AN INTEGRATED DEVELOPMENT ENVIRONMENT FOR THE DESIGN AND SIMULATION OF MEDIUM-GRAIN RECONFIGURABLE HARDWARE

By

KYLAN THOMAS ROBINSON

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

Master of Science in Computer Engineering

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2010

To the Faculty of Washington State University:

The members of the Committe appointed to examine the
thesis of KYLAN THOMAS ROBINSON find it satisfactory and recommed that
it be accepted.

José G. Delgado-Frias, Chair

David E. Bakken

Clint S. Cole

ACKNOWLEDGMENTS

I am extremely grateful...

To my employer, Schweitzer Engineering Laboratories, for funding the last two years of my graduate school education.

To Washington State University, for selecting me as a Distinguished Regents Scholar and supporting both my undergraduate and graduate studies.

To the members of my committee, Dr. David Bakken and Clint Cole. Thank you for taking the time to participate in this process.

To Dr. Mitchell Myjak and the rest of the HiPerCopS research group, for developing the HiPerCopS medium-grain reconfigurable hardware architecture.

To Dr. José G. Delgado-Frias, my mentor and guide throughout this journey. I was honored to be a part of you research group, and I have benefitted greatly from your expertise in the field. Thank you for your patience, encouragement, and feedback.

To my parents, Roland and Melissa Robinson. You taught me the value of a good education, and you have always encouraged me to reach my full potential.

To my wonderful wife Kristen, you are my strongest supporter and my best friend. Thank you for giving me perspective and for believing in me at every step along the way. I couldn't have done this without you.

AN INTEGRATED DEVELOPMENT ENVIRONMENT FOR THE DESIGN AND SIMULATION OF MEDIUM-GRAIN RECONFIGURABLE HARDWARE

Abstract

by Kylan Thomas Robinson
Washington State University
May 2010

Chair: José G. Delgado-Frias

Medium-grain reconfigurable hardware (MGRH) architectures incorporate the power of a dedicated digital signal processor (DSP) and the flexibility of a field-programmable gate array (FPGA). Traditionally, MGRH design has been performed using low-level tools. This paper presents a new software application, the HiPerCopS Integrated Development Environment (IDE), which facilitates the design and simulation of MGRH systems.

The HiPerCopS IDE uses a software architecture that mimics the physical setup of an MGRH device. It provides a set of libraries that take advantage of inherent hierarchical relationships, abstracting low-level details by handling them automatically. The IDE features a tool for graphical system design, allowing users to design systems by focusing on the relationships between components, not the inner workings of the components themselves.

Since it is targets a reconfigurable hardware architecture, the HiPerCopS IDE supports the the ability to map systems designs to a device. This paper introduces a number of mapping algorithms to the HiPerCopS architecture, and compares their respective merits. The IDE implements these algorithms, allowing users to choose the approach most suitable for their needs.

The HiPerCopS IDE has the ability to run simulations on a given system design. It processes input data from the user and returns a set of results and execution statistics. This allows the user to test their designs in a fast and efficient manner, without the costs associated with implementation on an actual physical device.

It is expected that the HiPerCopS IDE will help accelerate the research of MGRH architectures. A typical use case scenario is presented, giving a step-by-step example of system design and simulation. This paper also suggests future enhancements that will increase the usability and value of the IDE.

Contents

List of Figures

List of Tables

Dedication

Soli Deo gloria

Chapter 1

Introduction and Background

1.1 Introduction

Members of the High Performance Computing Systems (HiPerCopS) research group at Washington State University have developed a novel computer architecture with many potential applications. The uniqueness of this architecture is an asset because it represents uncharted territory in the field of computer engineering. Working in this area can be a problem, though, because standard methods for research and development have not yet been established. This means that HiPerCopS researchers spend most of their time grappling with low-level complexities instead of exploring high-level design concepts. This research group would benefit greatly from a complete toolset that can adequately meet their needs.

Until now, members of HiPerCops have been using tools intended for low-level design. These tools have facilitated the development and simulation of small-scale implementations, but they are not appropriate for exploring the full potential of the HiPerCopS architecture. This thesis report introduces an integrated development environment (IDE) that will assist the members of HiPerCopS as they explore and develop their architecture. This software will allow researchers to work with more speed and precision, testing new ideas and drawing conclusions with efficiency and confidence.

1.2 Background

Before delving into the details of the HiPerCopS IDE, it is important to understand the context and current state of the HiPerCopS hardware architecture. This section will explain the unique features of the architecture and the work that has been done up to this point. This background information was the fundamental driving force behind many of the decisions made during the development of the IDE.

1.2.1 Computers and their Applications

Computing devices are designed with specific applications in mind. For example, battery-powered mobile devices need processors that feature low power consumption. A typical coffee maker does not require top-of-the-line microprocessor technology, just a simple integrated circuit that can control the brewing process. In computer engineering, the rule is that the target application should specify the hardware requirements.

These days, computers are used in countless applications. In fact, it is difficult to identify an area of society that has not been affected by a digital device. With this ubiquity and pervasiveness, it is easy to imagine the diversity among the specific applications of computer technology. Each application has a unique set of requirements, and each set of requirements can be satisfied by a specific type of computing device. It makes sense, then, that the number of device types is proportional to the number of applications.

1.2.2 Hardware Architecture Classification

To organize this large field of digital technology, computer systems are usually grouped according to hardware architecture. Hardware architecture refers to the physical properties of the system components, how they are arranged, and how they interact with each other. There are four major categories of hardware architecture: application-specific integrated circuits (ASICs), reconfigurable hardware, digital signal processors (DSPs), and general purpose processors.

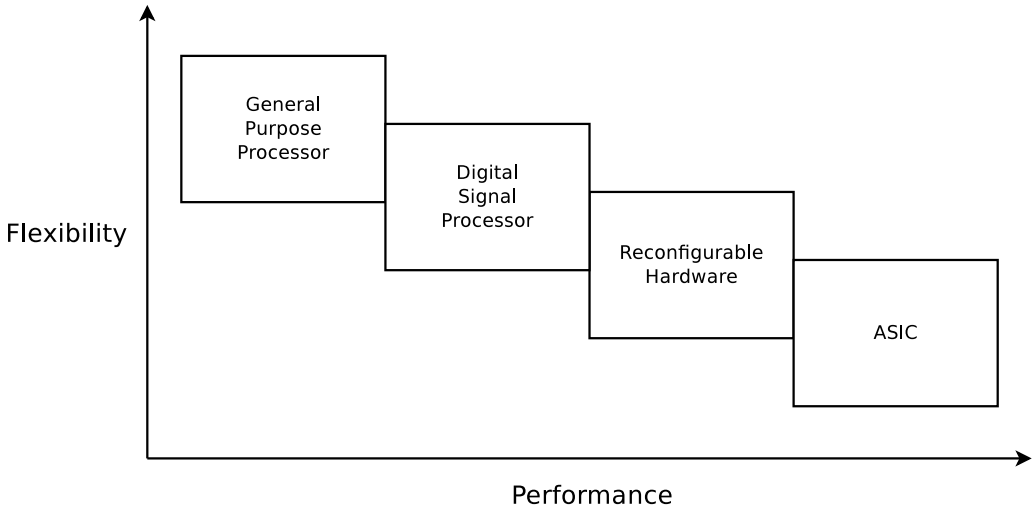


Figure 1.1: General hardware architectures

ASICs are typically small and inexpensive. Each ASIC is designed to perform a specific function with limited scope. This makes the architecture an ideal choice for embedded devices and simple consumer electronics.

Reconfigurable hardware is unique because it can be manipulated to perform a number of different tasks. Other hardware architecture categories are limited to a fixed structure, but the fundamental execution paths of reconfigurable devices can be altered to modify their behavior.

Digital signal processors are designed to handle multimedia data. They can be used to capture, encode, communicate, and decode digital signals such as images, video, and sound. The algorithms involved in these processes can be quite complex, so specialized hardware is needed to perform these tasks.

General purpose processors are found in a variety of applications. They are able to implement nearly any solution, and are found in modern PCs. Members of this class of hardware architecture are usually more expensive, due to the wide range of operations that they can perform.

Comparing these hardware architecture categories, it is interesting to note that each one represents a different choice in the tradeoff between flexibility and performance. ASICs

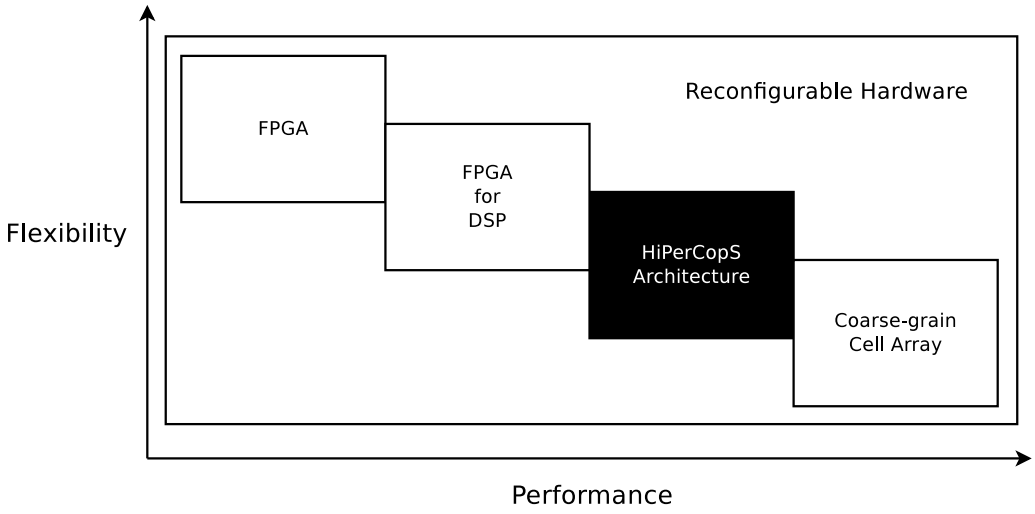


Figure 1.2: Reconfigurable hardware architectures

are highly optimized for their target application, resulting in high performance. However, this optimization comes at the price of a limited function set. On the other end of the spectrum, general purpose processors are powerful because they can perform any reasonable calcuation. Of course, these devices are not nearly as fast or efficient as ASICs. The flexibility/performance tradeoff is an important concept in computer architecture design. Figure 1.1 shows this relationship in a graphical presentation.

Each hardware architecture category can be divided into several subcategories. For the purposes of this discussion, the subcategories of the reconfigurable hardware realm will be examined more closely. They include: field-programmable gate arrays (FPGAs), FPGAs for DSP, and coarse-grain cell arrays.

FPGAs are devices that allow reconfiguration at the lowest possible level. Designers can specify exactly how each bit of their system behaves. While this allows complete control, it also requires significant amounts of work for applications of a significant size.

FPGAs for DSP still allow bitwise control, but they provide shortcuts to developers wishing to implement algorithms for digital signal processing. Usually, these devices will include one or more static, dedicated multipliers for the designer to incorporate into their system. These peripheral components speed up both the design process and the final

product, at the cost of a little freedom.

Coarse-grain cell arrays are groups of small functional units that can act together to form a system. The functional units typically implement addition, multiplication, bit shifting, and other simple computations. They usually work with 32 or 64 bits of data at a time, giving them the power to quickly perform large, complex computations.

1.2.3 The HiPerCopS Architecture

The HiPerCopS architecture falls into the reconfigurable hardware category, but it does not necessarily belong to any of the previously established subcategories. This new architecture provides more abstraction than an FPGA for DSP, but it works with smaller data chunks than a coarse-grain cell array. The HiPerCopS architecture is actually a member of a fourth type of reconfigurable devices - the medium-grain reconfigurable hardware architecture [1].

The term 'medium-grain' indicates that the architecture works with data that is divided into 4-, 8-, or 16-bit chunks. This increases the amount of control a system designer has over their implementation, but it also allows them to avoid repetitive tasks that are common in FPGAs. This type of architecture represents a reasonable compromise in the tradeoff between flexibility and performance, as shown in Figure 1.2.

Elements

The atomic unit of the HiPerCopS architecture is the *element*, shown in Figure 1.4. An element is an array of 32 bits, which can be treated as a 32x1 storage unit or a 16x2 look-up table (LUT) [2] [3]. Depending on the configuration, this basic building block can perform computations or store data. This is the real power of the HiPerCopS architecture.

An element is said to be in 'memory mode' when it is being used as a storage unit. Memory mode elements support read and write operations to allow access to the stored data. Both of these operations require five inputs bits to specify the target address. During a data write, a sixth input bit is used to determine whether a logic high or a logic low value should be written. For a data read, a single output bit returns the value at the given

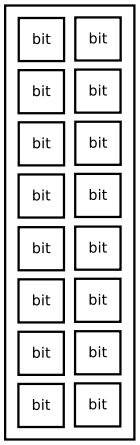


Figure 1.3: An element

address.

