1991

Use of functional programming paradigm in VLSI specification and design.

Wai S. Fong
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/4474

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000 ext. 3208.
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilming. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopy de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
USE OF FUNCTIONAL PROGRAMMING PARADIGM IN VLSI SPECIFICATION AND DESIGN

by

Wai S. Fong

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the School Of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Computer Science at the University of Windsor

Windsor, Ontario, Canada 1991
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

ABSTRACT

It has been claimed that pure functional programming languages facilitate the construction of complex programs and the subsequent proof of their correctness. In particular, claims have been made that pure functional languages are of value in VLSI specification and design where correctness is of importance. An investigation relevant to these claims has been carried out. A VLSI circuit specification transformer and an executable circuit specification language have been implemented in a pure lazy functional programming language. The advantages and disadvantages of using the functional programming paradigm in this application area are discussed.
To my parents.
ACKNOWLEDGMENTS

I would like to acknowledge the guidance and support provided by Dr. Richard Frost. He introduced me to Functional Programming and W/AGE. His help and encouragement were invaluable to me throughout the project. I would also like to thank Dr. S. Bandyopadhyay for the time and effort he has contributed in guiding me through this work. I would like to thank Dr. G.A. Jullien for his role as my supervisory committee member. Special thanks must go to Dimitris Phoukas, Bruce Erickson, Walid Saba, Martin Strobel, Martin Jaekel, and Arunita Sarkar who gave to me numerous suggestions. Finally, I would like to thank my parents, whose quiet encouragement has been a constant source of strength.
# TABLE OF CONTENTS

ABSTRACT .................................................................................. iv
DEDICATION ................................................................................. v
ACKNOWLEDGMENTS ................................................................. vi
LIST OF FIGURES ........................................................................ xiii

## PART I

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>INTRODUCTION .................................................................. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>THE THESIS UNDER INVESTIGATION .................................. 2</td>
</tr>
<tr>
<td>1.2</td>
<td>THE WINDSOR SYSTOLIC COMPILER ................................... 4</td>
</tr>
<tr>
<td>1.3</td>
<td>THESIS ORGANIZATION .................................................. 6</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>OUTLINE OF THE FUNCTIONAL CIRCUIT .............................. 9</td>
</tr>
<tr>
<td></td>
<td>SPECIFICATION TRANSFORMER : FCST ............................... 9</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>EXAMPLE USE OF FCST .................................................... 13</td>
</tr>
<tr>
<td>3.1</td>
<td>INVOKING FCST ............................................................. 13</td>
</tr>
<tr>
<td>3.2</td>
<td>HELP FACILITIES ........................................................... 14</td>
</tr>
<tr>
<td>3.3</td>
<td>TYPICAL EXAMPLE USE OF FCST ..................................... 14</td>
</tr>
<tr>
<td>3.4</td>
<td>CONCLUDING COMMENTS FOR THIS CHAPTER .................... 19</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>OVERVIEW OF TOOLS USED TO CONSTRUCT FCST ............. 20</td>
</tr>
<tr>
<td>4.1</td>
<td>THE ENTIRE SYSTEM IS IN A SINGLE PROGRAMMING ........... 20</td>
</tr>
<tr>
<td>4.1.1</td>
<td>The PARA Specification Language ................................. 21</td>
</tr>
<tr>
<td>4.1.2</td>
<td>The Circuit Specification Translators ............................ 22</td>
</tr>
<tr>
<td>4.2</td>
<td>OVERVIEW OF LAZY FUNCTIONAL PROGRAMMING LANGUAGES</td>
</tr>
<tr>
<td>4.3</td>
<td>OVERVIEW OF MIRANDA</td>
</tr>
<tr>
<td>4.3.1</td>
<td>The Miranda Programming Environment</td>
</tr>
<tr>
<td>4.3.2</td>
<td>The Type System</td>
</tr>
<tr>
<td>4.4</td>
<td>OVERVIEW OF W/AGE</td>
</tr>
<tr>
<td>4.5</td>
<td>THE WINDOW SYSTEM / HARDWARE PLATFORM</td>
</tr>
<tr>
<td>4.6</td>
<td>CONCLUDING COMMENTS FOR THIS CHAPTER</td>
</tr>
</tbody>
</table>

**PART II MORE DETAILED DESCRIPTION OF FCST**

**Chapter 1 THE SPECIFICATION LANGUAGES & DISPLAYS RELATED TO FCST**

| 1.1 | THE FIR FILTER SPECIFICATION LANGUAGE : FADL | 30 |
| 1.2 | THE EXECUTABLE CIRCUIT SPECIFICATION LANGUAGE : PARA | 31 |
| 1.3 | GRAPHIC DISPLAY OF THE SIMULATED CIRCUIT | 32 |
| 1.4 | THE MODULE DESCRIPTION LANGUAGE | 33 |
| 1.5 | THE NETLIST LANGUAGE: EDIF | 35 |
| 1.5.1 | EDIF 2.00 | 35 |
| 1.6 | CONCLUDING COMMENTS FOR THIS CHAPTER | 40 |

**Chapter 2 COMPONENTS OF FCST**

| 2.1 | FADL TO PARA TRANSLATOR | 41 |
| 2.1.1 | Definition Of a Grammar For The Input Language | 42 |
| 2.1.2 | Identification Of All Relevant Attributes And Based Types | 42 |
2.1.3  Specification Of An Attribute Grammar .......... 43
2.1.4  Definition Of All Attribute Functions .......... 45
2.2    PARA SPECIFICATION "EXECUTOR" .......... 46
       2.2.1  The Behavioral Model .......... 46
       2.2.2  Primitive Functions .......... 46
       2.2.3  Auxiliary Functions .......... 50
2.3    PARA OUTPUT TO GRAPHICAL DISPLAY
       TRANSLATOR .......... 52
2.4    PARA TO EDIF TRANSLATOR .......... 54
       2.4.1  General Discussion Of Attribute Types .......... 54
       2.4.2  Inherited Attributes Used In Translator .......... 55
2.5    TESTING THE VIABILITY OF FCST .......... 56
2.6    CONCLUDING COMMENTS FOR THIS CHAPTER .......... 57

Chapter 3
OUTLINE OF THE END-USER INTERFACE OF FCST .. 58

PART III  RELATED WORK AND CONCLUDING COMMENTS .. 60

Chapter 1
RELATED WORK .......... 61
       1.1  SUMMARY OF SURVEY .......... 61
       1.2  RELATIONSHIP TO THIS WORK .......... 62
       1.3  Relative advantages/disadvantages of FCST (compared
to closely related work) .......... 63
       1.4  CONCLUDING COMMENTS FOR THIS CHAPTER .......... 64
2.2 OVERVIEW OF THE FILTER ARCHITECTURE ........ 93

2.2.1 The Encoder ........................................... 94

2.2.2 The FIR Filter .......................................... 96

2.2.3 The Scaler ............................................... 97

APPENDIX 3 Summary: The Designer's Assistant .............. 100

3.1 Outline of the Windsor VLSI Designer's Assistant ....... 100

3.2 The KNOWLEDGE BASE ................................. 102

3.3 The INFERENCE ENGINE ................................. 103

3.4 The EXPLANATION FACILITIES ......................... 103

3.4.2 WHY Explanations ..................................... 104

3.4.3 HOW Explanations ..................................... 104

3.4.4 WHYNOT Explanation .................................. 104

3.5 The USER INTERFACE .................................... 104

3.6 The ABORT/RESET FACILITY ............................ 106

3.7 OUTPUT FROM THE DESIGNER'S ASSISTANT .......... 106

3.8 CONCLUDING COMMENTS ................................. 107

APPENDIX 4 FCST END-USER INSTRUCTIONS .................. 108

APPENDIX 5 FCST SCRIPTS ..................................... 109

5.1 FCST INTERFACE : FCST.m ............................. 109

5.2 TESTPARA : TESTPARA.m ............................... 110

5.3 PARATOEDIF : SIMPARA.m ............................... 111

5.4 FADL to PARA TRANSLATOR : FADLTOPARA.m ........ 112

xi
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The Systolic Compiler</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Overview of FCST</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Help facility and invoking testpara</td>
<td>14</td>
</tr>
<tr>
<td>Figure 4</td>
<td>A partial FADL description of a filter</td>
<td>15</td>
</tr>
<tr>
<td>Figure 5</td>
<td>A sample session of FCST</td>
<td>15</td>
</tr>
<tr>
<td>Figure 6</td>
<td>A sample of circuit specified in PARA</td>
<td>16</td>
</tr>
<tr>
<td>Figure 7</td>
<td>A sample of Waveform produced from FCST</td>
<td>17</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Netlist in EDIF produced by PARATOEDIF translator</td>
<td>18</td>
</tr>
<tr>
<td>Figure 9</td>
<td>An example function — gcd</td>
<td>24</td>
</tr>
<tr>
<td>Figure 10</td>
<td>An Example of FADL Description</td>
<td>31</td>
</tr>
<tr>
<td>Figure 11</td>
<td>An example of a FIR filter represented in PARA</td>
<td>33</td>
</tr>
<tr>
<td>Figure 12</td>
<td>An example use of the Module Description Language</td>
<td>34</td>
</tr>
<tr>
<td>Figure 13</td>
<td>EDIF format</td>
<td>36</td>
</tr>
<tr>
<td>Figure 14</td>
<td>EDIF Format (cont.)</td>
<td>37</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Relevant attributes and types</td>
<td>43</td>
</tr>
<tr>
<td>Figure 16</td>
<td>A Simple Attribute Grammar</td>
<td>45</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Attribute Functions</td>
<td>45</td>
</tr>
<tr>
<td>Figure 18</td>
<td>A simple circuit diagram</td>
<td>47</td>
</tr>
<tr>
<td>Figure 19</td>
<td>A complex PARA specification</td>
<td>48</td>
</tr>
<tr>
<td>Figure 20</td>
<td>General format of PARA</td>
<td>48</td>
</tr>
<tr>
<td>Figure 21</td>
<td>The special cell that we are using for filter design</td>
<td>51</td>
</tr>
<tr>
<td>Figure 22</td>
<td>A simple dictionary</td>
<td>55</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Inherited Attribute</td>
<td>56</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>24</td>
<td>More FCST functions</td>
<td>58</td>
</tr>
<tr>
<td>25</td>
<td>Simple combining forms and its abstract floorplans</td>
<td>76</td>
</tr>
<tr>
<td>26</td>
<td>Geometric interpretation of transition state</td>
<td>77</td>
</tr>
<tr>
<td>27</td>
<td>Bidirectional combining forms</td>
<td>78</td>
</tr>
<tr>
<td>28</td>
<td>Abstract schematic diagram in recursive equation system</td>
<td>80</td>
</tr>
<tr>
<td>29</td>
<td>Schematic diagram of FAC</td>
<td>82</td>
</tr>
<tr>
<td>30</td>
<td>Internal detail of a simple circuit</td>
<td>85</td>
</tr>
<tr>
<td>31</td>
<td>Structural primitive hbox and vbox in Hydra</td>
<td>87</td>
</tr>
<tr>
<td>32</td>
<td>Connections and layout</td>
<td>88</td>
</tr>
<tr>
<td>33</td>
<td>A block of five cells</td>
<td>93</td>
</tr>
<tr>
<td>34</td>
<td>Block diagram of the filter architecture</td>
<td>94</td>
</tr>
<tr>
<td>35</td>
<td>Selected encoder implementation for systolic compiler</td>
<td>95</td>
</tr>
<tr>
<td>36</td>
<td>Single block encoder implementation</td>
<td>96</td>
</tr>
<tr>
<td>37</td>
<td>Nth order FIR filter realization</td>
<td>97</td>
</tr>
<tr>
<td>38</td>
<td>Implementation chosen for systolic compiler</td>
<td>98</td>
</tr>
<tr>
<td>39</td>
<td>Overview of Designer's Assistant</td>
<td>101</td>
</tr>
</tbody>
</table>
PART I
INTRODUCTORY MATERIAL
1.1 THE THESIS UNDER INVESTIGATION

The thesis under investigation was that: Pure lazy higher order functional programming languages are of value in various aspects of VLSI specification and design.

Investigation of this thesis is of importance for the following reasons: There have been many claims that pure lazy higher-order functional programming languages possess certain features that make them superior to conventional programming languages. (See for example, Wadler 1989). The growing use of pure lazy higher-order functional programming languages in industry and as teaching languages in academic institutions suggests that there is some credence in the claims being made for functional languages.

Very few complex applications of pure lazy higher-order functional programming languages exist.

It is not clear what practical constraints, if any, result from the fact that pure functional languages do not have any notion of global variables or updateable data structures.

One of the features of pure functional languages is that they are completely declarative. Proof of properties of programs is therefore considerably simpler (involving classical
predicate logic only) than in conventional procedural languages where time order is a factor. Correctness of programs is crucial in the construction of VLSI circuit specification translators.

Pure functional languages are well suited to the use of 'constructive' techniques in programming (i.e. the transformation of a provably correct program specification to efficient executable code). Constructive methods would appear to be of relevance to VLSI design.

The investigation, undertaken as part of the thesis work, involved the construction of a 'transformer component' of a VLSI Systolic Compiler.

This component, called FCST (for Functional Circuit Specification Transformer), is described in detail in this thesis. The author was wholly responsible for the following work related to FCST:

• Design and construction of the Filter Architecture Description Language (FADL) to PARA and Module Description Language (MDL) translator.

• Construction of the PARA to Electronic Design Interchange Format (EDIF) translator using an initial simple design provided by one of the supervisors.

• Provide assistance in design of the PARA language.

• Design and construction of the PARA constructing functions
for the entire system.

- Design and construction of the user-interface with help facilities.
- Design and construction of auxiliary functions for generating graphical display.
- Design and construction of the auxiliary functions for simulating the PARA language

The viability of FCST has been demonstrated, as discussed later, and the advantages/disadvantages of the lazy functional programming paradigm are reviewed with respect to this application.

Although the correctness of the translators in FCST has not been proved, the groundwork for such proof has been done because the translators are all implemented as declarative executable specifications.

FCST is a component of a system called the Windsor Systolic Compiler which has been constructed at the University of Windsor. We begin by giving a brief overview of this system.

### 1.2 THE WINDSOR SYSTOLIC COMPILER

The Windsor Systolic Compiler has been developed for design of filter applications. It consists of three major components: the Designer's Assistant, a Functional Circuit Specification Transformer and Layout Generator. Figure 1 is an overview of the Systolic Compiler.
The Designer's Assistant is the knowledge-based front-end of the compiler. It is an expert system with a window-based user-interface which provides a user-friendly environment.
As in all expert systems, the Designer's Assistant contains a knowledge base (in this case, the knowledge base contains the design domain of filter design), and an inference engine which is used to make inferences using the rules in the knowledge base. An Explanation facility and Abort/Restart facility are also provided for explanations of some problems encountered during the design process and to terminate or restart the current session. A more detailed description of the Designer's Assistant is given in Appendix 3.

The second component of the Systolic Compiler is the Functional Circuit Specification Transformer (FCST) which has been implemented as part of the work towards this thesis. FCST provides a hardware description language called PARA, which allows designers to express their designs. PARA is then translated to the Electronic Design Interchange Format (EDIF) at the netlist level. The data passed from the Designer's Assistant to FCST is a text file for filter realization.

The last component is the Layout Generator which generates layouts automatically by feeding the EDIF description file and other related files to CADENCE™. The Layout Generator is a custom designed environment within CADENCE™.

1.3 THESIS ORGANIZATION

This thesis is subdivided into three parts. The following paragraphs briefly describe their contents:
Part 1 consists of 4 chapters: This is the first Chapter. Chapter 2 gives an overview of the FUNCTIONAL CIRCUIT SPECIFICATION TRANSFORMER. Chapter 3 covers some example uses of FCST with some partial screen output. Chapter 4 provides a general description of the FUNCTIONAL CIRCUIT SPECIFICATION TRANSFORMER, its structure and components.

Part 2 consists of three chapters which provide more detailed descriptions of the FCST. Chapter 1 of part 2 covers, in detail, four languages: (i) the Filter Architecture Description Language (FADL), (ii) PARA, (iii) the Electronic Design Interchange Format (EDIF), and (iv) Module Description Language (MDL). Chapter 2 of part 2 covers in detail the structure of each component in FCST. Chapter 3 explains the user interface provided to FCST.

Part 3 discusses some related research, concluding comments, and future directions of FCST in two chapters. Chapter 1 of part 3 begins with a summary of related research, and then a comparison between that work and the work that has been done here. Chapter 2 concludes this thesis and ends with comments on future directions.

Finally, the Appendices: Appendix 1, SURVEY, describes research done elsewhere which is related to this thesis. Appendix 2, THE SPECIAL ARCHITECTURE, discusses briefly the special architecture that we are using in this thesis.
Appendix 3, covers in detail the Designer's Assistant. Appendix 4, *FCST END-USER INSTRUCTIONS*, explains how users interact with FCST in a design session. Appendix 5, *FCST SCRIPTS*, lists all programs developed as part of this work. Appendix 6, *FORMAL SYNTAX OF SPECIFICATIONS*, contains formal syntax of specification languages that are used in the thesis. Appendix 7, *EXAMPLES OF SPECIFICATION*, shows a complete example of an application using FCST.
Chapter 2
OUTLINE OF THE
FUNCTIONAL CIRCUIT
SPECIFICATION
TRANSFORMER : FCST

The Functional Circuit Specification Transformer – FCST takes the FADL specification from the Designer’s Assistant as its input. The overall system overview of FCST is illustrated in figure 2.

Figure 2 Overview of FCST
The components of FCST are:

- the FADL to PARA translator,
- the PARA specification language,
- the PARA to EDIF translator, and
- the PARA graphical display.

Each individual component is discussed in detail in subsequent chapters.

The main function of FCST is to take a filter design description written in the FADL language, translate this description into the PARA specification language, determine the correctness of the design by verifying its behavior, and finally, to translate this PARA specification into EDIF at the Netlist level and produce a list of ROM contents for the required filter.

The responsibility of the Designer’s Assistant is to obtain the necessary information from the designer and provide default values for parameters whenever it is possible. This information is then written to a text file in FADL format. The FADL to PARA translator, named FADLTOPARA, takes this FADL description, and translates it into the PARA specification language. PARA contains constructs which can be used for specifying the connectivity and components of combinatorial circuits. These constructs can be found in the PRI_lib and SEC_lib library.
This transformation is done using the Windsor Attribute Grammar Environment (W/AGE) [FK89, Fro89, Fro90, Fro91]. W/AGE is based on Knuth's [Knu68] Attribute Grammar concepts (AG) which were introduced in 1968.

W/AGE provides an environment in which one can build a translator as follows:

- specify the grammar of the input language,
- identify appropriate attributes,
- specify attribute types based on primitive types,
- specify an appropriate attribute grammar, and
- define attribute functions for constructing the output language description.

The PARA specification is an executable specification. By providing appropriate stimulus to the design specified in PARA, it will produce signal vectors which represent the behavior of the circuit. Since program or function written in functional programming language is derived from proof in most of the cases, its result is correct.

The signal vector produced can be stored in a text file, or displayed graphically. Figure 7 illustrates a screen dump of a sample output. This graph is described using the PostScript [War89, War88] page description language. A PostScript script can be viewed using Pageview - a utility which can interpret and display PostScript script.
The output of FCST is an EDIF description at the Netlist level, and a list of ROM contents. This EDIF description specifies the connectivity and cells that are used in the design. These could be parts from the CMOS3_lib library or user defined. This EDIF description and the ROM contents are then used for generating the layout of the design by running them through CADENCE™.
Chapter 3
EXAMPLE USE OF FCST

The intended use of FCST is to accept a FADL description produced from the Designer's Assistant, and translate this into the PARA specification language. The PARA specification is then combined with appropriate signal vectors for verifying its behavior. The result can be displayed graphically. Finally, an EDIF file at the Netlist level is produced by using another translator (PARATOEDIF), which translates the PARA specification into an EDIF description. The rest of this chapter demonstrates an interactive session with FCST.

3.1 INVOKING FCST

To start FCST, the user must invoke the Miranda interpreter and supply "FCST" as the current script. At this stage, one can enter two lower levels: the first lower level takes a FADL description and translates it to PARA specification and Module Description Language (MDL), which contains the contents of ROM table of the realized filter. Then by executing the PARA specification by providing appropriate signal vector, one can verify the behavior of the filter. The other lower level takes a PARA specification and produces an EDIF in the Netlist level.
The Miranda System
version 2.04 last revised 24 May 1989
Copyright Research Software Ltd, 1990

(700000 cells)
FCST, for help type /help
Miranda help

Type

1) "testpara" to translate FADL to PARA and run simulation
2) "paratoedif" to translate PARA specification to EDIF Netlist

Miranda testpara.

Figure 3 Help facility and invoking testpara

To enter one of the two levels, after invoking FCST, as shown in figure 3, type "testpara" for converting a FADL description to PARA specification and behavior simulation. To produce Netlist in EDIF from PARA specification, enter "paratoedif" in FCST.

3.2 HELP FACILITIES

Help facilities are available at all levels in the FCST environment, any may be invoked by typing the word "help". For instance, after typing the word "help" at the top level of FCST, the help messages will be appear as illustrated in figure 3.

3.3 TYPICAL EXAMPLE USE OF FCST
The first step in the use of FCST would usually be to translate a FADL description to an executable PARA specification. Alternatively, one could specify your own design using PARA. The former can be done by issuing 'fadltocpara' followed by the name of the file which contains the FADL description.

Figure 4 contains an example of a FADL description. Figure 5 illustrates a sample session using FCST.
The corresponding PARA specification will appear in another window as illustrated in figure 6.

There are seven functions at this level. The specific formats for entering arguments for each function are given in the help facility. For instance, after converting a FADL description to PARA, one can simulate the behavior of this specification by entering the word "executecircuit" followed by the name of any circuit written in the PARA specification, followed by
the name of the input and output file. The input file should contain appropriate signal test vectors.

Figure 7 shows a sample output after issuing the sim command. The rest of the commands serve different purposes, which are necessary in testing PARA specifications. These commands have to be followed in sequence from top to bottom in a complete session with FCST. For instance, after converting a
FADL description to a PARA specification, the second and third command should be used for entering user-defined names for I/O pads. Then, you supply appropriate data, using the fourth command. And finally, use the sixth/seventh command to print the waveform of the circuit after simulation (as mentioned above). The difference between the last two commands is that in the last case, a hard copy is produced.

The second level of FCST produces a netlist in EDIF format. This level has a similar help facility to the other levels. To obtain a netlist of a PARA specification, enter "netlist" then the name of the files which contain the corresponding I/O pad name and PARA specification, as shown in figure 8.
this case, the intermediate file, "IF.m", contains the PARA specification for a circuit called chip_2 as shown earlier in figure 6. The last function, "netlistout" produces and stores in a file a netlist in EDIF by taking an extra argument.

3.4 CONCLUDING COMMENTS FOR THIS CHAPTER

The purpose of this chapter was to give a general description of the user-interface and the environment in which the user would be working. The help facility provides simple but sufficient instructions of showing how to use each command.
Chapter 4
OVERVIEW OF TOOLS USED TO CONSTRUCT FCST

4.1 THE ENTIRE SYSTEM IS IN A SINGLE PROGRAMMING LANGUAGE

One feature of FCST is that the whole system is written in a single programming language, Miranda™. Miranda™ is a pure, lazy functional programming language. Functional programming languages have several advantages over conventional procedural languages such as Fortran, C, Ada, etc., as discussed in Wadler [Wad89]. Functional languages also facilitate several modern programming techniques such as program from proof and synthesis through transformation during the construction of application programs.

The translators in FCST are constructed using a functional programming environment called W/AGE that has been implemented as an extension to the Miranda standard environment. Both translators can be regarded as interpreters, because the syntactic and semantic analysis are carried out during the parsing phase. W/AGE is based on Attribute Grammars which have been used for describing the syntax and semantics of programming languages [Wat79], in compiler generators [Far82], as the basis of syntax directed editors [RT88], and in construction of translators [Fon90].
4.1.1 The PARA Specification Language

The PARA specification language is defined as a set of Higher Order Functions (HOFs) which is added to the Miranda standard environment; therefore, the PARA language may also be regarded as an "extension" to the Miranda standard environment. This is the reason why PARA is executable. As a result there is no need to write a compiler for compiling PARA specifications into executable code.

