ABSTRACT

An abstract of the thesis of Chen, Qihong for the Master of Science in Electrical and Computer Engineering presented June 25, 1998.

Title: The Design of Cube Calculus Machine Using SRAM-based FPGA Reconfigurable Hardware DEC's PeRLe-1 Board

Cube calculus is an algebraic model used to process boolean functions. Cube calculus operations are widely used in logic optimization, logic synthesis, image processing and recognition, machine learning, and other applications which require massive logic operations.

The cube calculus operations can be carried out on general-purpose computers. Since these operations can involve several levels of nested loops, this approach has poor performance.

A cube calculus machine which has a special data path designed to speed up cube calculus operations is presented in this thesis. This cube calculus machine can execute cube calculus operations 10 to 25 times faster than the software approach on a general-purpose computer.

This thesis proposes a complete design of the Cube Calculus Machine Version II (CCM2). In this design, the CCM acts as a coprocessor of the host computer; it accepts a set of instructions that let the CCM carry out cube calculus operations. This design is mapped on a reconfigurable hardware DEC PeRLe-1 board.

ACKNOWLEDGEMENTS

I would like to thank Dr. Marek A. Perkowski, my advisor, who patiently guided and encouraged me through this master program.

I also would like to thank Dr. Douglas V. Hall and Dr. Sarah Mocas for their valuable comments, suggestions on my thesis and their willingness to be in my thesis defense committee.

Finally, I would like to thank Laura Riddell and Shirley Clark for their continuous support and help throughout my studies at Portland State University.

THE DESIGN OF CUBE CALCULUS MACHINE USING SRAM-BASED FPGA RECONFIGURABLE HARDWARE DEC'S PERLE-1 BOARD

by

CHEN, QIHONG

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1998

THESIS APPROVAL

The abstract and thesis of Chen, Qihong for the Master of Science in Electrical and Computer Engineering were presented June 25, 1998, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

____
Marek A. Perkowski, Chair

____
Douglas V. Hall

____
Sarah Mocas
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

____
Lee Casperson
Department of Electrical Engineering

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by ______________________________ on ______________

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Cube Calculus is an algebraic model used to process boolean functions [26, 27, 28, 29, 30, 34, 37, 16, 32, 43]. Cube calculus operations are widely used in logic optimization, logic synthesis, image processing and recognition, machine learning, and other applications which require massive logic operations.

The cube calculus operations can be implemented on general-purpose computers at the cost of control overhead. Almost all general-purpose computers are of traditional Von Neumann computer architecture. In traditional Von Neumann computer architecture, the control is located in the program that is stored in the memory. This results in a considerable control overhead. Since the instructions have to be fetched from the memory, if an algorithm contains loops, the same instruction will be read many times. This makes the memory interface the bottleneck of these architectures, especially when the memory bus is not as fast as the internal processor bus. Also, in many commonly used computer architectures, there is very little parallel processing, even in modern RISC or Pentium processors. The cube calculus operations involve several-level nested loops, that leads to poor performance on conventional computer architectures. The slow processing speed of the cube calculus operations is the bottleneck of many application such as logic synthesis, machine learning, and others.

This thesis presents a variant design of Cube Calculus Machine version II (CCM2 for short), a special hardware which has an innovative data path designed to execute cube operations efficiently. This design is mapped onto a DEC PeRLe-1 board, one of the earliest platforms for Reconfigurable Computing (RC). In our design, the CCM acts as a co-processor to the host computer. The host computer needs only to issue the commands (CCM instructions) to the CCM and obtain the result from the CCM.

The first version of the CCM was introduced in 1989 by Luis S. Kida and Dr. Marek Perkowski as a general purpose logic computer with a data path designed specifically to execute cube calculus operations. The second version of CCM (CCM2) was designed by a group of students in Dr. Perkowski's EE510 Computer Architecture for Robotics and Artificial Intelligence course during the 1991-1992 school year.

The CCM architecture is based on an observation that many cube calculus operations can be described in a general equation of certain logic/set equation (see Chapter 2 and Chapter 3). In this general format, the literals of the operand cubes are divided into three types of literals, according to a *relation* of literals of operand cubes. This relation is defined by the type of cube calculus operation. As the resultant cubes are generated, the literals of the operand cubes go through a sequence of literal types. The value of a literal in the resultant cube depends not only on the corresponding literals in the operand cubes, but also on the current types of these literals.

This general format of an equation is implemented in the Iterative Logic Unit (ILU for short) of the CCM architecture. The ILU consists of an array of Iterative Cells (ITs). Each IT can process two bits, either as a binary variable or a portion of a multi-valued variable in *positional notation*. Each IT contains a Finite State Machine (FSM) to keep track of the current type of literal that is being processed. The value of a resultant literal is not only calculated from the operand literals, but also from the current state of the FSM of the given IT.

A considerable part of the control of the CCM is implemented in its data path, which is the ILU. Once the ILU has received a cube calculus operation, the only control that is needed from the Control Unit of the ILU (called *Operation Control Unit* in our design) is the clock signal for the output of the resultant cubes.

One version of the ILU with 8 ITs of the CCM2 has been implemented by David W. Foote on 2 Xilinx XC3090 FPGAs [17]. Simulation and experimental results have shown that cube calculus operations executed on the FPGA Cube Calculus Machine are 10 to 25 times faster than those executed by using the software approach on a conventional computer. The larger the input cube, the more speed gain can be obtained by using the CCM.

Since our design is targeted for DEC PeRLe-1 , a pioneer of Reconfigurable Computing (RC), let us take a look at Reconfigurable Computing. Reconfigurable Computing [1, 2] is a relatively new term that describes computer and controller systems that can be configured "on-the-fly" to meet the needs of the target application.

FPGAs contain a large number of diverse logic gates and registers, which can be connected together in widely different ways to achieve a desired function. SRAM-based variants extend the capabilities of standard FPGA by allowing new configuration data to be downloaded into the device by the main system in a fraction of a second. Therefore, SRAM-based FPGAs are reconfigurable computing elements.

Computing circuits built from SRAM-based FPGAs can meet the true goal of parallel processing, executing algorithms in circuitry with inherent parallelism of hardware, while avoiding the instruction fetch and load/store bottlenecks of traditional Von Neumann architectures. There are many computationally intensive algorithms that can benefit from being partially or wholly implemented in hardware. Typically, these algorithms are too specialized to justify the expense of manufacturing custom IC devices. Just as often, the "algorithm space" is very large, and it may be impractical to perform enough simulations to find the optimal approach before committing to custom hardware.

With an FPGA-based "configurable co-processor" the user can design exactly the special hardware required for a given task without having to construct new hardware for each application. Different tasks can be time-multiplexed into the same FPGAs. Errors can be corrected and different algorithmic approaches explored, with no further hardware expense.

Several universities and research laboratories have been exploring the use of SRAM-based FPGAs to implement multi-purpose, high-speed co-processors for accelerating operations in computer systems. Using these systems, desktop workstations have delivered the performance of supercomputers for specific applications. In particular, two projects have gained considerable attention: the PeRLe systems from DEC's Paris Research Lab, and the SPLASH machines from the Supercomputer Research Center. These systems consist of FPGA-based attached processors in engineering workstations, complete with programming tools and run-time environments, and have been the target for a variety of "real world" applications.

DEC's Paris Research Laboratory has designed and implemented four generations of FPGA-based reconfigurable co-processors called Programmable Active Memories (PAMs) [5, 6, 7, 8, 9]. The most-widely used version, the PeRLe-1 , is based on a 4×4 array of XC3090 FPGAs. Developed applications include long multiplication, RSA cryptography, heat and LaPlace equations, a sound synthesizer and many others [9].

The Supercomputer Research Center (SRC) has designed two generations of the SPLASH processor based on a linear array of FPGAs [2, 4]. The SPLASH-1 includes a 32-stage linear logic array with a VME interface to a Sun workstation. Each stage consists of an XC3090 FPGA and a 128 Kbyte static memory buffer. The first application of the SPLASH-1 was to implement a systolic algorithm for one-dimensional pattern matching during DNA research, where it out-performed a Cray-2 by a factor of 325 and a custom-built nMOS device by a factor of 45. The SPASH-2 system is based on XC4010 FPGA devices, and has been used to implement a number of applications, including string searches and image processing[4].

The successes of these and other early projects have fueled the interest of the research community. The IEEE now devotes an entire annual workshop to FPGA-based computing, IEEE workshop on FPGAs for Custom Computing Machines (FCCM)[3].

FPGA-based reconfigurable processors are available for a broad range of applications, including scientific computing, database manipulation, design automation, cryptography, image processing and real-time digital signal processing. FPGA-based processors can exploit the fact that most of the processing time for a compute-intensive tasks is spent in a relatively small portion of the code, and hardware acceleration of that portion can significantly improve the overall performance.

In the long term, expected advances in FPGA density, performance and architecture may offer more significant advances than single processor solutions can promise. While many important hardware and software challenges remain, it is conceivable that reconfigurable processors constructed from SRAM-based programmable logic eventually will replace today's general-purpose processors, providing the basis for systems that automatically will alter their own hardware to best solve the problem at hand.

The CCM2 of PSU was originally meant to be fit into silicon, and the design of the CCM2 chip had been layed out and simulated from the extracted circuit. Due to the lack of VLSI fabrication funding, the attempt to fabricate CCM2 chip failed.

As of Spring 1992, the "old" Xilinx tools and two Xilinx XC3090 chips were available to the EE department at PSU, and the core of the CCM2, the ILU with 8 ITs, was mapped onto these two chips by David W. Foote [17].

As of 94/95, a DEC PeRLe-1 board was available to the EE department at PSU. Unfortunately, there was no documentation and development tools available. As of July 97, the documentation of the board became available. As of this writing, the development tools for DEC PeRLe-1 board are still not usable because a special C++ compiler that can collaborate with this board is needed. We expect the complete development tools to be available in the near future.

The design of the CCM presented in this thesis is a variant of CCM2, and is targeted at the PeRLe-1 board. Therefore, this design tightly depends on the architecture of the PeRLe-1 board. For the first time it is a complete design, including Global Control (GCU), ILU with 15 ITs, and Operation Control Unit (OCU). My design accepts a set of instructions that are optimized for practical applications. The CCM communicates with the host computer through the input and output FIFOs. The host issues the commands to the CCM through the input FIFO, and gets the results from the output FIFO. The instructions do not limit the CCM to execute only the cube calculus operations introduced in this thesis, actually, the CCM can execute all cube calculus operations that can be described in the general format of equations (see Chapter 2 and Chapter 3 for detail).

Once the development tools for the PeRLe-1 board become available, this design will be realized on the PeRLe-1 board, and the corresponding C/C++ based cube operation library (see section section 6.1) will be implemented on the host computer.

The following chapters are organized as follows: Chapter 2 presents a subset of cube calculus operations. Chapter 3 introduces the Iterative Logic Unit of the CCM2. This section gives a detailed description of the design and functionality of the ILU. Chapter 4 introduces the DEC PeRLe-1 board in detail, the programming method is also given in this chapter. Chapter 5 presents our design of the CCM2, including the design of Global Control Unit, and mapping on the PeRLe-1 board. Chapter 6 presents CCM assembly language. Chapter 7 present the simulation result of our design described in VHDL. Chapter 8 evaluates our design of the CCM. Chapter 9 shows some possible applications of the CCM. Chapter 10 gives a conclusion of the tasks that were accomplished in this thesis and a list of further work that is necessary to be performed to complete a ready-to-be-used Cube Calculus Machine.

In this thesis, the notations shown in Table 1.1 are used to represent different type of objects.

Table 1.1: Notation

Type	**Font**	**Example**
variable	lower case letter	$a,\ b,\ x,\ x_i$
literal	lower case letter	$\bar{a},\ x_i^{0,1,3}$
cube	upper case letter	$A,\ B$
array of cubes	upper case letter with arrow on head	$\vec{A},\ \vec{B}$
set	curly braces	$\{0, 1, \ldots, n\}$
the name of set	upper case letter	$P,\ S$

CHAPTER 2

A Review of Cube Calculus

Most of the efficient modern logic synthesis programs use the so-called *cube calculus* to represent and process Boolean functions. This representation is used in U.C.Berkeley programs, including the well-known Espresso [37], MIS II and SIS, and many others [29, 30, 16, 32]. This calculus has been used for Boolean minimizers, tautology and satisfiability checkers, verifiers and other software tools[31].

In this chapter, the concept of a *set* is presented first because it is used as a fundament of cube calculus; then the concept of cube and the cube calculus are presented. The last part of this chapter presents positional notation of cubes and how to carry out the cube calculus in positional notation.

2.1 Sets

A *set* is a collection of objects called *elements* or *members*. As listed in Table 1.1, we use curly braces to indicate sets.

For instance, the set of all integer between 0 and 5 is written as:

$$\{0,1,2,3,4,5\}$$

the infinite set of all positive, odd integers can be describe by:

$$\{1,3,5,7, \ldots \}$$

The membership of a element a in a set A is denoted by $a \in A$ to mean "a is an element of A". A set which has no element is called an *empty set* (denoted by $\emptyset$). The empty set is a subset of all sets. The elements contained in a set are either listed explicitly or described by their properties.

The following relations between two sets are used in cube calculus:

- Two sets A and B are *equal*, or *identical*, if they contain precisely the same elements. It is denoted by $A = B$.

- A set A is said to be a *subset* of B if every element of A is also an element of B. It it denoted by $A \subseteq B$.

- If $A \subseteq B$, and B contains at least one element which is not contained in A, then A is said to be *proper subset* of B. It is denoted by $A \subset B$.

The elements of the sets are taken from *universal set* (U). The following basic *set operations* are used in cube calculus:

- The *complement* of A in universal set U (denoted by $\neg A$) is the set of all elements of U that are not elements of A.

- The *intersection* of A and B (denoted by $A \cap B$) is the set containing the elements that are in both A and B.

- The *union* of A and B (denoted by $A \cup B$) is the set containing the elements that are in either A or B.

Example 2.1. Assuming that the universal set U is {0,1,2,3,4,5}, a set A is {0,1,2,3} and another set B is {2,3,4}. Then $\neg A = \{4, 5\}$, $A \cap B = \{2, 3\}$, and $A \cup B = \{0, 1, 2, 3, 4\}$.

The universal set U of possible values of a binary variable is {0,1}. For a binary variable a, literal $\bar{a}$ means that literal is true when variable a is 0, and can be described by $a^{\{0\}}$, where {0} is the true set of literal $\bar{a}$. More detailed discussion on sets can be found in [35, 36].

2.2 The Concept of A Cube

The basic concepts of cube calculus are a *cube* and an *array of cubes*. A *cube* is a *product of literals*. For example, product $\bar{a}bc\bar{d}$ is a cube. An *array of cubes* is a sum of cubes. A logic function can be represented by a cube or an array of cubes. For instance, a simple 2-input binary logic function AND can be described by a cube as: $f_{AND}(a,b) = ab$; another simple 2-input binary logic function XOR (exclusive OR) can be described by an array of cubes as: $f_{XOR}(a,b) = \bar{a}b + a\bar{b}$.

In a binary logic, a *literal* is a binary variable with negation or without negation (x or $\bar{x}$). In a multi-valued logic a literal ($x_i^{S_i}$) is a variable (x_i) with its set of values (S_i) for which the variable is true.

A multi-valued input, two-valued output, incompletely specified switching function (*multi-valued function* for short) is a mapping:

$$f(x_1, x_2, \cdots, x_n) : P_1 \times P_2 \times \cdots \times P_n \mapsto B \tag{2.1}$$

where x_i is a multi-valued (p_i-valued) variable, $P_i = \{0, 1, 2, \cdots, p_i - 1\}$ is the *set of values* that variable x_i may assume, $B = \{0, 1, \mathrm{x}\}$ (x denotes a *don't care value*). n denotes the number of variables (positions). For any subset $S_i \in P_i$, $x_i^{S_i}$ is a literal of x_i representing the function such that:

$$x_i^{S_i} = \begin{cases} 1 & \text{if } x_i \in S_i \\ 0 & \text{if } x_i \notin S_i \end{cases} \tag{2.2}$$

S_i is called *true values set* (*true set* for short) of literal $x_i^{S_i}$. For example, a four-valued input logic of $x^{\{1,2,3\}} = 1$ if $x \in \{1, 2, 3\}$, which means $x^{\{1,2,3\}} = 1$ if $x = 1$ or $x = 2$ or $x = 3$; otherwise, $x^{\{1,2,3\}} = 0$. We always assume that the set of possible values of a n-valued logic variable is $\{0, 1, 2, \ldots, n-1\}$.

A product of literals, $x_1^{S_1} x_2^{S_2} \cdots x_n^{S_n}$, is refereed to as a *product term* (also called *product* or *term* for short), and can be represented by a cube. A product term that includes literals for all function variables $x_1, x_2, \ldots, x_n$ is called a *full term.* Any literal of the form $x_i^{P_i}$ is always equal to 1 since the literal is true for all possible values of x_i, thus $x_i^{P_i} x_j^{S_j}$ can be written as $x_j^{S_j}$.

A sum of products is denoted as a *Sum-Of-Products Expression* (SOPE) while a product of sums is called a *Product-Of-Sums Expression* (POSE). An EXOR of products will be called a *Exclusive Sum Of Products* form (ESOP). A product of EXORs will be called a *Product Of Exclusive Sums* expression (POES). SOPE, POSE, ESOP and POES are all represented as an arrays of cubes. Products of SOPEs (PSOPEs) are also used for the *Generalized Propositional Formulas* form. They are represented as arrays of arrays of cubes.

The *degree* of a cube is the number of literals in the cube that are not equal to one (in other word, $P_i \neq S_i$).

Example 2.2. The degree of cube $a^{\{0\}}b^{\{1\}}c^{\{1\}}d^{\{0,1\}}$ is 3 (assuming a,b,c and d are binary variables).

The *difference* of two cubes is the number of variables for which the corresponding literals have different true sets. The *distance* of two cubes is the number of variables for which the corresponding literals have disjoint true sets.

Example 2.3. Given two cubes $A = a^{\{0,1\}}b^{\{0\}}c^{\{0\}}$, $B = a^{\{1,2\}}b^{\{1\}}c^{\{0\}}$. The difference of cubes A and B is 2 because they have different true sets on variables a and b. The distance of cubes A and B is 1 because they have disjoint true sets on variable b.

2.3 Cube Calculus Operations

The cube calculus operations presented in this thesis can be categorized into three groups: *simple combinational operations*, *complex combinational operations* and *sequential operations.*

Each cube operation has one or two operand cube(s). Cubes A and B are used to represent these arguments and they can be described by:

$$A = x_1^{S_1^A} x_2^{S_2^A} \cdots x_i^{S_i^A} \cdots x_n^{S_n^A} \tag{2.3}$$

$$B = x_1^{S_1^B} x_2^{S_2^B} \cdots x_i^{S_i^B} \cdots x_n^{S_n^B} \tag{2.4}$$

where S_i^A and S_i^B are the true sets of literal $x_i^{S_i^A}$ and $x_i^{S_i^B}$, respectively. n is the number of variables. In this chapter, S represents the true set of a literal, the subscript of S represents the index of the variable, the superscript of S represents the operand cube (A or B).

2.3.1 Simple combinational cube operations

Simple combinational cube operations are defined as single set operations. Such a set operation is applied on all pairs of true sets S_i^A and S_i^B of corresponding literals of operand cubes. A simple combinational cube operation produces one resultant cube. The *intersection* and the *supercube* are simple combinational cube operations presented in this section.

Intersection

The intersection of two cubes A and B is the cube that is included in both A and B. The intersection operation of cubes A and B is defined as follows:

$$A \cap B = \begin{cases} x_1^{S_1^A \cap S_1^B} \cdots x_n^{S_n^A \cap S_n^B} & \text{if there is no such } i \text{ that } S_i^A \cap S_i^B = \phi \\ \emptyset & \text{otherwise} \end{cases} \tag{2.5}$$

Where $S_i^A \cap S_i^B$ is a set intersection operation. ϕ denotes an empty set, and $\emptyset$ denotes a contradiction.

Example 2.4. Assuming two cubes $A = ab$ and $B = b\bar{c}$, where a, b and c are binary variables. The intersection of two cubes A and B is the following:

$$A = ab = ab\mathrm{x} = a^{\{1\}}b^{\{1\}}c^{\{0,1\}}$$

$$B = bc = \mathrm{x}b\bar{c} = a^{\{0,1\}}b^{\{1\}}c^{\{0\}}$$

$$A \cap B = a^{\{1\}\cap\{0,1\}}b^{\{1\}\cap\{1\}}c^{\{0,1\}\cap\{0\}} = a^{\{1\}}b^{\{1\}}c^{\{0\}} = ab\bar{c}$$

(a) Operand cubes (b) Resultant cube

Figure 2.1: Intersection operation example

Example 2.4 is illustrated in Figure 2.1 by Karnaugh map. The intersection operation can be used in function decomposition [40, 41].

Supercube

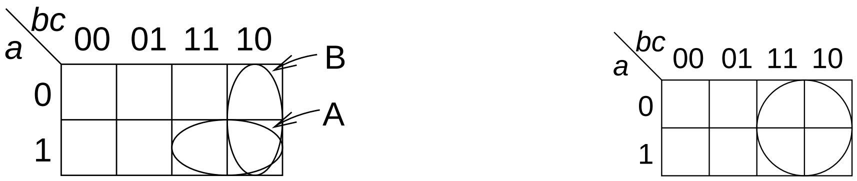
The supercube of two cubes A and B is the smallest cube that includes cubes A and B. The supercube operation of cubes A and B is defined as follows:

$$A \cup B = x_1^{S_1^A \cup S_1^B} \cdots x_n^{S_n^A \cup S_n^B} \tag{2.6}$$

Where $S_i^A \cup S_i^B$ is a set union.

Example 2.5. The supercube on two cubes A and B used in Example 2.4 follows:

$$A \cup B = a^{\{1\}\cup\{0,1\}} b^{\{1\}\cup\{1\}} c^{\{0,1\}\cup\{0\}} = a^{\{0,1\}} b^{\{1\}} c^{\{0,1\}} = b$$



(a) Operand cubes

(b) Resultant cube

Figure 2.2: Supercube operation example

Example 2.5 is illustrated in Figure 2.2 by Karnaugh map. The supercube operation can be used in graph coloring problem [42].

2.3.2 Complex combinational cube operations

Complex combinational cube operations are defined as two set operations and one set relation. These two set operations are called *before* (*bef* for short) and *active* (*act* for short). All variables whose pair of true sets S_i^A and S_i^B satisfy *relation* are said to be *special variables*. The *Active* set operation is applied on all pairs of true sets S_i^A and S_i^B of special variables. The *before* set operation is applied on the others. A complex combinational cube operation produces one resultant cube like simple combinational cube operation. Prime is an example of complex combinational cube operation presented in this section.

Prime

The prime operation of two cubes A and B is defined as:

$$A\,'B = x_1^{S_1^A} \cdots x_{k-1}^{S_{k-1}^A} x_k^{S_k^A \cup S_k^B} x_{k+1}^{S_{k+1}^A} \cdots x_{l-1}^{S_{l-1}^A} x_l^{S_l^A \cup S_l^B} x_{l+1}^{S_{l+1}^A} \cdots x_n^{S_n^A} \tag{2.7}$$

Where the relation for the prime operation is $S_i^A \cap S_i^B \neq \phi$. The *active* set operation is $S_i^A \cup S_i^B$, and the *before* set operation is S_i^A. In the above equation, variables x_k

and x_l are the special variables.

Example 2.6. Assuming two cubes $A = \bar{x}_1 x_2 x_3 x_4$ and $B = x_1 \bar{x}_3$, where x_1, x_2, x_3 and x_4 are binary variables. The prime of $A\,'B$ is defined as follows:

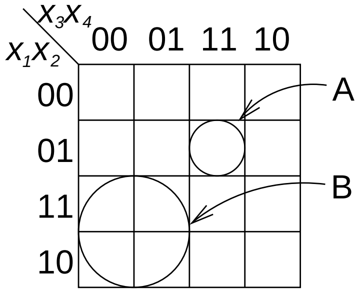
$$A = \bar{x}_1 x_2 x_3 x_4 = x_1^{\{0\}} x_2^{\{1\}} x_3^{\{1\}} x_4^{\{1\}}$$
$$B = x_1 \bar{x}_3 = x_1 \mathrm{x} \bar{x}_3 \mathrm{x} = x_1^{\{1\}} x_2^{\{0,1\}} x_3^{\{0\}} x_4^{\{0,1\}}$$

Because:

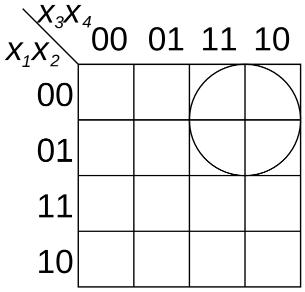
$$S_2^A \cap S_2^B = \{1\} \cap \{0,1\} = \{1\} \neq \emptyset$$
$$S_4^A \cap S_4^B = \{1\} \cap \{0,1\} = \{1\} \neq \emptyset$$

variable x_2 and x_4 are special variables. Therefore,

$$A\,'B = x_1^{S_1^A} x_2^{S_2^A \cup S_2^B} x_3^{S_3^A} x_4^{S_4^A \cup S_4^B} = x_1^{\{0\}} x_2^{\{0,1\}} x_3^{\{1\}} x_4^{\{0,1\}} = \bar{x}_1 x_3$$



(a) Operand cubes



(b) Resultant cube

Figure 2.3: Prime operation example

Example 2.6 is illustrated in Figure 2.3 by Karnaugh map. The prime operation is used in the ESOP minimization program EXORCISM, developed by Dr. Perkowski and his former students [29, 30].

Consensus

The consensus operation on cubes A and B is defined as follows:

$$A \; * \; B = \begin{cases} A \cap B & \text{when } distance(A,B) = 0 \\ \emptyset & \text{when } distance(A,B) > 1 \\ A *_{basic} B & \text{when } distance(A,B) = 1 \end{cases} \tag{2.8}$$

where $A *_{basic} B$ is defined as follows:

$$A *_{basic} B = x_1^{S_1^A \cap S_1^B} \cdots x_{k-1}^{S_{k-1}^A \cap S_{k-1}^B} x_k^{S_k^A \cup S_k^B} x_{k+1}^{S_{k+1}^A \cap S_{k+1}^B} \cdots x_n^{S_n^A \cap S_n^B} \tag{2.9}$$

where $S_k^A \cap S_k^B = \emptyset$. For basic consensus operation, the *before* set operation is $S_i^A \cap S_i^B$, the *active* set operation is $S_i^A \cup S_i^B$, and the *relation* is always true.

Example 2.7. Assuming two cubes $A = x_1 x_2 \bar{x_3}$ and $B = x_1 \bar{x_2}$, where x_1, x_2, x_3 and x_4 are binary variables. Because the distance of cubes A and B is 1, then cubes A and B have consensus as follows:

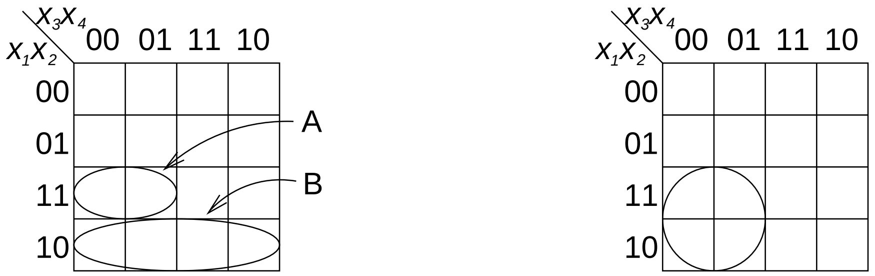
$$A = x_1 x_2 \bar{x_3} = x_1^{\{1\}} x_2^{\{1\}} x_3^{\{0\}} x_4^{\{0,1\}}$$
$$B = x_1 \bar{x_2} = x_1 \bar{x_2} \mathrm{xx} = x_1^{\{1\}} x_2^{\{0\}} x_3^{\{0,1\}} x_4^{\{0,1\}}$$

Because:

$$S_2^A \cap S_2^B = \{1\} \cap \{0\} = \emptyset$$

variable x_2 is a special variable. Therefore,

$$A * B = x_1^{S_1^A \cap S_1^B} x_2^{S_2^A \cup S_2^B} x_3^{S_3^A \cap S_3^B} x_4^{S_4^A \cap S_4^B} = x_1^{\{1\} \cap \{1\}} x_2^{\{0\} \cup \{1\}} x_3^{\{0\} \cap \{0,1\}} x_4^{\{0,1\} \cap \{0,1\}} = x_1 \bar{x_3}$$



(a) Operand cubes (b) Resultant cube

Figure 2.4: Prime operation example

Example 2.7 is illustrated in Figure 2.4 by Karnaugh map. The consensus operation is used for finding prime implicants, and finding prime implicant is a basic step of the well-known Quine-McCluskey algorithm that is used for two-level logic minimization and its variants [23], as well as many other basic algorithms for two-level, three-level and multi-level logic minimization and machine learning [33].

Cofactor

The cofactor operation of two cubes A and B is defined as:

$$A \mid B = \begin{cases} A \mid_{basic} B & \text{when } A \cap B \neq \phi \\ \phi & \text{otherwise} \end{cases} \tag{2.10}$$

where $A \mid_{basic} B$ is defined as follows:

$$A \mid_{basic} B = x_1^{S_1^A \cap S_1^B} \cdots x_{k-1}^{S_{k-1}^A \cap S_{k-1}^B} x_k^U x_{k+1}^{S_{k+1}^A \cap S_{k+1}^B} \cdots x_{l-1}^{S_{l-1}^A \cap S_{l-1}^B} x_l^U x_{l+1}^{S_{l+1}^A \cap S_{l+1}^B} \cdots x_n^{S_n^A \cap S_n^B} \tag{2.11}$$

Where the relation for cofactor operation is $S_i^A \supseteq S_i^B$. The result of the *active* set operation is always U (universal set), and the *before* set operation is $S_i^A \cap S_i^B$. In the above equation, variables x_k and x_l are special variables.

Example 2.8. Assuming two cubes $A = x_1x_2x_3$ and $B = x_1$, where x_1, x_2, x_3 and x_4 are binary variables. The cofactor of $A|B$ is defined as follows:

$$A = x_1x_2x_3 = x_1x_2x_3\mathrm{x} = x_1^{\{1\}}x_2^{\{1\}}x_3^{\{1\}}x_4^{\{0,1\}}$$
$$B = x_1 = x_1\mathrm{xxx} = x_1^{\{1\}}x_2^{\{0,1\}}x_3^{\{0,1\}}x_4^{\{0,1\}}$$

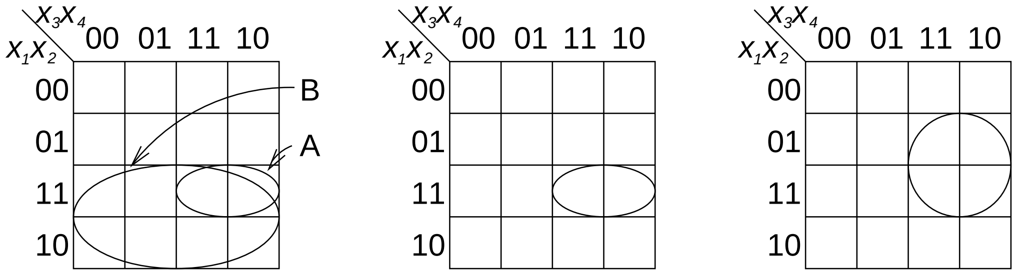
Because:

$$S_1^A = \{1\}\,, S_1^B = \{1\} \;\rightarrow\; S_1^A \supseteq S_1^B$$
$$S_2^A = \{1\}\,, S_2^B = \{0,1\} \;\rightarrow\; S_2^A \not\supseteq S_2^B$$
$$S_3^A = \{1\}\,, S_3^B = \{0,1\} \;\rightarrow\; S_3^A \not\supseteq S_3^B$$
$$S_4^A = \{0,1\}\,, S_4^B = \{0,1\} \;\rightarrow\; S_4^A \supseteq S_4^B$$

variable x_1 and x_4 are special variables. Therefore,

$$A|_B = x_1^U x_2^{S_2^A \cap S_2^B} x_3^{S_3^A \cap S_3^B} x_4^U = x_1^{\{0,1\}} x_2^{\{1\}\cap\{0,1\}} x_3^{\{1\}\cap\{0,1\}} x_4^{\{0,1\}} = x_1^{\{0,1\}} x_2^{\{1\}} x_3^{\{1\}} x_4^{\{0,1\}} = x_2x_3$$

This example is illustrated in Figure 2.5 by a Karnaugh map. In the Karnaugh map, first we calculate the intersection of cubes A and B, the intersection result is shown in Figure 2.5(b), then we remove variable x_1 which means in the result cube, variable x_1 can be either 1 or 0 (*don't care*); the result cube is shown in Figure 2.5(c). If there is no intersection of two operand cubes, then the cofactor is an empty cube. The cofactor operation is an important operation used in function decomposition [40, 41].



(a) Operand cubes (b) Intersection (c) Result cube

Figure 2.5: Cofactor operation example

2.3.3 Sequential cube operations

All sequential cube operations are defined as a single formula that has three set operations and one set relation. These three set operations are called *before* (bef for short), *active* (*act* for short) and *after*(aft for short), respectively. All variables whose pair of true sets S_i^A and S_i^B satisfy *relation* are said to be *special variables.*

The sequential cube operations produces m resultant cubes, where m is the number of special variables for a given operation. The sequential cube operations can be generally described by the following fundamental equation:

$$A(op)B = \Big\{ x_1^{aft(S_1^A,S_1^B)} \cdots x_{i-1}^{aft(S_{i-1}^A,S_{i-1}^B)} x_i^{act(S_i^A,S_i^B)} x_{i+1}^{bef(S_{i+1}^A,S_{i+1}^B)} \cdots x_n^{bef(S_n^A,S_n^B)} \;\Big|\; \text{for such } i = 1,\dots,n, \text{ that } relation(S_i^A, S_i^B) \text{ is true} \Big\} \qquad (2.12)$$

where x_i is a special variable, bef, act and aft are set operations. Every special variable produces a resultant cube according to Equation 2.12. This equation is a general pattern of all cube calculus operations and it was mentioned in the introduction.

Crosslink

The crosslink operation on cubes A and B produces an array of cubes defined as:

$$A \,\square\, B = \Big\{ x_1^{S_1^B} \cdots x_{i-1}^{S_{i-1}^B} x_i^{S_i^A \cup S_i^B} x_{i+1}^{S_{i+1}^A} \cdots x_n^{S_n^A} \;\Big|\; \text{for such } i = 1,\dots,n, \text{ that } S_i^A \cap S_i^B = \emptyset \Big\} \qquad (2.13)$$

For crosslink operation, the *before* set operation is $bef(S_i^A, S_i^B) = S_i^A$, the *active* set operation is $act(S_i^A, S_i^B) = S_i^A \cup S_i^B$, the *after* set operation is $aft(S_i^A, S_i^B) = S_i^B$, and the *relation* is $S_i^A \cap S_i^B = \emptyset$. The crosslink operation can only be applied to two cubes when the two operand cubes are of the same degree, and x (don't care) must be in same position(s).

Example 2.9. Assuming variables x_1, x_2, x_3 and x_4 are binary, two cubes $A = \bar{x}_1\bar{x}_3$ and $B = x_1x_3$, the crosslink operation $A \square B$ follows:

$$A = \bar{x}_1\bar{x}_3 = \bar{x}_1\mathrm{x}\bar{x}_3\mathrm{x} = x_1^{\{0\}}x_2^{\{0,1\}}x_3^{\{0\}}x_4^{\{0,1\}}$$
$$B = x_1x_3 = x_1\mathrm{x}\bar{x}_3\mathrm{x} = x_1^{\{1\}}x_2^{\{0,1\}}x_3^{\{1\}}x_4^{\{0,1\}}$$

Because:

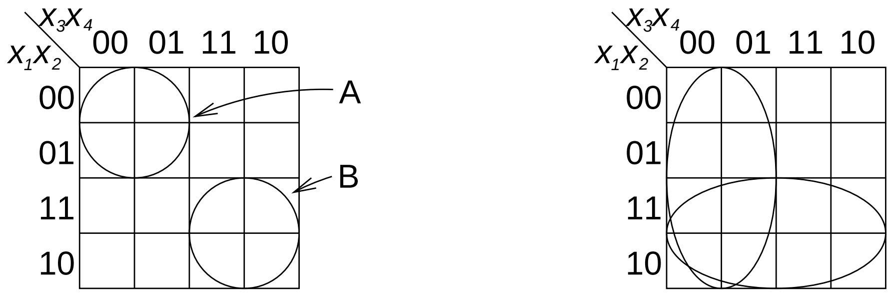
$$S_1^A \cap S_1^B = \{0\} \cap \{1\} = \emptyset$$
$$S_3^A \cap S_3^B = \{0\} \cap \{1\} = \emptyset$$

the variable x_1 and x_3 are special variables. And two resultant cubes are:

$$x_1^{S_1^A \cup S_1^B} x_2^{S_2^A} x_3^{S_3^A} x_4^{S_4^A} = x_1^{\{0\}\cup\{1\}}x_2^{\{0,1\}}x_3^{\{0\}}x_4^{\{0,1\}} = x_1^{\{0,1\}}x_2^{\{0,1\}}x_3^{\{0\}}x_4^{\{0,1\}} = \bar{x}_3$$
$$x_1^{S_1^B} x_2^{S_2^B} x_3^{S_3^A \cup S_3^B} x_4^{S_4^A} = x_1^{\{1\}}x_2^{\{0,1\}}x_3^{\{0\}\cup\{1\}}x_4^{\{0,1\}} = x_1^{\{1\}}x_2^{\{0,1\}}x_3^{\{0,1\}}x_4^{\{0,1\}} = x_1$$

Therefore,

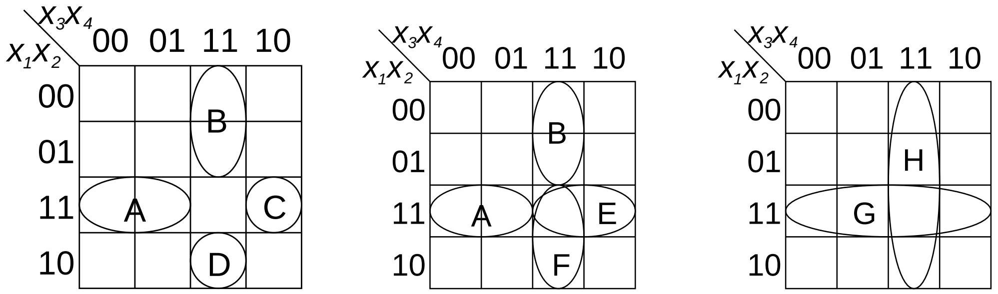
$$A \square B = \bar{x}_1\bar{x}_3 \square x_1x_3 = \bar{x}_3 + x_1$$



(a) Operand cubes (b) Resultant cube

Figure 2.6: Crosslink example

The Example 2.9 is illustrated in Figure 2.6 by Karnaugh map. The crosslink operation can be used in the minimization of logic functions in some canonical forms based on EXOR logic, for instance, the Generalized Reed Muller form [29], as well as the general-purpose AND/EXOR form called ESOPs. The function $f = \bar{x}_1\bar{x}_3 + x_1x_3$



(a) Operand cubes (b) intermediate result (c) Result cubes

Figure 2.7: A complex crosslink example

can be realized using EXOR gates as:

$$f = \bar{x_1}\bar{x_3} + x_1x_3 = \bar{x_3} \oplus x_1$$

Another more complex example is shown in Figure 2.7. As shown in Figure 2.7(a), we have a function $f = A \oplus B \oplus C \oplus D$, where A, B, C and D are four cubes. First, we calculate $C \square D$ and obtain cubes E and F, and the function becomes $f = A \oplus B \oplus E \oplus F$ as shown in Figure 2.7(b). Second we calculate $A \square E$ and $B \square F$, and obtain cubes G and H; therefore, the function is simplified as $f = G \oplus H$.
The crosslink operation is used in ESOP minimization program EXORCISM, developed by Dr. Perkowski and his former student Martin Helliwell in 1988/89 [29, 30]. A more powerful cube operation, called *exorlink*, and a new ESOP minimization program EXORCISM-MV-2 based on *exorlink* operation was developed by Dr. Perkowski and his former student Ning Song in 1993 [16, 32, 43].

Sharp

The (non-disjoint) sharp operation on cubes A and B is defined as follows:

$$A \# B = \begin{cases} A & \text{when } A \cap B = \emptyset \\ \emptyset & \text{when } A \subseteq B \\ A \#_{basic} B & \text{otherwise} \end{cases} \tag{2.14}$$

where $A \#_{basic} B$ is defined as follows:

$$A \#_{basic} B = \Big\{ x_1^{S_1^A} \cdots x_{i-1}^{S_{A,i-1}} x_i^{S_i^A \cap (\neg S_i^B)} x_{i+1}^{S_{A,i+1}} \cdots x_n^{S_n^A} \;\Big|\; \text{for such } i = 1, \ldots, n, \text{ that } \neg(S_i^A \subseteq S_i^B) \Big\} \tag{2.15}$$

For sharp operation, the *before* set operation is S_i^A, the *active* set operation is $S_{A_i} \cap (\neg S_i^B)$, the *after* set operation is S_i^A, and the *relation* is $\neg(S_i^A \subseteq S_i^B)$.

Example 2.10. Assuming variables x_1, x_2, x_3 and x_4 are binary, two cubes $A = \bar{x_3}$ and $B = x_2 x_4$, the sharp operation $A \# B$ follows:

$$A = \bar{x_3} = \mathrm{xx}\bar{x_3}\mathrm{x} = x_1^{\{0,1\}} x_2^{\{0,1\}} x_3^{\{0\}} x_4^{\{0,1\}}$$
$$B = x_2 x_4 = \mathrm{x}x_2\mathrm{x}x_4 = x_1^{\{0,1\}} x_2^{\{1\}} x_3^{\{0,1\}} x_4^{\{1\}}$$

Because:

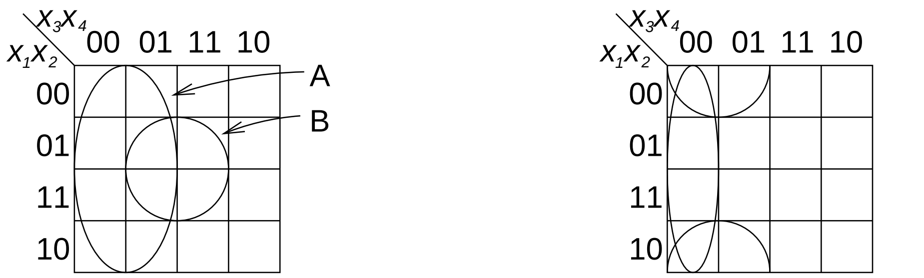
$$\neg(S_2^A \subseteq S_2^B) = \neg(\{0,1\} \subseteq \{1\}) = true$$
$$\neg(S_{A,4} \subseteq S_{B,4}) = \neg(\{0,1\} \subseteq \{1\}) = true$$

variables x_2 and x_4 are special variables. Thus, 2 resultant cubes are:

$$x_1^{S_1^A} x_2^{S_2^A \cap (\neg S_2^B)} x_3^{S_3^A} x_4^{S_4^A} = x_1^{\{0,1\}} x_2^{\{0,1\} \cap (\neg \{1\})} x_3^{\{0\}} x_4^{\{0,1\}} = x_1^{\{0,1\}} x_2^{\{0\}} x_3^{\{0\}} x_4^{\{0,1\}} = \bar{x_2}\bar{x_3}$$
$$x_1^{S_1^A} x_2^{S_2^A} x_3^{S_3^A} x_4^{S_4^A \cap (\neg S_4^B)} = x_1^{\{0,1\}} x_2^{\{0,1\}} x_3^{\{0\}} x_4^{\{0,1\} \cap (\neg \{1\})} = x_1^{\{0,1\}} x_2^{\{0,1\}} x_3^{\{0\}} x_4^{\{0\}} = \bar{x_3}\bar{x_4}$$

Therefore,

$$A \# B = \bar{x_3} \# x_2 x_4 = \bar{x_2}\bar{x_3} + \bar{x_3}\bar{x_4}$$



(a) Operand cubes

(b) Resultant cube $A \# B$

Figure 2.8: Sharp example

Remember, universal set U of possible values of a binary variable is {0,1}, therefore, $\neg\{1\} = \{0\}$. This example is also illustrated in Figure 2.8 by a Karnaugh map. The sharp operation can be used in the tautology problem [33].

Disjoint sharp

The disjoint sharp operation on cubes A and B is defined as follows:

$$A \;\#d\; B = \begin{cases} A & \text{when } A \cap B = \emptyset \\ \emptyset & \text{when } A \subseteq B \\ A \# d_{basic} B & \text{otherwise} \end{cases} \tag{2.16}$$

where $A \;\#d_{basic}\; B$ is defined as follows:

$$A \;\#d_{basic}\; B = \Big\{ x_1^{S_1^A \cap S_1^B} \cdots x_{i-1}^{S_{i-1}^A \cap S_{i-1}^B} x_i^{S_i^A \cap (\neg S_i^B)} x_{i+1}^{S_{i+1}^A} \cdots x_n^{S_n^A} \;\Big|\; \text{for such } i = 1, \ldots, n, \text{ that } \neg(S_i^A \subseteq S_i^B) \Big\} \tag{2.17}$$

For disjoint sharp operation, the *before* set operation is S_i^A, the *active* set operation is $S_{A_i} \cap (\neg S_i^B)$, the *after* set operation is $S_i^A \cap S_i^B$, and the *relation* is $\neg(S_i^A \subseteq S_i^B)$.

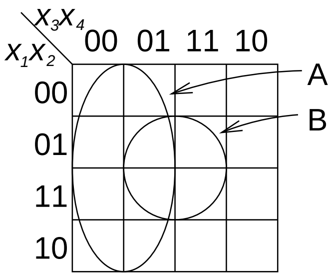
Example 2.11. The disjoint sharp operation $A \;\#d\; B$, where A and B are used in Example 2.5, is calculated as follows:

Since the *relation* of disjoint sharp is the same as sharp, therefore variables x_2 and x_4 are still special variables. Thus, two resultant cubes are:

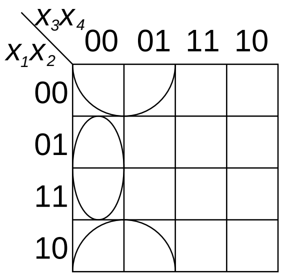
$$\begin{aligned} x_1^{S_1^A \cap S_1^B} x_2^{S_2^A \cap (\neg S_2^B)} x_3^{S_3^A} x_4^{S_4^A} &= x_1^{\{0,1\} \cap \{0,1\}} x_2^{\{0,1\} \cap (\neg \{1\})} x_3^{\{0\}} x_4^{\{0,1\}} \\ &= x_1^{\{0,1\}} x_2^{\{0\}} x_3^{\{0\}} x_4^{\{0,1\}} = \bar{x}_2 \bar{x}_3 \\ x_1^{S_1^A \cap S_1^B} x_2^{S_2^A \cap S_2^B} x_3^{S_3^A \cap S_3^B} x_4^{S_4^A \cap (\neg S_4^B)} &= x_1^{\{0,1\} \cap \{0,1\}} x_2^{\{0,1\} \cap \{1\}} x_3^{\{0\} \cap \{0,1\}} x_4^{\{0,1\} \cap (\neg \{1\})} \\ &= x_1^{\{0,1\}} x_2^{\{1\}} x_3^{\{0\}} x_4^{\{0\}} = x_2 \bar{x}_3 \bar{x}_4 \end{aligned}$$

Therefore,

$$A \;\#d\; B = \bar{x}_3 \;\#d\; x_2 x_4 = \bar{x}_2 \bar{x}_3 + x_2 \bar{x}_3 \bar{x}_4$$



(a) Operand cubes



(b) Resultant cube $A \# d\, B$

Figure 2.9: Disjoint sharp example

The Example 2.11 is also illustrated in Figure 2.9 by a Karnaugh map. The disjoint sharp operation can be used in tautology problem [33] and in conversions between SOP and ESOP representations.