When treated as a LUT, the element is said to be in 'math mode.' Math mode elements accept four bits of input and produce two bits of output. The input bits are referred to as α, β, γ, and δ and the output bits are $\psi_{1:0}$. In most cases, a math mode element uses the inputs and outputs to define a multiply-accumulate (MAC) function, shown in Equation 1.1.

$$\psi_{1:0} = \alpha * \beta + \gamma + \delta \tag{1.1}$$

Variations on the MAC can be used to implement a number of different arithmetic computations. For instance, to perform addition alone, α and β can be ignored as inputs. To multiply two negative numbers, α and β are negated while γ and δ are ignored. From an implementation standpoint, the four inputs represent the address of a LUT row and the output bits are found in the contents of that row. By changing row contents, any bitwise mathematical or logical functions can be implemented.

Cells

A 4×4 array of elements is called a *cell*, and cells are the basic building blocks of the HiPerCopS architecture. This grouping of elements abstracts the architecture away from

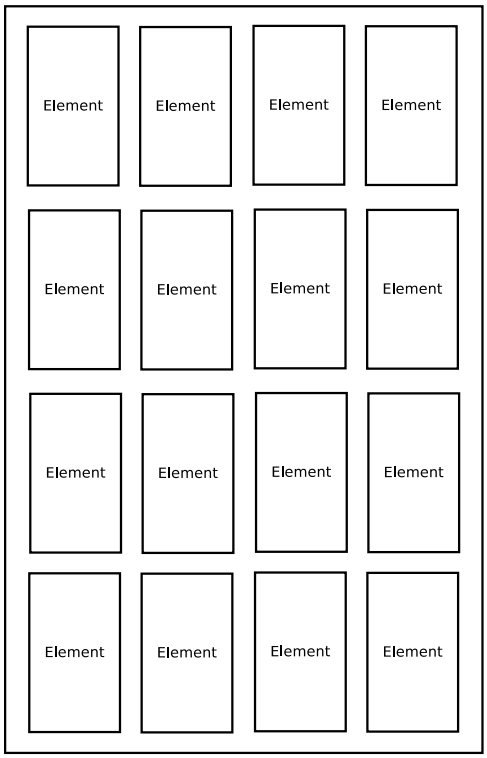


Figure 1.4: A cell

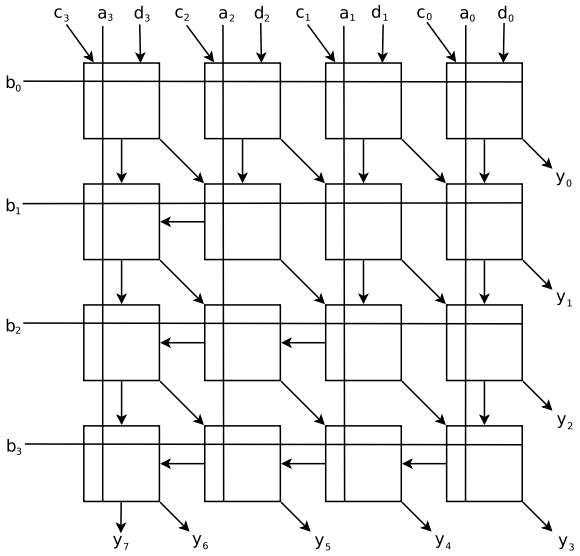


Figure 1.5: Element interconnect routing within a cell

fine-grain issues and elevates it to the medium-grain realm. As with elements, cells can be used in either memory mode or math mode.

Memory mode cells can store 128×4 bits, or 64 bytes of data. In this mode, seven input bits are used to select the target row. An eigth input bit indicates whether a read or a write operation should be performed. A four-bit data bus is used to provide the inputs for a write or accept the outputs of a read.

By configuring its 16 LUTs in different ways, a cell in math mode can perform mathematical and logical operations with 4-bit granularity. Again, MAC operations are very useful in this context (Equation 1.2).

$$y_{7:0} = a_{3:0} * b_{3:0} + c_{3:0} + d_{3:0} \tag{1.2}$$

The elements within a math mode cell must communicate with each other, so they are connected with a set of data busses [4]. The a and b inputs are broadcasted to every element their respective rows and columns. The c and d inputs are applied to the top row of the cell's elements, and the outputs are produced along the bottom rows and rightmost column, as shown in Figure 1.5.

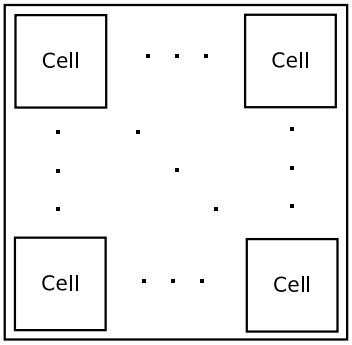


Figure 1.6: A cellfield

Within a cell, the routing of data from element to element follows a consistent pattern. The inputs propogate through the cell until the longest possible path has been followed. As a result, there is a constant delay between the application of the inputs and the availability of the outputs. In this architecture, the system cycle time is set to compensate for this inner-cell delay. As a result, cell computations complete in a single cycle.

Cellfields

Although a single cell can perform an impressive number of functions, it does not have the ability to implement complex applications on its own. In order to be useful, most cells need to be part of a larger system. Such a group of cells is called a *cellfield*.

In the most general case, the set of all reconfigurable cells on a piece of hardware can be considered a cellfield. Each of these cells has the ability to perform computations, store and retrieve data, and communicate with its neighbors. The term *device* is used to describe this general cellfield.

Usually, a cellfield is designed to perform a specific function or process. These types of cellfields are called *modules*. Module operations are often simple, involving fewer than forty cells. Since they are relatively small, a number of modules can be implemented on a single device. An application that utilizes multiple modules, routing the outputs of one module to the inputs of another, are called *systems*. The system is the highest level of abstraction in the HiPerCopS architecture.

Inter-cell data communication within a cellfield is not standardized like the inter-element routing within a cell. Every cell in a cellfield has access to two networks: a local mesh and a global H-tree. The local mesh allows the cell to communicate with its adjacent neighbors. This is how modules perform their operations. The H-tree network is the interstate highway system of the HiPerCopS network. It is most useful for inter-module communication within a system [5].

1.3 Outline

The remainder of this thesis report will describe the features and abilities of the HiPerCopS IDE. Chapter Two explains the context of the HiPerCopS architecture and some of its novel features. Prior research related to the IDE, general objectives, and program flow are described in Chapter Three. An in-depth look at the software architecture is given in Chapter Four. Chapter Five discusses mapping algorithms employed by the IDE. The application's libraries and system design process are described in Chapter Six, and Chapter Seven describes a typical use case scenario. A conclusion and suggestions for future work are presented in Chapter Eight.

Chapter 2

IDE Overview

2.1 The HiPerCopS CAD Application

The development of the HiPerCopS architecture has taken many years, and the HiPerCopS IDE represents another step in its evolution. The direct precursor of the IDE was the HiPerCopS Computer-aided Design (CAD) tool, which was important because it was the first software-based implementation of the HiPerCopS architecture. [6]. It allowed users to define cells, create basic modules, and run rudimentary tests using a graphical interface.

The HiPerCopS CAD tool was originally created to explore two local networking schemes. It implemented both methods for inter-cell connection and used the simulator to compare their performance. Since the focus was on this specific problem, the CAD tool required users to work at a low level. While the HiPerCopS CAD tools offered several features to assist the process of module design, the implementation of a complete system was still a tedious task.

The full potential of a medium-grain reconfigurable hardware architecture cannot be realized if designers only have access to fine-grain and low-level tools. Instead, the software tools for this problem domain should silently handle all low-level concerns. This will facilitate a one-to-one mapping between a system design's concept and its associated specification. The intermediate step of translating the concept to the fine-grain domain will be

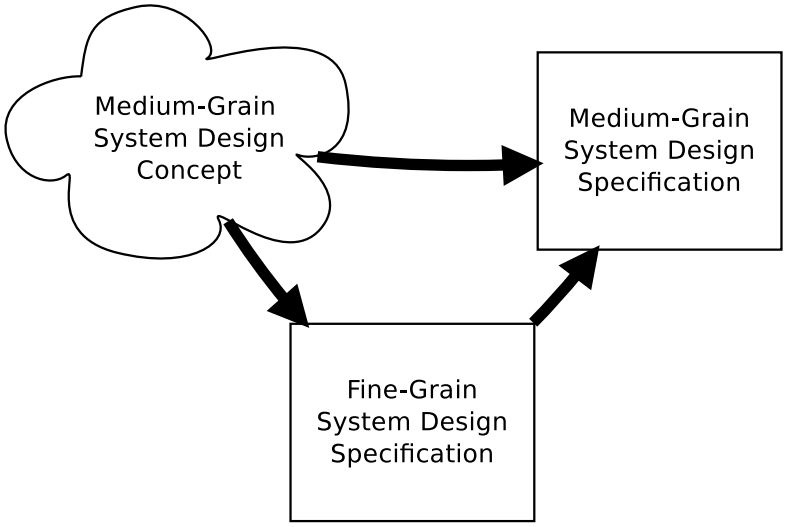


Figure 2.1: Fine-grain specification is an unnecessary intermediate step.

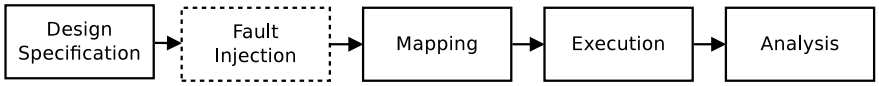


Figure 2.2: HiPerCopS IDE behavioral design

eliminated (Figure 2.1).

With that in mind, the HiPerCopS IDE has been created to streamline the entire design process for medium-grain reconfigurable hardware applications. It was intended to incorporate all of the major features of the HiPerCopS CAD application and expand the functionality to include tools that facilitate design. The IDE allows users to perform the following set of steps from a single interface. These steps constitute the general program flow (Figure 2.2).

2.2 Program Flow

Design specification takes a system concept and defines it as a set of interconnected modules. In most cases, the necessary modules can be reused from previous projects. When a new module needs to be created, it can be quickly built using predefined elements and cells. The new module can then be added to the module library for future reuse. Element, cell,

and module libraries will be discussed in depth later in this report.

Reconfigurable hardware architectures are a good choice for fault tolerant systems. In these systems, it is particularly easy to implement redundancy, error checking, and fault avoidance. To study the impact of faults and fault avoidance schemes, it may be desirable to inject hard faults into a system simulation. The HiPerCopS IDE allows users to create faults at specific locations or generate a random spread of faults across the device [7].

Mapping involves the placement of system modules onto a software representation of the target device. Parts of the device cellfield are reconfigured to perform module operations, and the appropriate local and global routes are established. In the HiPerCopS IDE, this process is performed automatically without the need for any user oversight. Chapter Five will discuss mapping in greater detail.

After the device is initialized with the desired application, users can simulate the execution of different input sets. The simulated modules process inputs in a pipelined manner, and data is passed in a scheme consistent with the local and global networks of the HiPerCopS architecture. Every step occurs exactly as it would in an actual physical implementation. With the ability to perform this type of execution, designers can be confident in their systems, even for large-scale applications.

The HiPerCopS IDE provides a number of resources for the inspection and analysis of systems under test. Log files and mapping records allow users to verify fine-grain operations that were performed automatically. Result files keep track of the applications outputs, and the HiPerCopS IDE provides a utility for comparing expected results against actual results. The IDE even allows users to step through the execution of their systems, pushing down to the element level to see the status of a particular state.

2.3 Objectives

The development of the HiPerCopS IDE was conducted according to a specific set of guiding principles. These objectives were designed to maximize the overall quality of the IDE, and

they are mentioned here because they had direct influences on the software implementation choices.

Hardware Imitation. The source code of the HiPerCopS IDE reflects the exact structure of the HiPerCopS architecture. This serves two purposes. First, it helps ensure that the software is a true representation of the hardware. It is important that the simulations behave exactly the same as physical implementations. Second, it provides a structured, explicit documentation regarding the inner workings of the hardware. Researchers interested in the HiPerCopS architecture can read the software source code to supplement the information contained in academic publications.

Ease of Use. The HiPerCopS IDE would not be necessary if existing tools allowed designers to implement and test designs in an intuitive manner. The user interface of the IDE was carefully designed to maximize productivity while minimizing confusion. Ideally, designers will be able to implement complex, large-scale DSP applications in a matter of minutes.

Code Portability. The HiPerCopS CAD application was written in C#. This choice allowed the software to be written quickly, but it greatly limited the number of platforms on which the application could run. The HiPerCopS IDE was implemented in Python, an interpreted cross-platform language that can be exectued on Windows, Mac, or Linux. This will allow designers to choose their operating system of choice when using the HiPerCopS IDE.

Segregation of Components. Special care was taken to design this application in such a way as to minimize coupling. The graphical user interface and the backend are completely separate entities, linked only by a common interface. The structural description of a module has minimal dependence on the interconnection framework. The source code that performs the mapping is ignorant of most of the other software components. This decoupling of software components increases the reliability, extensibility, and reusability of the code, and is usually considered to be a hallmark of good design [8].

Rigorous Testing. The HiPerCops IDE comes with a complete suite of unit tests. This ensures that each of the software components functions as intended. Future maintenance of the software source code will greatly benefit from this testing framework, and users can write additional tests to isolate bugs or improve their understanding of the application.

