High Level Design Methodology for Reconfigurable Systems

A dissertation presented to
the faculty of
the Russ College of Engineering and Technology
Ohio University

In partial fulfillment
of the requirements for the degree
Doctor of Philosophy

Mingwei Ding

November 2005

This dissertation entitled

HIGH LEVEL DESIGN METHODOLOGY FOR RECONFIGURABLE SYSTEMS

by

MINGWEI DING

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology
Ohio University by

Janusz Starzyk
Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology

DING, MINGWEI. Ph.D. November 2005. School of Electrical Engineering and Computer Science

High Level Design Methodology for Reconfigurable Systems (143pp.)
Director of dissertation: Janusz Starzyk

The rise of reconfigurable computing systems presents great challenges to traditional electronic system design mainly due to the dynamic reconfigurability requirements of these systems. At the same time, the chip capacity is growing at a faster pace than that of design productivity, bringing up the problem of a "productivity crisis". Thus, there is a strong demand in the microelectronic design industry for electronic design automation tools and design paradigms that will improve the design productivity and utilize the full power of reconfigurable architectures.

This dissertation addresses these needs and includes research on high level design methodology for reconfigurable systems and studies applications of such systems. Two ongoing projects — SOLAR and DRAW, which are dedicated to machine learning and next generation mobile station design, respectively, were selected as examples of reconfigurable system applications.

First of all, a novel design methodology and design language, Unified Algorithmic Design Language (UADL), have been proposed and developed. UADL allows the designer to enter a high level algorithmic description once and automatically generate executable code in different target languages, such as Matlab and VHDL. The proposed UADL tool has been applied to the design process of both projects with about 2 to 3 times savings in code sizes of their algorithmic parts.

Regarding the selected projects' needs, the following topics have been studied: optimum interconnection, reconfigurable routing, mapping of the Turbo decoder algorithm and dimensionality reduction. An optimum interconnection scheme has been derived and a novel pipeline structure has been devised to realize the reconfigurable routing with linear hardware costs. Dimensionality reduction is an indispensable step

for an efficient learning, feature extraction, classification, and image/pattern processing. Two postmapping algorithms have been developed and tested against real world data.

An important contribution of this dissertation is the proposed new design methodology based on UADL. The UADL methodology provides a high level, unified, tool-independent design interface for system designers, as Java has provided a platform independent programming language for software developers. It is my hope and desire that this design paradigm will gain such popularity as Java did.

Approved: Janusz Starzyk
Professor of Electrical Engineering and Computer Science

To My Dear Parents

Acknowledgments

After five years in Athens, my doctoral study is about to end. It would be impossible for me to finish this task without the help from many.

First of all, I would like to thank my advisor Dr. Starzyk whose guidance and supports are the key elements of this dissertation. Also I am grateful for his career advice as a friend.

Secondly, I want to extend my sincere thanks to my committee members, Dr. Curtis, Dr. Dill, Dr. Matolak and Dr. Mohlenkamp for reviewing my dissertation and giving me many helpful comments.

Special thanks go to Dr. Mohlenkamp who first introduced LaTeX to me and financed me to overcome last difficult time of my study life. The same special gratitude goes to Mrs. Zofia Starzyk as well, who had offered many financial support to me and cooked many delicious food for our research group parties. I need to thank Steven Diehl for helping me out with the LaTeX template.

Also I would like to express my thanks to my group members — Haibo, Yinyin, Yue, Zhen and James. At the same time, I owe a big "thank you" to all my friends who made my stay in Athens an enjoyable one.

Last but not least, I would like to thank my parents who have always been the primary source of support to me.

> *As its best was bought only at the cost of great pain. Driven to the thorn, with no knowledge of the dying to come. But when we press the thorn to our breast,*
> *We know........*
> *We understand.....*
> *And still......we do it.* — *Colleen McCullough*

Table of Contents

Abstract 3

Dedication 5

Acknowledgments 6

Table of Contents 7

List of Figures 9

List of Tables 12

1 Introduction 13
- 1.1 Motivation 13
- 1.2 Previous Works 17
- 1.3 Current Projects 19
  - 1.3.1 SOLAR 19
  - 1.3.2 DRAW 19
- 1.4 Objectives 20
- 1.5 Outline 22

2 Reconfigurable System Design Methodology 23
- 2.1 Managing System Complexity 23
- 2.2 UADL Motivation 24
- 2.3 UADL Structure 28
- 2.4 Comparison with Other Languages 35
- 2.5 FSM Examples 37

3 SOLAR System Design 41
- 3.1 Introduction 41
- 3.2 Input Selection and Weighting Scheme 43
  - 3.2.1 Random vs. Greedy 44
  - 3.2.2 Optimal and Binary Weighting Schemes 46

3.2.3 Simulation Results 50
3.3 Reconfigurable Routing Channel 54
3.3.1 Pipeline Structure 55
3.3.2 Data Flow Description 57
3.3.3 Node Operations 61
3.3.4 Simulation Results 65
3.4 Application Example 67
3.5 Contributions by UADL 73

4 DRAW System Design 78
4.1 Introduction 78
4.2 DRAW Structure 79
4.3 UADL Contribution 81
4.4 Design Example – Turbo Decoder 87
4.4.1 Introduction 87
4.4.2 Turbo Decoding Algorithm 87
4.5 DRAW Implementation Cost 91
4.6 UADL Contributions 93

5 Dimensionality Reduction 97
5.1 Preprocessing 97
5.2 Dimensionality Reduction and Postmapping 98
5.3 Principles of Postmapping 100
5.4 Linear Postmapping 101
5.5 SVD Postmapping 105
5.6 Example Applications 108
5.7 Conclusion 118

6 Conclusions and Future Work 119
6.1 Conclusions 119
6.2 Original Contribution 122
6.3 Future Work 122

Bibliography 124

A UADL Construction and Visitor Pattern 136

B List of Acronyms 140

C Source Code Used in Dissertation 143

List of Figures

1.1 Flexibility vs. performance. 14
1.2 Widening design productivity gap. 16

2.1 A typical VLSI design flow. 26
2.2 Similarity between Java design flow and UADL design flow. 27
2.3 Traditional design flow. 29
2.4 UADL design flow. 29
2.5 EBNF specification of UADL. 30
2.6 UADL MUX code for target VHDL. 31
2.7 UADL MUX code for target Matlab. 32
2.8 Module interchanging and detailing procedures. 33
2.9 Language commonalities and differences. 35
2.10 Incompatibleness among current tools. 36
2.11 UADL framework unifies the user design platform. 37
2.12 FSM diagram of the comment filter. 38
2.13 FSM code UADL implementation. 39
2.14 Generated VHDL FSM code (partial). 40

3.1 SOLAR structure. 42
3.2 Classification rate with different selection strategies 45
3.3 Model of weighted sum of noisy signals 46
3.4 P_{gain} vs. dP in a binary weighted connection 50
3.5 P_{gain} vs. dP at $P_{gain} = 0$ in a binary weighted connection 51
3.6 Visualization of Hamming distances for noisy vectors 52
3.7 A 31-neuron network topology 53
3.8 Pipeline structure overview. 57
3.9 Same network with two connection configurations. 57
3.10 Column 1, cycle 0 to cycle $L + P_k$. 59
3.11 Column 1, cycle $2L$ to cycle $2L + P_k$. 59
3.12 Column 1 and 2, cycle $3L$ to cycle $3L + P_k$. 60
3.13 Timing diagram between column 1 and 2. 60
3.14 KCPSM assembly code fragment for reading. 61
3.15 Single node read/write structure. 62

3.16 Node implementing $Lm(x)$ and $Em(x)$ operations. 64
3.17 Comparison between modified sigmoid and sigmoid. 65
3.18 Node implementing $Em(x)$ operation. 66
3.19 Single node read/write waveform. 66
3.20 Design area vs. network size. 67
3.21 4×7 Array processing 4-feature Iris data. 68
3.22 4×7 array RTL schematic. 68
3.23 Node RTL schematic. 69
3.24 One column RTL schematic. 70
3.25 One column RTL schematic (2). 71
3.26 4×7 array layout by Xilinx tools. 72
3.27 UADL code for one column design. 73
3.28 Generated VHDL code for one column design (partial). 74
3.29 Sample UADL testbench code. 76
3.30 Generated VHDL code from UADL testbench code (clock). 77
3.31 Generated VHDL code from UADL testbench code (reset). 77
3.32 Generated VHDL code from UADL testbench code (din,tin). 77

4.1 A block diagram of DRAP. 80
4.2 Crossbar implementation for 4-bit Barrel Shifter. 81
4.3 3 Stage logarithmic implementation for 7-bit Barrel Shifter. 82
4.4 UADL design flow with VHDL Matlab code auto-generation. 83
4.5 VHDL simulation results for one stage bypass/shift unit. 83
4.6 Matlab results for generated barrelshift.m code. 84
4.7 VHDL simulation results for 3-stage bypass/shift unit. 84
4.8 Matlab results for generated barrelshift3.m code. 85
4.9 Layout map for 3-stage Barrel Shifter. 86
4.10 PCCC encoder proposed by 3GPP. 88
4.11 Turbo Decoder structure. 88
4.12 Datapath for α calculation. 89
4.13 Datapath for LLR (4 states). 90
4.14 Sliding window scheme. 91
4.15 SISO decoding delay. 91
4.16 UADL code for LLR. 94
4.17 Generated VHDL code for LLR (partial). 94
4.18 Generated Matlab code for LLR. 95
4.19 Simulation results for LLR computing. 96

5.1 Preprocessing unit and SOLAR system. 98
5.2 Linear method reconstruction example 102
5.3 Linear postmapping error relation with k 103
5.4 Swiss Roll data in the original space 104
5.5 Unfolded Swiss Roll data with linear postmapped points 104

5.6 Comparison between linear and SVD postmapping 108
5.7 Data after SeDuMi process and postmapping 109
5.8 Histogram of distances between P_{post} and P_{LLE} 110
5.9 Illustration of the point shift by postmapping 111
5.10 Face data dimensionality reduction by ISOMAP 112
5.11 Face data postmapped SVD postmapping based on ISOMAP 113
5.12 MNIST data mapped by LLE 114
5.13 MNIST data postmapped by SVD postmapping based on LLE 115
5.14 Postmapping error vs. training set variability 116
5.15 Vehicle data set images with four positions 116

A.1 Visitor and node classes UML diagram. 137

List of Tables

3.1 Calculation of the output recognition rate 49
3.2 Comparison of binary and optimal weighting 53
3.3 $Li(x)$ value table. 63
3.4 Code size comparison for pipeline and testbench design. 76

4.1 Wireless standards for 2G and 3G network. 78
4.2 Output data for 1-stage Barrel Shifter. 85
4.3 Output data for 3-stage Barrel Shifter. 86
4.4 Generated code size compared with original UADL code size. 86
4.5 Code size comparison for α (or β) design. 95

5.1 Classification rate for SVM and Eigen-face methods 117
5.2 Classification rate for LLE+postmapping+SVM method 117

Chapter 1

Introduction

1.1 Motivation

The very first idea of reconfigurable computing (RC) can be dated back to 1960, when Gerald Estrin proposed an F+V computer structure in his conference paper [Est60]. In his paper, Estrin envisioned a novel computer structure that consists of fixed parts (F), variable structures (V) and supervisory control parts, where the variable structure can change the functionality according to the application needs. However, the technology at that time was not advanced enough to support his pioneering idea until the emergence of Field Programmable Gate Array (FPGA) technology in the 1980's.

After about two decades of development, the FPGA has evolved from initial *glue logic* components to today's major computing platform besides the general purpose processor (GPP) and the digital signal processor (DSP). GPP and DSP are fairly flexible and are able to cope with most applications, but at the cost of slow execution speed and a big chip area. Therefore, the dedicated Application Specific Integrated Chip (ASIC) still possesses a big market share for computation intensive tasks such as data encryption, image processing, wireless signal processing, etc.

However, the functional rigidness of the ASIC chip and the associated high fabrication cost have prevented the ASIC chip from being successful in a low to medium volume, fast-paced market. Whenever application requirements change, the devel-

oped ASIC chips most likely become useless. Particularly, in the field of wireless communication, due to the huge market and profit, we have several incompatible standards competing with each other. Therefore, if a user needs to roam globally, his mobile terminal should be able to support several different standards working on different frequencies. In this kind of scenario, traditionally, we need several sets of ASIC chips to support all the standards involved, which leads to a bulky, power hungry device and waste of the silicon area. In pattern recognition or image processing, different algorithms have been developed for different objectives due to the complicated nature of the task and, in practice, it is highly desirable to adaptively change the algorithm based on the outside environment and system status. This presents a difficult problem to the traditional ASIC design approach.

To address these problems, researchers resorted to the RC platform for a better solution. Significant research resources have been invested in this area, as summarized in [CH02; TS03]. The advantage of the RC platform over the ASIC and the GPP (DSP) approaches is its better trade-off between performance and flexibility. A carefully designed RC system can deliver a close-to-ASIC performance and maintain software-like flexibility for a given application field. The trade-off between flexibility and performance of these three technologies is shown in Figure 1.1.

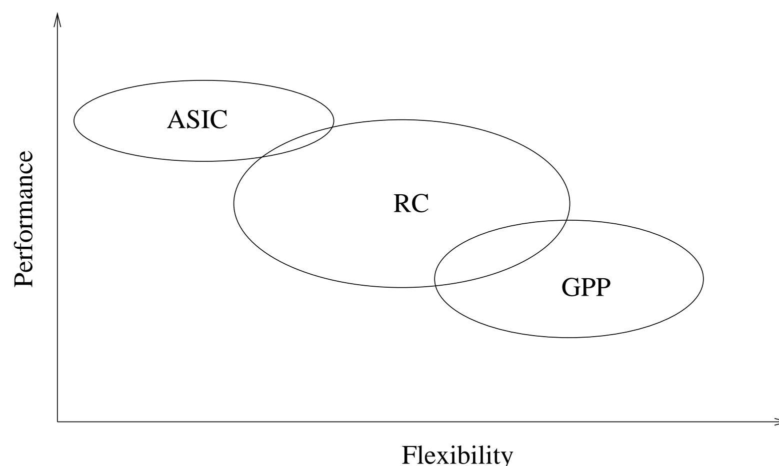


Figure 1.1: Flexibility vs. performance.

As pointed out in [LS05], it is the following two properties that brought prosperity to the RC industry.

- Dynamic reconfigurability, which allows the device to adapt to specific applications.
- Spatial computation in contrast to temporal processing with the GPP and the DSP.

The dynamic reconfigurability helps to reduce the silicon area and power consumption. The spatial computation has been proven to be a great performance booster. An encryption example given in [LS05] shows that an FPGA-based RC system is about two orders of magnitude faster than a high performance work station. In [BP02], the RC system is shown to have over 300 times speedup over a Cray II computer for DNA sequence matching problems.

Led by Moore's law, the semiconductor industry keeps pushing feature size down and the RC system became heterogeneous as researchers began to combine the best of the GPP (DSP) and RC worlds. For example, Xilinx has embedded Power PC CPU in its Virtex II-Pro product line [Xil], and Altera has put up to 96 DSP blocks in its Stratix II device family [Alt]. These new products from the two leading manufacturers indicate the GPP/DSP-RC marriage to be the direction of future development.

The new challenges for the RC system design come from the following two factors. The first one is the rapidly increasing system complexity. As the chip density continues to grow at a high rate, the conventional design approach has become inadequate. According to SemaTech's report, the designer's productivity grows at an average rate of 21% per year, while the chip capacity has a growth rate of 58% per year [Sem97]. As one can observe from Figure 1.2, the difference between chip capacity and design ability is widening every year, thus creating the term "productivity crisis". On one side, individual company will try hard to improve its design productivity to gain advantage oevr its competitors, on the other side, if the indsutry's design productivity is lagging behind Moore's Law too much, it will become harmful to the developement of semiconductor industry.

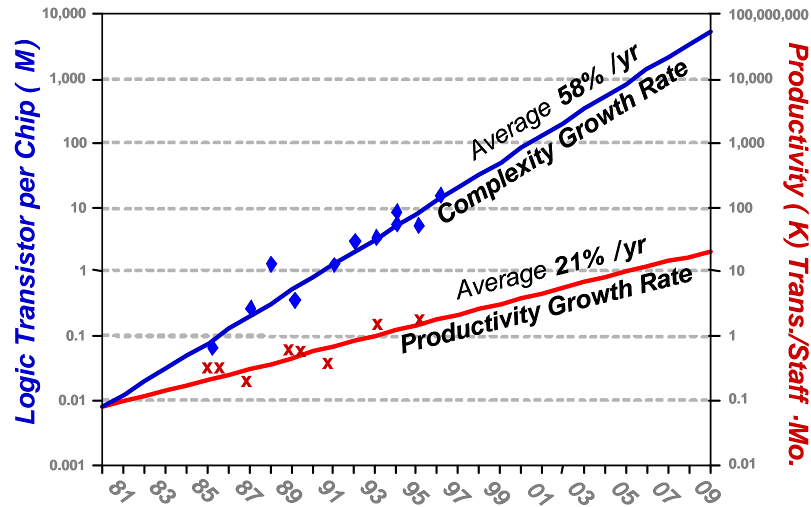


Figure 1.2: Widening design productivity gap (source: SemaTech).

New design methodologies and new design tools are necessary to curb this widening gap between system complexity and design productivity [KMN+00]. Since Gajski et al. had expressed the necessity to automate the whole design process from concept to silicon [GR94], there have been tremendous research efforts in the area of High Level Synthesis (HLS). The HLS research during the past decade are mostly directed to a better interpretation of the input behavioral descriptions, while the design migration between these behavioral descriptions are somehow neglected. Most of the existing HLS tools only accept input descriptions in one dedicated language, which implies that all the members within a design team need to use the chosen language. This has been proven to be a costly and difficult procedure to have hardware developers and software developers share the same programming paradigm and programming language.

In the following sections, I will summarize previous works in the RC system design and research on design productivity improvement like HLS and Object Oriented hardware design.

1.2 Previous Works

In spite of the good flexibility of the reconfigurable architecture, there is no such reconfigurable architecture that can fit all the possible applications. Therefore, many reconfigurable architectures dedicated to different applications have been devised and developed. Among these architectures, there are three major topologies adopted.

The first type of topology is a mesh style structure. In this type of structure, all the function units are arranged in a regular 2D array with local and global routing resources surrounding each unit. The majority of developed architectures belong to this type. Some of the mesh based architectures are listed as follows:

- CHESS array [MSK+99], aimed at multimedia applications, 4-bit granularity.
- RAW [WTS+97], aimed at general purpose computing tasks, 8-bit granularity.
- Garp [HW97], aimed at loop acceleration, 2-bit granularity.
- Morphosys [LSL+00], aimed at image processing and data encryption, 16-bit granularity.
- DReAM [BPG00], aimed at 3G wireless communication, 8/16-bit granularity.
- Adres [MLM+05], aimed at design space exploration, 32-bit granularity.

The second type is pipeline style structure. This structure is based on linear arrays of processing units to efficiently perform pipelining tasks.

- PipeRench [GSB+00], aimed at pipelining tasks, 128-bit granularity.
- RaPiD [ECF96], aimed at general purpose computing tasks, 8-bit granularity.
- CREC [CPVS03], aimed at general purpose computing tasks, 16-bit granularity.

The third type is crossbar style structure. This structure utilizes a crossbar to fulfill the interconnection between processing units.

- PADDI-1 [CR92] and PADDI-2 [YR95], aimed at digital signal processing, 16-bit granularity.

- Pleiades [Rab97], aimed at multimedia applications, multi-granularity.

At the same time, research is going on to improve design productivity. Two approaches have been researched and developed.

The first one is to raise the abstraction level of the design procedure. This is where HLS tools come into play. With the help of HLS tools, designers are able to express their ideas in a behavioral model without worrying too much about the hardware implementation details. Thus designers can produce much larger circuitry within the same time period without compromising the quality. To produce high quality netlist files from the behavioral model, several challenges need to be addressed. The first challenge is the translation from behavioral descriptions to internal presentation used by HLS tools. Currently the most widely used format is the Control-Data Flow Graph (CDFG). Although some other formats [Ber99] have been proposed to tackle this problem, the CDFG is still in dominance. After the internal presentation is established, three steps: *scheduling*, *allocation* and *binding* will be performed by HLS tools to generate a hardware oriented netlist. Many algorithms have been proposed to solve the scheduling and allocation problem for data-dominant behaviors [CP95; RJ97] and control-intensive behaviors [RDJW97; LKJ99].

The second approach is to promote design reuse. One way of promoting design reuse is to develop a rich library of reconfigurable Intellectual Property (IP) blocks. The popularity of the IP library has indicated the effectiveness of this method [RN04]. Another weapon to promote design reuse is borrowed from the software community — object oriented (OO) design methodology. Encouraged by the huge success of the OO method in the software design domain, there have been emerging research interests in object oriented hardware modeling in recent years [KOW+01; MPMF01; GO03]. Many OO concepts have been brought to the field of HLS such as design pattern [DMS03; RMBL05], interface [BLO98] and polymorphism [Pom04], etc. As the technology enters into the deep sub-micron (DSM) era, the interconnect delay begin to outweigh gate delay in system level design. It is important to take this effect into account during the synthesis procedure [MB04; SK04; MM04].

1.3 Current Projects

Two current projects at Ohio University, DRAW and SOLAR, based on [SZL05] and [ASBG00] respectively, are good examples of reconfigurable system designs. Both systems share similar mesh based structure and they both can dynamically reconfigure the content of individual units. The difference between these two architectures is that SOLAR has an array of identical processing units that perform nonlinear processing of input information and which can self-organize in terms of functionalities and connectivities, while DRAW has configurations that were developed ahead of time according to application algorithms. Some overview of these two projects is given below.

1.3.1 SOLAR

A data driven Self Organizing Learning Array (SOLAR) and its hardware structures are being developed in this project. The SOLAR is composed of an array of many simple processing nodes (neurons) with identical structures. Instead of making full connections between layers, each neuron is locally connected to its neighbors. What makes SOLAR a powerful analysis tool is the dynamic properties in two areas: neuron functionalities and inter-neuron connectivities. With a certain level of computing ability embedded in itself, a neuron can adjust its connections and its functionalities at running time. This gives the neuron a learning-like ability to adjust the network topology for a better processing result. The SOLAR array has been successfully applied to several problems, such as credit card problem, data classification, and pattern recognition problems.

1.3.2 DRAW

The evolving wireless communication industry has put wireless engineers in a situation they have never encountered before: the future mobile terminal needs to support higher and higher data rate air-links, video and audio enriched multimedia application, long battery life, light weight, low cost, etc. All these are critical to commercial success. Another factor that challenges system designers is a multitude of competing

standards, each of them claiming to be technically advanced. As a new computing architecture, reconfigurable computing has attracted significant research interests and is deemed a promising platform to solve this problem. Based on [Als02; ASBG00], the Dynamic Reconfigurable Architecture for Wireless (DRAW) project is aiming at third generation wireless communication. In this structure, several application tailored units are developed and a Turbo decoder application that is widely used in wireless communication will be studied.

1.4 Objectives

Given the needs for design productivity and design portability improvement in general, and specifically for the two ongoing projects SOLAR and DRAW, there are two primary objectives of this dissertation.

First of all, a new design methodology and corresponding design tool are to be developed. Instead of focusing on the translation from behavioral model to netlist, this new design methodology will concentrate on design portability between hardware and software designers, providing a unified design interface for algorithm level entry. Therefore, this new tool is called Unified Algorithmic Design Language (UADL). UADL aims to improve collaboration between different design groups and reduce redundant design efforts.

The second primary objective is directly related to the SOLAR and DRAW projects. Since the reconfigurable functionality and connectivity research needs to be carried on within a clearly defined application area, the second objective addresses the system level problems of these two projects.

Specifically, for the SOLAR project, the following sub-objectives apply.

1. The inter-connection problem. In the concept of SOLAR, each node (neuron) actively searches the most useful information to accomplish its association or learning task. In practice, the range of individual neuron's searching ability and the inter-neuron connectivity are limited by hardware resources; therefore, it is necessary to study the optimal way of organizing the network structure within the limits of the underlying hardware platform.

The self-adjusting connecting scheme of SOLAR has posed a great challenge for traditional routing and connectivity schemes since the connection changes are made by individual nodes at run-time. In our previous efforts, multiplexers were used to achieve the dynamic connection reconfiguration. However, the hardware cost brought by multiplexers increases quadratically with the number of interconnections, which is not desirable for a large network size. To address this problem, a novel routing scheme with less hardware cost is needed to fulfill the dynamic connection requirement by SOLAR. The UADL tool is to be applied to the SOLAR project to facilitate the design process.

2. Dimensionality reduction. The SOLAR system is designed for a variety of machine learning problems such as pattern recognition, clustering, and classification. In real world

   very high dimensional space. To effectively process such data, dimensionality reduction is an indispensable step. Concrete research tasks include dimensionality reduction algorithms improvement and the study of its effect on final classification (recognition) rates for various applications.

For the DRAW project, the two following sub-objectives apply.

1. The DRAW structure is first to be briefly reviewed with a key component Dynamic Reconfigurable Arithmetic Processor (DRAP) unit under focus. An important part of the DRAP unit, the Barrel Shifter, is to be re-designed with UADL language to illustrate the effectiveness of the proposed UADL tool. Two target languages – VHDL and Matlab will be used in the design process. It will be shown that the UADL tool can generate functionally correct codes in both target languages; consequently, its shows that the UADL can minimize redundant work for design migration between two different design flows.

2. The Turbo decoder is selected as one of the many possible applications of DRAW architecture due to its importance to wireless communication and its computation complexity. A hardware oriented Max-Log-MAP algorithm is to be presented and analyzed. A sliding window technique is to be adopted to reduce

the storage requirement. Major computing parts such as the LLR computing part and the α (or β) computing part are to be designed with the UADL and the improvement in design productivity will be displayed.

1.5 Outline

This dissertation is organized as follows.

Chapter 2 will discuss the problem in high level design and low level design and will propose a new tool aiming at reducing the redundant work for different design groups.

In Chapter 3, connectivity problems in the SOLAR system design are to be presented and discussed. First, the neuron's input selection and optimal input weighting scheme is analyzed. Second, a novel pipeline scheme is proposed and developed for implementing a reconfigurable routing channel with linear consumption of hardware resources.

Chapter 4 reviews the basic architecture of DRAW and a Turbo decoder design example is developed based on the DRAW architecture to illustrate its capability. The UADL tool is applied to the DRAW design process to demonstrate its effect in improving design productivity.

Chapter 5 discusses the dimensionality reduction problem as a preprocessing part of the SOLAR system. Current major dimensionality reduction algorithms have been inspected and two postmapping algorithms are proposed as a complementary step of the regular dimensionality reduction procedures.

Chapter 6 concludes this dissertation and gives recommendations for future research.

Chapter 2

Reconfigurable System Design Methodology

2.1 Managing System Complexity

After 40 years of dramatic development of integrated circuit technology, Gordon Moore's prediction [Moo65] is still amazingly accurate. The density of transistors on a chip increases exponentially against time which means the number of transistors in unit area doubles every eighteen months. Every few years, new technological innovations have appeared to strongly support semiconductor advancements tracking Moore's Law along the semiconductor technology roadmap. Today, technology has enabled people to squeeze millions of transistors into a single chip, compared with 2,500 transistors in Intel's 8008 in 1972. Confronted with the explosive growth of circuit complexity, traditional schematic entry was no longer productive enough to keep up.

Various VLSI CAD tools thus came to the stage. In the realm of digital design, Hardware Description Language (HDL) had changed the way of digital designing. By shifting design entry from graphics to text, HDL allowed more work to be done by computer programs rather than human designers. In addition, with the help of logic synthesis tools, designers had the opportunity to express their ideas at the Register Transfer Level (RTL), which is more abstract and more efficient compared with the

gate level description. However, after a few years of development, people found that working on the RTL level was not good enough for the ever increasing design size, and similar to earlier times, they found that it was an insufficient way of designing complex circuits. Therefore, many research efforts have been invested in making new tools that allow higher abstraction levels. High Level Synthesis (HLS) or Behavioral Synthesis(BS) is one of the results of these endeavors [TS03].

As defined in [Lin97], HLS is a translation process from a behavioral description into a netlist file which specifies the connectivities between electronic parts. After more than twenty years of development, HLS technology is still far from perfect. One of the reasons for this is that it is hard to evaluate the effect of high level algorithm decisions on the physical level implementation, thus the cost function used by the behavioral compiler can only give rough estimates of this effect. Another important reason, as pointed out in [Lin97], is that a good HLS tool must use domain knowledge (knowledge of the target application field) intelligently to make a satisfying translation, but the more domain knowledge it incorporates, the narrower its potential user base, which is harmful to further development of HLS.

On the other hand, the content on the silicon chip has evolved from pure digital or pure analog circuitry through digital-analog mixed signal structures to System-On-Chip (SoC). With the introduction of the SoC concept, it is natural to require a tighter collaboration between hardware and software designers. However, this collaboration is shown to be a difficult one due to the different working paradigms and different terminology sets each side uses [NBD+03].

There are two major problems that need to be solved before the RC concept can be adopted by more system designers: (1) a standard of programming paradigm, and (2) better design portability between different applications and design groups [VPI05].

2.2 UADL Motivation

Currently there exist many versions of HLS tools in both the commercial market and in the academic community. Most of the tools are based on existing popular

design languages such as C, JAVA, VHDL, Verilog, etc. and certain extensions and restrictions have been applied to a selected language for the purpose of better describing abilities of concurrent hardware i.e., JHDL [WHW01], SystemC [Ope], etc. The selection of the base language reflects the group's focus either on a system level or a physical level. Traditionally, the system level design is governed by software developers, and the physical level design is taken care of by hardware engineers. For many years, software engineers and hardware engineers have been working in two different domains without relating easily to each other's work. With the occurrence of HLS tools and HW/SW co-existence on SoC, this situation needs to be changed.

With the advance of VLSI technology and the development of SoC, the micro controller and embedded CPU have been integrated with memories and random logic circuitry, e.g. FPGA, into a single chip. This offers designers a great opportunity for a better combination of the power of hardware and software. For the reconfigurable computing system design, this means a chance for a better tradeoff between performance and flexibility. At the same time, making a successful system demands expert knowledge from all the fields, such as the application-related area, the software algorithm level, the Register Transfer level, and the physical level, as depicted in Figure 2.1, where the dotted arrows represent the design refining procedure based on the result of verification in each step.

Usually this expertise is possessed by different individuals on the team especially for a complex project. Therefore, frequent communication and cooperation between team members is required during different phases of the project.

