A SYSTEM FOR THE IMPLEMENTATION OF IMAGE PROCESSING ALGORITHMS ON CONFIGURABLE COMPUTING HARDWARE

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Benjamin Alexander Levine
August 1999

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Don Bouldin, for his support and guidance in this project. Without him, this work could never have been completed. I would also like to thank Dr. Bouldin for introducing me to the idea of configurable computing, an area I find very interesting and in which I hope to pursue further research. I am finally thankful to Dr. Bouldin for encouraging me to explore opportunities beyond the University of Tennessee, and for putting my best interests ahead of his own.

I have enjoyed working with the rest of the CHAMPION team, especially Dr. Chandra Tan, Dr. Danny Newport, and Mr. Senthil Natarajan. All of their efforts and assistance have contributed greatly to this work and to the CHAMPION project as a whole. I wish them all good fortune in their future endeavors. I would also like to acknowledge the Defense Advanced Research Projects Agency for their support of this research under grant F33615-97-C-1124.

Finally, I would like to thank my wife, Ruth, for her support and encouragement, and for being understanding about the long hours I have had to spend on this project. She has, in large part, enabled me to complete this thesis, and my research at the University of Tennessee, and to move on to new challenges.

ABSTRACT

Configurable computing hardware has many advantages over both general-purpose processors and application specific hardware. However, the difficulty of using this type of hardware has limited its use. An automated system for implementing image-processing applications in configurable hardware, called CHAMPION, is under development at the University of Tennessee. CHAMPION will map applications in the Khoros Cantata graphical programming environment to hardware. A relatively complex automatic target recognition (ATR) application was manually mapped from Cantata to a commercially available configurable computing platform. This manual implementation was done to assist in the development of function libraries and hardware for use in the CHAMPION systems, as well as to develop procedures to perform the application mapping. The mapping techniques used were developed in such a way that they could serve as the basis for the automated system. Many important considerations for the mapping process were identified and included in the mapping algorithms.

The manual mapping was successful, allowing the ATR application to be run on a Wildforce-XL configurable computing board. The successful application implementation validated the basic hardware design and mapping concepts to be used in CHAMPION. Nearly a tenfold performance increase was realized in the hardware implementation and performance bottlenecks were identified which should enable even greater performance improvements to be realized in the automated system. The manual implementation also helped to identify some of the challenges that must be overcome to complete the development of the automated system.

Table of Contents

I. INTRODUCTION 1
A. Motivation 1
B. Research Objectives 6

II. BACKGROUND 10
A. Configurable Computing 10
1) Hardware 10
2) Applications 13
3) Development Software 15
B. Khoros Cantata 17
C. CHAMPION 20

III. PROJECT GOALS 23
A. START Application 23
1) First Phase 27
2) Second Phase 30
B. Wildforce Board 32
1) Xilinx FPGAs 33
2) Board Architecture 35

IV. IMPLEMENTATION 39
A. Cantata Implementations of START 39
1) Standard Cantata 39
2) Hardware Equivalent Cantata 42
B. Glyph Libraries 44
1) Khoros Cantata Glyphs 45
2) Hardware Equivalent Glyphs 45
3) Hardware Glyphs 47
C. Differences Between Cantata and Hardware 50
1) Data Transfer 50
2) Data Synchronization 51
3) Data Conversion/Operator Sizing 54
D. Mapping Procedure 55
1) Workspace Conversion 55
2) Data Sizing and Glyph Selection 57
3) Data Synchronization 60
4) Partitioning 63
a) Background 63
b) Constraints 64
c) Partitioing Algorithm 66
5) Synthesis 67
6) Host Code Generation 68

V. RESULTS 71
A. Configurations 71
B. Application Performance 78
1) Target Recognition 78
2) Processing Time 80

VI. CONCLUSIONS 82
A. Glyph Libraries 82
B. Mapping Procedure 84
C. Performance 85
D. Future Work 86

REFERENCES 88

List of Tables

Table II-1.	*Comparison of Four Xilinx FPGA-based CCMs.*	11
Table III-1.	*Resources Available in FPGAs Used on Wildforce-XL.*	36
Table IV-1.	*Glyphs Used in Standard Cantata Implementation of START.*	40
Table IV-2.	*Glyphs Used in Hardware Equivalent Cantata Implementation of START*	43
Table IV-3.	*Hardware Glyph Control Line Combinations and Meanings*	49
Table V-1.	*Hardware Resource Usage for Each FPGA Configuration.*	73
Table V-2.	*Results from Cantata and Hardware Implementations of START*	79

List of Figures

Figure II-1.	*Khoros Cantata Glyphs and Command Line Equivalent.*	19
Figure III-1.	*Sample Images from START Algorithm.*	25
Figure III-2.	*First Phase of START Algorithm.*	28
Figure III-3.	*Second Phase of START Algorithm.*	31
Figure III-4.	*Xilinx 4000 Series FPGA Configurable Logic Block.*	34
Figure III-5.	*Basic Wildforce-XL Block Diagram.*	37
Figure III-6.	*Wildforce-XL Board As Used.*	38
Figure IV-1.	*Data Synchronization Problem and Solution.*	53
Figure IV-2.	*Example Hardware Equivalent Cantata Workspace – Low Pass Filter.*	56
Figure IV-3.	*Graph Form of Low-Pass Filter Workspace.*	57
Figure IV-4.	*Data Alignment Problem in Low-Pass Filter Graph.*	60
Figure IV-5.	*Low-Pass Filter Graph with Data Widths and Glyph Types Annotated.*	61
Figure IV-6.	*Partial Time Annotation for Low-pass Filter Graph.*	62
Figure IV-7.	*Low-pass Filter Graph with Complete Time Annotation.*	63
Figure IV-8.	*Pseudo-Code for START Host Program.*	70
Figure V-1.	*FPGA Programming for each Board Configuration.*	72
Figure V-2.	*First Board Configuration.*	75
Figure V-3.	*Second Board Configuration.*	76
Figure V-4.	*Third Board Configuration.*	77
Figure V-5.	*Fourth Board Configuration.*	78
Figure V-6.	*Breakdown of Image Processing Time for Hardware Implementation*	81