Chapter 3

Software Architecture

The HiPerCops IDE has many components. There are features that facilitate system design, perform device layout, simulate faults, execute programs, and assist in output analysis. From the highest level of abstraction, though, the application can be viewed as a simple process. Like any other piece of software, the IDE takes a set of inputs, executes instructions on the data, and produces a result. The IDE is described using a top-down approach, starting at that high level of abstraction and systematically delving into the details of each component.

3.1 Application Context

The basic concept of the HiPerCopS IDE is to draw data from two or three sources of input and produce a single output. This is shown as a data-flow diagram (DFD) [9] in Figure 3.1. The three input sources include a system concept, a set of input data, and, optionally, a fault model. The IDE returns the outputs of the system as its result.

A system concept is a general idea to be implemented on the targeted medium-grain reconfigurable hardware device. This system could be as simple as a small adder, or as complex as an entire DSP consisting of many units. In any case, a detailed design is not required as a prerequisite for using the IDE. The application provides all of the necessary

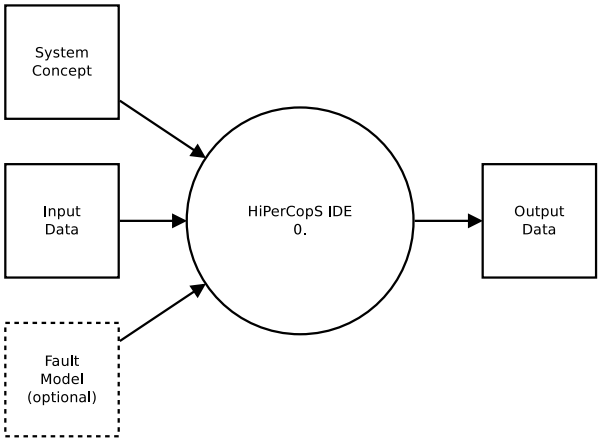


Figure 3.1: DFD context

tools and support for transforming an abstract concept into a complete system design, regardless of the scope of the problem domain.

In order to test a system design, the user must supply a set of data inputs to the simulator. The HiPerCopS IDE accepts a list in comma-separated value (CSV) format, arranged in columns and rows. The first row of the input file consists of system input labels, allowing each column to be mapped to a specific system input label. An n-input system will have n columns, with the first row containing n unique labels. The number of rows is arbitrary, but each column must have the same number of rows.

The fault model is an optional argument to the HiPerCopS IDE. It should be included when the user wants to test their system's robustness and ability to operate in the presence of hard faults. Faults can be generated randomly, specified explicitly, or a combination of the two. Random faults are distributed automatically by the IDE, with the user indicating the number of faults to be placed. Faults with explicit locations are listed in CSV format, with one fault entry per row. Fault entries have the following format, and the entry parameters are explained in Table 3.1:

```
<dev_r>,<dev_c>,<cell_r>,<cell_c>,<elem_r>,<elem_c>,<v>
```

The output of the application represents the results calculated by the system when the inputs are applied. The output file has the same format as the input file: CSV entries

Parameter	Description
dev_r	row index in the device cellfield
dev_c	column index in the device cellfield
cell_r	row index in the cell's element array
cell_c	column index in the cell's element array
elem_r	row index in the element's LUT
elem_c	column index in the element's LUT
v	stuck-at value (either 0 or 1)

Table 3.1: Fault entry parameters (indexes are zero-based)

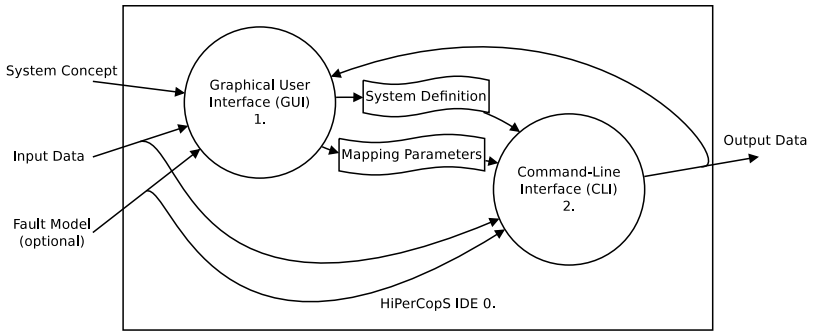


Figure 3.2: DFD 0 - HiPerCopS IDE

arranged in columns and rows, with the system output labels listed in the top row.

3.2 Graphical and Command-line Interfaces

The HiPerCopS IDE consists of two major components, the graphical user interface (GUI) and the command-line interface (CLI) (Figure 3.2). These two components work together to present the user with a unified application framework, and each is responsible for a specific set of functionality.

3.2.1 The Graphical User Interface

The GUI component is the portion of the application which interacts directly with the user. As shown in Figure 3.3, it is used to manage user inputs and facilitate system design. It also displays results, error messages, and other feedback to the user. Essentially, the GUI provides an easy-to-use interface to the functionality of the CLI.

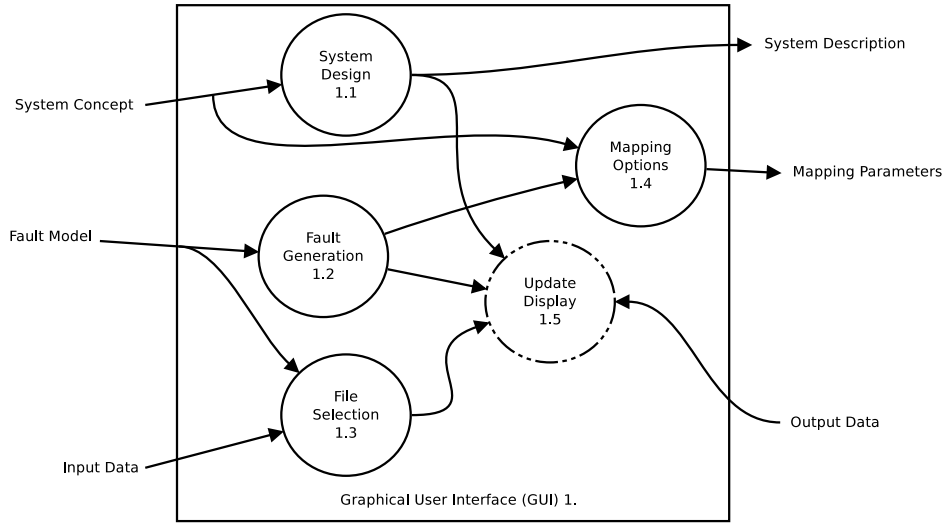


Figure 3.3: DFD 1 - Graphical User Interface

System Design. The system design process involves transforming an abstract system concept into a well-defined system description. The system description contains information on the required modules and how those modules are interconnected. The HiPerCopS IDE provides users with the ability to draw their system concepts as directed graphs, using nodes to represent modules and connectors to indicate data flow. After the system concept is drawn, the IDE uses the graph to generate a text-based representation of the system. The resulting text file represents the system definition, and it is formatted according to a formal programming language grammar. This format is so simple to read and understand that users may choose to write their system definitions directly, bypassing the graphing tool provided by the HiPerCopS IDE.

Fault Generation. As described before, the HiPerCopS IDE has the ability to generate random faults and inject faults at specified locations. The fault generation process is performed in the GUI portion of the IDE. To generate random faults, users can select a predetermined number of faults from the application menu or enter a different number using a dialog box. Explicitly located faults have to be listed in fault files, which are handled in the file selection process.

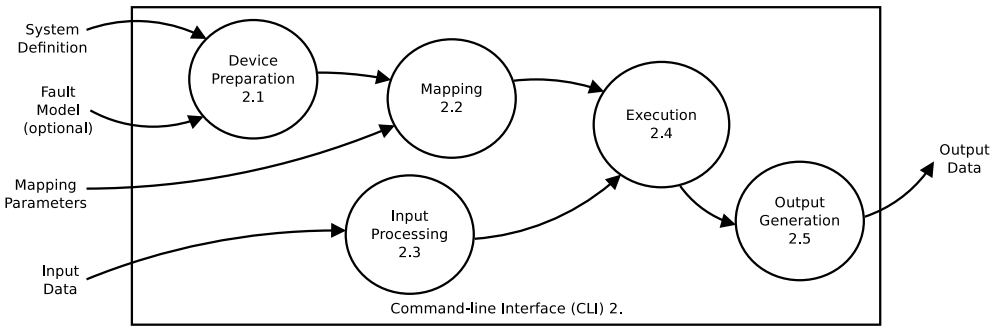


Figure 3.4: DFD 2 - Command-line Interface

File Selection. The file selection process involves the standard file browser paradigm. Users will be familiar with this interface, so they will have no trouble with finding and selecting their input data files and fault files. Currently, these files must be generated outside of the IDE using text processor or spreadsheet editor. In the future, the HiPerCopS IDE will support creating and editing of these files.

Mapping Options. In the GUI portion of the HiPerCopS IDE, the user must choose a specific algorithm for mapping system modules to the target device. There are a number of mapping options available, and they are selected using the application menu. The mapping process and the algorithms involved will be discussed in depth in Chapter Five.

3.2.2 The Command-line Interface

Device Preparation. The device preparation task takes a formal system description and implements the modules as a set of software class structure instantiations. This is a fully automated process that requires no input or oversight from the user. Since the member modules of a system can be quite diverse, device preparation relies on a set of libraries for implementation details. There is one library each for elements, cells, modules, and systems. These libraries are covered in more depth in Chapter Six.

The process of device preparation also involves handling the fault model. During this stage, the device is instantiated as a software object. The specified fault locations are then found on the device, and that particular location is marked as a “stuck-at” fault. This

means that the bit at the specified location will always be stuck at a logic high or a logic low level, unable to change during a reconfiguration or memory operation. Faults of this nature have the potential to cause undesired behavior during a simulation.

Mapping. Once the device is created, the faults are implemented, and the modules are established, the next step involves mapping the modules to the device. Since the HiPerCopS architecture is reconfigurable, systems can be mapped in a number of different ways and still produce the same behavior. As long as the modules are connected correctly via local and H-tree interconnections, mappings are very flexible. In order to take advantage of this feature, a number of different mapping strategies have been developed. These algorithms, fully described in Chapter Five, are optimized for different scenarios. For instance, one method minimizes the amount of device area required while another method avoids mapping modules to faulty areas. The user has the ability to select a particular mapping strategy, but the process itself is fully automated.

Input Processing. Input processing involves parsing the CSV file, normalizing the data, and applying each input entry to the correct system input module. Parsing is a simple task since the format of the file is a well-defined standard. To normalize the data and ensure that it can be processed by the system modules, the HiPerCopS IDE pads each binary string to the necessary length. For each processing interval, a new set of inputs, representing one row of the input file, are applied to the system. This is possible due to the pipelined nature of the HiPerCopS architecture.

Execution. To perform the execution of a system, the inputs are propagated from the modules to the cells and down to the elements. The elements perform their look-up operations, and the results are passed back up to the cells. The cells exchange data with each other and continue the processing loop until the module delays have been exhausted. At this point, the modules use the H-tree global interconnect network to exchange data at the module level. System execution is complete when the last set of inputs has traveled through the entire set of system modules and the results have been produced at the end of the pipeline.

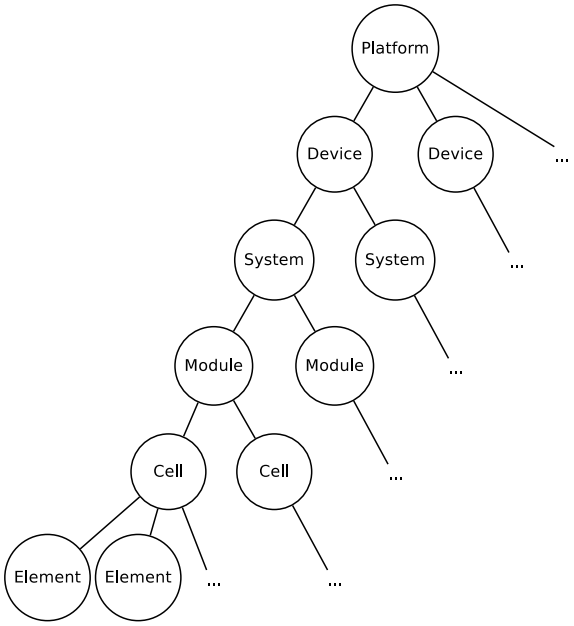


Figure 3.5: Recursive composition in the HiPerCopS IDE

Output Generation. The output generation process consists of gathering the results of the system execution, writing them to a CSV file, sending a copy to the GUI display, and performing analysis. The output file shares the same format as the input file, making it easy to understand from a user standpoint and easy to manipulate using common software tools. The HiPerCopS IDE provides a feature that allows users to compare the output file with a set of expected results, highlighting the differences between the two. This can be especially useful when running tests or debugging a particular system implementation.