To illustrate such design flow, let us consider the following example. Suppose we need to design a chip that can filter out the comment part in the C/C++ source code. The C/C++ comment has two different forms, a single line comment and a multi-line comment. Suppose the C/C++ source code input is in the form of a sequential stream of data. Finite State Machine (FSM) is an obvious solution to this problem. To build such a comment filter system, designers at different levels will have different issues to consider. How many states are necessary in this FSM? Is it a Mealy machine or a Moore machine? These are some questions a system designer needs to answer. After the designer has decided all these system level parameters, he needs to express

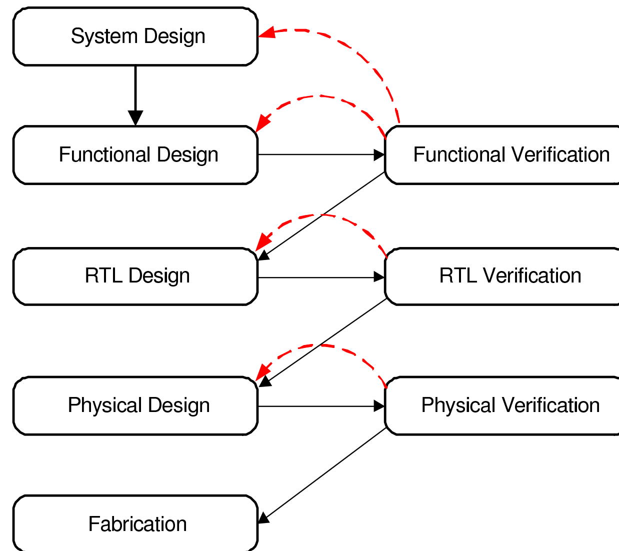


Figure 2.1: A typical VLSI design flow.

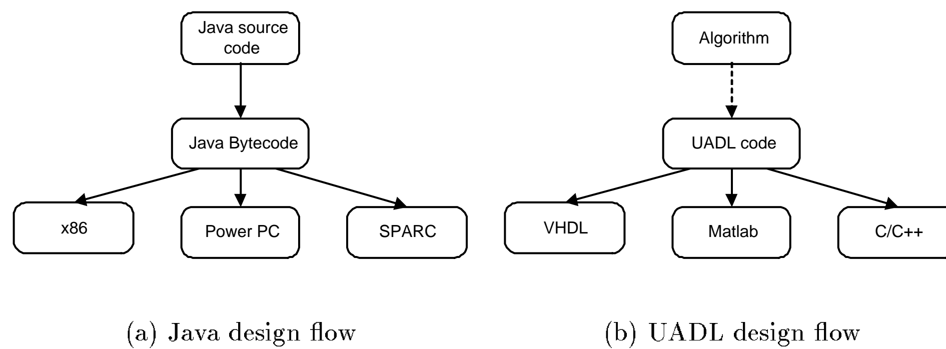
his idea in an executable language, say C language. He cannot ensure the correctness of his design until the C program is implemented and verified. During the period of software development, the designer will focus on how to fulfill the functionality of FSM in target language. Which statements should be used? Have all the situations been considered? For the RTL developer, the focus of design efforts will shift to a lower level. What is the word length, 8-bit or 16-bit? Which comparator should be used? Where is the critical path? Are the setup/hold timing requirements met for all the flipflops? For the physical design, the designer will worry about something else. Which technology to choose? Which library to use? What is the power consumption? Is the interconnection delay comparable to the clock period?

From the above description, it can be seen that designers at different levels focus on different aspects of the same object. From level to level, there are some redundant considerations and perhaps sub-optimum operations made when developers try to express essentially the same idea in different languages from different perspectives. This is particularly true for the software and logic levels. When a software developer finishes the FSM design and verifies its correctness, a lot of time and effort can be saved if certain EDA tools can translate the software implementation into a lower

level design, say the VHDL RTL model. However, the current HLS tools are still in the early stage and have a long way to go to alleviate the problems of integrating different design levels.

In my observation, the HLS tools from different groups all have their own choices of the design languages, even if they select the same base language e.g. C/C++; respective groups will implement different subsets of the original language specification and have different extensions, thus making it extremely difficult to migrate designs between different groups. Even within the same group, it is still a painful and frustrating process if designers need to change the tool in the middle of project even if the new tool tends to have a better potential. Take an analog example in the software domain, where a piece of C code compiled on Intel x86 architecture cannot run on SUN SPARC architecture without recompilation. Quite often, a considerable amount of modification of the original code is needed to make a C code run on another platform.

To solve this problem, SUN has invented the Java platform where the user code is first translated to an intermediate format – Bytecode. With the support from the computing architectures, this Bytecode is able to run on all the supporting platforms such as x86, SPARC, Power PC, etc.



(a) Java design flow

(b) UADL design flow

Figure 2.2: Similarity between Java design flow and UADL design flow.

To address the similar problem in the reconfigurable system design, I propose a Unified Algorithmic Design Language (UADL) based design methodology that allows

designers to use the same input language to express ideas regardless of the target platform, technology and tool selection. On the algorithmic level, UADL can let the user enter his/her design once and have different UADL engines create executable codes in respective target languages as illustrated in Figure 2.2. Please note the analogy between Java and UADL in the sense that they both provide a unified design interface to users and leave the interpretation to vendors. In following sections, a more detailed description of UADL is presented.

2.3 UADL Structure

Based on the analysis presented in the previous section, I propose a novel type of framework aiming at the maximum reduction of re-design efforts between software and register transfer level design cycles within one project development procedure. This framework, named Unified Algorithmic Design Language (UADL), is designed to provide a unified platform for designers at different levels with different tools. It alters the traditional design flow illustrated in Figure 2.3, where the dotted arrows represent design procedures that require a significant amount of manual work, and solid arrows represent the procedures that have been mostly automated by current compiler and synthesis tools.

In essence, UADL is a meta-language with relatively simple yet flexible syntax, because the output of UADL is a legal program in another language that can be compiled by corresponding tools. As can be observed from Figure 2.4, by inserting an intermediate layer between the high level system design idea that mostly resides in the human mind and the lower level, concrete, specific executable languages, UADL provides a unified design interface to all the design tasks for the designers in one design group. Therefore, the communication and cooperation between different members will become more effective.

Usually, a UADL program is composed of four parts – target options statement, configuration blocks, command blocks and comments. The formal Extened Backus Naur Form (EBNF) specification (except for comments) for UADL is given in Figure 2.5.

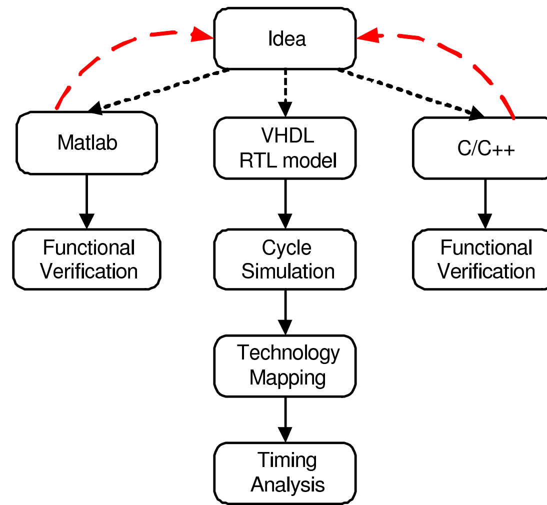


Figure 2.3: Traditional design flow.

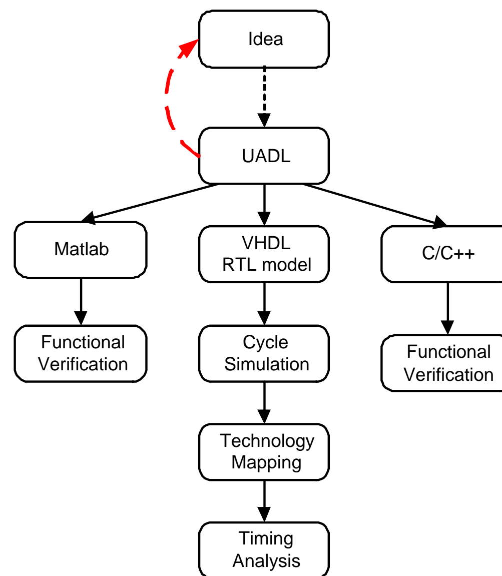


Figure 2.4: UADL design flow.

As defined by the syntax rule illustrated in Figure 2.5, a design unit is composed of one "targetBlock" and multiple "cfgBlock"s and "cmdBlock"s. Further explanation of these components are given below.

```
1    UNIT ::=targetBlock cfgBlock*   cmdBlock*
2    targetBlock ::= "target" "=" NAME "@" ["!"] FILENAME "~" NAME
3    cfgBlock ::="#" NAME [ simpleExpression ] "{"
4                         myStatement+
5                       "}" [ TARGETOPTION]
6    cmdBlock ::="$" <NAME> [ simpleExpression ] "{"
7                    (myStatement | cmdBlock)+
8                       "}" [ TARGETOPTION]
9    TARGETOPTION ::= "@[" NAMELIST "]"
10   NAMELIST ::= NAME [ ("," NAME)+ ]
11   NAME   ::=LETTER (LETTER| DIGIT )*
12   FILENAME ::= NAME [ ( ("." |"/" ) NAME)+ ]
13   myStatement ::= [ simpleExpression] ["#" [INTEGER] ] ";"
14   simpleExpression ::= (OPERATOR | FILENAME | INTEGER| NAME )+
15   INTEGER ::= (DIGIT)+
16   LETTER  ::= "_" | "a"-"z" | "A"-"Z"
17   DIGIT   ::= "0"-"9"
18   OPERATOR::= "*" | "/" | "%" | "+" | "-" | "<" |
19               ">" | "&" | "^" | "|" | ":" | "!" |
20               "." | "(" | ")" | "," | "?" | "[" | "]" | "="
```

Figure 2.5: EBNF specification of UADL.

1. targetBlock – a target statement specifies the target language, i.e. what language UADL will produce, which tool (engine) is to be used to perform the interpretation task for this piece of code, and what is the extension of the output file.

2. cfgBlock – Configuration blocks are used to address the non-algorithmic part of the design, like the unit input/output library unit path, internal variable, etc.

3. cmdBlock – Command blocks constitute the majority of the working code. They give designers the ability to express how to implement the desired algorithm. Each command block can have a different target tag to specify all of its possible targets. By changing the target option settings, the same UADL source file can have different command blocks to be interpreted for different target languages.

Currently supported command blocks include *if, else, for, case, var, port* blocks, etc.

4. Comments – comments are not listed in the EBNF specification. The comments in UADL adopt the same comment style as C/C++: single line comment begins with "//" and multi-line comment is enclosed by "/*" and "*/".

Configuration blocks and command blocks are the workhorse of the design file. In essence, the configuration block is a special kind of command block that is specifically designed for the non-algorithmic part of the design such as external unit inclusion, library definition, unit interfaces to the outside world, internal variables, constant variable definitions, etc. Therefore, the command block is dedicated to processing the algorithm parts, like value assignment, loop statement, conditional statement, etc.

A MUX Example

The following paragraphs use a simple MUX design targeting two languages – VHDL and Matlab to further explain the details of the UADL syntax.

```
1     target =v @VHDLWalker ~vhd
2     #ports{
3       in int d1, d2;
4       in int1 select;
5       out int dout;
6     }@[v,m]
7     $if select==0 {
8       dout = d1;
9     $else {
10      dout = d2;
11     }
12    }@[v, m]
```

Figure 2.6: UADL MUX code for target VHDL.

As displayed in Figure 2.6, line 1 is the target option statement which specifies the target name as "v", the UADL engine is "VHDLWalker" and the file extension of the generated code is "vhd". Lines 2 to 6 are the configuration block part where it

```
1     target =m @MatlabWalker ~m
2     #ports{
3       in int d1, d2;
4       in int1 select;
5       out int dout;
6     }@[v,m]
7     $if select==0 {
8       dout = d1;
9     $else {
10      dout = d2;
11     }
12    }@[v, m]
```

Figure 2.7: UADL MUX code for target Matlab.

specifies three input signals "d1", "d2", "select" and one output signal "dout". The "in" and "out" in lines 3 to 5 specify the direction of signal flow. "int" and "int1" are the type specifiers. By default, a "int" type represents an 8-bit integer, if there is a number right next to "int", this number specifies the bit width of the signal. Thus, the "select" signal has only one bit, while "d1", "d2" and "dout" have 8 bits. Lines 7 to 12 are the command block part which specifies the functionality of this component. It simply specifies an if-else logic control that routes either "d1" or "d2" to output "dout". Please note the target options at lines 6 and 12. The target option at these two lines mean the "port" configuration block and the "if" command block are both bound with target "v" and "m".

The second piece of the MUX code for the Matlab target is given in Figure 2.7. It is obvious to see that these two codes have exactly the same bodies except for the target option statements. In Figure 2.7, the target name is now changed to "m", the UADL engine is changed to "MatlabWalker" and the file extension is changed to "m".

Given the binding between command blocks and target tags, designers obtain the power to freely specify the design reuse parts between different phases, thus saving a lot of possibly redundant work as permitted by interpretation tools. For example, Figure 2.8 displays the module interchange and detailing procedures. The

interchange procedure refers to the change of certain modules for different targets. Suppose that initially we have a cat design consisting of seven modules aiming at target language A (Figure 2.8a). Later on, the same cat design needs to be re-targeted to another language B and one of the modules may need to be replaced by a new one to accomplish this re-targeting procedure. The new module may have included some features that are specific to language B, but it should have the same interface as that of the old one to work properly with other modules, as is shown in Figure 2.8b. Module detailing procedure means that a part of the design needs to be further developed with more details (usually lower level details) for another target language design. For example, the cat design in Figure 2.8a, needs to be further developed when target language C is desired. Thus, one module of the original design has been split into two modules as displayed in Figure 2.8c. These kinds of changes are very common when design hierarchy is shifted from a high level like behavioral level down to a lower level like register transfer level where more detailed design information is needed, especially, the hardware issues such as bit width, datapath delay, etc.

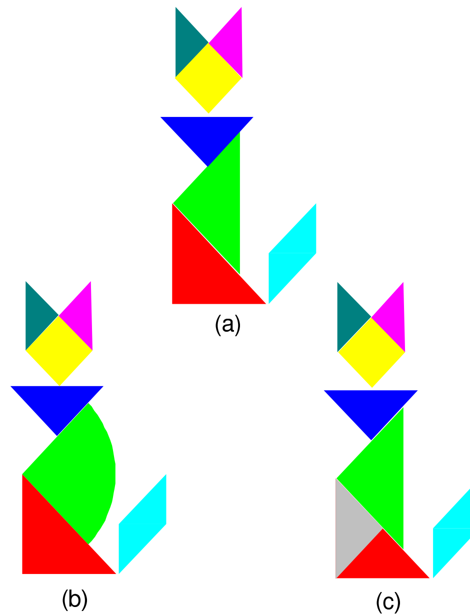


Figure 2.8: Module interchanging and detailing procedures.

As the capability varies from one synthesis tool to another, different subsets of the original design may need to be modified for different target languages and different tools. However, there quite often exist many parts of the design that can be recognized by those tools under consideration. Traditionally, these changes were made manually by human designers and the whole design needed to be translated even with the parts that can be recognized by these tools. This is the cause of low efficiency and repeated work, and UADL is aimed to address these problems.

In summary, UADL language has the following features.

1. Simple syntax – The syntax of UADL is fairly simple compared to many modern languages as one can observe from Figure 2.5. The exact meaning of a certain command block is defined by a command/engine pair.

2. Flexible statement – The statement of UADL is close to C/C++ style and supports some features that allow more concise description such as index expansion. Actually it does not force a strict syntax as there is no explicit syntax rules setup for the "simpleExpression" component (Figure 2.5). The reason for this flexibility is that in spite of many common functional blocks such as if-else, for/while loop, each language may have its own specialty that is lacking in other languages, as shown in Figure 2.9, where each of three target languages has its own statement to include external resources. To be able to adapt to as many target languages as possible, the task of recognizing different language elements is left for the corresponding UADL engine.

3. Focused on algorithmic levels – as mentioned before, the current development of UADL is mainly focused on algorithm translation from a high level (algorithm) to a low level (RTL).

4. Open to all the possible tools – UADL is open to all possible tools including existing and future tools. The interpretation task is left for the targeting tool vendor. User (designer) will be able to enjoy very little modification migrating designs between different tools. The source design file can freely specify which tool it wants for the target code generation.

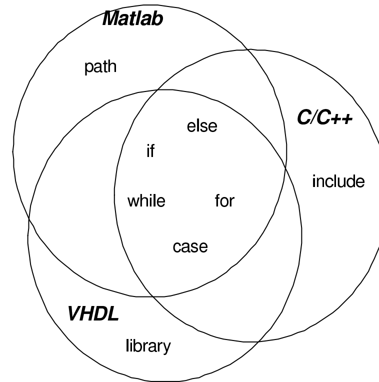


Figure 2.9: Language commonalities and differences.

5. Target binding – The binding between a certain configuration/command block and the target is achieved by the target option following that particular configuration/command block. As defined in line 9 in Figure 2.5, a target option is in the form of "@[*target1*, *target2*, ...]", where *target1* and *target2* and so on give a list of possible targets for this given configuration/command block.

2.4 Comparison with Other Languages

This section will list some comparisons between UADL and several other popular design languages that are aimed at unifying hardware and software design processes.

1. Forge language, formerly lava, now acquired by Xilinx Inc., is a Java-based HDL. By introducing several features and additional libraries, it offers designers the ability to infer hardware structure using a subset of Java. For example, with a "bit and" operation at assigning steps, Forge uses it as a data width indicator to infer the actual number of bits for synthesis.

2. SystemC, an initiative led by a group of leading EDA manufacturers to develop a system level hardware language leveraging the widest user base of C/C++. Similarly, SystemC introduces several new components to provide the concurrent description ability and channel communication to simulate the hardware.

3. SystemVerilog, developed by Accellera, is primarily aimed at the chip implementation and verification flow, with powerful links to the system level design flow.

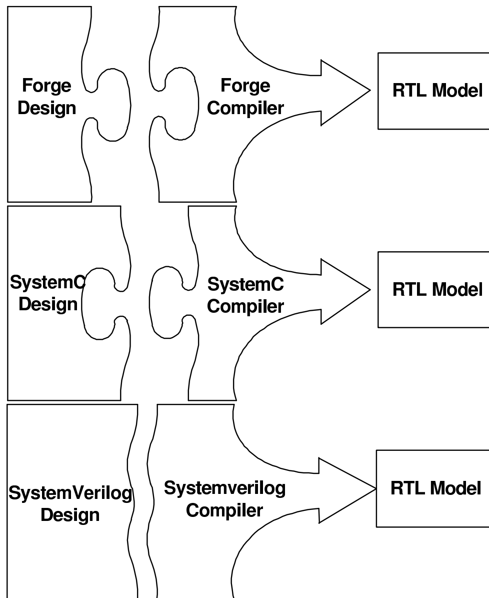


Figure 2.10: Incompatibleness among current tools.

There are dozens of other languages existing in industry and academia with the same goal in mind – unify the design language for both software and hardware engineers. However, the existence of multiple proprietary languages itself has prophesied the failure of their goals, as illustrated in Figure 2.10. To make it worse, those system design languages are either based on different existing languages or are a different subset/superset of the base languages, which makes it almost impossible to migrate from one platform to another without completely rewriting the application. By its design at the very beginning, UADL has moved this migration problem to engine vendors, rather than leaving it to the designers (Figure 2.11). Therefore, the designers can have more freedom in trying out different tool suits and better protection for the previous design legacy.

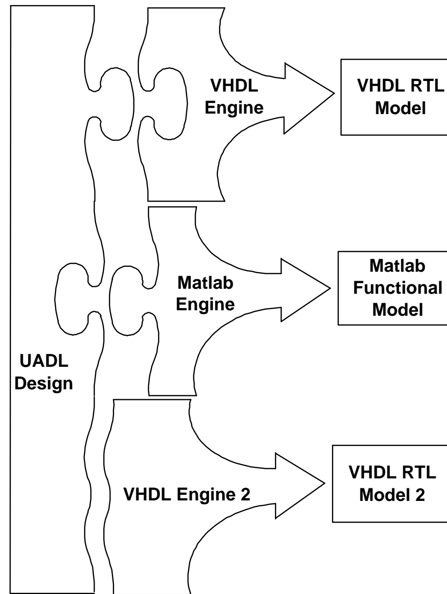


Figure 2.11: UADL framework unifies the user design platform.

2.5 FSM Examples

Based on the specification of UADL, a VHDL engine and a Matlab engine are developed and have been applied to the design process of SOLAR and DRAW projects. In the following paragraphs, a Finite State Machine (FSM) design sample is used to briefly illustrate the capability of UADL design flow. In Chapter 4 and 5, more details of the contribution of UADL design flow during the project design will be revealed.

The Finite State Machine is an important and widely used computing model in both hardware and software areas. Here the comment filter described in the previous section is used as a case study to show the capability of UADL language. Based on previous analysis of the requirement of the comment filter, five states can be established to represent the different steps in searching comment patterns in a C/C++ program.

1) *Init* state – this is the initial state of the FSM, should be set before reading the code stream.

2) *R* state – this is the ready state, machine should enter this state whenever a '/' is found.

3) *S* state – this is the single line comment state, machine should enter this state when two consecutive '/' are found.
4) *M* state – this is the multiple line comment state, machine should enter this state when pattern '/*' is found.
5) *M2* state – this is the multiple line comment state 2, machine should enter this state when a '*' is found and current state is in *M*.
The state transition conditions and corresponding outputs are displayed in Figure 2.12. On each transition arrow, state transfer condition and output are separated by a ":" symbol. For example, the *'/':0* string on the arrow from *Init* state to *R* state means this state transition will happen if the input character is '/' and the out of the FSM is *0* indicating the comment part is yet to come. Based on Figure 2.12,

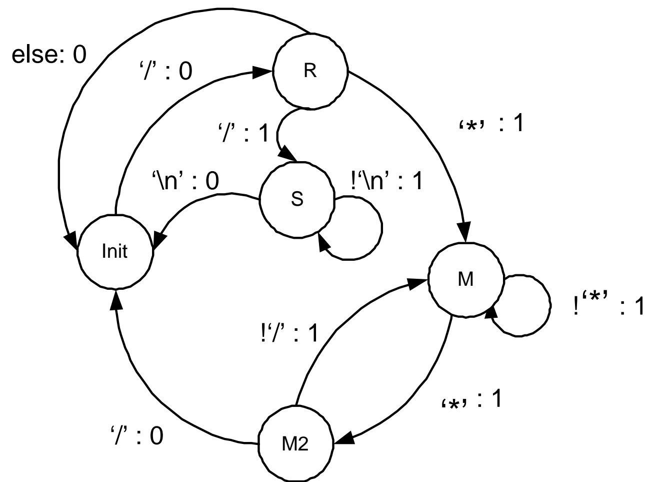


Figure 2.12: FSM diagram of the comment filter.

corresponding UADL code is developed and VHDL code is generated as shown in Figure 2.13 and Figure 2.14 (To limit the space, only part of the code is displayed).

In the following chapters, more design examples will be given to demonstrate how UADL can help to facilitate the design process.

```
1    ...
2    $if reset==0 {
3            state=INIT;
4      $else{
5              $switch state {
6                $case INIT {
7                  $if ch==SLASH{
8                   state=RDY;
9                  }
10               }
11             $case RDY {
12                ...
13             }
14                   ...
15             $case MULTI2 {
16                $if ch==SLASH {
17                    state=INIT;
18                $elseif ch!=STAR {
19                    state=MULTI;
20                  }
21                            }
22             }
23       }
24     }
25   }
26   ...
```

Figure 2.13: FSM code UADL implementation.

```
1     ...
2     begin
3     if reset ='0' then
4     state <="000";
5     else
6     case state  is
7     when "000" =>
8       if ch <="00101111" then
9         state <="001";
10      end if;
11    when "001" =>
12        ...
13            ...
14    when "100" =>
15      if ch <="00101111" then
16        state <="000";
17      elsif ch !="00101010" then
18        state <="011";
19      end if;
20    end case;
21    end if;
22    ...
```

Figure 2.14: Generated VHDL FSM code (partial).

Chapter 3

SOLAR System Design — It's all about connections!

3.1 Introduction

Based on [SZL05], a novel machine learning concept is proposed as Self Organizing Learning ARray (SOLAR). SOLAR is a multi-layered, hardware-oriented, local-connected learning structure aimed at the applications of data classification and clustering.

In past decades, many machine learning theories and systems [Wer74; Koh97; Vap98; HK01] have been proposed and applied to all sorts of application fields. SOLAR has approached the machine learning problem in a novel direction. Instead of setting up a fixed network topology and fixed processing functions, SOLAR allows an individual neuron to decide its own connections and functions. In addition, SOLAR has a structure of multiple layers where information has a better chance to be utilized by the network. In summary, the following characteristics differentiate SOLAR from other learning machines:

- the SOLAR structure is locally connected, which reduces the hardware implementation cost
- it has multiple layers allowing information to be fully digested by the network, unlike the traditional neural networks which adopt a three-layer structure

- the interconnections between neurons and neurons' functionalities are dynamically changed

The basic structure of the SOLAR system is illustrated in Figure 3.1. This concept has been successfully verified against various problems such as credit card problem, financial data analysis, data classification, pattern recognition, associative learning, and temporal sequence learning, etc.

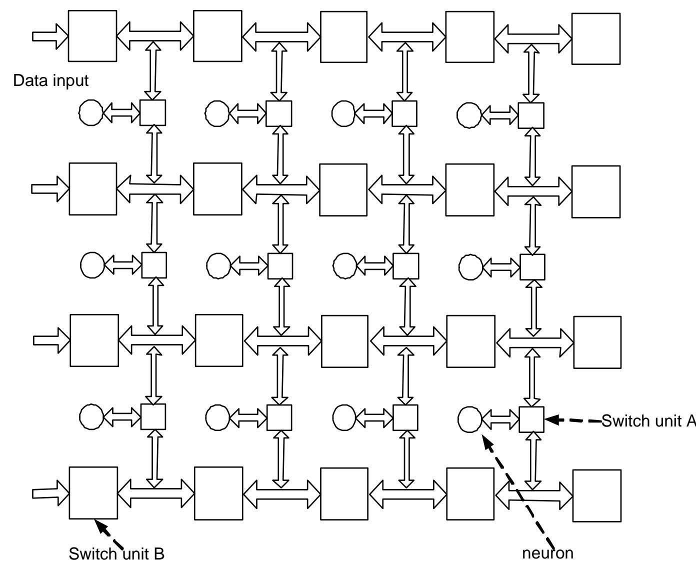


Figure 3.1: SOLAR structure.

One important study area of SOLAR system design is the input selection strategy applied by each individual neuron and the way the selected inputs are to be consumed by that neuron. The importance of neuron inputs is supported by recent research results in neuroscience which revealed incredible neural flexibility of the developing brain [EBJ+98]. In one experiment, the neurons in the visual cortex of rodents were transplanted to regions of the brain linked to bodily and sensory functions. The transplanted tissue begins to function like somato-sensory neurons and loses the capability of visual information processing [OS85]. Similarly, if the input from the eyes is rerouted from the normally visual area to the normally auditory area of the brain, the area that receives the visual input develops the visual processing ability

instead of audio processing ability. It can be concluded from these experiments that it is the input that determines the function of the brain [SPR90].

Inspired by these biological discoveries, SOLAR has assumed a networking scheme similar in fashion to the human brain, yet without the same level of complexity as human brain. Each neuron in SOLAR will decide its functionality and its connections to previous layers based on the input information and thus is dynamically changing with every input data sample. The dynamic connectivity and neuron functionality has empowered the SOLAR concept to deal with complex classification problems, but it also brings up many questions such as: how far can a neuron "see" from its layer? and what is the best way for a neuron to select its inputs within its reach?

In this chapter, I will focus on the connection problems in the SOLAR design. Basically, there are two topics covered.

1. Neuron input selection strategy. Given the limited hardware resources, each neuron has connections mostly to its neighbors. Therefore, it is worth studying the optimal way for a neuron to choose its connections and utilize the received information.

2. The hardware implementation for the reconfigurable inter-connection. The dynamic connection required by SOLAR has presented a great challenge to the system design. In previous works [SG03], ad hoc multiplexers are used to realize configurable connections between neurons. However, the multiplexer approach consumes too much hardware for a large array size, so a new routing scheme with less hardware demands is needed.

In the following sections, I will discuss these two topics and propose a workable solution to neurons' inter-connection problem.

3.2 Input Selection and Weighting Scheme

In the SOLAR system, each neuron will search for useful information from its neighbors to accomplish its own tasks. Since each neuron has limited connections due to hardware limitations, it is natural to ask which is the best way for a neuron

to select inputs from its neighbors. The following sections will discuss two different input selection strategies and their effects on the learning process. Since the strength of the information brought by each input is different, it is natural to assume a certain weighting scheme to best utilize the information. Detailed discussion of the optimal weight selection for each input is presented. A simplified binary weighting scheme is also discussed for the consideration of hardware implementation.

3.2.1 Random vs. Greedy

After a nonlinear transformations of the received signals, individual neuron's correct recognition rates are estimated based on the training sample probabilities. These probabilities are used in the merging neurons to arrive at the classification decision.

Merging neurons perform classification based on voting results from feature neurons or outputs of other merging neurons. Denote the jth merging neuron at ith layer as $n_{i,j}$. To optimize the learning network performance, each merging neuron will actively search for the most useful way to connect to the feature neurons and other merging neurons outputs. Due to the local and sparse connectivity structure of the SOLAR array, a neuron $n_{i,j}$ can only receive its inputs from neurons included in its *selection set* $R_{i,j}$. Each neuron $n_{i,j}$ will select $C_{i,j}$ neurons from $R_{i,j}$ as its actual input set, where $C_{i,j} \subset R_{i,j}$. When neuron $n_{i,j}$ receives all the information from its input set, a voting scheme is necessary to make correct classifications based on this information. In general, their selection sets $R_{i,j}$ and $R_{i,j+1}$ of two neighboring neurons in the same layer $n_{i,j}$ and $n_{i,j+1}$ will overlap. In this way, the neurons in ith layer will get different information from previous layer with some degree of redundancy. The size of the neurons' *selection set* and the amount of their overlap depend on the network size. Two strategies can be followed for the initial input set selection.

1. Random selection: Randomly select a connection from a neuron $\hat{n}$, where $\hat{n} \in R_{i,j}$ and $\hat{n} \notin C_{i,j}$, and check if $P(\hat{n}) > \min_{n_x \in C_y} P(n_x)$, where $P(\hat{n})$ is the recognition rate of the neuron $\hat{n}$, $P(n_x)$ is the recognition rate of n_x. If yes, $\hat{n}$ is selected to substitute neuron with the minimum $P(n_x)$, otherwise keep $C_{i,j}$ unchanged.

2. Greedy selection: Always choose connections from $C_{i,j}$ neurons which have the highest recognition rate within $R_{i,j}$.

These strategies result in various learning rates and classification qualities of the SOLAR array. In general, the neurons with recognition rates lower than 0.5 do not contribute to the final classification results and the connection to these neurons are useless or even harmful to a better classification rate. This is illustrated in Figure 3.2, where a merging neuron is connected to 40 inputs from other neurons, each with different classification probabilities ranging from 0.5 to 0.8. By combining inputs from these neurons using optimal weighting scheme (which will be discussed in later section), the merging neuron improves its recognition performance at a faster rate with the greedy selection strategy. In the next section, I will discuss how to combine the neural inputs using optimal weighting scheme and binary weighting scheme.