2.3.4 Summary of cube calculus operations

From the above formulas (2.3 to 2.15), it is can be seen that sequential cube operations are the most complex operations in three groups of cube operations. The sequential cube operations are defined by three set operations and one set relation. For the consistency of description, all cube operations in these three groups can be generally described by three set operations and one set relation. For simple combinational cube operations, only one set operation is used (called *before*); For complex combinational cube operations, two set operations and one set relation are used.

All cube operations (some of them are basic operations) described in this chapter are summarized in Table 2.1. Every row describes one cube operation. For each operation, its name, notation, set relation and three set operations (called *output functions* in the table) are listed from left to right, respectively.

Table 2.1: Cube Calculus Operations

Operation	**Notation**	**Relation**	**Output Functions**		
			$before$	$active$	$after$
crosslink	$A \square B$	$S_i^A \cap S_i^B = \emptyset$	S_i^A	$S_i^A \cup S_i^B$	S_i^B
sharp	$A \#_{basic} B$	$\neg(S_i^A \subseteq S_i^B)$	S_i^A	$S_i^A \cap (\neg S_i^B)$	S_i^A
disjoint sharp	$A \#d_{basic} B$	$\neg(S_i^A \subseteq S_i^B)$	S_i^A	$S_i^A \cap (\neg S_i^B)$	$S_i^A \cap S_i^B$
consensus	$A \ *_{basic} \ B$	1	$S_i^A \cap S_i^B$	$S_i^A \cup S_i^B$	$S_i^A \cap S_i^B$
intersection	$A \ \cap \ B$	1	$S_i^A \cap S_i^B$	–	–
supercube	$A \ \cup \ B$	1	$S_i^A \cup S_i^B$	–	–
prime	$A\,'B$	$S_i^A \cap S_i^B \neq \emptyset$	S_i^A	$S_i^A \cup S_i^B$	–
cofactor	$A \,|_{basic} B$	$S_i^A \supseteq S_i^B$	$S_i^A \cap S_i^B$	U	–

2.4 Positional Notation and Cube Operations in Positional Notation

From the above section, it can be seen that all cube operations are broken down into several set relations and set operations, and it is easy to carry out these set relations and set operations by hand. Now, the problem is how to represent sets in some way that they can be processed most efficiently by computers. Our answer to this problem is the *positional notation*.

2.4.1 Positional notation

In *Positional notation*, every possible value of a variable (binary or multi-valued) is represented by one bit, 0 or 1. Thus, a p-valued variable is represented by a string of p-bit; The i-th possible value is represented by the i-th bit. If the literal of this variable is true for a specific possible value (say the i-th possible value), the corresponding bit (the i-th bit) is set to 1, otherwise, it is set to 0.

For example, a four-valued variable x is represented by a string of 4-bit. Literal $x^{\{0,2\}}$ is represented by 1010 because the first and third possible values let the literal be true.

The positional notation for binary literals is shown in table 2.2. The *don't care* means the variable can be either 0 or 1, so both bits are set to 1. The *contradiction* means that the literal is not true for any possible value of variable, so both bits are set to 0. The last two cases, *don't care* and *contradiction*, can be extended to multi-valued variables. For p-valued variable, the string of 1's (the number of 1's is p) presents a *don't care*, and the string of 0's (the number of 0 is p) presents a *contradiction*.

Table 2.2: Positional Notation for binary literals

Binary literals	**Positional Notation**
$\bar{x}$	$\bar{x} = x^0 = x^{10} \rightarrow 10$
x	$x = x^1 = x^{01} \rightarrow 01$
x (don't care)	$\mathrm{x} = x^{0,1} = x^{11} \rightarrow 11$
ϵ (contradiction)	$\epsilon = x^{\emptyset} = x^{00} \rightarrow 00$

2.4.2 Set operations in positional notation

As listed in Table 2.1, all set operations used in cube operations are based on three basic set operations: intersection, union and complement. These three set operations can be executed using bitwise operations in positional notation:

- The set intersection operation can be executed using *bitwise AND* on two strings of bits that represent two true sets of literals in positional notation.

  Example 2.12. Assume two literals $x^{\{0,1,2\}}$ and $x^{\{0,2,3\}}$, where x is a 4-valued variable. Thus two true sets of these two literals are {0,1,2} and {0,2,3}, respectively. The intersection of these two true sets is $\{0,1,2\} \cap \{0,2,3\} = \{0,2\}$. In positional notation, set {0,1,2} is represented by 1110, and set {0,2,3} is represented by 1011. The bitwise AND of 1110 and 1011 is 1010, which means set {0,2}, and this is just what we want. Therefore, the set intersection operation is executed by bitwise AND in positional notation.

- The set union operation can be executed using *bitwise OR* on two string of bits that represent two true sets of literals in positional notation.

  Example 2.13. Assume two literals $x^{\{0,2\}}$ and $x^{\{3\}}$, where x is a 4-valued variable. Thus two true sets of these two literals are {0,2} and {3}, respectively. The union of these two true sets is $\{0,2\} \cup \{3\} = \{0,2,3\}$. In positional notation, set {0,2} is represented by 1010, and set {3} is represented by 0001. The bitwise OR of 1010 and 0001 is 1011, which means set {0,2,3}, and this is just what we want. Therefore, the set union operation is executed by bitwise OR in positional notation.

- The set complement operation can be executed using *bitwise NOT* on the string of bits that represents the true set of literal in positional notation.

  Example 2.14. Assume a literal $x^{\{0,2\}}$, where x is a 4-valued variables. Thus the true sets of the literal is {0,2}. The complement of the true set is $\neg\{0,2\} = \{1,3\}$ (the $U = \{0,1,2,3\}$ for 4-valued variable). In positional notation, set

{0,2} is represented by 1010. The bitwise NOT of 1010 is 0101, which means set {1,3}, and this is just what we want. Therefore, the set complement operation is executed by bitwise NOT in positional notation.

All other set operations can be done by combining these three basic set operations.

Example 2.15. Assume two literals $S^A = x^{\{0,2\}}$ and $S^B = x^{\{2,3\}}$, where x is a 4-valued variables. Thus two true sets of these two literals are $S^A = \{0,2\}$ and $S^B = \{2,3\}$, respectively. The set operation is:

$$S^A \cap (\neg S^B) = 1010\ AND\ (NOT\ 0011) = 1010\ AND\ 1100 = 1000$$

where AND and OR are bitwise operations. The result 1000 represents set $\{0\}$, which is correct result. This kind of set operation is called *set difference*, and is used in sharp and disjoint sharp cube operations.

2.4.3 Set relations in positional notation

The result of set relation is *true* or *false* and can be represented by one bit, 1 presents *true* and 0 presents *false*. The set relation can not be done by bitwise function because it is the function of all bits of two operand sets in positional notation.

Set relation is broken down into two parts in positional notation, *partial relation* and *relation type*. The *partial relation* determines whether or not a pair of the same possible value of two literals satisfy the *relation* "locally". The *relation type* determines the method of combining *partial relations*.

Assuming there are two literals x^A and x^B, where x is p-valued variable x, A is positional notation of true set of literal x^A, and $A = [a_0, a_2, \ldots, a_{p-1}]$, where a_i presents the $(i+1)$-th possible value of the literal (Note: the possible value starts with 0, ends with $p-1$), and $a_i \in \{0,1\}$. Similarly, $B = [b_0, b_1, \ldots, b_{p-1}]$, where $b_i \in \{0,1\}$.

For the crosslink operation, the set relation is $S^A \cap S^B = \phi$. Thus partial relation is $a_i \cdot b_i = 0$, or $\overline{a_i} + \overline{b_i} = 1$ (from De Morgan's theorem). If and only if all pairs of possible values satisfy this partial relation, then the set relation is satisfied. This can be written as:

$$relation(A,B) = (\overline{a_0} + \overline{b_0}) \cdot (\overline{a_1} + \overline{b_1}) \cdots (\overline{a_{p-1}} + \overline{b_{p-1}})$$

Therefore, the *partial relation* is $\overline{a_i}+\overline{b_i}=1$, and the *relation type* of crosslink operation is AND type because AND function is used to combining all *partial relations.*

An example of OR type relation is the one used in the sharp operation, where the *relation* is $\neg(S^A \subseteq S^B)$. Thus partial relation is $\neg(A_i \subseteq B_i)$, where A_i is the subset of the true set S^A. If the set S^A includes the possible value $i-1$, then the set A_i has one element that is the possible value $i-1$ and is represented by $a_i = 1$; otherwise, the set A_i is an empty set and is represented by $a_i = 0$. It can be seen that a_i is the i-th bit of the bit string that represents the set S^A in positional notation. The same thing is with B_i and b_i.

Table 2.3: The partial relation of sharp operation

a_i	b_i	$A_i \subseteq B_i$	$\neg(A_i \subseteq B_i)$	$a_i \cdot \overline{b_i}$
0	0	1	0	0
0	1	1	0	0
1	0	0	1	1
1	1	1	0	0

Table 2.3 shows how to find the *partial relation* function for the sharp operation. The first column shows two bits a_i and b_i. The next two columns show the value of the negated *partial relation* and the *partial relation* itself, the last column shows the bitwise function used to determine the *partial relation.* In the first two rows, $a_i = 0$ means that the set A_i is an empty set, and it is subset of all sets, thus $A_i \subseteq B_i$ are 1's (true). In the third row, $a_i = 1$ means that the set A_i includes one element that is possible value $i-1$, and $b_i = 0$ means that the set B_i is an empty set, thus $A_i \subseteq B_i$ is 0 (false). In the fourth row, $a_i = b_i = 1$ means that the sets A_i and B_i include one same element that is possible value $i-1$, thus sets A_i and B_i are equal, and $A_i \subseteq B_i$ is 1 (true). Therefore, the set relation of sharp operation can be determined by:

$$relation(A,B) = (a_0 \cdot \overline{b_0}) + (a_1 \cdot \overline{b_1}) + \ldots + (a_{p-1} \cdot \overline{b_{p-1}})$$

where the *partial relation* function is $a_i \cdot \overline{b_i}$, and the *relation type* is OR.

2.4.4 Summary of cube operations in positional notation

From the above discussion, *before*, *active*, *after* set operations and *partial (set) relation* can be defined by bitwise functions on bits. Therefore, all cube operations (some of them are basic operations) can be completely specified by 4 bitwise functions and *relation type*. Table 2.4 summarizes all cube operations described in this chapter in bitwise functions and relation type.

Table 2.4: Cube Calculus Operations in bitwise function and relation type

Operation	**Notation**	**Relation**		**Output Function**		
		partial relation	*relation type*	*before*	*active*	*after*
crosslink	$A \square B$	$\bar{a}_i + \bar{b}_i$	and	a_i	$a_i + b_i$	b_i
sharp	$A \#_{basic} B$	$a_i \cdot \bar{b}_i$	or	a_i	$a_i \cdot \bar{b}_i$	a_i
disjoint sharp	$A \# d_{basic} B$	$a_i \cdot \bar{b}_i$	or	a_i	$a_i \cdot \bar{b}_i$	$a_i \cdot b_i$
consensus	$A *_{basic} B$	1	and	$a_i \cdot b_i$	$a_i + b_i$	$a_i \cdot b_i$
intersection	$A \cap B$	–	–	$a_i \cdot b_i$	–	–
super cube	$A \cup B$	–	–	$a_i + b_i$	–	–
prime	$A\,'B$	$a_i \cdot b_i$	or	a_i	$a_i + b_i$	–
cofactor	$A\|_{basic}B$	$a_i + \bar{b}_i$	and	$a_i \cdot b_i$	1	–

The following examples show the entire procedure to carry out cube calculus operations in positional notation.

Example 2.16. Variables x_1, x_2 and x_3 all have 3 possible values, thus the sets of possible values are $P_1 = P_2 = P_3 = \{0,1,2\}$. A cube of $x_1^{\{0,1\}}x_2^{\{2\}}x_3^{\{0,1,2\}}$ is denoted as $x_1^{110}x_2^{010}x_3^{111}$ in positional notation; and it is written as [110-010-111] for simplifying.

Example 2.17. Assuming cubes $A = x_1x_2$ and $B = x_2x_3\bar{x}_4$, where variables x_1, x_2, x_3 and x_4 are binary, the intersection of cubes A and B follows:

$$\begin{aligned}A \cap B =& x_1x_2 \cap x_2x_3\bar{x}_4\\ =& x_1^{\{1\}}x_2^{\{1\}}x_3^{\{0,1\}}x_4^{\{0,1\}} \cap x_1^{\{0,1\}}x_2^{\{1\}}x_3^{\{1\}}x_4^{\{0\}}\\ =& x_1^{[01]}x_2^{[01]}x_3^{[11]}x_4^{[11]} \cap x_1^{[11]}x_2^{[01]}x_3^{[01]}x_4^{[10]}\end{aligned}$$

$$
\begin{aligned}
&= x_1^{[01]\cdot[11]} x_2^{[01]\cdot[01]} x_3^{[11]\cdot[01]} x_4^{[11]\cdot[10]} \\
&= x_1^{[01]} x_2^{[01]} x_3^{[01]} x_4^{[10]} \\
&= x_1 x_2 x_3 \bar{x_4}
\end{aligned}
$$

where '$\cdot$' is bitwise AND operation.

When two opposite literals are multiplied, if the contradiction is generated from the bitwise operation, then there is no resultant cube, which means that there is no intersection between two operand cubes.

Example 2.18. Assuming cubes $A = ab$ and $B = a\bar{b}$, where a and b are binary variables. Then the intersection of cubes A and B is:

$$ab \cdot a\bar{b} = [01 - 01] \cdot [01 - 10] = [01 - \underbrace{00}_{\epsilon}]$$

where 00 is a contradiction symbol for a binary variable, and the contradiction is denoted by ϵ (see Table 2.2).

Example 2.19. Redo the crosslink operation shown in Example 2.8 in positional notation as follows:

$$A = \bar{x_1}\bar{x_3} = x_1^{\{0\}} x_2^{\{0,1\}} x_3^{\{0\}} x_4^{\{0,1\}} = [\overset{a_{1,0}}{1}\,\overset{a_{1,1}}{0} - \overset{a_{2,0}}{1}\,\overset{a_{2,1}}{1} - \overset{a_{3,0}}{1}\,\overset{a_{3,1}}{0} - \overset{a_{4,0}}{1}\,\overset{a_{4,1}}{1}]$$

$$B = x_1 x_3 = x_1^{\{1\}} x_2^{\{0,1\}} x_3^{\{1\}} x_4^{\{0,1\}} = [\overset{b_{1,0}}{0}\,\overset{b_{1,1}}{1} - \overset{b_{2,0}}{1}\,\overset{b_{2,1}}{1} - \overset{b_{3,0}}{0}\,\overset{b_{3,1}}{1} - \overset{b_{4,0}}{1}\,\overset{b_{4,1}}{1}]$$

where the header of bit is the name of the bit. The subscript of bit name has two parts separated by comma (','), the first part represents the index of the variable, and the second part represents the possible value. This notation will also be used in the next chapter. Because:

$$
\begin{aligned}
relation(A_1, B_1) &= (\overline{a_{1,0}} + \overline{b_{1,0}}) \cdot (\overline{a_{1,1}} + \overline{b_{1,1}}) = (\bar{1} + \bar{0}) \cdot (\bar{0} + \bar{1}) = 1 \\
relation(A_2, B_2) &= (\overline{a_{2,0}} + \overline{b_{2,0}}) \cdot (\overline{a_{2,1}} + \overline{b_{2,1}}) = (\bar{1} + \bar{1}) \cdot (\bar{1} + \bar{1}) = 0 \\
relation(A_3, B_3) &= (\overline{a_{3,0}} + \overline{b_{3,0}}) \cdot (\overline{a_{3,1}} + \overline{b_{3,1}}) = (\bar{1} + \bar{0}) \cdot (\bar{0} + \bar{1}) = 1 \\
relation(A_4, B_4) &= (\overline{a_{4,0}} + \overline{b_{4,0}}) \cdot (\overline{a_{4,1}} + \overline{b_{4,1}}) = (\bar{1} + \bar{1}) \cdot (\bar{1} + \bar{1}) = 0
\end{aligned}
$$

the variables x_1 and x_3 are special variables. The two resultant cubes are:

$$
\begin{aligned}
x_1^{A_1+B_1} x_2^{A_2} x_3^{A_3} x_4^{A_4} &= x_1^{[10]+[01]} x_2^{[11]} x_3^{[10]} x_4^{[11]} = x_1^{[11]} x_2^{[11]} x_3^{[10]} x_4^{[11]} = \bar{x_3} \\
x_1^{B_1} x_2^{B_2} x_3^{A_3+B_3} x_4^{A_4} &= x_1^{[01]} x_2^{[11]} x_3^{[10]+[01]} x_4^{[11]} = x_1^{[01]} x_2^{[11]} x_3^{[11]} x_4^{[11]} = x_1
\end{aligned}
$$

where '+' is the bitwise OR operation.

CHAPTER 3

Cube Calculus Machine

The software-based approach is the simplest and cheapest way to realize cube calculus operations. Since the algorithm of a sequential cube calculus operation involves several-level nested loops, it leads to poor performance on general purpose computers. In some applications, such as logic minimization systems, or logic-based machine learning systems, there are so many cube calculus operations, that the software-based approach makes those applications unacceptable for practical sized problems due to poor performance. For speeding up the cube calculus operations, the cube calculus machine was invented.

3.1 Formalism for main algorithm for cube calculus operation

The architecture of the Cube Calculus Machine (CCM) results from an attempt to optimize the execution of the most complex cube operation, sequential cube calculus operations, like crosslink and sharp. Almost all cube calculus operations have two operand cubes described in Formulas 2.3 and 2.4. Re-write these two operand cubes in positional notation as:

$$\begin{aligned} A &= x_1^{A_1} x_2^{A_2} \cdots x_i^{A_i} \cdots x_n^{A_n} \\ B &= x_1^{B_1} x_2^{B_2} \cdots x_i^{B_i} \cdots x_n^{B_n} \end{aligned} \tag{3.1}$$

where $A_i = (a_{i,1}, a_{i,2}, \cdots, a_{i,m}, \cdots, a_{i,p_i})$, which is the true set of literal $x_i^{A_i}$ in position notation. For each bit $a_{i,m}$ from A_i, $a_{i,m} \in \{0,1\}$. The bit $a_{i,m}$ represents the m-th possible value of the i-th variable. Similarly, $B_i = (b_{i,1}, b_{i,2}, \cdots, b_{i,m}, \cdots, b_{i,p_i})$ and for each bit $b_{i,m}$ from B_i, $b_{i,m} \in \{0,1\}$. Example 2.18 shows how to use this notation.

The sequential cube operations are generally described by Equation 2.12. Re-write the equation in positional notation as:

$$A\ (op)\ B = \vec{C} \tag{3.2}$$

where resultant array of cubes $\vec{C}$ is:

$$\vec{C} = \Big\{ X_1^{aft(A_1,B_1)} \cdots X_{i-1}^{aft(A_{i-1},B_{i-1})} X_i^{act(A_i,B_i)} X_{i+1}^{bef(A_{i+1},B_{i+1})} \cdots X_n^{bef(A_n,B_n)} \Big| \text{ for such } i = 1, 2, \ldots, n, \text{ that } relation(A_i, B_i) \text{ is true} \Big\} \tag{3.3}$$

where bef, act and aft are bitwise function used to calculate set operations *before*, *active* and *after*. As discussed in section 2.4, an important property of output functions bef, act and aft is that they are bitwise functions.

As discussed in section 2.4.3, the *relation* function is broken down into two parts: *partial relation* and *relation type*. There are only two possible *relation types*: AND and OR. The *relation* can be described by:

$$relation(A_i, B_i) = \begin{cases} rel(a_{i,1}, b_{i,1}) + rel(a_{i,2}, b_{i,2}) + \cdots + rel(a_{i,p_i}, b_{i,p_i}) & \text{For } OR \text{ relation type} \\ rel(a_{i,1}, b_{i,1}) \cdot rel(a_{i,2}, b_{i,2}) \cdots rel(a_{i,p_i}, b_{i,p_i}) & \text{For } AND \text{ relation type} \end{cases} \tag{3.4}$$

where rel is the *partial relation* and it is bitwise function.

All simple combinational cube operations can be defined as a single set operation (see section 2.3.1), so there is no need to define *before*, *active* and *after* functions. For consistency of description, however, all variables in the case of the combinational cube calculus operations are of *before* type. So the same computational mechanism can be used to calculate combinational cube calculus operations. Therefore, combinational operations can be described by:

$$A\ (op)\ B = X_1^{bef(A_1,B_1)} \cdots X_i^{bef(A_i,B_i)} \cdots X_n^{bef(A_n,B_n)} \tag{3.5}$$

All complex combinational cube operations are defined by two set operations and one set relation (see section 2.3.2). For consistency of description, the complex combinational cube operations can be described by:

$$A\ (op)\ B = X_1^{bef(A_1,B_1)} \cdots X_{k-1}^{bef(A_{k-1},B_{k-1})} X_k^{act(A_k,B_k)} X_{k+1}^{bef(A_{k+1},B_{k+1})} \cdots \cdots X_{l-1}^{bef(A_{l-1},B_{l-1})} X_l^{act(A_l,B_l)} X_{l+1}^{bef(A_{l+1},B_{l+1})} \cdots X_n^{bef(A_n,B_n)} \tag{3.6}$$

where variables with indices k and l are special variables, and are actived at the same time, the values of other variables are calculated according to the $before$ function. The function $after$ is not used.

The variables whose pairs of literals (x^{A_i}, x^{B_i}) satisfy $relation(A_i, B_i)$ are said to be *specific variables*, and their positions are said to be *specific positions*. The variable x_i whose output function is *active* function is said to be an *active variable*. In the case of sequential cube operations, there is only one active variable at a time. In the case of complex combinational cube operations, it is possible to have multiple active variables at the same time.

From the above discussion, we can see that these cube operations can be described by functions $relation$, $before$, *active* and $after$, and have similar formulas.

3.2 The general programmable patterns

The *bitwise* function (of length k) is the k-time repetition of the same two-input, one-output Boolean function on argument vectors. For instance, the bitwise function $\vec{C} = AND(\vec{A}, \vec{B})$ is defined as follows:

$$(c_1, c_2, \ldots, c_k) = (AND(a_1, b_1), AND(a_2, b_2), \ldots, AND(a_k, b_k))$$

where $(a_1, a_2, \ldots, a_k)$, $(b_1, b_2, \ldots, b_k)$ and $(c_1, c_2, \ldots, c_k)$ are the vectors $\vec{A}$, $\vec{B}$ and $\vec{C}$, respectively. In the cube operations, there are four bitwise functions: *before*, *active*, *after* and *rel* (partial relation).

We can use a truth table to specify the output values for a Boolean function in terms of the values of input variables. For a bitwise function, there are $2 \times 2 = 4$ combinations of values of the variables. Therefore, the bitwise function can be completely specified by its four output values. This is shown in Figure 3.1(a).

An arbitrary Boolean function of two binary variables a and b can be realized by a 4-to-1 multiplexer as shown in Figure 3.1(b). The variables a and b are control inputs of the multiplexer, the output values of the function are the data inputs of the multiplexer. The values of a and b will select the valid output values from the data inputs.

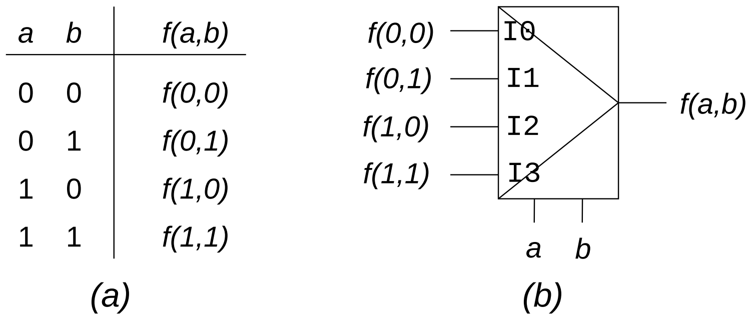


Figure 3.1: Realizing an arbitrary function of two binary variables

This general structure can be programmed to realize an arbitrary bitwise function. For example, the output values of function $f(a, b) = a + b$ are 0111. Set $I_0 I_1 I_2 I_3 =$ 0111, the output of multiplexer is the function of $f(a, b) = a + b$.

We derived all output values of the bitwise functions of the cube operations listed in Table 2.4, the results are listed in Table 3.1. This calculation is very simple. For example, suppose the function is $f(a_i, b_i) = a_i \cdot \bar{b}_i$, then

$$f(0,0) = 0 \cdot \bar{0} = 0$$

$$f(0,1) = 0 \cdot \bar{1} = 0$$

$$f(1,0) = 1 \cdot \bar{0} = 1$$

$$f(1,1) = 1 \cdot \bar{1} = 0$$

Therefore, the output values of function $f(a_i, b_i) = a_i \cdot \bar{b}_i$ are 0010.

Each row of Table 3.1 describes one cube operation. The operation name, notation, the output value of *rel* (partial relation) function, *and_or* (relation type), the output values of *before*, *active* and *after* functions are listed from left to right. The value of and_or equals to 1 means that the relation type is of AND type; otherwise, the relation type is of OR type.

3.3 The data path of CCM

Since the data path of CCM results from an attempt to optimize the execution of cube calculus operations, especially the sequential cube calculus operations, the CCM has been designed to directly implement Formulas 3.2 and 3.3.

Table 3.1: The Output Values of Bitwise Functions Used in Cube Operations

Operation	Notation	Relation		Output Function		
		rel	*and/or*	*before*	*active*	*after*
crosslink	$A \square B$	1110	1	0011	0111	0101
sharp	$A \#_{basic} B$	0010	0	0011	0010	0011
disjoint sharp	$A \#d_{basic} B$	0010	0	0011	0010	0001
consensus	$A *_{basic} B$	1111	1	0001	0111	0001
intersection	$A \cap B$	–	–	0001	–	–
super cube	$A \cup B$	–	–	0111	–	–
prime	$A\,'\,B$	0001	0	0011	0111	–
cofactor	$A\,\vert_{basic}\, B$	1011	1	0001	1111	–

From Formula 3.3, it can be seen that one relation function and one of three output functions (*bef*, *act* and *aft*) are applied on every pair of literals ($x_i^{A_i}$, $x_i^{B_i}$) in the same manner, which means we can use one combinational logic block to process every pair of literals in series, or use n identical logic blocks to process all pairs of literals in parallel (one pair of literals per logic block). The later is the so-called *iterative network*, and it has better performance than the former because of its parallelism. The CCM is realized using an iterative network. Before we describe the architecture of the CCM, let us take a look at the general concept of iterative networks.

3.3.1 Iterative network

An iterative network consists of a number of identical cells interconnected in a regular manner. Some operations, like binary addition, are naturally realized with an iterative network because the same operation is performed on each pair of input bits.

The simplest form of iterative network consists of a linear array of combinational cells with signals between cells traveling in only one direction.

As shown in Figure 3.2, each cell is a combinational network with one or more primary input(s) ($x[i]$) and possibly one or more primary output(s) ($z[i]$). In addition, each cell has one or more secondary input(s) ($a[i]$) and one or more secondary output(s) ($a[i+1]$). The $a[i]$ leads carry information about the "state" of the previous

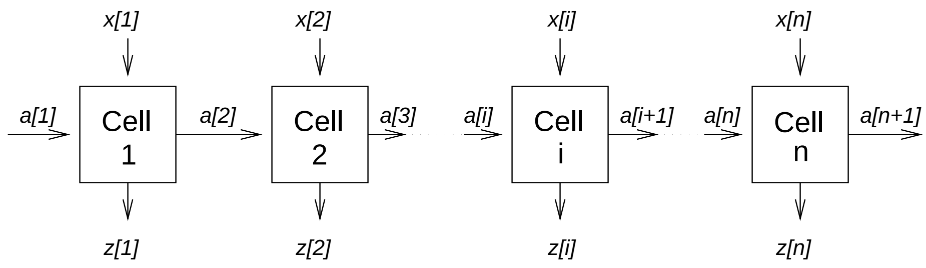


Figure 3.2: A Simple Iterative Network

cell. The primary inputs to the cells $(x[1], x[2], \ldots, x[n])$ are applied in parallel, that is, they are applied at the same time. The $a[i]$ signals then propagate down the line of cells. Since the network is combinational, the time required for the network to reach a stable state condition is determined only by the delay times of the gates in the cells. As soon as stable state is reached, the output may be read.

Therefore, the iterative network can function as a parallel input, parallel-output device, in contrast with the sequential network in which the input and output are provided in a serial manner.

Example 3.1. The *Parity Checker* determines whether the number or 1's in a n-bit word is even or odd. The Figure 3.3 shows the complete iterative network for $n = 4$. The output of it will be 1 if an odd number of x inputs are 1. The logic of a cell can be described by

$$a[i+1] = a[i] \oplus x[i]$$

Assuming the delay of a cell (an EXOR gate in this example) is t_{cell}. The $a[1]$ input to the first cell must be 0 since no ones are received to the left of the first cell and 0 is an even number. The delay of the output of the last cell $a[5]$ is $4t_{cell}$ in this example.

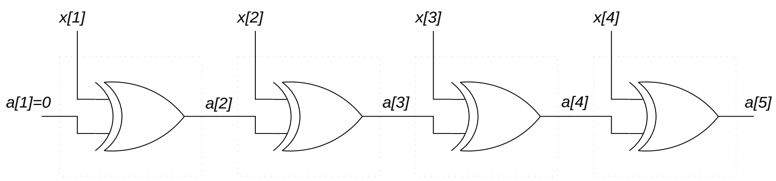


Figure 3.3: Parity Checker

Example 3.2. The *Ripple-carry binary adder* is used to perform addition on two binary numbers. A 4 bits adder can be constructed by cascading 4 full-adder circuit in series as shown in Figure 3.4. The logic of a full-adder can be described by

$$c[i+1] = g[i] + p[i]c[i]$$
$$s[i] = p[i] \oplus c[i]$$

where $c[i]$, $c[i+1]$ and s are carry input, carry output and sum output of cell i, respectively. g (generate) and p (propagate) are two intermediate signals and can be described by

$$g[i] = a[i] \cdot b[i]$$
$$p[i] = a[i] \oplus b[i]$$

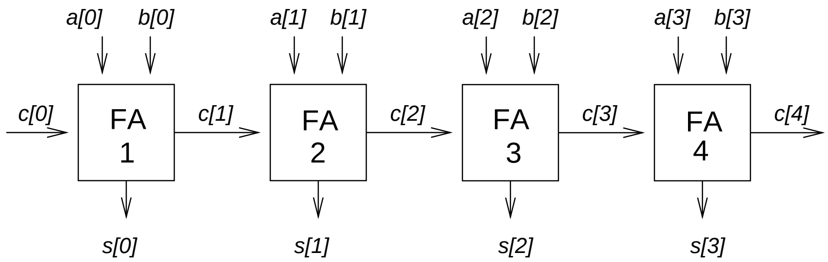


Figure 3.4: Ripple-Carry Binary Adder

Assuming the delay from inputs ($a[i]$ and $b[i]$) to intermediate signals (g and p) is t_1, and from intermediate signals and carry input ($c[i]$) to the outputs ($c[i+1]$ and $s[i]$) is t_2, and the input $c[0]$ is 0 (constant). It can be seen that the carry signal is propagated from left to right along the carry chain, and the carry chain is the worst-case delay path of the adder. Thus, the delay of this adder is $t_1 + 4t_2$.

For the design cases where an iterative network can be used, it offers several advantages over an ordinary combinational network:

- It is easier to design
- It is easily expanded to accommodate more inputs simply by adding more cells.

The principal disadvantage of the iterative network is that the signal must be propagated through a large number of cells, so the response time will be longer than in a corresponding combinational network with few levels.

3.3.2 The algorithm of the CCM

As mentioned above, the CCM is realized with an iterative network. The cell of this iterative network is more complex than those in the two examples given in section 3.3.1. The cell of the iterative network consists of a sequential logic block and several combinational logic blocks as shown in Figure 3.5. The inputs $A[i]$ and $B[i]$ are the pair of input literals. $C[i]$ is the output literal. *Relation*, bef, *act* and aft are binary bits used to specify these function. *clk* and $reset$ are the clock and reset inputs of the D flip-flops used in sequential logic block. The signal *next* ($next[i]$ and $next[i+1]$ in the figure) is the propagation signal (it will be discussed later). The signal $var[i]$ is generated by combinational logic block A, and represents whether the variable (x_i) is a special variable ($var[i] = 1$) or not ($var[i] = 0$).

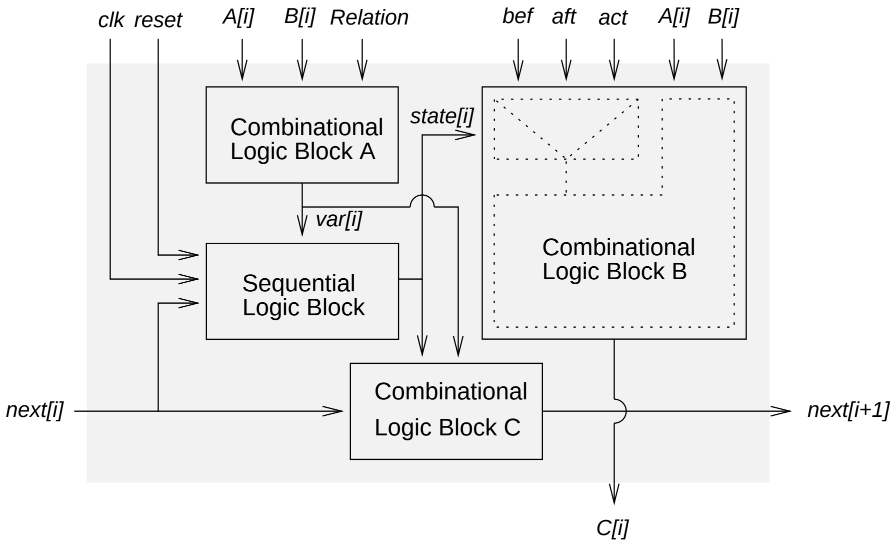


Figure 3.5: An simplified iterative cell of the CCM

For a sequential cube operation (section 2.2.3) we know that every special variable produces one resultant cube. For a given special variable (say x_i), the resultant cube is generated in the following manner: the output literal C_i is calculated by $act(A_i, B_i)$. The output literal C_k $(1 \leq k < i)$ is calculated by $aft(A_k, B_k)$. The output literal C_l $(i < l \leq n)$ is calculated by $bef(A_l, B_l)$. An output literal is calculated by combinational logic block B in Figure 3.5, and all output literals can be calculated in parallel with an iterative network.

A variable is to be called the *active variable* when it is in active state. Now we need to find a way to activate special variables in series, which means only one variable is in active state at a time, and all special variables become active one after another. All other variables should know their relative position with respect to the current active variable, left or right. Our solution to this problem is as follows:

- The propagation signal *next* activates the first special variable that it encounters. The signal *next* is propagated through combinational logic block C in Figure 3.5. The logic of the combinational logic block C can be described by:

  $$next[i+1] = act[i] + next[i] \cdot \overline{var[i]} \tag{3.7}$$

  where signal $act[i]$ is 1 when current state of the FSM is active (current state is represented by the signal $state[i]$).

- Every variable (cell) has a simple Finite State Machine (FSM) (the sequential logic block in the Figure 3.5) to memorize its relative position with respect to the active variable. This FSM has three states: *after*, *active* and *before*, which corresponds to the variables being on the left side of the active variable, the active variable itself, and the variables being on the right side of the active variable, respectively.

  The state machine flowchart[22] (or SM chart for short) of the FSM is shown in Figure 3.6. The states *bef*, *act* and *aft* are the short name of states *before*, *active* and *after*, respectively. The numbers under the state names are state assignments (encoding vectors). This state machine is realized by using D flip-flops with asynchronous reset, thus the reset inputs of the D flip-flops can be used to reset the FSM to *before* state. This reset logic is not shown in the SM chart. The output of the FSM is $state[i]$ signal which represents the state of the FSM (the index indicates the index number of the cell).

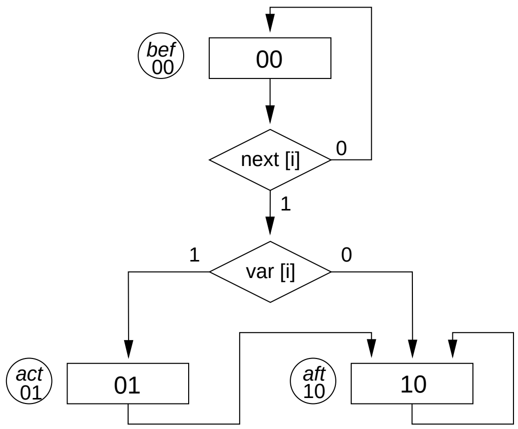


Figure 3.6: The SM chart of the FSM

The signal $state[i]$ is used as the select inputs of the multiplexer to select corresponding output function for combinational logic block B (see Figure 3.5).

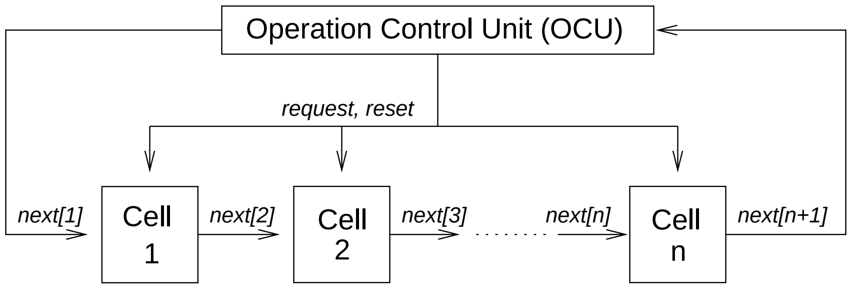


Figure 3.7: The iterative network of the CCM

The whole iterative network is shown in Figure 3.7. The inputs $A[i]$, $B[i]$, $Relation$, bef, act, aft come from register file which is not shown in the figure, and the output $C[i]$ is not shown either. The signal *request* is connected to the *clk* input of the iterative cell. The signals *request* and *reset* are generated by Operation Control Unit (OCU). The following example describes the procedure of executing a sequential cube operation by the core of the CCM, the sequential iterative network (also called *one-dimensional cellular automaton*).

Example 3.3. An example of a sequential cube operation is illustrated in Figure 3.8. In this example, there are two operand cubes that have 4 variables: x_1, x_2, x_3 and

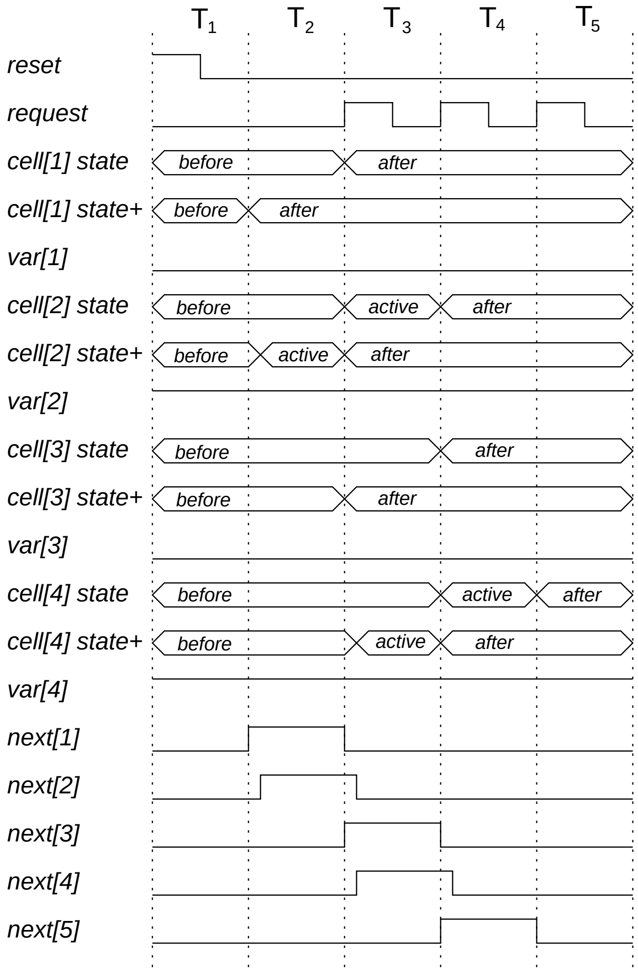


Figure 3.8: Timing diagram of Example 3.3

x_4, and variables x_2 and x_4 are special variables.

Assume we have a CCM that has 4 iterative cells. In Figure 3.8, "cell[i] state" is the *state* signal of the cell i; "cell[i] state+" is the next state signal of the cell i. "var[i]" is the *var* signal of cell i, where $i = 1, 2, 3, 4$. With 4 iterative cells, the CCM has 5 propagation signals *next*, *next*[1] to *next*[5]. This sequential cube operation takes 5 periods. Now let us take a close look at how the CCM works.

1. In period T_1:

   The inputs *relation*, *bef*, *act*, *aft* and the operand cubes are applied in parallel and keep stable during the whole cube operation.

After the operand cubes are applied, the combinational logic block A in each cell begins to evaluate signal $var[i]$. Assuming that the worst-case delay of logic block A is t_A, after the delay of t_A, all signals $var[i]$ $(i = 1, 2, 3, 4)$ become stable and will keep stable if and only if the inputs of operand cubes and the function *relation* keep stable. In this example, $var[1] = var[3] = 0$, and $var[2] = var[4] = 1$. The final states of *var* signals are shown in Figure 3.8; the delay of t_A is not shown in the figure.

All FSMs are reset to *before* state by setting signal *reset* to 1 (see Figure 3.8).

The signal $next[1]$ is set to 0. The signals $next[2]$ to $next[5]$ became 0's (see Equation 3.7). Since the states of FSMs are reset to the *before* state and all *next* signals are 0's, the next states of all FSMs become the *before* state (see Figure 3.6).

2. In period T_2:

   The signal $next[1]$ is set to 1.

   <u>For the cell 1</u>, substitution $act[1] = 0$, $next[1] = 1$ and $var[1] = 0$ into Equation (3.7) gives: $next[2] = act[1] + next[1] \cdot \overline{var[1]} = 0 + 1 \cdot \overline{0} = 1.$

   There is a delay of value 1 propagating from $nex[1]$ to $next[2]$, and the delay is shown in Figure 3.8. This delay comes from combinational logic that is described by Equation (3.7).

   The next state of the cell 1 is *after* (see Figure 3.6).

   <u>For the cell 2</u>, substitution $act[2] = 0$, $next[2] = 1$ and $var[2] = 1$ into Equation (3.7) gives: $next[3] = act[2] + next[2] \cdot \overline{var[2]} = 0 + 1 \cdot \overline{1} = 0.$

   The next state of the cell 2 is *active* (see Figure 3.6).

   <u>For the cells 3 and 4</u>, since there is no change on signal *next* ($next[3]$ and $next[4]$ keep 0), the next state of cells 3 and 4 are *before*.

3. In period T_3:

   There is a rising edge of signal *request*, then all cells change to the next state determined at previous period T_2. As shown in Figure 3.8, the cell 1 goes to *after* state, the cell 2 goes to *active* state, and the cell 3 and 4 keep *before* state.

   At this time, 4 $state[i]$ $(i = 1, 2, 3, 4)$ signals are used to select the corresponding output function. After that, the combinational logic block B in all cells begin to evaluate output literals $C[i]$ in parallel. Assume the clock-to-Q delay of the D flip-flops is t_{FF}, and the worse-case delay of the combinational logic blocks B is t_B. Thus, after the delay of $t_{FF} + t_B$, all output literals $C[i]$ become stable, which means the first resultant cube is generated and can be read.

   For the cell 1, the current state is *after*. From the SM chart of the FSM we know that the cell 1 will keep *after* state during the left time of the cube operation.

   For the cell 2, the current state is *active*, then $act[2] = 1$. Using Equation (3.7) gives: $next[3] = act[2] + next[2] \cdot \overline{var} = 1 + 0 \cdot \overline{1} = 1.$

   The next state of the cell 2 is *after* state (see Figure 3.6).

   For the cell 3, because the signal $next[3] = 1$ and $var[3] = 0$, using Equation (3.7) gives: $next[4] = act[3] + next[3] \cdot \overline{var} = 0 + 1 \cdot \overline{0} = 1.$

   Because $next[3] = 1$ and $var[3] = 0$, the next state of the cell 3 is *after* (see Figure 3.6).

   For the cell 4, because $next[4] = 1$ and $var[4] = 1$, using Equation (3.7) gives: $next[5] = act[4] + next[4] \cdot \overline{var} = 0 + 1 \cdot \overline{1} = 0.$

   The next state of the cell 4 is *active* because $next[3] = 1$ and $var[3] = 1$ (see Figure 3.6).

4. In period T_4:

   There is a rising edge of signal *request*, then all cells change to the next state determined at previous period T_3. The cell 1 keeps *after* state, the cells 2 and 3 goes to *after* state, and the cell 4 goes to *active* state.

At this time, 4 $state[i]$ $(i = 1, 2, 3, 4)$ signals are used to select the corresponding output function, and the second resultant cube is generated and can be read.

Because the cell 4 goes to *active* state, the signal $next[5]$ becomes 1 (Equation 3.7).

5. In period T_5:

   Since the value 1 reaches the last point of the propagation signal $next$ ($next[5]$ in this example), the cube operation is completed.

3.3.3 The signal *ready*

As mentioned in section 3.3.1, the disadvantage of the iterative network is that the propagation signal must propagate through a large number of cells, so the response time will be longer. The delay of propagational signal $next$ reaching the first special variable it encountered is:

$$T_{propagation} = t_{FF} + k \cdot t_C \tag{3.8}$$

where t_C is the worse-case delay of the combinational logic block C, and k is the number of cells the propagation signal going through. It can be seen that the larger the k, the longer the propagation delay.

When k is increased, the delay becomes longer. For the CCM working properly, we have two choices: one is to slow the clock signal which would slow down the entire CCM. The other is using a *ready* signal to tell the CU whether the ILU is ready or not, the CU generate *request* signal only when the ILU is ready. The *ready* signal is as follows:

$$subready[i] = \overline{request} \cdot next[i] \cdot var[i] \tag{3.9}$$

$$ready = subready[1] + subready[2] + \cdots + subready[n] \tag{3.10}$$

The $subready[i]$ signal is generated at the cell that represents a special variable ($var = 1$) and receives the $next$ signal. Any of $subready[i]$ signals becoming 1 means that the CCM is ready to output the result cube. Since we don't want to slow down the entire CCM, the second solution is used.