The CLI component could be used on its own without any interaction with the GUI. To do this, decisions made in the GUI are passed as command-line arguments. This might be desirable in some circumstances, including automated testing or long-running simulations. However, since a major goal of the HiPerCopS IDE is to be user-friendly, the GUI is really an essential feature. The GUI and the CLI were designed to be separate entities, though, to give users the option of command-line-only operation.

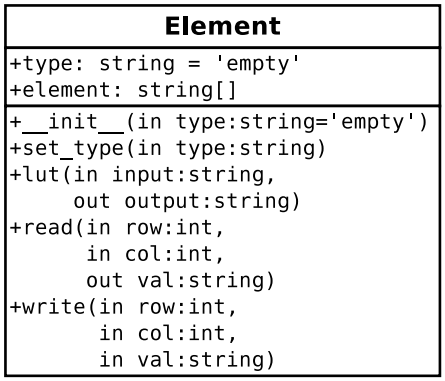


Figure 3.6: UML representation of the Element class

3.3 Recursive Composition

The software structure of the HiPerCopS IDE mimics the recursive composition of the HiPerCopS architecture. Cells are composed of elements, modules are composed of cells, systems are composed of modules, and devices are composed of systems. Figure 3.5 shows this concept, and adds another relationship: platforms are composed of devices. This is a future enhancement, not yet implemented in the IDE, that will allow for multi-core implementations of the architecture.

The class structure of the IDE is a faithful representation of the physical architecture. These objects are created during the device preparation step, process 2.1 of the DFD (Figure 3.4). Inspection of the source code reveals the implementation details of the HiPerCopS IDE and the HiPerCopS architecture simultaneously. The following class documentation, then, will be beneficial to both users of the simulator and designers of reconfigurable hardware.

3.3.1 The Element Class

The element is the basic building block of the HiPerCopS architecture. It also serves as the atomic structure for the HiPerCopS IDE (Figure 3.6).

Attributes

type. This string is used to describe which look-up table is implemented in the element. The name contained in this variable must match one of the entries in the element library. The type attribute can be ignored when the element is in memory mode.

element. This array is a 16x2 matrix of bits. In memory mode, this attribute is responsible for holding the stored values, reading from them and writing to them when required. In math mode, the array acts as a LUT that can be used to perform computations.

Operations

__init__(). This constructor initializes the element member variables. In memory mode, the bits of the element array are set to zero. In math mode, the type attribute is used to initialize the LUT with values from the element library.

set_type(). The type of the element can be changed with this operation. The element's type attribute is changed to the value passed into the function, and the element's LUT is modified using values from the element library.

lut(). The lut method takes a row address as its input, finds the corresponding entry in the element array, and returns the result. This is the basic math mode operation performed by the element class, and this simple function forms the basis for more complex operations at the cell and module levels.

read(). The read method is used when the element is in memory mode. It takes row and column indexes as its input and uses that information to address the element array. The bit found at that location is returned as a string, either '1' or '0'.

write(). The write function takes three inputs: a row index, a column index and a bit value. The row and column parameters are used to address the element

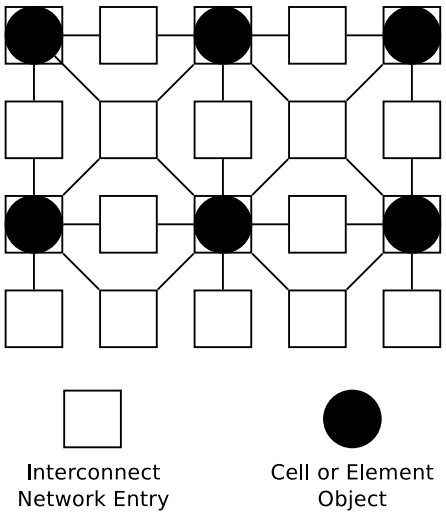


Figure 3.7: Example of a partial interconnect network

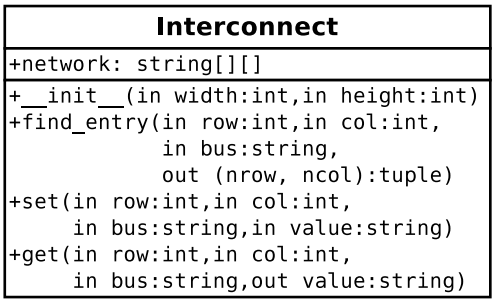


Figure 3.8: UML representation of the Interconnect class

array, and the bit value is written to that location. The write operation is used when the element is in memory mode.

3.3.2 The Interconnect Class

The interconnect class provides a medium for communication within cells or cellfields. Each entry in the interconnect network represents a local bus in the physical hardware. Inside a cell, communication busses connect each element to its eight closest neighbors (Figure 3.7). Cells are connected in the same way as members of a cellfield. Essentially, interconnect networks provide intermediate data storage for the object arrays they support.

Attributes

network. The network is an array of interconnect entries. Each interconnect entry can store a single string of binary data, meant to represent the output of an element or cell and, subsequently, the input of another element or cell.

Operations

__init__(). Given a target array of cells or elements, the initializer creates an interconnect network of the correct size. For an $m \times n$ array of elements or cells, the interconnect network will have $(2 * m + 1) \times (2 * n + 1)$ entries.

find_entry(). Element and cell arrays are addressed using row and column indexes, but the interconnect network has different dimensions. In order to work with the interconnect entries surrounding a specific element or row, an index conversion must be performed. This function takes an element or cell address along with the particular bus in question. The eight busses attached to each cell or element are referred to as 'nw', 'n', 'ne', 'e', 'se', 's', 'sw', and 'w'. The function returns the row and column indexes for the corresponding entry in the interconnect network.

set(). The set function is used to place a binary string value into the interconnect network. As arguments, it takes the row/column address of the element or cell, the name of the bus to be written to, and the value to be written. The row/column address is translated to an interconnect address by the find_entry utility, allowing cell or cellfield operations to proceed without any implementation details regarding the interconnect network.

get(). Binary string values can be read from the interconnect network entries using the get function. Like the set function, get accepts a row/column address and a bus name. The binary string value stored in that interconnect entry is returned.

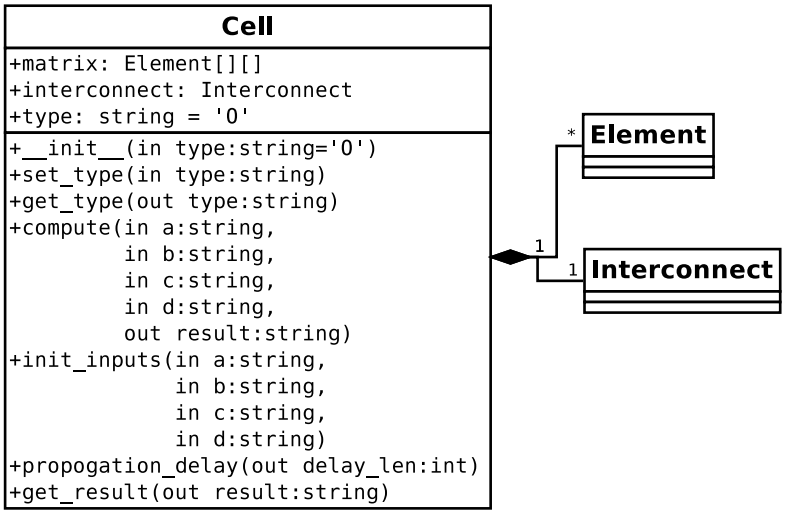


Figure 3.9: UML representation of the Cell class

3.3.3 The Cell Class

Cell objects are responsible for performing all of the computations in a system. Since this is the case, the cell class contains a number of utilities that facilitate math mode functionality (Figure 3.9).

Attributes

matrix. The matrix is an array of elements, arranged in 4 rows and 4 columns. This follows the conventions set in the HiPerCopS architecture.

interconnect. The interconnect attribute is an instance of the interconnect class. It allows elements in the matrix to communicate with each other.

type. The cell type is a string that corresponds to a particular file name in the cell library. This associated library file indicates the types and positions of the elements in the matrix.

Operations

__init__(). The cell initializer creates the cell matrix and sets the element LUTs according to the cell type.

set_type(). This function can be used to set or change the cell's type. It is useful during initial device setup or system reconfiguration.

get_type(). The get_type utility simply returns the current cell type. It is used by the GUI portion of the IDE to display a graphical representation of the cell.

compute(). The compute function takes a set of binary input strings, processes them using the elements in the cell's matrix, and returns the resulting binary output string. It uses init_inputs, propogation_delay, and get_result as helper functions.

init_inputs(). To seed the cell's interconnect network with the proper inputs, the compute function calls init_inputs. This function splits each binary input string and applies the resulting substrings to the appropriate interconnect entries. Then the elements process these inputs using their LUTs, and the data is propogated through the cell.

propogation_delay(). Cell computations run in a loop, allowing the elements to continue processing data contained in the interconnect network until the computation has completed. The amount of time necessary for inter-element propogation to complete depends on the dimensions of the cell. This propogation delay is a constant value for the HiPerCopS architecture, but it could be a variable quantity for other MGRH architectures. In order to make the HiPerCopS IDE as flexible as possible, the propogation delay is explicitly calculated for each cell.

get_result(). After the matrix elements have completed their computations and the binary input strings have propogated through the entire cell, the get_result function gathers the output substrings from the interconnect network. These substrings are concatenated together and returned as a single output.

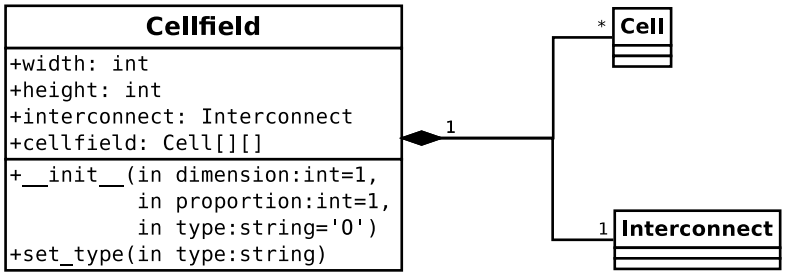


Figure 3.10: UML representation of the Cellfield class

3.3.4 The Cellfield Class

Cellfields are groups of related cells. In the HiPerCopS IDE, the cellfield class acts as a base class for modules and devices. It only provides general attributes and operations, leaving most implementation details up to its derived classes.

Attributes

width. Cellfields are arranged in rectangular arrays. The width attribute simply indicates the number of cells in each row of the cellfield.

height. The height attribute holds the integer value representing the number of rows in the cellfield.

interconnect. The cells of a cellfield must interact with one another, and the interconnect attribute is an interconnect object that facilitates inter-cell communication.

cellfield. The cellfield is an array of cell objects.

Operations

__init__(). The cellfield initialization function defines the cellfield array according to the cellfield type described in the cellfield library. It also assigns values to its height and width member attributes.

set_type(). Each cellfield has a type that corresponds to a particular file in the cellfield library. This library file specifies the types and positions of the cells in the cellfield array, as well as the local networking rules to be observed by the interconnect network.

Chapter 4

Mapping Algorithms

Users of the HiPerCopS IDE can inject hard faults into their systems. These hard faults, also known as 'stuck-at' faults, give insights into how the device might perform in harsh environments. Simulating a system on a faulty device can give a designer great confidence in the robustness of his or her creation.

Before beginning a simulation in the HiPerCopS IDE, the user is required to select a fault avoidance strategy to be used by the automated module mapper. Different fault avoidance strageties implement different algorithms, and it is interesting to note the differences between these algorithms, both in implementation and performance. First, though, it is important to understand the overall concept of fault avoidance.

4.1 Fault Avoidance

As explained in [10], fault tolerance consists of two major tasks: error detection and recovery. Error detection can occur concurrently with system operation, or preemptively while normal service is interrupted. Either way, its purpose is to identify errors within the system and begin the process of recovery. Recovery is further broken down into error handling and fault handling. Error handling takes care of the detected error, while fault handling prevents errors of the same type from occurring in the future. There are four steps included in fault

handling: diagnosis, isolation, reconfiguration, and re-initialization. Diagnosis isolates the source of the problem, and isolation removes this source from the system. Reconfiguration redefines the system in a way that avoids the removed source, and re-initialization performs 'bookkeeping' by updating affected tables and registers.

The HiPerCopS IDE is mainly concerned with reconfiguration and re-initialization in MGRH systems. Together, the activities of reconfiguration and re-initialization can be referred to as fault avoidance. In order to focus on fault avoidance, it is necessary to assume that mechanisms for error detection, fault diagnosis, and fault isolation already exist. Fault avoidance can be especially powerful in MGRH systems due to the flexibility and versatility of the general architecture. MGRH systems are particularly well-suited for fault avoidance mechanism implementation because the H-Tree routing scheme eliminates concerns related to routing and communication [11].

4.2 Placement Algorithms

If a design tool's structure placement process can be modified to avoid permanent faults on the device, an MGRH architecture will support the reconfiguration aspect of cell-level fault avoidance. A number of placement strategies have been developed to achieve this goal:

4.2.1 No Fault Avoidance

The unmodified placement algorithm serves as a baseline. It makes no attempt to avoid faulty areas of the device. A placement failure occurs if any of the placed structures contain faulty cells. In the simulation software, this is called the Naïve algorithm because it is not sensitive to faults on the device.

