler to map application based on the representation of GCI and it can be used for widely architecture exploration. Moreover, the configuration data can be generated to configure each element in form of RoMultiC [31] for fast configuration loading by using the GCI.

1.3 Thesis organization

The organization in this thesis is as follows: Chapter 2 introduces reconfigurable architectures and shows the problem of traditional retargetable compilers. Then, the GCI is proposed and used to represent a target DRPA. The coarse-grained multicontext reconfigurable architectures called *MuCCRA* as a case study are shown in Chapter 3. A simple retargetable compiler called *Black-Diamond*, was developed and used to map application into the architectures (Chapter 4). Finally, this work is concluded and discussed in Chapter 5.

Chapter 2

Survey

The reconfigurable device has ability to change its configuration to execute various kinds of application depending on demand. Basically, the device has programmable elements which can be reconfigured to form different datapath and computation. Two main advantages are high parallel computing and low power consumption compared to the same performance CPU. Recently, many researches focus on using the reconfigurable architecture as in off-loading engine for various types of System-on-a-Chip (SoC) for high performance computing, and especially using in the embedded or portable devices such as mobile phones, portable game machines, digital audio systems, DVD and MP3 players for battery saving propose.

2.1 Classification of reconfiguable architectures

A large number of reconfigurable architectures have been developed over the years by researchers and the industries. Reconfigurable architecture can be classified based on several different characteristics. This section shows some of the most distinguishing characteristics to classify the architectures.

2.1.1 Grain size

The granularity is the size of processed data (computational data and transferred data) which is common used in the entire system. There are two grain types: *fine-grained* and *coarse-grained*.

The fine-grained architecture has programmable elements working in bit-level called logic blocks which can be reconfigured to represent combination of logic gates. The input signals tend to select a configuration bit stored in a register to be an output signal as shown in Figure 2.1(a). Thus, the logic block is well known as Look-Up Table (LUT). Figure 2.1(b) and 2.1(c) show examples of configuration data to represent the different logic circuits. There are 3 input signals (a, b, and c) for selecting output data from a register. The configuration data which is a collection of logic bits 1 and 0 is delivered to fix the function of each LUT corresponding to the representing logic circuit. The LUTs are connected on reconfigurable interconnection network to form a large computational circuit. It is the most flexible architecture to implement any kind

of logic circuit since everything is controlled in bit-level. However, it requires huge configuration data to configure datapath and the LUTs itself, makes long time to load the configuration in term of millisecond.

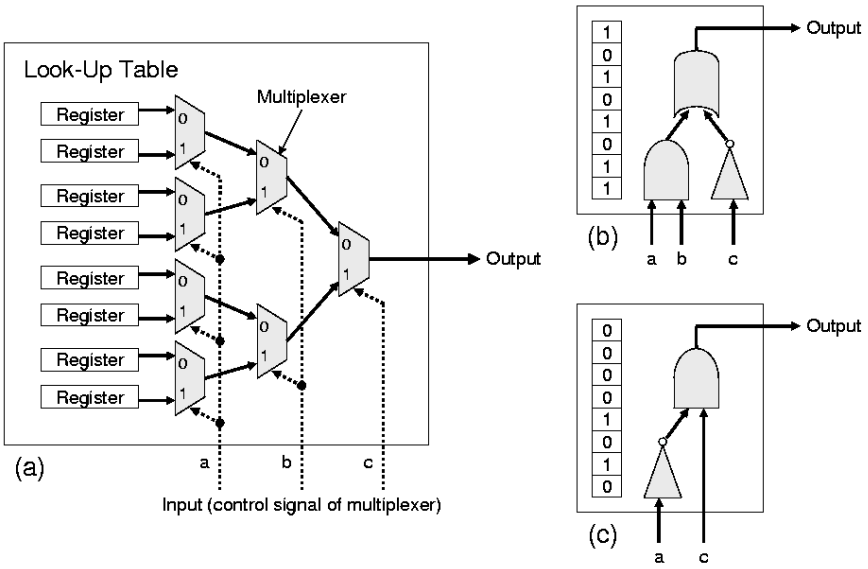

Figure 2.1: Example configuration of LUT (a) is LUT structure, (b) and (c) are examples of configuration corresponding to different logic gate functions

The coarse-grained architecture reduces the large size of configuration data by providing reprogrammable ability to process data in word-level. Every bit in the word shares the same control signal, thus, it requires less configuration data to change computational operation and perform datapath that allows fast configuration. Moreover, the arithmetic logic is optimizing for each operation leaded to low silicon-area implementation and also low power consumption. However, it has less flexibility when compared to the fine-grained architectures since the flexility was limitted by only the available operations.

2.1.2 Static and dynamic reconfiguration

The coarse-grained reconfigurable architecture and the fine-grained reconfigurable architecture can be classified whether they are statically reconfigurable or dynamically reconfigurable.

In the static reconfiguration, the configuration can not be changed during execution. It should provide large amount of the reprogrammable hardware to map large application. If the provided hardware is too small, the application can not be mapped into the device.

The dynamic reconfiguration is proposed as a relaxed solution to map the application into small size of hardware. The large application is divided into many small parts and only a executed part is configured into the hardware. When another part is required to be executed, it can be configured to replace

the old part into the same hardware. The intermediate computational data can be stored in register or memory module for transferring to the other parts during the time multiplex execution.

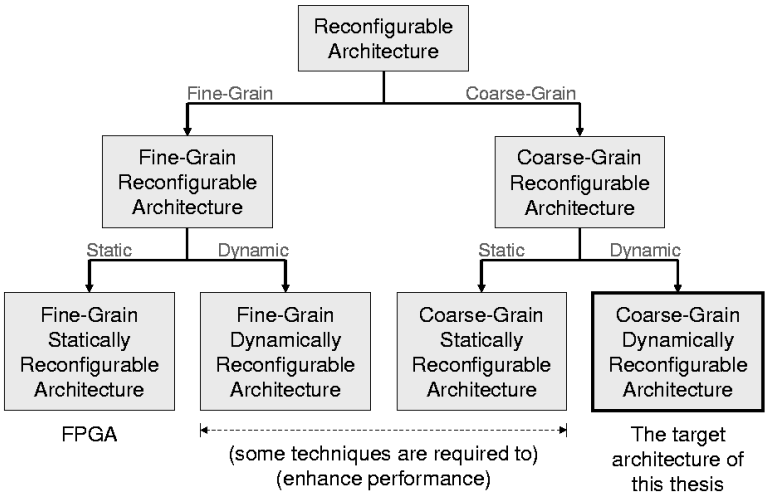

Figure 2.2: Classification of reconfigurable system

The reconfigurable architecture can be classified by the grain sizes and the types of reconfiguration into 4 groups as the diagram shown in Figure 2.2. This thesis focuses on only the coarse-grained dynamically reconfigurable architecture which discussed in the next sections.

The most popular fine-grained statically reconfigurable architecture is Field Programmable Gate Array (FPGA). The FPGA is consisting of a set of LUTs connected on global routing resources to form datapath of computational circuit. It can be reconfigured to execute different applications in bit-level and exchange data with outside of the chip by using I/O cells at the chip boundary.

The fine-grained dynamically reconfigurable architecture requires large configuration data, thus, it is not suitable to switch the context frequently. The performance can be enhanced by dynamically reconfiguring only a part of the entire chip (partial reconfiguration technique).

The coarse-grained statically reconfigurable architecture is not flexible and always requires large number of programmable elements to map application. In some FPGA architectures, they include a certain number of coarse-grained functional units instead of implementing the circuits on a large number of LUTs. For example: multiplication units or memory modules.

2.2 Coarse-grained dynamically reconfigurable architectures

Various kinds of coarse-grained dynamically reconfigurable architectures are developed as shown in the table 2.1. This Section shows the criteria to classify the architectures based on the classification in [4].

Table 2.1: Today's coarse-grained dynamically reconfigurable architectures

Name	Company	Reference
CS2112	Chameleon	[29]
DAPDNA-2	IPFlex	[28]
DRP-1	NEC electronics	[20]
FE-GA	Hitachi	[14]
Xpp-64	PACT	[24]
D-Fabrix	Elixent	[26]
Kilocore KC256	Rapport	[16]
ADRES	IMEC	[32]
PCA	-	[21]

2.2.1 PE structure

The coarse-grained dynamically reconfigurable architecture is consisting of an array of Processing Elements (PEs). A PE provides one or more functional units such as ALU or data manipulators, register files, and reconfigurable switches. Those functions and interconnection can be dynamically changed which allows time multiplex execution.

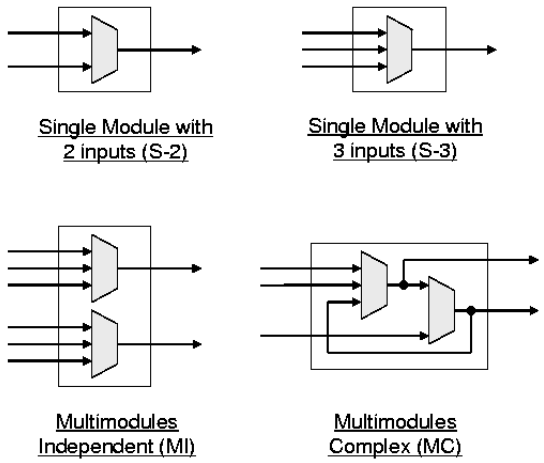

Figure 2.3: The example of different component structures inside PE body

Figure 2.3 shows the example of different component structures of PE body in the surveyed architectures. When the PE has only one component, it can be classified whether 2 inputs (S-2) or 3 inputs (S-3). When the PE includes multiple components, it can be classified whether the components can be connected to each other (MC) or work independently (MI). Some PE structures may include register files as the component (R) or are not any include register

file (N). The classifications of PE structures can be shown as Table 2.2 with its different grain size.

2.2.2 Dynamic reconfiguration

The datapath and functional unit are controlled by loading configuration data from the context memory. The configuration data for configuring several PEs must be loaded from a single on-chip memory module, and usually takes long configuring time. This kind of configuration method is called "delivery reconfiguration". On the other hand, the configuration data can be distributed to store in multiple memory modules, and entire configuration of PE array can be load in a clock. The configuration of PE array is called context related to the same context pointer broadcasting to all memory modules. This kind of configuration method is called "multicontext reconfiguration".

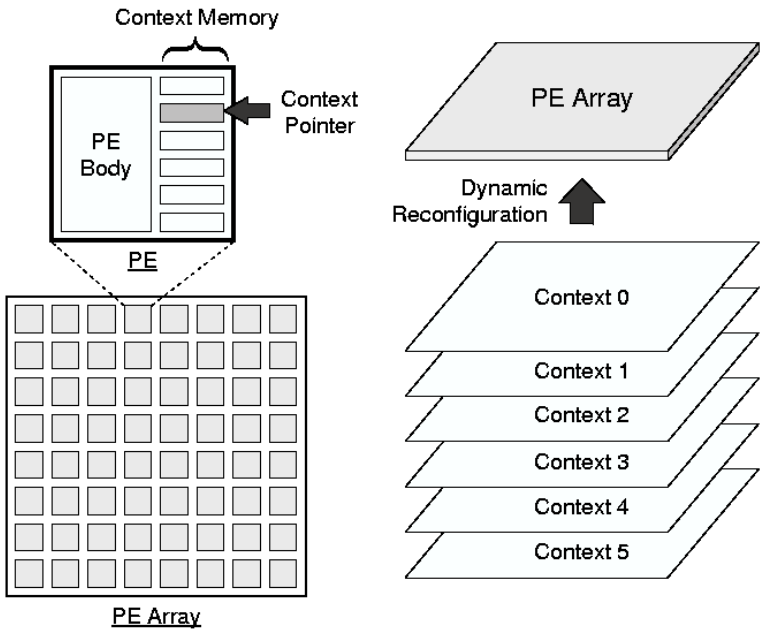

Figure 2.4: Multicontext reconfiguration

Figure 2.4 shows a PE array consisting of PEs with local context memories. By broadcasting the same context pointer to all PEs, configuration of the entire PE array can be changed in parallel. Although context switching can be done with a clock cycle in the multicontext devices, the area of each PE is increased with the distributed context memory.

The classification of surveyed architectures can be shown as the last column of Table 2.2. The multicontext reconfiguration architecture is followed by maximum number of contexts which can be stored in the local context memory. For the Kilocore KC256 architecture, the configuration data can be transferred from outside of the chip by using virtual pipeline with a powerful configuration bus.

Table 2.2: PE structure

Name	Bitwise	PE body	Register file	Reconfiguration
CS2112	16 / 32	S-2	N	Multicontext (8)
DAPDNA-2	32	M-C	N	Multicontext (4)
DRP-1	8	M-C	R	Multicontext (16)
FE-GA	16	M-I	N	Multicontext (4)
Xpp-64	24	M-I	N	Delivery
D-Fabrix	4	S-2	N	Delivery
Kilocore KC256	8	S-2	R	Multicontext / Delivery
ADRES	32	S-3	R	Multicontext (27)
PCA	4 / 8 / 16	S-3	R	Delivery

Thus, the number of context is unlimited.

2.2.3 Homogeneous and heterogeneous architecture

A certain number of PEs are connected with reconfigurable interconnection network for exchanging data between the PEs. A typical structure is square mesh. The examples of interconnection structures are shown in Figure 2.5. The number of PEs and interconnection structure are suitable for the different application domains. If the number of PEs becomes large, it allows more parallel function to execute in a clock. However, the size of PEs is limited by the silicon die area and more latency delay for transferring data via long distance wire.

The PE array can be classified whether it is homogenous (Homo) or heterogeneous (Hetero). Homogeneous means that all PEs are the same structure, while more than two types of PEs are used in heterogeneous structure. The classification result can be shown as Table 2.3.

Table 2.3: Interconnection structure

Name	Number of PEs	PE array
CS2112	108	Hetero
DAPDNA-2	376	Hetero
DRP-1	512	Homo
FE-GA	32	Hetero
Xpp-64	64	Homo
D-Fabrix	576	Homo
Kilocore KC256	256	Homo
ADRES	64	Hetero
PCA	64	Homo

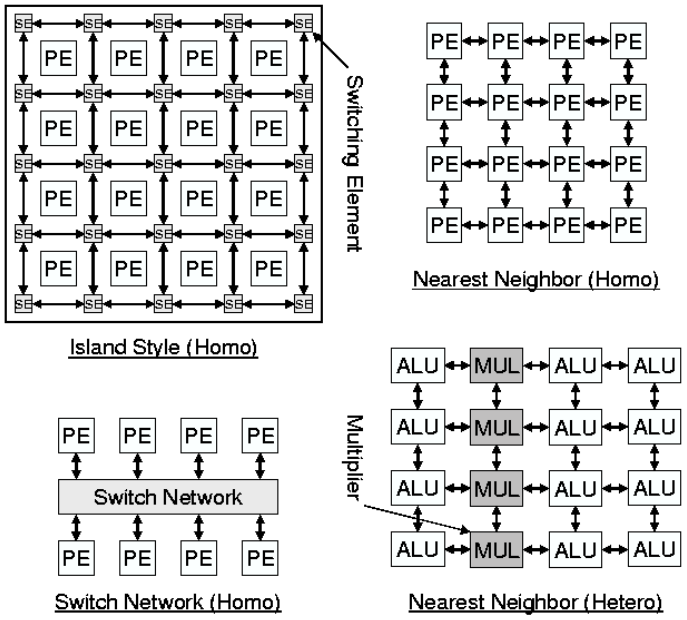

Figure 2.5: The examples of different interconnection network

2.3 Overview of the surveyed architectures

This Section shows the overview of surveyed architectures without detail since they are not related to this research.

2.3.1 CS2112

CS2112 is a processor-based reconfigurable architecture for high-performance telecommunication and data-communication. A RICS core, reconfigurable fabric, a fast bus, memory system, and I/O are built in a single chip. Figure 2.6 gives the structure of Chameleon CS2112 chip.

A 32-bit RISC core is used as a host processor to schedules computation intensive tasks onto the PE array. The PE array is consisting of 108 PEs called Data-Path Units (DPUs). The ALU inside the DPU can be dynamically reconfigured to execute one of eight instructions. The 108 DPUs are divided into four slices and each slice is partitioned into three tiles. In each tile, there are nine DPUs, of which seven are 32-bit ALUs and two are 16-bit multipliers. The configuration of DPU can be changed by loading from ConTroL Unit (CTL) which can store eight instructions at maximum. In addition, there are 8Kbytes of Local Memory (LM) for each slice which can read/write 32-bit data in two clock cycles. Inside the chip, a high speed bus links the core, PE array, main memory, and I/O module together. The configuration data is stored in the main memory and loaded to configure the PE array at runtime by DMA.

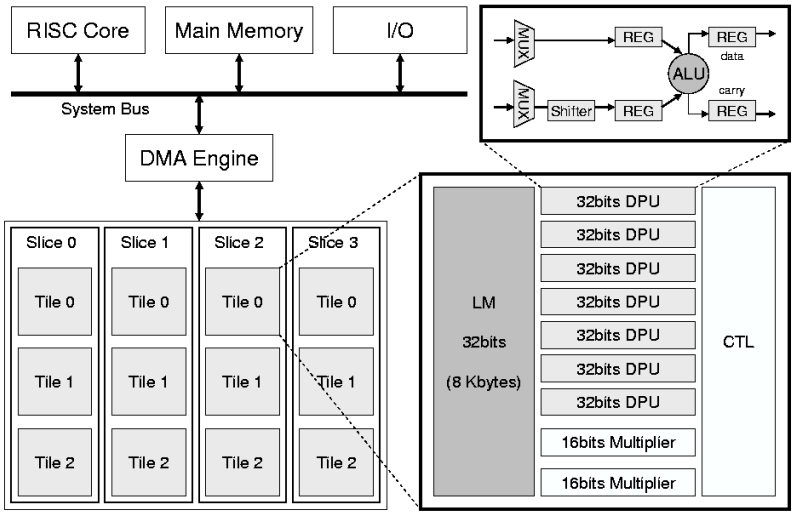

Figure 2.6: Hardware model of Chameleon CS2112 chip

2.3.2 DAPDNA-2

The DAPDNA-2 is the general purpose dynamically reconfigurable processor for commercial usage. It is a dual-core processor consisting of a custom 32 bits RISC core called Digital Application Processor (DAP), and a two dimensional PE array called Distributed Network Architecture (DNA). Both DAP and DNA are synchronized and co-work at the same clock frequency (166 MHz). The system includes 8 KB of instruction cache and 8 KB of data cache.

The DAP and DNA are connected with the peripheral of processor via system bus as shown in Figure 2.7. Inside the DNA, there is six PE arrays called segment with size 8x8 PEs. It provides rich interconnection between the PEs to guarantee operational speed of 166 MHz, regardless of the size and complexity of algorithms. There are 56 multiplier and 32 RAM elements, with 16 KB of memory each, (totaling 512 KB). This reduces the number of accesses to external memory that tends to be bottleneck. It also has a DDR-SDRAM interface to store large amount of data.

2.3.3 DRP-1

Dynamically Reconfigurable Processor (DRP) developed by NEC in 2002 can be shown as Figure 2.8. It provides 16 hardware contexts inside the chip. The context can be switched within a single clock by Instruction Pointer (IP) provided from State Transition Controller (STC). To configure the context, the current configuration memory of DRP-1 is accessed as a common memory, that is, configuration data is written by specifying address of the memory. Thus, it requires thousands of clock cycles to load the configuration data. It is connected in the system via PCI interface. There are 8 tiles and each tile consisted of 8x8 PE array controlled by STC at the center, 8 HMEMs (Horizontal Memory), and

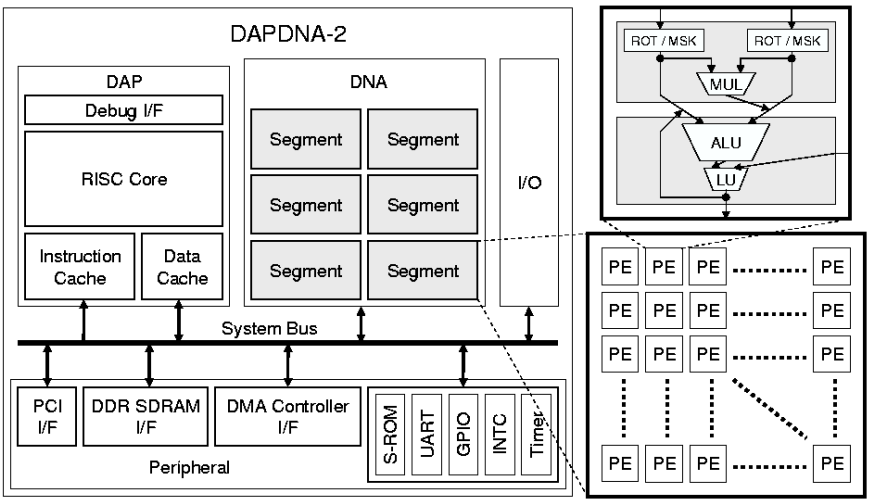

Figure 2.7: Hardware model of DAPDNA processor

16 VMEMs (Vertical Memory) at boundary of the PE array.

Each PE is consisting of ALU, DMU (Data Management Unit), register file, context memory, and inter-PE connections. Source and Destination operands can either come from or come to its own register file or other PEs. The interconnection network for transferring data between PEs is island style interconnection. There are 16 data buses for transferring pair of 8 bits data input from the column buses to a PE and 8 data buses for transferring 8 bits data to the next column bus. There is another island style interconnection network for transferring control or condition flag signal separated from the data network. Two flag bits can be picked in and picked out at the PE.

2.3.4 FE-GA

The FE-GA consists of 24 PEs in ALU type, 8 PEs in MLT type, 10 load/store (LS) cells with local RAM, a SEQuential Manager (SEQM), a ConFiGuration Manager (CFGM), a crossbar switch cell (XB), and a system bus interface as shown in Figure 2.9. Each PE is connected to its neighboring four PEs (north, south, east, and west), thus, the wire delay is short and can be executed with high frequency. The configuration data for each ALU, MLT, LS, and XB are managed by the CFGM and the context can be switched under control of the SEQM. The SEQM generates a switch trigger using its built-in counter or comparison result on the ALU.

Each PE in the ALU type consists of a general ALU (16 bits), a shifter, a data through logic block circuit, and an input delay logic circuit. All components are operated simultaneously without blocking each other. Each PE in the MLT type consists of a general multiplier, a data through logic block circuit, and an input delay logic circuit. Each LS cell can be connected to the XB, bus interface, and dual-port RAM (CRAM). Since the LS cells can generate load and store

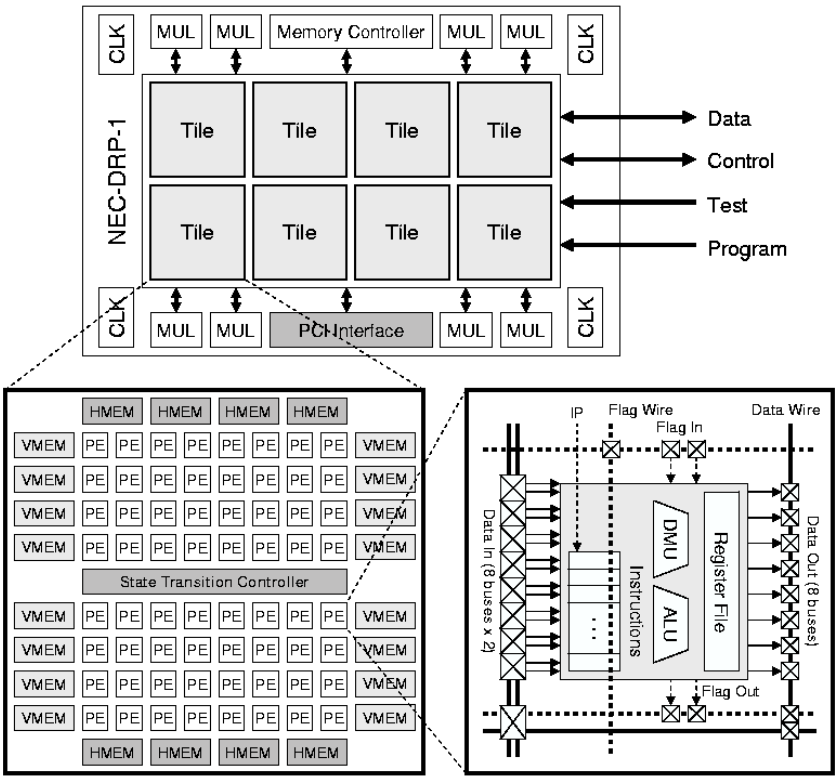

Figure 2.8: NEC-DRP-1 structure

addresses without using the ALU, the ALU can be saved for user applications.

2.3.5 Xpp-64

The eXtreme Processing Platform (XPP) is reconfigurable architecture optimized for parallel data stream processing where flexibility and fast reconfigurability are demanded. The XPP has similarities with other coarse-grained reconfigurable architectures which are designed and optimized for stream-based applications. However, the essential difference is the automatic packet-handing mechanism. All element objects communicate through a packet-oriented network, which route two types of packets: data packets and event packets. Configuration data is loaded from an external RAM into internal cache controlled by configuration manager, then, it configures the elements. As soon as an element is configured, it can start its operation if data is available.

An XPP array, shown in Figure 2.10, consists of a relatively small number of different components on homogenous structure. Each PE is integrating three types of components working independently: ALU which performs typical functions (suck as multiplication, addition, comparison, shifting), Back Register (BREG), and Forward Register (FREG). The BREG provides vertical paths from bottom to top of an ALU that can be used for addition, barrel shifting, and

normalization. The FREG provides routing paths from top to bottom and a specialized ALU that performs data stream control like multiplexing and swapping. The PE rows are connecting with horizontal routing channels between the rows. The RAM blocks are dual-ported, allowing simultaneous read and write, and have a typical size between 512 and 2K words. The PE array of XPP can exchange data with the outside of the chip via I/O blocks. It is two ports of an I/O element. The packets handling is performed through an asynchronous ready/acknowledge protocol.

2.3.6 D-Fabrix

In D-Fabrix, it contains two distinct types of component inside PE, which are ALU and multiplexer. The ALU is used for datapath purpose to process word-based data and the multiplexer is used as control block to manipulate data bits. Moreover, the multiplexers can be implemented as gates which are subset of the ALU functionality.

Figure 2.11 shows PE array and PE structure of which consists of ALU and multiplexer components. The D-Fabrix uses chess interconnection network for transferring data inside the PE array. In the PE, data from network or carryout of the ALU itself can become control signal of the multiplexer. It allows using the multiplexer as logic functions shown in Table 2.4.

Table 2.4: Logic funxtions constructed from multiplexer

Function	Implementation	Function	Implementation
0	A ? 0 : 0	1	A ? 1 : 1
A & B	A ? B : 0	$A \mid B$	A ? 1 : B
A & !B	B ? 0 : A	$A \mid !B$	B ? A : 1
!A & B	A ? 0 : B	$!A \mid B$	A ? B : 1
A	A ? 1 : 0	B	B ? 1 : 0
!A	A ? 0 : 1	!B	B ? 0 : 1

Eight logic functions can be implemented by using multiplexer. Since the multiplexers are smaller and faster then ALUs, there is a net reduction in both area and delay. The synthesis process needs only target the ALUs, and placement can then determine whether a given multiplexer in the netlist maps to an ALU or a switchbox multiplexer.

2.3.7 Kilocore KC256

Kilocore is a reconfigurable computing device comprised of two main components: (1) a PE array and (2) a peripheral called "wrapper" with interface logic, controllers, and memory as shown in Figure 2.12. It uses packet-based I/O model for exchanging operated data and configuration data with the host computer. All packets are 32 bits wide and are preceded by header data to identify the type of packet, destination, and other information.

The operated data can be stored up to 128 bits at the input controller for executing in parallel and the output controller can produce up to 4 result

packets to transfer 128 bits output data. The configuration packets are stored in configuration cache inside the configuration controller for reconfiguring a row of PEs within a clock when needs. The PE array consists of many rows of PEs called "stripe" with registers between the rows, thus, it can support pipeline execution. The interconnection network allows a PE can read data only from some of the nearest neighbor PEs in the stripe, not all of them.