3.4 Iterative Logic Unit and Iterative Cell

The iterative network and the iterative cell described in section 3.3 are called Iterative Logic Unit (ILU) and ITerative cell (IT), respectively. The IT enumerated from left to right: IT[1], IT[2], ... , IT[n]. The number of ITs is denoted by n.

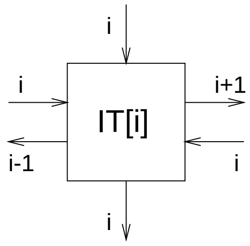


Figure 3.9: The rule of naming signals

Logic signals within each IT[i] have the index of i (as a subscript). Horizontal signals running from left IT (say IT[i]) to its right neighbor IT[i+1] have the index of $i+1$; Horizontal signals running from right IT (say IT[i]) to its left neighbor IT[i-1] have the index of $i-1$; Vertical signals both coming into or leaving from IT[i] have index of i. These principles of naming signals are shown in Figure 3.9.

3.4.1 Handling multi-valued variables

In the previous section, we just stated that one cell processes a single variable. Since one cell processing one variable with arbitrary number of possible values would be not practical, so in our design, one cell (IT) can process one binary variable. However, in addition, for processing variables with more than two possible values (multi-valued variables), multiple ITs are combined together to process a multi-valued variable.

Because the CCM is a hardware, when it has been realized, it has a fixed number of iterative cells. When we use the CCM to solve a problem, we can not always use all its iterative cells, sometimes, we just use part of it. Therefore, we need a signal vector to tell whether a given iterative cell is used by an operation or not. This signal vector is called $water[i]$ where $i = 1, 2, \ldots, n$ ($w[i]$ for short). The signal $w[i] = 1$ means that IT[i] is not used, and it should be transparent to all signals running horizontally (Signals running horizontally are the signals running between cells, like the signal *next*).

Example 3.4. Assume a CCM with 4 iterative cells. For a given cube operation, only two iterative cells are needed. The corresponding signal *water* should be $\overbrace{0}^{w[1]}\ \overbrace{0}^{w[2]}\ \overbrace{1}^{[w[3]}\ \overbrace{1}^{w[4]}$.

We use multiple iterative cells to handle a multi-valued variable. We need a signal vector to tell where is the boundary of a multi-valued variable. This signal vector is $right\_edge[i]$ ($re[i]$ for short), where $i = 1, 2, \ldots, n$. The signal $re[i] = 1$ means that IT[i] is the right edge of a variable. If all variables are binary, then all bits of $right\_edge$ are 1's. Since one iterative cell processes two possible values of a variable, we can process a multi-valued variable with an even number of possible values.

Example 3.5. Assume a CCM with 6 iterative cells. For a given cube operation, there are 3 variables with 2, 4 and 6 possible values, respectively. Therefore the signal $right\_edge$ is $\underbrace{\overbrace{1}^{re[1]}}_{variable1}\ \underbrace{\overbrace{0}^{re[2]}\ \overbrace{1}^{[re[3]}}_{variable2}\ \underbrace{\overbrace{0}^{re[4]}\ \overbrace{0}^{re[5]}\ \overbrace{1}^{[re[6]}}_{variable3}$. Since all iterative cells are used, so the signal *water* is 000000.

Take $w[i]$ and $re[i]$ into account, for handing multi-valued variables, the *next* signal is described now as:

$$next[i+1] = \overline{w[i]} \cdot \left(act[i] + next[i] \cdot \overline{re[i] \cdot var[i]}\right) + w[i] \cdot next[i] \tag{3.11}$$

which means that when the IT[i] is not used ($w[i] = 1$), the iterative cell is transparent to the signal *next*. Otherwise, the *next* signal will propagate till the right edge of the first special variable that it will encounter.

Two more propagation signals *carry* and *conf* are used to combine multiple iterative cells to process multi-valued variables. Let us discuss a simplified example that shows how to use these two signals first, then the general equations for these two signals will be derived after the example.

Example 3.6. Three iterative cells are combined together to process a pair of operand literals of a 6-valued variable as shown in Figure 3.10. The *water* and $right\_edge$

signals are also shown in the figure.

Now the problem is that no iterative cell receives all bits of operand cubes, then no single iterative cell can determine signal var of the variable by itself (the signal var represents whether the variable is a special variable or not). For a given OR_type cube operation, the signal var should be (Equation 3.5):

$$var = rel(a_1, b_1) + rel(a_2, b_2) + rel(a_3, b_3) + rel(a_4, b_4) + rel(a_5, b_5) + rel(a_6, b_6) \tag{3.12}$$

Since a single iterative cell processes just two possible values, then we can let them be:

$$carry[2] = rel(a_1, b_1) + rel(a_2, b_2) \tag{3.13}$$

$$carry[3] = carry[2] + rel(a_3, b_3) + rel(a_4, b_4) \tag{3.14}$$

$$carry[4] = carry[3] + rel(a_5, b_5) + rel(a_6, b_6) \tag{3.15}$$

Substituting Equations (3.13), (3.14) into (3.15) gives:

$$carry[4] = rel(a_1, b_1) + rel(a_2, b_2) + rel(a_3, b_3) + rel(a_4, b_4) + rel(a_5, b_5) + rel(a_6, b_6) \tag{3.16}$$

Comparing Equations (3.12) and (3.16), we know that the signal var is generated, and $var = carry[4]$. Please note that the signal var is always finally generated at the last cell of a variable.

All three cells that process the variable should know the signal var. For the last cell of the variable (IT[3] in this example), $var[3] = carry[4]$. All other cells that process the same variable receive the signal var through the propagation signal $conf$ from its successive cell. In other words, the signal var is propagated back to the preceding cells through the iterative signal $conf$. This can be described by:

$$conf[2] = carry[4] \quad var[2] = conf[2] \quad conf[1] = var[2] \quad var[1] = conf[1] \tag{3.17}$$

It can be seen that the signal $carry$ propagates from left to right until the right edge of the variable in order to generate signal var of the variable. Then signal var is propagated back (from right to left) through signal $conf$ (Equation 3.17). This propagation path is shown in Figure 3.10 by shadow big arrow.

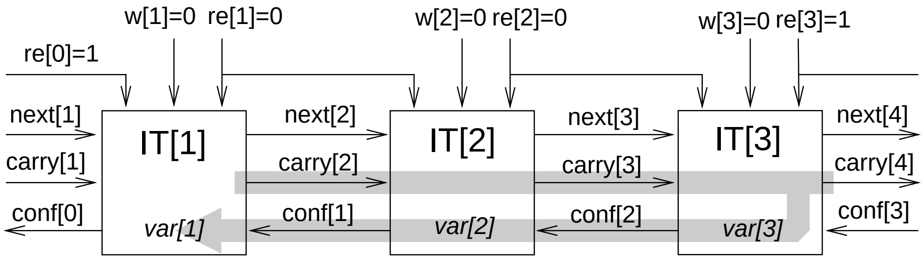


Figure 3.10: Three iterative cells combined together to process a 6-valued variable

For the AND_type cube operation, we only need to change "+" to "·" in the Equation (3.12) to (3.16).

Now we are ready to derive general formula for signals $carry[i]$, $conf[i]$ and $var[i]$. The IT[i] processes two possible values of a variable, then there are two partial relations in one iterative cell:

$$rel0[i] = rel(a0[i], b0[i]) \tag{3.18}$$

$$rel1[i] = rel(a1[i], b1[i]) \tag{3.19}$$

where $a0[i]$ and $a1[i]$ are two input bits from operand literal A, $b0[i]$ and $b1[i]$ are two input bits from operand literal B. For AND type relation, signal $carry\_and$ (signal $carry$ for AND type relation) can be described as:

$$carry\_and[i+1] = \begin{cases} rel0[i] \cdot rel1[i] & \text{if IT[i] is the first IT of a variable} \\ rel0[i] \cdot rel1[i] \cdot carry[i] & \text{otherwise} \end{cases} \tag{3.20}$$

For OR type relation, signal $carry\_or$ (signal $carry$ for OR type relation) can be described as:

$$carry\_or[i+1] = \begin{cases} rel0[i] + rel1[i] & \text{if IT[i] is the first IT of a variable} \\ rel0[i] + rel1[i] + carry[i] & \text{otherwise} \end{cases} \tag{3.21}$$

Signal $right\_edge$ can be used to determine whether or not a given IT[i] is the first/last IT of a variable:

$$re[i] = 1 \qquad \text{if IT[i] is the last IT of a variable} \tag{3.22}$$

$$re[i-1] = 1 \qquad \text{if IT[i] is the first IT of a variable} \tag{3.23}$$

Because IT[1] is always the first IT of a variable, then

$$re[0] \equiv 1 \tag{3.24}$$

Combining Equations (3.20) and (3.23), we obtain

$$\begin{aligned} carry\_and[i+1] &= rel0[i] \cdot rel1[i] \cdot re[i-1] \\ &\quad + carry[i] \cdot rel0[i] \cdot rel1[i] \cdot \overline{re[i-1]} \\ &= rel0[i] \cdot rel1[i] \cdot (re[i-1] + carry[i] \cdot \overline{re[i-1]}) \\ &= rel0[i] \cdot rel1[i] \cdot (re[i-1] + carry[i]) \\ &= rel0[i] \cdot rel1[i] \cdot re[i-1] + rel0[i] \cdot rel1[i] \cdot carry[i] \end{aligned} \tag{3.25}$$

Combining Equation (3.21) and (3.23), we obtain

$$\begin{aligned} carry\_or[i+1] &= (rel0[i] + rel1[i]) \cdot re[i-1] \\ &\quad + (carry[i] + rel0[i] + rel1[i]) \cdot \overline{re[i-1]} \\ &= (rel0[i] + rel1[i]) \cdot re[i-1] + carry[i] \cdot \overline{re[i-1]} \\ &\quad + (rel0[i] + rel1[i]) \cdot \overline{re[i-1]} \\ &= rel0[i] + rel1[i] + carry[i] \cdot \overline{re[i-1]} \end{aligned} \tag{3.26}$$

The signal $carry[i+1]$ is obtained by combining $carry\_and$ and $carry\_or$ as:

$$\begin{aligned} carry[i+1] &= carry\_and[i+1] \cdot and\_or \\ &\quad + carry\_or[i+1] \cdot \overline{and\_or} \end{aligned} \tag{3.27}$$

where signal and_or represents the relation type of the cube operation. $and\_or = 1$ means that the cube operation is of AND type, otherwise, the cube operation is of OR type. Because $carry\_or$ always equals 1 whenever $carry\_and$ equals 1, by combining Equation (3.25) and (3.26), we obtain:

$$\begin{aligned} carry[i+1] &= carry\_and[i+1] \cdot and\_or \\ &\quad + carry\_or[i+1] \cdot \overline{and\_or} \\ &= carry\_and[i+1] + carry\_or[i+1] \cdot \overline{and\_or} \\ &= rel0[i] \cdot rel1[i] \cdot re[i-1] + rel0[i] \cdot rel1[i] \cdot carry[i] \\ &\quad + (rel0[i] + rel1[i]) \cdot \overline{and\_or} + carry[i] \cdot \overline{re[i-1]} \cdot \overline{and\_or} \end{aligned} \tag{3.28}$$

As shown in Example 3.6, signal $conf$ can be generally described as:

$$conf[i-1] = \begin{cases} carry[i+1] & \text{if IT[i] is the last IT of a variable} \\ conf[i] & \text{otherwise} \end{cases} \tag{3.29}$$

Combining Equation (3.29) and (3.22), we obtain:

$$conf[i-1] = conf[i] \cdot \overline{re[i]} + carry[i+1] \cdot re[i] \tag{3.30}$$

The signal var always comes from signal $conf$, which is:

$$var[i] = conf[i-1] \tag{3.31}$$

If we also take the signal $water$ into account, we obtain:

$$\begin{aligned} carry[i+1] = \overline{w[i]} \cdot \Big(& rel0[i] \cdot rel1[i] \cdot re[i-1] \\ & + rel0[i] \cdot rel1[i] \cdot carry[i] + \big(rel0[i] + rel1[i]\big) \cdot \overline{and\_or} \\ & + carry[i] \cdot \overline{re[i-1]} \cdot \overline{and\_or}\Big) + w[i] \cdot carry[i] \end{aligned} \tag{3.32}$$

$$\begin{aligned} conf[i-1] = & \overline{w[i]} \cdot (conf[i] \cdot \overline{re[i]} + carry[i+1] \cdot re[i]) \\ & + w[i] \cdot conf[i] \end{aligned} \tag{3.33}$$

3.4.2 The design of an iterative cell

Now we know how iterative network works and how to combine multiple iterative cells to handle a multi-valued variable. This section will describe the details of the iterative cell that have not been discussed so far. The block diagram of one iterative cell is shown in Figure 3.11. As shown in the figure, one iterative cell can be divided into five blocks according to the function that they perform: IDENTIFY, STATE, OPERATION, COUNTER and EMPTY. All signals except the signals of COUNTER block in the figure were already discussed in the previous sections. The COUNTER block will be discussed in this section.

OPERATION block

The operation block is the combinational logic block B in Figure 3.5. This block creates bits of resultant cubes by performing the operation on bits of the argument cubes according to the state of IT. It takes the following inputs:

1. Two bits from operand literal $A[i]$ ($a0_i$, $a1_i$).

2. Two bits from operand literal $B[i]$ ($b0_i$ and $b1_i$).

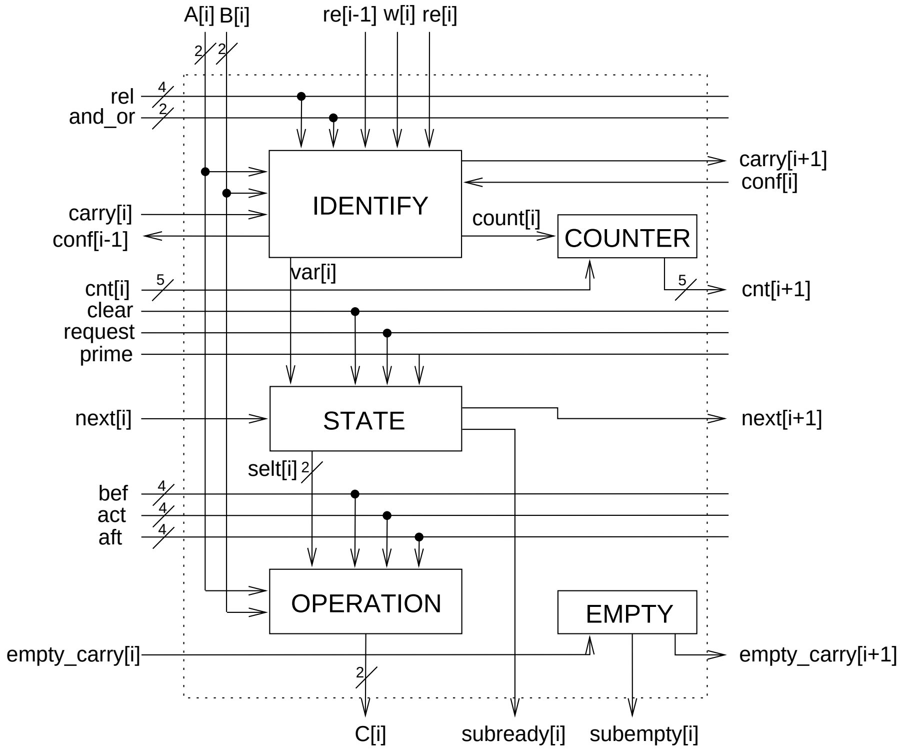


Figure 3.11: The block diagram of a Iterative Cell (IT)

3. Two bits signal $state[i]$ from block STATE.

4. 12 bits programmable inputs: 12 bits for functions $before$, $active$ and $after$ (4 bits each function).

It has two-bit output $C[i]$ ($c0_i$ and $c1_i$). The realization of OPERATION block follows the general programmable pattern (section 3.2) and is shown in Figure 3.12. This design takes one 4-bit 4-to-1 multiplexer and two 1-bit 4-to-1 multiplexers.

The signal $state[i]$ selects one function from three possible programmable functions by using one 4-bit 4-to-1 multiplexer, then realizes this function by using two 1-bit 4-to-1 multiplexers. Since there are only three possible output functions, so the last data input of the 4-bit 4-to-1 multiplexer is not used and is connected to a constant (0000 in the figure).

This is a general circuit for all kinds of cube calculus operations which can be described by Equations 3.2 and 3.3. The output values of functions $before$, $active$

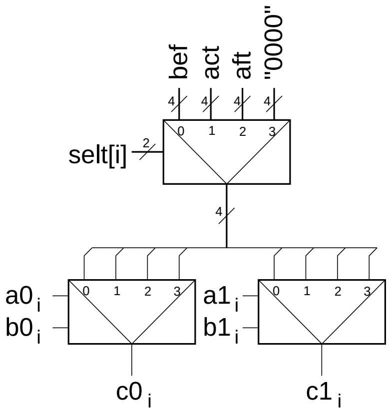


Figure 3.12: Block OPERATION of IT

and *after* of the cube operations described in Chapter 2 are listed in Table 3.1.

Block STATE

The STATE block is the combination of the sequential logic block and combinational logic block C in Figure 3.5. It has the following inputs:

1. signal $var[i]$ from IDENTIFY block.
2. signal $next[i]$ from preceding iterative cell.
3. global signals *reset*, *request* and *prime*.

The signal *prime* is used in complex combinational cube operations. As we discussed in section 3.3.2, there is a FSM in this block and the SM chart of the FSM was shown in Figure 3.6.

Since there are three states, this FSM can be realized by using two D flip-flops. The current state of the FSM is represented by the Q outputs of these two flip-flops, denoted by $state0[i]$ and $state1[i]$. The next state of the FSM is represented by D inputs of these two flip-flops, denoted by $ex0$ and $ex1$. This state machine can be described using the following formulas:

$$bef[i] = \overline{state1[i]} \cdot \overline{state0[i]} \tag{3.34}$$

$$act[i] = \overline{state1[i]} \cdot state0[i] \tag{3.35}$$

$$aft[i] = state1 \cdot \overline{state0[i]} \tag{3.36}$$

$$ex0[i] = bef[i] \cdot next[i] \cdot var[i] \tag{3.37}$$

$$ex1[i] = act[i] + aft[i] + bef[i] \cdot next[i] \cdot \overline{var[i]} \tag{3.38}$$

The signal $selt1[i]$ and $selt0[i]$ can be described by:

$$selt1[i] = state1[i] \cdot \overline{prime} \tag{3.39}$$

$$selt0[i] = state0[i] + var[i] \cdot prime \tag{3.40}$$

These two equations indicate that signal $selt[i]$ ($selt0[i]$ and $selt1[i]$) is set to 01 to select *active* function for the ITs that are specific positions when ILU executes the complex combinational cube operation (like prime). The signal $next[i+1]$ generated in this block is described by Equation (3.11).

IDENTIFY block

The IDENTIFY block is the combinational logic block A in Figure 3.5. The details of this block are fully discussed in section 3.4.1. The counter signal *count* generated in this block will be discussed in the next section, COUNTER block.

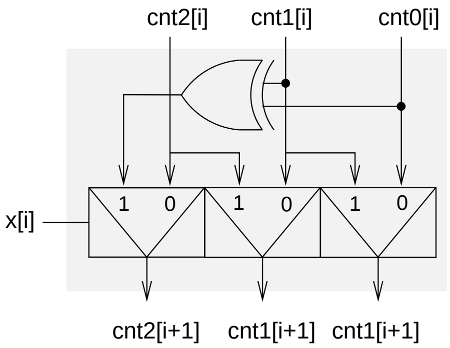
COUNTER block

The paper [31] shows that in addition to cube calculus operations presented in this thesis, the operation which take cubes as argument and return a number as a result is necessary. The simplest of such operation is calculating Hamming distance of two binary vector. All such operation require counting. In this simplified machine, we introduce counting, but it is restricted only to very simple operation used in pre-processing cubes. This counting is done by COUNTER block.

The COUNTER block counts the number of specific variables which is used in pre-relation/pre-operation (see section 3.5). The signal $count[i]$ equals 1 in the last IT of the specific variables. It can be described by:

$$count[i] = re[i] \cdot var[i] \tag{3.41}$$

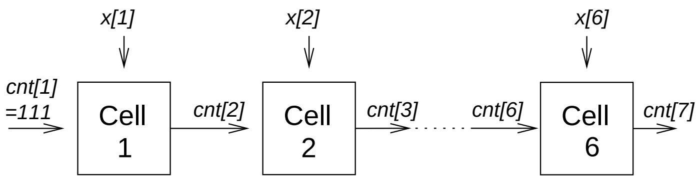
The $count[i]$ signal is generated in block IDENTIFY.



(a) cell

	cnt2	cnt1	cnt0
0	1	1	1
1	0	1	1
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	1
6	1	1	0
0	1	1	1

(b) sequence



(c) counter

Figure 3.13: counter

The counter is realized by an iterative network. A 3-bit counter is shown in Figure 3.13. The COUNT block is just a cell of the iterative network of the counter. To minimize the necessary logic for the cell, the cell of the counter is based on pseudo-random sequence generator. Figure 3.13 (a) shows a cell; and Figure 3.13 (c) shows the whole picture of the counter. This iterative counter can generate a fixed sequence of 7 numbers which is shown in Figure 3.13(b) (number 000 is not used). Therefore, this counter can be used to count the number of 1's in signal vector x.

The count begins with all bits are 1's. As the bits are shifted, a series of unique numbers will be generated and they are shown in Figure 3.13 (b). Since there must always be at least one bit equal to one in the pseudo-random counter, a k bits counter can count from 0 to $2^k - 2$. Therefore, a 3 bits counter can count from 0 to 6.

3 bits counter is sufficient for an ILU with 6 ITs. If all variables are binary, there are 6 specific positions at most. The counter must be able to signal 7 different numbers, to include the case where there are no specific positions. A decoder is

needed to convert the output of the last cell of counter to a binary number.

EMPTY block

One of the main design objectives idea of the CCM design was not to generate empty cubes. In order to do this, my idea is to simplify the design in past texts [15, 17, 18] which leads to design EMPTY block presented here. The role of this block is to generate signal *empty* to the control unit of the CCM informing whether the current resultant cube is a contradiction (empty cube) or not. This signal is used by the control unit of the CCM to remove empty cubes in the results, and makes the CCM not to generate empty cubes (see section section 5.4).

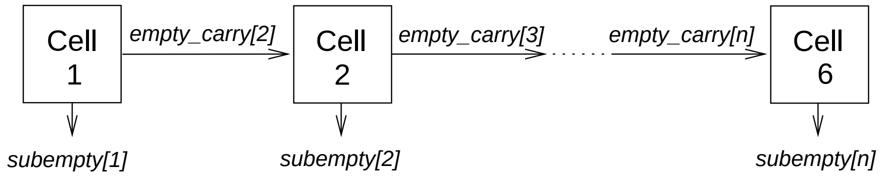


Figure 3.14: Iterative network used to generate *empty* signal

The *empty* signal is generated by a iterative network shown in Figure 3.14. The signal $empty\_carry[i]$ and $subempty[i]$ are as follows:

$$empty\_carry[i+1] = re[i-1] \cdot \overline{c0_i} \cdot \overline{c1_i} + \overline{re[i-1]} \cdot empty\_carry[i] \cdot \overline{c0_i} \cdot \overline{c1_i} \tag{3.42}$$

$$subempty[i] = empty\_carry[i+1] \cdot re[i] \cdot \overline{w[i]} \tag{3.43}$$

The *empty* signal is as follows:

$$empty = subempty[1] + subempty[2] + \cdots + subempty[n] \tag{3.44}$$

By comparing *empty* signal to *carry* signal, these equations are not hard to understand.

3.5 The architecture of the CCM

In our design, the cube calculus machine is a coprocessor to the host computer. The simplified block diagram of the CCM is shown in Figure 3.15; the thick arrow stands for data buses, and the thin arrow stand for control buses. As shown in the figure, the CCM communicates with the host computer through the input and the output FIFO. The ILU can take the input from register file and memory, and can write output to the register file[1], the memory, and the output FIFO. The ILU executes the cube operation under the control of *Operation Control Unit* (OCU). The *Global Control Unit* (GCU) controls all parts of the CCM and let them work together.

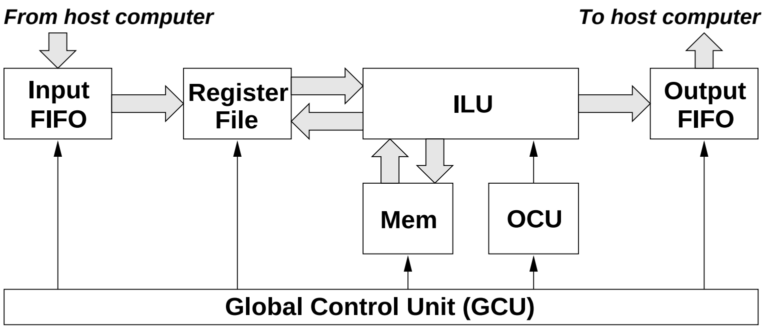


Figure 3.15: The simplified block diagram of the CCM

Since the design of the GCU depends on the design of the entire CCM, we will discuss the GCU in section 5.4 after the design of the CCM has been discussed in detail. In the next section, we will discuss the function of the OCU.

3.6 The Operation Control Unit (OCU)

The ILU executes cube operations under the control of the operation control unit (OCU) in our design; and the OCU is under the control of the GCU. The communication between the GCU and OCU is very simple: when the CCM is ready to execute the cube operation, the GCU set the signal *ilu_enable* to 1 to tell the OCU to execute the cube operation; then the OCU controls the ILU to execute the cube operation; after the cube operation is done, the OCU set the signal *ilu_done* to 1 to tell the GCU that the cube operation is done. This is illustrated in Figure 3.16.

[1]Actually, the ILU can write output to only one register of the register file, see Chapter 5 for detail.

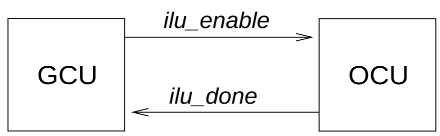


Figure 3.16: The communication between the GCU and the OCU

The algorithm of the CCM has been discussed in section 3.3.2. The state diagram of OCU is shown in Figure 3.17. Now let us take a look at these states.

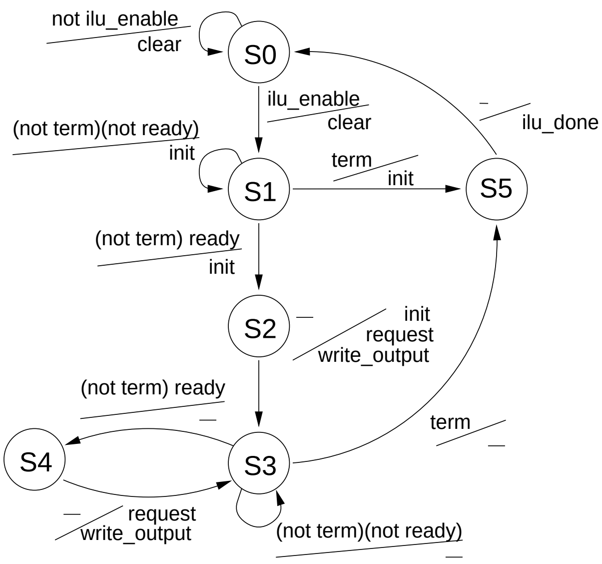


Figure 3.17: The state diagram of the OCU

- State S0: This is the initial state of the OCU. The *clear* signal (see Figure 3.7) is set to 1. The *clear* signal is connected to all synchronous reset inputs of the D flip-flops that are used to realize the state machines inside ITs, therefore, all ITs are reset to state *before*.

  If the signal *ilu_enable* is 1, then the OCU goes to state S1; otherwise, the OCU keeps in state S0.

  When the CCM is used to calculate combinational cube operations (including complex combinational cube operations), the OCU keeps in state S0 because the GCU will keep the signal *ilu_enable* to be 0.

- State S1: The OCU let the ILU begin to execute the sequential cube operation by setting the signal *init* (the first *next* signal) to 1 in this state. After that, the OCU will wait for signals *term* (the last *next* signal) and *ready*. If signal *term* becomes 1, which means the cube operation is done, then the OCU goes to state S5. If signal *ready* becomes 1 and signal *term* keeps 0, then the OCU goes to state S2; otherwise (both signals keep 0), the OCU will keep in state S1.

- State S2: The OCU generates the first resultant cube by setting signal *request* to 1; and it also generates signals *write_output* to write the result out. The OCU will always go to state S3 from state S2.

- State S3: The OCU will wait for signals *term* and *ready* again. If signal *term* becomes 1, then the OCU goes to state S5; if signal *ready* becomes 1 and signal *term* keeps 0), then the OCU goes to state S4; otherwise (both signals keep 0), the OCU will keep in state S3.

- State S4: The OCU generates one resultant cube by setting signal *request* to 1; and it also generates signals *write_output* to write the result out. The OCU will always go to state S3 from this state.

- State S5: The OCU informs the GCU that the cube operation is done by setting the signal *ilu_done* to 1. The OCU will always go to state S0 from this state.

3.7 Pre-relation/Pre-operation

To explain the concept of pre-relation/pre-operation, let us take a look at the sharp operation again. The sharp operation is defined by Equations (2.12) and (2.13). It can be seen that we can not use Equation (2.13) to carry out the operation unless two operand cubes satisfy $A \cap B \neq \emptyset$ and $A \not\subseteq B$. When $A \cap B = \emptyset$, the operation $A \# B = A$; when $A \subseteq B$, the operation $A \# B = \emptyset$. $A \cap B = \emptyset$ and $A \subseteq B$ are called *pre-relation* of the sharp operation; $A \# B = A$ and $A \# B = \emptyset$ are called *pre-operation* of the sharp operation. It can be seen that the pre-operation is the output function when pre-relation is satisfied.

Table 3.2: Pre-relation and Pre-relation of the Cube Calculus Operations

Operation	**Pre-relation**	**Pre-operation**
sharp/disjoint sharp	$A \cap B = \emptyset$	A
	$A \subseteq B$	$\emptyset$
consensus	$distance(A, B) = 0$	$A \cap B$
	$distance(A, B) > 1$	$\emptyset$
crosslink	$degree(A) \neq degree(B)$	$\emptyset$

Some cube operations which have pre-relation and pre-operation are listed in Table 3.2. The name, pre-relation and pre-operation of the operations are listed from left to right in the table, respectively. It can be seen that some cube operations have two pre-relations and pre-operations, such as the sharp and consensus operations.

As we discussed in section section 3.4.2, the COUNT block can count the number of specific variables. In another words, it can count the number of the pairs of literals A_i and B_i that satisfy the relation. If the relation (*rel* and *and_or*) are substituted by the pre-relation, the COUNT block can be used to count the number of the pairs of literals A_i and B_i that satisfy the pre-relation.

Example 3.7. For the pre-relation $A \cap B = \emptyset$, we can count the number (denoted by k) of the pairs of literals A_i and B_i that satisfy $A_i \cap B_i = \emptyset$. If $k > 0$, then $A \cap B = \emptyset$, otherwise $(k = 0)$, $A \cap B \neq \emptyset$.

The pre-relation can be represented by *partial pre-relation* (*prel*), *partial pre-relation type* (*pand_or*), *pre-relation compare type* (*pcmp*) and *pre-relation compare value* (*pval*).

Example 3.8. For the pre-relation shown in Example 3.7, *prel* is $\bar{a}_i + \bar{b}_i$; *pand_or* is *and*; *pcmp* is "$>$"; and *pval* is 0 ($A \cap B = \emptyset$ is the relation of the crosslink operation, so *prel* and *pand_or* are the same as *rel* and *and_or* of the crosslink operation, respectively).

The pre-operations listed in Table 3.2 are bitwise functions. The decomposed pre-relations and pre-operations of the sharp/disjoint sharp and consensus operations are listed in table 3.3. The crosslink operation can not be decomposed in this way (It needs two carry signals to compare the degrees of two operand cubes; but this simplified design of the CCM has only one carry signal, see [15]). Each row of the table describes one pair of pre-relation and pre-operation.

Table 3.3: Decomposed Pre-relation and Pre-operation of the Cube Operations

Operation	**Pre-relation**				**Pre-operation**
	prel	*pand_or*	*pcmp*	*pval*	(*poper*)
sharp/disjoint sharp	$\bar{a}_i + \bar{b}_i$	and	$>$	0	a_i
	$a_i\bar{b}_i$	or	$=$	0	$\emptyset$
consensus	$\bar{a}_i + \bar{b}_i$	and	$=$	0	$a_i \cdot b_i$
	$\bar{a}_i + \bar{b}_i$	and	$>$	1	$\emptyset$

The encoded pre-relations and pre-operations of the sharp/disjoint sharp and consensus operations are listed in table 3.4. The *prel* and pre-operation (*poper*) are decoded by their output values. The encoding of *pand_or* is the same as *and_or*. The encoding of *pcmp* is as follows: "<" — 00, "=" — 01 and ">" — 10. Each row of the table describes one pair of pre-relation and pre-operation.

Table 3.4: Encoded Pre-relation and Pre-operation of the Cube Operations

Operation	**Pre-relation**				**Pre-operation**
	prel	*pand_or*	*pcmp*	*pval*	(*poper*)
sharp/disjoint sharp	1110	1	10	0	0101
	0010	0	01	0	0000
consensus	1110	1	01	0	0001
	1110	1	10	1	0000

For realizing pre-relation/pre-operation, the ILU is modified as shown in Figure 3.18. Please note that only the modified part is shown in the figure. This design is very straightforward. In this design, a cube operation can have at most two pairs of pre-relation/pre-operation[2], and all pre-operations should be combinational operations,

[2]All cube operations presented in this thesis have at most two pairs of pre-relation/pre-operation. It is easy to extend this design to handle more than two pairs of pre-relation/pre-operation.

which means that the operation can be described as a bitwise function. Thus the signals *prel*, *pand_or*, *pcmp*, *pval* and *poper* shown in the figure have a suffix 1 or 2, which corresponds to the first or the second pair of the pre-relation/pre-operation, respectively. All input signals come from the registers (see section 5.2.3).

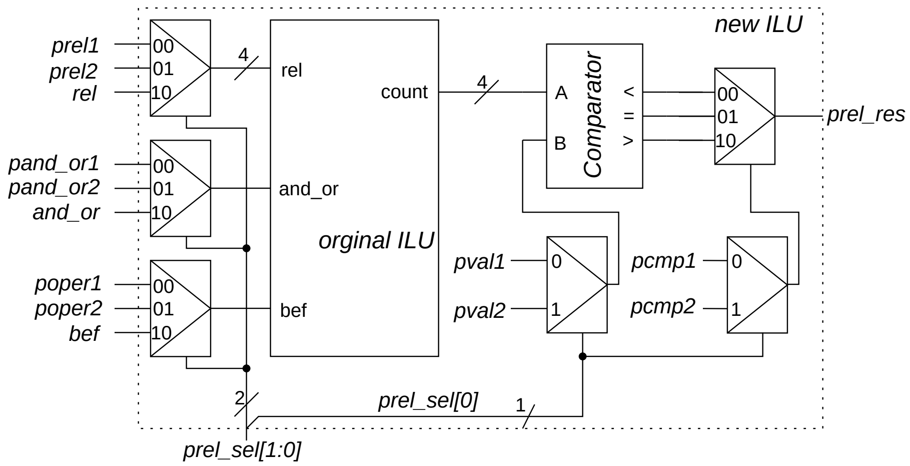


Figure 3.18: Realization of Pre-relation/Pre-operation

The signal *prel_sel* (having 2 bits) is generated by the GCU (see section 5.4). When the CCM executes the first pre-relation/pre-operation, the signal $prel\_sel = 00$; when the CCM executes the second pre-relation/pre-operation, the signal $prel\_sel = 01$; otherwise, the signal $prel\_sel = 10$.

The only output signal *prel_res* is the result of pre-relation. When $prel\_res = 1$, the pre-relation is satisfied, otherwise, the pre-relation is not satisfied. This signal is used by the GCU (see section 5.4).

If $prel\_sel = 00$ or 01, and all ITs of the ILU are in *before* state, then the cube operation is evaluated according to the function *poper*1 or *poper*2, respectively.

Let us observe that among combinational cube operations that have pre/relation/pre-operation, there exists certain subset operations, like intersection and cofactor operations, can be characterized by the following:

- These operations only have one pair of pre-relation/pre-operation, and the result of pre-operation is always an empty cube.

- These operations can be always carried out by following the basic equation of the operation without checking the pre-relation. After that the EMPTY blocks will check whether the result is an empty cube or not.

- These operations can be carried out using pre-relation/pre-operation, but it will take more time to execute this kind of operations because the GCU will go through more states to check the pre-relation (see section 5.4).

Based on these observations, this kind of operations will be carried out without using pre-relation/pre-operation in this thesis.

CHAPTER 4

PAM and DECPeRLe-1

There are two ways to implement a specific digital processing task: software approach and hardware approach.

In software approach, a general purpose computer is programmed to perform a processing task. The structure of any general purpose computer has been highly optimized to process arbitrary codes. In many cases, however, it is poorly suited to the specific algorithm, so the performance does not meet the requirement.

In the hardware approach, a specific circuitry/machine for the specific processing task is designed. The machine's structure, processors, storage and interconnect, are tailored to the application. The result is more efficient, with less actual circuitry than what general purpose computers would require.

The drawback of the hardware approach is that a specific architecture is usually limited to processing a small number of algorithms, often a single one. Attaching special purpose hardware to a general purpose computer, say for video compression, speeds up the system when the system is actually compressing video. It contributes nothing when the system is required to perform some different task, say machine learning.

An alternative way is a reconfigurable hardware, which combines software versatility and hardware performance. DEC PRL's Programmable Active Memory (PAM) was one of the earliest pioneers in this so-called *reconfigurable computing* field. PAM is a novel form of universal reconfigurable hardware co-processor based on SRAM-based FPGA technology.

This chapter presents the concept of PAM and the detail of one realization of PAM — PeRLe-1 board which is available in the Department of Electrical Engineering at

Portland State University. For more complete description about PAM and PeRLe-1 board, please see references [5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

4.1 PAM

A PAM concept is a uniform array of identical cells all connected in the same repetitive fashion. Each cell, called a PAM for Programmable Active Bit, must be general enough so that the following holds true: Any synchronous digital circuit can be realized, through suitable programming, on a large enough PAM, for a slow enough clock.

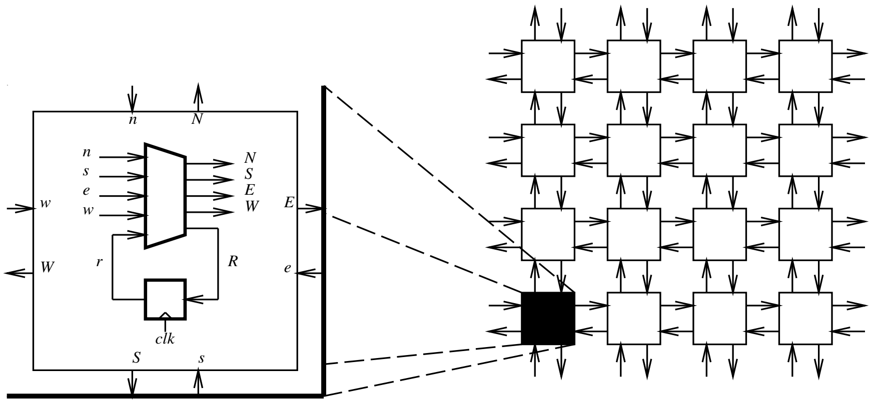


Figure 4.1: A simple PAM

Figure 4.1 shows a simple PAM implementation as a regular matrix of Manhattan connected identical PABs. Each PAB has:

- 4 inputs $< n, s, e, w >$
- 4 outputs $< N, S, E, W >$
- 1 register (flip-flop) with input R and output r, synchronous with the PAM's global clock *clk*
- 1 combinatorial gate F (5 inputs, 5 outputs) connected so that: $g(n, s, e, w, r) =< N, S, E, W, R >$

- $160 = 5 \times 2^5$ control bits which specify the truth table of function F.

A *program* for such a PAM with m active bits is a sequence of $160m$ control bits (*bitstream*) representing the truth tables for each PAM. This program can be *downloaded* into the *configuration memory* of the PAM. From this instant, and until the program is changed again, the PAM behaves as the particular finite state machine specified by the bitstream.

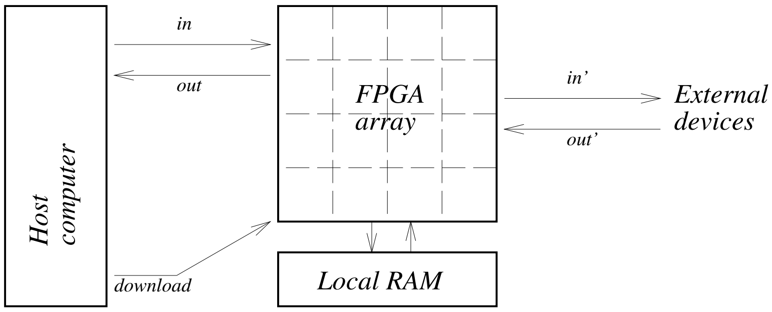


Figure 4.2: PAMs as virtual machines

The PAM is used to implement a *virtual machine* which can be dynamically configured as a large number of specific hardware devices. As shown in Figure 4.2, the PAM is connected to a host computer through the *in* and *out* links. The host can download configuration bitstreams into the PAM. After configuration, the PAM behaves, electrically and logically, like the ASIC (Application Specific Integrated circuit) defined by the specific bitstream. The PAM may operate in stand-alone mode, hooked to some external system through the *in'* and *out'* links. It may operate as a co-processor under host control, this co-processor is specialized to speed-up some crucial computations. It may operate in both modes simultaneously, connecting the host to some external system, like an audio or video device, or some other PAM.

The PAM as presented above is only a theoretical concept and has been not realized in practice. Digital's Paris Research Laboratory developed several boards based on the PAM concept, but realized with the existing FPGA chips, thus the actual cell structure and connections are different from the cell and connection model shown in Figure 4.1. DECPeRLe-1 is the third generation PAM board, and it was built at Digital's Paris Research Laboratory in 1992.

4.2 PeRLe-1 Board

The overall structure of PeRLe-1 is shown in Figure 4.3. The PeRLe-1 board is organized around a central computational matrix made up of 16 Xilinx XC3090 LCAs[1] (M00 to M15 in the figure), surrounded by a 4 1MB RAM bank, and 7 other LCAs to implement switching and controlling functions. The data buses and their width are also shown in the figure (everything shown in the figure will be explained later in this section). All control wires are shown in Figure 4.8 and will be discussed in related sections. This section describes all programmable resources and the way they are interconnected with each other.

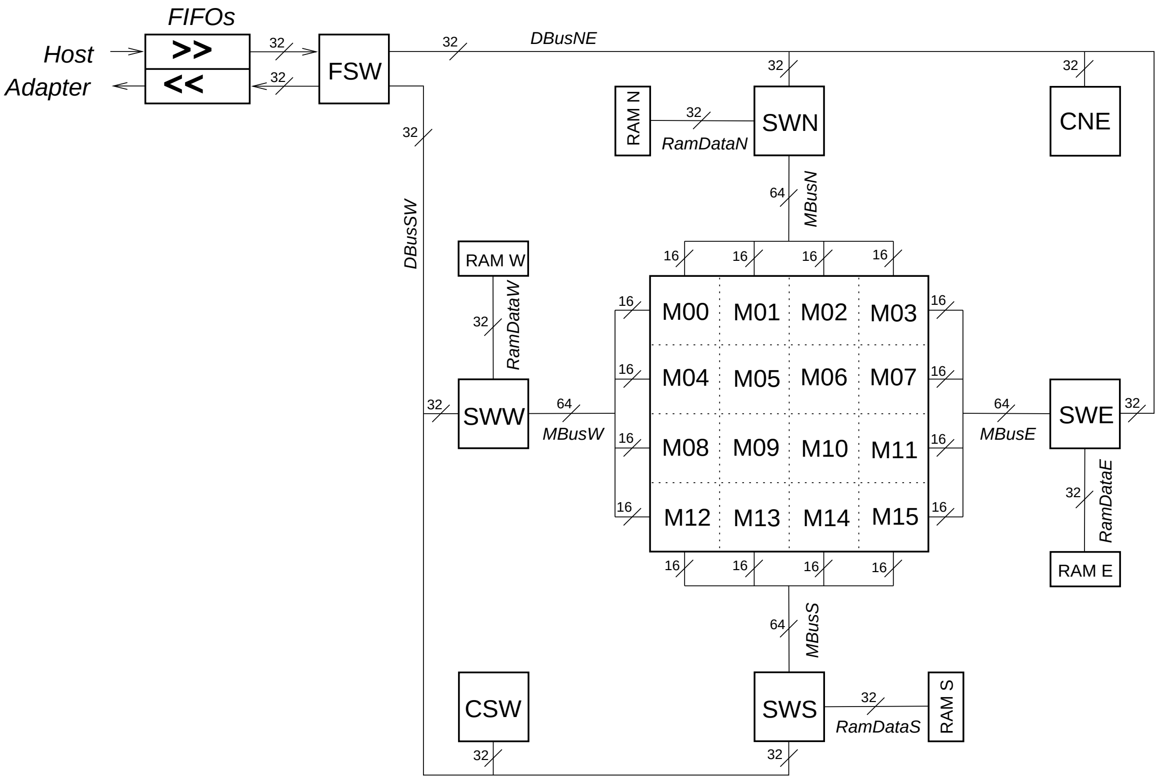
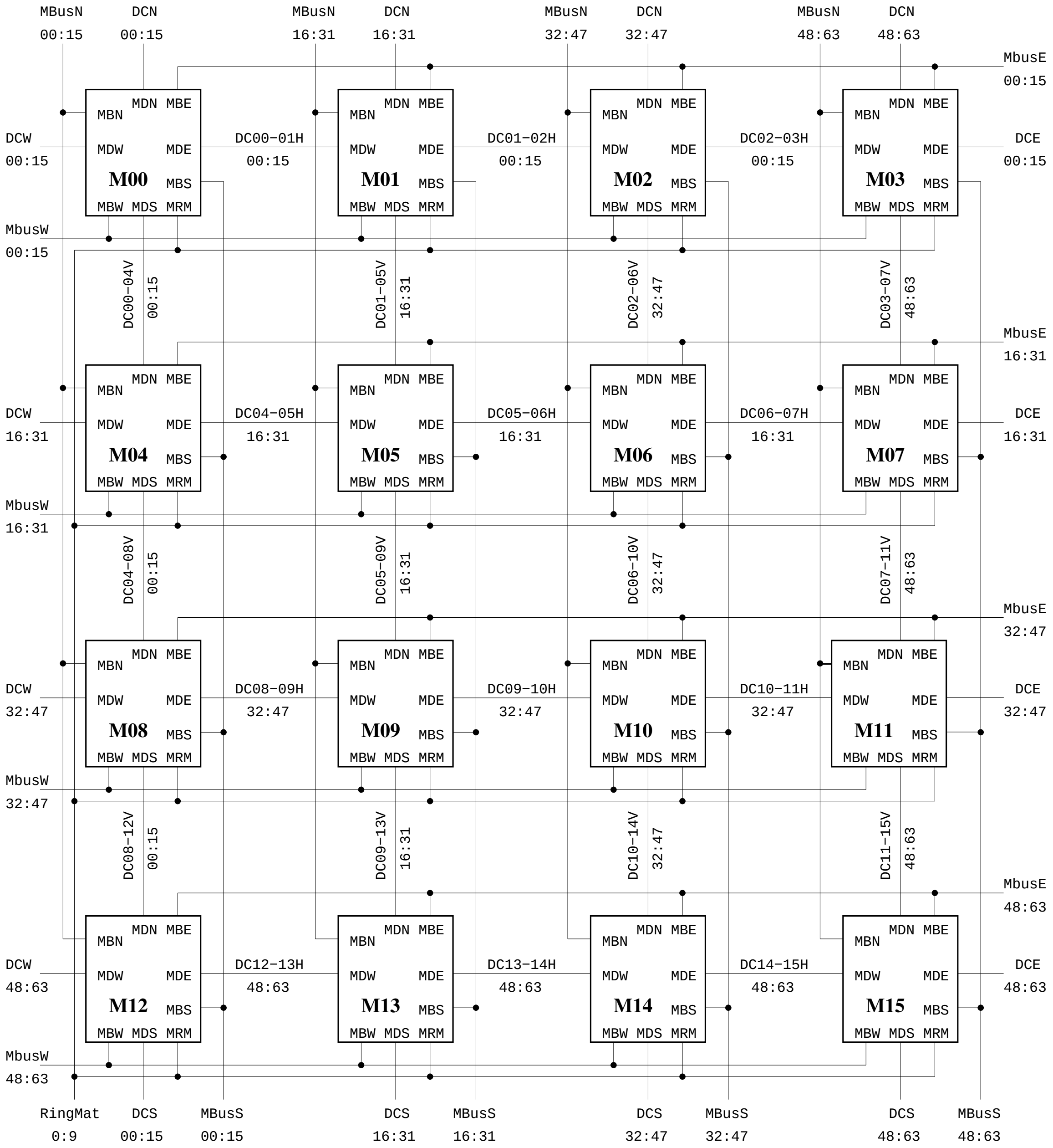


Figure 4.3: PeRLe-1 architecture

[1]LCA stands for Logic Cell Arrays



DCN, DCE, DCS and DCW: North/East/South/West matrix side to connectors

MDN, MDE, MDS and MDW: Matrix North/East/South/West direct connections

MBN, MBE, MBS and MBW: North/East/South/West matrix buses

Figure 4.4: PeRLe-1 matrix

4.2.1 Computational matrix

The central computational matrix is a 4×4 matrix of Xilinx 3090 LCAs. These LCAs are interconnected with each other. The internal structure of the matrix is shown in Figure 4.4. The LCAs are named LCA_M00 to LCA_M15 (M00 to M15 in the figure). This matrix can be used to develop any kind of digital circuitry: data path, control unit and others. But it is typically used to develop the data path of the application. The interconnection resource between them can be classified into the following three categories.