The Naïve algorithm is a reasonable choice when hard faults are not an issue. The algorithm is simple, intuitive, and efficient. However, hard faults are an eventual reality for every microprocessor. Without a more advanced approach to mapping systems to a device, an MGRH architecture has little advantage over a traditional, non-reconfigurable device.

4.2.2 Simple Fault Avoidance

This algorithm does not map structures to faulty areas of the device. Structures are mapped in the order that the user adds them, and a placement failure occurs if no region of contiguous fault-free cells can be found for a given structure. In the simulation software, this is called the Simple algorithm.

Compared to the Naïve algorithm, simple fault avoidance is a much more appropriate approach for an MGRH architecture. The Simple algorithm takes advantage of fault location information, which can easily be obtained if the device can perform a cell verification self-test.

4.2.3 Size-Aware Fault Avoidance

Before mapping the structures to the device, this algorithm orders them according to area. This causes the largest structures to be placed first and the smallest structures to be placed last. Besides this difference, this algorithm is identical to the Simple Fault Avoidance approach. In the simulation software, it is referred to as the Sizeaware algorithm.

It makes sense that some kind of ordering be imposed on a group of system structures waiting to be mapped. Hard faults effectively create partitions on a device. A significant number of partitions tends reduce the number of options for mapping large structures. By allowing large structures to be mapped first, areas of the device with high amounts of contiguous, available cells can be utilized efficiently. The smaller structures, then, can be appropriated to the (usually smaller) remaining partitions.

4.2.4 Size-Aware Fault Avoidance with Module Rotation

This algorithm takes Size-Aware Fault Avoidance a step further. If a fault-free placement cannot be found for a given structure, the structure is rotated 90 degrees and placement is attempted again. This approach could improve the probability of successful placement for non-square structures like adders and subtractors. In the simulation software, this is called the Sizeaware-rotate algorithm.

Table 4.1: FFT structures, dimensions and quantities.

Structure	Dimensions	Quantity
multiplier	4 × 4 cells	12
memory	4 × 4 cells	8
adder	1 × 2 cells	11
subtractor	1 × 2 cells	11

Returning to the concept of partitions, it is unlikely that a group of random faults will create partitions that are all perfectly square. A rectangular partition could have a square structure mapped to it, but this would not be the most efficient use of functional cells. The ideal solution would be to find a rectangular structure of the same dimensions as the partition. Once rectangular structures are considered, though, the problem of orientation arises. A 3 × 2-cell structure can fit into a 2 × 3-cell partition, but only if its orientation is changed through rotation. The Sizeaware-rotate algorithm extends the Sizeaware algorithm by adding support for this particular type of situation.

4.3 FFT Benchmark

Since MGRH architectures can be targeted toward DSP applications, consider a 32 × 32 cell device used to implement a 256-point fast Fourier transform (FFT) system. The FFT requires 364 cells for implementation, or just over a third of the total device area. The structures comprising this type of unit are listed in Table 4.1, and one possible layout of the unit is shown in Figure 4.1.

For this experiment, the HiPerCopS IDE system definition and fault specification utilities were used along with the mapping options. In this situation, a 'success' refers to a scenario in which an entire system was mapped to a device while avoiding all faults. A 'failure' occurs when there is not enough free, functional space on the device to map the entire set of system structures.

This simulation consists of four test cases, one for each of the previously mentioned placement algorithms. Each test case uses its placement algorithm to map a FFT system

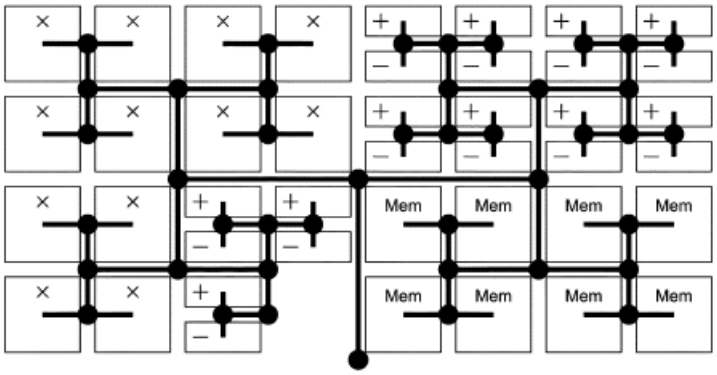


Figure 4.1: FFT system modules

to a 32 × 32 cell device. The independent variable is the number of faults injected into the device, which includes all of the integers between 0 and 199. The dependent variable is the percentage of successful system placement, defined by Equation 4.1.

$$P_{success} = \frac{Number\,of\,Successful\,Placements}{Total\,Number\,of\,Placement\,Attempts} \times 100 \tag{4.1}$$

The dependent variable is measured for each value of the independent variable. A successful placement involved mapping each structure in a system to a fault-free area of the device. The total number of placement attempts (i.e., the number of trials) at each value of the independent variable is set to 100. This number of trials gives very consistent and repeatable results.

4.4 Performance

The performance of the four placement algorithms is shown in Figure 4.2. The mapping success percentage of each algorithm at select points in the experiment is shown in Table 4.2.

As expected, the No Fault Avoidance approach performed relatively poorly. Even with only two faults injected into the device, the probability of successful placement is reduced to 41%. The presence of four faults lowers this figure to 13%, and eleven or more faults

Table 4.2: Mapping success rates for different numbers of injected faults.

	Success rate with:		
Algorithm	80 faults	100 faults	120 faults
Naïve	0%	0%	0%
Simple	100%	97%	48%
Sizeaware	100%	100%	67%
Sizeaware-rotate	100%	99%	67%

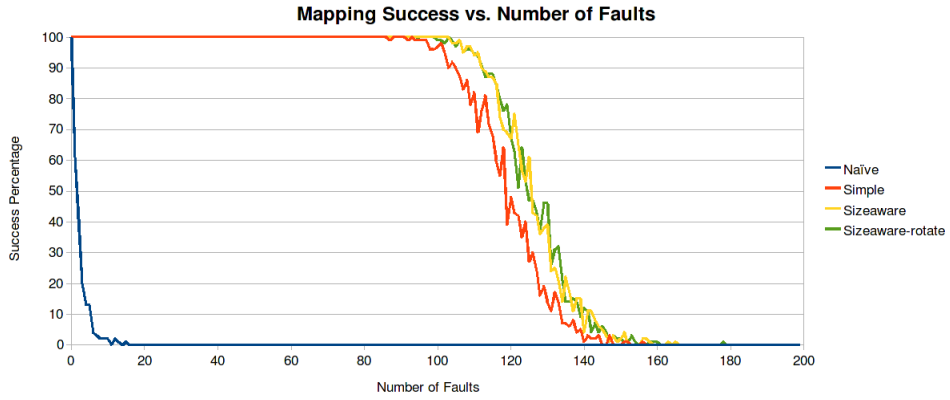


Figure 4.2: Mapping algorithm success versus faults injected

makes placement practically impossible.

The ability to perform Simple Fault Avoidance dramatically increases the chances of successful system placement. In the presence of 97 faults, 99% of placements were successful. Even with 118 faults on the device, there was a 60% success rate. When 128 faults were injected, 30% of the attempts still resulted in full system placement.

Size-Aware Fault Avoidance outperformed Simple Fault Avoidance, with the ability to achieve 99% placement success in the presence of 108 faults. This algorithm successfully placed the FFT system 61% of the time with 127 faults and 25% of the time with 134 faults.

Size-Aware Fault Avoidance with structure Rotation did not have a significant advantage over Size-Aware Fault Avoidance. However, it appeared to have more success when the number of faults was very high. The Sizeaware-rotate algorithm was the only one able to achieve a successful mapping when the number of faults on the device was higher than 170. It is hypothesized that this algorithm is more useful when systems have utilize larger

non-square structures. The FFT system of this experiment features a number of non-square adders and subtractors, but these structures may be too small to expose any benefits of rotation.

Chapter 5

Libraries and System Design

To develop systems using the HiPerCopS IDE, users must first understand the building blocks and tools available to them. The IDE provides all of the necessary resources for implementing a design from start to finish. Each step of the process serves a specific purpose, and the steps in the workflow are connected in an intuitive way. The HiPerCopS IDE employs a top-down approach to the design task, allowing users to work from a big-picture perspective.

The componentized nature of the HiPerCopS architecture lends itself to extensive reuse. The same types of elements are used in many kinds of cells, the same types of cells are used in many modules, and so on. To prevent the user from having to define the same elements and cells over and over again, the HiPerCopS IDE allows commonly-used structures to be defined in libraries.

The HiPerCopS IDE supports four different libraries, one each for elements, cells, modules, and systems. The default members of these libraries have been chosen for their frequent use in a wide variety of applications. Each library is a collection of plain-text files at a common location. The library files feature flexible syntax that is easy to understand. No experience with a particular programming language is required. This allows users to easily customize the libraries. They can edit the existing definitions or add library files of their own. Such control maximizes the efficiency of the designer and allows for extensive

Table 5.1: Basic element types

Element Type	ψ_1	ψ_0	Same as
A	$MAJ(\alpha \wedge \beta, \gamma, \delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	A
B	$MAJ(\alpha \wedge \beta, \gamma, \delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	A
C	$MAJ(\alpha \wedge \beta, \gamma, \neg\delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	C
D	$MAJ(\alpha \wedge \beta, \gamma, \neg\delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	C
E	$MAJ(\alpha \wedge \beta, \gamma, \neg\delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	C
F	$MAJ(\alpha \wedge \beta, \neg\gamma, \delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	F
G	$MAJ(\alpha \wedge \beta, \neg\gamma, \delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	F
H	$\neg MAJ(\alpha \wedge \beta, \neg\gamma, \neg\delta)$	$XOR(\alpha \wedge \beta, \gamma, \delta)$	H

customization.

Some elements, cells, modules, and systems are so common that they can be considered essential members of every HiPerCopS library. The files that describe these structures are included in every copy of the HiPerCopS IDE source distribution, and are described in Appendix A. Over time, it is expected that this common library will grow as the HiPerCopS architecture continues to develop. With this in mind, a global repository has been established to manage the common library. The repository enables users to update their local libraries whenever necessary, giving them the latest version of the common library. It also allows users to submit their custom library files to the repository, so others have the opportunity to leverage that work in their own designs. More information about the global repository can be found in Appendix A.

5.1 Element Library

In math mode, elements act as 4-input 2-output lookup tables. As shown in [12], most operations can be implemented by a small set of elements. These essential elements are listed in Table 5.1, and each of these elements is defined in the HiPerCopS element library by default. As shown in the table, this set can be reduced to the subset of element types A, C, F, and H, but all eight types are included in the library for the sake of completeness.

In the element library, the look-up table (LUT) of each element type is represented by a file with the *.lut extension. As an example, Figure 5.1 shows the contents of A.lut, which

```
#w1 = MAJ(alpha ^ beta, gamma, delta)
#w0 = XOR(alpha ^ beta, gamma, delta)
#w1,w0
0,0
0,1
0,1
1,0
0,0
0,1
0,1
1,0
0,0
0,1
0,1
1,0
0,1
1,0
1,0
1,1
```

Figure 5.1: Library file for the type A element.

Table 5.2: Truth table for the type A element.

$\alpha, \beta, \gamma, \delta$	ψ_1	ψ_0
0000	0	0
0001	0	1
0010	0	1
0011	1	0
0100	0	0
0101	0	1
0110	0	1
0111	1	0
1000	0	0
1001	0	1
1010	0	1
1011	1	0
1100	0	1
1101	1	0
1110	1	0
1111	1	1

```
#H cell

[cell]
elements: B,A,A,A
          D,Ctc,A,A
          D,A,Ctc,A
          H,F,F,E
```

Figure 5.2: Library file for the type H cell.

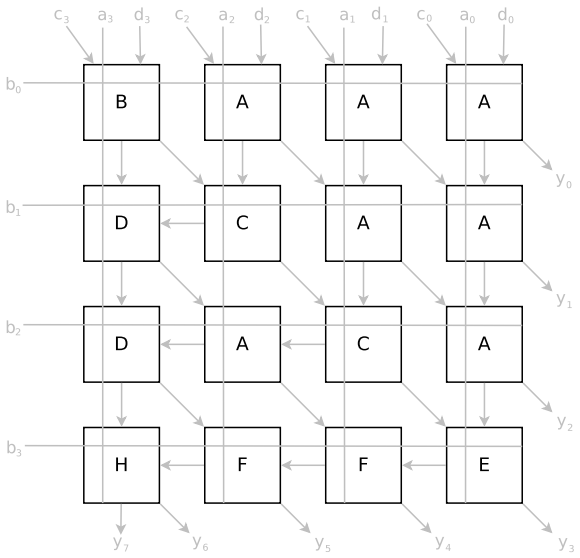


Figure 5.3: Physical layout of an H cell.

defines the type A element. Notice that the first three lines of the file begin with the '`#`' character. These indicate comments that act as notes to the user. The remaining sixteen lines define the contents of the LUT array and are written in comma-separated value (CSV) format. These rows describe the binary values to be written to the element's array, and they can be conceptualized as entries in a truth table to show the relationship between the 4-bit input and the 2-bit output (Table 5.2).