Specifying circuits using PARA is simple and straightforward. Designers can treat each component as a black box. The para function puts components in parallel to form a larger circuit. The function wire connects the output of a previous stage with the inputs of the next. Refer to figure 6 for an example PARA specification. These two construction functions plus another function, con, are devices that can be used to specify any uni-directional systolic array architectures. The function con is related to function composition. It works like a pipeline. con appears between each pair of wires and para and between two slices*.

Another advantage is that PARA makes explicit both the semantic and geometric properties of a circuit.

Graphical representations of circuit are very helpful in the debugging process. PARA offers a unique way of specifying

* Refer to the BNF notation of PARA language
circuits and this specification is similar to its gate level circuit diagram which is commonly used by digital designers. By comparing the PARA specification and the circuit diagram, one can easily verify the correctness of the PARA specification. The behavior of the circuit design can be simulated by providing appropriate signal vectors, i.e., streams of signals over time.

4.1.2 The Circuit Specification Translators

FCST contains two translators. One translates FADL descriptions to executable PARA specifications, the other translates PARA specifications to EDIF format. Both translators are implemented in the Windsor Attribute Grammars Environment (W/AGE), developed at the University of Windsor [Fro91]. W/AGE facilitates the design and construction of executable specifications of attribute grammars. W/AGE is also implemented using the functional language Miranda, which facilitates a simple and efficient implementation of attribute evaluations and its type system allows a simple and natural means for defining evaluation functions.

4.2 OVERVIEW OF LAZY FUNCTIONAL PROGRAMMING LANGUAGES

Functional programming languages such as Miranda offer the following advantages:
Functional languages are declarative in nature; therefore, the order of the program statements is irrelevant to program execution.

Their characteristic features of Strong typing, abstraction and modularization help programmers to control the complexity of their task by facilitating a black box approach to programming, thereby simplifying precise specification and allowing programmers to concentrate on the essential properties of a class of objects rather than on specific instances.

Higher Order Functions and partial application enable many related functions to be realized as specific instances of one generic function, resulting in shorter programs, improved readability and re-use of proofs of correctness.

Partial application enables functions to be defined as compositions of other functions with no need to make parameters explicit. The result is a more natural and compact code which is easier to manipulate for the purpose of proof of correctness and transformation.

4.3 OVERVIEW OF MIRANDA

When installed in the usual way, Miranda is invoked by the UNIX\(^1\) command 'mira'. Programs written in Miranda are called scripts and contain function definitions. The set of names

---

\(^{1}\) UNIX is a trademark of AT&T Bell Laboratories.
in scope at any time are those of the current script, together with the names of the standard environment, which are always in scope [Tur85] together with names of functions in any 'included' files. The Miranda system is interactive and runs under UNIX as a self contained subsystem. The basic action is to evaluate expressions in the environment established by the current script.

4.3.1 The Miranda Programming Environment

Miranda is a pure functional language, with no imperative features of any kind. A program is a collection of definitions, of functions and other data objects, written in the form of recursion equations. The order in which the definitions are written is not significant in general. An equation can have several alternative right hand sides distinguished by "guards" which are written on the right following a comma. For example the greatest common divisor can be written:

\[
gcd a b = gcd (a - b) b, a > b \\
= gcd (a - b) b, a < b \\
= a, a = b
\]

Figure 9 An example function — gcd

Miranda has a non-strict (i.e. lazy) semantics. There are two kinds of functional programming languages — strict and non-strict. Strict functional languages evaluate all the
arguments before invoking the function, whereas non-strict functional programming languages do not do any evaluation until it is needed. An example use of lazy evaluation is to define an infinite list of numbers as [1..].

Miranda is a fully higher order language – functions are first class citizens and can be both passed as parameters and returned as results. Function application is left associative, so when we write “f x y” it is parsed as “(f x) y”, meaning that the result of applying f to x is a function, which is then applied to y.

4.3.2 The Type System

Miranda is strongly-typed. It is not necessary to declare the types of functions and other objects. The types are deduced by the compiler from the data in the script.

The primitive types in Miranda are num, bool and char which stand for number, boolean (comprising ‘true’ and ‘false’) and character respectively. More complex types can be defined using tuples, represented by a pair of parentheses, and lists, represented by []). The type system serves two purposes. Firstly it permits a significant proportion of programming errors to be detected at compile time. Secondly, it provides a foundation on which to erect a system of user-defined types. There are three mechanisms available in Miranda for user-defined types. They are (i) type synonyms, (ii) algebraic
types, (iii) abstract data types. We will only discuss the first two of these. Type synonyms allow the user to introduce a name for an already existing type (we use \`\texttt{\textasciitilde\textasciitilde}'\ for these definitions, to distinguish them from ordinary value definitions) thus

\texttt{string \texttt{\textasciitilde\textasciitilde} [char]}

makes \texttt{string} a synonym for the type \[\texttt{[char]}\]. We can also introduce names for operators over type; this is done by using generic type variables as formal parameters, as in

\texttt{f * \texttt{\textasciitilde\textasciitilde} \texttt{\textasciitilde\textasciitilde} \rightarrow \texttt{\textasciitilde\textasciitilde} \rightarrow \texttt{\textasciitilde\textasciitilde}}

So now \texttt{f string} will mean \[\texttt{[char]}\rightarrow\texttt{[char]}\rightarrow\texttt{[char]}\], for example.

The basic method of introducing a new concrete data type, as in a number of other languages, is to declare a free algebra. In Miranda this is done by an equation using the symbol \texttt{\textasciitilde\textasciitilde\textasciitilde\textasciitilde}'. For example:

\texttt{tree \texttt{\textasciitilde\textasciitilde} Niltree \mid Node num tree tree}

This introduces three new identifiers, \texttt{tree} which is a new type, \texttt{Niltree} which is a tree-constructor of no arguments (i.e. an atomic tree), and \texttt{Node} which is a tree-constructor of type: \texttt{num->tree->tree->tree}. We call it a free algebra, because there are no associated laws, such as a law equating a tree with its mirror image. Two trees are equal if and
only if they are constructed in exactly the same way.

The subject of an algebraic type definition can be made a family of types, rather than just a single type, by introducing an appropriate number of generic type variables as formal parameters of the definition. So we can revise our earlier definition of trees to allow trees with labels of arbitrary type, built with polymorphic constructors, by writing

\[
\text{tree} * ::= \text{NilTree} \mid \text{Node} * (\text{tree} *) (\text{tree} *)
\]

This introduces an infinite family of types, of which 'tree int', 'tree bool', 'tree(num->num)', are members.

### 4.4 OVERVIEW OF W/AGE

W/AGE [Fro91] is based on Attribute Grammars as introduced by Knuth in 1968 [Knu68]. An Attribute Grammar may be used to formalize both the context-free and context-sensitive aspects of a subject language. An Attribute Grammar is basically a context-free grammar augmented with attributes, evaluation rules and conditions that enable the non-context-free aspects to be specified [Pag81].

The advantage of W/AGE is that it offers a mechanical way of constructing application programs as executable interpreters. The construction of programs as executable interpreters facilitates the separation of the specification of the grammar from the attribute evaluation functions. The attribute grammar of any problem can be specified with minimal concern
for efficiency and complexity of the interpreter, and these concerns can be handled in later steps by using behavior preserving transformations. The resulting interpreter is fully lazy, i.e., it evaluates only when necessary. Finally, evaluation of attribute functions is carried out during parsing by integrating the syntactic and the semantic analysis of the input language.

4.5 THE WINDOW SYSTEM / HARDWARE PLATFORM

FCST has been tested in the OpenWindows© System. This window environment provides easy access to view a PostScript script. OpenWindows is built on top of the X-Window System*. Therefore, users can access FCST in a networking system. Although OpenWindows is running in the Sun-4 architecture, FCST is not restricted to Sun-4. This is because X-Window System is device-independent and network transparent. FCST can also be executed in other window environments, such as SunView, with minor modifications.

4.6 CONCLUDING COMMENTS FOR THIS CHAPTER

The purpose of this chapter was to give a brief description of the tools and techniques that we have used to implement FCST. A detailed description of FCST is now given in PART II.

* X-Window System is a trademark of the Massachusetts Institute of Technology
PART II
MORE DETAILED
DESCRIPTION OF FCST
Chapter 1
THE SPECIFICATION LANGUAGES & DISPLAYS RELATED TO FCST

1.1 THE FIR FILTER SPECIFICATION LANGUAGE: FADL

The FADL specification produced by the Designer's Assistant is a text file which contains information for filter realization. The BNF notation of the relevant part of FADL description is shown in Appendix 6.

For simplicity, italic font is used to show the basic type of the terminal symbols. For instance, \textit{num} stands for a valid number, whereas \texttt{[char]} is used to represent a list of character strings. \textbf{Boldface} is used to represent constant or uninterpreted terminals.

A FADL description consists of two parts: the first part is the Module constructs which describe each module in a chip. The second part describes the topology of the interconnected modules and all the master cell names or map blocks to be used in the design. However, the second part of the FADL description is irrelevant to this part of the project.

Figure 10 illustrates a FADL description of a filter design. This filter contains 3 chips, namely chip_1, chip_2, and chip_3. Chip_1 is an encoder, which converts the input number to its residue form. The moduli selected are 32 and 31. And finally, the number of input bits for this design
Entity chip_1
Module mod_1 [ Purpose encoder,
    Moduli [32,31,29,27], Input_bits 8]
Floorplan [ Bias column, Array [[A]] ]
Map [ [A, mod_1] ]

Entity chip_2
Module mod_1 [ Purpose ipsp,
    Moduli [32,31,29,27], Coefficients
    [703,256,-152,36,36,-152,256] ]
Floorplan [ Bias column, Array [[A]] ]
Map [ [A, mod_1] ]

Entity chip_3
Module mod_1 [ Purpose scalar,
    Moduli [32,31,29,27], Output_bits 8 ]
Floorplan [ Bias column, Array [[A]] ]
Map [ [A, mod_1] ]

Figure 10 An Example of FADL Description

is 8, which is capable of handling the largest dynamic range.
Chip_2 is a different chip which works as an “Inner Product
Step Processor” or “ipsp”. It is using the coefficients 703,
256, -152, and 36. And finally, the last chip, chip_3, is
a scalar. And the number of output bits is 8. This step
converts a residue number back to the original frequency.

1.2 THE EXECUTABLE CIRCUIT SPECIFICATION
LANGUAGE : PARA

PARA is a hardware description language that allows hardware
designers to construct executable specifications for VLSI
circuits. It is implemented as a set of Higher Order Functions
(HOFs) added to the Miranda standard environment.
PARA consists of two sets of library functions: a set of primary functions and some auxiliary functions. The primary functions are basic constructs that facilitate the specification of interconnections and components of a circuit. They also provide the means for simulating the design and generating results in textual as well as graphical representations. Auxiliary functions define the behavioral semantics of standard cells. They contain all cells from the CMC library distribution.

The BNF syntax of PARA is illustrated in Appendix 6. A specification in FxA consists of a chip name, followed by an equal sign (=), then followed by one or more slices, and terminated with the semicolon symbol. Slices are separated by the terminal symbol `$con'`. Each slice contains a wiring part and a para part. The wiring part specifies one or more lists of incoming wire positions. For each list of incoming wire positions, there exists one corresponding component in the para portion. Figure 11 shows a filter design specified using PARA.

1.3 GRAPHIC DISPLAY OF THE SIMULATED CIRCUIT

PARA has the capability of producing graphical output of the simulated output of the design. Such output is the actual behavior of the circuit specified. The results generated are a series of instructions written in PostScript [War88, War89] which can be displayed on a screen using PageView, a graphic...
utility for viewing PostScript. Figure 7 shows a screen dump of an example output of a filter design.

1.4 THE MODULE DESCRIPTION LANGUAGE

The Module Description Language (MDL) is used for generating the layout of filter design. Files containing MDL descriptions are generated when the translation of FADL to PARA takes place. The default name of the file is "IFMDL.m". The MDL description contains information such as the contents of all the ROMs in a chip. An example specification in MDL is given in figure 12. The name of this chip is 'chip_1' and
it serves as a FIR filter. The moduli used in the design is 32 and 31. The coefficient used in this chip is 703 which is used to generate the values of content of the ROM as shown starting from line 4. The way that the ROM contents are created is discussed in detailed by [JBMF89, Tah88, Sar90].

This description is comprised of the name and purpose of the chip, the moduli used in the design, followed a list of the
contents of the ROM. The BNF notation for MDL is defined in Appendix 6.

1.5 THE NETLIST LANGUAGE: EDIF

1.5.1 EDIF 2.00

EDIF was designed to meet the needs of a wide range of users — from those who wish to transfer mask artwork data in its final form, to those who desire to transfer a behavioral description of a microcomputer. This interchange format is an unambiguous description of data.

EDIF descriptions can be specified in three different levels of abstraction. Level 0 is the basic level in which only literal constants are permitted. Level 1 is the same as Level 0 with the addition of parameters and expressions. And finally, Level 2, which is the level that we use, is the same as Level 1 with the addition of control-flow constructs.

EDIF is based on a Lisp-like syntax which was chosen for the ease of parsing and provision of a uniform method of extension of the format. A construct begins with an opening parenthesis and a keyword; this is followed by a list of items ending with a closing parenthesis. Those item may be other constructs which build a nested structure, or they may be tokens consisting of data items, such as integer tokens or string tokens, or EDIF identifiers.
This new EDIF\textsuperscript{1} has the format given in figure 13 and 14:

The interpretation of all the constructs in the above EDIF description [Edi87] are as follows: test\_circ\_edif is the name of the EDIF. editVersion indicates the version of the EDIF file. At present, this comprises the following versions: 1 0

\footnote{\textsuperscript{1}From now on, we will refer the new version simply by EDIF.}
Figure 14 EDIF Format (cont.)

0, 1 1 0, and 2 0 0. Presently SDA is using version 2 (2 0 0). edifLevel tells the reader the greatest EDIF level that will be found when process the file in library. The integer token will take the value zero, one or two. A level 0 structure will contain only the most primitive of the EDIF constructs.
KeywordMap isolates all keyword mapping in one area of the EDIF file. keywordLevel is used to specify the level of complexity that keyword mappings will have in the EDIF file. The integer token must have the value 0, 1, 2, or 3 which correspond to no keyword mappings, abbreviation only, keyword macros with no iterative parameter binding, and keyword macros with iterative parameter binding, respectively. status provides the information needed to trace back to the source of each piece of data, the human author or organization which owns the data, the program name, and the software revision used for creation of specific parts of an EDIF file. The timeStamp identifies the time in Universal Time Coordinated when the data was created or last modified. The example above indicates the EDIF file was last modified December 6, 1990 at 21:11:45.

External construct declares cells that exist outside of the current edif description so that those cells can be used in the current edif description. Thus any cells used in the edif description must be declared within this construct. The library that contains these cells must be specified. The technology construct contains the information related to the intended implementation of the library. The description applies to every component of the library. The technology block is required for every library. It includes scaling information, definitions of the figure groups and the default figure group attributes, and simulation
information. *numberDefinition* is required to ensure that scaling information is available before any values which may need to be scaled.

The *cell* construct is used to define a cell that may be instantiated later in another cell to build a design hierarchy. It holds all the information that logically pertains to a specific cell. *celltype* is an attribute of a cell which describes the use of the cell. The allowed cell types are GENERIC, TIE, and RIPPER. The cell type GENERIC indicates that the cell is a generic cell and covers all uses of cell other than TIE and RIPPER.

*ViewType* is used to specify a data representation, or perspective of a cell. *ViewType* is used to specify the type of a view. The viewType indicates the intended use of the view. The view NETLIST describes a netlist. The *interface* defines objects which can been seen. In this case, the objects are ports. *port* is the basic means of communicating signal information between cells and is the subject of all connectivity statements. *direction* construct is used to specify the direction of signal information flow at a port. The permitted values are input and output.

The *rename* construct names an object in the Design Framework database using the alternate time. The *library* construct
contains cell view definitions. EDIFIN' generates the views defined in a library construct; in this example the directory of the chips is CHIP_dir. libraryRef contains a reference to the name of a library.

1.6 CONCLUDING COMMENTS FOR THIS CHAPTER

The purpose of this chapter was to give a detailed description of the languages which FCST manipulates. The input and output of the system, including text and graphical input/output, and their complete syntax were also described.

* This program compiles the EDIF Netlist.
Chapter 2
COMPONENTS OF FCST

This chapter explains how each component is built in FCST. FCST is divided into two levels: the purpose of the first level is to provide a facility which allows users converting a FADL file to PARA specification, to test and verify this PARA specification. Users can enter their own PARA specifications as described later in this chapter. The second level allows the user to convert a PARA specification to EDIF at the netlist level.

FCST consists of five libraries: PRI_lib.m contains all of the functions for testing and verifying PARA specification. cell.m and SEC_lib.m contain all functions for simulating the special architecture [Tah88] and the standard cells, respectively. coms3lib.m provides information such as number of I/O pad for each standard cell which appears in the SEC_lib.m library. And finally, the W/AGE library, ambi_wage.m, provides all the functions which are used for constructing both translators: FADLTO PARA and PARATOEDIF. The functions in ambi_wage.m provide an easy way of constructing interpreters in a declarative manner with modular structure that can be modified easily.

2.1 FADL TO PARA TRANSLATOR

The translator, FADLTO PARA, was constructed as follows:
• We defined a grammar for the input language,
• We defined all relevant attributes,
• We defined all base types,
• We specified an Attribute Grammar, and
• We defined all attribute functions.

2.1.1 Definition Of a Grammar For The Input Language

The first step was to define a grammar for the input language. The BNF syntax of FADL has been given in the previous chapter. For simplicity, we will use a small portion of coding from the translator for explanation. The complete listing is attached at Appendix 6.

2.1.2 Identification Of All Relevant Attributes And Based Types

The next task, steps 2 and 3, was to identify the attributes that we will be using. We identified these attributes, as shown in figure 15, that are relevant to FADL:

These attributes are constructed from the basic types available in Miranda. They are num, char and bool which represent number, character and boolean respectively. We can use list ([]), and pair (()) to construct more complex type structures. For instance, the attribute PARA has type of list of string (list of character).

The function name, available in the W/AGE library, is used to associate each attribute with objects of type attribute.
attributes ::= CHIPNAME [char]
   | PARA [[char]]
   | PROPERTY property
   | MODULI [num]
   | PURPOSE [char]
   | DUMMY [char]
   | NUMVAL num

property ::= COEFFICIENTS [num]
   | INPUT_BITS num
   | OUTPUT_BITS num
   | COEFFVAL num

name (CHIPNAME x) = chipname
name (PARA x) = para
name (NUMVAL x) = numval
name (PROPERTY x) = property
name (MODULI x) = moduli
name (PURPOSE x) = purpose
name (DUMMY x) = dummy

Figure 15 Relevant attributes and types

2.1.3 Specification Of An Attribute Grammar

The next step was to specify the attribute grammar for the
input language. We begin by specifying a dictionary for the terminal symbols. A dictionary consists of all the valid tokens allowed in the input language. W/AGE generates Miranda code defining an interpreter for each category of terminal symbol using the higher order function, \texttt{mkint}. We then specify the interpreters for the non-basic syntactic categories using the higher function \texttt{rstc'}, \texttt{excl_orelse}, and \texttt{orelse}.

This translator does not need a dictionary because it takes text directly (synthesis attribute) from the input language. The retrieved text are verified by this translator. We will show, in the next translator, a dictionary.

Now, we needed to provide a skeleton attribute grammar by providing appropriate names for attribute synthesis functions. The result is a set of interpreters, one for each of the non-terminals in the grammar of the input language. The attribute grammar is no more than a grammar which has been augmented with semantic rules indicating how attributes of some syntactic constructs can be computed from attributes of other related syntactic constructs. Figure 16 show a simple attribute grammar which can be converted to this translator.

The first line of text constitutes a context-free grammar for the input language expression. The notation used is a variant of Backus-Naur Form in which comma (,) is a terminal symbol. This grammar involves only synthesized attributes. The next
properties = purpose, moduli, aproperty

MODULI ↑ properties = MODULI ↑ moduli

PROPERTY ↑ properties = PROPERTY ↑ aproperty

Figure 16 A Simple Attribute Grammar

A few lines are semantic rules. Each semantic rule indicates how the value of an attribute associated with the syntactic construct on the left hand side of a production is obtained from attributes of syntactic construct on the right hand side. An upward arrow signifies that the attribute is synthesized. For instance, MODULI ↑ properties should be read as the MODULI attribute that is synthesized for the expression.

2.1.4 Definition Of All Attribute Functions

The final task was to define all of the attribute functions which are defined in the semantic rules of the attribute grammar. Attribute functions take inherited/synthesized attributes as arguments as shown in figure 17.

get_fadl [CHIPNAME n, MODULI m, PROPERTY (INPUT_BITS i)]

= encoderspec n m i

get_fadl [CHIPNAME n, MODULI m, PROPERTY (COEFFICIENT c)]

= filterspec n m c

get_fadl [CHIPNAME n, MODULI m, PROPERTY (OUTPUT_BITS o)]

= scalarspec n m o

Figure 17 Attribute Functions
All these functions are evaluated after a successful parse of the input has been identified. This is an advantage of integrating the syntactic and semantic analysis of the linguistic objects in the input language.

2.2 PARA SPECIFICATION "EXECUTOR"

2.2.1 The Behavioral Model

The idea behind this behavioral model is that we are forming a larger computing network out of primitive ones. Each node represents a process (signal transformer). Each wire is an infinite list of values which represent the "history" of computation on this net. The values produced from each node represent the history of computation up to the current point in time (assuming a discrete model of time). Each sequence of values represents the state from some starting time at $t_0$ up to the current state.

2.2.2 Primitive Functions

PRI_lib.m contains all the main functions for specifying circuit specification, executing the PARA specification and plotting the result graphically. A complete listing of both libraries is in Appendix 5. The following is a brief description of each function/construct.

wire is used for describing the connectivity of a circuit. The wire construct specifies the position of the incoming wire. Incoming wire are counted from the bottom up to the top in a logic diagram. The para construct is used for specifying
the components of a circuit. Each component specified, using `para`, corresponds to a set of incoming wires specified in the `wire` construct. `wire` and `para` are used in a pair. These two constructs are glued together by the function `con`. `con` is a function composition which applies the output from a function to another function. Figure 18 illustrates a slice of a simple circuit and its corresponding circuit diagram.

\[
ha = (wire [[1, 2] [1, 2]])
\]

\[
$con
\]

\[
(para [and, xor]);
\]

Figure 18 A simple circuit diagram

The dollar sign ($) is an infix operator in Miranda. More slices can be glued together the similar way. For instance, `test_circ` is made up of a circuit defined previously, and its corresponding diagram is shown in figure 19.
test\_circ = (wire [1, 2], [3, 4])
$con
(para \text{nand, ha})
$con
(wire [1, 2])
$con
(para \text{ha});

Figure 19 A complex PARA specification

In general, a PARA specification has the following format:

chipname = (wire [p1, p2, ...], [p1, p2, ...], ...)
$con
(para \text{c1, c1, ...}, [c1, c1, ...], ...)
$con
...
$con
(wire [p1, p2, ...], [p1, p2, ...], ...)
$con
(para \text{c1, c1, ...}, [c1, c1, ...], ...);

Figure 20 General format of PARA
where test_circ is a name which can be used to refer to the entire circuit. More complex circuit can be made up of smaller user defined subcircuits. This would allow designers concentrate on a higher level abstraction and re-use these models.

The function, execute circuit, is used to execute a circuit specification by applying appropriate signal vector. execute circuit takes three arguments: a PARA specification, an input file and output file name. The input file should contain appropriate data which is input signals over a period of time. The output file would contain the results produced by applying the input signals to the circuit specification. The function sim takes the above output file, a circuit specification and some user-supplied I/O pads name, and produce a waveform of the result that is similar to figure 7 in the previous chapter. The user can plot this waveform using another function called plotsim.