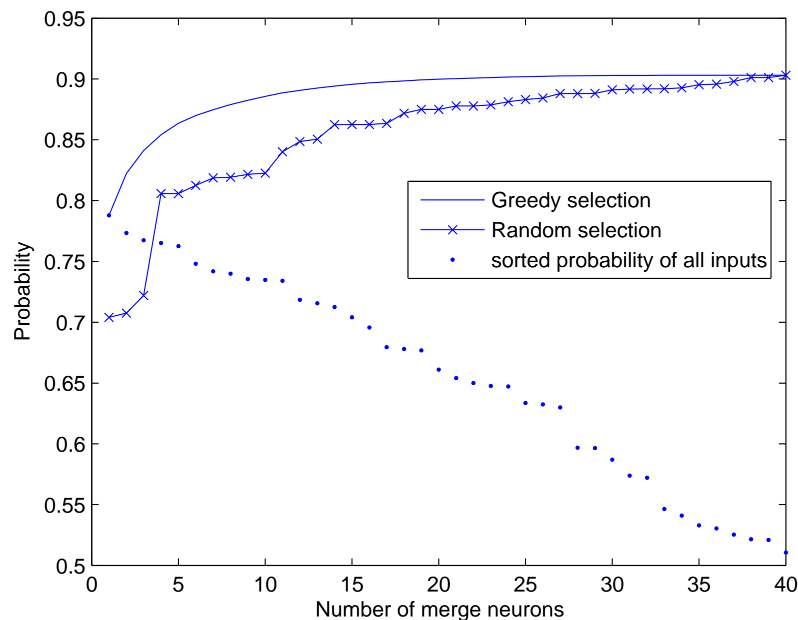


Figure 3.2: Classification rate with different selection strategies

3.2.2 Optimal and Binary Weighting Schemes

The optimal weighting scheme for the neuron input can be obtained by analogy to signal processing [SDH05].

First of all, a model of weighted sum of a set of noisy signals is established as shown in Figure 3.3, where k noisy signals with different weights are summed up to obtain a combined signal.

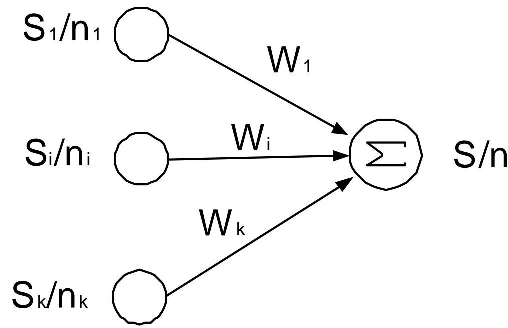


Figure 3.3: Model of weighted sum of noisy signals

For such an n input system with input signal s_i multiplied by a nonnegative weight, the combined signal energy is:

$$\hat{s}^2 = (w_1 s_1 + w_2 s_2 + \cdots + w_n s_n)^2 \tag{3.1}$$

Without loss of generality, it can be assumed that all the noise signals n_0 are the same and have noise energy equal to one. So the combined signal noise has energy

$$\hat{n}^2 = n_0^2 \left(w_1^2 + w_2^2 + \cdots + w_n^2\right) \quad . \tag{3.2}$$

In addition the weights are normalized by

$$w_1 + w_2 + \cdots + w_n = 1 \quad . \tag{3.3}$$

The objective is to find a set of weights that maximizes the combined signal to noise rate

$$\max_{w_i}(\hat{s}^2/\hat{n}^2) = F(w_i) \quad . \tag{3.4}$$

Take the gradient of $F(w_i)$

$$\nabla F_{w_i} = \frac{2\hat{s}\frac{\partial \hat{s}}{\partial w_i}\hat{n}^2 - \hat{s}^2\frac{\partial \hat{n}^2}{\partial \hat{w}_i}}{\hat{n}^4} = \frac{2\hat{s}s_i\hat{n}^2 - 2\hat{s}^2 w_i n_0^2}{\hat{n}^4} \quad . \tag{3.5}$$

For the maximum value of $F(w_i)$, it should satisfy

$$\nabla F_{w_i} = 0 \quad (i\text{=1, 2, \ldots n}), \tag{3.6}$$

therefore,

$$s_i\hat{n}^2 = w_i\hat{s}n_0^2 \tag{3.7}$$

and we have

$$\frac{w_i}{s_i} = \frac{\hat{n}^2}{\hat{s}n_0^2} = \frac{\sum_{k=1}^{n} w_k^2}{\hat{s}} \quad . \tag{3.8}$$

Since Eq. 3.8 should be satisfied for various signals s_i ($i = 1, 2, ...n$), with the identical terms on right side of all these equations, it must have

$$\frac{w_i}{s_i} = A = \text{const} \quad . \tag{3.9}$$

A. Optimal weights

In this section, Eq. 3.9 is used to derive optimal weighting for probabilistic self-organized learning. Thus the obtained weights that govern merging of classification probabilities from various neurons are optimum in the signal processing sense as described in the previous section. The two class classification problem will be discussed here. Neurons in the SOLAR network have different classification qualities measured by their recognition rate. In a two class problem this probability is based on the ratio of the number of samples from the class 1 over the total number of samples. Each neuron fires with a value representing probability that a sample came from class 1. The case of 0.5 represents the lowest amount of information, meaning that out of the two classes each one is equally likely. On the other hand, 1 or 0 represents full

information, meaning that the class type is known for certain. Analogous to signal and noise, a "signal" value and a "noise" value can be defined as follows:

$$\begin{cases} \bar{p} = p - 0.5 \\ \bar{n} = 0.5 - |\bar{p}| \end{cases} \tag{3.10}$$

where $\bar{p} \in [-0.5, 0.5]$ and $\bar{n} \in [0, 0.5]$. Then define the "signal-noise" ratio $\hat{p}$

$$\hat{p} = \frac{\bar{p}}{\bar{n}} = \begin{cases} \frac{p-0.5}{1-p} & p > 0.5 \\ \frac{p-0.5}{p} & p \leq 0.5 \end{cases} \tag{3.11}$$

Based on Eq. 3.9, the optimal weights are calculated by

$$w_i = \frac{|\hat{p}_i|}{\sum_{k=1}^{n} |\hat{p}_k|} \ . \tag{3.12}$$

Thus, the weighted result $\hat{p}_{out}$ is

$$\hat{p}_{out} = \frac{s}{n} = \frac{\sum_{i=1}^{n} w_i \hat{p}_i}{\sqrt{\sum_{i=1}^{n} w_i^2}} = \frac{\sum_{i=1}^{n} |\hat{p}_i| \hat{p}_i}{\sqrt{\sum_{i=1}^{n} |\hat{p}_i|^2}} \ . \tag{3.13}$$

Substituting $\hat{p}_{out}$ into Eq. 3.11 and get equivalent p_{out}

$$p_{out} = \begin{cases} \frac{\hat{p}_{out}+0.5}{1+\hat{p}_{out}} & \hat{p}_{out} > 0 \\ \frac{0.5}{1-\hat{p}_{out}} & \hat{p}_{out} \leq 0.5 \end{cases} \tag{3.14}$$

p_{out} represents the belief that the classified sample belongs to class 1. An experiment of three inputs combination is conducted to illustrate the combined results for different input combinations. Table 3.1 [SDH05] shows six sets of inputs (p_1, p_2, p_3) and the calculated p_{out} based on the above analysis. It can be seen that when all three inputs agree with the class information (all three greater than 0.5 or all smaller than 0.5), the combined result will further strengthen this agreement, otherwise, it will become weaker (the fourth case).

B. Binary Weights

The results obtained in Eq. 3.14 are based on the derived optimal weights applied

P_1	0.1	0.6	0.6	0.1	0.1	0.2
P_2	0.1	0.6	0.6	0.9	0.2	0.2
P_3	0.6	0.6	0.8	0.4	0.1	0.2
$\hat{P}_{out}$	-5.64	0.433	1.541	-0.011	-5.852	-2.598
P_{out}	0.0752	0.6511	0.8032	0.4945	0.073	0.139

Table 3.1: Calculation of the output recognition rate

to selected input neurons $C_{i,j}$. However, in practical hardware implementation, a simplified interconnection scheme is always desired. A binary weight does not require multiplication to obtain the combined input signal. A neuron is either wired to a node from its selection set $R_{i,j}$ or not. This corresponds to choosing $1/n$ as weights for all the connected inputs (n is the number of connections). Based on Eq. 3.13, the signal noise ratio is calculated as follows:

$$\frac{\hat{s}^2}{\hat{n}^2} = \frac{\left(\sum_{i \in C_{i,j}} \frac{1}{n}\hat{p}_i\right)^2}{\sum_{i \in C_{i,j}} \left(\frac{1}{n}\right)^2} = \frac{\left(\sum_{i \in C_{i,j}} \hat{p}_i\right)^2}{n} \tag{3.15}$$

or

$$\frac{\hat{s}}{\hat{n}} = \frac{\left|\sum \hat{p}_i\right|}{\sqrt{n}} \quad . \tag{3.16}$$

Eq. 3.16 will be used to study the effect of adding connections of different signal strengths at a neuron's input. Denote the stronger connection as P_{max}, and the weaker connection, that is to be added, as P_{mix}, assuming that P_{max} and P_{mix} are both greater then 0.5. Thus the corresponding scaled variables $\hat{P}_{max}$ and $\hat{P}_{max}$ are

$$\begin{aligned} \hat{P}_{max} &= \frac{P_{max} - 0.5}{1 - P_{max}} \\ \hat{P}_{mix} &= \frac{P_{mix} - 0.5}{1 - P_{mix}} \end{aligned} \tag{3.17}$$

The recognition rate will be calculated from the combination of the two inputs as $\hat{P}_{comb}$ and then obtain P_{comb}, which will give an estimate of the recognition rate if P_{mix} is

appended to P_{max}. The effect of combing this new connection with P_{mix} is shown in Figure 3.4, where $dP = P_{max} - P_{mix}$, is the difference between the existing stronger connection and the possible new connection, and P_{gain} defined as $P_{gain} = P_{comb} - P_{max}$, indicates the improvement of the final probability due to P_{mix}. From Figure 3.4 one can see that the P_{gain} decreases as the P_{mix} becomes smaller. Based on Figure 3.4,

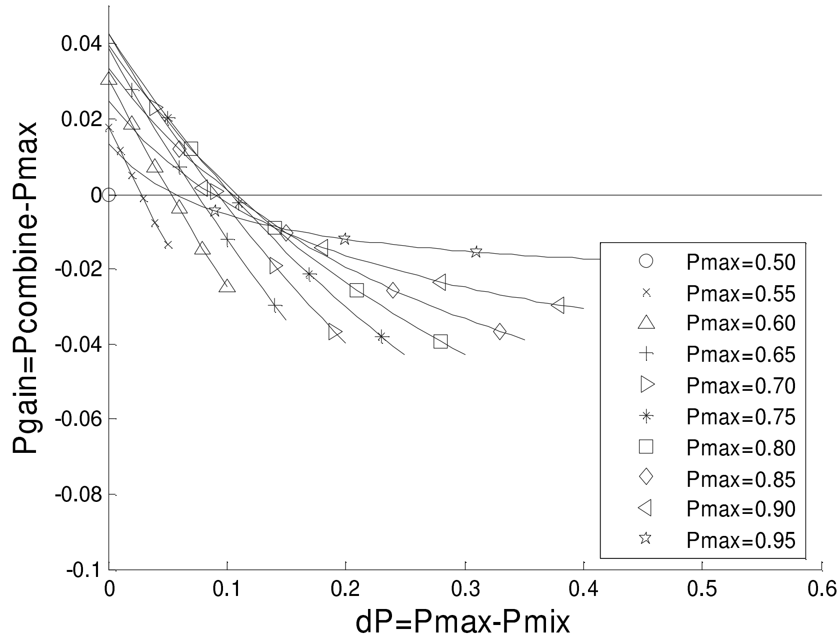


Figure 3.4: P_{gain} vs. dP in a binary weighted connection

the minimum acceptable P_{mix} can be obtained in order to have a positive P_{gain} as shown in Figure 3.5. This can be used as a criterion to decide which new connections are acceptable and which are not.

As can be seen from Figure 3.5, only neurons of similar recognition rates can be merged to improve the classification performance.

3.2.3 Simulation Results

In this section, several simulation results will be given to illustrate the effect of the weighting scheme.

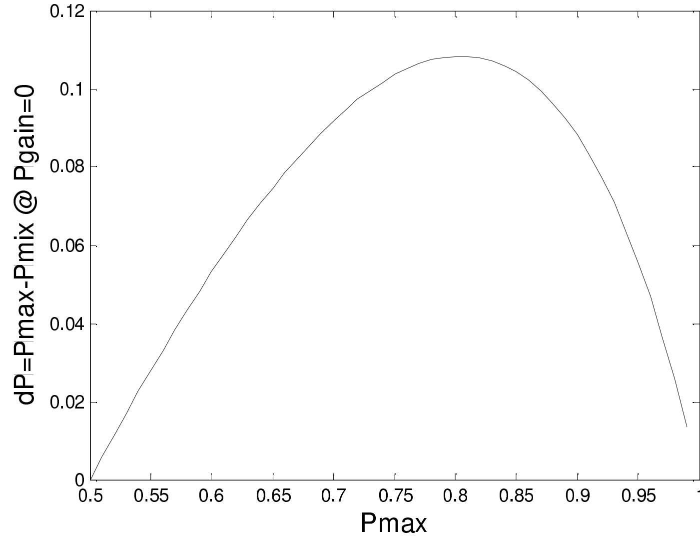


Figure 3.5: P_{gain} vs. dP at $P_{gain} = 0$ in a binary weighted connection

Experiment setting:

A 16-input binary neural network is constructed and tested against two classes of input patterns with different levels of noise.

Two 16-bit vectors have been selected as two original vectors representing class 0 and 1 respectively.

V_0 = "1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0"

V_1 = "1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1"

Based on these two original vectors, random noise is added to create two sets of vectors, the vectors generated from V_0 are marked as class 0, and vectors generated from V_1 are marked as class 1. When noise is added to these two original vectors, certain bits of these two vectors will flip from '0' to '1' or vice versa, a noise of strength n specifies the maximum number of bit reversing that can happen in the original vectors. Denote the noisy vector as V_n, and the Hamming distance from V_n to V_0 and V_1 are $d_{n,0}$ and $d_{n,1}$, respectively. For a weak noise, it can be sure that $d_{n,0} < d_{n,1}$ for each vector generated from V_0 and $d_{n,0} > d_{n,1}$ for each vector generated from V_1. When the noise is strong, there will be certain vectors whose $d_{n,0}$ and $d_{n,1}$

will not obey this rule. This imposes an upper limit for the correct recognition rate, since the Hamming distance is the only way to tell apart these vectors.

To visualize the distribution of these noisy vectors, vector V_0 and V_1 are translated to two squares in the 2D plane at position (0,0) and (8,0) since the original Hamming distance between vector V_0 and V_1 is 8. Each noisy vector V_n is translated to a point in the 2D plane whose Euclidean distances to those two squares are $d_{n,0}$ and $d_{n,1}$, as illustrated in Figure 3.6, where the two squares represent the original noise free vectors, dots represent the class 0 vectors, and circles class 1 vectors.

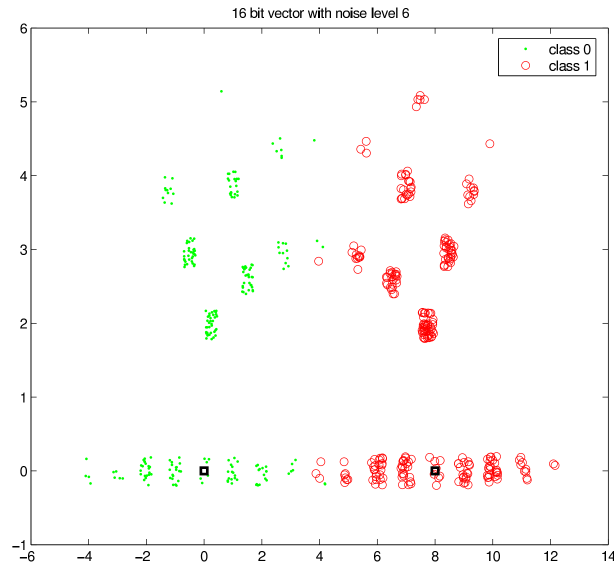


Figure 3.6: Visualization of Hamming distances for noisy vectors

A network of 31 neurons (Figure 3.7) is constructed and tested with optimal and binary weighting schemes and tested against four sets of data with different noise strength levels of 6, 8, 10 and 12. This means the maximum bit reversing that can happen on original vectors is 6, 8, 10 and 12 for these four datasets. The result is listed in Table 3.2. and the binary weighting has shown comparable performance as the optimal weighting for different strengths of noises.

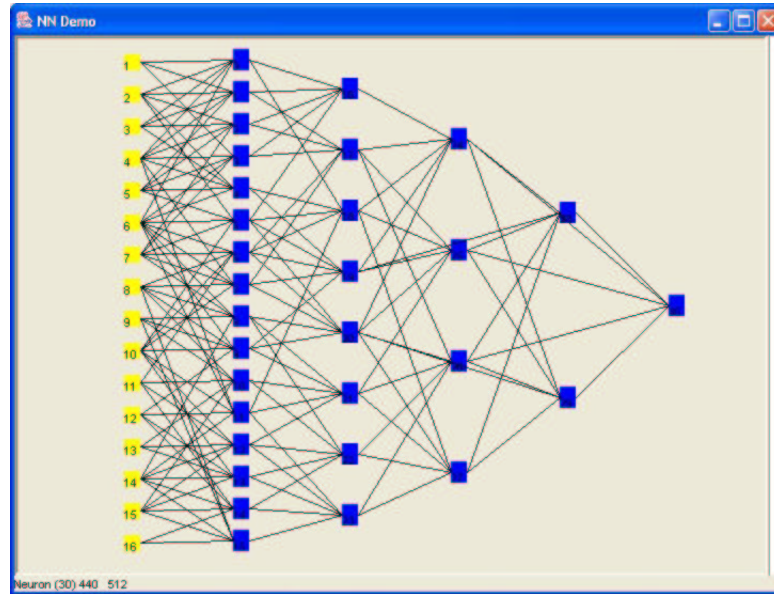


Figure 3.7: A 31-neuron network topology

dataset	weighting	recognition rate (%)	upper limit (%)
1	binary	96.1	98.1
1	optimal	95.9	98.1
2	binary	92.1	93.9
2	optimal	92.5	93.9
3	binary	88.1	90.5
3	optimal	88.3	90.5
4	binary	84.0	85.1
4	optimal	84.3	85.1

Table 3.2: Comparison of binary and optimal weighting

3.3 Reconfigurable Routing Channel

In the world of VLSI design, routing has been deemed as an extremely difficult yet very important problem from the very beginning of IC design technology and development of supporting EDA tools. As the transistor sizes keep shrinking, and operating frequency keeps soaring, the interconnect delay and capacity have shown ever greater impact on the final system performance. This problem has already attracted significant research interests and many routing algorithms have been developed with priority given to different factors like area, speed, delay time, power consumption, etc.

In the field of programmable devices, the flexibility of interconnection is usually achieved by switch boxes. Well designed switch boxes can provide very flexible and versatile routing for a wide spectrum of applications. However, the switch box is also known to consume a lot of energy and is slow to operate, as illustrated in [KAW04]. The appearance of the reconfigurable computing system requires dynamic changes in interconnections between functional units. This presents additional challenges to an already difficult routing problem since most routing algorithms treat all the functional units together. Thus a few changes in local connections are likely to cause a recalculation of the whole routing process, particularly for a big design.

Conventional EDA tools generally will create large number of multiplexers for configurable datapath [MM04]. This approach can achieve reconfigurability and does not involve complex routing algorithms. However, there are several drawbacks to this approach. First of all, a significant amount of silicon area is dedicated to wiring and multiplexing, because for the multiplexer-based wiring method, all the possible connections need to be pre-wired and switched on/off by multiplexers. The number of possible connections among n nodes grows at the rate of $O(n^2)$ and the average wire length increases at least at a rate of $O(n^{0.5})$ [Don81], thus the total design area occupied by wires grows at a level of $O(n^{2.5})$ or higher. This becomes prohibitively expensive if the number of nodes exceeds 1000. In addition, the growing network size requires fast growth in the number of wires and needs more multiplexers to configure. This shows that the multiplexer approach is only suitable when a small number of

reconfigurable connections are demanded. In the case of the SOLAR system, dynamic connections are required all over the array, while the switch box offers more flexibility than SOLAR needs. Therefore, a new routing scheme is desired to offer the flexibility SOLAR needs, with less overhead than switch boxes and multiplexers.

In the following sections, I will propose a novel pipeline structure focusing on the dynamically changing connectivity. The basic idea here is to utilize the computing ability of each node in the network to perform "soft" connections. First of all, the parallel input data are transformed to a sequential data stream with a predefined repeating ratio; this data stream is then bound with timing information. Each node can use this timing information to decide which slots of the data stream it will read from and which slots it will write back to. In this way, the nodes at earlier layers can pass the information to the nodes at the following layers. The advantage of this structure is that the connections between nodes are fully configurable inside the corresponding nodes without affecting other nodes' operations. Different connecting configurations can be implemented under the unified pipeline structure, thus saving the global wiring and multiplexing area, and avoiding complicated routing algorithms.

3.3.1 Pipeline Structure

The whole network is constructed in a rectangular 2D array. Each column of the array consists of a long shift register and several processing nodes (neurons) are attached to each column to perform the read/write operation. These shift registers implement routing channel through which all neurons are connected to input/output nodes and to each other. Each input data item is first repeated several times, and then fed to the network sequentially. This repetition of input data introduces necessary redundancy for pseudo-randomly connected neurons. After the first column finishes data processing, the results are shifted to the next column. Each processing node has four different working modes:

1. Idle mode
2. Reading mode
3. Processing mode

4. Writing mode

Before I continue with the detailed description of the structure and its data operations, let us first introduce some terminology to describe the network parameters:

c – Copy ratio, defines how many times every data is repeated before next data are fed in.

k – The number of nodes in each column.

N – The number of input data.

L – The total length of one column, $L = c \times N$.

R – The input range for a node (which is the same for all the nodes). It specifies the maximum range of nodes from previous layer a node is able to "see".

$\{P_1, P_2, \dots P_k\}$ – A vector of length k, the ith element of this vector specifies the read/write position for ith node in the current column. Since each data item is copied c times, P_i ($i = 1, 2, \dots k$) is always a multiple of c.

Because c indicates the effective size of the routing channel, it should increase proportionally with the average number of neuron inputs. k is related to the number of inputs, and R determines the neighborhood size of each neuron. R is small for locally connected neurons. The optimum choices of design parameters c, k, R, etc. are application specific. At the beginning, the node is in the idle mode for certain cycles, then it enters into the reading mode during which time it will read selected data copies from the data stream sequentially. A single 8-bit data can be read by each node at any given time during the reading mode. Then it begins the processing mode to process the read data and write back the results to the same slots from which data were read during the writing mode. In other words, the node consumes certain parts of the input data, and replaces them with its own result. By repeating each input data c times, it creates a chance for the data to survive for several processing layers. By adjusting the actual reading time of each node, many different connections between nodes can be implemented without increasing any wiring cost. For example, the structure in Figure 3.8 can implement different connection configurations as shown in Figure 3.9 with a simple change of the reading time for the corresponding nodes.

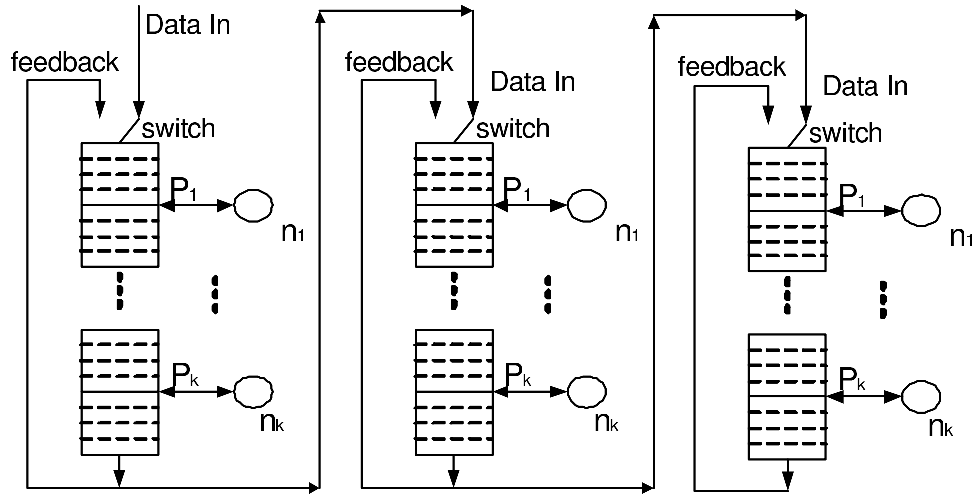


Figure 3.8: Pipeline structure overview.

Figure 3.9: Same network with two connection configurations.

Of course, this flexibility comes at the cost of extra timing and controlling hardware. In the following section a more detailed description of the dataflow in the implemented interconnect scheme will be presented.

3.3.2 Data Flow Description

First of all, the timing circuitry is based on the clock that drives the shift registers; all the operations are synchronized to the rising edge of this clock. Therefore, the input data will fill the whole column after $L = c \times N$ cycles. The operation of this structure is pipelined from column to column. Each data item of the first N input data is first repeated c times, and then fed to the first column sequentially. All the

nodes are in idle mode before P_k cycle, and then the nodes begin reading data from certain slot. At cycle L, the switch at the top of the column switches and the data begin to circulate in the first column. All the nodes enter the processing mode after reading the input data, and should finish their computing tasks no later than $L + P_k$ cycle (if the longest combined reading time and computing time of all the nodes exceed L cycles, an additional L cycles need to be inserted for these two operations), when all the nodes begin to write their processed results back to the specified slots as displayed in Figure 3.10. By the end of cycle $2L + P_k$, the nodes should finish all the writings and enter the idle mode as illustrated in Figure 3.11 with the same legend as in Figure 3.10.

At the arrival of the $3L$ cycle, the switch switches back and the next N input data begin feeding into the first column, and the content of the first column is copied to the second column (Figure 3.12 with the same legend as in Figure 3.10). From the above description, it can be concluded that the pipeline delay between two columns is $3L$ cycles. To further clarify the dataflow during the first $4L$ cycles, a timing diagram for the column 1 and column 2 nodes operations is illustrated in Figure 3.13.

This computing scheme in which pipeline data is transported sequentially will yield a performance slightly lower than full parallel hardware implementation, but significantly higher than sequential operation on a single processor. In general, if a node requires p cycles to process the input data, then fully parallel implementation requires $p + r$ cycles, where r is the average number of the neuron's inputs. The proposed pipeline scheme requires $L(2+\lceil p/L \rceil)$ cycles, and sequential implementation requires $N \times p$ cycles to complete. For large p, the performance of the proposed pipeline structure is similar to the fully parallel implementation, while for small p, its performance is a function of the channel size. Thus it is a good compromise between hardware cost and performance.

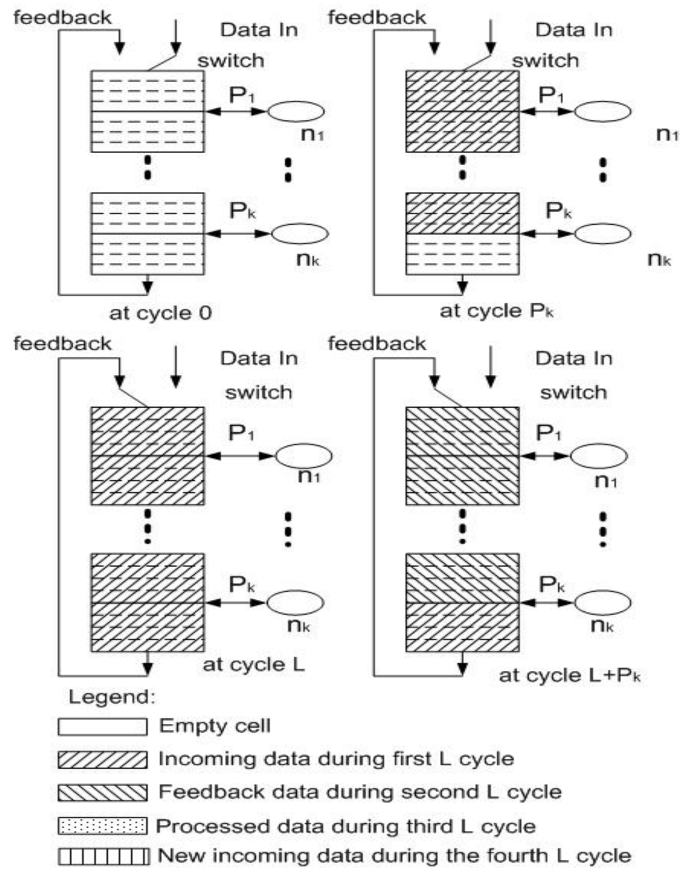


Figure 3.10: Column 1, cycle 0 to cycle $L + P_k$.

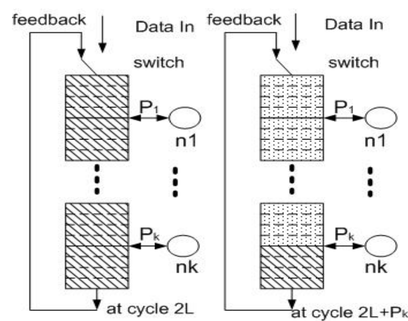


Figure 3.11: Column 1, cycle $2L$ to cycle $2L + P_k$.

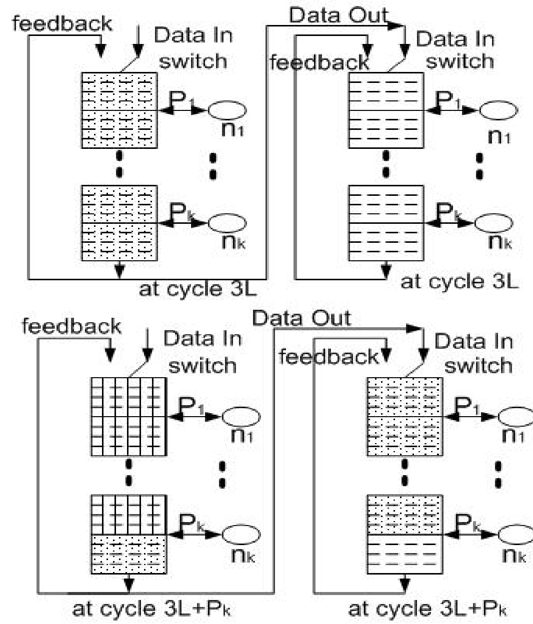


Figure 3.12: Column 1 and 2, cycle $3L$ to cycle $3L + P_k$.