5.2 Cell Library

Since cells have fixed dimensions and routing paths, they are even easier to describe than elements. The cell library file format is very simple, consisting of comments, a '`[cell]`'

heading, an '`elements:`' section, and a CSV list of the elements. Figure 5.2 displays the contents of library file H.cell. This library entry produces a type H cell, represented graphically in Figure 5.3.

It is important to note that the entries in the CSV portion of a cell file must match filenames in the element library directory. The HiPerCopS IDE recursively defines cells by building them out of their comprising elements.

5.3 Module Library

Entries in the module library are more complex than cell library files. Since modules can be composed of different numbers of cells, and since the inter-cell routing rules can be different from module to module, module library files require more detailed information. An example of a module library file is shown in Figure 5.4, which defines a 32-bit adder.

Entries in the module library consist of a single heading and associated sections:

Module heading. Every file in the module library must include the `[module]` heading. This marks the subsequent sections as belonging to a module description.

Cells section. The `cells` section is used to define the layout of the module's cellfield as well as the individual cell types for each position in the array. The names used in this CSV list must match filenames in the cell library directory.

Names section. Cells in a module cellfield cannot be identified by their cell type alone. Often, modules contain many cells of the same type. However, since a module requires explicit rules for inter-cell routing, there needs to be some way for individual cells to be uniquely identified. The `names` section contains a CSV list of the same dimensions as the `cells` section. Its entries are comprised of unique strings that correspond to the cell located in that position. By establishing a mapping between the cell type and its unique name, the HiPerCopS IDE can associate specific routing rules with each particular cell.

```
#adder

[module]
cells: add,add,add,add
       add,add,add,add

names: 00,01,02,03
       10,11,12,13

in1:     01,00,10,11,12,13,03,02
in2:     01,00,10,11,12,13,03,02
in3:     01,00,10,11,12,13,03,02
in4:     01,00,10,11,12,13,03,02

out:   o1,o2,o3,o4,o5,o6,o7,o8,o9

routing: digraph
         {
            01 -> o1  [chunk = msb, index = 1];
            01 -> o2  [chunk = lsb, index = 1];
            00 -> 01  [chunk = msb, index = 1];
            00 -> o3  [chunk = lsb, index = 1];
            10 -> 00  [chunk = msb, index = 1];
            10 -> o4  [chunk = lsb, index = 1];
            11 -> 10  [chunk = msb, index = 1];
            11 -> o5  [chunk = lsb, index = 1];
            12 -> 11  [chunk = msb, index = 1];
            12 -> o6  [chunk = lsb, index = 1];
            13 -> 12  [chunk = msb, index = 1];
            13 -> o7  [chunk = lsb, index = 1];
            03 -> 13  [chunk = msb, index = 1];
            03 -> o8  [chunk = lsb, index = 1];
            02 -> 03  [chunk = msb, index = 1];
            02 -> o9  [chunk = lsb, index = 1];
         }
```

Figure 5.4: Library file for an adder module.

Input sections. Depending on the module, computational inputs must be applied to different cells in the cellfield. Each computational input is a bitstring that is split into substrings called 'nybbles'. For the HiPerCopS architecture, nybbles are four bits long. Each nybble is applied to a different cell in the module cellfield. The `inputs` sections contain a list of cell names which map the nybbles to the appropriate cell. The most significant nybble is applied to the first cell in the list, the second-most significant nybble is applied to the second cell in the list, and so on. Of course, the input list must have enough entries to accomodate every nybble of the input string, and vice versa.

Modules can accept multiple inputs, so module library files have a section for each input. For example, an N-input module will have sections named `i1`, `i2`, ... `iN`. For the HiPerCopS architecture, modules take four bitstrings as arguments, so $N = 4$.

Output section. Just as input configurations differ from module to module, the outputs of a cellfield do not follow a standard rule. Since this is the case, the sources and order of the output nybbles need to be defined in the module library file. The most significant nybble is listed first, the second-most significant nybble is listed second, and so on. The library file's routing specification determines which cell outputs are applied to particular output nybbles.

Routing section. The `routing` section defines the module's cellfield interconnections. Its objective is to direct the outputs of each cell into the inputs of another. Routing sections in the module library are written using HiPerDOT, a custom variant of the DOT language (see Appendix B).

For modules, each line of a routing specification contains four parts, shown in Equation 5.1. Each routing entry can map a single nybble of data from a source's output to a destination's input. To begin an entry, {`source`} contains a cell name, indicating which cell's outputs are being routed. Another cell name

is listed in the {`destination`} field to show where the input should be routed. The 'chunk' attribute specifies which nybble of the source's output to use. The value of {`nybble`} can be either 'msb' or 'lsb,' referring to the most significant or least significant bits of the output. To direct the routed nybble to a particular destination cell input, the `index` field is used. The {`abcd`} entry is given a value in the range $[0, 1, 2, 3]$, which corresponds to cell inputs a, b, c, and d, respectively.

$$\{\texttt{source}\} \longrightarrow \{\texttt{destination}\}[\textbf{chunk} = \{\texttt{nybble}\}, \textbf{index} = \{\texttt{abcd}\}]; \quad (5.1)$$

Comments. As with the other library files, comment lines are indicated by the '`#`' character. Comments can be placed anywhere in the file and will be ignored by the library parser.

5.4 System Library

Systems use entries in the module library to create powerful, complex applications. Since the HiPerCopS IDE facilitates system design at a high level, users will spend most of their time working with system library files. For this reason, the syntax and format of system library files were designed to balance simplicity and verbosity.

As shown in Figure 5.5, every system library file has two major sections: module declaration and routing. The module declaration section lists variable names that will be used in the routing section. Each variable represents a module, and the module's type is determined by the label field. A variable's label must match the filename of a module library entry. If it does not, the HiPerCopS IDE will not be able to construct the associated underlying data structures or map the module to the device.

The routing section of a system library file describes how the inputs and outputs of the system's modules are connected. Each routing entry consists of a source and a destination, with directionality indicated by an arrow (Equation 5.2). The source and destination values

```
digraph test
{
    a [label="M"];
    b [label="M"];
    c [label="+"];
    d [label="-"];
    e [label="*"];
    aout [label="M"];
    bout [label="M"];

    a -> c;
    b -> c;
    a -> d;
    b -> d;
    a -> e;
    b -> e;
    a -> aout;
    b -> bout;
}
```

Figure 5.5: Library file for a simple system.

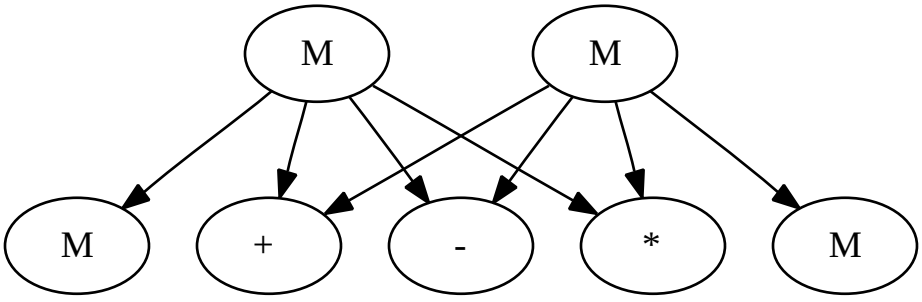


Figure 5.6: Simple system displayed as a diagram.

must correspond to a variable from the module declaration section.

$$\{\texttt{source}\} \longrightarrow \{\texttt{destination}\}; \tag{5.2}$$

Information contained in DOT files can be displayed graphically, giving the user another way to inspect their systems. For instance, the system described by the code in Figure 5.5 can be processed and displayed as the diagram in Figure 5.6.

Chapter 6

Case Study: CORDIC Implementation

To demonstrate the typical workflow of the HiPerCopS IDE, consider the design and simulation of a CORDIC unit. This description will follow the steps outlined in Figure 6.1, after some brief information about CORDIC units.

6.1 The CORDIC Unit

The CORDIC unit is a common component in digital signal processors [13]. It can be configured to perform a variety of mathematical operations, including square root, division, and rectangular-to-polar conversion. Each stage of a CORDIC computation implements Equation 6.1. A flowchart representing a single stage of the CORDIC unit is shown in Figure 6.2.

$$\begin{aligned} x' &= x \mp 2^{-i} y \\ y' &= y \pm 2^{-i} x \\ z' &= z + f(i) \end{aligned}$$

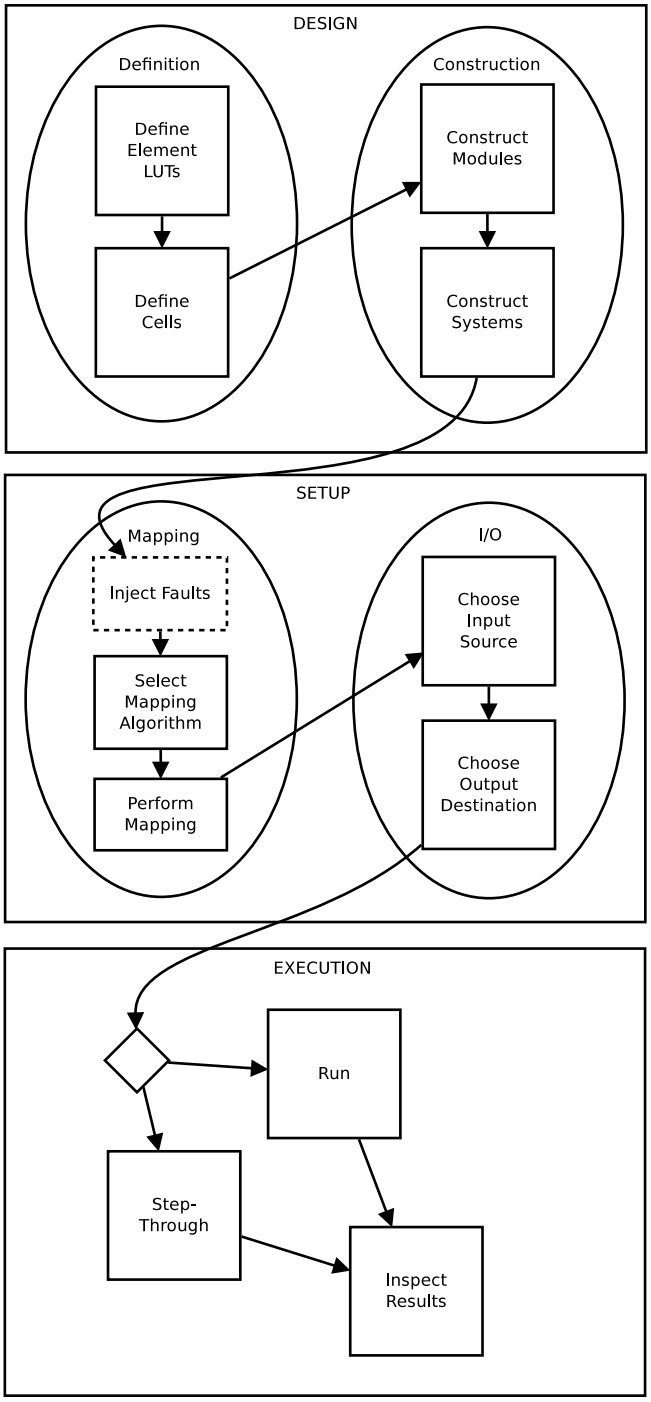


Figure 6.1: Detailed program flow for the HiPerCopS IDE

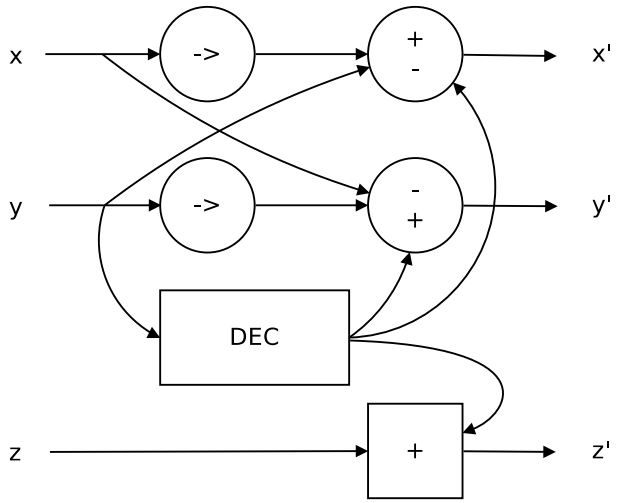


Figure 6.2: Cordic stage flowchart.

The CORDIC unit consists of six modules: two shifters, three adders, and a decoder. The shifters take bitstring inputs and shift them to the right by up to four bits. Two of the adders work on two's complement data, meaning that they can perform either addition or subtraction. The third adder is fixed to perform addition only. The decoder determines the sign of the y input and sets the adders accordingly.

6.2 Design

The design step involves the definitions of elements and cells and the construction of the modules and the system.