The functions input_pad_name and output_pad_name are the means of entering I/O pad name defined by the user. The number of names of I/O pad for each chip should correspond to the number of I/O pads in the circuit. The last function input_signal contains a signal over time. The proper format for each of the functions described here appears in the help facility. Therefore, designers can concentrate on their design rather
on how to use the system.

2.2.3 Auxiliary Functions

The library SEC_lib.m contains a set of standard cells which are commonly used for building more complex cells. These cells can be stored in this library for later use. Any function in this library serves as a behavior model for a particular cell. Any cell in this library that is being used should include the entire library in the current Miranda environment.

Cells are defined to accept particular type of input data, in this case, only vector signals which have the following type: [[signal]] -> [[signal]]. The interpretation of this type is that a cell will take some vector signals and return as output some signal vector over time. Here, time is implicitly stated.

coms3lib.m, which is closely related to SEC_lib.m, provides information such as that the I/O pad of each cell appears in the SEC_lib.m. Each cell in the SEC_lib.m should have a corresponding entry in the coms3lib.m. The information for these entries is used when generating the EDIF description.

cell.m contains a set of functions which consists of a user-defined cell and a special architecture which are used for producing high performance VLSI filters. Figure 21 shows the architecture of the cell that we are using for designing high
speed filter. A behavior model of this cell is specified using PARA which can be found in Appendix 5.

This cell is made up of three components: latches, steering switches, a ROM and an inverter. latch and inverter are available in the SEC_lib.m library. The other components are in the cell.m library. Rom_cr is used to create a five bit ROM by taking, as its arguments, a moduli, coefficient and the displacement of the rom. The function lookup searches and retrieves the rom content while decoder is used to decode the address of a rom. The function ssw which stands for steering switch will behave as a steering switch by taking a control signal and an incoming signal and distribute the incoming signal accordingly.
The function `make_cell` takes a ROM table and signal vector as its arguments. For every instance of a `make_cell`, one can supply different rom contents. This allows defining different cells as expected for designing filters. To cascade five cells, the function `make_sys_cell` is used. `make_sys_cell` does this by arranging one cell after another with appropriate displacement value when creating the rom contents. To use this function, only a moduli and coefficient are needed. This will result in a five cascaded cell with the right contents as explained in [Tah88].

There are also other functions available for converting signals from one representation to another. However, these functions will not be needed for testing filters generated from a FADL file. All the source listings for the above mentioned libraries are attached in Appendix 5.

### 2.3 PARA OUTPUT TO GRAPHICAL DISPLAY TRANSLATOR

The waveform produced in FCST is implemented using a page description language known as PostScript®. PostScript provides a large selection of graphical operators that allows it to precisely describe a desired page. These operators allow the control of placement of graphical objects and manipulations of graphical objects such as rotation, scaling, clipping etc.

---

PostScript is the trademark of Adobe Systems Incorporated
PostScript manipulates data directly by using a special entity called a stack. A stack is a piece of memory set aside for data which is to be immediately used by PostScript. This memory area is organized in such a way that the last item put in is the first item available to be removed. This type of data structure is referred to as last in, first out or LIFO stack.

When a word in the source file is found, the PostScript interpreter searches its internal dictionaries to see if that word is an operator name. The interpreter carries out whatever instructions are associated with that name and then continues on to the next word in the source file.

Waveforms are produced by using a pre-written PostScript program which will only recognise signals produced from the function `sim`. `sim` simply converts the output, produced from `executecircuit`, to an array structure and appends it to the end of this postscript program. The complete listing of the postscript program can be found in Appendix 5.

The function of `Title` is to display the name of the chip on the output. The function of `Display` is to draw a signal over time. `PinName` displays the name of the I/O pad on the first column followed by its corresponding signal. The last function `Clock` draws a timing scale on the bottom of the page.

The end of the program, referring to Appendix 5, shows how
signal data are actually stored and represented.

2.4 PARA TO EDIF TRANSLATOR

This translator, PARATOEDIF, is implemented the same way as the translator, FADLTOPARA, discussed previously. However, this translator consists of several dictionaries and is using both inherited and synthesized attributes.

2.4.1 General Discussion Of Attribute Types

To begin with the discussion of this translator, let’s explain the interpretation of some of the complex attributes which we will using later. The attributes INSTABLE have the type of $[(gcname, num)]^*$, where gcname, in turn has the type of [char]. INSTABLE is interpreted as a list of pairs; each pair consists of a general circuit name associated with a number which represents the number of occurrence of the same name. The attribute INSTDATA is a list of cname. Each cname is a pair; the first element is gcname and second element is num.

The attribute LCSPEC is a list of triples; the first element is cname which has been defined previously; the second and the last element are the name of input and output pad respectively. This attribute maintains a list of components which have appeared in circuit specification.

The attribute LNET has the type of $[(num, (wire_label, wire_label))]$; wire_label has the type of OUTPORT, INPORT

$^*$ refer to Appendix 5 for the detailed listing
and QUALITY. OUTPORT and INPORT have the same type which is [char], whereas QUALITY has the type of cname and pinname. Pinname has the type of [char].

The attribute NETNUM has the type of num. The attribute keeps track of the total number of connections which a component has. The attribute INLWIRE and OUTLWIRE are a list of wire_label. INLWIRE and OUTLWIRE are used to store the information of the input and output pads.

2.4.2 Inherited Attributes Used In Translator

Figure 22 shows a dictionary which is used in this translator. In this case, both lib_gc_name_list and wpos_list contain a list of possible strings which may appear in the input language.

```
libgcname = mkint lib_gc_name_list
wpos = mkint wpos_list
```

Figure 22 A simple dictionary

Figure 23 shows how inherited attributed is interpreted in the attribute grammar specification. Down arrows indicate inherited attributes. In this case, the fifth rule in this attribute grammar states that the attribute INLWIRE is inherited down to the expression wiring.
\texttt{slice ::= wiring \$con para}

\texttt{slice \uparrow INSTABLE = para \uparrow INSTABLE}

\texttt{slice \uparrow INSTDATA = make\_inst\_data(para \uparrow LCSPEC)}

\texttt{slice \uparrow LNET = make\_nets(slice \downarrow NETNUM,}
\texttt{\hspace{1cm} slice \downarrow OUTLLWIRE,}
\texttt{\hspace{1cm} para \uparrow LCSPEC)}

\texttt{slice \uparrow OUTLWIRE = get\_out\_wires(wiring \uparrow LCSPEC)}

\texttt{wiring \downarrow INLWIRE = slice \downarrow INLWIRE}

\texttt{para \downarrow INSTABLE = slice \downarrow INSTABLE}

\begin{figure}[h]
\begin{center}
\begin{tabular}{|c|}
\hline
\texttt{Figure 23 Inherited Attribute} \\
\hline
\end{tabular}
\end{center}
\end{figure}

\section*{2.5 TESTING THE VIABILITY OF FCST}

FCST has been tested in two ways:

(1) By generating PARA and EDIF output for a FIR filter as shown in Appendix 7. Appendix 7 section 1 contains a low pass filter in FADL. Appendix 7 section 2 contains a PARA specification for that filter generated by FCST. Appendix 7 section 3 is the EDIF obtained from FCST by translating the first part of the PARA specification given in section 2. Finally, the last section, in the same Appendix, is the MDL description for the first part of the FADL specification given in section 1. The PARA specification was tested through execution. And the EDIF was examined visually and appears to be correct, and
(2) By use on a smaller example, which was fed into CADENCE, and which would appear to have been correct. A VLSI layout was generated.

2.6 CONCLUDING COMMENTS FOR THIS CHAPTER

The purpose of this chapter was to give a detailed description of each component in FCST. The PARA specification language, both language translators and the related libraries were also discussed.
Chapter 3

Outline of the End-User Interface of FCST

Chapter 3 of Part I gave an overview of how users can start up the FCST system. The following shows in more detail the steps that are necessary to run FCST.

To use FCST in the OpenWindow System, the user should start the OpenWindow System before each session begins. To start FCST, make sure that a copy of the entire system, which consists of all libraries and translators, is in the current directory. Enter `mira FCST` at a command prompt, which begins a session right away. Figure 24 shows all the windows that might appear in a FCST session.

Figure 24 More FCST functions
The user-interface is straight-forward to use. Once the user is in FCST, help facilities are available. FCST provides all the information that a user needs for running a session. Basically, at the top level of FCST a user can enter one of two stages:

1. Convert a FADL to PARA, test the spec. and generate graphical output.

2. Convert PARA to EDIF at the netlist level.

The format for entering arguments for each function in the current script can be found in the help facility. The format is usually made up of a function name followed by one or more arguments. These functions have been discussed earlier. The list of argument (refer to figure 5) is usually composed of one or more file names, or a list of I/O pad names, or a list of signal vectors, or the combination of all of the above. The sequence of entering functions is designed in a logical way to help novice users.

However, certain items need to be entered only once; for example, for the same design specification, users can use the same set of signal vector or I/O pad name. The only thing that has to be changed is the PARA specification.

The purpose of this chapter was to give a brief description of the end-user interfaces. Chapter 3 of PART I provided some information in this aspect also.
PART III
RELATED WORK AND
CONCLUDING COMMENTS
1.1 SUMMARY OF SURVEY

In general, there are three different schools of thought in the use of functional languages in VLSI design (a comprehensive survey is attached in Appendix 1): The first, exemplified by Sheeran [She83, She84, She85a, She85c, She85b, She88] uses functions and combining forms with algebraic properties for describing both semantic and geometric interpretations of a circuit. Circuit descriptions are made from a set of primitive functions and combining forms, and manipulated by designers while testing its functionality. The meaning function is then used to generate a set of algebraic laws through transformations. These algebraic laws show the properties of the language and are used for refining circuits.

The alternative approach, for example [Joh82, Joh83, Joh84a, Joh84b], uses a set of recursion equations, as initial specification. The initial specification is first translated into iterative form which corresponds to the designer’s circuit. A realization language uses recursion to state the connectivity of a circuit’s components. And finally, the last step is to reduce external components in the combinational system using folding which can be easily implemented using Daisy.
The last approach [O'D87, O'D88] – Hydra, known as recursive circuit specification, uses both recursion equations and combining forms which are similar to Johnson’s and Sheeran’s approaches.

1.2 RELATIONSHIP TO THIS WORK

The above approaches are all functional. The first approach is a function-level approach. It is based on the ideas introduced by Backus [Bac78] in 1978. A function-level approach is a system which consists of a set of second-order functions which are built on top of basic functions.

The second approach is an applicative approach. An applicative approach is a system which consists of functions which are defined on top of data. The user can directly manipulate the data through these functions.

The third approach is a combination of the first two approaches, i.e. higher order functions are defined on top of Daisy which is defined on top of data.

All three approaches are closely related to the approach presented in this thesis. Primitive functions are the standard cells which are defined on top of data. However, a set of higher order functions/combining forms is defined which will take the primitive functions as arguments. These combining forms specify geometric descriptions of the circuit.
1.3 Relative advantages/disadvantages of FCST (compared to closely related work)

FCST allows users to specify their ideas in PARA. PARA is an executable specification language. There is no need for a compiler to compile specifications written in PARA. In the case of other hardware description languages, such as VHDL [Coe89], a compiler is needed to compile the description and to translate it to another executable representation. One problem in having to compile specifications is the difficulty in constructing, testing, and proving correctness of individual components of the specification.

The entire FCST is implemented in a single language. The elimination of compiling and linking of the source language and library functions simplifies the debugging process in the development stage of the system. The linking of any procedural language with the libraries is more complex than in Miranda.

The language translators are relatively easy to modify. From time to time, modification of the input or output language is inevitable for the accommodation of more complex tasks. W/AGE offers a mechanical way of constructing executable translators and easy modification of the translators. For instance, the grammar specification of the input language is separated from the attribute functions. One can concentrate on either task which simplifies the complexity of the entire task.
Each function written in Miranda can be verified formally through the use of mathematical techniques. For instance, the Principles of Mathematical Induction and Structural Induction can be used to verify the correctness of functions.

μFP, introduced by Sheeran, provided a set of algebraic laws which can be used for converting the structure of a circuit to another simpler or an efficient one. Synthesis from recursion equation, introduced by Johnson, provided a set of axioms which is being used for deriving alternative recursion equations. These axioms allow designers to transform their ideas into recursion equation and reduce the unnecessary external components; therefore, in a sense, this transformation also simplifies the circuit.

At this moment, PARA does not provide any algebraic laws or axioms. This could be the subject of future work. However, from the point of view of this project, PARA has enough expressive power for the special architecture (see Appendix 2 for a brief description) which the Windsor Systolic Compiler has been used.

1.4 CONCLUDING COMMENTS FOR THIS CHAPTER

The purpose of this chapter was to give a description of other related work. Then a comparison between this work with other related work were also discussed.
Chapter 2
CONCLUDING COMMENTS

This chapter summarizes the results of the thesis and outlines some important extensions.

2.1 SUMMARY OF WHAT HAS BEEN DONE

The work undertaken was empirical in nature. A system called FCST (Functional Circuit Specification Transformer) was constructed as part of a larger project whose objective was to build a systolic compiler to automate parts of the specification and design of high performance systolic VLSI circuits.

A number of programs were constructed in the pure lazy higher order functional programming language Miranda: a translator from the FIR filter description language FADL to the executable systolic circuit specification language PARA and Module Description Language (MDL), a translator from PARA to a netlist representation in the Electronic Data Interchange Format (EDIF), and a translator from PARA to a graphical display.

The executable specification language PARA was itself implemented as a number of higher order functions written in Miranda. The W/AGE package that was used to build the translators from FADL to PARA and PARA to EDIF is itself written in Miranda.
FCST was tested in two ways: (i) a relatively complex FIR filter description in FADL was translated to PARA and then to EDIF and the results visually inspected, (ii) a simpler circuit was translated to EDIF and the EDIF representation was fed into the CADENCE VLSI layout package and manipulated there. The first of these ‘tests’ identified no errors in the EDIF output. The latter test demonstrated that the EDIF was at least syntactically correct. Both of these (very informal) tests suggested that FCST was working as it should. The translator from PARA to EDIF which was written originally to generate EDIF 1 was subsequently modified to generate the new version EDIF II.

2.2 CONCLUSIONS THAT CAN BE DRAWN

Owing to the nature of the thesis, it is impossible to conclusively prove or disprove it. However, a number of facts have been established which may go some way towards supporting the thesis:

(1) It is possible to build useful circuit specification translators in a pure lazy higher-order functional programming language, within the constraints of MSc thesis work.

(2) It is possible to implement an executable circuit specification language as an extension to a pure lazy
higher-order functional programming language, within the constraints of MSc thesis work.

(3) It is possible to build translators which generate postscript output, within the constraints of MSc thesis work.

The clarity and modifiability of the translators in FCST cannot be measured. However, it can be stated that they are completely declarative and contain no updateable variables of any kind. The translators are also modular in the sense that the executable grammar productions interact with each other only through very limited and well defined means (through order of application of component interpreters, and through inherited or synthesised attributes).

2.3 DIRECTION FOR FUTURE WORK

There are two aspects of the work that deserve further attention. The first is the continued investigation of the use of pure lazy higher-order functional programming languages in VLSI specification and design. The second is the practical extension of the FCST package.

2.3.1 Continued investigation of pure functional languages

The 'expert system' front-end of the systolic compiler is written in Prolog, a natural choice for this task. It may be of value to investigate the use of a pure lazy functional
language for this task. There is little reference to the use of lazy functional languages in expert systems work.

The 'transformation' that is carried out by FCST is relatively straightforward involving direct translation from one circuit specification language to another. Such transformation is deterministic; there is only one answer and only one way to get it. However, transformation in general involves various possible paths leading to various end-results all of which might meet some given criteria. Providing automated assistance to such transformation is an active research area in pure functional programming. It would be useful to investigate the extent to which techniques that are being developed by the functional programming community can be modified for use in the transformation of VLSI circuit specifications.

2.3.2 Practical Extension Of The FCST Package

The current version of the hardware description language has a number of limitations. Further work may be directed towards overcoming these limitations in the future. Currently PARA does not have any algebraic laws which can be used for refining or simplifying a circuit design. A functionally correct design may not be always optimal. These laws provide a means of transforming from a circuit to another simpler, more efficient circuit.
Current version of PARA supports uni-directional data flow only. Bi-directional flow of data is commonly used in VLSI architectures. This would allow users to specify and verify more complex architectures.

The user-interface of the entire system is based on commands entered at the Miranda prompt. A better user-interface would make better use of a window manager. Another graphical user-interface with a set of standard cells for specifying circuit design would allow the system to be stand-alone. This would allow not only FADL descriptions, but also any other design ideas be specified using PARA by graphically manipulating objects.
PART IV
APPENDICES
APPENDIX 1
SURVEY: FUNCTIONAL APPROACHES
IN VLSI DESIGN

Abstract

It is often the case that designing a VLSI circuit within a reasonable time is very difficult without using efficient techniques and design languages. A concern here is to look at how to improve design procedures by testing circuit specifications using a pure, lazy functional programming language. Several techniques of specifying circuits are currently proposed. These methods are (1) stream recursion equation, (2) recursive circuit specifications, and (3) algebraic laws on combining forms. Stream recursion equations provide a natural way to specify the relationships among the values that appear on wires. Recursive circuit specification allows designers to describe a circuit in a simple, abstract manner. Functional or relational combining forms obey algebraic laws that support reasoning about circuit properties and make it possible to explain systolic array algorithms."

Categories and Subject Descriptors: Requirements/Specifications—methodologies; D.2.2 [Software Engineering]: Tools and Techniques—modules and interfaces, software libraries .......


Additional Key Words and Phrases: Higher order functions, combining form, stream recursion equations, recursive circuit specifications.

This research was supported in part NSERC grants held by Dr. R.A. Frost, School of Computer Science, and Dr. G. Jullien, Department of Electrical Engineering, University of Windsor.

Author's address: School of Computer Science, University of Windsor, Windsor, Ont. Canada N9B 3P4.
1.1 INTRODUCTION

As VLSI technology advances, designing hardware circuits within a reasonable time is increasingly difficult especially with the complex structures involved. Design languages play an important role when designing VLSI circuits. An effective and accurate design language will not only decrease the amount of time that designers have to spend when testing their design, it also produces a more correct design within the time constraint.

Functional programming (hereafter denoted as FP) languages have proven to be successful notations for hardware specification. Algebraic laws have been used in functional languages – μFP [She83, She85c, She84], which improve the performance and the geometrical interpretation. The language μFP is based on FP and the recent relational language is called “Ruby” [She88]. Program transformation techniques have been used on functional programs to synthesize circuit specifications from higher-level algorithms [Joh84a, Joh84b, Joh83], and higher order functions have been used to support recursive circuit specification in the “Hydra” system along with alternative libraries of primitive function definitions to obtain behavioral and structural interpretations of a circuit specification [O'D88].

In addition to the above, functional languages provide
clean semantics along with concise mathematical notation. Circuit specifications in functional languages may be directly executable. A brief introduction to the advantages of using pure lazy functional programming languages for specifying circuits will be given in the next section.

1.1.1 The Problem

The problem is to see how pure lazy functional programming (PLFP) is used to describe the hierarchical as well as the structural aspect of a circuit design, the connectivity between components as well as the direction of data flow, and how circuit specifications are described in different approaches.

1.1.2 Overview of Approaches

The approaches to the problem can be categorised as follows: In stream recursion equation, functional recursion equations serve as specification; afterward, a realization language uses recursion to state the connectivity of a circuit's components, and finally, the initial specification is translated into iterative form which corresponds to the designer's circuit.

In algebraic laws on combining form, circuit specifications are expressions which are made of a small number of primitive functions and combining forms (functionals that map functions onto functions). These functions and combining forms have nice algebraic properties and geometric interpretation. Every μFP expression has an associated floor-plan.
Hydra provides several specification styles, including defining systems of recursion equations, hierarchical specifications with combining forms, and recursive circuit specification with geometric combining forms that fully define both the structure and behavior of a circuit.

1.1.3 Organisation of The Survey

The remainder of the survey is structured as follows: Section 2 introduces the pure lazy functional programming paradigm. Sections 3, 4, and 5 discuss three different approaches: The first one considered is Sheeran’s μFP [She85c, She83, She84]. Then Johnson’s streams recursion equation [Joh83, Joh84a] is discussed. Finally, we review recursive circuit specification – Hydra [O'D88, O'D87]. Section 6 analyzes different approaches, and the shortcomings of each technique. Finally, this survey concludes with comments in Section 7.

1.2 Overview of Pure Lazy Functional Programming Paradigm

Functional Programming (FP) was introduced by Backus [Bac78]. A program in FP is an expression representing a function that maps objects onto objects. The objects on which a program operates can be undefined, atoms or a sequence of objects. A program is a recursive definition. FP enjoys algebraic properties. It also can be proven easily by induction [Hen80].
In the following, advantages of Functional Programming for circuit specification are addressed:

- Representation of signals as streams
- Suitability of functional notation as description language for connectivity
- Algebraic properties: the basic algebra of terms commutes freely with interpretation of symbols as components, as do the most common place decomposition techniques
- Directly executable design languages implies no need to write a compiler or interpreter to implement a new hardware description language
- Circuit is viewed as a sequential control algorithm acting on an architecture which implies that programmers view their product as an algorithm acting on an abstract type
- Functional languages can make circuit specification directly executable implies easy to simulate the circuit in order to check that its behavior is correctly.

1.3 APPROACH !: Algebraic Style

μFP is a variant of FP. It can describe both the semantics and the behavior of a circuit and its layout. It allows the designer to reason about his circuit descriptions by manipulating them. These descriptions are expressions which
are made from a set of primitive functions and combining forms (functions that map functions onto functions). These functions and combining forms have nice algebraic properties. Each combining form has a geometric interpretation which implies that each μFP expression has an associated floor-plan.

In μFP, functions take a sequence of inputs over time and space, and produce a sequence of outputs. The geometric interpretation of combining forms shows the relationships between the semantics and the layout. For example, Composition $f \circ g$, Construct $[f, g]$, and Apply all $af$. Their abstract floorplans are shown in figure 25.

Figure 25 Simple combining forms and its abstract floorplans

Primitive functions are basic components of circuits: such as and/or gates or combinations of gates (subcircuits). For example

$$HA = [XOR, AND]$$

$$XOR = AND \cdot [OR, NOT \cdot AND]$$
Combinational circuits can be easily expressed in this language. Circuits with transition states can also be implemented. For example, a latch is used to hold the state. A geometric interpretation of it is shown in figure 26:

![Diagram of latch and function](image)

Figure 26 Geometric interpretation of transition state

where \( f \) is a primitive function.

The semantics of \( \mu FP \) is defined by a "Meaning" function, \( \mathbf{M} \), which can be used to prove the correctness of the description based on a set of algebraic laws. Also it is used for transformation which will simplify the circuit design and improve its layout.

Functions have also been defined for bidirectional flow of signals. These functions allow designers to design linear systolic arrays as discussed in Mead and Conway [MC80]. For example, the geometric interpretations of \( f \uparrow g \) and \( f \downarrow g \) are shown in figure 27:
By combining these blocks, one can create two-dimensional grids of orthogonally connected cells. Interested readers are referred to [She83] in which all functions for the above combining forms are explained in detail.

1.4 APPROACH II: System of Recursion Equations

In this approach [Joh83], functional recursion equations serve as specifications. A recursion equation is an equation whose variables range over functions. A specification is a system of recursion equations. A realization language uses recursion to state the connectivity of a circuit's components, and finally, initial specifications are translated into iterative forms which correspond to the designer's circuit. Iterative
form is a characterization of "sequence control", which is analogous to a flowchart description of a computation.

This approach involves the following steps:

a. Transform an initial specification into iterative form.
b. Construct an equivalent synchronous system description.
c. Make refinements.

Step 1 formalizes the tasks of developing a control algorithm for the circuit. Step 2 translates iterative specifications into detailed descriptions called simple loops. And finally, step 3 simplifies the description using a set of axioms by folding and unfolding.

