Compiling for Coarse-Grain Reconfigurable Architectures

by

M. Morris E. Matsa

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

© M. Morris E. Matsa, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by ....
Thomas F. Knight, Jr.
Senior Research Scientist
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

OCT 29 1997

Compiling for Coarse-Grain Reconfigurable Architectures

by

M. Morris E. Matsa

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Increasing silicon area and inter-chip communication costs allow and require that modern general purpose computing devices incorporate large amounts of processing on a single die. The increased area permits a wider choice of architectures to perform this computation. The traditional approach to utilizing this silicon area is to place one, or a few, large processing elements on the die (microprocessors). This style of computation achieves its performance by dynamically issuing new instructions at a very high rate. On the other hand, it has long been recognized that much faster, more efficient processing is possible using application-specific and, even more significantly, computation-specific processing elements. More recently, academic and industry efforts have been working to use very fine-grain reconfigurable devices (such as FPGAs) in order to attain this performance. Instead of large processors, these devices place a large number of very small processing elements on the die, and connect them in a configurable network. Coarse Grain Reconfigurable Architectures (CGRAs) are a hybrid of these architectures, one that is capable of both dynamic instruction streams and application-specific optimizations. Depending on the way in which a CGRA is configured to run a given application, it can emulate a processor solution, a fine-grain reconfigurable device, or a hybrid between the two. Although initial indications are that CGRAs will be very efficient, it is not yet known how to best design these devices. Our group has implemented a prototype CGRA device called MATRIX. This thesis introduces several intermediate representations for the configuration of a MATRIX chip, each with the ability to specify higher level concepts than the previous languages and thus better suited to describing the algorithm being implemented while avoiding the unimportant configuration details. We then describe new tools that have been designed to convert these representations into an application to run on MATRIX. To compile the more advanced intermediate representations these tools include placement, routing, and power minimization algorithms. Using these tools, we compile down specific applications and emulations of many different general purpose computing architectures in order to achieve a better understanding of how to most efficiently program using MATRIX. After compiling for MATRIX, we vary the backend of the compiler so that we can experiment with compiling for similar architectures in an effort to analyze the usefulness of the various features of this

architecture. In this way, we learn which of MATRIX's features are useful for mapping various applications, and we make design suggestions for general CGRAs. We also develop insights into techniques for placement and heuristics for using these hybrid architectures. Finally, by designing an intermediate representation which can be compiled from a high-level language, as well as being easily converted into an application running on MATRIX, we will help define the role of high-level synthesis in compiling for Coarse Grain Reconfigurable Architectures.

Keywords: CAD, FPGA, Coarse Grain, Reconfigurable, Placement, Routing, Power Minimization, High-Level Synthesis

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Senior Research Scientist

Acknowledgments

I would like to thank everyone in my life, but it seems most appropriate to use this space to thank the people in the Reinventing Computing group at the MIT Artificial Intelligence Laboratory.

Ethan Mirsky and **Ian Eslick** for developing MATRIX and for introducing me to the group, without which I could not have written this thesis. I used several figures from Ethan's thesis.

Jeremy Brown and **Amit Patel** for being my officemates. Amit put up with me while I was testing MATRIX and Jeremy put up with me while I did my thesis.

Yael Levi, a high school student from the RSI program, for being the first person to attempt programming for MATRIX without understanding the hardware. Working with Yael gave me a better idea of how to abstract MDL+ from MATRIX. She also contributed to the polynomial example of chapter 6.

André DeHon. Many of the ideas in here originated as his. MATRIX is a direct result of his years of work into the nature of computing, and MDL+ is a direct result of discussions with him at the beginning of my research. My ramblings about high-level synthesis and the future of computing are largely a result of our discussions and our reading of the literature.

Daniel Hartman was involved in all levels of the design of the MATRIX chip, including most of the layout. An EECS renaissance man, he also discussed a few high-level synthesis papers with me, as well as discussing a plethora of MDL+ issues with me — both specific details and general views about the future of MATRIX, MDL, and computing. I believe that Dan is the only person who actually dared read through my code. "My god, you **are** a 6-3."

Thomas F. Knight, Jr., without whom the Reinventing Computing group would not even exist. It has been said behind his back that he is the kind of thesis advisor that other students wish they had.

The United States of America, the Department of Defense, the Advanced Research Projects Agency, and all United States taxpayers.

This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

Contents

1 Introduction **15**

2 The MDL Family of Languages **21**

2.1 Background 21

2.2 An Abstract view of MDL languages 24

2.3 The Anatomy of the MDL+ compiler 26

2.4 Multi-level MDL+ Programming 29

3 MATRIX **31**

3.1 Architecture Overview 31

3.2 Basic Unit 32

3.3 Interconnect Network 34

3.4 Network Ports 36

3.5 Control 38

3.6 Deployable Resources 39

4 MDL+ as Better MDL **41**

4.1 Language Basics 42

4.1.1 Basic Statement Structure 42

4.1.2 Identifiers 43

4.1.3 Numbers 43

4.1.4 Compare/Reduce Numbers 43

4.1.5 Don't Cares 44

4.1.6 Copying Constructs 44

4.1.7 Inheritance 44
4.1.8 Reserved Words 45
4.1.9 Lists 45
4.1.10 Ordering Directions 46
4.1.11 Going Beyond MATRIX 46
4.1.12 Definitions 47
4.1.13 Type Checking 49
4.1.14 Output 50
4.2 Language Constructs 51
4.2.1 Constants 51
4.2.2 Bitvecs 62
4.2.3 Sub-BFU Constructs 64
4.2.4 BFU Networks 65
4.2.5 BFU Configs 66
4.2.6 BFU Controls 70
4.2.7 BFU Powers 74
4.2.8 Ports 75
4.2.9 BFU Ports 78
4.2.10 Memories 80
4.2.11 BFUs 81
4.2.12 Layouts 84
4.2.13 Connects 88
4.2.14 IOports 90
4.2.15 Chips 91
4.2.16 Summary 94
4.3 Automatic Driving Phase 95
4.3.1 Motivation for Driving 95
4.3.2 What the Driver Does 96
4.3.3 Programming in MDL+1.1 101
4.4 Error Reporting and Interface Details 102

5 **Intelligent Phases of the MDL+ Compiler** **105**
5.1 The MDL+ Router 111
5.1.1 Syntax 111
5.1.2 Semantics 112
5.1.3 8-Bit Microprocessor 114
5.1.4 Benefits 115
5.2 The MDL+ Placer 116
5.2.1 Syntax 117
5.2.2 Semantics 117
5.2.3 8-Bit Microprocessor 119
5.2.4 Benefits 120
5.3 The MDL+ Grouper 121
5.3.1 Syntax 121
5.3.2 Semantics 122
5.3.3 8-Bit Microprocessor 123
5.3.4 Benefits 125
5.4 Improvements to MDL+ code 125

6 **Research done with MDL+** **127**
6.1 Examples 128
6.1.1 8-Bit Microprocessor 129
6.1.2 Polynomial Evaluator 133
6.1.3 Other Examples 137
6.2 General Purpose Computing Architectures 137
6.2.1 8-Bit Microprocessor 138
6.2.2 32-Bit Microprocessor 138
6.2.3 MIMD 142
6.2.4 VLIW 145
6.2.5 SIMD 149
6.2.6 ASIC 150

6.3 Other CGRAs 150
6.3.1 Eliminating Level-2 wires 150
6.3.2 Unregistered Level-2 Wires 151
6.3.3 No level-1 wires 151
6.3.4 No diagonal lines 152
6.3.5 No length-2 lines 153
6.3.6 Other Experiments 156
6.4 Questions about MATRIX 156
6.4.1 Level-2 lines 156
6.4.2 Level-1 lines 158
6.4.3 Inputs to BFU ports 159
6.4.4 OR Plane 159
6.4.5 Control 160
6.4.6 Other Questions 161

7 Conclusion **163**
7.1 Improvements from MDL 163
7.1.1 Code Generating MDL+ 163
7.1.2 MDL+'s Grammar 166
7.1.3 Output and Error Messages 170
7.2 New Insights 176
7.2.1 Manual Placement 177
7.2.2 How to use MDL+ 179
7.2.3 MATRIX's Strong Points 181
7.3 Future Work 182
7.3.1 Basic MDL+ 182
7.3.2 Driver 184
7.3.3 Router 186
7.3.4 Placer 186
7.3.5 Grouper 188

7.3.6 High-Level Synthesis 190
7.3.7 CGRAs 191

A MDL+ version 1.0 Grammar **193**

B MDL+ Man Pages **197**

C MDL+ Globally Reserved Words **201**

D 8 Bit Microprocessor implemented in MDL **205**

D.1 MDL 205
D.2 MDL+1.1 211
D.3 MDL+1.2 212
D.4 MDL+1.3 213
D.5 MDL+1.4 215

E Full MDL+ Grammar **217**

List of Figures

2-1 A Mathematical view of the various MDL Languages 25

2-2 A Mathematical view of the MDL+ Intermediate Languages 27

2-3 A Block Diagram of the proposed MDL++ Compiler 28

3-1 MATRIX Basic Functional Unit 33

3-2 Level 1 Network Connections 34

3-3 Level 2 Network Connections 35

3-4 MATRIX Network Switch Architecture - BFU Cell 36

3-5 Function Port Architecture 37

3-6 BFU Control Logic 38

3-7 Distributed PLA 39

4-1 Source code that does not type-check. 50

4-2 Typical error message for code that does not type-check. 50

4-3 Sample Bitvec Definition 63

4-4 BFU Control Logic 71

4-5 Four Ways to define a simple C/R II value 73

4-6 Nine ways to define a Port 77

4-7 Sample BFU Ports definition 80

4-8 Sample BFU Definition 83

4-9 Sample Error Message for over-shifting a non-DC BFU 86

4-10 One BFU replicated 36 times by placing Layouts in Layouts 87

4-11 A typical Connect 89

4-12 Block Diagram of a MATRIX Chip with IOport names 93

5-1 8 Bit Microprocessor on MATRIX 106
5-2 PC definition in MDL+1.1 107
5-3 I-store definition in MDL+1.1 108
5-4 Definitions of A, B, and F in MDL+1.1 109
5-5 Definition of ALU in MDL+1.1 110
5-6 Definition of Layout in MDL+1.1 110
5-7 Definition of Layout in MDL+1.3 120
5-8 Definition of Memory Objects in MDL+1.4 124
5-9 Definition of ALU Object in MDL+1.4 124

6-1 8 Bit Microprocessor on MATRIX, placed by a human 129
6-2 8 Bit Microprocessor on MATRIX, automatically placed with mdl -2 . 132
6-3 Polynomial Evaluator designed by a human 134
6-4 Polynomial Evaluator as placed by the compiler 136
6-5 32 Bit Microprocessor on MATRIX, placed by a human 139
6-6 32 Bit Microprocessor on MATRIX, automatically placed with mdl -2 140
6-7 MIMD Architecture on MATRIX, placed by a human 143
6-8 MIMD Architecture on MATRIX, automatically placed with mdl -2 . . 144
6-9 VLIW Architecture on MATRIX, placed by a human 145
6-10 VLIW Architecture on MATRIX, automatically placed with mdl -2 . . 147
6-11 SIMD Processor on MATRIX 149
6-12 8-bit microprocessor routed without level-1 lines 152
6-13 8-bit microprocessor routed without diagonal level-1 wires 153
6-14 8-bit microprocessor routed without length-2 level-1 wires 154
6-15 VLIW architecture routed without length-2 level-1 wires 155

7-1 Examining an MDL+ verilog output file 174
7-2 MDL+ code that does not make sense 184

List of Tables

4.1 String meanings in (const String) structure 53

4.2 Strings in Static Source Structures 54

4.3 Strings defining Selection Values 59

4.4 Strings defining Enable Values 60

4.5 Quick Summary of Types of Constants 61

4.6 Quick Summary of Constants 62

4.7 Values in a Bitvec 63

4.8 Quick Summary of Bitvecs 64

4.9 Meaning of l2_d1 and l2_d2 66

4.10 Quick Summary of BFU Networks 66

4.11 Flags allowed in BFU Configs 67

4.12 Sources for Left and Right Flags 69

4.13 Quick Summary of BFU Configs 69

4.14 Control argument keywords and associated hardware 70

4.15 ReduceII Arguments 72

4.16 ReduceII Bit-Desc Legal Arguments 72

4.17 Quick Summary of BFU Controls 74

4.18 Quick Summary of BFU Powers 75

4.19 Ways to define a Value of a Port 76

4.20 Ways to define a Constant Number, for TScycle 78

4.21 Quick Summary of Ports 78

4.22 Names for BFU Ports 79

4.23 Quick Summary of BFU Ports 80

4.24 Quick Summary of Memories 81
4.25 Quick Summary of BFUs 84
4.26 Ways to define a Constant Number for Layout Coordinates 85
4.27 Quick Summary of Layouts 87
4.28 Quick Summary of Connects 89
4.29 Quick Summary of IOports 91
4.30 Quick Summary of Chips 94
4.31 MDL+ Constructs 95

5.1 Length of 8-bit microprocessor implementations 126

C.1 Global Keywords in MDL+, part 1 of 2 202
C.2 Global Keywords in MDL+, part 2 of 2 203

Chapter 1

Introduction

Continuing advances in semiconductor technology have greatly increased the amount of processing that can be performed by single-chip general purpose computing devices. In addition, the relatively slow increase of inter-chip communication bandwidth requires that modern high performance devices use as much of their potential on-chip processing power as possible. This results in large, dense ICs and a large design space for general purpose computing architectures.

There are several ways of viewing this design space, one way being in terms of granularity. Designers have the option of building very large processing units, or many smaller ones, in the same space. Traditional architectures are either very coarse grain, such as microprocessors, or very fine grain such as FPGAs (Field Programmable Gate Arrays). Both have their own advantages and disadvantages.

Very coarse grain devices, such as microprocessors, incorporate very few large processing units which operate on wide data-words. Each unit is hardwired to perform a small set of instructions on these data-words. Usually each unit is optimized for a different set of instructions, such as integer and floating point, and the units are generally hardwired to operate in parallel. The hardwired nature of these units allows them to very rapidly perform their instructions. In fact, a great deal of area on modern microprocessor chips is dedicated to cache memories in order to support a very high rate of instruction issue. This allows these devices to efficiently handle very dynamic instruction streams.

Unfortunately, because microprocessors and other coarse-grain computing devices are highly optimized for simple, wide-word, dynamic instructions, they are relatively inefficient when performing other kinds of operations. For example, many cycles are required to build up a complex operations which are not part of the pre-selected instruction set, out of the processor's instructions. In addition, when performing short-word operations, a large amount of the device's processing power is not utilized.

On the other end of the design space, very fine grain devices, such as FPGAs, incorporate a large number of very small processing elements. These elements are arranged in a configurable interconnect network so that larger structures can be built out of them. The configuration data used to define the functionality of the processing units and network can be thought of as a very large, semantically powerful, instruction word. Nearly any operation can be described, and mapped to hardware. In general, this allows these devices to perform any particular operation faster than a coarse-grain, microprocessor-like, device.

However, the size of this "instruction word" creates a number of problems with fine-grain devices. First of all, reloading this instruction takes a relatively long time, making dynamic instruction streams very difficult for these devices. Secondly, if the operation being performed is, in fact, a wide word operation, a great deal of this "instruction word" must be dedicated to re-describing the operation for each of the small processing elements. Thus, fine grain processing elements are also not equipped to take advantage of a large number of common computing operations.

The increasing available silicon area means that it is now possible to build a large number of intermediate-grain processing elements. MATRIX (*Multiple ALU Architecture with Reconfigurable Interconnect Experiment*) [MD96] is the first such Coarse Grain Reconfigurable Architecture (CGRA) exploiting the regularity and rapid instruction issue features of coarse-grain units, but still allowing these units to be connected in an application-specific manner. This means that it is capable of deploying its coarse-grain resources, such as memory and processing, in a way that takes advantage of the opportunities for optimization present in any given problem.

Another way to view the design space is by the way in which the resources of the

chip are being reused. Designers have the option of building processing units which can be quickly reused in time, changing the operation they perform each cycle, or many smaller ones which will not change what they do as quickly, but will make use of all of the space on the chip.

Very coarse-grain devices, such as microprocessors, although they do waste a lot of area (such as an FPU during an application that only does integer arithmetic), reuse the complete ability of the chip each cycle, by doing an entirely new operation. Very fine-grain devices, such as FPGAs, have small hard-wired units which do not change their operation often, but there are usually tens of thousands of these units on a single chip, all of which can be doing useful operations in parallel at the same time, thus all of the spatial resources of the chip are being used. Chips that are using all of the temporal resources of the chip can not run faster than their slowest operation, and can not use their wasted area to parallelize a problem. Thus, they will not be fast, but they will not take up a lot of space. On the other side, chips that are using all of their spatial resources can run as quickly as the application dictates, but will spread out in area while parallelizing the application in order to compensate for using generalized fine-grain units instead of instruction-specific hardware (e.g. a multiplier). Thus, they will use a lot of space but will run quickly.

This space thus constitutes a view of the basic space-time tradeoff. CGRAs can, after fabrication time, allocate parts of the chip to a microprocessor-like implementation for a part of an application that does not require high-throughput, while using more space for the parts of the application that do require high-throughput by using FPGA-like implementations. This means that CGRAs can make better space-time tradeoffs on a per-application basis by deploying a lot of its space resources when faced with high-throughput constraints, and using its time resources to save on space by reusing some of its area through time when faced with low-throughput constraints.

In this thesis, we study only the CGRA MATRIX, and attempt to apply our knowledge to CGRAs and this entire space. Since MATRIX is a means of combining elements of different architectures, there should be an intermediate language expressible in terms of certain architectures which compiles to MATRIX. While other group

members have designed the actual layout for MATRIX, and a verilog simulation of it, this thesis will design this intermediate language as well as the tools to compile it down, and investigate high-level synthesis possibilities for MATRIX.

The first step in this process is specifying this new intermediate language, the new MATRIX Description Language (MDL+), to succeed the old MATRIX Description Language (MDL). Chapter 2 discusses the high-level motivations for designing MDL+, and the mathematical view of MDL+ as a member of a family of intermediate languages, such that it is a step on the road towards a high-level language which is easily compiled down for CGRAs using high-level synthesis techniques.

Once the framework for the design of MDL+ is set, chapter 3 will describe the actual MATRIX chip in sufficient detail to allow for a complete specification of the MDL+ language. Chapter 4 will then go into great detail about the syntax and semantics of MDL+, as well as describing the effects of the MDL+ compiler's Automatic Driving Phase, and providing a basic coverage of the interfaces and error reporting mechanisms of the new MDL+ compiler.

Chapter 5 will go on to discuss the intelligent phases of the MDL+ compiler which can automatically group high-level functionality into MATRIX constructs, place MATRIX constructs on a MATRIX chip, and route the interconnect between these units on a MATRIX chip. Through some discussion of how these phases of the compiler were designed and how they act, we will begin to understand how to better program for MATRIX, how to design CGRAs to be easily programmed, and how a high-level synthesis compiler might compile high-level code to an MDL+ backend.

Chapter 6 will then attempt to learn more about these issues by using the MDL+ compiler to study both MATRIX and MDL+. This will include implementing example applications in MDL+, examining various general-purpose computing architectures modeled in MDL+, and studying the effects of modifying the backend of the MDL+ compiler.

Finally, Chapter 7 will summarize the benefits of MDL+ and the MDL+ compiler over the previous state-of-the-art, as well as discussing what we have learned about programming for CGRAs and about designing CGRAs. It will end by suggesting

directions for future work.

The most interesting material is the research that culminates in new heuristics for designing coarse-grain reconfigurable architectures and for using these architectures, including ideas about how to place and route for them. This material is located in chapter 6 and the second half of chapter 7.

Chapter 2

The MDL Family of Languages

This chapter is intended to convey the big picture of what MDL+ is about. This extends into the motivation behind MDL+ including the desires for an eventual language MDL++, and leads to an understanding of both the way MDL+ is implemented and the way it is explained in this document. Finally, understanding this view of MDL+ should enable people to understand the abstraction layers available when programming in MDL+ as well as the ways to take advantage of this power. We start with a brief description of the mathematical ideal (section 2.2) and quickly progress through the anatomy of an MDL+ compiler (section 2.3) to suggestions for how to program using the power of MDL+. (section 2.4) This section should provide the motivation and high-level understanding that will assist with the later chapters on the syntax and semantics details of MDL+. (chapters 4,5)

2.1 Background

Before beginning the meat of this chapter, we provide a brief background of the languages and compiler phases that we will be referring to. They were all mentioned in chapter 1, and will be discussed in much more detail in later chapters, but this section is intended to provide enough detail about each until it itself is discussed.

MDL - This is the original Matrix Description Language, first introduced in [Esl95]. MDL is basically just a hardware description language for the MATRIX

chip. MDL was quickly designed and a compiler was developed for it as a quick hack. Both the language and compiler were developed as a first-cut implementation. The language was a good start, but not able to support much functionality due to the amount of time put into the compiler. The language was also never completely specified. The MDL compiler eventually got most of the bugs worked out of it, but is probably still fairly buggy. It needs more power and flexibility, it does not have a sufficient interface for error reporting, and was never well documented. Reading through some of the MDL compiler's code verifies that it was a quick-hack project.

MDL+ - This is the programming language developed for this thesis. It attempts to clean up both the specification of MDL and the implementation of the MDL compiler, as well as adding many nice programming features that MDL lacked or did not implement correctly, and adding higher-level programming possibilities. It is intended to be able to serve, with only slight changes, as a front-end to a compiler for various CGRAs besides MATRIX, and as a back-end to a compiler which performs high-level synthesis on a high-level language. Several of the ideas for improving MDL given in [Mat96b] have been incorporated into MDL+. Specification of the basic MDL+ language will be given in chapter 4.

MDL++ - This name is being reserved for the possible occurrence of a truly high-level compiler for MATRIX or other CGRA. MDL++ will be used throughout this thesis to describe an unspecified high-level language such as C or Silage, perhaps developed with MATRIX in mind, and the associated compiler which would turn this high-level code into configuration bits for MATRIX. Topics relating to MDL++ will be discussed in chapters 5 and 6, and a summary of suggestions for the design and implementation of MDL++ will be given in section 7.3.6.

Automatic Driving Phase - There are many interconnect wires on a MATRIX chip, and some must not be driven at any given time in order to conserve power and decrease heat. This phase of the MDL+ compiler determines whether each line is used by an MDL+ design, and might turn some wires on or off. The driver will be discussed in section 4.3.

Automatic Routing Phase - It is possible to specify an MDL+ design such that two units know that they are communicating with each other, but they do not know which of the many interconnect wires they are using. This phase of the MDL+ compiler sets both of those units so that they are communicating over a specific wire instead of merely knowing which unit they are communicating with. Once this is done, the driving phase can set used wires to be driven and others to be turned off. The router will be discussed in section 5.1.

Automatic Placing Phase - It is possible to specify in MDL+ that a unit should be on a chip, but not specify where on the chip that unit should be placed. This phase of the MDL+ compiler puts such units at specific locations on the chip. Once this is done, the routing phase can figure out which wires can and will be used to communicate with the unit. The placer will be discussed in section 5.2.

Automatic Grouping Phase - It is possible to specify in MDL+ that a chip should contain a higher-level object that does not precisely correspond to a physical unit that exists in the MATRIX hardware. This phase of the MDL+ compiler expresses such units as pieces of actual MATRIX hardware, and combines them to some extent, so that, for example, one basic unit on MATRIX might have one part of it acting as a logical register between two logical-objects, and also have another part of it acting as a logical memory unit. Once this is done, the placing phase can figure out where to put the actual MATRIX basic units on the chip. The grouper will be discussed in section 5.3.

High-Level Synthesis - This is the added functionality of the MDL++ compiler over the MDL+ compiler, or the process of turning a high-level language like C or Silage into a more hardware-oriented description. While high-level synthesis has gathered more definitions in the literature, this is the one we are referring to when we use the term in this thesis.

2.2 An Abstract view of MDL languages

When beginning the planning stages for MDL+, several objectives naturally come to mind:

- **Programming Extensibility** It would be desirable to be able to use MDL code with the MDL+ compiler. In fact, it would be best if we could interlink pieces of MDL code with pieces of MDL+ code, for example have a legal MDL+ design where an MDL unit interacts with an MDL+ unit in the same chip layout.

- **Multi-level code** It would also be advantageous to possess the ability to let users enter code at many levels – even given the power of an automatic grouper, placer, and router, the user might have specifically tightly designed a certain subpart of the cell, and want it placed in a specific part of the chip or even routed exactly as specified. Thus, we want to provide the user access to MDL+ without automatic features, MDL+ with a router, MDL+ with a router and placer, and a completely automatic MDL+ with a grouper. Upon reviewing many projects and papers on high level synthesis (such as [RP90, GR94, WC95, CHM91, Wak91, NON91, HDDW93, MCG+90, LND+91, CSW93, RvSC+93]), it becomes clear that this is a key aspect of a high-level chip compiler.

- **Human Interactions** It would be best if human interaction could occur in both directions, receiving information from the compiler at each stage and giving it to the compiler at each stage. This means that in addition to being able to tell the compiler where you want particular units placed, you should be able to ask the compiler how the chip looked just before it was placed, so that you can try to then place by hand.

Given these objectives we decided to design a family of MDL languages, each being a superset of the previous ones, all based on MDL. Thus, any MDL code will be legal in any MDL+ intermediate language, and MDL+ code will be allowed in MDL++, but not necessarily in MDL. Finally, MDL++ is a subset of Σ^* (Everything). This "MDL view of the Universe" is shown in figure 2-1.

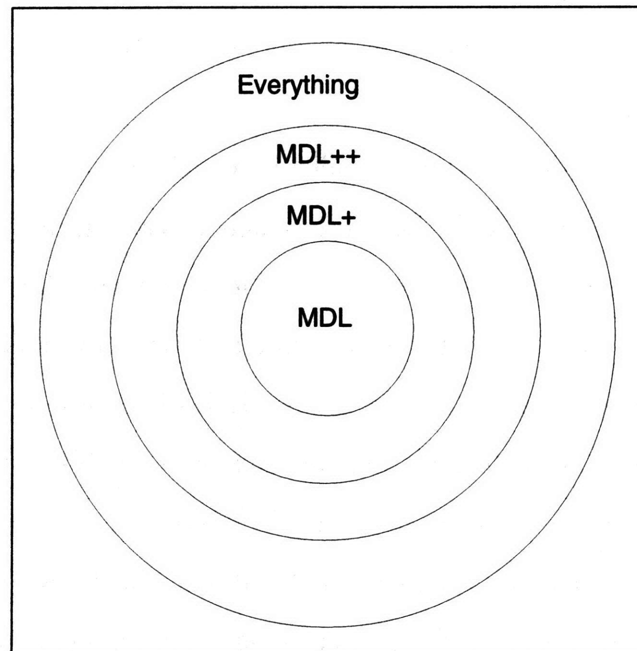

Figure 2-1: A Mathematical view of the various MDL Languages

Once we have decided on such a worldview, and given our objectives, it seems a natural decision to specify MDL+ as a family of languages that are strictly subsets of each other, all being supersets of MDL and all being subsets of MDL++.[1] We assign these MDL languages version numbers to correspond to the versions of the MDL+ compiler releases. Given that each language is a superset of the previous ones, each version of the compiler will be completely backwards-compatible, which was an implicit desire in our Programming Extensibility objective.

MDL+ version 1.0 is the same as MDL except that it is much easier to program in, essentially a language which could be desugared into the MDL Kernel. MDL+ 1.1 is a small step ahead, adding minor functionality which we no longer considered "just better MDL" but rather actually extensions to MDL. In practice, it is difficult to decide which basic improvements belong in MDL+1.0 and which should be considered additions available in MDL+1.1. Thus, we group all extensions besides the Automatic Driving Phase, and consider them to be in MDL+1.0 for practical discussions. Then,

[1]Keep in mind through this discussion that MDL++ has never been specified, and might never be specified. This is just a suggestion for it.

the sole addition to the MDL+ compiler for version 1.1 becomes the Automatic Driving Phase. MDL+1.1 is then considered "basic MDL+" and it is this language that is discussed in chapter 4.

MDL+ 1.2 is a version of the compiler in which routing can be done automatically, and thus the language MDL+ 1.2 is one where explicit routing information can be omitted. MDL+ 1.3 can then omit placing information, and MDL+ 1.4 actually adds new constructs which can be defined (math-functional constructs instead of physical-functional constructs) which the MDL+1.4 compiler then automatically groups together into physical units that are available on MATRIX. These extensions to MDL+ are discussed in chapter 5. All of the MDL+ languages fit onto a "Bullseye of Languages" that is the middle piece of the above worldview. (Figure 2-2)

Designing the languages such that each one is a superset of the previous ones is natural, and thus easy, because each compiler does more automatically than the previous ones, allowing the user to either use new higher-level constructs (more stuff, so a superset) or allows the user to omit more information (more possibilities allowed, less constrained, so a superset). In short, the more the compiler does, the more the options the user has, and since we never remove any possibilities the user now has a superset of the possible programs that he can use in the new language.

2.3 The Anatomy of the MDL+ compiler

Now that we have a good idea of how the various MDL+ languages will be related, in a mathematical sense, we need to specify how the compiler will be written. The mathematical definition gives us a clear idea of possible intermediate languages for the MDL+ compiler, namely the MDL+ 1.0 - MDL+ 1.4 languages. This is basically the direction we proceeded in. Since the compiler has an internal state that is not the lisp-like MDL+ code, it will not actually be storing these intermediate languages, but it will have internal state that will have a one-to-one mapping to these languages, and thus have the ability to print out the intermediate code as a user would understand it at any point. Although this code will lack the nice human-designed hierarchy of

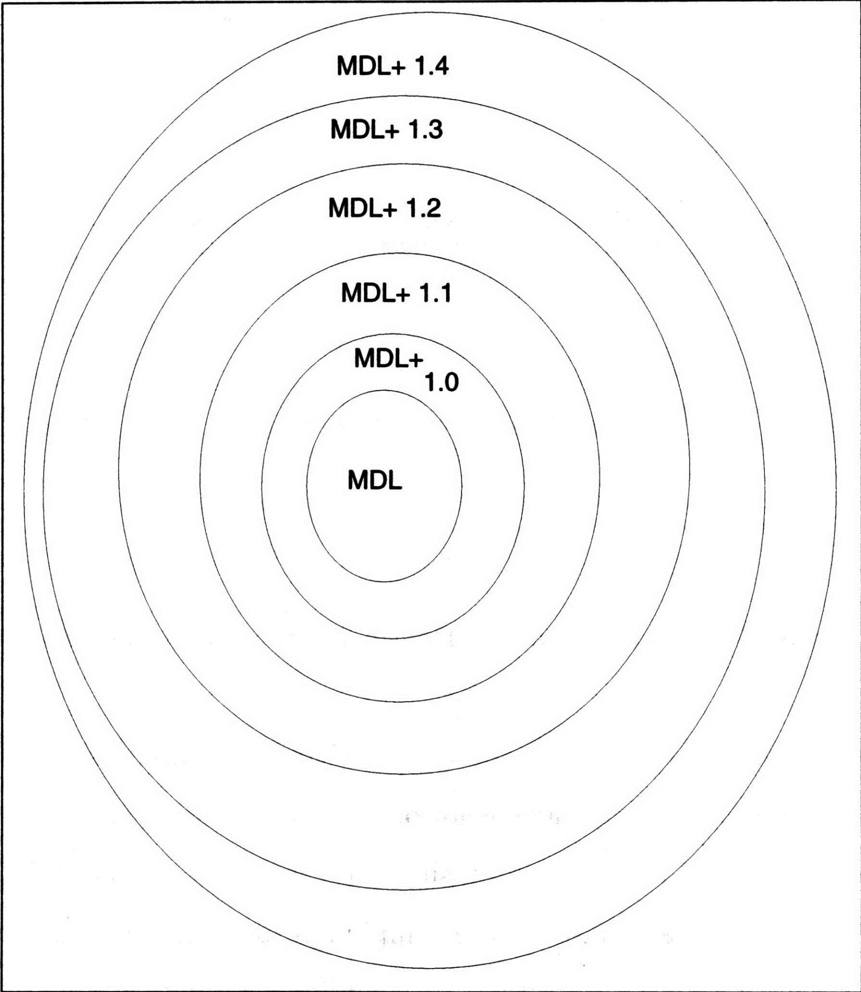

Figure 2-2: A Mathematical view of the MDL+ Intermediate Languages

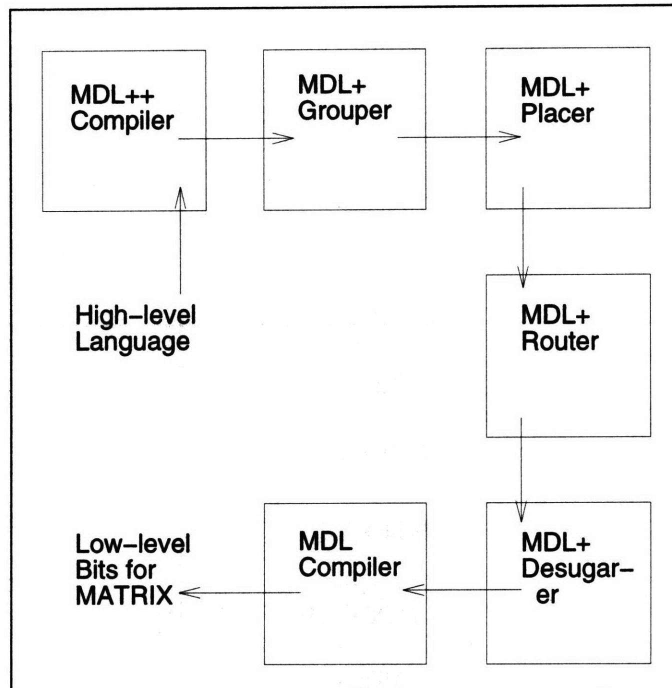

Figure 2-3: A Block Diagram of the proposed MDL++ Compiler

the original code, it still enables us to achieve our Human Interactions objective from above.

This design brings us to the picture of the MDL++ compiler shown in figure 2-3. Note that the output at each stage is not only corresponding to a legal intermediate-MDL+ code, but to all previous codes, since they were supersets of this intermediate language. However, these phases of the compiling process are not all just desugaring into some kernel of MDL+, but rather the compiler is actually automatically working on the code and adding new detail (without a unique choice for which detail), for example in the grouper, placer, and router pieces.

Of course, this figure is just a high-level view of the compiler as the actual MDL+ compiler does not make use of the MDL compiler. Nevertheless, the MDL+ compiler has a code generator which (while being better than the MDL compiler) could function as an MDL compilers' code generator. Similarly, MDL+ has a parser and scanner that can parse a superset of MDL, and thus could parse MDL.

2.4 Multi-level MDL+ Programming

Proceeding through chapters 4 and 5, they will start by explaining the "inner" languages, that are subsets of the others, and then work outwards until they have explained all of MDL+. In this way we will progress forwards through the languages (simple to complex) and backwards through the phases of compilation (last to first). On a very real level, understanding of all intermediate languages is essential to the writing of good efficient MDL+ code.

By Multi-level programming we mean that we can achieve the Multi-level Code objective from above. That is, MDL+ code can include code from all intermediate languages, and thus some of the code might completely specify a piece of the chip while other code might just ask the compiler to place an ALU in one of the units somewhere on the chip and connect it to other ALUs, whichever units they should fall in.

For applications which are either very important to optimize or just not being compiled to a state acceptable to the programmer, there also becomes a whole other way to view Multi-level programming. By describing the MDL+ compiler the way that we have, we have enabled the user to get into the guts of the compiler's work at each level. Thus, an ambitious multi-level programmer can write multi-level code (as described in the last paragraph) and then tell the MDL+ compiler to group it and print out the resulting MDL+1.3 code. He can then re-group the pieces which he does not like the automatic grouping of manually and run the MDL+ compiler on this modified output, telling it to only run the placer and output MDL+1.2 code, and so on. In this way, the programmer can view some of the compiler's automated actions as suggestions, while allowing the compiler complete control over the aspects that the programmer does not care about, is not capable of doing on his own, or is content with.[2]

It is now clear how all of the objectives which we sought are realized by this implementation of MDL+, and the reader is ready to proceed through the next few

[2]In professorial lingo, the compiler has become a grad student.

chapters secure enough in the big picture to understand the details.

Chapter 3

MATRIX

Before proceeding to the specification of the basics of MDL+ we take a break to discuss the hardware present on a MATRIX chip. This discussion of the theoretical basis for CGRAs and the actual hardware on MATRIX provides an ability to understand what the various constructs of MDL+ correspond to. This thesis is not actually about MATRIX, but since this thesis is about compiling for CGRAs and MATRIX, a lot of background about MATRIX is necessary before being able to understand the work that we did. This chapter provides that background, so that the next chapter can resume a discussion of the work actually done for this thesis. For further clarification of material in this chapter, see the thesis that proposed the idea [DeH96], the original paper to introduce MATRIX [MD96], and the MATRIX Micro-Architecture specification [Mir95a].

Much of the basic description of MATRIX in this chapter is taken directly from [Mir95b]. Many of the figures are from [Mir95b] and [Mir95a]. All of this other material was produced for Reinventing Computing and is used with permission of the authors.

3.1 Architecture Overview

MATRIX attempts to bridge the gap between microprocessors and FPGAs, such as those offered by Xilinx. [Xil94] The best way to achieve this is by examining the way

in which the two computing machines limit and specify their operations. [DeH95] Microprocessors place strict limits on their sets by specifying them in advance at fabrication time. The set is often limited to less than 256 instructions and therefore require very little "configuration data" (the actual instruction word - often 8 bits or less). FPGAs on the other hand, have a large amount of configuration data (up to 30 KBytes per configuration on some of the larger parts), but have an infinite instruction set which is not fixed at fabrication time. MATRIX, therefore, aims to limit its instruction set, but not absolutely fix it, at fabrication time. It should also reduce the amount of configuration data required to specify each instruction, which will help make each instruction much faster and allow for rapid instruction issue rates.

This is accomplished by creating a microprocessor-like basic block, consisting of a Memory and fixed ALU (Arithmetic Logic Unit), and connecting them in a reconfigurable mesh. Therefore, while each unit has an easy-to-specify configuration, the combination of many such elements allows for a great deal of flexibility.

3.2 Basic Unit

The Basic Functional Unit (BFU) of MATRIX is the coarse grain building block out of which more complex processing units can be built. It primarily consists of a memory block and basic ALU. Figure 3-1 shows the block diagram for a BFU cell.

The main BFU memory is a 256 word by 8 bit wide memory block, which is arranged to be used in either single or dual port modes. In dual port mode, the memory size is reduced to 128 words in order to be able to perform both read operations without increasing the read latency of the memory. In both modes this read operation takes place during the first half of the clock cycle and the values are latched for the rest of the cycle. Write operations take place on the second half of the cycle. This allows a BFU to perform register-file like operations such as A op B $\rightarrow$ A in one cycle.

The MATRIX ALU is a basic 8 bit arithmetic logic processing unit. It is capable of performing the following operations:

- **Pass**

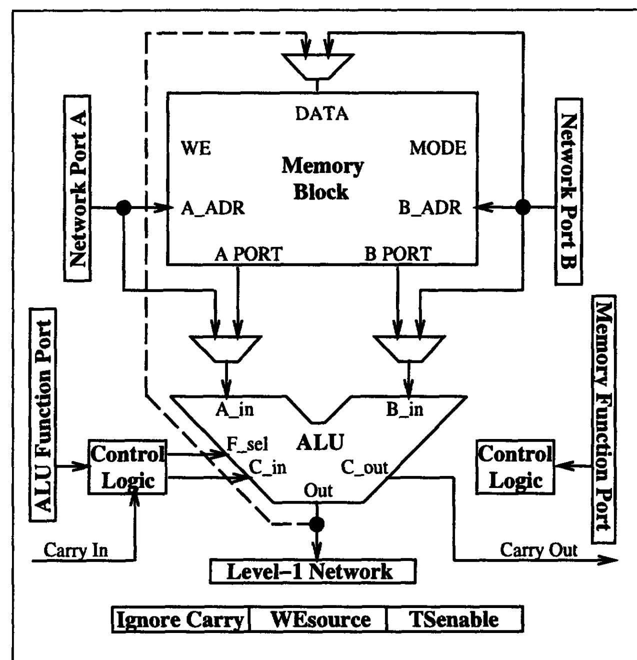

Figure 3-1: MATRIX Basic Functional Unit

- **Logic Operations** - AND, OR, XOR
- **Shifts** - Left and Right, with and without carry
- **Add**
- **Multiply**

With optional input inversion this also allows NOT, NOR, NAND, and subtraction.

Several adjacent blocks can also be composed to form wider-word ALU's. For example, four basic units can perform a 32-bit operation. The blocks can also be configured so that they can serve as a pure memory, pure ALU, or an ALU/memory (ALU/register file) combination.

Every input to the BFU is registered. This means that every BFU will be in its own pipeline stage which allows for highly pipelined systems.

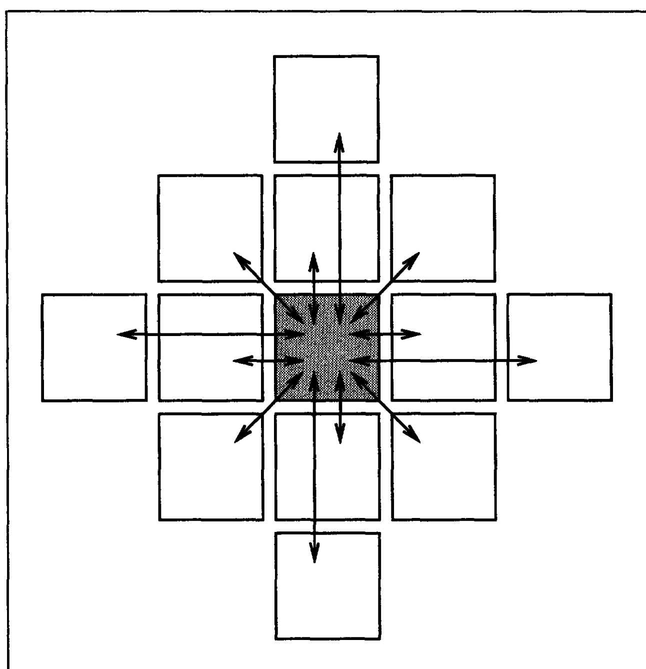

Figure 3-2: Level 1 Network Connections

3.3 Interconnect Network

The MATRIX network consists of three levels of interconnect structuring: a regular neighbor mesh, longer switchable lines, and long broadcast lines.