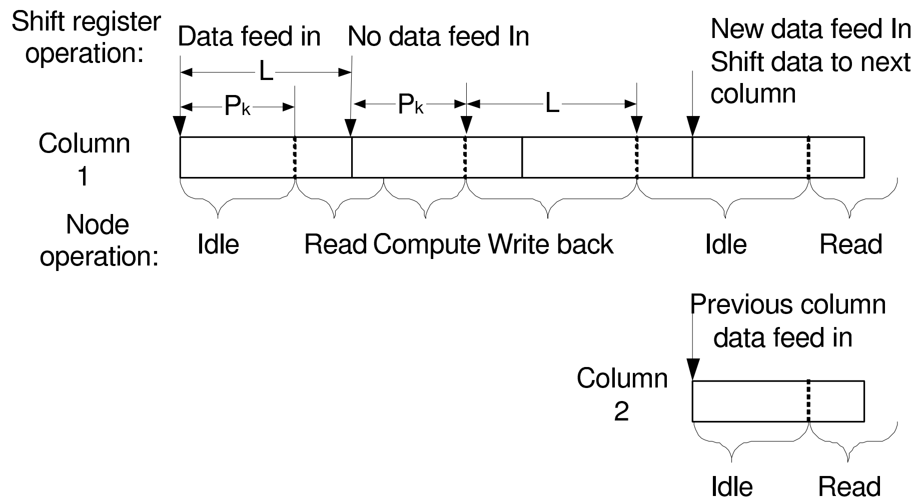


Figure 3.13: Timing diagram between column 1 and 2.

3.3.3 Node Operations

A. Read/Write operation

In the pipeline design, each node is implemented with a Xilinx picoBlaze soft processor. Based on the Constant (k) Coded Programmable State Machine (KCPSM), the picoBlaze processor is a fully embedded 8-bit RISC micro controller soft core optimized for Xilinx devices. It has one 8-bit input port, one 8-bit output port and one 8-bit address port. To correctly operate as described in the previous section, the node must be running at a higher speed than the shift register, because during one shift register clock period, the node needs to read timing information, make comparison with its own storage and read data at that time if necessary. For a clear description of the operation, the period of that higher speed clock is denoted as *node-cycle*. Figure 3.14 shows a piece of KCPSM assembly code which instructs the node to read data from slot 4. First, it needs to read the current timing value, and check if it is equal to 4, if yes then it needs to read current data and store it to its internal register, otherwise it keeps waiting for the next data.

```
1     index1:
2        IN     s0,  iport
3              SUB    s0,  4
4              JUMPZ  read1
5              JUMP   index1
6     read1:
7        IN     s0,  dport
```

Figure 3.14: KCPSM assembly code fragment for reading.

After a node finishes reading all the slots of data as instructed, it will begin working in the processing mode. From Figure 3.10 to Figure 3.12 and the previous description, the node has $L - c \times R$ cycles to perform its computing task. Figure 3.15 illustrates how each processing node is attached to the column to perform read/write operations. Register 1 and register 2 are parts of the shift register that implement the routing channel of the hybrid pipeline structure. The "sel" signal is always switched

to let data flow from register 1 to register 2 except when the node decides to output its processed result.

As stated before, the picoBlaze core is running at a frequency higher than the shift register clock frequency. Denote the ratio between two clocks as m. The computing time for the required calculation must be the multiple of m. For example, if a node is to perform an exponent operation on slot 4, it needs to finish the operation in km ($k = 1, 2, 3, \ldots$) *node-cycles*. Based on previous data flow analysis, the node's computation time has an upper limit in order for the whole pipeline to work correctly. Thus, optimization of computing time is necessary especially for complicated operations, like exponent, logarithm, sigmoidal, etc.

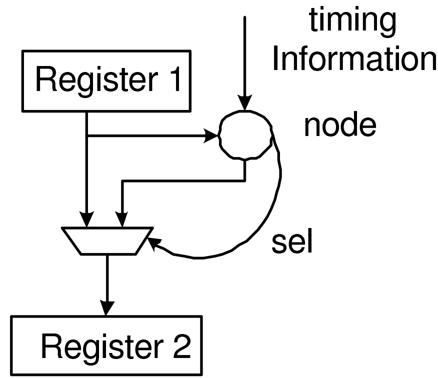


Figure 3.15: Single node read/write structure.

B. Arithmetic operation

After data have been correctly read, each node in the array will perform certain arithmetic functions on these input data. Currently the nodes can perform the following functions:

1. unary operations: *half, identity, log, exp, sigmoid*

2. binary operations: *add, sub*

Among these functions, *half* and *identity* function are the easiest to obtain. Binary functions *add* and *sub* are also very close to their conventional meanings:

$$Am(a, b) = (a/2 + b/2) \tag{3.18}$$

$$Sm(a,b) = \begin{cases} a-b & a \geq b \\ 0 & a < b \end{cases} \tag{3.19}$$

where $Am(a,b)$ is the modified *add* function and $Sm(a,b)$ is the modified *sub* function. The remaining functions need some simplification from traditional mathematical calculation with the consideration of hardware implementation. For example, the conventional exponent calculation $y = \exp(x)$ is modified so that for input included in interval $[0, 255]$, the output belongs to $[0, 255]$ as well. Therefore, no overflow or underflow will ever occur in the whole array computations. Denote $Em(x)$ as the modified exponent function, we have

$$Em(x) = [1 + frac(x/32)] \times 2^{floor(x/32)} \tag{3.20}$$

where $frac(x)$ is the fraction part of x and $floor(x)$ is the integer part of x in binary format. To implement this equation in the KCPSM assembly code, the procedure is as following: for the 8-bit integer x, take its lower 5 LSBs as its fraction part then add 1 to it making a temporary value s, take its 3 MSBs as its integer part, then s is shifted left according to the value of x's integer part. Denote $Lm(x)$ as the modified logarithm function, we have

$$Lm(x) = 32 \times [Li(x) + frac(x/2^{Li(x)})] \tag{3.21}$$

where $Li(x)$ is actually the bit position of x's most significant bit of '1' in its binary expression. Table 3.3 lists the $Li(x)$ values for different x values.

x	0	1	2	4	8	16	32	64	128
Li(x)	0	0	1	2	3	4	5	6	7

Table 3.3: $Li(x)$ value table.

The assembly code for $Lm(x)$ is developed in a similar fashion. Figure 3.16 illustrates the differences between the modified logarithm, modified exponent, and their conventional scaled peers. Many other useful functions can be created from the combinations of these basic functions. Since sigmoidal function is an important function in neural networks, it is useful to have a hardware implementation for this function.

In order to make use of the simplified calculations based on the $Em(x)$ function, I am going to use sigmoidal transformation in the following format.

$$f(x) = sign(x)(1 - e^{-|x|}) \tag{3.22}$$

First, the MSB of input x is used to decide if x is treated as positive or as negative, then based on $Em(x)$ function, modified sigmoidal function is obtained. Such defined sigmoidal function and its approximation based on $Em(x)$ are computed (actually the conventional sigmoid curve is shifted and scaled before comparison) and displayed in Figure 3.17

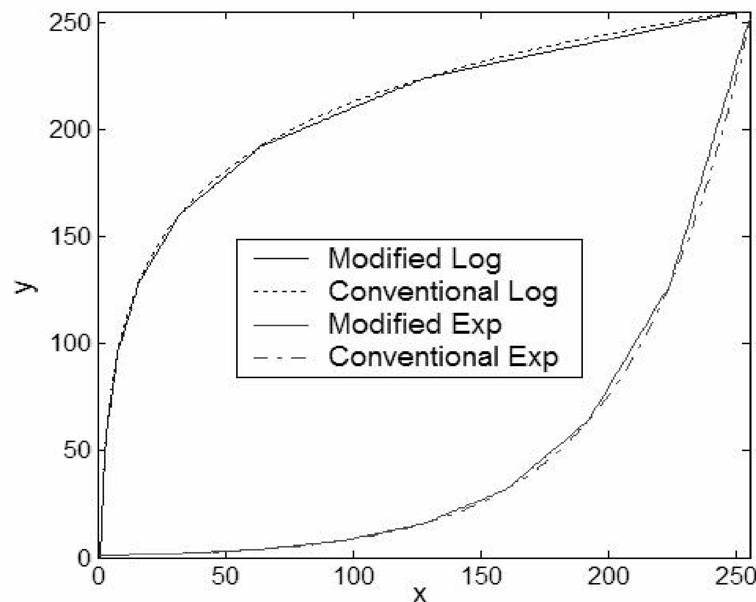


Figure 3.16: Node implementing $Lm(x)$ and $Em(x)$ operations.

Assembly codes for all these functions have been developed and special care has been taken to ensure the calculation time is optimized and is the multiple of *m node-cycles* to make the node computation synchronized with the shift register speed.

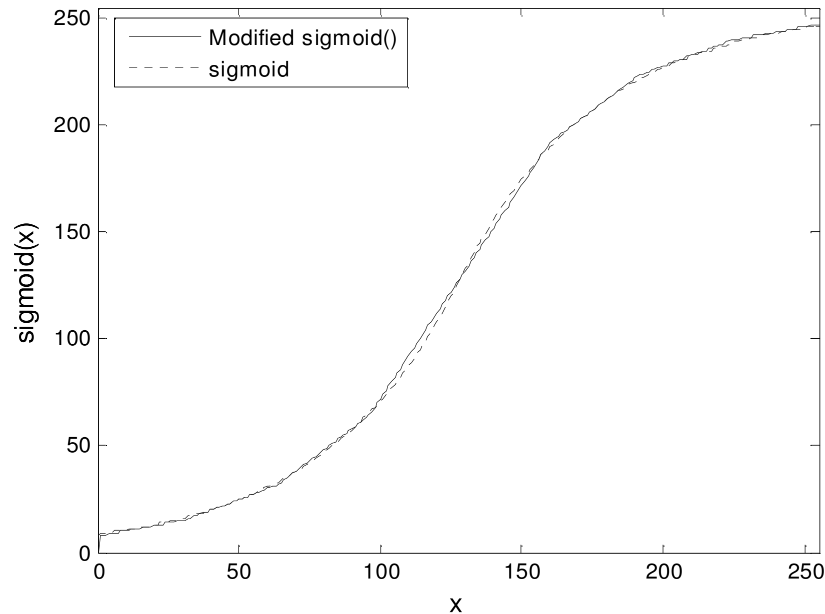


Figure 3.17: Comparison between modified sigmoid and sigmoid.

3.3.4 Simulation Results

In this section, I will first give the simulation results of a single node performing read/write and exponent calculation. Then a 4-row array is built for reading and processing Iris database samples. Lastly, four arrays of different sizes are constructed and their design areas are compared. The respective read/write waveform for a single node is illustrated in Figure 3.19. The node is configured to read data at slot 4 and 5, and perform an add operation, and then write the results back to slots 4 and 5. Here the timing signal tin is artificially set for a quick verification of one single node's operations. As one can see, the node reads the data values 47 and 57 at slot 4 and 5, according to the previous description, $Am(47, 57) = (47/2 + 57/2) = (23 + 28) = 51$, and the summation 51 is output by the node when it "sees" slot 4 and 5 again. Corresponding assembly codes are developed for all the arithmetic operations discussed in the previous section and VHDL test benches are created to test the correct response for all possible 256 x values. Figure 3.18 shows simulation result for a node performing $Em(x)$ function ($Em(192) = 64$).

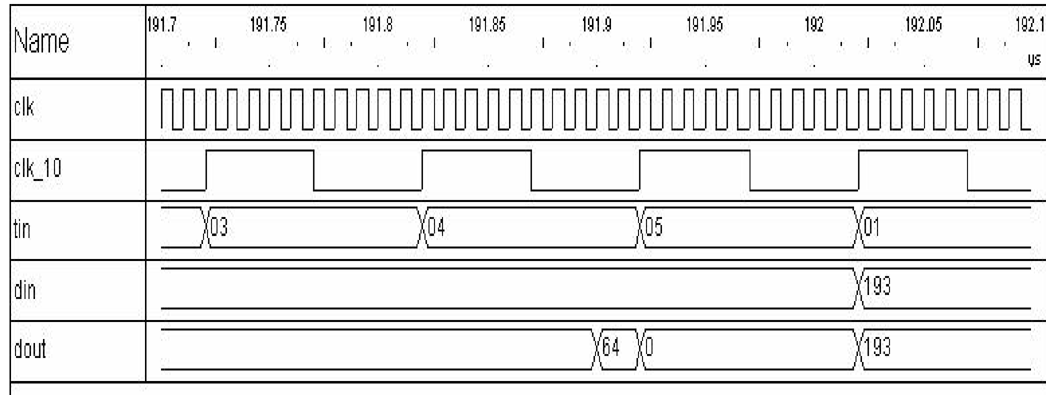


Figure 3.18: Node implementing $Em(x)$ operation.

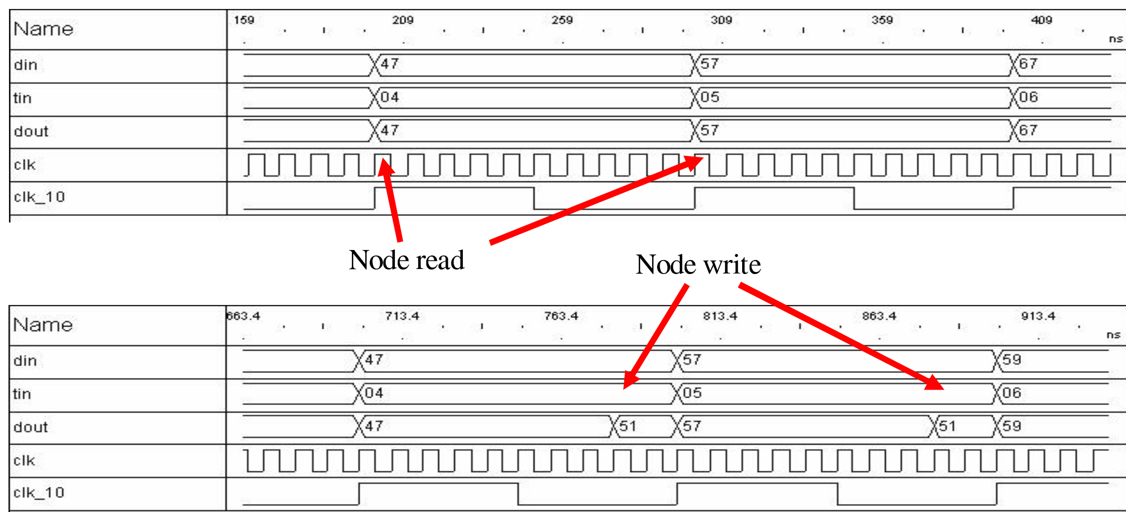


Figure 3.19: Single node read/write waveform.

To demonstrate the proposed pipeline approach's advantages when expanded to a network of such size, additional arrays of 4×6, 4×12 and 4×24 were constructed and synthesized targeting Xilinx Virtex II 8000 chip. The design area (number of slices) is observed to be a linear function of the network size (number of nodes) as shown in Figure 3.20, while the maximum system clock is consistently kept at 81.1 MHz.

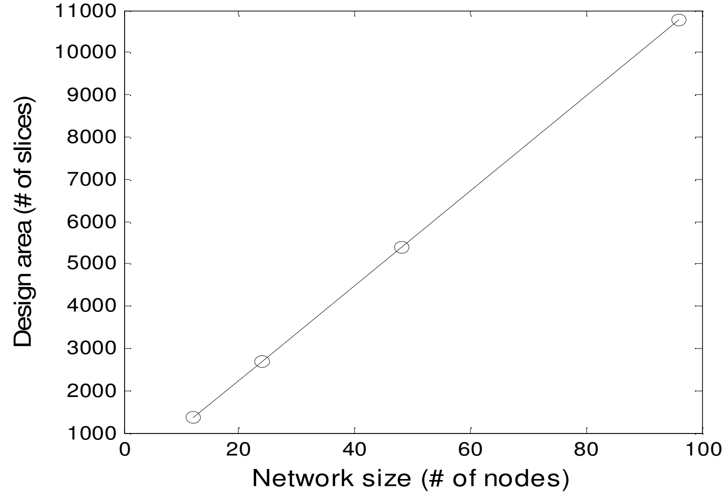


Figure 3.20: Design area vs. network size.

3.4 Application Example

In this section, an Iris data application sample will be given to illustrate the working results of the proposed pipeline scheme. The Iris database [UCI05] has 3 classes, 150 samples total with each sample consisting of 4 features. The features are first normalized to [0, 255] and then fed to the SOLAR array as input data. A 4×7 SOLAR array was established to process this 4-feature data.

The connections of the learned array are shown in Figure 3.21, where the symbol inside the circle shows the functionality of that neuron (an empty circle means that neuron just passes the information without any changes) and the letters on the arrows represent the arithmetic operation for a two-input neuron. Letter "H" means *half* operation, "L", *logarithm*, "E", *exponential* and "=", *equal*.

The final output is read from the last column and verified with the results by Matlab simulation.

In the following paragraphs are shown the schematics for this 4×7 array after synthesis and the corresponding layout after mapping is displayed. From Figure 3.22 to Figure 3.25 the RTL schematics are shown for the components at different levels. Figure 3.26 illustrates the layout after place and route process.

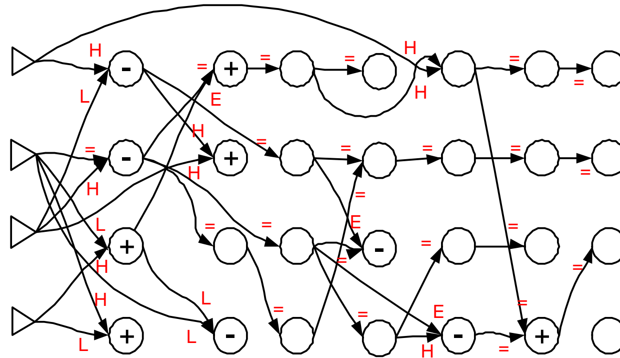


Figure 3.21: 4 × 7 Array processing 4-feature Iris data.

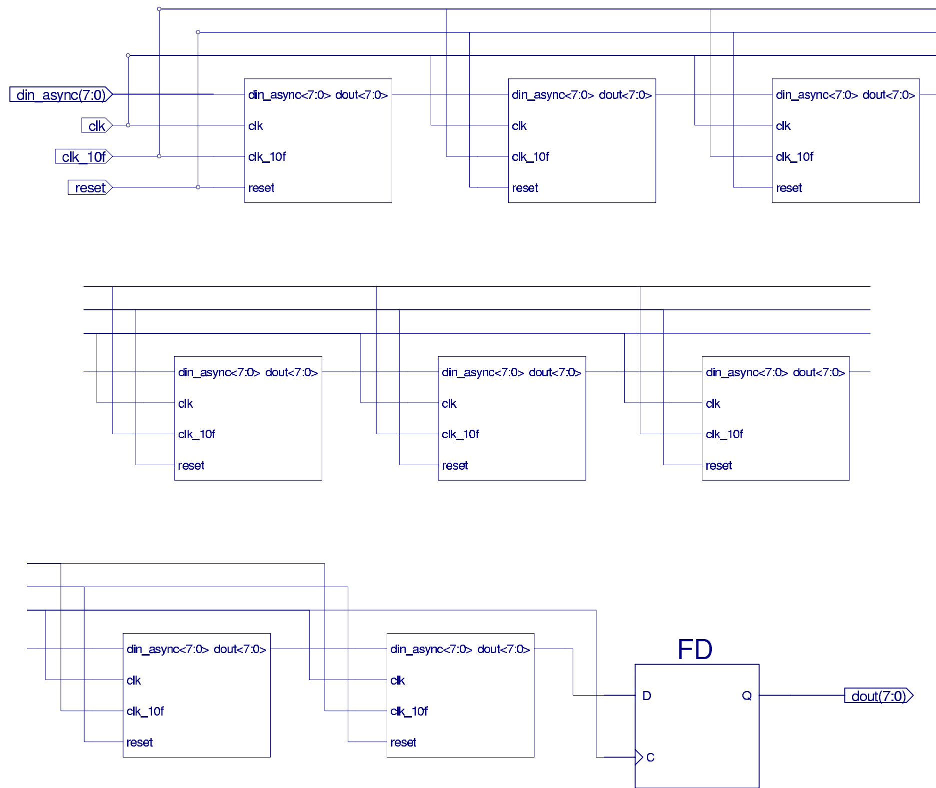


Figure 3.22: 4 × 7 array RTL schematic.

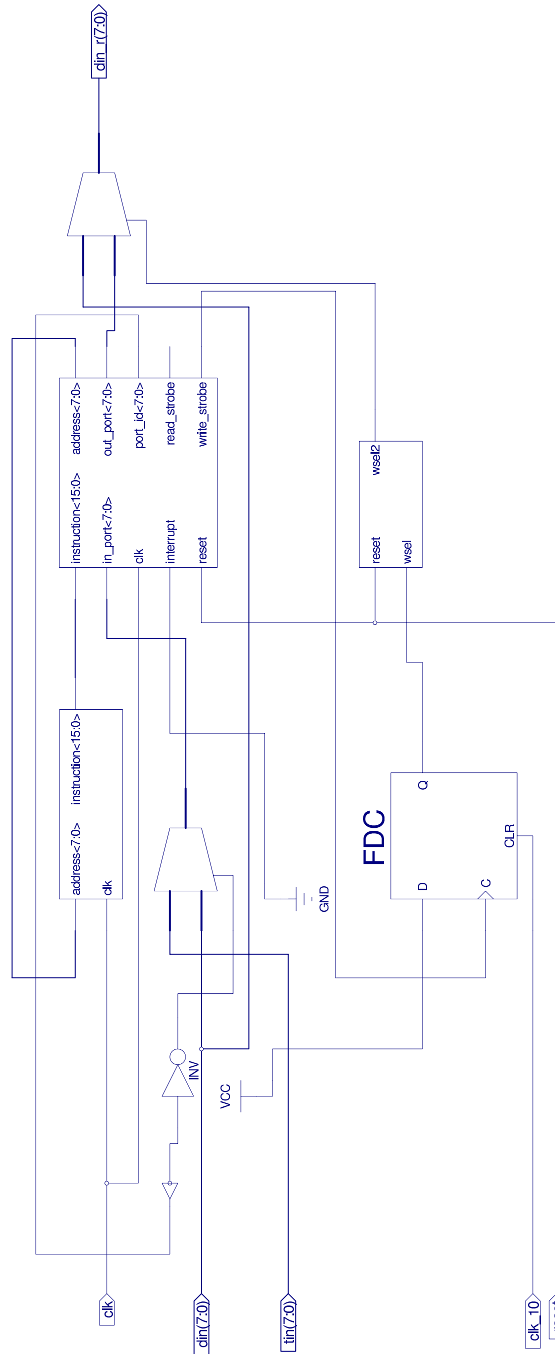


Figure 3.23: Node RTL schematic.

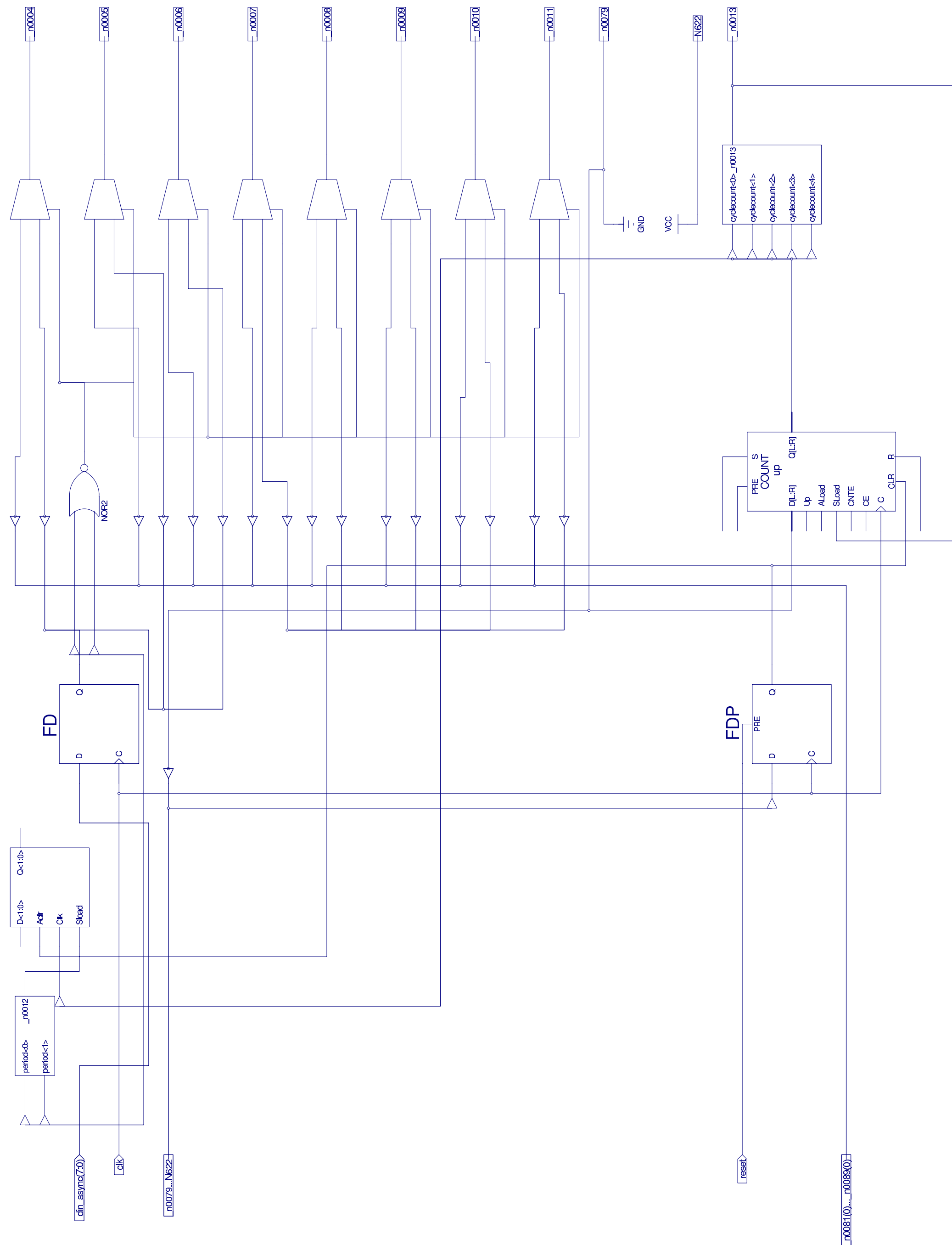


Figure 3.24: One column RTL schematic.

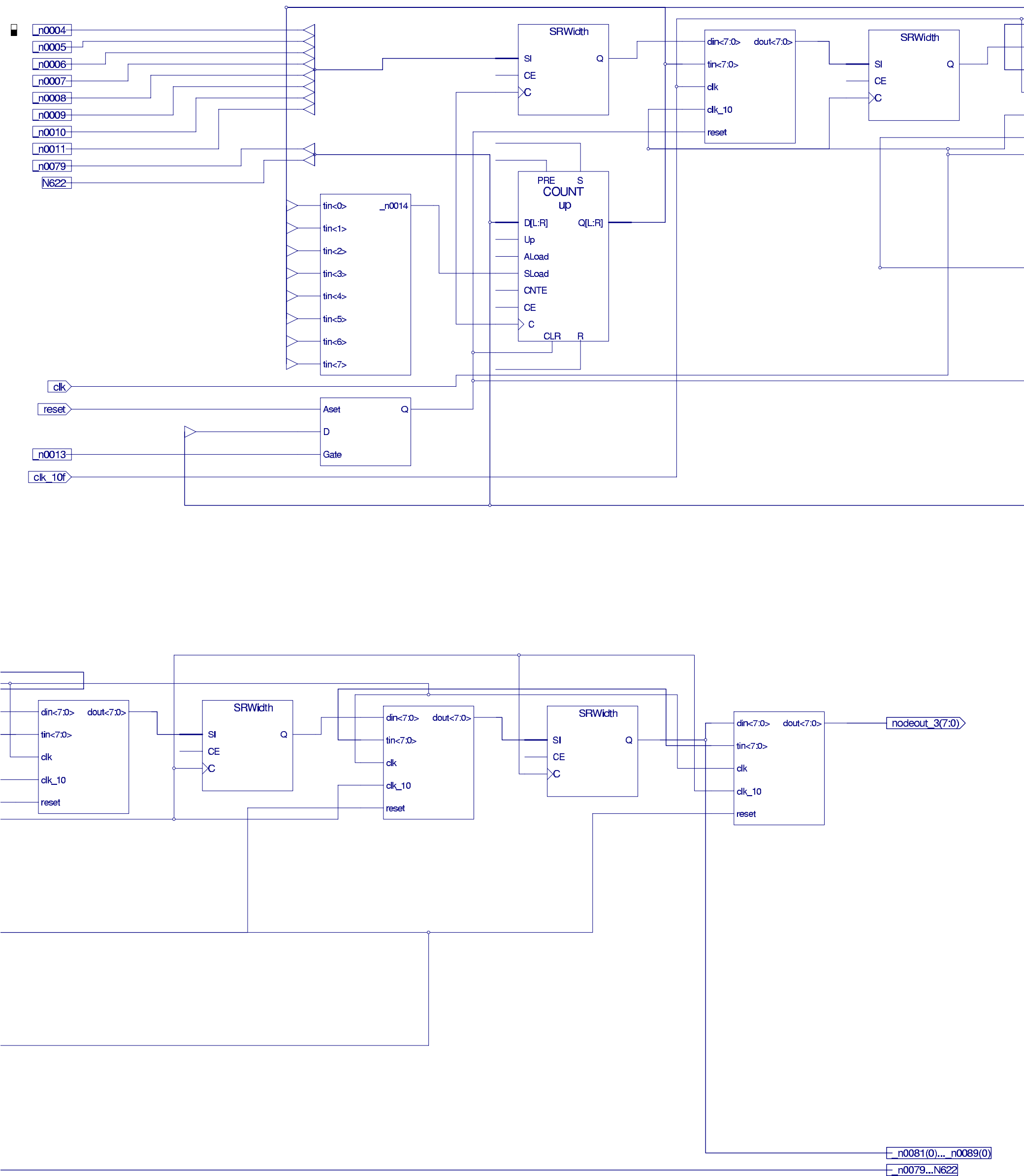


Figure 3.25: One column RTL schematic (2).

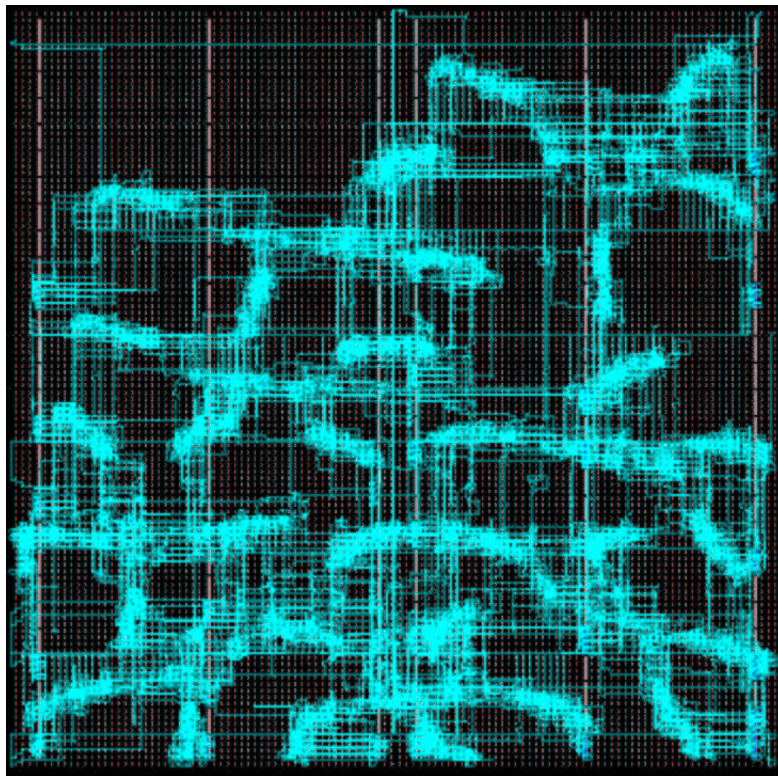


Figure 3.26: 4×7 array layout by Xilinx tools.