I. INTRODUCTION

A. Motivation

Since the first electrical computing devices were created in the 1940s, an ever-increasing variety of computations has been done by electronic hardware. The earliest computers calculated artillery tables and decrypted codes for the military. There is now virtually no type of computation that is not performed by computer at least some of the time. Clearly, in the intervening years an extremely large number of new applications for computing hardware have been developed. The calculations performed in these applications range from those that were previously done by hand, such as accounting and scientific calculations, to entirely new sorts of computations that can only be done by computer, such as digital image processing and complex simulations. Regardless of the type of computation being done, the same basic operations are used ubiquitously. For example, arithmetic operations, such as addition and multiplication, and logical operations, such as Boolean ANDs and ORs, are used for virtually all types of calculations. While the same operations are used repeatedly, the type of hardware performing these basic operations can vary widely.

For many years, electronic hardware used for computation could be divided into two main types, general purpose, and application specific. General-purpose hardware is exemplified by microprocessors such as the Intel 80x86 family and the Motorola 68000 family, which serve as the main processing unit in most personal computers. The architecture of these devices is fixed and includes specific hardware to implement a limited, pre-defined, set of instructions. These microprocessors run programs, which are lists of instructions to be executed that are stored in external memory. New programs can be loaded into memory from disk or other storage as needed. The software program determines the computation to be done, not the hardware. Because their instruction sets include very general operations such as arithmetic and logical operators, branching and recursion, and memory access, this type of hardware can be programmed to perform any conceivable function. In fact, as the work of Church and Turing shows, *all* computations can be performed by such a general-purpose computer. However, general-purpose computers can be very slow at performing certain kinds of operations, such as those involving floating-point calculations or complex mathematical functions. For this reason, most modern computers have one or more coprocessors, which are application specific hardware that performs certain functions very quickly. Examples include math coprocessors that perform floating-point calculations and other mathematical operations, and graphics coprocessors that perform 3D rendering.

Application-specific computing hardware performs functions very quickly, but the price of this speed is limited flexibility. As their name implies, this type of hardware can only perform one function, or a group of closely related functions. The hardware determines the type of computation to be done. They cannot be reprogrammed to perform entirely new functions that were not anticipated and included in the original design. If application specific hardware is needed to perform a new function, then a new hardware design will have to be created. Since this type of computation hardware is generally implemented as carefully designed Application Specific Integrated Circuits (ASICs), creating a new design takes a great deal of effort and knowledge. Since they are custom ICs, they are also very expensive to fabricate, and it takes week or months to design a new ASIC and have it fabricated. Therefore, application specific hardware is only useful if the functions needed are known in advance and the requirements of the functions they perform are not expected to change during the useful lifetime of the hardware.

In spite of these drawbacks, application specific hardware is widely used whenever speed is an important design consideration. By structuring the hardware to match the problem, application specific hardware can often achieve computation speeds several orders of magnitude faster than general-purpose hardware. This high level of performance is obtained by utilizing several techniques, including the performance of operations in parallel; the organization of hardware for efficient data transfer, which reduces delays while data is moved from place to place; and the utilization of hardware structures that permit efficient data scheduling, reducing inefficiencies introduced when computation is halted to wait for new data.

In recent years, a new class of computing hardware has been gaining increasing research interest. Configurable computing hardware has some of the advantages of both general-purpose and application-specific hardware. This type of hardware may be based on commercially available Field Programmable Gate Arrays (FPGAs), or on ICs designed specifically for the purpose. In either case, this type of hardware consists of a relatively large number of functional units with programmable interconnections. The functionality of the hardware is determined by how the interconnections between functional units are configured, and in most, but not all, architectures, how the functional units themselves are configured. By changing the configuration, the hardware can be made to perform a completely different function. Since the configuration is specific to the application at hand, it is in effect a custom computer for the particular design. For this reason, computing devices using configurable hardware are often called Custom Computing Machines (CCMs).

The functional units in a CCM are usually relatively simple logical functions equivalent to a few gates, or a few tens of gates. Some configurable computing hardware has slightly more complex functional units such as small Arithmetic Logic Units (ALUs). The functional units may be fixed in function, or their functionality may be configurable. For instance, many FPGAs use Look-Up Tables (LUTs), for which the output (or outputs) for every possible input can be programmed. In this way, LUTs can implement any possible logical function of the inputs, giving great flexibility. Most architectures also include flip-flops in each functional unit, to register the outputs and allow for synchronous designs.

The interconnections between functional units provide much of the utility of most configurable computing architectures. Programmable switches can selectively connect tracks wired to the inputs and outputs of the functional blocks, allowing the desired functions to be implemented. These interconnections are internal to the IC; if multiple ICs are used in a CCM, then there are usually programmable interconnections between the ICs as well.