6.2.1 Definition

The HiPerCopS IDE library contains all of the necessary elements for constructing the shifters and the adders. It does not define the particular element needed for the decoder module, but this turns out to be a trivial problem based on the y input. If y is positive, the decoder returns '0'. If it is a negative value, the decoder returns '1'. This simple truth table can be implemented as an element library file in the HiPerCopS IDE, as shown in Figure 6.3.

```
#Decoder element for the CORDIC unit
#w1 = alpha
#w0 = alpha
#w1, w0
0,0
0,0
0,0
0,0
0,0
0,0
0,0
0,0
1,1
1,1
1,1
1,1
1,1
1,1
1,1
1,1
```

Figure 6.3: CORDIC decoder element library file, cordic_decode.lut.

```
#Decoder cell for a CORDIC unit

[cell]
elements: cordica,cordic0,cordic0,cordic0
          cordica,cordicc,cordic0,cordic0
          cordica,cordicc,cordicc,cordic0
          cordica,cordicc,cordic1,cordic0
```

Figure 6.4: CORDIC decoder cell library file, cordic_decode.cell.

```
#CORDIC decoder module

[module]
cells: cordicX

names: cell

in1:   cell
in2:   cell
in3:   cell
in4:   cell

out:   o1

routing: digraph
         {
            cell -> o1   [chunk = msb, index = 1];
            cell -> cell [chunk = lsb, index = 3];
         }
```

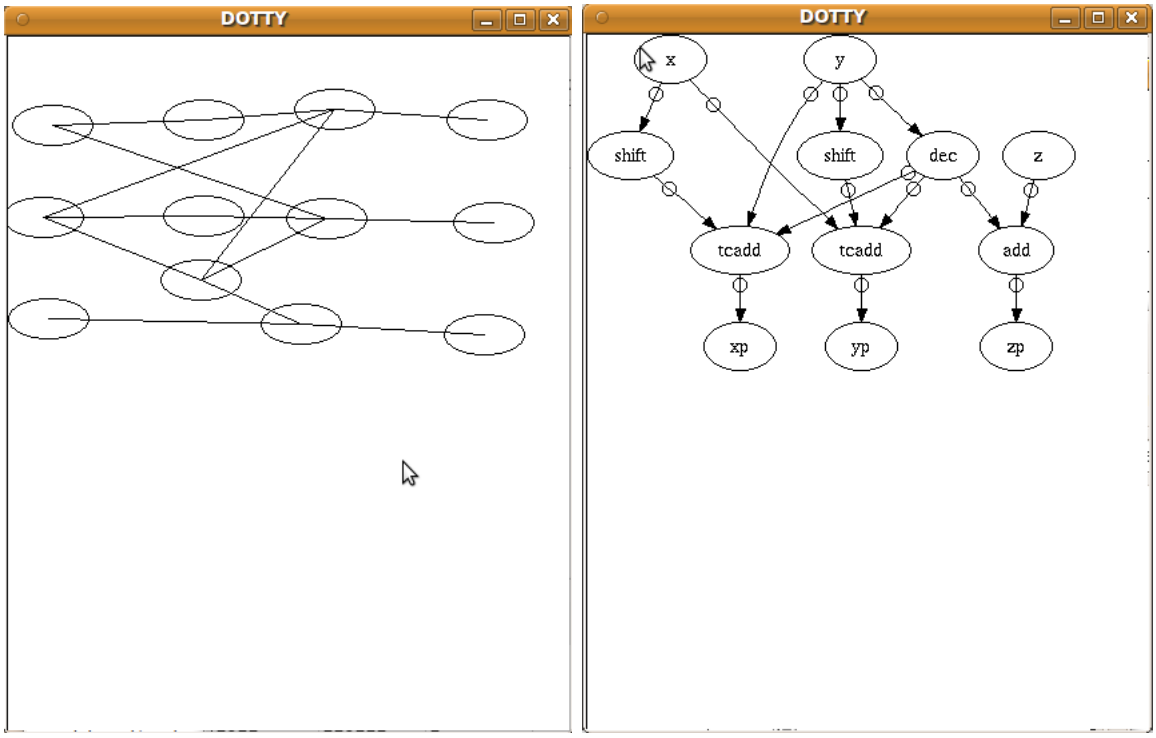
Figure 6.5: CORDIC decoder module library file, cordic_decode.module.

As with the elements, the only thing missing from the HiPerCopS cell library is an entry that performs the CORDIC decode. Since the CORDIC decoder only pays attention to a single input bit, most of the elements in the new cell can be ignored. The resulting cell is shown in Figure 6.4.

6.2.2 Construction

The HiPerCopS IDE library already has modules defined for shifting, unsigned addition, and two's complement addition. This means that the only module that needs to be created is the decoder module. In a text editor, reproduce the code shown in Figure 6.5 and save it in the module library.

To create the system file, start the HiPerCopS IDE by calling the command `python spoc_gui.py`. Then, from the menu, select Systen → New System. This will open the design window. In the design window, click the left mouse button to create three nodes down the left side of the window. Then use the right mouse button to click on the first



(a) Before layout (b) After layout

Figure 6.6: System construction

node. Select the `set attr` option, type `label=x` in the text box, and click the `OK` button. Repeat these steps for the other two nodes, labeling them `y` and `z`, respectively.

Use this tool to continue mapping out the nodes of the system. Name the nodes according to the module names defined earlier, and create three more nodes labeled `xp`, `yp`, and `zp`. These will serve as the system outputs. Use the diagram in Figure 6.2 as a guide.

To connect the nodes together, use the center mouse button to click in the middle of the source node. Then drag the mouse to the middle of the destination node and release the center mouse button. This will draw a connecting line between the two nodes. Use this method to define all of the connections in the CORDIC unit.

After this is done, right click on the canvas and select `do layout`. This will rearrange the nodes and apply the labels, shown in Figure 6.6. Notice that the result looks very similar to the CORDIC unit flowchart (Figure 6.2).

```
"x","y","z"
0,10,10011101
101,11,1010101
1011,110111,1
1011100,10011101,101011
10101010,1010101,11101
111101,1,11
0,101011,1010
11011,11101,0
1100,11,101
11111111,1010,1011
1,1111001,1011100
10000000,11111110,11111110
1,11101,11101
```

Figure 6.7: CORDIC unit simulation input.

Save the system file by right-clicking on the canvas and selecting `save graph as`. Name the system `cordic_system.dot` and close the editor. In the HiPerCopS menu, go to System → System from File and select the new system file. It will be displayed in the system pane of the IDE. By clicking on the Text tab, the source code behind the system graph can be inspected.

6.3 Setup

6.3.1 Mapping

Since there were no faults injected into this system, there is no need to select a mapping algorithm that avoids them. As a result, the naïve algorithm can be selected by choosing "naive" under Faults → Fault Avoidance Strategy.

6.3.2 Inputs and Outputs

Construct an input file, open a text editor and reproduce the contents shown in Figure 6.7. Save this file as input.csv.

To select the input file, go to Data → Input File. This will bring up a file selection

```
"x","y","z"
0,10,10011101
101,11,1010101
1011,110111,1
1011100,10011101,101011
10101010,1010101,11101
111101,1,11
0,101011,1010
11011,11101,0
1100,11,101
11111111,1010,1011
1,1111001,1011100
10000000,11111110,11111110
1,11101,11101
```

Figure 6.8: CORDIC unit simulation output.

dialog box. Navigate to your input.csv file and select it. The contents of the file will be displayed in the application's input pane.

The output file is selected under the Data → Output File menu entry. The file selection dialog box can be used to specify which file shall be overwritten by the outputs of the system. If the selected file name does not exist, it will be created.

6.4 Execution

To execute the simulation, select Execute → Run from the application menu. This will display the device map and the system outputs inside the IDE. The system outputs will also be saved to the output file specified earlier. Check the contents of that file against the data in Figure 6.8 to verify that everything worked correctly.

Chapter 7

Conclusion and Future Work

7.1 Contributions

The HiPerCopS IDE contains all of the features necessary for the design and simulation of medium-grain reconfigurable hardware systems. Its notable contributions to the research field include:

- **Ease of Use.** The IDE abstracts unnecessary information away from the user, allowing them to focus on system-level issues instead of low-level details.
- **Graphical design tools.** Although entire system designs can be performed using a simple text editor, users have the option to create and view their designs as graphs. This visual representation facilitates a smooth transition from system concept to implementation.
- **System simulation.** Previously, system simulation meant tracing a mess of bit-strings through a paper-based diagram. With the IDE, users are able to apply a set of inputs to the software representation of a system and instantly have access to the results.
- **Fault injection.** With the ability to add "stuck-at" faults to a device, users can get an idea of how their systems might perform in hostile environments.

- **Realistic performance.** The internal data structures of the IDE mimic the behavior of the physical architecture. This allows users to verify the correctness of their systems, rapidly and accurately with a high degree of confidence.

- **Libraries.** The HiPerCopS architecture uses a hierarchical approach, and the libraries use data encapsulation to take advantage of this fact. The IDE's libraries promote reuse by storing commonly used structures in a standard location.

- **Mapping algorithms.** Before the introduction of the HiPerCops IDE, all system designs had to be manually mapped to a given device. Now users can choose from a range of mapping algorithms that will perform this task automatically.

7.2 Future Work

Even though the HiPerCopS IDE is a complete software application, the addition of a few more features could greatly increase its value. These opportunities for future work include:

- **Step-through debugging.** The IDE gives users an unprecedented level of control over their system designs. However, the simulation features do not give much feedback regarding the inner workings of the system under test. It might be helpful if users had the ability to halt execution and inspect the current states of their modules, cells, and elements.

- **Multi-core support.** Currently, the HiPerCopS IDE systems can be mapped to a single device. To provide greater redundancy and reliability, it might be advantageous to support systems that span multiple devices.

- **Device programming.** The data structures of the IDE represent systems in a realistic manner, so it should be possible to implement these structures on an actual device. A interface utility could be developed to enable the IDE to program HiPerCopS hardware.

7.3 Conclusion

It is expected that the HiPerCopS IDE will be a helpful tool for researchers interested in medium-grain reconfigurable hardware. With its system-level focus and graphical design tool, the IDE will facilite more rapid application development. Hopefully, the growth rate of medium-grain reconfigurable hardware architectures will increase as a result. Since the HiPerCopS IDE was designed to be as flexible as possible, the application has the potential to grow with the technology it targets. By adding new entries to the structure libraries or making small modifications to the source code, the IDE can be used for a wide variety of architectures. Adequate design tools are critical to the success of any computer hardware, and the HiPerCopS IDE represents a step in the right direction.

Appendix A

The Standard Library

The HiPerCopS IDE Standard Library is distributed with each copy of the IDE source code. The Standard Library contains definitions for all of the elements, cells, and modules that are considered to be essential building blocks for most applications.

A.1 Elements

The element directory of the Standard Library contains all of the elements listed in Table 5.1. The truth table for each of these elements is listed here in Table A.1.

The 'Ctc', 'add', and 'empty' elements are also included in the Standard Library. The 'Ctc' element is used in multiply-accumulate cells when two's complement numbers are involved. The 'add' element is used in an unsigned addition cell, and the 'empty' cell returns '00' no matter what the input is.

A.2 Cells

Cells of type 'A', 'B', 'C', 'D', 'E', 'F', and 'H' all perform the multiply-accumulate operation, $y = (a \times b) + c + d$. Inputs a, b, c, and d are each four bits long and are in either unsigned or two's complement form, depending on the cell.

In the following descriptions, the four entries listed as the input format correspond to

Table A.1: Elements in the Standard Library.

$\alpha\beta\gamma\delta$	A	B	C	D	E	F	G	H	Ctc	add	empty
0000	00	00	00	00	00	00	00	00	00	00	00
0001	01	01	01	01	01	11	11	11	11	01	00
0010	01	01	11	11	11	01	01	11	01	01	00
0011	10	10	00	00	00	00	00	10	00	10	00
0100	00	00	00	00	00	00	00	00	11	01	00
0101	01	01	01	01	01	11	11	11	01	10	00
0110	01	01	11	11	11	01	01	11	00	10	00
0111	10	10	00	00	00	00	00	10	00	11	00
1000	00	00	00	00	00	00	00	00	11	00	00
1001	01	01	01	01	01	11	11	11	01	01	00
1010	01	01	11	11	11	01	01	11	00	01	00
1011	10	10	00	00	00	00	00	10	11	10	00
1100	01	01	11	11	11	11	11	01	11	01	00
1101	10	10	00	00	00	10	10	00	10	10	00
1110	10	10	10	10	10	00	00	00	00	10	00
1111	11	11	11	11	11	11	11	11	11	11	00

the four cell inputs. A + indicates that the input is in unsigned form, while a − indicates that the input is in two's complement form.

The two entries of the output format represent the upper and lower halves of the cell output. These halves can also be in either unsigned or two's complement form, represented by the + and − symbols, respectively.