2.3.8 ADRES

ADRES (Architecture for Dynamically Reconfigurable Embedded System) includes PE array which is divided to be VLIW part and RC array part. There are VLIW (Very Long Instruction Word) processor as the main processing unit and a tightly coupled PE array for purpose of acceleration. The PE array is well known as array of Reconfigurable Cells (RCs) which is optimized for data-flow while the VLIW part is optimized for control and load/store operations. The PE array size is 8x8 PEs which the first row was implemented as the VLIW part connecting by using horizontal data bus as shown in the Figure 2.13.

The VLIW part includes up to eight functional units organized in a row. The attached array is composed of several rows of RCs organized in a matrix form implemented as orthogonal interconnection network. The RCs at VLIW part do not provide the register file while other include register file (RF). Data exchange with external memory is through the default path of the VLIW processor. There are global buses for each row or column which allows a RC can get input data directly from each of its horizontal or vertical neighbors.

2.3.9 PCA

The Plastic Cell Architecture (PCA) is a special class of dynamically reconfigurable devices each of whose basic cell can be reconfigured individually. The device has a uniform array structure of basic cells called "PCA Cells", which work independently to realize distributed processing. It includes "Plastic Parts" and "Built-in Part" respectively as shown in Figure 2.14.

The plastic part consists of basic cells as shown in Figure 2.15. It has an ALU and a register surrounded by wires and multiplexers. Each basic cell is connected to four-direction neighbor cells via multiplexers located in four side of the basic cell.

The built-in part controls data flow and reconfiguration of the plastic parts inside or outside of the cell. It accepts commands from the accompanied plastic part or other cells. Each PCA cell is connected to the neighbor cells and this provides scalability of a PCA device. With self-reconfiguration, PCA can create, copy, or delete circuit modules to perform flexible and adaptive processing.

The possible architecture parameters include reconfigurable resource in each basic cell (ALUs with various input word length and supported operations), wire resources in the array (structure and bandwidth), and mechanism for configuration delivery.

2.4 Survey on retargetable compilers

Generation of configuration data costs large man-hours without any development tools dedicated for the architecture. Retargetability is important for a compiler or synthesis tool used for architecture exploration. The recent compilers or synthesis tools for reconfigurable devices are developed and targeted only for a specific architecture, they do not suit for a use of developing new reconfigurable architecture. A architecture designer must spend a lot of time to create a compiler for each architecture and modify the compiler when the architecture is re-designed. The design time can be saved by using a retargetable which can customize the target architecture.

Even, there are many reconfigurable architectures have been proposed, there are only a few works on the retargetable compiler. Many researches tend to develop a compiler for only a single target architecture. This chapter shows example of retargetable compiler for coarse-grained reconfigurable architecture.

2.4.1 DRESC compiler

IMEC develops a coarse-grained reconfigurable array called *Architecture for Dynamically Reconfigurable Embedded Systems (ADRES)* with the tools to design an application specific processor instance based on an architecture template [5]. This enables the designer to combine an arbitrary number of functional units, interconnects and register files.

Tool flow

Figure 2.16 depicts the proposed tool flow for architectural exploration. It consist of basically three parts: (1) Compilation and Assembly providing binary files and a compiled instruction level simulator, (2) Synthesis providing gate level netlist and physical characteristics needed for power calculation and (3) Simulation to obtain power and performance figures.

The *Compile and Assemble* part transforms the ANSI-C application code into an optimized binary file. The code is first processed by the IMPACT [6] frontend that performs various ILP optimization and transforms it into an intermediate representation called Lcode. The DRESC2.0 compiler in Figure 2.16 reads the Lcode, performs ILP scheduling, register allocation and modulo scheduling for CGA mode and generates the optimized DRE files. These files are used to create a high level simulator which provides basic performance parameters such as instructions per cycle (IPC). In addition, the Assembler tool creates the binary files needed for the cycle true Esterel and ModelSim simulations.

The ADRES instance XML description is transformed into VHDL files and is synthesized in the *Synthesize* part. *Synopsys Physical Compiler v2004.12-SP1* [2] is used to create the netlist for 90nm CMOS library, from which area, capacitance and resistor values are extracted to be used multiple times.

In the *Simulate* part three different simulators are used. The compiled ISA simulator (marked as A in Figure 2.16) provides the performance numbers.ModelSimRTL simulator is used to simulate the generated RTL VHDL files at highest level of hardware accuracy and to obtain the switching activity figures needed for RTL power calculations. The Esterel simulator [1] is based on the Esterel synchronous language dedicated to control dominated reactive

systems and models the entire ADRES instance as a state machine. The advantage of the Esterel simulator is the reduction in time to simulate a benchmark compared to ModelSim RTL simulations, as it is 8 - 12 times faster reducing overall time by a factor of 3 - 6.5 depending on the application. In addition, it is the behavior level reference model for the core architecture available much earlier than the verified RTL.

Facilitated by its language, the Esterel simulator has the same structure as the actual implementation. Thus by enhancing it with bit toggle counting functions written in C, we are able of capturing the signal activity statistics of almost all relevant connections. This switching activity is what an RTL HDL simulator also generates for the power analysis tool. The match between the generated data is established by using the same signals as defined in the XML architecture description file.

The switching activities obtained after simulations are annotated on the gate level design created in the synthesize part. The toggling file and the gate level design are used by the *PrimePower v2004.12-SP1* of Synopsys [2] to estimate power.

Architecture exploration options

Fourteen different architectures are constructed from 7 different interconnection options as depicted in Figure 2.17. The architectures are described in the XML architecture file, which will be used in the tool flow described earlier. The simplest interconnection option is *mesh*. It creates connections between destinations and sources of adjacent FUs in horizontal and vertical directions. The *mesh_plus* interconnection is an extension of mesh with additional connections that routes over the neighboring FUs. The *reg_con1* and *reg_con2* options create diagonal connections between neighboring FUs and RFs. This creates additional routing and the possibility of data sharing among FUs connecting directly to the RFs. The difference is that *reg_con1* receives data from its neighbors while *reg_con2* sends data to them. The *extra_con* option offers an extra FU bypass to enable parallel processing and routing. The *enhance_rf* option has shared read and write data ports from the global DRF to the vertically connected FUs. This is beneficial for communication as the global DRF is used as communication medium in the array, however it increases power consumption due to the frequent accesses to the global DRF. Splitting up the powerhungry DRF into smaller, local DRFs was one of the main features of power reduction. The option *has_busses* determines if predicate and data busses of 1 and 32 bits wide respectively are implemented in the design. By combining various interconnection options 14 different architectures are created as noted in Table 2.5.

2.4.2 Kyoto-University's compiler

For the retargetable compiler from Kyoto-University [9], the target architecture can be customized by changing parameters to control: size of PE-Array, number of connections between PEs, bitwidth and ALU instructions, for evaluating device on PCA [12][22] platform. Given an application in C-language, the tool automically executes data-flow analysis, technology mapping, and layout synthesis.

Table 2.5: Architectural exploration selection options

Architecture	mesh	mesh plus	reg con1	reg con2	extra con	enhance RF	has busses
mesh	X						
mesh_plus	X	X					
xtra_con	X	X			X		
reg_con1	X	X	X				
reg_con2	X	X		X			
reg_con_all	X	X	X	X			
enh_rf	X	X				X	
busses	X	X					X
arch_1	X	X		X	X	X	
arch_2	X	X	X		X	X	
arch_3	X	X		X		X	
arch_4	X	X	X			X	
all	X	X	X	X	X	X	X
ref	X	X	X		X	X	X

To enable description of applications easy, it is desired to support high-level language (such as C) for source code of compilation. Figure 2.18 shows a processing flow of proposed compiler. Input of the compiler is simplified C language that does not support all the C grammar. The compiler first converts a given C code to "GCC Tree" expression using a frontend of GNU Compiler Collection version 4.0 (GCC-4.0). The GCC Tree, which is intermediate expression used inside GCC, represents syntax trees of an input C code and architecture-independent optimizations are performed on the GCC Tree. Next, the compiler generates a Data Flow Graph (DFG), which represents a flow and dependency of data and control, from the GCC Tree. Then nodes in the DFG are assigned to ALUs to generate a netlist of ALUs. Finally, all the ALUs are placed and routed according to the netlist and configuration data is generated.

2.5 Summary

This Chapter shows survey of the coarse-grained dynamically reconfigurable architectures, which are the target architectures of this thesis. The configuration data structures are different cause by the differents reconfigurable resources on the architectures. It must be generated to evaluate those architectures on real applications at design time.

The retargetable compilers can generate configuration data for many target architectures by changing paramters and options. However, the flexibility is limited within the available options. For developing brand new architectures, the compiler must be modified to support the new design in architecture exploration. This is the reason why many researches tend to develop a compiler for only a single target architecture.

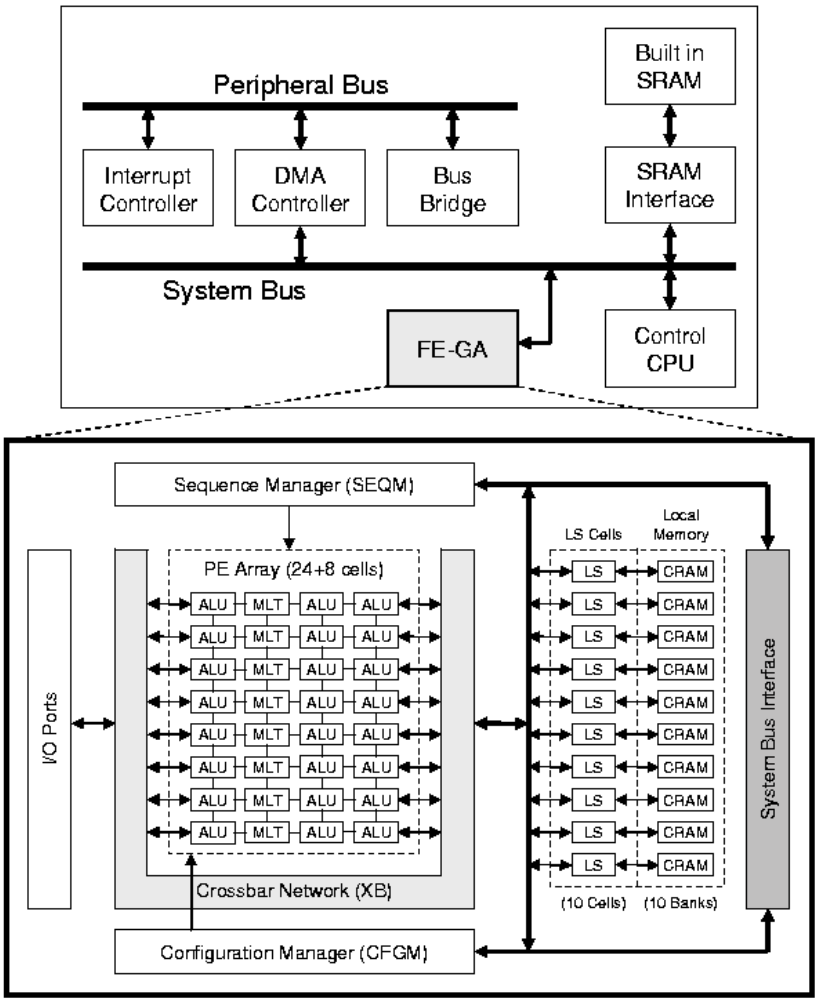

Figure 2.9: FE-GA structure

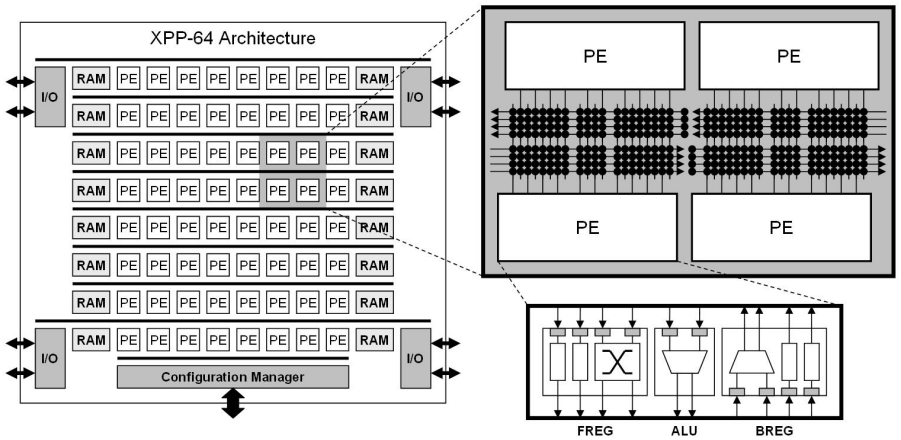

Figure 2.10: XPP-64 structure

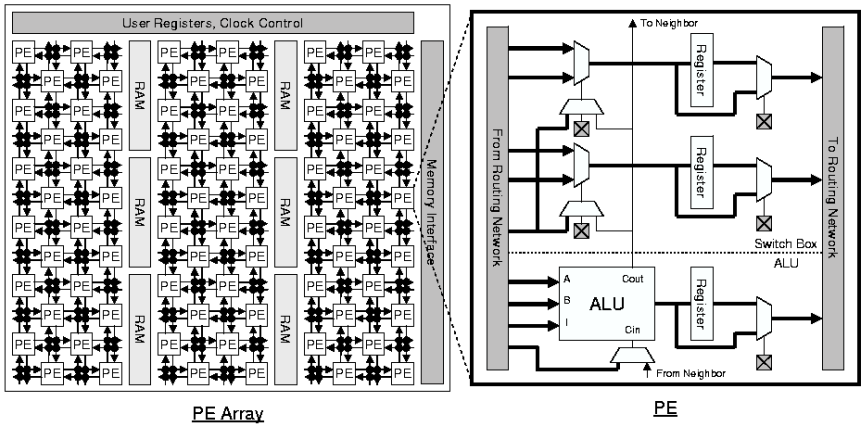

Figure 2.11: D-Fabrix structure

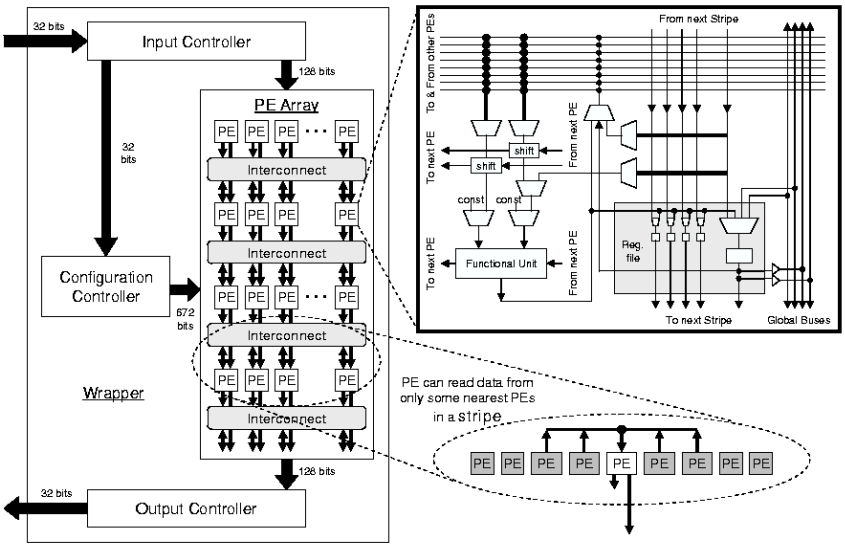

Figure 2.12: Kilocore structure

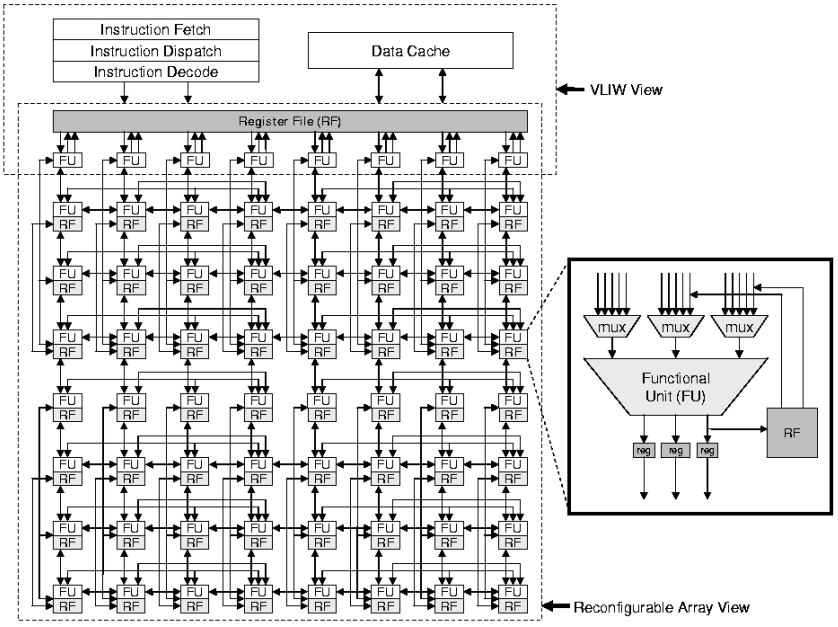

Figure 2.13: ADRES structure

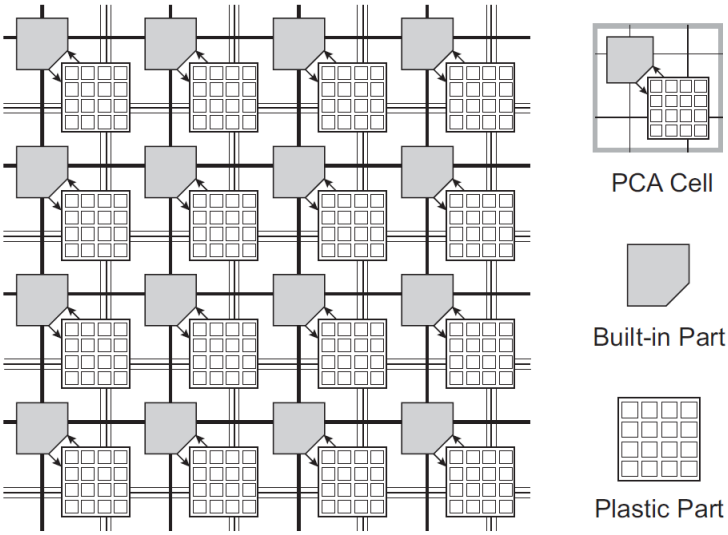

Figure 2.14: Array structure of PCA

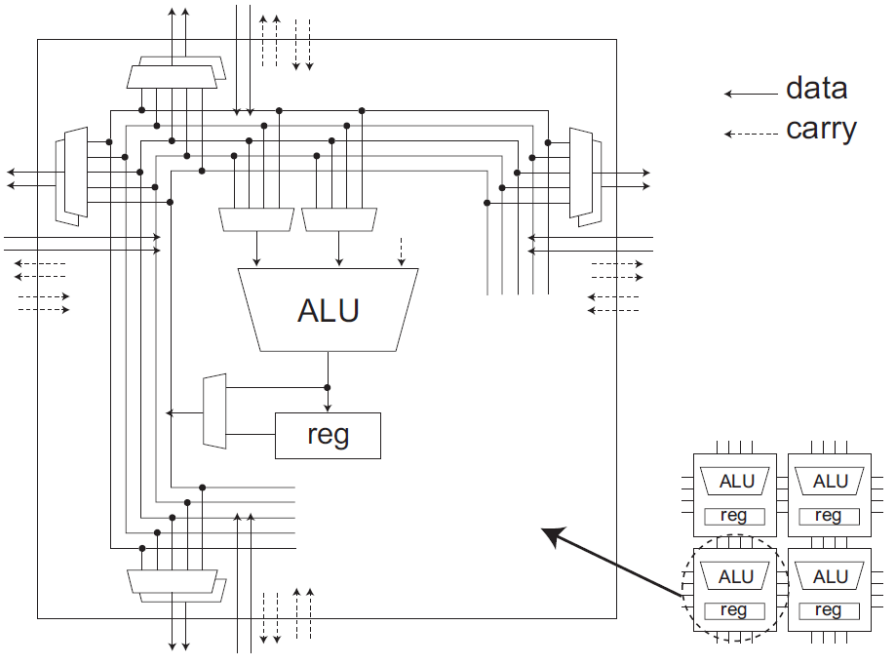

Figure 2.15: A structure of a basic cell

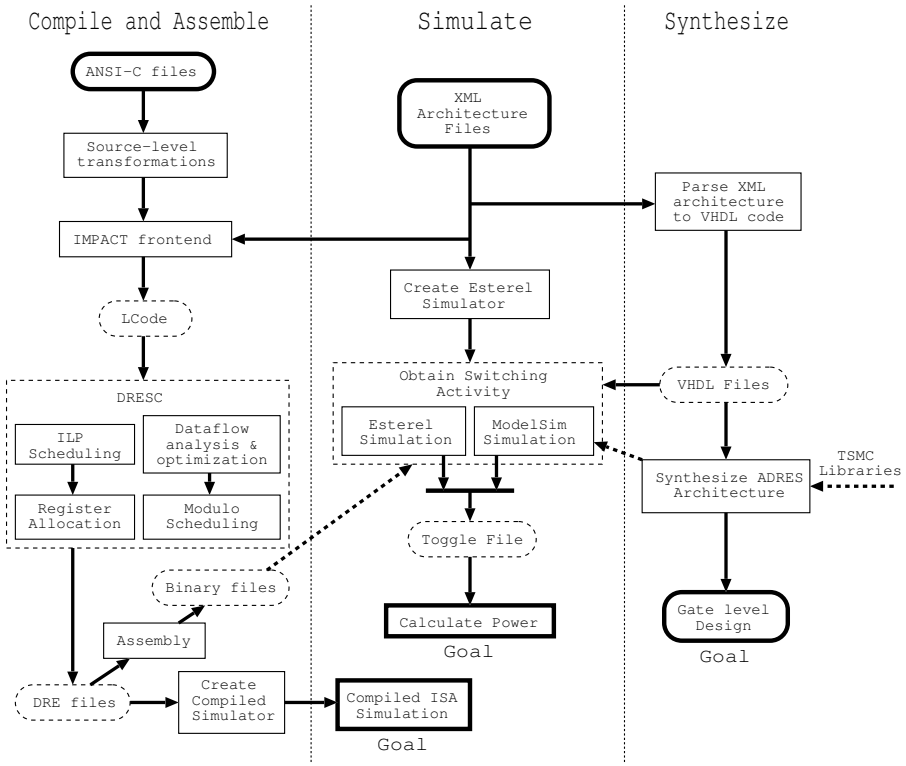

Figure 2.16: Overview of tool flow

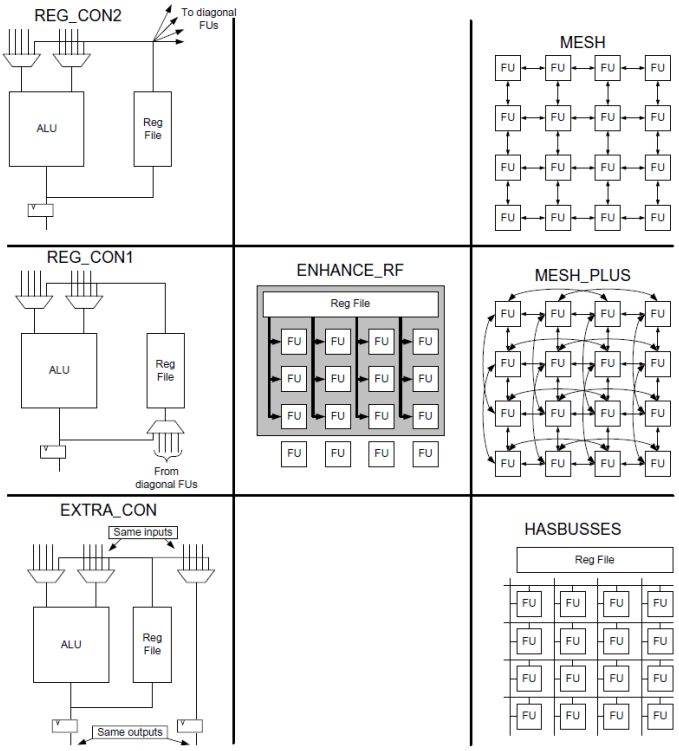

Figure 2.17: Interconnection options for architectural experiments

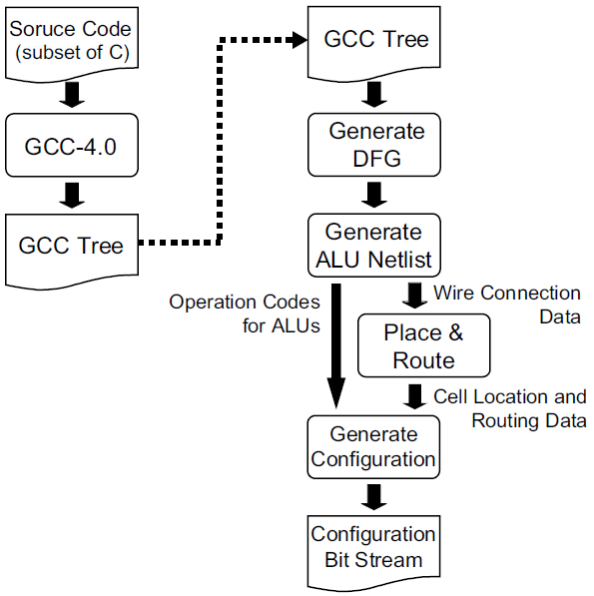

Figure 2.18: Overview of compile flow

Chapter 3

The GCI representation model

The target architectures can be customized by changing the providing parameters and options. For developping brand new architectures, the compiler must be modified to support the new design in architecture exploration.

We need standard model to represent different target architectures without changing the retargetable compiler itself. This chapter describes about the representation model used in this research and shows case study of using the model to represent different target architectures.

3.1 Approach

In order to propose the standard model, we must start from considering the requirement of representation model used for mapping application:

- In order to map application into target architecture, the compiler must generate configuration data to control reconfigurable resources on the device. The different resources are controlled by different configuration codes. The representation model must include all selectable configuration codes of the target architecture for generating the configuration data.
- The mapping process can be divided to be placement and routing. The computations are placed into the architecture by selecting a corresponding configuration code at functional units. Many routing algorithms requires the information of interconnection structure in form of directed graph which represents selectable links to perform datapath between the functional units.
- The different reconfigurable architectures have different hardware restrictions. For example, the number of computations is always limilted by the number of sharing hardware resources, thus, all reconfigurable units can not be configured to use the same hardware in parallel. The representing model must represent the restriction inside the target architecture.

By combining all of those requirements, I have an idea to represent the target architecture by using directed graph directly. The selectable configuration codes

and restriction to control hardware conflict are also added into the graph. Thus, the representation graph is called *Graph with Configuration Information (GCI)*.

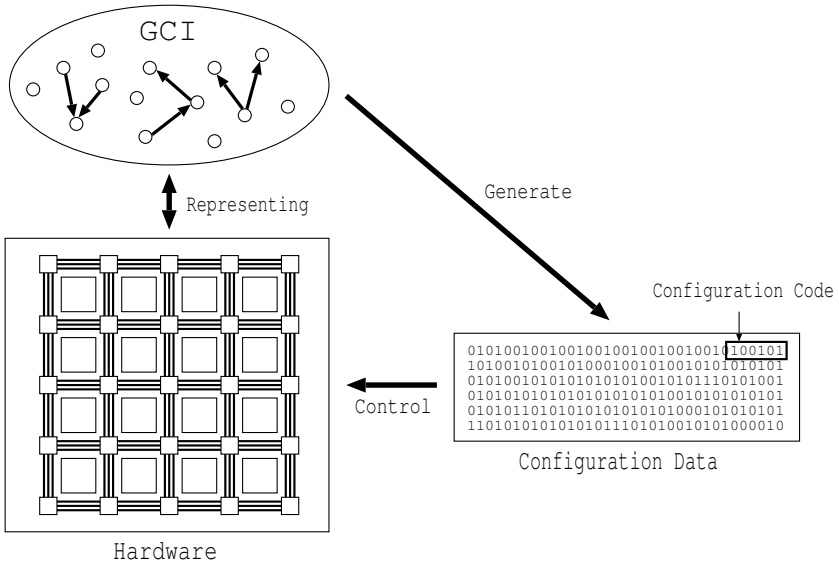

Figure 3.1: The relationship between hardware, GCI, and configuration data

Figure 3.2 shows the relationship between hardware, GCI, and configuration data. The hardware or target reconfigurable architecture is controlled by configuration data to change computaional or interconnection functions. The configuration data is a collection of configuration codes to select the diffrent functions. The bits length of the code can be vary depended on the number of selectable functions. For example, 2 configuration bits can be used to select 4 different functions, and 3 configuration bits can be used to select 8 different functions. In this research, both interconnection structure and selectable configuration codes of the target architecture are represented by the GCI graph in the same time. And, the configuration data can be generated by mapping application on the GCI.