Since the structure of the hardware has effectively been changed for the specific function to be implemented, many types of computations can be performed by CCMs at speeds close to those obtained using application specific hardware. In addition, the configuration can be changed relatively quickly from one function to another, giving some of the same flexibility as general-purpose hardware. In effect, configurable computing allows the benefits of both application-specific and general-purpose computing to be present in the same hardware.

Despite these advantages, configurable computing hardware has not been widely used. Perhaps the main reason is the difficulty involved in configuring this hardware to perform a specific function. A typical CCM may consist of several interconnected FPGAs, as well as memory and other components. In order to map an application to this hardware, the designer must first design the hardware configuration needed to perform the necessary functions. This is usually done with either schematic capture, or increasingly with a Hardware Description Language (HDL) such as VHDL or Verilog. In either case, the designer must understand digital design and be able to separate an application into data processing and control elements. The design must then be partitioned spatially, so that the design is spread across the resources available on the FPGAs. If the design does not fit in the available FPGAs, then it must also be partitioned temporally, by allocating functional units to different configurations of the same FPGA. In addition, any configuration of the CCM itself must be determined, and a program to communicate with a host computer must usually also be created. While commercial tools exist to help with parts of this process, it still requires a great deal of skill, knowledge of digital design, time, and effort. These steep requirements have severely limited the potential users of this type of hardware and prevented its widespread acceptance.

The development of methods to allow CCMs to be used more easily will make their advantages available to more users. Ideally, tools will be developed to allow users with no knowledge of digital design to use CCMs, just as users who know nothing about the design of a microprocessor can use computers. The less important knowledge of the underlying hardware is, the larger the pool of potential users of CCMs will be. The current pool of users consists almost exclusively of researchers and digital hardware engineers. If the use of CCMs is to spread beyond this limited group, then much more research into better tools to use them will have to be done.

B. Research Objectives

Research at the University of Tennessee is currently underway to develop an automated system for mapping image processing applications in a graphical programming environment called Khoros Cantata to configurable computing hardware. It is expected that this system, called CHAMPION, will help make the power of configurable hardware more accessible to users who lack digital design experience. It is also expected that the system will allow new applications to be implemented in much less time than is required now, since many portions of application mapping that must currently be done manually will be automated. The work described in this thesis is a part of the overall research being conducted as part of the CHAMPION project.

The author's research concerns the manual mapping of a complex application running in Khoros Cantata to a commercially available CCM, the Annapolis Micro Systems Wildforce-XL. A systematic method of doing this application mapping was developed. The manual mapping used a library of parameterizable hardware modules developed concurrently for CHAMPION. The entire manual mapping was done with the goals of CHAMPION considered at every step. The resulting implementation was thus quite different from that which would have been developed if the only goal were to implement the application in hardware. However, the goal of the research was not just to simply implement the application in hardware, but to provide information on the requirements of the implementation to the developers of the automated system.

The proposed benefits of performing a manual mapping were fourfold. First, to help develop application-mapping methods that could be used to help design the automated system by determining the requirements of the system and possible methods by which the mapping could be performed. Second, to help guide the library development process by showing what types of library modules would be needed and what the requirements of those modules should be. Third, to show that the mapping methods and library modules developed could be used to implement a significant application. Fourth, the successful implementation of the application would give researchers measurements of the time required to manually map the application and the performance of a manually mapped implementation.

The process of mapping an application from software to hardware is a complicated one and there are many factors that had to be considered at each step of the process. Some of these were readily apparent, such as ensuring that the hardware chosen implements the same operations as the software application. Others were not considered until the mapping process had begun. By performing a mapping manually, all of the steps needed could be determined, and all of the factors that needed to be considered could be identified. The mapping process was successful, showing that the methods determined in the mapping process were viable, and could be used as the basis for an automated system. The manual mapping also clearly delineated the requirements for an automated system to perform the mapping process.

There are hundreds of functions available in the Khoros Cantata toolboxes and in publicly available extensions. Since not all of these could or should be included in the CHAMPION system, decisions had to be made as to which functions would be needed. Some of the decisions were obvious; simple arithmetic operators such as addition, and the basic Boolean operators should definitely be included in a system such as CHAMPION. Other choices were not as obvious. By determining the types of functions needed to implement a real application, better decisions as to which functions to include could be made. The implementation process also helped determine the requirements of the hardware implementing these functions. Design details of the hardware modules such as control lines and I/O ports were determined to large extent by what would be necessary to make the initial application work.

In order to evaluate the effectiveness of the automated system, some idea of the time required to do a manual mapping is necessary. Only by having information on the time required for a manual implementation can the productivity gains provided by CHAMPION be measured. The manual implementation also provided information on the performance of the application, in terms of the rate at which data can be processed. While the performance of the applications mapped by the automated system is not a primary goal of the project, it is still valuable to be able to measure any differences in performance between the results obtained by CHAMPION and the results obtained by a human designer.

The manual implementation did provide the type of information that was expected, as discussed above. This information should prove useful in the development of the automated system. However, the development of the automated system is mostly outside of the scope of the author's research and as such, will not be discussed in detail in this paper. It is to be expected that later publications by others working on the CHAMPION system will provide more information on its development.