The idea behind this approach is that any specification has a canonical solution; it is a unique set of minimally defined functions that satisfy the definition. This specification is unambiguous. Iterative form is a characterization of "sequential control", and thus coincides with the class of specifications that can be associated with a flowchart description of a computation.

The following example shows a simple circuit in the register transfer level. Figure 28 shows the schematic diagram. The component (!) is a clocked storage element or register. The token (•) indicates that the register has been initialized with the value a. Moreover, p, f, and g are primitive
operations. These functions operate over time.

Figure 28 Abstract schematic diagram in recursive equation system

The following expressions, (S), (P), and (C), are equivalent:

\[ F(a) \quad \text{Where } F(x) \leftarrow p(x) \rightarrow f(x), F(g(x)) \quad \text{(S)} \]

\[ x := a; \]

\[ \text{while } \neg p(x) \text{ do } x := g(x); \quad \text{(P)} \]

\[ x := f(x) \]

\[ X = a!g(X) \]

\[ \text{READY} = p(x) \quad \text{(C)} \]

\[ \text{VALUE} = f(x) \]
$S$ is a general recursion equation where $p(x)$ is a conditional expression and is referred to as a proposition expression. $S$ is interpreted as: if $p(x)$ is true, then $f(x)$ where $f(x)$ is a primitive function, otherwise, perform $F$ recursively by applying $g(x)$ where $g(x)$ is a primitive function. Equation $P$ shows a corresponding while loop in conventional imperative programming languages. The system $C$ is the realization language which is a linear description of the schematic* for the register transfer circuit.

The formal connection between source and target languages is made on the basis of the forms $S$ and $C$ above. Consequently, synthesis decomposes into a subtask of transforming an initial specification to an instance of $S$, followed by a sequence of refinements to the corresponding instances of $C$. The following is an example of implementing the Factorial function:

$$FAC(x) \leftarrow \text{zero?}(x) \rightarrow 1, mpy(x, FAC(dcr(x)))$$  \hspace{1cm} (S_1)

An equation of this kind can be reasoned using Structural Induction or Subgoal Induction. Synthesis from specification to iterative form can be achieved using mathematical transformations, multiplexors, and combined operations. By introducing an accumulator, the factorial specification has

* Left-right flow is used in schematics wherever possible, so that a component's input is on the left. This is informal notation and is used accompanied by systems like C that state the input-output relationship explicitly.
the following iterative version:

\[ G(x, y) \leftarrow \text{zero?}(x) \rightarrow y, \ G(\text{dcr}(x), \ mpy(x, y)) \]

where \( \text{dcr} \) and \( \text{mpy} \) have the following definition:

\[ \text{dcr}(c) = x - 1 \]

\[ \text{mpy}(x, y) = x \times y \]

Figure 29 shows its corresponding schematic diagram followed by its linear description.

![Figure 29 Schematic diagram of FAC](image)

\[ X = x^0! \ DCR(X) \]
\[ Y = 1! \ MPY(X, Y) \]

\[ \text{READY} = \ \text{ZERO?}(X) \]
\[ \text{VALUE} = \ Y \]
The system of recursion equations such as $C_1$ is then translated to **DAISY**, a descendant of Pure Lisp introduced by McCarthy, as follows:

$$\text{Fac} : \ x_0 \leftarrow \text{rec} \ \text{Experiment}$$

$$\text{where}$$

$$X = \langle x_0! \text{DCR} : \langle X \rangle \rangle$$

$$Y = \langle 1 \ ! \text{MPY} : \langle XY \rangle \rangle$$

$$\text{READY} = \ \text{ZERO}? : \langle X \rangle$$

$$\text{VALUE} = \ Y.$$ 

Now, the final step is the circuit refinement. The task is to modify a large combinational system so that it has fewer external components. This goal is attained by folding the system so that a component cannot simultaneously produce two results, thus it is necessary to serialize its behavior [Joh83]. Since **DAISY** is useful in folding, it was used to build algebra of synthesis on **DAISY**-like notation. Since it is not the concern of this survey to explore of this technique, interested readers are referred to Johnson’s thesis for detailed explanations.

### 1.5 APPROACH III: HYDRA

The method used in Hydra for circuit specification involves both stream recursions equation and combining forms which are similar to Johnson’s and Sheeran’s techniques respectively. The method is known as recursive circuit specification [O'D88]. Designers can choose different specification
styles: defining systems of recursion equations, hierarchical specifications with combining forms, or recursive circuit specifications with geometric combining forms.

Hydra represents each type of component with a function that hides the internal stream representation of the component’s behavior. A Hydra circuit behavior function takes a list of input signals and returns a list of signals. Circuit specification is treated as a functional program where the component names are free variables which may be bound by supplying a library of component function definitions. For example, a function $F$ which describes the behavior of the combinational components over time, assuming that $f$ gives its instantaneous behavior during one cycle. The input $X$ and the output $Y$ are streams of individual values, and $Y = F : X$ describes the component’s behavior, where

$$F = \lambda s. [f : head : s ! F : tail : s].$$

The notation $f = \lambda x. y$ means “$f$” is a function which returns the value of $y$ when given input $x$. The (!) operator is equivalent to the append or concat operation in Lisp. For example,

$$[a ! [x_0, x_1, x_2, ...]] \text{ returns } [a, x_0, x_1, x_2, ...].$$

A component with 2 inputs has a similar description. The component’s description equation is $Y = G : [A \ B]$ where

$$G = \lambda [r s]. [g : [head : r head : s] ! G : [tail : r tail : s]]$$
where $g$ is its \textit{instantaneous behavior function}.

For clocked components, it is defined by constructing a \texttt{Letric} expression. This is an extension of Johnson's work. \texttt{Letric} is used to hide the internal details of a subsystem. This allows the circuit specification to maintain a hierarchical structure and it is easier to understand complex specifications. The \texttt{Letric} expression implements recursive definitions of data structures. For example:

$$C = \lambda \text{in}.$$  

\texttt{letrec } $X = \text{XOR} : [\text{D in}]$

$$D = [0 ! X]$$

in D

and can be represented graphically, as in figure 30.

![Figure 30: Internal detail of a simple circuit](image)

This function takes an input and produces an output, but the streams $X$ and $D$ are defined locally within the \texttt{Letric} expression. Nested \texttt{Letric} expressions support a structured, hierarchical design style.
This approach is generalized by building a set of primitive component behavior definitions. Instead of constructing the corresponding stream behavior function, it is convenient to introduce a function $behavior$ that creates stream functions from instantaneous behavior functions:

$$behavior = \lambda f. \lambda inputs. (\text{star} : f) : inputs$$

where $\text{star}$ maps $f$ over the input streams in order to compute the output streams. Stream behavior functions are restricted to the set of primitive component behavior functions. For example:

$$Y = \text{AND}[A \ \text{NOT} : B]$$

where AND and NOT are primitive component behavior functions:

$$\text{AND} = \text{head} : behavior : \lambda [a \ b] . a \land b$$

$$\text{NOT} = \text{head} : behavior : \lambda a . \overline{a}$$

In the case of synchronized circuits, stream behavior functions will be defined as follows:

$$\lambda inputs.$$

$$letrec$$

$$\text{AND} = \ldots$$

$$\text{OR} = \ldots$$

$$\text{Latch} = \ldots$$

in ...expression using AND, OR, Latch, etc...

A set of structural primitives is constructed for implementing the layout of the designers' circuits. A system of recursive
equations will result in a circular graph structure that is isomorphic to the graph structure of the actual circuit. Consider the circuit $Y = F : G : X$, the output of $G$ becomes the input of $F$. The outputs of a component representation are the targets of pointers from devices that read those outputs, while the inputs of the component are represented by pointers to the sources that provide the input values.

The representation of a component contains the type of component, a pointer to each of its inputs, and a port for each of its outputs. Each output port points to the component, and other circuits may contain pointers to whichever output ports are needed. Consider component $A$ in Figure 31. It has input ports $a$, $c$, $e$, $g$, and $h$, and output ports $b$, $d$, and $f$.

Two components are connected only if the circuit ports match up properly along their common edges; then the circuits can be combined to form a larger circuit. Figure 31 shows two circuits: $A$ and $B$, and all the ports of each circuit. Two circuits can be connected using the primitives $hbox$ and $vbox$.

![Circuit A](image)

![Circuit B](image)

Figure 31 Structural primitive $hbox$ and $vbox$ in Hydra
Figure 32 shows how hbox connects two adjacent circuits A and B and the interconnections between them. vbox works the same way as hbox. The geometric combining forms are implemented in two ways: the first implementation, as demonstrated above, uses the TeX typesetting system which is based on functional geometry [Hen82]. The second implementation is a program that takes a Hydra geometric specification, works out the coordinates, and calls the standard Unix graphics plotting procedures.

![Interconnection diagram of hbox A B](image)

![Geometric layout of hbox A B](image)

Figure 32 Connections and layout

1.6 ANALYSIS OF APPROACHES

This first approach inherits most of the properties of FP. The specification itself is an executable description. Each of the combining forms is associated with a layout. It is suitable for regular systolic array structures. Designers are able to improve their designs through transformations based on a set of algebraic laws.
The disadvantages of the second approach are that there is no obvious way to describe "stability" in the specification language. Circuits are assumed to be synchronous. What happens, for example, if one of the components comes with its own clock. Another problem is that components map inputs to outputs making the treatment of tri-state logic problematical. And since recursion corresponds directly to connectivity in realizations, there is an overhead of creating a graphical data base for evaluating a realization in an environment where ground symbols are bound to graphical primitives.

The third approach, Hydra, is not useful in the final design of the actual VLSI layout, because it is not integrated into a standard set of VLSI design tools. It can only construct a VLSI layout for the part of the system which was specified with combining forms. Hydra cannot describe the behavior of a clock in a synchronous circuit as temporal logic. Bidirectional dataflow is another problem that is not solved. Moreover, the stream recursion equations are too abstract to find timing errors, clock skew, and capacitance problems.

1.7 CONCLUDING COMMENTS

This paper began by introducing the advantages of Functional Programming for specifying circuit design and followed the different approaches using functional notations for circuit specifications. Then the relative merits and disadvantages
of using such approaches as hardware specification languages were discussed.

Besides the above mentioned alternatives, HOL [Gor86, Gor88] is also being used to describe digital hardware. For example, look at a transistor \( T \) with gate \( g \) and diffusion ports \( a \) and \( b \). The expression

\[
T(g,a,b) \equiv (g \lor (a = b))
\]

specifies exactly what the transistor does: if the value of \( g \) is 1, then we can deduce that \( a=b \). Another hardware description language, introduced by Patel et al., is known as \( \nu \)FP [PSE85] and is also based on FP. In this approach, the following set of functions is defined: primitive functions, structure modifying functions, and functional forms that combine primitive functions into more complex functions. Designers are free to choose whichever level is “best” for their current purposes.

**Miranda**, a pure, lazy functional programming language, is also being used for simulating digital circuits [Hi87]. In this approach, the properties of **Miranda**, full laziness, simple syntax based on recursion equations, and polymorphic typing, are exploited. The simulation of a circuit is also a description of it.

The main issue when discussing hardware description language is how simple is the language? A hardware description
language should be easy to read, to learn and to use to ensure its usefulness. Designers are used to express their ideas using concise and compact notations, as for instance, blocks, diagrams and logics. Hardware description language should support this approach. It is more likely for a simple language to have formal semantics. Secondly, what is the expressive power of this language? Designers should be able to express any system they have in mind. And finally, is the language mathematically tractable? As systems become more and more complex, methodologies for formal verification should be integrated.
Appendix 2
The Special Architecture

This Appendix gives an overview of the cell that we are using in this project. The cell is developed locally at the University of Windsor (see [Tah88, JBMF89, Sar90]). It is specially designed for designing filters.

2.1 The Standard Cell

The basic building block is illustrated in figure 21. All components of the filter, i.e., the encoder, the filter computational element and the scaling block, are made up of the same cell. This cell consists of a 32x5 bit ROM along with the associated decoding circuitry. This cell basically has ten input and output pads (disregarding the clock, power and ground). Five of these bits \( y_0 \) to \( y_4 \) (refer to the figure) act as address lines for the ROM, and one of the bits from the remaining five \( (x_0 \) to \( x_4 \)) acts as a steering bit [Tah88]. If the steering bit is 1 the output is taken from the ROM. If it is 0, the output is simply the input value.

Five of these cells connected in series to form an single operational unit (as shown in figure 33) and perform the following function:

\[
Y_{out} = Y_{in} \oplus_m A \otimes_m X_{in}
\]

\( Y_{in} \), \( X_{in} \) and \( Y_{out} \) represent three five bit numbers that are input and output to the first cell in the block. \( A \) represents a
number which determines the contents of the ROMs. The contents of the $i^{th}$ ROM is determined by the following equation:

$$
Content = 2^{i-1} \otimes_m A \otimes_m Address
$$

The ROMs basically act as look up tables where the appropriate answers are stored.

All computations are performed using residue arithmetic [JBMF89]. The advantage of using this Residue Number System is that it permits faster computations.

### 2.2 OVERVIEW OF THE FILTER ARCHITECTURE

A filter usually consists of three main functional blocks: the encoder, the filter computational element, and the scaling block as shown in Figure 34.
2.2.1 The Encoder

The function of the encoder is to map a binary weighted number to a residue given a specified modulus. This mapping operation is simply a modulo reduction of the input number. A $S$-bit binary number can be reduced modulo $m$ as follows:

$$|X|_m = \sum_{b=0}^{S-1} 2^b \otimes_m X^b$$

where the notation $|X|_m$ represents modulo $m$ reduction of $X$.

There are several ways in which the encoder may be implemented. We are only interested in two ways as shown in figures 36 and 35.
These implementations are selected for the construction of encoders. The implementation being used depends on the number of input bits. The parallel structure as shown in figure 35 is used for a 9 to 18 bit input encoder. The equations being computed are:

\[
\sum_{b=0}^{3} 2^b \otimes_m x^{[b]} \oplus \sum_{b=4}^{8} 2^b \otimes_m x^{[b]}
\]

\[
\sum_{b=0}^{3} 2^b \otimes_m x^{[b+9]} \oplus \sum_{b=4}^{8} 2^b \otimes_m x^{[b+9]}
\]

\[2^9 \otimes A_2 \oplus A_1\]

BLOCK A, BLOCK B and BLOCK C of figure 35 carry out computations specified by these equations.

The single block encoder implementation as shown in Figure 36 is being used for 9 bits or less of input. This block (BLOCK A) carries out the same computation as the first block (also called BLOCK A) of the previous implementation.
2.2.2 The FIR Filter

The computations for the FIR filter are done using residue arithmetic. Using residue arithmetic an $N^{th}$ order FIR filter can be expressed by the following equation:

$$|Y(n)|_m = \sum_{i=0}^{N-1} A(i) \otimes_m X(n-i)$$

where $X$, $A$ and $Y$ represent the inputs, the filter coefficients and the outputs respectively, and are all reduced modulo $m$. This filter implementation uses a word serial/bit parallel input form. This equation can be further decomposed to the bit level as follows:

$$|Y(n)|_m = \sum_{i=0}^{N-1} \sum_{b=0}^{4} 2^b \otimes_m A(i) \otimes_m x[0](n-i)$$

The inner summation represents multiplication of one input word by a single coefficient. The multiplication requires a block of five cells. For an $N^{th}$ order FIR filter $N$ such blocks must be connected in series as shown in figure 37.
The partial products calculated in the inner summations are added by the outer summation, thus eliminating the need for a separate adder. Also, the addition is performed in a distributed manner, a part of it in each cell.

### 2.2.3 The Scaler

The final component is the scaling block. It accepts data in RNS form from the previous section which is the FIR filter. Scaling performs two main functions:

1. Reduces the dynamic range so that the final answer may be accommodated in a smaller number of bits.
2. By choosing appropriate moduli, it is possible to obtain a 'free' conversion to binary format.
The implementation chosen for the systolic compiler is shown in figure 38. In this case, we will only accept four moduli.

![Figure 38 Implementation chosen for systolic compiler](image)

The black block consists of 5 latches connected in serial. The data line shown on the blocks represents the cyclically rotated parallel data path with access to the top serial bit. The arrow between blocks B1 and B2 indicates that the serial bit used is from block B2, rather than its own serial bit.

We first defined $\nu_1$, $\nu_3$, and $\nu_4$ as follows:

\[ \nu_1 = m_4^{-1} \otimes_{m_1} m_3^{-1} \otimes_{m_1} m_2^{-1} \]

\[ \nu_3 = m_3^{-1} \otimes_{m_1} m_4^{-1} \]

\[ \nu_4 = m_3^{-1} \otimes_{m_1} m_2^{-1} \]
The input to the decoder, \(X_1, X_2, Y_3,\) and \(X_4,\) can be mapped to \(\overline{X}_1, \overline{X}_2, \overline{X}_3,\) and \(\overline{X}_4\) employing only a single cell for each input (this cell will be the first of a block of 6 cells) as follows:

\[
\overline{X}_1 = (X_1 \oplus_{m_1} \gamma) \oplus_{m_1} \nu_1
\]

\[
\overline{X}_2 = (X_2 \oplus_{m_2} \gamma)
\]

\[
\overline{X}_3 = (X_3 \oplus_{m_3} \gamma) \oplus_{m_3} \lfloor m_2^{-1} \rfloor_{m_3}
\]

\[
\overline{X}_4 = (X_4 \oplus_{m_4} \gamma) \oplus_{m_4} \nu_4
\]

The additional constant \(\gamma = \frac{m_1 m_2}{2}\) is used to allow rounding of the estimate to the nearest integer.

The decoding and scaling operation can be performed using mod \(m\) adder blocks if the free constant multiplication property of the cells is used. The details of the operation are given as follows:

\[
B_1 = \overline{X}_3 \oplus_{m_3} \{[-\overline{X}_2] \oplus_{m_3} m_2^{-1}\}
\]

\[
B_2 = \overline{X}_1 \oplus_{m_1} \{[-\overline{X}_2] \oplus_{m_1} \nu_1\}
\]

\[
\overline{B}_3 = B_2 \oplus_{m_1} \{[-B_1] \oplus_{m_1} \nu_3\}
\]

\[
B_4 = \overline{X}_4 \oplus_{m_4} \{[-\overline{X}_2] \oplus_{m_4} \nu_4\}
\]

\[
B_5 = B_4 \oplus_{m_4} \{[-B_1] \oplus_{m_4} m_3^{-1}\}
\]

\[
B_6 = B_3 \oplus_{m_1} \{[-B_5] \oplus_{m_1} m_4^{-1}\}
\]
APPENDIX 3
Summary: The Designer's Assistant

3.1 Outline of the Windsor VLSI Designer's Assistant

The Windsor VLSI Designer's Assistant is the knowledge-based front-end of the Systolic Compiler. The description here is derived from [Sar90]. Its domain target is high speed digital filter design. The Designer's Assistant provides a window-based, interactive, user-friendly environment with minimal manual intervention for the entire filter design process. This environment helps novice users to learn the techniques of filter design and expert users to design advanced filters.

In the design process, users are required to supply a description of the frequency response of the filter by answering a number of questions posed by the expert system. Additional parameters such as hardware restrictions – maximum number of chips, number of input/output bits (dynamic range) etc. – are also required for generating filter coefficients for the filter. These coefficients are then used to reconstruct the frequency response of the designed filter. The design cycle is completed if the differences between the actual and desired filters are within specified tolerance limits. Otherwise, selected parameters are modified and the design cycle is iterated. Figure 39 shows the design cycle of a filter design session.
Figure 39 Overview of Designer's Assistant

In a design session, three databases are generated. User-provided information is stored in a database file called the user-response-database, which is useful for resuming the session later. A calculated values database is used to store parameter values. These values may be calculated from
user specified values, or suggested by the expert system as defaults; these will be modified during the design process. The last database is the default database which contains default values for certain parameters which may be required during the design process. These values are set up at the beginning of a filter design session as soon as the field of application has been specified. These values may or may not be used depending on user specified value for that parameter. The Designer's Assistant consists of the following main components:

(1) The knowledge base,
(2) The inference engine,
(3) The explanation facilities,
(4) The user interface, and
(5) The abort facility.

3.2 The KNOWLEDGE BASE

The knowledge base consists of a number of rules and facts concerning the design of digital FIR filters. Facts are not associated with any condition, whereas rules are associated with at least one condition. The rules and facts can be used to generate new facts which are used only in the current session. Conclusions are drawn based on information provided by the user and also by verifying preconditions, weights, and conditions of the specific rule. The preconditions are an
optional feature that can be used to speed up the processing. The preconditions are a list of one or more goals that must be satisfied in order to use a specific rule. The use of weights is an optional feature that can be used to decide which rule should be attempted first. This feature is applicable only in the case where more than one rule is corresponding to a particular goal. And finally, the conditions are one or more conditions which determines if the conclusion is valid.

3.3 The INFEERENCE ENGINE

The inference engine is used to make inferences using the rules in the knowledge base. The inferencing scheme is similar to that of Prolog. The heart of the inference engine is the confirmed procedure which first checks if the goal corresponds to a built-in Prolog procedure. If this is the case control is passed to the Prolog processor to determine whether the goal is true or not, otherwise confirmed calls preprocess to determine the condition sets that can be used to prove the goal. After the list of possible condition sets is returned, confirm_rule is invoked to determine which of the condition sets will satisfy the predicate. confirm then attempts to prove if they are true.

3.4 The EXPLANATION FACILITIES

The explanation facilities provide three types of explanations:
(1) WHY Explanations,
(2) HOW Explanations, and
(3) WHYNOT Explanations.

3.4.2 WHY Explanations

In a design session, the system asks the users for information on the design. The WHY facility remains in the entire design process and provides the users with the reason why is each question asked by the system. The system will then display the sequence of rules that was used to reach the current state. Further clarification on each individual rule is available, if asked. In addition, this facility also explains why and how the value for a particular rule is used.

3.4.3 HOW Explanations

The HOW facility explains how a particular conclusion was reached at the end of a successful design session. This is useful in explaining a conclusion since it shows each intermediate step that is taken in reaching the final answer.

3.4.4 WHYNOT Explanation

The WHYNOT facility, in converse to HOW, explains to the user why a design session has ended unsuccessfully. This facility provides a list of rules that have failed at the point where the design failed.

3.5 The USER INTERFACE

The user interface was implemented by Sarkar and using the
windowing facilities of the Prowindows system [Qui88]. The methods of interaction include

(1) Entering numerical data
(2) Selection from menu
(3) Entering points
(4) Graphical interface
(5) User-log-window

In a design process, certain information is needed from the designer. For instance, filter response is one of the parameters which can be expressed numerically. Numerical values are accepted when the design is being queried.

Menus are provided in the case where choices are limited. The user interface provides two types of menus – single-selection and multi-selection. The user may select at most one item from the single-selection menu. For instance, you can only design a specific type of filter. In the multi-selection menu, the user can select more one item. In this case, one might prefer one filter algorithm over another one; however, multi-selection allows one to select different algorithms according to his/her preference.

There are two other ways of specifying frequency response of a desired filter: entering all the points in the spectrum or using the graphical interface. In the former case, each point must be specified by two coordinates separated by a
comma and enclosed in parenthesis. The graphical interface provides a graphical way of inputting desired filter response and examining the filter designed by the system at the end of a session by comparing two curves: one represents the ideal response and the other represents the actual response of the filter.

The user-log-window displays, on the left side of the screen, keep a log of the current session with the answers provided by the user. The user has the option of modifying any of these answer during the session. However, the entire session has to be re-initialized if any change is made. Otherwise, the system will use the information provided.

3.6 The ABORT/RESET FACILITY

The Abort/Restart facility enables the user to discontinue a session in the middle of a design process. The current state of the program can be saved in a file and the user can resume at the same point in a later session. This file contains all the responses given by the user in the current session.

3.7 OUTPUT FROM THE DESIGNER’S ASSISTANT

The output from the Designer’s Assistant is a text file which is called Filter Architecture Description Language or FADL in short. FADL contains information of a filter realization, such as the name of the chips, its purposes, the moduli and coefficients used in the filter.
3.8 CONCLUDING COMMENTS

The purpose of this Appendix was to give an overview of the knowledge-based front-end of the Windsor Silicon Compiler. The different components in the Designer’s Assistant were discussed briefly, the system architecture and methodologies that were used in this front-end were also described.
APPENDIX 4
FCST END-USER INSTRUCTIONS

The following lists show all the valid commands and the type of arguments each command accepts in FCST.