Direct Connections

As shown in Figure 4.5, these wires connect the adjacent sides of adjacent LCAs. The main purpose of direct connections is to extent the internal regularity of the LCA to the matrix level. The matrix can be seen as a large FPGA with 64×80 PABs (one XC3090 FPGA has 16×20 PABs). Each LCA has 16 such wires on each side.

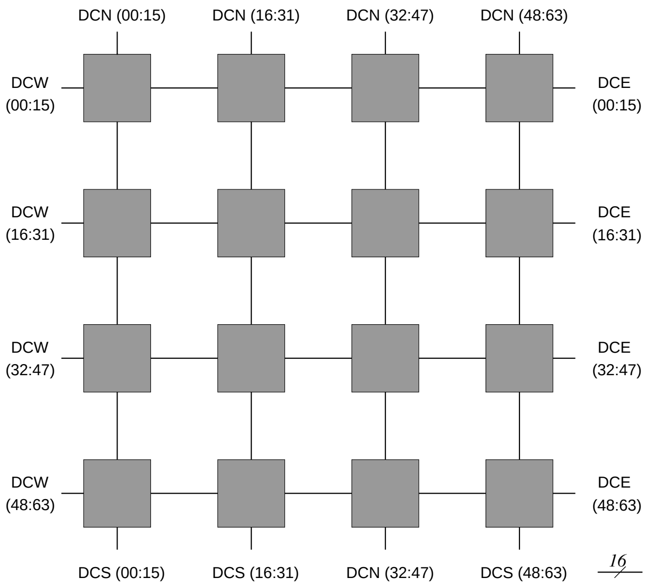


Figure 4.5: PeRLe-1 Direct connection

As shown in Figure 4.4, the horizontal and vertical direct connections are named DCmm_nnH and DCmm_nnV, respectively, where mm and nn are the numbers of the connected LCAs. For instance, DC00_04V represents vertical direct connections between LCA_M00 and LCA_M04.

The direct connections at the edges of the FPGA matrix are called DCN, DCE, DCS and DCW as shown in Figure 4.5. These four 64-bit-wide connections are connected to external connectors, which can be used to connect other devices, for example, another PeRLe-1 board.

Buses

As shown in Figure 4.6, the horizontal or vertical wires connect the corresponding side of all 4 LCAs in the same row or column. They can thus efficiently distribute global data in one direction, and are comparable to the *longline* interconnections resources in Xilinx internal architecture. Each LCA has 16 such wires on each side.

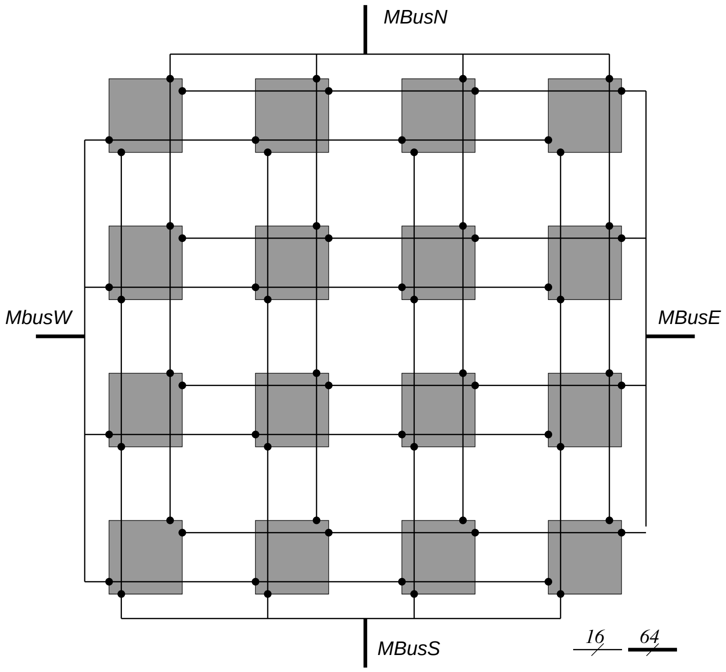


Figure 4.6: PeRLe-1 Matrix data buses

According to their directions, these buses are named *matrix North, East, South and West bus*, respectively, and represented by MBusN, MBusE, MBusS and MBusW for short. Each bus has 64 wires which are connected to switches on the corresponding side of matrix FPGAs.

Rings

As shown in Figure 4.7, the wires connect all the matrix LCAs and two control LCAs. These connections are very useful for global control signals distribution since they connect to all the matrix LCAs. There are 10 such wires. Note that because of their electrical loading (they are used to connect 18 LCAs, 16 matrix LCAs and two control LCAs), these wires are slower than the buses and should be used with care in high performance designs.

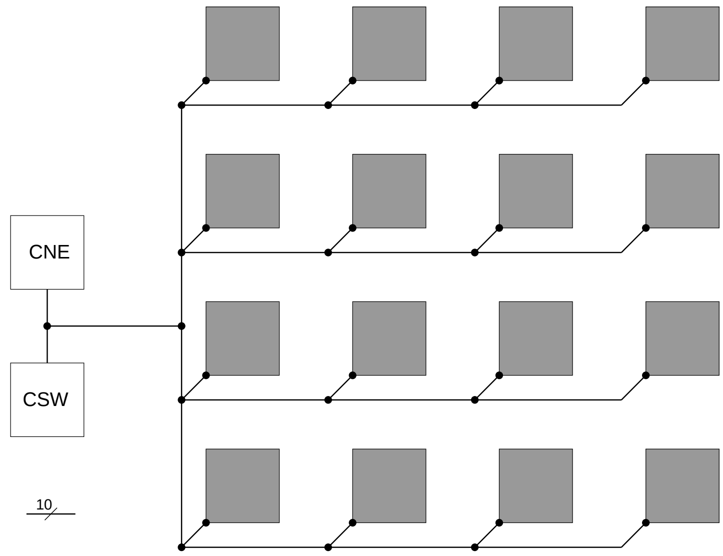


Figure 4.7: PeRLe-1 Matrix rings

4.2.2 Switches and I/O buses

Figure 4.3 shows the way that FIFOs, RAM banks and the central matrix are connected through two 32-bit data buses and five programmable switches (FPGAs).

As shown in Figure 4.3, there is one matrix switch on each side of the matrix, respectively called *North Switch* (SWN), *East Switch* (SWE), *South Switch* (SWW) and *West Switch* (SWW), which connect the corresponding matrix data buses and corresponding RAM banks. These 4 switches (SWN, SWE, SWS, SWW) also connect to two 32 bits I/O buses, called *North-East Bus* (DBusNE) and *South-West Bus* (DBusSW) after the names of the switches they respectively connect.

Two I/O buses (DBusNE, DBusSW) connect to the input and output FIFOs through the fifth switch called *Fifo Switch* (FSW), and also connect to corresponding controllers, called *North-East Controller* (CNE) and *South-West Controller* (CSW).

As their names imply, the FPGAs CNE and CSW are typically used to develop the controller of the application because they connect to all other parts of the PeRLe-1 , FIFOs, Memory banks and other FPGAs.

4.2.3 Control resource

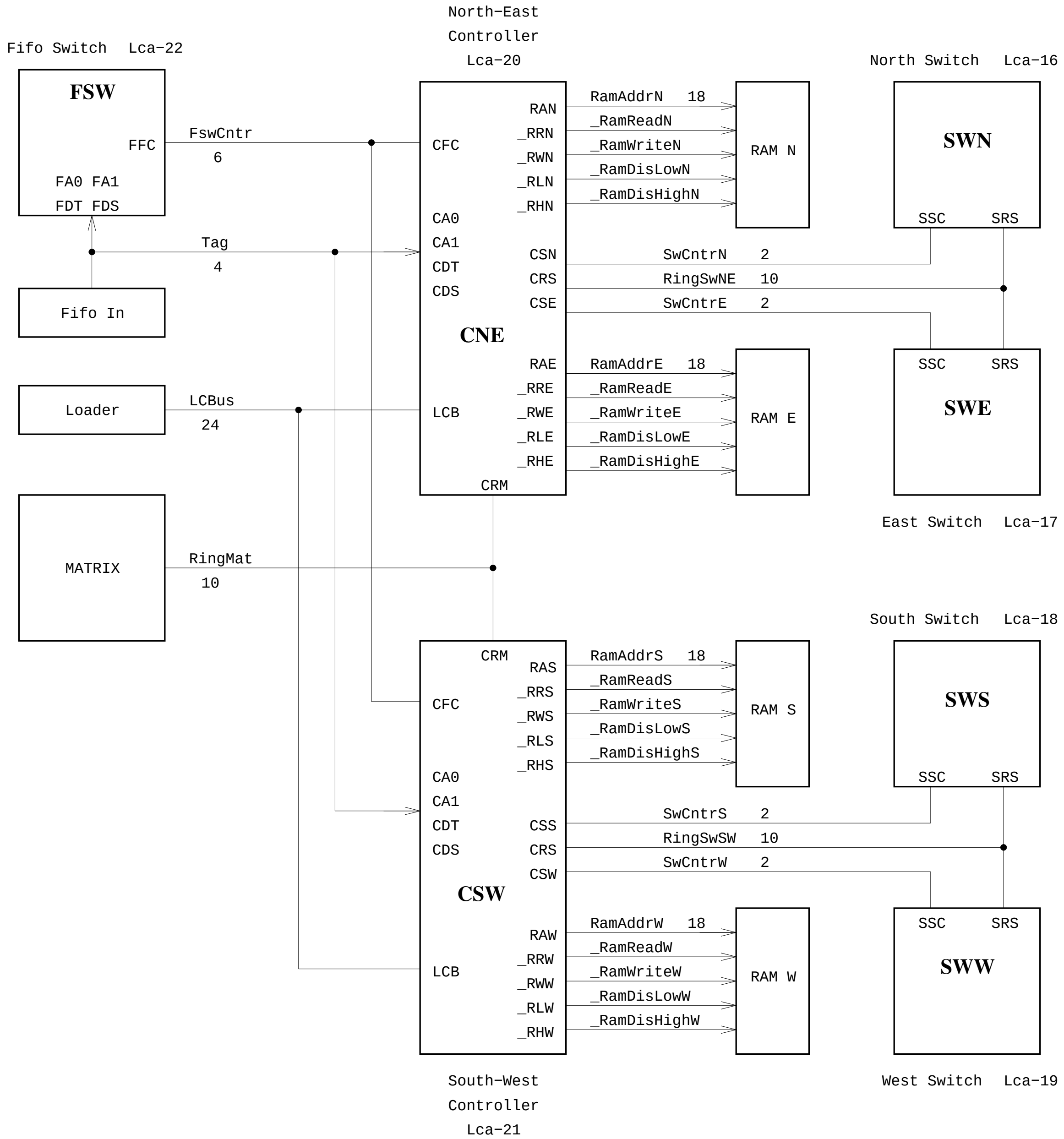
The control resource is the programmable resource that can be used to develop the control part of the application other than data path part. As shown in Figure 4.8, the data path resource (matrix, RAM banks, FIFOs and switches) need the following set of control wires:

MATRIX RINGS: There are 10 matrix global wires (see section 4.2.1). These wires are not used in our CCM design.

RAM ADDRESS: Each RAM bank has a 18-bit-wide address, that specify the word address of the current read or write operation. Since our CCM design uses two of four memory banks, two addresses are used in our CCM design.

RAM CONTROLS: Each RAM bank has 4 control signals (see section 4.2.4). Since our CCM design uses two memory banks, the corresponding controls are used in our CCM design,

SWITCH CONTROLS: Each pair of matrix switches (*North* and *East* / *South* and *West*) has 10 control wires that are the equivalent of the matrix rings, and are called *switch ring*. The *Fifo Switch* has 6 control wires. In addition, each of the matrix switches has 2 dedicated control wires (see Figure 4.8). These wries are used in our CCM design.



(see section sections 4.2.3, 4.2.4 and 4.2.6 for more detail)

Figure 4.8: PeRLe-1 control wires

FIFO CONTROLS: Each of the two FIFOs has one status wire: empty flag for input FIFO / full flag for output FIFO; and one control wire: write for output FIFO / read for input FIFO (see Figure 4.8). These wires are used in our CCM design.

TAGS: Four "tag" wires along the input data wires on the input FIFO (see section 4.2.6). These wires are not used in our CCM design.

CLOCK CONTROL: The clock generator has two control wires that can be driven by the application design (see section 4.2.5). These wires are not used in our CCM design.

LCBus: There is a 24-bit-wide communication path between the board and the host, called *LCBus* (see section 4.2.6). These wires are not used in our CCM design.

As shown in Figure 4.8, all these control wires are connected to one of two controller LCAs (CNE, CSW) or both of them. These two controllers are identical except that each of them controls two of the four switches and memory banks. These two controllers also connect to corresponding I/O bus in order for it to be able to communicate with the main datapath.

4.2.4 Memory subsystem

PeRLe-1 contains 4MB of high-speed static RAM organized in 4 banks of 256K 32-bit words (4 bytes a word). These banks are named *North, South, East* and *West* after the matrix switch to which they are connected. Each bank is completely independent of the others and has its own data, address and control signals:

DATA BUS: 32 data wires connect to the corresponding matrix switch. They are represented by $RamdataX$, where X is one of *N,S,E,W* to respectively specify the North, South, East or West RAM bank.

ADDRESS BUS: 18 address wires ($1MB = 2^{20} = 2^{18} \times 2^2$) connect to the corresponding controller. They are represented by $RamAddrX$.

CONTROL BUS: 4 active-low control signals to specify the read/write operation, connect to the corresponding controller.

- $\overline{RamReadX}$: read command.
- $\overline{RamWriteX}$: write command.
- $\overline{RamDisLowX}$: disable lower half-word (bits 0 to 15).
- $\overline{RamDisHighX}$: disable upper half-word (bits 16 to 31).

In our CCM design, we always read/write memory by a 32-bit word a time. Therefore, the signal $\overline{RamDisLowX}$ and $\overline{RamDisHighX}$ are always set to 1 (not actived).

The basic read and write transactions both last one clock cycle, and either may be performed at every cycle.

Read memory

To read a particular word of memory, the word address ($RamAddrX$) must be presented and the read command ($\overline{RamReadX}$) must be asserted at the beginning of a cycle; the data word read from memory will be available on the data wires at the end of the same cycle and may be latched on the next clock tick. A RAM bank can be seen as a combinational device when read.

Write memory

To write a particular word of memory, the word address ($RamAddrX$), the data ($RamDataX$) and the write command ($\overline{RamWriteX}$) must be asserted during the same cycle; the word will have been written by the end of the same cycle, and the address and the data may be removed after the next clock tick.

The reading or writing of either half of the data word may be independently disabled by asserting the corresponding disabled command ($\overline{RamDisLowX}$ or $\overline{RamDisHighX}$) during the transaction cycle. The memory system is clocked by *clock1* signal (see section 4.2.5).

4.2.5 Clock subsystem

Two global synchronous clock signals, *clock0* and *clock1*, are available to all PeRLe-1 LCAs for proper synchronous operation. These clock signals are generated by a phase-locked-loop oscillator synchronized to the host bus master clock. When PeRLe-1 is connected to a DEC 5000/24 workstation (25MHz TURBOchannel), its frequency can be programmed under software control to be any value from 360 KHz to 120 MHz, with an average resolution of 0.01%.

4.2.5.1 Clock modes

Under software (the program running on the host) control, the clock generator may be put in the following operation modes:

STOP MODE: No clock is generated in this mode.

FREE-RUN MODE: This is the normal operating mode, where the clock continuously runs at the prescribed frequency. Our CCM design will run in this mode.

BURST MODE: This is a mode where, under software control, the clock generator will generate a burst of 1 to 31 clock ticks at the prescribed frequency, then stop. This is useful to implement step and double-step debugging modes. We will use this clock mode to debug our CCM design.

AUTOSTOP MODE: There are two autostop modes: *FifoIn-Autostop* and *FifoOut-Autostop*. In the *FifoIn-Autostop* mode, *clock0* will automatically stop whenever the design attempts to read an empty input FIFO. Similarly, in the *FifoOut-Autostop* mode, *clock0* will automatically stop whenever the design attempts to write a full output FIFO. These two modes can be enabled at the same time. Our CCM design will always run in this mode.

CLOCK1-DIV2: This mode is useful for very high performance designs. *clock1* runs at half the speed of *clcok0*. This allows the RAM and FIFOs to be operated on half the speed of the matrix. This clock mode is not used in our CCM design.

4.2.5.2 *clock0* stop

The *clock0* may stop under control of the application on the board. This is usually used to implement *flow-control*, where the entire datapath is stopped waiting for input data (when the input FIFO is empty) or output space (when the output FIFO is full). It is much more efficiently and easily implemented this way than through the global distribution of a clock enable signal. In effect, when application runs entirely on *clock0* and both autostop modes are enabled, the application can be seen as a perfect synchronous system without flow-control concern.

The *clock0* signal will stop under one or more of the following conditions:

- The active-low $\overline{ClkStop}$ signal is asserted from one of the controllers.
- In the *FifoIn-autostop* mode, the input FIFO is empty and the active-low $\overline{FifoInRead}$ signal is asserted from one of the controllers.
- In the *FifoOut-autostop* mode, the output FIFO is full and the active-low $\overline{FifoOutWrite}$ signal is asserted from one of the controllers.

The memory subsystem and the FIFOs are clocked by *clock1*. This means that it is still possible to perform memory and/or FIFO operations even when *clock0* is stopped. This feature is not used in our CCM design.

4.2.5.3 Slow mode

Under control of an application on the board, it is possible to slow down the clock (divide its frequency by 4) by asserting the active-low $\overline{ClkSlow}$ signal from one of the controllers. This is useful when an application can run at a very high speed, but must infrequently perform an operation that is impossible to be performed at the high speed (like stopping the clock, or accessing the FIFOs). The $\overline{ClkSlow}$ can be asserted at any speed, but its operation is asynchronous, that is, it will take an unpredictable number of cycles for it to be effective. If the operation frequency is less than 80 MHz, this number of cycles is however guaranteed to be less than or equal to 6. This feature is not used in our CCM design.

4.2.6 Host interface

The PeRLe-1 application is running under the control of the software program executed on the host computer. The communication between PeRLe-1 application and its driving software program can be done through FIFOs or LCBus.

4.2.6.1 FIFOs

There is a 32-bit-wide, 512-word-deep FIFO in each direction (see Figure 5.1). These FIFOs are called *input FIFO* for the Host-to-PAM direction and *output FIFO* for the PAM-to-Host direction, respectively. On the application side, their data wires are connected to the *Fifo Switch* LCA and their control wires to the two Controller LCAs. Both FIFOs are purely synchronous devices when operated from the application side. They appear to be always available for reading or writing in *autostop* mode.

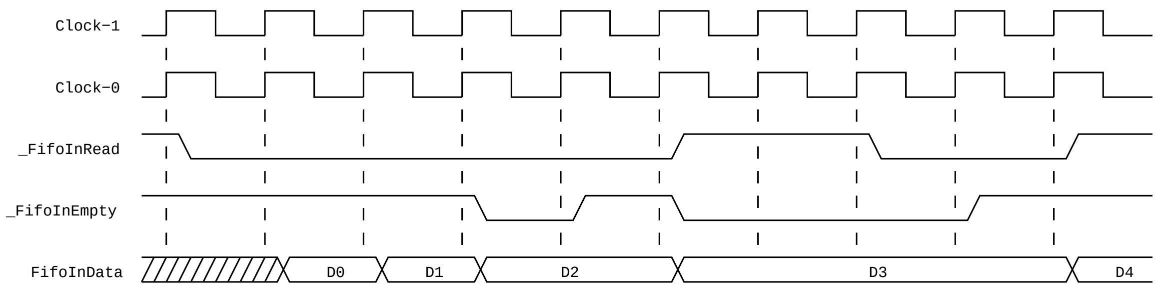
The input FIFO and output FIFO are synchronous devices that offer two active-low status signals $\overline{FifoInEmpty}$ and $\overline{FifoOutFull}$ and two active-low command signals $\overline{FifoInRead}$ and $\overline{FifoOutWrite}$. These four signals are connected to the two Controller LCAs CNE and CSW (see Figure 4.8). The Input FIFO is operated as follows:

- When the design is in *FifoIn-autostop* mode, if on one cycle $\overline{FifoInRead}$ is asserted and $\overline{FifoInEmpty}$ is inactive, the FIFO is read and the data word shows on the *FifoInData* wires in the next cycle (see Figure 4.9 (a)).

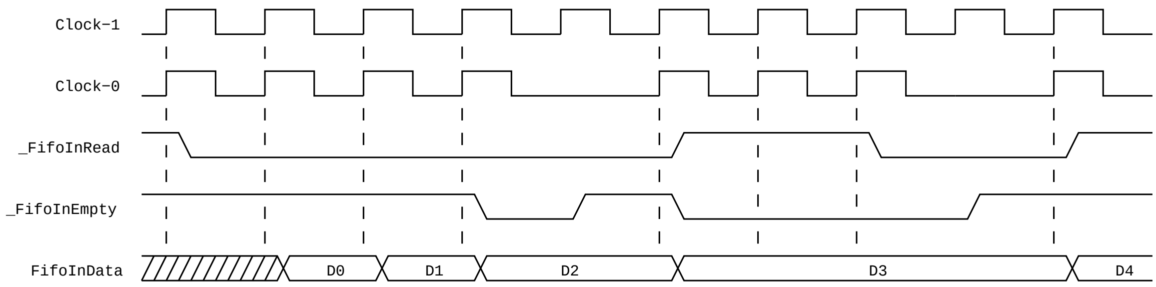
- In any other situation except in the *FifoIn-autostop* mode, no operation is performed in this cycle and the current value of *FifoInData* is held to the next cycle.

- When the design is in *FifoIn-autostop* mode and the design is entirely clocked by *clock0*, if $\overline{FifoInRead}$ is asserted, the data word is available on the next cycle (see Figure 4.9 (b)).

The output FIFO is operated as follows:



(a) the input FIFO read operation (no autostop)



(b) the input FIFO read operation (autostop)

Figure 4.9: The input FIFO operation

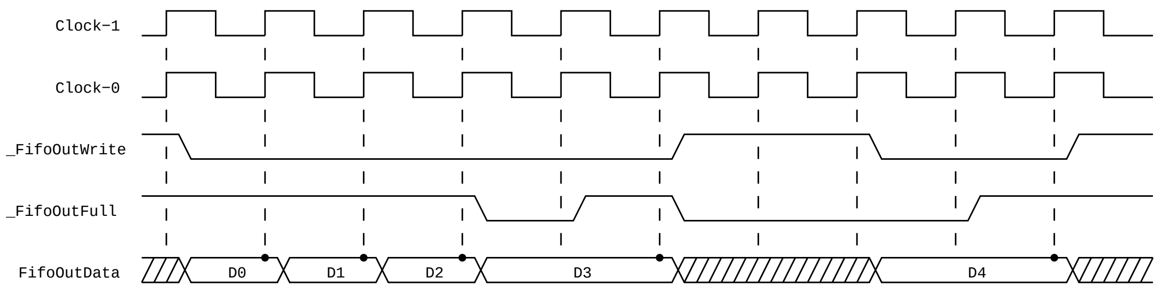
- When the design is not *FifoOut-autostop* mode, if on one cycle $\overline{FifoOutWrite}$ is asserted and $\overline{FifoOutFull}$ is inactive, the FIFO is written and the data word present on the *FifoOutData* wires in this cycle is pushed into the output FIFO (see Figure 4.10 (a)).

- In any other situation except in *FifoOut-autostop* mode, no operation is performed in this cycle and the data present on the *FifoOutData* is ignored.

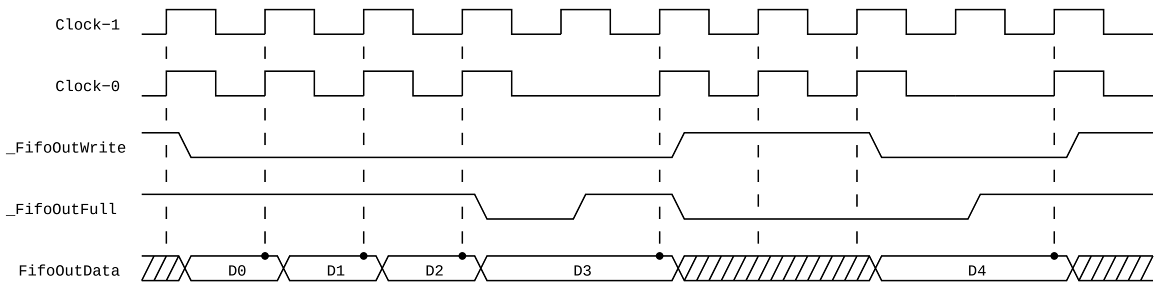
- When the design is put in *FifoIn-autostop* mode and the design is entirely clocked off *clock0*, if $\overline{FifoOutWrite}$ is asserted, the data present on the *FifoOutData* is pushed into the output FIFO (see Figure 4.10 (b)).

As shown in Figure 4.9 and 4.10, all operations of the input and output FIFOs occur at the raising edge of the *clock1* signal.

The input FIFO can be written and the output FIFO can be read by the driving software through the runtime library. The program running on the host computer can access the input and output FIFOs through the following FIFO operations:



(a) the output FIFO write operation (no autostop)



(b) the output FIFO write operation (autostop)

Figure 4.10: The output FIFO operation

WriteFifo: this operation pushes 32-bit data word into the input FIFO. The corresponding runtime library is *P1WriteFifo()*.

WriteFifoTagged: this operation pushes one 32-bit data word into the input FIFO and sets the tag bits. The corresponding runtime library *P1WriteFifoTagged()*.

ReadFifo: this operation reads one 32-bit data word from output FIFO. The corresponding runtime library is *P1ReadFifo()*.

WritePAM: this operation successively pushes a 19-bit address word and a 32-bit data word in the input FIFO; the address word is present on bits 5 to 23. The corresponding runtime library is *P1WritePAM()*.

ReadPAM: this operation successively pushes a 19-bit address word in the input FIFO and reads a 32-bit data word form the output FIFO; the address word is present on bits 5 to 23. The corresponding runtime library is *P1ReadPAM()*.

4.2.6.2 LCBus

The LCBus is a 24-bit-wide general purpose register that can be read and written by both the software and the application design. The LCBus can be used for asynchronous communication between the Controller LCAs and the software program. Under the software control, the direction of each bit can be set independently of the others. Initially (after download), all bits are set for PAM-to-Host communication. This feature is not used in our CCM design.

4.2.6.3 Tags

Every word that the software (the program running on the host) pushes into the input FIFO is "tagged" with 4-bit value. These tag bits are read from the input FIFO at the same time as the data word, and are available on both Controller LCAs and on the *Fifo Switch*. The meaning of these bits is as follows:

TagDataSource (TDS): When set, the word was pushed by a DMA[2] transfer operation.

TagDataType (TDT): This bit is only valid when *TagDataSource* is 0 (not set) . When valid and set, the word is a 19-bit address pushed by a *WritePAM* or *ReadPAM* transaction.

TagAddr0, TagAddr1 (TA0, TA1): these two bits can be defined by the user.

These tags are not used in our CCM design.

4.2.7 Performance

The main timing characteristics of the PeRLe-1 is shown in table 4.1. For a given design, the worst-case propagation delay can be determined by combining these timing characteristics and Xilinx chip time characteristics.

[2]DMA stands for *Direct Memory Access.* The DMA is a fast way of transferring data between memory and other devices (not CPU) in a computer system. For example, the DMA operation can be used to transfer data between memory and hard disk in a PC system.

Table 4.1: PeRLe-1 Timing Characteristics

Connection	Name	Delay
Matrix direct connection	DCnn-mmX	24 ns
Matrix bus I/O bus Switch ring Switch control Fifo switch control	MBusX DBusXX RingSwXX SwCntrX FswCntr	28 ns
Matrix ring	RingMat	43ns
Memory read, address to data	RamAddrX to RamDataX	46ns
Memory read, control to data	RamReadX to RamDataX	35ns
Memory write, from any source		33ns
Fifo data Tags	FifoInData FifoOutData TagAddr TagDataType TagDataSource	23 ns
Fifo Status	FifoInEmpty FifoOutFull	27 ns
Fifo Command	FifoInRead FifoOutWrite	30 ns

The goal of this table is to provide the further designers with data allowing the understanding of delays of different kinds of connections, so that they can make reasonable trade-off decisions for their designs. For instance, as shown in the table, the delay of matrix rings is 43ns, and the delay of matrix direct connection is 24ns. For a given signal, if we can use either matrix rings or matrix direct connection, the matrix direct connection should be a better choice.

4.2.8 The runtime library

The runtime library of PeRLe-1 is essential to the developer who develops the *driving program* which runs on the host computer and controls the PeRLe-1 hardware for the application. The runtime library is the only way to access PeRLe-1 hardware for the driving program. The runtime library developed by Digital Paris Research Laboratory provided a few essential controls to the application driving program:

- A UNIX I/O interface, with open, close, read and write.
- Download the configuration bitstreams from host to PeRLe-1 , and/or read back the values of all the flip-flops of all the LCAs.
- Read/write static RAM on PeRLe-1 by the software program.
- Control the mode and speed of PeRLe-1 clock by the software program.

4.3 Programming

For using PeRLe-1 board, we must run an application-specific program on the host computer which connects to the PeRLe-1 board. On the other hand, the 23 FPGA chips of the PeRLe-1 must be programmed to realize an application-specific hardware. Therefore, A PeRLe-1 program consists of two parts:

- the *driving program* which runs on the host and controls the PeRLe-1 hardware.
- A 1.5 MB bitstream which programs the 23 XC3090 FPGAs of the PeRLe-1 to realize an application-specific hardware.

The driving program is written in C or C++ and is linked to the runtime library encapsulating a device driver. The requirement for developing the driving program is the C or C++ programming environment and the PeRLe-1 runtime library.

For generating 1.5MB bitstream that programs the XC3090 FPGAs to realize application-specific hardware, the following steps are involved:

1. Design Partition

   In this step, you map your design onto 23 FPGA chips according to your design and the constraint of PeRLe-1 board. Some of FPGA chips may be not used in your design. For example, our CCM design uses only 17 FGPA chips of all 23 chips. The steps 2 and 3 should be carried out separately for each FPGA chip that is used in your design.

2. Design Entry

   In this step, you create your design using a Xilinx-supported schematic editor or hardware description language (like VHDL) for each FPGA used in your design separately. This step produces a Xilinx netlist file (XNF file) for the next step. There are three kinds of design entry methods:

   (a) Schematic editor: you can use schematic editor to create your design, then your schematic editor should be able to generate the XNF file.

   (b) Hardware description language: you can use VHDL or other hardware description language to create your design, then you need a synthesis software to synthesize and optimize your design and produce the XNF file.

   (c) Another possible way is to use a C++ program and the PerleDC library to describe your design. Individual configuration of each FPGAs involved in your design are described by this C++ program. Compiling and running this C++ program generates the XNF file of your design [9, 13, 14].

   There are two sets of tools available at EE of PSU as of this writing: *Xilinx Foundation Series* and *OrCAD Express 7.0*. Both of them support schematic editor and hardware description language. Due to the lack of license, the *Xilinx Foundation Series* only supports schematic editor. *Xilinx Foundation Series* was used to capture our CCM design. For more information about this software, please see [52] and online reference. I do not have experience with *OrCAD Express 7.0*. For more information about this software, please see the documentation shipped with the software and online reference.

3. Design Implementation

   Map, place and route your design, and finally generate the bitstream file by using Xilinx development tools.

Since all FPGAs used on DEC PeRLe-1 board are XC3090 FPGAs, we need a Xilinx development tools that support XC3090 FPGA. As of this writing, the latest Xilinx development tool, *Xilinx's Alliance Series Release Version 1.4* (also known as *M1 software*), is available at EE of PSU. Unfortunately, M1 software does not support XC3090 FPGA anymore since XC3090 FPGA is obsolete. M1 software does support two families of Xilinx 3000 series FPGA, XC3000A and XC3000L. XC3090 belongs to XC3000 family. Therefore, our CCM design are mapped to XC3090A chips. For more information about M1 software, please see [49, 50] and online reference.

4. Design Verification

   At this step, the bitstream generated at previous step is downloaded into the PeRLe-1 board and the design is tested. If something goes wrong, you may need to modify your design at design entry step, then regenerate the bitstream file, download it to PeRLe-1 board and test your design again.

CHAPTER 5

The Design of Cube Calculus Machine Co-processor

In our design, the cube calculus machine acts as a co-processor to the host computer, and it will be realized on the PeRLe-1 board. Therefore, the architecture of PeRLe-1 is our only design constraint.

The input FIFO and the output FIFO on PeRLe-1 allowed us to significantly simplify the communication between the host and the PeRLe-1 board. This feature lets the host and the PeRLe-1 board work asynchronously. Therefore, we use these two FIFOs as the way of communicating between the host and the CCM.

By using the input and output FIFOs, the communication between the host and the CCM is as follows. The host just puts instructions into the input FIFO, and receives the results from the output FIFO. On the other side, the CCM takes an instruction from the input FIFO, executes this instruction and puts the results back into the output FIFO. This is shown in Figure 5.1.

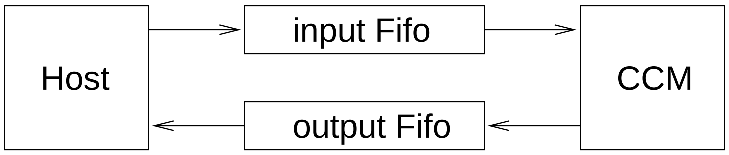


Figure 5.1: Communication between the host and the CCM

As described in Chapter 4, the width of FIFOs is 32 bits, which means the data transferred between the host and the CCM is 32-bit-wide. At this time, we want to keep the CCM as simple as possible, so we just use fixed-length instructions, and the width of all instructions will be 32 bits. This means that the opcode and the actual data of a CCM instruction are both included in one 32-bit-wide word.

5.1 Executing Patterns

Before we design the CCM instructions, we need to know what kind of execution patterns happen often in practical applications of the CCM, and how can our design be able to execute cube operations efficiently for these execution patterns.

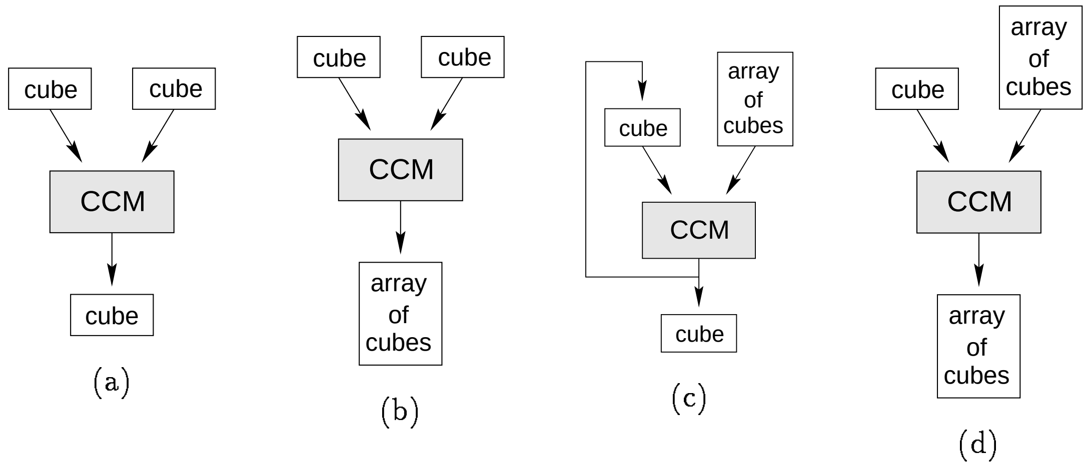


Figure 5.2: Cube operation patterns

We found four patterns based on the analysis of many algorithms in logic synthesis, such as satisfiability, tautology, complementation, solving of equations and others that must be speeded-up. These four basic patterns are shown in Figure 5.2. Many practical complex patterns can be created by repeating or combining these basic patterns.

Pattern (a) (Figure 5.2 (a)) is the general form of combinational cube operations. A combinational cube operation produces one resultant cube.

Pattern (b) (Figure 5.2 (b)) is the general form of sequential cube operations. A sequential cube operation produces as many as n resultant cubes, where n is the number of variables in the operand cubes.

Pattern (c) (Figure 5.2 (c)) is used in some combinational cube operations on an array of cubes, for example, the result of intersection operation on an array of cubes

$(A_1 \cdot A_2 \cdots A_n)$ is a single cube or an empty cube.

Pattern (d) (Figure 5.2 (d)) can be used both in combinational and sequential cube operations. A combinational operation example is a cofactor operation on an array of cubes:

$$\vec{C}|_A = (C_1|_A \; C_2|_A \dots C_n|_A)$$

A sequential operation example is a sharp operation on two arrays of cubes, and this is the most complicated case:

$$\begin{aligned}\vec{C} = \vec{A}\#\vec{B} &= (A_1 \; A_2 \dots A_m)\#(B_1 \; B_2 \dots B_n)\\ &= \Big(\big((A_1\#B_1 \; A_2\#B_1 \dots A_m\#B_1)\#B_2\big) \cdots \#B_n\Big)\\ &= \big((C_1^1\#B_2 \; C_2^1\#B_2 \dots C_k^1\#B_2) \cdots \#B_n\big)\\ &\;\;\vdots\end{aligned}$$

where $(C_1^1 \, C_2^1 \dots C_k^1)$ is the result of operation $(A_1\#B_1 \, A_2\#B_1 \dots A_m\#B_1)$. As we can see from the equation, the basic step for sharp operation on two arrays of cubes is the sharp operation on one array of cubes and one cube. This is what Pattern (d) describes. Therefore, the pattern of sharp operation on two arrays of cubes repeats pattern (d) as many times as the number of cubes in the array of cubes $\vec{B}$.

It can be seen from these execution patterns that the same cube operation is executed very many times before another kind of cube operation is executed in a practical application. Also, sometimes one operand cube does not change in subsequent operations or comes from the result of the previous cube operation. Thus, we have the following design considerations:

- We need an *accumulator* register for pattern (c), this *accumulator* can be set by the user or it receives the data being the result of a previous cube operation. As discussed in Chapter 2, most cube operations have two operand cubes. Thus, we need another *general* data register to store another operand cube.

- The CCM can execute cube operations by just accepting operand cube(s) without re-setting the *instruction* register.

5.2 The Design of the CCM

The block diagram of our design is shown in Figure 5.3. In this design, there are 5 data buses, 2 banks of memories, Global Control Unit (GCU), ILU and it's controller Operation Control Unit (OCU), two address units, registers, tri-state buffers and three multiplexers. The following section will discuss them in detail. The control signals are not shown in the figure, and they are all generated by GCU, which means that all components of the CCM work together under the control of the GCU.

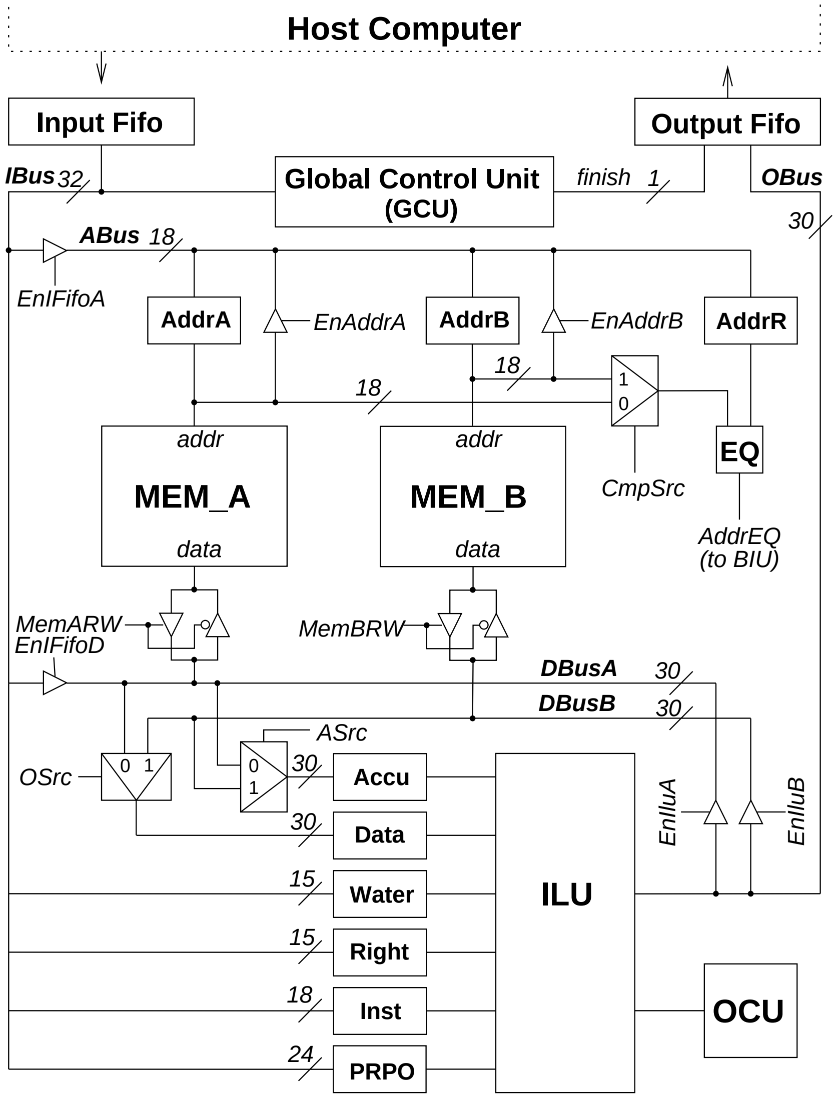


Figure 5.3: The Block Diagram of Our Design

5.2.1 Data Bus

Five data buses are used in the CCM. They are described as follows:

IBus is the short name of *input FIFO data bus.* The CCM receives the instruction from the input FIFO through this data bus. Only input FIFO can write this data bus.

OBus is the short form of *output FIFO data bus.* The CCM puts the results into output FIFO through this data bus. Only the ILU can write this data Bus.

ABus is the short name of *Address data bus.* The CCM sets the contents of two address units AddrA and AddrB and one address register AddrR through this data bus. The input FIFO and two address units can write this data bus, and they are controlled by three control signals: *EnIFifoA*, *EnAddrA* and *EnAddrB*, which control the corresponding tri-state buffers[1].

DBusA is the short name of *Data Bus A*. This data bus connects to the input FIFO, memory bank A (MEM_A) and the input and output of the ILU. The input FIFO, MEM_A and the ILU can write this data bus, and they are controlled by three control signals: *EnIFifoD*, *MemARW*, and *EnIluA*, which control the corresponding tri-state buffers.

DBusB is the short name of *Data Bus B*. This data bus connects to the memory bank B (MEM_B), the input and output of the ILU. The MEM_B and the ILU can write this data bus, and they are controlled by two control signals: *MemBRW* and *EnIluB*, which control the corresponding tri-state buffers.

The examples that show how to use these buses will be given in section 6.2.

5.2.2 Memory and Address Units

In this design, we use two banks of memory, MEM_A and MEM_B, to store intermediate results. Each bank of memory connects to one data bus: MEM_A connects to the *DBusA* and *MEM_B* connects to the *DBusB*.

[1]There are programmable tri-state buffer resources in Xilinx XC3090 FPGA

The address signals of MEM_A come from *Address Unit A* (*AddrA* for short). The address signals of MEM_B come from *Address Unit B* (*AddrB* for short). The contents of these two address units can be set or incremented under the control of GCU. These two address units are realized by 18-bit-wide loadable-up-counters. A *Address Register* (*AddrR* for short) is used to store an address data shown on *ABus*. As we mentioned before, *ABus* can be written by *IBus*, *AddrA* or *AddrB*, so the address data could be one of these three sources. The example of using *AddrR* is shown in section 6.2.

The control signal *MemARW* controls the MEM_A in read mode or write mode, which means the MemARW can read from or write to the data bus *DBusA*. When the *MemARW* is set, the MEM_A is in read mode, otherwise, the MEM_A is in write mode. The control signal *MemBRW* controls the MEM_B in the same manner.