This chapter presented an introduction to the research covered by this thesis. Chapter II provides background information on configurable computing, Khoros Cantata, and the CHAMPION project. Chapter III discusses the application to be mapped and the CCM used to implement it. Chapter IV discusses the application mapping process, including the libraries and procedures used. Chapter V gives the results of the hardware implementation, including specific information on the final implementation and its performance, as well as the time taken to complete it. Chapter VI presents overall conclusions about the research.

II. BACKGROUND

A. Configurable Computing

To date, most research in configurable computing can be divided into three main areas. The first has been the development of configurable computing hardware. The second has been the implementation of various types of applications on configurable computing hardware. The third has been the creation of software to assist in the implementation of applications on configurable computing hardware. All three of these areas have helped determine which types of applications are best suited to implementation on CCMs, and the benefits of running these applications on this type of hardware as compared to either general-purpose or application-specific hardware. Portions of each of these areas are relevant to the current research and will be discussed in the following sections.

1) Hardware

Numerous configurable computing machines have been built by research groups. More recently, commercial vendors have begun producing CCMs. While some research groups have designed their own ICs for configurable computing, (e.g., PipeRench [1] and RaPiD [2]), most CCMs have been constructed using commercially available FPGAs. A list of dozens of FPGA-based CCMs is given in [3]. Some of the more significant FPGA-based CCMs include Splash 2 [4], DECPeRLe-1 [5], and the Virtual Computer [6]. Versions of these CCMs, and CCMs derived directly from them, are used today by many CCM researchers.

Many different CCMs have been built with the same FPGAs, especially those produced by Xilinx, Inc. of San Jose, California. Despite their use of the same FPGAs, CCMs can vary widely in a number of other specifications. CCMs can vary in the number and size of FPGAs used; the number of interconnections between the FPGAs and whether or not these interconnections are configurable; the topology of the interconnections; whether or not other resources are available in the CCM, such as RAM or DSP chips; and how the CCM connects to an external host computer. Table II-1 shows some of the specifications for several different CCMs that use Xilinx FPGAs and illustrates some of the differences between them.

Much of the research in CCMs has attempted to determine what kinds of CCM architectures are most efficient, fastest, cheapest, and so forth. The answer to these questions usually varies depending on the characteristics of the applications that are being run. For instance, some applications require the frequent storage of intermediate

Table II-1. *Comparison of Four Xilinx FPGA-based CCMs.*

	DECPeRLe-1	Splash 2 (per array board; up to 16 array boards may be used)	Virtual Computer	Wildforce-XL (as used at UT-Knoxville)
FPGA Family	Xilinx XC3000	Xilinx XC4000	Xilinx XC4000	Xilinx XC4000XL
Number of FPGAs (available for computation)	16 x XC3090	17 x XC4010	52 x XC4010	4 x XC4013XL 1 x XC4036XL
Total Gate Equivalents	96,000	170,000	520,000	88,000
RAM	4 MB	8.5 MB	8 MB	640 KB
Interconnection Topology	4 x 4 Matrix	Linear Array with Programmable Crossbar	Spherical	Linear Array with Programmable Crossbar
Host Interface	TurboChannel	SUN S-Bus	SUN S-Bus	PCI Bus

results in RAM. If these applications were implemented on a hypothetical CCM, denoted CCM-A, which has relatively low RAM access bandwidth, than the performance would be lower than if they were run on another hypothetical CCM, denoted CCM-B, which has higher RAM access bandwidth. However, CCM-B might achieve its higher RAM access bandwidth by devoting more I/O pins on each FPGA to memory interfacing, leaving fewer available pins for interconnection between FPGAs. If CCM-A and CCM-B both have the same number and type of FPGAs, then all else being equal, CCM-B will have less connectivity between FPGAs than CCM-A. Another class of applications might require many interconnections between functional units. These applications would fit better into CCM-A than into CCM-B, using computation resources more efficiently in the former. For these and other obvious reasons, there can be no one CCM architecture that is ideal for all applications.

However, there are CCMs with architectures that seem to work relatively well for a range of problems. CCMs are now available from commercial vendors such as Annapolis Micro Systems and the Virtual Computer Company. These systems offer many of the same features found on research CCMs including: multiple FPGAs on a single board, state readback of processing elements for diagnostic purpose, direct access to RAM from each FPGA, host I/O through a fast interface bus such as PCI or VME, and pre-developed APIs for host interface code and hardware configuration. These commercial CCMs can be used to implement fairly complex and demanding applications.

Missing from these commercial offerings are the powerful tools necessary to make these CCMs easy to use. Once again, the most pressing problem in configurable computing seems to be the development of tools to efficiently use CCMs. Tool development is lagging behind the development of configurable hardware, and seems to be a limiting factor in its adoption, so there is currently somewhat less interest in hardware development than in earlier years. Research does continue in various areas of CCM hardware development. Recent research in architectures often involves the integration of configurable hardware and general-purpose processors ([7], [8], and [9]) or DSP chips [10]. New architectures are also being developed to take advantage of developments in commercial FPGA designs. Xilinx Corporation's new Virtex family, which offers much higher densities than its other FPGAs, faster reconfiguration, and partial reconfiguration, seems to be of particular interest to the CCM community.

2) Applications

In addition to studying CCM hardware and the software tools needed to use it, researchers have been investigating the types of applications that are well suited for implementation on CCM hardware. CCM hardware is especially good at applications that require the repeated application of simple operators on large amounts of data. Performing operations in parallel and optimizing them so that the operations can be performed at high speed is one of the primary ways that CCM architectures can achieve good performance. If applications show some exploitable parallelism and use relatively simple operators, then they may be good candidates for implementation using a CCM.