3.2 Graph with configuration information

The GCI is directed graph consisting of nodes and links. The node is used to represent where configuration code can be changed in the configuration data to control the hardware function. The link is used to represent interconnection structure between the nodes where direction of transferred data can be changed. Thus, some nodes are used to represent only reconfigurability without connecting to the other nodes.

The directed links can be classified to be input link and output link. The input links are fixed at a node to represent all selectable configuration codes as shown in Figure 3.2(a). The configuration code, which is a collection of logic bits 1 and 0, is attached at each input link. Since only an input link must be selected at configuration data when mapping application, a node tends

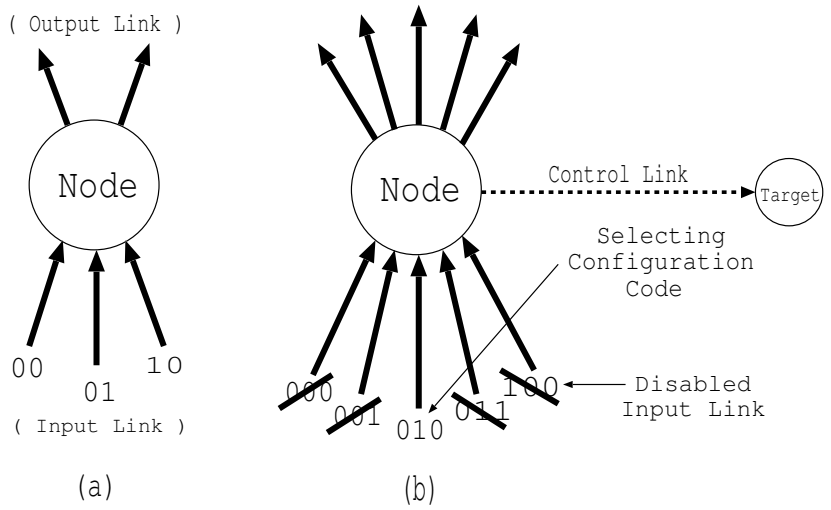

Figure 3.2: Example of GCI nodes (a) a node with 3 selectable configuration codes, (b) a node which selected a configuration code by disabling the other 4 input links and a control link to disable input links at the other node

to broadcast the same data to all output links corresponding to the selecting function. The un-selected input links are marked as disabled input links at the same node as shown in Figure 3.2(b).

3.3 Disable configuration testing

In order to embed the whole information of the target architecture into GCI, the restriction of the target architecture must be represented. We can add flexibility to represent the restriction by allowing selecting an input link to disable input links at the other nodes. It can be shown in the graph by using *control link* to connect source node and target node as shown in Figure 3.2(b). Each control link sends information whose input links of the target node are disabled. This information is depending on the selected input link of the source node. A control link is activated when an input link of the source node is selected temporarily and the corresponding inputs of the target node are disabled. A target node can be controlled by multiple source nodes. When multiple control links are activated, only control information which results at least one enabled input at the target node is accepted, otherwise, the selection of some source nodes must be canceled. Note that, the disabled link can not be used for placement or routing, thus, the target node must select an enabled input for generating the configuration data. In order to make the above explanation become clear, we define three *Disabling Configuration Testing (DisCounT)* rules to control every node in the GCI as followed:

- When a node selects an input link (by placement or routing), the other input links must be disabled by the node itself. This rule allows selecting only one input link at a node for generating configuration data.

- When a node selects an input link, it can send control information to disable input links at the other nodes. It can be applied to represent the hardware restriction of a target architecture. The examples will be shown on case study later.
- Since a target node can be controlled by multiple source nodes, there is a rule to avoid the situation that all input links at a node become disabled. At least one input link must be enabled.

3.4 Case study

Here, as a case study, an example of representing a dynamically reconfigurable processor called MuCCRA (Multi-Core Configurable Reconfigurable Architecture) based on the GCI is shown. Since MuCCRA is designed for investigating an optimized structure of DRPA for a given application, several prototypes with different structures have been designed. Three different MuCCRA structures: MuCCRA-1, MuCCRA-2 and MuCCRA-D are treated as targets shown in Table 3.1. Although the same size of array (4x4 PEs) is used in all architectures, the bit-width, number of contexts, interconnection structure, and configuration data structure are different.

Table 3.1: The difference between MuCCRA-1, MuCCRA-2 and MuCCRA-D

	MuCCRA-1	MuCCRA-2	MuCCRA-D
Bit-width	24 bits	16 bits	24 bits
Contexts	64	16	64
PE structure	Heterogeneous: including Multiplier PE	Homogeneous: All PE provides a Multiplier	Homogeneous: All PE provides a Multiplier
Interconnection	2 bi-direction	3 bi-direction	NN-network
Process	Rohm's 0.18um	ASPLA's 90nm	Rohm's 0.18um

3.4.1 Overview of MuCCRA project

The object of MuCCRA project is to develop a design methodology and framework which generate various types of DRPAs easily by selecting the specific parameters.

As shown in Figure 3.3, the final goal is evaluating the target customized architecture on both timing analysis and RTL simulation. It starts by reading architectural parameter file, and generates the Verilog-HDL descriptions of DRPAs. The fundamental DRPA architecture template is fixed, and the designers can generate their desired DRPAs by controlling parameters described in the parameter file. A simple test bench is generated for simulating the target architecture immediately. At the same time, architectural description file for a retargetable compiler called Black-Diamond which generates configuration data from C-like language is also created. Now, the generator can only generate a single core DRPA which can be used as an element of multi-core systems.

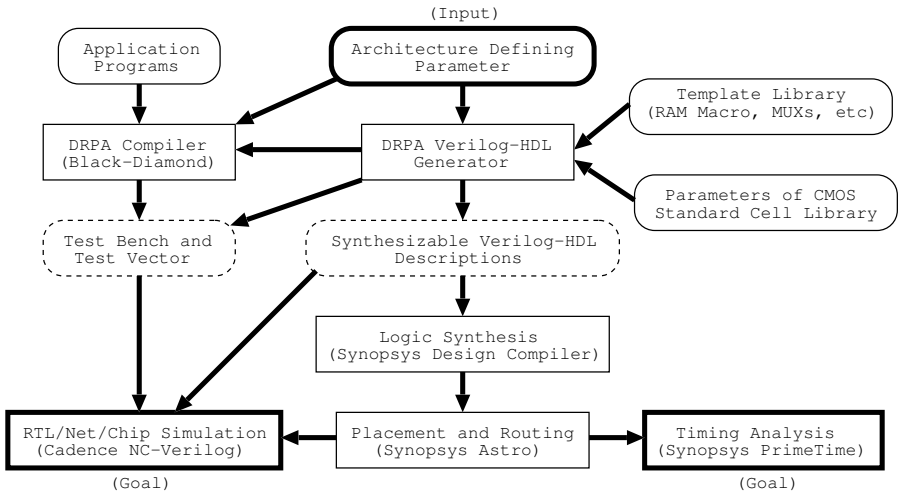

Figure 3.3: MuCCRA design environment

3.4.2 MuCCRA architectures

The basic building unit of MuCCRA architecture is a Processing Element (PE). The PE array structure of MuCRRA is parametrized by the size of PE array, bitwidth of data transferred between PEs, number of hardware contexts, selectable operations at each functional unit, and PE structure can be flexibly defined. Like a lot of existing DRPA devices, a data manipulator called Shift & Mask Unit (SMU), an Arithmetic Logic Unit (ALU), and a Register File Unit (RFU) are provided on PE as shown in Figure 3.4. A PE can exchange data with other PEs by surrounding global routing wires, and can exchange data between local ALU, SMU, and RFU. The flexibility of interconnection of PE structure can be defined with the number of selectors provided on inputs and outputs of the functional units.

Each PE equips local context memory which provides multiple sets of configuration data to control ALU, SMU, RFU, and interconnection. Context Switching Controller (CSC) broadcasts a context pointer to all PEs and a context is read out from the context memory according to the context pointer. This type of dynamic reconfiguration is called a multicontext scheme, and a lot of current devices support it. Since the number of contexts at the context memory is limited by area of silicon die, the configuration data which cannot be stored in the context memory is stored in the central configuration memory, and distributed to unused area of each context memory during the execution. This mechanism, called the virtual hardware, has been proposed and researched long time but rarely implemented in real chips. However, all MuCCRA chips provide this mechanism, and application which requires exceeded contexts number can be executed. For high speed configuration data distribution, a multicast mechanism called RoMultiC [31] is adopted.

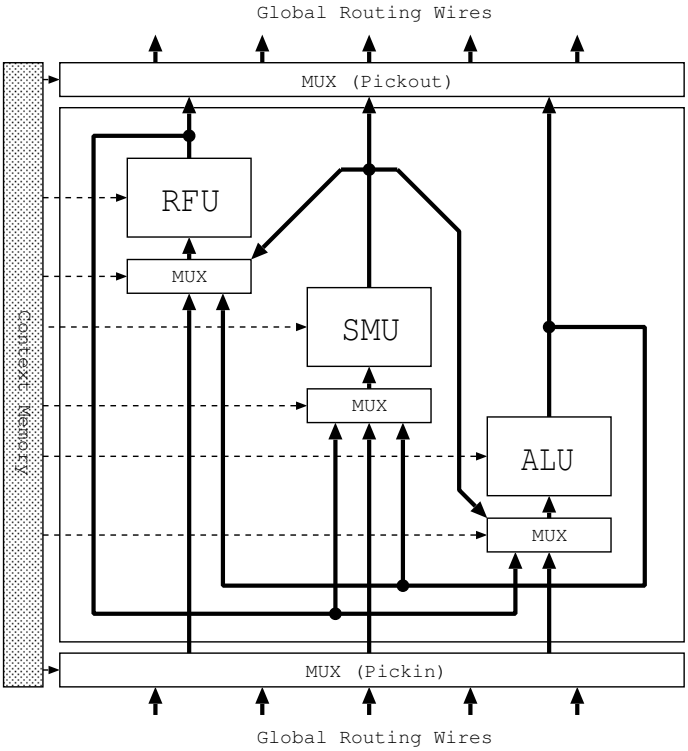

Figure 3.4: PE structure in MuCCRA architecture

MuCCRA-1

The first prototype, MuCCRA-1, was designed with Rohm's 0.18um process, and implemented on 5-mm square die with 189 I/O pads. An island-style interconnection structure like traditional FPGAs is adopted in MuCCRA-1 and MuCCRA-2. As shown in Figure 3.5, an island-style 2-dimensional interconnection is provided, and each PE is surrounded by programmable routing wire segments. On the intersection of a vertical and a horizontal channel, a Switching Element (SE) is placed. The SE is a set of simple programmable switches in which an incoming link is connected to the other SEs. Since it is designed for multi-media processing, the bitwidth of routing channel is 24 bits. Multiplier modules (MULs) are provided at the edge of the PE array since each PE does not include multiply operation, makes the PE array become hetrogeneous structure.

MuCCRA-2

MuCCRA-2 was implemented on 2.5-mm square die with 51 I/O pads. ASPLA's 90nm process was used. The main challenge of MuCCRA-2 is the reduction of the die area without degrading its performance. For this purpose, the bit-width and context size are smaller than those of MuCCRA-1. A context memory module is shared by two PEs and four SEs on chip layout for reducing the number

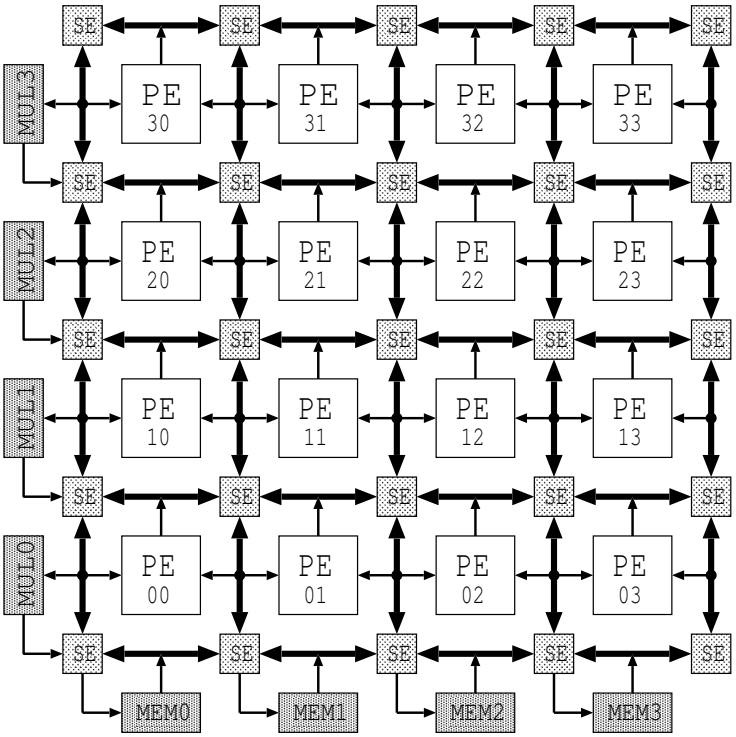

Figure 3.5: MuCCRA-1 array structure

of memory modules. On the other hand, multiply operations are provided in all PEs, since it appeared that the number of multiplier often dominates performance in MuCCRA-1. As a result, the array becomes a homogeneous structure. MuCCRA-2 uses a 16 bits architecture and 16 contexts can be held in the context memory. The interconnection capacity is also enchanced to improve the utilization ratio of PEs. Three routing channels are available on MuCCRA-2, while MuCCRA-1 has only two routing channels.

MuCCRA-D

Although the island-style adopt on MuCCRA-1 and MuCCRA-2 has an advantage of flexibility on PE-to-PE connections, there are two major problems: (1) the large delay time caused by passing multiple switches, and (2) the maximum operating frequency is degraded by the maximum delay of longest connection path on each context. MuCCRA-D uses direct interconnection between neighboring PEs as shown in Figure 3.7 instead. A register which stores the output data at each component before transferring to the other PEs allows MuCCRA-D executing at high clock frequency. However, the direct interconnection has disadvantage of taking a few cycles to transfer data to distance PEs. In order to reduce the number of hops for transferring data to distant PEs, one routing channel is connected on both horizontal and vertical direction to the next neighboring PE, while torus connection is available on only horizontal direction.

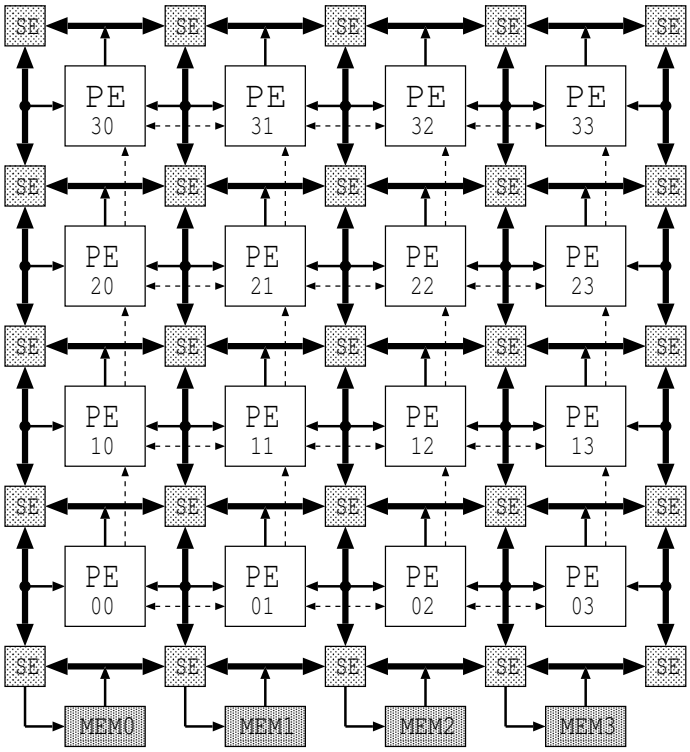

Figure 3.6: MuCCRA-2 array structure

The memory modules (MEMs) are connected on upper and lower side of the PE array, and the computation flow can go both up and down direction to exchange data between the memory modules, unlike MuCCRA-1 and MuCCRA-2 which provide the MEM module only on lower side of the PE array.

3.5 Representing GCI structures

This section shows the representing GCI structure of each component in the target MuCCRA architectures. First, the GCI of each component in DRPA (SE, PE, MEM, and MUL) are constructed. Some components require adding control links to represent hardware restriction. Next, they are connected in the same graph to represent datapath in a context. Finally, Several duplicated graphs of a context can be connected together to form a large GCI to represent entire of the multicontext architecture.

3.5.1 Representing each component of DRPA

The GCI structure to represent each component is constructed only by nodes and links. There are nodes to represent input ports and output ports of the components. It is used to connect with the other components to form a large GCI structure of a context.

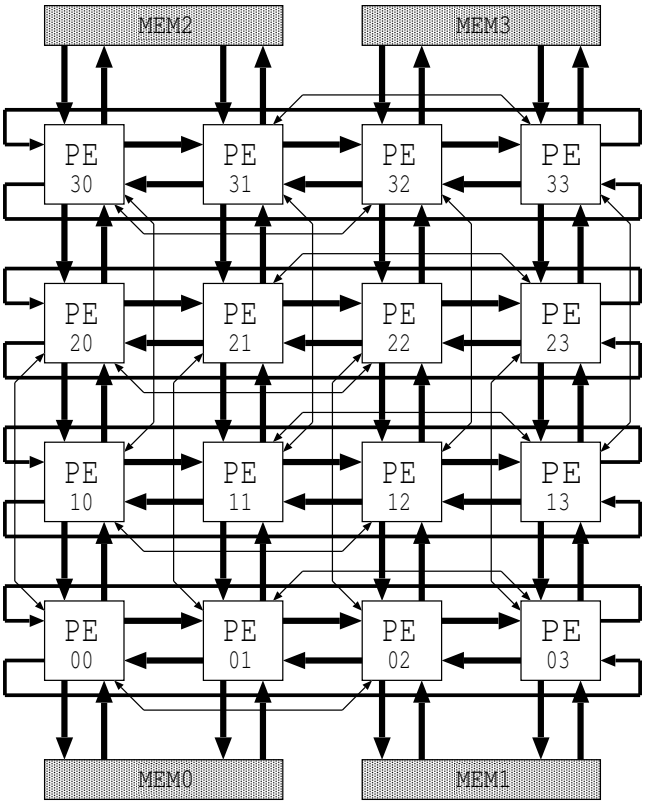

Figure 3.7: MuCCRA-D array structure

GCI structure of SE

MuCCRA-1 and MuCCRA-2 use island-style interconnection structure to exchange data between PEs. The network can be constructed by connecting the SEs together. The SE can route data in 4 directions (NEWS) based on cross-bar connection without loop back to the input direction as shown in Figure 3.8.

A set of cross-bar connection controls routing data in a routing channel (4 directions) and multiple sets of cross-bar connection can be implemented on a SE. Figure 3.8(a) shows GCI structure of SE for only a routing channel. The number of routing channels can be increased by adding duplicated structure for another routing channel as shown in Figure 3.8(b).

GCI structure of PE

The island-style interconnection is adopted in MuCCRA-1 and MuCCRA-2. For island-style interconnection, each PE provides "Pickin" and "Pickout" nodes for exchanging data with the surrounding global routing wires. Figure 3.9(a) shows abstract GCI structure of PE surrounding by one bi-directional routing channel. Two sets of pickin and pickout nodes are used for the forward and backward directions. There are 3 FUs inside PE with 1 input port and 1 output port represented by nodes. The input nodes can receive data from output nodes of

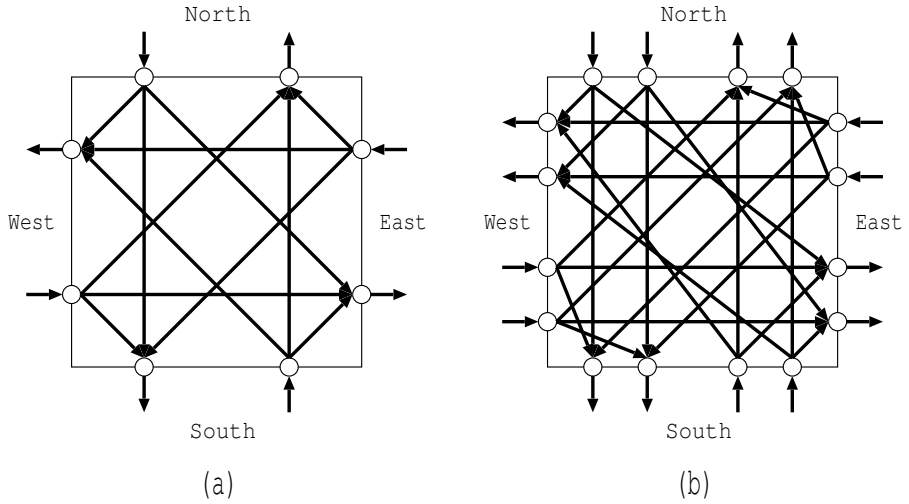

Figure 3.8: Representing GCI of SE ((a) 1 routing channel, (b) 2 routing channels)

the other FUs or input data from the "Pickin" node. The "Pickout" nodes are broadcasted by output data from all FUs and data from SE for passing to the opposite SE (upper horizontal routing channels).

Unlike island-style interconnection adopted in MuCCRA-1 and MuCCRA-2, MuCCRA-D [13] architecture uses Nearest Neighbor interconnection (NN-network) in order to transfer data between the PEs quickly. There are 3 bi-directional routing channels in 4 directions (NEWS) for transferring data from each FU to the next PEs. Note that, the input and output nodes of each FUs are connected to every nodes at PE boundary, but the links are omitted in the Figure since it is difficult to see.

GCI structure of functional unit and constant unit

Both ALU and SMU are functional units, but the different is that there is a constant unit in SMU. Normally, the constant unit is a register for storing constant data loaded at configuration. It is used in SMU computation, for example shifting input data with a certain number of bits or masking input data using logic AND with the constant data. The GCI representation of functional unit and constant unit can be shown as Figure 3.10.

Figure 3.10(a) shows the representation of an FU with four operations. The input port A and B receive data from the other FUs or the network. Each input link has configuration code for selecting the input data, thus, it is used as a control information of input functions. Here, four operations of the FU are represented with 2 bits. Fixing the operation is done by disabling the other un-selected operation codes.

The constant unit to store constant data loading from configuration data can be represented by using a node. Figure 3.10(b) shows the representation of constant unit to stored 3 bits constant data. There are 8 input links associated

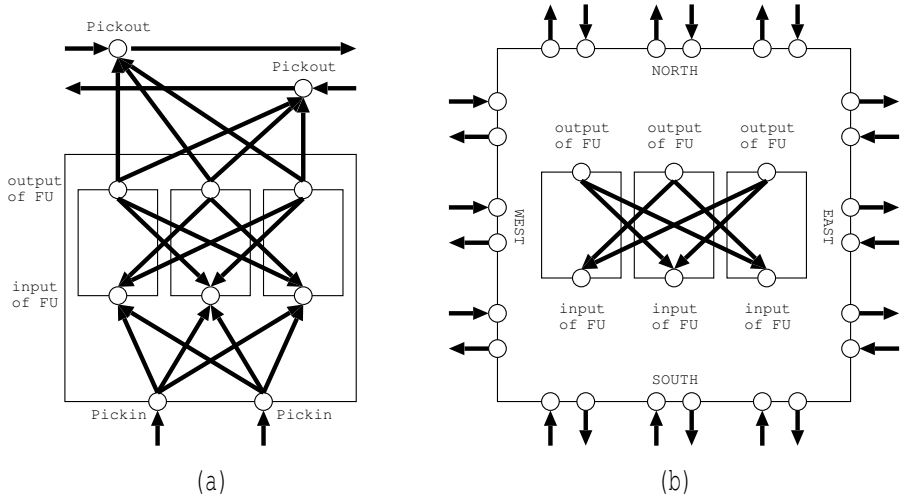

Figure 3.9: GCI structure of connection inside PE ((a) for island-style interconnection, and (b) for NN-network)

with a constant data, and selecting one of them is deciding constant data. Note that, in practical, the constant node is implemented by storing number of bits of the constant data rather than adding real input links since 65,536 input links are required for 16 bits constant data.

GCI structure of MUL and MEM

The GCI structure of multiply unit (MUL), which is available only on MuCCRA-1, is shown in Figure 3.11(a). It receives 2 input data selected from "Pickin" nodes and the multiply result is broadcasted from "Output" node to all "Pickout" nodes.

Figure 3.11(b) shows a GCI structure of memory module using in both MuCCRA-1 and MuCCRA-2. It is 2 ports memory which can read data from 2 addresses in parallel. The input addresses can be selected at "In A" and "In B" nodes, then the reading data are broadcasted from "Out A" and "Out B" nodes to all "Pickout" nodes respectively. For writing data selected at the "In Data" node, an input link corresponding to configuration code "1" must be selected at "W/E" node, and the data is written to the input address at port A.

In MuCCRA-D, there are the same GCI structure of MEM modules at upper and lower of PE array as shown in Figure 3.11(c). It is 2 ports memory connected to 2 PEs in different columns in order to reading data from the different ports in parallel. When an input link "1" is selected at the "W/E" node for writing data, an input link at "Select Data" node must be also selected to indicate that the input data at "In A" or "In B" nodes is written data or written address.

3.5.2 Constructing a context

After constructing each component on the target architecture, a large GCI structure to represent a context can be constructed by connecting all components

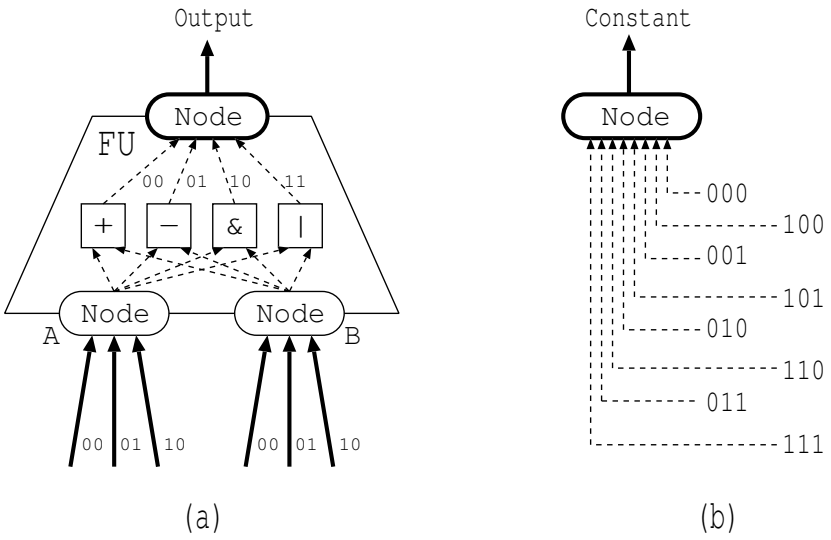

Figure 3.10: (a) Operation node, and (b) Constant node (the dash links are not actual connected in the graph because it is not necessary to transfer data from any existing node)

together. Note that, the numbers of "Pickin" and "Pickout" nodes are different from the previous abstract GCI structures.