3.5 Contributions by UADL

In this section, the contribution of UADL to the designing process of this pipeline structure is given. There are two places UADL has been applied to improve the design productivity. The first place is in the design process of the pipeline structure. As illustrated in Figure 3.8, the pipeline is constructed from columns and each column is regularly composed of shift registers and neurons. With the ability of index expanding, UADL can express this kind of structure very concisely. One column structure is built with UADL as displayed in Figure 3.27. The "$libunit" command

```
1     target = v @visitor.VHDLWalker ~vhd
2     #ports{
3       in int din, tin;
4       in int1 reset,tinReset;
5       in int1 clk,clk_10f;
6       out int dout;
7     }@[v]
8     #var{
9       int [4*5] SR ;
10      int[4] Nout;
11    }@[v]
12    $_ clk {
13      SR[0] = din #;
14      SR[1:4]=SR[0:3] #;
15      SR[5] = Nout[0] #;
16      SR[6:9]=SR[5:8] #;
17      SR[10] = Nout[1] #;
18      SR[11:14]=SR[10:13] #;
19      SR[15] = Nout[2] #;
20      SR[16:19]=SR[15:18] #;
21      dout=Nout[3] #;
22    }@[v]
23    $libunit{
24       Nout[0:3]=NodeRW(SR[4:5:19],tin,reset,clk_10f,clk);
25    }@[v]
```

Figure 3.27: UADL code for one column design.

block used in Figure 3.27 is to instantiate an existing VHDL component "NodeRW"

which implements the neuron (node) functionalities. Since UADL supports the index expansion, it is very convenient to specify such a regular structure, e.g. it only takes UADL one line to implement all four "NodeRW" instantiations, which would be quite verbose for VHDL. The "#" symbols at the end of lines 13 to 21 indicate clocked signal transfers. The generated VHDL code for one column is shown in Figure 3.28.

```
1    ...
2    entity oneCol is port (
3      reset :in STD_LOGIC;
4      ...
5      dout :out STD_LOGIC_VECTOR (7 downto 0)
6    );
7    end oneCol;
8    architecture oneCol_arch of oneCol is
9    signal
10     Nout_0,Nout_1,Nout_2,Nout_3: STD_LOGIC_VECTOR (7 downto 0);
11     ...
12   begin
13     ...
14     if clk'event and clk  ='1' then
15           SR_1<=SR_0;
16       ...
17     end if;
18     ...
19   end process;
20   NodeRW_Nout_0: NodeRW port map (Nout_0,SR_4,tin,reset,clk_10f,clk);
21   ...
22   NodeRW_Nout_3: NodeRW port map (Nout_1,SR_9,tin,reset,clk_10f,clk);
23   end oneCol_arch;
```

Figure 3.28: Generated VHDL code for one column design (partial).

Due to the space limit, only part of the VHDL code is shown. The 25-line UADL has generated a 121-line VHDL code, about four times savings in coding. For a bigger column, even more savings would be expected.

The second area of UADL application is in testbench creation. When simulating the node operation, it is desirable to check the correctness of the node operation under all possible input values. For example, does the output of the node match

the theoretical results for all the 256 input values? To manually create the testing stimulus will be a painful, error-prone process, although most of the EDA tool vendors provide certain commands and scripts for users to specify simulation stimulus. However, there are two drawbacks using those provided commands and scripts:

1. These commands and scripts are not capable to generate complicated test patterns and usually they only provide weak support for mathematical functions.
2. The format of the commands and scripts are incompatible between different vendors, which causes problems for designs that encompass different EDA tools.

To overcome these disadvantages of using the vendor dependent commands and scripts, a new testbench generator is created with UADL framework. This generator allows two types of signals: periodic and non-periodic. Figure 3.29 displays the UADL code used to generate the VHDL testbench code for the functional verification of node operations.

This 25-line testbench code is composed of two clock signals, one reset signal, and two input data. The generated VHDL code has about 7,800 lines. The corresponding processes that generate clock, reset and input data are illustrated in Figures 3.30, 3.31 and 3.32.

The developed testbench generator supports most of the common mathematical functions and with the support from other UADL engines, the same testbench code can be used to generate testbench codes in other forms such as Verilog, C and Matlab code with very little or no modifications. This feature will greatly reduce the unnecessary work for design migration from tool A to tool B and it is very convenient for a complicated design that involves several different development tools from different vendors.

At last, a comparison of the code sizes between the UADL code and the generated VHDL code is given in Table 3.4. Note that the huge difference in code sizes for the testbench is simply because a regular repetition of the signal patterns, while for algorithmic part of the pipeline design, UADL achieves about 4 times savings.

```
1    target = v @VHDLTBWalker ~vhd
2    #timeUnit {ns;}
3    $period clk(1)=0 for 20 {
4      1 for 5;
5      0 for 5;
6    }
7    $period clk_10f(1)=0 for 20 {
8      1 for 50;
9      0 for 50;
10   }
11   $nonPeriod reset{
12     1 for 20;
13     0 for 100000;
14   }
15   $nonPeriod   din(8), tin(8){
16     0, 0 for 20;
17     $for lp=1:255{
18           $for x=1:5 {
19                   lp, x for 100;
20           }
21           $for y=1:5 {
22                   0, y for 100;
23           }
24     }
25    }
```

Figure 3.29: Sample UADL testbench code.

code size (byte)	UADL	VHDL
one column of pipeline	463	1,781
testbench	370	16,471

Table 3.4: Code size comparison for pipeline and testbench design.

```
1     process
2             variable bflag:boolean :=true;
3     begin
4     if bflag then
5             clk  <='0';
6             wait for 20 ns ;
7             bflag:=false;
8     else
9             clk <='0';
10      wait for 500ns ;
11      clk <='1';
12      wait for 500ns ;
13    end if;
14    end process;
```

Figure 3.30: Generated VHDL code from UADL testbench code (clock).

```
1     process
2     begin
3       reset<='1';
4       wait for 20ns ;
5       reset<='0';
6       wait for 1000000ns ;
7       wait;
8     end process;
```

Figure 3.31: Generated VHDL code from UADL testbench code (reset).

```
1     process
2     begin
3       din <="00000000";
4       tin <="00000000";
5       wait for 20ns ;
6       din <="00000001";
7       tin <="00000001";
8       wait for  100ns;
9       ...
10    end process;
```

Figure 3.32: Generated VHDL code from UADL testbench code (din,tin).

Chapter 4

DRAW System Design

4.1 Introduction

The rapid growth of wireless communication had surprised almost every one including many experts. Here in a small town in the Midwest, you can see most students talking on the phone while walking down the street, a phenomenon that you could rarely observe about two or three years ago. The explosive market and evolving technology have created several wireless standards and groups, each claiming its own advantage over others. Table 4.1 lists the major standards for 2G, 2.5G and 3G networks.

Generation	ETSI, GSM Ass.	UWC	TIA, CDG	CATT, Siemens
2G	GSM	IS-136	IS-95	
2.5G	GPRS/EDGE	GPRS/EDGE	1xRTT	
3G	WCDMA	UWC-136	CDMA2000	TD-SCDMA

Table 4.1: Wireless standards for 2G and 3G network.

Behind this fight for 3G standards is really a competition for a huge, lucrative market. This situation has presented additional difficulty to wireless designers besides the regular design challenges i.e. low power consumption, low cost, support for value-added services (internet, multimedia), reliability, etc. Which standards to choose is the primary question for a wireless designer before any further considerations.

Incompatible standards, complicated error correction coding algorithms and higher data link rates have driven the complexity of wireless systems to outpace the available silicon implementation [Rab99].

To address this problem, a new computing platform is needed to increase the efficiency of silicon usage. Reconfigurable computing has attracted significant attention for its flexibility and on-line dynamic reconfigurability. In the domain of reconfigurable computing, the same silicon area can be dynamically redefined to perform different functions without an obvious deterioration of performance.

As J. Rabaey pointed out, by confining the target application within certain domains like wireless communication, a better use of the silicon area can be achieved with comparable performance to ASIC or DSP chips.

Based on the previous work on dynamically reconfigurable architecture called DReAM [ASBG00], a new version of architecture called Dynamic Reconfigurable Architecture for Wireless communication (DRAW) was developed. It is aimed to provide a promising SoC solution for the third generation wireless mobile stations. DRAW has been used as a prototyping platform for the development of algorithms for wireless signal processing.

In the following sections, I will give a brief introduction to the structure of DRAW and focus on the arithmetic function units and UADL design flow application. For the details of other function units overall DRAW structure, please see [Als02].

4.2 DRAW Structure

According to [Als02], the DRAW platform mainly consists of the following components: Dynamically Reconfigurable Processing Unit (DRPU), Configuration Management Unit (CMU) Switching Box (SWB), and Communication Switching Unit (CSU), etc. DRPU is the basic computing unit of DRAW, and (Dynamic Reconfigurable Arithmetic Processor) DRAP is the computing engine behind DRPU, which performs all the the arithmetic operations required in wireless communication. In the following paragraphs, I will briefly introduce the DRAP unit and apply UADL tool to the Barrel Shifter design as a part of the DRAP unit.

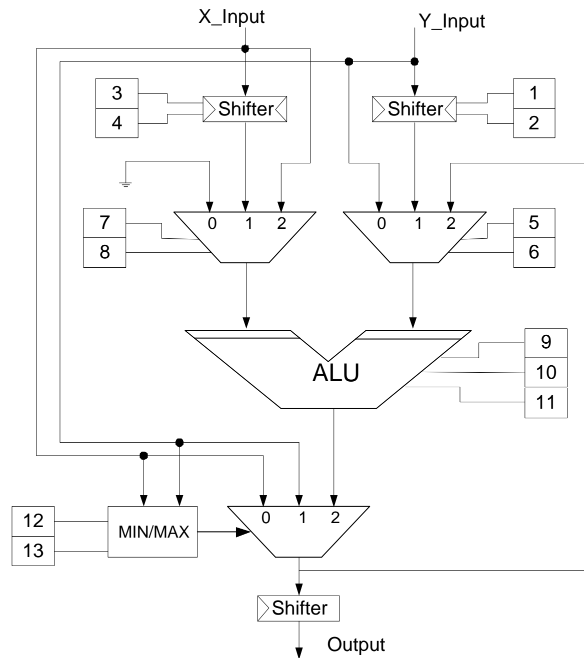


Figure 4.1: A block diagram of DRAP [Als02].

The detailed structure of the DRAP unit is illustrated in Figure 4.1. The DRAP unit is designed to support 13 basic arithmetic operations such as addition, subtraction, multiplication, shifting, etc. The major challenge of designing this unit is to make DRAP an efficient computing engine capable of fast numeric processing and to keep it simple at the same time. As pointed out on page 111 of [Als02], the most frequently used operations in wireless signal processing are addition and subtraction, while multiplication is far less frequent. Therefore, the speed of multiplication is not the bottleneck and it is implemented in the form of a sequential multiplier with a booth decoder [Boo51; Hwa79]. The configuration bits and the generic ALU work together to fulfill all the 13 numeric operations as described in [Als02].

There are three shifters in the DRAP unit, which are dedicated to scaling two inputs and one output. Scaling is important for computer arithmetics due to the limited word width. In DRAP design it is mainly used to prevent overflow/underflow during the signal processing. The Barrel Shifter is chosen as the shifter of DRAP unit for its fast processing [HJ04]. In the following section, I will use the implementation of

the Barrel Shifter design as a demonstration of how the UADL tool can help accelerate the design process and reduce the redundant works for developers.

4.3 UADL Contribution

The Barrel Shifter is a hardware device that can shift an incoming data word by any number of bits in a single operation. There are two major architectures of the Barrel Shifter design, one is the crossbar Barrel Shifter which uses crossbar structure to realize any number of shifts in one operation cycle. Another one is the logarithmic Barrel Shifter which involves a series of bypass/shift units; each can either shift data or not. Figure 4.2 and Figure 4.3 illustrate these two Barrel Shifter structures respectively. The crossbar design can achieve a maximum operating speed, but as we can see from Figure 4.2, this design requires N^2 control switches for data width N. The logarithmic Barrel Shifter has less demand for hardware resources at the cost of increased delay time. As observed from Figure 4.3, each stage of the bypass/shift unit is connected to one bit of the input shift count number. Suppose that a bypass/shift unit is connected to the kth bit of the shift count number. Depending on the bit value on kth position, this bypass/shift unit will either shift data by 2^k bits or just directly produce output from input bypassing the shift operation.

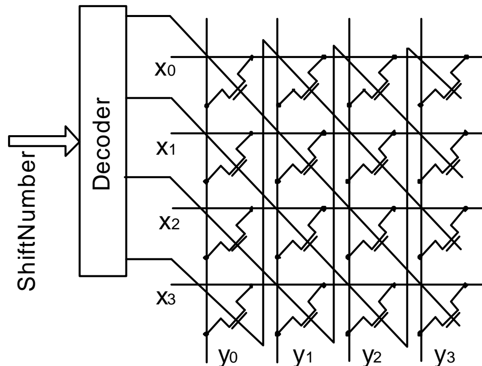


Figure 4.2: Crossbar implementation for 4-bit Barrel Shifter.

In case of DRAW architecture, the logarithmic Barrel Shifter is chosen over the crossbar for two reasons. The first reason is that the data path within the DRAW

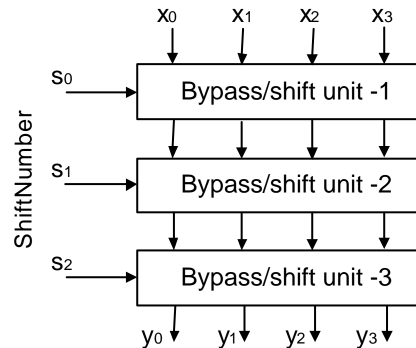


Figure 4.3: 3 Stage logarithmic implementation for 7-bit Barrel Shifter.

structures is 16 bit wide and it is sufficient to use only 7 bit shift for the purpose of scaling. The second reason is that the hardware resource is of more concern to make a compact DRAP unit. Therefore a 3-stage logarithmic Barrel Shifter is needed.

The design flow for this Barrel Shifter implementation using the UADL tool is straightforward. Firstly, a one stage bypass/shift unit is written in the developed UADL tools, then two UADL engines are applied to this same UADL code to generate different target codes – VHDL code and Matlab code. Both of these codes are executable and yield simulation results that can be verified against each other. After the one-stage design passes the verification in both VHDL and Matlab, a 3-stage bypass/shift unit is developed in UADL and a similar procedure follows. The design flow is depicted in Figure 4.4. On the top of this figure is the UADL code used to generate two different target codes. The same piece of UADL code **barrelshift.tgr** is fed to the UADL VHDL engine and the UADL Matlab engine respectively, except for a few minor changes in the first target line as shown in the gray boxes in Figure 4.4. Correspondingly, the target files **barrelshift.vhd** and **barrelshift.m** are created automatically by its UADL engine.

The simulation results of the generated VHDL and Matlab codes are displayed in Figure 4.5 and Figure 4.6 and have been verified to have same results for all four combinations of *dir* and *arithLogic* values which are used respectively to specify the shift direction and the type of the shift operation.

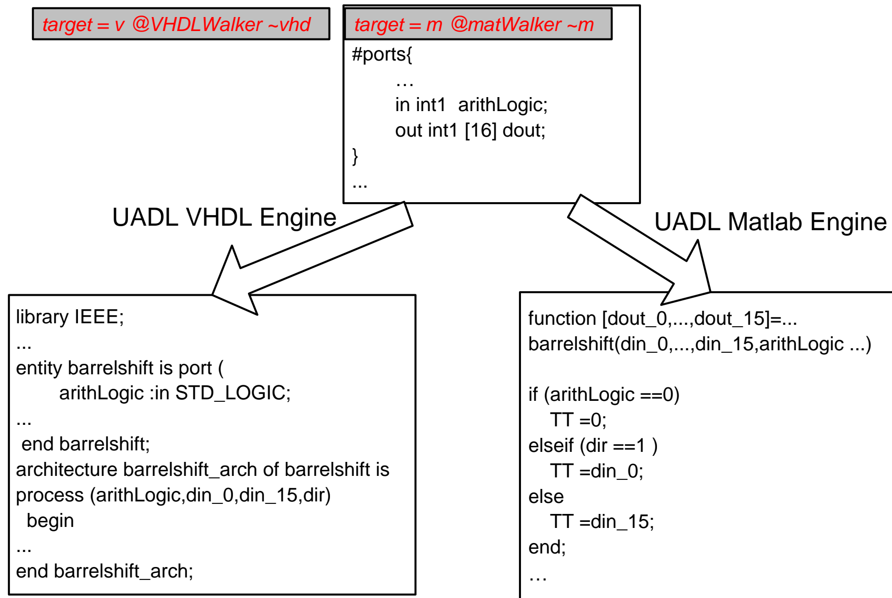


Figure 4.4: UADL design flow with VHDL Matlab code auto-generation.

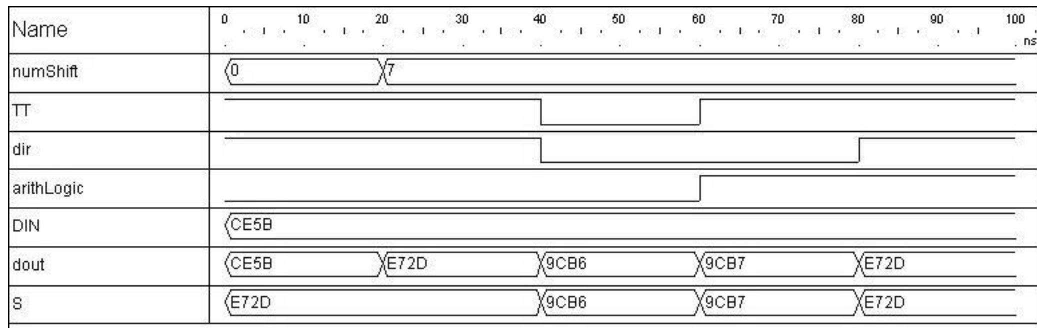


Figure 4.5: VHDL simulation results for one stage bypass/shift unit.

Table 4.2 shows the results in the binary format for a better illustration of the two different types of shifting it implements.

In a similar way, the final 3-stage Barrel Shifter is implemented and corresponding VHDL and Matlab codes are generated and simulated with the results displayed in Figure 4.7 and Figure 4.8.

The generated VHDL codes have also been successfully synthesized and mapped to Xilinx Virtex device as displayed in Figure 4.9

```
>>Matlab results:
dir=0, arithLogic=0, numShift_0=1
input data="1100111001011011"
output data="[1 0 0 1 1 1 0 0 1 0 1 1 0 1 1 0]"
dir=1, arithLogic=0, numShift_0=1
input data="1100111001011011"
output data="[1 1 1 0 0 1 1 1 0 0 1 0 1 1 0 1]"
dir=0, arithLogic=1, numShift_0=1
input data="1100111001011011"
output data="[1 0 0 1 1 1 0 0 1 0 1 1 0 1 1 1]"
dir=1, arithLogic=1, numShift_0=1
input data="1100111001011011"
output data="[1 1 1 0 0 1 1 1 0 0 1 0 1 1 0 1]"
```

Figure 4.6: Matlab results for generated barrelshift.m code.

Name 0 10 20 30 40 50 60 70 80 90 100 ns
numShift 0 7
TT
dir
arithLogic
DIN CE5B
dout CE5B FF9C 2D80 2DE7 B79C
S E72D 9CB6 9CB7 E72D

Figure 4.7: VHDL simulation results for 3-stage bypass/shift unit.

Now let us briefly revisit Figure 2.4. As it can be seen that with the help of the individual UADL engine, the redundant work of migrating the Matlab design flow to the VHDL code design flow (or vice versa) has been greatly simplified to only one line change. To further demonstrate the savings introduced by the UADL flow, a comparison of code sizes are given in Table 4.4. It lists the size of the original UADL

(dir, arithLogic)	Hexadecimal	Binary
0,0	9CB6	1001110010110110
0,1	9CB7	1001110010110111
1,0	E72D	1110011100101101
1,1	E72D	1110011100101101
input data	CE5B	1100111001011011

Table 4.2: Output data for 1-stage Barrel Shifter.

```
>> Matlab results
dir=0, arithLogic=0, numShift=111
input data="1100111001011011"
output data="[0 0 1 0 1 1 0 1 1 0 0 0 0 0 0 0]"
dir=1, arithLogic=0, numShift=111
input data="1100111001011011"
output data="[1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 0]"
dir=0, arithLogic=1, numShift=111
input data="1100111001011011"
output data="[0 0 1 0 1 1 0 1 1 1 1 0 0 1 1 1]"
dir=1, arithLogic=1, numShift=111
input data="1100111001011011"
output data="[1 0 1 1 0 1 1 1 1 0 0 1 1 1 0 0]"
```

Figure 4.8: Matlab results for generated barrelshift3.m code.

code and generated VHDL and Matlab codes and shows a significant savings in the designer's programming efforts.

In the following section, I will discuss a Turbo decoder design as one of the many possible applications of DRAW.

(dir, arithLogic)	Hexadecimal	Binary
0,0	2D80	0010110110000000
0,1	2DE7	0010110111100111
1,0	FF9C	1111111110011100
1,1	B79C	1011011110011100
input data	CE5B	1100111001011011

Table 4.3: Output data for 3-stage Barrel Shifter.

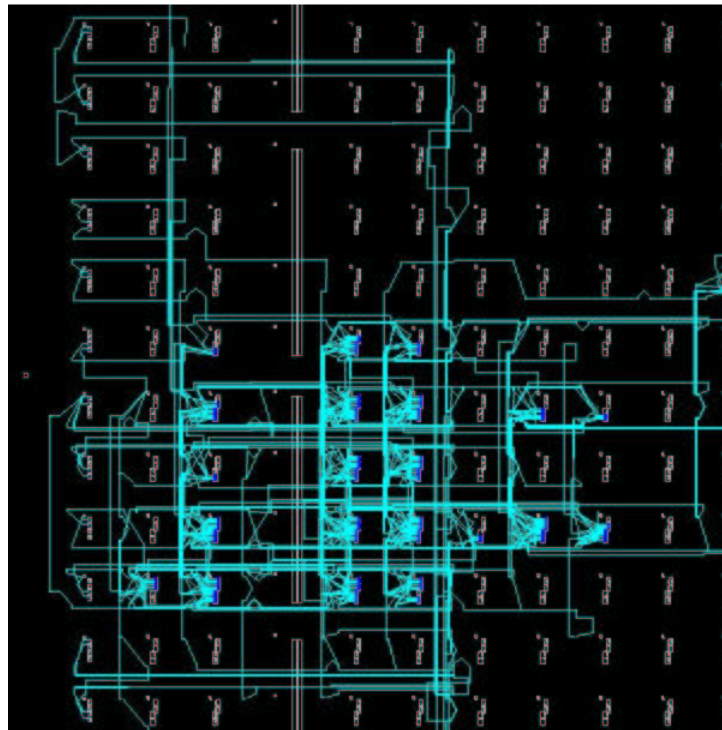


Figure 4.9: Layout map for 3-stage Barrel Shifter.

code size (byte)	UADL	Matlab	VHDL	average of Matlab and VHDL
barrelshift	597	1,252	2,325	1,788
barrelshift3	1,888	2,939	5,558	4,248

Table 4.4: Generated code size compared with original UADL code size.

4.4 Design Example – Turbo Decoder

4.4.1 Introduction

Since its introduction in 1993 [BGT93], Turbo code has attracted significant interest from research and industry for its superior error correction capability, which is close to the Shannon limit. The major 3G standard groups such as 3GPP [3GP] have already included Turbo code as part of future wireless communication systems. As a class of convolutional code, Turbo code is known to be extremely computing intensive in decoding procedure. This presents a great challenge to the design of 3G mobile stations where power consumption, circuit size and cost are of great concern. In the literature there are many implementation attempts [GG98; BC01; VMP+00; MPRZ99] aiming to address both power and adaptability of the Turbo decoder. The RC system offers a suitable implementation platform for such problems. By dynamically swapping different blocks of a Turbo decoder in time, a smaller area and significant power savings can be gained. This is an alternative to power saving techniques where unused parts of a system are powered down. Those powered down subsystems still occupy a valuable area which, in RC architecture, can be reused to run other applications.

4.4.2 Turbo Decoding Algorithm

There are two types of procedures to generate a concatenated Convolutional coding: Serial Concatenated Convolutional Coding (SCCC) and Parallel Concatenated Convolutional Coding (PCCC). The PCCC scheme is the one proposed by 3GPP. This type of Turbo codes can be decoded in an iterative manner [CGK00; GMG+01]. The encoding of the data bits is performed by using two Recursive Systematic Coders (RSC) concatenated in a parallel, as shown in Figure 4.10. This encoder is rate 1/3 encoder, which means that for every one input bit, three output bits are generated. The input to the second RSC2 is first interleaved using a random-like interleaving process. The encoder operates on a block of bits, which is from 40 to 5,114 bits long according to [3GP; MW01]. Note that puncturing is omitted here for the sake of simplicity.

A general Turbo decoder structure is displayed in Figure 4.11. The Turbo decoder consists of two Soft-Input Soft-Output (SISO) decoders. An SISO decoder is capable of computing the a posteriori likelihood of the constituent codes. The original Maximum a *posteriori* (MAP) algorithm is very expensive to implement, due to the required multiplication and exponential operations [WLW01; RHV97]. Therefore, the Max-Log-MAP algorithm [CGK00] is chosen as the algorithm of implementation, and a sliding window technique is used to reduce the storage requirements.

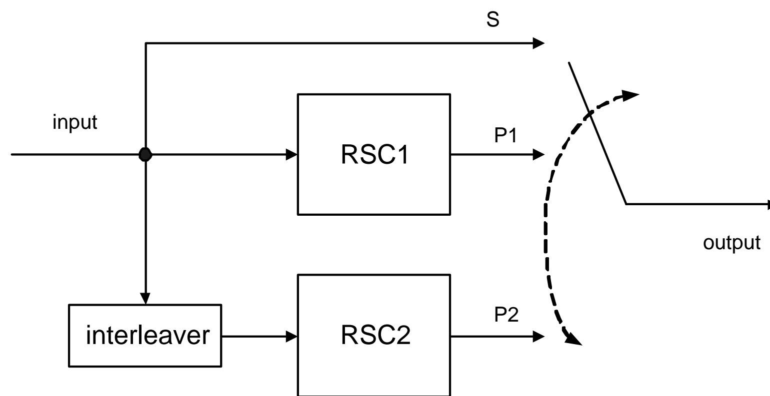


Figure 4.10: PCCC encoder proposed by 3GPP.

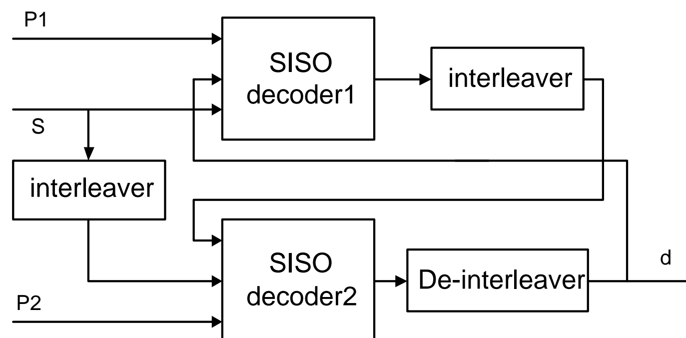


Figure 4.11: Turbo Decoder structure.

According to the BCJR algorithm [BCJR74], the calculation of a *posterior* probability p can be decomposed to a calculation of α, β and γ parameters, as follows:

$$p(s', s, y) = \alpha_{k-1}(s')\gamma_k(s', s)\beta_k(s) \tag{4.1}$$

where s and s' are the encoder states at time k and $k-1$ respectively, and

$$\alpha_k(s) = \sum \alpha_{k-1}(s')\gamma_k(s', s) \tag{4.2}$$

$$\beta_{k-1}(s) = \sum \beta_k(s')\gamma_k(s', s) \tag{4.3}$$

In Max-Log-MAP algorithm, calculation of α, β and γ is simplified to a batch of addition, subtraction and maximum operations. Figure 4.12 shows the datapath for the calculation of parameter for state m at time k. β has the same datapath as α.

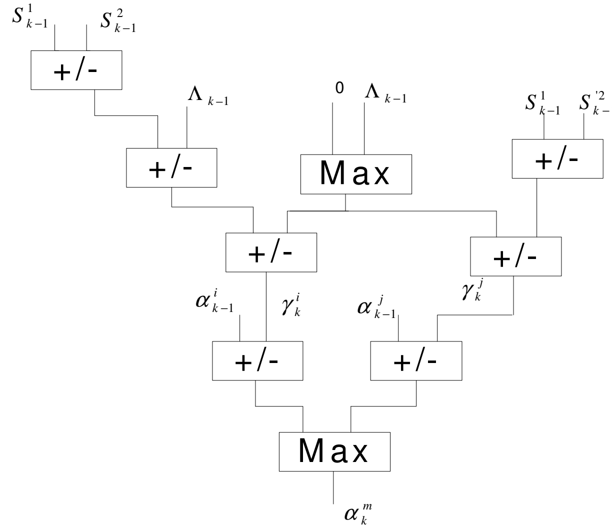


Figure 4.12: Datapath for α calculation.

Similarly, the Log-Likelihood Ratio (LLR) can be calculated as

$$LLR_k = \max_{s1}[\alpha_k(s) + \gamma_k(s, s') + \beta_{k+1}(s)] - \max_{s0}[\alpha_k(s) + \gamma_k(s, s') + \beta_{k+1}(s)] \tag{4.4}$$

where $s1$ stands for all the states related to information bit 1, while $s0$ stands for all the states related to information bit 0. Due to the space limit, Figure 4.13 shows the datapath of LLR computation for only 4 states.

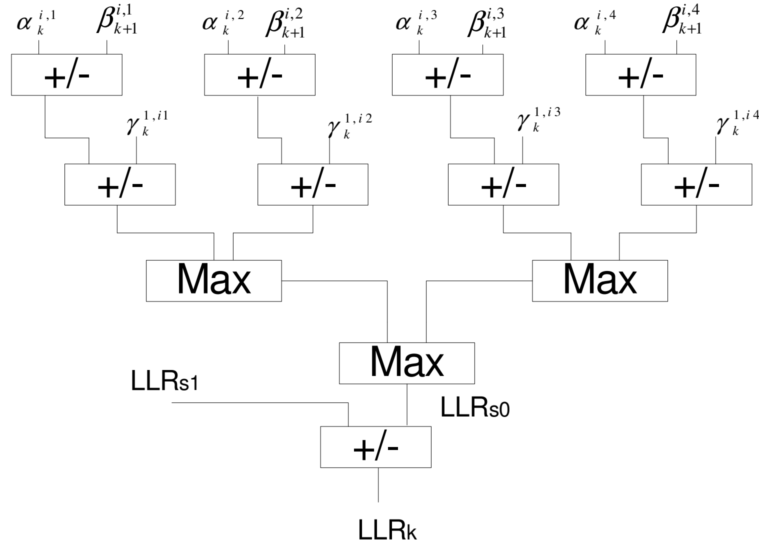


Figure 4.13: Datapath for LLR (4 states).