5.2.3 Registers

There are six registers used by ILU in our design (*AddrR* register is mentioned in the above section). They are described as follows:

`Accu` is an *accumulator register* used to store one operand cube for the cube operation. It is 30 bits wide.

`Data` is a general *data register* used to store one operand cube for the cube operation. It is 30 bits wide.

`Water` is a 15 bits wide register used to store *water* signals.

`Rightedge` is a 15 bits wide register used to store *right_edge* signals.

`Inst` is a 21 bit wide register used to store cube operation instruction. The content of the *inst* register is shown in Figure 5.4. The meaning of these nine fields are as follows:

p1 field represents whether the first pre-relation/pre-operation is valid or not.

p2 field represents whether the second pre-relation/pre-operation is valid or not when $p1 = 1$.

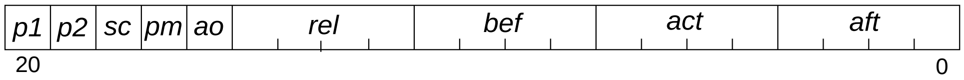


Figure 5.4: The content of *instruction* register

sc is sequential/combinational bit. When it is 1, the operation is a sequential operation, otherwise, the operation is combinational operation.

pm is prime bit. When it is 1, the operation is a complex combinational operation, otherwise, the operation is a simple combinational operation.

ao is *and_or* bit. When it is 1, the relation type of the operation is "AND", otherwise, the relation type is "OR".

rel, *bef*, *act* and *aft* are the four bitwise functions used to describe the operation (see Chapter 2 and 3).

PRPO is a 24 bits wide register used to store two pairs of pre-relation/pre-operation. The content of the *prpo* register is shown in Figure 5.5. The meaning of these eight fields are as follows:

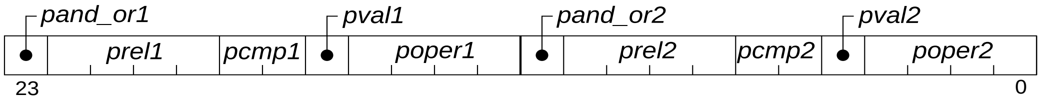


Figure 5.5: The content of *prpo* register

pand_or1, *prel1*, *pcmp1*, *pval1* and *poper1* are the partial pre-relation type, partial pre-relation, pre-relation compare type, pre-relation compare value and pre-operation for the first pair of pre-relation/pre-operation, respectively.

pand_or2, *prel2*, *pcmp2*, *pval2* and *poper2* are the partial pre-relation type, partial pre-relation, pre-relation compare type, pre-relation compare value and pre-operation for the second pair of pre-relation/pre-operation, respectively.

For the more information about pre-relation/pre-operation, please see section 3.7.

As shown in Figure 5.3, the input of these six registers is connected to either *DBusA* or *DBusB*. The signal *ASrc* controls to which bus the *Accu* is connected. The signal *OSrc* controls to which bus the *data* register is connected. Every register has a load signal used to load data from its inputs, and all load signals are generated by GCU (they are not shown in Figure 5.3).

There is one more register called *config* register (not shown in Figure 5.3), and it will be discussed in section 5.3.

5.2.4 Dataflow mode

A simplified block diagram of the CCM is shown in Figure 5.2.3 (a). It can be seen that the CCM has two data buses that connect the input/output FIFOs, memory and data path (ILU in the CCM) together.

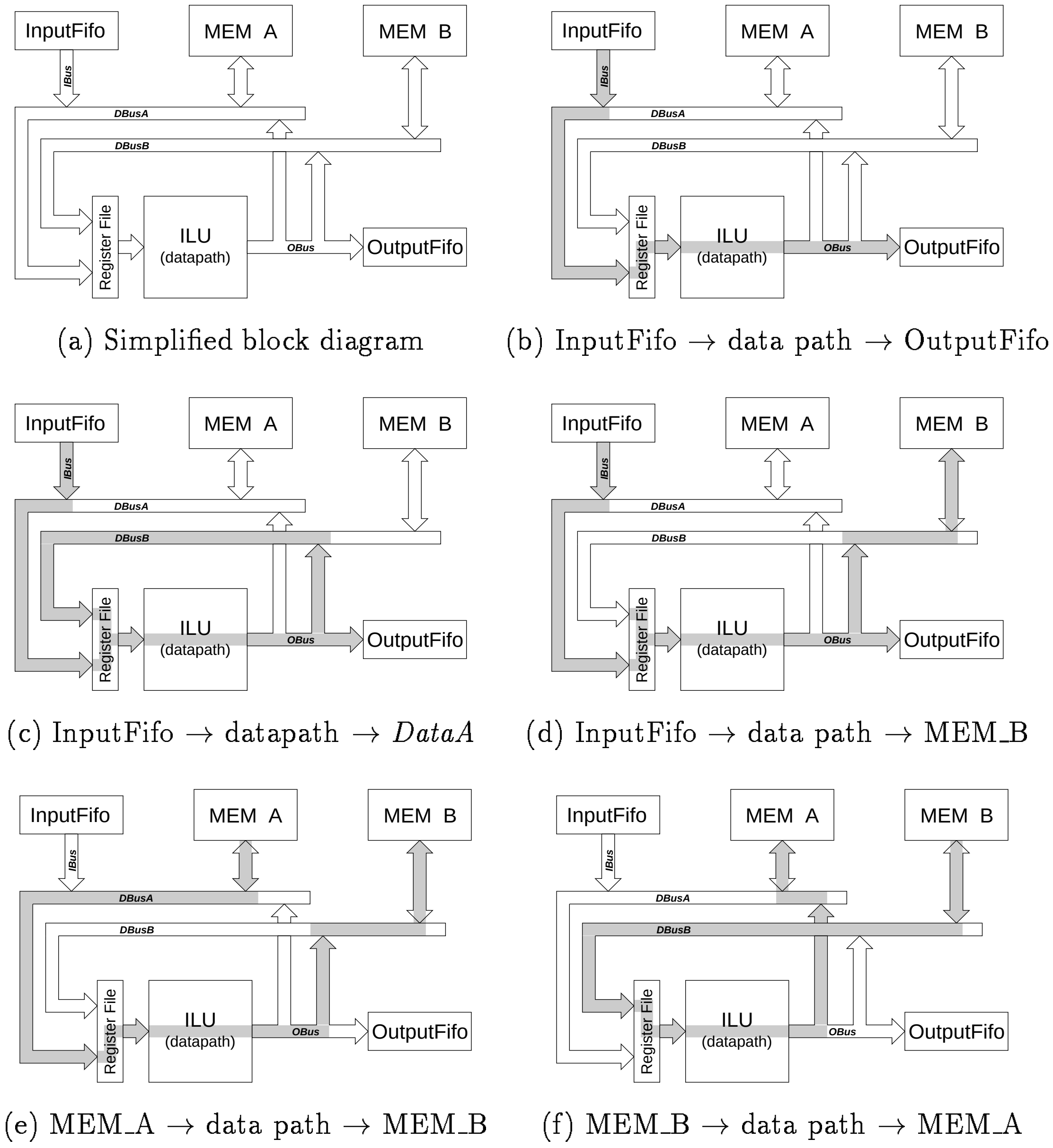
This two-bus structure has better performance than the single bus structure. Suppose we use single bus structure, which means that the input/output FIFOs, memory and ILU are connected together by one data bus. For a sequence of cube operations that read data from memory and write the results back to the memory, the algorithm would be:

```
1.  for (i=0; i++; i<n)
2.  {   set MEM be the writer and ILU be the reader of the data bus
3.      read data from MEM to ILU
4.      execute cube operation
5.      set MEM be the reader and ILU be the writer of the data bus
6.      write data from ILU to MEM
7.  }
```

It is easy to observe that the lines 2 through 6 are executed n times. With our two data buses structure, the algorithm is now changed to:

```
1.  set MEM be the writer and ILU be the reader of the data bus A
2.  set MEM be the reader and ILU be the writer of the data bus B
3.  for (i=0; i++; i<n)
4.  {   read data from MEM to ILU through data bus A
5.      execute cube operation
6.      write data from ILU to MEM through data bus B
7.  }
```

This time, the lines 1 and 2 are outside of loop and are only executed one time in our



(a) Simplified block diagram

(b) InputFifo → data path → OutputFifo

(c) InputFifo → datapath → *DataA*

(d) InputFifo → data path → MEM_B

(e) MEM_A → data path → MEM_B

(f) MEM_B → data path → MEM_A

Figure 5.6: The dataflow modes of the CCM

architecture. Therefore, we improve the performance by using two data buses. Some of useful dataflow modes are shown in Figure 5.2.3 (b) to (f). The examples of using these modes are given in Chapter 6.

5.3 Instructions and Their Encoding

The CCM has two categories of instructions called "CCM instructions", *config* instructions and *execute* instructions. The *config* instructions set the CCM to be ready to execute a specified cube operation. The *execute* instructions let the CCM executes cube operation(s) currently set in the *instruction* register.

There are three *config* instructions: *Set Accumulator*, *Set Tri-state Buffers* and *Set Registers*. And there are two *execute* instructions: *Execute* and *Loop*. This section will discuss these instructions and their encoding in detail (the examples are given in section 6.2).

5.3.1 Set Accumulator

The "set accumulator" instruction loads the data into the accumulator (*Accu*) from its input. The encoding of the instruction is as follows:

31	30	29 … 0
0	1	30-bit data

The first two bits "01" is the opcode of this CCM instruction. The *30-bit data* in the instruction will be shown on the bus *IBus*. For loading the correct data into *Accu*, the control bits *EnIFifoD* and *Asrc* (see Figure 5.3) must be set properly (by issuing *set tri-state buffers* and *set register* instructions) before issuing this instruction. For example, when *EnIFifoD* is 1 and *Asrc* is 0, this instruction will load the data shown on bits 29 to 0 of the instruction word into the *Accumulator*.

5.3.2 Set Tri-state Buffers

The "set tri-state buffers" instruction sets the control bits of tri-state buffers that control the data flow. Some useful dataflow modes are discussed in section 5.2.4.

There are 8 bits of this kind in our design.

These 8 bits are registered by an 8 bits register in the CCM, and this register can be set by one CCM instruction. As we discussed in section 5.2.1, the three data buses (ABus, DBusA and DBusB) have more than one possible driver, but at any given time, there is only one driver for each data bus. If there were more than one driver for a given bus at a given time, then the FPGA chip would be destroyed permanently.

For protecting the hardware from destroying by a "bad program", in our design, the *set tri-state buffers* instruction can only set one control bit a time, and a special circuit is used to check potential contention (multiple drivers). The idea of this special circuit is shown in Figure 5.7.

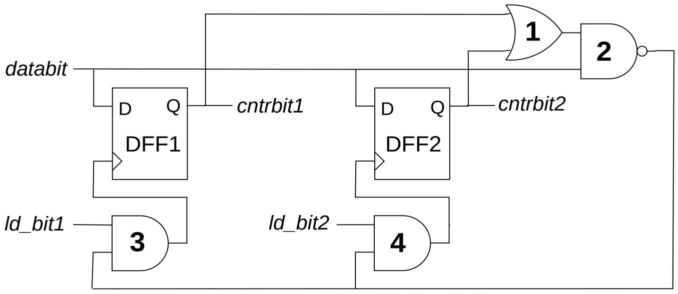


Figure 5.7: Avoiding contention which would result from multiple drivers

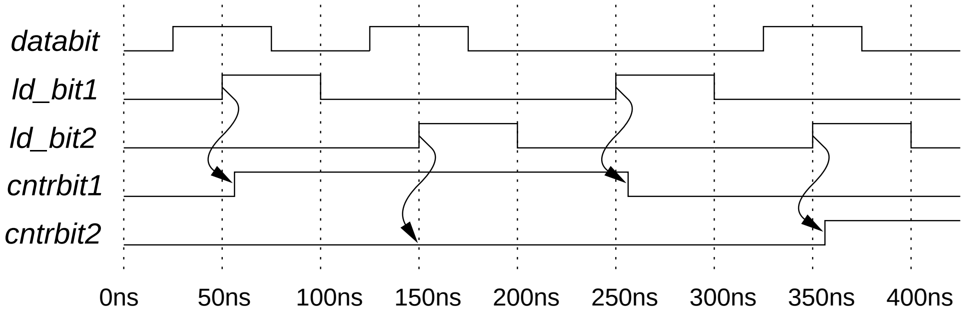


Figure 5.8: Timing diagram of special circuit for avoiding bus contention

As shown in Figure 5.7, two control bits (*cntrbit*1 and *cntrbit*2) control two tri-state buffers that drive one data bus (the tri-state buffer and data bus are not shown in the figure). At any given time, at most one control bit can be set to 1.

For understanding how this circuit works, let's see a timing diagram shown in Figure 5.8. At time point 0ns, all signals are 0. For setting *cntrbit*1 to 1, the signal *databit* is set to 1 first at 25ns, then there is a raising edge on the signal *ld_bit*1 at 50ns. As shown in the figure, after a little delay, the signal *cntrbit*1 is set to 1.

Now let us try to set *cntrbit*2 to 1 to create a bus contention. The signal *databit* is set to 1 at 125ns, then there is a raising edge on the signal *ld_bit*2 at 150ns. Since one of two control bits (*cntrbit*1) is 1, then both inputs of gate 2 (NAND gate) are 1's, thus, the output of gate 2 is 0. Then the raising edge on the signal *ld_bit*2 can not go through the gate 4 (AND gate), which means that the raising edge can not reach the clock input of DFF2 (D flip-flop), thus Q output of the DFF2 does not change. Therefore, this circuit ensures that at most one control bit can be set to 1.

For setting *cntrbit*2 to 1 at this time, two steps are needed. First step is to set *cntrbit*1 to 0 at 250ns (please note that the signal *databit* is set to 0 before the raising edge on the signal *ld_bit*1), then second step is to set *cntrbit*2 to 1 at 350ns. By using this kind of circuit, nothing happens when the CCM encounters a "bad instruction" that tries to create multiple drivers. This circuit can be described by the following VHDL code:

```
...
signal databit, ld_bit1, ld_bit2, cntrbit1, cntrbit2 : std_logic;
signal dff1clk, dff2clk, gate2output: std_logic;
...
begin
  ...
  DFF1: dff port map ( d=>databit, clk=>dff1clk, q=>cntrbit1);
  DFF2: dff port map ( d=>databit, clk=>dff2clk, q=>cntrbit2);

  gate2output <= not ((cntrbit1 or cntrbit2) and databit);
  dff1clk <= ld_bit1 and gate2output;
  dff2clk <= ld_bit2 and gate2output;
  ...
end;
```

The encoding of the "set tri-state buffers" instruction is as follows:

31	30	29	28	27	26	25	24 … 0
0	0	0	m	m	m	1/0	unused

1: Enable, 0: disable

The first three bits "000" is the opcode of this CCM instruction. The bit 25 is the *databit* signal in Figure 5.7. The bits 28 to 26 (`mmm` in the encoding format) is the "address" of these eight control bits of tri-state buffers. The address of these control bits is as follows: 000 is EnAddrA, 001 is EnAddrB, 010 is EnIFifoA, 011 is MemARW, 100 is EnIluA, 101 is EnIFifoD, 110 is MemBRW, and 111 is EnIluB (all these eight control bits are shown in Figure 5.3).

5.3.3 Set Registers

The "set registers" instruction loads the data into registers (except *Accu* and *Data*) from their inputs. For loading the correct data into registers, the tri-state buffers must be set properly before issuing this instruction. The encoding of the instruction is as follows:

31	30	29	28	27	26	25 … 0
0	0	1	m	m	m	data

The first three bits "001" is the opcode of this CCM instruction. The bits 28 to 26 (`mmm` in the encoding format) is the "address" of the target register. The addresses of these registers are as follows: 000 is AddrA, 001 is AddrB, 010 is AddrR, 011 is WATER, 100 is RightEdge, 101 is INST, 110 is CONF, and 111 is PRPO.

AddrA and AddrB are the *Address Units* (see section 5.2.2). Since they can be set in the same way we set registers, the same instruction is used to set *Address Units* and registers. AddrA, AddrB and AddrR are 18-bit wide, so bits 17 to 0 are used when the target register is one of them. WATER and RightEdge registers are 15-bit wide, so bits 14 to 0 are used when the target register is one of them.

When the target register is the *instruction* register, 21-bit data is needed. The bits 25 to 23 represent the highest 3 bits of the *instruction* register. The bits 17 to 0 represents the lowest 18 bits of the *instruction* register (see the format of *instruction* register discussed in section 5.2.3). When the target register is the PRPO register, the bits 23 to 0 are used (see the format of *prpo* register discussed in section 5.2.3).

When the target register is *config* register (CONF), the bits 8 to 0 of the instruction are used. The *config* register is the collection of eight configuration bits of the CCM. The content of *config* register is shown in Figure 5.9. The meaning of these eight bits

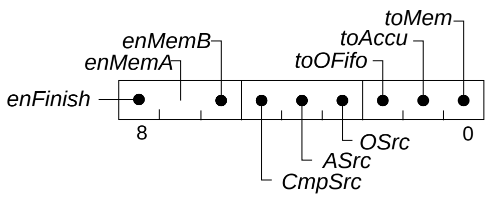


Figure 5.9: the format of *config* register

is as follows:

- *enFinish* determine whether the instructions *Execute* and *Loop* will generate "finish word" or not. The "finish word" will be discussed in section 5.3.4.

- *enMemA* and *enMemB* determine whether the memory banks are used in cube operation or not. Do not confuse with tri-state buffer control signals *memARW* and *memBRW*; the signals *memARW* and *memBRW* determine the operation mode (*read* or *write*) of memory banks (see section 5.2.2). If *enMemA* is set to 1, then the MEM_A is used in the following cube operation, otherwise MEM_A is not used. The bit *enMemB* controls MEM_B in the same manner. Only *Loop* instruction will use memory banks, and it will be discussed later in this section.

- *CmpSrc*, *ASrc* and *Osrc* are three "select" signals of the multiplexers (see Figure 5.3).

- *toOFifo*,*toAccu* and *toMem* are three output control signals. These three signals tell the GCU just whether or not to generate the corresponding control signals to load data into the output FIFO, and/or *Accu*, and/or the memory from their inputs after a resultant cube is generated. The GCU doesn't care about where these inputs come from. For using proper dataflow mode, the tri-state buffers must be set properly before executing the operation. It is possible to write the resultant cube to all these three targets at the same time.

A cube operation is completely described by functions *relation*, *before*, *active*, *after* and *pre-relation/pre-operation*. As it can be seen from this instruction, all these functions can be programmed by users through setting registers *inst* and *prpo* instead of "hard-code" values. This is similar to microprogramming and makes it easy to execute a "new" cube operation that is not discussed in this thesis and can be classified into one of three classes of cube operations without re-designing of the entire CCM. For example, the cofactor operation is a "new" operation to the CCM. I was asked by Dr. Perkowski to implement the cofactor operation on the CCM, which did not exist there yet. Therefore, I described the cofactor operation using Equation 3.6, and I derived $before$, *active* and $relation$ functions. The final equation used to describe cofactor operation is shown in Equation 2.11. Now we can perform cofactor operation without changing the design of the CCM. This is a powerful feature of our design and it is hoped that it will find many applications in CCM assembly programs.

5.3.4 Execute

The "execute" instruction is used to execute only one cube operation. It is the realization of the executing patterns (a) and (b) (see section 5.1). When the CCM receives this instruction, the CCM loads data into *Data* register from its input, then executes cube operation on two operand cubes currently stored in *Accu* and *Data* registers. The resultant cubes being written to *Accu*, and/or memory, and/or the output FIFO depending on three output control bits (The address of memory will be automatically increased by one after every memory write operation). The encoding of the instruction is as follows:

31	30	29 ... 0
1	0	30-bit data

The first two bits "10" is the opcode of this CCM instruction. The *30-bit data* in the instruction will be shown on the bus *IBus* after the instruction is read from the input FIFO. For loading the correct data into *Data* register, the control bits *EnIFifoD* and *Osrc* must be set properly before issuing this instruction.

The last step of executing *execute* instruction is that a "finish word" is pushed into the output FIFO if the bit *enFinish* is 1. The *finish word* is a special 32-bit word whose highest two bits are set to 10. On the other hand, the highest two bits of the general data word (represents cubes) are always 00. The finish word is used to separate two arrays of (resultant) cubes of two set of cube operations.

The finish word is necessary for our CCM co-processor. For example, we calculate a sharp operations on two arrays of cubes. The sharp operation is carried out by a set of CCM instructions, and produces an array of cubes. Without the finish word, the host computer would never know where is the end of the resultant cubes even if the host computer fetches all words in the output FIFO. The question is how the host computer could determine whether a operation (or a set of operations) is completed or not. Therefore, we introduce here the concept of finish word to solve this problem. We can let the CCM generate a finish word after the operation is completed. With the finish word, the host computer can tell whether a operation is already completed or not. Anther example of use of the finish word is when we calculate two sharp operations, each of them produces an array of cubes as the result. Without the finish word, the two resultant cubes would be concatenated together, so the host computer would be not not able to separate these two arrays of cubes. With the finish word, the host computer has a way to separate these two arrays of cubes.

5.3.5 Loop

The "loop" instruction is used to execute multiple cube operations continuously without fetching the input FIFO. It is the realization of the executing patterns (c) and (d) (see section 5.1). When the CCM receives *LOOP* instruction, the CCM loads the data from memory into *Data* register (The control signals *MemARW*, *MemBRW*, *EnIluA*, *EnIluB*, *OSrc* and *ASrc* must be set properly before issuing *LOOP* instruction). Then the CCM executes cube operation on two operand cubes currently stored in *Accu* and *Data* registers. The resultant cubes are written to *Accu*, and/or memory, and/or the output FIFO determined by three output control signals. After that, the CCM loads the next data from memory into *Data* register and executes the same cube operation again (The address of memory will be automatically increased by one after every memory read/write operation). This procedure is repeated until the signal *AddrEQ* (see Figure 5.3) becomes 1, which means that the memory address (the signal *Cmpsrc* determines which memory address is used, see Figure 5.3) is equal to the content of the *AddrR* register. The encoding of the instruction is as follows:

31	30	29 … 0
1	1	30-bit data

The first two bits "11" is the opcode of this CCM instruction. The *30-bit data* in the instruction will be shown on the bus *IBus*, but typically it is not used.

Similarly as in the *execute* instruction, the last step of executing the *loop* instruction is also that a finish word is pushed into the output FIFO if the bit *enFinish* is 1.

5.4 Global Control Unit

The Global Control Unit (GCU) handles the communication between the host computer and the CCM, and it is also the controller of the whole CCM. As mentioned in section 3.5, another controller OCU is used to control the datapath of the CCM. Certainly, we can design a single controller to control all of them. The reason why we design two controllers in our CCM is that it is easier to design and test two simple controllers than one complex controller.

The algorithm of the CCM is very simple: under the control of GCU, the CCM fetches an instruction from the input FIFO; then the CCM executes the instruction: set the contents of registers, or tri-state buffers, or executes cube operation(s). After that, the CCM is ready to process next instruction from the input FIFO. The GCU will remove an empty cube by using signal *empty* (see section 3.4.2). The state diagram of the GCU is shown in Figure 5.10. The signals *opc* in the state diagram is the opcode the CCM instruction, and it occupies the highest 3 bits of the CCM instruction. The other signals will be discussed when we will discuss the related states. Now let us take a look at these states.

- State S0: This is the initial state of the GCU. In this state, the GCU checks if there are CCM instruction(s) in the input FIFO by asserting signal *FifoInEmpty* (see section 4.2.6). If there are CCM instruction(s) in the input FIFO, then the GCU goes to state S1, otherwise, the GCU stays in state S0.

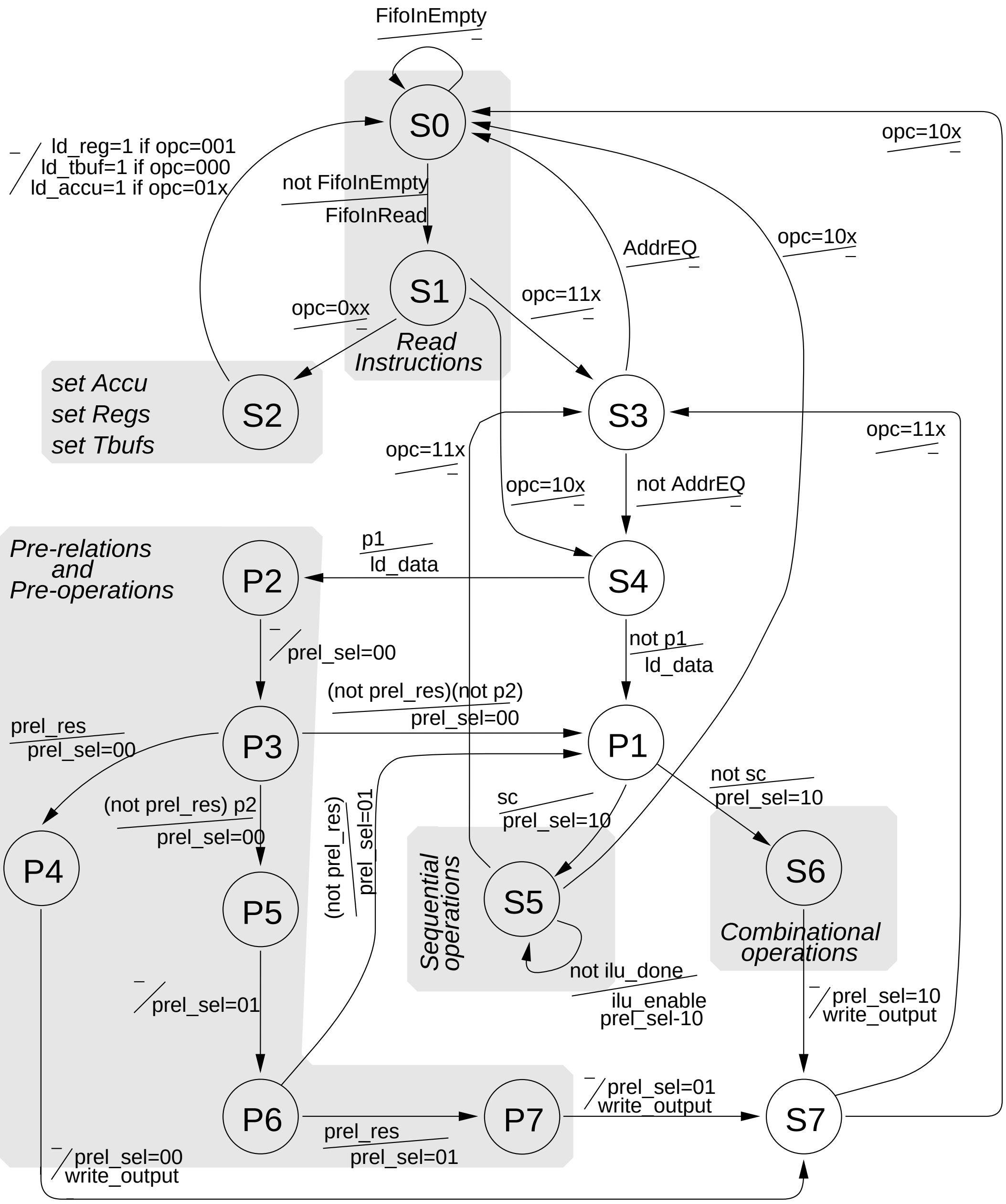


Figure 5.10: The state diagram of the GCU

- State S1: In this state, the GCU generates signal *FifoInRead* (see section 4.2.6) to fetch a CCM instruction from the input FIFO. This state has three exits, states S2, S3 and S4. If the instruction is one of instructions "set accumulator", "set tri-state buffers" or "set registers" (the highest bit of opcodes of these three instructions all are 0's), then the GCU goes to state S2; if the instruction is the instruction "exec" (the corresponding opcode is 10x), then the GCU goes to state S4; if the instruction is the "loop" instruction (the corresponding opcode is 11x), the GCU goes to state S3.

- State S2: In this state, the GCU generates the load signals to load data into the corresponding register, or the accumulator, or the 1-bit register that store the control bit of the corresponding tri-state buffer. For example, if the instruction is "set instruction register", then the GCU generates signal *ld_reg*, and the signal *ld_inst* that is used to load data into the instruction register is generated as follows:

  $$ld\_inst = ld\_reg \cdot b_{28} \cdot \overline{b_{27}} \cdot b_{26}$$

  (please note that the instruction register is encoded as 101, see section 5.3.3). where b_{28}, b_{27} and b_{26} are the 28th to 26th bit of the CCM instruction. Load signals for other registers and 8 1-bits registers for 8 tri-state buffers' control bits are generated similarly. When the instruction is "set accumulator", the GCU generates signal *ld_accu*.

  After that, the GCU always goes back to state S0 and becomes ready to process the next CCM instruction.

- State S3: In this state, the GCU checks if the loop operation is completed by asserting signal *AddrEQ* (see Figure 5.3 and section 5.3.5). If the loop operation is completed, then the GCU goes back to state S0 and becomes ready to process the next CCM instruction; otherwise, the GCU goes to state S4.

- State S4: In this state, the GCU generates signal *ld_data* to loads the data into the *data* register from its input. If the CCM is executing "exec" instruction, the input of the *data* register should come from the *IBus*, otherwise (the CCM is executing "loop" instruction), the input of the *data* register should come from one of the two memory banks. This state has two exits, states P2 and P1. If the first pre-relation/pre-operation is used in the operation (represented by *p1* field of the instruction register, see section 5.2.3), then the GCU goes to state P2, otherwise, the GCU goes to state P1.

- State P2: In this state, the GCU sets the signal *prel_sel* to 00. After that, the pre-relation/pre-operation circuitry begins to evaluate the first pre-relation (see section 3.7). The GCU always goes to state P3 from state P2.

- State P3: In this state, the GCU still keeps the signal *prel_sel* to be 00, and checks if the first pre-relation is satisfied by asserting signal *prel_res* (see section 3.7). If the first pre-relation is satisfied (the signal *prel_res* is 1), then the GCU goes to state P4; otherwise, the GCU will check if the second pre-relation/pre-operation is used in the operation (represented by *p2* field of the instruction register, see section 5.2.3). If the second pre-relation/pre-operation is used, then the GCU goes to state P5; otherwise, the GCU goes to state P1.

- State P4: Achieving this state means that the first pre-relation is satisfied. The GCU calculates the cube operation by using the first pre-operation function (the signal *pre_sel* keeps 00), and the GCU will generate signal *write_output*. By using signal *write_output*, the proper write signals can be generated to write the resultant cube to *Accu*, and/or memory, and/or the output FIFO depending on three output control bits if the *empty* signal is 0 in this state. For example, the signal *write_fifo* that writes the result to the output FIFO is generated as follows:

$$write\_fifo = write\_output \cdot toOFifo \cdot \overline{empty}$$

where the signal *toOfifo* is discussed in section 5.3.3, and the signal *empty* is discussed in section 3.4. The GCU always goes to state S7 from state P4.

- State P5: This state is similar to state P2, and the difference is that the GCU will check the second pre-relation rather than check the first pre-relation. The signal *prel_sel* is set to 01 in this state. The GCU always goes to state P6 from this state.

- State P6: In this state, the GCU still keeps the signal *prel_sel* to be 01, and checks if the second pre-relation is satisfied by asserting signal *prel_res* (see section 3.5). If the second pre-relation is satisfied, then the GCU goes to state P7; otherwise, the GCU will go to state P1.

- State P7: Achieving this state means that the second pre-relation has been satisfied. The GCU calculates the cube operation by using the second pre-operation function (the signal *pre_sel* keeps 01), and the GCU will generate proper write signals (in the same way with state P4) to write the resultant cube to *Accu*, and/or memory, and/or the output FIFO depending on three output control bits if the *empty* signal is 0 in this state. The GCU always goes to state S7 from state P7.

- State P1: This state means that the cube operation that is executed on the CCM does not have the pre-relation/pre-operation or these pre-relations are not satisfied. In the state P1, state S5, and state S6, the signal *prel_sel* is set to 10, which means the cube operation will be carried out by using relation/operation specified by the *instruction* register. State P1 has two exits, states S5 and S6. If the cube operation is a sequential operation (The *sc* field of the *instruction* register represents whether a operation is sequential or combinational, see section 5.2.3), then the GCU goes to state S5; otherwise (combinational operation), the GCU goes to state S6.

- State S5: This state means that the operation is a sequential cube operation. In this state, the GCU will generate signal *ilu_enable* which enables the ILU to execute sequential operation under the control of OCU (the control unit of the ILU). After the operation is done, the OCU will generate signal *ilu_done* to tell the GCU that the cube operation is done. The GCU will keep checking the signal *ilu_done* to see if the cube operation is done. If not, the GCU will remain in state S5; otherwise (the operation is done), the GCU will check if this is a loop operation by asserting the opcode of the current CCM instruction on the *IBus*. If this is a loop operation ($opc = 11x$), then the GCU will go to state S3, otherwise, the GCU goes back to state S0 and becomes ready to process the next CCM instruction.

- State S6: This state means that the operation is a combinational cube operation. In this state, the signal *ilu_enable* keeps 0, which means that all ITs in the ILU will remain in *before* states, or goes to *active* states if the current cube operation is a complex combinational cube operation (the *pm* field of the *instruction* register represents whether the operation is complex or not, see section 5.2.3) and the given IT is a special variable[2] (or part of special variable). The ILU executes this (complex) combinational cube operation by using *before* and *active* functions (see section 2.3.1 and 2.3.2).

  The GCU will generate proper write signals (in the same way with state P4) to write the resultant cube to *Accu*, and/or memory, and/or the output FIFO depending on three output control bits if the *empty* signal is 0 in this state. The GCU always goes to state S7 from state S6.

- State S7: In this state, the GCU will check if this is a loop operation by asserting the opcode of the current CCM instruction on the *IBus*. If this is a loop operation ($opc = 11x$), then the GCU will go to state S3, otherwise, the GCU goes to state S0 and becomes ready to process the next CCM instruction.

  In this state, the GCU also adjusts the address of the memory if the CCM read the data from and/or write the data to the memory bank(s).

5.5 Mapping CCM onto PeRLe-1 board

This design of the CCM is mapped onto DEC PeRLe-1 board, and the mapping is described by VHDL codes. Since the mapping detail is too tedious to list here, we only give the outline of idea in this section.

The outline of mapping is shown in Figure 5.11. As shown in the figure, for using Matrix as regular as possible, only ITs are mapped in it; The GCU is mapped in

[2]Do not confuse states *before* and *active* with the states of the GCU, these two states are the states of the state machine inside IT.

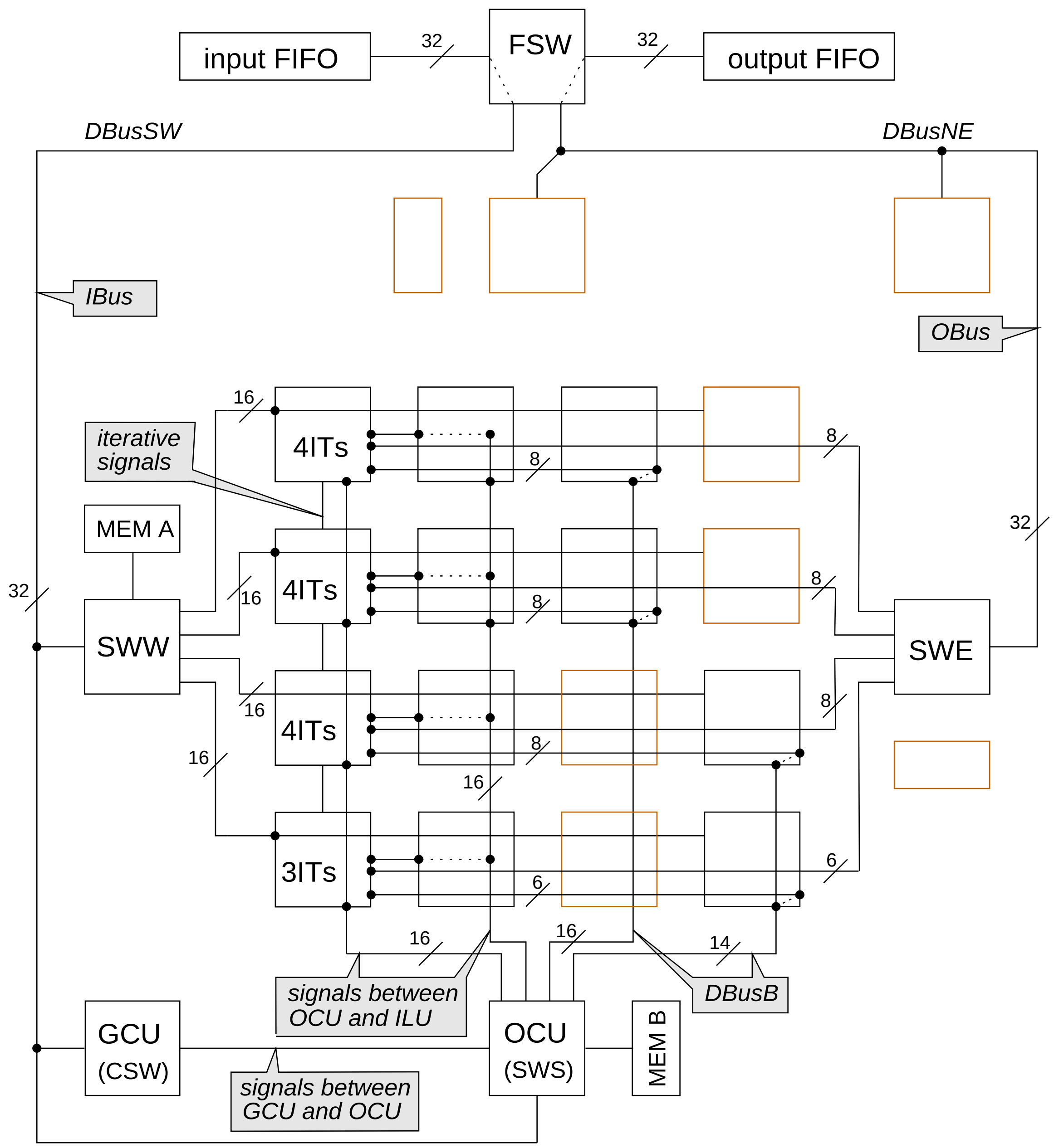


Figure 5.11: The outline of mapping

South-West controller (*CSW*) because it is easy to control input and output FIFOs from the controller, OCU is mapped in *South Switch* (*SWS*) because it is easy to control all matrix chips from the switches. All other parts are shown in the figure. The components of PeRLe-1 that are represented by dotted lines are not used in this design. The following section will discuss how we derived this mapping.

One important mapping principle is that the mapping should be as simple as possible, which means the designer should try to use direct connections as more as possible.

In my design, the input FIFO connects the bus *IBus*, and the output FIFO connects to the bus *OBus*. Both these two buses are 32-bit wide. Since the PeRLe-1 has two 32-bit-wide I/O buses: *DBusSW* and *DBusNE*, the bus *IBus* is mapped to *DBusSW*, and the bus *OBus* is mapped to *DBusNE*. It is very easy to connect the input FIFO to *DBusSW* and the output FIFO to *DBusNE* through *Fifo Switch* (FSW).

Since the GCU is connected to the bus *IBus*, and it should be able to control the input and output FIFOs directly, it should be mapped to a FPGA chip that connects to *IBus* and the control signals of the input and output FIFOs directly[3]. In the PeRLe-1 , the only FPGA chip that meets this requirement is *South-West Controller* (CSW). Therefore, the GCU is mapped into CSW.

Our design needs two memory banks. It can be seen from Chapter 4 that the address signals of the memory banks are connected to two controller FPGAs CSW and CNE. In our design, the two *Address Units* and *Address Register* should be able to copy their 18-bit-wide contents to each other, which means they should reside in one FPGA chip; and they are also connected to the bus *IBus*. So I mapped these two *Address Units* in CSW. This means also that we use the west and south memory banks to map two memory banks: MEM_A and MEM_B.

The ILU, the data path of our CCM, is mapped into the matrix of PeRLe-1 since the matrix is the only place that is large enough to hold the ILU. In the previous

[3]We can map GCU to a FPGA chip that does not connect to *IBus* and the control signals of the input and output FIFOs directly, but it will take more time to transfer signal between them, see section 4.2.7.

experiment [17], David W. Foote mapped four ITs in one Xilinx 3090 FPGA chip, which took about 50% CLB resources of the FPGA. In our design, more functions are added to one IT, like COUNT and EMPTY blocks; and I still want to reserve some resource for the future modifications. Thus, four ITs are mapped into one matrix FPGA chip in our mapping, which means we need 4 matrix chips to map 15 ITs. We can use any 4 FPGAs inside matrix to map the ILU, but we want to keep the structure as regular as possible. So I decided to use one column or one row of the matrix to map the ILU. The bus *DBusA* is connected to the MEM_A and the bus *IBus*. Suppose MEM_A is mapped to west memory bank[4], so the bus *DBusA* is routed into the matrix from the west side. There are only 16 connections from *West Switch* to any row of the matrix. The width of *DBusA* is 30-bit, so we use the first column FPGAs of the matrix to map ILU that has 15 ITs.

The registers (*Accu, Data, Water, Right, Inst* and *PRPO* are also mapped in the first column of the matrix because that there is not enough direct connections between the matrix and the west/south switches if these registers are mapped into the *west/south switches*.

The last unmapped component is OCU. The OCU is connected to every ITs in the ILU and the GCU. The only FPGA chip that has direct connection to the first column FPGAs of matrix and the GCU (mapped in CSW) is *South Switch* (SWS). Therefore, the OCU is mapped into SWS.

[4]There exists another choice of mapping MEM_A to the south memory bank. These two mapping choices are symmetrical.

CHAPTER 6

CCM Assembly

The programs written for the CCM are in the form of a list of binary bytes which are CCM instructions encoded by the programmer. This method of programming is very tedious and error prone. All instruction encoding formats had to be remembered in order to use them, and it is painful to look at and maintain this kind of programs. Therefore, a very simple assembly language, called *CCM assembly* was created. With the CCM assembly, the programmer needs only to remember the name of the CCM instructions. This programming methodology still requires the programmer to think in terms of registers and individual instructions. This chapter will describe the CCM assembly and will give some examples.

6.1 CCM Assembly

In the CCM assembly, one instruction has two or three fields and occupies one line; the fields are separated by blank spaces; and the comments can be added after the last field of the instruction. The names of the instructions, called *mnemonics*, occupy the first field in an instruction line. The subsequent fields are the operands of the instruction.

In the syntax of CCM assembly, keywords are represented in upper case, operands are represented in lower case, and they should be substituted by actual operands when they are used.

There are four instructions in the CCM assembly, and their corresponding instructions are discussed in detail in section section 5.3. The syntax of them is described as follows:

- `ENABLE/DISABLE`: The `ENABLE/DISABLE` instructions are used to enable/disable tri-state buffers in the CCM. The corresponding CCM instruction is *set tri-state buffers*. The syntax is as follows:

  ```
  ENABLE control_signal_name
  ```

  and

  ```
  DISABLE control_signal_name
  ```

  where `control_signal_name` is one of `ENADDRA`, `ENADDRB`, `ENIFIFOA`, `ENIFIFOD`, `ENMEMAWR`, `ENMEMBRW`, `ENILUA` and `ENILUB` (see section section 5.3.2).

- `SET`: The `SET` instructions are used to load the data into registers and address units from their inputs. The corresponding CCM instructions are *set accumulator* and *set registers*. The syntax is as follows:

  ```
  SET register_name, operand
  ```

  where `register_name` is one of `ADDRA`, `ADDRB`, `ADDRR`, `WATER`, `RIGHT`, `INST`, `ACCU`, `CONF`, `PRPO`; the `operand` in the syntax is a binary number that will show up on *IBus*, the width of this number depends on the register of the instruction (see section section 5.3.3).

- `EXEC`: The `EXEC` instruction is used to execute only one cube operation. The corresponding CCM instruction is *execute*. The syntax is as follows:

  ```
  EXEC operand
  ```

  where `operand` is a 30-bit-wide binary data that will show up on *IBus* when the instruction is executed (see section section 5.3.4).

- `LOOP`: The `LOOP` instruction is used to execute multiple cube operations continuously without fetching the input FIFO. The corresponding CCM instruction is *loop*. The syntax is as follows:

  ```
  LOOP operand
  ```

  where `operand` is a 30-bit-wide binary number that will show up on *IBus* when the instruction is executed (see section section 5.3.5).

Two more efforts are made to make the CCM assembly program easy to understand:

- The symbol slash ('-') can be inserted into binary number. The program that interprets the CCM assembly should ignore these slashes.

  Example 6.1. The functions of the following two instructions are identical.

  ```
  set inst 000000000000100000000
  set inst 00-0-0-0-0000-0001-0000-0000
  ```

  It is obvious that the second instruction is easier to understand.

- The unused bits at the end of a binary number can be omitted. The program that interprets the CCM assembly should fill these bits with 0's or 1's (only the omitted bits of the binary number, that is the operand of "`set water`" instruction, will be filled with 1's, see the definition of *water* signal in section section 3.4.1).

  Example 6.2. The functions of the following two instructions are identical.

  ```
  set water 000011111111111
  set water 0000
  ```

  Both of these two instructions mean that only first 4 ITs are used; and it is obvious that the second instruction is easier to understand.

  Example 6.3. The functions of the following two instructions are identical.

  ```
  set right 111100000000000
  set right 1111
  ```

The CCM assembly is a very simple assembly language. It is easy to develop an interpreter for the CCM assembly, this CCM interpreter accepts CCM assembly programs as its input, then executes CCM instructions by calling proper PeRLe-1 runtime library routines, and passes the result back to the host program.

For making the CCM easy to use, the CCM runtime library, a set of library calls which can be called from C/C++ programs, need to be developed by the next student in the CCM project group. For example, this library should have a routine (function) to carry out sharp operation on two arrays of cubes. The CCM runtime library can hide unnecessary details about the CCM hardware from the programmer, and enables programmers to think at a higher level and develop applications more efficiently.

6.2 Examples of Using CCM Assembly

This section presents several examples of solving some cube operation problems in CCM assembly. These examples serve as a tutorial about how to use the CCM to solve the problems. All these programs have an assumption that the CCM is reset, which means that all registers and control signals of tri-state buffers are zeroed.

Example 6.4. Assuming two cubes $A = ab$ and $B = b\bar{c}$, where a, b and c are binary variables. Write a CCM assembly program to calculate the intersection of cubes A and B.

Solution. The intersection operation is a simple combinational cube operation, it does not have *pre-relation/pre-operation* (see section 3.5), and

$$rel = \text{xxxx}, \quad and\_or = \text{x}, \quad bef = 0001$$

Since we process cubes with 3 binary variables, we have

$$water = \text{ 000-1111-1111-1111}, \quad rightedge = \text{ 111-xxxx-xxxx-xxxx}$$

Cubes A and B can be described in positional notation as:

$$A = ab \rightarrow \text{01-01-11}, \quad B = b\bar{c} \rightarrow \text{11-01-10}$$

Therefore, the program is as follows:

```
1. enable    enififod
2. set conf  100000100
3. set water 000
4. set right 111
5. set inst  00-0-0-0-0000-0001-0000-0000  ;intersection
6. set accu  01-01-11                      ; cube A
7. exec      11-01-10                      ; cube B
```

Line numbers are not part of the CCM assembly, they are used here to help identify specific lines of code in our discussion. Everything to the right of the semi-colon ";" are the comments.