On the first level, output of every BFU is passed to its nearest neighbors in all directions, its neighbors 2 cells away in the cardinal directions, and itself. As a result every cell receives 13 Level-1 inputs. This network is intended to provide fast connections between tightly packed cells. Figure 3-2 shows the connections on the Level-1 network.

The second level provides length-4 broadcast lines along rows and columns of BFUs. Level-2 switches could operate in two modes. In one, the data passing through the switch would be registered, creating an extra pipeline stage in the network. This would be useful for synchronization. In the other mode, data would be passed without registering. This would allow for longer chains of network connections to be built. The possible number of links in these chains would depend on particular implementations of this design as well as the internal clock speed, but probably be as long as

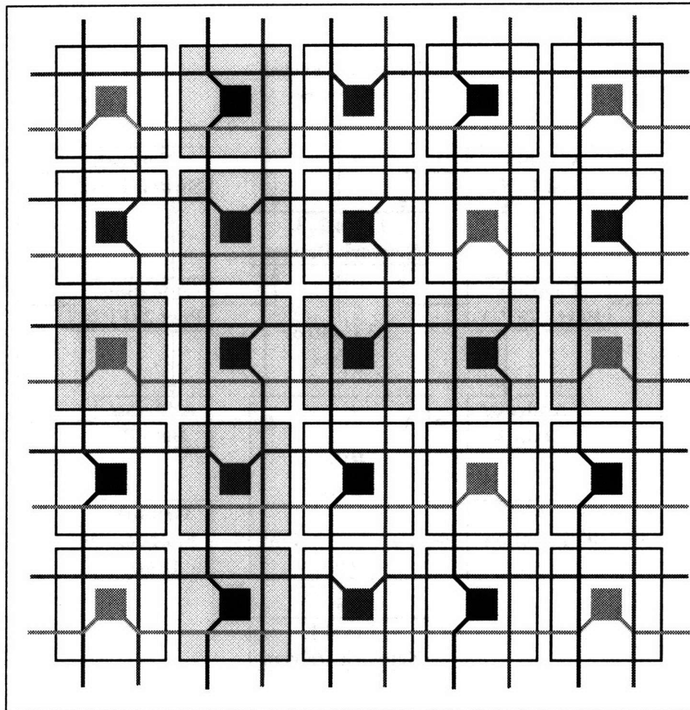

Figure 3-3: Level 2 Network Connections

three or four links. Due to the reduction in time to fabricate the MATRIX chip, as well as simplicity in design, the option of not registering level 2 lines has not been implemented in the current version of MATRIX, but it is still unclear whether it would have been worth implementing. We will, by using the MDL+ compiler, discuss whether the inability to leave level-2 lines unregistered has significantly hurt the ability to automatically route MATRIX.

The BFUs on the main diagonal drive horizontal level-2 lines, and every other BFU drives in the same direction, creating a checkerboard pattern. Each BFU drives two level-2 lines, and can view all eight level-2 lines passing over it. Figure 3-3 shows the level-2 connections on a five by five subgrid of BFUs.

The MATRIX level-3 network is intended to carry data long distances as rapidly as possible. It consists of 4 shared network lines spanning every MATRIX row and column. Each BFU cell gets to drive up to 4 inputs onto the level-3 network. The delay across level-3 is one clock cycle per step, with steps at this level being up to a full-chip long. Thus it is possible to get from any BFU to any other BFU in a

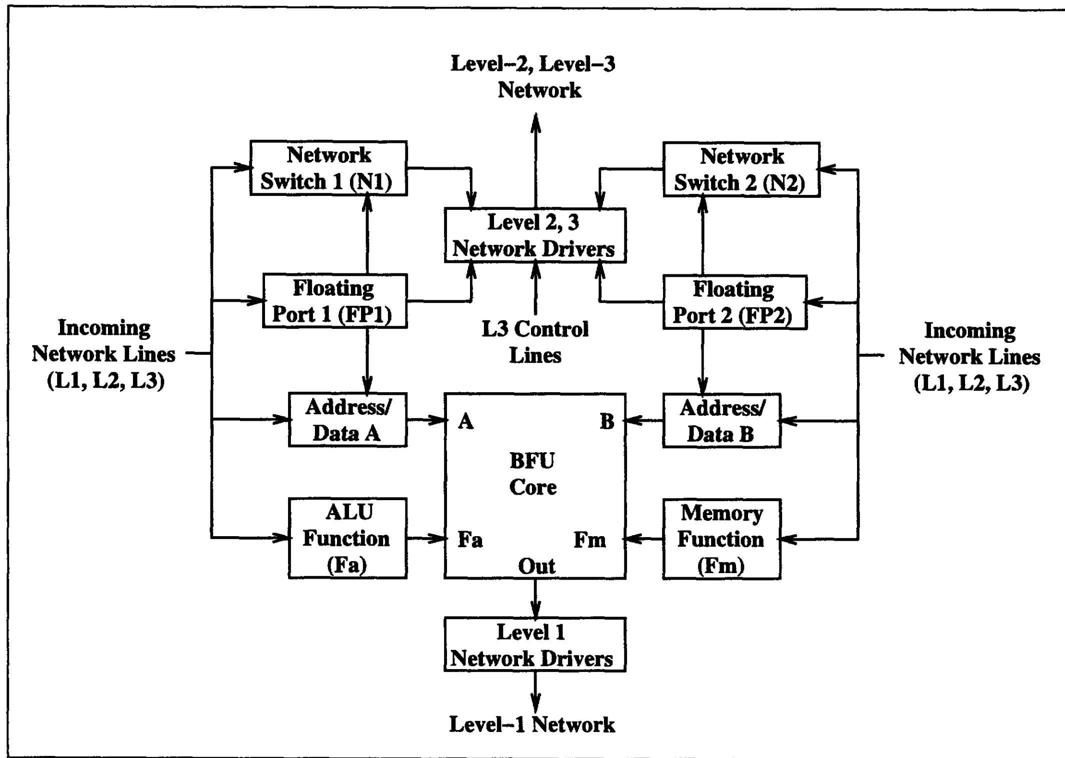

Figure 3-4: MATRIX Network Switch Architecture - BFU Cell

MATRIX array in 2 clock cycles.

3.4 Network Ports

The network is connected to the BFUs through the network ports, as shown in Figure 3-4. It is in these ports that much of the flexibility of the MATRIX architecture is found.

A typical port is shown in Figure 3-5. The port is controlled by a 9 bit configuration word. This word can be interpreted in two ways, based on the value of the 9th bit. In *Static-Value* (Constant) mode, the lower eight bits of the configuration word are passed directly to the BFU. This allows the port's value to be fixed to a pre-programmed value. In the other mode, *Static-Source*, the lower 5 bits of the configuration word select from the 30 incoming lines and the value on this line is propagated to the BFU. In this mode, the BFU can be controlled by another BFU,

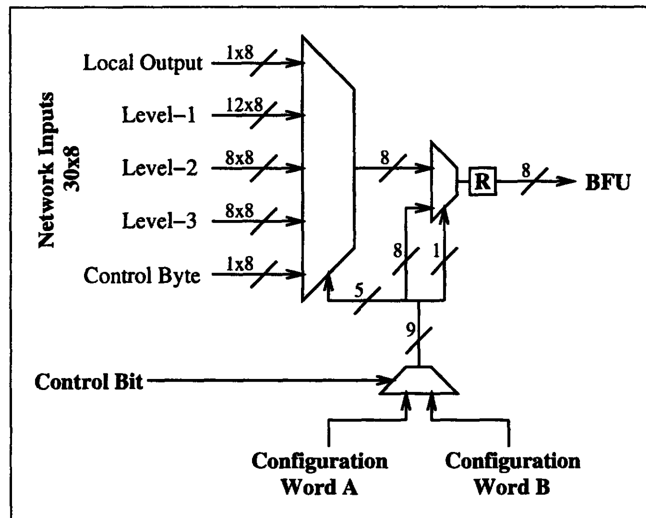

Figure 3-5: Function Port Architecture

or from an external source.

For the Address/Data ports and network ports, a third mode is allowed: *Dynamic Source.* In this mode the value coming from another network switch, the Floating Port on the same side of the BFU (see figure 3-4), can be used to be select from the 30 incoming network lines. Thus, the port's value can be fixed by the configuration word, come from a fixed source, or come from a source that can be selected on a cycle by cycle basis.

While three modes on 30 input lines gives MATRIX a lot of flexibility, the ports need to be very large in order to accommodate this many input lines and very large MUXs. The BFU would be a lot smaller if less inputs were available, thus providing more BFUs per chip. One goal of this thesis is to compile down our intermediate representation to BFUs while watching to see how many inputs are really needed per port, in order to determine if some of the current flexibility is not useful, or not as efficient as having several more smaller BFUs.

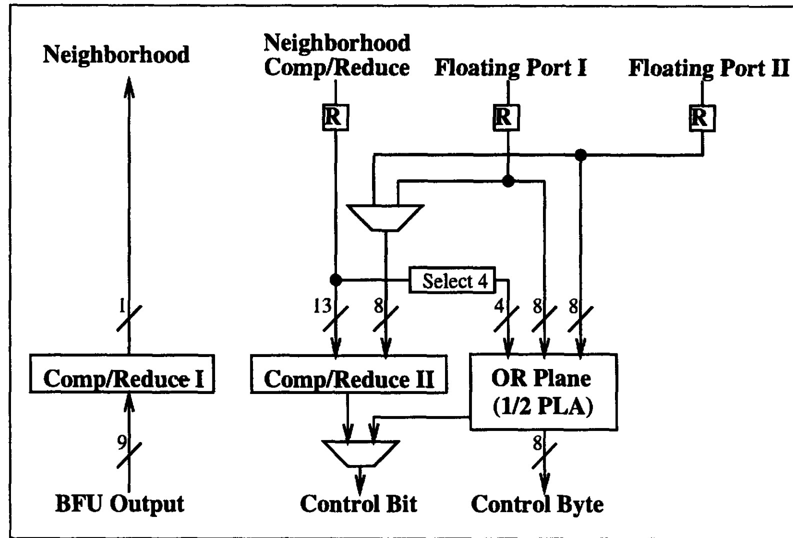

Figure 3-6: BFU Control Logic

3.5 Control

There are three portions of control logic in each MATRIX BFU, as shown in figure 3-6. First is Compare/Reduce I which compares the local ALU output to a given pattern and outputs a bit to every BFU which receives the level-1 output from the local BFU. Second, there is Compare/Reduce II which can compare any of the BFU's neighbors' Compare/Reduce I outputs and either of the floating ports, and outputs one bit. Third, there is the OR plane which can choose for input any four of the neighborhood Compare/Reduce I values and both of the floating port inputs, and can output 9 bits. Either the high bit from the OR plane output or the Compare/Reduce II bit can determine the local context of the BFU for the next cycle.

This control logic can probably be improved. The OR-plane is useful because it is the only thing on a MATRIX chip which can compose any arbitrary bit-level logic when combined with a second OR plane to form a PLA, as shown in figure 3-7. However, it takes up a lot of area, and is very complicated to test, and is therefore being left out of the initial design of MATRIX. One goal of this thesis will be to determine its usefulness, and whether it should be added to the next chip design. Similarly, we will explore other logic options, such as changing the Compare/Reduce networks,

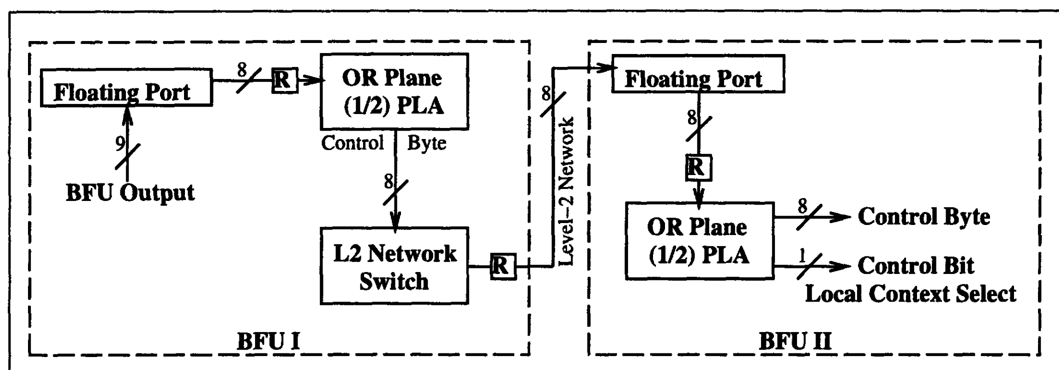

Figure 3-7: Distributed PLA

not broadcasting Compare/Reduce I as widely, or changing the entire control logic scheme.

3.6 Deployable Resources

The ability of one BFU to control another is one of the main fundamental advantages of MATRIX's coarse-grain architecture. For larger primitive units, there are not enough on a single die to make this kind of self-control interesting. For smaller cells, one cell is not large enough to generate the control information for other cells. Thus, it is only at this intermediate granularity that it becomes possible to deploy a chip's resources at this level of flexibility. We thus mention this aspect of MATRIX before completing our discussion of MATRIX and this chapter.

In terms of MATRIX, deployable resources means that every BFU can be configured to serve whatever function it is most needed for. Depending on the type of architecture MATRIX is imitating at the time, BFUs will be allocated to different functions.

Examples of how MATRIX can be configured to imitate various architectures, thus deploying its resources in different ways, will be given in chapter 6 where in addition to discussing MATRIX's ability to implement these architectures, we will discuss MDL+'s ability to compile them. In addition to compiling versions of several existing general purpose computing architectures, that chapter will discuss ways in which MDL+ and MATRIX can deploy resources on an application-specific ASIC-type basis. The earlier

part of the chapter will include several ASIC-type MDL+ designs for MATRIX.

Thus, we will give a lot of evidence that MATRIX's resources are truly deployable, in a way that cannot be achieved by very-coarse, or very-fine grain systems. Designers simply need to decide what kind of processing style suits their needs, and then descriptions of those architectures (or ASIC-like descriptions) can be compiled automatically into an application which will run on MATRIX, deploying resources to support the designers' preferred style. Developing this software is a large part of this thesis, and we therefore return to discussion of MDL+ with the next chapter.

Chapter 4

MDL+ as Better MDL

This chapter describes the basic syntax and semantics of the MDL+ language that corresponds to configuration details that are actually loaded onto the chip. Thus, it will be describing how MDL+ includes the functionality of MDL, and what basic forms of syntactic sugar MDL+ offers. This chapter will not discuss the aspects of the MDL+ compiler which are at a fundamentally higher level, such as the automatic routing, placement, and grouping that the compiler can do. Those aspects, as well as the syntactic elements of MDL+'s grammar needed to support them, are discussed in Chapter 5.

The first part of this chapter is devoted to explaining the basics behind MDL+. The next part discusses the additions to MDL+ that are more than just cleaning up MDL, but not actually intelligent phases of the compiler, namely MDL+'s automatic driving phase. The end of this chapter is devoted to explaining the basic output modes and compiler options that MDL+ provides, in order to make this chapter complete and sufficient to enable the reader to write an MDL+ source file and then compile and use it himself. For the complete grammar defined in this chapter, see Appendix A.

4.1 Language Basics

MDL+ is a lexically scoped call-by-value language using a LISP-style syntax. MDL+ uses a LISP-like syntax for several reasons. It is easier to parse, seems to fit the problem in that the language constructs actually correspond to physical resources, and because MDL used a LISP-like syntax and it is easiest on MDL programmers if we change the appearance of the language as little as possible.

The actual layout of an MDL source file is very simple: it is a list of successive definitions. These definitions are read by the compiler in-order, and thus are read in the environment defined by the definitions above the current one in the input file. If any variables are defined above the current definition in the file, they are thus bound in the current defining-environment, and so they can be used by the current definition. Since MDL+ is lexically-scoped, they will be assumed to mean whatever they were defined as in the file at the point above their use that is nearest their use. Any places where variables are used in a more dynamically-scoped way will be explicitly mentioned when the construct is explained.

4.1.1 Basic Statement Structure

Since this is a LISP-like syntax, each definition is enclosed in parenthesis. Each definition is a definition of a specific type of construct, and thus begins with a statement that indicates which type of construct it is defining, such as **def-bfu** or **def-port**. Since each definition is defining a specific instance of that type, the instance must be named, and this name is always the second item listed, just after the **def-** item. Finally, prior to the end parenthesis, the construct is defined with appropriate arguments. These are discussed for each construct in the next section. Often it will be allowed for these arguments to be omitted entirely, producing the default item of that construct type bound to the appropriate name for future definitions to use.

4.1.2 Identifiers

The name given to an instance of some construct type must be a legal identifier, which probably includes anything that it might have been named. Specifically, it must start with a letter (lowercase or capital), an underscore (_), a colon (:), a period (.), a hyphen (-), a forward slash (/), a dollar sign ($), a less-than sign (<), or a greater than sign (>). Any subsequent characters can be any character that could have started the identifier, but they can also be digits.

It is recommended that names be constrained to letters and digits but it does not have to be so. Since MDL+ is context-insensitive where identifiers are concerned, using an uppercase version of a letter yields the identical effect of using the lowercase version of that letter.

4.1.3 Numbers

Whenever a number is required in an MDL+ file, it can either be represented by a sequence of digits which will be interpreted as a decimal number, or precede a sequence of hexadecimal digits with **0x** or **0X** which will be interpreted as a hexadecimal number.

4.1.4 Compare/Reduce Numbers

Besides decimal and hexadecimal numbers, MDL+ will also recognize Compare/Reduce-Numbers or rednumbers. They should be prefixed with a **0r** or **0R** and each character or bit of the number must be a **0**, **1**, **x**, or **f**. These letters are case insensitive, and an **x** means that the bit compared against can be either a 0 or a 1, while an **f** means to always fail. These numbers are only for use with the Compare/Reduce control-system of MATRIX and are not allowed most places where normal numbers and hexadecimal numbers are allowed. These numbers are also known as rednums.

4.1.5 Don't Cares

One concept that will appear throughout MDL+ is that of *Don't Cares*, or *DCs*. Basically, whenever there is a configuration value or argument that could be entered for some construct in MDL+ that the programmer does not care about at all, he can enter "DC" in the MDL+ code instead, or omit the argument entirely. For specific usage options, see the part of the next section that deals with the specific construct that is of interest, since sometimes there are limited options, in order to clean up unnecessary aspects of the grammar a bit. Usually, everything can either be omitted entirely, or its values filled in with Don't Cares.

4.1.6 Copying Constructs

It is almost always permissible to define a construct with only one argument, where that argument is either DC or the name of an instance of the same type of construct which is defined in the current environment. Thus, for almost any legal type, it could be substituted in the following code for *mytype*, thus producing two legal copies of the type in the environment, both completely unspecified.

```
(def-mytype a dc)
(def-mytype b a)
```

At the point of the MDL+-code just after these two lines, *a* and *b* would both be bound in the environment to completely unspecified (not caring) instances of the type replacing mytype. Since MDL+ definitions are all passed arguments call-by-value, a and b would refer to separate but identical instances, and not be merely aliases of each other.

4.1.7 Inheritance

Another recurring theme in MDL+ is that of inheritance. With DCs, the ability to rearrange and omit a lot of the MDL+ code, and inheritance, MDL+ files are much shorter, more manageable, easier to write, and easier to understand than MDL files

used to be. Inheritance is not as pervasive in MDL+ as DCs are in that not every construct can inherit from other constructs of its type. But the more useful examples do have inheritance. For example, a BFU can be defined as being exactly the same as another already-defined BFU, except for some explicit differences. Since the parent BFU is evaluated at the time of definition, in a lexically-scoped manner, changes to the parent BFU later in the source file will not affect the child BFU at all. Thus, it might be best to view inheritance in MDL+ as copying an old instance of the type to the new instance and then specifying changes to it.

This other view is not only more accurate, but it also provides insight into other possibilities for how to use inheritance in MDL+ that a traditional view might not encourage. For example, in a BFU definition, the changes to the parent BFU can be specified prior to the parent BFU being named. In that case, the new BFU is formed by first specifying a new implicit BFU from the new specifications and then treating it as the parent of the actual explicitly-defined parent, and inheriting in reverse-order. This will be explained further when there is a basis for understanding the specifics, once the BFU has been discussed, in the next section.

4.1.8 Reserved Words

MDL+ has two types of reserved words that will thus be interpreted with special meanings: Globally reserved words and locally reserved words. Globally reserved words are special everywhere in an MDL+ file and thus can not be used as identifiers. Locally reserved words are only special in specific contexts and thus are fine for use in most cases. All globally reserved words are listed in appendix C, while locally reserved words should be obvious for each context.

4.1.9 Lists

There are many parts of the MDL+ grammar that call for lists. Sometimes these are sequences which are being defined, such as the sequence of all memory cells for a BFU. Other times, each element of the list is intended to convey more information

about the configuration of a single item being defined. In this case, it is possible for the configuration information from some of the list to contradict with information from other elements of the list. When there are contradictions, the later elements (right-most) will take precedence over the earlier elements (left-most). However, any pieces of the earlier definition which are not contradicted by the later definitions will remain.

For example, saying that the last two bits of a number are "01" followed by saying that the second to last bit of the number is "1" will be understood as saying that the last two digits of the number are "11". In general, this is the way the compiler will understand items of configuration at least as large as a single number, while any information smaller than a single number (a piece of a number) will actually be completely replaced by later references. If our example had been the number 1 and then the number 2, the compiler would understand the list as just the number 2.

4.1.10 Ordering Directions

There is a convention throughout MDL+ and MATRIX programming in general of an implicit ordering on the opposite cardinal directions. South or down is "less than" or "comes before" north or up. Similarly, west or left is less than or comes before east or right. This convention is prevalent in many elements of the syntax and semantics of MDL+, as explained in the next section.

4.1.11 Going Beyond MATRIX

Proceeding through the explanations of basic MDL+ in this chapter, there will be several elements for which configuration is provided even though they do not exist on the MATRIX hardware. These are generally elements that could have been used in MATRIX, but were taken out of the design at some point either to conserve area on the chip or to reduce the amount of time until the chip could be fabricated. They have been included in MDL+ because they might be useful for simulation of hypothetical chips, as well as a desire to be compatible with MDL source code.

4.1.12 Definitions

Here we define many words that will be used a lot in the upcoming section to describe aspects of MDL+. All of them should seem to fit with their definitions fairly naturally, but they do mean more in the context of MDL+ than they would in standard language.

- **Construct** - This is a datatype for MDL+. In MDL+ code, there can be definitions of constructs, which would then leave an instance of that construct in the environment. They generally correspond directly to hardware on MATRIX, but not always. The only types in MDL+ that are not constructs are simple types: Numbers, strings, rednumbers, and Don't Cares. Examples are *BFUs* and *Memories.*

- **Instance** - A specific example of a construct. After definition of a construct, there is an instance of the type in the environment. Two instances of a construct are always unique and not aliases of each other.

- **Constructor** - A globally reserved word in MDL+ that is used to indicate the beginning of the definition of some construct. After that definition, there will be a new instance of that construct. Each construct has its own constructor. Examples are *def-bfu* and *def-mem.*

- **Structure** - A description of a piece of the grammar of MDL+. This is a sequence of words and parenthesis that explain how a certain amount of the MDL+ syntax is used. Some of the words will need to be written in the MDL+ file exactly as they are in the structure, and some are arguments that will need to be replaced. The only other notation used in structures is Kleene closure. There may be multiple structures allowable in MDL+ to do the same thing: One structure for defining BFUs is **(def-bfu Name Old)** where Name is an argument that is the string which names the BFU being defined, and Old is an argument that is the string which names an already defined BFU. A second structure for defining BFUs is **(def-bfu Name DC)** where Name is the same

kind of argument as above. Structures listed in the text will always be boldfaced.

- **Basic Structure** - All constructs in MDL+ can be defined with two structures such as those listed above for BFUs, one to copy an old instance of the same type and one to create a completely unspecified instance of the type. Clearly, these are not a sufficient set of structures. The basic structure is the structure that includes all alternatives besides these two for defining a given construct. It is considered basic because it is the only one that the language could not do without.

- **Keyword** - Many basic structures will be of the form **(Keyword Stuff)** where Keyword will need to be replaced by a specific string that is either a global or local reserved word, and Stuff will need to be replaced by something special to the construct. The word replacing Keyword for one of these constructs is considered its keyword because it identifies the type of definition that the program is making. Constructs can have no keywords, one keyword, or many keywords.

- **Argument** - This is a word in a structure that is not meant to be used exactly as written in the MDL+ code, but rather to be replaced with something else, perhaps something described by its own structure.

- **Default** - If it is possible to define a construct with only a constructor and a new name inside a pair of parenthesis, then the default for that construct is whatever instance such a definition would produce. If not, then the construct can still be said to have a default if it is possible to define it with the basic structure of only a sole keyword inside a pair of parenthesis.

- **Replacing** - This means that one thing will get completely eliminated and another put in its place. When A is replaced by B, the result is always exactly the same as B. This word is used in conjunction with *overriding* in order to be more specific about the behavior of inheritance and lists in MDL+.

- **Overriding** - This means that one thing will use its parts which are specified to replace those parts in another thing. When A is overridden by B, the result has in common with B every piece of B which was specified, and in common with A any piece of information that was not specified in B. When a child inherits from a parent, the child overrides the parent, as right-most elements of lists override left-most elements of lists.

- **Reverse Inheritance** - Whenever inheritance is allowed in MDL+, the parent may also be listed after the definition of the child in which case the parent overrides the child instead of the child overriding the parent.

- **Full Inheritance** - This refers to a situation in which the parent may be listed at any point in the child's definition, and multiple parents may be listed. The items listed later always override the items listed earlier. Whenever inheritance is available in MDL+, full inheritance is available.

4.1.13 Type Checking

MDL+ is a type-checking compiler, and will thus halt with an error message if a name is used which is currently bound to an instance of a type which is not one of the types allowable in the place that the name was used. For example, the first local context of the A port of a BFU must be set to either the name of another BFU (a string that is the name of another BFU)[1] or to a constant value. Thus, if it is given a variable that is of type chip, the compiler will halt, since the user program did not type-check.

For example, consider the code in figure 4-1. The first line defines a chip named *a* and the remaining lines attempt to define a bfu named *b* which has the first local context of its A port defined to be that chip named *a*. It could have defined the A port value to be a number, e.g. *8*, or the name of a constant variable, or even another BFU, but not a chip. (See the explanations of BFUs (4.2.11), constants (4.2.1), and

[1]The syntax for this case actually includes more than just the string, e.g. "(static bfuname)"

```
(def-chip a)
(def-bfu b
  (ports
   (aport a)
   )
  )
```

Figure 4-1: Source code that does not type-check.

```
Parse Error:
  need a constant or aluobj, you have an identifier that has been
defined to be something else.  Only constants and aluobjs can be used
here.  If your ID is a BFU, you might want to wrap it in a (static) if
you meant to use it, automatically routed, as static source.  Bad ID: A
Last Line Read: 4
```

Figure 4-2: Typical error message for code that does not type-check.

chips (4.2.15) to avoid any confusion and completely understand this.) Thus, trying to run this code through the MDL+ compiler would cause it to halt with the message in figure 4-2.

4.1.14 Output

One of the things that is often most mysterious when learning a new programming language is how to get any output out of it. In the case of MDL+, it is clear that there are no statements other than definitions for the environment, so it is especially unclear how the compiler will know to output something. In fact, if an input file does not complete any chip definitions, the compiler will parse the input, and if it is correct, the compiler will halt without outputting anything. However, when one or more chips are defined in an MDL+ file, the compiler will process them with its various automatic phases (discussed at the end of this chapter and in the next chapter) and then output each chip to its own file. The form of the output will depend on the command line arguments provided to the MDL+ compiler. It is illegal to define more than ten chips in any one MDL+ file, but there is not much reason why that would

be done anyway.

The forms of output, and how to ask the compiler for them, will be discussed in much more detail in section 4.4 at the end of this chapter. At this point, all that is important is that if an MDL+ source file has a chip construct defined in it, then the compiler will be able to produce from it a configuration file which can be loaded onto a MATRIX chip to run the way the chip was specified in MDL+.

4.2 Language Constructs

The previous section provided a basic overview to the syntax and semantics of MDL+. As with most LISP-like languages, the basics are very simple and all of the functionality lies in the details of the primitives. In this case, the primitives correspond to the types of the language, or the various constructs. An instance of each type can be created and placed in the environment with an appropriate definition statement, as stated in the previous section, and then used later in the file.

All that is left is understanding each construct, which will correspond to some hardware on the actual MATRIX chip. This section will go into detail to explain each construct. After reading about any construct, it should be a simple matter to write an MDL+ file that uses that construct.

4.2.1 Constants

Constants are the lowest level type that MDL+ has. (Numbers, Rednumbers, and name strings are not types.) There are many different kinds of constants and they are used in many different kinds of other situations. Many times, using one kind of constant in a situation intended for another kind of constant will be a mistake, but MDL+'s type-checking will not catch these errors, because as far as the compiler is concerned all constants are equally appropriate for any places that need constants. The benefit to this is that there are times when it might be desirable to use one kind of constant in a place where a different kind is desired, and the compiler will not stop you.

Constants are the lowest-level type in MDL+ and very useful because they are basically just numbers. The difference between a constant and a number are twofold: Constants can be bound to names and stored in the environment whereas numbers can not be, and constants can also be DC, meaning that it does not matter what number the constant represents, whereas numbers are just integers.

But the power of constants should not be underestimated just because they are almost equivalent to numbers. Their greater power lies in providing an abstraction layer between concepts and the lower-level number that is required for a MATRIX chip to be configured to behave with the concept. Several ways to define constants will shortly be described and they will be interpreted by the compiler to mean some number and thus stored as constants that are just numbers, but they will only be seen by the human user as higher-level concepts.

The only exception to the rule that all constants are as good as any others, is that sometimes situations that call for constants will require constants in a certain range (such as 0..31 for constants that are supposed to fit into five bits). In these cases, the compiler will complain if the constant variable it is given has a value not in the appropriate range. These are errors that are closer to run-time than the other type-checking errors, but they are still compile-time errors.

Before continuing into the details of the ways in which constants can be defined, there is one last warning. The compiler will always turn a constant definition into either a number or a DC value, but that does not mean that it turns the definition into the same number that you might think it should. If you then use that constant in appropriate places, the compiler will know what values it meant by the number it assigned to the constant and it will operate correctly. However, if you use it in a place where a different kind of constant was assumed, then it is not guaranteed to work correctly (it might, but it should particularly be avoided when it seems most unusual). An example of a constant definition and use that would not be guaranteed to do what you thought it should[2] is defining a constant to be an Enable-value and then using it as a Selection-value.

[2]It's not even clear that there would be a unique assumption of what it should do.

Description	String
leftmost or bottom BFU	BFU1
nth BFU, $1 \leq n \leq 6$	BFUn
Opposite-side perimeter VBFU	otherio, otherside
Same-side perimeter VBFU	sameio, thisside, thisio, sameside
Undriven	zero, constant0, none

Table 4.1: String meanings in (const String) structure

Basic Definitions

Constants can be defined to be DC, like most other constructs. They can also be defined to be just a number, or to be the value of a name that is bound to another constant. Although these simple definitions are the only means of creating constants that are ever needed, they are far from the only means that would ever be desired.

All definitions of constants use the constructor **def-constant**, and all must provide the definition with some argument, since there is no default value.

Constant Value

There are two ways in which you can define a constant to be of the "Constant Value" or "Static Value" kind. In the realm of BFUs "Constant Value" and "Static Value" refer to 8-bit numbers that any of the eight ports can be set to. Thus, a constant can be defined to be **(constant NUM)** where **NUM** is any 8-bit number (Any number zero through 255.)

In the realm of VBFUs, or the perimeter of the MATRIX chip, "Constant Value" and "Static Value" refer to the information that controls which BFU is allowed to write a given Level-3 line. This information is considered constant or static because it is configured to be the same BFU every cycle. To define a constant this way, define it as **(constant String)** where **String** is **BFU1** for the first BFU (in order bottom to top and left to right), **BFU2** for the second BFU, and so on, or other values as explained in table 4.1.

Any other strings used in the **(const String)** structure must be names of con-

Description	String
Local BFU	local
L1,L2,or L3 line	l1_n1, l1_sw, l2_e2, l3_h4, etc.
Control Byte	ctrl
Memory Read-Out Data	md
Constant Value 1/Config read data	c0
VBFUs diagonals	l1_n, l1_nd, etc.
VBFUs L1,L2,L3 lines	l1_2, l2_1, l3_4, etc.
Ioport	io_port, ioport
L3_3 High bits	highbits
Undriven/No source	none, constant, constant0, zero

Table 4.2: Strings in Static Source Structures

stants in the environment. As explained above, these constants could have been defined using any means of constant definition.

Static Sources

Just like constant values or static values, there are two places that static sources are used on a MATRIX chip. One is in the realm of BFUs where a port can be defined to have whatever value is coming from a specific input line to that BFU. The port does not provide a constant or static value, but it does provide the value that is coming from a static or constant source every cycle. In the realm of VBFUs, the arbitration of which BFU gets to drive a Level-3 line can be left up to the low bits from an input line to that VBFU on a cycle-by-cycle basis. The choice of which BFU gets to drive the line is not constant or static, but the source of the decision is static.

Both of these situations are encoded into MDL+ as constants with the **(static String)** structure. When using this structure, the String must either be the name of a line that enters the BFU ports (e.g. l1_n1) or the name of a line that enters the VBFUs and can thus arbitrate the control of level-3 lines (e.g. l1_1). The possible values the String can take on are listed in table 4.2.

No other values are allowed for the string in a **static** structure, not even the names of already-defined constants.

Dynamic Sources

BFU ports can be set to be in Dynamic Source mode, in addition to Constant Value Mode or Static Source Mode. In this case, the port looks to one of the Floating Ports (the one on the same side of the BFU as it is) and uses the value of that port each cycle to determine the line it will choose as the source of its value. Since no further configuration is possible once a BFU is in dynamic source mode, and there is no such mode available for VBFUs when determining which unit drives a level-3 line, thus the syntax in MDL+ of specifying a constant to be a dynamic source is very simple: it is a **(dynamic)** structure. So, for example, a line in an MDL+ file might read:

```
(def-constant dyn-con (dynamic))
```

ALU Values

Describing the functioning that the ALU should be performing in a single cycle is complex as a concept, but that complex concept is actually implemented as an 8-bit configuration byte. Thus, ALU values are legal constants. A novice MDL+ programmer might wonder what additional benefit is gained by allowing ALU values to be interchangeable with other constants instead of just allowing them to be inserted in the ALU port configuration of BFUs. The answer to this is simply that ALU values belong almost anywhere in a chip's configuration that general constants belong. An ALU value might go into a memory cell or into another BFU's port such that it could eventually be passed into the BFU whose ALU port will use it to run an actual ALU.

Just as constant values, static sources, and dynamic sources have keywords to tell the compiler what kind of constant is being defined, ALU values have the keyword **aluval**. The structure of an ALU value definition is as follows: **(aluval (Inst Ctx We))** where any of the three arguments Inst, Ctx, and We can be omitted, but those present must come in order.

If it matters which of the instructions the ALU will carry out in the cycle that this ALU value controls an ALU, then the Inst argument to the ALU Value structure must be specified. There are three types of instructions that might be passed in: shift

instructions, pass instructions, and general instructions.

To generate a shift instruction, use the following structure for the Inst argument: **(shift arg1 arg2 arg3)**, where arg1 is simply the letter **a** or **b**, arg2 is the letter **r** or **l** and arg3 is **f**, **c**, **0**, or **1**. The first argument determines whether the ALU will shift its A input or its B input. The second argument determines whether the ALU will shift right or left (towards LSB or MSB). The third argument helps to determine what will be done with the newly-empty position after shifting: **0** or **1** will insert a 0 or 1, respectively. **f** will override the LSB/MSB setting of the BFU and force the shift to use the carry-in from its designated Leftsource or Rightsource, which can be useful for shift-rotations. **c** will keep the same bit that used to be in that slot, essentially duplicating the low or high bit of the shifted number. For a better understanding of these various configurations that a BFU could be in (MSB, LSB, Leftsource, Rightsource), see section 4.2.5 on BFU-Configuration constructs.

To generate a pass instruction use the **(Pass Inv)** structure, where **Pass** is either the string **PassA** or the string **PassB** depending on which ALU input the ALU should be passing, and **Inv** is either the string **Inv** or omitted depending on whether or not the ALU should be inverting all bits of the value it is passing before passing it along.

To generate an arithmetic or logical instruction use the **(Arith Inv)** structure, where Arith is one of the legal MATRIX arithmetic or logical operations that the BFUs can perform, and the Inv argument is **InvA**, **InvB**, **InvA InvB**, or omitted depending on which (if any) of the inputs to the ALU are supposed to be inverted prior to the operation taking place. The legal operations are represented by the following strings: and, or, xor, add, add0 (add but the carry-in of any LSB BFU is forced to 0), add1 (the carry-in of any LSB BFU is forced to 1), mul, mula (Multiply-Add, A*B+FP1), mulaa (Multiply-Add-Add, A*B+FP1+FP2), and mcon (Multiply-Continue, get the high byte of the previous cycle's mul, mula, or mulaa).

Now that the ways to specify an ALU value's Inst argument have been explained, we continue to the Ctx and We arguments.

If it matters which of the two Compare/Reduce I words the output from the ALU

is compared with, specify the Ctx argument as **(ctx 0)** or **(ctx 1)**.

If it matters whether the BFU's memory has the write enable bit set for that cycle, specify the We argument as **(we 0)** or **(we 1)**.

If any of the three arguments are passed to the ALU value structure, the omitted arguments will all default to zero. But, if all three defaults are desired then they must be written out explicitly, **(aluval ((mul) (ctx 0) (we 0)))**, since omitting all arguments will result in an ALU Value of Don't Care, and a resulting constant defined in the environment of DC.

Memory Values

Just as there are constant values to control the ALU function of a BFU, there are constant values to control the memory function of a BFU (that is inputted through the MEM port of a BFU). The keyword for this kind of constant definition is **memval**. The structure to use is **(memval (Port AluA AluB Memdat Cfgwrite Cfgread))**. Any of the six arguments can be omitted. As with ALU values, if all of them are omitted then the value will be the DC constant, but omitting any less than all of them will just cause the omitted values to default.[3] Most of the arguments have an obvious connection to the MATRIX hardware in figure 3-1.

The Port argument can be passed as either **(Port single)** to indicate one 256-byte memory, or **(Port double)** to indicate that the memory of the BFU is to be used as two 128-byte memories, which can be accessed by different BFU ports (A,B). The default is a single port.

The AluA argument controls one of the muxs just after the Memory unit of a BFU. The AluA argument can be passed as either **(AluA Input)** to flip the mux to passing the BFUs A port value, or **(AluA Memory)** to flip the mux to passing the output of the memory's A port. The default is Input.

The AluB argument controls the other mux just after the Memory unit of a BFU. The AluB argument can be passed as either **(AluB Input)** to make the mux pass

[3] Due to a known bug in MDL+, you can not omit arguments to a Memval prior to any that you do not omit.

the BFU's B port value, or **(AluB Memory)** to make the mux pass the memory unit's B port. The default is Input.

The Memdat argument controls the mux that comes just before the Data input port of the Memory Unit. The argument can be passed as either **(Memdat InputB)** to have the mux pass the BFUs B port value to the memory as its data, or **(Memdat local)** to have the mux pass the ALU's output from the current cycle back to the memory unit to act as the data for this cycle. The (Memdat local) memory configuration is required for doing operations of the form $\mathbf{A} \leftarrow \mathbf{A}$ **op B** in one cycle. The default is InputB.

The Cfgwrite argument controls the configuration memory write enable. The two options for the argument are **(Cfgwrite disable)** to not assert the enable, and **(Cfgwrite enable)** to assert the enable. The default is **(Cfgwrite disable)**.

The Cfgread argument controls the configuration memory read enable. The two options for the argument are **(Cfgread disable)** to not assert the enable, and **(Cfgread enable)** to assert the enable. The default is **(Cfgread disable)**.

Select Values

Select Values are constants that are used for the 12 IO Ports that are at the perimeter of the chip. Each has three configuration words for output selection (One for the data word, and one for each of two I/O bits.) These values decide which values are outputted off-chip in the event that the port's output enable is set.

The structure of a constant definition for select values uses the keyword **sel** and looks like **(sel NUM)** or **(sel Words)**. If a number NUM is given, it must be a four bit number, and it will be directly used to code the low-level bits. If Words, or a sequence of strings, is given then they are used to determine as much as possible about which line the programmer meant to select. The descriptions of what these strings can choose, and the related strings that can be used, are shown in table 4.3.