Sliding Window

To reduce the memory demand of the Turbo decoder, the sliding window technique is necessary (Figure 4.14). The whole data frame is divided into several sliding windows accompanied with a training window. α is computed in forward recursion as usual, while β is computed from the end of each training window instead of from end of frame backward to the head of each sliding window. In this way, the calculation of LLR can be performed as soon as β backward recursion is finished. Then move to the next sliding window performing the same operation. Thus output delay can be reduced proportional to the size of sliding window N_s instead of frame length L.

Considering the decoding delay for the SISO module, there will be a slight difference between the first iteration and subsequent iterations. For the first iteration, it will take about $k(2N_s + 2N_t)$ cycles to get the first block of output, where k is the number of clock cycles needed to process every information bit, N_s is the sliding window size, and N_t is the training window size. In the subsequent iterations the decoding delay can be reduced to $k(N_s + N_t)$ cycles, since data is already stored and ready. As illustrated in Figure 4.15, decoding delay per iteration(in clock cycles) is

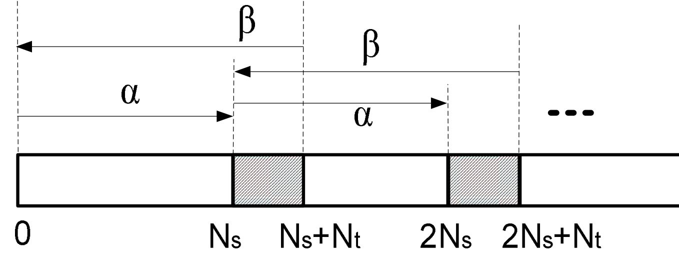


Figure 4.14: Sliding window scheme.

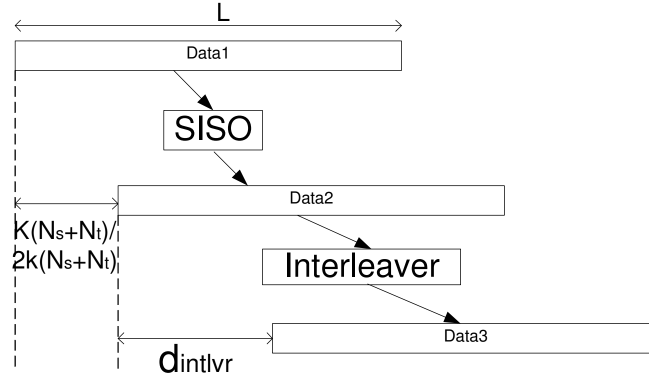


Figure 4.15: SISO decoding delay.

$$\text{Delay} = \begin{cases} k(2N_s + 2N_t) + d_{intlvr} & n = 1 \\ k(N_s + N_t) + d_{intlvr} & n > 1 \end{cases} \tag{4.5}$$

where n is the iteration number for the whole decoder, d_{intlvr} is the processing delay for the interleaver (de-interleaver) and is affected by the size of frame length L as shown in Figure 4.15. According to the 3GPP standard, the size of the permutation matrix is decided by L [3GP]. Assuming the matrix has R rows and C columns, d_{intlvr} can be expressed as $C(R-1)+1$ cycles.

4.5 DRAW Implementation Cost

The cost of implementing this Turbo decoder into DRAW is listed in the following paragraphs in terms of memory, arithmetic, and timing requirements.

Memory requirement for one MAP

The storage for α is $S \times N_S$ bytes where S is the number of encoder states and equals to 8 [3GP]. The storage for data is $3L$ bytes, where L is the frame length. In case of DRAW, one DRPU has 16 byte memories, thus it takes $(8N_s + 3L)/16$ DRPUs. The storage for the interleaver can be estimated as L bytes.

Arithmetic requirement for one MAP

Area for α (or β) computing is $(6{+}3) \times S = 72$ DRPUs and for LLR it is $3S - 1 = 23$ DRPUs, so the total area for one MAP module is 95 DRPUs. Consequently, the mapping of the SISO module to DRAW architecture will take an 5×19 array of DRPUs. As one can see from Figure 4.11, one input of the MAP module is the output of the (de)interleaver, thus the total required DRPUs for a Turbo decoder would not incur extra storage cost for the (de)interleaver and it will be $2 \max[(8N_s{+}3L)/16, 95]$.

Timing requirement for one MAP

As one can see from Figure 4.12 the calculation of α or β costs 5 clock cycles, and from Figure 4.13 it can be figured out that the calculation of LLR for 8 states, whenever α, β and γ are ready, takes 6 clock cycles. So the computation cost for α (or β) plus LLR is 16 cycles/iteration/bit (Computing α, β in parallel can achieve faster speed at the cost of extra area, which is a desirable option). Based on previous analysis, the processing delay for the decoder is

$$\begin{aligned} \delta =& 16(2N_s + 2N_t) + 2[C(R-1) + 1] \\ &+ (n-1)\{16(N_s + N_t) + [C(R-1) + 1]\} \\ =& (n+1)[16(N_s + N_t) + C(R-1) + 1] \text{ (cycles)} \ . \end{aligned} \tag{4.6}$$

Note that $C(R-1)+1$ is usually close to or comparable with L, have

$$\delta \approx (n+1)[16(N_s + N_t) + L] \ . \tag{4.7}$$

This gives a good estimate of δ, and can be a useful guide for N_s and N_t choice with respect to L. From the above discussion, please note that the choice of n, N_s

and N_t should consider all three factors in balancing the area and speed constraint. The throughput bottleneck will shift to the (de)interleaver as L increases, and data storage will have a larger effect on the processing delay.

Using the design parameters mentioned above, the total number of DRPUs required for the Turbo decoder application is $2\max[(8N_s+3L)/16, 95] = 280$. A 15×19 array of DRPUs including other auxiliary units (like configuration unit, switch box, etc.) is required. From Eq. 4.7, it can be found that for a frame length of 512 bits, the required clock rate is about 2.5MHz for 10 iterations. According to 3GPP standard, maximum frame length is 5,114 bits; it is estimated that the required clock rate will be 7.4MHz for 10 iterations, which is well within DRAW's capability.

4.6 UADL Contributions

The UADL tool has been applied to implement the Turbo decoder. Specifically, the LLR computing part and α (or β) computing part are implemented in UADL. Since the basic computing unit in DRAW is DRPU, the LLR computing task is accomplished in form of a network of DRPUs with different configurations. To simplify the interface of DRPU, a "DRPUwrapper" VHDL module, and a Matlab function are created to expose only the interface signals that are related to LLR computing. The UADL code is listed in Figure 4.16.

The generated VHDL and Matlab codes for the LLR part are listed in Figure 4.17 and Figure 4.18 respectively. The VHDL simulation for the LLR part is displayed in Figure 4.19 and it is verified with Matlab results.

Similarly, the α (or β) computing part is designed in UADL which generated VHDL and Matlab codes. A comparison of code sizes is given in the following table. It can be seen that there is about 2 times savings between UADL and the averaged VHDL and Matlab code sizes.

```
1    target = v @visitor.matWalker ~m
2    #ports{
3     in int[4] alpha, beta, gamma;
4     in int3 cfgAdd, cfgSub, cfgMax;
5     out int LLRout;
6    }@[v]
7    #var{
8     int [4] sumAlphaBeta ;
9     int [4] sumABG;
10    int [2] maxL1;
11   }@[v]
12   $libunit {
13    sumAlphaBeta[0:3]=DRPUwrapper(alpha[0:3],beta[0:3],clk,cfgAdd);
14    sumABG[0:3]=DRPUwrapper(sumAlphaBeta[0:3],gamma[0:3],clk,cfgAdd);
15    maxL1[0:1]=DRPUwrapper(sumABG[0:2:2],sumABG[1:2:3],clk,cfgMax);
16    LLRout=DRPUwrapper(maxL1[0],maxL1[1],clk,cfgMax);
17   }@[v]
```

Figure 4.16: UADL code for LLR.

```
1    ...
2    entity LLR is port
3    ( gamma_0,
4      gamma_1,
5      ...
6      LLRout :out STD_LOGIC_VECTOR (7 downto 0)
7    );
8    end LLR;
9    architecture LLR_arch of LLR is
10   signal
11     maxL1_0,
12     ...
13   begin
14   DRPUwrapper_sumAlphaBeta_0: DRPUwrapper port map (
15     sumAlphaBeta_0, alpha_0, beta_0, clk, cfgAdd);
16     ...
17   end LLR_arch;
```

Figure 4.17: Generated VHDL code for LLR (partial).

```
1   function [LLRout]=...
2   LLR(gamma_0,gamma_1,gamma_2,gamma_3,cfgAdd,cfgMax,...
3      beta_0,beta_1,beta_2,beta_3,alpha_0,...
4      alpha_1,alpha_2,alpha_3,clk,cfgSub)
5
6   [sumAlphaBeta_0] = DRPU(alpha_0,beta_0,clk ,cfgAdd);
7   [sumAlphaBeta_1] = DRPU(alpha_1,beta_1,clk ,cfgAdd);
8   [sumAlphaBeta_2] = DRPU(alpha_2,beta_2,clk ,cfgAdd);
9   [sumAlphaBeta_3] = DRPU(alpha_3,beta_3,clk ,cfgAdd);
10  [sumABG_0] = DRPU(sumAlphaBeta_0,gamma_0,clk ,cfgAdd);
11  [sumABG_1] = DRPU(sumAlphaBeta_1,gamma_1,clk ,cfgAdd);
12  [sumABG_2] = DRPU(sumAlphaBeta_2,gamma_2,clk ,cfgAdd);
13  [sumABG_3] = DRPU(sumAlphaBeta_3,gamma_3,clk ,cfgAdd);
14  [maxL1_0] = DRPU(sumABG_0,sumABG_1,clk ,cfgMax);
15  [maxL1_1] = DRPU(sumABG_2,sumABG_3,clk ,cfgMax);
16  [LLRout ] = DRPU(maxL1_0,maxL1_1,clk ,cfgMax);
```

Figure 4.18: Generated Matlab code for LLR.

code size (byte)	UADL	Matlab	VHDL	average of Matlab and VHDL
LLR	505	736	1,865	1,300
$\alpha(\beta)$	718	631	1,783	1,207

Table 4.5: Code size comparison for α (or β) design.

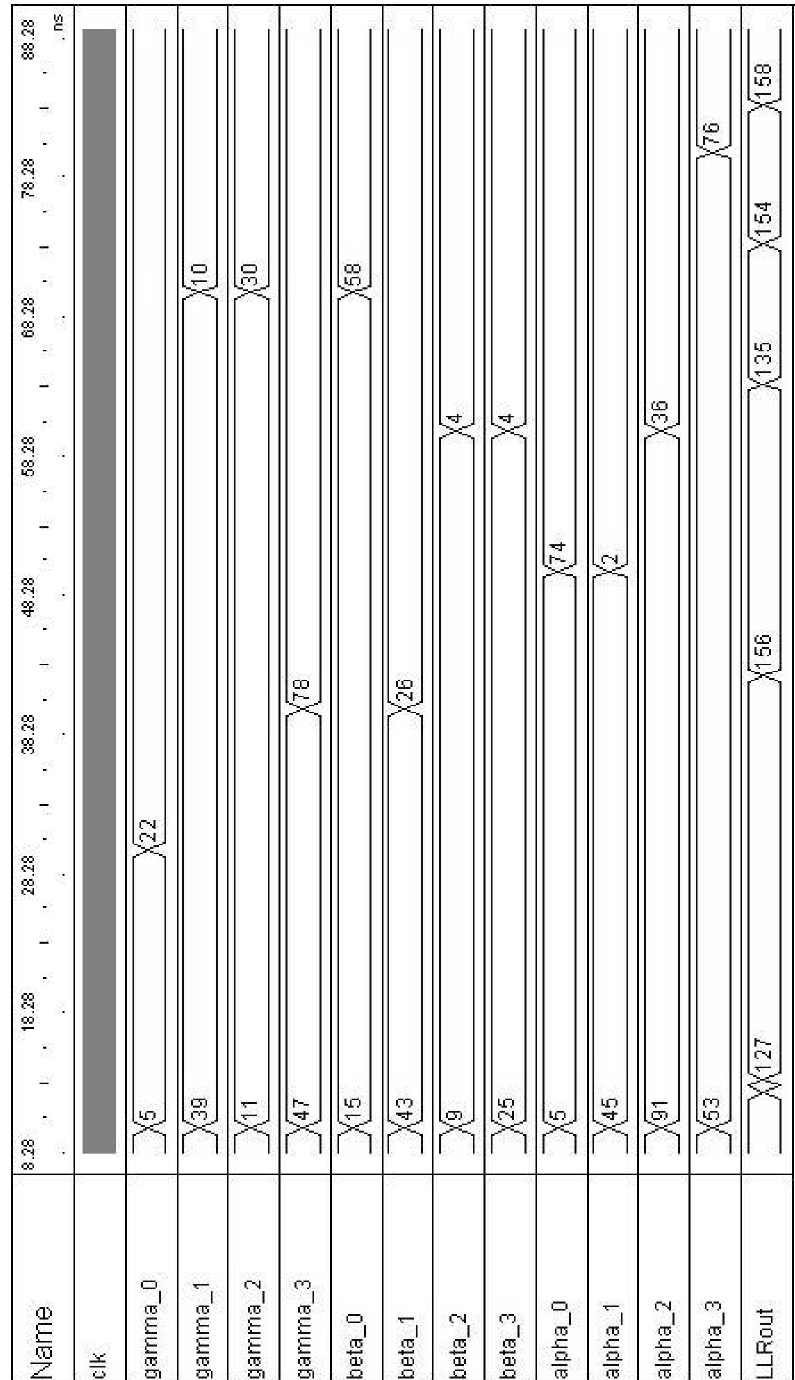


Figure 4.19: Simulation results for LLR computing.

Chapter 5

Dimensionality Reduction

5.1 Preprocessing

In the targeting fields of SOLAR system such as pattern classification and data clustering, many data of real world applications such as image and speech data usually have very high dimensionality: a set of 60×80 pixel images can easily reach a very high dimensionality if each image is treated as a single data point in the 4800 dimensional space. Depending on the nature of the classification problem that needs to be solved, there might be only a handful of factors affecting the classification. For instance, a set of face images could be classified into several expressions like smile, sad, cry, anger, etc, or different positions such as face turning left, right, up, etc. A few major features will be sufficient to do this kind of job with decent accuracy. To have SOLAR directly process those high dimensional data, both hardware implementation cost and computation cost will be prohibitively high. Dimensionality reduction thus plays a key role to lower the computing burden of the down-stream unit, yet retains high performance of the whole system. From a system level point of view, a preprocessing unit is necessary for a data processing unit like SOLAR to deal with a broad spectrum of real-world data as shown in Figure 5.1.

In the following sections, the dimensionality reduction algorithms will be discussed as the preprocessing for the SOLAR computing system and in next chapter discus-

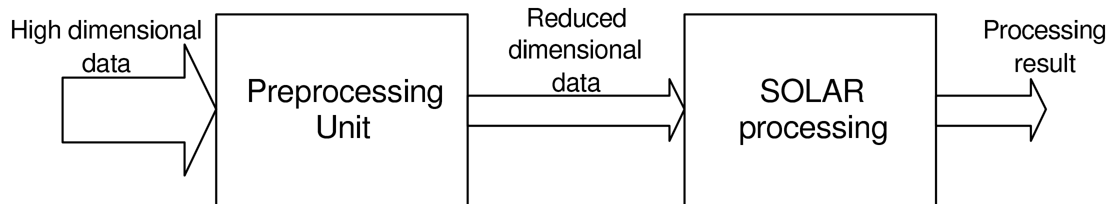


Figure 5.1: Preprocessing unit and SOLAR system.

sion and analysis will be given on the input selection and weighting schemes and reconfigurable routing problems with the related hardware implementation.

5.2 Dimensionality Reduction and Postmapping

One of the critical problems in machine learning and pattern recognition is the development of appropriate representation of complex real-world data. Quite often these data are intrinsically low dimensional, but are lying in a very high dimensional space. Finding the intrinsic low dimensionality in such data is called the manifold embedding problem. Classical linear dimensionality reduction methods like Principal Component Analysis (PCA) and Multi-Dimensional Scaling (MDS) cannot deal very well with this problem when data is distributed in a high dimensional space over a nonlinear low dimensional manifold. [dST03]. Recently, several non-linear dimensionality reduction (NLDR) algorithms such as ISOMAP by [TdSL00], Laplacian Eigenmap (LE) by [BN03a], kernel based principal component analysis (kernel PCA) by [SSM98], and Locally Linear Embedding (LLE) by [RS00] have been developed to address this problem. Based on the kernel method presented by [SSM98], an unsupervised learning algorithm is proposed by [WSS04], which can find the optimum input space transformation kernel via semi-definite programming (SDP) optimizing procedure [VB96]. Due to recent substantial progress in the field of SDP, efficient algorithms have been devised to perform optimization over convex cones [OW97], and these algorithms facilitate finding the optimum transformation kernel for multidimensional data with thousands of data points. The learned kernel has shown superior performance over polynomial or Gaussian kernels used in other dimensionality reduc-

tion techniques [WSS04]. LLE is another unsupervised algorithm that attracts recent attention. Based on a simple geometric intuition, it exploits the local neighborhood to discover the global manifold (manifold—a topological space that is locally Euclidean) structure. Compared with its peers, LLE has less free parameters, does not involve local minima, and is numerically more efficient than kernel optimization [RS00].

Current major NLDR methods, including LE, LLE and ISOMAP algorithms, base their computation on local neighborhood structure, and use this structure to globally map the manifold to a lower dimensional space [HLMS04]. As described by [VRV04], these NLDR methods do not provide useful mapping to lower dimensional space of new test data, which is needed in many application areas. Therefore, when a new test data sample comes, the whole computing procedure needs to be repeated at a high computing cost. For instance, NLDR procedure complexity is at the order of $O(nD^2)$ for LLE [SR03], $O(D^2 logD)$ for ISOMAP and LE algorithm [VVK02], where D is the cardinality of the data set and n is the reduced dimensionality. In applications such as facial recognition, the system is first trained over a set of face images, then based on the training result, the system should be able to handle a new face image to perform tasks such as identification or classification by the expression, light angle or pose. It would be uneconomic to recalculate the whole data set whenever a new face image needs to be positioned in reduced dimension within the same training set. Therefore, an extension method that allows new test data to be directly mapped to the reduced dimension space is desirable.

Two possible directions: the non-parametric model and the parametric model are suggested by [SR03] for the LLE extension. In the non-parametric model, authors suggest using the same underlying intuition as LLE for the mapping of new data; in the parametric model, a probabilistic distribution model is to be learned and used for mapping of new data. In the paper by [BPV+04], a unified framework is proposed to extend the dimensionality reduction mapping to test (out-of-sample) data points. This framework has been applied to five different algorithms, with slight modification from one algorithm to another and has shown a comparable error level as caused by training set stability.

In this chapter a different approach to this problem is proposed. The two proposed algorithms provide a simple solution to the extension problem for many possible NLDR algorithms by treating the mapping procedure as a black box. In other words, the proposed method only needs to know the original data set and the after-reduction data set. Therefore, it can be applied to not only existing NLDR algorithms, but also to any future algorithms as long as they preserve local isometry. This black-box property can help to simplify the test data extension procedure, especially when the original training algorithm is not available at the test phase. Since this extension procedure is applied after the dimensionality reduction procedure that maps the original set of points, it is called "postmapping". It uses the training data in the original high dimensional space and reduced dimensional space created by NLDR algorithms to map new test data into the lower dimensional space.

5.3 Principles of Postmapping

Nonlinear dimensionality reduction algorithms were developed to overcome the inability of linear methods to adequately represent dominant features of observed data in high dimensional feature space. Popular NLDR methods like ISOMAP, LLE and LE were related to kernel PCA algorithms [HLMS04]. Although methods like LLE and ISOMAP have different global properties, they preserve local distances between neighborhood points with sufficiently dense sampling of that neighborhood area. Local isometry preservation is also an important constraint in kernel matrix learned through semi-definite programming [WSS04]. The proposed postmap methods are based on the assumption that the local isometry is preserved by NLDR procedure and that data samples come from smooth manifolds.

To better illustrate the proposed postmapping method, first some terminology will be introduced. Let us denote the original space of N dimensions by $\Re^N$, the original space after dimensionality reduction will be denoted by $\Re^n$, (where $N > n$), the original data set is denoted by D, and the same data after dimensionality reduction will be denoted by R. Then consider a new data point $d \notin D$. The postmapping method needs to find a corresponding point $r \in R$, that maintains the relationship

between d and D. In particular, it is expected that r will maintain a topological structure of data and consequently the local isometry around d will be well preserved.

To obtain such point r from the given data d, first find k nearest neighbors of d in D, and call them D_{nbr}. It is easy to find the corresponding neighborhood set $R_{nbr} \subset R$. The target point r will be constructed based on d, D_{nbr} and R_{nbr}. For convenience I will denote by the same symbols D_{nbr} and R_{nbr} as $k \times N$ and $k \times n$ matrices that contain all k neighborhood points as rows and points coordinates as columns. Two methods were developed to make this kind of postmapping.

5.4 Linear Postmapping

The most natural approach for postmapping is a method that preserves local distances between neighbors in the original and mapped spaces. Here I will show how such a method can be simply implemented by solving linear equations, therefore this method is called a linear postmapping. This postmapping method constructs r from R_{nbr} using distances between d and D_{nbr}.

Generally, it needs $k \geq n+1$ to have sufficient number of quadratic equations to define r in n dimensional space using a linear method. First, the case for $k = n+1$ is considered, and the cases for $k > n+1$ will be discussed later. The following $n = k-1$ linear equations can be used to obtain coordinates of a point r in the reduced dimensional space that preserves local isometry.

$$\begin{pmatrix} R_1 - R_2 \\ R_2 - R_3 \\ \vdots \\ R_{k-1} - R_k \end{pmatrix} r^T = -\frac{1}{2} \begin{pmatrix} \gamma_1^2 - \|R_1\|^2 - (\gamma_2^2 - \|R_2\|^2) \\ \gamma_2^2 - \|R_2\|^2 - (\gamma_3^2 - \|R_3\|^2) \\ \vdots \\ \gamma_{k-1}^2 - \|R_{k-1}\|^2 - (\gamma_k^2 - \|R_k\|^2) \end{pmatrix} \tag{5.1}$$

where R_i $(i = 1, 2, \ldots, k)$ is the ith point in R_{nbr} and $\gamma_i = ||d - D_i||$, where D_i $(i = 1, 2, \ldots, n)$ is the ith point in D_{nbr}. When local distances in R_{nbr} are preserved by NLDR algorithm, then Eq. 5.1 has unique solution r. In the the case when the distances between the neighbors in R_{nbr} were disturbed by the nonlinear dimensionality reduction technique, the obtained solution r may not preserve this local isometry. In

such case, local distances are only approximately maintained after mapping into lower dimensionality space and the postmapping error is a function of both the disturbance introduced by NLDR procedure and the number of neighborhood points $k \geq n$ used in Eq. 5.1.

This sensitivity of the linear postmapping is verified in experiments with Swiss Roll data (available at `http://isomap.stanford.edu`) as illustrated in Figure 5.2. Two test points are presented in that figure. The first one marked with diamond shape is outside of the distribution that generates the Swiss Roll data, the second one marked with square shape is part of this distribution. After mapping the Swiss Roll data using Kernel PCA powered by SeDuMi tool [Stu99], a flat 2D manifold is obtained as shown on the right side of Figure 5.2. Using linear postmapping I can generate mapped images of the test points and both of them are mapped outside of the 2D manifold that represents the Swiss Roll distribution. This indicates that the linear procedure is very sensitive to the distortion from local isometry introduced by nonlinear dimensionality reduction.

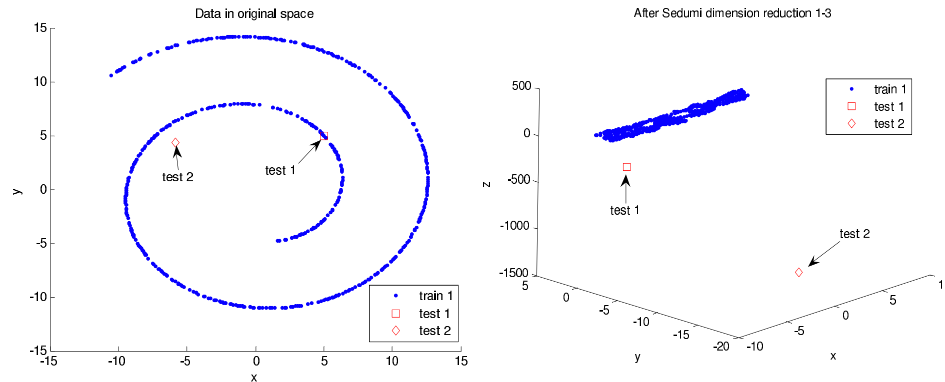


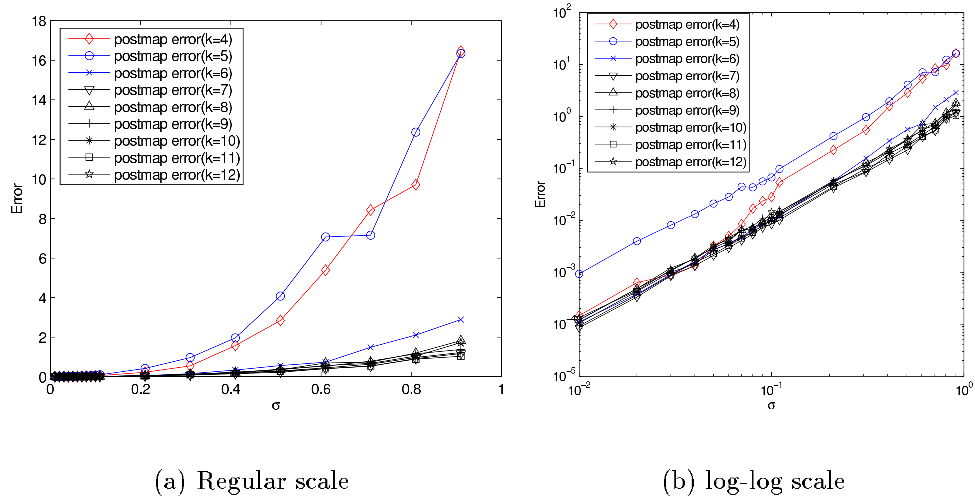
Figure 5.2: Linear method reconstruction example

Over-determined case

A systematic analysis of the linear postmapping in the over-determined case ($k > n + 1$) is conducted in the following experiment in the 3D space. The test cases were randomly generated for this experiment and do not represent the result of a specific nonlinear dimensionality reduced mapping of a database. However, they do emulate

the expected variance of local distances induced by those dimensionality reduction algorithms. The purpose of this experiment is to see how the disturbance of local mapped points will affect the accuracy of linear postmapping.

Choosing the size of neighborhood set $k > 4$ leads to an over-determined equation Eq. 5.1. This over-determined case will be studied to learn about the statistical stability of the linear postmapping method. First k points are randomly picked as the neighbors of a selected test point, distances are calculated from this test point to its neighbors, and then these distances are disturbed by a zero mean Gaussian noise with a standard deviation σ. Then the linear postmapping method is applied to reconstruct the position of the test point based on the disturbed distances. The error between the reconstructed test point and the true test point is plotted for different σ and k, as illustrated in Figure 5.3, where each data point in the curve is the mean value of 100 experiments for statistical stability.



(a) Regular scale (b) log-log scale

Figure 5.3: Linear postmapping error relation with k

It can be seen that the error of linear postmapping increases proportionally against the logarithm of the strength of disturbing noise over several orders of magnitude. The number of neighbors k moderately larger than reduced space dimensionality helps to lower the linear postmapping error.

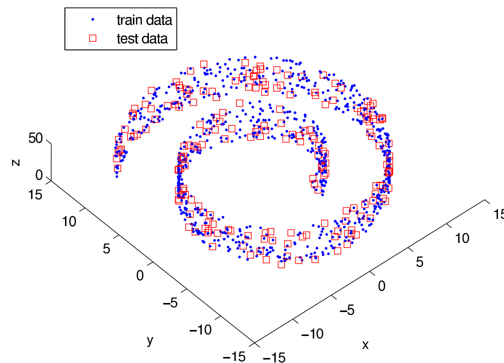
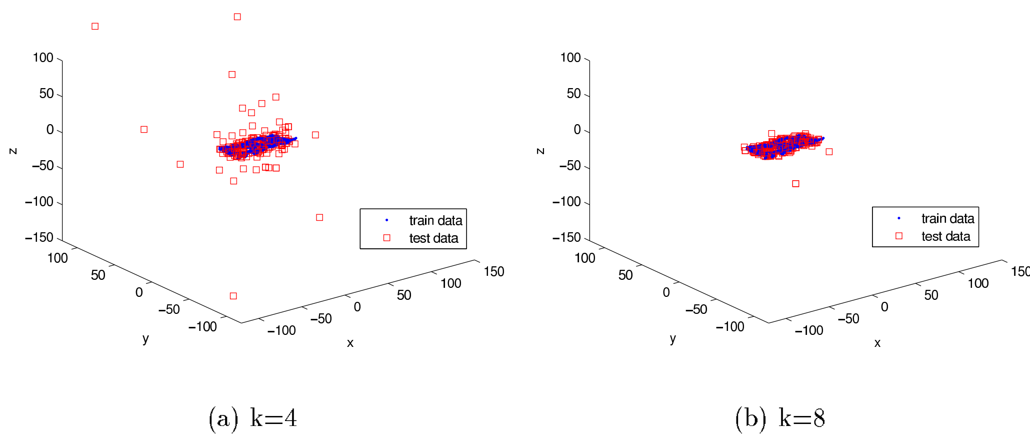


Figure 5.4: Swiss Roll data in the original space

To illustrate this dependency of the postmapping error on the neighborhood size in a concrete example, the linear postmapping method is applied to 200 test data points with $k = 4$ and 8 respectively, and 800 points from the same roll (Figure 5.4) are picked as training data and the SeDuMi tool is used to create the initial mapping. From results shown in Figure 5.5, it is obvious that the increase of k from 4 to 8 has greatly reduced the blow out effect of the linear postmapping.



(a) k=4 (b) k=8

Figure 5.5: Unfolded Swiss Roll data with linear postmapped points

In summary, it may be concluded that linear postmapping works fine if the dimensionality reduction maintains local distances between points in the neighborhood

of a given point. This method is very fast and inexpensive to use, however it is sensitive to errors in preserving local distances in the training dimensionality reduction procedure, thus it is inappropriate for NLDR methods that do not preserve distances (like LLE algorithm).