Most configurable computing hardware is very inefficient at performing floating point calculations. These calculations require large amounts of FPGA resources and run relatively slowly. Applications requiring many floating-point calculations are not usually good candidates for implementation on CCM hardware. In some cases, however, an application originally developed using floating point numbers can be implemented using fixed-point data representation without a significant degradation of its performance. Determining the number of bits necessary to represent data at various stages of an application and the effect of changing floating-point representations to fixed-point can be an important step in determining the suitability of application to CCM implementation, and there has been increasing research on this topic [11], [12].

Several application domains have proved particularly suitable to implementation on CCMs. In particular, applications involving image and video processing [13], [14],[15], communications [16],[17], and CAD applications [18],[19],[20], have been the subject of recent research. Since image-processing applications have been widely implemented on CCM hardware, image processing seemed to be a particularly fruitful application domain for an automated system such as CHAMPION. The availability of a widely used tool for image processing application development, Khoros Cantata, made image processing even more attractive as the focus for CHAMPION and for the initial manual implementation presented in this paper.

Image processing applications are well suited for implementation on configurable computing hardware, for several reasons. They involve relatively large amounts of data. For example, one 640 x 480 image with 256 gray levels represents 300 kilobytes of data. If the images being processed come from a standard real time source at 30 frames/second, nearly 8 megabytes per second must be processed. Many image processing operations require only simple arithmetic and logical operations that are well suited to configurable hardware and avoid more complex mathematical operators and floating point calculations that are more poorly suited. Image-processing applications exhibit a high degree of parallelism, in that the same operations are often applied to each pixel in an image or portion of an image. In a general-purpose computer, each operation on each pixel must be done sequentially, which can be time consuming for large images. CCM implementations of image processing applications can often be designed so that multiple pixels can be processed simultaneously by arrays of functional units, greatly improving performance. This kind of array processing has been widely used in application specific VLSI designs [21] and much of this experience is transferable to CCM designs.

3) Development Software

Ideally, software should be developed that would allow a user to capture their design at a high level, without any consideration of digital design or the specific CCM architecture being targeted. The high level design would then automatically be converted into an equivalent hardware description and this hardware description partitioned across the available resources. Finally, the software would generate the host code and programming files to run the application on the targeted architecture. This would allow a user to run their application on CCM hardware without having any knowledge of digital electronics and without even being aware of the mapping process taking place. Unfortunately, this is a very difficult task. No one has yet come even close to meeting these specifications. Much research has been done into tools that can perform parts of the application mapping process, however, and it eventually should be possible to approach this hypothetical ideal case.

Much research has been performed recently to make it easier to run applications on CCMs by allowing high level design information to be entered using general purpose programming languages. Many papers, such as [22], [23], [24], [25], [26], and [27] have reported work on systems to map programs in C or C++ to various types of configurable hardware. Since C is a widely used language, the ability to map applications from C to CCMs should greatly increase the number of people with the ability to utilize configurable hardware. Other work has been done mapping applications from other programming languages to CCMs [28], [29], and [30]. All of these language-based efforts usually work with subsets of the programming languages and require not only programming ability, but also varying degrees of knowledge of the underlying hardware.

These efforts do increase the pool of potential users of CCMs, but mapping applications to hardware with these tools is not completely automated and still requires significant technical expertise.

There have been successful efforts to create tool suites specifically designed for implementing designs on CCM architecture (e.g., [31] and [32]). This type of system provides powerful tools to help the hardware designer create and test designs on CCMs. They are intended for technical users, however, and aim more at increasing the efficiency of the users, rather than widening the pool of potential users. Their primary goal is not to isolate the user from detailed technical information about the hardware, but to make access to that information easier and more useful.

Other systems have attempted to automate portions of the application mapping process. For instance, one style of design for CCMs uses libraries of parameterizable modules that perform standard functions. A design can be created by combining these modules in a design with the necessary interfacing and interconnections. Several systems have been developed that are focused on creating, maintaining, and implementing these libraries of modules (e.g. [33], and [34]). Again, these tools greatly assist the hardware designer, but do not significantly lower the skill requirements for using CCMs.

One system that does make CCMs more accessible to users who are not necessarily experienced hardware designers was developed at Virginia Tech [35]. It allows the implementation of image processing applications by connecting pre-designed modules that perform common operations, such as filtering and thresholding, using a schematic capture tool. These applications are then implemented on a configurable computing engine developed at Virginia Tech, using a set of specific software tools. The system is designed to work only with one specific architecture, however, and many of the steps in the mapping process are manual. Nonetheless, the system does greatly decrease the skill required to create CCM implementations of relatively complex applications.