FCST:

a. testpara
   • convert “fadlspec”
   • input_pad_name “inpadfilename” [“In1”, ...]
   • output_pad_name “outpadfilename” [“Out1”, ...]
   • input_signal “signalfilename” [“HLLHL..”, ...]
   • execucircuit cirname “signalfilename” “outfilename”
   • sim “cirname” “padfilename” “signalfilename”
   • plotsim “cirname” “padfilename” “signalfilename”

b. paratoedif
   • netlist “inpadfilename” “outpadfilename” “paraspec”
   • netlistout “inpadfilename” “outpadfilename” “paraspec” “netlistfile”

After entering the Miranda interpreter with the current script FCST.m, the valid commands are testpara and paratoedif without any argument. There are subcommands, as shown above highlighted in italic font, followed by one or more arguments. These argument could be a file name, list of strings, or circuit name specified in PARA specification.
APPENDIX 5
FCST SCRIPTS

This Appendix contains complete Miranda scripts: FCST.m, TESTPARA.m, SIMPARA.m, FADLTOPARA.m, PRI_lib.m, SEC_lib.m, cell.m, plot.ps, and PARATOEDIF.m.

5.1 FCST INTERFACE: FCST.m

help
  = lay ["\n","\n","Type","\n"]
  ++
  inst
  ++
  "\n"
  ++
  "\n"

inst
  = layn ["\"testpara\" to translate FADL to PARA
          and run simulation",
          "\"paratoedif\" to translate PARA
          specification to Edif Netlist"]

testpara
  = [System "/usr/openwin/bin/xview/cmdtool mira
      -heap 700000 TESTPARA.m"]

paratoedif
  = [System "/usr/openwin/bin/xview/cmdtool mira
       PARATOEDIF.m"]
5.2 TESTPARA : TESTPARA.m

#include "FADLTOPARA.m"
#include "SIMPARA.m"
#include "pri_lib.m"
#include "sec_lib.m"
#include "5bitcell.m"

help
  = lay [" ","Steps to be followed in sequence,
         type ", ",

++
help0

help0
  = layn [" convert "fadlspecfile" ",
            " input_pad_name "namefile"
             
            [[]"In1"","In2"",...]]",
            " output_pad_name "namefile"
             
            [[]"Out1"","Out2"",...]] ",
            " input_signal "infile" 
            [[]"HLLL"
            [,"LHHL","LHHL",...]]",
            " execute_circuit circuitname "inputfile"
             "outputfile" ",
            " sim "circname" "namefile"
             "outputfile" ",
            " plotsim "circname" "namefile"
             "outputfile" "]
5.3 PARATOEDI5 : SIMPARA.m

/// This file is an intermediate library file
/// which is used to include all necessary
/// intermediate files.
#include "5bitcell.m"
#include "pri_lib.m"
#include "sec_lib.m"
#include "sigcell.m"
#include "circ0.m"
5.4 FADL to PARA TRANSLATOR: FADLTOPARA.m

| This is a translator from "FADL" to "PARA" and "MDL". |

```
#include "ambi_wage.021890.m"
#include "SIMPARA.m"
#include "pri_lib.m"
#include "sec_lib.m"
#include "5bitcell.m"

u = wage_uparrow name
d = wage_downarrow name
rule = wage_rule
rstc = wage_recognises name

fadls = rstc (s1 fadl ++ s2 fadls)
        [rule 10.10 (PARA $u lhs) EQ conc_para [PARA $u s1, PARA $u s2],
         rule 10.20 (ROM_CONTENTS $u lhs) EQ
         conc_rom [ROM_CONTENTS $u s1, ROM_CONTENTS $u s2]]

$excl_orelse
rstc (s1 fadl)
        [rule 15.10 (PARA $u lhs) EQ same_as [PARA $u s1],
         rule 15.20 (ROM_CONTENTS $u lhs) EQ
         same_as [ROM_CONTENTS $u s1]]

conc_para [PARA p1, PARA p2] = PARA (p1++p2)

conc_rom [ROM_CONTENTS x, ROM_CONTENTS y]
    = ROM_CONTENTS (x ++ y)

epsilon = succeed
initialize_para [] = PARA []
```
fadl = rstc (s1 entity ++ s2 module ++ s3 floorplan ++ s4 mapblk)
    [rule 20.10 (PARA $u lhs) EQ get_fadl [CHIPNAME $u s1,
        MODULI $u s2,
        PROPERTY $u s2],
    
    rule 20.20 (ROM_CONTENTS $u lhs) EQ
    
        rom_contents [ENT $u s1,
          PURPOSE $u s2,
          MODLIST $u s2,
          ROM_PROPERTY $u s2]]

rom_contents [ENT n,PURPOSE p,MODLIST m,ROM_PROPERTY (INPUT_BITS i)]
  = ROM_CONTENTS ["chipname = " ++ n ++ "\n"
      ++
      "purpose = " ++ p ++ "\n"
      ++
      "moduli = \n" ++ (showmod m) ++ "\n"
      ++
      "rom = \n\{( " ++ (showcoeff m [16]) ++ ", i <= 9

  = ROM_CONTENTS ["chipname = " ++ n ++ "\n"
      ++
      "purpose = " ++ p ++ "\n"
      ++
      "moduli = \n" ++ (showmod m) ++ "\n"
      ++
      "rom = \n\{( " ++ (showcoeff m [16,16,512]) ++ "\n"
          i <= 18

rom_contents [ENT n,PURPOSE p,MODLIST m,ROM_PROPERTY(COEFFICIENTS c)]
  = ROM_CONTENTS ["chipname = " ++ n ++ "\n"
      ++
      "purpose = " ++ p ++ "\n"
      ++
      "moduli = \n" ++ (showmod m) ++ "\n"
      ++
      "rom = \n\{( " ++ (showcoeff m c) ++ "\n"

showmod [] = []

showmod (a:as)
  = "\n\{( " ++ (shownum a) ++
        concat [" " ++ (shownum y) | y <- as] ++ "\n"

showcoeff ms cs
  = concat [g m cs | m <- ms]

where
  g m cs = concat [f m (c*2^disp) | c <- cs; disp <- [0..4]]
f m s = showmod (multi_coef m s)

entity = rstc (s1 (uterm "Entity") ++ s2 chipname )
    [rule 30.1 (CHIPNAME $u lhs) EQ same_as [CHIPNAME $u s2],
     rule 30.2 (ENT $u lhs) EQ chipname_to_entity [CHIPNAME $u s2]]

chipname = translate_from_charstring_to CHIPNAME

chipname_to_entity [CHIPNAME n]
    = ENT n

module
    = rstc (s1 (uterm "Module") ++ s2 dummy ++
        s3 (uterm "]") ++ s4 properties ++ s5 (uterm ",")
    [rule 40.10 (MODULI $u lhs) EQ same_as [MODULI $u s4],
     rule 40.20 (PROPERTY $u lhs) EQ same_as [PROPERTY $u s4],
     rule 40.30 (PURPOSE $u lhs) EQ same_as [PURPOSE $u s4],
     rule 40.40 (MODLIST $u lhs) EQ mod_to_modlist [MODULI $u s4],
     rule 40.50 (ROM_PROPERTY $u lhs) EQ
        prop_to_rom_prop [PROPERTY $u s4]]

mod_to_modlist [MODULI m]
    = MODLIST m

prop_to_rom_prop [PROPERTY p]
    = ROM_PROPERTY p

properties
    = rstc (s1 purpose ++ s2 ( uterm ",") ++ s3 moduli ++
        s4 (uterm ",") ++ s5 aproperty)
    [rule 50.10 (PURPOSE $u lhs) EQ same_as [PURPOSE $u s1],
     rule 50.20 (MODULI $u lhs) EQ same_as [MODULI $u s3],
     rule 50.30 (PROPERTY $u lhs) EQ same_as [PROPERTY $u s5]]

purpose
    = rstc (s1 (uterm "Purpose") ++ s2 purposes)
[rule 55.10 (PURPOSE $u$ lhs) EQ same_as [PURPOSE $u$ s2]]

purposes = mkint purposes_list

purposes_list = ["encoder", [PURPOSE "encode" ]],
               ["ipap", [PURPOSE "FIR_Filter" ]],
               ["scalar", [PURPOSE "scalar" ]]

moduli
     = rstc (s1 (u term "Moduli") ++ s2 (u term "[")) ++
       s3 listofmodi ++ s4 (u term "]" )
     [rule 61.10 (MODULI $u$ lhs) EQ same_as [MODULI $u$ s3]]

aproperty
     = rstc (s1 input_bits)
     [rule 60.10 (PROPERTY $u$ lhs) EQ same_as [PROPERTY $u$ s1]]
     $orelse
     rstc (s1 coefficients)
     [rule 60.20 (PROPERTY $u$ lhs) EQ same_as [PROPERTY $u$ s1]]
     $orelse
     rstc (s1 output_bits)
     [rule 60340 (PROPERTY $u$ lhs) EQ same_as [PROPERTY $u$ s1]]

input_bits
     = rstc (s1 (u term "Input_bits") ++ s2 numofinput)
     [rule 62.10 (PROPERTY $u$ lhs) EQ same_as [PROPERTY $u$ s2]]

numofinput [(s, (w:ws))]
     = [[PROPERTY (INPUT_BITS (numval w))], ws], numstr w
     = [], otherwise

coefficients
     = rstc (s1 (u term "Coefficients") ++ s2 (u term "]") ++
           s3 listofcoeff ++ s4 (u term "]" ))
     [rule 63.10 (PROPERTY $u$ lhs) EQ same_as [PROPERTY $u$ s3]]
output_bits
    = rstc (s1 (uterm "Output_bits") ++ s2 numofoutput)
        [rule 64.10 (PROPERTY $u 1hs) EQ same_as [PROPERTY $u s2]]

numofoutput [(s, (w:ws))]
    = [[(PROPERTY (OUTPUT_BITS (numval w))), ws]], numstr w
    = [], otherwise

floorplan
    = rstc (s1 (uterm "Floorplan") ++ s2 (uterm "[") ++
        s3 (uterm "Bias") ++ s4 bias ++ s5 (uterm ",") ++
        s6 (uterm "Array") ++ s7 arrays ++ s8 (uterm "]")])
        [rule 50.1 (DUMMY $u 1hs) EQ same_as [DUMMY $u s4],
         rule 50.1 (DUMMY $u 1hs) EQ same_as [DUMMY $u s7]]

bias = mkint bias_list

bias_list = [("column", [DUMMY "column"]),
             ("row", [DUMMY "row"])]

arrays = rstc (s1 (uterm "[") ++ s2 (uterm "[") ++
        s3 array ++ s4 (uterm "]") ++ s5 (uterm "]")])
        [rule 55.1 (DUMMY $u 1hs) EQ same_as [DUMMY $u s3]]

array = translate_from_charstring_to DUMMY

mapblk
    = rstc (s1 (uterm "Map") ++ s2 (uterm "]") ++
        s3 (uterm "[") ++ s4 block ++ s5 (uterm ",") ++
        s6 modname ++ s7 (uterm "]") ++ s8 (uterm "]")])
        [rule 60.1 (DUMMY $u 1hs) EQ same_as [DUMMY $u s4],
         rule 60.2 (DUMMY $u 1hs) EQ same_as [DUMMY $u s6]]

block = translate_from_charstring_to DUMMY
modname = translate_from_charstring_to DUMMY

dummy = translate_from_charstring_to DUMMY

property_list
= [["Input_bits", [DUMMY "Input_bits"]],
  ("Coefficients", [DUMMY "Coefficients"]),
  ("Output_bits", [DUMMY "Output_bits"])]

translate_from_charstring_to con input
= concat (map (tfct con) input)

tfct con (inh,[]) = []
tfct con (inh, (u:us))
= [[(con u,us)], charstring u
  = [], otherwise
  where
    charstring [c] = ischar c
    charstring (c:cs) = (ischar c) & (charstring cs)
    ischar c = ('a' <= c <= 'z') \ (c = '_' \ (c = '"') \ /
                 ('0' <= c <= '9') \ ('A' <= c <= 'Z')

conc_coeff [COEFFICIENTS x, COEFFICIENTS y]
  = COEFFICIENTS (x+y)

conc_mod [MODULI s, MODULI t] = MODULI (s+t)

listofcoeff
  = rstc (s1 coeff ++ s2 (uterm ",") ++ s3 listofcoeff)
    [rule 4.2 (PROPERTY $u lhs) EQ makenumlist2 [NUMVAL $u s1,
          PROPERTY $u s3]]

$excl_orelse
rstc (s1 coeff)
  [rule 4.1 (PROPERTY $u lhs) EQ makenumlist1 [NUMVAL $u s1]]

coeff [(s, (w:ws))] = [[NUMVAL (numval w)], ws]},
(w ^= "-") & numstr w
    = [{[NUMVAL (- (numval y))], wss}],
    (x ^= "-") & numstr y
    = [], otherwise
where
    (x:y:wss) = (w:ws)

numstr [] = False
numstr ls = and (map digit ls)

makenumlist1 [NUMVAL x] = PROPERTY (COEFFICIENTS (x:[]))

makenumlist2 [NUMVAL x, PROPERTY (COEFFICIENTS y)]
    = PROPERTY (COEFFICIENTS (x:y))

makenumlist3 [NUMVAL x, MODULI y] = MODULI (x:y)

makenumlist4 [NUMVAL x] = MODULI (x:[])

listofmodi
    = rstc (s1 modi)
        [rule 6.1 (MODULI $u lhs) EQ makenumlist4 [NUMVAL $u s1]]
        Sorelse
        rstc (s1 modi ++ s2 (uterm ",") ++ s3 listofmodi)
        [rule 6.2 (MODULI $u lhs) EQ makenumlist3 [NUMVAL $u s1,
            MODULI $u s3]]

modi = coeff

‖ Attribute type

attributes ::= CHIPNAME [char]
    | PARA [[char]]
    | PROPERTY property
    | MODULI [num]
    | PURPOSE [char]
property ::= COEFFICIENTS [num]
    | INPUT_BITS num
    | OUTPUT_BITS num
    | COEFFVAL num

name (CHIPNAME x) = "chipname"
name (PARA x) = "para"
name (NUMVAL x) = "numval"
name (PROPERTY x) = "property"
name (MODULI x) = "moduli"
name (PURPOSE x) = "purpose"
name (DUMMY x) = "dummy"

!! Attribute Functions

same_as [x] = x

get_fadl [CHIPNAME n, MODULI m, PROPERTY (INPUT_BITS i)]
    = encoderspec n m i

get_fadl [CHIPNAME n, MODULI m, PROPERTY (COEFFICIENTS c)]
    = filterspec n m c

get_fadl [CHIPNAME n, MODULI m, PROPERTY (OUTPUT_BITS o)]
    = scalarspec n m o

filterspec p m c
    = PARA ([lay2 (p ++ " " ++ "=")]
        ++
        (parspec c m) ++ [";"])

parspec [] ms = []
parspec (c:cs) [] = []
paraspec (c:cs) ms
    = initspec c ms ++ restspec cs ms

initspec c ms
    = ["(wire " ++ (wspec2 ms) ++ ")" ++ "\n"
        ++
        "$con" ++ "\n"
        ++
        "(para" ++ " " ++ (pspec c ms) ++ ")" ]

pspec :: num -> [num] -> [char]
pspec c ms
    = lay2 (show ["msc" ++ " " ++ (show m) ++ " " ++
                    "(" ++ (show c) ++ ")" | m <- ms ])

lay2 [] = []
lay2 (a:x) = lay2 x, a = '\\'
            = [a] ++ lay2 x, otherwise

restspec cs ms
    = ["$con" ++ "\n" ++ "(wire " ++ (wspec2 ms) ++
            " )" ++ "\n"
            ++
            "$con" ++ "\n"
            ++
            "(para" ++ " " ++ (pspec c ms) ++ ")" | c <- cs]

wspec ms = show (rep (#(ms)) (take 10 [1..]))

wspec2 ms = show (take (#(ms)) (ltof10 [1..]))

ltof10 [] = []
ltof10 ls = [take 10 ls] ++ ltof10 (drop 10 ls)

encoder spec p m i
    = PARA ([lay2 (p ++ " " ++ ":")]}
++
(enc_restspec3 [16] m) ++ [";"] , i <= 9
= PARA ([lay2 (p ++ " " ++ "=")]
++
(enc_restspec1 m) ++ ["$con"]
++
(enc_restspec2 m) ++ [";"], otherwise

enc_restspec1 ms
= ["(wire " ++ wspec4 ms ++ " " ) " ++ "\n"
++
"$con" ++ "\n"
++
"(para" ++ " " ++
pspec 16 (concat (transpose (rep 2 ms))) ++ " ) " ]

enc_restspec2 ms
= ["(wire " ++ (lay2 (wspec3 ms)) ++ " " ) " ++ "\n"
++
"$con" ++ "\n"
++
"(para" ++ " " ++ (pspec 512 ms) ++ " ) " ]

enc_restspec3 cs ms
= ["(wire " ++ (wspec2 ms) ++ " " ) " ++ "\n"
++
"$con" ++ "\n"
++
"(para" ++ " " ++ (pspec c ms) ++ " ) " | c <- cs]

wspec3 ms
= show ( take (#(ms)) (lst10 (choplstto5 [1..])))

lst10 [] = [[]]
lst10 ls = [flip2 a b] ++ (lst10 (drop 10 ls))
    where
a = take 5 ls
b = take 5 (drop 5 ls)

flip2 a b = b ++ a

choplstto5 [] = []
choplstto5 ls = take 5 (drop 5 ls) ++ (choplstto5 (drop 10 ls))

wspec4 ms = show (take (#(ms) * 2) (1stof10 [1..]))

scalarspec p m o
  = PARA (1:2 (p ++ " " ++ "="))
    ++
    (scalarspec m) ++ [";"]) , o <= 10
  = error "Number of output bits is greater than 10", otherwise

scalarspec :: [num] -> [[char]]
scalarspec ms
  = ["wire " ++ "[" ++ show (take 40 [1..]) ++ "]" ++ "]"
    ++
    ["$con"]
    ++
    ["(para " ++ "[" ++ "scalar " ++ (show ms) ++ "]" ++ "])"

fadltopara file = layout (fadls [[], (newwords (read file))])

layout [] = show "empty output from fadlstruct"
layout [[[PARA x], y]] = lay x
layout [(x,y)] = show x

‘run’ takes two files as arguments, the first one is the file
that contains ‘FADL’ specification, and the other one is the output
which contains the executable ‘PARA’ specification.

run infile outfile = [Tofile outfile (fadltopara infile)]
startsims filespec = [System "/usr/openwin/bin/xview/cmdtool
             mira filespec", Exit 0]

convert fadlspec
    = [Tofile "IF.m" (fadltopara fadlspec),
        Closefile "IF.m",
        Tofile "IFMDL.m" (fadltomdl fadlspec),
        Closefile "IFMDL.m",
        Tofile "SIMPARA.m"
            (lay ["\%include \"5bitcell.m\"
                  ++ "\n" ++
                  "\%include \"pri_lib.m\"
                  ++ "\n" ++
                  "\%include \"sec_lib.m\"
                  ++ "\n" ++
                  "\%include \"sigcell.m\"
                  ++ "\n" ++
                  "\%include \"circ0.m\"
                  ++ "\n"]),
        System "/usr/openwin/bin/xview/textedit IF.m",
        Tofile "SIMPARA.m" (read "IF.m"),
        Closefile "SIMPARA.m"
]

fadltomdl file = layout_mdl (fadls [[[]], newwords (read file))]

layout_mdl [[(PARA x, ROM_CONTENTS y), z]] = lay y
5.5 PRIMARY FUNCTIONS FOR CONSTRUCTING PARA LANGUAGE : PRI_lib.m

signal ::= L | H | U

stream ::= [signal]

|==================================|
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
| PRIMITIVE FUNCTIONS             |
|                                 |
|                                 |
| |==================================|

------------- Wire function -------------

wire:::[num]-->[stream]-->[[stream]]
wire components stream
= [g comp stream | comp <- components]
where
  g comp stream
  = [f comp sig | sig <- stream]
  where
    f comp sig = [sig!(index - 1) | index <- comp]

------------- Parallel Processing Function -------------

para:::[*->[[signal]]]-->[*]-->[[signal]]

para:::[signal->[stream]]->stream->[stream]
para fs streams
= map concat (transpose [[(fs!n) (streams!n)]
  | n <- [0..(#fs) -1]])

------------- Con Function -------------

con:::(*->**)-->(**->***)-->*--->***
con f g = g . f

|==================================|
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
|                                 |
| AUXILIARY FUNCTIONS             |
|                                 |
|                                 |
| |==================================|