5.5 SVD Postmapping

A more accurate postmapping method is developed to overcome the aforementioned problem. It uses singular value decomposition (SVD) to find a more robust mapping between $\Re^N$ and $\Re^n$. The difference of SVD postmapping with linear postmapping begins after obtaining D_{nbr} and R_{nbr}. First of all, D_{nbr} and R_{nbr} are centralized as follows

$$\begin{cases} \hat{D}_{nbr} = D_{nbr} - \bar{D} \\ \hat{R}_{nbr} = R_{nbr} - \bar{R} \end{cases} \tag{5.2}$$

where $\bar{D}$ is the mean value of D, $\bar{R}$ is the mean value of R . Then SVD is applied to the centralized data sets $\hat{D}_{nbr}$ and $\hat{R}_{nbr}$ to obtain

$$\begin{cases} \hat{D}_{nbr} = U_D S_D V_D^T \\ \hat{R}_{nbr} = U_R S_R V_R^T \end{cases} \tag{5.3}$$

where S_D and S_R contain singular values in descending order. Let us first assume that the two sets of data $\hat{D}_{nbr}$ and $\hat{R}_{nbr}$ have the same dimension, i.e. $n = N$. Since the sets of points in D_{nbr} and R_{nbr} are isometric, then the centralized data $\hat{R}_{nbr}$ and $\hat{D}_{nbr}$ are linearly related by an orthonormal matrix P

$$\hat{R}_{nbr} = \hat{D}_{nbr} P \tag{5.4}$$

and P can be obtained via SVD of $\hat{D}_{nbr}$ and $\hat{R}_{nbr}$. Using Eq. 5.3, the orthonormal matrix P in Eq. 5.4 can be obtained as follows

$$P = V_D S_D^{-1} U_D^T U_R S_R V_R^T \tag{5.5}$$

where S_D^{-1} is the pseudo inverse of S_D.

To find a mapped point r for a new data point d, first shift it to obtain $\hat{d} = d - \bar{D}$. Assume that the local isometry is also preserved for $\hat{d}$ and the neighborhood $\hat{D}_{nbr}$, the desired embedding $\hat{r}$ has same relation with $\hat{d}$ and can be obtained as follows

$$\begin{aligned} \hat{r} &= \hat{d}P \\ &= \hat{d}V_D S_D^{-1} U_D^T U_R S_R V_R^T \quad . \end{aligned} \tag{5.6}$$

After obtaining $\hat{r}$, shift it to get the mapped point r in $\Re^n$

$$r = \hat{r} + \bar{R} \quad . \tag{5.7}$$

This result is true for the case of exact isometry between D_{nbr} and R_{nbr}. However, since, in NLDR methods, local isometry is only approximately maintained (isometry is maintained in sufficiently small neighborhoods), a robust matching procedure needs to be developed to obtain a desired transformation. This matching will be based on the coordinates matching in the orthogonal coordinate systems by relating U_D and U_R with a diagonal matrix M.

Lemma 1

If the set of points $\hat{D}_{nbr}$ and $\hat{R}_{nbr}$ are isometric then their orthonormal matrices U_D and U_R are simply related through a diagonal matrix M which contains 1 or -1 on its diagonal.

Proof :

Since P is orthonormal, we have

$$PP^T = V_D S_D^{-1} U_D^T U_R S_R^2 U_R^T U_D S_D^{-1} V_D^T = I \tag{5.8}$$

consequently, we get

$$U_D^T U_R S_R^2 U_R^T U_D = S_D^2 \tag{5.9}$$

and since $\hat{D}_{nbr}$ and $\hat{R}_{nbr}$ are isometric, thus both S_R and S_D have identical singular values in descending order, therefore $S_R = S_D$. Defining $M = U_D^T U_R$, from Eq. 5.9 we have

$$MS_R^2 M^T = S_R^2 \tag{5.10}$$

Note that M is also orthonormal, thus from Eq. 5.10 $MS_R^2 = S_R^2 M$, and since S_R^2 is diagonal, M is diagonal too. From Eq. 5.10, since M and S_R^2 are diagonal, then $M^2 = I$, then the diagonal elements on M are either 1 or -1. i.e. $m_{ii} = \pm 1$. From the definition of M, have

$$U_R = U_D M \tag{5.11}$$

which proves the lemma. □

Case with dimensionality reduction

Now let us consider the case when dimensionality of data D is reduced from N to n, for which $N > k \geq n$. Note that, in general, the number of neighbors k should not be less than the intrinsic dimensionality n, so $k \geq n$. Thus U_D and U_R are $k \times k$ matrices, since the pseudo inverse S_R^{-1} is a $n \times k$ matrix and only first n columns have non-zero elements on diagonal. The calculated U_R can be trimmed to a $k \times n$ matrix $\tilde{U}_R$.

$$\tilde{U}_R = \hat{R}_{nbr} V_R \tilde{S}_R^{-1} \tag{5.12}$$

where $\tilde{S}_R^{-1}$ consists of the first n columns of S_R^{-1}. Accordingly, U_D needs to be trimmed to $k \times n$ as well.

$$\tilde{U}_D = \hat{D}_{nbr} \tilde{V}_D \tilde{S}_D^{-1} \tag{5.13}$$

where $\tilde{V}_D$ is the first n columns of V_D, and $\tilde{S}_D^{-1}$ is the first n columns and first n rows of S_D^{-1}. Actually, $\tilde{U}_D$ and $\tilde{U}_R$ can be expressed as

$$\begin{aligned} \tilde{U}_D &= (d_1, d_2, \ldots, d_n) \\ \tilde{U}_R &= (r_1, r_2, \ldots, r_n) \end{aligned} \tag{5.14}$$

where d_i $(i = 1, 2, \ldots, n)$ is the ith column vector of U_D and r_i $(i = 1, 2, \ldots, n)$ is the ith column vector of U_R. Similar to the case of $n = N$, a matching matrix $\tilde{M}$ can be found to relate $\tilde{U}_R$ with $\tilde{U}_D$

$$\tilde{U}_R = \tilde{U}_D \tilde{M} \tag{5.15}$$

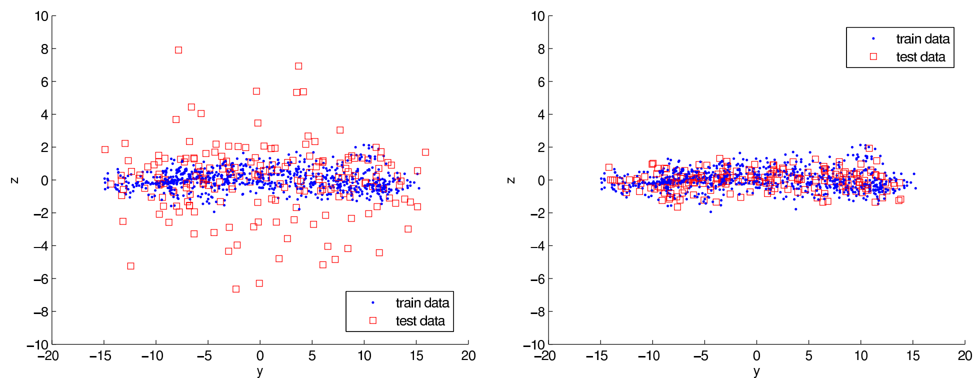
and used in Eq. 5.6 to determine the desired embedding.

5.6 Example Applications

To check the effectiveness of the proposed SVD postmapping method, several dimensionality reduction algorithms and various data sets including synthetic and real world data have been examined.

Example 1

In the first example, I will repeat the Swiss Roll experiment as displayed in Figure 5.5(a) and (b) with the SVD postmapping method. With exactly the same settings for training data and test data, the SVD postmapping has produced a better result than linear postmapping (k=8), as illustrated in Figure 5.6(a) and (b), where Figure 5.6(a) is actually a zoomed in view of Figure 5.5(b). The range of the SVD postmapped test data is about the same as the range of dimensionality reduced training data, while linear postmapping has produced a much wider spread of data values.



(a) Linear postmapped results, $k = 8$ (zoom in)

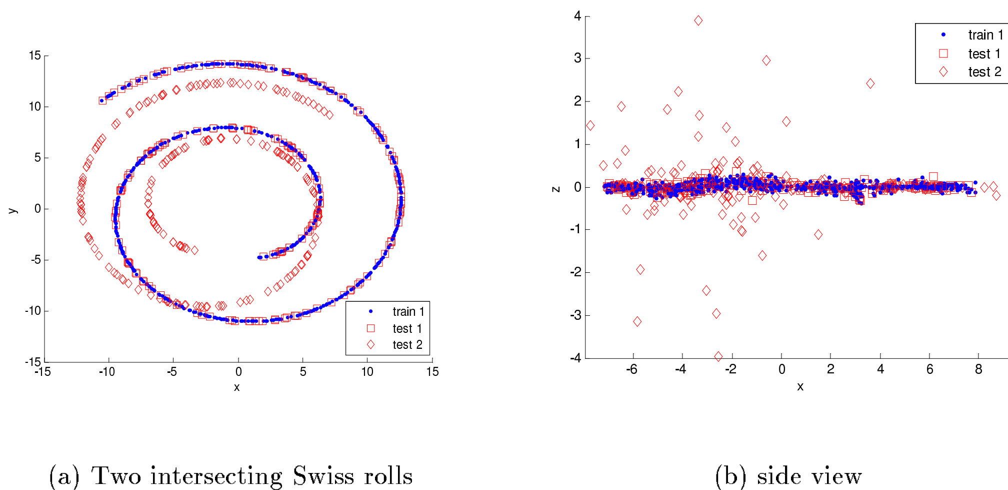
(b) SVD postmapped results (zoom in)

Figure 5.6: Comparison between linear and SVD postmapping

Example 2

The second example also involves the SeDuMi optimization tool and uses two intersecting Swiss Roll data sets as shown in Figure 5.7(a). It can be seen that the two rolls are closely intertwined with each other, making it difficult to separate with a linear classification method. The experiment is organized like this: first select some

data points from one Swiss Roll as class 1 training data, and apply kernel-PCA to the selected training data to obtain an unrolled Swiss Roll on a 2D plane after the dimensionality reduction procedure. Then select a number of points from both rolls, excluding the training data points, and label them as testing data class 1 and class 2, depending on which roll they are from. The SVD based postmapping method is thus applied to both classes of testing data and the mapped points are checked to see if the proposed postmapping method facilitates the separation of data from different classes in $\Re^n$ space.



(a) Two intersecting Swiss rolls (b) side view

Figure 5.7: Data after SeDuMi process and postmapping

In Figure 5.7(b), the mapping results (zoomed in for better illustration) are shown from the side view. It can be observed that most of the postmapped class 1 data are confined within the area of the mapped space defined by nonlinear dimensionality reduction of training data, and the majority of postmapped class 2 points are outside this area, which makes it easier to separate two classes in the reduced space (along the z axis) than in the original space.

Example 3

This example uses the Frey face database (available at `http://www.cs.toronto.edu/~roweis/data.html`). To make a fair comparison with the results by [RS00],

The LLE algorithm is used for nonlinear dimensionality reduction. Then the SVD postmapping procedure is applied using the mapped points as its base. First of all, one picture is picked out of a total of 1965 pictures as a test sample P for postmapping. The remaining 1964 pictures are sent to LLE for dimensionality reduction, generating a 2D pattern, then sample P is also transformed to this 2D space by SVD postmapping and is denoted as P_{post}. To check the accuracy of the SVD postmapping method, the 1965 pictures are mapped to 2D space by the LLE algorithm, creating a reference pattern. The point in 2D space corresponding to P is denoted as P_{LLE}. The Euclidean distance between P_{post} and P_{LLE} is calculated and denoted as $dist$. This procedure is repeated for 634 different pictures in the data set, and the distribution of $dist$ is displayed in Figure 5.8. The majority of such pairs (474 out of 634 or about 75%) have distances of less than 0.0839 (the stand deviation of $dist$).

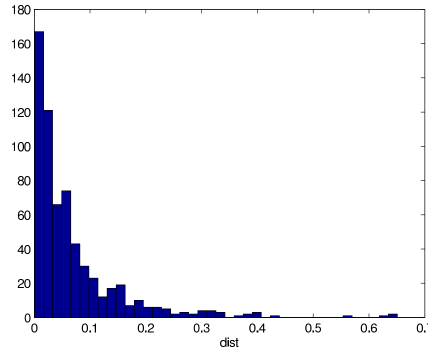


Figure 5.8: Histogram of distances between P_{post} and P_{LLE}, $\overline{dist} = 0.0676$, $\sigma_{dist} = 0.0839$

To learn more about the nature of the postmapping error, a mapped point with the maximum value of $dist$ equal to 0.65 is picked and its corresponding mapping is shown in Figure 5.9, where the neighbors of the test data are marked with triangles.

The test data mapped by LLE and SVD postmapping are marked as circle and square, respectively. It can be observed that the test data have been pushed away from its five neighbors by the LLE algorithm, while the SVD postmapping method reconstructs the test data closer to the set of neighbors. However, even in this worst case distance measure, the facial expression of all illustrated neighborhood pictures are similar. This indicates that in spite of the apparent shift of the neighborhood points,

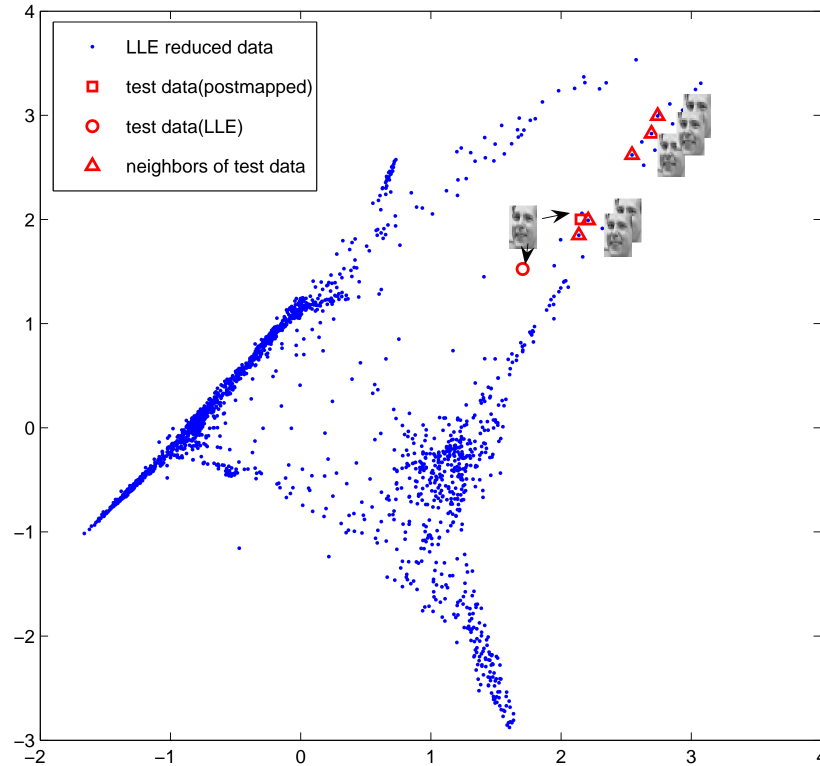


Figure 5.9: Illustration of the point shift by postmapping (dist=0.65)

the SVD postmapping procedure mapped the similar images in close proximity groups in lower dimensional space.

In the next two examples, the postmapping method will be shown to be able to preserve meaningful properties for two different databases working with two different NLDR algorithms.

Example 4

This experiment uses a rotating face database (available at `http://isomap.stanford.edu`) with different poses and lighting directions. 630 out of 698 faces are picked as training data and ISOMAP is used to obtain a set of data in a reduced dimensional space as displayed in Figure 5.10. The remaining 68 faces are used as test data for the SVD postmapping method. The postmapped result is shown in Figure 5.11,

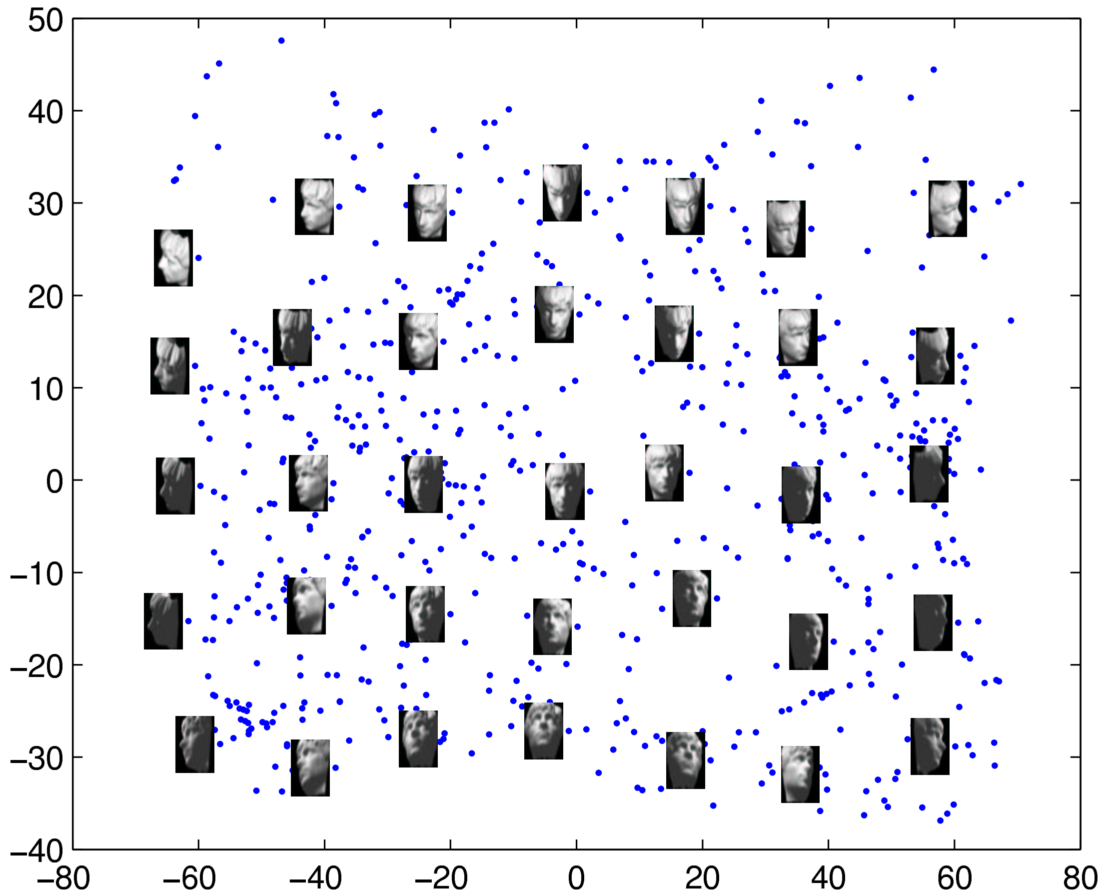


Figure 5.10: Face data dimensionality reduction by ISOMAP

where squares mark the postmapped points with corresponding faces on the left. As observed from Figure 5.10 and Figure 5.11, the postmapped faces match well in expression and poses with training faces displayed in Figure 5.10.

Example 5

This experiment uses the LLE procedure and a modified NIST hand written digit database [LBBH01], which has 60000 samples in the training set and 10000 samples in the testing set. Due to the computing cost, 1000 samples are selected as a training set, another 1000 samples are picked as a testing set. The results of mapping the training and testing sets are displayed in Figure 5.12 and Figure 5.13. Notice that

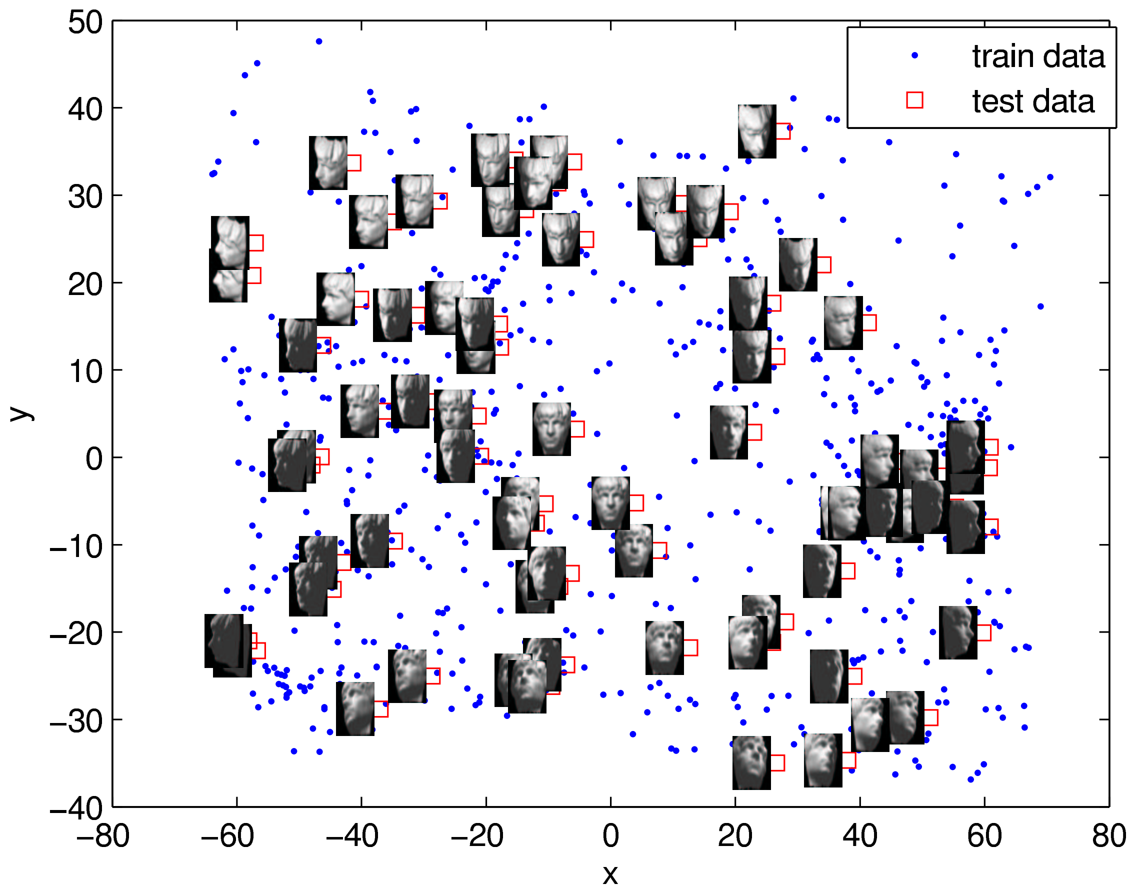


Figure 5.11: Face data postmapped SVD postmapping based on ISOMAP

the similarity between the distribution of these two figures is clearly seen especially for the numbers “1”, “7” , “5”, “6” and “0” that are at the edge of the test patterns.

Results of these examples show that the proposed postmapping method correctly interpolates the test data in a reduced dimensional space, which shows its potential application for classification and machine learning.

Example 6

In this example, a comparison of SVD postmapping with the method presented by [BPV+04] applied to the LLE algorithm is given. To make a fair comparison with that method, the same experiments were repeated on the same data (rotating Face Data). First, the whole face data D is split into three sets, F, R1 and R2, with the

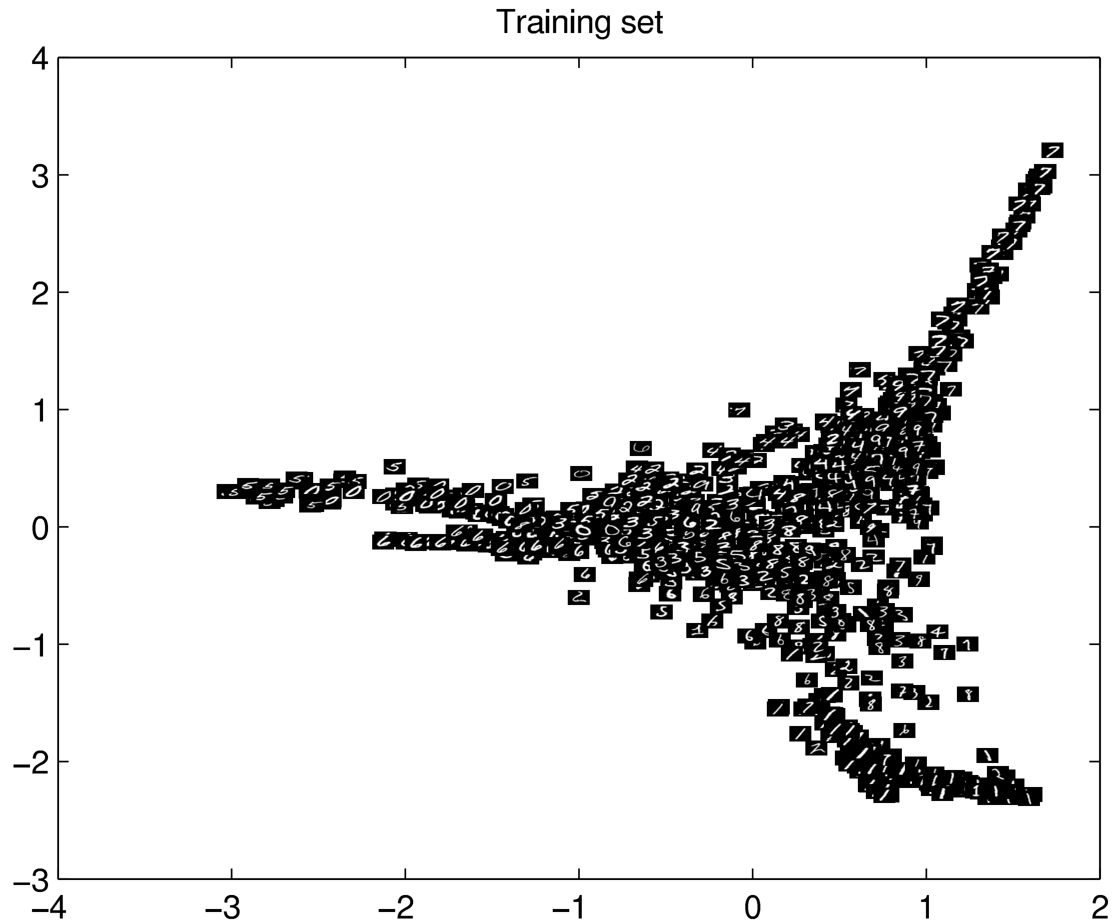


Figure 5.12: MNIST data mapped by LLE

same size for R1 and R2, then the experiment is organized in 3 steps as described by [BPV+04].

1. First train over sets $F \cup R1$ and $F \cup R2$. Then an affine alignment between the two embeddings is made to remove possible translation of the eigenvectors resulting from LLE algorithm. The Euclidean distance between the aligned embeddings is then obtained for each point in F. This represents the training set variability.

2. For each sample $S_i \in F$, training is performed over $\{F \cup R1\} \setminus \{S_i\}$. Then the SVD postmapping method is applied to S_i to find its embedding in the reduced dimensionality space. Then calculate the Euclidean distance between this embedding and the one obtained when training with $F \cup R1$, i.e. with S_i in the training set.

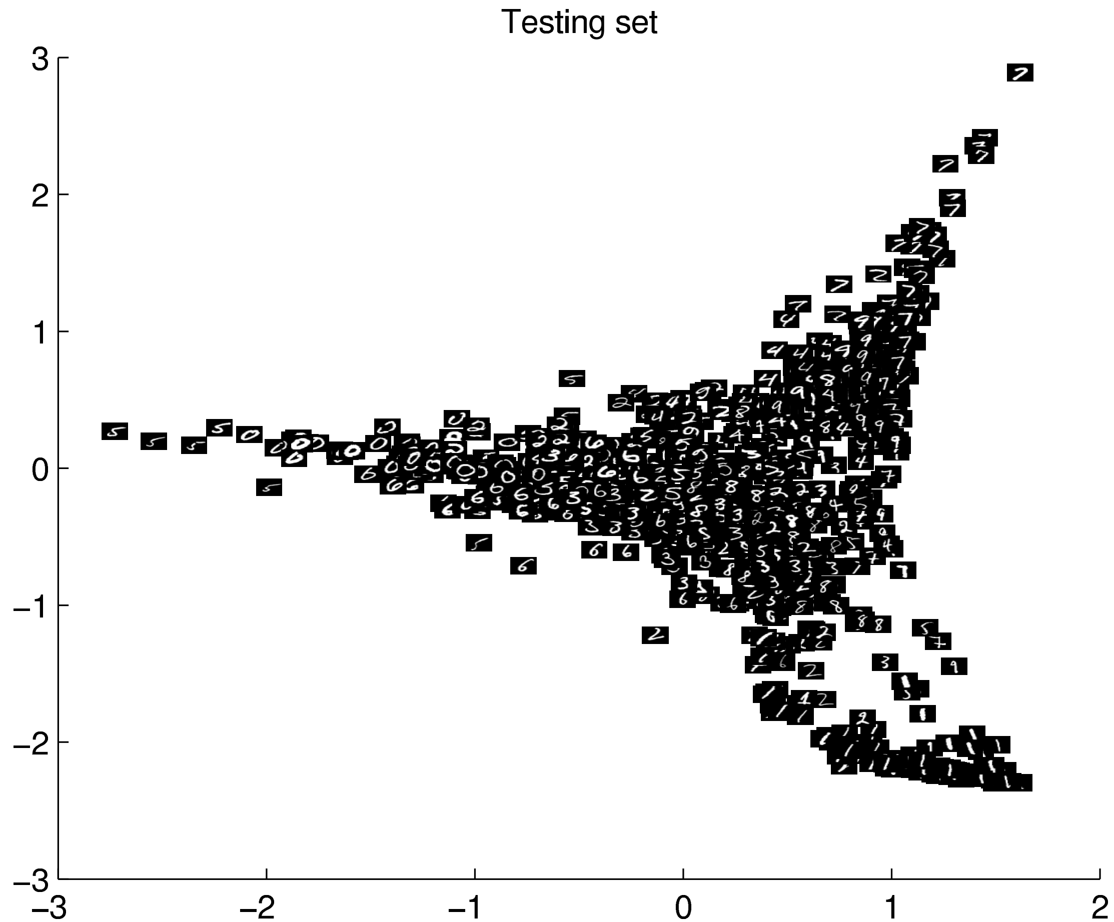


Figure 5.13: MNIST data postmapped by SVD postmapping based on LLE

This represents the postmapping error.

3. The mean difference between the distance obtained in step 1 and the one obtained in step 2 is calculated for each sample $S_i \in F$, and this experiment is repeated for various sizes of F.

The results are presented in Figure 5.14 with x axis representing the proportion of the substitution in training set and y axis representing the logarithm of the ratio between postmapping error and training set variability. From the plotted results, it can be observed that the postmapping error is always smaller than the training set variability for Laplacian, for ISOMAP and LLE procedure, the postmapping error is

bigger only for a small proportion of the substitution. These results are in agreement with the result in [BPV$^+$04].

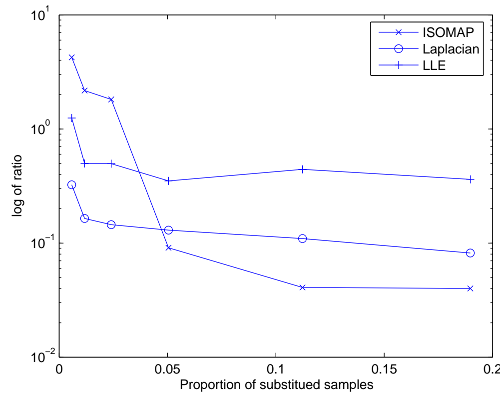


Figure 5.14: Postmapping error vs. training set variability

Example 7

To make a comparison between non-reduced data and dimensionality reduced data on the effect to final classification rate, a vehicle data set experiment is conducted. This vehicle data set is composed of 116 samples from 29 different vehicles with four different positions for each vehicle. Four sample vehicle images with left, right, front and back positions are displayed in Figure 5.15.