The GCI structure of a context in MuCCRA-1 can be constructed by connecting PE and SE to form the island-style interconnection as shown in Figure 3.12. The pickout node of each routing channel can select input from SE or the components inside PE and pass data to the next SE. Then, the MUL module is connected to SEs on the leftmost side of PE array. Finally, the MEM module is connected to the lowest SEs of PE array.

The GCI structure of PE array in MuCCRA-2 is also constructed by connecting the pickout node inside PE to each routing channel at SE as MuCCRA-1. There are 3 routing channel on MuCCRA-2 and the connection can be shown as Figure 3.13. For the MEM modules, only 2 routing channels of the lowest row of SEs are connected to the pickout nodes in MEM module and the rest routing channel is connected directly to the next SE.

In MuCCRA-D, the PEs are connected together to form GCI structure of a context as shown in Figure 3.14. The MEM modules can be connected at upper and lower of PE array. Note that, the torus links and bypass links are not shown in the Figure.

3.5.3 Constructing entire architecutre

Since the MuCCRA architectures have several context memories to store and execute a context in a clock, the same GCI structures are used to represent configuration on each context. They are connected by adding links between the nodes in different contexts as shown in Figure 3.15. The nodes for transferring data to the next context is called "write nodes" and the nodes for receiving data

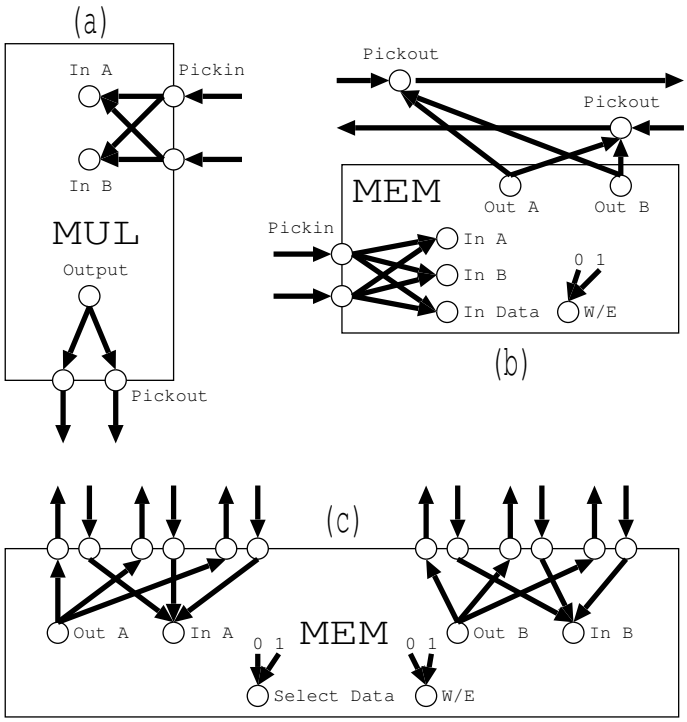

Figure 3.11: GCI structures of (a) MUL module, (b) and (c) MEM modules

from the previous context is called “read node”. They are implemented to be register components which always transfer data to the next context like register “REG-1”, or which has configuration codes to enable writing like register “REG-2”. The link from read node to write is used for holding data from the previous context when the write enable bits is set to be “0”.

3.5.4 Adding hardware restriction

In order to map application into target architecture, the placement process assigns computational operation to functional units, and the routing process select configuration codes in order to transfer the computational data. However, some components include hardware restriction which must be controlled in program code of traditional retargetable compiler. The GCI supports to control the hardware restriction during placement and routing by just adding control information between the nodes. The examples are shown as below:

Hardware restriction on RFU

A register file is a common component of DRPA, and often consisting of two-port structure, that is, port A can be used for both reading and writing, but port B is only for reading. In this case, when port A is used for writing data, it cannot be used for reading simultaneously except that the reading address and writing address are the same. Figure 3.16 shows GCI representation of

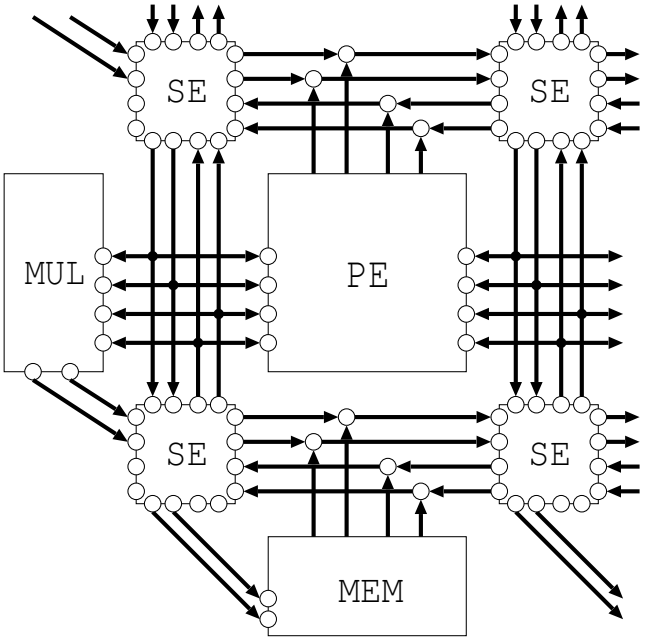

Figure 3.12: Constructing PE array of MuCCRA-1

a two-port register file with four register entries. Each register is consisting of read node for receiving data from the previous context and write node for transferring data to the next context. A register address for reading data at output B can be generated corresponding to the configuration bits of a selected input link directly. However, the register address at output A must be used for both reading and writing. The shaded node (named "DisCounT") is used to generate the address instead of the output port A. If input from "R3" is selected at the "Output A" node, the control link disables all virtual links other than R3 at the "DisCounT" node. In this case, only a control link from "Write" node corresponding to R3 is acceptable. With the similar manner, it is found from the graph that connections are allowed only when the register selected at "Output A" node and the register address of "Write" node are the same.

Hardware restriction on SE

In MuCCRA-2, the fully routing capability is not allowed for reducing the number of bits in configuration data, and only the switching patterns shown in the Table 3.2 are allowed. Five bits configuration codes can be used for each channel instead of 8 bits to control selecting output data of 4 directions.

The *no,so,eo, and wo* are output ports and the *ni,si,ei, and wi* are input ports. There is a register at the north direction to store input data before transferring to east and west direction to avoid combinatorial loop in the interconnection network. The *ni* is write node to transfer input data to read node ("nR") of the next context. The configuration code *00000* is used in the case that no input to be routed by sending zero value instead to save energy.

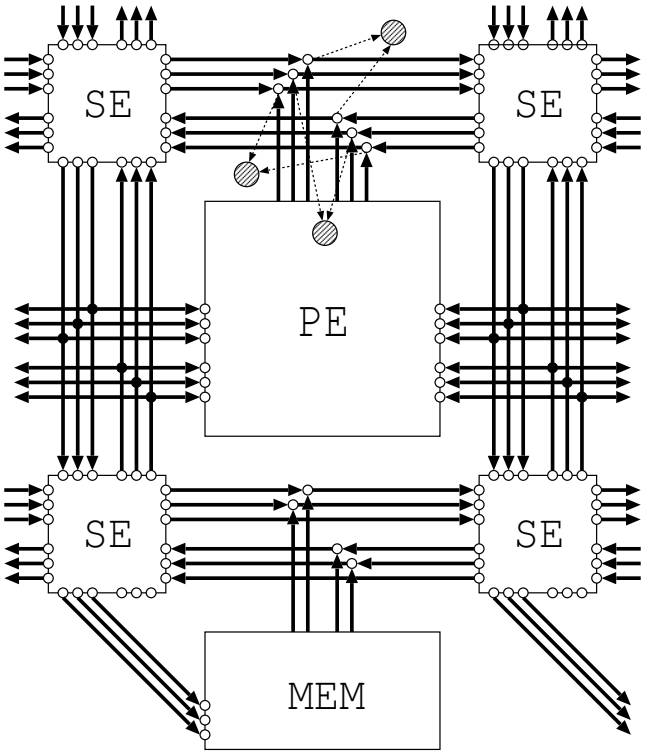

Figure 3.13: Constructing PE array of MuCCRA-2

The corresponding GCI is shown in Figure 3.17. The shaded nodes ("SW0", "SW1", and "SW2") are used to generate configuration data of the combination of routing for each channel as shown in the Table 3.2. The connecting path can be established from input to output if there is still one or more enable inputs after the output sends control information to disable the other input links at the shaded node.

Hardware restriction on PE

The PE of MuCCRA-1 can transfer output data from ALU, SMU, and RFU to 2 bi-directional routing channels by using 4 "Pickout" nodes. Each node is controlled by 2 bits configuration code to select data from the PE or pass SE data, making 8 bits to control the pickout unit. For the pickout unit of MuCCRA-2, a pair of "Pickout" nodes to transfer forward and backward data in each routing channel are controlled by 3 bits configuration code. A shaded node is added into GCI graph for selecting the different 3 bits configuration code controlled by control information from each pair of pickout nodes, like in the case of SE as shown in Figure 3.13. Thus, it requires 9 bits to control pickout of 3 routing channels instead of 12 bits.

The MuCCRA-2 has carry network separated from the island-style data network. The carry network can transfer carry signal to the north, east, and west neighboring PEs as shown by dashed lines in Figure 3.6. The GCI structure of

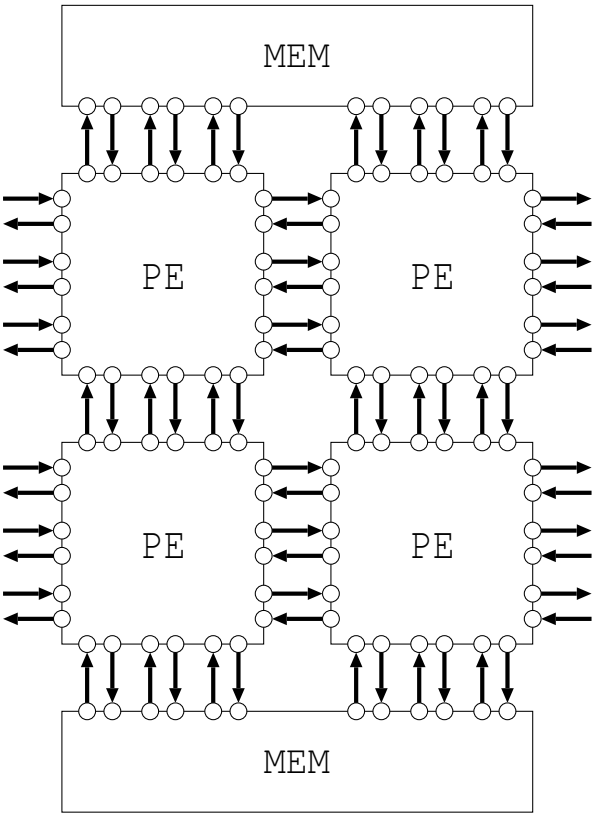

Figure 3.14: Constructing PE array of MuCCRA-D

carry network can be shown as Figure 3.18. There are pickin and pickout nodes for transferring carry signal to the next PEs. The pickin nodes can select output carry from functional units inside the PE and send control informations to the shaded node to select a corrsponding configuration code. 3 bits configuration data for controlling 3 output nodes can be reduced to be 2 bits configuration code at the shaded node. Note that, the input links at shaded node are omitted in the Figure.

Hardware restriction on ALU

The GCI structure of ALU in MuCCRA-2 can be shown as Figure 3.19. Some operations are limited in order to pass data from "Carry In" to output directly without any computation. So, an "Operation" node for selecting the operation codes is controlled by the link from the node "Carry Out".

Hardware restriction on SMU

The GCI structure of SMU in MuCCRA-2 can be shown as Figure 3.20. Four shaded nodes are used to represent two types of operation code. In order to avoid the redundancy in configuration code, SMU uses two modes of operations:

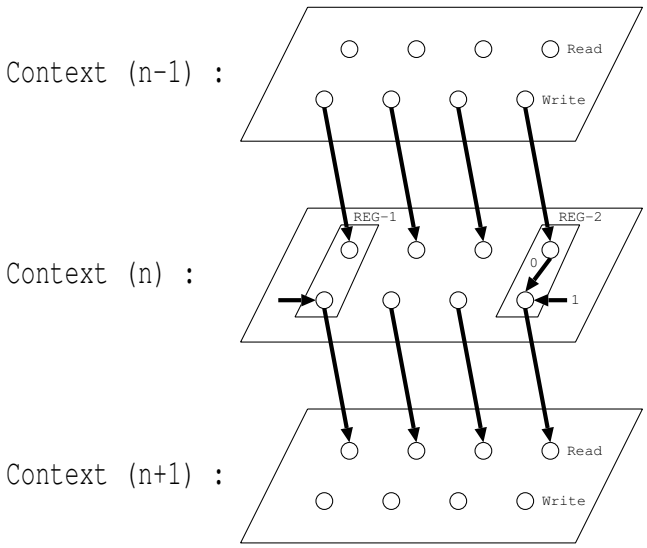

Figure 3.15: Constructing GCI of multicontext architecture

"Short-Constant" with 6 bits operation code and 14 bits constant value, and "Long-Constant" with 4 bits operation code and 16 bits constant value. A pair of shaded node is used for selecting one from two modes. In this case, the input which generates an empty string (NULL) is enabled when the corresponding mode is not selected, and thus, all other input links are disabled in the shaded node. Note that, in practical, there is special control information to disable generating configuration code at the shaded node rather that disable the real input links.

Hardware restriction on RFU of MuCCRA-2

Since the MuCCRA-2 architecture includes carry network, the RFU can also select writting carry signal independent from the writting data. The data selector and carry selector require 3 bits configuration codes to control (6 bits in total), and it can be reduced to be 4 bits at the shaded node "W/E" as shown in Figure 3.21.

The duplicated set of register and output node are added into the GCI structure for the carry signal. The shaded nodes "Address A" and "Address B" are added in the graph for generating configuration codes to read data from the same register number of data and carry. With the same manner as commond two-port register file, the configuration code of port A indicates a written register of both data and carry signal. Every write port transfers control information to select a configuration code which is corrsponding to the written register number, thus, writting data and carry to the same register number is acceptable. Note that, the input links at shaded node are omitted in the Figure.

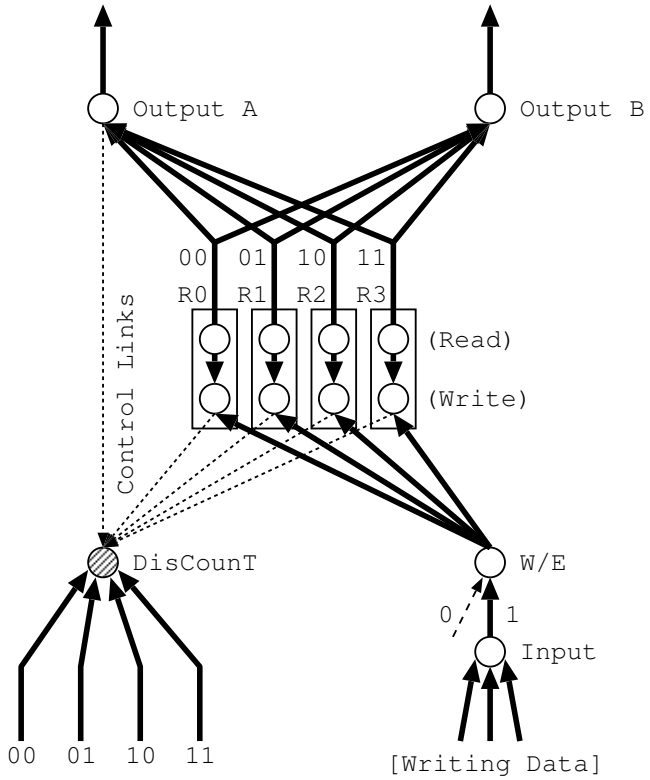

Figure 3.16: An example of two-port register file with 4 register entries

3.6 Summary

The GCI includes all possible configuration codes on the target architecture. It can be used to represent functional unit, constant unit, interconnection structure, and register file which are important components of DRPA. Since registers component can be represented in GCI as links between contexts, the intermediate data transferred to other contexts can be routed as well as routing connections in the same context.

In order to fix a selecting configuration code at a node, the other input links must be disabled. The restriction in hardware can be represented by adding control information into the GCI to disable input links at the other nodes, and there are 3 DisCounT rules to control every node during placement and routing.

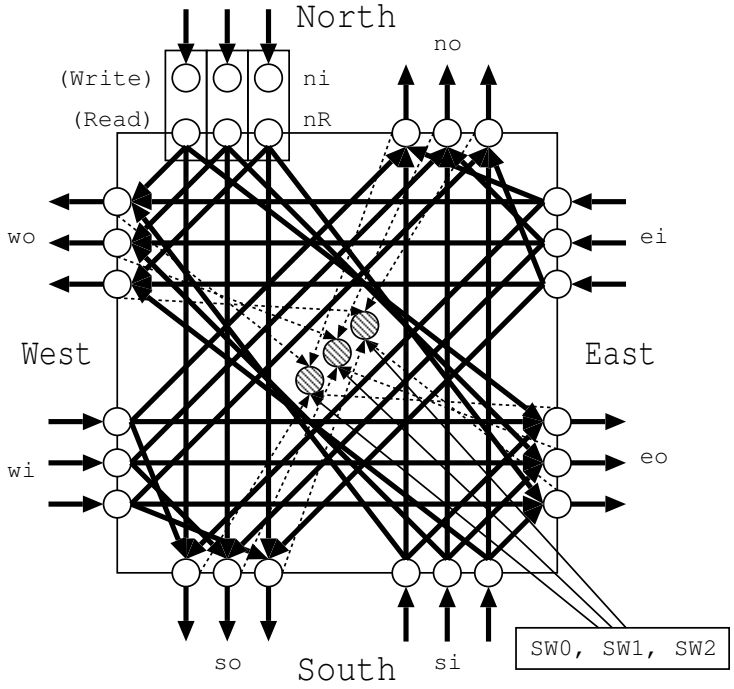

Figure 3.17: Switch box architecture

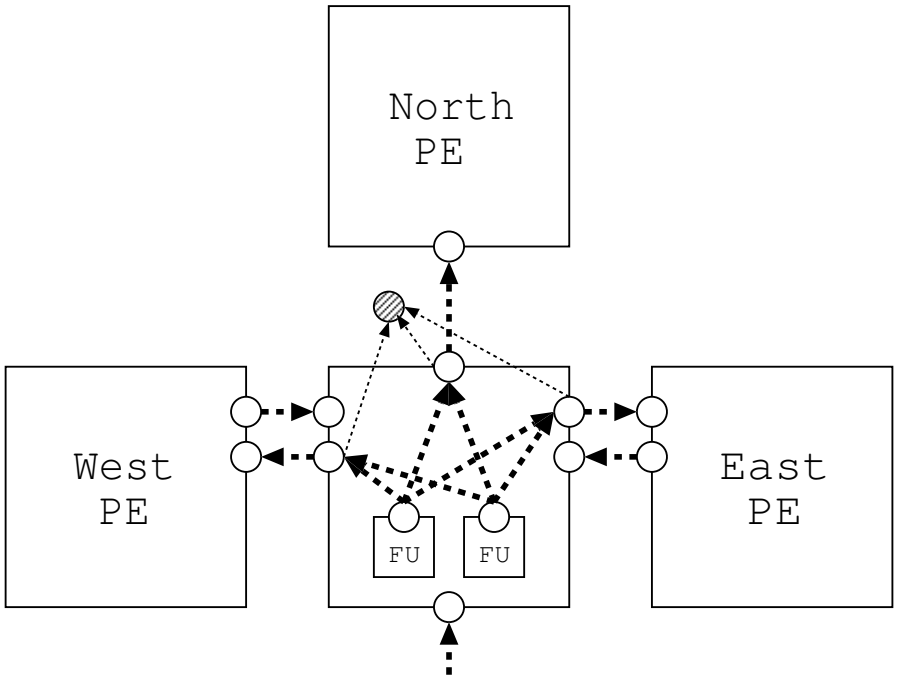

Figure 3.18: GCI structure of carry network

Table 3.2: Routing table of SE in MuCCRA-2

Configuration Code	Output			
	no	so	eo	wo
00000	–	–	–	–
00001	si	ni	wi	ei
00010	si	ei	wi	NR
00011	si	wi	NR	ei
00100	ei	ni	NR	ei
00101	ei	ni	si	ei
00110	ei	ni	wi	si
00111	ei	ni	wi	ei
01000	ei	ei	NR	ei
01001	ei	ei	si	ei
01010	ei	ei	wi	ei
01011	ei	wi	NR	si
01100	ei	wi	NR	ei
01101	ei	wi	si	NR
01110	ei	wi	si	ei
01111	ei	wi	wi	ei
10000	wi	ni	si	ei
10001	wi	ni	wi	NR
10010	wi	ni	wi	si
10011	wi	ni	wi	ei
10100	wi	ei	NR	si
10101	wi	ei	si	NR
10110	wi	ei	wi	NR
10111	wi	ei	wi	si
11000	wi	ei	wi	ei
11001	wi	wi	wi	NR
11010	wi	wi	wi	si
11011	wi	wi	wi	ei

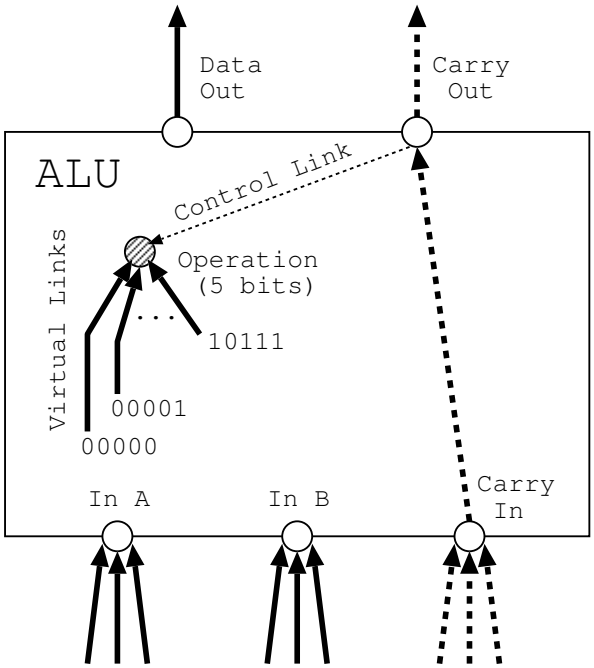

Figure 3.19: A target ALU architecture and corresponding GCI

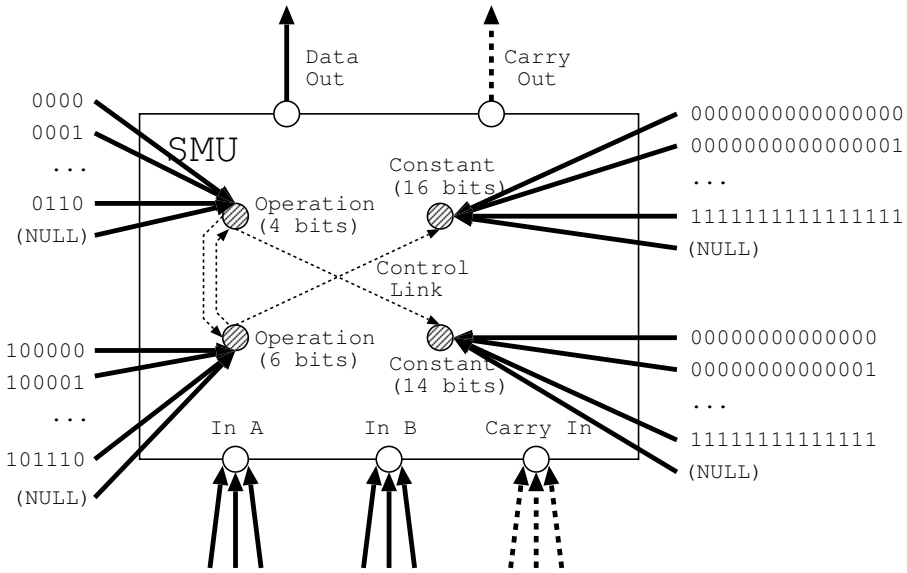

Figure 3.20: A target SMU Architecture and Corresponding GCI

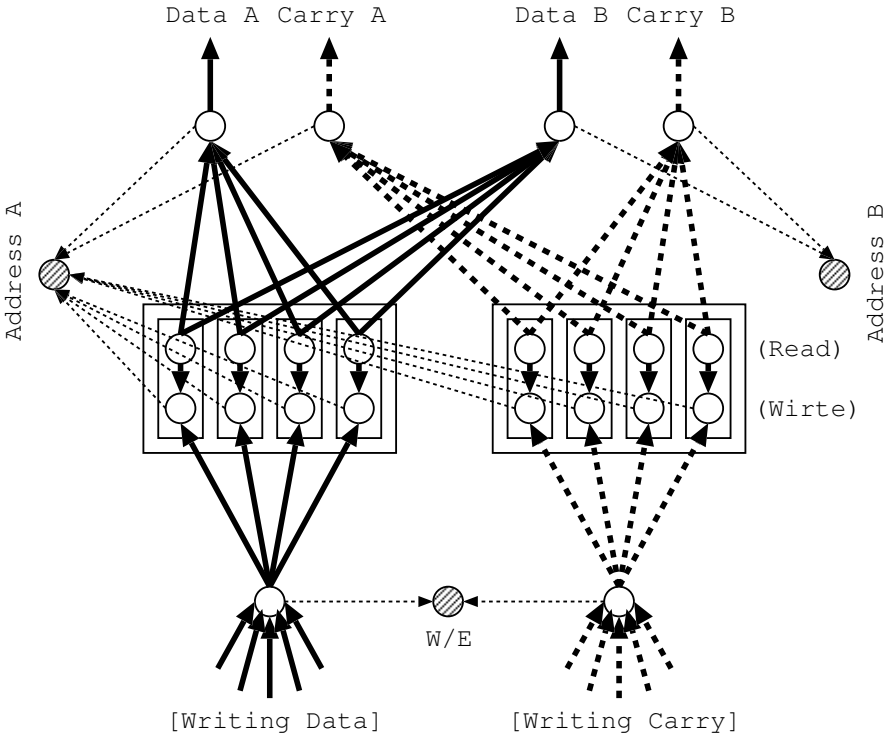

Figure 3.21: GCI structure of RFU in MuCCRA-2

Chapter 4

A retargetable compiler using GCI

A simple retargetable compiler called Black-Diamond was developped by using the GCI to represent a target architecture. Now, it can generate configuration data of three different models of DRPA; MuCCRA-1, MuCCRA-2 and MuCCRA-D.

4.1 Black-Diamond compiler

The flow of compilation is shown in Figure 4.1. Although common placement and routing algorithms used for FPGAs can be applied on the GCI, we adopt the simplest method in order to develop the retargetable compiler as quick as possible by making the best use of the characteristics of the GCI.

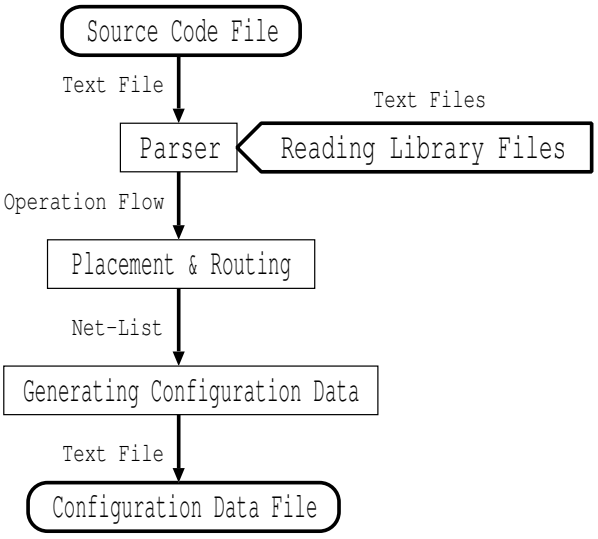

Figure 4.1: Compile flow of the Black-Diamond Compiler

The target application is described in a C-like language as shown in Figure 4.2. After giving source code file into the compiler, library file corresponding to the header file name in the source code is read out. It is consisting of GCI and library functions to be placed in the target architecture. Once the application is mapped into the architecture, the compiler generates configuration data in the text format as output.