------------- execute circuit -------------
```
|| Execute Circuit Function

|| Format:
|| executecircircuit circuit inputfile outputfile

|| To execute:
|| executecircircuit fir "inile" "outile"

executecircircuit::
    ([signal]-->[signal])-->[char]-->[char]-->[sys_message]
executecircircuit circuit [] outsig
   = error "Input file is not specified"
executecircircuit circuit insig []
   = error "Output file is not specified"
executecircircuit circuit insig outsig
   = [ Tofile outsig (outfchar outputsignals)]
      where
      outputsignals = circuit (inftosig (read insig))

|| convert character type to signal type (input stream)
||

inftosig infilestring
    = map (map chartosignal) (lines infilestring)
chartosignal 'H' = H
chartosignal 'L' = L
chartosignal 'U' = U

|| Convert signal type to character type (output file)
||

outfchar outfilestring
    = lay (map (map signaltochar) outfilestring)
signaltochar H = 'H'
signaltochar L = 'L'
signaltochar U = 'U'

|| RUN SIMULATION
|| To execute: sim "namefile" "outfile"

sim cn nf df
    = [Tofile "pf.ps"
       (lay ([read "plot_prolog.ps"] ++
            pt3 cn nf df)), Closefile "pf.ps",
```
System "pgview pf.ps", System "rm pf.ps"

pt1nf nf
    = map concat ((map cvt1) (take10 (lines (read nf))))
pt2df df
    = map concat (take10 (map cvt2 (transpose (lines (read df)))))
take10 [] = []
take10 ls = [take 10 ls] ++ (take10 (drop 10 ls))

pt3 cn nf df
    = foldr (++) []
        [ [ "(" ++ cn ++ ")" ++ "Title"] ++ [n] ++ [d]
        ++ ["Clock"] ++ ["showpage"]
        | (n, d) <- zip2 (pt1nf nf) (pt2df df) ]

||   PLOT SIMULATION
|| To execute: plotsim "namefile" "outfile"

plotsim cn nf df
    = [Tofile "pf.ps"
        (lay ([read "plot_prolog.ps"] ++
        pt3 cn nf df)), Closefile "pf.ps",
        System "lpr -Plw pf.ps", System "rm pf.ps"
    ]

pt1 file
    = foldr (++) [] (cvt1 (lines (read file)))

pt2 file
    = lay [ cvt2 a | a <- (transpose (lines (read file))) ]

cvt1 ls = ["["] ++ [f ls] ++ ["]"] ++ ["PinName"]
    where
    f [] = []
    f (a:as) = ("(" ++ a ++ ")") ++ (f as)
cvt2 [] = []
cvt2 ls = foldr (++) [] (["["] ++
    [ foldr (++) [] [ f a | a <- ls ] ]
    ++ ["]"] ++ ["Display"] )
    where
    f a = ("(" ++ [a] ++ ")")

||   || Transpose the content of input file ,
||   || then create files in binary type from
||   || character type
||   ||
chartobin infilename outfilename
   = [Tofile outfilename
       (outbin (transpose (inftobin (read infilename))))]

||   Convert from character type to binary type

inftobin infilestring
   = map (map signaltobin) (lines infilestring)
signaltobin 'L' = 0
signaltobin 'H' = 1
signaltobin 'U' = 9

||   Convert from binary type to character type

outbin outfilestring
   = lay (map (map bintochar) outfilestring)
bintochar 0 = '0'
bintochar 1 = '1'
bintochar 9 = '9'

||       Output to a file       ||

outfile::[num]->[sys_message]
outfile ls = [Tofile "outfile" ((format.print) ls)]

print l = [ (a,l!(a-l)) ] a<-[1..#l] ]

format :: [(num,num)] -> [char]
format [] = []
format [(x,y):l]
   = (show x) ++ " " ++ (show y) ++
     label ++ "\n" ++ format !
   where
     label
       = " " ++ "\" ++ (show x) ++ "\"

||-----------binary to decimal-----------||

bintodec = reverse_lst $con dec
dec [] = 0
dec (a:x) = a + 2 * (dec x), a = 1 \ a = 0
   = error "not binary number", otherwise