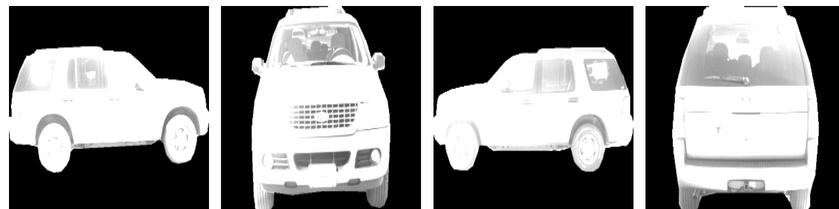


Figure 5.15: Vehicle data set images with four positions

Two popular algorithms — Support Vector Machine (SVM) [SWG+96] and Eigen-face [BHK97] have been used to get classification rates as a reference. Specifically, the OSU SVM tool[MZA] and eigen-face algorithm [Sun] are applied to this data set and the results are given in Table 5.1.

statistics of classification rate	SVM	Eigen-face
mean	91.2%	92.3%
standard deviation	0.035	0.031

Table 5.1: Classification rate for SVM and Eigen-face methods

The experiment is conducted as follows. First randomly pick 90 out of 116 samples as training set and remaining 26 samples as testing set. Then the SVM and Eigen-face methods are applied to both sets to get a classification rate. To get statistically stable results, this procedure is repeated 100 times with different random picks on training and testing sets each time. The averaged classification rate and related standard deviation are listed in Table 5.1 and will be used later to compare with the classification results based on postmapping.

For dimensionality reduction, the LLE+SVD postmapping algorithms are used to reduce the size of training data and testing data respectively. SVM and Eigen-face methods are used respectively as the classification methods after nonlinear (LLE) and linear (SVD) dimensionality reduction to check the postmapping effect on the classification rate.

	statistics of classification rate	$dim = 4$	$dim = 8$
SVM	mean	86.4%	91.8%
	standard deviation	0.046	0.038
Eigen-face	mean	84.1%	82.7%
	standard deviation	0.048	0.030

Table 5.2: Classification rate for LLE+postmapping+SVM method

From Table 5.2 it can be seen that the postmapping method works well with the SVM classifier, for reduced dimensionality $dim = 8$, the classifier rate is even slightly

higher than the result from non-reduced data. The reason for worse results from Eigen-face is because the Eigen-face method uses linear average as the center of the cluster, which is corrupted by the nonlinearity of the postmapping procedure.

5.7 Conclusion

It can be concluded that the proposed postmapping algorithms give a good estimate in the reduced dimensionality space for the new data points with no prior knowledge of specific NLDR algorithm and without repeating the dimensionality reduction computation. This is useful when the dimensionality reduction procedure demand for memory and computing time increases fast with the number of data points or when the dimensionality reduction procedure is not available during test phase. This is particularly useful in optimized kernel method [WSS04] when explicit mapping is determined only for training points.

Two postmapping methods have been developed in this chapter. The linear postmapping method is simple to implement and costs less to compute, but it is sensitive to noise. SVD postmapping is more complex but shows robustness to noise. Postmapping of a single data point using both linear and SVD methods have computational complexity of $O(D) + O(d^3)$, so for the cases with large D, it is almost linear. The examples in section 5.6 have illustrated that SVD postmapping can maintain meaningful properties in a reduced space for both synthetic and real-world data. Example 6 shows that the developed postmapping method has a similar error property that agrees with a recently published out-of-sample extension approach for three major NLDR procedures. Example 7 shows that for certain classifiers, the NLDR+postmapping preprocessing can significantly reduce the data size and keep the classification result as good as the original data.

There are two advantages of the proposed postmapping method — first, it reduces the computational cost of new data point mapping based on calculated dimensionality reduction results; second, it does not require any knowledge of the dimensionality reduction algorithm used. Therefore, it can help to simplify the extension procedure of the test data.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

As Moore's law predicts an exponential increase in the number of transistors per unit area, design productivity has been subjected to a permanent stress to grow at an equally fast pace. However, in spite of progress made in design automation, it has been lagging behind all the time. This dissertation addresses some problems and challenges of design automation for reconfigurable systems. RC systems, a very promising platform for next generation computing, have greatly expanded their spectrum of applications by integrating powerful GPP/DSP with reconfigurable logics. The market share of GPP/DSP has been eroded by a growing demand for RC systems. However, the new heterogeneous RC systems require a tighter collaboration between hardware and software designers and a better HW-SW co-design methodology.

"Productivity crisis" has become the Sword of Damocles hanging over almost every designer following the projection made by SemaTech in 1997 and increased the pressure to deliver a tool that will improve design productivity. Higher design productivity not only requires better EDA tools but also better integration of hardware and software and increased collaboration of HW-SW design efforts. Raising abstraction levels and promoting design-reuse have become a savior to stressed out designers. We have seen a trend to improve design productivity by switching design methodology

from an RT level to a behavioral level, as we have seen the switch from graphic-base methodology to text-base methodology decades ago.

Today's High Level Synthesis tools are mainly focused on better interpretation of behavioral descriptions and most of them are proprietary. Even if two HLS tools accept the same design language, it is almost certain that they are speaking two incompatible dialects. This impedes migration between different languages and is detrimental to design reuse. A significant amount of work has to be done manually if a system designed in one language is to be migrated to another language.

Noting the needs to improve design productivity and design reuse and the two ongoing RC system projects at OU, the work of this dissertation has been carried out in two overlapping domains. The first domain deals with high level design methodology. To ease the design migration between design flows and to reduce the redundant work, a novel design methodology and corresponding design tool – UADL are introduced and developed. This new design methodology is mainly focused on facilitating design flow, especially between the hardware and software designers. It is designed to provide an open, unified design interface to engineers of different domains. UADL has been successfully applied to the DRAW project and has shown considerable savings in redundant coding between VHDL and Matlab design flow. It is also used in the SOLAR project for a unified testbench which provides a tool independent interface for VHDL testbench creation.

The second domain deals with SOLAR and DRAW projects. Both of these two projects are based on the idea of the RC platform, and they both have similar mesh style structures. But SOLAR is focused on self-organization of reconfigurable connections and neuron functions for machine intelligence, while DRAW is aimed at providing flexibility, high performance devices for wireless baseband signal processing. Specifically, the following topics have been researched for these two projects.

SOLAR Project

Two research tasks have been accomplished for the SOLAR project. First of all, connectivity challenges in SOLAR are inspected. There are two issues related to the connectivity challenges. The first issue is to study the optimal input selection

strategy and the optimal input weighting scheme for individual neurons. The second issue is to develop a reconfigurable routing scheme with lower hardware cost. A novel pipeline scheme was devised and implemented in the HDL model to achieve a reconfigurable routing channel with linear hardware cost and a time delay as a function of the channel size. Another important feature of the developed pipeline structure is that the pipeline parameters can be adjusted to obtain different levels of tradeoffs between flexibility and throughput. The UADL tool has been used in SOLAR to create a tool independent testbench and to improve the design productivity in testbench coding. Secondly, as an indispensable preprocessing step of SOLAR, NLDR algorithms are studied and two postmapping algorithms have been developed to lower the computational burden brought by NLDR.

DRAW Project

The DRAW project was initially based on my cooperation with Ahmad's research work [Als02]. The project's goal is to develop a reconfigurable architecture that is able to fulfill the needs of future generation mobile communication. I had co-developed HDL models for several key components such as DRPU, DRAP, etc. In this dissertation, part of the DRAP design is revisited and the UADL design flow is applied to the Barrel Shifter (a part of DRAP) design for two target languages — VHDL and Matlab. Considerable reduction of redundant work between VHDL and Matlab design flows has been shown.

Second, a Turbo decoder, an important and complex unit in wireless communication, is used to demonstrate the capability of the DRAW structure. The LLR and α (or β) computing parts of the Turbo decoder are implemented in UADL, and VHDL and Matlab codes are automatically generated and functionalities verified. About 2 times savings in coding effort is observed.

In summary, in this dissertation I have addressed various research topics within the RC system domain, which includes design methodology and tool development, reconfigurable interconnect and reconfigurable routing, DRAW implementation of Turbo decoder, UADL applications to SOLAR and DRAW projects and theoretical research in dimensionality reduction and postmapping algorithms,.

6.2 Original Contribution

In this section, I reiterate the novelty and major advantages of the developed UADL design methodology and software tool. The problem with existing HLS tools is their mutual incompatibility. A design developed for HLS tool A cannot be directly used for tool B. Many of the migration works need to be done manually, causing low design productivity and redundant works.

Inspired by Java, a cross-platform programming language, UADL has adopted a similar programming paradigm. By inserting UADL as an intermediate layer in the design flow (Figure 2.2), the end-users (system designers) do not have to manually redesign their works when a change in design tool selection occurs.

With the openness of tool selection implemented in the UADL concept, one is able to combine the best from all HLS tools. Suppose there is a design that consists of a control-intensive part and an arithmetic-intensive part, UADL can assign these two parts to two different HLS tools that have better algorithms dealing with control logic and arithmetic operations, respectively, assuming there is not much interaction between the control part and the arithmetic part. This kind of combination would be impossible within the existing HLS tools.

The popularity of Java language has proved the effectiveness of this programming paradigm where the end-user's design productivity is treated with a higher priority. It is reasonable to expect that a similar paradigm in RC system design focused on improving the designer's productivity will gain its wide acceptance in the future.

6.3 Future Work

Reconfigurable computing system design is a relatively young field. In spite of many great discoveries and accomplishments, there are still many problems that remain open. In my opinion, the following research topics are worth further study and research.

1. Develop additional UADL engines to support other design languages such as Verilog, C/C++ and Java, etc. Another important improvement includes better support for system level algorithm design.

2. As numerous efforts have shown the potential of manifold learning in the field of visualization and classification [VDG+02; DRD02; BN03b], future work needs to be carried out to study the effect of the SVD postmapping method on classification accuracy with different NLDR algorithms.

3. Develop accessory tools for the proposed pipeline scheme. For example, it would be nice to have an algorithm that could read in any connection configuration of the array, output suitable parameters for the pipeline, and automatically generate the read/write slots information for each node. Another improvement worth investigation is that currently the proposed pipeline scheme only supports a feed forward array. It would be an interesting work to develop a dual pipeline structure to support bi-directional information transmission, which would give the pipeline structure a wider spectrum of applications.

As a new computing platform, I believe that the reconfigurable computing market will see a rapid growth in the near future. As the transistor size is shrinking to the physical limits, a smarter way of using the silicon area is a natural choice to support the continuous development of the semiconductor industry.

Bibliography

[3GP] 3GPP. 3GPP standard organization. http:/www.3gpp.org.

[Als02] Ahmad Alsolaim. *Dynamically Reconfigurable Architecture for Third Generation Mobile Systems.* PhD thesis, Ohio University, 2002.

[Alt] Altera. Altera inc. http://www.altera.com/products/prd-index.html.

[ASBG00] Ahmad Alsolaim, Janusz Starzyk, Jürgen Becker, and Manfred Glesner. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. In *FCCM*, pages 205–216, 2000.

[BC01] T. Blankenship and B. Classon. Fixed-point performance of low complexity turbo decoding algorithms. In *Proc. IEEE Veh. Tech. Conf.*, Rhodes, Greece, May 2001.

[BCJR74] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Trans. Inf. Theory*, 20:284–287, March 1974.

[Ber99] Reinaldo A. Bergamaschi. Behavioral network graph: Unifying the domains of high-level and logic synthesis. In *DAC*, pages 213–218, 1999.

[BGT93] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding. In *ICC'93*, pages 1064–1070, Genève, Switzerland, May 1993.

[BHK97] P.N. Belhumeur, J.P. Hespanha, and D.J. Kriegman. Eigenfaces vs. fisherfaces: recognition using class specific linear projection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(7):711–720, 1997.

[BLO98] Gaetano Borriello, Luciano Lavagno, and Ross B. Ortega. Interface synthesis: a vertical slice from digital logic to software components. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 693–695, New York, NY, USA, 1998. ACM Press.

[BN03a] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Comput.*, 15(6):1373–1396, 2003.

[BN03b] Mikhail Belkin and Partha Niyogi. Using manifold stucture for partially labeled classification. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 929–936. MIT Press, Cambridge, MA, 2003.

[Boo51] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[BP02] K. Bondalapati and V.K. Prasanna. Reconfigurable computing systems. *Proceedings of the IEEE*, 90(7):1201–1217, July 2002.

[BPG00] J. Becker, T. Pionteck, and M. Glesner. An application-tailored dynamically reconfigurable hardware architecture for digital baseband processing. In *SBCCI '00: Proceedings of the 13th symposium on Integrated circuits and systems design*, page 341, Washington, DC, USA, 2000. IEEE Computer Society.

[BPV+04] Yoshua Bengio, Jean-François Paiement, Pascal Vincent, Olivier Delalleau, Nicolas Le Roux, and Marie Ouimet. Out-of-sample extensions for lle, isomap, mds, eigenmaps, and spectral clustering. In Sebastian Thrun,

Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16.* MIT Press, Cambridge, MA, 2004.

[CGK00] A. Chass, A. Gubeskys, and G. Kutz. Efficient software implementationn of the max-log-map turbo decoder on the starcore sc140 dsp. In *Int. Conf. Signal Process Application and Technology*, Herzelia, Israel, Octobor 2000.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[CP95] Jui-Ming Chang and Massoud Pedram. Register allocation and binding for low power. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 29–35, New York, NY, USA, 1995. ACM Press.

[CPVS03] O. Cret, K. Pusztai, C. Vancea, and B. Szente. CREC: a novel reconfigurable computing design methodology. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 175–182, April 22-26 2003.

[CR92] D. C. Chen and J. M. Rabaey. A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, December 1992.

[DMS03] Robertas Damasevicius, Giedrius Majauskas, and Vytautas Stuikys. Application of design patterns for hardware design. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 48–53, New York, NY, USA, 2003. ACM Press.

[Don81] W.E. Donath. Wire length distribution for placement of computer logic. *IBM Journal of Research and Development*, 25(3):152–155, 1981.

[DRD02] D. De Ridder and R.P.W Duin. Locally linear embedding for classification. Technical report PH-2002-01, Delft University of Technology, Delft, The Netherlands, 2002.

[dST03] Vin de Silva and Joshua B. Tenenbaum. Global versus local methods in nonlinear dimensionality reduction. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 705–712. MIT Press, Cambridge, MA, 2003.

[EBJ+98] J. L. Elman, E. A. Bates, M. H. Johnson, A. Karmiloff-Smith, D. Parisi, and K. Plunkett. *Rethinking Innateness.* MIT press, Cambridge, MA, February 1998.

[ECF96] C. Ebeling, D.C. Cronquist, and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, 1996. Springer-Verlag.

[Est60] G. Estrin. Organization of computer systems–the fixed plus variable structure computer. In *Proc. Western Joint Computer Conf.*, pages 33–40, New York, 1960.

[GG98] W. Gross and P. Gulak. Simplified map algorithm suitable for implementation of turbo decoders. *Electronics Letters*, 34(16):1577–1578, August 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GMG+01] G.Masera, M.Mazza, G.Piccinini, F.Viglione, and M.Zamboni. Low-cost ip-blocks for umts turbo decoders. In *Proceedings of ESSCIRC*, Villach, Austria, September 2001.

[GO03] Eike Grimpe and Frank Oppenheimer. Extending the systemc synthesis subset by object-oriented features. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 25–30, New York, NY, USA, 2003. ACM Press.

[GR94] D. D. Gajski and L. Ramachandran. Introduction to High-level synthesis. *IEEE Design and Test of Computers*, Winter 1994.

[GSB+00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[HJ04] Vincent P. Heuring and Harry F. Jordan. *Computer Systems Design and Architecture*. Prentice Hall, Upper Saddle River, NJ, 2004.

[HK01] Fred Ham and Ivica Kostanic. *Principles of Neurocomputing for Science and Engineering*. McGraw-Hill, New York, 2001.

[HLMS04] Jihun Ham, Daniel D. Lee, Sebastian Mika, and Bernhard Schölkopf. A kernel view of the dimensionality reduction of manifolds. In *ICML '04: Twenty-first international conference on Machine learning*, New York, NY, USA, 2004. ACM Press.

[HW97] John R. Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[Hwa79] Kai Hwang. *Computer Arithmetic: Principles, Architecture and Design*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[KAW04] Sami Khawam, Tughrul Arslan, and Fred Westall. Synthesizable reconfigurable array targeting distributed arithmetic for system-on-chip applications. In *IPDPS*, 2004.

[KMN+00] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. L. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform–based design. *IEEE Trans. on CAD*, 2000.

[Koh97] Teuvo Kohonen, editor. *Self-organizing maps*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[KOW+01] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, Marc Edwards, and Yaron Kashai. A framework for object oriented hardware specification, verification, and synthesis. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 413–418, New York, NY, USA, 2001. ACM Press.

[LBBH01] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Intelligent Signal Processing*, pages 306–351. IEEE Press, 2001.

[Lin97] Youn-Long Lin. Recent developments in high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 2(1):2–21, 1997.

[LKJ99] G. Lakshminarayana, K.S. Khouri, and N.K. Jha. Wavesched: a novel scheduling technique for control-flow intensive behavioral descriptions. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(5):505–523, May 1999.

[LS05] P. Lysaght and P.A. Subrahmanyam. Guest editors' introduction: Advances in configurable computing. *Design and Test of Computers, IEEE*, 22(2):85–89, March 2005.

[LSL+00] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M. C. Filho, and Vladimir Castro Alves. Design and implementation of the morphosys reconfigurable computing processor. *J. VLSI Signal Process. Syst.*, 24(2-3):147–164, 2000.

[Mar02] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[MB04] Fan Mo and Robert K. Brayton. A timing-driven module-based chip design flow. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 67–70, New York, NY, USA, 2004. ACM Press.

[MLM$^+$05] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *Design and Test of Computers, IEEE*, 22(2):90–101, March 2005.

[MM04] Mahmoud Meribout and Masato Motomura. A combined approach to high-level synthesis for dynamically reconfigurable systems. *IEEE Trans. Computers*, 53(12):1508–1522, 2004.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[MPMF01] O. Mencer, M. Platzner, M. Morf, and M.J. Flynn. Object-oriented domain specific compilers for programming fpgas. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(1):205–210, Feburary 2001.

[MPRZ99] Guido Masera, Gianluca Piccinini, Massimo Ruo Roch, and Maurizio Zamboni. Vlsi architectures for turbo codes. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(3):369–379, 1999.

[MSK$^+$99] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143, Monterey, California, United States, 1999. ACM Press.

[MW01] H. Michel and N. Wehn. Turbo-decoder quantization for UMTS, 2001.

[MZA] Junshui Ma, Yi Zhao, and Stanley Ahalt. OSU SVM classifier matlab toolbox). `http://eewww.eng.ohio-state.edu/~maj/osu_svm/`.

[NBD$^+$03] W.A. Najjar, W. Bohm, B.A. Draper, J. Hammes, R. Rinker, J.R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69, August 2003.

[Ope] Open SystemC Initiative. SystemC. `http://www.systemc.org/`.

[OS85] D.D. O'Leary and B.B Stanfield. Occipital cortical neurons with transient pyramidal tract axons extend and maintain collaterals to subcortical but not intracortical targets. *Brain Research*, May 1985.

[OW97] M. L. Overton and H. Wolkowicz, editors. *Semidefinite Programming.* 1997. Dedicated to the memory of Svatopluk Poljak, Math. Programming **77** (1997), no. 2, Ser. B.

[Pom04] Luigi Pomante. Exploiting polymorphism in hw design: a case study in the atm domain. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 81–85, New York, NY, USA, 2004. ACM Press.

[Rab97] Jan M. Rabaey. Reconfigurable computing: The solution to low power programmable dsp. In *ICASSP Conference*, Munich, Germany, April 1997.

[Rab99] Jan M. Rabaey. Beyond the third generation of wireless communications. In *ICICS*, Singapore, December 1999. Keynote presentation.

[RDJW97] Anand Raghunathan, Sujit Dey, Niraj K. Jha, and Kazutoshi Wakabayashi. Power management techniques for control-flow intensive designs. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 429–434, New York, NY, USA, 1997. ACM Press.

[RHV97] P. Robertson, P. Hoher, and E. Villebrun. Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding, 1997.

[RJ97] A. Raghunathan and N.K. Jha. Scalp: an iterative-improvement-based low-power data path synthesis system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(11):1260–1277, November 1997.

[RMBL05] Fernando Rincon, Francisco Moya, Jesus Barba, and Juan Carlos Lopez. Model reuse through hardware design patterns. In *DATE '05: Proceedings*

of the conference on Design, Automation and Test in Europe, pages 324–329, Washington, DC, USA, 2005. IEEE Computer Society.

[RN04] T. Ristimaki and J. Nurmi. Reconfigurable ip blocks: a survey. In *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, pages 117–122, 16-18 November 2004.

[RS00] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, December 2000.

[SDH05] Janusz Starzyk, Mingwei Ding, and Haibo He. Optimized interconnections in probabilistic self-organizing learning. In *Proc. IASTED Int. Conf. on Artificial Intelligence and Applications*, Innsbruck, Austria, Feburary 14-16 2005.

[Sem97] Sematech, 1997. `http://www.sematech.org`.

[SG03] J. Starzyk and Y. Guo. Dynamically self-reconfigurable machine learning structure for fpga implementation. In *ERSA*, Las Vegas, Nevada, June 23-26 2003.

[SK04] Keoncheol Shin and Taewhan Kim. An integrated approach to timing-driven synthesis and placement of arithmetic circuits. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 155–158, Piscataway, NJ, USA, 2004. IEEE Press.

[SPR90] M. Sur, S.L. Pallas, and A.W Roe. Cross-modal plasticity in cortical development: Differentiation and specification of sensory neocortex. *Trends in Neuroscience*, 13:227–233, 1990.

[SR03] Lawrence K. Saul and Sam T. Roweis. Think globally, fit locally: unsupervised learning of low dimensional manifolds. *J. Mach. Learn. Res.*, 4:119–155, 2003.

[SSM98] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Comput.*, 10(5):1299–1319, 1998.

[Stu99] J.F. Sturm. Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11-12:625–653, 1999.

[Sun] Wei Sun. Shape analysis in computer vision final project report: Face recognition. `http://cim.mcgill.ca/~wsun/sa/project/report.html`.

[SWG+96] M. O. Stitson, J. Weston, A Gammerman, V. Vapnik, and V. Vovk. Theory of sv machines. Technical report, University of London, Royal Holloway, Egham, UK, 1996.

[SZL05] J.A. Starzyk, Zhen Zhu, and Tsun-Ho Liu. Self-organizing learning array. *Neural Networks, IEEE Transactions on*, 16(2):355–363, March 2005.

[TdSL00] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

[TS03] Nick Tredennick and Brion Shimamoto. The rise of reconfigurable systems. In *Engineering of Reconfigurable Systems and Algorithms*, pages 3–12, 2003.

[UCI05] UCI. UCI machine learning repository, 2005. Available at `http://http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[Vap98] V. Vapnik. *Statistical Learning Theory.* Wiley, New York, 1998.

[VB96] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Rev.*, 38(1):49–95, 1996.

[VDG+02] Michail Vlachos, Carlotta Domeniconi, Dimitrios Gunopulos, George Kollios, and Nick Koudas. Non-linear dimensionality reduction techniques

for classification and visualization. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 645–651, New York, NY, USA, 2002. ACM Press.

[VMP+00] F. Viglione, Guido Masera, Gianluca Piccinini, M. Ruo Roch, and Maurizio Zamboni. A 50 mbit/s iterative turbo-decoder. In *DATE*, pages 176–180, 2000.

[VPI05] M. Vuletid, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *Design and Test of Computers, IEEE*, 22(2):102–113, March 2005.

[VRV04] J. Verbeek, Sam T. Roweis, and Nikos Vlassis. Non-linear cca and pca by alignment of local models. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[VVK02] J. Verbeek, N. Vlassis, and B. Kröse. Fast nonlinear dimensionality reduction with topology preserving networks. In *Proc. 10th European Symposium on Artificial Neural Networks*, 2002.

[Wer74] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[WHW01] Michael J. Wirthlin, Brad L. Hutchings, and Carl Worth. Synthesizing rtl hardware from java byte codes. In *FPL*, pages 123–132, 2001.

[WLW01] Alexander Worm, Holger Lamm, and Norbert Wehn. Design of low-power high-speed maximum a priori decoder architectures. In *DATE*, pages 258–267, 2001.

[WSS04] Kilian Q. Weinberger, Fei Sha, and Lawrence K. Saul. Learning a kernel matrix for nonlinear dimensionality reduction. In *ICML '04: Twenty-first international conference on Machine learning*, New York, NY, USA, 2004. ACM Press.

[WTS+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[Xil] Xilinx. Xilinx virtex products. `http://www.xilinx.com/products/`.

[YR95] A.K. Yeung and J. M. Rabaey. A 2.4 gops data-drivern reconfigurable multiprocessor ic for dsp. In *In Proceedings of the 1995 IEEE International Solid-State Circuits Conference*, pages 108–109, February 1995.

Appendix A

UADL Construction and Visitor Pattern

The construction and design methodology behind UADL development are given in this section. In essence, UADL is a light weight compiler that accepts a set of syntax rules and translates it to another set of syntax rules in the target language. Since the introduction of classic "Gang of Four" Design Pattern book [GHJV95], design pattern methodology has acquired wide acceptance in software engineering and many design patterns have been proposed and developed. For concrete implementation of the UADL tool development, Visitor pattern [Mar02] has been chosen. The Visitor pattern allows programmers to add new operations to classes without changing the existing code using a double-dispatch technique. Figure A.1 illustrates the class diagram of the Visitor pattern.

As one can notice the node class in Figure A.1 "accepts" a Visitor object as its input, then it calls that Visitor object's visit method, passing in itself as parameter. Based on the pass-in object type, the Visitor object will then execute corresponding visit method and its particular operations for that node object. Since the actual visiting function of a certain node is always implemented in two consecutive "accept" and "visit" function calls, this process is called "double-dispatch".

The UADL package consists of two major parts: syntax tree part and visitor part. The syntax tree part is responsible for parsing the input code according to the syntax

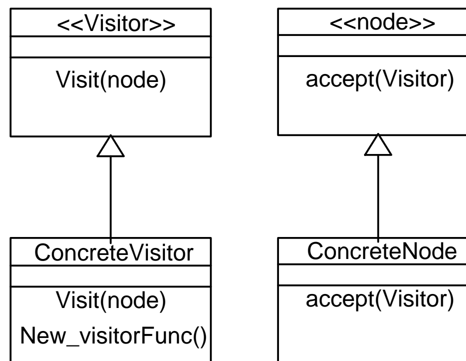


Figure A.1: Visitor and node classes UML diagram.

rule as specified in Figure 2.5. The visitor part is responsible for correct interpretation of the given element in the syntax tree. After a piece of UADL code is parsed and abstract syntax tree is generated, the traverse of the abstract syntax tree is achieved via Visitors.

Currently the UADL interpreter is developed in Java language. However, with the openness of the UADL structure, further implementation can be developed in other languages.

The following lists the major Java files for UADL syntax parsing.

```
/* basic node analysis */
Node.java
NodeChoice.java
NodeList.java
NodeListInterface.java
NodeListOptional.java
NodeOptional.java,
NodeSequence.java
NodeToken.java

/* UADL syntax component analysis */
Unit.java,
cfgBlock.java
cmdBlock.java
fileName.java
myStatement.java
nameList.java
```

```
simpleExpression.java
targetBlock.java
targetOption.java
```

The following lists the major UADL API for the visitor class with brief explanation of respective functionalities.

```
/**
   visit targetBlock, obtain target name,
 UADL engine and file extension
**/
public Object visit(targetBlock n, Object argu)

/**
   visit fileName, obtain the file name
**/
public Object visit(fileName n, Object argu)

/**
   visit targetOption, obtain the target options
 bound with current block.
**/
public Object visit(targetOption n, Object argu)

/**
   visit nameList, obtain a list of names
**/
public Object visit(nameList n, Object argu)

/**
   visit myStatement, perform correct interpretation
 according to the block that statement belong to
**/
public Object visit(myStatement n, Object argu)

/**
   visit simpleExpression, analyze the expression
 index expansion
**/
public Object visit(simpleExpression n, Object argu)
```

```
/**
   visit cmdBlock, obtain commmand type,
 decide the target binding.
**/
public Object visit(cmdBlock n, Object argu)

/**
   visit cfgBlock, obtain configuration information
 generate correspinding decalration for target language
**/
public Object visit(cfgBlock n, Object argu)
```

Appendix B

List of Acronyms

3GPP 3rd Generation Partnership Project

ANN Artificial Neural Network

ASIC Application Specific Integrated Chip

CAD Computer Aided Design

CDFG Control-Data Flow Graph

CDMA Code Division Multiple Access

CPU Central Processing Unit

DNA DeoxyriboNucleic Acid

DRAP Dynamic Reconfigurable Arithmetic Processor

DRAW Dynamic Reconfigurable Architecture for Wireless applications

DSP Digital signal Processor

EBNF Extended Backus Naur Form

EDA Electronic Design Automation

EDGE Enhanced Data rate for GSM Evolution

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPP General Purpose Processor

GPRS General Packet Radio Service

GSM Groupe Spécial Mobile (Global System for Mobile communications)

HDL Hardware Description Language

HLS High Level Synthesis

HW Hardware

IP Intellectual Property

JHDL Java HDL

KCPSM constant(K) Coded Programmable State Machine

LE Laplacian Eigenmap

LLE Locally Linear Embedding

LLR Log-Likelihood Ratio

LSB Least Significant Bit

MAP Maximum a *posteriori*

MSB Most Significant Bit

MUX Multiplexer

NLDR Non-Linear Dimensionality Reduction

OO Object Oriented

PCA Principal Component Analysis

PCCC Parallel Concatenated Convolutional Coding

RC Reconfigurable Computing

RSC Recursive Systematic Coder

RT(L) Register Transfer (Level)

SDP Semi-Definite Programming

SISO Soft-Input Soft Output

SoC System-on-Chip

SOLAR Self-Organizing Learning ARray

SVD Singular Value Decomposition

SVM Support Vector Machine

SW Software

UADL Unified Algorithmic Design Language

UMTS Universal Mobile Telecommunication System

VHDL VHSIC Hardware Description Language

VLSI Very Large Scale Integrated chip

Appendix C

Source Code Used in Dissertation

The source code packages used in this dissertation are available at following address:

`http://www.ent.ohiou.edu/~webcad/Graduate_Research`
`/Students_thesis_dissertation/Mingwei`

Under the **Mingwei** directory, there are four sub-directories:

- **UADL**, package for UADL tool in Chapter 2.
- **postmapping**, package for postmapping algorithms in Chapter 3.
- **pipeline**, package for pipeline design in Chapter 4.
- **DRAW**, package for DRAW design in Chapter 5.

Access to these packages is to be granted upon request.