A

Input format: $(+, +, +, +)$

Output format: $(+, +)$

A	A	A	A
A	A	A	A
A	A	A	A
A	A	A	A

B

Input format: $(-, +, -, -)$

Output format: $(-, -)$

B	A	A	A
D	Ctc	A	A
D	A	Ctc	A
D	A	A	Ctc

C

Input format: $(+, +, +, -)$

Output format: $(+, -)$

C	A	A	A
A	Ctc	A	A
A	A	Ctc	C
A	A	A	Ctc

D

Input format: $(-, +, -, +)$

Output format: $(-, +)$

D	A	A	A
D	A	A	A
D	A	A	A
D	A	A	A

E

Input format: $(+, -, -, +)$

Output format: $(-, +)$

G	A	A	A
A	Ctc	A	A
A	A	Ctc	A
F	F	F	E

F

Input format: $(+, -, +, -)$

Output format: $(-, +)$

C	A	A	A
A	Ctc	A	A
A	A	Ctc	A
F	F	F	E

H

Input format: $(-, -, -, -)$

Output format: $(-, +)$

B	A	A	A
D	Ctc	A	A
D	A	Ctc	A
H	F	F	E

O

The type 'O' cell is simply an empty cell. Its elements' lookup tables are filled with zeros. The type O cell can be used to represent an unconfigured cell.

empty	empty	empty	empty
empty	empty	empty	empty
empty	empty	empty	empty
empty	empty	empty	empty

add

The 'add' cell is used to perform unsigned addition. To see how it is used, consider the adder module in the next section.

A	A	A	add
A	A	A	add
A	A	A	add
A	A	A	add

A.3 Modules

Add

The add module is a 32-bit unsigned adder, built using cell types A and add.

```
#adder

[module]
cells: add,add,add,add
       add,add,add,add

names: 00,01,02,03
       10,11,12,13

in1:     01,00,10,11,12,13,03,02
in2:     01,00,10,11,12,13,03,02
in3:     01,00,10,11,12,13,03,02
in4:     01,00,10,11,12,13,03,02

out:   o1,o2,o3,o4,o5,o6,o7,o8,o9

routing: digraph
         {
            01 -> o1  [chunk = msb, index = 1];
            01 -> o2  [chunk = lsb, index = 1];
            00 -> 01  [chunk = msb, index = 1];
            00 -> o3  [chunk = lsb, index = 1];
            10 -> 00  [chunk = msb, index = 1];
            10 -> o4  [chunk = lsb, index = 1];
            11 -> 10  [chunk = msb, index = 1];
```

```
            11 -> o5  [chunk = lsb, index = 1];
            12 -> 11  [chunk = msb, index = 1];
            12 -> o6  [chunk = lsb, index = 1];
            13 -> 12  [chunk = msb, index = 1];
            13 -> o7  [chunk = lsb, index = 1];
            03 -> 13  [chunk = msb, index = 1];
            03 -> o8  [chunk = lsb, index = 1];
            02 -> 03  [chunk = msb, index = 1];
            02 -> o9  [chunk = lsb, index = 1];
        }
```

MAC

The two's complement MAC module performs the multiply-accumulate function, $Y = (a \times b) + c + d$, on binary strings in two's complement format. It utilizes all of the MAC cells in the Standard Library.

```
#MAC

[module]
cells: B,A,A,A
       D,C,A,A
       D,A,C,A
       H,F,F,E

names: 00,01,02,03
       10,11,12,13
       20,21,22,23
       30,31,32,33

in1:     00,01,02,03
       10,11,12,13
       20,21,22,23
       30,31,32,33
in2:     30,20,10,00
       31,21,11,01
       32,22,12,02
       33,23,13,03
in3:     00,01,02,03
in4:     00,01,02,03

out:   o1,o2,o3,o4,o5,o6,o7,o8
```

```
26
27  routing: digraph
28           {
29              00 -> 10  [chunk = msb, index = 2];
30              00 -> 11  [chunk = lsb, index = 2];
31              01 -> 11  [chunk = msb, index = 3];
32              01 -> 12  [chunk = lsb, index = 2];
33              02 -> 12  [chunk = msb, index = 3];
34              02 -> 13  [chunk = lsb, index = 2];
35              03 -> 13  [chunk = msb, index = 3];
36              03 -> o8  [chunk = lsb, index = 1];
37              10 -> 20  [chunk = msb, index = 2];
38              10 -> 21  [chunk = lsb, index = 2];
39              11 -> 10  [chunk = msb, index = 3];
40              11 -> 22  [chunk = lsb, index = 2];
41              12 -> 22  [chunk = msb, index = 3];
42              12 -> 23  [chunk = lsb, index = 2];
43              13 -> 23  [chunk = msb, index = 3];
44              13 -> o7  [chunk = lsb, index = 1];
45              20 -> 30  [chunk = msb, index = 2];
46              20 -> 31  [chunk = lsb, index = 2];
47              21 -> 20  [chunk = msb, index = 3];
48              21 -> 32  [chunk = lsb, index = 2];
49              22 -> 21  [chunk = msb, index = 3];
50              22 -> 33  [chunk = lsb, index = 2];
51              23 -> 33  [chunk = msb, index = 3];
52              23 -> o6  [chunk = lsb, index = 1];
53              30 -> o1  [chunk = msb, index = 1];
54              30 -> o2  [chunk = lsb, index = 1];
55              31 -> 30  [chunk = msb, index = 3];
56              31 -> o3  [chunk = lsb, index = 1];
57              32 -> 31  [chunk = msb, index = 3];
58              32 -> o4  [chunk = lsb, index = 1];
59              33 -> 32  [chunk = msb, index = 3];
60              33 -> o5  [chunk = lsb, index = 1];
61           }
```

A.4 Repository

Over time, the Standard Library may grow to include more entries. To allow users to update their library files, the Standard Library is stored in an online repository. This repository

can be accessed using following Subversion command:

```
svn checkout http://hipercops-washington.googlecode.com/svn/trunk/lib
standardlib
```

This will download the latest versions of all of the Standard Library entries and store them in a directory named `standardlib`. For more information on Subversion, see Appendix C.

Appendix B

The HiPerDOT Language

HiPerDOT is a specialized subset of the DOT language. DOT is covered by the Common Public License (CPL) [14] and is defined on the Graphviz website [15].

B.1 Grammar

The following is an abstract grammar defining the HiPerDOT language. Terminals are shown in bold font and nonterminals in italics. Literal characters are given in single quotes. Parentheses (and) indicate grouping when needed. Square brackets [and] enclose optional items. Vertical bars | separate alternatives.

graph : **digraph**) [*ID*] '**{**' *stmt_list* '**}**'
stmt_list : [*stmt* ['**;**'] [*stmt_list*]]
stmt : *node_stmt*
| *edge_stmt*
| *attr_stmt*
| *ID* '**=**' *ID*
attr_stmt : (**graph** | **node** | **edge**) *attr_list*
attr_list : '**[**' [*a_list*] '**]**' [*attr_list*]
a_list : *ID* ['**=**' *ID*] ['**,**'] [*a_list*]
edge_stmt : (*node_id* | *subgraph*) *edgeRHS* [*attr_list*]
edgeRHS : *edgeop* (*node_id* | *subgraph*) [*edgeRHS*]
node_stmt : *node_id* [*attr_list*]
node_id : *ID*
subgraph : [**subgraph** [*ID*]] '**{**' *stmt_list* '**}**'

The keywords **node**, **edge**, and **digraph** are case-independent.

An *ID* is any string of alphanumeric([a-zA-Z0-9]) characters or underscores ('_'), not beginning with a digit.

An *edgeop* is ->in directed graphs and - - in undirected graphs.

An *a_list* clause of the form *ID* is equivalent to *ID*=`true`.

The language supports C++-style comments: /* */ and //. In addition, a line beginning with a '#' character is considered a line output from a C preprocessor (e.g., # 34 to indicate line 34) and discarded.

Semicolons aid readability but are not required. Also, any amount of whitespace may be inserted between terminals.

As another aid for readability, HiPerDOT allows single logical lines to span multiple physical lines using the standard C covention of a backslash immediately preceeding a newline character. In addition, double-quoted strings can be concatenated using a '+' operator. As HTML strings can contain newline characters, they do not support the concatenation operator.

B.2 Semantic Notes

If a default attribute is defined using a **node**, **edge**, or **graph** statement, or by an attribute assignment no attached to a node or edge, any object of the appropriate type defined afterwards will inherit this attribute value. This holds until the default attribute is set tot a new value, from which point the new value is used. Objects defined before a default attribute is set will have an empty string value attached to the attribute once the default attribute definition is made.s

If an edge belongs to a cluster, its endpoints belong to that cluster. Thus, where you put an edge can effect a layout, as clusters are sometimes laid out recursively.

B.3 Character encodings

The SpocDOT language assumes the alphanumeric character set. DOT is more flexible in this regard, and SpocDOT may be expanded in the future to accomodate more character sets.

Appendix C

Installation

The HiPerCopS IDE is a cross-platform application. It will run on any operating system that supports Python. The IDE was primarily developed on Ubuntu Linux, but some testing was performed under Windows XP and Mac OS X. The portability of the HiPerCopS IDE is an important feature because it makes the application accessible to a wide array of potential users.

Source Code

The source code for the HiPerCopS IDE is located in a public Subversion repository. To access this repository, first install Subversion [16]. The following command will install the latest version of the source in a directory named `hipercops-ide`:

```
svn checkout http://hipercops-washington.googlecode.com/svn/trunk/
hipercops-ide
```

Dependencies

The HiPerCopS IDE relies on the following software packages:

- Python 2.6 [17]
- wxPython [18]
- Pydot [19]

Installation Script

For Windows and Mac OS, the dependencies can be installed manually from their respective websites. For Debian-based operating systems like Ubuntu, an installation script is included with the source code. From the `hipercops-ide` directory, run the command:

```
sudo scripts/installdeps
```

Execution

The HiPerCopS IDE is installed once the source is downloaded and the dependencies are satisfied. To start the IDE, execute the following command from the `hipercops-ide` directory:

```
./hipercops-ide
```

Bibliography

[1] J. G. Delgado-Frias, M. J. Myjak, F. L. Anderson, and D. R. Blum, "A medium-grain reconfigurable cell array for dsp," in *IASTED International Conference on Circuits, Signals, and Systems*.

[2] M. Myjak, *A medium-grain reconfigurable architecture for digital signal processing*. PhD thesis, Washington State University, 2006.

[3] M. Myjak and J. Delgado-Frias, "A two-level reconfigurable architecture for digital signal processing," in *Proc. 2003 International Conference on VLSI*.

[4] M. Myjak and J. G. Delgado-Frias, "Medium-grain cells for reconfigurable dsp hardware," *IEEE Transactions on Circuits and Systems, I: Fundamental Theory and Applications*, 2007.

[5] A. Widjaja and J. G. Delgado-Frias, "A high-performance unicast configuration scheme for an h-tree based reconfigurable hardware," in *48th IEEE Int. Midwest Symposium on Circuits and Systems*.

[6] J. K. Larson, "Cad tool emulation for a two-level reconfigurable cell," Master's thesis, Washington State University, 2005.

[7] K. Robinson and J. G. Delgado-Frias, "Fault avoidance in medium-grain reconfigurable hardware architectures," in *International Conference on Engineering of Reconfigurable Systems and Algorithms*.

[8] *Modelling and simulation: exploring dynamic system behaviour*. Springer, 2007.

[9] *Modern Structured Analysis*. Prentice Hall, 1989.

[10] A. Aviienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, 2004.

[11] K. Robinson, "Fault recovery mechanisms for medium-grain reconfigurable hardware architectures.".

[12] M. J. Myjak and J. G. Delgado-Frias, "Pipelined multipliers for reconfigurable hardware," in *Proc. 11th Reconfigurable Architectures Workshop.*

[13] M. Myjak and J. G. Delgado-Frias, "A medium-grain reconfigurable architecture for dsp: Vlsi design, benchmark mapping, and performance," *IEEE Transactions on VLSI Systems*, 2008.

[14] "Common public license." http://www.opensource.org/license/cpl1.0.php.

[15] "Graphviz." http://www.graphviz.org/.

[16] "Subversion." http://subversion.apache.org/.

[17] "Python." http://www.python.org/.

[18] "wxpython." http://www.wxpython.org/.

[19] "Pydot." http://code.google.com/p/pydot/.

[20] M. Myjak and J. G. Delgado-Frias, "A two-level reconfigurable cell array for digital signal processing," *The Microelectronic Journal*, 2007.

[21] M. J. Myjak, J. K. Larson, and J. G. D. Frias, "Mapping and performance of dsp benchmarks on a medium-grain reconfigurable architecture," in *International Conference on Engineering of Reconfigurable Systems and Algorithms.*

[22] M. J. Myjak and J. G. Delgado-Frias, "Superpipelined reconfigurable hardware for dsp," in *2006 IEEE Symposium on Circuits and Systems.*

[23] M. J. Myjak and J. G. Delgado-Frias, "A bit-serial cell for reconfigurable dsp hardware," in *48th IEEE Int. Midwest Symposium on Circuits and Systems.*

[24] M. J. Myjak and J. G. Delgado-Frias, "A symmetric differential clock generator for bit-serial hardware," in *The 2005 International Conference on Computer Design.*