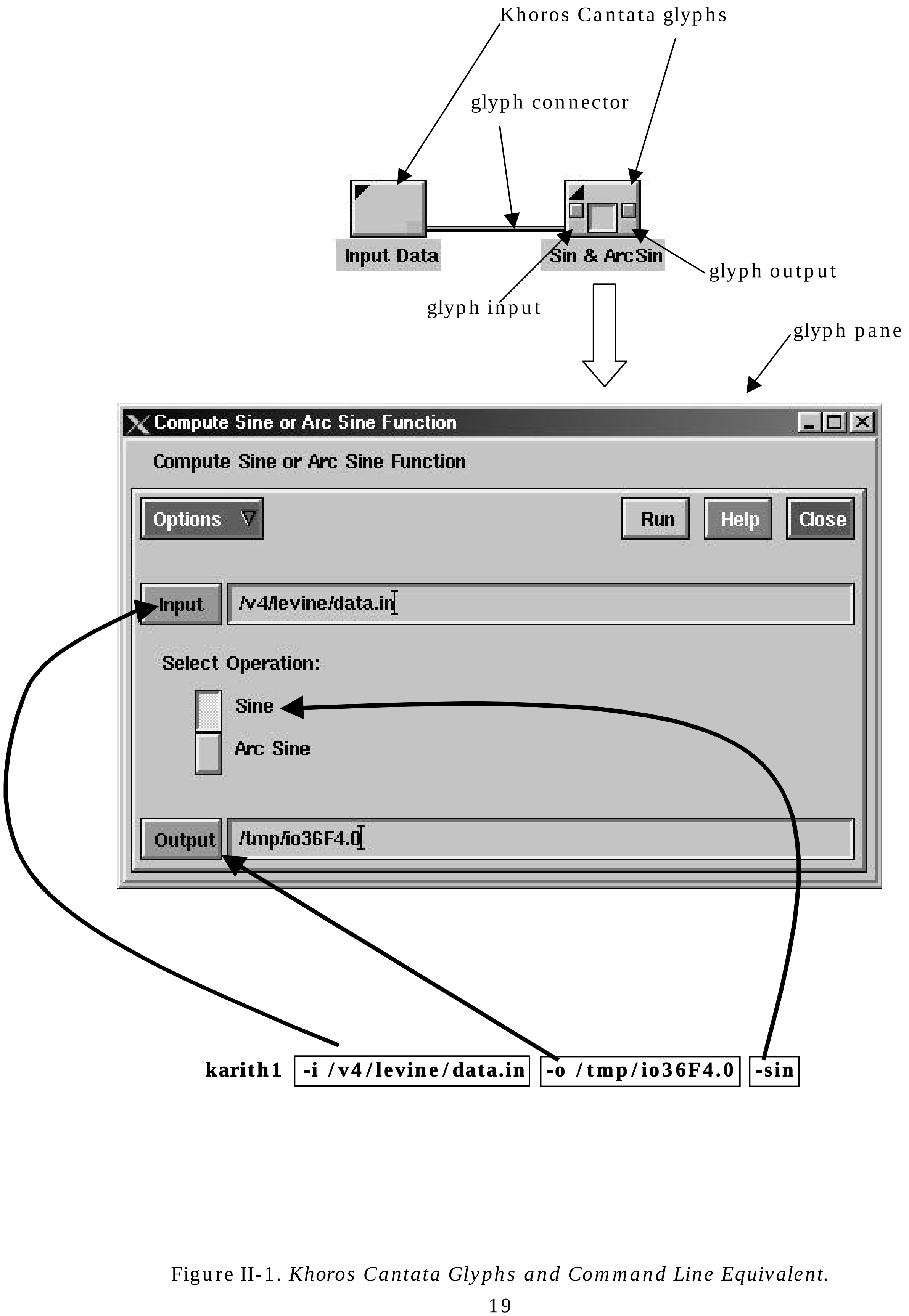
B. Khoros Cantata

Khoros is a software system from Khoral Research Incorporated (KRI). At the core of Khoros is a set of toolboxes containing over 300 operators. These operators include arithmetic operators for scalars, vectors, and matrices; image and signal processing functions; data visualization and display operations; and many functions for manipulating and examining sets of data. The operators can be run as stand alone programs from the command line, or as functions called by a C or Fortran program.

In addition to the operators, Khoros defines three robust data structures. The geometry data model allows for representation and storage of complex geometric structures, and allows for easy access to, and manipulation of these structures. The color data model is used for the storage of color maps in a format that allows for easy transformations of the color space. The polymorphic data model is the most flexible data model. It is capable of storing multi-dimensional data including audio signals, images, video, vector spaces, or virtually any other type of data that can be represented with up to three spatial dimensions and optionally one time dimension.

While Khoros can be used by running commands manually, it is perhaps most widely used with its graphical programming environment called Cantata. Cantata allows the user to create an application by drawing a graphical representation of it. Each function in the Khoros toolbox is represented on the screen by a small icon called a glyph. The glyph has an input terminal corresponding to each of the possible inputs to the function and output terminals for each of the outputs. In addition, each glyph has a pane, which is a set of interface objects that allow the user to set options for the operation of the glyph. Each of these objects corresponds directly to a parameter that can be passed to each function on the command line.

Figure II-1 shows a simple Cantata workspace. A glyph for calculating the trigonometric sine is shown, as well as another glyph that reads in the input data. The pane for the sine glyph is shown directly below the glyph itself. It has a text box for the input and output filenames and radio buttons to select whether the sine or arc sine is to be computed. The input filename is set automatically to be the same as the output of the preceding glyph. In this case, the first glyph is set to read in a file called /v4/levine/data.in. This filename is then passed to the sine glyph. The output filename is automatically picked by Cantata, and corresponds to a temporary file. This filename will be passed to any following glyphs, and the data in the file can be further processed, written to another file, or displayed, depending on which glyphs are used. The user does not have to worry about the filenames for the inputs and outputs, as they are chosen by Cantata. The user also does not have to be concerned about the type of data in the files; Cantata will take care of any conversions between types necessary. It does not matter whether the input is one floating-point value or a three-dimensional array of double precision values. In every case, Cantata determines how to handle the data without user intervention. As mentioned previously, anything in Khoros can be done at the command line as well as in Cantata. The command line equivalent to the two glyphs is shown at the bottom of Figure II-1. Each parameter in the command line has a corresponding user interface object in the glyph pane. On the command line, the user has to type each parameter in manually. In Cantata, the parameters are generated automatically from the settings in the glyph pane. Regardless of whether it is called



Compute Sine or Arc Sine Function

Input /v4/levine/data.in

Select Operation:

Sine

Arc Sine

Output /tmp/io36F4.0

```
karith1 -i /v4/levine/data.in -o /tmp/io36F4.0 -sin
```

Figure II-1. *Khoros Cantata Glyphs and Command Line Equivalent.*

from the command line or from Cantata, the same software routine performs the desired operation on the data.