In any case where an entire word is selected but only for an I/O Bit, the low bit is chosen. Note that no one string will completely determine which line is selected. If only some of where the line must be is determined, then the rest will default to

Description	String
Lower or leftmost Column	A, columnA
Higher or rightmost Column	B, columnB
CR from a BFU 1 or 2 away for I/O Bits	CR_1, CR_2
L1 lines a BFU 1 or 2 away, for I/O Word	l1_1, l1_2
L2 lines a BFU 1 or 2 away	l2_1, l2_2
L3 lines	l3_1, l3_2, l3_3, l3_4
ID naming a constant with value 0..15	ID

Table 4.3: Strings defining Selection Values

one of the lines which the specification did not eliminate. It is possible to completely specify a line, for example in this constant definition of a selection value:

```
(def-constant sel-con (sel B L2_1))
```

Finally, since the sequence of strings can be an empty sequence, a Selection Value can be completely unspecified, as in this code:

```
(def-constant sel-any (sel))
```

Enable Values

Just like Select Values, Enable Values are constants that are used for the IO Ports that are at the perimeter of the chip. Each has three configuration words for the output enables (One for the data word, and one for each of two I/O bits.) These values decide whether each of the I/O Word and bits are inputted on-chip and which are outputted off-chip.

The structure of a constant definition for enable values uses the keyword **en** and looks like **(en NUM)** or **(en Words)**. If a number NUM is given, it must be a four bit number, and it will be directly used to code the low-level bits. If Words, or a sequence of strings, is given then they are used to determine as much as possible about what the programmer wants the enable value to be which is often a question of which line the programmer meant to select as the line whose bit will dynamically

Description	String
Lower or leftmost Column	A, columnA
Higher or rightmost Column	B, columnB
L2 lines a BFU 1 or 2 away	l2_1, l2_2
L3 lines	l3_1, l3_2, l3_3, l3_4
I/O Bit 0 or I/O Bit 1	io_0, io_1
Always Input	input, zero
Always Output	output, one
ID naming a constant with value 0..15	ID

Table 4.4: Strings defining Enable Values

decide the output enable value each cycle. The descriptions of what these strings can choose, and the related strings that can be used, are shown in table 4.4.

Note that while any Enable Value can be chosen from any of these possibilities, only the Output Enable of the Data Word at each port has much of the flexibility. The two I/O Bit output enables must be constant, either always input or always output.

Also, note that like Selection Values some strings will not completely specify the output enable behavior (when the it depends on a Level-2 or Level-3 line that is not completely specified) but unlike Selection Values some strings will completely specify the output enable behavior (such as io_0 or input). If a list of strings contradicts somewhat, the rules in the previous section will still hold and thus the rightmost strings will determine the constant value stored. While this is still not recommended for final code, the functionality is included since it might be useful for debugging. However, use of this functionality should be done even more carefully with Enable Values than with Selection values and some other lists, since the effects might not always be clear to the programmer.

Just as with Selection Values, the sequence of strings can be an empty sequence, resulting in a completely unspecified Enable Value:

```
(def-constant en-any (en))
```

Kind of Constant	Keyword	Related Hardware	Default
Number or DC	N/A	Any	N/A
Constant Value	constant	BFU Ports, L3 Controllers	N/A
Static Source	static	BFU Ports, L3 Controllers	N/A
Dynamic Source	dynamic	BFU Ports	0x1DC
ALU Value	aluval	BFU's ALU Port	(Mul) (Ctx 0) (We 0)
Memory Value	memval	BFU's MEM Port	Sing-Port AluA&B&Memdat-In CfgR&W-Dis
Select Value	sel	IO Port Select Words	DC
Enable Value	en	IO Port Enable Words	DC

Table 4.5: Quick Summary of Types of Constants

Summary

One of the most important things to remember is that although a constant that has been defined will likely have an integer value associated with it that can be retrieved from it, a programmer should not make use of that value unless he is using it for the purpose which is paired with the means he used to create it or he is sure that he knows what he is doing. The danger is that the number has been changed in order to indicate information about it to the system which will use it, but this information might mean something different to another system that might try to use it. The power is there to do some nasty hacks, and that power should be avoided or at most used sparingly and carefully.

Since there are so many different kinds of constants, table 4.5 is provided as a reference to them. For complete descriptions of their associated syntactic structures and their semantic meanings, see the parts of this section that describe them in detail and the parts of the later sections that describe the constructs that use constants defined in the appropriate manner.

Although most constructs will not be nearly as involved as constants, each will have a table such as table 4.6 which will summarize the basic qualities of that construct. These tables are all combined in one large table that should be a useful reference about the various MDL+ constructs while programming in MDL+. That

Type	Constant
Constructor	def-constant
Keyword	Several
Default	None
Inheritance	No

Table 4.6: Quick Summary of Constants

table (table 4.31) is at the end of this section.

Constants can not inherit from each other. Table 4.6 mentions this fact for completeness, as there is not likely ever a time when it would be helpful for one constant to inherit from another, instead of simply copying the "parent" constant to the "child".

4.2.2 Bitvecs

Now that we have defined the Constant construct which represents any single number, it makes sense to have a construct for a list of such numbers. That is what the Bitvec construct is. Note that the name Bitvec is somewhat misleading, since these are usually representing an array of byte (8-bit) quantities and would thus better be referred to as Bytevectors. Of course, any long list of bits (regardless of what word-length they are grouped into) is still a bitvector, and this is the name that MDL used to refer to such items, so MDL+ uses it as well.

Bitvecs are useful in two places: Or-planes and Memory configurations. The Or-plane requires a bit vector to configure it, but the Or-plane was removed from MATRIX prior to the chip fabrication, and was not used all that often beforehand anyway. The other use is for the memory in each BFU, which consists of 256 one-byte (8 bit) quantities.

Bitvecs use the keyword **bitvec** and have the basic structure **(bitvec Value*)** where each of the list of arguments Value can take on many different forms, basically corresponding to the various ways to define constants. Each one of those forms will correspond to a constant value (integer number or Don't Care). The options and examples of their use are listed in table 4.7.

Description	Example	Constant
Don't Care	DC	DC
Number	8	8
ID of constant	C	Value of C
Constant Value	(constant 8)	8
Static Source	(static l1_n1)	769
Dynamic Source	(dynamic)	476
ALU Value	((Mul) (ctx 1))	64
Memory Value	((port double))	32

Table 4.7: Values in a Bitvec

```
(def-constant c 9)
(def-bitvec b
  (bitvec DC 8 C (constant 8) (static l1_n1) (dynamic)
  ((mul) (ctx 1)) ((port double)))
  )
```

Figure 4-3: Sample Bitvec Definition

Thus, a Bitvec could be defined as shown in figure 4-3. Note that while this is a legitimate Bitvec, it would not be a good definition of the memory cells of a BFU, because it contains values that are not 8-bit quantities. From table 4.7 it is clear that one of the values is 769 and another one of the values is 476, both larger than 256. This is because they were defined with ways intended for defining BFU port configurations, which are 10-bit values. (8 bits plus 2 bits for determining whether it is Constant Value, Static Source, or Dynamic Source.)

Bitvec constructs are summarized in table 4.8. The table says that the default value of a Bitvec is DC because defining a Bitvec to be DC yields the same thing as defining a Bitvec with its basic structure containing an empty list of Values. Bitvec constructs are identical to Memory constructs. (see later part of this section on Memory constructs.)

Type	Bitvec
Constructor	def-bitvec
Keyword	bitvec
Default	DC
Inheritance	No

Table 4.8: Quick Summary of Bitvecs

4.2.3 Sub-BFU Constructs

BFUs are very large and complex, and thus require a lot of configuration. To this end, there are many constructs in MDL+ which specify a subset of the configuration of a BFU. An MDL+ programmer can always specify these elements of functionality directly in a BFU definition, but when defined separately once, they can be combined in unique ways to form several similar BFUs.

Although there is not much known about whether these will turn out to be a good idea, since they have never been provided prior to MDL+, it is felt that they will also sometimes provide a nice amount of abstraction when defining BFUs. It has been pointed out that other times they will only serve to make an MDL+ source file longer and potentially more confusing to read, and not assist much in reducing the amount of complexity via abstraction. It is thus recommended that the MDL+ programmer figure out when separate specification of these constructs is aiding and impeding code readability, and then either use them or not accordingly. The MDL+ compiler outputs the exact same code regardless of whether pieces of BFUs are defined separately and then incorporated into the BFU definitions or first defined as part of the BFU definitions.

Whichever side the reader may take in this debate, the pieces of the BFU's configuration will be described with their individual constructs, and then the means of incorporating the already-defined pieces and of defining them directly in the BFU definition will both be explained in the section about the BFU construct. Since the guts of the structure for each construct will only be explained in the sections for the various constructs, even though these structures can be used directly in the BFU

construct, all of the sections about sub-BFU constructs will be important reading.

4.2.4 BFU Networks

This is the first of the sub-BFU constructs. The purpose of a BFU Network is to specify the crossbar between the BFU's network-capable ports (N1,N2,FP1,FP2) and the network lines it might output onto the higher-level-networks (2 L2 lines, 8 L3 lines).

The keyword for BFU Networks is **network**. The structure for a BFU Network that is not being defined to be merely a copy of another BFU Network or DC, is **(network (Line Port)*)**. The asterisk indicates Kleene closure, or that there is a sequence of zero or more **(Line Port)** arguments to the BFU Network construct. Each Line argument needs to be one of the Level-2 or Level-3 lines that a BFU can drive (l2_d1, l2_d2, l3_v1, l3_v2, ... , l3_h4) and each Port argument needs to be either one of the BFU's network ports (N1, N2) or one of the BFU's floating ports which can be used as network ports (FP1, FP2). The Level-2 lines are referred to in a different way than usual, with "d1" and "d2", because while 8 Level-2 lines cross over each BFU (2 in each direction), only 2 emanate from each BFU. If the BFU has horizontal L2 lines, "d1" points south or down and "d2" points north or up. As mentioned in chapter 3, BFUs are viewed on a checkerboard-grid to determine whether they drive horizontal or vertical Level-2 lines.

After some time programming for MATRIX programmers are expected to prefer using "d1" and "d2" to actual directions, as it was with MDL, but people first starting to program in MDL+ will likely find the "d" notation confusing. Table 4.9 should provide these people all of the assistance they need in determining what "d1" and "d2" mean for each BFU, as well as which BFUs drive horizontal Level-2 lines, and which BFUs drive vertical Level-2 lines.

Use of table 4.9 is very simple. Add up the X-coordinate and the Y-coordinate of the BFU in question to determine which half of the table to peruse. For example, any BFU on the middle diagonal (X=Y) will fall in the top half of the table. Notice that the convention of West and South being "lesser" and East and North being "greater"

X+Y	L2 Driven	l2_d1	l2_d2
Even	Horizontal	West/Left	East/Right
Odd	Vertical	South/Down	North/Up

Table 4.9: Meaning of l2_d1 and l2_d2

Type	BFU Network
Constructor	def-bfu-network
Keyword	network
Default	DC
Inheritance	No

Table 4.10: Quick Summary of BFU Networks

is preserved here.

If it does not matter at all which port a line will be driven from, because for example, the line is not going to be driven at all, then it can be omitted from the definition of the BFU Network. If all arguments after the **network** keyword are omitted, the result will be the same kind of BFU Network that would have resulted from simply defining a BFU Network to be DC, thus DC is listed as the default in table 4.10.

4.2.5 BFU Configs

This is the sub-BFU construct which specifies the BFU configuration bits that deal with higher-level concepts. These concepts include, for example, whether a BFU should think of itself as the most or least significant block of a wider than one-BFU operation. At the lower-level of actual configuration writes to a MATRIX chip these bits are also associated with each other. If an MDL+ programmer does not agree that they deal with higher-level concepts, it might be useful to think of a BFU Config as being the sub-BFU construct that deals with any remaining BFU configuration information that is not included in any of the other sub-BFU constructs. That is, as the miscellaneous sub-BFU construct.

Flag	MATRIX	Meaning
ignorecarry	No	Unclear
carrypipeline	Yes	Register incoming carry prior to use
madd1dyn	No	Use FP1 to select source for first add in MULA/A
madd2dyn	No	Use FP2 to select source for second add in MULAA
tsenable	Yes	Time-Switching Enabled for this BFU.
msb	Yes	This BFU is most significant byte in multi-word op
lsb	Yes	This BFU is least significant byte in multi-word op
left:Source	Yes	Source is next most significant byte, for left carries
right:Source	Yes	Source is next least significant byte, for right carries

Table 4.11: Flags allowed in BFU Configs

The constructor for BFU Configs is **def-bfu-config** and the basic structure is very simple: **(config Flag* ConTscycle)** where the keyword is **config**, the first argument is a list of flags, and the second argument sets the Time-Switching value for the Write-Enable of the BFU's memory.

Each **Flag** argument simply sets one of those high-level concepts to be true for the appropriate BFU. The possible flags and their meanings are explained in Table 4.11.

The Ignorecarry flag is included in MDL+ because it was in MDL, but it is unclear what the meaning was intended to be. It might mean that the carry is always zero. If that is the desired functionality, then the MDL+ programmer will have to implement it on his own by making all shift operations insert 0 and specifying LSB for the sake of the arithmetic operations.

The Carrypipeline flag is useful both for operations that the programmer needs, because of the nature of the program, to occur over multiple cycles (such as a single BFU computing a 16-bit addition) and for operations that are too wide for the carry-chain to complete all BFU operations correctly within the clock period. (Extremely long additions can not be completed in one cycle.)

The Madd1dyn and Madd2dyn options have been removed from MATRIX, essentially choosing to always assert them, hardwired. (This saves area and complexity in the hardware design.) Non-asserted Madd1dyn and Madd2dyn would free up the

FP1 and FP2 Ports during MULA and MULAA operations, instead always using the default sources of L1_N1 and L1_NE, respectively. This implies that with the current design of MATRIX, and MDL+, setting a BFU's ALU to a MULA or MULAA operation precludes that BFU from using its FP1 or FP2 ports for other activities, such as dynamic sourcing the A and B ports, or rebroadcasting certain Level-2 lines in different directions.

The Tsenable flag means that all time-switched registers will only take new values on mini-cycles that they are set to register on. The default is to turn time-switching off. Time-Switching is a very powerful tool, especially for routing hard-to-route situations. However, it is recommended to leave Time-Switching off, for the current MATRIX chip, since this is the one major facet of MATRIX that was not tested prior to fabrication, and is never necessary for simple designs.

The MSB and LSB flags are for BFUs that are involved in multi-word operations consisting of chains of BFUs. The most and least significant bytes should be configured as MSB and LSB, respectively, so that they will act properly, such as the LSB knowing that on an add it ignores its carry-in, and uses a zero instead.

The Left:Source and Right:Source flags are the only flags that contain arguments, each having the argument Source. These flags, like MSB and LSB, are also flags for BFUs that are involved in multi-word operations consisting of chains of BFUs. A BFU's left source is used to determine its left carry-in and its right source is used to determine its right carry-in. The Source argument can take on eight different values, all strings, as listed and explained in Table 4.12.

The option of the Source argument being Control, would make the carry-in to a BFU be the result of its Compare/Reduce II, in the current version of MATRIX. But, before the OR-plane was eliminated from MATRIX to cut down the area of the chip, this was the general Control Bit, which was chosen by the CtrlMux (see BFU Control construct) to be either the result of Compare/Reduce II or the first bit of the result from the OR-plane (MSB).

The final argument to BFU Configs is the ConTscycle, which is an optional argument. It has keyword **tscycle** and if given must have the structure **(tscycle We**

Source	Meaning
North	One slot up
East	One slot right
South	One slot down
West	One slot left
Local	Take carry from local carry out
Control	Control bit from Compare/Reduce II as carry in
Zero	Always constant 0
One	Always constant 1

Table 4.12: Sources for Left and Right Flags

Type	BFU Config
Constructor	def-bfu-config
Keyword	config
Default	DC
Inheritance	No

Table 4.13: Quick Summary of BFU Configs

Madd1 Madd2) where any of the three arguments may be omitted.

The We argument has keyword **WE** and structure **(WE NUM)** where the NUM argument can be any number. This number is the value of the current mini-cycle which the BFU's memory's write-enable will latch on. The Madd1 and Madd2 arguments use similar structures with keywords **Madd1** and **Madd2**, but do not correspond to anything in MATRIX. It is not even completely clear what they might be considered to have corresponded to in the past. Thus, you should always omit them. They are included here, and in the grammar of MDL+, for completeness since they were included in MDL.

The list of flags in a BFU Config works as most lists do in MDL+: The later arguments override the earlier arguments. A brief review of BFU Configs is available in Table 4.13. It lists a default of DC because omitting all flags and omitting the ConTscycle argument will provide a BFU Config identical to one defined merely to be DC.

Keyword	MATRIX	Explanation	Legal Values
Inputsel	Yes	Mux before C/R II	FP1, FP2
Ctrlmux	No	Mux determining Control Bit	Reduce, Or
Or	No	Or-plane	An ID or Bitvec instance
ReduceI	Yes	Compare/Reduce I	one or two 9-bit rednums
ReduceII	Yes	Compare/Reduce II	21-bit rednum and TS
Crselect	No	Select 4 C/R I for Or-plane	0 to 4 CR-Dirs

Table 4.14: Control argument keywords and associated hardware

4.2.6 BFU Controls

One of the biggest problems with MATRIX is its lack of a good control system. As opposed to its straightforward and natural ALU, memory systems, ports, and networks, the control system is completely unnatural and unintuitive as well as not very powerful. This sub-BFU construct defines that control system.

The keyword for the main BFU Control structure is **control**, and the structure is simply **(control Piece*)** where each possible argument Piece defines one element of the control for that BFU.

The mapping from the possible arguments in the control structure to the control hardware of a BFU is very straightforward. Figure 4-4 shows the basic control logic in a MATRIX BFU, and it includes 6 configurable pieces: Two MUXs, two Compare/Reduce boxes, the OR-plane, and which four of thirteen Neighborhood Compare/Reduce values are selected for the OR-plane input. These six pieces each have their own structure which can configure them in an argument Piece. The keywords for these six structures and which pieces of hardware they can control are listed in Table 4.14. The MATRIX column of the Table lists whether this hardware exists in the actual MATRIX chip, after the elimination of the OR-plane. The Legal Values column explains which arguments are allowed in the structure **(Keyword Legal-Values)** for that keyword.

The TS argument for the ReduceII keyword is optional, has structure **(Tscycle NUM)** and specifies the time-switching mini-cycle on which the Compare/Reduce II output value is supposed to latch while the BFU is in time-switching mode. The

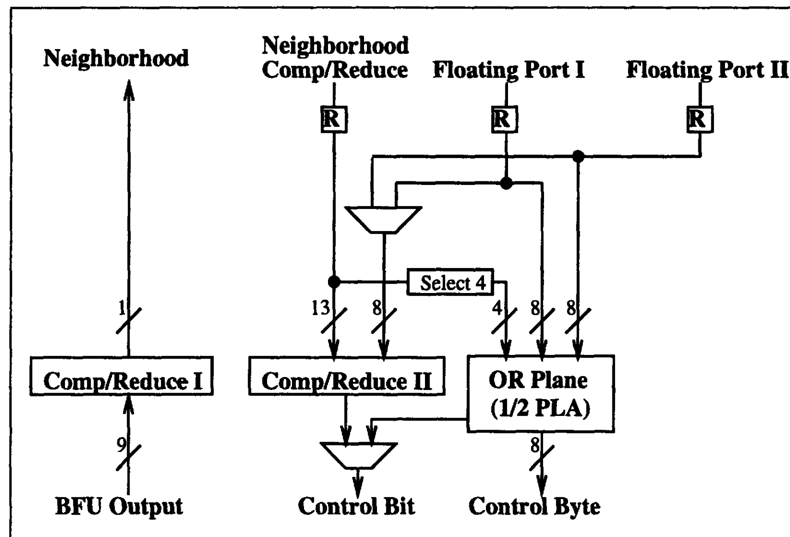

Figure 4-4: BFU Control Logic

Crselect keyword has a structure that takes a list of zero to four distinct **CR-Dirs** as arguments. The legal **CR-Dir** arguments are: n1, n2, ne, e1, e2, se, s1, s2, sw, w1, w2, nw, local, zero. Local refers to the local BFU C/R I value, and zero refers to constant value "0" instead of any C/R, since these were both options before the Or-plane was removed, even though they are not present on the diagram in figure 4-4.

MDL+ will not currently accept an Or-plane if you include it in your file, but it will accept any of the other information that is only needed when Or-planes are around, in case you wish to specify designs which will show how useful Or-planes are in an attempt to show that they should be included on the next chip. Another known bug at this time is that Compare/Reduce II will not accept the optional TS argument.

A useful abstraction in MDL+ is that the ReduceII argument can, but need not, be specified with a 21-bit rednumber value. Instead, it can be defined with a list of arguments that each describe some subset (not necessarily strict) of the 21-bit reduction number. Each argument in the list may be in any of the forms listed in table 4.15. This improvement to MDL was initially suggested in [Mat96b].

There are four times in table 4.15 where there is an argument **Bit-Desc** that must

Argument Desc	Explanation
21-bit compare	Actually specify all bits, e.g. 0rFFFXXX000111FFFXXX000
(Bit-Desc)	Forces that bit to 1
(not Bit-Desc)	Forces that bit to 0
(either Bit-Desc)	Forces that bit to X
(fail Bit-Desc)	Forces that bit to F
fail	C/R II should always Fail/return 0
accept	C/R II should always Accept/return 1

Table 4.15: ReduceII Arguments

Bit-Desc	Explanation
FPn, $0 \leq n \leq 7$	The nth Floating Port bit (FP1 or FP2)
Bit$n$, $0 \leq n \leq 20$	The nth C/R II bit
Local	C/R II bit corresponding to Local C/R I
n1,n2,ne,etc (12 total)	C/R II bit corresponding to that BFU

Table 4.16: ReduceII Bit-Desc Legal Arguments

be given. That argument can always be chosen to be any of the options given in table 4.16. There are two basic kinds of advantages gained by using this system over just using 21-bit rednumbers. First, it can shorten source code and make the code more readable. Second, it can abstract away more details of MATRIX, such as which bit encodes the reaction to which C/R I bit.

For example, consider the code in figure 4-5. The first two definitions are at the same abstraction layer, but the second uses the new system and is more readable. Someone reading the MDL+ source file does not need to count the "X"s and can thus not miscount, just as the programmer did not need to count the "X"s and thus could not have miscounted. It's also easy to read the line across as "The Reduce II accepts any situation as long as the twelfth bit is asserted."

The third definition also uses the abstraction capability, since the programmer probably knew he cared about the C/R I bit coming from the north-west neighbor and then had to look that up on page 43 (appendix D, section 2) of the MATRIX Micro-Architecture Specification [Mir95a] in order to determine that north-west cor-

```
(def-bfu-control old
  (control
   (ReduceII 0rXXXXXXXXXXX1XXXXXXXX)
   ))
(def-bfu-control new
  (control
   (ReduceII accept (bit12))
   ))
(def-bfu-control abst
  (control
   (ReduceII accept (nw))
   ))
(def-bfu-control best
  (control
   (ReduceII (nw))
   ))
```

Figure 4-5: Four Ways to define a simple C/R II value

responded to bit 12. This way (with the **abst** definition) the programmer does not need to consult [Mir95a] and can not mistakenly copy down the wrong bit number. Additionally, a person reading the MDL+ source code can not mistakenly look up the bit number in [Mir95a] and can simply read the line across as "The ReduceII accepts any situation as long as the North-West neighbor's Compare/Reduce I value is set."

Finally, the fourth definition is the best one. We consider it best because it is shortest and omits any unnecessary details. However, if unsure about which of the final two definitions is best, then the earlier one should be used since it is more clearly correct. The fourth definition (the **best** definition) relies on the fact that the default value of each ReduceII is **accept**. Thus, putting an **accept** at the beginning of a ReduceII argument list never actually does anything. It is a natural default to assume that nothing is cared about until those particular things are specified.

The BFU Control construct is summarized in table 4.17. The default listed is DC because a BFU Control defined with an empty argument list is identical to one that is defined to be DC.

Type	BFU Control
Constructor	def-bfu-control
Keyword	control
Default	DC
Inheritance	No

Table 4.17: Quick Summary of BFU Controls

4.2.7 BFU Powers

Even with MDL, the BFU configuration dealing with power could be optionally omitted. With MDL+ this section is even less needed than it was with MDL. That is because of MDL+'s Automatic Driving phase, which will be discussed near the end of this chapter. Basically, BFU Powers can be set to configure a MATRIX chip to drive or not drive any Level-1 or Level-2 line. All lines which are not so configured are somewhat like Don't Cares, they might end up driven, not driven, or in some other strange and possibly meaningless state.

For now, assume that every line which should be driven should specify enabled, and every line which should not be driven should specify disabled. When you get to the section about the Driver, there will be more information on how to best use BFU Powers.

The keyword for BFU Powers is **power** and the basic structure is **(power Pair*)** where any of the list of arguments Pair has the structure **(Line Status)**. The Status argument is simply one of two strings: **Disable** or **Enable**. The Line argument is simply one of twelve Level-1 lines (l1n1, l1n2, l1ne, etc.) or one of two level two lines (l2d1, l2d2). Note that the level-2 lines are marked "d" for direction, and their meaning is once again that given in table 4.9. Also note that these lines have no underscores (_) in their definitions, in an attempt to be consistent with MDL which was inconsistent with itself.

Since BFU Powers with an empty list of arguments end up leaving all lines neither driven nor undriven, like a BFU Power which is defined to be DC, the default behavior listed in summary table 4.18 is that of DC.

Type	BFU Power
Constructor	def-bfu-power
Keyword	power
Default	DC
Inheritance	No

Table 4.18: Quick Summary of BFU Powers

4.2.8 Ports

Just as BFU Ports (see the next construct) are sub-constructs of BFUs, so Ports are sub-constructs of BFU Ports. A piece of the BFU definition is that of a single one of its ports, and that is exactly what a Port defines. Since this is such a small amount of configuration, the Port is a construct with a very simple structure. Though simple, it is a very useful definition when building up our modular definition of BFUs.

The keyword for a Port is **port**, and its simple basic structure is **(port Value Value Tscycle)** where the first Value is the port's configuration in the first local context and the second Value is the port's configuration in the second local context. All three of the arguments to the Port structure are optional, which leads to the question of how to interpret only one Value argument. When provided only one Value argument, the compiler assumes that it is intended for the first local context (local context 0). Thus, for example, if the second local context needed the port's configuration set to 8, but the first local context did not matter at all, the only way to indicate that would be as:

```
(def-port Sec (port dc 8))
```

Meanwhile, if only the first local context was relevant and needed to be set to 8, either of these lines of code could be used:

```
(def-port Fir1 (port 8 dc))
(def-port Fir2 (port 8))
```

It should be clear by now that a Value argument can be a number or a Don't Care, but there are also many other possibilities. They are basically the core structures

Description	Example	Exception
Don't Care	DC	
Number	8	
ID of constant	C	
Constant Value	(constant 8)	
Static Source	(static l1_n1)	(static B)
Dynamic Source	(dynamic)	
ALU Value	((Mul) (ctx 1))	
Memory Value	((port double))	

Table 4.19: Ways to define a Value of a Port

of several kinds of constants. Since numbers and DCs are also core structures of constants, these are not exceptions. Table 4.19 gives all possible means of defining these Value arguments.

Note the one exception: That in addition to being able to use static structures the way that they can be used to define constants, they can also be used here with the name of a BFU as the only argument to a static structure. That is, in the example from the table, **B** must already have been defined to be a BFU. This option represents another level of indirection, saying that the static source is to come from a certain other high-level BFU wherever it may be placed on the grid (at l1_n1, l1_e2, l3_h3, or anywhere else). When using this option, it will later be replaced by the actual relationship between the BFUs during the Automatic Routing Phase which will be explained in the next Chapter.

All of the other options listed in table 4.19 are not explained here because those structures were already explained in the earlier part of this section that explained the Constant construct. However, it is important that some of the structures here maintain their keyword as necessary, and some do not have their keyword (specifically, ALU Values and Memory Values). Since there are as many as nine different ways to define each context of a Port and they are not actually explained at this point, we provide figure 4-6 with MDL+ source code that defines nine Ports, using a different structure to define each of them.

The third and final optional argument to the basic Port structure (after the two

```
(def-constant c 8)
(def-bfu b )

(def-port p1 (port dc))
(def-port p2 (port 8))
(def-port p3 (port C))
(def-port p4 (port (constant 8)))
(def-port p5 (port (static l1_n1)))
(def-port p6 (port (static B)))
(def-port p7 (port (dynamic)))
(def-port p8 (port ((Mul) (ctx 1))))
(def-port p9 (port ((port double))))
```

Figure 4-6: Nine ways to define a Port

Value arguments) is the Tscycle argument. This argument deals with the mini-cycle on which the value coming into this port is supposed to be latched when the BFU is in time-switching mode. For ports which do not have time-switching capability this argument should be omitted.

The structure of the Tscycle argument is **(TScycle Value)** where this Value is an argument almost like the two Value arguments at the beginning of the main Port structure. Normally this Value argument should just be a simple integer number or the name of a constant variable that has been defined to be an integer number. But, it is allowed to be almost anything that the other Value arguments could be. The one exception is the case that was listed in the exception column of table 4.19. That is, static structures must be used only as they can be used in constant definition. It would not make any sense to use the name of a BFU in this case, especially if that BFU were not even placed yet (see the next chapter's discussion of the Automatic Placement Phase). Table 4.20 shows the possibilities that do exist for this Value argument.

Ports are summarized in table 4.21. The default listed is DC because omitting all three arguments will produce the same Port as declaring it to be DC.

Description	Example
Don't Care	DC
Number	8
ID of constant	C
Constant Value	(constant 8)
Static Source	(static l1_n1)
Dynamic Source	(dynamic)
ALU Value	((Mul) (ctx 1))
Memory Value	((port double))

Table 4.20: Ways to define a Constant Number, for TScycle

Type	Port
Constructor	def-port
Keyword	port
Default	DC
Inheritance	No

Table 4.21: Quick Summary of Ports

4.2.9 BFU Ports

Now that the Port construct is defined, it is time to define the BFU Ports construct. Whereas the Port construct explained how any of the eight ports of a BFU are configured, a BFU Ports construct explains how all eight ports of a BFU are configured. To that end, it makes sense that the basic structure of a BFU Ports is basically just a list of which Port constructs get mapped to which of the BFU's ports.

The constructor for BFU Ports is **def-bfu-ports** and the keyword is **ports**. Be aware that this is only one letter different from the Port construct keyword of **port**. The basic structure for the BFU Ports construct is **(ports Pair*)**. In this case any of the list of Pair arguments is meant to define one of the BFU's eight ports to be a specific port configuration. As such, a Pair argument must fit the structure **(Name Value)** where the Name argument must be a string that represents one of the eight BFU ports, and the Value argument represents a port configuration.

The valid strings that can be given for the Name argument are listed in table

Name	Description
alu	F_a Port, for ALU function
mem	F_m Port, for MEM function
aport	A port
bport	B Port
n1port	First Network Port, N1
n2port	Second Network Port, N2
fp1port	First Floating Port, FP1
fp2port	Second Floating Port, FP2

Table 4.22: Names for BFU Ports

4.22. The Value argument is optional and may be DC (for the default all-DC port configuration), the name of a previously defined Port, or the structure used as the basic structure of a Port construct. If using the basic structure from a Port construct, the outer wrapping with the keyword **port** must be omitted.

These various ways of defining pieces of a BFU Ports are present in the MDL+ source code of figure 4-7.[4] In the sample code, the aport has both local contexts defined in one line, but the ALU has both local contexts defined in different lines. The two A port lines could separated but the two ALU lines could not be united because one of them is defining the ALU configuration to be a port which includes both local contexts even though the programmer knows that one of them is about to get overwritten.

Finally, note that the N1 port is defined to be DC, but this line of the code accomplishes nothing because that is the default anyway. The ability to make DC definitions like this is provided for two reasons: So that MDL files are better able to compile with MDL+ (since MDL required all ports to be specified, and had a non-working DC option) and for debugging purposes (replacing a definition with DC can effectively comment-out a line when it is easier to do it this way).

Table 4.23 is provided as a review of BFU Ports. If no ports are specified at all inside a basic structure for a BFU Ports definition, the result will be identical to the

[4]This code will actually not compile due to a known bug that prevents defining a piece of a BFU Ports to be the name of an already defined port.

```
(def-port p1 (port 8))
(def-port p2 (port ((mul))))
(def-bfu b )
(def-bfu-ports bp
  (ports
   (alu p2) ;; start off with whatever p2 is defining lcl ctx 0 of ALU
   (mem ((port double)))
   (aport (static l1_n1) (static l1_n2))
   (bport (static b))
   (alu dc ((add) (we 1))) ;; change the ALU lcl ctx 1
   (n1port dc)
   (fp2port 9)
   ))
```

Figure 4-7: Sample BFU Ports definition

Type	BFU Ports
Constructor	def-bfu-ports
Keyword	ports
Default	DC
Inheritance	No

Table 4.23: Quick Summary of BFU Ports

case where the BFU Ports was defined to be DC. Thus, the listed default is DC.

4.2.10 Memories

The last of the sub-BFU constructs, Memories describe the initial contents of the memory unit of a BFU. The Bitvec construct already seems suited to this task. A BFU's starting state consists of initial values for 256 8-bit cells, any of which the programmer might not care about. Thus a Bitvec of up to 256 values, none of whose values are greater than 256, is an ideal way to specify this configuration.

Memories are merely another means of saying the same thing as a Bitvec. Instead of **def-bitvec** the constructor is **def-mem**, and instead of **bitvec** the keyword is **cells** because it is defining memory cells. Besides that, there are no differences. In fact, internal to the compiler, these two constructs are currently viewed as the

Type	Memory
Constructor	def-mem
Keyword	cells
Default	DC
Inheritance	No

Table 4.24: Quick Summary of Memories

same construct, and it will even accept a mix-and-matching of the constructors and keywords. Any place that an ID for a Bitvec or Memory is required, the other will be accepted.

It is not surprising that the summary table for Memories (table 4.24) mirrors the table for Bitvecs (table 4.8).

4.2.11 BFUs

After understanding all six sub-BFU constructs, a programmer should not have a hard time learning about the BFU construct. Basically, the BFU construct is designed to contain all of the configuration information possible for any single BFU. This is divided into the six non-overlapping subsets that are each part of their own sub-BFU construct.

The BFU construct is the first construct mentioned in this thesis to not have a keyword of its own. Instead, it merely uses keywords of the sub-BFU constructs, when necessary. The basic structure of a BFU construct is simply **Piece***, which is to say a list of Piece arguments, without any surrounding parenthesis or keyword. It is because this list can be empty that several source-code figures above have defined a BFU with a line such as:

```
(def-bfu b )
```

The empty space left after the ID "b" is simply one convention, used to suggest that this is the definition of a BFU, that just happens to not have any parts of it specified. The same convention calls for no space to be left when the same BFU

will be defined later in the file. In that case, this line is merely a declaration for the purpose of telling the compiler that "b" is a BFU, and not actually meant to define that BFU at all. The compiler does not care which convention is used, and where blank spaces are left, but we recommend this as a good MDL+ programming convention.

Of course, interesting BFUs are defined with non-empty lists. Each Piece in the list can be an ID or the basic structure of any of the six sub-BFU constructs. If an ID, it must name either a previously defined BFU or a previously defined sub-BFU construct. If the basic structure of a sub-BFU construct, it should be complete, including the keyword.

The obvious change since MDL is that names of sub-BFU constructs can now be used, where MDL only provided the actual structures themselves. But that is not the only change. Due to the argument to the BFU structure being any list of Pieces, some pieces may be omitted, pieces may be given in any order, and pieces can even be given more than once. That is, the programmer may make use of MDL+'s list convention, to put things at the end of the list that are supposed to override things at the beginning of the list. This ability to override can be several layers deep.

First of all, we observe that BFU constructs have inheritance. That is, the first Piece in the list can be another BFU. Then, the new BFU being defined will have all of the qualities of the parent BFU, except the ones overridden by later arguments. And there are many more twists on this strategy. Several BFU names may be listed, essentially saying to the compiler to "take BFU A and use it's configuration, except for the details provided by BFU B, and I don't care what's in either A or B as soon as C has anything to say."

Furthermore, sub-BFU Pieces, either names or structures, can override each other. A BFU Ports with the ALU port and MEM port set can override a BFU Ports with the ALU port and A port set, leaving the old A port, changing the ALU port, and adding a new MEM port specification. But, it is important to remember that it is never quite as simple as that. Constructs will "inherit" from each other in this way that we call overriding instead of just "replace"ing each other, unless they are at the

```
(def-bfu-network net (network (l2_d1 n1)))
(def-bitvec bits (bitvec DC 8 DC 9 10))
(def-bfu bfu1 bits net
  (cells 1)
  )
(def-bfu bfu2 bfu1
  (ports
   (Aport (dynamic))
   )
  (ports
   (Aport dc (static bfu1))
   )
  (network
   (l3_h1 fp2)
   )
  (cells dc dc 2)
  )
```

Figure 4-8: Sample BFU Definition

lowest level of Constants. So, when we say that the new ALU port definition will override the old ALU port definition, that is only in the ways that it is defined to care about. For example, if the old ALU Port definition specified the port value for both local contexts, and the TScycle value, and the new ALU Port definition only specified the second local context value, then the final ALU Port value for the BFU will still retain the old value for the first local context and the TScycle value, but get the new value for the second local context. This can be summed up very simply as "If you don't care about what a value is, and someone else does, then their value will persist."

Figure 4-8 contains MDL+ source code for an illustrative example of BFU definition. All of the structures should be familiar from earlier parts of this section, only the combination and overriding of them should be new.

The first BFU in this code, **bfu1**, has its l2_d1 line driven by its first Network port (N1), and four of its first five memory slots are initialized (1, 8, DC, 9, 10). The second BFU inherits from the first one, and replaces none of its values. It does add the definition of the remaining uninitialized memory cell of the first five memory

Type	BFU
Constructor	def-bfu
Keyword	None
Default	DC
Inheritance	Yes

Table 4.25: Quick Summary of BFUs

cells, add a network definition, and add a single port definition. **bfu2** has its initial memory cells starting with the values (1,8,2,9,10) and it has its Aport configured to be in Dynamic Source more for the first local context and in Static Source mode during the second local context.

This code is slightly nonsensical since it makes no sense to define the Aport to be in dynamic source mode without defining the FP1 port configuration for the same local context. However, the MDL+ compiler will still allow this case, because it does describe a configuration of MATRIX, and this configuration might be a desirable one to load onto the chip during some phase of testing the chip, or as a parent to inherit from.

The behavior of the BFU construct is summarized in table 4.25. Recall that just because BFUs have no keyword does not mean that all BFU construct definitions will be devoid of keywords. Furthermore, the advanced MDL+ programmer should take special note of the fact that the inheritance of BFUs allows him to use inheritance-like overriding with many other constructs.

4.2.12 Layouts

Once all necessary BFUs are defined, the next step is to put them in the layout. The layout is just the 6x6 grid that the BFUs eventually need to get placed in. Being a higher-level construct like the BFU construct, a Layout contains a lot of configuration information, is very much just a collection of other constructs, and has more higher-level functionality like inheritance and the assistance of automatic compiler phases that will be discussed in the next chapter.

Description	Example	Constant
Number	4	4
ID of constant	C	$1 \leq C \leq 6$
Constant Value	(constant 3)	3
ALU Value	((Mulaa))	2
Memory Value	((cfgread enable))	1

Table 4.26: Ways to define a Constant Number for Layout Coordinates

Also like the high-level BFU construct, Layouts have no keyword, and the basic structure is just **Piece***, a list of Piece arguments. This means that a layout, like a BFU, can be defined with just an empty list. Unlike a BFU, there is no convention to leave a blank space in such a definition, since there is no reason other than defining an empty layout to use an empty list.

In the case of the Layout structure, a **Piece** represents something that is supposed to be on the chip in the layout; for now, either a BFU or another layout. And since it is going to be somewhere in the layout, it needs to be put in a specific location. Thus the structure for the Piece argument is **(Name Value Value)** where Name is the string that names a previously defined BFU or Layout, and each Value argument is almost any means of defining a constant. (See table 4.26 which is the same as table 4.7 and table 4.20 except that here all of the numbers must resolve to values between 1 and 6, inclusive.) The first Value corresponds to the X-coordinate on the grid, and the second Value to the Y-coordinate on the grid. Thus the BFU's location is said to be (X,Y) in (Column,Row) notation.

It is not recommended to use ALU values or Memory Values to specify coordinates in a layout. Also note that there will become more options for how to specify the Piece argument to a layout once we introduce the Automatic Placement phase. The coordinates will be allowed to be specified as DC, or omitted. There will also become more possibilities after the Automatic Grouping Phase is introduced, with the option to include other constructs besides BFUs and Layouts in a Layout. For now, all three arguments to the structure for each Piece are mandatory, and only BFUs or Layouts can be named by the first of the three.

```
Parse Error:
  You try to put a shifted layout into a layout but: One of the bfus
that you're shifting over is not dc and shifts to a location off of
the layout (x or y coord greater than six).
Last Line Read: 3
```

Figure 4-9: Sample Error Message for over-shifting a non-DC BFU

If a BFU is named, it is fairly obvious what the consequences will be: That BFU will be placed in the specified slot of the grid on the chip with this layout. It is less clear what placing a Layout on another Layout at slot (x, y) means, but it is not difficult to understand. It merely means that the old Layout will be translated so that its lower left BFU slot ((1,1) is the "least" slot) is over the new slot (x, y) and then all BFUs that have been placed in the old Layout will get placed at the location they are over, in the new layout. Essentially, the compiler translates all BFUs in the old layout by $(x - 1, y - 1)$ and then places them in the new Layout at those new coordinates.

This does in fact mean that many BFUs in the old Layout will be translated to a point on the infinite grid that is not in the 6x6 grid on the new Layout. It is required that any such BFUs that "fall over the edge of the world" are completely unspecified (equivalent to the default, or to having been defined to be DC). If one of these BFUs is partly specified, then the compiler will halt and output a compile-time error message such as the one in figure 4-9.

Note that this error message will come about even if the BFU being shifted off of the layout has been given a name and not specified at all besides that. This is because the name might be later used to route signals from that location (and this is in fact the only good reason to create such a BFU in the first place) in which case the BFU must be on the layout.