Line 1 enables the tri-state buffer from the bus *IBus* to the *DBusA*, which means the data existing on the *IBus* exist also on the bus *DBusA* at the same time. Line 2 sets $ASrc = OSrc = 0$, $toOFifo = 1$ and $enFinish = 1$, which means that the inputs of registers *accu* and *data* are connected to the bus *DBusA* (see Figure 5.3), and the CCM will write the result(s) to the output FIFO. This instruction means also that the CCM is enabled to generate finish word. In this example, we don't care about the other bits of the *config* register.

Lines 3 to 5 are very straightforward. Line 6 lets the CCM load the cube A into the *accumulator*. Line 7 lets the CCM load the cube B into the *data* register and execute the cube operation (defined by *inst* and *prpo* registers) on the operand cubes (stored in the *Accu* and the *Data* registers). Please note that the CCM does not know (or does not care) the configuration of the datapath, it is the programmer's responsibility to set *config* register and the control bits of tri-state buffers correctly before issuing the EXEC command. In this example, these signals are set in Lines 1 and 2 of above program.

This is an example of executing pattern (a) (see section section 5.1), and the dataflow mode used by this example is shown in 5.2 (b) (see section section 5.2.4).

Example 6.5. Assuming four cubes $A = ac$, $B = ad$, $C = bd$ and $D = cd$, where a, b, c and d are binary variables. Write a CCM assembly program to calculate the intersection of these four cubes: $A \cdot B \cdot C \cdot D$ (Try to use as few instructions as possible).

Solution. The program is as follows:

```
1. enable    enififod
2. set water 0000
3. set right 1111
4. set inst  00-0-0-0-0000-0001-0000-0000 ; intersection operation
5. set accu  01-11-01-11                  ; cube A
6. set conf  000010010                    ; sent result back to ACCU
7. enable    enIluB
8. exec      01-11-11-01                  ; cube B
9. exec      11-01-11-01                  ; cube C
10. set conf 100000100
11. disable  enIluB
12. exec     11-11-01-01                  ; cube D
```

As mentioned before, there is an assumption that the CCM is reset, which means the *config* register is set to 000000000. Lines 2 to 5 set the registers *water*, *right_edge*, *inst* and *accu*. Lines 6 and 7 set a feedback path from the output of the ILU to the input of the *Accu* and let the CCM write the results back to the *Accu*.

Line 8 let the CCM calculate $A \cdot B$ and write the result back to *Accu*. Line 9 let the CCM calculate $[Accu] \cdot C$ and write the result back to *Accu*, where $[Accu]$ represents the content of the *Accu*. At this time, the content of *Accu* is the intersection of cubes A, B and C.

Line 10 let the CCM write the result to the output FIFO; and enables the CCM to generate "finish word". Line 11 breaks the feedback path created by Lines 6 and 7. Line 12 let the CCM calculate $[Accu] \cdot D$ and write the result to the output FIFO.

This is an example of executing pattern (c) (see section 5.1), and the dataflow mode used by this example is shown in 5.2 (c) (see section 5.2.4).

Example 6.6. Let us assume two cubes $A = \bar{c}$ and $B = bd$, where a, b, c and d are binary variables. We present a a CCM assembly program to calculate the basic sharp of cubes A and B: $A \#_{basic} B$ (Cube A is stored in the *Accu* and cube B is stored in the *data* register).

Solution. The program is as follows:

```
1. enable    enififod
2. set conf  100000100
3. set water 0000
4. set right 1111
5. set inst  00-1-0-0-0010-0011-0010-0011 ; basic sharp [Accu]#[Data]
6. set accu  11-11-10-11                  ; cube A
7. exec      11-01-11-01                  ; cube B
```

This example is the same as Example 6.4 except that it uses a different operation on different operand cubes. This is an example of executing pattern (b) (see section 5.1), and the dataflow mode used by this example is shown in 5.2 (b) (see section 5.2.4).

Example 6.7. Write a program in the CCM assembly to calculate the basic sharp operation of two cubes A and B: $B \#_{basic} A$, where cubes A and B are the same as in the previous example (Again, cube A is stored in the *Accu*, and cube B is stored in the *data* register).

Solution. It can be seen from the definition of sharp that $(A \#_{basic} B) \neq (B \#_{basic} A)$, and $B \#_{basic} A$ is not listed in Table 3.1. This is a "new" cube operation. This operation is very useful to execute the sharp operation on an array of cubes and a cube.

The functions rel, bef, act and aft are 2 inputs Boolean function $f(a_i, b_i)$. The 4 output values of each function are corresponding to minterms $\bar{a}_i\bar{b}_i$, $\bar{a}_i b_i$, $a_i\bar{b}_i$ and $a_i b_i$, respectively, where a_i comes from operand cube stored in *Accu*, and b_i comes from operand cube stored in *data* register.

Now, we want to perform sharp operation $[Data]\#_{basic}[Accu]$ (where $[Data]$, $[Accu]$ represent the contents of *data* and *Accu* registers, respectively), therefore, we have to substitute a_i with b_i and b_i with a_i in function $f(a_i, b_i)$ in order to obtain function $f(b_i, a_i)$. Its minterms are (b_i is the most significant bit now): $\bar{b}_i\bar{a}_i$, $\bar{b}_i a_i$, $b_i\bar{a}_i$ and $b_i a_i$, respectively. We have to use the format of $f(a_i, b_i)$ to represent $f(b_i, a_i)$ in the instruction, and we can obtain it by swapping minterms $\bar{b}a$ and $b\bar{a}$. Therefore, we just swap second and third output values of the functions rel, bef, act and aft. Therefore, only Line 5 of previous example needs to be changed. The whole program is as follows:

```
1. enable    enififod
2. set conf  100000100
3. set water 0000
4. set right 1111
5. set inst  00-1-0-0-0100-0101-0100-0101 ; basic sharp [Data]#[Accu]
6. set accu  11-11-10-11                  ; cube A
7. exec      11-01-11-01                  ; cube B
```

Example 6.8. Write a program in the CCM assembly to calculate basic sharp operation: $\vec{B} \#_{basic} A = (B_1\, B_2\, B_3) \# A = (a\, b, c) \# bd$, where B is an array of 3 cubes, A is a cube, and a, b, c and d are binary variables (Try to use as few instructions as possible).

Solution. The program is as follows:

```
1. enable    enififod
2. set conf  000000100
3. set water 0000
4. set right 1111
5. set inst  00-1-0-0-0100-0101-0100-0101 ; basic sharp [Data]#[Accu]
6. set accu  11-11-10-11                  ; cube A
7. exec 01-11-11-11                       ; cube B1
8. exec 11-01-11-11                       ; cube B2
9. set conf  100000100                    : generate finish word
10. exec     11-11-01-11                  ; cube B3
```

This is an example of executing pattern (d) (see section 5.1).

Example 6.9. Write a program in the CCM assembly to calculate disjoint sharp operation: $A \,\#d\, \vec{B} = A \,\#d\, (B_1 + B_2 + B_3) = 1 \,\#\, (ab + \bar{a}c + \bar{b}\bar{c})$ where A is a cube, $\vec{B}$ is a array of cubes, and a, b and c are three binary variables. Please note that this example shows how to use the loop instruction.

Solution. The program is as follows:

```
set conf  000000000
enable    enififoa
set addrb 0
disable   enififoa
enable    enififod
set water 000
set right 111
set prpo  1-1110-10-0-0101--0-0100-01-0-0000 ; disjoint sharp [D]#d[A]
set inst  11-1-0-0-0100-0101-0100-0001       ; disjoint sharp [D]#d[A]
set accu  01-01-11                           ; cube B1
set conf  001000001                          ; write result to MEM_B
enable    enIluB
disable   MemBRW
exec      11-11-11                           ; cube A1
disable   enIluB
enable    enaddrb                            ; [AddrB] => [AddrR]
set addrr 0
disable   enaddrb
enable    enIFifoA
set addrb 0
disable   enIFifoA
set accu  10-11-01                           ; cube B2
disable   enIFifoD
set conf  011101001                          ; memB=>ILU=>memA
enable    MemBRW
disable   MemARW
enable    enIluA
loop 0
disable   MemBRW
disable   enIluA
enable    enaddra                            ; [AddrA] => [AddrR]
set addrr 0
disable   enaddra
enable    enIFifoA
set addra 0
disable   enIFifoA
set conf  000000000
enable    enIFifoD
set accu  11-10-10                           ; cube B3
```

```
40. disable   enIFifoD
41. set conf  110000100                         ; memA=>ILU=>OFifo
42. enable    MemARW
43. loop 0
```

This is a little bit more complex example. Lines 2 to 4 set *AddrB* to 0. Lines 5 to 9 set *water right_edge*, *inst* and *prpo* registers. Please note that this example shows how to carry out pre-relation/pre-operation.

Lines 10 to 15 calculate $A \# B_1$ and write result (called $\vec{I1}$ here) to MEM_B (please note that the result can not write to MEM_A at this time because the *execute* instruction needs *DBusA* to load cubes A). The dataflow mode used in this step is shown in Figure 5.2 (d) (see section 5.2.4).

Lines 16 to 30 calculate $\vec{I1} \# B_2$ and write result (called $\vec{I2}$) to MEM_A. This is an example of using loop instruction. Now what we want to do is to calculate the following operation:

$$\vec{I1} \# B_2 = (I1_1 \# B_2 \; I1_2 \# B_2 \; \ldots)$$

Because the array of cube $\vec{I1}$ is stored in memory bank MEM_B, we can use the "loop" instruction to carry this out.

The loop instruction loads one cube from one of two memory banks (determined by signal *enMemA*, *enMemB*, *MemARW* and *MemBRW*, see section 5.2) to the *Data* register. Then the memory address pointer will be increased by 1, and the operation currently set in the *inst* and the *prpo* registers is executed on two cubes stored in *Accu* and *Data* registers. After the operation is done, the GCU checks if the loop operation is done by comparing the content of *AddrR* with *AddrA* or *AddrB*. If their contents are not the same, the GCU will load one cube from the memory to the *Data* register again, and will repeat the whole process until the contents of *AddrR* and *AddrA* (or *AddrB*) become the same.

The array of cubes $\vec{I1}$ is stored in MEM_B. For using loop instruction, we need to set the content of *AddrR* to the number of cubes of $\vec{I1}$, which is currently stored in *AddrB*. Lines 16 to 18 copy the content of *AddrB* to *AddrR*. After that, the content of *AddrB* is set to 0 (Lines 19 to 21) to point the beginning of $\vec{I1}$. Line 22 loads cube B_2 into *Accu*.

The operation $\vec{I1}\#B_2$ is not the last operation of this example, and the result will be used in the subsequent operation, therefore, the result array of cubes of the operation will be stored in MEM_A. The dataflow mode used in this step is shown in Figure 5.2 (f) (see section 5.2.4). Lines 23 to 27 set the data flow mode.

The loop instruction is issued in Line 28. Lines 29 and 30 remove the drivers of buses *DBusA* and *DBusB* for the subsequent operation since the subsequent operation will use different dataflow mode, which means that the driver of *DBusA* and *DBusB* will be changed (remember for setting new bus driver, we have to remove the previous driver first, see section 5.3.2).

Lines 31 to 43 calculate $\vec{I2}\#B_3$ in the similar way with Lines 16 to 30. The difference is that the GCU loads the array of cubes $\vec{I2}$ from MEM_A, and write the results to the output FIFO this time. By comparing to Line 16 to 30, Line 31 to 43 are not hard to understand. The dataflow mode used in this step is shown in Figure 5.2 (e) (see section 5.2.4).

The operation $1\#(ab+\bar{a}c+\bar{b}\bar{c}) = \bar{a}b\bar{c}+a\bar{b}c$. This example is used as a test program to test the entire CCM (see section 7.2.6).

Example 6.10. Write a program in the CCM assembly to calculate the following operation:

$$\vec{A}\cdot\vec{B}\cdot\vec{C} = (A_1 + A_2 + A_3)\cdot(B_1 + B_2 + B_3)\cdot(C_1 + C_2 + C_3)$$
$$= (ab + bc + cd)\cdot(bc + cd + ad)\cdot(\bar{a} + \bar{b} + \bar{c})$$

where A, B and C are three arrays of cubes, and a, b, c and d are four binary variables.

Solution. The program is as follows:

```
set conf  000000000
enable    enififoa
set addrb 0
disable   enififoa
enable    enififod
set water 0000
set right 1111
set inst  00-0-0-0-0000-0001-0000-0000 ; intersection
set accu  01-01-11-11                  ; cube A1
set conf  001000001                    ; write result to MEM_B
enable    enIluB
disable   MemBRW
```

```
13. exec      11-01-01-11                 ; cube B1
14. exec      11-11-01-01                 ; cube B2
15. exec      01-11-11-01                 ; cube B3
16. set accu  11-01-01-11                 ; cube A2
17. exec      11-01-01-11                 ; cube B1
18. exec      11-11-01-01                 ; cube B2
19. exec      01-11-11-01                 ; cube B3
20. set accu  11-11-01-01                 ; cube A3
21. exec      11-01-01-11                 ; cube B1
22. exec      11-11-01-01                 ; cube B2
23. exec      01-11-11-01                 ; cube B3
24. disable   enIluB
25. enable    enaddrb                     ; [AddrB] => [AddrR]
26. set addrr 0
27. disable   enaddrb
28. enable    enIFifoA
29. set addrb 0
30. set conf  001101100                   ; memB=>ILU=>OFifo
31. enable    MemBRW
32. set accu  10-11-11-11                 ; cube C1
33. loop 0
34. set addrb 0
35. set accu  11-10-11-11                 ; cube C2
36. loop 0
37. set addrb 0
38. set accu  11-11-10-11                 ; cube C3
39. set conf  101101100                   ; generate finish word
40. loop 0
```

This program is very straightforward. Lines 1 to 23 calculate $\vec{A} \cdot \vec{B}$ and write the results (called $\vec{I}$) to MEM_B. Lines 24 to 38 calculate $\vec{I} \cdot \vec{C}$ and write the result to the output FIFO. This example is used as a test program to test the entire CCM (see section 7.2.6), the result is also shown there.

CHAPTER 7

Simulation

A complete design of the Cube Calculus Machine version II (CCM2) is accomplished in this thesis. This design is captured in VHDL and is simulated by QuickHDL [44, 45], a VHDL/HDL simulator from Mentor Graphics. The functionality of this design is tested and approved to be correct.

7.1 Design Capture

The design of CCM2 is captured in VHDL code hierarchically, which means that the VHDL codes of the lower level design blocks were captured first, then these blocks are tested (through simulation) and modified until their function was proved to be correct. This way, the design bugs can be identified earlier and can be fixed easier. Later, these blocks were used in the VHDL code of upper level design blocks. Figure 7.1 shows the hierarchical structure of the CCM. The rectangular boxes in the figure represent design blocks, their names and corresponding VHDL file names (in parentheses) are shown in the boxes. The "other logic" rectangular box includes VHDL modes of some basic components, like D flip-flop, multiplexer and others. Details about VHDL language can be found in [19, 20]; details about how to use QuickHDL tool can be found in [45]. The VHDL code of this design is available by contacting Dr. Perkowski (mperkows@ee.pdx.edu).

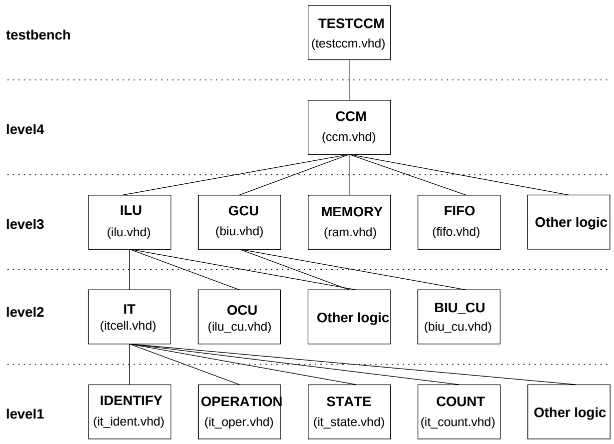


Figure 7.1: Hierarchical structure of the CCM

7.2 Functional Verification

After the design of the CCM is captured, its functionality needs to be fully tested to make sure that it does what it is supposed to. Functional verification of the CCM was performed through simulation using the QuickHDL tool.

A test bench file (testccm.vhd) was created. This test bench file realizes CCM assembly described in Chapter 6, which means that it accepts the CCM assembly instructions as input instead of a “force” file, and prints out the resultant cube(s). The test bench file greatly improves the efficiency of the functional verification. Due to the VHDL not accepting variable-length strings, this test bench file always uses “test.ccm” as the input file name, which means that the user has to rename his/her CCM assembly program to “test.ccm” before he/she tests it.

The test bench is very simple, it just simulates the function of the host computer (just interface part). The test bench contains two VHDL processes:

- Verify: This process reads one CCM instruction from file test.ccm, then encodes this instruction into binary format and pushes it into the input FIFO of the CCM. This procedure will be repeated until all CCM instructions in file test.ccm are processed. This process simulates the host computer sending CCM instructions to the CCM.

- Read Output FIFO: This process fetches the resultant cubes from the output FIFO of the CCM. This process simulates the host computer receiving the result from the CCM.

For more information about the test bench, please read Chapter 10 of [20].

A test plan was drafted to systematically verify the functionality of the CCM. The test procedure is as follows:

- A single combinational operation (like Example 6.4) was selected to be tested first because this is the simplest case.
- A single complex combinational operation without pre-relation/pre-operation has been tested.
- A single complex combinational operation with pre-relation/pre-operation had been tested.
- A single sequential operation without pre-relation/pre-operation had been tested.
- A single sequential operation with pre-relation/pre-operation had been tested.
- Complicated programs shown in Examples 6.9 and 6.10 have been tested. These two tests both perform multiple cube operations. The memory banks and several data flow modes are also tested in these two tests.

This following section will represent some tests that we have performed. All test programs (CCM assembly programs) are given in Appendix A, except the last two tests.

7.2.1 Simple combinational cube operation

This test is to test an intersection operation. TEST1.A tests an intersection operation which creates a resultant cube. TEST1.B tests an intersection operation which creates a contradiction.

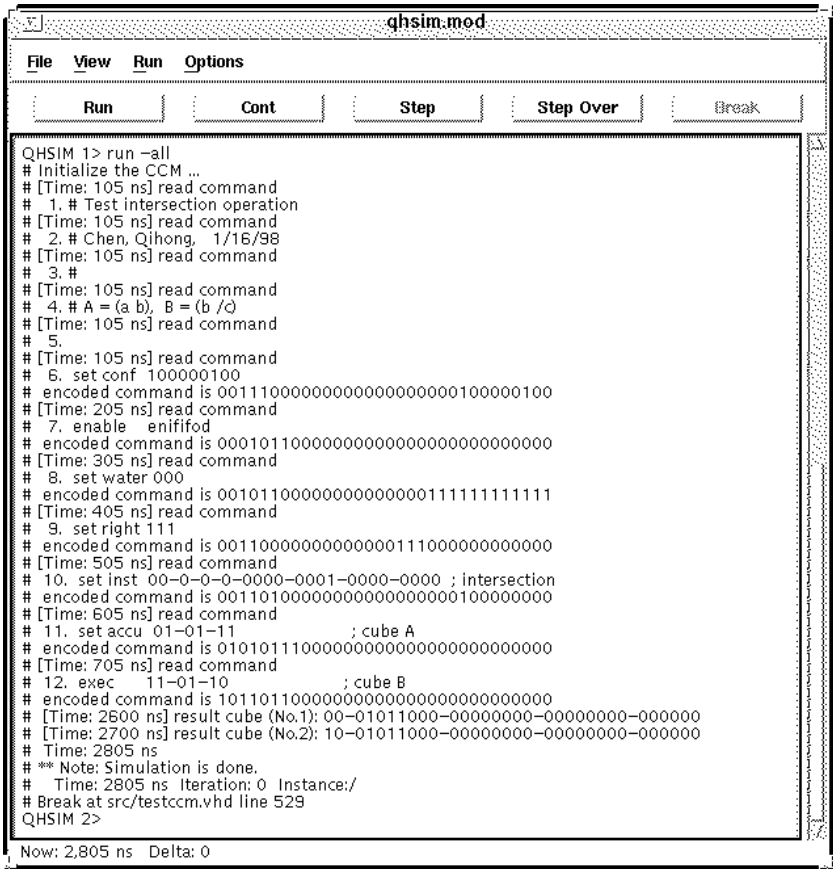


Figure 7.2: The simulation of Test1

TEST1.A

Example 2.4 is used as the test operation. The screen of the simulation is shown in Figure 7.2. As shown in the figure, every line read from file "test.ccm" shows on the simulation window, and if the input line is a valid instruction, its encoding shows on the simulation windows too. There are two resultant cubes for this test (near the bottom of the figure):

```
result cube (No.1):  00-01011000-00000000-00000000-000000
result cube (No.2):  10-01011000-00000000-00000000-000000
```

If the highest 2 bits of the resultant cube are "00", then the lower 30 bits are the resultant cube; if the highest 2 bits of the resultant cube are "10", then this word represents the "finish word" (see section §5.3, Execute instruction). For this test, the first resultant cube is a valid resultant cube which represents $ab\bar{c}$; the second resultant cube is the "finish word". This result is correct.

TEST1.B

Example 2.18 is used as a test operation. This test just produces a "finish word" which means there is no resultant cube, and it is correct.

7.2.2 Complex combinational operation without pre-relation

This test is to test a cofactor operation. Example 2.8 is used as the test operation. There are two resultant cubes for this test:

```
result cube (No.1):  00-11010000-00000000-00000000-000000
result cube (No.2):  10-11010000-00000000-00000000-000000
```

which means that the resultant cube is x_2, and it is correct.

7.2.3 Complex combinational operation with pre-relation

This test is to test a consensus operation. TEST3.A tests first pre-relation/pre-operation ($distance(A, B) = 0$). TEST3.B tests second pre-relation/pre-operation ($distance(A, B) > 1$). TEST3.C test the basic consensus operation ($distance(A, B) = 1$).

TEST3.A

Assuming two cubes $A = x_1\bar{x_3}$ and $B = x_1\bar{x_2}$, where x_1, x_2, x_3 and x_4 are binary variables. Because the distance of cubes A and B is 0, then the consensus of cubes A and B is: $A * B = A \cap B = x_1\bar{x_2}\bar{x_3}$. The outputs of the simulation are:

```
result cube (No.1):  00-01101011-00000000-00000000-000000
result cube (No.2):  10-01101011-00000000-00000000-000000
```

which is correct.

TEST3.B

Assuming two cubes $A = \bar{x_1}x_2\bar{x_3}$ and $B = x_1\bar{x_2}$, where x_1, x_2, x_3 and x_4 are binary variables. Because the distance of cubes A and B is 2 (> 1), then there is no consensus of cubes A and B. The output of the simulation is a "finish word", which is correct.

TEST3.C

Example 2.7 is used as the test operation. There are two resultant cubes for this test:

```
result cube (No.1):  00-01111011-00000000-00000000-000000
result cube (No.2):  10-01111011-00000000-00000000-000000
```

which means that the result cube is $x_1\bar{x_3}$, and it is correct.

7.2.4 Test sequential cube operation without pre-relation

This test is to test a crosslink operation. Example 2.9 is used as the test operation. There are two resultant cubes for this test:

```
result cube (No.1):  00-11111011-00000000-00000000-000000
result cube (No.2):  00-01111111-00000000-00000000-000000
result cube (No.3):  10-01111111-00000000-00000000-000000
```

which means that the result cubes are $\bar{x_3}$ and x_1, and it is correct.

7.2.5 Test sequential cube operation with pre-relation

This test is to test disjoint sharp operation. TEST3.A tests first pre-relation/pre-operation ($A \cap B = 0$). TEST3.B tests second pre-relation/pre-operation ($A \subseteq B$). TEST3.C tests the basic disjoint sharp operation.

TEST5.A

Assuming two cubes $A = \bar{x_3}$ and $B = x_2x_3x_4$, where x_1, x_2, x_3 and x_4 are binary variables. Because $A \cap B = \phi$, then the disjoint sharp $A \#d\, B = A$. The outputs of the simulation are:

```
result cube (No.1):  00-11111011-00000000-00000000-000000
result cube (No.2):  10-11111011-00000000-00000000-000000
```

which means that the resultant cube is $\bar{x}_3$ (cube A), and it is correct.

TEST5.B

Assuming two cubes $A = x_1 x_2 \bar{x}_3$ and $B = x_1 x_2$, where x_1, x_2, x_3 and x_4 are binary variables. Because $A \subseteq B$, then the disjoint sharp $A \# d\, B = \phi$. The output of the simulation is a "finish word", which is correct.

TEST5.C

Example 2.11 is used as the test operation. There are two resultant cubes for this test:

```
result cube (No.1):  00-11101011-00000000-00000000-000000
result cube (No.2):  00-11011010-00000000-00000000-000000
result cube (No.3):  10-11011010-00000000-00000000-000000
```

which means that the resultant cubes are $\bar{x}_2 \bar{x}_3$ and $x_2 \bar{x}_3 \bar{x}_4$, and it is correct.

7.2.6 Test two complex cases

This test is to test cube operation on array of cubes. The memory read/write operations and several data flow modes are verified in this test.

TEST6.A

Example 6.9 is used as the test operation. There are three resultant cubes for this test:

```
result cube (No.1):  00-10011000-00000000-00000000-000000
result cube (No.2):  00-01100100-00000000-00000000-000000
result cube (No.3):  10-00000000-00000000-00000000-000000
```

There are one "finish" words and two valid resultant cube $\bar{a}b\bar{c}$ and $a\bar{b}c$, which means $1\#(ab + \bar{a}c + \bar{b}\bar{c}) = \bar{a}b\bar{c} + a\bar{b}c$, and it is correct.

TEST6.B

Example 6.10 is used as the test operation. Let us multiply out the function manually first:

$$
\begin{aligned}
& (ab + bc + cd)(bc + cd + ad)(\bar{a} + \bar{b} + \bar{c}) \\
= & (abc + abcd + abd + bc + bcd + abcd + bcd + cd + acd)(\bar{a} + \bar{b} + \bar{c}) \\
= & \bar{a}bc + \bar{a}bcd + \bar{a}bcd + \bar{a}cd + \\
& barbcd + a\bar{b}cd + \\
& ab\bar{c}d
\end{aligned}
$$

Therefore, there are 7 cubes in the result array of cubes. Please note that the duplicated cubes are not removed, and the function is not simplified. The simulation produced 8 resultant cubes for this test:

```
result cube (No.1):  00-10010111-00000000-00000000-000000
result cube (No.2):  00-10010101-00000000-00000000-000000
result cube (No.3):  00-10010101-00000000-00000000-000000
result cube (No.4):  00-10110101-00000000-00000000-000000
result cube (No.5):  00-11100101-00000000-00000000-000000
result cube (No.6):  00-01100101-00000000-00000000-000000
result cube (No.7):  00-01011001-00000000-00000000-000000
result cube (No.8):  10-00000000-00000000-00000000-000000
```

The resultant array of cubes is correct.

CHAPTER 8

Design Evaluation

The design of CCM was captured using schematic editor of Xilinx Foundation Series software, and was implemented on 17 Xilinx XC3090A FPGA chips (see Figure 5.11) using M1 software from Xilinx. Now that the CCM has been verified in its operation, a proper timing analysis must be done to evaluation of the design of the CCM.

Since our design was mapping on multiple FPGA chips, we will focus on some paths that span multiple FPGA chips and are likely to have greater delays. The following paths will be discussed in this section:

- The path begins from the outputs of registers *accu* and *data*, and goes to the input of the output FIFO. The delay of this path is the time that the CCM takes to compute a combinational cube operation once the content of registers are set properly. This path will be refereed as *vertical path*[1] in the following section.

- The *counter carry path* includes the entire iterative network of counter blocks and the circuit that is used to evaluate pre-relation. The delay of this path is the time that the CCM takes to evaluation the signal *pre_res* (see section 3.7) once the registers and control signals are set properly.

- The *empty carry path* is the data path used to generate signal *empty*. The delay of this path is the that time the CCM takes to generate the result cube, and then determine whether the resultant cube is an empty cube or not once the registers and control signals are set properly.

[1]This path just goes through one IT, while the horizontal signals like *empty* and *ready* go through all ITs.

- The *memory path* connects two memory banks (MEM_A and MEM_b) and the registers *accu* and *data*.

The delay of the *ready* signal will not be discussed here since our design is already able to handle it (see section 3.3.3). Actually, the delay of *ready* signal is approximate to that of the *empty carry path*. Now let us analysis the time characteristics of these paths.

- The vertical path.

  All ITs are mapped in the first column of the matrix of the PeRLe-1 . The output of the ITs (resultant cubes) goes to the output FIFO through MBusE, SWE, DBusNE, FSW and FifoOutData. The delay of 385 ns is the greatest delay of this path.

- The counter carry path.

  This path goes to CSW through 4 matrix FPGAs, 3 segments of matrix direct connections, MBusS, SWS and RingSW. The delay of 643 ns is the greatest delay of this path.

- The empty carry path

  This path goes to CSW through 4 matrix FPGAs, 3 segments of matrix direct connections and RingMat. The delay of 648 ns is the greatest delay of this path.

- The memory path.

  The memory path that connects the memory bank MEM_A and the registers goes through RamDataW, SWW and MBusW. This path has a delay of 104 ns. The other memory path that connects the memory bank MEM_B and the registers goes through MBusE, one matrix FPGA, MBusS, SWS and RamDataS. This path has a delay of 160 ns.

As we discussed in section 5.4, the CCM evaluates pre-relation in states P2 and P5 of GCU, and this should be done in one clock period. Therefore, the clock period should be greater than 643ns.

For comparing the performance of the CCM and that of the software approach, a program can carry out disjoint sharp operation on two arrays of cubes was created using C language. Then this program and the CCM are used to solve the following problems:

- Three variables problem: 1# (all minterm with 3 binary variables).
- Four variables problem: 1# (all minterm with 4 binary variables).
- Five variables problem: 1# (all minterm with 5 binary variables).

The C program is compiled by GNU C compiler version 2.7.2, and is run on Sun Ultra5 workstation with 64MB real memory. The CCM is simulated using QuickHDL software from Mentor Graphics. We simulated the VHDL model of CCM, got the number of clocks used to solve the problem, then calculated the time used by CCM using formula: clock $\times$ clock-period. A clock of 1.33 MHz (clock period: 750 ns) is used as the clock of the CCM. The experiential result is shown in Table 8.1.

Table 8.1: Compare CCM (1.33 MHz) with software approach

Problem	3 variables	4 variables	5 variables
Ultra5	111 usec	268 usec	812 usec
CCM	546×0.75 $= 409$ usec	1285×0.75 $= 963.75$ usec	3405×0.75 $= 2553.75$ usec
speedup	0.27	0.28	0.32

It can be seen from Table 8.1 that our CCM is about 4 times slower than the software approach. But, the clock of the CPU of Sun Ultra5 workstation is 270 MHz, which is 206 times faster than the clock of the CCM. Therefore, we still can say that the design of the CCM is very efficient for cube calculus operations.

It also can be seen from Table 8.1 that the more variables the input cubes have, the more efficient the CCM is. This is due to the software approach need to iterate through one loop for each variable that is presented in the input cubes.

However, the clock period of 750ns is too slow. From the state diagram of the GCU (shown in Figure 5.10), it can be found that the delays of *empty carry path* and *counter carry path* only occur in a few states. Thus, if we can just give more time to these states, then we can speedup the clock of the whole CCM. This is very easy to achieve: for example, the state P2 of GCU need more time for the delay of counter carry path, so add two more states in series between states P2 and P3. These two extra states do nothing but give the CCM two more clock periods to evaluate the signal *prel_res*, which means that the CCM has 3 clock periods to evaluate signal *prel_res* in state P2 after adding two more "delay" states. After making similar modifications to all these kind of states, the CCM can run against a clock of 4 Mhz (clock period of 250 ns). The CCM was simulated again, and the result is shown in Table 8.2.

Table 8.2: Compare CCM (4MHz) with software approach

Problem	3 variables	4 variables	5 variables
Ultra5	111 usec	268 usec	812 usec
CCM	611×0.25 $= 152.75$ usec	1486×0.25 $= 371.5$ usec	4078×0.25 $= 1019.5$ usec
speedup	0.72	0.72	0.80

It is very hard to increase the clock frequency again with this mapping because some other paths like *memory path* have delays greater than 150 ns.

From the above comparison result, I have to say that a design like CCM with a complex control unit and complex data path is not good for the architecture of the PeRLe-1 board. It can be seen from our CCM mapping that since a lot of signals must go through multiple FPGA chips, this leads to greater signal delays. For instance, if we can connect the memory banks and the registers directly, then the memory path has a delay of only 35 ns. But our current memory path has a delay of 160 ns. Another issue is that XC3090 FPGA is kind of "old" now (6 to 8 years old technology). The latest FPGA from Xilinx or other vendors has more powerful CLBs and more routing resource, and they are made using deep sub-micron process technology.

If we can map the entire CCM inside one FPGA chip, then we can speedup the CCM from the following aspects:

- If we map entire CCM into one FPGA chip, the signals do not need to go through multiple chips again, which means the routing delay is reduced.

- Since the new FPGA chip has more powerful CLBs and routing resource, we can map the CCM denser. This also reduces the routing delays.

- Since new FPGA chips are made using deep sub-micron technology, the delay of CLB and routing wires are both reduced. For example, the delay of the CLB of XC3090A is 4.5 ns while the delay of CLB of XC4085XL (0.35 micron technology) is only 1.2 ns. This means that it is very easy to achieve 3 times faster mapping.

XC4085XL FPGA, a new FPGA from Xilinx, has a CLB matrix of 56×56 and up to 448 user I/O pins. The CCM should be able to map into one XC4085XL FPGA. With this new chip, it should not be difficult to run the CCM against a clock of 20 MHz (clock period: 50 ns). This means that our CCM will be about 4 times faster than the software approach while the system clock of the CCM is still 5 times slower than that of the workstation.

As said by the designers of the PeRLe-1 board in paper [9]: *PAM technology is currently best applied to low-level, massively repetitive task such as image or signal processing.* The example applications are a long integer multiplier, RSA cryptography and Fast Hough transform [9]. All these applications have no or very simple control units, and their data paths can be easily pipelined.

The CCM has a complex control unit, and a complex data path. It is difficult to pipeline the data path of the CCM. Therefore, the PeRLe-1 board is not good for the CCM.

CHAPTER 9

Applications of Cube Calculus Machine

Many logic minimization software tools such as ESPROSSO[37], MIS, SIS[38] and EXORCISM-MV-2[16, 32] may benefit from the introduction of the CCM. David W. Foote analyzed how much the CCM can improve the performance of ESPRESSO-II in his thesis[17].

The CCM can be used to perform set operation, like set intersection, set union, set complement, and set relations such as subset as we discussed in Chapter 2. The CCM can be seen as a machine for set-theoretical problems when it is configured to process only one variable with many possible values. The Examples 2.12 to 2.15 are this kind of examples.

The CCM can also speed up the process of solving small size of the satisfiability problem and the tautology problem[1] that are the two most fundamental combinational problems used in many research and application areas. As shown in [53], these two problems can be used to solve many other more complicated problems in CAD, Machine Learning and other fields. The point here is that we speed up the process of solving small size of these two problems by speeding up the very basic operator.

9.1 Satisfiability Problem

Given a product of terms, each term being a Boolean sum of literals; the *satisfiability problem* is to find any product of literals that satisfies all terms or prove that

[1]Both satisfiability problem and tautology problem are NP-complete problem, which means we are unlikely to find a polynomial-time algorithm for solving it exactly. In practice, only small size NP-complete problems can be solved and it may still be possible to find near-optimal solutions in polynomial-time using approximation algorithm, and the near-optimal solutions is often good enough. For more information about NP-complete problem, please see [54].

such product does not exist.

Example 9.1. Find all products of literals that let function $f(a,b,c,d) = (\bar{a}+b)(\bar{b}+c)(a+\bar{c})$ be 1.

Solution. It is very easy to rewrite this problem by multiplying out the expression as follows:

$$\begin{aligned} f(a,b,c,d) &= (\bar{a}+b)(\bar{b}+c)(a+\bar{c}) \\ &= (\bar{a}\bar{b}+\bar{a}c+bc)(a+\bar{c}) \\ &= abc+\bar{a}\bar{b}\bar{c} \end{aligned}$$

The Covering Problem is used in many two level logic minimization algorithms (being various improvements and extensions of the classical Quine-McCluskey algorithm[23]) and it can be reduced to the Petrick Function Minimization Problem, a special case of satisfiability problem. This means that the Covering Problem can be solved by minimizing the Petrick Function. Let us see the following example.

Example 9.2. The following is a covering table. The problem is to find the smallest set of rows that covers all columns.

	C_1	C_2	C_3	C_4	C_5	C_6
R_1	X	X		X		X
R_2			X	X		X
R_3	X	X				
R_4			X		X	

Solution. First, let us find all sets of rows that cover all columns. The column C_1 is covered by rows R_1 and R_3; the column C_2 is covered by rows R_1 and R_3; the column C_3 is covered by rows R_2 and R_4; the column C_4 is covered by rows R_1 and R_2; the column C_5 is covered by row R_4; the column C_6 is covered by rows R_1 and R_2; Therefore, the problem can be solved by solving the following Petrick function (PF):

$$
\begin{aligned}
PF \equiv 1 \equiv & (R_1 + R_3)(R_1 + R_3)(R_2 + R_4)(R_1 + R_2)(R_4)(R_1 + R_2) \\
= & (R_1 + R_3)(R_2 + R_4)(R_1 + R_2)(R_4) \\
= & (R_1R_2 + R_1R_4 + R_2R_3 + R_3R_4)(R_1 + R_2)(R_4) \\
& \dots \\
= & R_1R_4 + R_1R_2R_4 + R_1R_3R_4 + R_2R_3R_4 + R_1R_2R_3R_4
\end{aligned}
$$

The last sum shows that there are five sets of rows that cover all columns: $\{R_1,R_4\}$, $\{R_1,R_2,R_4\}$, $\{R_1,R_3,R_4\}$, $\{R_2,R_3,R_4\}$ and $\{R_1,R_2,R_3,R_4\}$. Now use the absorption law $A + AB = A$ to simplify the decision function and obtain $PF = R_1R_4 + R_2R_3R_4$, which means $\{R_1,R_4\}$ is the smallest set of rows that covers all columns.

The above example shows how to reduce the Covering Problem to the Petrick function problem. The CCM can be used to multiply out the function, then the function can be simplified by the software or Sorter/Absorber circuitry designed by Dr. Perkowski and his students[39]. It has to be kept in mind by the reader that CCM was designed because real life covering problems have matrices with hundreds of thousands of rows and columns. Because, however the number of variables in the CCM is limited, a large problem has to be first decomposed to many smaller problems that fit the CCM word length and can be solved by it sequentially.

9.2 Tautology Problem

The tautology problem is the verification a logic function to see if it is always true or not.

Example 9.3. Is the function $f(a,b,c,d) = a + b + c + d$ a tautology (always be 1)?

Solution. If the function $f(a,b,c,d) \equiv 1$, this implies that $1\#f(a,b,c,d) = 0$, and the function $f(a,b,c,d)$ is a tautology. If $1\#f(a,b,c,d) \neq 0$, then the function $f(a,b,c,d)$ is not a tautology.

Because $1\#(a + b + c + d) = \bar{a}\bar{b}\bar{c}\bar{d} \neq 0$, then the function $f(a,b,c,d)$ is not a tautology. The CCM assembly program used to solve this problem is very similar to the Example 6.9.

CHAPTER 10

Conclusions and Future Work

The concept of Cube Calculus, Cube Calculus Machine and a complete design of the Cube Calculus Machine have been presented. This is the first complete design of the CCM that has been done so far.

The design of the ILU presented in this thesis is a collaboration of many students' ideas over the years. I have evaluated two designs from past texts [15, 17, 18] and made some very important changes (like EMPTY block, Pre-relation/pre-operation logic block).

The author's contributions to the CCM project are outlined below in the manner in which the project pieces were completed.

- Design a complete CCM which is presented in this thesis. For the most part of ILU comes from past texts [15, 17, 18], all other parts have been designed by the author. The pre-relation/pre-operation that were missing in past texts [15, 17, 18] have been designed by the author.

- Modeled the CCM in VHDL code and simulated it using QuickHDL tool from Mentor Graphics. This is the first simulation of the entire CCM.

- Created CCM assembly language for the CCM and realized it in the test bench. This made the VHDL model of the CCM a research tool. Future students can explore cube operations by using this model with having the knowledge of the encoding scheme of the CCM instructions, but they still need to think in terms of registers and individual instructions of the CCM. There are several complete examples given in this thesis, and they can be used as a tutorial.

- Improve and unify the descriptions of cube calculus from previous texts.

- Wrote an introduction to the DEC PeRLe-1 board.

- Derived the formula to calculate cofactor operation by using cube calculus. The result is shown in Equation 2.11.

- Mapped this design onto the PeRLe-1 board. The entire design has been captured using Xilinx Foundation software, and has been implemented in 17 XC3090A FPGA chips using Xilinx M1 software.

Suggestions are made here for future work that will build a ready-to-use Cube Calculus Machine.

- Develop the CCM runtime library (see section 6.1). DEC C++ compiler is required.

- Build several complete demo applications of practical use such as tautology, satisfiability, or set covering.

- Run the demos on benchmark functions.

REFERENCES

[1] J. Villasenor and W. H. Mangione-Smith *Configurable Computing*, Scientific American, June 1997.

[2] URL http://www.reconfig.com/

[3] URL http://www.fccm.org/

[4] URL http://www.ccic.gov/pubs/blue97/nsa/splash.html

[5] URL http://pam.devinci.fr/

[6] URL http://www.research.digital.com/PRL/publications/pam.html

[7] P. Bertin, D. Roncin, and J. Vuillemin, *Introduction to Programmable Active Memories*, PRL Research Report 3, Digital Equipment Corp., Paris Research Lab, June 1989.

[8] P. Bertin, D. Roncin, and J. Vuillemin, *Programmable Active Memories: a Performance Assessment*, PRL Research Report 24, Digital Equipment Corp., Paris Research Lab, March 1993.

[9] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, *Programmable Active Memories: Reconfigurable Systms Come of Age*, IEEE Transactions on VLSI System, Vol 4, No. 1, pp.56-69, March 1996.

[10] P. Bertin and P. Boucard, *DECPeRLe-1 Hardware Installation Manual*, Paris Research Laboratory, Digital Equipment Corp., January 1993.

[11] P. Bertin and P. Boucard, *DECPeRLe-1 Hardware Programmer's Manual*, Paris Research Laboratory, Digital Equipment Corp., January 1993.

[12] P. Bertin, *DECPeRLe-1 Software Version 1.2 Installation Guide and General Description*, Paris Research Laboratory, Digital Equipment Corp., January 1993.

[13] Paris Research Laboratory, Digital Equipment Corp., *DECPeRLe-1 Introductory Tutorial*, April 1993

[14] H. Touati, *Perle1DC: a C++ Library for the Simulation and Generation of DECPeRLe-1 Designs*, PRL Technical Note 4, Digital Equipment Corp., Paris Research Lab, February 1994.

[15] C. Engelbarts, *The Multiple-valued Cube Calculus Machine Version 2.5*, Dept. of Electrical Engineering, Portland State University, August 1993.

[16] N. Song, *Minimization of Exclusive Sum of Products Expressions for Multiple-valued Input Incompletely Specified Functions*, Master thesis, Dept. of Electrical Engineering, Portland State University, 1993.

[17] D.W. Foote, *The design, realization and testing of the ILU of the CCM2 using FPGA technology*, Master thesis, Department of Electrical Engineering, Portland State University, 1994.

[18] L. Zhou, *Testability Design and Testability Analysis of Cube Calculus Machine*, Master thesis, Dept. of Electrical Engineering, Portland State University, 1995.

[19] S. Mazor and P. Langstraat, *A Guide to VHDL*, 2nd edition, Kluwer Academic Publishers, Boston, 1993.

[20] K. Skahill, *VHDL for Programmable Logic*, Addison-Wesley Publishing Inc., Reading, Massachusetts, 1996.

[21] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

[22] C.H. Roth, *Fundamentals of Logic Design*, 4th edition, West Publishing Company, St. Paul, 1992.

[23] O. Coudert, *Two-level Logic Minimization: an Overview*, Integration, the VLSI Journal, Vol. 17, No. 2, pp. 97-140.

[24] R.G. Casey and C. Delobel, *Decomposition of a Database and the Theory of Boolean Switching Functions*, IBM Journal Research and Development, Vol. 17, No. 5 (Sep. 1973), pp. 374-386.

[25] L.C. Farrell and M.E. Balogh, *An Application of Boolean Minimization to Database Normalization*, Technical Report, No. TR-90-25, Department. of Computer Science, Portland State University.

[26] T. Downs and M.F. Schulz, *Logic Design with Pascal Computer-Aided Design Techniques*, Van Nostrand Reinhold, New York, 1988.

[27] D.L. Dietmeyer, *Logic Design of Digital Systems*, 2nd Edition, Allyn and Bacon, Boston, 1978

[28] M.E. Ulug and B.A. Bowen, *A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems*, IEEE Transaction on Computers, Vol. C-23, No. 3 (Mar. 1974), pp. 255-267.

[29] M. Helliwell and M.A. Perkowski, *A Fast Algorithm to Minimize Multi-output Mixed-polarity Generalized Reed-Muller Forms*, Proc. 25-th ACM/IEEE Design Automation Conference, pp.427-432, Jun 12-15, 1988.

[30] M.A. Perkowski, M. Helliwell and P. Wu, *Minimization of Multiple-valued Input Multi-output Mixed-radix Exclusive Sum of Products for Incompletely Specified Boolean Functions*, Proc. of the 19th International Symposium on Multiple-Valued Logic, pp. 256-263, May 1989.

[31] M.A. Perkowski, *A Universal Logic Machine*, Proc. of the IEEE ISMVL'92, the 21st International Symposium on Multiple-Valued Logic, Sendai, Japan, May 27-29, 1992, pp. 262-271.

[32] N. Song and M.A. Perkowski, *EXORCISM-MV-2: Minimization of Exclusive Sum of Products Expression for Multiple-valued Input Incompletely Specified*

Boolean Functions, Proc. of the 23th International Symposium on Multiple-Valued Logic, pp. 132-137, May 24-27, 1993.

[33] G.D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, Boston, 1996.

[34] J.P. Roth, *Computer Logic, Testing, and Verification*, Computer Science Press, Potomac Maryland, 1980.

[35] R.R. Stoll, *Set Theory and Logic*, Dover Publications, Inc., New York, 1979.

[36] K. Hrbacek and T. Hech, *Introduction to Set Theory*, Marcel Dekker, Inc., New York, 1984.

[37] R.K. Brayton, G.D. Hachtel, C.T. McMullen and A.L. Sangiovanni-Vincentelli *Loigc Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.