4.1.1 Front-end language

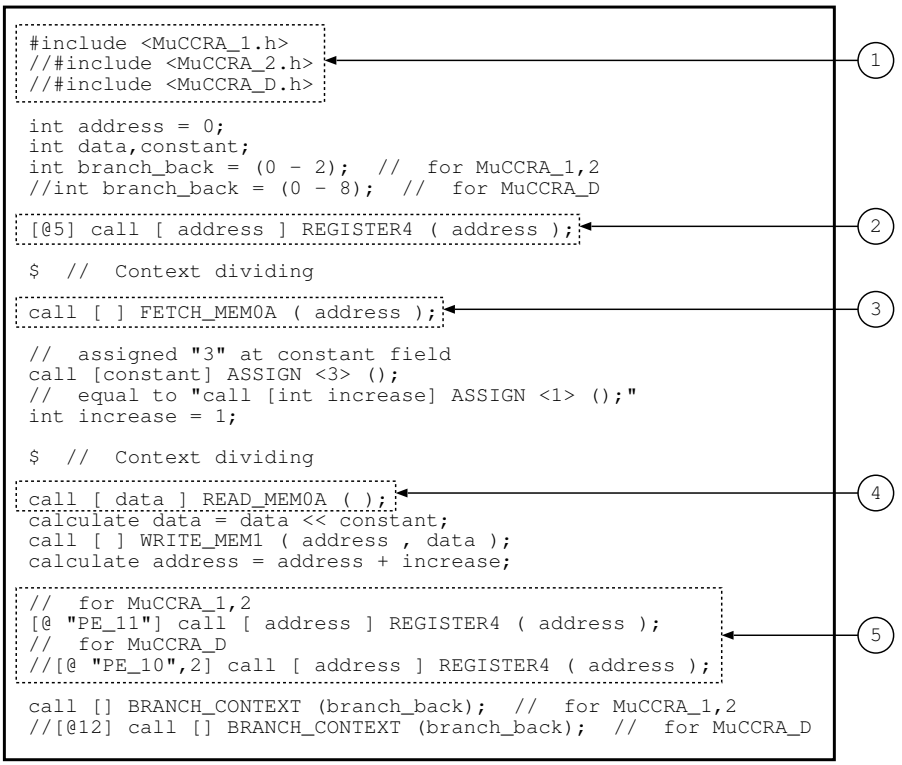

Figure 4.2: An example input source code ((1) header file for indicating target architecture name, (2 and 5) register used in looping, (3) giving reading address to memory, and (4) reading data from memory in the next context)

The parser used to translate the input source code to a data flow graph is developed by using LEX & YACC [3] in Linux. The data flow graph represents dependency by connecting the nodes corresponding to computations (library functions) used in the application. The output node of an function is linked to connect input node of the other functions by using a variable. The same variable name can be used to connect many different pairs of source and sink nodes depending on the last return value from the library function.

There are 3 parameter fields to call a function: *output field* (in front of function name), *input field* (behind function name), and *constant field* (between function name and input field if needed). The output field can return multiple

variables since pointers and structures are not allowed, and the input field can receive multiple variables to link the returning data from other functions. The constant field is provided to assign the given constant values to constant nodes where the function is placed.

4.1.2 Architecture description language

A target architecture can be declared in the header of the input source code file. Similar to the header file of common C language, it consists of prototype functions which are interface to call functions in a library file. Normally, the hader file has ".h" extension and the same file name with ".lib" extension is read out as the library file. In Black-Diamond compiler, the header file is declared with the same name as the target architecture for loading library file to describe the architecture.

The content inside library is written as a human readable description language. It can be divided into three parts: (1) constructing GCI graph, (2) definding library functions which can be called in the source code, and (3) generating output configuration data file.

Constructing GCI graph

In order to construct the GCI graph, first, template nodes with different configuration codes are definded. Each configuration code is considered as an input link which can be selected for generating configuration data or connected to another node for routing data when the template node is added into each element as a real node. Since the PE array of target architecture is constructed by connecting many elements which have the same configuration structures, the same type elements (PEs, SEs, MULs, and MEMs) use the same group of template nodes to construct GCI structure. There are descriptions to add control information to disable input links at the other nodes when some input links are selected for representing hardware restriction of the target architecture.

The multicontext architecture can become target of Black-Diamond compiler. It requires a large GCI graph to represent every context. However, only a context is definded as template in the library file for constructing every contexts. Some nodes are declared to be read node and write node for adding links between the nodes in different contexts to represent registers.

Definding library functions

For mapping application into the target architecture, the compiler must know information where to place the calling library functions into functional units at each element. Since the functional unit is represented by nodes, each function tends to indicate: (1) input nodes as target of routing data returned from the other functions, (2) nodes to select an input link corresponding to the computation of function, (3) constant nodes to assign constant value from the input source code, and (4) output nodes as source of routing data transferred to the other functions. When the function is called, the number of input variables, output variables, and constant values must be matched to the number of nodes. The prototype of each function is available at the header file for reference.

Generating configuration data

From the results of placement and routing in the GCI, the configuration data can be directly generated by just combining the configuration bits of a selected input link at each node in a specific order. If the target component is not used, the node must select an input link associated with a *default configuration code* to fulfill entire code of the configuration data.

The configuration data can be generated in order to enable multicast for reducing the number of configuration clock cycles. The multicast mechanism called RoMultiC [31] is adopted in all MuCCRA architectures, and the duplicated configuration data found in the different elements can be configured in parallel by using bitmap pattern. The node which does not select input link during the placement and routing can become the same selection as in another element to reduce the number of configuration clock cycles. The effective way to generate the bitmap pattern can be found in [30].

4.2 Mapping application

After giving source code, the compiler maps the application and generate configuration data for a target architecture corresponding to the header file. The mapping process can be divided to be placement and routing.

4.2.1 Placement algorithm

In order to map application into the architecture represented in GCI, the application is translated into a directed graph representing the data flow between computational nodes called *data-flow graph* as shown in Figure 4.3. The computational node is corresponding to the library function to fix configuration at the target placement (ex. ALU, SMU, or MEM). The input source code may consist of user function which is the collection of library functions, however, it must be translated to be the sequence of library functions. From this point, the word *"function"* means only the library function.

The function has a list of nodes in different types; *output nodes*, *input nodes*, *constant nodes*, and *fixed nodes*. The output and input nodes are source and sink nodes to be connected to form the datapath in the data-flow graph. Some functions such as initial constant data (*a*ssignment function) or *s*hifting function requires constant value assignment. The constant node is used to indicate the node to be assigned the value. The fixed node is configured to select an input link corresponding to the computation.

In the target architecture, there are many possible elements to place a function. The possible target placement is called *target element* as shown in Figure 4.4. It is a group of nodes to be referred by the existing node types in the function. In this example, the target architecture has 4 target elements (FUs) consisting of 7 nodes. The list of FUs is attached to the function in order to find possible placement solution. At the output node of FU, there is an input from the read node of register. With the same manner as in the ALU of MuCCRA-2 architecture, when the output data is routed from the register, the control link disables all input links except "10" at the "OPERATION" node. The selection at "OPERATION" node of lower right FU is set without placing any function by routing data "A" from the previous context.

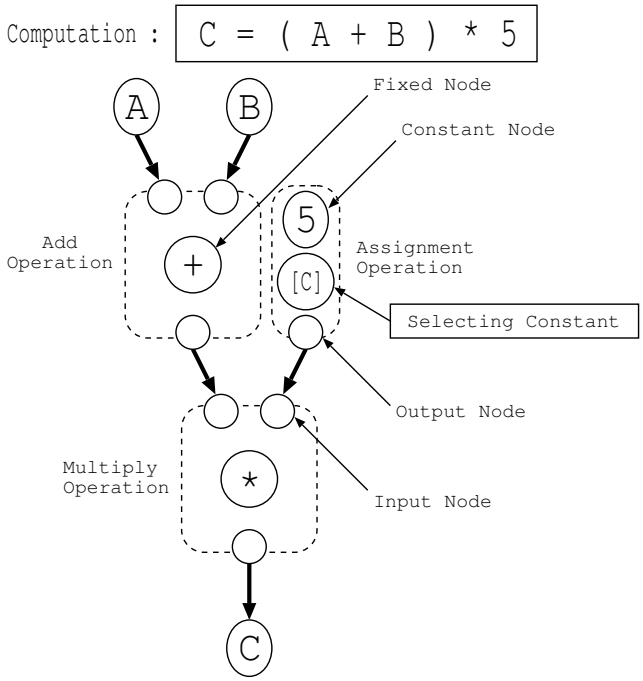

Figure 4.3: Data-flow graph

Here, a placement position is obtained from a simple greedy algorithm shown in Figure 4.5. Since the GCI applies hardware restriction, some input links are disabled by the first or second DisCounT rules after the placement. Thus, the order of calling function becomes important. The algorithm tries to place each function according to the list of target elements in order to route all input data successfully. We also need to ensure that, after the fixed node selects an input link and activates the control information, every node still has at least an enable input link due to the third DisCounT rule (can active control information), otherwise, the control information activated by placing and routing at the target must be canceled. In usual case, each calling function is placed into the first possible position, and the search starts from context zero by setting "shifting_value" and initialing "start_context" values to be zero. Once the placing position is obtained, the fixed node activates control information due to the first and second DisCounT rules, the constant node selects an input link corresponding to the constant data, and return variable at output field of function recognizes output node at the target element. A user can control to arrange the placement by inserting *pragma* in the source code before calling function. In the case that all target elements can not be used for a function placement, it tries to be placed in the next context.

4.2.2 Routing algorithm

In order to find the minimum cost path for routing data in the GCI, the shortest-path algorithm with obstacle avoidance [11] is used. The routing algorithm

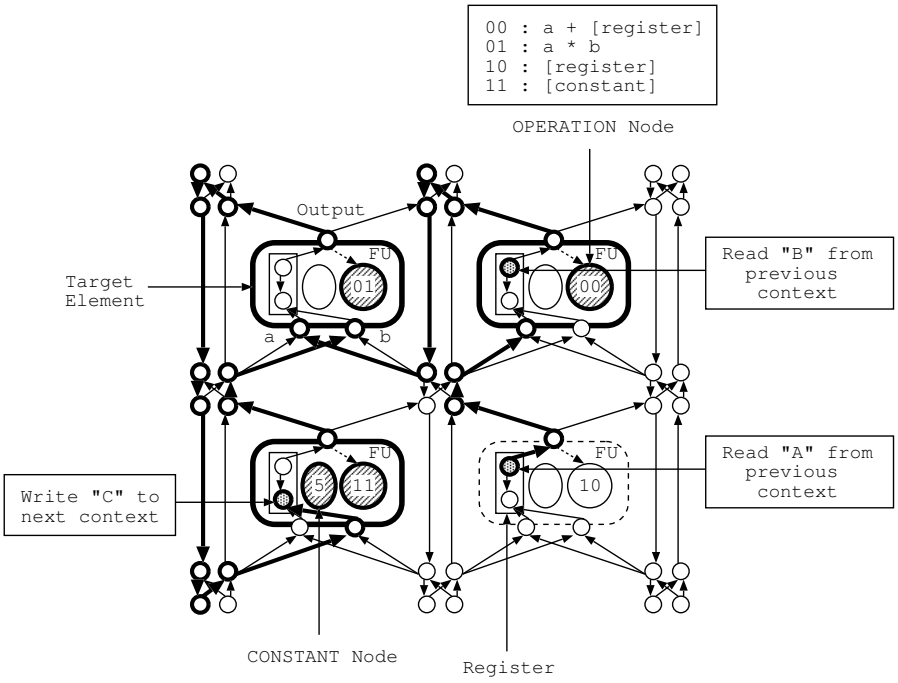

Figure 4.4: Example placing in the GCI representing a context of target architecture

shown in Figure 4.6 returns value to the placement algorithm indicating whether there exists path from the source node to the sink node or not.

The node in GCI also has cost information to represent wire delay of the target architecture. It is used in order to search the minimum cost path to route data. Since each node broadcasts the same output data inputted to other nodes, the list of input links can be the list of nodes where the input comes from. The path from the source to sink node is found by using Breadth First Search (BFS). The algorithm is started by adding the sink node into the searching list which is empty when starting. If there is no source node in the list, the input node with minimum propagation cost is added into the list. The process is repeated until the source node is found or there is no more un-reachable input node (not found). If there exists the connecting path, it can be obtained by backtracking from the source node.

All nodes on the connecting path are set to select an input link related to the backtracked input node. Since the GCI represents a constraint graph using DisCounT nodes, the input nodes, whose can not activate control information when the corresponding input links is selected, are not added into the searching list. All input nodes on the connecting path become disable except the selected routing input node by the first DisCounT rule. Thus, the input connection of the followed calling function can share the routing path to the same source node by using the rest enabling input links of pre-routing connection.

```
Position *PLACEMENT ( start_context , shifting_value ,
                      function ) {
  FOR ( each context ; start_context to MAX_CONTEXT ) {
    FOR ( each target in function ) {
      can_place = true ;
      FOR { each input of function } {
        IF ( can not ROUTING(sink at target,source) ) {
          can_place = false ;
        }
      }
      FOR ( each fixed_node in function at target ) {
        IF ( can not active control information ) {
          can_place = false ;
        }
      }
      IF ( can_place ) {
        shifting_value = shifting_value - 1 ;
        IF ( shifting_value < 0 ) {
          return [ context , target ] ;
        }
      }
      ELSE {
        Cancel the activate control information ;
      }
    }
  }
  return [ ERROR ] ;
}
```

Figure 4.5: Placement algorithm

4.2.3 The example application

Figure 4.2 shows an example application for shifting all data stored in MEM0 and writing result into MEM1 at the same address. By changing the header file declared at Figure 4.2(1), the same application can be mapped into different target architectures. The placement of each function can be automatically decided based on the restriction of GCI in order to route all input data to the function. The first possible placement position is selected, however, a user can control to place into the other positions by using the pragma.

The "@" pragma controls shifting the placement to be another position. In the example at Figure 4.2(2), the shifting value is 5 and the following function is placed into the sixth possible position (placing at "PE11" in MuCCRA-1 and MuCCRA-2, and at "PE10" in MuCCRA-D due to the different interconnection networks). If there is not possible position to place the function, it tries to find in the next context by routing input data via register automatically since the register is represented as link between contexts. The pragma can also control the placement to a target element by indicating its name as shown at Figure 4.2(5).

The context looping can be performed by calling "BRANCH_CONTEXT" function to transfer negative value related to the next executed context. Even the register can be automatically assigned to transfer data to the next context, the current version of compiler can not automatically assign the same register for storing data between the last context in the loop and the first context in the

```
Fact *ROUTING ( sink_node , source_node ) {
  search_list -> clear () ;
  search_list -> add ( sink_node ) ;
  WHILE ( searching_list has not source_node ) {
    add_node = NULL ;
    FOR ( each node in search_list ) {
      FOR ( each input_node at the node ) {
        IF ( search_list has not input_node )
          IF ( can activate control information )
            IF ( propagation cost less than add_node )
              add_node = input_node ;
              add_node -> backtrack = node ;
      }
    }
    IF ( add_node is NULL ) {
      return [ FALSE ] ;
    }
    search_list -> add ( add_node ) ;
  }
  node = source_node ;
  WHILE ( node is not sink_node ) {
    node -> backtrack -> select ( node ) ;
    node = node -> backtrack ;
  }
  return [ TRUE ] ;
}
```

Figure 4.6: Routing algorithm

next iteration. The "REGISTER4" function shown in Figure 4.2(2) and 4.2(5) is used to ensure that the register for holding the counting value "address" at the last context and reading at the first context in the loop are the same. Otherwise, the routing algorithm assigns register based on the minimum cost path.

In many architectures, the PE array can read data from memory module with a clock delay. The reading address is sent, and the reading data can be available in the next context since the multicontext architecture can switch context within a clock. A user can insert "$" pragma between the function "FETCH_MEM0A" and "READ_MEM0A" to start placing the following function in the next context.

The interconnection architecture of MuCCRA-1 and MuCCRA-2 is almost the same, but different style is adopted in MuCCRA-D. Thus, 3 contexts are used in the MuCCRA-1 and MuCCRA-2, while 9 contexts are used in the MuCCRA-D. In the case of MuCCRA-D, different set of pragmas which is commented out in the source code is used. By activating these lines, the code can be used for MuCCRA-D.

4.3 Graphic user interface

Mapping of application onto the PE array can be controlled by the user in Black-Diamond compiler. As the initial placement, it maps functions to the

first possible target elements automatically. However, the user may want to arrange the placement manually, for example, in order to reduce the loading configuration time in RoMultiC scheme or increasing usage of PE in a context. In this case, the user can insert pragmas in the source code to control the placement.

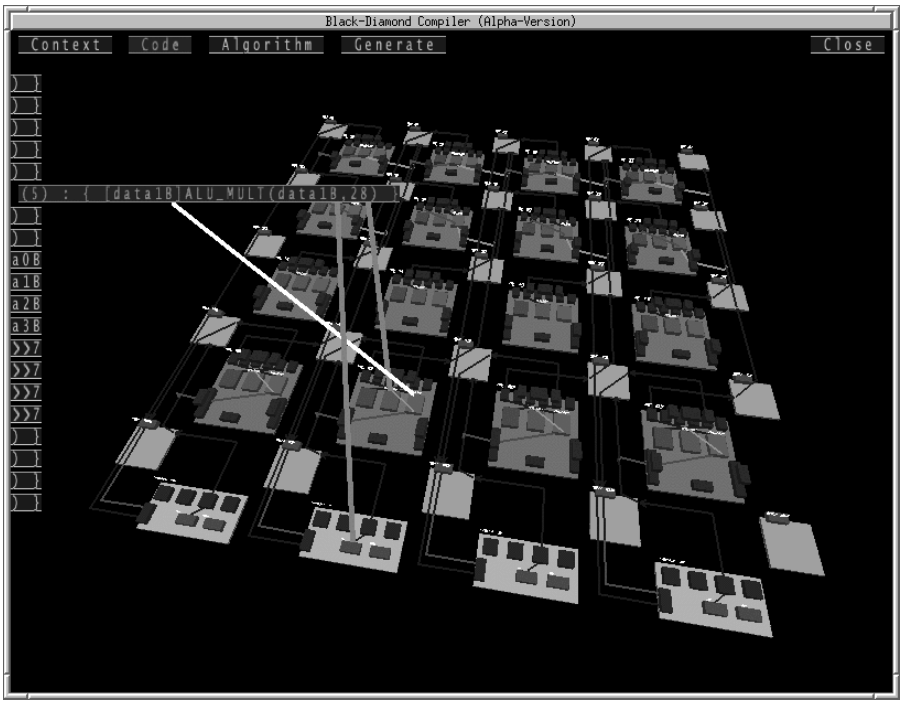

Figure 4.7: Screen shot of the Black-Diamond Compiler

For such cases, Black-Diamond supports Graphic User Interface (GUI) to show the placement and routing graphically. An example is shown in Figure 4.7. The target architecture is shown in 3-dimensional graphic. It can show the list of functions placed in each context with the same variable name in the source code. There is a pointer to show the node of function in the target placed element corresponding to each variable at the parameter fields. By selecting input link, the routed path is highlighted in the picture, so, the user can easily trace the path. This GUI is also helpful for debugging.

4.4 Designs using the retargetable compiler

It is difficult to demonstrate the benefit of the retargetable compiler, that is, how it can treat various target architectures easily. By using the GCI, Black-Diamond can treat three different types of DRPAs: MuCCRA-1, MuCCRA-2 and MuCCRA-D.

4.4.1 Evaluated applications

Black-Diamond can generate configuration data for all architectures just by changing the header file. Here, application implementation examples on three architectures are shown. Since the purpose of MuCCRA architectures here is slightly different from each other, implemented applications are not always the same. Five applications are used here for evaluation: *Alpha-Blender* combines two input images depending on a constant *alpha*. Three pairs of color data (RGB) can be combined in parallel. *Discrete Cosine Transform (DCT)* is a part of JPEG coder, and treats 8×8 image matrices. First, 1 dimentional DCT is computed in the row direction, then the similar computation is done to the transposed matrix. Thus, it is consisting of two processes: 1D-DCT and transpose. *Contrast* is "Histogram Equalization" used to enhance contrast of input image. It includes two iterations, one is for uniforming the histogram and the other is for replacing color. *Secure Hash Algorithm (SHA-1)* is required in the Digital Signature Standard (DSS). It is computationally infeasible to find two different messages which produce the same message digest. *Viterbi* algorithm is widely used to enhances the performance of digital communication systems by providing error correction over a noisy channel. The algorithm receives bits stream as input data and computes several path metrics in parallel to obtain the corrected bits stream data as output.

4.4.2 Mapping results

Table 4.1, 4.2, and 4.3 show the required contexts, maximum clock frequency and execution time. The clock frequency of MuCCRA-2 is higher than those of MuCCRA-1, since it is designed with the advanced process. In MuCCRA-D, there is a register to store data before transferring it to other PEs, thus, all applications are executed in the same clock frequency.

Table 4.1: Mapping and execution results on MuCCRA-1

Application	Contexts	Clk	Exe. time
Alpha-Blender	6	38MHz	6682nsec
DCT:1D-DCT	12	27MHz	3240nsec
DCT:Transpose	6	45MHz	924nsec
SHA1	12	20MHz	20900nsec
Viterbi	22	31MHz	15040nsec

Table 4.2: Mapping and execution results on MuCCRA-2

Application	Contexts	Clk	Exe. time
Alpha-Blender	5	90MHz	5643nsec
Contrast	11	76MHz	5057nsec
DCT:1D-DCT	12	66MHz	1350nsec
DCT:Transpose	6	76MHz	572nsec

Table 4.3: Mapping and execution results on MuCCRA-D

Application	Contexts	Clk	Exe. time
Alpha-Blender	11	125MHz	7200nsec
DCT:1D-DCT	17	125MHz	928nsec
DCT:Transpose	23	125MHz	184nsec
SHA1	29	125MHz	5816nsec

The Alpha-Blender requires smaller number of contexts in MuCCRA-2 than in MuCCRA-1 since the number of routing channels is larger. Thus, many computational functions can be placed and executed in the same context even there are limited switching patterns at the SE of MuCCRA-2 architecture. For the DCT application, it requires the same number of contexts because the mappings on both architectures are almost the same. All applications mapped on MuCCRA-D architecture require the number of contexts about 2 times compared to the MuCCRA-1 and MuCCRA-2, since the interconnection structure transfers data to only vertical and horizontal PEs So, it takes at least 2 contexts to transfer data to diagonal PEs. However, the applications can be executed in higher clock frequency.

The execution time of every application is superior to those from TI's DSP which works at 225MHz. Those results demonstrate that the practical applications can be developed by using Black-Diamond with multiple architectures.

4.4.3 Compilation time

The compilation time to map each application into the target architecture is measured from giving input source code to generating the configuration data. The result is measured on Intel(R) dual Core(TM) CPUs (2.40GHz) with 4096KB cache. Table 4.4 shows the compilation time to map each application on the MuCCRA architectures is relatively short (tens of seconds). The major reason is that, MuCCRA is a coarse-grained architecture consisting of small number of PEs (16 PEs). We will check the cases with larger size arrays in the next subsection.

Table 4.4: Compilation time of mapping each application on the different target architectures

Application	MuCCRA-1	MuCCRA-2	MuCCRA-D
Alpha-Blender	0.81sec	3.39sec	3.66sec
Contrast	-	23.46sec	-
DCT:1D-DCT	6.83sec	25.34sec	12.38sec
DCT:Transpose	1.35sec	9.22sec	16.37sec
SHA1	4.71sec	-	23.98sec
Viterbi	8.70sec	-	-

4.4.4 Manual mapping and mapping without pragma

In the Black-Diamond compiler, users can control the initial placement using pragmas. Since the compilation time is not so long, the user can make a lot of trial placement with various initial placement. On the other hand, perfectly manual mapping is extremely hard for the programmer. For MuCCRA-1, as an initial programming tool, a graphic editor tool called "MuCCRA editor" is available. In this tool, the programmer must fix the location of PEs and paths between them graphically for all contexts.

Table 4.5: Number of context required for mapping application on different methods on MuCCRA-1

Application name	Black-Diamond		Manual mapping
	with pragma	without pragma	
Alpha-Blender	6	7	8
DCT:1D-DCT	12	19	13
DCT:Transpose	6	6	15

The table 4.5 shows the required number of contexts from three cases: the case when the pragma is reasonably used, all pragmas are removed, and everything is completely done manually with "MuCCRA editor" [7]. Only three applications were implemented manually, since the program using "MuCCRA editor" takes extremely long time. The manual mapping results are not always better than the result without pragma. Especially, it requires a large number of contexts for matrix transpose in DCT which requires complicated access of shared memory. This type of application is hard to be treated by manual mapping. It also appears that the initial placement by pragma can reduce the number of contexts in some cases. So, the optimization by pragma for initial placement in Black-Diamond gives the better results with reasonable effort.

4.4.5 Mapping on larger size arrays

Since Black-Diamond uses simple placement and routing algorithm, in the worst case, if there are "n" target elements (count all contexts) and "m" functions to be placed, it requires the time in term of $O(n^m)$. However, in the real application, the target element to place function is usually found in only 1 or 2 contexts distance in a target architecture with reasonable amount of routing resources. Table 4.6 shows compilation time of mapping Alpha-Blender on MuCCRA-2 architecture with larger PE array sizes (8x8 and 16x16). The number of nodes of GCI shown in the second column is increased in linear due to the number of PEs (5 contexts). In the large PE array size, the routing algorithm must search connection on the large network, and there are many target elements to seek before placing the calling functions in the same positions as 4x4 PE array. Although the compiling time shown in the last column is increased in exponential, the result shows that Black-Diamond is useful at least for arrays with hundreds of PEs. Note that, sophisticated place and routing algorithms can be used with the GCI proposed here. For a larger array with thousands of PEs, such sophisticated algorithms should be introduced.

Table 4.6: Compilation time of mapping Alpha-Blender on different array sizes of MuCCRA-2 architecture

PE array size	Number of nodes	Compiling time
4 x 4	2758 x 5	3.40sec
8 x 8	11146 x 5	29.13sec
16 x 16	51154 x 5	503.12sec

4.5 Summary

A simple retargetable compiler called Black-Diamond has been implemented by using GCI to represent target architecture. A user can insert pragma into source code to arrange the placement manually. For this purpose, the GUI is also available to show mapping result graphically.

The Black-Diamond compiler can map applications into three different architectures by changing the header file to indicate the target architecture in the source code. A user can make a lot of trial to optimize the placement by using pragma compared to manual mapping since the compilation time is relatively short (tens of seconds).

Chapter 5

Conclusions and discussions

5.1 Conclusions

Retargetability is important for a compiler or synthesis tool used for architecture exploration. The traditional retargetable compilers can generate configuration data for many target architectures by changing paramters and options. However, the flexibility is limited within the available options. For developing brand new architectures, the compiler must be modified to support the new design. This thesis shows a standard model to represent different target architectures without changing the retargetable compiler itself.

The *Graph with Configuration Information (GCI)* is proposed to represent configurable resource in the target dynamically reconfigurable architecture. The GCI nodes represent selectable configuration codes as input links and interconnection structure in the same time. In order to fix a selecting configuration code at a node, the other input links must be disabled. The restriction in hardware can be represented by adding control information into the GCI to disable input links at the other nodes, and there are 3 DisCounT rules to control every node during placement and routing. Since a user can apply the restriction to control every node in the GCI directly, it allows higher degree of flexibility in the design-space exploration than the traditional parameter based retargetable compilers which can selecting only a set of available interconnection options. The MuCCRA architectures have been selected as case study, and many examples of using the GCI to represent each component with different hardware restrictions are shown. Other target architectures can be easily treated by representing many aspects of architectural property into a GCI.

A prototype compiler called Black-Diamond with GCI is now available for three different dynamically reconfigurable architectures. It translates data-flow graph from C-like front-end description, applies placement and routing by using the GCI, and generates configuration data for each element of the DRPA in the form of multicasting. A user can insert pragma into source code to arrange the placement manually. For this purpose, the GUI is also available to show mapping result graphically. Since the compilation time is relatively short (tens of seconds), a user can make a lot of trial to optimize the placement. Evaluation results of simple applications show that Black-Diamond can generate reasonable designs for three different architectures.