The capability of putting a layout into another layout can be used for several reasons. First, it could be used to get a stock-design that's been used before and stick it onto the chip that is currently being defined, perhaps as a module. Second, the programmer might be tiling some design onto the chip and this way he only needs

```
(def-bfu b
  (ports
   (alu ((passa)) ((passa))
   (aport 4 4)))
(def-layout row (b 1 1) (b 2 1) (b 3 1) (b 4 1) (b 5 1) (b 6 1))
(def-layout all (row 1 1) (row 1 2) (row 1 3) (row 1 4) (row 1 5) (row 1 6))
```

Figure 4-10: One BFU replicated 36 times by placing Layouts in Layouts

Type	Layout
Constructor	def-layout
Keyword	None
Default	DC
Inheritance	Yes

Table 4.27: Quick Summary of Layouts

to define it once and can then place the entire design several times.

For example, consider figure 4-10 in which a single stock BFU has been placed at every single location on the chip. (Every location is broadcasting the number 4 every round.) Instead of 36 Pieces in the Layout, only 12 were required.

The Layout construct is summarized in table 4.27. The only potentially confusing entry in this table is the one that lists Layouts as having inheritance. It does this because an inheritance scheme is present for Layouts that is as robust as that for BFUs by including a parent Layout in a child Layout at location (1,1). It is important to remember that this means that placing two BFUs in the same spot of a Layout grid will not simply replace the first one with the second one, but rather override the first one with the second one. If having the two BFUs end up in the same location was a mistake on the part of the programmer, this can lead to strange behavior that might be difficult to diagnose correctly, however the grammar is kept this way because it provides the potential for this very robust form of inheritance.

4.2.13 Connects

The only information about the interior of a chip that is not configured by a Layout is the decisions of which BFUs will drive which Level-3 lines. Although this greatly affects the interior of the chip, the arbitration is actually left to the VBFUs on the perimeter of the chip, and thus the configuration of a Connect is really perimeter configuration. That is why Connects are separate constructs from Layouts.

However, the programmer should be able to mostly stick with the abstraction that a Connect is part of the interior of the chip. That is, a Connect merely specifies for each Level-3 line, who gets to drive that line, and this is usually an interior-of-the-chip problem. It becomes connected to the perimeter when the answer is that the line will get driven by whoever the IOport says should be driving, or by the IOport itself.

The basic structure of a Connect construct is similar to that of BFUs and Layouts: There is no keyword, and the basic structure is **Pair*** where each of the arguments in the list, Pair, has the structure **(Line Value)**. The Line argument is merely a string that specifies one of the 48 Level-3 lines that a chip has. The Value argument can be omitted, can be DC, or can be a constant defined with either a Constant Value structure or a Static Source structure. Basically, it is intended that these Value arguments should all be defined with the Constant Value Level-3 structures if the line is always to be driven by the same source, and to be defined with the Static Source Level-3 structures if the line is to be driven by different BFUs based on the input of a certain line.

The possible Line arguments are all four-letter long strings. The first letter determines whether the line is over a Row (R, horizontal) or Column (C, vertical). The second character is a digit, one through six, that specifies which row (R) or column (C) the Level-3 line runs over. The third character is an underscore (_) to separate the row or column determination from the rank within the row or column. The final character is a digit from one through four, since each row and column on a MATRIX chip has 4 level-3 lines running over it. Thus, some allowed Line arguments would be R1_3, R3_1, R5_4, C6_1, and C2_2.

The structures for the Value argument were discussed in the earlier part of this

```
(def-connect c
  (r1_1 (constant bfu1))
  (r2_2 (constant bfu2))
  (c1_3 (constant bfu1))
  (c1_4 (constant bfu2))
  (c2_1 (static none))
  (c2_2 (constant none))
  )
```

Figure 4-11: A typical Connect

Type	Connect
Constructor	def-connect
Keyword	None
Default	DC
Inheritance	No

Table 4.28: Quick Summary of Connects

section that introduced constants, so we will not repeat that discussion here. But, the most common Value arguments to a Connect are all **Bfu**n for some n, $1 \leq n \leq 6$. These arguments represent saying that the Level-3 line will always be driven by the same BFU in that row or column, in particular the BFU that is the nth one in the row or column, counting from the lower end (south or down for columns, west or left for rows). Some examples of these typical Connect arguments are given in figure 4-11.

Note that while the final two definition Pairs in the figure should have the same operational effect on their lines, they ask for different configuration bits to be written into the MATRIX chip. Thus, the Connect **C** will actually have different values associated with its lines c2_1 and c2_2 (28 and 15 respectively).

Connects are summarized in table 4.28. Be sure to remember that when defining a piece of a Connect with a static structure, it must fit the definition of the static structure that is used for constants, not the one used for ports. In other words, it can not refer to another BFU by name.

4.2.14 IOports

With a Layout and a Connect a MATRIX chip's internal state is completely configured and specified. All that remains to be defined is how the chip will input and output data from and to the off-chip world. This takes place through 12 IOports that are on the perimeter of the chip, and the configuration information for each one can be set in MDL+ using its own IOport construct.

At each IOport, there is an 8 bit data bus which can either input or output data, and two I/O control bits which can each either input or output data. Thus, each of the three (one 8-bit bus, two bits) needs an Output Enable to tell it whether to be inputting or outputting. If in output mode, these same three things also need to know which line to take the output from, which it knows based on a Selection word. Remember that when in output mode, the outputted data will come back and appear on the input lines, after two cycles of delay (because the input lines are always reading the perimeter busses.)

The three words of output enables, and the three words of selection configuration, together form the six words of configuration possible at each IOport. With an IOport construct, any of these configuration words can be included or omitted. The IOport construct has no keyword, and has the simple basic structure of **Pair*** where each element of the list is an argument Pair whose structure is either **(SelField SelVal)** or **(EnField EnVal)**. The first option should be used for all Selection words being configured, and the second for any Enable words being configured.

We will discuss selection words first. The SelField argument can take on any of three strings as its value, to represent the three words it might be configuring: **datasel** (to configure the source of the output onto the 8-bit data bus), **bit1sel** (to configure the source of the bit that can be outputted at the first bit, or bit 0), and **bit2sel** (to configure the source of the bit that can be outputted at the second bit, or bit 1).

The **SelVal** argument can take on the same values as an argument to a constant definition's **(sel Arg)** structure. That is, referring back to the Select Values part of the part of this section to discuss Constants, SelVal can be either a NUM or a Words

Type	IOport
Constructor	def-ioport
Keyword	None
Default	DC
Inheritance	No

Table 4.29: Quick Summary of IOports

argument. The SelVal argument is the sole reason for Select Values to exist, and one legal SelVal is, of course, the name of a constant that was defined to be a Select Value.

Next, we discuss output enables and their **(EnField EnVal)** structure. The EnField argument can take on any of three strings as its value, to represent the three words it might be configuring: **dataen** (to configure the 8-bit data bus' output enable), **bit1en** (to configure the first bit output enable, or that for bit 0), and **bit2en** (to configure the second bit output enable, or that for bit 1).

The **EnVal** argument can take on the same values as an argument to a constant definition's **(en Arg)** structure. That is, referring back to the Enable Values part of the part of this section to discuss Constants, EnVal can be either a NUM or a Words argument. The EnVal argument is the sole reason for Enable Values to exist, and one legal EnVal is, of course, the name of a constant that was defined to be an Enable Value.

A summary of the IOport construct is in table 4.29. The toughest thing about IOports when programming in MDL+ is integrating them with the interior of the chip, and remembering that sometimes changes to the interior of the chip will then require modification of the IOports in the perimeter.

4.2.15 Chips

At this point, all configuration bits on a MATRIX chip can be set, but they all still need to be brought together and identified as a single chip. That is the job of the Chip construct — to unify a Layout, a Connect, and 12 IOports into one chip.

Similar to a BFU or a Layout, a Chip is a high-level construct that basically just groups functionality from the lower level constructs. As such, it contains a great deal of configuration information, and has a similar structure. A Chip construct has no keyword, can be defined with an empty basic structure, has full inheritance capability, and has a basic structure that is basically just a list of pieces. The basic structure is slightly more complicated than some of the previous ones:[5] **Piece*** where each of the list's elements is an argument Piece which can be passed either just a string that is the name of a piece of a chip or the structure **(Name Value)** which represents the placement of an IOport.

When using Piece as just a name it must be a name that is defined in the current environment to be a Layout, a Connect, or a Chip. Since this argument can be a parent Chip, we have inheritance. Since a parent chip can come anywhere in the list, we have what we have been referring to as full inheritance. Also relevant is that we can override Layouts or Connects on their own or with Chips.

The only problem with just using the string option for the Piece argument is that it leaves no way to specify IOports. To do that, use the **(Name Value)** structure. Name is a two-character string where the first character is the letter representing the side of the chip that the IOport should be on (N,W,E,S), and the second character is the digit describing which IOport it is on that side, starting from the "lesser" side of that side of the chip (0,1,2). A picture of a MATRIX chip, with all 12 IOports pictured and labeled by name, should make figuring out the name of an IOport much easier, and is provided in figure 4-12.

The Value argument needs to specify the configuration of the port which the Name argument has just placed. Thus, the Value argument is just a string which is the name of an already defined IOport construct.

Between the two means of defining the Piece argument, a Chip construct can specify all of the configuration for the interior of a MATRIX chip (Layout, Connect) as well as all of the configuration for that chip's communication with the off-chip world (IOport) and can even inherit from previous such definitions (Chip).

[5] The reason being that there was no Chip-IOports construct.

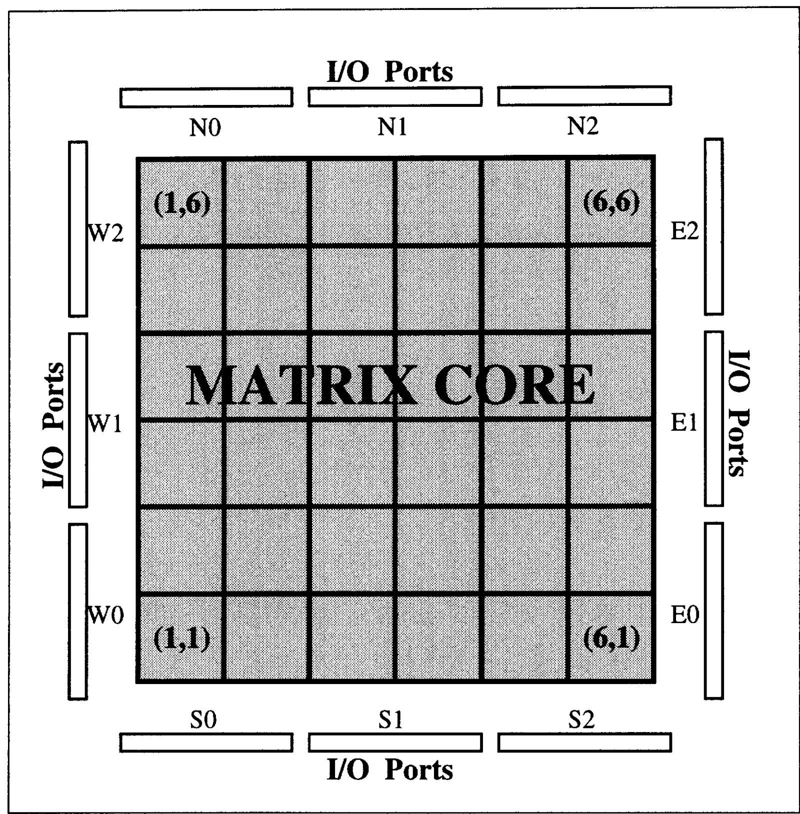

Figure 4-12: Block Diagram of a MATRIX Chip with IOport names

Type	Chip
Constructor	def-chip
Keyword	None
Default	DC
Inheritance	Yes

Table 4.30: Quick Summary of Chips

However, recall that the Chip basic structure only takes identifiers of previously defined pieces of chips, and not those definitions themselves. This keeps MDL+ source code much neater and more abstract, but can be easily forgotten. A summary of the Chip construct is available in table 4.30. The default is listed as DC because a Chip defined with an empty basic structure (empty list) is the same as one defined to be DC.

4.2.16 Summary

As a summary of the various construct types that MDL+ offers, and an easy reference for programming in MDL+, much of the information that has been given in this section for each construct is summarized in table 4.31. It contains the constructors and keywords for all MDL+ types, what each constructs' default value is for those which have default values, and whether they can inherit from other instances of the same type.

This section should be sufficient preparation for programming in MDL+. The remainder of this chapter will deal with the Automatic Driving Phase of the compiler and then an overview of the error messages and output modes of the compiler. Both of these remaining sections should serve to further facilitate understanding about the MDL+ compiler and make programming using the MDL+ tools much easier, but this section has already concluded the vast majority of the MDL+ language specification and explanation.

Type	Constructor	Keyword	Default	Inheritance
Constant	def-constant	Several	None	No
Bitvec	def-bitvec	bitvec	DC	No
BFU Network	def-bfu-network	network	DC	No
BFU Config	def-bfu-config	config	DC	No
BFU Control	def-bfu-control	control	DC	No
BFU Power	def-bfu-power	power	DC	No
Port	def-port	port	DC	No
BFU Ports	def-bfu-ports	ports	DC	No
Memory	def-mem	cells	DC	No
BFU	def-bfu	None	DC	Yes
Layout	def-layout	None	DC	Yes
Connect	def-connect	None	DC	No
IOport	def-ioport	None	DC	No
Chip	def-chip	None	DC	Yes

Table 4.31: MDL+ Constructs

4.3 Automatic Driving Phase

The principal addition to the MDL+1.0 compiler that makes the MDL+1.1 compiler is the automatic driving phase. All other improvements in the MDL+ grammar over the MDL grammar have been rolled into the description of the MDL+1.0 grammar in the previous section. While adding the driver to the compiler does add functionality useful to the programmer and thus does change the look of MDL+ programs, it does not actually change the grammar of MDL+ at all. Thus, while the MDL+1.1 compiler is upgraded from the MDL+1.0 version, the two languages are exactly the same.

Since there is no need to explain the changes in language syntax, this section quickly describes why the driver is needed, what the driver does, and finally how driving should now be specified in MDL+.

4.3.1 Motivation for Driving

MDL+ has always provided the ability to enable or disable driving any line, which in itself was an improvement over MDL. In MDL, level-3 lines could be specified much as they can be in MDL+, but level-1 and level-2 lines could only be explicitly turned

off. The MDL+ power section of BFU definition (or BFU Power construct) was a Power-Disable section, and all level-1 and level-2 lines were implicitly defaulted to being turned on.

When writing MDL code to be run on a simulation this was fine, since the programmer could omit the power-disable section (this was one of the few section that an MDL programmer had the option to omit), and be sure that the lines he needed were in fact being driven. The problem would arise when moving to running MDL-generated code on an actual fabricated piece of silicon, which has never happened. A MATRIX chip has so much interconnect on it, that actually driving all of the lines on the chip at once would cause it to overheat and be destroyed. Thus, with MDL the programmer would need to go through and specify all of the level-1, level-2, and level-3 lines that he was not using to be not driven (or at least most of them). To address this problem there was the assumption that the MDL compiler would be altered at a point in time before the fabricated chips were received to default all lines to not being driven.[6] This would of course solve the problem of accidentally blowing out the chip, but not save MDL programmers from having to explicitly specify power information for each BFU.

The MDL+ driver, on the other hand, should solve all of these problems. A similar concept to it was suggested at one point as an extension to MDL, but was never implemented. The basic idea is that the driver figures out which lines (level-1, level-2, and level-3) are read and which lines are not. It then drives the lines that are read and disables the driving of the lines that are not.

4.3.2 What the Driver Does

In MDL+ source code terms, keeping a line from being driven means specifying it disabled in a BFU Power construct for the appropriate BFU for level-1 or level-2 lines, and specifying the line to be driven by **none** in a Connect construct for level-3 lines. Driving a line means specifying it enabled in a BFU Power construct of the

[6]The people doing the circuit design and layout for MATRIX were quite adamant about this being the only acceptable situation.

appropriate BFU for level-1 or level-2 lines, and specifying someone to drive the line via a constant value or static source structure for level-3 lines.

Because the programmer might wish to override the wishes of the compiler, and to support the option of human-involvement at all levels of multi-level programming, the driver does not ever actually force a line to be driven or disabled. Instead, it tries to drive lines or tries to disable them. It does this by, in MDL+ source code terms again, using inheritance to cause the programmers specifications for each line to override its decisions on each line. This way, if the programmer forces a line to be driven or disabled, he will still get his way, but in the event that the programmer does not care then the driver will do the most intelligent thing it can think of.

The only question remaining is how the driver decides which lines are being read and thus requiring its attempt to have them be driven. This is not a simple question, and there is no easy way for the driver to determine exactly which lines are going to be read.

For any level-1 or level-2 line, if there is some BFU port that reads that line in static source mode then it is considered read and thus the driver attempts to have the source of the line drive it. One can imagine a driver that is more conservative than this and checks to make sure that the data is used. For example, imagine a BFU that has its ALU port in constant value mode with a value that does not write the memory (memory write enable is disabled). Furthermore, this BFU does not write any level-3 lines, and no other BFUs read from the level-1 or level-2 lines that come from this BFU. Clearly, the output of this BFU does not matter, and thus even if its A port is in static source mode from one of its neighbors, an intelligent driver might not require that line to be driven.

And there are many more examples, such as a BFU with its memory always set in single port mode, the memory data mux coming from the local feedback line, and the ALU port always a Pass A operation. Clearly, the B port of the BFU can never have any effect. So what if the B port is in a mode where it reads one or more network wires? Again this reading can be ignored, by a very advanced driver. It was deemed that the MDL+ driver need not be this advanced, but it is important to understand

that these cases exist and not count on them being driven, since the driver might be improved at some point. If an application wants a line to be driven, it should specify that properly — in a BFU Power or Connect construct — and not merely add an arbitrary read of the line.

In this first case, the driver was not made as conservative as possible because it would require a vast amount of complexity in the algorithm, or else merely be checking for a couple of cases which are very unlikely to happen anyway. Most importantly, the driver already turns off enough lines that we do not believe any reasonable application will overheat the chip beyond its tolerances.

The second way in which the driver could be far more intelligent is when it comes to ports of BFUs that are in dynamic source mode. This means that depending on the values of the associated floating ports, the dynamically sourced port might be reading any of the 30 input lines that it can choose from. The safest way to handle this, and the way in which the MDL+ driver does in fact handle it, is to attempt to drive these 30 lines. (Actually, the local line is always driven, and the same with the control byte, so it is really trying to drive 28 lines.)

This is the place in which the programmer can assist the driver the most, by explicitly disabling the never-used lines that flow into a dynamically-sourced port. The reason this job falls on the programmer is that it is a very hard, and sometimes impossible, job for the compiler to do. To figure out which of the lines it could disable, the compiler would need to determine what values could be flowing into the appropriate floating port. Fortunately, floating ports can not themselves be in dynamic source mode, and since it would not make any sense to put a floating port that was arbitrating a dynamic port into constant value mode (because you could just put the other port into static value mode with the same value and free up the floating port), thus we can assume that the floating port is in static source mode.

For simplicity, and without loss of generality, assume that it is the A port that is in dynamic source mode, and thus the FP1 port whose static source line we are analyzing. The driver could reasonably check to make sure that the FP1 port is in static source mode and then check its value and thus find the line that is determining

which line will source the A port. However, determining all of the possible values on this line might be extremely complicated. In the general case, this problem is provably harder than solving the halting problem[7], and thus the driver does not attempt to solve it.

Of course there are problems that are theoretically very difficult but in practice the common case is never one of the more difficult ones, (e.g. there are many programs for which it is easy to determine if they will halt) but we do not believe this to be the case with the value-on-a-line problem. In an intuitive sense, if it were easy to figure out exactly which values were going to be on the line, then it would not be as useful a line for determining the source of our A port. There will be some cases where dynamic source mode is being used to toggle between two lines, but we feel that more of the time it will be used to toggle between many lines, perhaps determined by some Instruction-Store memory. In a practical sense, figuring out all of the values a level-1 line can take on means figuring out all of the values that some BFU can calculate with its ALU or pass out of its memory. Figuring out the values on a level-2 line includes these possibilities but also includes any values that the BFU might have re-transmitted from its input (these are included even in situations where the A, B, Mem, and ALU ports are simple constant values, if any of the other four ports are not). Finally, the line might be a level-3 line and this can be the most difficult of all, because even figuring out who drove the unknown values on the level-3 line might require figuring out all of the possible values that are being passed from the static source that the control of that line was specified as. And all of this work would be required to rediscover something that the programmer already thought he knew when writing the code.

Now that we have discussed how the driver determines whether level-1 and level-2 lines are being read, we move on to level-3 lines. The driver takes the simple way out here as well. Basically, the MDL+ driver looks for each line at whether it has been specified that someone gets to drive it. If so, then the driver assumes the line is read

[7]Consider a machine that simulates a MATRIX chip, observing this wire. If it ever hits a specific value, the machine halts. Otherwise, it keeps running the simulation forever.

(the logic is that there would be no other reason for the programmer to specify who is driving it.) If not, then the driver assumes that the line is not read (the logic is that if it does not matter who is driving it then it can not matter whether it is being driven.) This is a sufficient analysis, and saves the chip a large amount of power again.

Of course, the driver could be far more intelligent in this case as well. Instead of merely looking at whether a level-3 line is driven, it could also look at whether the line is read by any BFUs and whether any BFUs thought that they were driving the level-3 line (specified which network or floating port would drive it). In this case the driver could observe that a given level-3 line is read, and is only being attempted to be driven by a single BFU, and it could then assign that BFU to drive the line in the Connect construct. For now, this seems like too much initiative being taken on the part of the compiler, and like a response to a situation that should not happen, and thus it has not been implemented. The most we have considered implementing is an algorithm that would consider these cases, and issue a warning at compile-time if something did not make sense. For example, if a level-3 line was read but no one was allowed to drive it, a warning would be issued, possibly halting compilation or possibly just alerting the user to the chance of an error in his code.

The final concern about the driver is that we set out when writing the MDL+ compiler to have it read and generate manageable and readable code, in part by minimizing code-size through a process of eliminating the unnecessary configuration data. The driver seems to counter this effort by putting in a piece of configuration for each and every line on the chip. After the driver runs, the code is overrun by long BFU Power specifications and long Connects. The verilog output can become mostly power information, hiding away the programmer-relevant lines of configuration writes.

About these concerns we make two notes: First, if the programmer wishes MDL+ code to be outputted, then he always has the option of getting it either with or without the driver's changes, essentially by asking for the input in MDL+1.0 or MDL+1.1. Second, if the code is to be run on an actual fabricated chip then the driver must turn off all of the unused lines regardless of any alternative wishes on our part, but if

the code is to be run only on a simulation then all that is required is to turn on the lines that are read, and it becomes irrelevant whether the other lines are turned on or off.

This first note leads to the "-0" and "-1" options to the compiler, so that the user can choose his MDL+ source-code output with or without the extra driving information. The second note leads to the "-F" and "-S" options to the MDL+ compiler, which are both options to the Automatic Driving Phase. "-F" stands for Fabricated chip, and tells the driver to do as much as it can, including turning off all unused lines. "-S" stands for Simulation, and tells the driver to turn on all used lines so that the simulation will work, but not to bother with turning off any other lines, leaving the code size minimized and quite manageable to read. For more information about these and all options to the MDL+ compiler, see the end of this chapter when the interfaces of the compiler are discussed.

4.3.3 Programming in MDL+1.1

After understanding how the driving phase works, all that is left for the MDL+ programmer is to learn how to use it most effectively. These tips should seem fairly obvious after the last section.

First there is the way in which the programmer can be most assisted by the driver: That is, in not turning off lines. Every line that the programmer does not care about in a Connect will be turned off, so there is no need for him to explicitly turn any level-3 line off. The power will still be conserved, and his code will only state the relevant information and thus be much more readable. Even the verilog output, given compilation with the "-S" option, will be fairly readable and easy to debug.

When it comes to level-1 and level-2 lines, it is always safest to enable any lines that are to be read, but if they are read with a static source port then they can be omitted from the specification. As mentioned above, some future implementation of the driver might disable driving of the line, but only if the programmer had already messed up by making that port unable to affect any state of the chip.

Of course, interaction between the programmer and the compiler should include

help in both directions. The place where the programmer can best help out the compiler is with ports in dynamic source mode. If it is necessary to have a port in dynamic source mode, and the programmer knows that the choice is really between two of the input lines, and not all 30, then he should explicitly disable driving of the rest of those lines. It will save the chip a lot of power, and is a case where it is actually much easier for the programmer to do the specification then for the compiler to deduce it, as explained above.

4.4 Error Reporting and Interface Details

Now that we have covered all of the syntax and semantics of basic MDL+ (MDL+1.1), how the compiler will act on the code, and how the programmer can best use the various constructs to program, all that remains to be said in order to assist a person in actually using the MDL+ compiler to program for MATRIX is specifying its interface and methods of error reporting. After this section, which ends this chapter, the reader should be able to program in MDL+1.1. The next chapter will then describe the various intelligent phases of the compiler, bringing the reader to competency on MDL+1.4 and thus releasing all of the power of MDL+.

A good explanation of all of the command line arguments to MDL+ is given in the MDL+ man pages (Appendix B), thus we will merely refer to that appendix here for the interface details, and only explain the error reporting available in MDL+.

The error reporting available from MDL+ is far superior to what was possible with MDL, and most of it is done at the point of parsing the input. Error messages from the intelligent phases of the compiler (which are discussed in the next chapter), are mostly warnings that it was not able to accomplish the job it was assigned. For example, if the compiler is not able to route all of the wires on the chip (possibly because it was impossible to do so) then it will do as much as it can, and print a warning message for the user that it was unable to complete the job. The programmer should then look at the output (perhaps run it again with the "-0" or "-1" option to see what it looks like after routing), and change something. He might change the placement (by

running it with the "-2" option, changing the placement of a couple BFUs, and then re-running the code), or he might actually route in a way too complicated for the router to have figured it out (by running it with the "-1" option, and then routing the un-routed wires).

The parsing phase of the compiler can output many more errors, and as opposed to the intelligent-phase errors, these errors halt the compiler and do not output any data. All errors from the parsing phase of the compiler will be labeled as "Parse Error:", with a description of the particular error, and the line number that the compiler was parsing when it found the error. In most cases, this is the line that the error occurs on, but it is possible that an error might occur at an earlier point in the file, say at the definition of some variable, but not be noticed until later in the file, say when the variable is used. Through a large amount of testing of MDL+ and use of MDL+ no error has been on a line other than the one the compiler indicated, but we still mention the possibility since it did happen with MDL.

The most basic kind of parse error is a syntax error. This means that the source code file does not contain legal MDL+ source code matching the descriptions in this chapter and the grammar in appendix E. The error is not a semantic one, but merely a case where there is no possible way to understand it as a legal MDL+ statement-list. To deal with a syntax error, the programmer should start by looking at the flagged line of the source file, as the error will usually be obvious at this point. If it is not obvious, he should look at the point in this file where the appropriate construct is explained, and at the grammar in the appendix. If many tokens are on the flagged line, he can add arbitrary line breaks at all points where white space is allowed, and thus get a more specific error location from the compiler. Unfortunately, for syntax errors the compiler will never be more specific than "syntax error."

Semantic errors, on the other hand, will be accompanied by extremely verbose error messages. Since all of these error messages should sufficiently represent themselves, and in many cases offer assistance for how to alleviate the conditions that caused them, we will not go into any further details here.

This completes this chapter and this tutorial for MDL+1.1. The next chapter will

continue by discussing the intelligent automatic phases of the MDL+ compiler.

Chapter 5

Intelligent Phases of the MDL+ Compiler

The MDL+ compiler has several intelligent phases of compilation which were initially explained back in chapter 2. This chapter will delve into more detail about each of them: The router which determines which wires two BFUs will use to communicate with each other, the placer which decides where in a Layout a BFU will be placed, and the grouper which decides how to group various definitions of high-level functionality into BFUs.

To elucidate the functionality of each phase, as well as its effect on the MDL+ code, we will follow the definition of a simple 8-bit microprocessor in MATRIX. First we will introduce it as it would look in MDL and basic MDL+1.1, a language which has already been specified in chapter 4. Then, as each amount of functionality is added to the compiler and thus the MDL+ language expanded to be able to make use of that functionality, the 8-bit microprocessor's definition will be refined to take advantage of these changes. Seeing the example evolve one step at a time should make the job of each phase clear, and seeing the code overall migrate from the unreadable MDL to the reasonable MDL+1.4 should make the usefulness of this thesis apparent.

Since the MDL code itself is very long, it is not included in this chapter, but rather in appendix D. While reading through it laboriously is not advised, skimming it over will help to make this chapter more meaningful. In total, the MDL code is 245

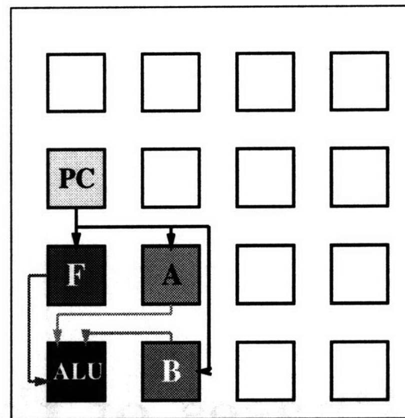

Figure 5-1: 8 Bit Microprocessor on MATRIX

lines long, and it compiles to a file that is 12,888 lines long (and thus 12,888 MATRIX configuration writes).

The MDL+ code is much easier to understand. Recall that there are five basic units in the most-basic MATRIX 8-bit processor, as shown in figure 5-1. At Layout location (1,3) there is a program-counter (PC) which counts through the addresses of all of the instructions. It gives this data to the three instruction store memories (I-stores) and they in turn determine the instruction appropriate to the current cycle and pass it on to the BFU that is acting as the arithmetic logic unit of this microprocessor (ALU), which in turn outputs the results each cycle. The F I-store determines which function the ALU will perform, and the A and B I-stores provide the data to be operated on.

Even given the above specification of the microprocessor, there is still a lot left to be determined, which will specify the nature of what it does. The A and B units could be providing a list of constants, which the ALU then operates on, they could be providing the locations of the BFUs where the ALU will then take data from (maybe the output of some other assemblies on the MATRIX chip, possibly even other microprocessor assemblies), or the A and B units could be providing addresses while the ALU BFU is always using its memory and ALU to perform **A ← A op B** operations each cycle. For simplicity, we will implement the first case, which is least likely to occur on an actual chip. The last case, which is the most interesting and

```
(def-bfu PC
  (ports
   (alu ((passa)) ((add0))) ;; default to the first ctx for C/R I
   (Aport (const 0) (static local))
   (Bport dc (const 1))
   )
  (control
   (ReduceI 0rx00000100)
   (ReduceII (not local))
   )
  )
```

Figure 5-2: PC definition in MDL+1.1

useful for a stand-alone microprocessor assembly, is a simple analogue of this one.

We build up the definition of this microprocessor for MATRIX in MDL+1.1 one piece at a time, starting with the PC. While each piece of the code is in an appropriate figure here, the entire MDL+1.1 source code is in appendix D.

The PC BFU is fairly simple. It counts up through the legal instruction addresses, and outputs the numbers as it counts. When it has reached the last legal address, it starts counting from the beginning again. Loops like this are more likely to occur on a MATRIX chip than completing counters, that perhaps stay at the final instruction number when done. Either is implementable and so we choose to implement the more useful one, as shown in figure 5-2.

This is a PC that counts through the five instructions (0 through 4) repeatedly. It does not take input from any other source, instead starting to count out at 0, and progressing through its range over and over again. The reason that it is guaranteed to start at zero, is because of the MATRIX convention to start chips out by configuration writing all zeros to the address that is all zeros for several cycles before running a program in a programmable global context. This has been guaranteed to put zeros on all of the wires, with no other effect, thus convincing the PC that it has just outputted zero, and everyone else will get that zero as well.

Once the current cycle number is generated by the PC, the design requires three instruction stores (A,B,F) to read in that cycle and output an appropriate instruction.

```
(def-bfu I-Store
  (ports
   (mem ((port single) (alua memory)))
   (alu ((passa)))
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )
```

Figure 5-3: I-store definition in MDL+1.1

Thus, our basic MDL+ design starts with an **I-store** BFU from which they will all inherit their basic I-store properties. The motivation for a common parent for these three BFUs is that they only differ in the actual memory cells that they posses, and cell initialization is a very well abstracted module in the definition of BFUs. The definition of the BFU I-store is provided in figure 5-3.

This definition is fairly straightforward. The only relevant pieces of configuration for the memory are that it is single ported in case we should wish some I-store to possess more than 128 usable memory addresses, and that the ALU takes its first input from the memory, which will thus be valued MEM[A]. Note that the importance of the omitted memory configuration bits differs: We do not care at all how the MUXs leading into the ALU B port or memory data port act, but we do care that the configuration read and write enables are both disabled. Here, we count on MDL+ to default all obvious defaults correctly (enable signals all default to disabled) when they are a small piece of a constant.

Once the MEM[A] value is passed into the A port of the ALU it is also passed out of the ALU since the operation for the ALU is PASSA. Finally, all of this would have to be specified in both local contexts if not for the control configuration. Here, the Compare/Reduce II stage is always set to fail, and thus the control bit will always be 0. (The Or-plane was removed, thus the control bit always comes from the Compare/Reduce II value.)

Once the I-store parent BFU is set it is fairly easy to define all three I-stores required for this problem, as is done in figure 5-4. The F I-store has its contents

```
(def-bfu F I-store
  (ports
   (Aport (static l1_n1)))
  (cells ((add0)) ((and)) ((xor)) ((or)) ((add1 invb)) ;; last is a SUB
   ))

(def-bfu A I-store
  (ports
   (Aport (static l1_nw)))
  (cells 0 1 2 3 4
   ))

(def-bfu B I-store
  (ports
   (Aport (static l2_w1)))
  (cells 5 4 3 2 1
   ))
```

Figure 5-4: Definitions of A, B, and F in MDL+1.1

entered as we would think of them at a high level (as ALU instructions), whereas the A and B I-stores are entered as the constant numbers, while they are all stored internally by MDL+ as constant numbers. Note the ease with which the definitions can be read: "Define the BFU F to be an I-store, except that its A port comes from the north and its memory cells get initialized to this."

The A, B, and F I-stores provide the entire 24-bit instruction needed by the ALU, and thus, we are now ready to define the ALU BFU, as in figure 5-5. To confirm the directions of the level-1 lines controlling the A, B, and ALU function ports, consult with figure 5-1.

Now all five relevant BFUs have been completely defined, and thus all that is left for the MDL+ code to do is to place them at the proper locations in a Layout and put that layout information in a chip definition, as in figure 5-6. Once this is done, the source code can be compiled with the MDL+ compiler and it will generate appropriate output code for the chip which can then be loaded into either a verilog simulation of a MATRIX chip, or a MATRIX chip itself.

Now that the MDL+1.1 version of the 8-bit microprocessor has been completely

```
(def-bfu ALU
  (network (l2_d2 n1))
  (ports
   (Aport (static l1_ne))
   (Bport (static l1_e1))
   (ALU (static l1_n1))
   (N1port (static l1_n2)) ;; to rebroadcast to B from PC
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )
```

Figure 5-5: Definition of ALU in MDL+1.1

```
(def-layout Micro8-layout
  (PC 1 3)
  (F 1 2)
  (A 2 2)
  (B 2 1)
  (ALU 1 1)
  )

(def-chip Micro8
  Micro8-layout)
```

Figure 5-6: Definition of Layout in MDL+1.1

defined, the later sections of this chapter will only discuss changes from it for the MDL+1.2, MDL+1.3, and MDL+1.4 versions. All of those versions of the code are available in their entirety in appendix D.

5.1 The MDL+ Router

The first of the MDL+ intelligent automatic phases is the automatic routing phase. In this phase, MATRIX wires which have been specified as communicating information from some named BFU A to some other BFU B, will be re-configured into low-level MATRIX configuration information. For example, if A is just to the north of B then the BFU B will be written expecting this information from the l1_n1 line instead of from the BFU A.

5.1.1 Syntax

The syntax added to the MDL+1.2 language to enable it to use the power of the router is very simple and has in fact already been introduced in chapter 4: When setting a port of a BFU to a static source mode value, it can be set to a static source that is the name of a BFU in addition to any other kind of static source. This is true when defining a port as a part of a BFU definition or even just as part of a BFU Ports definition or a Port definition.

The relevant structure is the **(static Name)** structure, with the same keyword **static** that is used to set port values to other more basic kinds of static source mode values. When using this structure, the Name argument must be a string that names a BFU in the current environment. This means that that name need not refer to a BFU at all points in the file, but it must refer to a BFU at the point of the use of this structure. Thus, if the desired effect is to cause a static source mode of a yet to be defined BFU, then an empty BFU definition of the source BFU can be used.

For example, assume again that BFU B is to have one of its ports set to static source mode with that source being BFU A, but in this case BFU A has not been defined yet. This situation is allowed if BFU A has been *declared* earlier, such as:

```
(def-bfu A)
```

Of course, the compiler merely sees this as a definition of A to be the Don't Care BFU, which will be replaced in the environment at the time of the actual declaration of A. But, to the programmer this can be seen as a type-declaration. It is most important when doing things like this in an MDL+ source file to remember that the version of A that will source this line is the current definition of A in the environment *in the same chip as B once that chip is completely defined at the end of the file.*

If there is only one definition of a chip in the file, and it has only one copy of A in it, then this situation is very simple. If there are no BFUs named A in the chip with B, or there is more than one BFU with that name, then the compiler will complain. In some sense, this can be seen as a dynamically scoped variable, whose value is found in the environment at the time of passing arguments to the **def-chip** structure.

5.1.2 Semantics

When used properly, the semantics of this new static source structure are fairly simple: At the time of chip-definition (actually the slightly later routing phase) the reference in the static source to another BFU is traced to that unique BFU in the chip's layout, and a reference to its location relative to the destination BFU is put in place of the reference to the name. Thus, the static source structure might be acting as **(static l1_n1)** or **(static l2_w2)**.

The only exception is that this is the behavior when the compiler can figure out how to route the wire. If the compiler can not figure out how to route the wire, it will warn the user once per routing failure with the message:

```
Warning: routing not good enough yet...
```

If any wires could not be routed, the user must have the compiler output MDL+1.1 code (with "mdl -1") which will include one or more un-routed wires, then explicitly route those wires and re-run MDL+ on the new code (effectively using "mdl -D"). The other alternative is to change the highest-level source code so that the MDL+

router can handle routing all of the wires. One of these alternatives must be chosen, because the driver and code-generator do not know how to handle un-routed wires (as it makes no sense at that level) and will thus halt without completing. This is as much a safeguard for the programmer as a result of the lack of other options since the programmer might accidentally use code generated without all of the wires routed, and it would probably cause strange errors, in effect "not being all there."

Finally, there is no guarantee that the MDL+ router will not do more complicated things, but it is guaranteed to do any simple one-hop routing. In fact, it is guaranteed to use more-local resources before less-local resources. If a level-1 line can be used to route a wire, it is used. If not, and a level-2 line can be used, it is used. If not, but a level-3 wire could be used and is available then the source BFU is set to drive a level-3 line which the destination BFU is then set to receive in static source mode. In this final case, being available means not only that nothing is driving the level-3 line, but also that the source BFU either has a network or floating port free to drive its local value onto that level-3 line, or is already passing its local output into one of those network-capable ports. Alternatively, the compiler could use a level-3 line that the source BFU is already driving its local output onto and the destination BFU can receive.

If none of these one-hop paths are available to route the wire on, more than one wire is necessary for the journey, and at least one BFU must be used to rebroadcast the signal in the middle of this journey. In this case, the MDL+ router might make use of many other wires and many other BFUs, or it might fail to route the signal, with the error behavior described above.

These requirements that we have set up for the router, of always using the best possible wire as long as they're available up until a certain point, leads us to using a simple greedy algorithm for its implementation. The actual algorithm starts by using level-1 lines and then level-2 lines whenever possible, since these resources are not used up. It then uses level-3 lines to satisfy as many as possible of the remaining wires that still need routing, starting with those which can be routed in one-hop.

Note that in theory all MATRIX chips are probably routable, when using time-

switching. The BFUs can be slowed down enough that they will be able to bounce and re-route many signals around until they can all get to the places where they were meant to go, and all of these signals can be registered at their final destinations in the appropriate mini-cycle when the data reaches the destination BFU. The reason that this type of routing is not done by the MDL+ router, besides not being needed by most designs we have looked at, is that time-switching was the one major component of the MATRIX chip which was not tested prior to fabrication. In fact, the functionality of time-switching on the MATRIX chip has never been verified in the verilog model, and it is highly unlikely that it will work completely (if it works at all) on the hardware once it returns from fabrication and a board is built to test it. With this in mind, we did not implement time-switching as a means of routing.

5.1.3 8-Bit Microprocessor

Now that we have described the updates to the syntax of MDL+, and the meaning of what the automatic router does, we are ready to update the code for the 8-bit microprocessor to make full use of the ability to route. This changes it in several ways.

The first change is that simple references to other locations become references to the BFUs that are at those locations. Thus, in the definition of the BFU named "ALU" the lines

```
(Aport (static l1_ne))
(Bport (static l1_e1))
(ALU (static l1_n1))
```

can be replaced by the lines:

```
(Aport (static A))
(Bport (static B))
(ALU (static F))
```

The second change is that the complex routing that we did to get the PC to the B I-source by rebroadcasting it from the BFU named "ALU" can all be removed and

replaced by a request to the router to find a way to route that. Really, this is all that we wanted as programmers — for the value from the PC to somehow get to B. We really did not care how the signal got there, and bouncing it through the BFU "ALU" and then out through a level-2 line was just one possibility. Since we did not have any particular attachment to that implementation and all we really cared about was that the signal somehow got from the PC to the B I-store, the new code is much more appropriate. The new code has the A port input on the "B" BFU configured with the line:

```
(Aport (static PC))
```

The third change comes after noticing the results of the second one. Now, the BFUs A, B, F all specify the A port to be configured as the above line of code, to take their value from the PC. Since all of the I-stores have now agreed on where to get their A port value from (not agreed in physical location but in abstract BFU name), this information can be incorporated into the parent I-store BFU definition instead of being in each one of the children definitions. In addition to making the code shorter, this makes the code more natural since it is in fact a basic piece of an instruction store that it gets its data from the PC, and not something that should be specified with each I-store. The only difference between various instruction stores is the instructions they output, which are stored in their memory cells, and now that is all that is specified with the definition of each child instruction store.