[38] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A.Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, *SIS: A System for Sequential Circuit Synthesis*, Technical Report Memorandum No. UCB/ERL M92/41, University of California Berkeley, 1992.

[39] P. Kashubin, A. Ojha and E. Tuers, *An Scaleable Minterm Sorter/Absorber Using Iterative Circuit Techniques*, Project report, Dept. of Electrical Engineering, Portland State University, Spring 1997.

[40] C. Files, R. Drechsler, and M.A. Perkowski, *Functional Decomposition of MVL Functions using Multi-Valued Decisions Dragrams*, Proc. of the IEEE ISMVL'97, Halifax, Nova Scotia, May 1997, pp. 27-32.

[41] M.A. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J.S. Zhang, *Decomposition of Multi-Valued Relations*, Proc. of the IEEE ISMVL'97, Halifax, Nova Scotia, May 1997, pp. 13-18.

[42] S. Mohamed, M.A. Perkowski, and L. Jozwiak, *Fast Approximate Minimization of Multi-Output Boolean Functions in Sum-of-Condition-Decoders Structures*, Proc. Euromicro'97, September 1997.

[43] N. Song and M.A. Perkowski, *New Fast Approach to Approximate ESOP Minimization for Imcompletely Specified Multi-Output Boolean Functions*, Proc. RM'97 Conference, Oxford Univ., U.K., September 1997, pp 61-72.

[44] *QuickHDL Release Notes*, Software Version 8.5_4.6c, Mentor Graphics Corporation, Wilsonville, OR, 1996.

[45] *QuickHDL User's and Reference Manual*, Software Version 8.5_4.6c, Mentor Graphics Corporation, Wilsonville, OR, 1996.

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

[47] *XILINX: User Guide and Tutorials*, Xilinx Inc., San Jose, CA, 1991.

[48] *XILINX: XAPP Applications Handbook*, Xilinx Inc., San Jose, CA, 1992.

[49] *Quick Start Guide for Xilinx Alliance Series 1.4*, Xilinx Inc., San Jose, CA, 1997.

[50] *Development System User Guide for Xilinx Alliance Series 1.4*, Xilinx Inc., San Jose, CA, 1997.

[51] *Xilinx Library Guide*, Xilinx Inc., San Jose, CA, 1997.

[52] Dave Van Den Bout, *The Practical Xilinx Designer Lab Book*, Prentice Hall Inc., New Jersey, Dec. 1997.

[53] M.A. Perkowski's book in preparation on Finite State Machine, winter 1998.

[54] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts, 1989.

APPENDIX A

Test Programs

All CCM assembly programs listed in this appendix are used to test the functionality of our CCM design, see section 7.2 for detail.

TEST1.A

```
# Test intersection operation
# Chen, Qihong,   1/16/98
#
# A = (a b),  B = (b /c)

 set conf  100000100
 enable    enififod
 set water 000
 set right 111
 set inst  00-0-0-0-0000-0001-0000-0000  ; intersection
 set accu  01-01-11                      ; cube A
 exec      11-01-10                      ; cube B
```

TEST1.B

```
# Test intersection operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 00
 set right 11
 set inst  00-0-0-0-0000-0001-0000-0000  ; intersection
 set accu  01-01                         ; cube A
 exec      01-10                         ; cube B
```

TEST2

```
# Test cofactor operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
```

```
 set water 00
 set right 11
 set inst  00-0-1-1-1011-0001-1111-0000  ; cofactor
 set accu  01-01                         ; cube A
 exec      01-11                         ; cube B
```

TEST3.A

```
# Test consensus operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 0000
 set right 1111
 set prpo  1-1110-01-0-0001--1-1110-10-1-0000 ; consensus
 set inst  11-0-1-1-1110-0001-0111-0000       ; consensus
 set accu  01-11-10-11                        ; cube A
 exec      01-10-11-11                        ; cube B
```

TEST3.B

```
# Test consensus operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 0000
 set right 1111
 set prpo  1-1110-01-0-0001--1-1110-10-1-0000 ; consensus
 set inst  11-0-1-1-1110-0001-0111-0000       ; consensus
 set accu  10-01-10-11                        ; cube A
 exec      01-10-11-11                        ; cube B
```

TEST3.C

```
# Test consensus operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 0000
 set right 1111
 set prpo  1-1110-01-0-0001--1-1110-10-1-0000 ; consensus
 set inst  11-0-1-1-1110-0001-0111-0000       ; consensus
 set accu  01-01-10-11                        ; cube A
 exec      01-10-11-11                        ; cube B
```

TEST4

```
# Test crosslink operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 0000
 set right 1111
 set inst  00-1-0-1-1110-0011-0111-0101       ; crosslink
 set accu  10-11-10-11                        ; cube A
 exec      01-11-01-11                        ; cube B
```

TEST5.A

```
# Test disjoint sharp operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 0000
 set right 1111
 set prpo  1-1110-10-0-0011--0-0010-01-0-0000 ; disjoint-sharp [A]#[D]
 set inst  11-1-0-0-0010-0011-0010-0001       ; disjoint-sharp [A]#[D]
 set accu  11-11-10-11                        ; cube A
 exec      11-01-01-01                        ; cube B
```

TEST5.B

```
# Test disjoint sharp operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 0000
 set right 1111
 set prpo  1-1110-10-0-0011--0-0010-01-0-0000 ; disjoint-sharp [A]#[D]
 set inst  11-1-0-0-0010-0011-0010-0001       ; disjoint-sharp [A]#[D]
 set accu  01-01-10-11                        ; cube A
 exec      01-01-11-11                        ; cube B
```

TEST5.C

```
# Test disjoint sharp operation
# Chen, Qihong,   1/16/98

 set conf  100000100
 enable    enififod
 set water 0000
 set right 1111
 set prpo  1-1110-10-0-0011--0-0010-01-0-0000 ; disjoint-sharp [A]#[D]
 set inst  11-1-0-0-0010-0011-0010-0001       ; disjoint-sharp [A]#[D]
 set accu  11-11-10-11                        ; cube A
 exec      11-01-11-01                        ; cube B
```

APPENDIX B

VHDL Codes

it_ident.vhd

```
-- The identify block of IT.
--
-- Signals:
--   rel: 4 bits output value of partial relation function.
--   a: two bits from operand literal A.
--   b: two bits from operand literal B.
--   and_or: and_or signal
--   redge: re[i], right edge signal
--   redge_pre: re[i-1], right edge signal
--   water: w[i], water signal
--   carry: carry[i], carry signal
--   carry_nxt: carry[i+1], carry signal
--   conf: conf[i], confirm signal
--   conf_pre: conf[i-1], confirm signal
--   count: count[i], count signal

library ieee;
use ieee.std_logic_1164.all;
use work.all;
use work.parts.all;

entity it_identify is
  port( rel: in  std_logic_vector (0 to 3);
        a, b: in  std_logic_vector (0 to 1);
        and_or, water:    in std_logic;
        redge, redge_pre: in std_logic;
        carry, conf:      in std_logic;
        carry_nxt, conf_pre, var: out std_logic;
        count : out std_logic
      );
end;

architecture dataflow of it_identify is

signal u0sel, u1sel: std_logic_vector (1 downto 0);
signal rel0, rel1, i_carry_nxt, i_conf_pre, i_var: std_logic;
```

```
begin

  u0sel <= a(0) & b(0);
  U0: parts.mux41 port map (din=>rel, sel=>u0sel, dout=>rel0);

  u1sel <= a(1) & b(1);
  U1: parts.mux41 port map (din=>rel, sel=>u1sel, dout=>rel1);

  i_carry_nxt <= ((not water)
                  and (   (rel0 and rel1 and redge_pre)
                       or (rel0 and rel1 and carry)
                       or (carry and (not redge_pre) and (not and_or))
                       or ((rel0 or rel1) and (not and_or))
                      )
                 )
              or (water and carry) after 2 ns;

  i_conf_pre <= (i_carry_nxt and redge) or (conf and (not redge)) after 2 ns;

  i_var <= i_conf_pre and (not water) after 2 ns;

  carry_nxt <= i_carry_nxt;

  conf_pre <= i_conf_pre;

  count <= i_var and redge and (not water) after 2 ns;

  var <= i_var;

end dataflow;
```

it_oper.vhd

```
-- The operation block of IT cell
--
-- Signals:
--   bef: 4 bits output value of before function.
--   act: 4 bits output value of active function.
--   aft: 4 bits output value of after function.
--     a: two bits from operand literal A.
--     b: two bits from operand literal B.
--     c: two bits of the output literal.
--   state: state of the IT, comes from state block.

library ieee;
use ieee.std_logic_1164.all;
use work.all;
use work.parts.all;
```

```
entity it_operation is
  port (bef,act,aft: in  std_logic_vector (0 to 3);
        state:       in  std_logic_vector (1 downto 0);
        a,b:         in  std_logic_vector (0 to 1);
        c:           out std_logic_vector (0 to 1));
end;

architecture dataflow of it_operation is

-- temp signals.
signal u0sel,u1sel: std_logic_vector (1 downto 0);
signal bus4, fourzero: std_logic_vector (0 to 3);

begin

  fourzero <= "0000";

  U0: parts.mux441 port map
     (din0=>bef, din1=>act, din2=>aft, din3=>fourzero, sel=>state, dout=>bus4);

  u0sel <= a(0) & b(0);
  U1: parts.mux41 port map (din=>bus4, sel=>u0sel, dout=>c(0));

  u1sel <= a(1) & b(1);
  U2: parts.mux41 port map (din=>bus4, sel=>u1sel, dout=>c(1));

end dataflow;
```

it_state.vhd

```
-- The state block of IT cell
--
-- Signals:
--   clear: used to reset IT to "before" state.
--   request: the clock signal of FSM of the IT
--   reset: global reset. this signal can be seen as chip reset.
--   prime: prime signal
--   nxt: next[i] signal ("next" is a reserved word in VHDL).
--   nxt_nxt: next[i+1] signal.
--   var: var[i], variable signal
--   water: w[i], water signal
--   redge: re[i], right edge signal
--   state: state signal
--   ready: subready[i], ready signal of the IT

library ieee;
use ieee.std_logic_1164.all;
use work.all;
```

```
use work.parts.all;

entity it_state is
  port (clear, request, reset, prime: in std_logic;  -- global signals
        nxt, var, water, redge: in std_logic;
        state: out std_logic_vector (1 downto 0);
        nxt_nxt, ready: out std_logic);
end;

architecture dataflow of it_state is

signal st1, st0 : std_logic;   -- current state
signal nst1, nst0: std_logic;  -- next state
signal dff_reset: std_logic;

begin

  dff_reset <= reset or clear;

  U0: parts.dff port map
    (d=>nst0, clk=>request, reset=>dff_reset, q=>st0);

  U1: parts.dff port map
    (d=>nst1, clk=>request, reset=>dff_reset, q=>st1);

  nst0 <= (not st1) and (not st0) and (not clear) and nxt and var after 3 ns;
  nst1 <= st0 or st1 or  (nxt and (not var)) after 3 ns;

  state(1) <= st1;
  state(0) <= st0 or (var and prime) after 1 ns;

  nxt_nxt <= (nxt and water) or
             ((not water) and
             (((not st1) and st0) or (nxt and (not (var and redge)))))
             after 3 ns;

  ready <= redge and nxt and var and (not request) after 3 ns;

end dataflow;
```

it_count.vhd

```
-- The counter block of IT
--
-- Signals:
--   cnt_in: cnt[i], counter carry signal
--   cnt_out: cnt[i+1], counter carry signal
--   count: count[i], count signal
--
```

```
-- This is a 4-bit counter. In this VHDL mode of CCM, there are 15 ITs
-- in the ILU, which means the counter should be able to count from 0
-- to 15. As described in the thesis, 4-bit counter can count from 0 to
-- 14. By combine the "count" signal of the last IT, we are able to
-- count from 0 to 15. By using 4-bit counter instead of 5-bit counter,
-- we save 1 iterative signal. The decoder of this counter is described
-- by pcountd.vhd

library ieee;
use ieee.std_logic_1164.all;
use work.all;
use work.parts.all;

entity it_count is
  port (cnt_in  : in std_logic_vector (3 downto 0);
        count   : in std_logic;
        cnt_out : out std_logic_vector (3 downto 0)
       );
end;

architecture arch of it_count is

  signal tmp_sig : std_logic_vector(1 to 1);

begin

  u0: parts.mux21N generic map (size=>1) port map
   ( din0=>cnt_in(0 downto 0), din1=>cnt_in(1 downto 1),
     sel=>count, dout=>cnt_out(0 downto 0));

  u1: parts.mux21N generic map (1) port map
   ( din0=>cnt_in(1 downto 1), din1=>cnt_in(2 downto 2),
     sel=>count, dout=>cnt_out(1 downto 1));

  u2: parts.mux21N generic map (1) port map
   ( din0=>cnt_in(2 downto 2), din1=>cnt_in(3 downto 3),
     sel=>count, dout=>cnt_out(2 downto 2));

  tmp_sig(1) <= cnt_in(0) xor cnt_in(1);

  u3: parts.mux21N generic map (1) port map
   ( din0=>cnt_in(3 downto 3), din1=>tmp_sig,
     sel=>count, dout=>cnt_out(3 downto 3));

end arch;
```

it_empty.vhd

```
-- The empty block of IT
```

```
--
-- Signals:
--   redge_pre: re[i-1], right edge signal
--   redge: re[i], right edge signal
--   water: w[i], water signal
--   empty_pre: empty_carry[i], empty carry signal
--   empty_nxt: empty_carry[i+1], empty carry signal
--   empty: subempty[i], empty signal
--   c: two bits of the output literal.

library ieee;
use ieee.std_logic_1164.all;

entity it_empty is
  port (c : in std_logic_vector (0 to 1);
        redge_pre, redge, water : in std_logic;
        empty_pre : in std_logic;
        empty_nxt, empty : out std_logic);
end;

architecture dataflow of it_empty is

signal empty_nxt_tmp : std_logic;

begin

  empty_nxt_tmp <= (redge_pre and (not c(0)) and (not c(1))) or
            ((not redge_pre) and empty_pre and (not c(0)) and (not c(1)));

  empty <= empty_nxt_tmp and redge and (not water);

  empty_nxt <= empty_nxt_tmp;

end dataflow;
```

itcell.vhd

```
-- IT cell
--
-- Signals:
--   bef: 4 bits output value of before function.
--   act: 4 bits output value of active function.
--   aft: 4 bits output value of after function.
--   rel: 4 bits output value of partial relation function.
--   a: two bits from operand literal A.
--   b: two bits from operand literal B.
--   c: two bits of the output literal.
--   and_or: and_or signal
--   redge: re[i], right edge signal
```

```
--   redge_pre: re[i-1], right edge signal
--   water: w[i], water signal
--   reset: global reset. this signal can be seen as chip reset.
--   request: the clock signal of FSM of the IT
--   clear: used to reset IT to "before" state.
--   prime: prime signal
--   nxt: next[i] signal ("next" is a reserved word in VHDL).
--   nxt_nxt: next[i+1] signal.
--   carry: carry[i], carry signal
--   carry_nxt: carry[i+1], carry signal
--   conf: conf[i], confirm signal
--   conf_pre: conf[i-1], confirm signal
--   ready: subready[i], ready signal of the IT
--   empty_pre: empty_carry[i], empty carry signal
--   empty_nxt: empty_carry[i+1], empty carry signal
--   empty: subempty[i], empty signal
--   cnt_in: cnt[i], counter carry signal
--   cnt_out: cnt[i+1], counter carry signal
--   count: count[i], count signal

library ieee;
use ieee.std_logic_1164.all;
use work.all;
use work.parts.all;

entity itcell is
  port (rel, bef, act, aft: in std_logic_vector (0 to 3);
        a, b: in std_logic_vector (0 to 1);
        and_or, redge, redge_pre, water: in std_logic;
        reset, request, clear, prime: in std_logic;
        nxt,     carry,     conf    : in std_logic;  -- propagation signals
        nxt_nxt, carry_nxt, conf_pre: out std_logic; -- propagation signals
        ready: out std_logic;
        c: out std_logic_vector (0 to 1);
        empty_pre : in std_logic;
        empty_nxt, empty : out std_logic;
        cnt_in : in std_logic_vector (3 downto 0);
        cnt_out : out std_logic_vector (3 downto 0);
        count : out std_logic );
end;

architecture dataflow of itcell is

component it_operation
  port (bef,act,aft: in  std_logic_vector (0 to 3);
        state:       in  std_logic_vector (1 downto 0);
        a,b:         in  std_logic_vector (0 to 1);
        c:           out std_logic_vector (0 to 1));
end component;
```

```
component it_state
  port (clear, request, reset, prime: in std_logic;
        nxt, var, water, redge: in std_logic;
        state: out std_logic_vector (1 downto 0);
        nxt_nxt, ready: out std_logic);
end component;

component it_identify
  port (rel: in  std_logic_vector (0 to 3);
        a, b: in  std_logic_vector (0 to 1);
        and_or, water:    in std_logic;
        redge, redge_pre: in std_logic;
        carry, conf:      in std_logic;
        carry_nxt, conf_pre, var: out std_logic;
        count : out std_logic );
end component;

component it_empty
  port (c : in std_logic_vector (0 to 1);
        redge_pre, redge, water : in std_logic;
        empty_pre : in std_logic;
        empty_nxt, empty : out std_logic);
end component;

component it_count
  port (cnt_in  : in std_logic_vector (3 downto 0);
        count   : in std_logic;
        cnt_out : out std_logic_vector (3 downto 0));
end component;

-- state: state[i], state signal of FSM within the IT
signal state : std_logic_vector (1 downto 0);

-- var: var[i], variable signal
-- tmp_count signal is the count signal.
signal var, tmp_count : std_logic;
-- c_tmp signal is the c signal (the output literal)
signal c_tmp : std_logic_vector (0 to 1);

begin

  U0: it_identify port map
      (rel=>rel, and_or=>and_or, a=>a, b=>b,
       water=>water, redge=>redge, redge_pre=>redge_pre,
       carry=>carry, carry_nxt=>carry_nxt,
       conf=>conf, conf_pre=>conf_pre, var=>var, count=>tmp_count);

  U1: it_state port map
      (clear=>clear, request=>request, reset=>reset, prime=>prime,
       nxt=>nxt, var=>var, water=>water, redge=>redge,
```

```
      state=>state, nxt_nxt=>nxt_nxt, ready=>ready);

  U2: it_operation port map
     (bef=>bef, act=>act, aft=>aft, state=>state,
      a=>a, b=>b, c=>c_tmp);

  c <= c_tmp;

  U3: it_empty port map
     (c=>c_tmp, redge_pre=>redge_pre, redge=>redge, water=>water,
      empty_pre=>empty_pre, empty_nxt=>empty_nxt, empty=>empty);

  U4: it_count port map
     (cnt_in=>cnt_in, count=>tmp_count, cnt_out=>cnt_out);

  count <= tmp_count;

end dataflow;
```

pcountd.vhd

```
-- Pseudo-random number decoder
--
-- Signals:
--   din: 5 bits pseudo-random number input
--   dout: 4 bits binary number output

library ieee;
use ieee.std_logic_1164.all;

-------------------------------------------------------
-- psedo-counter decoder
-------------------------------------------------------
entity pcount_decoder is
  port (din  : in std_logic_vector (4 downto 0);
        dout : out std_logic_vector (3 downto 0)
       );
end;

architecture arch of pcount_decoder is

begin

  dout <= "0000" after 5 ns when (din = "01111") else
          "0001" after 5 ns when (din = "00111") else
          "0001" after 5 ns when (din = "10111") else
          "0010" after 5 ns when (din = "00011") else
          "0010" after 5 ns when (din = "10011") else
          "0011" after 5 ns when (din = "00001") else
```

```
        "0011" after 5 ns when (din = "10001") else
        "0100" after 5 ns when (din = "01000") else
        "0100" after 5 ns when (din = "11000") else
        "0101" after 5 ns when (din = "00100") else
        "0101" after 5 ns when (din = "10100") else
        "0110" after 5 ns when (din = "00010") else
        "0110" after 5 ns when (din = "10010") else
        "0111" after 5 ns when (din = "01001") else
        "0111" after 5 ns when (din = "11001") else
        "1000" after 5 ns when (din = "01100") else
        "1000" after 5 ns when (din = "11100") else
        "1001" after 5 ns when (din = "00110") else
        "1001" after 5 ns when (din = "10110") else
        "1010" after 5 ns when (din = "01011") else
        "1010" after 5 ns when (din = "11011") else
        "1011" after 5 ns when (din = "00101") else
        "1011" after 5 ns when (din = "10101") else
        "1100" after 5 ns when (din = "01010") else
        "1100" after 5 ns when (din = "11010") else
        "1101" after 5 ns when (din = "01101") else
        "1101" after 5 ns when (din = "11101") else
        "1110" after 5 ns when (din = "01110") else
        "1110" after 5 ns when (din = "11110") else
        "1111" after 5 ns when (din = "11111") else
        "----" after 5 ns;

end arch;
```

ilu_cu.vhd

```
-- File:   ilu_cu.vhd
-- Author: CHEN, Qihong,  Portland State University
-- Date:   12/9/97
--
-- The control unit of ILU (OCU in the thesis)
--
-- Signals:
--   reset: global reset. this signal can be seen as chip reset.
--   clk: global clock signal.
--   enable: ilu_enable signal in the thesis.
--   done: ilu_done signal in the thesis.
--   init: next[1], first next signal
--   term: next[n+1], last next signal.
--   ready: ready signal
--   to_mem: toMem signal, one bit of config register.
--   clear: used to reset all ITs to "before" state.
--   request: the clock signal of FSM of the IT
--   write_output: this signal is generated when there is a resultant cube.
--   inc_waddr: this signal used to increase mem address unit by 1.
```

```
--
--  The section 3.6 of the thesis give a brief introduction to this finite
--  state machine. The OCU is only used to deal with sequential cube operation.
--  The ilu_enable (enable in the VHDL code) will be 0 when the CCM is used to
--  carry out the combinational cube operation (including complex combinational
--  cube operation). In this case, the clear signal is set to 1 (in state st0)
--  to keep all ITs in "before" state.

library ieee;
use ieee.std_logic_1164.all;

entity ilu_cu is
  port (reset, clk, enable, ready, term, to_mem: in std_logic;
        clear, request, init, write_output, inc_waddr, done: out std_logic );
end;

architecture behavior of ilu_cu is

  -- In figure 3.17 of the thesis, the states of OCU are s0 to s5, and they
  -- are corresponding to st0 ... st5, respectively.
  type ILUstate is (st0, st1, st2, st3, st4, st5);

  signal present_state, next_state: ILUstate;

begin


  state_clocked: process (clk)
  begin
    if (clk'event and clk='1') then
       present_state <= next_state;
    end if;
  end process state_clocked;

  state_comb: process (present_state, enable, reset, ready, term)
  begin

    if (reset = '1') then

       next_state <= st0;

    else

       case present_state is

         when st0 =>

           if (enable  = '1') then
              next_state <= st1;
           else
```

```
      next_state <= st0;
   end if;

   clear         <= '1';
   request       <= '0';
   init          <= '0';
   write_output  <= '0';
   inc_waddr     <= '0';
   done          <= '0';

when st1 =>

   if (term = '1') then
      next_state <= st5;
   else
     if (ready = '1') then
        next_state <= st2;
     else
        next_state <= st1;
     end if;
   end if;

   clear         <= '0';
   request       <= '0';
   init          <= '1';
   write_output  <= '0';
   inc_waddr     <= '0';
   done          <= '0';

when st2 =>

   next_state <= st3;

   clear         <= '0';
   request       <= '1';
   init          <= '1';
   write_output  <= '1';
   inc_waddr     <= '0';
   done          <= '0';


when st3 =>

   if (term = '1') then
      next_state <= st5;
   else
     if (ready = '1') then
        next_state <= st4;
     else
        next_state <= st3;
```

```
          end if;
        end if;

        clear         <= '0';
        request       <= '0';
        init          <= '0';
        write_output  <= '0';
        inc_waddr     <= to_mem;
        done          <= '0';

      when st4 =>

        next_state <= st3;

        clear         <= '0';
        request       <= '1';
        init          <= '0';
        write_output  <= '1';
        inc_waddr     <= '0';
        done          <= '0';

      when st5 =>

        next_state <= st0;

        clear         <= '0';
        request       <= '0';
        init          <= '0';
        write_output  <= '0';
        inc_waddr     <= '0';
        done          <= '1';

     end case;
   end if;
  end process state_comb;

end behavior;
```

ilu.vhd

```
-- The ILU
--
-- Signals:
--   reset: global reset. this signal can be seen as chip reset.
--   clk: global clock signal.
--   ilu_enable: ilu_enable signal.
--   ilu_done: ilu_done signal.
--   bef: 4 bits output value of before function.
--   act: 4 bits output value of active function.
```

```
--   aft: 4 bits output value of after function.
--   rel: 4 bits output value of partial relation function.
--   a: two bits from operand literal A.
--   b: two bits from operand literal B.
--   c: two bits of the output literal.
--   and_or: and_or signal
--   prime: prime signal
--   to_mem: toMem signal, one bit of config register.
--   redge: re in the thesis, right edge vector signal.
--   water: w in the thesis, water vector signal.
--   empty: empty signal.
--   write_output: this signal is generated when there is a resultant cube.
--   inc_waddr: this signal used to increase mem address unit by 1.
--   cnt_val: counter value, the output of pseudo-random number decoder.
--
-- ILU consists of the iterative network (IT is the cell) and the control
-- unit of the iterative network (called OCU in the thesis).

library ieee;
use ieee.std_logic_1164.all;
use work.all;

-------------------------------------------------------
-- ILU
-------------------------------------------------------
entity ilu is
  generic
   ( NumberOfIT : integer := 4 );  -- The number of ITs
  port
   ( reset, clk, ilu_enable, prime, to_mem, and_or : std_logic;
     rel, bef, act, aft: in std_logic_vector (0 to 3);
     water, redge: in std_logic_vector (0 to NumberOfIT - 1);
     a, b: in std_logic_vector (0 to (NumberOfIT * 2 - 1));
     c: out std_logic_vector (0 to (NumberOfIT * 2 - 1));
     ilu_done, write_output, inc_waddr, empty : out std_logic;
     cnt_val : out std_logic_vector (3 downto 0));
end;

architecture dataflow of ilu is

component ilu_cu
  port (reset, clk, enable, ready, term, to_mem: in std_logic;
        clear, request, init, write_output, inc_waddr, done: out std_logic);
end component;

component itcell
  port (rel, bef, act, aft: in std_logic_vector (0 to 3);
        a, b: in std_logic_vector (0 to 1);
        and_or, redge, redge_pre, water: in std_logic;
        reset, request, clear, prime: in std_logic;
```

```
        nxt,     carry,     conf    : in std_logic;  -- propagation signals
        nxt_nxt, carry_nxt, conf_pre: out std_logic; -- propagation signals
        ready: out std_logic;
        c: out std_logic_vector (0 to 1);
        empty_pre : in std_logic;
        empty_nxt, empty : out std_logic;
        cnt_in : in std_logic_vector (3 downto 0);
        cnt_out : out std_logic_vector (3 downto 0);
        count : out std_logic );
end component;

component pcount_decoder
  port (din  : in std_logic_vector (4 downto 0);
        dout : out std_logic_vector (3 downto 0));
end component;

-- carry, nxt (next in the thesis), conf, empty_carry, nct_carry
-- all are propagational signals of the iterative network.
signal clear, request, ready, init, term : std_logic;
signal carry, nxt : std_logic_vector(1 to (NumberOfIT+1));
signal conf : std_logic_vector(0 to NumberOfIT);
signal empty_carry: std_logic_vector(1 to NumberOfIT+1);
signal subready, subempty: std_logic_vector(1 to NumberOfIT);
signal const_one : std_logic := '1';
signal tempready, tempempty: std_logic_vector(1 to NumberOfIT);
signal subcount : std_logic_vector (1 to NumberOfIT);
signal decoder_in: std_logic_vector (4 downto 0);

type sigNx4 is array (1 to NumberOfIT+1) of std_logic_vector(3 downto 0);
signal cnt_carry: sigNx4;

begin

  cu: ilu_cu port map
      (reset=>reset, clk=>clk, enable=>ilu_enable, ready=>ready,
       term=>term, to_mem=>to_mem, clear=>clear, request=>request,
       init=>init, write_output=>write_output,inc_waddr=>inc_waddr,
       done=>ilu_done );

  carry(1) <= '0';            -- carry(1) is "don't care"
  conf(NumberOfIT) <= '0';   -- conf(NumberOfIT) is "don't care"
  empty_carry(1) <= '0';     -- empty_carry(1) is "don't care"

  nxt(1) <= init;
  term <= nxt(NumberOfIT+1);

  cnt_carry(1) <= "1111";

  it: for i in 1 to NumberOfIT generate
    it1: if i=1 generate
```

```
    it1: itcell port map
     ( rel=>rel, bef=>bef, act=>act, aft=>aft,
       a=>a(0 to 1), b=>b(0 to 1),
       and_or=>and_or, redge_pre=>const_one, redge=>redge(0), water=>water(0),
       reset=>reset, request=>request, clear=>clear, prime=>prime,
       nxt=>nxt(1),  carry=>carry(1), conf_pre=>conf(0),
       nxt_nxt=>nxt(2), carry_nxt=>carry(2), conf=>conf(1),
       ready=>subready(1), c=>c(0 to 1),
       empty_pre=>empty_carry(1), empty_nxt=>empty_carry(2),
       empty=>subempty(1), count=>subcount(1),
       cnt_in=>cnt_carry(1), cnt_out=>cnt_carry(2)
     );
  end generate it1;

  iti: if i>1 generate
    iti: itcell port map
     ( rel=>rel, bef=>bef, act=>act, aft=>aft,
       a=>a(2*i-2 to 2*i-1), b=>b(2*i-2 to 2*i-1), and_or=>and_or,
       redge_pre=>redge(i-2), redge=>redge(i-1), water=>water(i-1),
       reset=>reset, request=>request, clear=>clear, prime=>prime,
       nxt=>nxt(i),  carry=>carry(i), conf_pre=>conf(i-1),
       nxt_nxt=>nxt(i+1), carry_nxt=>carry(i+1), conf=>conf(i),
       ready=>subready(i), c=>c(2*i-2 to 2*i-1),
       empty_pre=>empty_carry(i), empty_nxt=>empty_carry(i+1),
       empty=>subempty(i), count=>subcount(i),
       cnt_in=>cnt_carry(i), cnt_out=>cnt_carry(i+1)
     );
  end generate iti;
end generate it;

-- the count signal of last IT is combined with 4-bit pseudo-random number.
-- By using 4-bit counter instead of 5-bit counter, we save 1 iterative
-- signal. The decoder of this counter is described by pcountd.vhd
decoder_in <= subcount(NumberOfIT) & cnt_carry(NumberOfIT);

-- see pcountd.vhd
decoder : pcount_decoder port map
 ( din=>decoder_in, dout=>cnt_val );

-- ready = subready(1) or subready(2) or ... or subready(NumberOfIT)
tempready(1) <= subready(1);
readys: for i in 2 to NumberOfIT generate
  tempready(i) <= tempready(i-1) or subready(i);
end generate readys;
ready <= tempready(NumberOfIT) after 3 ns;

-- empty = subempty(1) or subempty(2) or ... or subempty(NumberOfIT)
tempempty(1) <= subempty(1);
emptys: for i in 2 to NumberOfIT generate
  tempempty(i) <= tempempty(i-1) or subempty(i);
```

```
  end generate emptys;
  empty <= tempempty(NumberOfIT) after 3 ns;

end dataflow;
```

biu_cu.vhd

```
-- File:   biu_cu.vhd
-- Author: CHEN, Qihong,  Portland State University
-- Date:   3/30/97
--
-- The control unit of GCU (it was called BIU, Bus Interface Unit)
--
-- Signals:
--   reset: global reset. this signal can be seen as chip reset.
--   clk: global clock signal.
--   infifoempty: whether the input FIFO is empty or not.
--   ilu_enable: ilu_enable signal.
--   ilu_done: ilu_done signal.
--   seq_com: sequential or combinational operation.
--   loop_done: whether a loop operation is done or not.
--   to_mem: toMem signal, one bit of config register.
--   opc: the highest 3 bits of CCM instructions, it's the op-code.
--   read_fifo: read the CCM instructions from the input FIFO.
--   write_fifo: write the result to the output FIFO.
--   ld_tbufs: load tri-state buffers control bits.
--   ld_regs: load registers.
--   ld_accu: load accumulator.
--   ld_data: load data register.
--   inc_raddr1: increase the source mem (for reading operands) address by 1
--   inc_waddr1: increase the target mem (for writing results) address by 1.
--   finish: finish bit, see section 5.3.4 of the thesis.
--   write_output1: this signal is generated when there is a resultant cube.
--   mem_read: mem read signal, it is a temporary signal, see biu.vhd.
--   prel1,prel2,prel_res,prel_sel: see pre-relation/per-operation section in
--                                 the thesis.
--
--  The section 5.4 of the thesis give a brief introduction to this finite
--  state machine. The GCU is used to deal with combinational cube operation
--  and pre-relation/pre-operation.

library ieee;
use ieee.std_logic_1164.all;
-- use work.parts;
use work.ccmtype.all;

entity biu_cu is
  port (reset, clk, ilu_done, infifoempty : in std_logic;
        seq_com, loop_done, to_mem: in std_logic;
```

```
        opc : in std_logic_vector (0 to 2);
        read_fifo, ilu_enable : out std_logic;
        ld_tbufs, ld_regs, ld_accu, ld_data : out std_logic;
        inc_raddr1, inc_waddr1, finish: out std_logic;
        write_fifo, write_output1, mem_read : out std_logic;
        state : out BIUstate;
        prel1, prel2, prel_res : in std_logic;
        prel_sel : out std_logic_vector(1 downto 0) );
end;

architecture behavior of biu_cu is
--  type BIUstate is (s0, s1, s2, s3, s4, s5, s6, s7, p1, ..., p7);
  signal present_state, next_state: BIUstate;
  signal cur_prel_sel, next_prel_sel: std_logic_vector(1 downto 0);

begin

  state <= present_state;
  prel_sel <= cur_prel_sel;

  state_clocked: process (clk)
  begin
    if (clk'event and clk='1') then
       present_state <= next_state after 5 ns;
       cur_prel_sel <= next_prel_sel;
    end if;
  end process state_clocked;

  state_comb: process
    ( present_state, reset, loop_done, infifoempty,
      to_mem, ilu_done, seq_com, opc(0), opc(1), opc(2),
      prel1, prel2, prel_res )
  begin

    if (reset = '1') then

       next_state <= s0;

    else

       case present_state is

         when s0 =>

           if (infifoempty = '1') then
              next_state <= s0;
           else
              next_state <= s1;
           end if;
```

```
      ilu_enable <= '0';
      ld_tbufs   <= '0';
      ld_regs    <= '0';
      ld_accu    <= '0';
      ld_data    <= '0';
      inc_raddr1 <= '0';
      inc_waddr1 <= '0';
      finish     <= '0';
      read_fifo  <= not infifoempty;
      write_fifo <= '0';
      write_output1 <= '0';
      mem_read   <= '0';
      next_prel_sel   <= cur_prel_sel;

   when s1 =>

      if (opc(0) = '1') then
         if (opc(1) = '1') then
            next_state <= s3;
         else
            next_state <= s4;
         end if;
      else
         next_state <= s2;
      end if;

      ilu_enable <= '0';
      ld_tbufs   <= '0';
      ld_regs    <= '0';
      ld_accu    <= '0';
      ld_data    <= '0';
      inc_raddr1 <= '0';
      inc_waddr1 <= '0';
      finish     <= '0';
      read_fifo  <= '0';
      write_fifo <= '0';
      write_output1 <= '0';
      mem_read   <= '0';
      next_prel_sel   <= cur_prel_sel;

   when s2 =>

      next_state <= s0;

      ilu_enable <= '0';
      ld_tbufs   <= (not opc(0)) and (not opc(1)) and (not opc(2))
                    after 2 ns;
      ld_regs    <= (not opc(0)) and (not opc(1)) and opc(2)
                    after 2 ns;
      ld_accu    <= (not opc(0)) and opc(1)
```

```
                after 2 ns;
   ld_data    <= '0';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
   finish     <= '0';
   read_fifo  <= '0';
   write_fifo <= '0';
   write_output1 <= '0';
   mem_read   <= '0';
   next_prel_sel   <= cur_prel_sel;

when s3 =>

   if (loop_done = '1') then
      next_state <= s0;
   else
      next_state <= s4;
   end if;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
   finish     <= loop_done;
   read_fifo  <= '0';
   write_fifo <= loop_done;
   write_output1 <= '0';
   mem_read   <= '1';
   next_prel_sel <= cur_prel_sel;

when s4 =>

   if (prel1 = '1') then
      next_state <= p2;
      next_prel_sel   <= "00";
   else
      next_state <= p1;
      next_prel_sel   <= "10";
   end if;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '1';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
```

```
      finish     <= '0';
      read_fifo  <= '0';
      write_fifo <= '0';
      write_output1 <= '0';
      mem_read   <= '1';

   when p1 =>

      if (seq_com = '1') then
         next_state <= s5;
      else
         next_state <= s6;
      end if;

      ilu_enable <= '0';
      ld_tbufs   <= '0';
      ld_regs    <= '0';
      ld_accu    <= '0';
      ld_data    <= '0';
      inc_raddr1 <= '0';
      inc_waddr1 <= '0';
      finish     <= '0';
      read_fifo  <= '0';
      write_fifo <= '0';
      write_output1 <= '0';
      mem_read   <= '0';
      next_prel_sel <= cur_prel_sel;

   when s5 =>

      if (ilu_done = '1') then
         if opc(1) = '1' then
            next_state <= s3;
         else
            next_state <= s0;
         end if;
      else
         next_state <= s5;
      end if;

      ilu_enable <= not ilu_done;
      ld_tbufs   <= '0';
      ld_regs    <= '0';
      ld_accu    <= '0';
      ld_data    <= '0';
      inc_raddr1 <= ilu_done and opc(1);
      inc_waddr1 <= '0';
      finish     <= ilu_done and (not opc(1));
      read_fifo  <= '0';
      write_fifo <= ilu_done and (not opc(1));
```

```
   write_output1 <= '0';
   mem_read   <= '0';
   next_prel_sel <= cur_prel_sel;

when s6 =>

   next_state <= s7;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
   finish     <= '0';
   read_fifo  <= '0';
   write_fifo <= '0';
   write_output1 <= '1';
   mem_read   <= '0';
   next_prel_sel <= cur_prel_sel;

when s7 =>

   if (opc(1) = '1') then
      next_state <= s3;
   else
      next_state <= s0;
   end if;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= opc(1);
   inc_waddr1 <= to_mem;
   finish     <= not opc(1);
   read_fifo  <= '0';
   write_fifo <= not opc(1);
   write_output1 <= '0';
   mem_read   <= '0';
   next_prel_sel <= cur_prel_sel;

when p2 =>

   next_state <= p3;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
```

```
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
   finish     <= '0';
   read_fifo  <= '0';
   write_fifo <= '0';
   write_output1 <= '0';
   mem_read   <= '0';
   next_prel_sel <= cur_prel_sel;

when p3 =>

   if (prel_res = '1') then
      next_state <= p4;
      next_prel_sel <= cur_prel_sel;
   else
      if (prel2 = '1') then
         next_state <= p5;
         next_prel_sel <= "01";
      else
         next_state <= p1;
         next_prel_sel <= "10";
      end if;
   end if;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
   finish     <= '0';
   read_fifo  <= '0';
   write_fifo <= '0';
   write_output1 <= '0';
   mem_read   <= '0';

when p4 =>

   next_state <= s7;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= '0';
```

```
   inc_waddr1 <= '0';
   finish     <= '0';
   read_fifo  <= '0';
   write_fifo <= '0';
   write_output1 <= '1';
   mem_read   <= '0';
   next_prel_sel <= cur_prel_sel;

when p5 =>

   next_state <= p6;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
   finish     <= '0';
   read_fifo  <= '0';
   write_fifo <= '0';
   write_output1 <= '0';
   mem_read   <= '0';
   next_prel_sel <= cur_prel_sel;

when p6 =>

   if (prel_res = '1') then
      next_state <= p7;
      next_prel_sel <= cur_prel_sel;
   else
      next_state <= p1;
      next_prel_sel <= "10";
   end if;

   ilu_enable <= '0';
   ld_tbufs   <= '0';
   ld_regs    <= '0';
   ld_accu    <= '0';
   ld_data    <= '0';
   inc_raddr1 <= '0';
   inc_waddr1 <= '0';
   finish     <= '0';
   read_fifo  <= '0';
   write_fifo <= '0';
   write_output1 <= '0';
   mem_read   <= '0';

when p7 =>
```

```
          next_state <= s7;

          ilu_enable <= '0';
          ld_tbufs   <= '0';
          ld_regs    <= '0';
          ld_accu    <= '0';
          ld_data    <= '0';
          inc_raddr1 <= '0';
          inc_waddr1 <= '0';
          finish     <= '0';
          read_fifo  <= '0';
          write_fifo <= '0';
          write_output1 <= '1';
          mem_read   <= '0';
          next_prel_sel <= cur_prel_sel;

      end case;
    end if;
  end process state_comb;

end behavior;
```