The power of Cantata, like other graphical programming environments, is that it allows a user to simply draw a diagram of their application and have it run. They do not have to worry about the details of how data is passed or where it is stored. They do not have to be concerned about converting data from one data type to another. They do not even have to compile their code. This means that users do not even have to be conversant with traditional programming methods to quickly implement and test their algorithms and ideas. Since Cantata can work on such a wide range of data, users can test their applications using real data. The visual nature of Cantata also makes it easy to modify an existing application by simply adding or removing glyphs and changing connections.

Cantata can be used by users who may be skilled in their own area of expertise, but who may not necessarily be able to program well enough to test their ideas. For instance, a mathematician who is adept at creating algorithms for computer vision, but might otherwise be limited by a lack of programming ability, can use Cantata to implement and test algorithms using real data. Cantata has become widely used partly because of its ability to isolate users from the underlying technology and still let them exploit the power of the computing platforms they are using.

C. CHAMPION

The goal of research currently being pursued at the University of Tennessee is to produce a system, called CHAMPION, which will automatically map applications implemented in Khoros Cantata to configurable computing hardware. This would greatly increase the number of potential users of CCMs, as little or no knowledge of hardware design would be required. Khoros Cantata was chosen as a front end to CHAMPION due to its widespread use in the image processing community, and also because it isolates the user from the technical details of the hardware doing the computations. This is important because it will allow the same application to run on a workstation or on configurable hardware without visible differences in the application, as long as the CHAMPION libraries are used.

CHAMPION will require the creation of a set of libraries and a set of tools or procedures. The libraries will consist of Cantata glyphs that are compatible with hardware implementation and a corresponding hardware version of each of the Cantata glyphs. As long as the user uses these CHAMPION glyphs to create their Cantata workspace, CHAMPION should be able to convert the Cantata application into a hardware implementation to be run on a CCM. The Cantata glyphs and the corresponding hardware will perform the same operations, so the hardware implementation should perform identically to the Cantata implementation. The determination of the types of glyphs needed and how these will be implemented in hardware is an important part of the development of CHAMPION, and will be guided by the results obtained in the research presented in this paper.

In addition to the two sets of libraries, CHAMPION will require the development of a set of tools to perform the application. Tools will need to be developed for each step of the mapping process. First the Cantata workspace for the application must be read and converted into an intermediate form for use by CHAMPION. Each glyph in the Cantata workspace must be replaced by its hardware equivalent, and then the hardware partitioned to fit the target architecture. The programming files for each FPGA must be generated and the necessary code for the host processor must be generated. Exactly what is required at each step of this process will be determined in large part by the results of the manual implementation presented herein.

III. PROJECT GOALS

The goals of this research project were to manually implement a significant Khoros Cantata application on configurable hardware, using glyph libraries and procedures that will be usable in the context of the larger CHAMPION project. The specific application chosen was an automatic target recognition application called START and the specific CCM was a commercial product called the Wildforce-XL. Details of the application and the hardware it was implemented on are presented in this chapter. Details about the implementation process are presented in Chapter IV, and the results of the implementation are presented in Chapter V.

A. START Application

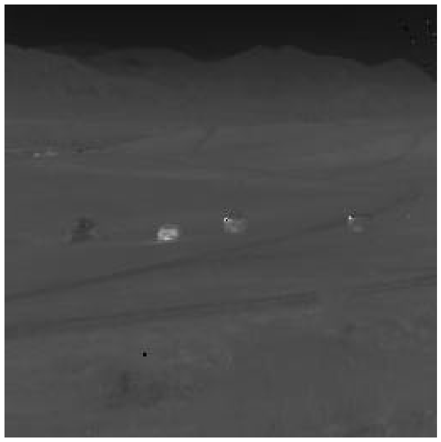
A relatively complex image processing application that was well understood by the author was desired for the first manual application implementation. Using a complex application would provide several benefits. It would help identify more useful hardware glyphs than would a simpler algorithm. Using an algorithm large enough to require reconfiguration of the board would test that capability of the board and require the determination of techniques to partition the design temporally as well as spatially. Finally, using a complex algorithm would validate that the libraries and procedures developed were sufficient to complete problems of a significant nature.

An Automatic Target Recognition (ATR) application developed by the author for a digital image processing course was picked as the application to be implemented. The application was named Simple, Two-criterion, Automatic Recognition of Targets, or START. The application was not chosen for its merit as an ATR algorithm; rather it was chosen because it was sufficiently complex and because it was an existing application developed by the author. This familiarity with the application allowed the efforts of the author to be concentrated on the hardware implementation and not on understanding the application to be implemented, or on devising a new one.