reverse_lst x = rev x []
rev [] y = y
rev (a:as) y = rev as (a:y)
decimal to binary

```
dtobin n
  = reverse_lst . map (mod 2) . take n . iterate (div 2)

dtobc dec
  = (reverse_lst . map (mod 2) . take n . iterate (div 2)) dec
    where
    (n, y) = [(x, y) | (x, y) <- [(i,2^i)|i<-[0..]]; y >= dec+1]!0

dectobin = bin $con reverse_lst
bin = map (mod 2) . take 4 . iterate (div 2)
  || change 4 -> 5 for different size

dectobin5
  = bin $con reverse_lst
    where
    bin = map (mod 2) . take 5 . iterate (div 2)

--binary to High-Low--

binto.hl [] = []

binto.hl (b:bs)
  = H : binto.hl bs, b = 1
  = L : binto.hl bs, b = 0
  = error "unrecognizable character", otherwise

--elect to binary--

hltobin [] = []

hltobin (sig:sigs)
  = 1 : hltobin sigs, sig = H
  = 0 : hltobin sigs, sig = L
  = error "non-digital signal found", otherwise

--elect to decimal--

hltodec = hltobin $con bintodec

--decimal to High-Low--

dectohl :: num -> [signal]
dectohl = dectobin $con binto.hl

-- Input/Output functions--
write_file infile [] = []
write_file infile ls = [Tofile infile (lay ls)]

input_pad_name = write_file
output_pad_name = write_file
input_signal = write_file

||************ END OF PROGRAM ************||
5.6 STANDARD FUNCTIONS : SEC_lib.m

#include "pri_lib.m"

|| ------- Buffer ------- ||
buf in = [ out | out <- in ]
buffer = map buf

|| ------- D-type single phase flip flop ------- ||
dff [ck,d] = [d] ++ notgate d, ck = H

|| ------- Latch ------- ||
lat = map lat0
lat0 l = [L] : l

|| --- mux2 is a two input active multiplexor --- ||
mux2 = map mux0
mux0 [sb,a,b] = [L], a = L \ b = L
 = [H], a = H \ b = H

|| ------- n-transistor ------- ||
n_trans0 [in] = [ L ], in = H
 = [ U ], otherwise

n_trans = map n_trans0

|| -------------- Nand Gate -------------- ||
nand_2:[[signal]-->[[signal]]
nand_2 = map nand02
nand02 [a,b] = [ U ], a = U \ b = U
 = [ H ], a = L \ b = L
 = [ L ], otherwise
nand_3::[[signal]]->[[[signal]]
nand_3 = map nand03

nand03 [a,b,c] = [ U ], a = U \ b = U \ c = U
   = [ H ], a = L \ b = L \ c = L
   = [ L ], otherwise

nand_4::[[signal]]->[[[signal]]
nand_4 = map nand04

nand04 [a,b,c,d]
   = [ U ], a = U \ b = U \ c = U \ d = U
   = [ H ], a = L \ b = L \ c = L \ d = L
   = [ L ], otherwise

|| --------------- Nor Gate --------------- ||

nor_2::[[signal]]->[[[signal]]
nor_2 = map nor02

nor02 [a,b] = [ U ], a = U \ b = U
   = [ L ], a = H \ b = H
   = [ H ], otherwise

nor_3::[[signal]]->[[[signal]]
nor_3 = map nor03

nor03 [a,b,c] = [ U ], a = U \ b = U \ c = U
   = [ L ], a = H \ b = H \ c = H
   = [ H ], otherwise

nor_4::[[signal]]->[[[signal]]
nor_4 = map nor04

nor04 [a,b,c,d] = [ U ], a = U \ b = U \ c = U \ d = U
   = [ L ], a = H \ b = H \ c = H \ d = H
   = [ H ], otherwise

|| --------------- p-transistor --------------- ||
p_trans0 [in] = [ H ], in = L
   = [ U ], otherwise

p_trans = map p_trans0
|| -------- Exclusive Or -------- ||

xor02 [a,b] = { U }, a = U \( \lor \) b = U
   = { L }, (a = H \& b = H) \( \lor \) (a = L \& b = L)
   = { H }, otherwise

xor_2 : [[signal]] -> [[signal]]
xor_2 = map xor02

|| -------- And Gate -------- ||

and2 = and_2
or2 = or_2
inv = notgate

and_2 : [[signal]] -> [[signal]]
and_2 = map and02

and02 [a,b] = { U }, a = U \( \lor \) b = U
   = { H }, a = H \& b = H
   = { L }, otherwise

and_3 : [[signal]] -> [[signal]]
and_3 = map and03

and03 [a,b,c] = { U }, a = U \( \lor \) b = U \( \lor \) c = U
   = { H }, a = H \& b = H \& c = H
   = { L }, otherwise

and_4 : [[signal]] -> [[signal]]
and_4 = map and04

and04 [a,b,c,d]
   = { U }, a = U \( \lor \) b = U \( \lor \) c = U \( \lor \) d = U
   = { H }, a = H \& b = H \& c = H \& d = H
   = { L }, otherwise

|| -------- Or Gate -------- ||

or_2 : [[signal]] -> [[signal]]
or_2 = map or02

or02 [a,b] = { U }, a = U \( \lor \) b = U
   = { L }, a = L \& b = L
or_3::[[signal]]->[[[signal]]]
or_3 = map or03

or03 [a,b,c] = [ U ], a = U \ b = U \ c = U
    = [ L ], a = L & b = L & c = L
    = [ H ], otherwise

or_4::[[signal]]->[[[signal]]]
or_4 = map or04

or04 [a,b,c,d]
    = [ U ], a = U \ b = U \ c = U \ d = U
    = [ L ], a = L & b = L & c = L & d = L
    = [ H ], otherwise

|| ------------------ Inverter ------------------ ||

notbit::[[signal]]->[[[signal]]]
notbit [H] = [L]
notbit [L] = [H]
notbit [U] = [U]

notgate::[[signal]]->[[[signal]]]
notgate = map notbit

|| -------------- Gate With Delay -------------- ||

latch_gate gate delay ls = (mklist delay) ++ ( gate ls )

mklist 0 = []
mklist n = [ U ] : ( mklist (n-1) )

|| ------ Half Adder with Delay ------ ||

half_adder2
    = (wire [[1,2], [1,2]] )
    $con
    (para [latch_gate and_2 1 , latch_gate xor_2 1])

|| ----- Simple Circuit with Delay ------- ||

cir_wd
    = (wire [ [1,2], [3,4], [5,6] ] )
    $con
(para [ latch_gate and_2 1, latch_gate and_2 1,
latch_gate and_2 1 ] )
$con
(wire [ [1], [2,3] ] )
$con
(para [ id, latch_gate or_2 1 ] )
$con
(wire [ [1,2] ] )
$con
(para [ latch_gate and_2 1 ] )
$con
(wire [ [1] ] )
$con
(para [ latch_gate notgate 1 ] )

---------- Simple Circuit ----------

The number indicates the positions correspond
to the function in the input stream.

cir::[[signal]]->[[[signal]]]
cir = (wire [ [1,2], [3,4], [5,6] ] )
$con
(para [ and_2, and_2, and_2 ] )
$con
(wire [ [1], [2,3] ] )
$con
(para [ id, or_2 ] )
$con
(wire [ [1,2] ] )
$con
(para [ and_2 ] )
$con
(wire [ [1] ] )
$con
(para [ notgate ] )

cir0 = (wire [ [1,2], [3,4], [5,6] ] )
$con
(para [ and_2, and_2, and_2 ] )

test_cir = (wire [ [1,2], [3,4,5,6] ])
$con
(para [ and_2, cir1 ] )
cirl = (wire [ [1,2], [3,4] ] )
  $con
  (para [ and_2, and_2 ] )

||--- Half Adder Using 1-andg ate and 1 xor gate ---||

half_adder = (wire [[1,2], [1,2]] )
  $con
  (para [and_2, xor_2])

|| Full Adder Implementing by using Half_adder Above ||

full_adder2
  = (wire [[1], [2,3] ] )
     $con
     (para [id, half_adder ] )
     $con
     (wire [[2], [1,3] ] )
     $con
     (para [id, half_adder ] )
     $con
     (wire [[1,2], [3] ] )
     $con
     (para [or_2, id ] )

|| Full Adder Implementation using Nand gates only ||

full_adder
  = (wire [[1], [2], [2,3], [3] ] )
     $con
     (para [id, id, nand_2, id ] )
     $con
     (wire [[3], [1], [2,3], [3,4] ] )
     $con
     (para [id, id, nand_2, nand_2 ] )
     $con
     (wire [[1], [2], [3,4] ] )
     $con
     (para [id, id, nand_2 ] )
     $con
     (wire [[1], [2], [2,3], [3] ] )
     $con
     (para [id, id, nand_2, id ] )
     $con
     (wire [[1,3], [2,3], [3,4] ] )
     $con
(para [nand_2, nand_2, nand_2 ] )
$con
(wire [[1], [2,3] ] )
$con
(para [id, nand_2 ] )

|| Multiplexor Implementation using Nandgate  ||

mux = (wire [[1,2], [3,4] ] )
$con
(para [nand_2, nand_2] )
$con
(wire [[1,2] ] )
$con
(para [nand_2] )

|| --- Full Adder implementation using Mux ---  ||

full_adder3
= (wire [[1], [2], [2,3], [3] ] )
$con
(para [id, id, nand_2, id ] )
$con
(wire [[3], [1], [2,3,3,4] ] )
$con
(para [id, id, mux ] )
$con
(wire [[1], [2], [2,3], [3] ] )
$con
(para [id, id, nand_2, id ] )
$con
(wire [[1,3], [2,3,3,4] ] )
$con
(para [nand_2, mux ] )

|| ------- bit sequences -------  ||

bit_seq_2 = [[0,0],[0,1],[1,0],[1,1]]
bit_2 = map bintohl bit_seq_2
bit_seq_3 = [[0,0,0],[0,0,1],[0,1,0],[0,1,1],
            [1,0,0],[1,0,1],[1,1,0],[1,1,1]]
bit_3 = map bintohl bit_seq_3

|| ------- End Of Program -------  ||
5.7 SPECIAL PURPOSE CELL DEFINITION : cell.m

%include "pri_lib.m"
%include "sec_lib.m"

%include "pri_lib.m"
%include "sec_lib.m"

|| -------    Steering Switch    ------- ||

||------------------------------------------------------------------------||
|| "st_sw" simulates the function of a steering                           ||
|| switch.                                                              ||
||------------------------------------------------------------------------||

st_sw ls
  = (drop 1 $con take (#ls div 2)) ls, ls!0 = L
  = (drop 1 $con drop (#ls div 2)) ls, otherwise

steering_sw :: [[signal]] -> [[signal]]
steering_sw = map st_sw

ssw [sig, ctl] = [sig], ctl = H
  = [], otherwise

sssw = map ssw

|| -------    Latch    ------- ||

|| defined in sec_lib.m ||
table :: [num] -> [[[signal]]]
table = maketable 4 8

maketable row col ls
  = [map cvtfn signal | signal <- f row col ls]
  where
    cvtfn = (dectobin $con reverse_1st $con bintohl)
f r c [] = []
f r c ls = take r ls : f r (c-1) (drop r ls), c > 1
          = [], otherwise

rom_cr modi coeff displ
  = table (multi_coeff modi (coeff * 2 ^ displ))

Rom contents are created column-wise

lookup :: [[[signal]]] -> [[[signal]]] -> [[[signal]]]
lookup rom_table ls
  = [ decoder rom_table a | a <- ls ]

decoder :: [[[signal]]] -> [signal] -> [signal]
decoder rom_table ls
  = ((rom_table!row)!col)
  where
    row = hltodec [ls!4,ls!3,ls!2]
    col = hltodec [ls!1,ls!0]

Definition Of Cell
make_cell rom_table
= (wire[[2],[3],[4],[5],[1],[1],[6],[7],[8],[9],[10],[6,7,8,9,10]])
 $con
 (para [id,id,id,id,id,notgate,id,id,id,id,lookup rom_table])
 $con
 (wire [[1],[2],[3],[4],[5],[7,6],[8,6],[9,6],[10,6],[11,6],
        [12,5],[13,5],[14,5],[15,5],[16,5]]
 )
 $con
 (para [id, id, id, id, lat0, sssw, sssw, sssw, sssw, sssw,
         sssw, sssw, sssw, sssw, sssw ] )
make_sys_cell :: num -> num -> [[signal]] -> [[signal]]

make_sys_cell modx coeff
  = (wire [ [1,2,3,4,5,6,7,8,9,10] ] )
    $con
    (para [ make_cell (rom_cr modx coeff 0) ] )
    $con
    (wire [ [1,2,3,4,5,6,7,8,9,10] ] )
    $con
    (para [ make_cell (rom_cr modx coeff 1) ] )
    $con
(wire [ 1,2,3,4,5,6,7,8,9,10 ] )
\$con
(para [ make_cell (rom_cr modx coeff 2) ] )
\$con
(wire [ 1,2,3,4,5,6,7,8,9,10 ] )
\$con
(para [ make_cell (rom_cr modx coeff 3) ] )
\$con
(wire [ 1,2,3,4,5,6,7,8,9,10 ] )
\$con
(para [ make_cell (rom_cr modx coeff 4) ] )

msc m c = make_sys_cell m c
mc m c d = make_cell (rom_cr m c d)

scalar_cell_1 m round
 = map (decoder (table (multi_coeff m round)))

scalar_cell_2 modx1 modx2
 = map (decoder (table (scalar_with_rounding modx1 modx2)))

ccts file = [ map chartosignal x | x <- lines (read file)]

||
|| encoder with less than or equal 9 bits of input
||

|| x x y y x x y y
|| 4 0 3 0 0 4 0 4
|| eg: 193 = [0,1,1,1,0,0,0,0,0,0,0] --> [l,l,h,h,l,h,l,l,l,l] || x x
|| 8 0
||
|| encoder9 [[[l,l,h,h,l,h,l,l,l], [h,l,h,h,l,h,l,h,l,l]],
\[ \text{display}_\text{in}\_\text{decimal} \implies [(0,1), (12,18), (13,3), (14,20)] \]

dtoe dec

\[ = [\text{bintohl} \ (\text{reverse}_\text{lst} \ (\text{take 5 bitbin}) ++ \\text{reverse}_\text{lst} \ (\text{drop 5 bitbin}) ++ [0])) \]

where

\[ \text{bitbin} = \text{dtonb} \ 9 \ \text{dec} \]

take5 [] = []

take5 1s = take 5 1s ++ (drop 5 1s), 1s \equiv []

testend9 dec = \text{display}_\text{in}\_\text{decimal} \ (\text{encoder9} \ (\text{dtoe} \ \text{dec}))

\text{encoder9} = (\text{wire} \ [ [1,2,3,4,5,6,7,8,9,10] ])

\text{\textcolor{red}{\text{\$con}}}

(\text{para} \ [\text{msc} \ 32 \ 16])

\text{br} = [(i, 2^i)|i<0.19]

\text{encoder18 m}

\[ = (\text{wire} \ [[1,2,3,4,5,6,7,8,9,10], [11,12,13,14,15,16,17,18,19,20]]) \]

\text{\textcolor{red}{\text{\$con}}}

(\text{para} \ [\text{msc m 16}, \text{msc m 16}])
$con
(wire [ [16,17,18,19,20,6,7,8,9,10] ])
$con
(para [ msc m 512 ])) || 2^7 ==> 128

cvt18bit dec
  = bintohl (num bit1 ++ num bit2)
  where
    num bit
      = (reverse_lst (take 5 bit)
          ++ reverse_lst (drop 5 bit) ++ [0])
    bit2 = take 9 (dtobn 18 dec)
    bit1 = drop 9 (dtobn 18 dec)

  ||
  || e.g. test18 [143564,123455,219474,153363]
  || ==> [(0,12),(24,31),(17,18),(12,19)]
  ||

test18 m = display_in_decimal . encoder18 m. map cvt18bit

  ||
  || SCALAR
  ||

scalar4
  = (wire [ [1..10], [11..20], [21..30], [31..40] ])
$con
(para [ msc 32 16, msc 31 16, msc 29 16, msc 27 16 ])
$con
(wire [ [1..10], [11..20], [21..30] ])
$con
(wire [ [1..10], [11..20] ])
$con
(para [ msc 32 16, msc 31 16 ])
$con
(wire [ [16,17,18,19,20,6,7,8,9,10] ])
$con

||
|| PRE-MULTIPLIER
||

pl m0 m1 = add_inverse m1 m0
5.8 WAVEFORM POSTSCRIPT PROLOGUE : plot.ps

%! 
/t xpos 72 def % initial x point of title
/vdis 48 def % vertical distance
/wd 9 def % width of timing diagram
/ht 12 def % height of timing diagram
/htd ht neg def % negation of height

/Times-Roman findfont 15 scalefont setfont

% -------- Define Procedure -------- %

/InitParm
{/typos 672 def % initial y point of title
/initypos 672 def % initial y position for diagram
/ypos 672 def % y starting coordinate of graph
.5 setlinewidth
} def

/FontSize {/Times-Roman findfont exch scalefont setfont} def

>Title {24 FontSize 144 720 moveto show InitParm} def

/PinName % Print PinName
{15 FontSize
t xpos typos moveto % Go to initial position
{show % Show signal name
/typos vdis sub def % Go to next position
t xpos typos moveto} forall % Print all titles in array
} def

/Display % Print timing diagram
{newpath
/xpos 144 def % x starting coordinate of graph
/countsignal 1 def % Accumulates same the signal
}
/idx 1 def % Array index
dup /prevsignal exch 0 get def % Previous signal
(prevsignal eq (/ypos ypos ht add def) if
  % Is the first signal H?
xpos ypos moveto
  % Initial x-y position
dup /lengthofsignal exch length 1 sub def
  % Get the length of array for looping

lengthofsignal
%
% Main Program Loop
%
{
  dup idx get prevsignal eq
  (/countsignal countsignal 1 add def) if
    % Same signal encountered
dup idx get prevsignal gt
  {dup /prevsignal exch idx get def OnetoZero} if
  dup idx get prevsignal lt
  {dup /prevsignal exch idx get def ZerotoOne} if
  /idx idx 1 add def
} repeat

DrawHorBar
/initypos initypos vdis sub def
/ypos initypos def
stroke pop
} def

/ZerotoOne
{DrawHorBar DrawVerUp} def

/OnetoZero
{DrawHorBar DrawVerDown} def

/DrawHorBar
{wd countsignal mul 0 rlineto
 /countsignal 1 def} def

/DrawVerUp
{0 ht rlineto
 /ypos ypos ht add def} def

/DrawVerDown
{0 htd rlineto
 /ypos ypos htd add def} def

/Clock
{newpath
 /lengthofsignal lengthofsignal 1 add def
 /xpos 144 def
 /counter 1 def
 txpos ypos moveto (Clock) show
 xpos ypos moveto
 {counter lengthofsignal le {ZerotoOne} {exit} ifelse
 /counter counter 1 add def
counter lengthofsignal le {OnetoZero} {exit} ifelse
 /counter counter 1 add def} loop
 stroke} def
5.9 PARA TO EDIF TRANSLATOR: PARATOEDIF.m

This is a translator from the "PARA" specification language to EDIF description language in the NETLIST level.
This translator is implemented using the W/AGE tool developed at the University of Windsor.
The translator function is called "run". This function takes three arguments, a set of input identifiers, a set of output identifiers, and a file which contains a circuit specification.

#include "ambi_wage.021890.m"
#include "CMOS3LIB.m"

A SPECIFICATION OF AN EXECUTABLE ATTRIBUTE GRAMMAR

rstc = wage_recognises name

u = wage_uparrow name
d = wage_downarrow name
rule = wage_rule

INPUT GRAMMAR FOR THE PARA SPECIFICATION LANGUAGE

paraspecs
  = rstc (s1 paraspec ++ s2 paraspecs)
    [rule 0.1 (EDIF $u lhs) EQ conc_edif [EDIF $u s1,
EDIF $u s2],
rule 0.2 (INLWIRE $d s1) EQ same_as [INLWIRE $d lhs ],
rule 0.3 (OUTLWIRE $d s1) EQ same_as [OUTLWIRE $d lhs ],
rule 0.4 (INLWIRE $d s2) EQ same_as [INLWIRE $d lhs ],
rule 0.5 (OUTLWIRE $d s2) EQ same_as [OUTLWIRE $d lhs ]
$excl_orelse
rstc (s1 paraspec)

[rule 0.6 (EDIF $u lhs) EQ same_as [EDIF $u s1],
rule 0.2 (INLWIRE $d s1) EQ same_as [INLWIRE $d lhs ],
rule 0.3 (OUTLWIRE $d s1) EQ same_as [OUTLWIRE $d lhs ]]
paraspec
= rstc (s1 circdecl)

[rule 1.1 (EDIF $u lhs) EQ get_text[INLWIRE $d lhs,
OUTLWIRE $d lhs,
CIRCNAME $u s1,
INSTDATA $u s1,
INSTABLE $u s1,
LNET $u s1 ],
rule 1.2 (INLWIRE $d s1) EQ same_as [INLWIRE $d lhs ],
rule 1.3 (OUTLWIRE $d s1) EQ same_as [OUTLWIRE $d lhs ]]
circcdecl
= rstc (s1 circ_name ++ s2 (uterm ":" ) ++
s3 slices ++ s4 (uterm ":" ))

[rule 2.1 (INSTABLE $u lhs) EQ same_as [INSTABLE $u s3],
rule 2.2 (INSTDATA $u lhs) EQ same_as [INSTDATA $u s3],
rule 2.3 (LNET $u lhs) EQ finish_net [LNET $u s3,
OUTLWIRE $u s3,
OUTLWIRE $d lhs ],
rule 2.4 (CIRCNAME $u lhs) EQ same_as [CIRCNAME $u s1],
rule 2.5 (INSTABLE $d s3) EQ init_instable[INLWIRE $d lhs,
OUTLWIRE $d lhs ],
rule 2.6 (NETNUM $d s3) EQ init_netnum[],
rule 2.7 (INLWIRE $d s3) EQ same_as [INLWIRE $d lhs ]]
slices

\[
\text{rsc} \ (s1 \ \text{slice} \ ++ \ s2 \ (\text{uterm} \ "\$con") \ ++ \ s3 \ \text{slices})
\]

- rule 3.8 (INSTABLE $u \ \text{lhs}) \ EQ \ same\_as \ [\text{INSTABLE} \ $u \ s3],
- rule 3.9 (INSTDATA $u \ \text{lhs}) \ EQ \ \text{app\_instdata} \ [\text{INSTDATA} \ $u \ s1,
\text{INSTDATA} \ $u \ s3],
- rule 3.10 (LNET $u \ \text{lhs}) \ EQ \ \text{app\_lnet} \ [\text{LNET} \ $u \ s1,
\text{LNET} \ $u \ s3],
- rule 3.11 (OUTLWIRE $u \ \text{lhs}) \ EQ \ same\_as \ [\text{OUTLWIRE} \ $u \ s3],
- rule 3.12 (INSTABLE $d \ s1) \ EQ \ same\_as \ [\text{INSTABLE} \ $d \ \text{lhs}],
- rule 3.13 (INLWIRE $d \ s1) \ EQ \ same\_as \ [\text{INLWIRE} \ $d \ \text{lhs}],
- rule 3.14 (NETNUM $d \ s1) \ EQ \ same\_as \ [\text{NETNUM} \ $d \ \text{lhs}],
- rule 3.15 (INSTABLE $d \ s3) \ EQ \ same\_as \ [\text{INSTABLE} \ $u \ s1],
- rule 3.16 (INLWIRE $d \ s3) \ EQ
\text{conv\_out\_to\_in\_lw} \ [\text{OUTLWIRE} \ $u \ s1],
- rule 3.17 (NETNUM $d \ s3) \ EQ \ \text{add\_net\_nums} \ [\text{NETNUM} \ $d \ \text{lhs},
\text{LNET} \ $u \ s1])

$excl\_orelse

\[
\text{rsc} \ (s1 \ \text{slice})
\]

- rule 3.1 (INSTABLE $u \ \text{lhs}) \ EQ \ same\_as \ [\text{INSTABLE} \ $u \ s1],
- rule 3.2 (INSTDATA $u \ \text{lhs}) \ EQ \ same\_as \ [\text{INSTDATA} \ $u \ s1],
- rule 3.3 (LNET $u \ \text{lhs}) \ EQ \ same\_as \ [\text{LNET} \ $u \ s1],
- rule 3.4 (OUTLWIRE $u \ \text{lhs}) \ EQ \ same\_as \ [\text{OUTLWIRE} \ $u \ s1],
- rule 3.5 (INSTABLE $d \ s1) \ EQ \ same\_as \ [\text{INSTABLE} \ $d \ \text{lhs}],
- rule 3.6 (INLWIRE $d \ s1) \ EQ \ same\_as \ [\text{INLWIRE} \ $d \ \text{lhs}],
- rule 3.7 (NETNUM $d \ s1) \ EQ \ same\_as \ [\text{NETNUM} \ $d \ \text{lhs}]}

slice

\[
\text{rsc} \ (s1 \ \text{wiring} \ ++ \ s2 \ (\text{uterm} \ "\$con") \ ++ \ s3 \ \text{para})
\]

- rule 4.1 (INSTABLE $u \ \text{lhs}) \ EQ \ same\_as \ [\text{INSTABLE} \ $u \ s3],
- rule 4.2 (INSTDATA $u \ \text{lhs}) \ EQ \ \text{make\_inst\_data} \ [\text{LCSPEC} \ $u \ s3],
- rule 4.3 (LNET $u \ \text{lhs}) \ EQ \ \text{make\_nets} \ [\text{NETNUM} \ $d \ \text{lhs},
\text{OUTLWIRE} \ $u \ s1,
\text{LCSPEC} \ $u \ s3],
- rule 4.4 (OUTLWIRE $u \ \text{lhs}) \ EQ \ \text{get\_out\_wires} \ [\text{LCSPEC} \ $u \ s3],
- rule 4.5 (INLWIRE $d \ s1) \ EQ \ same\_as \ [\text{INLWIRE} \ $d \ \text{lhs}],
- rule 4.6 (INSTABLE $d \ s3) \ EQ \ same\_as \ [\text{INSTABLE} \ $d \ \text{lhs}])
wiring
  = rstc (s1 (uterm "(") ++ s2 (uterm "wire") ++
          s3 (uterm "]") ++ s4 substrams ++
          s5 (uterm ")") ++ s6 (uterm ")"))

  [rule 5.1 (OUTLWIRE $u lhs) EQ same_as [OUTLWIRE $u s4],
     rule 5.1 (INLWIRE $d s4) EQ same_as [INLWIRE $d lhs]]

substrams
  = rstc (s1 substram ++ s2 (uterm ",") ++ s3 substrams)
     [rule 6.3 (OUTLWIRE $u lhs) EQ conc_outlw [OUTLWIRE $u sl,
                                                OUTLWIRE $u s3],
     rule 6.4 (INLWIRE $d s1) EQ same_as [INLWIRE $d lhs],
     rule 6.5 (INLWIRE $d s3) EQ same_as [INLWIRE $d lhs]]

$excl_orelse
  rstc (s1 substram)
     [rule 6.1 (OUTLWIRE $u lhs) EQ make_outllw [OUTLWIRE $u s1],
     rule 6.2 (INLWIRE $d s1) EQ same_as [INLWIRE $d lhs]]

substram
  = rstc (s1 (uterm "[") ++ s2 wireposits ++ s3 (uterm "]")])
     [rule 7.1 (OUTLWIRE $u lhs) EQ same_as [OUTLWIRE $u s2],
     rule 7.2 (INLWIRE $d s2) EQ same_as [INLWIRE $d lhs]]

wireposits
  = rstc (s1 wireposits ++ s2 (uterm ",") ++ s3 wireposits)
     [rule 8.3 (OUTLWIRE $u lhs) EQ conc_outw [OUTWIRE $u sl,
                                                OUTLWIRE $u s3],
     rule 8.4 (INLWIRE $d s1) EQ same_as [INLWIRE $d lhs],
     rule 8.5 (INLWIRE $d s3) EQ same_as [INLWIRE $d lhs]]

$excl_orelse
  rstc (s1 wireposits)
     [rule 8.1 (OUTLWIRE $u lhs) EQ make_outlw [OUTWIRE $u s1],
     rule 8.2 (INLWIRE $d s1) EQ same_as [INLWIRE $d lhs]]

wireposits
= rstc (s1 wpos)
   [rule 9.1 (OUTWIRE $u lhs) EQ lookup_wire [WPOS $u s1,
            INLWIRE $d lhs]]

para
   = rstc (s1 (uterm "(") ++ s2 (uterm "para") ++
            s3 (uterm "]") ++ s4 gcnames ++
            s5 (uterm "]") ++ s6 (uterm "]"))
   [rule 10.1 (INSTABLE $u lhs) EQ same_as [INSTABLE $u s4],
    rule 10.2 (LCSPEC $u lhs) EQ same_as [LCSPEC $u s4],
    rule 10.3 (INSTABLE $d s4) EQ same_as [INSTABLE $d lhs]]

gcnames
   = rstc (s1 gcname2 ++ s2 (uterm ")) ++ s3 gcnames)
   [rule 11.4 (INSTABLE $u lhs) EQ same_as [INSTABLE $u s3],
    rule 11.5 (LCSPEC $u lhs) EQ cons_cspec [CSPEC $u s1,
            LCSPEC $u s3],
    rule 11.6 (INSTABLE $d s1) EQ same_as [INSTABLE $d lhs],
    rule 11.7 (INSTABLE $d s3) EQ same_as [INSTABLE $u s1]]

$excl_orelse
   rstc (s1 gcname2)
   [rule 11.1 (INSTABLE $u lhs) EQ same_as [INSTABLE $u s1],
    rule 11.2 (LCSPEC $u lhs) EQ make_cspec_list [CSPEC $u s1],
    rule 11.3 (INSTABLE $d s1) EQ same_as [INSTABLE $d lhs]]

gcname2
   = rstc (s1 libgcname)
   [rule 12.1 (CSPEC $u lhs) EQ gcspec_to_cspec[GCSPEC $u s1,
           INSTABLE $d lhs],
    rule 12.2 (INSTABLE $u lhs) EQ
       update_instable [GCSPEC $u s1,
           INSTABLE $d lhs]]

$orelse
   rstc (s1 libgcname ++ s2 arguments)
   [rule 13.1 (CSPEC $u lhs) EQ
       gcname_to_cspec3 [GCSPEC $u s1,
PGCNAME $u s2,  
INSTABLE $d lhs],

rule 13.2 (INSTABLE $u lhs) EQ
  update3_instable [GCSPEC $u s1,  
                   PGCNAME $u s2,  
                   INSTABLE $d lhs]]

arguments
  rstd (s1 modi ++ s2 (uterm "(") ++  
        s3 coeff ++ s4 (uterm ")"))
  [rule 16.1 (PGCNAME $u lhs) EQ  
      conc_makepgclst2 [PGCNAME $u s1,  
                       PGCNAME $u s3]]

$orelse
  rstd (s1 (uterm "[" ++ s2 listmodi ++ s3 (uterm "]"))
  [rule 16.2 (PGCNAME $u lhs) EQ same_as [PGCNAME $u s2]]

||
||  DICTIONARY
||

wpos_list = [(shownum n, [WPOS n]) | n < [1..90] ]

||
||  BASIC INTERPS
||

circ_name = translate_from_charstring_to CIRCNAMEN

translate_from_charstring_to con input
  = concat (map (tfct con) input)

tfct con (inh,[]) = []
tfct con (inh, (u:us))
  = [[(con u),us]], charstring u
  = [], otherwise
where

charstring [c] = ischar c
charstring (c:cs) = (ischar c) & (charstring cs)
ischar c = ('a' <= c <= 'z') \ (c = '_') \/
    (c = '\"') \/
    ('0' <= c <= '9') \ (chr <= c <= 'z')

libgcname = mkint lib_gc_name_list

wpos = mkint wpos_list

modi [(s, []]) = []
modi [(s, (w:ws))] = [[PGCNAME w], ws]]

coeff :: [[attributes], [[char]]] -> [[attributes], [[char]]]
coeff [(s, (w:ws))] = 
    [[PGCNAME (x++y)], z], x = "-"
    [[PGCNAME w], ws]], otherwise
where
    (x:y:z) = (w:ws)

lstofmodi
    = rstc (s1 modi ++ s2 (uterm ",") ++ s3 lstofmodi)
    [rule 6.2 (PGCNAME $u lhs) EQ conc_makepgclst [PGCNAME $u s1,
        PGCNAME $u s3]]

$excl_orelse
    rstc (s1 modi)
    [rule 6.1 (PGCNAME $u lhs) EQ same_as [PGCNAME $u s1]]

||
|| SPECIFYING RELEVANT ATTRIBUTE TYPES
||

attributes ::= WPOS num
    | CIRCNME gcname
    | GCSPEC (gcname,[inpiname],[outpiname])
| INSTABLE [(gcname, num)]
| CSPEC (cname, [inpinname], [outpinname])
| LCSPEC [(cname, [inpinname], [outpinname])]
| INSTDATA [cname]
| OUTWIRE [wire_label]
| INLWIRE [wire_label]
| OUTLWIRE [wire_label]
| OUTLLWIRE [(wire_label)]
| NETNUM num
| LNET [(num, (wire_label, wire_label))]
| EDIF [(char)]
| PGCNAME gcname

wire_label ::= OUTPORT [char]
| INPORT [char]
| QUALIFY cname pinname

pinname == [char]
inpinname == [char]
outpinname == [char]
gcname == [char]
cname == (gcname, num)

name (WPOS x) = "wpos"
name (CIRCNAMe x) = "circname"
name (GCspec x ) = "gcspec"
name (CSPEC x) = "cspec"
name (LCSPEC x) = "lcspec"
name (INSTABLE x) = "instable"
name (INSTDATA x) = "instdata"
name (OUTWIRE x) = "outwire"
name (INLWIRE x) = "inlwire"
name (OUTLWIRE x) = "outlwire"
name (OUTLLWIRE x) = "outllwire"
name (NETNUM x) = "netnum"
name (LNET x) = "lnet"
name (EDIF x) = "edef"
name (PGCNAME x) = "pgcname"

||
|| ATTRIBUTE FUNCTIONS
||

conc edif [EDIF x, EDIF y] = EDIF (x ++ y)

conc_outw [OUTWIRE x, OUTWIRE y] = OUTWIRE (x:y)

lookup_wire [WPOS n, INIWIRE y] = OUTWIRE (y ! (n -1))

make_outlw [OUTWIRE x] = OUTWIRE [x]

same_as [x] = x

conv_out_to_in_lw [OUTLWIRE x] = INLWIRE x

conc_outlw [OUTLWIRE x, OUTLWIRE y] = OUTLLWIRE (x:y)

make_outllw [OUTLWIRE x] = OUTLLWIRE [x]

gspec_to_cspspec [GCSPEC (x,y,z), INSTABLE t] =
    CSPEC ((x, instnum rec), y, z)
    where instnum x = 1, x = [],
         = (z!0) + 1, otherwise
    rec = [n | (gc, n) <- t; gc = x]

update_instable [GCSPEC (x, y, z), INSTABLE t]
    = INSTABLE ((x,1):t), (#[n | (gc, n) <- t; gc = x]) = 0
    = INSTABLE (p:q), otherwise
    where q = [(gc,n) | (gc,n) <- t; gc = x]
    p = (x, instnum + 1)
    instnum = [n | (gc, n) <- t; gc = x]!0
gcname_to_cspec3 [GCSPEC (cn,i,o), PGNAME p2, INSTABLE t]
  = gcspec_to_cspec [GCSPEC (x,i,o), INSTABLE t]
  where
      x = cn ++ "_" ++ p2

update3_instable [GCSPEC (cn,i,o), PGNAME p2, INSTABLE t]
  = update_instable [GCSPEC (x,i,o), INSTABLE t]
  where
      x = cn ++ "_" ++ p2

conc_makepgclst2 [PGNAME x, PGNAME y]
  = PGNAME ( x ++ "(" ++ y ++ ")")

conc_makepgclst [PGNAME x, PGNAME y]
  = PGNAME (x ++ "_" ++ y)

makepgclst [PGNAME p] = PGNAME (p++[])

make_cspec_list [CSPEC x] = LCSPEC [x]

cons_cspec [CSPEC x, LCSPEC y] = LCSPEC (x:y)

make_inst_data [LCSPEC l]
  = INSTDATA [cn | (cn, inp, outp) <- l]

get_out_wires [LCSPEC l]
  = OUTLWIRE (concat (map getwires l))
  where getwires (cn, inps, outps) =
          [QUALIFY cn pin | pin <- outps]

make_nets [NETNUM n, OUTLWIRE w, LCSPEC s] =
  INET (zip2 [(n+1) ..] nets)
  where
      nets = concat [convert (lw,circ) | (lw,circ) <- (zip2 w s)]
      where
convert (lw, (id, inps, outps)) =
    zip2 lw [QUALIFY id pin | pin <- inps]

finish_net [LNET n, OUTLWIRE x, OUTLWIRE y]
    = LNET (n ++ t)
      where
      t = zip2 [(#n + 1) ..] links
           where
           links = zip2 x y

init_instable [INLWIRE i, OUTLWIRE o]
    = INSTABLE ["inpadx",numi],("outpadx",numo),
              ("gpx",1),("ppx",1)]
      where
      numi = #(i)
      numo = #(o)

init_netnum [] = NETNUM 0

add_net_nums [NETNUM x, LNET y] = NETNUM (x + (#y))

app_instdata [INSTDATA d, INSTDATA m] = INSTDATA (d ++ m)

app_lnet [LNET m, LNET n] = LNET (m ++ n)


FUNCTIONS TO CONVERT FROM ATTRIBUTES TO EDIF TEXT

string_of_net (netnum, (inpin,outpin)) =
    " (net " ++ "NET" ++ (shownum netnum) ++
    "\n" ++
    " (joined\n" ++ (string_of_wlabel inpin) ++
    "\n" ++ (string_of_wlabel outpin) ++ "))

string_of_wlabel (IMPORT x)
string_of_wlabel (OUTPORT x)  
   = " 
       (portRef OUT (instanceRef " ++ x ++ "))"

string_of_wlabel (QUALIFY (nam, num) pin)  
   = " 
       (portRef "++pin++" (instanceRef " 
         ++nam++"_"++(shownum num)""_"++")"
    
    timestamp  
    = concat [yy, " ", mm, " ", dd, " ", tt]  
    where  
        yy = (ls!5)  
        mm = show ([y | (x, y) <- lstofmonth; x = (ls!1)]!0)  
        dd = (ls!2)  
        tt = time (ls!3)  
        ls = words (date!0)  
    
    date = [ x | (x, y, z) <- [system "date"]]

    time [] = []  
    time ls = concat [hrs, " ", min, " ", sec]  
    where  
        hrs = (ls!0) : [ls!1]  
        min = (ls!3) : [ls!4]  
        sec = (ls!6) : [ls!7]  

    lstofmonth = [("Jan",1),("Feb",2),("Mar",3),("Apr",4)S (),  
        ("May",5),("June",6),("July",7),("Aug",8),  
        ("Sep",9),("Oct",10),("Nov",11),("Dec",12)]

ediflevel  
    = lay ["   (edifLevel 0)"",  
        "   (technology",  
        "   (numberDefinition" ++ " "))"
    
    cmos3lib = "cmos3lib"
library = "FIR_Filer"

extractcell lstcname
  = ["  (cell " ++ lcname ++ "\n " ++
    (celltype GENERIC)\n " ++
    (view symbol)\n " ++
    (viewType NETLIST)\n " ++
    (interface" ++ checkioport inps outps ++
""))\n"]
  (cname, inps, outps) <- database;
  (lcname,n) <- lstcname;
  cname = take 3 lcname
  \ cname = take 6 lcname
  \ cname = lcname }

checkioport inps outps
  = inports inps ++ outports outps

inports inps
  = concat ["\n    (port " ++ inp ++
      " (direction input))" | inp <- inps ]

outports outps
  = concat [ "\n    (port " ++ outp ++
      " (direction output))" | outp <- outps ]

iports iops
  = concat [ "\n    (port " ++ iop ++
      " (direction InOut))" | iop <- iops ]

instance ls
  = ["  (instance " ++ i ++ "\n " ++
    (viewRef symbol)" ++
    (cellRef " ++ "inpadx")" ++
    (libraryRef " ++ "cmos3lib " ++
")\n"] | (IMPORT i) <- ls]
oinstance ls
   = [" (instance " ++ i ++ "\n" ++
        (viewRef symbol"
        " (cellRef " ++ "outpadx" ++
           (libraryRef " ++ cmos3lib ++
        ")\n" | (OUTPORT i) <- ls]

pginst
   = [" (instance vddcore"
        " (viewRef symbol"
        " (cellRef ppx"
           (libraryRef " ++ cmos3lib ++
        ")\n"
    ++
    [" (instance gndcore"
        " (viewRef symbol"
        " (cellRef gpx"
           (libraryRef " ++ cmos3lib ++
        ")\n"
    "])"

pgnet = pnet ++ gnet

pnet = [" (net " ++ io1 ++ "\n" ++
        (joined (portRef " ++ io2 ++
        " (instanceRef " ++ cname ++ " \n" |
        (cname, [io1], [io2]) <- database; cname = "ppx"
    )

gnet = [" (net " ++ io1 ++ "\n" ++
        (joined (portRef " ++ io2 ++
        " (instanceRef " ++ cname ++ " \n" |
        (cname, [io1, io2], s) <- database; cname = "gpx"
    )

get_text [INLWIRE i; OUTLWIRE o, CIRCNAME name,
         INSTDATA d, INSTABLE s, LNET l]
   = EDIF ["(EDIF " ++ name ++ ".edif",
          " (edifVersion 2 0 0)",
          " (edifLevel 0)"
]
" (keywordMap",
" (keywordLevel 0" ++ ")"),
" (status",
" {written ",
" (timestamp "++ timestamp ++ ")"),
" (program " ++ "\"" ++ name ++ "_program" ++ "\"" ++ " ) )))" ++ "\n",
" (external " ++ cmos3lib ++ "\n" ++ edificelevel ] ++
extractcell s ++ [" )\n"] ++
[" (library " ++ library ++ "\n" ++ edificelevel ++ "\n" ++
" (cell " ++ name,
" (celltype GENERIC)",
" (view netlist" ++ "\n" ++
" (viewType NETLIST)"
++]
[" (interface)"
++]
[" (contents\n"
++]
iinstance i
++
oinstance o
++
pгинст
++
(map string_of_inst d)
++
(map string_of_net l)
++
pгnet ++ [" \n \n \n \n\n")

string_of_inst (nam,num)
  = " " ++
    (instance " ++ nam ++ "." ++ (shownum num) ++ "\n"
" " ++
" (viewRef symbol " ++ "\n"
" (cellRef " ++ nam ++ "\n" ++
(libraryRef " ++ cmos3lib ++"))
"

author = getenv "USER"

layout [] = lay ["empty string"]
layout [[(EDIF x),y]] = lay x

|| THE TOP LEVEL FUNCTION

run inps outps file
   = layout res
   where
     res = paraspecs [[make_imports inps,
                      make_outports outps], newwords {read file}]

||
|| FUNCTIONS FOR MAKING IMPORTS AND OUTPORTS
||

make_imports ins = INLWIRE (map IMPORT ins)

make_outports outs = OUTLWIRE (map OUTPORT outs)

||
|| THE CELL LIBRARY READ FROM THE FILE "CMC_lib"
||

lib_gc_name_list
   = [(name,[GCSPEC (name, ins, outs)])
      | (name,ins,outs) <- database]

|| The Help facility

help
   = lay [" ", "Type:", " ",
          " netlist "inputname" "outputname" "paraspec" "]
" netlistout "inputname" "outputname"
"paraspec" "netlistfile" ")

netlist inputname outputname paraspec
    = run (lines (read inputname)) (lines (read outputname)) paraspec

netlistout inputname outputname paraspec netlistfile
    =[Tofile netlistfile(netlist inputname outputname paraspec)]
APPENDIX 6
FORMAL SYNTAX OF SPECIFICATIONS

6.1 BNF Notation of FADL

<fadls> ::= <fadl> <fadls>
       | <fadl>
<fadl> ::= <entity> <module>
       <floorplan> <mapblk>
<entity> ::= Entity <chipname>
<chipname> ::= [char]
<module> ::= Module <modname>
       [ <properties> ]
<modname> ::= [char]
<properties> ::= <purpose>, <moduli>,
       <aproperty>
<purpose> ::= Purpose <purposes>
<purposes> ::= encoder | isps | scalar
<moduli> ::= Moduli [ <listofmodi> ]
<listofmodi> ::= <modi>, <listofmodi>
       | <modi>
<modi> ::= num
<aproperty> ::= <input_bits>
       | <coefficients>
       | <output_bits>
<input_bits> ::= Input_bits <numofinput>
<numofinput> ::= num
<coefficients> ::= Coefficients [ <listofcoeff> ]
<listofcoeff> ::= <coeff>, <listofcoeff>
       | <coeff>
<coeff> ::= num
<output_bits> ::= Output_bits <numofoutput>
<numofoutput> ::= num

italic => basic types, Boldface => lateral, Regular => terminal
6.2 BNF Notation of PARA

This Appendix contains the BNF notation of the specifications used in this thesis.

<paras> ::= <para>
         | <para> <paras>
<para>  ::= <cirdef>
<cirdef> ::= <chipname> = <slices> ;
<chipname> ::= <str>
<str> ::= [char]
<slices> ::= <slice>
         | <slice> $con <slices>
<slice> ::= <wiring> $con <para>
<wiring> ::= ( wire [ <substreams> ] )
<substreams> ::= <substream>
         | <substream>, <substreams>
<substream> ::= [ <wireposits> ]
<wireposits> ::= <wireposit>
         | <wireposit>, <wireposits>
<wireposit> ::= <wpos>
<wpos> ::= num
<para> ::= ( para [ <gcnames> ] )
<gcnames> ::= <gcname2>
         | <gcname2>, <gcnames>
<gcname2> ::= <libgcname>
         | <libgcname> <pgcname>
<pgcname> ::= <modi> ( <coeff> )
         | [ <listofmodi> ]
<libgcname> ::= <str>
<pgcname> ::= <str>
<modi> ::= num
<coeff> ::= num
<listofmodi> ::= <modi>
         | <modi>, <listofmodi>
6.3 BNF Notation of MDL

<mdl> ::= <chip> <purposes> <moduli> <rom_contents>
<chip> ::= chipname = <chipname>
<chipname> ::= [char]
<purposes> ::= purpose = <purpose>
<purpose> ::= encoder | FIR_Filter | scaling
<moduli> ::= moduli = ( <list_of_mod> )
<list_of_mod> ::= num <list_of_mod> | num
<rom_contents> ::= rom = ( <list_of_rom> )
<list_of_rom> ::= num <list_of_rom> | num
APPENDIX 7
EXAMPLES OF SPECIFICATIONS

7.1 Example of FADL: A Low Pass Filter

This is an example of FADL output generated by the Designer’s Assistant front-end.

Entity chip_1
Module mod_1 [ Purpose encoder, Moduli [32,31,29,27],
    Input_bits 8 ]
Floorplan [ Bias column, Array [[A]] ]
    Map [[A, mod_1]]
Entity chip_2
Module mod_1 [ Purpose ipsp, Moduli [32],
    Coefficients [184,67,-39,8,11,-15,7,2,-8,6,0,-4,4,-1,-1] ]
Floorplan [ Bias column, Array [[A]] ]
    Map [[A, mod_1]]
Entity chip_3
Module mod_1 [ Purpose ipsp, Moduli [32],
    Coefficients [3,-1,0,1,-1,0,1,-1,0,0,-1,-1,0,0,-1] ]
Floorplan [ Bias column, Array [[A]] ]
    Map [[A, mod_1]]
Entity chip_4
Module mod_1 [ Purpose ipsp, Moduli [32],
    Coefficients [1,0,-1,1,0,-1,3,-1,-1,4,-4,0,6,-8,2] ]
Floorplan [ Bias column, Array [[A]] ]
    Map [[A, mod_1]]
Entity chip_5
Module mod_1 [ Purpose ipsp, Moduli [32],
    Coefficients [7,-15,11,8,-39,67] ]
Floorplan [ Bias column, Array [[A]] ]
    Map [[A, mod_1]]
Entity chip_6
Module mod_1 [ Purpose ipsp, Moduli [31],
    Coefficients [184,67,-39,8,11,-15,7,2,-8,6,0,-4,4,-1,-1] ]
Floorplan [ Bias column, Array [[A]] ]
Map \([ [A, \text{mod}_1] ]\)

Entity \text{chip}_7

Module \text{mod}_1 \ [ \text{Purpose} \ ipsp, \text{Moduli} \ [31],
\ Coefficients \ [3,-1,0,1,-1,0,1,-1,0,0,-1,-1,0,0,-1] ]

Floorplan \ [ \text{Bias column}, \ \text{Array} \ [[A]] ]

Map \([ [A, \text{mod}_1] ]\)

Entity \text{chip}_8

Module \text{mod}_1 \ [ \text{Purpose} \ ipsp, \text{Moduli} \ [31],
\ Coefficients \ [1,0,-1,1,0,-1,3,-1,-1,4,-4,0,6,-8,2] ]

Floorplan \ [ \text{Bias column}, \ \text{Array} \ [[A]] ]

Map \([ [A, \text{mod}_1] ]\)

Entity \text{chip}_9

Module \text{mod}_1 \ [ \text{Purpose} \ ipsp, \text{Moduli} \ [31],
\ Coefficients \ [7,-15,11,8,-39,67] ]

Floorplan \ [ \text{Bias column}, \ \text{Array} \ [[A]] ]

Map \([ [A, \text{mod}_1] ]\)

Entity \text{chip}_10

Module \text{mod}_1 \ [ \text{Purpose} \ ipsp, \text{Moduli} \ [29],
\ Coefficients \ [184,67,-39,8,11,-15,7,2,-8,6,0,-4,4,-1,-1] ]

Floorplan \ [ \text{Bias column}, \ \text{Array} \ [[A]] ]

Map \([ [A, \text{mod}_1] ]\)

Entity \text{chip}_11

Module \text{mod}_1 \ [ \text{Purpose} \ ipsp, \text{Moduli} \ [29],
\ Coefficients \ [3,-1,0,1,-1,0,1,-1,0,0,-1,-1,0,0,-1] ]

Floorplan \ [ \text{Bias column}, \ \text{Array} \ [[A]] ]

Map \([ [A, \text{mod}_1] ]\)

Entity \text{chip}_12

Module \text{mod}_1 \ [ \text{Purpose} \ ipsp, \text{Moduli} \ [29],
\ Coefficients \ [1,0,-1,1,0,-1,3,-1,-1,4,-4,0,6,-8,2] ]

Floorplan \ [ \text{Bias column}, \ \text{Array} \ [[A]] ]

Map \([ [A, \text{mod}_1] ]\)

Entity \text{chip}_13

Module \text{mod}_1 \ [ \text{Purpose} \ ipsp, \text{Moduli} \ [29],
\ Coefficients \ [7,-15,11,8,-39,67] ]

Floorplan \ [ \text{Bias column}, \ \text{Array} \ [[A]] ]

Map \([ [A, \text{mod}_1] ]\)
Entity chip_14
Module mod_1 [ Purpose ipsp, Moduli [27],
    Coefficients [184, 67, -39, 8, 11, -15, 7, 2, -8, 6, 0, -4, 4, -1, -1] ]
Floorplan [ Bias column, Array [[A]] ]
Map [[A, mod_1]]
Entity chip_15
Module mod_1 [ Purpose ipsp, Moduli [27],
    Coefficients [3, -1, 0, 1, -1, 0, 1, -1, 0, 0, -1, 0, 0, -1] ]
Floorplan [ Bias column, Array [[A]] ]
Map [[A, mod_1]]
Entity chip_16
Module mod_1 [ Purpose ipsp, Moduli [27],
    Coefficients [1, 0, -1, 1, 0, -1, 3, -1, -1, 4, -4, 0, 6, -8, 2] ]
Floorplan [ Bias column, Array [[A]] ]
Map [[A, mod_1]]
Entity chip_17
Module mod_1 [ Purpose ipsp, Moduli [27],
    Coefficients [7, -15, 11, 8, -39, 67] ]
Floorplan [ Bias column, Array [[A]] ]
Map [[A, mod_1]]
Entity chip_18
Module mod_1 [ Purpose scalar, Moduli [32, 31, 29, 27],
    Output_bits 8 ]
Floorplan [ Bias column, Array [[A]] ]
Map [[A, mod_1]]
7.2 Example of PARA

This is the PARA specification generated by FCST for the FADL description given in section 1 of this Appendix