5.2 Discussions

5.2.1 A problem on mapping algorithm

In Chapter 4, Table 4.5 shows the manual mapping using a graphic tool does not always give good results. However, theoretically, it has advantage over the current version of routing algorithm in the following case. Since the connections are routed separately, the first connection has higher priority than the routing path in the next connection. From this property, the placing at the nearest PE may be blocked.

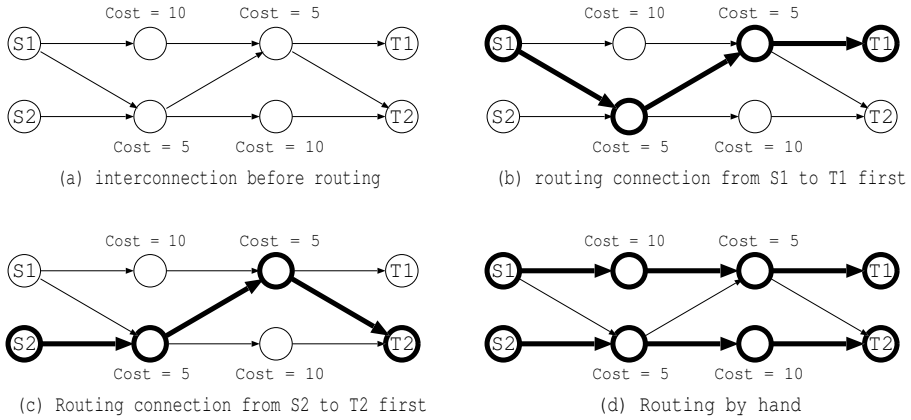

Figure 5.1: Problem of minimum cost path routing

The example can be shown as Figure 5.1, a function transfers 2 output data from node "S1" and "S2" to inputs at another function at nodes "T1" and "T2" respectively, and the bold line shows the routing path of the connections. In the Figure 5.1(b) and 5.1(c), a connection is selected to be routed first which results propagation cost equal to 10, however, another connection can not be routed. The functions must be replaced into the other target elements in order to route the connections, which may degrade the placement performance. On the other hand, Figure 5.1(d) shows the route decided manually. Both connections can be routed successfully with propagation cost equal to 15. Since GCI applied DisCounT rules at every node to represent hardware restriction, this example reveals that the order of disabling link becomes important.

5.2.2 Future work

The placement and routing algorithms used in Black-Diamond is simple, and sometimes requires human effort to add controlled pragmas. More sophisticated methods to automatic placement and routing are left as future work.

For the front-end of Black-Diamond compiler, it can be divided into input source code and architecture description. The input source code is simplified C language that does not support all the C grammar. There is only an "int" variable type for connections between functions. When the target architecture provides data network and carry network separately like MuCCRA-2, sometimes

a user confuses and try to form connections on the different networks that is impossible. The pointer grammar, structure grammar, and object-oriented programming style should be adopted to write the source code easily.

In order to construct GCI of target architecture, the current version of Black-Diamond uses simple architecture description which every element and link must be constructed from scratch. So, the architecture description files become very large (12,026 lines, 59,243 lines, and 26,111 lines for MuCCRA-1, MuCCRA-2, and MuCCRA-D respectively). The description of duplicated element structures are constructed by writing another program to generate the same structures repeatly. It can be improved by constructing duplicated structures from a template in the description file.

Appendix A

Architecture Description

A.1 The work before hand

The Black-Diamond compiler can be customized to map application into many target architectures by changing a header file at source code. The header file name has ".h" extension corresponding to read the same library file name with ".model", ".type", ".architecture", and ".code" extension. They are used to define 3D model to draw in Graphic User Interface (GUI) corresponding to each configuration code which is attached at each node, interconnection structure, and sequence of the nodes to generate configuration data of the target architecture. However, it is too compilcate and a designer must have very deep knowledge in order to define the target architecture by using those architecture descriptions. So, I simplify the architecture description to be understood easily. The tool called *Diamond-Dust* is used for generating 4 library files from an architecture description file with extension ".gci". Note that, the Black-Diamond compiler will be developed to construct the Graph with Configuration Information (GCI) to represent the target architecture from the new architecture description directly in the future.

A.2 Diamond-Dust tool

To use the tool, the name of target architecture file is giving followed in the execute command without extension. For example command line "dust TEST", the input file name "TEST.gci" is read out for generating architecture description for the target architecture named "TEST". The output files are: (1) "TEST.model", (2) "TEST.type", (3) "TEST.architecture", (4) "TEST.code", and (5) "TEST.h". They are library description used in the Black-Diamond compiler when indicating header file in source code with the name "TEST.h" (#include ⟨ TEST.h ⟩).

In order to create the architecture description file (.gci), first, the nodes inside the architecture are defined with configuration information. Then, they are grouped as reconfigurable element inside the target architecture. Next, the nodes are connected together by adding link between them. Finally, the restriction information is added into the graph. The graph representing target architecture is described only one context, and the Black-Diamond compiler

duplicates the graph if more number of contexts is needed to map application. Some nodes are identified for linking between the contexts. There are also description for defining functions which can be mapped into the architecture, and description for generating configuration data to configure the target architecture. The examples of creating architecture description are described step by step as below:

A.3 Example 1: Example Target Architecture

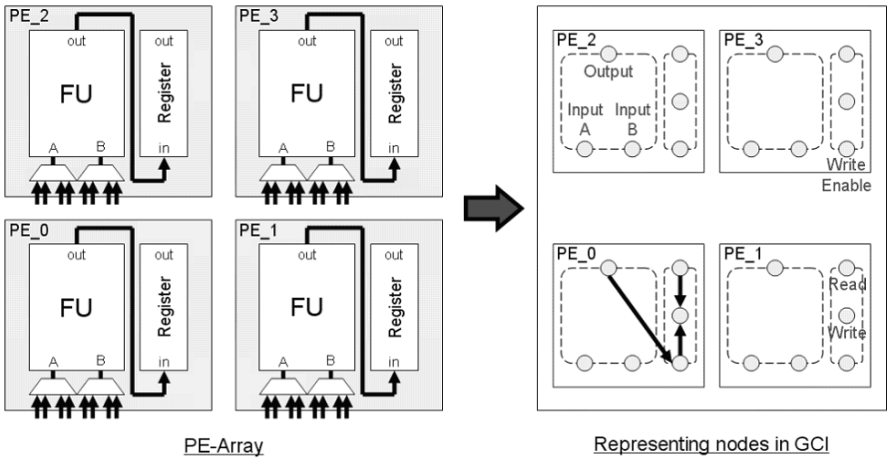

Figure A.1: Representing PE in GCI

In this example, the architecture consists of 4 Processing Elements (PEs). Each PE has a Functional Unit (FU) and a register as shown in Figure A.1. Since the FU receives 2 input data ("A" and "B") to calculate output data, it requires 3 nodes to represent input ports and an output port. The register has "Read" node and "Write" node for transferring data between contexts. The "Write" node can select input data from "Read" node for transferring to the next context and can select input data from "Write Enable" node to store the output data from FU. The architecture description to represent the nodes attached with configuration bits at each input can be described as below:

```
NODE  "Output"  //  Output of Functional Unit
    POSITION  0.5  0.0  0.0
    COST  20
    CONFIGURATION  "000_"  //  Addition Operation
    CONFIGURATION  "001_"  //  Substraction Operation
    CONFIGURATION  "010_"  //  Multiply Operation
    CONFIGURATION  "011_"  //  Receive I/O data
    CONFIGURATION  "100_"  //  Send input A to I/O Bus
    CONFIGURATION  "101_"  //  Send input B to I/O Bus
    DEFAULT 0
```

```
NODE  "Input A"  //  Input A of Functional Unit
    POSITION  0.0  0.0  1.0
    COST  10
    CONFIGURATION  "00_"  //  Get input from horizontal PE
    CONFIGURATION  "01_"  //  Get input from vertical PE
    CONFIGURATION  "10_"  //  Get input from Register
    CONFIGURATION  "11_"  //  Get input from horizontal Register
    DEFAULT  0
NODE  "Input B"  //  Input A of Functional Unit
    POSITION  1.0  0.0  1.0
    COST  10
    CONFIGURATION  "00_"  //  Get input from horizontal PE
    CONFIGURATION  "01_"  //  Get input from vertical PE
    CONFIGURATION  "10_"  //  Get input from Register
    CONFIGURATION  "11_"  //  Get input from vertical Register
    DEFAULT  0

NODE  "Read"  //  Output of Register
    POSITION  2.0  0.0  0.0
    COST  30
    CONFIGURATION  "(R)"
NODE  "Write"  //  For sending data to next context
    POSITION  2.0  0.0  0.5
    COST  0
    CONFIGURATION  "(Holding)"
    CONFIGURATION  "(Writing)"
NODE  "Write Enable"  //  Input of Register
    POSITION  2.0  0.0  1.0
    COST  30
    CONFIGURATION  "0_"  //  No writing
    CONFIGURATION  "1_"  //  Enable writing
    DEFAULT  0
```

The keyword “NODE” is followed by the node name to create a new node in the architecture and the following keywords are used in order to explain property of the node. The keyword “POSITION” indicates location of the node on the PE in GUI while the keyword “COST” indicates delay to transfer input data to output at the reconfigurable point in real hardware. At the application mapping, Black-Diamond compiler uses this information to search routing path with minimum delay to transfer data.

In order to configure the device, it requires configuration codes for (1) selecting computational operation at FU, (2) selecting input data at FU, and (3) controlling writing data into register. The keyword “CONFIGURATION” is used to describe the different configuration codes corresponding to each input. The keyword “DEFAULT” indicates selecting an input for generating configuration data if the node is not routed or set to select a configuration code. In the case of “Read” node and “Write” node, it also has input link to transfer data between contexts but they are not generated in configuration data. So, the keyword “DEFAULT” is not necessary to be defined.

After all nodes with different reconfigurable property are defined, they are grouped to be reconfigurable element inside the architecture as described below:

```
ELEMENT  "PE_0"
    WHERE  -3.0  0.0  3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Read"
    ADD  "Write"
    ADD  "Write Enable"
ELEMENT  "PE_1"
    WHERE  3.0  0.0  3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Read"
    ADD  "Write"
    ADD  "Write Enable"
ELEMENT  "PE_2"
    WHERE  -3.0  0.0  -3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Read"
    ADD  "Write"
    ADD  "Write Enable"
ELEMENT  "PE_3"
    WHERE  3.0  0.0  -3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Read"
    ADD  "Write"
    ADD  "Write Enable"
```

The keyword "ELEMENT" is used for creating reconfigurable element in the architecture. The property "WHERE" indicates position to draw the PE in GUI of the Black-Diamond compiler. And, all nodes identifying by keyword "ADD" are also drawn related to the position of PE locally as shown in the Figure A.1.

Next, the nodes in each element are connected together by adding link between the nodes. Each link is consisting of two nodes represented transferring data in a direction. The links in GCI can be described as shown below:

```
//  Connection inside PE

//  (1) Holding Register data
```

```
//  (2) Receive writing data
//  (3) Connect Register between contexts
//  (4) Receive writing data from Output

NET  "PE_0"@"Read"  ->  "PE_0"@"Write","(Holding)"
NET  "PE_0"@"Write Enable"  ->  "PE_0"@"Write","(Writing)"
REGISTER  "PE_0"@"Write"  ->  "PE_0"@"Read","(R)"
NET  "PE_0"@"Output"  ->  "PE_0"@"Write Enable","1_"

NET  "PE_1"@"Read"  ->  "PE_1"@"Write","(Holding)"
NET  "PE_1"@"Write Enable"  ->  "PE_1"@"Write","(Writing)"
REGISTER  "PE_1"@"Write"  ->  "PE_1"@"Read","(R)"
NET  "PE_1"@"Output"  ->  "PE_1"@"Write Enable","1_"

NET  "PE_2"@"Read"  ->  "PE_2"@"Write","(Holding)"
NET  "PE_2"@"Write Enable"  ->  "PE_2"@"Write","(Writing)"
REGISTER  "PE_2"@"Write"  ->  "PE_2"@"Read","(R)"
NET  "PE_2"@"Output"  ->  "PE_2"@"Write Enable","1_"

NET  "PE_3"@"Read"  ->  "PE_3"@"Write","(Holding)"
NET  "PE_3"@"Write Enable"  ->  "PE_3"@"Write","(Writing)"
REGISTER  "PE_3"@"Write"  ->  "PE_3"@"Read","(R)"
NET  "PE_3"@"Output"  ->  "PE_3"@"Write Enable","1_"

//  Connection between PEs

NET  "PE_1"@"Output"  ->  "PE_0"@"Input A","00_"
NET  "PE_2"@"Output"  ->  "PE_0"@"Input A","01_"
NET  "PE_0"@"Read"  ->  "PE_0"@"Input A","10_"
NET  "PE_1"@"Read"  ->  "PE_0"@"Input A","11_"
NET  "PE_1"@"Output"  ->  "PE_0"@"Input B","00_"
NET  "PE_2"@"Output"  ->  "PE_0"@"Input B","01_"
NET  "PE_0"@"Read"  ->  "PE_0"@"Input B","10_"
NET  "PE_2"@"Read"  ->  "PE_0"@"Input B","11_"

NET  "PE_0"@"Output"  ->  "PE_1"@"Input A","00_"
NET  "PE_3"@"Output"  ->  "PE_1"@"Input A","01_"
NET  "PE_1"@"Read"  ->  "PE_1"@"Input A","10_"
NET  "PE_0"@"Read"  ->  "PE_1"@"Input A","11_"
NET  "PE_0"@"Output"  ->  "PE_1"@"Input B","00_"
NET  "PE_3"@"Output"  ->  "PE_1"@"Input B","01_"
NET  "PE_1"@"Read"  ->  "PE_1"@"Input B","10_"
NET  "PE_3"@"Read"  ->  "PE_1"@"Input B","11_"

NET  "PE_3"@"Output"  ->  "PE_2"@"Input A","00_"
NET  "PE_0"@"Output"  ->  "PE_2"@"Input A","01_"
NET  "PE_2"@"Read"  ->  "PE_2"@"Input A","10_"
NET  "PE_3"@"Read"  ->  "PE_2"@"Input A","11_"
NET  "PE_3"@"Output"  ->  "PE_2"@"Input B","00_"
NET  "PE_0"@"Output"  ->  "PE_2"@"Input B","01_"
```

```
NET  "PE_2"@"Read"  ->  "PE_2"@"Input B","10_"
NET  "PE_0"@"Read"  ->  "PE_2"@"Input B","11_"

NET  "PE_2"@"Output"  ->  "PE_3"@"Input A","00_"
NET  "PE_1"@"Output"  ->  "PE_3"@"Input A","01_"
NET  "PE_3"@"Read"  ->  "PE_3"@"Input A","10_"
NET  "PE_2"@"Read"  ->  "PE_3"@"Input A","11_"
NET  "PE_2"@"Output"  ->  "PE_3"@"Input B","00_"
NET  "PE_1"@"Output"  ->  "PE_3"@"Input B","01_"
NET  "PE_3"@"Read"  ->  "PE_3"@"Input B","10_"
NET  "PE_1"@"Read"  ->  "PE_3"@"Input B","11_"
```

In the description, the node on the left side (source node) transfers data to the node on the right side (sink node). The nodes are referred with the element name followed by "@" in front of the node name. Since each input at a node attached with different configuration bits, the sink node also requires to be followed by "," and configuration bits to indicate connected input.

There are two keywords for adding link into GCI: (1) keyword "NET" and (2) keyword "REGISTER". The source node indicated by using the keyword "NET" is same context as the sink node while the source node indicated by using the keyword "REGISTER" is located in the previous context executed before the context of sink node. When Black-Diamond duplicates the graph to add new context for mapping application, the GCI of new context is connected with the previous context by adding the "REGISTER" links.

Now, the nodes and interconnection network of the target architecture have been represented as architecture description based on GCI. By using Diamond-Dust to translate the description (dust EXAMPLE1) and execute the Black-Diamond compiler with input source code indicating the target architecture (#include ⟨ EXAMPLE1.h ⟩), the screenshot can be shown as Figure A.2. All nodes are shown as small white boxes. Each element shown in different position has the same set of nodes to represent input ports and output port of FU and register.

A.4 Example 2: Adding Description for Mapping Application

In the first example, the target architecture has been represented, however, it can not be used for mapping application since the Black-Diamond compiler also requires information of available computational operation and a sequence of the nodes for generating configuration data. The description for adding computational operation of the target architecture can be shown as below:

```
FUNCTION  "Addition"
    OUTPUT  "Output"
    FIX  "Output"  ,  "000_"
    INPUT  "Input A"
    INPUT  "Input B"
```

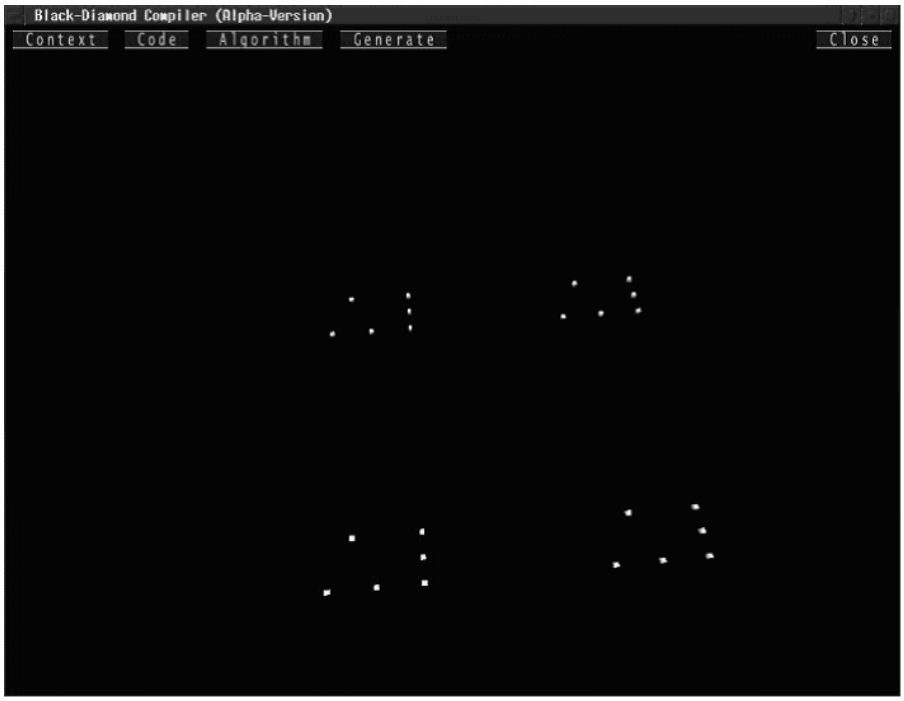

Figure A.2: Screenshot of example 1

```
    PLACE  "PE_0"
    PLACE  "PE_1"
    PLACE  "PE_2"
    PLACE  "PE_3"
FUNCTION  "Substraction"
    OUTPUT  "Output"
    FIX  "Output"  ,  "001_"
    INPUT  "Input A"
    INPUT  "Input B"
    PLACE  "PE_0"
    PLACE  "PE_1"
    PLACE  "PE_2"
    PLACE  "PE_3"
FUNCTION  "Multiply"
    OUTPUT  "Output"
    FIX  "Output"  ,  "010_"
    INPUT  "Input A"
    INPUT  "Input B"
    PLACE  "PE_0"
    PLACE  "PE_1"
    PLACE  "PE_2"
    PLACE  "PE_3"
FUNCTION  "Receive"  //  For receiving data from global bus
```

```
    OUTPUT  "Output"
    FIX  "Output"  ,  "011_"
    PLACE  "PE_0"
    PLACE  "PE_1"
    PLACE  "PE_2"
    PLACE  "PE_3"
FUNCTION  "Send_A"  //  For sending input data A to global bus
    OUTPUT  "Output"
    FIX  "Output"  ,  "100_"
    INPUT  "Input A"
    PLACE  "PE_0"
    PLACE  "PE_1"
    PLACE  "PE_2"
    PLACE  "PE_3"
FUNCTION  "Send_B"  //  For sending input data B to global bus
    OUTPUT  "Output"
    FIX  "Output"  ,  "101_"
    INPUT  "Input B"
    PLACE  "PE_0"
    PLACE  "PE_1"
    PLACE  "PE_2"
    PLACE  "PE_3"
```

The computational operation can be called in source code of the Black-Diamond compiler to be placed into the architecture as library function. Each function is declared by using keyword "FUNCTION" and followed by the function name. It requires property "PLACE" to indicate reconfigurable element whose function can be placed. The property "OUTPUT" and "INPUT" are used to indicate the node in the placed element related to parameter list of the calling function for exchanging data with other functions. The property "FIX" is used for selecting input at the node where the computational operation is placed element. The node name inside the element is fixed to select the following configuration code.

By using Diamond-Dust to translate the description (dust EXAMPLE2) and execute the Black-Diamond compiler with the following input source code, the screenshot can be shown as Figure A.3. There are input links to transfer data from output node of PE_0 and PE_3 to the input nodes of PE_1 for multiplying.

```
//  Input source code of Black-Diamond compiler

#include < EXAMPLE2.h >

int  a , b , c ;

call [ a ] Receive ( );  //  Getting data by accessing bus at PE_0
[ @2 ] call [ b ] Receive ( );  //  Getting data by accessing bus at PE_3
call [ c ] Multiply ( b , a );  //  Multiply data at PE_1
```

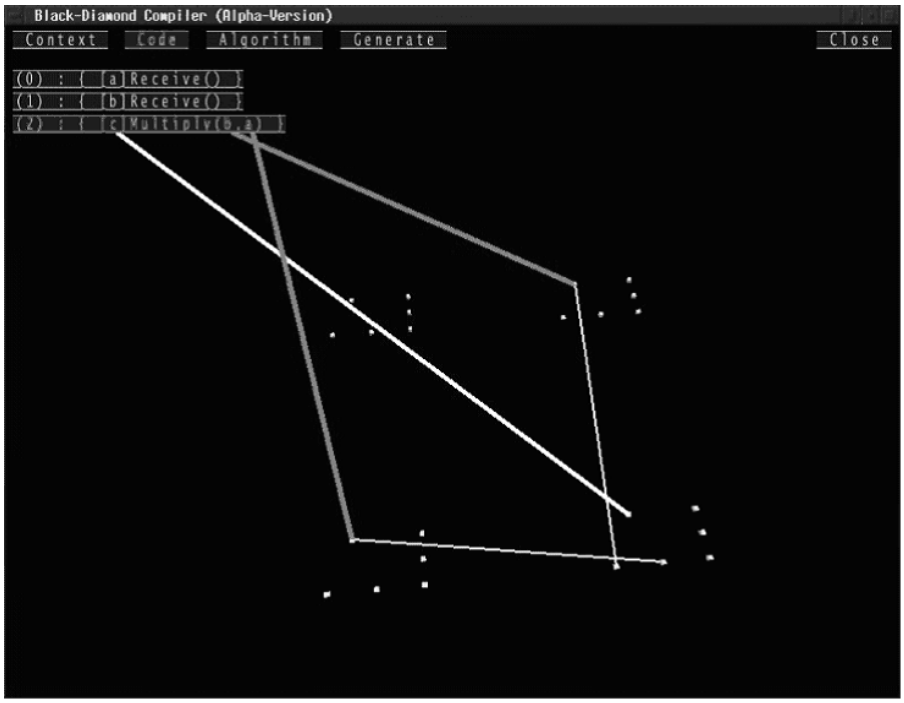

Figure A.3: Screenshot after mapping application

There are reserved functions which user can use to calculate data without writing calling function as the previous example input source code. They are shown in Table A.1 and used to declare the library function instead of the keyword "FUNCTION". All of those reserved functions require declaring 2 input nodes for receiving data and 1 output node for transferring the computational result except the "ASSIGN_FUNCTION" (be explained in Example 5). Some architectures require to set configuration codes at the first context and last context different from the other contexts. The reserved function "FIRST_CONTEXT_FUNCTION" and "LAST_CONTEXT_FUNCTION" can be added to set static configuration codes at the first context and last context respectively.

By rewriting the architecture description to be:

```
//  Replace FUNCTION "Addition"
ADD_FUNCTION  "+"

//  Replace FUNCTION "Substraction"
SUB_FUNCTION  "-"

//  Replace FUNCTION "Multiply"
MUL_FUNCTION  "*"
```

Table A.1: Reserved functions

Keyword	Example Source Code
ASSIGN_FUNCTION	int a = 5 , b , c; assign b = 8 ;
ADD_FUNCTION	calculate c = a + b ;
SUB_FUNCTION	calculate c = a - b ;
SHIFT_LEFT_FUNCTION	calculate c = a ¡¡ b ;
SHIFT_RIGHT_FUNCTION	calculate c = a ¿¿ b ;
MUL_FUNCTION	calculate c = a * b ;
FIRST_CONTEXT_FUNCTION	(automatic called at the first context)
LAST_CONTEXT_FUNCTION	(automatic called at the last context)

The previous input source code becomes:

```
//  Input source code of Black-Diamond compiler

#include < EXAMPLE2.h >

int  a , b , c ;

call [ a ] Receive ( ) ;  //  Getting data by accessing bus at PE_0
[ @2 ] call [ b ] Receive ( ) ;  //  Getting data by accessing bus at PE_3
//call [ c ] Multiply ( b , a ) ;  //  Multiply data at PE_1
calculate  c  =  b * a ;
```

A.5 Example 3: Adding Description for Generating Configuration Data

Since each node selects a configuration code corresponding to select an input at the node, the configuration data to configure device can be generated as sequence of the node in GCI. Suppose each PE in the target architecture requires configuration codes for: (1) indicating computational operation, (2) selecting input data, (3) controlling writing register, and (4) selecting configured PE (2 bits). So, 4 clock cycles are needed to configure a context. The description for generating configuration data can be shown as followed:

```
MAX_CONTEXT  16

FILE  "conf.dat"
    GENERATE  "PE_0"@"Output"
    GENERATE  "PE_0"@"Input A"
    GENERATE  "PE_0"@"Input B"
```

```
    GENERATE  "PE_0"@"Write Enable"
    STATIC  "_00_"
    CONTEXT_INDEX  4
    COUNT_CONFIGURATION
    NEW_LINE

    GENERATE  "PE_1"@"Output"
    GENERATE  "PE_1"@"Input A"
    GENERATE  "PE_1"@"Input B"
    GENERATE  "PE_1"@"Write Enable"
    STATIC  "_01_"
    CONTEXT_INDEX  4
    COUNT_CONFIGURATION
    NEW_LINE

    GENERATE  "PE_2"@"Output"
    GENERATE  "PE_2"@"Input A"
    GENERATE  "PE_2"@"Input B"
    GENERATE  "PE_2"@"Write Enable"
    STATIC  "_10_"
    CONTEXT_INDEX  4
    COUNT_CONFIGURATION
    NEW_LINE

    GENERATE  "PE_3"@"Output"
    GENERATE  "PE_3"@"Input A"
    GENERATE  "PE_3"@"Input B"
    GENERATE  "PE_3"@"Write Enable"
    STATIC  "_11_"
    CONTEXT_INDEX  4
    COUNT_CONFIGURATION
    NEW_LINE

    GEN_ALL_CONTEXT
    NEW_LINE

FILE  "tft.dat"
    NUM_CONFIGURATION  8
    STATIC  "_"
    NUM_CONTEXT  5
```

The keyword “FILE” is followed by the file name for storing configuration data generated as a sequence of the node in GCI indicated by keyword “GENERATE”. Since the configuration code to indicate configured PE is static for each PE, it can be added into the configuration data by using keyword “STATIC”. The number of available context in the target architecture is indicated by keyword “MAX_CONTEXT” at the beginning of description. The keyword “CONTEXT_INDEX” is followed by the number of digit to indicate the configured context. In some architectures, the number of configuration clock cycles is also

necessary. The keyword "COUNT_CONFIGURATION" is used for counting increased by 1 per found keyword. Noth that, the number of configuration clock cycles is counted on every context used for mapping the application. It is always paired with the keyword "NEW_LINE" to put ⟨ ENTER ⟩ character into the configuration data file to indicate new configuration clock cycle (new memory block of initial memory data in Verilog HDL). At the end of file "conf.dat", the keyword "GEN_ALL_CONTEXT" is added to generate configuration data for every context. Finally, the file "tft.dat" is created. It contains information of the number of configuration clock cycle and number of context by the keyword "NUM_CONFIGURATION" and "NUM_CONTEXT" respectively (the number of digit is indicated by the followed number).