The fully modified and updated MDL+1.2 code for the 8-bit MATRIX microprocessor is available in appendix D.

5.1.4 Benefits

As a quick summary of the MDL+ automatic routing phase we review the benefits of having a router in MDL+.

- **Abstraction** - The programmer is not generally concerned with the details such as that one BFU is just to the left of another. He knows that a certain

BFU should be receiving input from another BFU, and this is the level at which he is now required to specify that information. As with abstraction in general, this enables easier changes to the source code: For example, if one of the BFUs moves then the programmer only needs to move it in the layout of the chip, instead of the large job of changing all of the static sources of the wires it read from and the ones that read input from it.

- **Analysis** - Using the automated tool we can get a better idea of what is useful to routing. We can let MDL+ route several designs of ours, and see when it runs into trouble. This can give us better insights into how we should be manually placing BFUs. This is not the most informative automatic phase because what it is doing is simpler than what other phases do.

- **Readability** - It is easier to read an MDL+ source file since it is no longer necessary to look at the Layout to find out which BFU another is reading at every encounter of a static source mode. It is also easier to write the code, because the programmer does not even need to know the details of where MATRIX provides wires. Instead, he can just put the BFUs that need to communicate with each other near each other, and if the compiler can not route it then he can move them around a little. The easiest way to understand how it makes the code shorter and more manageable to read and write is to compare it to the MDL+1.1 code for the 8-bit microprocessor in appendix D.

5.2 The MDL+ Placer

The next of the MDL+ intelligent phases is the automatic placement phase. This is the phase of the compiler that takes completely specified BFUs which it knows to be on a chip, and puts them at particular locations of the layout in that chip. It should be clear that this requires the additions that we made to the syntax for the automatic routing phase, since without knowing where on the chip various BFUs are to be located, it is impossible to actually route the BFUs together with specific wires.

5.2.1 Syntax

There is only one small change to the syntax of MDL+1.2 to make MDL+1.3, and that change occurs in the definition of Layout constructs. First of all, instead of just being able to specify the location of a BFU in a layout as two arguments which resolve to integer constants, they can also resolve to the Don't Care value, or even be explicitly "DC". Instead of listing both coordinates as "DC" it is even permissible to omit them both in which case two "DC" values are assumed.

Note that while it is permissible to make both row and column coordinates DC or omit them both, specifying one as an integer constant and the other as DC is not permitted, and will cause the compiler to halt with an informative error message. Similarly, omitting both coordinates is allowed but omitting one of them is not.

5.2.2 Semantics

If both coordinates of a BFU are DC or if they are both omitted, then the compiler assumes that the BFU is meant to be somewhere in the layout, but that the programmer does not wish to specify where. In this case, prior to running the automatic routing phase, the automatic placement phase will place it at one of the locations in the layout that is not taken by a partially-specified BFU which is forced to be in exactly that position by the programmer.

In other words, the compiler treats any BFUs explicitly placed by the programmer as stuck in those locations, and then finds places in the layout that are empty (or occupied by completely unspecified (DC) BFUs) and puts the non-stuck BFUs in them. In effect, it replaces the DCs or omitted arguments in the Layout construct's structure with integers.

The placer is guaranteed to place all of the unplaced BFUs, but while it will try to place them such that the router will be able to route the wires between them, it does not guarantee that the router will be able to route the results. If the programmer is unhappy with the automatic placement, he has several options:

- Using "mdl -2" to suggest placements, and then altering the placement in the

code and re-compiling.

- Changing the original source code and re-compiling.
- Using "mdl -2" and then recompiling the input with "mdl -2Z" or the output with "mdl -2UZ" until an acceptable placement is found, ending with recompiling the acceptable MDL+1.2 output code.
- Using "mdl -2" and then recompiling the output with "mdl -2UY" until the placement is acceptable, then recompiling the final output all the way.
- Using some combination of the above methods.

With all of the above in mind, the placer usually provides a good placement. Since it does attempt to place things such that the router's preferred routing mechanisms will be useful, placements are generally routable.

The actual algorithm controlling the placer is a simulated annealing algorithm, [KGV83] thus it includes information about the placement problem but not a human-designed way of solving the problem. This is designed to be the case in order to maximize the chance of finding out useful insight from the placer about placement, MATRIX, and CGRAs.

The details of the simulated annealing algorithm are as follows:

- Initial temperature is always 10.
- Maximum number of swaps is 30,000 plus 1,000 per wire that needs to be routed.
- The initial number of swaps performed at each temperature is 20 per BFU that is not stuck to a particular location or is completely non specified plus 10 per wire needing to be routed.
- The rate at which the temperature is changed after the maximum number of swaps at a given temperature is 0.5 plus 0.002 per wire needing to be routed, up to a maximum of 0.9.

- The rate at which the number of swaps at each temperature is increased is always 1.1.

- The function being minimized is a weighted sum of the wires needing to be routed, including wires passing to, from, or to and from BFUs which are stuck in place. Each wire routable with level-1 resources is weighted as 0 points, each wire routable with level-2 is weighted as 1 point, each wire routable with level-3 in one hop is weighted as 2 points, and each wire not routable in one hop is weighted as 10 points. Since the router is very bad at dealing with multiple-hop paths, and even if it can handle them they require a lot of retiming because of the registers involved, the placer attempts to avoid such paths completely. These are all equally weighted because any connection in MATRIX is routable with two level-3 lines.

- If all of the wires needing routing are routable with level-1 lines after the swaps for a certain temperature have been completed, the algorithm halts immediately, giving that answer. Otherwise, it continues until it has exceeded the maximum number of swaps.

5.2.3 8-Bit Microprocessor

Now that we have described the updates to the syntax of MDL+, and the meaning of what the automatic placer does, we are ready to update the code for the 8-bit microprocessor to make full use of the ability to automatically place. Much simpler than the changes because of the router, this merely removes all mention of exactly where each BFU is to be placed in the layout definition, leaving the layout definition in figure 5-7.

The fully modified and updated MDL+1.3 code for the 8-bit MATRIX microprocessor is available in appendix D.

```
(def-layout Micro8-layout
  (PC)
  (F)
  (A)
  (B)
  (ALU)
  )
```

Figure 5-7: Definition of Layout in MDL+1.3

5.2.4 Benefits

Though adding the placer does not make the MDL+ code much more readable or much shorter, it does aid a lot by making the code more abstract, taking tough work out of the hands of the human programmer, and aiding in our analysis.

This phase aids in giving the code extra potential for abstraction because the programmer can now abstract away the unimportant details of exactly where BFUs are placed, when these details are unimportant. If the programmer needs certain BFUs to be placed in certain places, such as nearby an I/O port to input or output data from the outside world to the chip, he can still specify their location. The difference is that other BFUs that are placed near these BFUs just to be near them are not specified anywhere in particular, while the reasons they were once placed there, if any (such as to be near other BFUs who they share wires with), remain in the specification.

The placer aids in removing tough work from being the programmer's responsibility and aids in our analysis. It is able to aid in these areas because placing is such a tough problem, and it removes this tough problem from the programmer's domain. The placer also gives us insight into how to solve this problem for MATRIX by seeing it solved many times and gives us insight into how to design CGRAs such that placement will be easy by varying the placer's assumptions about the routing phase and seeing how that affects its behavior.

Finally, it helps us achieve insight into these various areas better because of the algorithm's nature as a generic algorithm. If the algorithm was a specific implemen-

tation of the way a human would place, then it would still be useful in making the code more abstract and easier to modify and doing work so that the human programmer did not have to, but it would cease to be as useful a tool to study MATRIX and CGRAs.

All of the ways in which the placer will help us learn more about manual placing, MATRIX, and CGRAs are discussed in chapter 6.

5.3 The MDL+ Grouper

The last of the intelligent phases of the MDL+ compiler is the automatic grouping phase. This is the phase that takes higher level notions such as ALUs, memories, and registers, and groups their functionality into BFUs. Since these higher-level constructs might include several pieces of several different BFUs, it would make no sense to put them at specific places in the layout. Thus, while they can be included in layouts, they can not be placed at specific locations in layouts, and they would therefore make no sense if not for our placer. In this way, as the placer was only possible because of the router, so the grouper is only possible because of the placer.

5.3.1 Syntax

The changes to the grammar of MDL+1.3 to give us MDL+1.4 are all in the form of adding definition constructs and their appropriate structures in order to make the high-level objects which can then be grouped. Currently there are three objects available in MDL+:

- **ALU Objects** - The constructor is **def-alu-obj** and the basic structure is **Piece*** where each argument Piece in the list is like the definition of a single port in a BFU Ports definition, but only for the ALU port, A port, or B port. The ALU port keyword is **ALU**, the A port keyword is **Aport**, and the B port keyword is **Bport**. The difference between these definitions and the ones in a BFU or BFU ports is that here they can only define at most one local context, not two.

- **Memory Objects** - The constructor is **def-mem-obj** and the basic structure is **From (cells Piece*)** where From is an optional argument that is of the form of a single local context argument to a BFU Ports or Port, and the rest is an optional argument that looks just like the memory definition in a BFU definition.

- **Registers** - The constructor is **def-reg-obj** and the basic structure is **Name** where the Name argument is just a string that is the name of a BFU or an object (ALU object, Memory Object, or another register).

Registers have an extremely simple syntax, and the other types of objects have examples given as part of the new version of the MDL+ code for the 8-bit microprocessor.

5.3.2 Semantics

Each of the existing MDL+ objects describes a subset of a BFU. An ALU object is just the ALU of a BFU, taking its two inputs and performing certain operations on them. A memory object is just the memory unit of a BFU, with its first argument specifying where the address it should be reading out comes from, and the cells argument specifying the initial configuration of the memory. Since memory objects implement ROMs, this is the only configuration of the memory. The memory might be implemented on actual BFUs as a single-ported memory, or half of a double-ported memory depending on how much it is used and other considerations by the grouper. A register object serves the sole purpose of adding a one-cycle delay to some signal. In particular it delays the signal of the unit that it specifies with its argument.

The current implementation of the grouper possesses all of its intelligence in the ways in which it abstracts up from the BFU level to the object level, and does not do much grouping. ALU objects get their own BFU, and memory objects get their own BFU, partly because of difficulties with MATRIX itself in combining these objects. Then, registers are added to any ALU or memory object BFUs, as network or floating ports that were previously unspecified but now drive level-2 or level-3 lines. The other

option is to add the register as an entirely new BFU that then just passes its static source mode input from another unit straight through its ALU. With the current version of MATRIX (and likely any later ones), passing something through the level-2 and level-3 lines will actually end up adding on several more registered-delays, thus the latter method is currently the preferred one.

Finally, note that while the syntax of layouts did not change at all, their semantics changes slightly. The strings that are names given in Layout definitions used to be able to represent BFUs or other Layouts. Now they can also include objects — ALU objects, memory objects, or register objects — but when they do represent objects they no longer have the option of being explicitly placed. Objects can not even be specified to be "DC DC", even though that is the implicit definition. Instead, objects should be defined to be in layouts of chips just by placing their name alone between parenthesis as one of the arguments to a Layout definition.

5.3.3 8-Bit Microprocessor

Now that we have described the updates to the syntax of MDL+, and the meaning of what the automatic grouper does, we are ready to update the code for the 8-bit microprocessor to make full use of the ability to automatically group high-level objects.

First, since I-stores are just memory objects, we can eliminate the definition of an I-store and instead simply define A, B, and F to be memory objects, as is done in figure 5-8. An advanced grouper might even notice that these memories were small enough to be combined and thus include two in a double-ported memory unit of a BFU, and then read out two values each round with time-switching or a new ability of BFUs to do such things at once. This would be especially savory if the implementation did not require high-throughput due to the problem it was a part of. In this way, unnecessary throughput would be replaced by minimized area.

As is obvious from the figure the only negative development here is that all of the memory objects need to repeat that they are taking their addresses for read-out from the PC. Still, the tradeoff is a positive one.

```
(def-mem-obj F (static PC)
  (cells ((add0)) ((and)) ((xor)) ((or)) ((add1 invb)) ;; last is a SUB
   ))

(def-mem-obj A (static PC)
  (cells 0 1 2 3 4
   ))

(def-mem-obj B (static PC)
  (cells 5 4 3 2 1
   ))
```

Figure 5-8: Definition of Memory Objects in MDL+1.4

```
(def-alu-obj ALU
  (Aport (static A))
  (Bport (static B))
  (ALU (static F))
  )
```

Figure 5-9: Definition of ALU Object in MDL+1.4

Second, the BFU named "ALU" can be turned into an ALU object, since all it does is take in two signals and perform some ALU function on them. Thus, the definition of the ALU turns into the code in figure 5-9. The benefit here is that some of the code that was designed only to keep the BFU in the first local context can now be eliminated, and the grouper might be able to combine the ALU and one of the memory units for example, into a single BFU. This is especially the case if the microprocessor has very low throughput requirements, since then some sort of swapping between contexts or time-switching is possible.

Finally, note that while the definition of four out of the five components has changed, and their types with them, the layout definition does not need to change at all since the layout still includes the five objects with the same names.

The fully modified and updated MDL+1.4 code for the 8-bit MATRIX microprocessor is available in appendix D.

5.3.4 Benefits

The addition of the grouper to MDL+ makes code shorter, more readable, and more abstract. It could also do a lot of work that the programmer would otherwise have to do in combining these elements as tightly and appropriately as possible, and then we would be able to gain a lot from analysis of how it did that. However, with the current grouper that does not do much work these benefits are lost.

Thus, the benefits of the current grouper are mostly the effects it has on the source code. When defining a memory object, it's actually called a memory object, and the same with the other objects. This does make the code shorter and more readable but the key point is that it makes the code more abstract, at a very high abstraction level. After extending MDL+ this far, it is easy to see that any compiler that can compile to a description of ALUs, memories, and registers, can then compiler for MATRIX with MDL+ as a backend. Furthermore, if there are any other basic objects that the compiler would need to have to compile to, or additional fields of specification for the existing ones (such as a way to write the memories) then those new objects could be created for MDL+ or the old objects modified.

In the end, it is this connection to the back-end of high-level synthesis and a potential MDL++ compiler that is the most important part of the automatic grouping phase.

5.4 Improvements to MDL+ code

As has been shown throughout this chapter, MDL+ code improves in readability and shortens in length, as well as increasing in power, each time one of the intelligent phases is added to the MDL+ compiler. While we have no definitive evidence or wide-reaching study of this decrease in code length, we feel that it is an obvious side effect, and provide table 5.1. This table lists the number of lines in the coded versions of the MATRIX 8-bit microprocessor that are discussed through this chapter and are all provided in their entireties in appendix D.

This table shows the decrease in code size that accompanies the increase in power

Language	Lines	Words	Bytes
MDL	245	701	5102
MDL+1.1	61	168	1074
MDL+1.2	54	146	896
MDL+1.3	54	136	876
MDL+1.4	42	104	672

Table 5.1: Length of 8-bit microprocessor implementations

of the compiler. It shows, as expected, that the greatest improvement lies in moving from MDL to MDL+, although part of that benefit might be exaggerated since there might be ways of implementing the MDL code in a shorter way, using MDL templates. We do not believe that it would be legal in MDL to do so, but are not sure. Finally, small differences should be neglected as they might be due to comments, except for the differences from MDL+1.2 to MDL+1.3 which are precisely due to the omission of the placement data in the definition of the Layout construct. All of the data in this table was gathered with a UNIX 'wc' command and is not meant as anything more than a summary of the feelings generated by the 8-bit microprocessor example.

For more substantive research and data see chapter 6, and for a discussion of the improvements MDL+ has made over MDL see the appropriate section of chapter 7.

Chapter 6

Research done with MDL+

This thesis has laid the background for MDL+ and discussed the reasons for doing it, specified the basic and advanced features of MDL+, and referred to the research done with MDL+. In this chapter, all of that research and discussion will be explained and discussed. While no questions will be definitively answered, many will be addressed, and many useful intuitions will be set up.

The research done using MDL+, and thus this chapter, can be split up into three major categories. First, there are specific examples of MDL+ programs that have been written and compiled using MDL+. Second, there are architectures which have been compiled down with MDL+ to determine how the compiler and MATRIX would handle different styles of designs. Third, MDL+ has been converted for use with different backends. We will treat these three areas, which overlap somewhat, in this order.

Of course, it is very hard to make general conclusions from the evidence that will be given. Before beginning, we note that one of the more subtle reasons for this is that the design of MDL+ affects the outcome of how it compiles user programs. While this seems obvious, it is very important to keep it in mind while considering any results given in this chapter. For example, consider the possible human bias to not use level-2 lines. If we felt level-2 lines were useless, we might then write a compiler that never considered level-2 lines as a routing option. We would be able to use it to compile several examples and architectures and note that it never made use of level-2

lines, and we could claim this as evidence that they are useless and should be omitted from CGRA designs in the future. This would be a proof by circular reasoning, but a large-scale computer project in the middle would make it seem much more solid.

While we are probably not guilty of circular reasoning, the same problem can easily arise on a more subtle and innocent level. It is possible that for some intricate and emergent reasons the compiler which we have designed can not make use of certain characteristics even though they are useful. For example, the router might be able to use level-2 lines but the placer might, because of the way that the grouper worked or the way that we tend to describe designs, have placed the BFUs such that no one could route them using level-2 lines. Of course, just because the placer placed BFUs so that they could not use level-2 lines does not mean that the optimal placement would not be much better because it could then use level-2 lines. Nonetheless, throughout this chapter we will use the reasonable assumption that if our MDL+ compiler can not make use of something, then it is hard to use. Given that we make this assumption, this warning has been given at the outset.

At the end of this chapter we will specifically review some of the basic questions about MATRIX, CGRAs, and MDL+ that were not addressed explicitly in the rest of the chapter.

6.1 Examples

This section will discuss examples of programs that have been implemented in MDL+ for MATRIX. We will attempt to examine how the compiler treated these programs with its automatic systems and thus learn about MDL+'s phases as well as MATRIX.

Though these examples are specific data points and not giving us general facts about MATRIX and MDL+, it is still a useful starting point for this study as well as giving us many useful intuitions which we believe will continue to hold as people learn more about these sorts of problems.

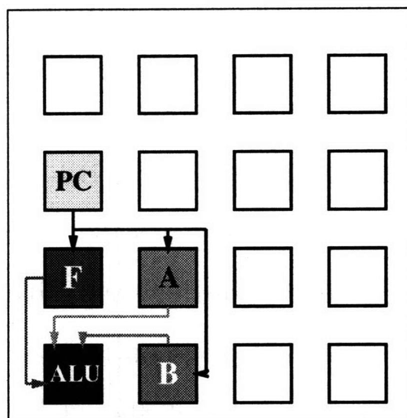

Figure 6-1: 8 Bit Microprocessor on MATRIX, placed by a human

6.1.1 8-Bit Microprocessor

We will start with the example that was strung throughout chapter 5, that of a simple 8-bit microprocessor. While this code has been discussed in that chapter, and is in appendix D in its entirety, the ways in which the MDL+ compiler treats it have yet to be discussed. The implications for small-word microprocessor architectures on MATRIX and with MDL+ will be handled in section 6.2, while this section will treat it as a good introductory example about how to compile for MATRIX.

Since the MDL+ grouper is in its infant stages of development, the MDL+1.4 code turns out to be no better for us than the MDL+1.3 code. The placer thus needs to place five BFUs (PC, A, B, F, ALU) onto the layout of a chip. When we designed this chip originally, we saw it placed as in figure 6-1 (as seen in [Mat96a]), but that is not an easy to route design.

This human design has the three I-stores all clustered around the ALU, easily providing their distributed 24-bit instruction to it each cycle over level-1 lines, and similarly the PC is near enough to the F and A I-stores to get the cycle number to them each cycle over level-1 lines. The problem here is that the B I-store is two rows down and one column over from the PC, and thus no level-1 wires run between the PC and the B I-store in order to communicate the address from which B should be serving its stored value. Since level-2 and level-3 lines only run in a single row or

column, they are always helpless to communicate data directly between two BFUs that are neither in the same row nor the same column, thus there are no one-hop paths at all between the PC and the B I-store.

In the MDL+1.2 code that we provided, the B I-store merely asked for the value anyway, leaving it up to the router to figure out how to route this line. In the MDL+1.0 code, the routing needed to be done explicitly, and so some rebroadcasting was added. When doing this, the first question to be answered is which BFU should be doing the rebroadcasting. If possible it should be a BFU that already existed (non-DC) but has enough power left unspecified that it can add the rebroadcasting functionality. It should also be in a position to receive from PC and transmit to B. All of A, F, and ALU are in ideal positions for this. The only problem is that while all three of them can read PC's level-1 output and all three can have B read their level-1 output, all three are already using their level-1 output to communicate a different signal. (The ALU is communicating the final output, and not read by anyone in this description, but we clearly care which value it outputs from its ALU.)

This means that either one of these three must use a level-2 or level-3 line to transmit the data to the B I-store, or a new BFU must be created. If we were to create a new BFU we would have it be placed in the (2,3) spot to fill in the 2x3 grid that this microprocessor is on, and it would merely read l1_w1 into its A port for the PC value, and pass that value through its ALU, which could then be read by the B I-store as its l1_n2 value. Instead, we chose the option of using re-transmission on network wires, which is what level-2 lines are generally used for. The ALU can read the PC (although the basic design does not require it to) using level-1 lines. Since it is at location (1,1) it can transmit horizontally with its level-2 lines, which can then be read by the B-Istore, thus controlling the data that the B-Istore sends right back to the ALU.

This rebroadcasting design requires the memory of the B I-store to be slightly off of the other memories, so that they (A, F) output the values they would-have-outputted a cycle later than they used to, to keep them in-sync with the B ALU that is getting each PC one cycle later. This detail was omitted from the previous

chapter, as it is not even clear that the difference is exactly one cycle (depending on network, network port, and level-2 line design it might be 2 cycles, which is the way the current implementation works). Although the fix for this problem is simple, it is a good indication of why the compiler was designed to always search out designs that solely relied on one-hop paths, and if possible only level-1 lines (thus no chance of extra registers).

And, in this case, the compiler succeeded in its goal. The placement of the 8-bit microprocessor by MDL+ is shown in figure 6-2 and it clearly is routable with only level-1 lines. Note that this, and the later figures of actual MDL+ placements, are not bottom-left justified. The MDL+ placer has not implemented the proposed bottom-left magnetism, so this is to be expected. Of course, they can always be shifted down diagonally towards the bottom-left, but not all designs will be able to occupy the BFU at (1,1). For example, this design can have the F BFU shifted to (1,2) or (2,1), but not (1,1) because of the checkerboard pattern of level-2 lines. While the direction of level-2 transmission is important to keep in mind, and will be relevant in later examples, we could actually shift this example so that the F BFU moved to (1,1) after noticing that the design uses only level-1 lines.

In fact, this design is superior to the human placement of figure 6-1 in almost every way. It only occupies a 2x3 grid, it uses only level-1 lines, and it can easily be shifted in any direction, for example in order to fit in a 3x2 space. Finally, the ALU, where the only output comes from (and the only input goes to if the A and B instructions tell it where else on the chip to dynamically source lines from) is located in the corner, and thus (depending on how the design is shifted) is near the most other units. Much of this was true for the human design, but not all of it. While using level-2 lines it was troublesome because it could not be arbitrarily shifted, in addition to the reasons above. The only potential drawback to this machine-placed design is that the empty slot in the 2x3 grid is not at one of the corners, potentially making it harder to get another unit to overflow into this slot.

There are a few things we can learn from this example. First, we see the usefulness of length-2 level-1 lines. The human design puts all BFUs that need each other as

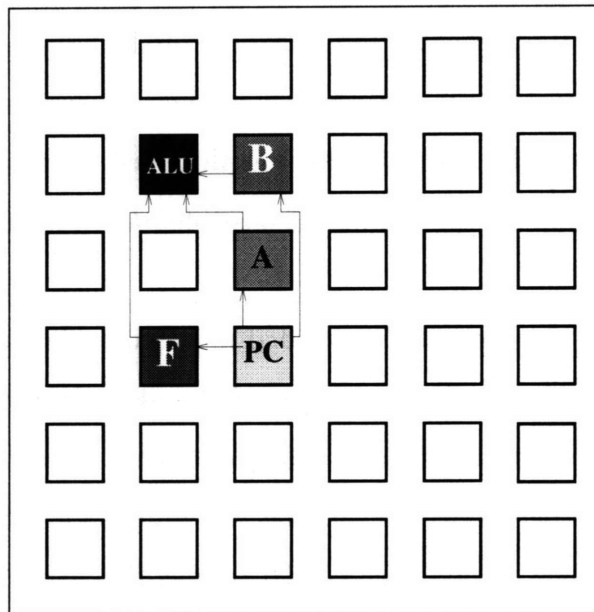

Figure 6-2: 8 Bit Microprocessor on MATRIX, automatically placed with mdl -2

close to each other as possible. But, the machine design shows an understanding that being two units away in a cardinal direction is the same distance away as being only one unit away, when using the most relevant distance metric. This results in the three units that need to be communicating with the ALU and the PC being spread out a bit more. Only one of the three is next to both the ALU and PC, while the other two are each next to one and two away from the other in a cardinal direction.

Second, we start to notice a certain symmetry in the machine design. This is a trend that will continue through many of the MDL+ placements. The B BFU and the F BFU have the same needs, and so they end up in symmetrical positions: One shifted a column from the PC and two rows from the ALU and the other shifted a column from the ALU and two rows from the PC. Meanwhile, the third BFU with the same needs is as close as possible to in between them, and could be placed one column and one row off of either the ALU or PC, and merely one row off of the other. Finding similar units placed at similar points in the design will be a recurring theme.

Third, we begin to notice the Knight-pairs or k-pairs. This is how we refer to any two BFUs that are displaced from each other by one level in one cardinal direction

and two in the other, the places where a knight would be able to move on a chess board. BFUs that are k-pairs are as close as they can be to each other without actually having any wires to communicate over in one-hop. One possible addition to MATRIX would be adding wires for just such communication, but it is an unlikely development.

Instead of changing MATRIX, we can use the notion of k-pairs to help our designing ability. Notice that in the human design (figure 6-1) the only k-pair is the PC and the B-Istore which need to communicate and thus there is a problem, in fact the only major problem with that design. Next, consider the machine design (figure 6-2) where there are two sets of k-pairs. The B and F I-stores form a k-pair but they have no dependency on each other for signals so it does not matter. Similarly, the PC and ALU BFUs form a k-pair with each other but they do not need any wires between them. The absence of problem k-pairs in this design is a key to its success.

Gaining intuition about k-pairs starts us on the notion that one of the paths to designing a MATRIX placement might be placing disjoint BFUs in k-pair positions. Thus, we have acquired an intuition about how to place BFUs that do not depend on each other in addition to the easier intuition on how to place BFUs that do depend on each other. This intuition was, of course, not gleaned merely from this example, but by watching MDL+ place many examples. Finally, the best non-dependent BFUs to share a k-pair are the ones that share other BFUs which they both share wires with. The reason is clear: They do not take up the few usable positions for the other BFUs, and there are several slots remaining where those BFUs can communicate with both of them. This is one example of logic which might be explicitly added to the MDL+ placer in the future.

6.1.2 Polynomial Evaluator

Another basic example of functionality on MATRIX is that of a basic group of BFUs that would calculate a quadratic equation. That is, given any a, b, c, x, it would output $ax^2 + bx + c$ each cycle. This problem was initially posed by André DeHon to Ian Eslick. By the time people were done passing around the responsibility for it, Ian

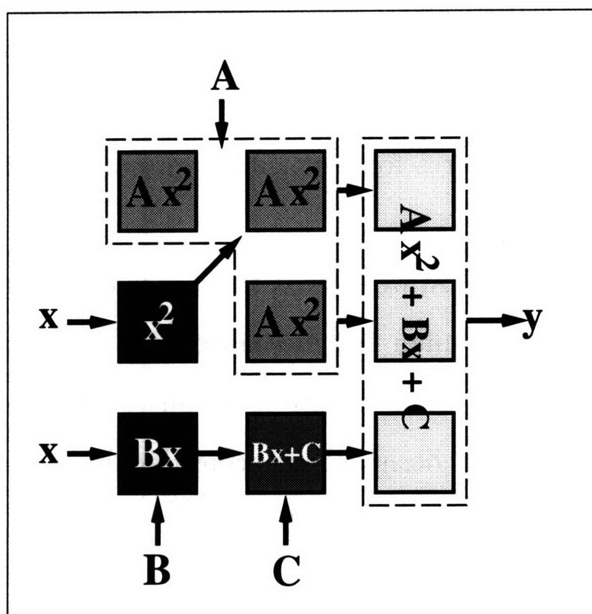

Figure 6-3: Polynomial Evaluator designed by a human

and Morris Matsa had developed a version of it on an old specification for MATRIX, and Dan Hartman and Ethan Mirsky had given input on a few of the unclear details of how MATRIX acts in special circumstances. Since the look of MATRIX changed a lot, the initial design became out of date prior to MATRIX fabrication.

Finally, during the summer of 1996, Yael Levi designed a MATRIX chip to solve this problem and implemented the solution in MDL. We have taken this solution and converted it into MDL+. The MDL design, as shown in Yael Levi's figure 6-3 involves a lot of complicated level-2 wires rebroadcasting signals so that all timing issues are worked out correctly to maximally pipeline the design. The MDL+ source code instead asks the compiler to place and route an unpipelined version of the same code. Should we wish to achieve the maximum throughput of one calculation every two cycles we could always add registers to the MDL+ code before compiling it, or add level-2 retiming to the design after the compiler places and routes it.

The polynomial evaluator's design is somewhat simple, working with a 2-cycle period. All nine BFUs switch between local contexts each cycle, all being in the first local context at the same time and then the second local context the next cycle. It

starts with four one-byte values: x, a, b, and c. The BFU at (1,1) multiplies $b * x$ to get a 2-byte value over two cycles, outputting the low byte in the first cycle and the high byte in the second cycle. The BFU at (1,2) does the same thing for x^2.

The 3 BFUs labeled Ax^2 then try to calculate that 3-byte value. The BFU at (2,2) multiplies a by the low byte of x^2 while the BFU at (1,3) multiplies a by the high byte of x^2, one starting in the first local context, and the other starting in the second. Each of these two BFUs outputs a 16-bit result, but since they received their byte of the x^2 input off by a cycle, and output their output off by a cycle, thus the high byte of the low bytes and the low byte of the high bytes is available at the same time, and the BFU at (2,3) adds them together in order to calculate the middle of the three bytes that make up ax^2.

At the same time, the BFU at (2,1) is adding the 8-bit constant c to the 16-bit bx that it gets from the BFU at (1,1). It does this addition over the course of its two cycles. When it gets the low byte from the BFU at (1,1) it adds in c with an ADD0 ALU operation. The next cycle it adds the high byte of bx from the BFU at (1,1) to the constant 0, with the ADD operation taking the carry-in from its own carry-out from the previous cycle. This requires the configuration of the BFU being set to **Right:Local** as well as **CarryPipeline**. The result is that this BFU outputs the quantity $bx + c$ starting with the low byte in the second cycle and then the high byte in the subsequent first cycle.

Finally, the three BFUs in the third column add the various pieces. The BFU at (3,1) takes the low byte of $bx + c$ from the BFU at (2,1) and the low byte of ax^2 from the BFU at (2,2) and adds them together to get the low byte of the answer. The BFU at (3,2) adds the high byte of $bx + c$ from the BFU at (2,1) and the middle byte of ax^2 from the BFU at (2,3), to form the middle byte of the answer. And the BFU at (3,3) adds the high byte of ax^2 from the BFU at (1,3) to the constant 0 to produce the high byte of the answer. Since all of this data is delayed until a specific cycle (with level-2 wires for retiming) and the third column BFUs are not set to **CarryPipeline**, the carry bits work correctly and these three BFUs compute a 24-bit ADD in one cycle. The output is then the 24-bit answer of $ax^2 + bx + c$.

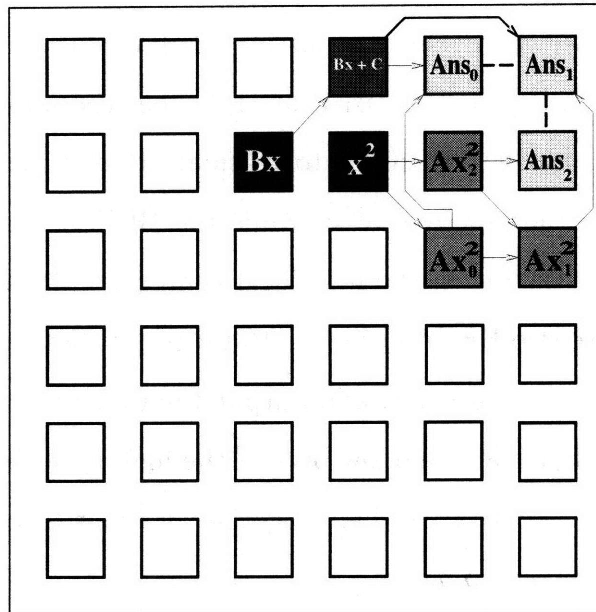

Figure 6-4: Polynomial Evaluator as placed by the compiler

The MDL+ compiler took these same BFUs (the algorithm still designed by a human) and placed them as shown in figure 6-4. This design is not compact in a 3x3 grid as the human design was, but the BFU at (3,5) can clearly be moved to (4,4) to make the design fit into a 3x3 grid. The compiler's design only uses level one wires, but so did the human design (leaving the level-2 wires free for retiming, and the level-3 wires free to input the new parameters every other cycle).

In many ways this design is similar to the human design, both being equally good. However, there are differences that we can learn from even if they are not too important. For example, the compiler does not try to put the x^2 BFU next to the BFU that calculates the middle bits of ax^2. The compiler does not place them next to each other because they do not need to be placed near each other. Although there are two BFUs which they both share communications with, they do not communicate with each other. That is why they are placed in a k-pair.

The only thing unique about this example is that the compiler placed the three units computing the 24-bit add next to each other. It even placed them in order, though not in a single row or column. While placing them next to each other is

necessary for a working design (and not even diagonally next to each other) we will see later in this chapter that the MDL+ compiler does not have enough information to always do so, and that in this example it just got lucky.

Finally, this example shows us two possible uses for the MDL+ placer. First, it can be used to show us alternate ways to place designs. While the human designer came up with a suitable placement, the computer came up with a different way to place that still works and has different benefits. The compiler's design still fits in a 3x3 grid (after a slight adjustment) and still only uses level-1 wires for basic communication. However, it has shifted the BFUs around enough that the three BFUs with the answer are at one of the corners of the unit instead of on one of the sides. Depending on the design of the rest of the chip, some other units might be using this answer instead of merely having an IO port outputting them. In that case, their design would impact which of these two designs we would choose for the polynomial evaluator.

Second, on a very basic level the human's work at placement was not necessary. It would be far easier on the programmer to run the MDL+ compiler on his problem and then possibly rearrange one or two BFUs. Given that the design the compiler came up with was just as good as that of the human programmer, partially relying on the compiler becomes a reasonable model for programming on MATRIX.

6.1.3 Other Examples

There are several other examples of designs on MATRIX available in [DeH96, Mir96, Mat96b].

6.2 General Purpose Computing Architectures

This section contains many examples of how the MDL+ placer might place various architectural styles that might be implemented on MATRIX. Every one of these possible placings was in fact an actual placing calculated by the MDL+ placer when the MDL+ compiler was run on the example, but since the placer has been optimized over time, such as by changing the parameters for the simulated annealing algorithm, and

since the placer's algorithm is non-deterministic, these are still only possible placings, and not necessarily the ones we would get by running the program again. They were all compiled with "mdl -2".

6.2.1 8-Bit Microprocessor

The first example is that of an eight-bit microprocessor. We viewed this as seen in figure 6-1 while the placer produced a chip as shown in figure 6-2. The ramifications on general design have already been discussed above, and we will now consider the use of these small microprocessors in MATRIX designs.

As these units are small and compile well, it seems advisable to use them when applicable in MDL+ programs. They can be used to control a part of an algorithm that runs slower than other parts, and thus does not need high throughput. If they are placed well, as in figure 6-2, they can even run high-throughput parts of an algorithm, since they can be doing something different every cycle. This architecture seems like such a good building block that we have considered adding a "micro8" object to MDL+ and having the grouper expand one of them into several BFUs. At first it could expand one of these objects into five BFUs, but later it might group these pieces with other pieces in the design or even with each other, as discussed in section 5.3.

6.2.2 32-Bit Microprocessor

The next sample architecture is that of a 32-bit microprocessor. We viewed this as seen in figure 6-5. Even though the PC can reach all of the I-stores in one cycle with level-1 lines, this human design leaves much to be desired. Because of the design, the I-stores share their row with none of the ALUs they need to supply instruction to, and only one ALU shares each of their columns. Furthermore, that ALU was already reachable with their level-1 lines. Thus, none of the ALUs can receive instruction with level-2 or level-3 lines in one-hop.

This would be all right if all wires were routable with level-1 lines, but the F BFU

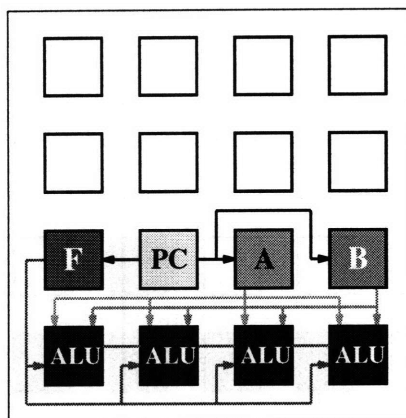

Figure 6-5: 32 Bit Microprocessor on MATRIX, placed by a human

and B BFU each can not reach two of the other-side ALUs with their level-1 lines and the A BFU can not reach the ALU at (1,1) with level-1 lines. Meanwhile, the PC can easily be connected with three ALUs even though it does not need to, and so its position is wasted. The PC was placed there so that it was in between the I-stores and as close as possible to all of them, but this ends up being the wrong decision. Of course, the great benefit to this design is that is fits into a 4x2 grid.

Fortunately, this design is easily routable using multiple hops. Each of the instructions from the I-stores can reach the ALU directly beneath it in one cycle over the ALU's l1_n1 line. Then, the ALU can take that instruction, and rebroadcast it on the horizontal level-3 lines (say, l3_h1 for F, l3_h2 for A, and l3_h3 for B). Finally, all four of the ALUs are set to read their A, B, and Function ports from the level-3 lines. Since all bytes of the 24-bit instruction word arrive via the same type of path, none of them need to be sped up or slowed down relative to the others.

Our intuition tells us, as we learned in the previous section, that using all level-1 lines and never needing to rebroadcast is always a good idea. However, in this case it is not immediately clear what the drawback to using multiple-hops is. Nonetheless, our intuition does hold up and is thus strengthened, since the ALUs will be computing based on an instruction several cycles old, and thus every branch will necessarily have several branch delay slots.

To remedy this situation, in hopes of finding a placement that will not require

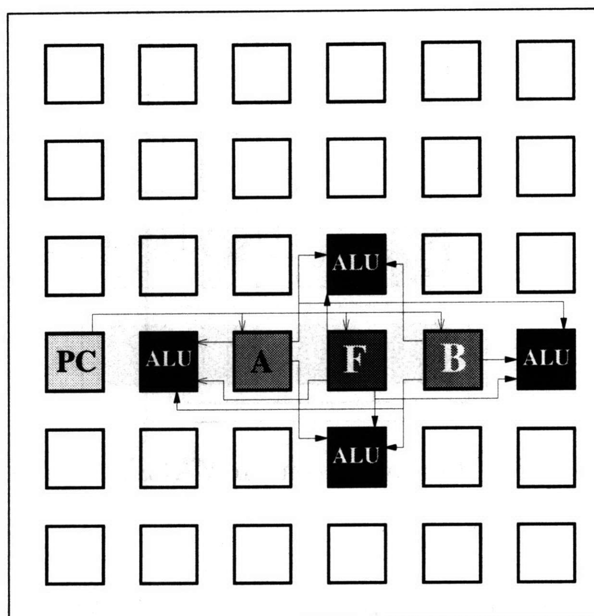

Figure 6-6: 32 Bit Microprocessor on MATRIX, automatically placed with mdl -2

multiple-hop paths for any wires, we run this through the MDL+ placer. The automatic compiler produces a chip as shown in figure 6-6. The first thing apparent about this design is that it is far less compact than the human one. Nonetheless, it might fit onto certain chips better, given their other prior designs, since it takes up only one row besides two jutting-out BFUs. The second most noticeable quality of this design is that it is a machine-looking design. That is, it does not seem that a human would be as likely to lay the pieces out in this manner.

Upon deeper consideration of the design, it becomes clear that there are several things we can learn from it, as we did from the previous examples. Furthermore, there are many advantages to it, and one big disadvantage. The biggest advantage is that all wires are routable with one-hop level-1 and level-2 wires, not even using up any of the precious level-3 resources. Note that the PC, A, and B BFUs are all two BFUs over from each other, thus all get to transmit horizontal level-2 lines. The PC can reach the A I-store with level-1 lines, but needs its l2_d2 wire to transmit the cycle to the F and B I-stores. Being in the middle of the ALUs, the F I-store can transmit to all four ALU BFUs with level-1 lines, while the A and B I-stores can each

communicate with three of the ALUs with level-1 lines and transmit in the correct direction to give their byte of the instruction to the fourth ALU with a level-2 wire. In all, three level-2 wires are needed (four, but two of them are the same l2_d2 wire from the PC).