```
chip_1 =
  (wire [[1,2,3,4,5,6,7,8,9,10]],
   [11,12,13,14,15,16,17,18,19,20],
   [21,22,23,24,25,26,27,28,29,30],
   [31,32,33,34,35,36,37,38,39,40]])
$con

(para [msc 32 (16), msc 31 (16),
       msc 29 (16), msc 27 (16)])
```

```
chip_2 =
  (wire [[1,2,3,4,5,6,7,8,9,10]])
$con

(para [msc 32 (184)])
$con

(wire [[1,2,3,4,5,6,7,8,9,10]])
$con

(para [msc 32 (67)])
$con

(wire [[1,2,3,4,5,6,7,8,9,10]])
$con

(para [msc 32 (-39)])
$con

(wire [[1,2,3,4,5,6,7,8,9,10]])
$con

(para [msc 32 (8)])
$con

(wire [[1,2,3,4,5,6,7,8,9,10]])
$con

(para [msc 32 (11)])
$con

(wire [[1,2,3,4,5,6,7,8,9,10]])
```

```
(para [msc 32 (-1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)] )
;
chip_3 =
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (3)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(chip_4 =
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (1)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (3)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (0)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-8)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (2)])
;
chip_5 =
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (7)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (-15)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (11)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 32 (8)])
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (3)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (-9)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (10)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (-10)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (11)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (-11)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 61 (15)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con

(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (0)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (3)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (-4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (6)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (-8)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (2)])
$con
chip_9 =
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (7)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (-15)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (11)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (8)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (-39)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 31 (67)])
$con
chip_10 =
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (184)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (67)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (-39)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (8)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (11)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (-15)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (7)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (2)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (-8)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (6)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (0)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (-4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (-1)])

; chip_11 =
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (3)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 29 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
```plaintext
$\text{ex}
\begin{align*}
\text{num} & : \{0,1\}^N, \\
\text{chip}_{12} & := (\text{num})^{12}, \\
\text{para} & : \{0,1\}^{12}, \\
\text{wire} & : \{0,1\}^{12}.
\end{align*}
\text{ex}
```
(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (-1)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (3)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (-1)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (4)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (-4)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (0)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (6)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (7)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (-15)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (11)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (8)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (-39)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (67)]) $con

(chip_14 =
(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 27 (184)]) $con

(wire [[1,2,3,4,5,6,7,8,9,10]]) $con
(para [msc 29 (-8)]) $con

$con
(para [msc 27 (67)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (-39)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (8)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (11)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (-15)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(chip_15 =
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (7)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (2)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (-8)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (6)] )
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])

EXAMPLES OF SPECIFICATIONS • 181
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (-1)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (-4)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (0)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (6)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (-8)])
$con
(wire [[1,2,3,4,5,6,7,8,9,10]])
$con
(para [msc 27 (2)])

chip_17 =

chip_18 =
(wire [[1,2,3,4,5,6,7,8,9,10,11,
12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,
34,35,36,37,38,39,40]])
$con
(para [scalar [32,31,29,27]])

;
7.3 Example of EDIF

This EDIF format was obtained by using FCST to translate the first part of the PARA specification given in section 2 of this Appendix.

(EDIF chip_1_edif
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap
  (keywordLevel 0))
 (status
  (written
   (timestamp 1991 4 16 12 47 59)
   (program "chip_1_program" )))
(external cmos3lib
 (edifLevel 0)
 (technology
  (numberDefinition ))
 (cell msc_27_(16)
  (celltype GENERIC)
  (view symbol
   (viewType NETLIST)
   (interface
    (port X1 (direction input))
    (port X2 (direction input))
    (port X3 (direction input))
    (port X4 (direction input))
    (port X5 (direction input))
    (port Y1 (direction input))
    (port Y2 (direction input))
    (port Y3 (direction input))
    (port Y4 (direction input))
    (port Y5 (direction input))
    (port OUT1 (direction output))
    (port OUT2 (direction output))
    (port OUT3 (direction output))
    (port OUT4 (direction output))
    (port OUT5 (direction output))
    (port OUT6 (direction output))
    (port OUT7 (direction output))
    (port OUT8 (direction output))
    (port OUT9 (direction output))
    (port OUT10 (direction output))))
 (cell msc_29_(16)
  (celltype GENERIC)
(view symbol
  (viewType NETLIST)
  (interface
    (port X1 (direction input))
    (port X2 (direction input))
    (port X3 (direction input))
    (port X4 (direction input))
    (port X5 (direction input))
    (port Y1 (direction input))
    (port Y2 (direction input))
    (port Y3 (direction input))
    (port Y4 (direction input))
    (port Y5 (direction input))
    (port OUT1 (direction output))
    (port OUT2 (direction output))
    (port OUT3 (direction output))
    (port OUT4 (direction output))
    (port OUT5 (direction output))
    (port OUT6 (direction output))
    (port OUT7 (direction output))
    (port OUT8 (direction output))
    (port OUT9 (direction output))
    (port OUT10 (direction output))))
  (cell msc_31_16)
  (celltype GENERIC)
  (view symbol
    (viewType NETLIST)
    (interface
      (port X1 (direction input))
      (port X2 (direction input))
      (port X3 (direction input))
      (port X4 (direction input))
      (port X5 (direction input))
      (port Y1 (direction input))
      (port Y2 (direction input))
      (port Y3 (direction input))
      (port Y4 (direction input))
      (port Y5 (direction input))
      (port OUT1 (direction output))
      (port OUT2 (direction output))
      (port OUT3 (direction output))
      (port OUT4 (direction output))
      (port OUT5 (direction output))
      (port OUT6 (direction output))
      (port OUT7 (direction output))
      (port OUT8 (direction output))
(port OUT9 (direction output))
(port OUT10 (direction output)))
(cell msc_32_16)
(celltype GENERIC)
(view symbol
(viewType NETLIST)
(interface
(port X1 (direction input))
(port X2 (direction input))
(port X3 (direction input))
(port X4 (direction input))
(port X5 (direction input))
(port Y1 (direction input))
(port Y2 (direction input))
(port Y3 (direction input))
(port Y4 (direction input))
(port Y5 (direction input))
(port OUT1 (direction output))
(port OUT2 (direction output))
(port OUT3 (direction output))
(port OUT4 (direction output))
(port OUT5 (direction output))
(port OUT6 (direction output))
(port OUT7 (direction output))
(port OUT8 (direction output))
(port OUT9 (direction output))
(port OUT10 (direction output))))
(cell inpadx (celltype GENERIC)
(view symbol (viewType NETLIST)
(interface (port OUT (direction output))))))
(cell outpadx (celltype GENERIC)
(view symbol (viewType NETLIST)
(interface (port IN (direction input))))))
(cell pp (celltype GENERIC)
(view symbol (viewType NETLIST)
(interface
(port (rename vdd "vdd!") (direction input))
(port vddcore (direction output))))))
(cell gpx
(celltype GENERIC)
(view symbol (viewType NETLIST)
(interface
(port (rename gnd "gnd!") (direction input))
(port gndcore (direction input))))))
(library FIR_Filter
(edifLevel 0) (technology (numberDefinition ))
(cell chip_1 (celltype GENERIC)
(view netlist (viewType NETLIST)
(interface)
(contents
(instance INPUT0 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT1 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT2 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT3 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT4 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT5 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT6 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT7 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT8 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT9 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT10 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT11 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT12 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT13 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT14 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT15 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT16 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT17 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT18 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT19 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT20 (viewRef symbol
  (cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT21 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT22 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT23 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT24 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT25 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT26 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT27 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT28 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT29 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT30 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT31 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT32 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT33 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT34 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT35 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT36 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT37 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT38 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance INPUT39 (viewRef symbol
(cellRef inpadx (libraryRef cmos3lib ))))
(instance OUTPUT0 (viewRef symbol
(cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT1 (viewRef symbol
(cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT2 (viewRef symbol
(cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT3 (viewRef symbol
(cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT4 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT5 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT6 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT7 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT8 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT9 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT10 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT11 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT12 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT13 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT14 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT15 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT16 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT17 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT18 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT19 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT20 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT21 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT22 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT23 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT24 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT25 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT26 (viewRef symbol  
  (cellRef outpadx (libraryRef cmos3lib ))))
(instance OUTPUT27 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT28 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT29 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT30 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT31 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT32 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT33 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT34 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT35 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT36 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT37 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT38 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance OUTPUT39 (viewRef symbol
(cellRef outpadx (libraryRef_cmos3lib ))))
(instance vddcore (viewRef symbol
(cellRef ppx (libraryRef_cmos3lib ))))
(instance gndcore (viewRef symbol
(cellRef gpx (libraryRef_cmos3lib ))))
(instance msc_32_(16)_1 (viewRef symbol
(cellRef msc_32_(16) (libraryRef_cmos3lib))))
(instance msc_31_(16)_1 (viewRef symbol
(cellRef msc_31_(16) (libraryRef_cmos3lib))))
(instance msc_29_(16)_1 (viewRef symbol
(cellRef msc_29_(16) (libraryRef_cmos3lib))))
(instance msc_27_(16)_1 (viewRef symbol
(cellRef msc_27_(16) (libraryRef_cmos3lib))))

(net NET1 (joined
(portRef OUT (instanceRef INPUT0))
(portRef X1 (instanceRef msc_32_(16)_1 ))))

(net NET2 (joined
(portRef OUT (instanceRef INPUT1))
(portRef X2 (instanceRef msc_32_(16)_1 ))))

(net NET3 (joined
(portRef OUT (instanceRef INPUT2))
(portRef X3 (instanceRef msc_32_(16)_1)))
(net NET4 (joined
  (portRef OUT (instanceRef INPUT3))
  (portRef X4 (instanceRef msc_32_(16)_1))))
(net NET5 (joined
  (portRef OUT (instanceRef INPUT4))
  (portRef X5 (instanceRef msc_32_(16)_1))))
(net NET6 (joined
  (portRef OUT (instanceRef INPUT5))
  (portRef Y1 (instanceRef msc_32_(16)_1))))
(net NET7 (joined
  (portRef OUT (instanceRef INPUT6))
  (portRef Y2 (instanceRef msc_32_(16)_1))))
(net NET8 (joined
  (portRef OUT (instanceRef INPUT7))
  (portRef Y3 (instanceRef msc_32_(16)_1))))
(net NET9 (joined
  (portRef OUT (instanceRef INPUT8))
  (portRef Y4 (instanceRef msc_32_(16)_1))))
(net NET10 (joined
  (portRef OUT (instanceRef INPUT9))
  (portRef Y5 (instanceRef msc_32_(16)_1))))
(net NET11 (joined
  (portRef OUT (instanceRef INPUT10))
  (portRef X1 (instanceRef msc_31_(16)_1))))
(net NET12 (joined
  (portRef OUT (instanceRef INPUT11))
  (portRef X2 (instanceRef msc_31_(16)_1))))
(net NET13 (joined
  (portRef OUT (instanceRef INPUT12))
  (portRef X3 (instanceRef msc_31_(16)_1))))
(net NET14 (joined
  (portRef OUT (instanceRef INPUT13))
  (portRef X4 (instanceRef msc_31_(16)_1))))
(net NET15 (joined
  (portRef OUT (instanceRef INPUT14))
  (portRef X5 (instanceRef msc_31_(16)_1))))
(net NET16 (joined
  (portRef OUT (instanceRef INPUT15))
  (portRef Y1 (instanceRef msc_31_(16)_1))))
(net NET17 (joined
  (portRef OUT (instanceRef INPUT16))
  (portRef Y2 (instanceRef msc_31_(16)_1))))
(net NET18 (joined
  (portRef OUT (instanceRef INPUT17))
  (portRef Y3 (instanceRef msc_31_(16)_1))))
(portRef OUT (instanceRef INPUT33))
(portRef X4 (instanceRef msc_27_(16)_1 )))

(net NET35 (joined
(portRef OUT (instanceRef INPUT34))
(portRef X5 (instanceRef msc_27_(16)_1 )))

(net NET36 (joined
(portRef OUT (instanceRef INPUT35))
(portRef Y1 (instanceRef msc_27_(16)_1 )))

(net NET37 (joined
(portRef OUT (instanceRef INPUT36))
(portRef Y2 (instanceRef msc_27_(16)_1 )))

(net NET38 (joined
(portRef OUT (instanceRef INPUT37))
(portRef X3 (instanceRef msc_27_(16)_1 )))

(net NET39 (joined
(portRef OUT (instanceRef INPUT38))
(portRef Y3 (instanceRef msc_27_(16)_1 )))

(net NET40 (joined
(portRef OUT (instanceRef INPUT39))
(portRef Y5 (instanceRef msc_27_(16)_1 )))

(net NET41 (joined
(portRef OUT1 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT0))))

(net NET42 (joined
(portRef OUT2 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT1)))

(net NET43 (joined
(portRef OUT3 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT2)))

(net NET44 (joined
(portRef OUT4 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT3)))

(net NET45 (joined
(portRef OUT5 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT4)))

(net NET46 (joined
(portRef OUT6 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT5)))

(net NET47 (joined
(portRef OUT7 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT6)))

(net NET48 (joined
(portRef OUT8 (instanceRef msc_32_(16)_1 ))
(portRef IN (instanceRef OUTPUT7)))

(net NET49 (joined
(portRef OUT9 (instanceRef msc_32_(16)_1 ))
(net NET65 (joined
(portRef OUT5 (instanceRef msc_29_(16)_1 ))
(portRef IN (instanceRef OUTPUT24))))

(net NET66 (joined
(portRef OUT6 (instanceRef msc_29_(16)_1 ))
(portRef IN (instanceRef OUTPUT25))))

(net NET67 (joined
(portRef OUT7 (instanceRef msc_29_(16)_1 ))
(portRef IN (instanceRef OUTPUT26))))

(net NET68 (joined
(portRef OUT8 (instanceRef msc_29_(16)_1 ))
(portRef IN (instanceRef OUTPUT27))))

(net NET69 (joined
(portRef OUT9 (instanceRef msc_29_(16)_1 ))
(portRef IN (instanceRef OUTPUT28))))

(net NET70 (joined
(portRef OUT10 (instanceRef msc_29_(16)_1 ))
(portRef IN (instanceRef OUTPUT29))))

(net NET71 (joined
(portRef OUT1 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT30))))

(net NET72 (joined
(portRef OUT2 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT31))))

(net NET73 (joined
(portRef OUT3 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT32))))

(net NET74 (joined
(portRef OUT4 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT33))))

(net NET75 (joined
(portRef OUT5 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT34))))

(net NET76 (joined
(portRef OUT6 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT35))))

(net NET77 (joined
(portRef OUT7 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT36))))

(net NET78 (joined
(portRef OUT8 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT37))))

(net NET79 (joined
(portRef OUT9 (instanceRef msc_27_(16)_1 ))
(portRef IN (instanceRef OUTPUT38))))

(net NET80 (joined
(portRef OUT10 (instanceRef msc_27_(16)_1 ))
  (portRef IN (instanceRef OUTPUT39)))
(net (rename vdd "vdd!"))
  (joined (portRef vddcore (instanceRef ppx ))))
(net (rename gnd "gnd!"))
  (joined (portRef gndcore (instanceRef gpx ))))))))))
7.4 Example of ROM CONTENTS

This MDL description was obtained by using FCST to translate the first part of the FADL description given in section 1 of this Appendix.

chipname = "chip_2"
purpose = "FIR_Filter"
moduli = '(32 31)
rom = '( (24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8
         9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
        (16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)
        (30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
        (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)
        (18 19 20 21 22 23 24 25 26 27 28 29 30 31)
        (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)
        (18 19 20 21 22 23 24 25 26 27 28 29 30 31)
        (3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
        (20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5)
        (6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
        (22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5)
        (12 13 14 15 16 17 18 19 20 21 22 23 24 25 26)
        (27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11)
        (24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9
         10 11 12 13 14 15 16 17 18 19 20 21 22 23)
        (16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)
        (30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
        (25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10)
        (11 12 13 14 15 16 17 18 19 20 21 22 23 24)
        (18 19 20 21 22 23 24 25 26 27 28 29 30 31 0)
        (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)
        (4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
        (21 22 23 24 25 26 27 28 29 30 31 0 1 2 3)
        (8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
        (23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7)
        (16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)
        (31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
        (8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
        (24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7)
        (16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)
        (31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
        (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)
        (18 19 20 21 22 23 24 25 26 27 28 29 30 31)
EXAMPLES OF SPECIFICATIONS • 197

(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10)
(22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11)
(24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11)
(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11)
(24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10)
(12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7)
(14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13)
(22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13)
(14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
(24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
EXAMPLES OF SPECIFICATIONS • 198

(16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5)
(12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11)
(24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
EXAMPLES OF SPECIFICATIONS - 199

(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31)
(31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26 27)
(24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26 27)
(24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(27 28 29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26 27)
(23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 0 1 2 3 4 5)
(10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8 9)
(20 21 22 23 24 25 26 27 28 29 30 0 1 2 3 4 5
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
(9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8 9)
(18 19 20 21 22 23 24 25 26 27 28 29 30 0 1
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18)
(23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(27 28 29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26 27)
(0 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 0 1)
(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 0 1 2)
(4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 0 1 2 3)
(11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 0 1 2 3 4 5 6 7 8 9 10 11)
(22 23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
(13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13)
(26 27 28 29 30 0 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27)
(21 22 23 24 25 26 27 28 29 30 0 1 2 3 4 5 6
7 8 9 10 11 12 13 14 15 16 17 18 19 20 21)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 0 1)
(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 0 1 2)
(4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 0 1 2 3)
(8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8)
(7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7)
(14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
(28 29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
13 16 17 18 19 20 21 22 23 24 25 26 27 28)
(25 26 27 28 29 30 0 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23 24 25)
(19 20 21 22 23 24 25 26 27 28 29 30 0 1 2 3
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
(30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 0 1 2 3 4)
(6 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8)
(16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 0 1)
(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 0 1 2)
(30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(27 28 29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29)
(23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
(30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30)
(29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29)
(27 28 29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29)
(23 24 25 26 27 28 29 30 0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
(15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15))
REFERENCES


[Joh82] Steven D. Johnson. Circuits and systems: Implementing communication with streams. Proc. of the 10th IMACS World


VITA AUCTORIS

Wai Suen Fong was born in 1960 in Hong Kong. He completed his high school education at Rosemount High School in 1978. He then graduated from Dawson College and obtained a Diploma des Etudiants des College in 1983. He completed his Honors degree in Computer Science from the University of Windsor in 1989. In May 1991 he finished his Masters of Computer Science in School of Computer Science at the University of Windsor.