A.6 Example 4: Adding Graphic to the Target Architecture

The screenshot shown at Figure A.3 is difficult to understand. The example of description for improving the graphic can be shown as follows:

```
//  Insert before declaring NODE
NODE_SIZE  0.1
NODE_COLOR  0.0  0.0  1.0

TEXT_SIZE  0.003

COLOR  0.4  0.4  0.7  //  PE_0
    BOX  ( -3.75  -0.3  2.5 ) , ( -0.25  -0.15  4.5 )
    COLOR  1.0  1.0  1.0
    TEXT  ( -3.75  0.2  2.5 ) , "PE_0"

COLOR  0.4  0.4  0.7  //  PE_1
    BOX  ( 2.25  -0.3  2.5 ) , ( 5.75  -0.15  4.5 )
    COLOR  1.0  1.0  1.0
    TEXT  ( 2.25  0.2  2.5 ) , "PE_1"

COLOR  0.4  0.4  0.7  //  PE_2
    BOX  ( -3.75  -0.3  -3.5 ) , ( -0.25  -0.15  -1.5 )
    COLOR  1.0  1.0  1.0
    TEXT  ( -3.75  0.2  -3.5 ) , "PE_2"

COLOR  0.4  0.4  0.7  //  PE_3
    BOX  ( 2.25  -0.3  -3.5 ) , ( 5.75  -0.15  -1.5 )
    COLOR  1.0  1.0  1.0
    TEXT  ( 2.25  0.2  -3.5 ) , "PE_3"

COLOR  0.5  0.5  0.5  //  Frame of PE_Array
    LINE  ( -4.25  -0.225  5.0 ) , ( 6.25  -0.225  5.0 )
    LINE  ( -4.25  -0.225  5.0 ) , ( -4.25  -0.225  -4.0 )
    LINE  ( -4.25  -0.225  -4.0 ) , ( 6.25  -0.225  -4.0 )
```

```
LINE  ( 6.25  -0.225  5.0 ) , ( 6.25  -0.225  -4.0 )
```

The keyword “NODE_SIZE” and “NODE_COLOR” must be defined before using keyword “NODE” to control size and color of the new node. The keyword “NODE_SIZE” requires following value to indicate the size (default value is 0.05) and three values following the keyword “NODE_COLOR” are used to indicate red, green, and blue colors of the node (default values are 1.0, 1.0, and 1.0 (white color)).

Keyword “BOX” and “LINE” require two position values (x, y, and z) to indicate starting point and ending point of drawing while keyword “TEXT” requires only a position to draw the following text. The drawing color can be changed by adding keyword “COLOR” before the drawing keywords. The screenshot after adding the graphic description can be shown as Figure A.4.

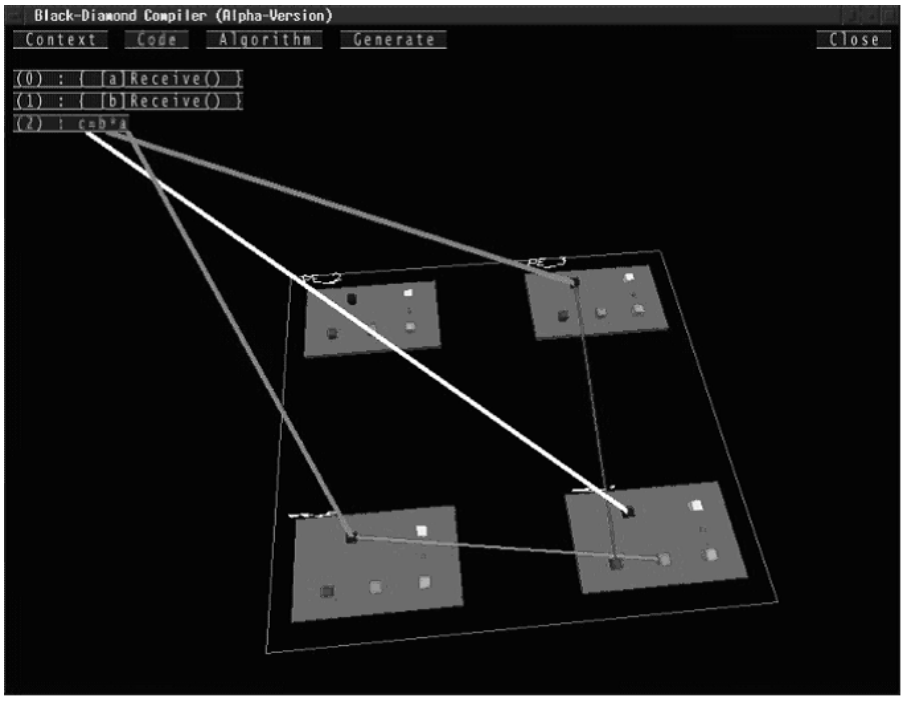

Figure A.4: Screenshot after adding graphic description

A.7 Example 5: Adding Constant Node to the Target Architecture

Some architectures can load constant data from configuration data to store in constant register of each PE and use for calculation. It requires a node in each PE to represent the constant value. In this example, we change the target architecture by adding a constant register to store the constant data inside FU.

A new FU operation is also added to read the constant data at the output of the FU. The additional description can be shown as below:

```
//  Add new operation to "Output" node of FU
CONFIGURATION  "111_"  //  Send constant data to output

CONSTANT_NODE  "Constant"  ,  16

//  Add the new node to each PE
    ADD  "Constant"

//  Add generating node into configuration data of each PE
    GENERATE  "PE_0"@"Constant"
    STATIC  "_"

ASSIGN_FUNCTION  "Assign"
    OUTPUT  "Output"
    FIX  "Output"  ,  "111_"
    CONSTANT  "Constant"
    PLACE  "PE_0"
    PLACE  "PE_1"
    PLACE  "PE_2"
    PLACE  "PE_3"
```

First, a new operation is added into the "Output" node for sending constant data to the output. The node for representing constant value can be created by adding a lot of virtual links at the node to represent all possible constant bits. If the constant data is 16 bits, it requires 65536 virtual links. It makes the description become large and waste a lot of memory. The effective way is recognizing only the number of bitwidth and generating configuration bits corresponding to selecting the virtual links. Keyword "CONSTANT_NODE" used for creating the constant node is followed by the constant node name and the number of bitwidth. Next, the constant node is inserted into each PE and also inserted to generate the configuration data. Finally, a function to set the constant value is needed. When the function is placed into architecture, it fixes computational operation to be the new operation and indicates the output node. Normally, the function can be called in the source code by inserting "⟨" and "⟩" to set selecting input at the node indicated by using keyword "CONSTANT". The keyword "ASSIGN_FUNCTION" can be used for representing reserved function and assign the constant value by using "=" sign in the source code instead.

By using Diamond-Dust to translate the modified description (dust EXAMPLE5), the following input source code can be compiled. The comment line bellowed the variable declaration can be used instead of the "int a = 3".

```
//  Input source code of Black-Diamond compiler

#include < EXAMPLE5.h >
```

```
int  a = 3 , b , c ;  //  Assign constant value at PE_0
//call [ a ] Assign < 3 > ( ) ;
//assign a = 3 ;

[ @2 ] call [ b ] Receive ( ) ;  //  Getting data by accessing bus at PE_3
calculate c = a - b ;  //  Subtract data at PE_1
```

A.8 Example 6: Adding restriction to Control Accessing Global Bus

This example shows how to add restriction into the architecture description. Since each PE in the architecture can access to write data into global bus (using the operation corresponding to configuration codes "100_" and "101_"), it requires restriction to limit accessing the bus by only a PE per context. The description to add the restriction can be shown as below:

```
DISCOUNT  "PE_0"@"Output"
    CONTROL  "PE_1"@"Output","100_"  ->  "100_"
    CONTROL  "PE_1"@"Output","100_"  ->  "101_"
    CONTROL  "PE_1"@"Output","101_"  ->  "100_"
    CONTROL  "PE_1"@"Output","101_"  ->  "101_"
    CONTROL  "PE_2"@"Output","100_"  ->  "100_"
    CONTROL  "PE_2"@"Output","100_"  ->  "101_"
    CONTROL  "PE_2"@"Output","101_"  ->  "100_"
    CONTROL  "PE_2"@"Output","101_"  ->  "101_"
    CONTROL  "PE_3"@"Output","100_"  ->  "100_"
    CONTROL  "PE_3"@"Output","100_"  ->  "101_"
    CONTROL  "PE_3"@"Output","101_"  ->  "100_"
    CONTROL  "PE_3"@"Output","101_"  ->  "101_"
DISCOUNT  "PE_1"@"Output"
    CONTROL  "PE_0"@"Output","100_"  ->  "100_"
    CONTROL  "PE_0"@"Output","100_"  ->  "101_"
    CONTROL  "PE_0"@"Output","101_"  ->  "100_"
    CONTROL  "PE_0"@"Output","101_"  ->  "101_"
    CONTROL  "PE_2"@"Output","100_"  ->  "100_"
    CONTROL  "PE_2"@"Output","100_"  ->  "101_"
    CONTROL  "PE_2"@"Output","101_"  ->  "100_"
    CONTROL  "PE_2"@"Output","101_"  ->  "101_"
    CONTROL  "PE_3"@"Output","100_"  ->  "100_"
    CONTROL  "PE_3"@"Output","100_"  ->  "101_"
    CONTROL  "PE_3"@"Output","101_"  ->  "100_"
    CONTROL  "PE_3"@"Output","101_"  ->  "101_"
DISCOUNT  "PE_2"@"Output"
    CONTROL  "PE_0"@"Output","100_"  ->  "100_"
    CONTROL  "PE_0"@"Output","100_"  ->  "101_"
    CONTROL  "PE_0"@"Output","101_"  ->  "100_"
```

```
    CONTROL  "PE_0"@"Output","101_"  ->  "101_"
    CONTROL  "PE_1"@"Output","100_"  ->  "100_"
    CONTROL  "PE_1"@"Output","100_"  ->  "101_"
    CONTROL  "PE_1"@"Output","101_"  ->  "100_"
    CONTROL  "PE_1"@"Output","101_"  ->  "101_"
    CONTROL  "PE_3"@"Output","100_"  ->  "100_"
    CONTROL  "PE_3"@"Output","100_"  ->  "101_"
    CONTROL  "PE_3"@"Output","101_"  ->  "100_"
    CONTROL  "PE_3"@"Output","101_"  ->  "101_"
DISCOUNT  "PE_3"@"Output"
    CONTROL  "PE_0"@"Output","100_"  ->  "100_"
    CONTROL  "PE_0"@"Output","100_"  ->  "101_"
    CONTROL  "PE_0"@"Output","101_"  ->  "100_"
    CONTROL  "PE_0"@"Output","101_"  ->  "101_"
    CONTROL  "PE_1"@"Output","100_"  ->  "100_"
    CONTROL  "PE_1"@"Output","100_"  ->  "101_"
    CONTROL  "PE_1"@"Output","101_"  ->  "100_"
    CONTROL  "PE_1"@"Output","101_"  ->  "101_"
    CONTROL  "PE_2"@"Output","100_"  ->  "100_"
    CONTROL  "PE_2"@"Output","100_"  ->  "101_"
    CONTROL  "PE_2"@"Output","101_"  ->  "100_"
    CONTROL  "PE_2"@"Output","101_"  ->  "101_"
```

When a FU selecting operation at FU to transfer input data to the global bus, the other FUs can not access to the bus in the same context. The keyword "DISCOUNT" is used for indicating the "Output" node which is disabled inputs by the other nodes. If the other nodes indicated by using keyword "CONTROL" selects an input corresponding to the followed configuration code, the input corresponding to the configuration code followed "-⟩" is disabled at the "Output" node indicated by the keyword "DISCOUNT". The example source code to show the different between before and after adding the restriction can be shown as following:

```
//  Must be selected only a included file to indicate a target architecture

#include  < EXAMPLE5.h >  //  Architecture before adding restriction
//#include  < EXAMPLE6.h >  //  Architecture after adding restriction

int  a = 1 , b = 2 , output ;

call [ output ] Send_A ( a ) ;
call [ output ] Send_B ( b ) ;
```

Figure A.5 shows the screenshot of architecture before adding the restriction. Two calling functions to access the global bus can be the same context. The "Send_A" function is placed at "PE_2" and the "Send_B" function is placed at "PE_3".

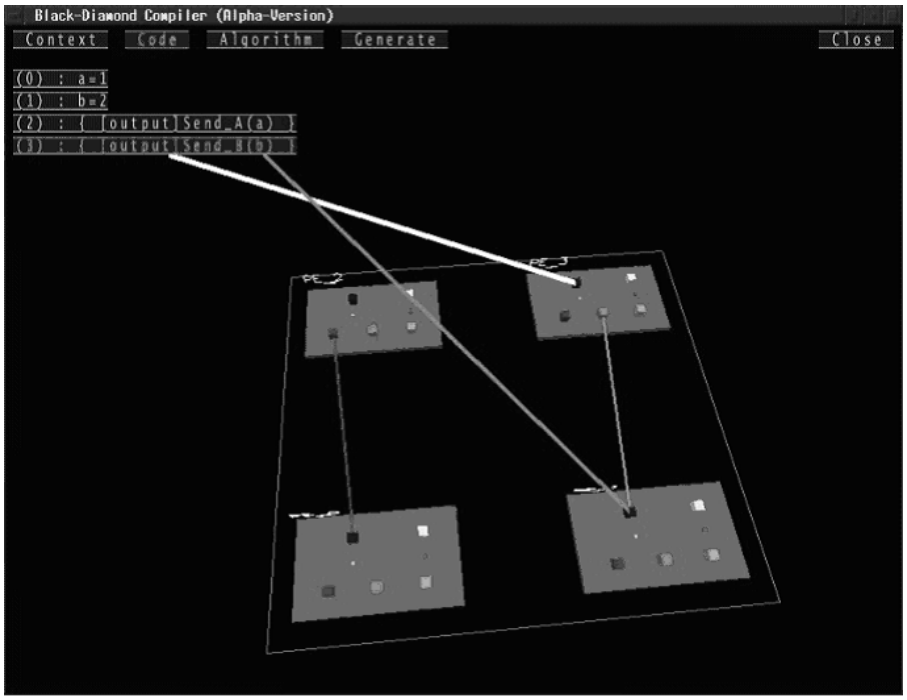

Figure A.5: Screenshot before adding restriction

Figure A.6 shows the screenshot of architecture after adding the restriction. The application requires 2 contexts for placing the two calling functions. In context 0, the constant value "b" is stored in register of "PE_1" at first. Then, the "Send_B" function access the global bus in the next context.

A.9 Example 7: Changing Register to be Register File

In all previous examples, each PE in the architecture has only a register. Many architectures store intermediate by using register file. This example explains how to replace the register by 1 port register file with 4 register entries. The register file requires 2 configuration bits to indicate both reading and writing register number and 1 configuration bits to enable the writing. If the reading and writing are occurred in the same context, they must use the same register number.

A new node is added into each PE for generating the register number (represented by 4 input links). This node is used for controlling the register number of reading and writing to be the same. Since the register file is consisting of 4 entries, it requires 4 sets of "Read" node and "Write" node to connect each entry between the different contexts. Selecting register reading data at the new node and the "Write" node of each entry disables un-corresponding generating register number. If reading and writing disable the different register number, the

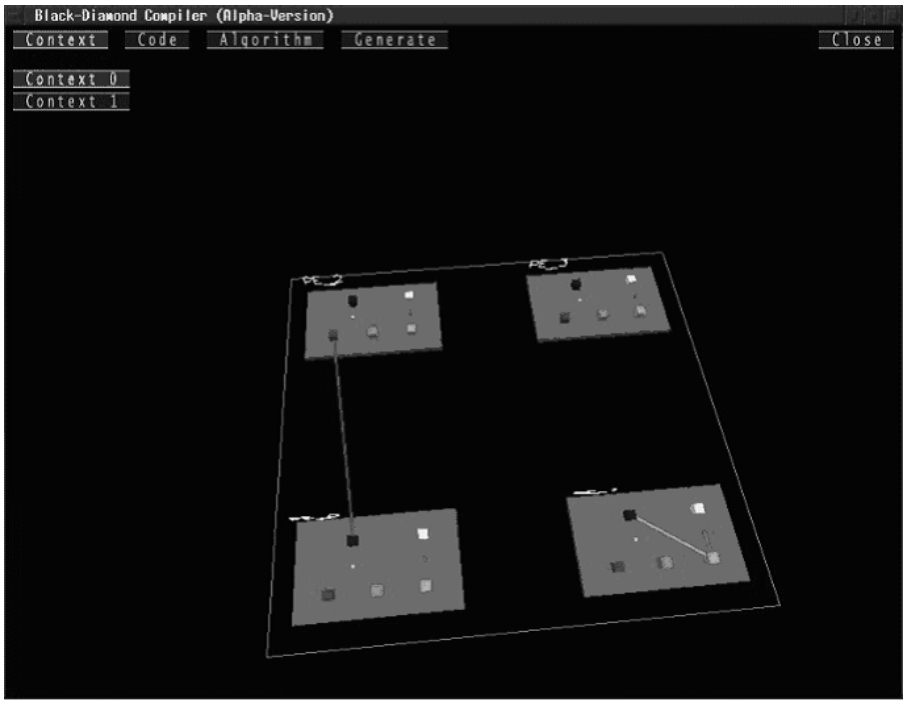

Figure A.6: Screenshot after adding restriction

one must be canceled to keep at least an enable input link. The corresponding description can be shown as below:

```
NODE  "DisCounT"  //  New Output of Register
    POSITION  2.0  0.0  0.0
    COST  20
    CONFIGURATION  "00_"  //  Reading register R0
    CONFIGURATION  "01_"  //  Reading register R1
    CONFIGURATION  "10_"  //  Reading register R2
    CONFIGURATION  "11_"  //  Reading register R3

//  Replace the "READ" node
NODE  "Read R0"  //  Read node of register R0
    POSITION  2.0  0.2  0.4
    COST  11
    CONFIGURATION  "(R0)"
NODE  "Read R1"  //  Read node of register R1
    POSITION  2.0  0.4  0.4
    COST  12
    CONFIGURATION  "(R1)"
NODE  "Read R2"  //  Read node of register R2
    POSITION  2.0  0.6  0.4
    COST  13
```

```
    CONFIGURATION  "(R2)"
NODE  "Read R3"  //  Read node of register R3
    POSITION  2.0  0.8  0.4
    COST  14
    CONFIGURATION  "(R3)"

//  Replace the "Write" node
NODE  "Write R0"  //  For sending data from register R0 to next context
    POSITION  2.0  0.2  0.7
    COST  0
    CONFIGURATION  "(Holding R0)"
    CONFIGURATION  "(Writing R0)"
NODE  "Write R1"  //  For sending data from register R1 to next context
    POSITION  2.0  0.4  0.7
    COST  0
    CONFIGURATION  "(Holding R1)"
    CONFIGURATION  "(Writing R1)"
NODE  "Write R2"  //  For sending data from register R2 to next context
    POSITION  2.0  0.6  0.7
    COST  0
    CONFIGURATION  "(Holding R2)"
    CONFIGURATION  "(Writing R2)"
NODE  "Write R3"  //  For sending data from register R3 to next context
    POSITION  2.0  0.8  0.7
    COST  0
    CONFIGURATION  "(Holding R3)"
    CONFIGURATION  "(Writing R3)"

//  Replace ADD "Read" and ADD "Write" at each PE
    ADD  "DisCounT"
    ADD  "Read R0"
    ADD  "Read R1"
    ADD  "Read R2"
    ADD  "Read R3"
    ADD  "Write R0"
    ADD  "Write R1"
    ADD  "Write R2"
    ADD  "Write R3"
```

Since the register file requires configuration bits to select reading register data, new node called “DisCounT” is added into the architecture and used as output instead of the “Read” node. The read node and write node are duplicated to be 4 sets for each register entry. The cost property of every read node is different. In the case, it tends to use register “R0” for storing intermediate data first since the Black-Diamond compiler routes data in minimum cost connection. Next, the links to connect the new node and restriction information must be added to control the register file at all PEs. The “DisCounT” node has 4 input links to select data from the “Read R0”, “Read R1”, “Read R2”, and “Read R3” nodes. The additional description of “PE_0” can be shown as below:

```
NET  "PE_0"@"Read R0"  ->  "PE_0"@"DisCounT","00_"
NET  "PE_0"@"Read R1"  ->  "PE_0"@"DisCounT","01_"
NET  "PE_0"@"Read R2"  ->  "PE_0"@"DisCounT","10_"
NET  "PE_0"@"Read R3"  ->  "PE_0"@"DisCounT","11_"

//  Replace NET "PE_0"@"Read" -> "PE_0"@"Write","(Holding)"
NET  "PE_0"@"Read R0"  ->  "PE_0"@"Write R0","(Holding R0)"
NET  "PE_0"@"Read R1"  ->  "PE_0"@"Write R1","(Holding R1)"
NET  "PE_0"@"Read R2"  ->  "PE_0"@"Write R2","(Holding R2)"
NET  "PE_0"@"Read R3"  ->  "PE_0"@"Write R3","(Holding R3)"


//  Replace NET "PE_0"@"Write Enable" -> "PE_0"@"Write","(Writing)"
NET  "PE_0"@"Write Enable"  ->  "PE_0"@"Write R0","(Writing R0)"
NET  "PE_0"@"Write Enable"  ->  "PE_0"@"Write R1","(Writing R1)"
NET  "PE_0"@"Write Enable"  ->  "PE_0"@"Write R2","(Writing R2)"
NET  "PE_0"@"Write Enable"  ->  "PE_0"@"Write R3","(Writing R3)"

//  Replace REGISTER "PE_0"@"Write" -> "PE_0"@"Read","(R)"
REGISTER  "PE_0"@"Write R0"  ->  "PE_0"@"Read R0","(R0)"
REGISTER  "PE_0"@"Write R1"  ->  "PE_0"@"Read R1","(R1)"
REGISTER  "PE_0"@"Write R2"  ->  "PE_0"@"Read R2","(R2)"
REGISTER  "PE_0"@"Write R3"  ->  "PE_0"@"Read R3","(R3)"

DISCOUNT  "PE_0"@"DisCounT"
    CONTROL  "PE_0"@"DisCounT","00_"  ->  "01_"
    CONTROL  "PE_0"@"DisCounT","00_"  ->  "10_"
    CONTROL  "PE_0"@"DisCounT","00_"  ->  "11_"
    CONTROL  "PE_0"@"DisCounT","01_"  ->  "00_"
    CONTROL  "PE_0"@"DisCounT","01_"  ->  "10_"
    CONTROL  "PE_0"@"DisCounT","01_"  ->  "11_"
    CONTROL  "PE_0"@"DisCounT","10_"  ->  "00_"
    CONTROL  "PE_0"@"DisCounT","10_"  ->  "01_"
    CONTROL  "PE_0"@"DisCounT","10_"  ->  "11_"
    CONTROL  "PE_0"@"DisCounT","11_"  ->  "00_"
    CONTROL  "PE_0"@"DisCounT","11_"  ->  "01_"
    CONTROL  "PE_0"@"DisCounT","11_"  ->  "10_"
    CONTROL  "PE_0"@"Write R0","(Writing R0)"  ->  "01_"
    CONTROL  "PE_0"@"Write R0","(Writing R0)"  ->  "10_"
    CONTROL  "PE_0"@"Write R0","(Writing R0)"  ->  "11_"
    CONTROL  "PE_0"@"Write R1","(Writing R1)"  ->  "00_"
    CONTROL  "PE_0"@"Write R1","(Writing R1)"  ->  "10_"
    CONTROL  "PE_0"@"Write R1","(Writing R1)"  ->  "11_"
    CONTROL  "PE_0"@"Write R2","(Writing R2)"  ->  "00_"
    CONTROL  "PE_0"@"Write R2","(Writing R2)"  ->  "01_"
    CONTROL  "PE_0"@"Write R2","(Writing R2)"  ->  "11_"
    CONTROL  "PE_0"@"Write R3","(Writing R3)"  ->  "00_"
    CONTROL  "PE_0"@"Write R3","(Writing R3)"  ->  "01_"
    CONTROL  "PE_0"@"Write R3","(Writing R3)"  ->  "10_"
```

```
//  Add "DisCounT" node for generating configuration data
GENERATE  "PE_0"@"DisCounT"
```

The restriction information forces to disable un-corresponding configuration bits at the "DisCounT" node when the write nodes or the "DisCounT" node itself select input data. By using the same source code as Example 6, the screenshot can be shown as Figure A.7. All register entries are stacked for representing register "R0", "R1", "R2", and "R3" from lower to upper.

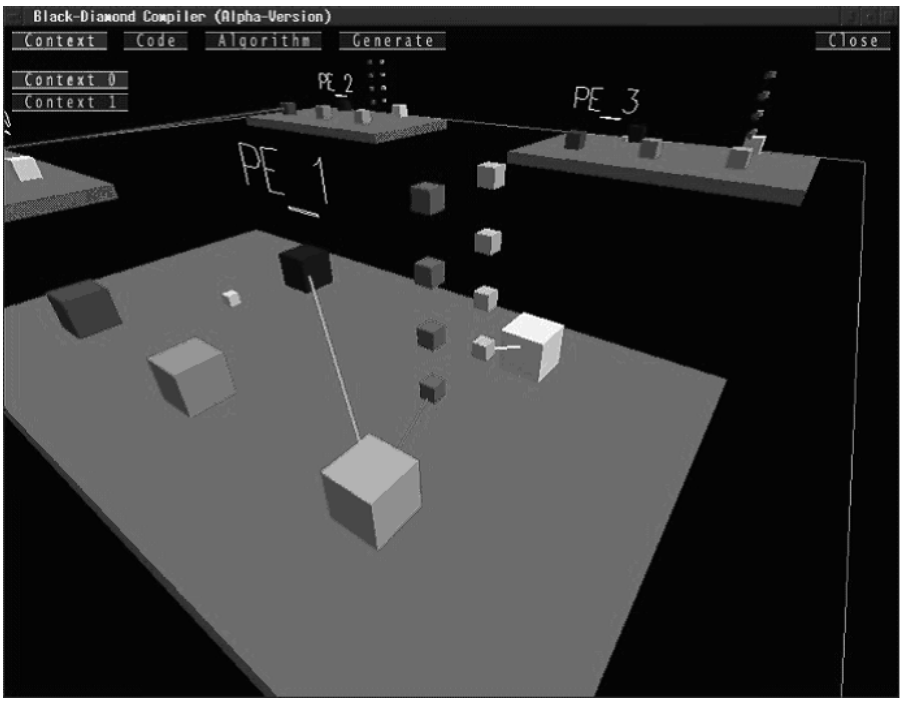

Figure A.7: Screenshot after replacing register file

A.10 Example 8: Automatic Passing Input Data to Output

Now, let turn back to consider the input source code at Example 2, it requires shifting pragma ("[@2]") to place the two calling functions into "PE_0" and "PE_3" for multiplying. Without the pragma, the multiply function must be placed in the next context since data from diagonal PE can not become input.

Some architectures have operation to transfer data from input of FU to output directly. The operation can be fixed to transfer data from the diagonal PE by selecting input link from the input of FU. After the operation is fixed for transferring the data, it can not be overwritten by placing the other operation anymore. In order to modify, it requires additional description as below:

```
//  Add new operation to "Output" node of FU
CONFIGURATION  "110_"  //  Pass input data to output

//  Add to control passing input data to output at each PE
NET  "PE_0"@"Input A"  ->  "PE_0"@"Output","110_"
```

The new operation for passing input data to output is added into the "Output" node of FU (operation "110_"). Then, a link is added to transfer data from "Input A" node to "Output" node at each PE. By using the same source code as Example 2 with shifting pragma, the two calling functions are placed into "PE_0" and "PE_1". However, the multiply function can be also placed in the same context at "PE_2" by passing the variable "b" data at "PE_3" as shown in Figure A.8.