We can make several observations about this design. First, the length-2 level-1 wires were useful again. Without them, this design would not be possible, since this design makes use of the every-other BFU that can not transmit level-2 wires in a direction still being able to transmit level-1 to half that distance of two units away.

Second, the entire width of the chip really can be used. This design would not be possible on a chip that was less than six units wide. It is not merely a case of this design spreading out when it could be more compact — other designs could place these units in a more compact area but this design requires six BFUs wide. The rightmost five BFUs in the third row are integral to the design, and there is no way (given the rest of the design) that the PC could reach all of the I-stores in one-hop besides being in the sixth slot on that line.

Third, we see once again that form follows function. The PC is needed by the three I-stores but no one else, so it is moved to a spot far out of the main design, but where it can still reach those I-stores. The three I-stores, on the other hand, are very necessary for many units, and to be close to all of them they are placed in the middle of the main design. The other option would be placing the ALUs in the middle and the I-stores around them, but then there would be no way for the PC to transmit the cycle to all of the I-stores in one-hop.

Fourth, again the design displays a lot of symmetry. Looking at the main design, which is to say everything except for the PC, and treating the I-stores as all the same (since they are as far as the placer is concerned), the design has both horizontal and vertical symmetry. The A and B I-stores have the same set up as each other, with one ALU next to them in a cardinal direction, two ALUs next to them diagonally, and one ALU three units away but reachable with a level-2 line. And the final I-store, F, is in the average of these situations, two ALUs two away in opposite cardinal directions and two ALUs next to it.

Fifth, we see that there is still a lot to be learned by examining the k-pairs. Only the ALUs are in k-pairs in this design, and each of the four ALUs is in two k-pairs, each with a different ALU. These k-pairs join the units that are most numerous but do not need contact with each other at all. Once they are formed, they are slid into a position where they do not take any of the valuable resources (in this case, horizontally-transmitting level-2 BFUs in the third row). Then, the rest of the design falls into place.

Finally, after having extolled this design's virtues, we mention the one great fault: The ALUs are not near each other. While this was one of the key points of the design, putting the ALUs in k-pairs with each other thus forming the diamond, in practice it kills the design. The problem is that there is currently no way in MDL+ to tell the compiler that the ALUs want to be next to each other in order to do a wide-word operation. They can be set to take their left and right carries from specific directions, but not from specific other BFUs, and thus there was no constraint keeping them near each other. Thus, this design turns out to be an ideal 4-way SIMD architecture, but not a real possibility for a 32-bit microprocessor.

6.2.3 MIMD

Devices utilizing more than one program counter (control unit) per die are considered MIMD (Multiple-Instruction, Multiple-Data) machines. Figure 6-7 shows a generic 2-PC, 8-bit MIMD machine implemented on MATRIX. Just as in the VLIW case, a variety of devices such as PADDI-2 [YR95] have chosen a specific data point of these architectures, while MATRIX gives a designer the option of changing those choices. [Mir96]

Since this is equivalent to placing two of our 8-bit microprocessors on the same MATRIX chip, the observations we made earlier apply here as well. The points that were mentioned about placing the "holes" in an 8-bit microprocessor design such that many may easily be placed on the same chip start to become more relevant when we consider using n-PC MIMD machines for larger values of n.

The MDL+ placer produced a chip for this 2-way MIMD design as shown in figure

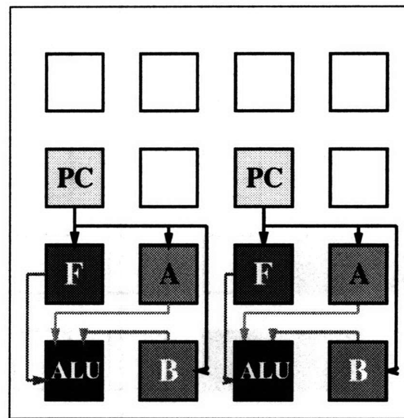

Figure 6-7: MIMD Architecture on MATRIX, placed by a human

6-8. This design is interesting both in that it shows us how two separate pieces of a MATRIX design might be placed near each other, and because it gives us two new ideas of how 8-bit microprocessors might be placed.

The 8-bit microprocessor with its PC at (3,1) is an obvious addition to our set of possibilities. Since the PC needs to communicate with the three I-stores and they in turn need to pass information on to the ALU, it puts each register stage in a separate row. The drawback of this approach is of course that it requires three rows and three columns, thus the minimum enclosing rectangle is 3x3, bigger than the other ways of designing the microprocessor. The benefit of the layout of this design is that the four corners of the 3x3 box are all empty, thus allowing other units to easily make use of these resources, as the other unit in the figure did by sticking its B I-store in the upper-left hand corner. Once again, this design uses only level-1 lines. Note that none of its BFUs are in k-pairs.

The other microprocessor on this chip is in a more compact 2x3 design, again only using level-1 wires, and again in a new configuration. The only k-pair here includes the A and B I-stores which do not need to communicate, but both share communications with the PC and ALU. There are, of course, many k-pairs with one member in the upper microprocessor and one in the lower microprocessor, as should be expected.

With this chip, we will make a few new observations. First, there are many ways

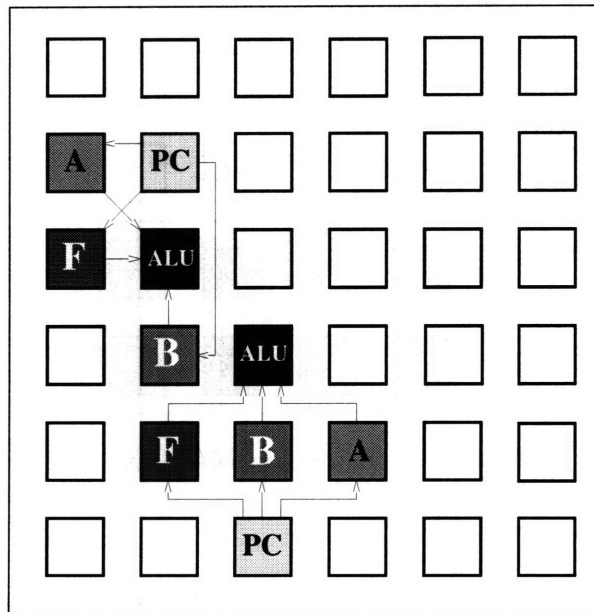

Figure 6-8: MIMD Architecture on MATRIX, automatically placed with mdl -2

of laying out any small design, such as the 8-bit microprocessor that we have seen a lot of. This implies a benefit of using an automatic placer, since it will be able to try out each of these designs or even come up with new designs fitting the problem, while the human would more likely default to an old design. Of course, part of the beauty of our system is that the human can observe the final product and rearrange units if he sees a better old design to put them into, or even a new one.

Second, the computer-designed chip seems more haphazardly-placed than the human-designed chip. Given the placement algorithm this is not a coincidence, nonetheless it is noteworthy. Furthermore, it is not merely true of this one example: it will be more obvious in later examples and is present in earlier examples. The human designs all show a desire to fix everything in a simple form and then tile that form, while the computer designs use very irregular forms, because the placer's algorithm gives no bias toward "regular" forms, likely bringing about better designs since benefits arise on a MATRIX chip when the design includes more level-1 lines, not when the design looks more regular to a human.

Third, we note that the way in which any multiple-BFU structure is laid-out,

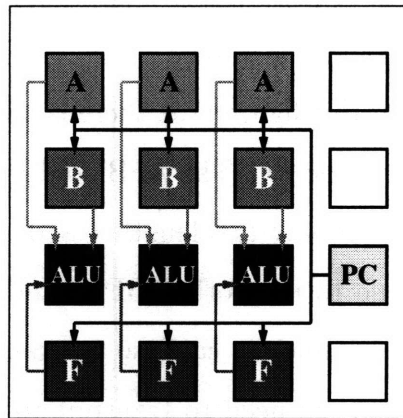

Figure 6-9: VLIW Architecture on MATRIX, placed by a human

but in particular 8-bit microprocessors, will likely ultimately depend on the external constraints on its data. The 8-bit microprocessor might need to receive data at the ALU, and will almost always need to transmit data from the ALU. Thus, the design which is chosen will most likely be the one that fits in the available space and puts the ALU the closest to the other BFUs it needs to communicate with. This is why having so many different designs is so useful. The human design and above computer design leave the ALU at the corner of a 2x3 box, the upper design in this case leaves the ALU at the middle of a 2x3 box, and the lower design on this chip leave the ALU on its own column or row, jutting out of a 2x3 box with the two far corners empty.

6.2.4 VLIW

When 24 bits of instruction are insufficient, it is easy to deploy extra resources to the instruction caches. This could result in the system shown in Figure 6-9, which is a VLIW system. Here we can see how we do not need to replicate the program counter and control. This allows for a much higher efficiency of silicon usage.

In a higher-form of MDL+ leading towards MDL++ this might be implemented as a VLIW object construct which would require the user to specify all of the processor's configuration. Then, the grouper would determine how many BFUs must be allocated to being ALUs, and which of the ALU's ports will be in static value mode, static

source mode, and dynamic source mode. The compiler could then allocate as many additional BFUs to control as are necessary.

It should be obvious from this figure that there is only one problem from a placement and routing perspective: the PC. The A, B, and F I-stores associated with each ALU can reach their ALU with only two length-1 and one length-2 level-1 lines, but the PC needs to get its value to nine other BFUs and that is difficult. Given this placement, the PC has level-1 lines it can use to transmit to only two ALUs and two I-stores, and it does not even need to transmit to the ALUs. This means that of the 12 slots that it could transmit to over level-1 wires, two are filled by BFUs it does not need to transmit to, and eight are filled by nothing at all, with only 2 of 12 filled with useful BFUs that are recipients. Most likely, the best way to route this placement is to pass the PC value over its l2_d2 line horizontally to the three ALUs, have them each rebroadcast on a level-3 line vertically, and have the I-stores pick up these vertical level-3 lines and use it as the PC, then issuing their instructions to the ALUs.

It might even seem that this is the best we can do, and that there would be no way to place this design such that it would be routable without using the depletable level-3 resources. However, the placer did an excellent job as can be seen in figure 6-10. Not only did the MDL+ compiler manage to avoid level-3 lines, it routed the entire VLIW design using only level-1 wires. The three ALUs receive all of their unique 24-bit instruction over level-1 lines (sometimes making use of the length-2 lines), and the PC has the 9 I-stores placed in nine of the twelve locations it can transmit to using level-1 wires. The remaining three locations are unused, two to the left and one because it is past the boundary of the chip to the right.

This design seems less compact, but it is actually very compact. While the compiler was not told to make the design compact, the user could easily shift the ALU at (3,3) to the empty slot at (4,2) after placement, and the router would still be able to route the entire chip using only level-1 lines. Then this design would use up only a 3x5 sub-grid, whereas the human design required a 4x4 grid. Thus, the computer design uses a smaller box (15 instead of 16 BFUs, or a box with 2 empty "wasted"

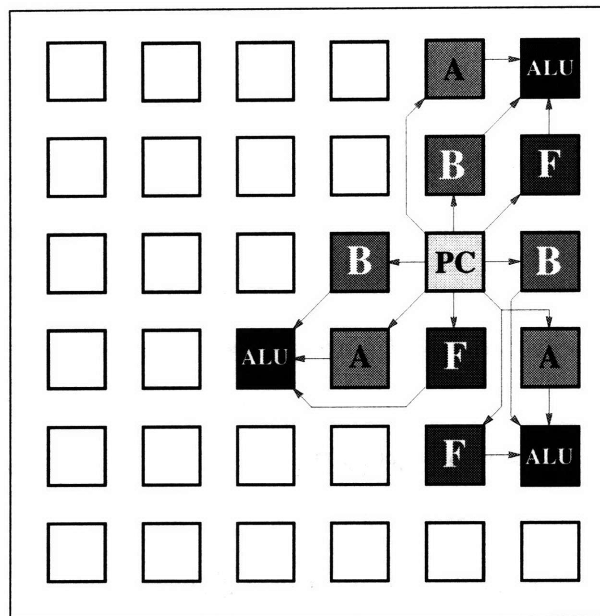

Figure 6-10: VLIW Architecture on MATRIX, automatically placed with mdl -2

slots instead of three). Furthermore, two 3x5 units will fit on a single chip, whereas two 4x4 units will not. Finally, since the computer's version uses only level-1 lines, it can be flipped and translated and rotated as much as it wants and still work, thus the two empty slots can be shifted to BFUs (4,3) and (5,3) which would be the only corner of the unit not on a boundary edge of the chip, for maximal chance of being used by other units.

Of course, this design faces the same major drawback of the 32-bit microprocessor design: The ALUs can not form up wide-word operations because they are not all near each other. As before, there are several ways we might deal with this problem, but ultimately being able to place based on this information will be added to the compiler when the ability to specify this information is added to MDL+. Also, as before, this design might be useful, but probably not as the architecture it was intended for.

We can also learn from this automatically-compiled design. First, the placement looks very much less structured and regular than the human placement. Once again, realizing the actual goals of the design, the machine was able to find a better solution when ignoring these human concerns. Of course, it was not as impressive this time,

since the human needed to place the chip while keeping the ALUs near each other, which might have on its own forced the more regular design.

Second, k-pairs have become a key point in this design. There are only three other BFUs with which the PC does not need to communicate, and it has great need for slots with which it can communicate. Furthermore, of the three BFUs it need not transmit to, all must communicate with several of the BFUs that the PC does transmit to. Thus, these three ALUs are perfect candidates for being in k-pairs with the PC, and in fact all three end up in k-pairs with the PC — and end up the only used k-pair slots for the PC, the other three that are on chip being left empty. Once these BFUs are placed, everything else falls into place.

Third, the design is roughly symmetrical. Now that the design has become more complex it is not as perfect a symmetry, and thus harder to see. Nonetheless, our intuition gleaned from simpler examples does pay off. The three ALUs are located as close to 120 degrees apart from each other as possible on this grid. The two to the right are at the closer k-pair slots, and the one at the left is at one of the farther k-pair slots. Considering the 8 possible k-pairs the PC could form on an infinite grid, there are two empty slots between the ALUs at (6,6) and (6,2), one empty slot between the ALUs at (6,2) and (3,3), and two empty slots between the ALUs at (3,3) and (6,6). This is as close as possible to evenly distributed.

Fourth, certain design motivations can be traded-off once the design is mostly set. For example, symmetry and k-pair concerns cause us to place the ALUs in their current locations. Then, once the design is mostly set we decided that the ALU at (3,3) should be moved to (4,2) to make the design more compact. This leaves it in a k-pair with the PC, but makes the design no longer symmetric. Although trading off symmetry for compactness at this stage is good, we recommend using the symmetry heuristic when designing and only worrying about compactness once the design is mostly set.

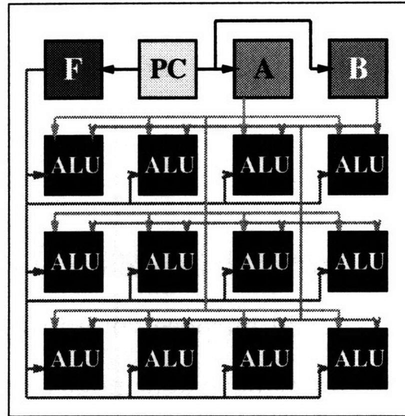

Figure 6-11: SIMD Processor on MATRIX

6.2.5 SIMD

Instead of the extra instruction caches, if more processing power is needed to run off of the single control, a SIMD (Single-Instruction, Multiple Data) parallel system might be created. Figure 6-11 shows an example of this. Further decisions might include whether the multiple data streams should arrive from off chip and be pipelined through MATRIX, or be cached on-chip in which case more BFUs will need to be allocated to being memories for the cache. In the SIMD case, the amount of processing power is only limited by the number of BFUs on the die, and MATRIX can accomplish much more computation than a processor would.

MDL+ has a very hard time placing this design, and fails to route it with only one-hop paths. Ultimately, if a large amount of processing power is desired, the best option might be to manually place the three I-stores, and several rebroadcasting BFUs each of which follows some convention, such as F onto l3_h1, A onto l3_h2, B onto l3_h3. Then, ALUs can be placed in any of the rebroadcasted rows (or columns) and could even double as the BFUs doing the rebroadcasting with their network ports. In fact, one BFU could serve as an ALU and do all three rebroadcasts, if it could dedicate one of its floating ports to this cause. The other option, is to have different ALUs rebroadcast each signal. For example, in the figure, the ALUs in the first column could each broadcast F to their rows, the third column could rebroadcast A

to their rows, and the fourth column could rebroadcast B to their rows. This method clearly scales to any size of chip.

6.2.6 ASIC

Finally, and perhaps most importantly, in addition to these general-purpose structures, it is possible to create special purpose computing engines. As was mentioned earlier, these kind of systems can achieve enormous speedups over a general-purpose system. And, because MATRIX doesn't need to be configured until run time, users do not need to spend the money or time developing custom silicon for these applications. A couple of simple applications were described as examples at the beginning of this chapter.

6.3 Other CGRAs

The final area of research using MDL+ that is available is changing the picture of the chip that the placer and router see in order to see how they would act when compiling for different backend chips. This should shed some light on how MATRIX should be changed and how CGRAs in general should be designed. While the experiments themselves are discussed here, the conclusions about MATRIX and CGRAs are discussed in the next section.

6.3.1 Eliminating Level-2 wires

This is the easiest possible area for research of this type, since we do not even need to change the MDL+ backend at all. The placer already placed so that level-1 wires would be used when possible, and it never ended up using up all of the level-3 resources. Thus, the few times that it used level-2 lines, they could be easily substituted with level-3 lines that were unused. So, everything that the compiler was able to place and route successfully it would still be able to place and route successfully.

However, the MDL+ compiler has also shown us that level-2 wires are not useless.

There have been several times when it has made use of level-2 wires, and since it does not do so unless it can not use level-1 wires, we know that without level-2 wires these connections would have to be made with level-3 wires, which are longer than the level-2 wires and thus some area on the chip would be wasted.

6.3.2 Unregistered Level-2 Wires

If level-2 wires were not registered then 2-hop and 3-hop level-2 paths would have the same latency as 1-hop level-2 paths. As long as level-2 paths are registered at the source port and the destination port, there is already plenty of time in the cycle for this and so it would only require a little more hardware design and area, but not slow down the rest of the chip. However, not registering a level-2 line at its source at all, and thus making level-2 lines as quick as level-1 lines would slow down the cycle considerably.

Nonetheless, we can change the backend of MDL+ to assume that 2-hop level-2 paths are traversable in one cycle. This makes very many BFUs accessible in one-cycle and not surprisingly makes almost anything routable, though this is not an incredible gain since almost every design we tried to compile had already been routable. Unfortunately, this would slow down the clock of the MATRIX chip too much to have the few extra routable designs outweigh the slow-down to all other designs.

6.3.3 No level-1 wires

In an attempt to confirm that bad design choices for MATRIX would not appear positive in our research environment, we tried changing the backend of MDL+ so that it would compile to an architecture just like MATRIX except without any level-1 lines.

The most noticeable effect is that it becomes hard to route any significant number of units to each other unless they are all in the same row or column (since they must use level-2 and level-3 lines which all connect BFUs that are in the same row or

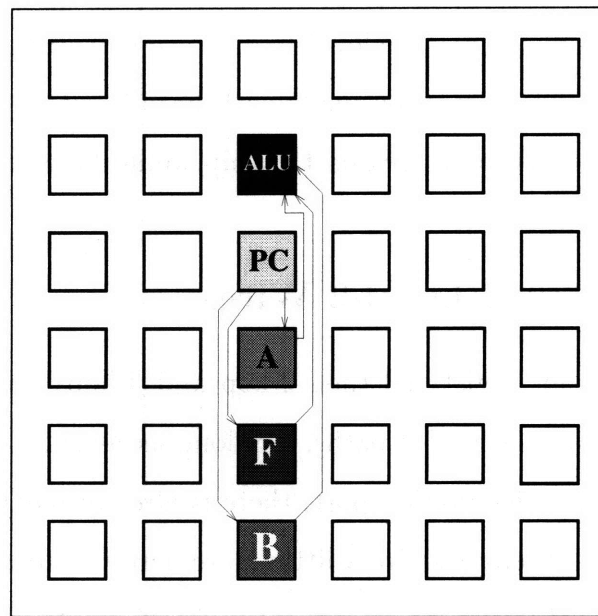

Figure 6-12: 8-bit microprocessor routed without level-1 lines

column. Thus, when we run our favorite 8-bit microprocessor example through the new version of the compiler it turns out as in figure 6-12. Note that the PC is next to the ALU even though they do not communicate with each other, because being next to another BFU is no longer any more useful than being within a distance of three or four.

MATRIX would not be as useful if every unit on a chip needed to be on the same row or column, thus we were correct that it would be ridiculous to eliminate level-1 lines, and our method of research did not fail this test.

6.3.4 No diagonal lines

Even though it is unreasonable to think of eliminating all level-1 lines, it has been suggested that we should eliminate the diagonal level-1 lines and only leave the level-1 lines in the cardinal directions. We implemented this change to the backend of MDL+ and recompiled the 8-bit microprocessor example yet again.

Once again, upon running the 8-bit microprocessor example we got a result of BFUs that "lined up." This time, they lined up in a row instead of a column, as

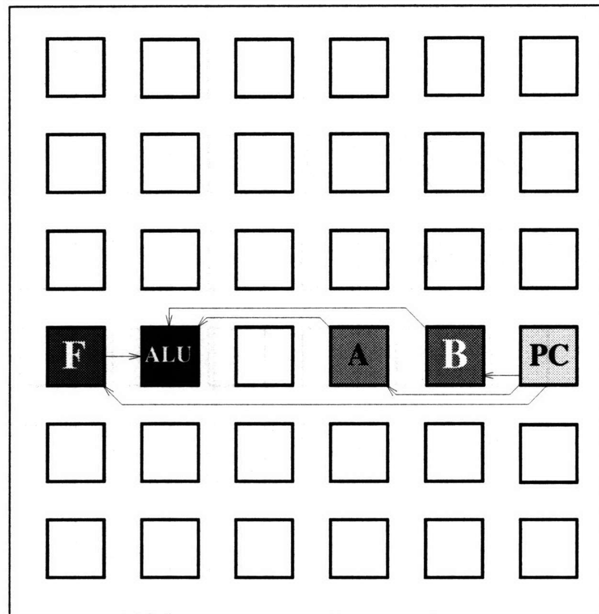

Figure 6-13: 8-bit microprocessor routed without diagonal level-1 wires

shown in figure 6-13. Due to the existence of short level-1 wires the PC and ALU were divided, and each is within level-1 range (2 slots) of two of the three I-stores.

While this design can be improved (by moving the PC to slot (3,3)), it would still not be able to use only level-1 lines. Even though we have not completed a large enough survey to be able to conclusively decide whether diagonal level-1 lines are useful, it seems that they are very useful, even in some of the simplest designs.

6.3.5 No length-2 lines

Besides suggesting that all diagonal level-1 lines be eliminated, another popular suggestion has been to eliminate all length-2 level-1 lines. If both diagonal and length-2 level-1 lines were eliminated, then the cycle length on MATRIX could be shortened. Although eliminating only length-2 lines would not cause as large an advantage, it still would decrease the size of a BFU, thus we examine the possibility.

We have already seen that length-2 level-1 lines have often been used by the compiler and seem to have enabled certain chips to be routed when they might not have otherwise been routable. Nonetheless, we went ahead and altered the backend

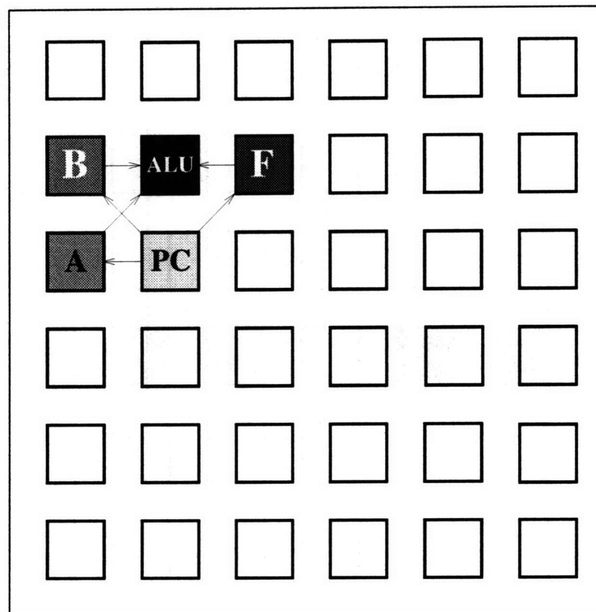

Figure 6-14: 8-bit microprocessor routed without length-2 level-1 wires

of MDL+ to eliminate length-2 level-1 lines. Since several of our previous 8-bit microprocessor designs have relied on length-2 level-1 lines, we started by recompiling that example, with the results shown in figure 6-14.

The compiler managed to find a new way that we have not seen yet to place the microprocessor into a 2x3 grid while using only length-1 level-1 lines to route. While this is a positive result, routing the 8-bit microprocessor with only length-1 level-1 wires is not very difficult. Thus, to test whether the compiler would be able to place and route effectively without length-2 level-1 lines, we tried running this modified MDL+ on the VLIW example of figure 6-9. The unmodified MDL+ compiler is able to place and route this design using only level-1 lines, but it seems to be at the limit of its capabilities and it does use several length-2 level-1 lines, as shown in figure 6-10, thus this seems like a great test for the case of eliminating length-2 lines. If the VLIW architecture could still be routed, then length-2 lines must not be as important as we had assumed. Expecting an answer to be strong in that the design either would or would not be routable, we found an inconclusive answer, as shown in figure 6-15.

The first thing we can learn from this figure is that the absence of length-2 level-1

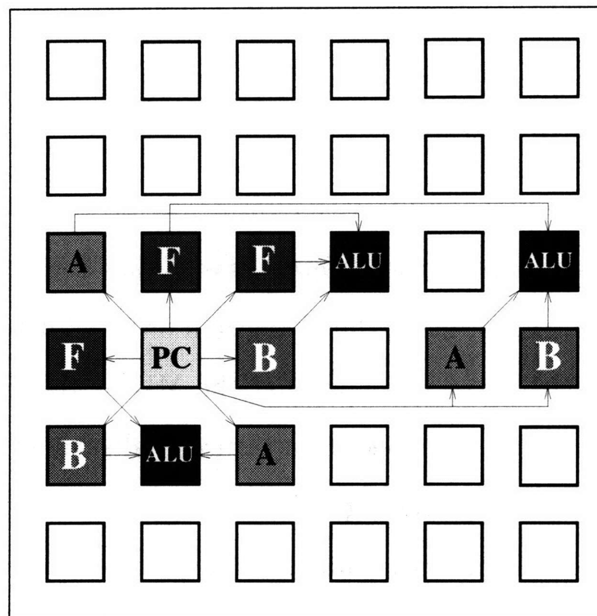

Figure 6-15: VLIW architecture routed without length-2 level-1 wires

lines definitely made the situation more difficult for the compiler, as it did not find an entirely-level-1 solution, and did not come up with the optimal solution. While placing and routing this design would be hard for a human, it is not hard to notice from the compiler's solution that some changes can be made to improve the design.

While these changes would be difficult for the machine to consider at once, it is easy for a human to notice them. To start with, move the three BFUs in the final two columns each one to the left or west. The PC still needs a level-3 line to reach those A and B I-stores, and the F I-store still needs a level-2 line to reach the ALU. The difference is that now the design is only five columns wide, and so the entire design can be shifted over one column to the right or east. Shifting the entire design this way does not effect level-1 lines, but it does mean that BFUs which used to be able to use level-2 lines for some direction can not, and those that could not now can. Since there were four non-level-1 lines of which three were only routable with level-3 wires, there are now 4 of which only one requires a level-3 wire.

Therefore, the designs that are barely able to use only level-1 lines on MATRIX rely on length-2 level-1 lines, but they would be able to be automatically placed and

routed without length-2 lines while only using a small number level-2 and level-3 lines.

6.3.6 Other Experiments

We have not done other experiments with altering the backend of the MDL+ compiler, because they would be more subtle differences and thus require a much larger set of applications to compile down using the changes in order to determine the true effects of making those changes. However, now that the MDL+ tool is available, and easily configurable in this manner, we expect that future studies might be done to determine where the optimal point is for trading off more lines on one level with less lines on a different level, as well as other experiments that we have not even thought of yet.

6.4 Questions about MATRIX

This chapter concludes by discussing the answers that our research has led us to with regard to the questions asked about the architecture in chapter 3. Once again, we can only give discussion about these issues and not definitive answers.

6.4.1 Level-2 lines

There are two questions dealing with level-2 lines. First, are they useful or would it be better to replace them with more level-1 or level-3 lines? And second, assuming they remain will it be advisable to allow the option of not registering them or was that wasted functionality?

It is hard to answer the first question. As mentioned in the last section, in all of the examples that we have run level-2 wires have been neither necessary nor useless. However, we did not implement any examples that taxed the amount of resources in MATRIX, and thus more level-1 or level-3 lines would be neither useless nor necessary as well.

However, we have come up with dense designs, and even with our somewhat dense designs they did not run out of wires to communicate on. Thus, it is our conclusion

that level-2 lines can be eliminated and not replaced at all. This would start to put a burden on the level-3 lines, and would start causing us to see a greater percentage of the wires used. As long as we are correct and the amount of useful functionality is not decreased, this would be a great tradeoff if it would release enough area to allow for more BFUs on the chip. While none of our designs ran out of room on the chip, it is easy to make such designs, even by tiling the old smaller designs.

We can not be sure how much more area would be freed up by eliminating the level-2 lines, but it might be considerable. It includes 8 wires over each BFU on the grid, as well as 8 of 30 inputs to each of 8 ports on each of the BFUs on the grid. It also reduces the size of some of the perimeter of the chip.

It might not even matter if the number of BFUs on the chip increased, since the largest benefit of eliminating the level-2 lines might be the decrease in complexity of MATRIX. Even though MATRIX is a first implementation, it has the second system effect [Bro95].[1] MATRIX is very complex, and thus harder to design, understand, and use. Eliminating level-2 lines would remove the need to view a MATRIX grid as a checkerboard pattern, and make all designs translatable by any number of rows or columns. It would also remove the need for trying to figure out what capabilities each BFU had (and what "d1" and "d2" mean) since all BFUs would have the same abilities.

Given that we recommend eliminating the level-2 lines, it seems less important to answer the question of whether unregistering level-2 lines would help. However, the answer is clearly that it would help. In fact, it might help so much that level-2 lines would be worth keeping.

Therefore, if many MDL+ designs had previously proved unroutable we would suggest keeping level-2 lines around if they could be opted unregistered. However, since most MDL+ designs are already routable and thus the great benefit of unregistered level-2 lines not adding much real functionality to MATRIX, we continue to recommend the removal of level-2 lines from MATRIX and CGRAs in general, even if

[1]This was intentionally designed into the system so that useless features could be eliminated in a streamlined second system.

they are unregistered.

Still, if the level-2 lines were going to be unregistered including being unregistered at their source port then they would essentially become length-4 level-1 lines, and would aid in our placement and routing to the point that we might actually recommend keeping level-2 lines in CGRAs. However, as we know that such a system is not needed much, and would greatly increase the length of the critical path in a MATRIX cycle, we still do not recommend keeping level-2 lines, even on this basis.

6.4.2 Level-1 lines

In the previous section we examined the possibility of compiling to a MATRIX chip that had no diagonal level-1 lines, and it seemed like a very bad idea. Thus, we recommend keeping the diagonal level-1 lines. We believe that their great benefit lies in the ability to create larger completely-connected and mostly-connected subgrids of a MATRIX layout.

Length-2 level-1 lines, on the other hand, gave us only a slight improvement. The elimination of length-2 lines, even in the worst case of a barely routable chip, only required use of a few level-2 and level-3 lines. However, it seems that manually placing and routing for such a chip would be less intuitive than for a MATRIX chip, since being two spaces away is now equivalent to being four or five spaces away in certain directions.

Therefore, we recommend eliminating length-2 level-1 lines from MATRIX and CGRAs in general only if the decrease in area (due to the absence of the lines and the smaller switches at the ports) is sufficient to add another row and column of BFUs. In other words, it is not very bad to lose these lines, but it is bad. Thus, we would suggest moving from a 6x6 grid of BFUs to a 7x7 grid of BFUs at the expense of the length-2 lines, but not eliminating them for only small gains such as the gains that would be sufficient to convince us to eliminate the level-2 lines.

In the event that both length-2 level-1 lines and level-2 lines were eliminated, as per our suggestions, there would be no intermediate between length-1 lines and global level-3 lines, and more level-3 lines might need to be added. Thus, in the event

that level-2 lines were already eliminated, we would suggest studying the system carefully before eliminating the length-2 level-1 lines, since their importance may have increased.

6.4.3 Inputs to BFU ports

Here we must ask ourselves whether each BFU port really needs all 30 inputs to it, or if that number can be reduced. The easiest answer, given that we have recommended eliminating either level-2 or length-2 level-1 wires, is that by committing either of those actions the BFU ports would be reduced in size. We have seen the local input to be very useful in some designs, as well as the level-1 lines (besides possibly the length-2 level-1 lines). Thus, we do not recommend the elimination of any of these.

The only other remaining possibilities for elimination are the level-3 lines. One way to reduce the number of level-3 lines entering each BFU port is to have each BFU only be able to read 2 of the level-3 lines in each direction (in a checkerboard pattern) or having half of the BFU's ports able to read half of the level-3 lines while the other half of the ports can read the other lines. Any of these options seem to raise the complexity level of the MATRIX grid a lot, and we therefore do not support them, in an attempt to keep the grid as uniform as possible. However, given an improvement in the automatic phases of the MDL+ compiler, and other CGRA programming tools, it might become easier to think about and program devices that are complex in these ways. Therefore, further study in these areas might be warranted, even though we do not suggest adopting them at this time.

6.4.4 OR Plane

The OR plane was eliminated in the initial design of MATRIX, but the question remains as to whether it should be reinstated and whether the chip lost too much functionality with that decision. While it is very useful, in theory, to be able to construct a PLA on the chip, such constructions had very long latencies, and were hard to think about while programming, and thus were not used much in practice.

Removing the OR plane from the design of MATRIX reduced the time to design the chip, the time to test the chip, and the area of the chip, all by significant amounts. Thus, given that we have several other means of control on the chip (including small constructed microprocessors), we do not recommend adding an OR plane to CGRAs in the future. Basically, through many designs we never desired an OR plane of the type that would be available, and the time and area are thus not worth the effort.

6.4.5 Control

In general, the control options on a MATRIX chip are poor. Thus, in chapter 3 we promised to consider which changes would be appropriate to the compare/reduce systems and whether the control system should be entirely redone. While it is clear to any MDL+ programmer or other user of MATRIX that the control system should be entirely redone, we found that it was usable through these examples. It seems that, as with other things on MATRIX, the difference will end up being decided by the amount of automatic assistance in MDL+ and the programming environment.

If a high-level MDL++ compiler is available, and it has found sufficient ways to compile down to the MATRIX control system, then at that point the control system will not need to be redone. However, it seems that this scenario is unlikely, as the current MATRIX control system leaves much to be desired. Therefore, we still recommend an entire overhaul of the control system for any future version of MATRIX or other CGRA.

We do not describe possible alternatives here for two reasons. First, it is a complex question and beyond the scope of this project. It is sufficiently complex that the MATRIX project went on for a couple of years before anything was done about this problem. Second, something has since been done about this problem. The MATRIX architects have devised a proprietary new control system, thus any quick suggestions here would not be able to contribute much.

6.4.6 Other Questions

By no means have we addressed all of the questions about the MATRIX architecture which can be asked. Many more questions can also be discussed by using the MDL+ compiler as a tool, and yet more questions can be discussed by extending the MDL+ compiler with more functionality. Hopefully, this work has paved the way for much more analysis which will now be easier to conduct.

Chapter 7

Conclusion

We have covered a lot: The motivation for MDL+ was given in chapters 1 and 2, the hardware MDL+ compiles to was described in chapter 3, the language and features of the compiler were described in chapters 4 and 5, and what we learned from MDL+ was discussed in chapter 6. In this chapter, we will start by discussing the improvements that MDL+ represents over the previous state of the art. Then we will summarize what we have learned with this thesis. We will conclude by suggesting future work on MDL+ and related software and hardware.

7.1 Improvements from MDL

This section will discuss the improvements that MDL+ presents over the original MDL. They are divided up into three sections: Improvements to the code that generates the compiler, improvements to the grammar that is accepted by the compiler, and improvements to the output and error messages generated by the compiler.

7.1.1 Code Generating MDL+

A major advantage of MDL+ over MDL, even though it is the one that will be least noticed by users, is that the code generating the executable is much improved. It will hopefully crash less, be needing corrections less, and be easier to correct and improve.

The major change in order to improve the code that generates MDL+ is that it is

all written in high-level object-oriented C++ instead of C. For MDL+, flex and bison are still used to automatically generate code, and bison still generates C code (which is now compiled with a C++ compiler), but all of the other code used for MDL+ is in the form of one top-level procedure, a handful of helper procedures, and many C++ objects developed specially for MDL+. Some of these C++ classes are generated by Perl scripts written specially for generating MDL+ classes.

The basic advantages of reliability and extensibility of C++ over C should be obvious, but the greatest advantages come because MDL+ was written using the high-level object-oriented features of C++ such as extreme modularity and abstraction. Software hacks were not allowed to cross over between objects, as opposed to MDL which had many really bad hacks of the type that might be expected in a C program. For example, some bugs found long after the release of MDL were caused by some C code accessing a piece of a data object (a C struct) by accessing the memory a certain number of bytes displaced from the object pointer (an incorrect number of bytes). With MDL+, these kinds of errors will not creep up.

Even when bugs are inevitably found in MDL+, they should be easier to track down and correct than they were for MDL, because the code is written better. The improvement in the quality of the code is a direct result of the motivations going into this project. MDL was intended as a quick hack. Indeed, the MDL code itself warns "It really sucks - I'm ashamed I havn't deleted it off the face of the earth...but that's what I get for trying to hack up a quick solution!" in a comment that was dated more than half a year before work began on MDL+.[1]

Being a quick hack, the MDL code was not organized in an easy to read or reason about fashion or an easy to modify fashion. It was not written modularly in an object-oriented style, and it was not written with good error reporting. MDL+ has been designed from the start to be code that will last for a long time[2] and be easy to understand and alter. MDL+ has been written in an abstraction-oriented

[1]There's a lesson to be learned here. As the adage goes, if it's important enough to do, it's important enough to do well.

[2]The irony is that the ARPA contract is running out soon, and MDL+ will in practice receive much less use than MDL did.

style with verbose and useful errors reported both in the event that the user inputs something unexpected and in the event that the program itself acts in an improper and unforeseen way.

Finally, the MDL+ code will be easier to upkeep than the MDL code because of its superior documentation. The only specification of MDL was [Esl95], a document which contained a sparsely explained (at points) description of an old temporary version of MDL. This document differed from the actual functionality provided by the MDL executable in many ways, and ultimately some of the most annoying errors in it were details such as misspecifying aspects of the grammar, and listing keywords for language constructs which were not the actual keywords used by the executable. [Esl95] promised an appendix with the complete grammar of MDL, but ultimately the only way to get this necessary information was to look at the hard to understand yacc file.

MDL+ instead has a plethora of documentation, most likely too much. There is high level information in the form of what is basically a programmer's manual (chapter 4 of this thesis), the complete grammar (appendix E), and the yacc file itself is more commented and thus more readable. The programmer's manual and the grammar, as well as the list of known bugs and the man pages (like typical UNIX man pages, very different from a programmer's manual, and provided in appendix B), should be up to date.

Furthermore, when alterations are made to the code they will probably not result in emergent errors which are only later observed and thus hard to track down. This benefit is due to the regression test suite that has been set up to test MDL+'s parsing. These tests, as they have been built up over the project, have been run almost every time the MDL+ executable has been recompiled, and while they are not nearly as complete as regression tests should be, they should provide a certain level of confidence that errors have not been injected when the code is altered.

With much more attention to detail, and a perception that this project was supposed to last and be extensible and correctable, MDL+ has developed much better code than MDL had.

7.1.2 MDL+'s Grammar

The actual grammar, or syntax, of MDL+ is a dramatic improvement over that of MDL, and thus the source code is greatly cleaned up. The best possible evidence of this is the code in appendix D where hard to read MDL code and easy to read MDL+ code with the same meaning are listed. In this section, we will discuss some of the reasons why the MDL+ code looks so much better, and why it is easier to write, use, and read.

Shorter Files

As has been alluded to at many points in this thesis, one of the biggest advantages of MDL+ over MDL is that the code is shorter. There are several causes of this decreased length:

- **Don't Cares** - A late patch of MDL included Don't Cares, but they were not often actually used. One reason was that the code was in a legacy state already, but the more important reasons were that they were only usable in certain circumstances, and even in those circumstances they were quickly-developed functionality and thus quite buggy. They were intended, when added to MDL, as means of not overwriting all parental fields when a BFU inherited from another BFU, not as general Don't Cares as they exist in MDL+. In their new incarnation, Don't Cares enable a programmer to not specify details that he does not care about, and allow the compiler to use this increased freedom when compiling, and to in turn not generate more output code that it does not care about.

- **Omissions** - Besides being able to substitute some real values with DCs, MDL+ allows almost all arguments to be omitted. A BFU's configuration provides a lot of functionality, and most BFUs are simply not going to make use of most of this, thus file sizes are greatly decreased. For example, most BFUs will not be transmitting on all 10 possible level-2 and level-3 lines.