biu.vhd

```
-- File:   biu.vhd
-- Author: CHEN, Qihong,  Portland State University
-- Date:   3/30/97
--
-- Global Control Unit (GCU for short, it was called BIU, Bus Interface Unit)

library ieee;
use ieee.std_logic_1164.all;
use work.parts.all;
use work.ccmtype.all;

entity biu is
  port
   ( -- global signals and input data bus
     reset, clk : in std_logic;
     InstBus : in std_logic_vector (31 downto 23);
     -- InstBus is IBus(31 downto 23) in the thesis.
     DataBus : in std_logic_vector (8 downto 0);
     -- DataBus is IBus(8 downto 0) in the thesis.

     -- signals between GCU and ILU
     loop_done, ilu_done, write_output2, inc_waddr2 : in std_logic;
     ilu_enable : buffer std_logic;
     to_mem : buffer std_logic;
```

```
    ilu_empty : in std_logic;

    -- signals between GCU and the input/output Fifos
    infifoempty : in std_logic;
    read_fifo, write_fifo, finish: out std_logic;

    -- signals to Memory Address Units and Memory Banks from GCU
    ld_addrA, ld_addrB, ld_addrR : out std_logic;
    inc_addrA, inc_addrB : out std_logic;
    MemAwrite, MemAread : out std_logic;
    MemBwrite, MemBread : out std_logic;

    -- Load signals for registers generated by GCU
    ld_accu, ld_data, ld_water, ld_rightedge : out std_logic;
    ld_inst : buffer std_logic;
    ld_prpo : out std_logic; -- PreRelation and PreOperation

    -- multiplexers select signals (Src signals)
    CmpSrc, ASrc, OSrc: out std_logic;

    -- Control signals of Tri-state buffers
    EnAddrA, EnAddrB, EnIFifoA, EnIFifoD : buffer std_logic;
    MemARW, EnIluA, MemBRW, EnIluB       : buffer std_logic;

    -- current state of BIU-CU (just for debug)
    state : out BIUstate;

    -- signals for pre-relation and pre-operation
    prel_res : in std_logic;
    prel_sel : out std_logic_vector (1 downto 0)
  );
end;

architecture arch of biu is

component biu_cu
  port (reset, clk, ilu_done, infifoempty : in std_logic;
        seq_com, loop_done, to_mem: in std_logic;
        opc : in std_logic_vector (0 to 2);
        read_fifo, ilu_enable : out std_logic;
        ld_tbufs, ld_regs, ld_accu, ld_data : out std_logic;
        inc_raddr1, inc_waddr1, finish: out std_logic;
        write_fifo, write_output1, mem_read : out std_logic;
        state : out BIUstate;
        prel1, prel2, prel_res : in std_logic;
        prel_sel : out std_logic_vector(1 downto 0) );
end component;

-- Sequential/combinational operation
signal seq_com : std_logic;
```

```
-- Output control bits
signal to_ofifo, to_accu : std_logic;

-- Load signals
signal ld_tbufs, ld_regs : std_logic;
signal ld_conf, ld_accu1, ld_accu2 : std_logic;

-- Output signals for conf_reg and 3-to-8 decoders
signal conf_reg : std_logic_vector (8 downto 0);
signal decoder_out : std_logic_vector (7 downto 0);

-- Tri-state buffers control bits and Buses status bits
signal ld_EnAddrA, ld_EnAddrB, ld_EnIFifoA, ld_EnIFifoD : std_logic;
signal ld_MemARW, ld_EnIluA, ld_MemBRW, ld_EnIluB : std_logic;

-- tempory tri-state control signals
signal  tEnAddrA, tEnAddrB, tEnIFifoA, tEnIFifoD : std_logic;
signal  tMemARW, tEnIluA, tMemBRW, tEnIluB       : std_logic;

-- Status of the 3 Buses
signal ABusStatus, DBusAStatus, DBusBStatus: std_logic;

-- Singals for Address Units
signal inc_raddr, inc_waddr, inc_waddr1 : std_logic;

-- Signals for memory and outputs
signal mem_read, write_output, write_output1, write_fifo1 : std_logic;

-- Config bits
signal enMemA, enMemB, enFinish : std_logic;

-- temp signal
signal t_ilu_enable : std_logic;

-- pre-relation flags
signal prel1, prel2 : std_logic;

begin

  prel1_dff: dff port map
   ( d=>InstBus(25), clk=>ld_inst, reset=>reset, q=>prel1);

  prel2_dff: dff port map
   ( d=>InstBus(24), clk=>ld_inst, reset=>reset, q=>prel2);

  seq_com_dff: dff port map
   ( d=>InstBus(23), clk=>ld_inst, reset=>reset, q=>seq_com);

  cu: biu_cu port map
```

```
 ( reset=>reset, clk=>clk, ilu_done=>ilu_done, infifoempty=>infifoempty,
   seq_com=>seq_com, loop_done=>loop_done, to_mem=>to_mem,
   opc=>InstBus(31 downto 29),
   read_fifo=>read_fifo, ilu_enable=>t_ilu_enable,
   ld_tbufs=>ld_tbufs, ld_regs=>ld_regs,
   ld_accu=>ld_accu1, ld_data=>ld_data,
   inc_raddr1=>inc_raddr, inc_waddr1=>inc_waddr1, finish=>finish,
   write_fifo=>write_fifo1, write_output1=>write_output1,
   mem_read=>mem_read, state=>state,
   prel1=>prel1, prel2=>prel2, prel_res=>prel_res, prel_sel=>prel_sel);

ilu_enable <= t_ilu_enable;

----------------------------------------------------------------
-- Address pointer
----------------------------------------------------------------

-- Both BIU_CU and ILU_CU are able to generate inc_waddr (call inc_waddr1
-- and inc_waddr2 here), the inc_waddr signal is finally generated here.
inc_waddr <= (inc_waddr1 or (ilu_enable and inc_waddr2)) and (not ilu_empty);

-- inc_addrA is used to increase the address of mem bank A by 1
inc_addrA <= enMemA and
             ((MemARW and inc_raddr) or ((not MemARW) and inc_waddr));

-- inc_addrB is used to increase the address of mem bank B by 1
inc_addrB <= enMemB and
             ((MemBRW and inc_raddr) or ((not MemBRW) and inc_waddr));

----------------------------------------------------------------
-- Load signals for output devices (mem, ofifo, accumulator)
----------------------------------------------------------------

-- Both BIU_CU and ILU_CU are able to generate write_output (call
-- write_output1 and write_output2 here), the write_output signal
-- is finally generated here.
write_output <= write_output1 or (ilu_enable and write_output2);

-- write_fifo is used to write resultant cube to the output FIFO.
write_fifo <= (write_output and to_oFifo and (not ilu_empty))
              or (write_fifo1 and enFinish);

-- ld_accu1 is used to load operand cube into accumulator
-- ld_accu2 is used to load resultant cube back to accumulator
-- ld_accu combines ld_accu1 and ld_accu2 together.
ld_accu2   <= write_output and to_accu;
ld_accu    <= ld_accu1 or ld_accu2;

-- MemAwrite is used to write resultant cube to the mem bank A.
MemAwrite <= write_output and to_mem and (not MemARW) and enMemA
```

```
            and (not ilu_empty) after 3 ns;

-- MemBwrite is used to write resultant cube to the mem bank B.
MemBwrite <= write_output and to_mem and (not MemBRW) and enMemB
             and (not ilu_empty)  after 3 ns;

----------------------------------------------------------------
-- Memory read signals
----------------------------------------------------------------

-- MemAread is used to read operand cube from the mem bank A.
MemAread <= mem_read and MemARW and enMemA after 3 ns;

-- MemBread is used to read operand cube from the mem bank B.
MemBread <= mem_read and MemBRW and enMemB after 3 ns;

----------------------------------------------------------------
-- Config-register is used to store 8 config bits:
-- enMemA, enMemB, CmpSrc, ASrc, OSrc, to_oFifo, to_accu, to_mem
----------------------------------------------------------------

config_reg: regN generic map (Size=>9)  port map
 ( d=>DataBus(8 downto 0), load=>ld_conf, reset=>reset, q=>conf_reg);

enFinish <= conf_reg(8);
enMemA   <= conf_reg(7);
enMemB   <= conf_reg(6);
CmpSrc   <= conf_reg(5);
ASrc     <= conf_reg(4);
OSrc     <= conf_reg(3);
to_oFifo <= conf_reg(2);
to_accu  <= conf_reg(1);
to_mem   <= conf_reg(0);

decoder: decoder3to8 port map
 ( din=>InstBus(28 downto 26), dout=> decoder_out);

ld_addrA     <= ld_regs and decoder_out(0);
ld_addrB     <= ld_regs and decoder_out(1);
ld_addrR     <= ld_regs and decoder_out(2);
ld_water     <= ld_regs and decoder_out(3);
ld_rightedge <= ld_regs and decoder_out(4);
ld_inst      <= ld_regs and decoder_out(5);
ld_conf      <= ld_regs and decoder_out(6);
ld_prpo      <= ld_regs and decoder_out(7);

----------------------------------------------------------------
-- tri-buffers connected to ABus
----------------------------------------------------------------
```

```
-- Avoiding contention on ABus which would result from multiple
-- drivers, see section 5.3.2 of the thesis.

EnAddrA_dff: dff port map
 ( d=>InstBus(25), clk=>ld_EnAddrA, reset=>reset, q=>tEnAddrA);
EnAddrA <= tEnAddrA;

EnAddrB_dff: dff port map
 ( d=>InstBus(25), clk=>ld_EnAddrB, reset=>reset, q=>tEnAddrB);
EnAddrB <= tEnAddrB;

EnIFifoA_dff: dff port map
 ( d=>InstBus(25), clk=>ld_EnIFifoA, reset=>reset, q=>tEnIFifoA);
EnIFifoA <= tEnIFifoA;

ABusStatus  <= not ((tEnAddrA or tEnAddrB or tEnIFifoA) and InstBus(25));
ld_EnAddrA  <= decoder_out(0) and ABusStatus and ld_tbufs after 3 ns;
ld_EnAddrB  <= decoder_out(1) and ABusStatus and ld_tbufs after 3 ns;
ld_EnIFifoA <= decoder_out(2) and ABusStatus and ld_tbufs after 3 ns;

----------------------------------------------------------------
-- tri-buffers connected to DBusA
----------------------------------------------------------------

-- Avoiding contention on DBusA which would result from multiple
-- drivers, see section 5.3.2 of the thesis.

MemARW_dff: dff port map
 ( d=>InstBus(25), clk=>ld_MemARW, reset=>reset, q=>tMemARW);
MemARW <= tMemARW;

EnIluA_dff: dff port map
 ( d=>InstBus(25), clk=>ld_EnIluA, reset=>reset, q=>tEnIluA);
EnIluA <= tEnIluA;

EnIFifoD_dff: dff port map
 ( d=>InstBus(25), clk=>ld_EnIFifoD, reset=>reset, q=>tEnIFifoD);
EnIFifoD <= tEnIFifoD;

DBusAStatus <= not ((tMemARW or tEnIluA or tEnIFifoD) and InstBus(25));
ld_MemARW   <= decoder_out(3) and DBusAStatus and ld_tbufs after 3 ns;
ld_EnIluA   <= decoder_out(4) and DBusAStatus and ld_tbufs after 3 ns;
ld_EnIFifoD <= decoder_out(5) and DBusAStatus and ld_tbufs after 3 ns;

----------------------------------------------------------------
-- tri-buffers connected to DBusB
----------------------------------------------------------------

-- Avoiding contention on DBusB which would result from multiple
-- drivers, see section 5.3.2 of the thesis.
```

```
  MemBRW_dff: dff port map
   ( d=>InstBus(25), clk=>ld_MemBRW, reset=>reset, q=>tMemBRW);
  MemBRW <= tMemBRW;

  EnIluB_dff: dff port map
   ( d=>InstBus(25), clk=>ld_EnIluB, reset=>reset, q=>tEnIluB);
  EnIluB <= tEnIluB;

  DBusBStatus <= not ((tMemBRW or tEnIluB) and InstBus(25));
  ld_MemBRW   <= decoder_out(6) and DBusBStatus and ld_tbufs after 3 ns;
  ld_EnIluB   <= decoder_out(7) and DBusBStatus and ld_tbufs after 3 ns;

end arch;
```

ccm.vhd

```
-- File:   ccm.vhd
-- Author: CHEN, Qihong,  Portland State University
-- Date:   3/30/97
--
-- Cube Calculus Machine Version 2 (CCM2 for short)
--
-- In this file, all components of CCM2 are combined together according
-- to figure 5.3 of the thesis. The reader should compare this code with
-- the figure 5.3 to understand it.
--
-- Signals:
--   ififo_din: the input of the input FIFO.
--   ofifo_dout: the output of the output FIFO.
--   ififo_we: write enable signal of the input FIFO.
--   ififo_ff: full flag signal of the input FIFO.
--   ofifo_re: read enable signal of the output FIFO.
--   ofifo_ef: empty flag signal of the output FIFO.

library ieee;
use ieee.std_logic_1164.all;
use work.all;
use work.parts.all;
use work.ccmbasic.all;
use work.ccmtype.all;

entity ccm is
  port ( -- global signals
         reset, clk : in std_logic;

         -- the input/output FIFOs
         ififo_din : in std_logic_vector (31 downto 0);
         ofifo_dout : out std_logic_vector (31 downto 0);
```

```
        ififo_we, ofifo_re : in std_logic;
        ififo_ff, ofifo_ef : buffer std_logic;

        -- current state of BIU (just for debug)
        state : out BIUstate
      );
end;

architecture arch of ccm is

  constant memSize : integer := 64;

  -- Bus signals
  signal IBus, OBus : std_logic_vector (31 downto 0);
  signal ABus : std_logic_vector (17 downto 0);
  signal DBusA, DBusB : std_logic_vector (29 downto 0);

  -- Tri-state buffer enable signals
  signal EnAddrA, EnAddrB, EnIFifoA, EnIFifoD : std_logic;
  signal MemARW, EnIluA, MemBRW, EnIluB       : std_logic;

  -- multiplexers select signals
  signal CmpSrc, ASrc, OSrc: std_logic;

  -- register/address unit load signals
  signal ld_accu, ld_data, ld_water, ld_rightedge, ld_inst : std_logic;
  signal ld_addrA, ld_addrB, ld_addrR : std_logic;

  -- memory address unit inc signals
  signal inc_addrA, inc_addrB : std_logic;

  -- memory read/write signals
  signal MemAwrite, MemAread, MemBwrite, MemBread : std_logic;

  -- signals related to the input/output Fifos
  signal ififo_ef, ofifo_ff, ififo_re, ofifo_we : std_logic;

  -- signals between BIU and ILU
  signal ilu_enable, ilu_done, write_output2, inc_waddr2, to_mem: std_logic;
  signal ilu_empty : std_logic;
  signal cnt_val : std_logic_vector(3 downto 0);
  signal cmp_q : std_logic_vector(0 to 3);

  -- enent flag signal
  signal addrEQ, finish : std_logic;

  -- pre-relation and pre-operation
  signal ld_prpo : std_logic;
  signal prpo_q : std_logic_vector (23 downto 0);
  signal rel, bef, tmp_and_or : std_logic_vector (0 to 3);
```

```
  signal and_or, prel_res : std_logic;
  signal prel_sel: std_logic_vector (1 downto 0);
  signal fourzero: std_logic_vector (0 to 3) := "0000";
  signal prel_val : std_logic_vector (3 downto 0);
  signal res_sel : std_logic_vector (1 downto 0);

  -- others
  signal const_one  : std_logic := '1';
  signal const_zero : std_logic := '0';
  signal addrA_q, addrB_q, addrR_q, CmpMux_q : std_logic_vector(17 downto 0);

  signal memaBus, membBus : std_logic_vector (31 downto 0);
  signal memaDump, membDump : std_logic := '0';
  signal a_dump_start, b_dump_start : integer := 0;
  signal a_dump_end, b_dump_end : integer := memSize;
  signal MemARWn, MemBRWn : std_logic;
  signal OMux_q, AMux_q : std_logic_vector (29 downto 0);

  signal accu_q, data_q : std_logic_vector (29 downto 0);
  signal water_q, right_q : std_logic_vector (14 downto 0);
  signal inst_q : std_logic_vector (17 downto 0);

  signal addra_clk, addrb_clk : std_logic;

begin

  biu: ccmbasic.biu port map
   ( reset=>reset, clk=>clk,
     InstBus=>IBus(31 downto 23), DataBus=>IBus(8 downto 0),
     loop_done=>AddrEQ, ilu_done=>ilu_done, write_output2=>write_output2,
     inc_waddr2=>inc_waddr2, ilu_enable=>ilu_enable, to_mem=>to_mem,
     infifoempty=>ififo_ef, read_fifo=>ififo_re,
     write_fifo=>ofifo_we,  finish=>finish,
     ld_addrA=>ld_addrA, ld_addrB=>ld_addrB, ld_addrR=>ld_addrR,
     inc_addrA=>inc_addrA, inc_addrB=>inc_addrB,
     MemAwrite=>MemAwrite, MemAread=>MemAread,
     MemBwrite=>MemBwrite, MemBread=>MemBread,
     ld_accu=>ld_accu, ld_data=>ld_data, ld_water=>ld_water,
     ld_rightedge=>ld_rightedge, ld_inst=>ld_inst,
     CmpSrc=>CmpSrc, ASrc=>ASrc, OSrc=>OSrc,
     EnAddrA=>EnAddrA, EnAddrB=>EnAddrB, EnIFifoA=>EnIFifoA,
     EnIFifoD=>EnIFifoD, MemARW=>MemARW, EnIluA=>EnIluA,
     MemBRW=>MemBRW, EnIluB=>EnIluB, state=>state, ilu_empty=>ilu_empty,
     ld_prpo=>ld_prpo, prel_res=>prel_res, prel_sel=>prel_sel
   );

  tbuf_ibus_A : parts.tbufN generic map (18) port map
   ( en=>EnIFifoA, din=>IBus(17 downto 0), dout=>ABus );

  addra_clk <= inc_addrA or ld_AddrA;
```

```
addrA: parts.counterN generic map (18) port map
 ( reset=>reset, clk=>addra_clk, ld=>ld_addrA, ce=>const_one,
   d=>ABus, q=>addrA_q );

tbuf_addrA: parts.tbufN generic map (18) port map
 ( en=>EnAddrA, din=>addrA_q, dout=>ABus );

addrb_clk <= inc_addrB or ld_AddrB;
addrB: parts.counterN generic map (18) port map
 ( reset=>reset, clk=>addrb_clk, ld=>ld_addrB, ce=>const_one,
   d=>ABus, q=>addrB_q );

tbuf_addrB: parts.tbufN generic map (18) port map
 ( en=>EnAddrB, din=>addrB_q, dout=>ABus );

addrR: parts.regN generic map (18) port map
 ( reset=>reset, load=>ld_addrR, d=>ABus, q=>addrR_q );

mux_cmp: parts.mux21N generic map (18) port map
 ( din0=>addrA_q, din1=>addrB_q, sel=>CmpSrc, dout=>CmpMux_q );

equ: parts.equalN generic map (18) port map
 ( din0=>CmpMux_q, din1=>addrR_q, o=>addrEQ );

mema: ccmbasic.ram
  generic map
   ( Size=>memSize, AddrWidth=>18, DataWidth=>32, download_on_power_up=>false)
  port map
   ( ce=>const_one, memread=>MemAread, memwrite=>MemAwrite,
     clk=>clk, addr=>addrA_q, dbus=>memaBus, dump=>memaDump,
     dump_start=>a_dump_start, dump_end=>a_dump_end
   );

memb: ccmbasic.ram
  generic map
   ( Size=>memSize, AddrWidth=>18, DataWidth=>32, download_on_power_up=>false)
  port map
   ( ce=>const_one, memread=>MemBread, memwrite=>MemBwrite,
     clk=>clk, addr=>addrB_q, dbus=>membBus, dump=>membDump,
     dump_start=>b_dump_start, dump_end=>b_dump_end
   );

tbuf_mema_r: parts.tbufN generic map (30) port map
 ( en=>MemARW, din=>memaBus(29 downto 0), dout=>DBusA );

tbuf_mema_w: parts.tbufN generic map (30) port map
 ( en=>MemARWn, din=>DBusA, dout=>memaBus(29 downto 0) );
MemARWn <= not MemARW after 1 ns;

tbuf_memb_r: parts.tbufN generic map (30) port map
```

```
 ( en=>MemBRW, din=>membBus(29 downto 0), dout=>DBusB );

tbuf_memb_w: parts.tbufN generic map (30) port map
 ( en=>MemBRWn, din=>DBusB, dout=>membBus(29 downto 0) );
MemBRWn <= not MemBRW after 1 ns;

tbuf_ibus_D : parts.tbufN generic map (30) port map
 ( en=>EnIFifoD, din=>IBus(29 downto 0), dout=>DBusA );

mux_o: parts.mux21N generic map (30) port map
 ( din0=>DBusA, din1=>DBusB, sel=>OSrc, dout=>OMux_q );

mux_a: parts.mux21N generic map (30) port map
 ( din0=>DBusA, din1=>DBusB, sel=>ASrc, dout=>AMux_q );

reg_accu: parts.regN generic map (30) port map
 ( reset=>reset, load=>ld_accu, d=>AMux_q, q=>accu_q );

reg_data: parts.regN generic map (30) port map
 ( reset=>reset, load=>ld_data, d=>OMux_q, q=>data_q );

reg_water: parts.regN generic map (15) port map
 ( reset=>reset, load=>ld_water, d=>OMux_q(14 downto 0), q=>water_q );

reg_rightedge: parts.regN generic map (15) port map
 ( reset=>reset, load=>ld_rightedge, d=>OMux_q(14 downto 0), q=>right_q );

reg_inst: parts.regN generic map (18) port map
 ( reset=>reset, load=>ld_inst, d=>OMux_q(17 downto 0), q=>inst_q );

reg_prpo: parts.regN generic map (24) port map
 ( reset=>reset, load=>ld_prpo, d=>OMux_q(23 downto 0), q=>prpo_q );

mux_rel: parts.mux441 port map
 ( din0=>prpo_q(22 downto 19), din1=>prpo_q(10 downto 7),
   din2=>inst_q(15 downto 12), din3=>fourzero,
   sel=>prel_sel, dout=>rel  );

mux_bef: parts.mux441 port map
 ( din0=>prpo_q(15 downto 12), din1=>prpo_q(3 downto 0),
   din2=>inst_q(11 downto 8), din3=>fourzero,
   sel=>prel_sel, dout=>bef  );

tmp_and_or <= prpo_q(23) & prpo_q(11) & inst_q(16) & const_zero;

mux_and_or:parts.mux41 port map
 ( din=>tmp_and_or, sel=>prel_sel, dout=>and_or);

ilu: ccmbasic.ilu generic map (15) port map
 ( reset=>reset, clk=>clk, ilu_enable=>ilu_enable,
```

```
   prime=>inst_q(17), to_mem=>to_mem, and_or=>and_or,
   rel=>rel, bef=>bef,
   act=>inst_q(7 downto 4), aft=>inst_q(3 downto 0),
   water=>water_q, redge=>right_q, a=>accu_q, b=>data_q,
   c=>OBus(29 downto 0), ilu_done=>ilu_done,
   write_output=>write_output2, inc_waddr=>inc_waddr2, empty=>ilu_empty,
   cnt_val=>cnt_val
 );

mux_pval: parts.mux21N generic map (size=>1) port map
 ( din0=>prpo_q(16 downto 16), din1=>prpo_q(4 downto 4),
   sel=>prel_sel(0), dout=>prel_val(0 downto 0));

prel_val(3 downto 1) <= "000";

cmp_prel: parts.cmpN generic map (size=>4) port map
 ( dinA=>cnt_val, dinB=>prel_val,
   lt=>cmp_q(0), eq=>cmp_q(1), gt=>cmp_q(2));

cmp_q(3) <= '0';

mux_res_sel: parts.mux21N generic map (size=>2) port map
 ( din0=>prpo_q(18 downto 17), din1=>prpo_q(6 downto 5),
   sel=>prel_sel(0), dout=>res_sel);

mux_prel_res: parts.mux41 port map
 ( din=>cmp_q, sel=>res_sel, dout=>prel_res);

tbuf_IluA: parts.tbufN generic map (30) port map
 ( en=>EnIluA, din=>OBus(29 downto 0), dout=>DBusA );

tbuf_IluB: parts.tbufN generic map (30) port map
 ( en=>EnIluB, din=>OBus(29 downto 0), dout=>DBusB );

ififo: ccmbasic.fifo
  generic map (Width=>32, depth=>512)
  port map
   ( data=>ififo_din, q=>IBus, clk=>clk, reset=>reset,
     re=>ififo_re, we=>ififo_we, ef=>ififo_ef, ff=>ififo_ff
   );

ofifo: ccmbasic.fifo
  generic map (Width=>32, depth=>64)
  port map
   ( data=>OBus, q=>ofifo_dout, clk=>clk, reset=>reset,
     re=>ofifo_re, we=>ofifo_we, ef=>ofifo_ef, ff=>ofifo_ff
   );

OBus(30) <= const_zero;
OBus(31) <= finish;
```

```
end arch;
```

testccm.vhd

```
-- File:   testccm.vhd
-- Author: CHEN, Qihong,  Portland State University
-- Date:   3/30/97
--
-- Test bench of CCM2

library ieee;
use std.textio.all;
use ieee.std_logic_textio.all;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.stringpkg.all;
use work.ccmtype.all;

entity testccm is
end testccm;

architecture test of testccm is

component ccm
  port ( -- global signals
         reset, clk : in std_logic;

         -- the input/output fifos
         ififo_din : in std_logic_vector (31 downto 0);
         ofifo_dout : out std_logic_vector (31 downto 0);
         ififo_we, ofifo_re : in std_logic;
         ififo_ff, ofifo_ef : buffer std_logic;
         state : out BIUstate
       );
end component;

constant cmd_filename: string := "test.ccm";
signal clk: std_logic := '0';
signal reset : std_logic;
signal ififo_din, ofifo_dout : std_logic_vector (31 downto 0);
signal ififo_we, ofifo_re : std_logic := '0';
signal ififo_ff, ofifo_ef : std_logic;
signal state : BIUstate;
signal init_done : boolean := false;

BEGIN

  u0: ccm port map
```

```
 ( reset=>reset, clk=>clk,
   ififo_din=>ififo_din, ofifo_dout=>ofifo_dout,
   ififo_we=>ififo_we,   ofifo_re=>ofifo_re,
   ififo_ff=>ififo_ff,   ofifo_ef=>ofifo_ef,
   state=>state
 );

-- clock process is used to generated clock signal.
clock: process
begin
  clk <= not clk after 50 ns;
  wait for 50 ns;
end process;

-- read output FIFO whenever there is a cube in it.
read_ofifo : process
  variable outbuf: line;
  variable resultno: integer := 0;
begin

  wait for 100 ns;

  if (init_done) then
     if (ofifo_re = '1') then
        resultno := resultno + 1;
        write(outbuf, string'(" [Time: "));
        write(outbuf, now);
        write(outbuf, string'("] result cube (No."));
        write(outbuf, resultno);
        write(outbuf, string'("): "));
        write(outbuf, ofifo_dout(31 downto 30));
        write(outbuf, string'("-"));
        write(outbuf, ofifo_dout(29 downto 22));
        write(outbuf, string'("-"));
        write(outbuf, ofifo_dout(21 downto 14));
        write(outbuf, string'("-"));
        write(outbuf, ofifo_dout(13 downto 6));
        write(outbuf, string'("-"));
        write(outbuf, ofifo_dout(5 downto 0));
        writeline(output, outbuf);
     end if;

     if (ofifo_ef = '1') then
        ofifo_re <= '0';
     else
        ofifo_re <= '1';
     end if;
  end if;
end process;
```

```
-- This is the main process of the test bench.
verify: process
  type CCMSymbol is
   ( -- command symbols
     ccmSET, ccmENABLE, ccmDISABLE, ccmEXEC, ccmLOOP,

     -- registers
     regAddrA, regAddrB, regAddrR, regWater, regRight,
     regInst, regConf, regACCU, regPRPO,

     -- tri-buffers
     tbufAddrA, tbufAddrB, tbufIFifoA, tbufIFifoD,
     tbufMemARW, tbufMemBRW, tbufIluA, tbufIluB,

     -- others
     ccmINVALID
   );

  -- convert ccm assembly command into CCMSymbol.
  procedure to_command(token: inout line; symID: out CCMSymbol) is
    variable cmpret : boolean;
  begin

    cmp_string(token, string'("SET"), cmpret);
    if (cmpret) then
       symID := ccmSET;
    else
       cmp_string(token, string'("ENABLE"), cmpret);
       if (cmpret) then
          symID := ccmENABLE;
       else
          cmp_string(token, string'("DISABLE"), cmpret);
          if (cmpret) then
             symID := ccmDISABLE;
          else
             cmp_string(token, string'("EXEC"), cmpret);
             if (cmpret) then
                symID := ccmEXEC;
             else
                cmp_string(token, string'("LOOP"), cmpret);
                if (cmpret) then
                   symID := ccmLOOP;
                else
                   symID := ccmINVALID;
                end if;
             end if;
          end if;
       end if;
    end if;
    Deallocate(token);
```

```
end to_command;

-- convert register name into CCMSymbol.
procedure to_register(token: inout line; symID: out CCMSymbol) is
  variable cmpret : boolean;
begin

  cmp_string(token, string'("AddrA"), cmpret);
  if (cmpret) then
     symID := regAddrA;
  else
     cmp_string(token, string'("AddrB"), cmpret);
     if (cmpret) then
        symID := regAddrB;
     else
        cmp_string(token, string'("AddrR"), cmpret);
        if (cmpret) then
           symID := regAddrR;
        else
           cmp_string(token, string'("Water"), cmpret);
           if (cmpret) then
              symID := regWater;
           else
              cmp_string(token, string'("Right"), cmpret);
              if (cmpret) then
                 symID := regRight;
              else
                 cmp_string(token, string'("Inst"), cmpret);
                 if (cmpret) then
                    symID := regInst;
                 else
                   cmp_string(token, string'("Conf"), cmpret);
                   if (cmpret) then
                      symID := regConf;
                   else
                     cmp_string(token, string'("Accu"), cmpret);
                     if (cmpret) then
                        symID := regAccu;
                     else
                        cmp_string(token, string'("Prpo"), cmpret);
                        if (cmpret) then
                           symID := regPRPO;
                        else
                           symID := ccmINVALID;
                        end if;
                     end if;
                   end if;
                 end if;
              end if;
           end if;
```

```
        end if;
     end if;
  end if;
  Deallocate(token);
end to_register;

-- convert tri-state buffer name into CCMSymbol.
procedure to_tribuf(token: inout line; symID: out CCMSymbol) is
  variable cmpret : boolean;
begin

  cmp_string(token, string'("enIFifoA"), cmpret);
  if (cmpret) then
     symID := tbufIFifoA;
  else
     cmp_string(token, string'("enIFifoD"), cmpret);
     if (cmpret) then
        symID := tbufIFifoD;
     else
        cmp_string(token, string'("enIluA"), cmpret);
        if (cmpret) then
           symID := tbufIluA;
        else
           cmp_string(token, string'("enIluB"), cmpret);
           if (cmpret) then
              symID := tbufIluB;
           else
              cmp_string(token, string'("enAddrA"), cmpret);
              if (cmpret) then
                 symID := tbufAddrA;
              else
                 cmp_string(token, string'("enAddrB"), cmpret);
                 if (cmpret) then
                    symID := tbufAddrB;
                 else
                   cmp_string(token, string'("MemARW"), cmpret);
                   if (cmpret) then
                      symID := tbufMemARW;
                   else
                     cmp_string(token, string'("MemBRW"), cmpret);
                     if (cmpret) then
                        symID := tbufMemBRW;
                     else
                        symID := ccmINVALID;
                     end if;
                   end if;
                 end if;
              end if;
           end if;
        end if;
```

```
      end if;
    end if;
    Deallocate(token);
  end to_tribuf;

  file ifile : text is in cmd_filename;
  variable inbuf, outbuf, tempstr, token : line;
  variable good, equal, valid_cmd : boolean := false;
  variable lineno, state_s0_no : integer := 0;
  variable ch : character;
  variable cursymID, argsymID : CCMSymbol;
  variable vec_addra, vec_addrb, vec_addrr : std_logic_vector(1 to 18);
  variable vec_water, vec_right : std_logic_vector(1 to 15);
  variable vec_inst : std_logic_vector(1 to 21);
  variable vec_conf : std_logic_vector(1 to 9);
  variable vec_accu, vec_data : std_logic_vector(1 to 30);
  variable vec_prpo : std_logic_vector(1 to 24);
  variable encoded_cmd : std_logic_vector(31 downto 0);

begin
  -- initialize the CCM.
  if (not init_done) then
    -- ififo_we <= '0';
    -- ofifo_re <= '0';
    reset <= '1';
    wait for 75 ns;
    reset <= '0';
    wait for 30 ns;
    write(outbuf, string'("Initialize the CCM ..."));
    writeline(output, outbuf);
    init_done <= true;
  end if;

  -- the following while loop read one line of CCM assembly, converts
  -- into CCM instruction represented by binary number. If it encounts
  -- a line of invalid CCM assembly (like comments), it will read next
  -- line of CCM assembly.

  while not endfile(ifile) loop

    write(outbuf, string'("[Time: "));
    write(outbuf, now);
    write(outbuf, string'("] read command"));
    writeline(output, outbuf);

    lineno := lineno + 1;
    readline(ifile, inbuf);
    tempstr := new string'(inbuf.all);
    write(outbuf, lineno, right, 3);
    write(outbuf, string'(". "));
```

```
write(outbuf, string'(tempstr.all));
writeline(output, outbuf);

read(inbuf, ch, good);
if (not good) or (ch /= ' ') then next; end if;

get_string(inbuf, token, good);
if (not good) then next; end if;

-- encoded_cmd := "00000000000000000000000000000000";
encoded_cmd := (others => '0');
valid_cmd := true;
to_command(token, cursymID);
case cursymID is
  when ccmSET =>
    get_string(inbuf, token, good);
    if (not good) then
       write(outbuf, string'(" CCM ERROR: Invalid set command."));
       writeline(output, outbuf);
       next;
    end if;
    encoded_cmd(31 downto 29) := "001";
    to_register(token, argsymID);
    case argsymID is
      when regAddrA =>
        read_std_logic_vector(inbuf, vec_addra, '0');
        encoded_cmd(28 downto 26) := "000";
        encoded_cmd(17 downto 0) := vec_addra;
        write(outbuf, string'(" encoded command is "));
        write(outbuf, encoded_cmd);
        writeline(output, outbuf);
      when regAddrB =>
        read_std_logic_vector(inbuf, vec_addrb, '0');
        encoded_cmd(28 downto 26) := "001";
        encoded_cmd(17 downto 0) := vec_addrb;
        write(outbuf, string'(" encoded command is "));
        write(outbuf, encoded_cmd);
        writeline(output, outbuf);
      when regAddrR =>
        read_std_logic_vector(inbuf, vec_addrr, '0');
        encoded_cmd(28 downto 26) := "010";
        encoded_cmd(17 downto 0) := vec_addrr;
        write(outbuf, string'(" encoded command is "));
        write(outbuf, encoded_cmd);
        writeline(output, outbuf);
      when regWater =>
        read_std_logic_vector(inbuf, vec_water, '1');
        encoded_cmd(28 downto 26) := "011";
        encoded_cmd(14 downto 0) := vec_water;
        write(outbuf, string'(" encoded command is "));
```

```
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when regRight =>
      read_std_logic_vector(inbuf, vec_right, '0');
      encoded_cmd(28 downto 26) := "100";
      encoded_cmd(14 downto 0) := vec_right;
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when regInst  =>
      read_std_logic_vector(inbuf, vec_inst, '0');
      encoded_cmd(28 downto 26) := "101";
      encoded_cmd(25 downto 23) := vec_inst(1 to 3);
      encoded_cmd(17 downto 0) := vec_inst(4 to 21);
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when regConf  =>
      read_std_logic_vector(inbuf, vec_conf, '0');
      encoded_cmd(28 downto 26) := "110";
      encoded_cmd(8 downto 0) := vec_conf;
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when regAccu  =>
      read_std_logic_vector(inbuf, vec_accu, '0');
      encoded_cmd(31 downto 30) := "01";
      encoded_cmd(29 downto 0) := vec_accu;
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when regPRPO  =>
      read_std_logic_vector(inbuf, vec_prpo, '0');
      encoded_cmd(28 downto 26) := "111";
      encoded_cmd(23 downto 0) := vec_prpo;
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when others   =>
      write(outbuf, string'(" invalid set command. "));
      writeline(output, outbuf);
      valid_cmd := false;
  end case;
when ccmENABLE =>
  get_string(inbuf, token, good);
  if (not good) then
     write(outbuf, string'(" CCM ERROR: Invalid enable command."));
     writeline(output, outbuf);
     next;
  end if;
```

```
  encoded_cmd(31 downto 29) := "000";
  encoded_cmd(25) := '1';
  to_tribuf(token, argsymID);
  case argsymID is
    when tbufAddrA =>
      encoded_cmd(28 downto 26) := "000";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when tbufAddrB =>
      encoded_cmd(28 downto 26) := "001";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when tbufIFifoA =>
      encoded_cmd(28 downto 26) := "010";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when tbufMemARW =>
      encoded_cmd(28 downto 26) := "011";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when tbufIluA =>
      encoded_cmd(28 downto 26) := "100";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when tbufIFifoD =>
      encoded_cmd(28 downto 26) := "101";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when tbufMemBRW  =>
      encoded_cmd(28 downto 26) := "110";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when tbufIluB =>
      encoded_cmd(28 downto 26) := "111";
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when others   =>
      write(outbuf, string'(" invalid enable command. "));
      writeline(output, outbuf);
      valid_cmd := false;
  end case;
when ccmDISABLE =>
```

```
get_string(inbuf, token, good);
if (not good) then
   write(outbuf, string'(" CCM ERROR: Invalid disable command."));
   writeline(output, outbuf);
   next;
end if;
encoded_cmd(31 downto 29) := "000";
to_tribuf(token, argsymID);
case argsymID is
  when tbufAddrA =>
    encoded_cmd(28 downto 26) := "000";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when tbufAddrB =>
    encoded_cmd(28 downto 26) := "001";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when tbufIFifoA =>
    encoded_cmd(28 downto 26) := "010";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when tbufMemARW =>
    encoded_cmd(28 downto 26) := "011";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when tbufIluA =>
    encoded_cmd(28 downto 26) := "100";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when tbufIFifoD =>
    encoded_cmd(28 downto 26) := "101";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when tbufMemBRW  =>
    encoded_cmd(28 downto 26) := "110";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when tbufIluB =>
    encoded_cmd(28 downto 26) := "111";
    write(outbuf, string'(" encoded command is "));
    write(outbuf, encoded_cmd);
    writeline(output, outbuf);
  when others   =>
```

```
          write(outbuf, string'(" invalid disable command. "));
          writeline(output, outbuf);
          valid_cmd := false;
      end case;
    when ccmEXEC =>
      read_std_logic_vector(inbuf, vec_data, '0');
      encoded_cmd(31 downto 30) := "10";
      encoded_cmd(29 downto 0) := vec_data;
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when ccmLOOP =>
      read_std_logic_vector(inbuf, vec_data, '0');
      encoded_cmd(31 downto 30) := "11";
      encoded_cmd(29 downto 0) := vec_data;
      write(outbuf, string'(" encoded command is "));
      write(outbuf, encoded_cmd);
      writeline(output, outbuf);
    when others =>
      write(outbuf, string'(" invalid command. "));
      writeline(output, outbuf);
      valid_cmd := false;
  end case;

  if (not valid_cmd) then next; end if;

  -- push the command into the input fifo
  ififo_we <= '1';
  ififo_din <= encoded_cmd;
  wait for 100 ns;
end loop;   -- read command file

ififo_we <= '0';

-- if CCM keeps in state S0 in two continuous clock periods, then
-- the simulation is done.
while ( state_s0_no < 2 ) loop
  if (state = s0) then
     state_s0_no := state_s0_no + 1;
  else
     state_s0_no := 0;
  end if;
  wait for 100 ns;
end loop;

-- simulation is done
write(outbuf, string'(" Time: "));
write(outbuf, now);
writeline(output, outbuf);
assert false
```

```
      report "Simulation is done."
      severity note;
    wait;
  end process;

END;
```

APPENDIX C

The C program to perform disjoint-sharp operation

This is a C program to perform disjoint-sharp operation on two arrays of cubes, See Chapter 8 for detail. This program consists of three files: main.c, cubeoper.h, cubeoper.c.

MAIN.C

```
/*  File: main.c,    Chen, Qihong    Apr 1, 1998
**
**  This program is used to perform disjoint sharp operation on two
**  arrays of cubes. Please note that this program only handles binary
**  variables.
**
**  This program reads two arrays of cubes from two files. The file
**  format is similar to PLA file format, but it only support
**  ".i", ".e" and cubes in PLA format. There is only input cubes
**  in this file format.
**
**  Example: the follow lines represent a array of cubes with 3 variables
**     .i 3
**     000
**     011
**     11-
**     .e
*/

#include <stdio.h>
#include <sys/time.h>
#include "cubeoper.h"

int debug = 0;

main(int argc, char *argv[])
{
  ARRAYOFCUBES a, b, temp;
  FILE *fp1, *fp2;
```

```
struct timeval  tv1,tv2,tvdiff;
struct timezone tz;
int res1, res2;

/* initialize arrayofcubes */
a.numvar  = 0;      a.numcube = 0;
b.numvar  = 0;      b.numcube = 0;
temp.numvar  = 0;   temp.numcube = 0;

if (argc<3 || argc>4)
 { printf("Usage: dsharp <infile1> <infile2> <d>\n");
   exit(0);
 }

if (argc==4)
  if (strcmp(argv[3],"d")==0) debug=1;

if ((fp1=fopen(argv[1],"r")) == 0)
 { printf("cannot open file %s\n", argv[1]);
   exit(0);
 }

read_cubes(fp1, &a);
fclose(fp1);
if (debug) showarray(&a, "A");

if ((fp2=fopen(argv[2],"rw")) == 0)
 { printf("cannot open file %s\n", argv[2]);
   exit(0);
 }

read_cubes(fp2, &b);
fclose(fp2);
if (debug) showarray(&b, "B");

res1 = gettimeofday(&tv1, &tz);
dsharparr(&a, &b, &temp);
res2 = gettimeofday(&tv2, &tz);
showarray(&temp, "Result");

if (debug)
 { if (! res1)
     printf("start: %ld sec %ld usec\n", tv1.tv_sec, tv1.tv_usec);
   else
     printf("first gettimeofday failed\n");

   if (! res2)
     printf("  end: %ld sec %ld usec\n", tv2.tv_sec, tv2.tv_usec);
   else
```

```
      printf("second gettimeofday failed\n");
    }

  difftv(&tv1, &tv2, &tvdiff);
  printf("time difference is %ld sec %ld usec\n",tvdiff.tv_sec,tvdiff.tv_usec);
}
```

CUBEOPER.H

```
/*  File: cubeoper.h */
#include <sys/time.h>
#define MAXCUBES 1024   /* the max cubes in a array of cubes */

/* The cube is represent by a long int(4-byte, 32-bit). Since this program
** only handle binary variables, a cube has at most 16 binary variables.
** The 32 bits are used from left to right, which means the first cube is
** represented by the highest two bits.
*/
typedef unsigned long CUBE;

/* The structure is used to represent an array of cubes */
struct ArrayOfCubes
 unsigned char numvar;
  int           numcube;
  CUBE          cube[MAXCUBES];
;

typedef struct ArrayOfCubes ARRAYOFCUBES;

/* Functions' definition. see file cubeoper.c */
void getvar(CUBE *cube, short idx, short *value);
void setvar(CUBE *cube, short idx, short *value);
void storecube(ARRAYOFCUBES *parray, CUBE *pcube);
void skip_line(FILE *fp);
char *get_word(FILE *fp, char *word);
int str2pn(char *str, int nvar, CUBE *pcube);
void pn2str(CUBE *pcube, int nvar, char *str);
void read_cubes(FILE *fp, ARRAYOFCUBES *cubes);
void showarray(ARRAYOFCUBES *parray, char *name);
void dsharp(CUBE cubea, CUBE cubeb, int numvar, ARRAYOFCUBES *parray);
void copyarrcube(ARRAYOFCUBES *parraya, ARRAYOFCUBES *parrayb);
void dsharparr(ARRAYOFCUBES *parraya,ARRAYOFCUBES *parrayb,ARRAYOFCUBES *pres);
void difftv(struct timeval *tv1, struct timeval *tv2, struct timeval *tvdiff);
```

CUBEOPER.C

```
/* File: cubeoper.c */

#include <stdio.h>
```

```
#include "cubeoper.h"

extern int debug;

/* get a literal of the given cube */
void getvar(CUBE *pcube, short idx, short *pvalue)
{ CUBE temp;

  temp = *pcube;
  *pvalue = (temp >> (30-idx*2)) & 0x03L;
}

/* set a literal of the given cube */
void setvar(CUBE *pcube, short idx, short *pvalue)
{ CUBE temp;

  temp = *pvalue & 0x0003;
  *pcube = (*pcube) & (~(0x03L << (30-idx*2)));
  *pcube = *pcube | (temp << (30-idx*2));
}

/* store a cube into the given array of cubes */
void storecube(ARRAYOFCUBES *parray, CUBE *pcube)
{
  parray->cube[parray->numcube] = *pcube;
  parray->numcube += 1;
}

/* skip the rest of line when read file */
void skip_line(FILE *fp)
{ int ch;

  while ((ch=getc(fp)) != EOF && ch != '\n');
}

/* get a word from the file, the words are seperated by white spaces */
char *get_word(FILE *fp, char *word)
{ int ch, i = 0;

  while ((ch = getc(fp)) != EOF && isspace(ch));

  word[i++] = ch;
  while ((ch = getc(fp)) != EOF && ! isspace(ch))
   { word[i++] = ch;
   }
  word[i++] = '\0';

  return word;
}
```

```
/* convert a cube from PLA format to positional notation format */
int str2pn(char *str, int nvar, CUBE *pcube)
{ short i, val, res=0;

  *pcube = 0L;

  if ( (strlen(str) != nvar) || (nvar >= 16))
   { printf("wrong length (%s,%d).\n", str, nvar);
     res = 1;
   }
  else
   { for (i=0; i<nvar; i++)
      { switch (str[i])
         { case '0': val=2; break;
           case '1': val=1; break;
           case '-': val=3; break;
           default : res = 2;
         }
        setvar(pcube, i, &val);
      }
   }

  return res;
}

/* convert a cube from positional notation format to PLA format */
void pn2str(CUBE *pcube, int nvar, char *str)
{ short i, val;

  for (i=0; i<nvar; i++)
   { getvar(pcube, i, &val);

     switch (val)
      { case 0: str[i] = 'e'; break;
        case 1: str[i] = '1'; break;
        case 2: str[i] = '0'; break;
        case 3: str[i] = '-';
      }
   }
  str[nvar] = '\0';
}

void read_cubes(FILE *fp, ARRAYOFCUBES *cubes)
{
  int i, ch, lineno, numvar, res;
  char word[256];
  CUBE tempcube;

  while (1)
   { switch(ch = getc(fp))
```

```
      { case EOF: return;

        case '\n':
        case ' ':
        case '\t': break;

        case '#': (void) ungetc(ch, fp);
                  skip_line(fp);
                  break;

        case '.':
          get_word(fp, word);

          /* .i gives the cube input size (binary-functions only) */
          if ( strcmp(word, "i") == 0 )
            { if (fscanf(fp, "%d", &numvar) != 1)
                printf("error reading .i");

              cubes->numvar = numvar;
              /* printf("there are %d variables\n", numvar); */
            }

           /* .e and .end specify the end of the file */
           if (strcmp(word, "e") == 0) return;

           break;

        default:
           (void) ungetc(ch, fp);
           get_word(fp, word);
           /* printf("cube is >%s< ", word); */
           res = str2pn(word, numvar, &tempcube);
           /* printf("res=%d, cube is >%08X< \n", res, tempcube); */
           storecube(cubes, &tempcube);
       }
   }
}

/* display an array of cubes */
void showarray(ARRAYOFCUBES *parray, char *name)
{ int  i;
  char word[17];

  printf("Array Of Cubes [%s]: %d variables, %d cubes\n",
          name, parray->numvar, parray->numcube);

  for (i=0; i<parray->numcube; i++)
   { printf(" cube[%d]: %08X (", i, parray->cube[i]);
     pn2str(&parray->cube[i], parray->numvar, word);
     printf("%s)\n", word);
```

```
   }
}

/* disjoint sharp operation on two cubes */
void dsharp(CUBE cubea, CUBE cubeb, int numvar, ARRAYOFCUBES *parray)
{ short vala, valb, valc;
  int i, j, res=0;
  CUBE tempcube = 0L;

  int subset[4][4] = { 1, 1, 1, 1,
                       0, 1, 0, 1,
                       0, 0, 1, 1,
                       0, 0, 0, 1 };

  int active[4][4] = { 0, 0, 0, 0,
                       1, 0, 1, 0,
                       2, 2, 0, 0,
                       3, 2, 1, 0 };

  /* check the first pre-relation */
  for (i=0; i<numvar; i++)
   { getvar(&cubea, i, &vala);
     getvar(&cubeb, i, &valb);

     if (!(vala & valb)) /* there is no intersection */
      { res = 1;
        break;
      }
   }
  if (res == 1)
   { storecube(parray, &cubea);
     return;
   }

  /* check the second pre-relation */
  for (i=0; i<numvar; i++)
   { getvar(&cubea, i, &vala);
     getvar(&cubeb, i, &valb);

     if (!subset[vala][valb]) /* A is not the subset of B */
      { res = 1;
        break;
      }
   }
  if (res != 1) return;  /* no result cube */

  /* calculate disjoint sharp */
  for (i=0; i<numvar; i++)
   { getvar(&cubea, i, &vala);
     getvar(&cubeb, i, &valb);
```

```
      if (!subset[vala][valb]) /* i is the special position */
       { tempcube = 0L;

         /* printf("the position %d is a special positon ...\n", i); */

         for (j=0; j<i; j++)  /* after function */
          { getvar(&cubea, j, &vala);
            getvar(&cubeb, j, &valb);
            valc = vala & valb;
            setvar(&tempcube, j, &valc);
          }

         getvar(&cubea, i, &vala);
         getvar(&cubeb, i, &valb);
         valc = active[vala][valb];
         setvar(&tempcube, i, &valc);

         for (j=i+1; j<numvar; j++)  /* before function */
          { getvar(&cubea, j, &vala);
            valc = vala;
            setvar(&tempcube, j, &valc);
          }

         storecube(parray, &tempcube);
       }
    }
}

/* copy one array of cubes to another array of cubes */
void copyarrcube(ARRAYOFCUBES *parraya, ARRAYOFCUBES *parrayb)
{ int i;

  parrayb->numvar = parraya->numvar;
  parrayb->numcube = parraya->numcube;

  for (i=0; i<parraya->numcube; i++)
    parrayb->cube[i] = parraya->cube[i];
}

/* disjoint sharp operation on two arrays of cubes */
void dsharparr(ARRAYOFCUBES *parraya,ARRAYOFCUBES *parrayb,ARRAYOFCUBES *pres)
{ ARRAYOFCUBES tmparraya, tmparrayb;
  ARRAYOFCUBES *pcubesa, *pcubesb, *ptmp;
  int i, j;

  if (parraya->numvar != parrayb->numvar)
   { pres->numvar = 0;
     pres->numcube = 0;
     return;
```

```
    }

  copyarrcube(parraya, &tmparraya);
  pcubesa = &tmparraya;
  pcubesb = &tmparrayb;

  pcubesb->numvar = parraya->numvar;

  /* showarray(pcubesa, "Copy arrA to tmp buf A"); */

  for (i=0; i<parrayb->numcube; i++)
    { pcubesb->numcube = 0;
      for (j=0; j<pcubesa->numcube; j++)
        dsharp(pcubesa->cube[j], parrayb->cube[i], pcubesa->numvar, pcubesb);

      if (debug) printf("Iteration %d: %d cubes\n", i, pcubesb->numcube);
/*    showarray(pcubesb, "intermedia result");
*/
      ptmp = pcubesa;
      pcubesa = pcubesb;
      pcubesb = ptmp;
    }

  /* showarray(pcubesa, "final result"); */
  copyarrcube(pcubesa, pres);
  pres->numvar = pcubesa->numvar;
  pres->numcube = pcubesa->numcube;
}

/* calculate the difference time between two moments (two timeval structure) */
void difftv(struct timeval *tv1, struct timeval *tv2, struct timeval *tvdiff)
{
  if (tv2->tv_usec < tv1->tv_usec)
    { tvdiff->tv_usec = tv2->tv_usec - tv1->tv_usec + 1000000;
      tvdiff->tv_sec = tv2->tv_sec - tv1->tv_sec - 1;
    }
  else
    { tvdiff->tv_usec = tv2->tv_usec - tv1->tv_usec;
      tvdiff->tv_sec = tv2->tv_sec - tv1->tv_sec;
    }
}
```