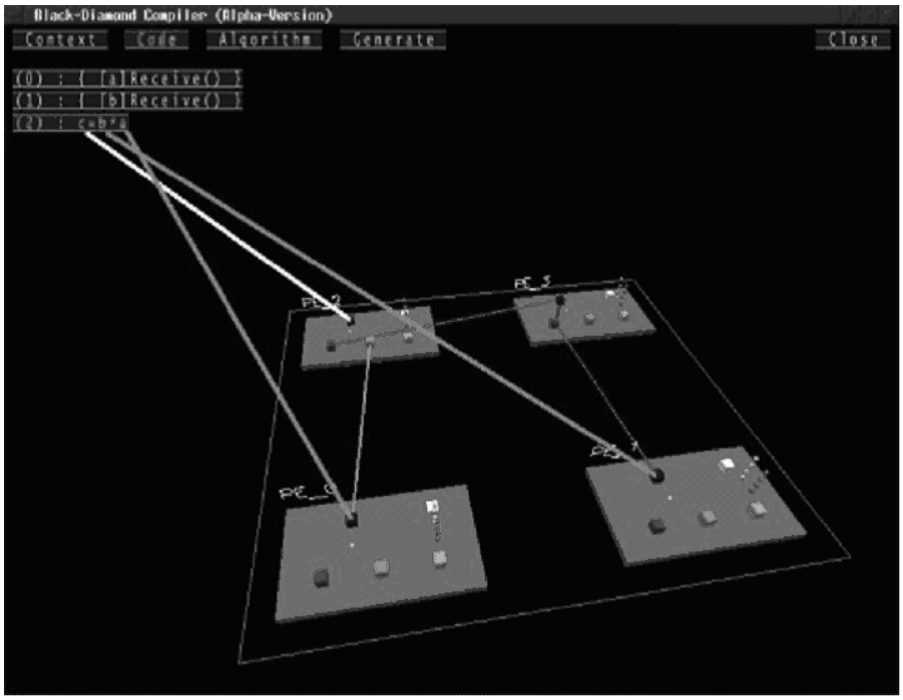

Figure A.8: Screenshot after adding by-passing operation into FU

A.11 Example 9: Applying RoMultiC in Configuration Data Generating

Some architectures apply RoMultiC scheme in configuration. Instead of using address to indicate a PE to be configured one by one, RoMultiC uses bitmap pattern to configure multiple PEs in parallel. Figure A.9 shows an example of

configuring in the RoMultiC scheme. There are bitmaps (multicast bits) in X and Y direction. Many PEs at the crossing points (digit "1" means activate configuring) can be configured with the same configuration data.

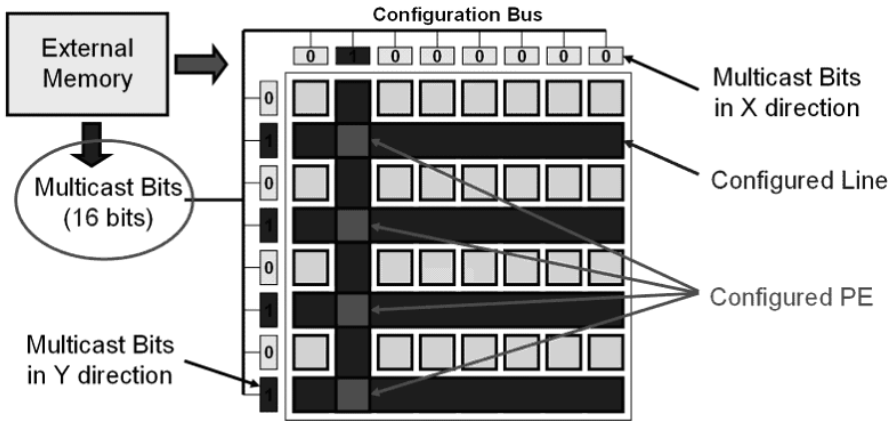

Figure A.9: Example of configuring RoMultiC scheme

In Figure A.10, each PE requires 16 multicast bits for multicasting on PE-Array size 8x8. Since the multicast bits are available on X and Y dimension, there are 2 activate bits to configure a PE. In order to configure multiple PEs with the same configuration data, The Black-Diamond compiler combines the bitmap by calculating logic "OR" all multicast bits to activate the bits for RoMultiC. It may require different number of configuration clock cycles depending on the pattern of duplicating configuration data in the PE-Array.

In Example 3 (Adding Description for Generating Configuration Data), the configuration data is generated by using keyword "GENERATE" to indicate the generated node. But in order to generate configuration data in RoMultiC, Black-Diamond compiler must have information of the bitmap on each PE for combining. And every node inside the PE must be known for finding the duplicated configuration data in different PE. Thus, the configuration data must be generated by indicating element instead of the node. Since every node inside the element is not generated (for example, read node and write node), the element should be modified as the description below:

```
ELEMENT  "PE_0"
    WHERE  -3.0  0.0  3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Constant"
    ADD  "DisCounT"
    ADD  "Write Enable"
    ADD_NOGEN  "Read R0"
    ADD_NOGEN  "Read R1"
    ADD_NOGEN  "Read R2"
```

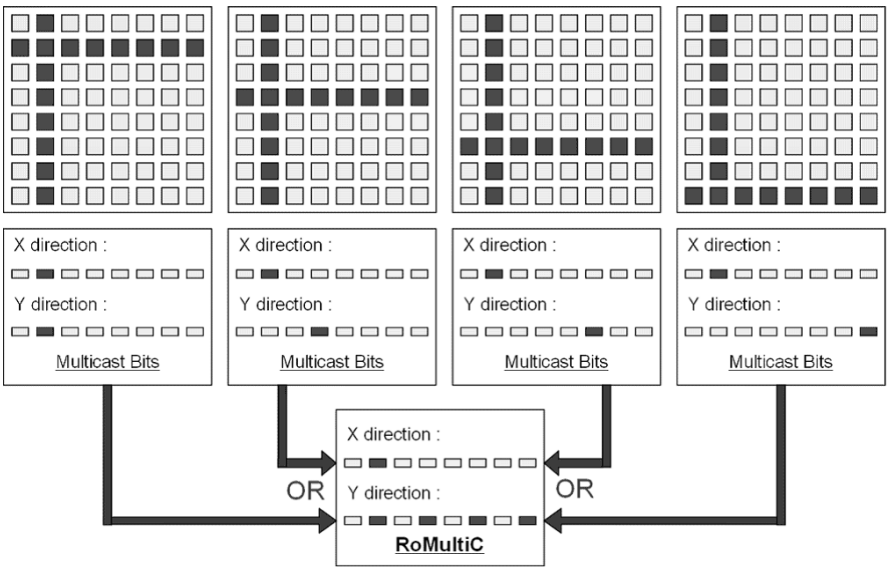

Figure A.10: Combining bitmap of each PE

```
    ADD_NOGEN  "Read R3"
    ADD_NOGEN  "Write R0"
    ADD_NOGEN  "Write R1"
    ADD_NOGEN  "Write R2"
    ADD_NOGEN  "Write R3"
    ADD_STATIC  "_"
    ADD_BITMAP_0
    ADD_BITMAP_1

    ADD_STATIC  "_"
    ADD_BITMAP_0
    ADD_BITMAP_1
    ADD_STATIC  "_"
    ADD_CONTEXT_INDEX  4
    ADD_COUNT
    ADD_NEWLINE
ELEMENT  "PE_1"
    WHERE  3.0  0.0  3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Constant"
    ADD  "DisCounT"
    ADD  "Write Enable"
    ADD_NOGEN  "Read R0"
    ADD_NOGEN  "Read R1"
    ADD_NOGEN  "Read R2"
```

```
    ADD_NOGEN  "Read R3"
    ADD_NOGEN  "Write R0"
    ADD_NOGEN  "Write R1"
    ADD_NOGEN  "Write R2"
    ADD_NOGEN  "Write R3"
    ADD_STATIC  "_"
    ADD_BITMAP_0
    ADD_BITMAP_1
    ADD_STATIC  "_"
    ADD_BITMAP_1
    ADD_BITMAP_0
    ADD_STATIC  "_"
    ADD_CONTEXT_INDEX  4
    ADD_COUNT
    ADD_NEWLINE
ELEMENT  "PE_2"
    WHERE  -3.0  0.0  -3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Constant"
    ADD  "DisCounT"
    ADD  "Write Enable"
    ADD_NOGEN  "Read R0"
    ADD_NOGEN  "Read R1"
    ADD_NOGEN  "Read R2"
    ADD_NOGEN  "Read R3"
    ADD_NOGEN  "Write R0"
    ADD_NOGEN  "Write R1"
    ADD_NOGEN  "Write R2"
    ADD_NOGEN  "Write R3"
    ADD_STATIC  "_"
    ADD_BITMAP_1
    ADD_BITMAP_0
    ADD_STATIC  "_"
    ADD_BITMAP_0
    ADD_BITMAP_1
    ADD_STATIC  "_"
    ADD_CONTEXT_INDEX  4
    ADD_COUNT
    ADD_NEWLINE
ELEMENT  "PE_3"
    WHERE  3.0  0.0  -3.0
    ADD  "Output"
    ADD  "Input A"
    ADD  "Input B"
    ADD  "Constant"
    ADD  "DisCounT"
    ADD  "Write Enable"
    ADD_NOGEN  "Read R0"
```

```
    ADD_NOGEN  "Read R1"
    ADD_NOGEN  "Read R2"
    ADD_NOGEN  "Read R3"
    ADD_NOGEN  "Write R0"
    ADD_NOGEN  "Write R1"
    ADD_NOGEN  "Write R2"
    ADD_NOGEN  "Write R3"
    ADD_STATIC  "_"
    ADD_BITMAP_1
    ADD_BITMAP_0
    ADD_STATIC  "_"
    ADD_BITMAP_1
    ADD_BITMAP_0
    ADD_STATIC  "_"
    ADD_CONTEXT_INDEX  4
    ADD_COUNT
    ADD_NEWLINE
```

Every node inside the element is going to be generated. The keyword "ADD" is replaced by keyword "ADD_NOGEN" for indicating un-generated node. The keyword "ADD_BITMAP_0" and "ADD_BITMAP_1" are used for adding bitmap node to the PE (BITMAP_1 is active bit). In the same manner as the example 3, some special commands can be added to control the generating as shown in the Table A.2.

Table A.2: Pair of generating commands

Outside Element	Inside Element
STATIC	ADD_STATIC
CONTEXT_INDEX	ADD_ CONTEXT_INDEX
COUNT_CONFIGURATION	ADD_COUNT
NEW_LINE	ADD_NEWLINE

In order to configure the PE-Array, it may require different number of configuration clock cycle depending on the pattern of duplicate configuration data in the PE-Array. Black-Diamond compiler requires information of elements to be configured in RoMultiC. Only a duplicated configuration data are multicasted when the command is executed. User must indicate looping to loop generating the configuration data in RoMultiC until all PEs are configured. The executed command for generating configuration data in RoMultiC can be shown as the description below:

```
//  Add keyword "SELECT" to "Write Enable" node
    SELECT  0

    RETURN_POINT
        GEN_ROMULTIC
```

```
            CONFIGURE  "PE_0"
            CONFIGURE  "PE_1"
            CONFIGURE  "PE_2"
            CONFIGURE  "PE_3"
    LOOP_ROMULTIC
```

The keyword "GEN_ROMULTIC" is used for indicating group of PE which can be configured in RoMultiC scheme by using the followed keyword "CONFIGURE". The keyword "RETURN_POINT" and keyword "LOOP_ROMULTIC" are used for looping to generate configuration data until all indicated PEs are configured.

In order to reduce the number of configuration clock cycle, Black-Diamond changes selecting input at the node which is not fixed automatically. However, the "Write Enable" node should not be changed since it may overwrite data storing in the register without routing between contexts. The keyword "SELECT" must be used to fix selecting input at the "Write Enable" node to protect from the automatic changing.

Some architectures can load multiple sets of RoMultiC to configure in parallel (For example, the switching element of MuCCRA-1 and MuCCRA-2 architecture). Multiple keyword "GEN_ROMULTIC" can be used as the description below:

```
//  Add new elements for generating white space and executing special command
ELEMENT  "Space Element"
    ADD_STATIC  "_"
ELEMENT  "End Element"
    ADD_STATIC  "_"
    ADD_CONTEXT_INDEX 4
    ADD_COUNT
    ADD_NEWLINE

//  Remove the duplicate part as the "End Element" in each PE

//  Replace the RoMultiC looped
    RETURN_POINT
        GEN_ROMULTIC
            GEN_ORDER  2
            CONFIGURE  "PE_0"
            CONFIGURE  "PE_1"
            CONFIGURE  "PE_2"
            CONFIGURE  "PE_3"
        GEN_ELEMENT  "Space Element"
            GEN_ORDER  1
        GEN_ROMULTIC
            GEN_ORDER  0
            CONFIGURE  "PE_0"
            CONFIGURE  "PE_1"
            CONFIGURE  "PE_2"
            CONFIGURE  "PE_3"
```

```
    GEN_ELEMENT  "End Element"
        GEN_ORDER  3
LOOP_ROMULTIC
```

Between the RoMultiC configuration data, a user can add white space or execute special command by generating the new element by using keyword "GEN_ELEMENT". Moreover, the generating order can be switched by setting "GEN_ORDER" property.

A.12 Example 10: Improving Graphic

This example is not so important but allows a user to understand the architecture by graphic user interface easily. One of difficulty to understand this example target architecture is: where is the component inside the reconfigurable element? All previous screenshots show flowing node represented input and output of FU and register. Adding drawing box (using keyword "BOX") into graphic makes it become clearly to be understood as shown in Figure A.11.

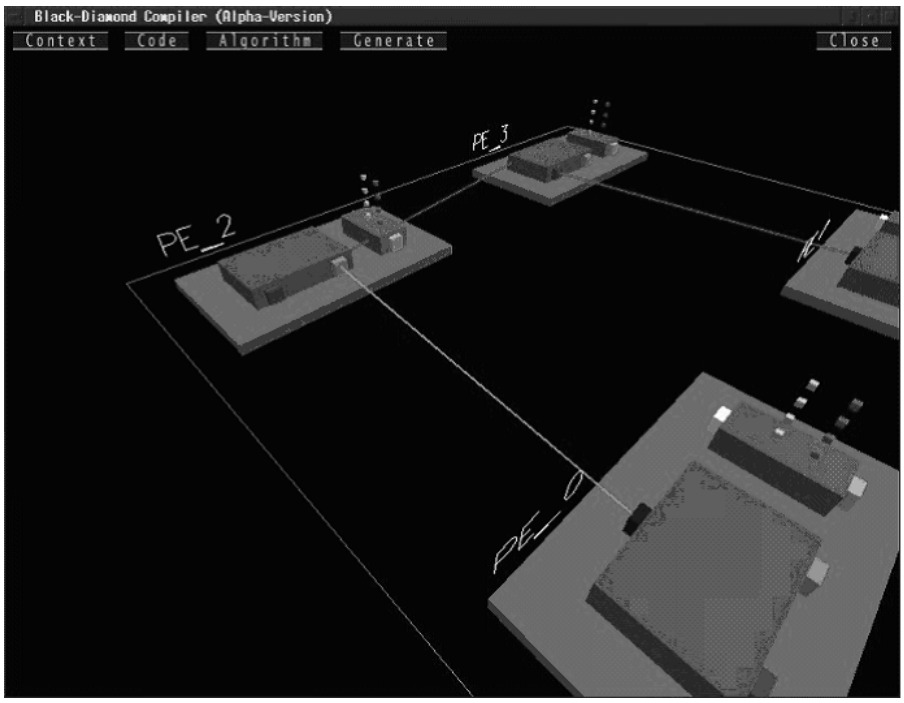

Figure A.11: Screenshot after adding drawing box to FU and register

However, a new problem occurs. The strain line to represent connection from "PE_3" to "PE_2" looks like drilling the register component and does not connect to the "Input A" of FU. This problem can be solved by adding new nodes ("NODE_SIZE 0" and "COST 0") floating over the input node for receiving data at each input link before reaching to the input node. The screenshot

can be shown as Figure A.12. This technique can be also used for making arrow at the end of connection too.

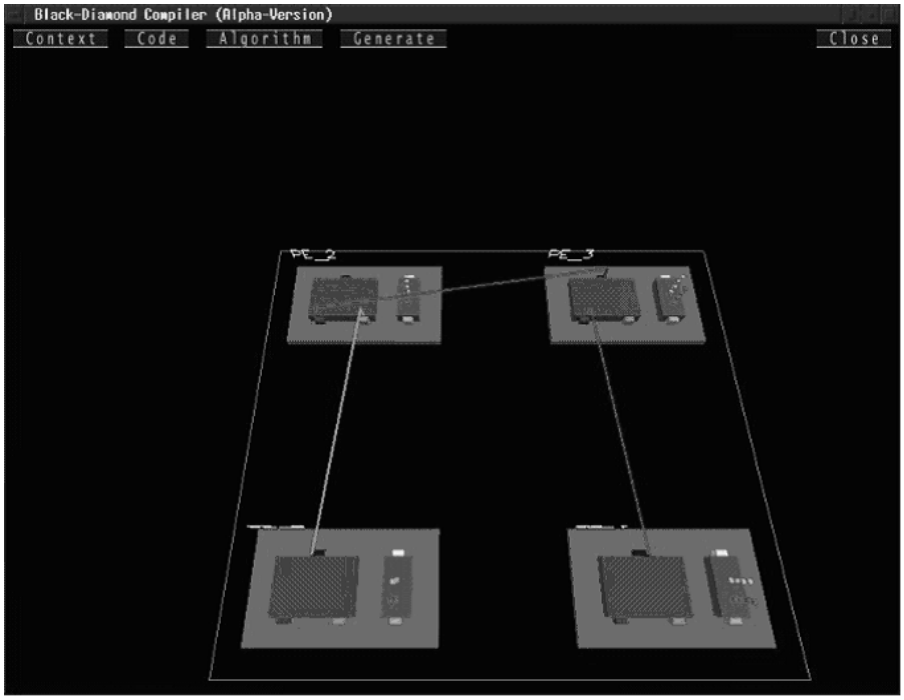

Figure A.12: Screenshot after adding floating nodes over input and output node

At the output node of FU, the different operations are represented and a user can not know what operation is placed at the FU without looking at the generated configuration data. The selecting input link can be shown by using drawing text in the graphic. By replacing the keyword "CONFIGURATION" with "CONFIGURATION_TEXT" at the "Output" node as the description below, the text followed "," is draw in graphic corresponding to the selection as shown in Figure A.13. The register file can be also modified to show the reading and writing register number too.

```
CONFIGURATION_TEXT  "000_" , "Addition"  //  Addition Operation
CONFIGURATION_TEXT  "001_" , "Subtraction"  //  Subtraction Operation
CONFIGURATION_TEXT  "010_" , "Multiply"  //  Multiply Operation
CONFIGURATION_TEXT  "011_" , "Receive"  //  Receive I/O data
CONFIGURATION_TEXT  "100_" , "Send_A"  //  Send input A to I/O Bus
CONFIGURATION_TEXT  "101_" , "Send_B"  //  Send input B to I/O Bus
CONFIGURATION_TEXT  "110_" , "Pass_Input"  //  Pass input data to output
CONFIGURATION_TEXT  "111_" , "Constant"  //  Send constant data to output
```

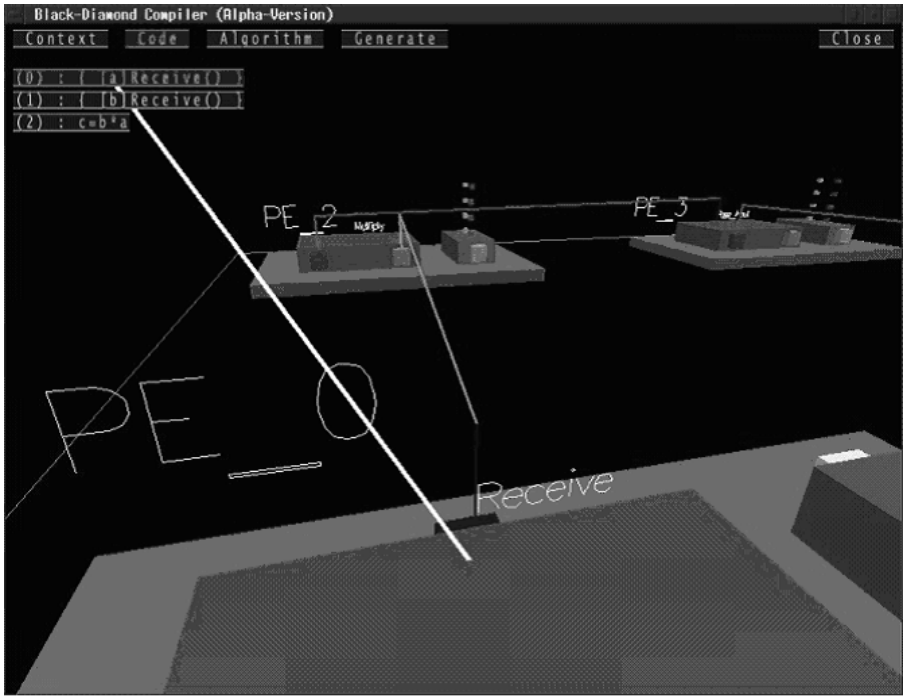

Figure A.13: Screenshot after modifying the "Output" node to show text corresponding to selecting input

Bibliography

[1] Esterel simulator. *http://www-sop.inria.fr/esterel-org/*.

[2] Synopsys physical compiler v2004.12-sp1. *http://www.synopsys.com/*.

[3] Yacc. *http://dinosaur.compilertools.net*.

[4] Hideharu Amano. A survey on dynamically reconfigurable processors. *IEICE Transactions on Communications*, E89-B(12):3179–3187, 2006.

[5] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. *ARC 2007*, LNCS 4419:1–13, 2007.

[6] The IMPACT Group. *http://www.crhc.uiuc.edu/Impact/*.

[7] Y. Hasegawa, S. Tsutsumi, T. Nakamura, T. Nishimura, T.Sano, M. Kato, S. Saito, and H. Amano. Design and implementation of the dynamically recofigurable processor muccra-1. *Proc. of SACSIS 2007*, pages 95–102, 2007.

[8] Yohei Hasegawa, Shohei Abe, Shunsuke Kurotaki, Vu Manh Tuan, Naohiro Katsura, Takuro Nakamura, Takashi Nishimura, and Hideharu Amano. Performance and power analysis of time-multiplexed execution on dynamically reconfigurable processor. *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[9] M. Hiromoto, S. Kouyama, H. Ochi, and Y. Nakamura. A retargetable compiler for cell-array-based self-reconfigurable architecture. *IJCSNS Computer Science. & Network Security*, 7(4):131–139, 2007.

[10] Masayuki Hiromoto, Shin'ichi Kouyama, Hiroyuki Ochi, and Yukihiro Nakamura. A retargetable compiler for cell-array-based self-reconfigurable architecture. *IJCSNS Computer Science. & Network Security*, 7(4):131–139, 2007.

[11] Ishikawa Hiroyuki, Shimizu Sho, Arakawa Yutaka, Yamanaka Naoaki, and Shiba Kosuke. Shortest path algorithm on parallel reconfigurable processor dapdna-2. *IEICE Technical Report*, NS2005-162:17–20, 2006.

[12] H. Ito, K. Oguri, K. Nagami, R. Konishi, and T. Shiozawa. The plastic cell architecture for dynamic reconfigurable computing. *In Proceedings of 9th International Workshop on Rapid System Prototyping*, pages 39–44, 1998.

[13] M. Kato, Y. Hasegawa, and H. Amano. Evaluation of muccra-d: A dynamically reconfigurable processor with directly interconnected pes. *Proc. of ERSA 2008*, 2008.

[14] T. Kodama, T. Tsunoda, M. Takada, H. Tanaka, Y. Akita, M. Sato, and M. Ito. Flexible engine: A dynamic reconfigurable accelerator with high performance and low power consumption. *Proc. of COOL Chips IX*, pages 393–408, 2006.

[15] Y. Kurose, I. Kumata, M. Okabe, H. Hanaki, K. Seno, K. Hasegawa, H. Ozawa, S. Horiike, T. Wada, S. Arima, K. Taniguchi, K. Ono, H. Hokazono, T. Hiroi, T. Hirano, and S. Takashima. A 90nm embedded dram single chip lsi with a 3d graphics, h.264 codec engine, and a reconfigurable processor. *Hot Chips*, 16, 2004.

[16] B. Levine. Kilocore: Scalable, high-performance, and power efficient coarse-grained reconfigurable fabrics. *Proc. of Int. Symp. on Advaned Reconfigurable Systems*, pages 129–158, 2005.

[17] Bingfeng Mei, Andy Lambrechts, Jean-Yves Mignolet, Diederik Verkest, and Rudy Lauwereins. Architecture exploration for a reconfigurable architecture template. *IEEE Design & Test of Computers*, pages 90–101, March.April 2005.

[18] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. *Proc. IEEE Int 'l Conf. Field-Programmable Technology, IEEE Press*, pages 166–173, 2002.

[19] M. Motomura. A dynamically reconfigurable processor architecture. *Microprocessor Forum*, 2002.

[20] M. Motomura. A dynamically reconfigurable processor architecture. *Microprocessor Forum*, 2002.

[21] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi. Plastic cell architecture: A scalable device architecture for general-purpose reconfigurable computing. *IEICE Trans. on Electronics*, E81-C(9):1431–1437, 1998.

[22] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi. Plastic cell architecture: A scalable device architecture for general-purpose reconfigurable computing. *IEICE Transaction on Electronics*, E81-C(9):1431–1437, 1998.

[23] M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, and M. Glesner. The xpp architecture and its co-simulation within the simulink environment. *Proc. of FPL*, pages 761–770, 2004.

[24] M. Petrov, T. Murgan, F. May, M. Vorbach, P. Zipf, and M. Glesner. The xpp architecture and its cosimulation within the simulink environment. *Proc. FPL*, pages 761–770, 2004.

[25] Inc. Rapport. *http://www.rapportincorporated.com.*

[26] T. Stansfield. Using multiplexers for control and data in d-fabrix. *Proc. of FPL*, pages 416–425, 2003.

[27] T. Sugawara, K. Ide, and T. Sato. Dynamically reconfigurable processor implemented with ipflex's dapdna technology. *IEICE Trans. Inf. & Syst.*, E87-D:1997–2003, 2004.

[28] Takayuki Sugawara, Keisuke Ide, and Tomohiro Sato. Dynamically reconfigurable processor implemented with ipflex's dapdna technology. *IEICE Trans. on Inf.&Syst.*, E87-D(8):1997–2003, 2004.

[29] Xinan Tang, Manning Aalsma, and Raymond Jou. A compiler directed approach to hiding configuration latency in chameleon processors. *Proc. FPL*, LNCS 1896:29–38, 2000.

[30] Satoshi Tsutsumi, Vasutan Tunbunheng, Yohei Hasegawa, Hiroki Matsutani, Adepu Parimala, Takuro Nakamura, Takashi Nishimura, Toru Sano, Masaru Kato, Shotaro Saito, Naomi Seki, Keiichiro Hirai, Mao KaiYi, and Hideharu Amano. A scheduling algorithm for multicast configuration. *RECONF2006-73*, pages 49–54, 2007.

[31] V. Tunbunheng, M. Suzuki, and H. Amano. Data multicasting procedure for increasing configuration speed of coarse grain reconfigurable devices. *IEICE Trans. Inf. & Syst.*, E90-D(2):473–481, 2007.

[32] F. J. Veredas, M. Scheppler, W. Moffat, and B.Mei. Custom implementation of the coarse-gained reconfigurable adres architecture for multimedia purposes. *Proc. of FPL*, pages 106–111, 2005.