- **Inheritance** - As with Don't Cares, MDL implemented BFU inheritance at a late date and then very quickly, as a hack upon a hack. For the same reasons as with DCs, programmers did not use this inheritance. The additional reason that people did not use BFU inheritance was because a child still needed to have all of its fields specified, the difference being that DC values would not override parental values. This meant that MDL's inheritance both did not decrease the code length by much and was much less useful. With MDL+, inheritance means that BFU functionality that is common to several BFUs can be abstracted, as with the I-stores in chapter 5. This makes the code shorter and more natural. And in MDL+ there is a very rich version of inheritance which includes what we have referred to as reverse-inheritance.

As there are many causes of the decrease in length, there are also many benefits derived from the decrease in code length:

- **Writable** - It is much easier to write MDL+ files, since all that is required is figuring out which functionality is desired on the MATRIX chip and then figuring out how to explain that functionality in MDL+. With MDL, the programmer also had to determine which values he would write for the fields he did not care about, and what those fields were. It is also easier to abstract code with the combination of omissions and inheritance.

- **Short Output** - The MDL compiler outputted the entire configuration data of a chip, the MDL+ compiler outputs the configuration data that was requested by the source code.

- **Readable** - Viewing the source file no longer requires sifting through all of the randomly chosen values to find the ones that the programmer actually cared about, since only the relevant ones are printed. Additionally, more relevant data ends up on each page, making it easier to view at once. Many questions (such as "Did he want that BFU at (1,1) or did he just need it on the chip?") are no longer evoked by reading the source files.

- **Modifiable** - Since it is easier to understand the code, and easier to write it, it is thus easier to modify it.

Although shortening the source code is a major advantage of using MDL+ over MDL, it is far from the only advantage. We will summarize some of the other benefits of MDL+'s syntax.

One major advantage comes from the syntax changes associated with the intelligent phases of the MDL+ compiler (discussed in chapter 5) and even the MDL+ driver (section 4.3). Each of these four phases reduces the complexity of trying to read an MDL+ source file. Even when the code does not get shorter, it does reduce the amount of "noise" data. For example, if the programmer cares that one wire is driven but not another, he can merely use the first and ignore the second without specifying BFU Power constructs. The person that reads his file can then assume that used wires will be driven and other will not be driven, not needing to read through extra parts of code.

Similarly, if the programmer wants two BFUs to communicate but does not care which wire they use, he can "write" this in MDL+, instead of specifying which line they use. And it is much easier to read a file as "There is a memory unit and it gets its address from this ALU unit" than reading a file with many completely-specified BFUs that happens to implement the same functionality.

Besides the multiple input modes (MDL+1.0, MDL+1.2, etc.) thus allowing multi-level programming, the syntax is also cleaned up by certain command line options to the executable. Being able to use the options to the driver (-S/-F) means that a single source file can be used to compile an algorithm both for simulations and for actual fabricated chips. Without these options, two source files would be needed — one with minimal length not specifying any wires disabled, and one with all wires explicitly configured as driven or not driven. While this is not a change to the grammar of MDL+, it is an improvement that affects the MDL+ source code.

Similarly, the -U option implements the first "sureness" functionality of the MDL+ compiler. In general, the idea of sureness values is that besides just specifying, say, a BFU to be placed somewhere or to not place it anywhere in particular, the program-

mer could place it in a specific location *with a certain amount of conviction*, which is to say a certain number that vaguely corresponds to the probability that the compiler will follow his suggestion. What the -U option provides is not yet implementing sureness values, but it does give the programmer the option of using the same exact source code file as either having some BFUs explicitly placed or not, which can be seen as corresponding to MDL+ assuming default surenesses on placement to be either 100% or 0%. Once again, this is not a change in the grammar, but it does affect the source code.

A change that is actually a change to the grammar, and makes programming much easier, is the small extensions to the basic MDL+ kernel, aspects of the grammar which essentially desugar into MDL. For example, there are many places in MDL where language structures require not only all fields to be defined, but for all of them to be defined in order. MDL+ generally allows any order and any omissions, though there are exceptions. This eliminates many careless syntax errors in the code, which in the case of MDL were actually very difficult to track down.

Another improvement is the added datatypes available with MDL+, not just the objects available with the grouper but the explicit control over each of the six pieces of BFU configuration, the Port construct, etc. These constructs make more abstraction possible in the MDL+ source code, since BFUs can then share these constructs, without wasting all of the memory that would be required to inherit from an entire other BFU. Additionally, if one of these objects changes a lot then its definition can be placed separately from the rest of the larger object, perhaps at the beginning of the file with other frequently changed data.

As has been mentioned, one of the most annoying aspects about MDL could be that the keywords were often different than they had been specified. While this problem in particular has been solved by specifying the language as it is implemented, a related problem exists of the user using a wrong word. For example, a user of MDL might specify a level-3 line to be driven by "thisside" when he meant "thisio". To alleviate this problem, there are many more locally reserved words in MDL+ (but not globally reserved words), so that in this example, while "thisio" is still accepted,

"thisside" is accepted as well.

Another similar problem comes about because the globally reserved words are case-sensitive, and MDL had an ad hoc method of deciding which letters would be capitalized, so that the user would need to try each possibility to be sure. (There was no accurate manual to consult, but there was the flex file.) There were often multiple ways to specify the word, but even then many were capitalized in an unpredictable manner. MDL+ globally reserved words, on the other hand, were designed with consideration of a general convention: It should always be all right to completely capitalize any of these words, to leave only the first letter of each sub-word capitalized, and to leave the entire word lower-case. Sometime, where it might not be clear what the middle option means, it might be necessary to consult the complete list of ways to specify each globally reserved word in appendix C.

Finally, a feature which should soon be added to the MDL+ syntax is the ability to output chip descriptions to global context 1. Each MATRIX chip has two programmable global contexts. MDL always outputs configuration information for global context 0, and thus can never be used to produce configurations for global context 1. In real situations, this can be devastating, vastly decreasing the power of a MATRIX chip by increasing its context switching time which can be seen as its major-instruction issue rate. The MDL+ compiler has been designed to output chip descriptions in either global context. In fact the code generation routine that actually prints the configuration information to the output file takes an argument of which global context to print out writes for. This argument is hardcoded to 0, but could easily be passed as 1 when the user desires it given a small change to the MDL+ syntax. As with all of the other changes to the syntax, with this change we would achieve a large improvement to the MDL+ source code and compiler.

7.1.3 Output and Error Messages

Last, but not least, the MDL+ compiler is much better than the MDL compiler because of the interfaces it provides for the user. The command line options provide the user with much useful power, the output is much clearer and more concise, and

the errors are far more explanative.

User Options and Output

For each of the intermediate languages of MDL+ (1.0,1.1,1.2,1.3) there is an input mode and an output mode. By input modes, we mean that the source code may contain elements from any of these languages. The existence of output modes is much more explicit: Using the appropriate command line argument the user can get the MDL+ compiler to output its configuration information for each chip in any of the MDL+ intermediate languages. This realizes the multi-level programming objective that we set out to possess in chapter 2.

To further achieve our human interaction objective, beyond just enabling the user to access the data before and after the driver, router, placer, and grouper act, there are other command line arguments that affect particular phases. For example, there are the -S and -F options discussed in the previous section for the driving phase. More useful for multi-level programming are the -Y and -Z options to the placement phase which toggle whether the placer will act randomly or deterministically. These options can be very useful for a user interacting with MDL+ when combined with the -U option (see appendix B).

Having five normal output modes (and a sixth output mode for debugging), enables MDL+ to output data in any form that the user might desire it. In addition, MDL+ offers the user control over which automatic phases are run on each input file, starting from any phase and then continuing all the way down through the lowest-level phase. Yet, MDL+'s advantage when it comes to output is not limited to how it gets to the point when it outputs data and at which points it will output data — it also provides a great improvement in the actual form of the data it outputs.

MDL never outputted high-level MDL code, so there is no comparison possible here, but MDL+ code that is outputted by the MDL+ compiler is formatted well in that it is pretty-printed and only includes the specified configuration bits. Whatever configuration data is left unspecified is omitted wherever the MDL+ grammar allows it to be omitted, and listed as DC in the few remaining places. Thus, while lacking

the abstraction and hierarchy built into human programs, MDL+ output from the MDL+ compiler (at any of the intermediate stages) is still much more readable and easy to understand than human-generated MDL code ever was.

Even the basic verilog output mode of the MDL+ compiler is far superior to the output mode of the MDL compiler. The verilog output now consists of a list (one per line) of verilog procedure calls which will write either configuration memory or main memory to a MATRIX chip. The MDL compiler had outputted a list (one per line) of numbers that included the addresses and data for memory writes, and information to specify whether it was intended for a configuration memory write or a main memory write, but all of this needed to be converted by another program to the form of the MDL+ output files. Thus, MDL+ has removed the necessity of using that extra program while making the file more readable. (Configuration writes actually say the word "Config" in them, address and data are separated, etc.)

Of course, there are several other benefits to the MDL+ verilog output besides the basic format. The verilog output code is now shorter, more correct, and much more readable and thus debugable.

The verilog files are now shorter — often in total byte-length, always in number of lines and thus number of cycles that it takes to load into an actual MATRIX chip. They are shorter because the MDL+ compiler outputs only the configuration bytes that the user cares about, whereas the MDL compiler specified configuration and main memory writes for every memory byte on the MATRIX chip. Besides taking up less space and requiring much less time to load into a MATRIX simulation (and an actual MATRIX chip) the advantage behind this is that the code is more readable.

Furthermore, recall that we set out in chapter 1 to create a general purpose computing device that had a shorter instruction word than FPGAs. While this has been considered a benefit of MATRIX and CGRAs for a long time, this benefit has only been realized with MDL+. Now, the instruction word is a lot shorter and only as long as it takes to specify the configuration that the programmer cares about, as opposed to FPGAs which generally must have their entire configuration loaded even when only a small subset is relevant to the programmer.

In fact, had the MDL compiler possessed this quality it would have greatly affected the work that the MATRIX design team did to test the verilog model of the chip, and to write tests for the fabricated chip itself. For the simulation, a complete global context being loaded onto the chip takes about 20 minutes to half an hour to run on a SPARC 20 with 132-196 megabytes of RAM. This is a long time, especially since it is required upon starting up any test of MATRIX, and always needs to occur before even the first test involving that configuration has run. With a very long list of regression tests to be run on the verilog model of MATRIX we felt a need to avoid this cost.

Some people proposed elaborate systems to examine each MDL file and compare it to the previously loaded MDL file and a user-specified list of unpredictable events that might happen when the previous MDL code runs, with the elaborate system then producing a new MDL output file that was the subset of the file that actually needed to be loaded in. Instead of this project, in practice the tests were designed by writing a lot of code to generate the MATRIX configuration writes, re-implementing the abstraction barriers in MDL, and going to great lengths and efforts to avoid MDL in order to produce tests that would be able to run in a reasonable amount of time. With MDL+, none of this effort would have been necessary, and the MATRIX testing effort would likely have required less time to implement and been superior in its correctness and ability to find errors in the hardware design.

Not only are the verilog output files more concise, but they are most likely more accurate and correct. MDL output was periodically found to have bugs in it, but the code generation in MDL+ has been debugged extensively and is written in a very easy to understand and modify manner. The code has been checked by multiple people, been used several times outside of the compiler and had its output checked separately, and then inserted into the compiler and had its output checked again.

Of course, we do not mean to suggest that there will be no code generation errors or bugs in MDL+. But, even when bugs are found, they will be easier to debug and fix. While there are many reasons contributing to the ease of debugging which have been mentioned, the most important aspect is that the verilog output is commented.

```
% grep "Config" Micro8-mdl+.v | grep "16'h23" | grep -i "drive l1"
WriteConfig(16'h231c, 8'h0, 1'b1);// Drive L1 - 4 unused, N,E,S,W
% grep "Config" Micro8-mdl+.v | grep "16'h22" | grep -i "msb"
WriteConfig(16'h220c, 8'hff, 1'b1);// Carry Byte - leftsrc, rightsrc, MSB, L
```

Figure 7-1: Examining an MDL+ verilog output file

Each line of the verilog output from MDL+ is a memory write to a MATRIX chip. After the call to the verilog procedure which performs the write, the compiler outputs the verilog comment characters and then prints a comment on the meaning of the particular configuration byte written. These meanings are based only on the address, but the non-obvious ones explain what the various data values would indicate.

This enables a user to easily read the verilog output and figure out what it is doing. If a line is incorrect, or there when it should not be he can look at the code-generator C++ code and grep through for the comment, quickly tracking down the bug. Of course, the most difficult part of tracking the bug down in the past (with MDL and while working on MDL++) has been determining which configuration write was incorrect. For example, even after determining that an incorrect value is on a wire, the user still needs to determine why the configuration made that possible. This is exactly where it assists to have the comments in the verilog file. Where it used to take a long time to track down specific facts about the output file, it is now one command-line grep away, as shown in figure 7-1.

In this figure, we examine the verilog output from MDL+ running on one of the versions of the 8-Bit microprocessor. Since we had named the chip in that case "micro8", MDL+ put the output in a file named "Micro8-mdl+.v". This filename specifies "mdl+" so as not to be confused with MDL output, ".v" so that we can tell the verilog output from the MDL+ output modes, and it capitalizes the filename so that a simple convention of always naming source files starting with lowercase letters will ensure that source code is never overwritten by MDL+'s output.

Both lines of this example start by searching through the file for configuration memory writes (including "Config") that were for a specific BFU (addresses are the

only 16 bit fields, and the high byte is made up of two hexadecimal characters that each specify a BFU or VBFU by row-then-column notation). The final search limits the returned lines to those with a certain explanation in their comment portion. Thus, the first search is to determine whether BFU (3,2) is driving its level-1 lines, and the second search is to determine whether BFU (2,2) has been configured to be the MSB for wide-word operations. From this figure it should be clear how easy it is to get information even from the lowest level of MDL+ output, an incredible leap forward from MDL.

Error Reporting

There are too many different parse errors that MDL+ reports for us to attempt to give specific details of them all here. Besides, the important point is not what help they give, but that they do give help to the MDL+ programmer.

MDL's error checking and reporting is described completely in five lines from [Esl95] which end by warning the user to "be careful!" While there are several situations which will prompt MDL to halt with some other error message, in practice most people go through months of MDL programming without receiving any error other than "Parse error: Parse error" which is not very helpful. Those error messages that do specify more will sometimes explain which procedure of the MDL code received an incorrect value, instead of explaining what that means in terms of MDL or the user. Finally, there have been some basic parse errors that have caused the MDL compiler to segmentation fault and core dump, although all such known instances have been patched.

MDL+, while its error reporting of basic syntax errors could be greatly improved, does explain many more parse errors. There are still designs that are so syntacticly wrong that the compiler will merely terminate with "Parse Error: syntax error", but in practice a vast majority of the errors detected leave long explanatory error messages. And these explanations are always addressed to the MDL+ programmer who need not be a C programmer.

Furthermore, MDL+ has not been known to segmentation fault due to bad syntax

on the part of the MDL+ programmer. Even if something does go wrong with the MDL+ executable (unforeseen bugs must be in there), they will likely be easier to debug than MDL bugs were, since most unexpected events are actually checked for by the executable, causing it to immediately halt and display an explanative error message about what was detected at which point. These error messages are intended for the C++ programmer who should then have the bug in the executable completely diagnosed and be ready to attempt to track the cause down and fix the problem. While these messages should never be seen by the user, it is worth noting that MDL+ has been built with several strong independent layers of error reporting.

There are also some errors for the various phases of the compiler but these are not as well developed, and it is hard to compare them with MDL, since MDL did not have the automatic phases, let alone the error messages. Thus, we will not discuss those here.

In conclusion, the MDL+ compiler provides much more functionality, is easier to use, is better explained and specified, uses and produces code that is shorter and more user-friendly, and is written with code that is much more understandable, debugable, and extensible. Whereas MDL code warns "Don't even think about trying to read or understand this code..." and suggests that it is bad because it was a quick hack, MDL+ code was designed from the start to be robust and extensible. These are all major improvements of MDL+ over the previous state of the art.

7.2 New Insights

This section will summarize, from the last couple chapters, the insights that we have gleaned from designing and using MDL+. We will suggest heuristics for manual placement, ideas of how to best use the MDL+ compiler, and opinions on the types of designs that are more successful on MATRIX. Throughout this section we will emphasize the useful heuristics that we have learned while discussing these insights. Suggestions for future work will be reserved for the next section.

7.2.1 Manual Placement

Through chapter 6 we have analyzed many different automatic placements, and discussed the ways in which we could learn from the ways the compiler treated those designs. The first thing that we learned is that there are multiple ways of using the placer:

- When working on manually placing a design, MDL+ can be used to suggest several alternatives, eventually leading the programmer to a design of his own.
- The programmer can leave all of the work for the placer, since it will likely do a decent job, and then if necessary clean up the automatically compiled design after it is done.

Heuristic #1: *Let the compiler do as much work as possible.* While both of these are legitimate alternatives, and any programmer is likely to pick and choose between them at different times, we recommend relying heavily on the compiler. There will be many times when the designs will need a lot of work, but they will often be sufficient or need very little touching-up and thus the compiler can save the programmer from a lot of tedious work.

Heuristic #2: *Level-1 lines should be used whenever possible.* Level-1 lines are not registered, while all other lines are. Level-1 lines are equally available to all BFUs thus allowing designs with only level-1 lines to be translated freely around the layout grid. Since level-1 lines around a BFU are rotationally symmetric, designs with only level-1 lines can also be rotated freely on the layout. Level-1 lines emit the least heat of any lines, as soon as some are used several more must be driven and so they should be used as well, and level-1 lines are not depletable resources. Since level-1 lines are short-range, using them will encourage designs to stay compact. Especially in the initial design, other lines should not be used when a level-1 line is available.

Heuristic #3: *Use symmetry when initially placing.* All of the best designs seem to have some sort of natural symmetry in them, and the symmetry is always explainable in terms of similar units each getting to be placed in similar circumstances.

In many disciplines optimal solutions to similar problems result in symmetric designs, and it will usually help to look out for symmetry even if the search only gets as far as a better understanding of the needs of the units in a design and which ones are in similar situations. While symmetry should be a major concern early in the design process, it should cease to be a concern once the basic design is set.

Heuristic #4: *Encourage compactness only after a basic design is achieved.* Once a basic design has been found, local optimizations that are sure to improve it should be made. The most common of these is moving around a few BFUs so that the design fits in a smaller rectangle, thus enabling it to be more easily inserted on a chip or tiled several times.

Heuristic #5: *Use Knight-pairs to place un-connected units.* It is a good idea to spend as much effort on placing un-connected units as is spent on placing connected units. This is especially true when the un-connected units are each connected with many of the same BFUs. Placing these un-connected BFUs in k-pairs with each other can sometimes make the rest of the design obvious in a way that placing a few of the connected BFUs next to each other never would. This is partially because there are fewer options for slots to place BFUs where they will *not* be connected combined with the fact that there are generally fewer once-removed un-connected BFU pairs than there are BFUs that need to communicate with each other.

Heuristic #6: *Place holes at edges and corners of designs.* Besides placing un-connected pairs, another way to place in a "reverse" order is by placing the holes, or slots on the grid without used BFUs, first. Even if the holes are not placed first, they should be considered before the design is complete. A hole in the center of a design is not likely to ever get used by other units on a chip, most likely becoming wasted area and a lost opportunity for high-connectivity between units. Always place the holes near the BFUs that are likely to take input and provide output for the design, otherwise it might be very hard to incorporate the design onto a chip that will use it.

Heuristic #7: *Do not place in a vacuum.* A very large effect on the design of a unit will come from the other units on the same chip — they will determine which shape is best, where holes should be to correspond to other units' jutting-out

points, and how much level-3 usage is reasonable. Putting a design on several chips with different other units present will cause MDL+ to place it differently, and manual placement should be no different. Of all effects from other units, the largest will stem from a design's interactions with those other units.

Heuristic #8: *Ignore artificial regularity and structure.* As we saw with several of the examples, humans tend to have preconceived notions that add artificial constraints to the placement problem. Sometimes this is a benefit, such as when it keeps BFUs that are performing a multi-word operation next to each other, but it is usually harmful. BFUs should not be placed next to each other because they need to share a line. Instead, the designer should always be asking himself the question "Why do they need to be near each other?" and if the only answer deals with sharing a level-1 line then the implication is clearly that they can be two units apart in a cardinal direction, not only next to each other. Insufficient questioning of design practices can lead to an understandable reliance on imaginary constraints.

7.2.2 How to use MDL+

Since MDL+ was designed carefully from the objectives in chapter 2, the best ways to use it are largely encompassed in the goals from that chapter that govern the ways in which it can be used. After describing the implementation of MDL+ in detail in chapters 4 and 5, we provide general insights on how to use MDL+ effectively here.

Heuristic #9: *Use multi-level programming.* One of the few common themes that can be found when searching the literature on high-level synthesis is the need for some element of multi-programming, as described when we set it as one of our goals behind MDL+. In fact, multi-level programming can be very useful in any environment such as MDL+, due to the compiler's interaction with the programmer in the form of several intelligent automatic phases. Since MDL+ was designed in an effort to be as multi-level programmable as possible, this turns out to be one of its most useful features.

Programmers should not be afraid to use this feature. The best way to make use of it is to start by programming at a high level and then compiling down to low level

MDL+ code. If the code seems fine, then it can be left alone, but if it leaves something to be desired then the programmer should interact with the compiler. Several ways have been suggested for this interaction through this thesis, and any of them are all right, as it is the interaction itself that is the key.

Heuristic #10: *Let the driver do all of the driving except for dynamic sources.* The driver is quite good at doing a non-intelligent phase, and should therefore be left alone to do its work. Unless there is some special reason to act differently, programmers should not explicitly enable or disable any level-1 or level-2 lines and should not turn off any level-3 lines. This will keep the code shorter while it remains just as effective. The exception is that MDL+ programmers should explicitly disable all lines that can be read by a dynamically sourced port but will not be read, for reasons mentioned in section 4.3.

Heuristic #11: *Use the "-S" option whenever possible.* There is no reason to make the code longer and thus slower to load and harder to read as well as having it take up more space unless it will actually be loaded onto a physical fabricated chip.

Heuristic #12: *Do not interfere with intelligent phases until they mess up.* Intelligent phases of the MDL+ compiler are not very intelligent, but they are intelligent enough to do most of the things that they are responsible for. There is no reason to manually route a chip until the compiler has been given a chance to route it, or place it before the compiler has been given a chance to place it.

Heuristic #13: *Try using "-2UZ" and "-2UY" when necessary.* If the compiler has a hard time placing a chip then let it try re-placing the chip again and again for a long time until it finds a solution. After a while the compiler becomes less likely to find a better solution than it has already found, but programmers should not be afraid to try the "-Z" nondeterminism or "-Y" simulated nondeterminism. When the initial placement is very bad, there is a good chance that the placer can do better.

Heuristic #14: *Compiled designs can be incorporated into a chip.* While it would be unwise to forget the option of compiling an entire complicated chip at once, programmers should remember that each design of a smaller unit can be compiled on its own and then incorporated into a complete chip design after it is placed and

routed. However, as suggested by an earlier piece of insight, such an approach should only be taken after the compiler has failed to make a satisfactory design for the chip as a whole.

7.2.3 MATRIX's Strong Points

After attempting to compile several different general purpose computing architectures as well as several other MDL+ programs in the last chapter, we offer some ideas about which architectures and types of programs MATRIX and CGRAs are best suited to. We also review the elements of MATRIX that are not very useful for compilation.

Heuristic #15: *If the architecture is important enough then it should be used.* While MATRIX and MDL+ performed better on some architectures than others, none of them were so bad that they need to be avoided at all costs. If there is some general purpose computing architecture that is very appropriate to an application, then it should be emulated on MATRIX.

Heuristic #16: *Small units are better than large units.* High throughput might require a large unit, but in general smaller units are easier to design and route, and then connect to each other on a MATRIX chip. Therefore, whether the unit is emulating a general purpose computing architecture or an ASIC design, it is better if it is smaller.

Heuristic #17: *Do not worry about running out of wires.* MATRIX has many wires on it, at least until our suggestions for eliminating some of them are implemented. While we did not consider any examples that are heavily connected to the point of risking running out of wires, we believe that we did consider sufficiently dense designs to conclude that the ratio of wires to BFUs is large enough to prevent a reasonably-connected group of BFUs from running out of wires to connect the signals they share.

Heuristic #18: *A chip can be modeled as connected clusters of BFUs.* The small heavily connected clusters of BFUs can be sparsely connected to each other to form almost any design that might be desired on MATRIX. This type of implementation is likely to use the smallest number and cheapest type of MATRIX resources while

being able to modularly form most algorithms that might be desired on MATRIX. Each cluster of BFUs might be emulating a general purpose computing architecture or it might be an ASIC design.

Heuristic #19: *Avoid level-2 lines and if possible length-2 level-1 lines. Use as little of the control system as possible.* In chapter 6 we suggested eliminating the level-2 lines because they were not very useful for designs, while adding a lot of complexity and area to a CGRA. We also suggested attempting to eliminate length-2 level-1 lines because they were not very useful while adding some area to the chip. Finally, we suggested a complete overhaul of the control system. These resources should be avoided because they are not very useful and in some cases not very easy to use and also because they might not exist on CGRAs for a very long time. Thus, designs that make heavy use of these features might soon require an extensive redesign.

In conclusion, we have gained many insights through this work with MDL+. To be used best they should probably be treated like any group of heuristics, with common sense arbitrating disputes between them. It is also important to remember that no one has used MDL+ or programmed for MATRIX or any CGRA very extensively and thus a lot of experimentation will be as helpful for an MDL+ programmer as any of our other heuristics.

7.3 Future Work

While no one will likely ever do more work to improve MDL+ and extend it into MDL++, this section is devoted to suggesting future work that should be done. Some of it is work that would have been included in MDL+ if there had been more time, and some of it involves major projects that should follow the MDL+ project.

7.3.1 Basic MDL+

Even though MDL+ has improved upon MDL in a lot of ways, there are still many more ways in which MDL+ can be fixed and made even better. This is a list of some of them.

- Fix the known bugs. A file has been kept through the development of MDL+ that lists all currently known bugs in the program and even suggests ways to fix many of them. Any further effort on MDL+ should start by working on these problems.

- Add macro functionality to MDL+. A macro or package would be a group of BFUs which could then be placed in a layout. When being placed, any references in the package, including the names of the BFUs, would be changed into unique names. In this way, a unit could be added to a layout several times without the different copies of it confusing each other.

- Every list structure in the language should be able to be rearranged in order. Currently, most lists can be arbitrarily ordered in the syntax of MDL+, but some can not.

- Add explanations of basic syntax errors. When the compiler halts with most parse errors it provides a good explanation of the cause of the error, but for basic syntax errors is provides no explanation.

- Add an option to output verilog for a chip to be placed in the second programmable global context. There are two programmable global contexts, and MDL+ has the ability to output a chip to either one but the syntax does not provide a way for the user to ask MDL+ to output to the second context, thus it always outputs to the first.

- Add sureness amounts to the syntax of many statements. Once the programmer has a means of telling the compiler how sure he is that he wants a BFU placed in a certain position or routed over a certain line the compiler can use those suggestions to help its algorithm without being bound to them. This will add to the amount of ability the MDL+ compiler has to interact with its user.

- Incorporate the Connect construct into the Layout construct. This idea was implemented a long time ago but then abandoned, however it should be considered again. The Connect is usually thought of by the programmer as part

```
(def-constant x 3)
(def-constant x (static l1_n1))
(def-bfu-ports d (ports (fp1port x x (tscycle 2)) (fp1port 1)
              ))
(def-bfu b d)
(def-layout l (b 1 1))
(def-chip BadChip l)
```

Figure 7-2: MDL+ code that does not make sense

of the internal MATRIX chip and thus should be specified with the rest of the chip. Then, shifting a chip Layout across the grid would shift control of the appropriate level-3 lines as well, as it should.

- Get feedback from users. At this point, not many people have used MDL+. If a group of people used it for a while and then provided feedback on its capabilities, it could be improved even further.

7.3.2 Driver

When discussing the driving phase we concluded that there was no need to make the driver more intelligent to the extent that it would correct problems by, for example, driving a level-3 line with some constant because it was being read. However, it would be great to augment the driver to search out as many of these nonsensical elements of design as possible and warn the user.

Consider figure 7-2. The only specification on the entire chip is that the BFU at (1,1) has its first floating port (FP1) in static source mode receiving the level-1 line from its north as input.

Compiling this code with the MDL+ compiler and the -dS options shows that the Driving Phase of the compiler has added a specification to the BFU just north of "b", the BFU at location (1,2). That BFU, even though it has no configuration telling it what to drive, has its l1_s1 enabled. We could have the Driving Phase set the BFU at (1,2) to have its Fm port always set to the constant ALU value PASSA, and the A port set to the constant value 0, so that the BFU "b" is at least getting valid input

of some kind for its floating port, with the minimum power dissipated.

However, the more important thing would be to warn the user that he is having a BFU look for a piece of its input to come from an unspecified value. Perhaps the user meant to specify the BFU at (1,2) or thought that some package of lower-level definitions that he had included would define the output of BFU (1,2). If so, this warning would be very valuable. If not, in the less likely case, the user could ignore the warning. So that the user has the option of ignoring the warning, it should merely be a warning as the errors from the intelligent phases of the compiler are, and not halt the compiler as the errors from the parser do.

Besides adding these warnings, the best way that the driver can be improved is by adding to the MDL+ grammar a means of specifying where a dynamic source might take input from. This would make it very easy on the programmer to indicate the range of places that the driver needs to turn on, thus eliminating many of the 30 wires which the driver could not have easily eliminated.

Although adding warnings and adjusting the dynamic source syntax are the two best ways that the driving phase can be improved, there are also ways in which it might improve its analysis. First, there could be more analysis to determine whether a line that is read actually has the value used somewhere. If the value is never used then the line can be disabled even though it is read.

Second, in the absence of a change to the MDL+ syntax for dynamic sources, the driver might be able to find a way of easily eliminating many of the possible sources that the dynamic port might have chosen. Although figuring out exactly which lines the dynamic source will choose in general is hard, the driver might be able to do some analysis so that it can turn off many more lines in the cases where the analysis is easy.

With each change to the automatic driving phase, we are trying to get closer to the goal of MDL+ programmers never needing to see or use a BFU Power and never needing to disable driving a line.

7.3.3 Router

The syntax for the router and the router's algorithm could both be improved. The syntax handles routing between BFUs, but should be extended to handle routing between VBFUs as well as routing carry chains together for single-cycle wide word operations or even multi-cycle pipelined operations.

The routing algorithm itself should be improved in at least two ways. First, it should be extended to handle more cases by doing a lot of routing that requires two hops. While the router currently handles routing every wire that can be routed in one hop, it is very bad at routing other wires.

Second, the router should do a lot more timing analysis. At this point it merely routes between two BFUs, and extra register objects can be added so that there will be delays between the signal's generation and consumption, but it would be better if the syntax allowed a user to specify that a BFU wanted to receive another BFU's output after a delay of n cycles. Then, the router could be more intelligent, using long paths and higher level network wires to route signals that need to take a long time and using level-1 lines for signals that can not have any delay at all.

Third, if the time switching capabilities of MATRIX are working once the chip is fabricated then a router that always succeeds should be built using an algorithm that keeps slowing the chip down until every line can be routed to its destination in the appropriate number of macro-cycles.

7.3.4 Placer

As the placer has been the subject of most of our research, since it is one of the most difficult parts of designing a MATRIX chip, it is also the target for a lot of suggested future work.

To start with, there are several ways that an expanded MDL+ syntax might enable a better placer to be designed. If MDL+ adds optional sureness values to each placement, then the compiler could start using different algorithms. The simulated annealing algorithm is a good one in general because it does not require a starting

point that is near the optimal solution, but if the user provides a placement that is close to the optimal solution then the placer could decide to use a different algorithm to do a more conservative hill climbing of the problem.

Additionally, the compiler would be able to output the user-placed BFUs with 100% sureness values, and other BFUs with lower sureness values so that a "-2Z" option would replace the old "-2UZ" option when the user merely wanted the originally unplaced BFUs to be re-placed. Furthermore, the compiler might update the sureness values of BFUs depending on how its search went so that progressive calls to "mdl -2Z" might become calls to the compiler to continue work instead of calls to starting over and retrying as they are now.

Even without sureness values, future work on the placer might include modifying the algorithm or even replacing it. Since we have developed several intuitions on how to manually place, discussed in the last section, these heuristics might now be incorporated as problem-specific patches to the algorithm or could even replace the old algorithm. After all, part of the reason that a general simulated annealing algorithm was chosen was because we did not have any heuristics teaching us how to place MATRIX chips yet.

Future work could also take the opposite direction, realizing that there is still more research to be done identifying how MDL+ would compiler to various chips. In this case, a general algorithm might still be necessary, but it might be better to replace the simulated annealing algorithm with an algorithm that is still general but which can be designed more specifically for the problem of placing. One such class of algorithms is genetic algorithms,[3] and there are many other algorithms which might start to become more appropriate to this problem as we learn more about it. Figuring out which algorithms are best suited to solving this problem is one of the continuing tasks for MDL+ research.

One of the advancements that could change the preferred placement algorithm is getting a better interface to the router. Currently, the placer has a very efficient means of keeping a good estimate of how routable the chip is at every point during

[3]This idea was suggested by Thomas Colthurst.

the placement phase. However, as the router becomes more intelligent the placer will either need to change the way it makes these calculations or actually interface with the router and attempt to route the chip several times while arranging possible placements.

Finally, there are several ways in which the current placer could be easily modified that should be considered for future work. First, it could implement a magnetic attraction to the bottom-left of a layout grid. In other words, the minimization function could favor designs that kept themselves closer to the bottom-left of the grid, in an effort to make layouts easily placeable inside other layouts as pieces of a larger design.

Second, the function could be tweaked to encourage compactness of the design. Due to the heuristic of not worrying about compactness until the basic design is set, it might be best to add this to the minimization function only at low temperatures. Such a change would not prevent programmers from needing to shift large segments of the design over to points nearer each other, but it would probably snap all little easily movable pieces into positions as close as possible, and there were several examples in the last chapter of cases when this would have been beneficial.

Third, once the syntax of MDL+ has been extended to allow the compiler to route carry chains together, the placer should be modified to use this information. Once the placer has done its job, the routing of these wires should be trivial, but the placement of BFUs based on this information would be invaluable. Until that time, certain architectures will not usually be effectively automatically compiled on MATRIX. With this adaption of the placer, the 32-bit microprocessor and the VLIW example from the last chapter should be automatically placeable.

7.3.5 Grouper

Since the grouper is in a much more primitive state than the placer, initial future work on it will probably be of a simpler nature. To start with, someone should make the basic functionality work, having it actually group some objects together.

Once the basic grouper is working, more objects should be added to build up

the primitives at this higher abstract layer so that there becomes less of a reason for programmers to ever delve deeper. One high-level object for each basic general purpose computing architecture might be a good way to start.

After the grouper is working and compiling from a larger set of objects, work will need to be done to make it more intelligent. That is, it should perform analysis to group objects into single BFUs as tightly as possible and it should be able to better realize what an object needs. With these added forms of intelligence and analysis the grouper might realize that two objects each only need a 128-byte memory and a single local context, then combining them into a single BFU with its memory in dual-port mode.

Next, as a more advanced placer should be developed that receives feedback from the router it would be good for the grouper to receive some form of feedback from the placer and router to help it determine which units should be grouped together in order to make placement easier. If not feedback from the other phases, the grouper should at least be given heuristics to help it guess at which groupings will make placement easier.

Finally, some analysis should be done to discover the best way to view the resources of a MATRIX chip. The grouper can use the determined metric as a minimization function to help it figure out how it should group various units. One possible way to view the chip is in term of the lines, since there are a finite number of them and due to power considerations it is better to use fewer of them.

Another way to view the chip is in terms of BFUs since there are a finite number of them available and a design that uses fewer BFUs can be included more times on a chip, can have more other units put on a chip along with it, and in some cases being smaller is what enables a design to fit on a single MATRIX chip instead of needing to be spread out over multiple chips.

A third way to view the chip is in terms of the floating ports since they are a limited resource that is in high demand. They can float between many different uses, but each floating port can only be doing one thing at a time: enabling another port on the BFU to be in dynamic source mode, enabling an additional signal to be driven

onto the level-2 or level-3 lines, or enabling the ALU to calculate a MULA or MULAA operation. In the end, the best metric of resource usage on a MATRIX chip might be a combination of these three views.

7.3.6 High-Level Synthesis

After some use of MDL+, people should be able to determine which objects would do a good job of compiling down to MATRIX. After these objects have been added to the language and they can be grouped well, an MDL++ compiler should be designed to compile a high-level language down to these constructs.

Even the small number of objects that are currently available should be a good start for this, as Silage or any general dataflow-oriented language would probably compile down somewhat easily to registers, ALUs, and memory units. Whenever the high-level synthesis research and development indicates that another object would be useful, it can be added to MDL+ and the grouper adjusted to group it.

This model of an MDL++ compiler should provide for a very good first implementation since it provides a very good abstraction barrier. First, it separates the high-level synthesis from the low-level details of the design of CGRAs. A person writing the MDL++ compiler which uses MDL+ as a backend would not even need to know that there are BFUs on MATRIX or what their capabilities include. Instead, he could constrain his thoughts to compiling for the MDL+ objects. A side effect of this is that if the MATRIX hardware changes the MDL++ compiler will not need to change. Instead, the MDL+ compiler will just be redesigned to provide the same objects, grouping them into a different type of BFU.

Second, this division separates the problem of deciding on an architecture from the problem of implementing that architecture. A large part of the job for the MDL++ compiler will be deciding which architecture is best for implementing each piece of the design. Then, it will use the objects for those architectures and the MDL+ compiler will worry about how to best place and route that architecture to make it efficient in time and space.

Part of the reason that it is so hard to program for MATRIX is that there is an

added dimension of efficiency being sought, since the design must be efficient not only in space and time but over all possible architectural designs as well. [Mat96b] This is a drawback of using MATRIX that directly comes from the feature that it does not fix the application's architecture at fabrication time. By removing this problem to a place where only it needs to be solved (architectural efficiency with MDL++) and then solving the other problems separately (time and space efficiency with MDL+), a useful abstraction line has been drawn.

While any actual input on the design of a high-level synthesis compiler is beyond the scope of this project, MDL+ should have created an environment in which the design work will be much easier. This is one of the most important results of MDL+, and thus this future work is highly encouraged.

7.3.7 CGRAs

We have suggested many design changes for CGRAs and in particular for future versions of the MATRIX chip. The OR plane should not be reinstated, the level-2 lines should be eliminated, the control system should be entirely redesigned, and then the length-2 level-1 lines should be studied and possibly eliminated as well.

Even given all of those changes, CGRAs will continue to rely heavily on the assistance provided for them by automatic compilation tools and CGRA-specific CAD tools. An acceptance of CGRAs in the world and an increase in their use will only come about if there is either a lot of work done in development of these tools or a very good high-level synthesis compiler is built for CGRAs.

CGRAs are a new form of general purpose computing device that has advantages over any single other form of computing device, but they are significantly harder to program and use. Thus, CGRAs have the potential to be widely adopted in academic and industry circles, but only when the software tools accompanying them make them a better choice than combining several other architectures onto one chip in hardware.

Appendix A

MDL+ version 1.0 Grammar

```
<program> = <statement>*
<statement> = <def-const> | <def-bfu-network> | <def-bfu-config> |
              <def-bfu-control> | <def-bfu-power> | <def-port> |
              <def-bfu-ports> | <def-bitvec> | <def-bfu> |
              <def-layout> | <def-ioport> | <def-connect> | <def-chip>

<def-const> = (def-constant <id> <const_val>)
<const_val> = dc|<num>|<checked-const-val>|<checked-static-val>|
              <checked-dynamic-val>|<wrapped-checked-alu-val>|
              <wrapped-checked-mem-val>|<wrapped-checked-sel-val>|
              <id>
<checked-const-val> = (constant <num>) | (constant <l3-driver>)
<l3-driver> = <id> | Bfu1 | Bfu2 | Bfu3 | Bfu4 | Bfu5 | Bfu6 |
              otherio | otherside | sameio | thisio | sameside | thisside |
              zero | constant0 | none
<checked-static-val> = (static <srcaddr>) | (static <l3-src-addr>)
<srcaddr> = local | l1_n1 | l1_n2 | l1_ne | ... | l3_h4 | ctrl | md | c0
<l3-src-addr> = L1_nd | L1_sd | L1_ed | L1_wd | L1_n | L1_s | L1_e | L1_w |
              L1_1 | L1_2 | L2_1 | L2_2 | L3_1 | L3_2 | L3_3 | L3_4 |
              io_port | ioport | none | constant | constant0 | zero |
              highbits
<checked-dynamic-val> = (dynamic)
<wrapped-checked-alu-val> = (aluval <checked-alu-val>)
<checked-alu-val> = (<inst>? (ctx 0|1)? (we 0|1)?)
<inst> = (shift a|b r|l f|c|0|1) | (passa|passb inv?) |
              (and|or|xor|add|add0|add1|mul|mula|mulaa|mcon inva? invb?)
<wrapped-checked-mem-val> = (memval <checked-mem-val>)
<checked-mem-val> = ((port single|double)? (alua input|memory)?
              (alub input|memory)? (memdat inputb|local)?
              (cfgwrite disable|enable)? (cfgread disable|enable))
<wrapped-checked-sel-val> = (sel <checked-sel-val>)
<checked-sel-val> = <num> | <sel-val-item>*
```

```
<sel-val-item> = A | B | columnA | columnB | CR_1 | CR_2 | l1_1 | l1_2 |
                 l2_1 | l2_2 | l3_1 | l3_2 | l3_3 | l3_4 | <id>
<wrapped-checked-en-val> = (en <checked-en-val>)
<checked-en-val> = <num> | <en-val-item>*
<en-val-item> = A | B | columnA | columnB | l2_1 | l2_2 | l3_1 | l3_2 |
                l3_3 | l3_4 | IO_0 | IO_1 | zero | one | input | output | <i

<def-bfu-network> = (def-bfu-network <id> <bfu-network-val> )
<bfu-network-val> = dc | <checked-bfu-network-val> | <id>
<checked-bfu-network-val> = (network (<driven_l2_l3_line> <n_or_fp_port>)*)
<driven_l2_l3_line> = l2_d1 | l2_d2 | l3_v1 | ... | l3_h4
<n_or_fp_port> = n1 | n2 | fp1 | fp2

<def-bfu-config> = (def-bfu-config <id> <bfu-config-val> )
<bfu-config-val> = dc | <checked-bfu-config-val> | <id>
<checked-bfu-config-val> = (config <flag>* (tscycle (we <num>)?
                (madd1 <num>)? (madd2 <num>)?)?)
<flag> = ignorecarry | carrypipeline | madd1dyn | madd2dyn | tsenable |
                msb | lsb | left:<source> | right:<source>
<source> = north | east | south | west | local | control | zero | one

<def-bfu-control> = (def-bfu-control <id> <bfu-control-val> )
<bfu-control-val> = dc | <checked-bfu-control-val> | <id>
<checked-bfu-control-val> = (control (inputsel fp1|fp2)? (ctrlmux reduce|or)
                (or <bfu-mem-val>)?
                (reducei <9bit-compare> <9bit-compare>)?
                (reduceii <reduceii-item>*)?
                (crselect <cr_dir>? <cr_dir>? <cr_dir>? <cr_dir>?)?)
<reduceii-item> = <21bit-compare> | (<reduceii-bit-desc>) |
                (not|either|fail <reduceii-bit-desc>) | fail | accept
<reduceii-bit-desc> = fp[0..7] | bit[0..20] | local |
                n1|n2|ne|e1|e2|se|s1|s2|sw|w1|w2|nw
<9bit-compare> = 0[r,R](0|1|x|f)^9
<21bit-compare> = 0[r,R](0|1|x|f)^21
<cr_dir> = n1 | n2 | ne | ... | local | zero

<def-bfu-power> = (def-bfu-power <id> <bfu-power-val> )
<bfu-power-val> = dc | <checked-bfu-power-val> | <id>
<checked-bfu-power-val> = (power (<output-line> disable|enable)*)
<output-line> = l1n1 | ... | l2d1 | l2d2

<def-port> = (def-port <id> <port-val> )
<port-val> = dc | <checked-port-val> | <id>
<checked-port-val> = (port <const-num>? <const-num>? (tscycle <const-num>?)?
```

```
<def-bfu-ports> = (def-bfu-ports <id> <bfu-ports-val>)
<bfu-ports-val> = dc | <checked-bfu-ports-val> | <id>
<checked-bfu-ports-val> = (ports (<port-name> <port-val>?)*)
<port-name> = alu|mem|aport|bport|n1port|n2port|fp1port|fp2port
<port-val> = <unchecked-port-val> | id | dc
<unchecked-port-val> = <const-num>? <const-num>? (tscycle <const-num>?)?

<def-bitvec> = (def-mem <id> <bfu-mem-val>) | (def-bitvec <id> <bfu-mem-val>)
<bfu-mem-val> = dc | <checked-bfu-mem-val> | <id>
<checked-bfu-mem-val> = (cells <const-num>*) | (bitvec <const-num>*)

<def-bfu> = (def-bfu <id> <bfu-val>)
<bfu-val> = dc | <bfu-piece>* | <id>
<bfu-piece> = <checked-bfu-config-val> | <checked-bfu-network-val> |
                <checked-bfu-ports-val> | <checked-bfu-control-val> |
                <checked-bfu-power-val> | <checked-bfu-mem-val> | <id>

<def-layout> = (def-layout <id> <layout-val>)
<layout-val> = dc | <layout-piece>* | <id>
<layout-piece> = (<id> <const-num> <const-num>)

<def-ioport> = (def-ioport <id> <ioport-val>)
<ioport-val> = dc | <ioport-piece>* | <id>
<ioport-piece> = (<sel-field> <checked-sel-val>) |
                 (<en-field> <checked-en-val>)

<def-connect> = (def-connect <id> <connect-val>)
<connect-val> = dc | <connect-piece>* | <id>
<connect-piece> = (<l3-spec> <connect-val-spec>?)
<l3-spec> = [R,C][1..6]_[1..4]
<connect-val-spec> = <checked-const-val> | <checked-static-val> | dc

<def-chip> = (def-chip <id> <chip-val>)
<chip-val> = dc | <chip-piece>* | <id>
<chip-piece> = <id> | (<portname> <id>)
<portname> = [N,W,E,S][0,1,2]

<const-num> = dc | <num> | <id> | <checked-const-val> | <checked-static-val> |
                <checked-dynamic-val> | <checked-alu-val> |
                <checked-mem-val>
<num> = {digit}+ | {0}{x}{hex-digit}+
<id> = <idfirst>(<idfirst>|{digit})*
<idfirst> = {letter}|_|:|.|-|/|$|<|>
```

Appendix B

MDL+ Man Pages

```
File Errors
        In the event that mdl+ can not open up a file that you have
asked it to compile, it will exit giving you the name of the
troublesome file and an error code which can be looked up in
"/usr/include/sys/errno.h".  No later files will be compiled.

Command Line Args
        Although the program's output does not actually indicate it,
mdl+ can compile different files with different options set, all by
one call to the compiler.  You can set the options with the command
line flags, and the compiler will observe them left-to-right.
Whenever you give it a filename (arg not prefixed with "-") it will
compile that file with the args set at that point.  Then it will
continue with the rest of the command line.  If at any point it can
not read a file, all processing stops (see section File Errors).
        Some possible command line args control what kind of output the
program provides after parsing the input files.  The default is to
output verilog files for each chip in the input file.

  -v    this is the default.  Outputs a verilog file for each chip in
        <chipname>-mdl+.v which when included in a verilog model of
        the chip will load the chip's configuration.

  -d    this is the debug output.  It outputs to standard output the
        final object defined in the file and the entire contents of
        the symbol table.

  -0    this is the mdl+ 1.0 output mode.

  -1    this is the mdl+ 1.1 output mode for chips that have not yet been
        put through the Automatic Driving Phase.  Running code through
        mdl -0 is the same as running mdl -D on code generated from the
```

original source by mdl -1. This option is for people who want to get mdl+ 1.0 output, but don't want all of the added lines dealing with power. It does clean up the code considerably.

-2 this is the mdl+ 1.2 output mode for chips that have not yet been routed. Running code through mdl -0 is the same as running mdl -R on code generated from the original source by mdl -2. This option is useful sometimes in helping to understand better what was automatically placed, since there is still an added level of abstraction present that will be removed by the router. Using mdl -2 and mdl -1 should help understand how the router deals with problems

-3 this is the mdl+ 1.3 output mode for chips that have not yet been placed. Running code through mdl -0 is the same as running mdl -P on code generated from the original source by mdl -3. This option is useful sometimes in helping to understand better what was automatically grouped, since there is still an added level of abstraction present that will be removed by the placer and router. Using mdl -3 and mdl -2 should help understand how the placer deals with problems.

Other possible command line args control what automatic improving or optimizing tools are used on your input, after it is parsed, before it is outputted.

-N Do nothing.

-D Do the driving phase. If lines are specified driven or not, leave that alone, otherwise if the line is used then drive it, and finally if the user didn't specify that line and it isn't used, then don't drive it in order to conserve power. This phase has the options -S and -F associated with it.

-R In addition to driving, do the routing phase. This routes wires that were only specified previously as coming from another high-level construct, such as a BFU or a VBFU. (Currently only BFUs.) You can not do the routing phase without the driving phase, and get verilog output. You would have to run mdl -Nv on the output of mdl -R1.

-P In addition to driving and routing (-R) also do the placement phase. This places BFUs that were only specified previously as being in a chip's layout, but not at which location. If you wish to do the placement phase without the routing phase, try mdl -P2 to do phases starting with placement, and print

output just after the placement phase. You can not do the placement phase without the routing and driving phases if you wish chip-loadable verilog output. If that was what you wanted you would have to run mdl -Nv on the output of mdl -P2.

-S This is an option for the Driving Phase. It is for when you are compiling MDL+ code to be used in simulations, but not for use on actual silicon. In this case, you need not worry about turning off unused lines, and in the interest of shorter output files, the driving phase does not specify these lines as powered down. It does still force some lines to be driven.

-F This is an option for the Driving Phase. It is for when you are compiling MDL+ code to be used on an actual Fabbed chip. A Matrix chip would be ruined if all lines were driven, as it would dissipate enough power to fry the chip. Thus, turning off as many lines from being driven as possible is important. This option will have the Driving Phase cause every line to not be driven unless it is either specified by the user to be driven or being read by another unit while the user did not specify anything about it. This is the default.

-U This is an option useful when you're trying to either find out different random possible placings of a chip (thus using the -Z option) or trying to find different deterministic placings (thus using the -Y option). It means that if the Placement Phase is executed, it will ignore the placements you have forced and re-place all BFUs on its own. You can use this as a way to have suggestive-placings in your layout definition which will get ignored, but it is intended so that you can keep running mdl -2UZ or mdl -2UY on its output until you like the result. This is most useful when you have many chip definitions in a single file and only wish to work on the placement of one of them. Note that mdl -2UY continuously being executed will produce different results, just as mdl -2UZ will, except that it is more likely to converge, and is far easier to re-produce the result.

-Y This is the default. It is an option for the Placement Phase, which causes the Automatic Placer to always yield the same placement result. In other words, it makes the MDL+ compiler deterministic. This option can be used in conjunction with the -U option to produce iteratively different but reproducibly random results. Repeatedly executing mdl -2UY on the same file will cause the Placer to re-place the chip, each time with the

initial placement based on the previous placement. This process can converge, as opposed to using mdl -2UZ, but may be preferable since you can reproduce your result by starting with the human-written source code and applying mdl -2UY repeatedly again.

-Z This is the opposite of the -Y option. It makes the MDL+ compiler non-deterministic. It is an option for the Placement Phase. Each call to mdl -Z on the same input file will produce a possibly-different placement. For most chips, you will probably get decent placement from the "standard" deterministic placer, so this option is only recommended to "Shake things up" when you aren't getting good results. To return a novel result to the "standard" placement, assuming you produced a placed but un-routed output file with mdl -2Z, you can not run mdl -2UY on the output from the non-deterministic run. This is because the BFUs are likely listed in a different order than they used to be, and thus a deterministic random algorithm will likely produce different results.

WARNINGS

Warning: WE on the memory of a BFU does in fact default to 0 (no writes), in any specified ALU instruction. However, if you do not tell the compiler that you care about the Fa port of the BFU at all, then the WE becomes unspecified. MDL+ might or might not set it to zero. (Although MDL+ will never actually set it to one, it could end up as one if the config byte is not overwritten and the previous application in the global context had it as a one. This should not be a problem, since using the memory means you most likely pass its values out through the ALU, but we figured we should mention it.

Warning: If MDL+ wants to write output to a file that already exists, it will write over it. Often this is the desired effect since the source file is really an automatically-generated file from the previous MDL+ run. To ensure that you do not write over your own human-generated code it is suggested that you maintain a convention of not beginning your MDL+ source files' names with a capital letter. MDL+ will only produce output file names that begin with a capital letter.

KNOWN BUGS

Bug: The output mdl files generated by mdl -3 do not actually produce correct output. mdl -3 code will ignore any unplaced BFUs.

Bugs: See the /mdl/known_bugs file. There's lots of them.

Appendix C

MDL+ Globally Reserved Words

These are all words which are reserved with special meanings in all of any MDL+ program. There are also many other words which are reserved with special meanings in specific contexts. For those words, look at the part of chapter 4 that is relevant to the particular context.

The globally reserved words are reserved in a case-sensitive manner. Locally reserved words are always reserved in a case-insensitive manner.

[1] This is also a locally reserved word, in the definition of static sources.

Reserved Word	Other versions
dc	DC
def-constant	DEF-CONSTANT
constant [1]	CONSTANT const CONST
static	STATIC
dynamic	DYNAMIC
shift	SHIFT
aluval	ALUVAL
passa	PASSA
passb	PASSB
invA	inva INVA
invB	invb INVB
inv	INV
ctx	CTX
we	WE
memval	MEMVAL
port	Port PORT
alua	ALUA
alub	ALUB
memdat	MEMDAT
cfgwrite	CFGWRITE
cfgread	CFGREAD
network	Network NETWORK
def-bfu-network	DEF-BFU-NETWORK
def-bfu-config	DEF-BFU-CONFIG
CONFIG	Config config
TSCYCLE	TSCycle tscycle
MADD1	MAdd1 Madd1 madd1
MADD2	MAdd2 Madd2 madd2
DEF-BFU-CONTROL	def-bfu-control
CONTROL	Control control

Table C.1: Global Keywords in MDL+, part 1 of 2

Reserved Word	Other versions
INPUTSEL	InputSel Inputsel inputsel
CTRLMUX	CtrlMux Ctrlmux ctrlmux
REDUCEI	ReduceI Reducei Reduce1 reduce1 reducei reduceI
REDUCEII	ReduceII Reduceii Reduce2 reduce2 reduceii reduceII
CRSELECT	CRSelect CRselect crselect
DEF-BFU-POWER	Def-Bfu-Power def-bfu-power
POWER	Power power
DEF-BFU-PORTS	Def-Bfu-Ports def-bfu-ports
PORTS	Ports ports
DEF-PORT	Def-Port def-port
DEF-BITVEC	Def-Bitvec def-bitvec
DEF-MEM	Def-Mem def-mem
CELLS	Cells cells
BITVEC	Bitvec bitvec
DEF-BFU	Def-Bfu def-bfu
DEF-LAYOUT	Def-Layout def-layout
DEF-IOPORT	DEF-IO-PORT Def-IOport Def-IO-Port def-ioport def-io-port
DATAEN	DataEn Dataen dataen
BIT1EN	Bit1En Bit1en bit1en
BIT2EN	Bit2En Bit2en bit2en
DataSel	Datasel datasel
BIT1SEL	Bit1Sel Bit1sel bit1sel
BIT2SEL	Bit2Sel Bit2sel bit2sel
SEL	Sel sel
EN	En en
DEF-CONNECT	Def-Connect def-connect
DEF-CHIP	Def-Chip def-chip
DEF-MEM-OBJ	Def-Mem-Obj def-mem-obj
DEF-ALU-OBJ	Def-Alu-Obj def-alu-obj
DEF-REG-OBJ	Def-Reg-Obj def-reg-obj

Table C.2: Global Keywords in MDL+, part 2 of 2

Appendix D

8 Bit Microprocessor implemented in MDL

This appendix includes many versions of MDL and MDL+ source code which implement the 8-bit microprocessor that is depicted in figure 5-1 on MATRIX. Many excerpts from these implementations are given in chapter 5. The implementations are given in an order from smallest language (MDL) through ever increasing languages (more functionality and more automatic phases) to the largest language (MDL+1.4).

D.1 MDL

This is the code for the MATRIX 8-bit microprocessor implemented in MDL:

```
;; MDL definition of an 8-bit microprocessor

(def-bitvec test_bits 0x0)

(def-constant dontcare (const 0))

(def-bitvec dontcare_bits 0x0)

(def-constant basic-mem
  ((Port Single) (ALUA Input)
   (ALUB Input) (MemDat Input)
   (CtxWrite Disable)
   (CtxRead Disable))
  )

(def-bfu PC
  (config)
  (network
   (L2_d1 N1)
   (L2_d2 N1)
```

```
   (L3_v1 N1)
   (L3_v2 N1)
   (L3_v3 N1)
   (L3_v4 N1)
   (L3_h1 N1)
   (L3_h2 N1)
   (L3_h3 N1)
   (L3_h4 N1))
  (ports
   (ALU ((passa) (ctx 0) (we 0))
((add0) (ctx 0) (we 0)))
   (MEM basic-mem basic-mem)
   (Aport (const 0) (static local))
   (Bport dontcare (const 1))
   (N1port dontcare dontcare)
   (N2port dontcare dontcare)
   (FP1port dontcare dontcare)
   (FP2port dontcare dontcare)
   )
  (control
   (InputSel FP2port) ;; dc
   (CtrlMux REDUCE) ;; dc
   (OR test_bits) ;; dc
   (ReduceI  0rx00000100 0rx00000100)
   (ReduceII 0r0XXXXXXXXXXXXXXXXXXX) ;; (not local)
   (CRSelect N1 N2 NE W2)
   )
  (mem dontcare_bits)
  )

(def-bitvec f-bits
  10 ;; ((add0) (ctx 0) (we 0))
  13 ;; ((and) (ctx 0) (we 0))
  15 ;; ((xor) (ctx 0) (we 0))
  14 ;; ((or) (ctx 0) (we 0))
  43 ;; ((add1 invb) (ctx 0) (we 0)) which is SUB
  )

(def-bfu F
  (config)
  (network
   (L2_d1 N1)
   (L2_d2 N1)
   (L3_v1 N1)
   (L3_v2 N1)
```

```
   (L3_v3 N1)
   (L3_v4 N1)
   (L3_h1 N1)
   (L3_h2 N1)
   (L3_h3 N1)
   (L3_h4 N1))
  (ports
   (ALU ((passa) (ctx 0) (we 0))
dontcare)
   (MEM ((port single) (alua memory) (ALUB Input) (MemDat Input)
 (CtxWrite Disable) (CtxRead Disable))
basic-mem)
   (Aport (static l1_n1) dontcare)
   (Bport dontcare dontcare)
   (N1port dontcare dontcare)
   (N2port dontcare dontcare)
   (FP1port dontcare dontcare)
   (FP2port dontcare dontcare)
   )
  (control
   (InputSel FP2port) ;; dc
   (CtrlMux REDUCE) ;; dc
   (OR test_bits) ;; dc
   (ReduceI  0r000000000 0r000000000) ;; dc
   (ReduceII 0rFXXXXXXXXXXXXXXXXXXXX) ;; always stay in lcl ctx 0
   (CRSelect N1 N2 NE W2) ;; dc
   )
  (mem f-bits)
  )

(def-bitvec a-bits 0 1 2 3 4)

(def-bfu A
  (config)
  (network
   (L2_d1 N1)
   (L2_d2 N1)
   (L3_v1 N1)
   (L3_v2 N1)
   (L3_v3 N1)
   (L3_v4 N1)
   (L3_h1 N1)
   (L3_h2 N1)
   (L3_h3 N1)
   (L3_h4 N1))
```

```
  (ports
   (ALU ((passa) (ctx 0) (we 0))
dontcare)
   (MEM ((port single) (alua memory) (ALUB Input) (MemDat Input)
 (CtxWrite Disable) (CtxRead Disable))
basic-mem)
   (Aport (static l1_nw) dontcare)
   (Bport dontcare dontcare)
   (N1port dontcare dontcare)
   (N2port dontcare dontcare)
   (FP1port dontcare dontcare)
   (FP2port dontcare dontcare)
   )
  (control
   (InputSel FP2port) ;; dc
   (CtrlMux REDUCE) ;; dc
   (OR test_bits) ;; dc
   (ReduceI  0r000000000 0r000000000) ;; dc
   (ReduceII 0rFXXXXXXXXXXXXXXXXXXX) ;; always fail, stay in lcl ctx 0
   (CRSelect N1 N2 NE W2) ;; dc
   )
  (mem a-bits)
  )

(def-bitvec b-bits 5 4 3 2 1)

(def-bfu B
  (config)
  (network
   (L2_d1 N1)
   (L2_d2 N1)
   (L3_v1 N1)
   (L3_v2 N1)
   (L3_v3 N1)
   (L3_v4 N1)
   (L3_h1 N1)
   (L3_h2 N1)
   (L3_h3 N1)
   (L3_h4 N1))
  (ports
   (ALU ((passa) (ctx 0) (we 0))
dontcare)
   (MEM ((port single) (alua memory) (ALUB Input) (MemDat Input)
 (CtxWrite Disable) (CtxRead Disable))
basic-mem)
```

```
  (Aport (static l2_w1) dontcare) ;; rebroadcast from alu
  (Bport dontcare dontcare)
  (N1port dontcare dontcare)
  (N2port dontcare dontcare)
  (FP1port dontcare dontcare)
  (FP2port dontcare dontcare)
  )
 (control
  (InputSel FP2port) ;; dc
  (CtrlMux REDUCE) ;; dc
  (OR test_bits) ;; dc
  (ReduceI  0r000000000 0r000000000) ;; dc
  (ReduceII 0rFXXXXXXXXXXXXXXXXXXX) ;; always fail
  (CRSelect N1 N2 NE W2) ;; dc
  )
 (mem b-bits)
 )

(def-bfu ALU
 (config)
 (network
  (L2_d1 N1)
  (L2_d2 N1) ;; this one matters
  (L3_v1 N1)
  (L3_v2 N1)
  (L3_v3 N1)
  (L3_v4 N1)
  (L3_h1 N1)
  (L3_h2 N1)
  (L3_h3 N1)
  (L3_h4 N1))
 (ports
  (ALU (static l1_n1) dontcare)
  (MEM basic-mem basic-mem)
  (Aport (static l1_ne) dontcare)
  (Bport (static l1_e1) dontcare)
  (N1port (static l1_n2) dontcare) ;; to rebroadcast PC
  (N2port dontcare dontcare)
  (FP1port dontcare dontcare)
  (FP2port dontcare dontcare)
  )
 (control
  (InputSel FP2port) ;; dc
  (CtrlMux REDUCE) ;; dc
  (OR test_bits) ;; dc
```

```
   (ReduceI  0r000000000 0r000000000) ;; dc
   (ReduceII 0rFXXXXXXXXXXXXXXXXXXXX) ;; fail, stay in lcl ctx 0
   (CRSelect N1 N2 NE W2) ;; dc
   )
  (mem dontcare_bits)
  )

(def-layout chip_core
  (PC 1 3)
  (F 1 2)
  (A 2 2)
  (B 2 1)
  (ALU 1 1)
  )

;; Chip boundary definitions

;; MDL Assembler automatically converts the appropriate N1 to E1 to
;; maintain symmetry, etc...
(def-ioport anIOport ;; all DC
  (DataSEL A_L1_1)
  (DataEN A_L2_1)
  (Bit1SEL A_L1_2)
  (Bit1EN input)
  (Bit2SEL B_L3_1)
  (Bit2EN output)
  )

(def-boundary-column aColumn ;; all DC
  (L3Ctrl1 (static L1_1))
  (L3Ctrl2 (const bfu2))
  )

(def-edge anEdge ;; all DC
  (Ports anIOport anIOport anIOport)
  (Columns aColumn aColumn aColumn aColumn aColumn aColumn)
  )

(def-chip chip
  (North anEdge) ;; dc
  (East anEdge) ;; dc
  (South anEdge) ;; dc
  (West anEdge) ;; dc
  (Core chip_core) ;; here i care
  )
```

D.2 MDL+1.1

This is the code for the MATRIX 8-bit microprocessor implemented in MDL+1.1, which uses the basic version of the MDL+ compiler with the automatic driving phase.

```
;; this is an 8-bit microprocessor written in MDL+1.1

(def-bfu PC
  (ports
   (alu ((passa)) ((add0))) ;; default to the first ctx for C/R I
   (Aport (const 0) (static local))
   (Bport dc (const 1))
   )
  (control
   (ReduceI 0rx00000100)
   (ReduceII (not local))
   )
  )

(def-bfu I-Store
  (ports
   (mem ((port single) (alua memory)))
   (alu ((passa)))
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )

(def-bfu F I-store
  (ports
   (Aport (static l1_n1)))
  (cells ((add0)) ((and)) ((xor)) ((or)) ((add1 invb)) ;; last is a SUB
   ))

(def-bfu A I-store
  (ports
   (Aport (static l1_nw)))
  (cells 0 1 2 3 4
   ))

(def-bfu B I-store
  (ports
   (Aport (static l2_w1)))
  (cells 5 4 3 2 1
```

```
   ))

(def-bfu ALU
  (network (l2_d2 n1))
  (ports
   (Aport (static l1_ne))
   (Bport (static l1_e1))
   (ALU (static l1_n1))
   (N1port (static l1_n2)) ;; to rebroadcast to B from PC
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )

(def-layout Micro8-layout
  (PC 1 3)
  (F 1 2)
  (A 2 2)
  (B 2 1)
  (ALU 1 1)
  )

(def-chip Micro8
  Micro8-layout)
```

D.3 MDL+1.2

This is the code for the MATRIX 8-bit microprocessor implemented in MDL+1.2, which uses an MDL+ compiler like the one for MDL+1.1 except that it also has the automatic routing phase.

```
;; this is an 8-bit microprocessor written in MDL+1.2

(def-bfu PC
  (ports
   (alu ((passa)) ((add0))) ;; default to the first ctx for C/R I
   (Aport (const 0) (static local))
   (Bport dc (const 1))
   )
  (control
   (ReduceI 0rx00000100)
   (ReduceII (not local))
   )
  )

(def-bfu I-Store
```

```
  (ports
   (mem ((port single) (alua memory)))
   (Aport (static PC))
   (alu ((passa)))
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )

(def-bfu F I-store
  (cells ((add0)) ((and)) ((xor)) ((or)) ((add1 invb)) ;; last is a SUB
   ))

(def-bfu A I-store
  (cells 0 1 2 3 4
   ))

(def-bfu B I-store
  (cells 5 4 3 2 1
   ))

(def-bfu ALU
  (ports
   (Aport (static A))
   (Bport (static B))
   (ALU (static F))
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )

(def-layout Micro8-layout
  (PC 1 3)
  (F 1 2)
  (A 2 2)
  (B 2 1)
  (ALU 1 1)
  )

(def-chip Micro8
  Micro8-layout)
```

D.4 MDL+1.3

This is the code for the MATRIX 8-bit microprocessor implemented in MDL+1.3, which uses an MDL+ compiler like the one for MDL+1.2 except that it also has the

automatic placement phase.

```
;; this is an 8-bit microprocessor written in MDL+1.3

(def-bfu PC
  (ports
   (alu ((passa)) ((add0))) ;; default to the first ctx for C/R I
   (Aport (const 0) (static local))
   (Bport dc (const 1))
   )
  (control
   (ReduceI 0rx00000100)
   (ReduceII (not local))
   )
  )

(def-bfu I-Store
  (ports
   (mem ((port single) (alua memory)))
   (Aport (static PC))
   (alu ((passa)))
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )

(def-bfu F I-store
  (cells ((add0)) ((and)) ((xor)) ((or)) ((add1 invb)) ;; last is a SUB
   ))

(def-bfu A I-store
  (cells 0 1 2 3 4
   ))

(def-bfu B I-store
  (cells 5 4 3 2 1
   ))

(def-bfu ALU
  (ports
   (Aport (static A))
   (Bport (static B))
   (ALU (static F))
   )
  (control (ReduceII fail)) ;; keep in lcl ctx 0
  )
```

```
(def-layout Micro8-layout
  (PC)
  (F)
  (A)
  (B)
  (ALU)
  )

(def-chip Micro8
  Micro8-layout)
```

D.5 MDL+1.4

This is the code for the MATRIX 8-bit microprocessor implemented in MDL+1.4, which uses an MDL+ compiler like the one for MDL+1.3 except it also has the automatic grouping phase.

```
;; this is an 8-bit microprocessor written in MDL+1.4

(def-bfu PC
  (ports
   (alu ((passa)) ((add0))) ;; default to the first ctx for C/R I
   (Aport (const 0) (static local))
   (Bport dc (const 1))
   )
  (control
   (ReduceI 0rx00000100)
   (ReduceII (not local))
   )
  )

(def-mem-obj F (static PC)
  (cells ((add0)) ((and)) ((xor)) ((or)) ((add1 invb)) ;; last is a SUB
   ))

(def-mem-obj A (static PC)
  (cells 0 1 2 3 4
   ))

(def-mem-obj B (static PC)
  (cells 5 4 3 2 1
   ))

(def-alu-obj ALU
```

```
  (Aport (static A))
  (Bport (static B))
  (ALU (static F))
  )

(def-layout Micro8-layout
  (PC)
  (F)
  (A)
  (B)
  (ALU)
  )

(def-chip Micro8
  Micro8-layout)
```

Appendix E

Full MDL+ Grammar

```
<program> = <statement>*
<statement> = <def-const> | <def-bfu-network> | <def-bfu-config> |
              <def-bfu-control> | <def-bfu-power> | <def-port> |
              <def-bfu-ports> | <def-bitvec> | <def-bfu> |
              <def-layout> | <def-ioport> | <def-connect> | <def-chip>

<def-const> = (def-constant <id> <const_val>)
<const_val> = dc|<num>|<checked-const-val>|<checked-static-val>|
              <checked-dynamic-val>|<wrapped-checked-alu-val>|
              <wrapped-checked-mem-val>|<wrapped-checked-sel-val>|
              <id>
<checked-const-val> = (constant <num>) | (constant <l3-driver>)
<l3-driver> = <id> | Bfu1 | Bfu2 | Bfu3 | Bfu4 | Bfu5 | Bfu6 |
              otherio | otherside | sameio | thisio | sameside | thisside |
              zero | constant0 | none
<checked-static-src-val> = (static <srcaddr>) | (static <l3-src-addr>) |
              (static ID)
<checked-static-val> = (static <srcaddr>) | (static <l3-src-addr>)
<srcaddr> = local | l1_n1 | l1_n2 | l1_ne | ... | l3_h4 | ctrl | md | c0
<l3-src-addr> = L1_nd | L1_sd | L1_ed | L1_wd | L1_n | L1_s | L1_e | L1_w |
              L1_1 | L1_2 | L2_1 | L2_2 | L3_1 | L3_2 | L3_3 | L3_4 |
              io_port | ioport | none | constant | constant0 | zero |
              highbits
<checked-dynamic-val> = (dynamic)
<wrapped-checked-alu-val> = (aluval <checked-alu-val>)
<checked-alu-val> = (<inst>? (ctx 0|1)? (we 0|1)?)
<inst> = (shift a|b r|l f|c|0|1) | (passa|passb inv?) |
              (and|or|xor|add|add0|add1|mul|mula|mulaa|mcon inva? invb?)
<wrapped-checked-mem-val> = (memval <checked-mem-val>)
<checked-mem-val> = ((port single|double)? (alua input|memory)?
              (alub input|memory)? (memdat inputb|local)?
              (cfgwrite disable|enable)? (cfgread disable|enable))
```

```
<wrapped-checked-sel-val> = (sel <checked-sel-val>)
<checked-sel-val> = <num> | <sel-val-item>*
<sel-val-item> = A | B | columnA | columnB | CR_1 | CR_2 | l1_1 | l1_2 |
                 l2_1 | l2_2 | l3_1 | l3_2 | l3_3 | l3_4 | <id>
<wrapped-checked-en-val> = (en <checked-en-val>)
<checked-en-val> = <num> | <en-val-item>*
<en-val-item> = A | B | columnA | columnB | l2_1 | l2_2 | l3_1 | l3_2 |
                l3_3 | l3_4 | IO_0 | IO_1 | zero | one | input | output | <i

<def-bfu-network> = (def-bfu-network <id> <bfu-network-val> )
<bfu-network-val> = dc | <checked-bfu-network-val> | <id>
<checked-bfu-network-val> = (network (<driven_l2_l3_line> <n_or_fp_port>)*)
<driven_l2_l3_line> = l2_d1 | l2_d2 | l3_v1 | ... | l3_h4
<n_or_fp_port> = n1 | n2 | fp1 | fp2

<def-bfu-config> = (def-bfu-config <id> <bfu-config-val> )
<bfu-config-val> = dc | <checked-bfu-config-val> | <id>
<checked-bfu-config-val> = (config <flag>* (tscycle (we <num>)?
                (madd1 <num>)? (madd2 <num>)?)?)
<flag> = ignorecarry | carrypipeline | madd1dyn | madd2dyn | tsenable |
                msb | lsb | left:<source> | right:<source>
<source> = north | east | south | west | local | control | zero | one

<def-bfu-control> = (def-bfu-control <id> <bfu-control-val> )
<bfu-control-val> = dc | <checked-bfu-control-val> | <id>
<checked-bfu-control-val> = (control (inputsel fp1|fp2)? (ctrlmux reduce|or)
                (or <bfu-mem-val>)?
                (reducei <9bit-compare> <9bit-compare>)?
                (reduceii <reduceii-item>*)?
                (crselect <cr_dir>? <cr_dir>? <cr_dir>? <cr_dir>?)?)
<reduceii-item> = <21bit-compare> | (<reduceii-bit-desc>) |
                (not|either|fail <reduceii-bit-desc>) | fail | accept
<reduceii-bit-desc> = fp[0..7] | bit[0..20] | local |
                n1|n2|ne|e1|e2|se|s1|s2|sw|w1|w2|nw
<9bit-compare> = 0[r,R](0|1|x|f)^9
<21bit-compare> = 0[r,R](0|1|x|f)^21
<cr_dir> = n1 | n2 | ne | ... | local | zero

<def-bfu-power> = (def-bfu-power <id> <bfu-power-val> )
<bfu-power-val> = dc | <checked-bfu-power-val> | <id>
<checked-bfu-power-val> = (power (<output-line> disable|enable)*)
<output-line> = l1n1 | ... | l2d1 | l2d2

<def-port> = (def-port <id> <port-val> )
<port-val> = dc | <checked-port-val> | <id>
```

```
<checked-port-val> = (port <const-src>? <const-src>? (tscycle <const-num>?)?)

<def-bfu-ports> = (def-bfu-ports <id> <bfu-ports-val>)
<bfu-ports-val> = dc | <checked-bfu-ports-val> | <id>
<checked-bfu-ports-val> = (ports (<port-name> <port-val>?)*)
<port-name> = alu|mem|aport|bport|n1port|n2port|fp1port|fp2port
<port-val> = <unchecked-port-val> | id | dc
<unchecked-port-val> = <const-src>? <const-src>? (tscycle <const-num>?)?

<def-bitvec> = (def-mem <id> <bfu-mem-val>) | (def-bitvec <id> <bfu-mem-val>)
<bfu-mem-val> = dc | <checked-bfu-mem-val> | <id>
<checked-bfu-mem-val> = (cells <const-num>*) | (bitvec <const-num>*)

<def-bfu> = (def-bfu <id> <bfu-val>)
<bfu-val> = dc | <bfu-piece>* | <id>
<bfu-piece> = <checked-bfu-config-val> | <checked-bfu-network-val> |
                <checked-bfu-ports-val> | <checked-bfu-control-val> |
                <checked-bfu-power-val> | <checked-bfu-mem-val> | <id>

<def-layout> = (def-layout <id> <layout-val>)
<layout-val> = dc | <layout-piece>* | <id>
<layout-piece> = (<id> <const-num> <const-num>) | (<id>)

<def-ioport> = (def-ioport <id> <ioport-val>)
<ioport-val> = dc | <ioport-piece>* | <id>
<ioport-piece> = (<sel-field> <checked-sel-val>) |
                 (<en-field> <checked-en-val>)

<def-connect> = (def-connect <id> <connect-val>)
<connect-val> = dc | <connect-piece>* | <id>
<connect-piece> = (<l3-spec> <connect-val-spec>?)
<l3-spec> = [R,C][1..6]_[1..4]
<connect-val-spec> = <checked-const-val> | <checked-static-val> | dc

<def-chip> = (def-chip <id> <chip-val>)
<chip-val> = dc | <chip-piece>* | <id>
<chip-piece> = <id> | (<portname> <id>)
<portname> = [N,W,E,S][0,1,2]

<def-mem-obj> = (def-mem-obj <id> <memobj-val>)
<memobj-val> = dc | <const-src>? <checked-bfu-mem-val>? | <id>

<def-alu-obj> = (def-alu-obj <id> <aluobj-val>)
<aluobj-val> = (<aluobj-port-name> <const-src>?)*
<aluobj-port-name> = alu|aport|bport
```

```
<def-reg-obj> = (def-reg-obj <id> dc|<id>)

<const-src> = dc | <num> | <id> | <checked-const-val> |
                <checked-static-src-val> | <checked-dynamic-val> |
                <checked-alu-val> | <checked-mem-val>
<const-num> = dc | <num> | <id> | <checked-const-val> | <checked-static-val>
                <checked-dynamic-val> | <checked-alu-val> |
                <checked-mem-val>
<num> = {digit}+ | {0}{x}{hex-digit}+
<id> = <idfirst>(<idfirst>|{digit})*
<idfirst> = {letter}|_|:|.|-|/|$|<|>
```

Bibliography

[Bro95] Frederick P. Brooks Jr. *The Mythical Man-Month.* Addison-Wesley, 1995. ISBN: 0-201-83595-9.

[CHM91] Albert E. Casavant, Ki Soo Hwang, and Kristen N. McNall. PISYN — High-Level Synthesis of Application Specific Pipelined Hardware. In Raul Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, pages 55–78. Kluwer Academic Publishers, 1991. ISBN 0-7923-9159-4.

[CSW93] Francky Catthoor, Lars Svensson, and Klaus Wolcken. Application-Driven Synthesis Methodologies for Real-Time Processor Architectures. In Francky Catthoor and Lars Svensson, editors, *Application-Driven Architecture Synthesis*, pages 1–22. Kluwer Academic Publishers, 1993. ISBN 0-7923-9355-4.

[DeH95] André DeHon. Specialization Theory. Transit Note 123, MIT Artificial Intelligence Laboratory, March 1995. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn123.ps.Z`.

[DeH96] André DeHon. *Reconfigurable Architectures for General-Purpose Computing.* PhD thesis, MIT, 545 Technology Sq., Cambridge, MA 02139, September 1996. Anonymous FTP `transit.ai.mit.edu:papers/rcgp-tr.ps.Z`.

[Esl95] Ian Eslick. MDL: A MATRIX Description Language. Transit Note 132, MIT Artificial Intelligence Laboratory, November 1995. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn132.ps.Z`.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, pages 44–54, Winter 1994.

[HDDW93] Peter Held, Patrick Dewilde, Ed Deprettere, and Paul Wielage. HIFI: From Parallel Algorithm to Fixed-Size VLSI Processor Array. In Francky Catthoor and Lars Svensson, editors, *Application-Driven Architecture Synthesis*, pages 71–94. Delft University of Technology, Kluwer Academic Publishers, 1993. ISBN 0-7923-9355-4.

[KGV83] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, May 1983. Yorktown Heights, New York.

[LND+91] Dirk Lanneer, Stefaan Note, Francis Depuydt, Marc Pauwels, Francky Catthoor, Gert Goossens, and Hugo De Man. Architectural synthesis for medium and high throughput signal processing with the new Cathedral environment. In Raul Camposano and Wayne Wolf, editors, *Trends in high-level synthesis*, IMEC, Kapeldreef 75, B-3001 Leuven, Belgium, 1991. Kluwer Academic Publishers.

[Mat96a] Morris Matsa. Compiling for Coarse-Grain Reconfigurable Architectures. Master's thesis proposal, MIT, 545 Technology Sq., Cambridge, MA 02139, September 1996.

[Mat96b] Morris Matsa. Implementing RSA on MATRIX. Advanced Undergraduate Project, Massachusetts Institute of Technology, 545 Technology Sq., Cambridge, MA 02139, May 1996.

[MCG+90] Hugo De Man, Francky Catthoor, Gert Goossens, Jef Van Meerbergen, Jan Vanhoof, Stefaan Note, and Jos Huisken. Architecture driven synthesis techniques for VLSI implementation of DSP algorithms. *Proceedings of the IEEE*, 78(2):319–335, FEBRUARY 1990.

[MD96] Ethan Mirsky and André DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996. Anonymous FTP `transit.ai.mit.edu:papers/matrix-fccm96.ps.Z`.

[Mir95a] Ethan Mirsky. MATRIX Micro-Architecture. Transit Note 130, MIT Artificial Intelligence Laboratory, November 1995. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn130.ps.Z`.

[Mir95b] Ethan A. Mirsky. Coarse-Grain Reconfigurable Computing. Master's thesis proposal, MIT, 545 Technology Sq., Cambridge, MA 02139, December 1995.

[Mir96] Ethan Mirsky. Course-Grain Reconfigurable Computer. Master's thesis, Massachusetts Institute of Technology, 545 Technology Sq., Cambridge, MA 02139, June 1996. Anonymous FTP `transit.ai.mit.edu:papers/eamirsky-matrix-meng.ps.Z`.

[NON91] Yukihiro Nakamura, Kiyoshi Oguri, and Akira Nagoya. Synthesis From Pure Behavioral Descriptions. In Raul Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, pages 205–229. NTT Communications and Information Processing Laboratories, Kluwer Academic Publishers, 1991. ISBN 0-7923-9159-4.

[RP90] J. Rabaey and M. Potkonjak. Resource Driven Synthesis in the HYPER System. *ISCAS-90*, 4:2592–2595, May 1990. New Orleans, LA.

[RvSC+93] Jan Rosseel, Michael van Swaaij, Francky Catthoor, Hugo De Man, Hervé Le Verge, and Patrice Quinton. Regular Array Synthesis for Image and Video Applications. In Francky Catthoor and Lars Svensson, editors, *Application-Driven Architecture Synthesis*, pages 119–165. Kluwer Academic Publishers, 1993. ISBN 0-7923-9355-4.

[Wak91] Kazutoshi Wakabayashi. Cyber: High Level Synthesis System from Software into ASIC. In Raul Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, pages 127–151, 4-1-1 Miyazaki Miyamae-ku Kawasaki 216, Japan, 1991. C&C Systems Research Laboratories, NEC Corporation, Kluwer Academic Publishers. ISBN 0-7923-9159-4.

[WC95] Robert A. Walker and Samit Chaudhuri. Introduction to the Scheduling Problem. *IEEE Design & Test of Computers*, pages 60–69, Summer 1995.

[Xil94] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Logic Data Book*, 1994.

[YR95] Alfred K. Yeung and Jan M. Rabaey. A 2.4 GOPS Data-Drivern Reconfigurable Multiprocessor IC for DSP. In *Proceedings of the 1995 IEEE International Solid-State Circuits Conference*, pages 108–109. IEEE, February 1995.

5466-34