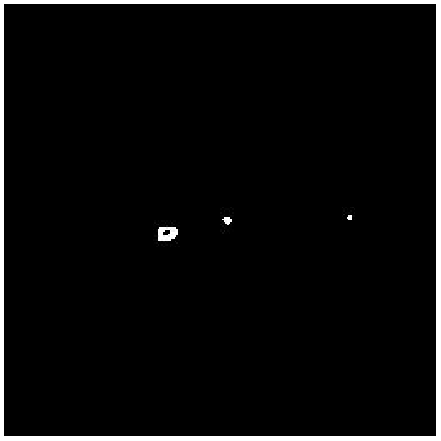
The START application uses a statistical algorithm to find regions in Forward-Looking InfraRed (FLIR) images where a target may exist and draws a box around such regions. Algorithms of this sort are often used for target queuing; that is, they automatically identify areas of possible interest for further examination by human operators. The algorithm does not positively identify a target, nor does it identify the type of target, as a true ATR algorithm does. It simply identifies segments of the image as areas where there is a high probability of there being one or more targets.

The START algorithm was tested with FLIR images from Colorado State University's Fort Carson RSTA Data Collection. This is a freely accessible collection of image data available over the Internet [36]. Forty images were chosen from the entire set of available images. The images chosen had corresponding color visual light images available so that the actual location of targets could be determined more accurately, as identifying targets in the FLIR image can sometimes be difficult for a human observer. The FLIR images were taken of varied, generally hilly terrain, with either no vehicles present, or up to four vehicles present. The vehicles used to represent targets were an M60 main battle tank, an M113 armored personnel carrier, an M901 anti-tank vehicle, and a GMC pickup truck. A representative FLIR image is shown in Figure III-1(a).

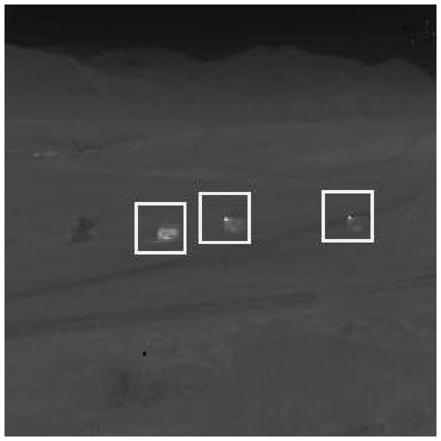
Many variables were present in the image set. Vehicles varied in their distance from the sensors and in their orientation, greatly changing their apparent size and shape. Vehicles were sometimes partially concealed by terrain features, which also introduced



(a) FLIR Image Used as Input to START Algorithm.



(b) Target Pixel Map from First Phase of START Algorithm.



(c) Output image from START Algorithm.

Figure III-1. *Sample Images from START Algorithm.*

variation in size and shape. The overall image contrast was variable, as was the relative background intensity. These variables made the determination of target regions challenging. A straightforward algorithm based on statistical evaluation of two criteria was found to provide reasonably good results, however.

Areas of interest are assumed to have two characteristics. First, they must contain pixels hotter than the surrounding terrain. In a FLIR image, hotter pixels are brighter or higher in numeric value. An area of interest must also contain pixels having a high numerical value after the application of an edge detection filter. A high numerical value indicates large gradient intensity between adjacent pixels, which corresponds to a sharp temperature gradient between adjacent regions. Vehicles are likely to exhibit sharp temperature gradients, either between the vehicle and the surrounding terrain, or between different components of the vehicle, such as an exhaust port and the chassis. An area of interest must satisfy these two different criteria; it must contain hot pixels and it must contain pixels that exhibit large gradients. These are the two criteria in the name Simple, Two-criterion, Automatic Recognition of Targets.

While the core idea of the algorithm is relatively simple, many details of the START algorithm are important to its success. First, the images are low-passed filtered initially to remove noise, as image noise is generally high frequency in nature and produces strong responses in edge detection filters. The threshold values for each of the two criteria are specified as a certain number of standard deviations from the mean. These values are important to the performance of the algorithm and were determined empirically during processing of a large set of input images. The intermediate binary images in the target identification process are low-pass filtered and then thresholded. This favors larger clusters of pixels and eliminates the occasional falsely identified pixel.

1) First Phase

Figure III-2 shows a block diagram of the first part of START algorithm, in which a map of potential target pixels is generated. The input image is first low-pass filtered to remove noise. The unfiltered input image is also passed to the second portion of the algorithm through the connector marked “A”. The low-pass filter is implemented by convolving the image with the 3 x 3 mask shown here:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The resulting image is then divided by 8 to normalize the values. Since the actual maximum multiplier from the mask is 9, dividing by 8 causes some intensity scaling of the low-passed image. Since the relative pixel values remain the same, this has so significant effect on the results. Dividing by 8 can be implemented by a simple right shift, while dividing by 9 is much more expensive in terms of hardware, so the former was chosen.

The filtered image is passed to a block that computes the mean and standard deviation of the intensities of the pixels. These statistics are then used to check for pixels that are more than two standard deviations above the mean. Pixels meeting this criterion are given the value 1 and all others are given the value 0. This produces a map of the pixels that meet the criterion of being relatively hot, since the intensities are linearly proportional to temperature in a FLIR image.

The low-pass-filtered image is also passed to a block that performs edge detection. Edge detection is essentially a high-pass filtering operation, which gives high values to pixels with sharp gradients and low values to pixels with low gradients. The particular edge

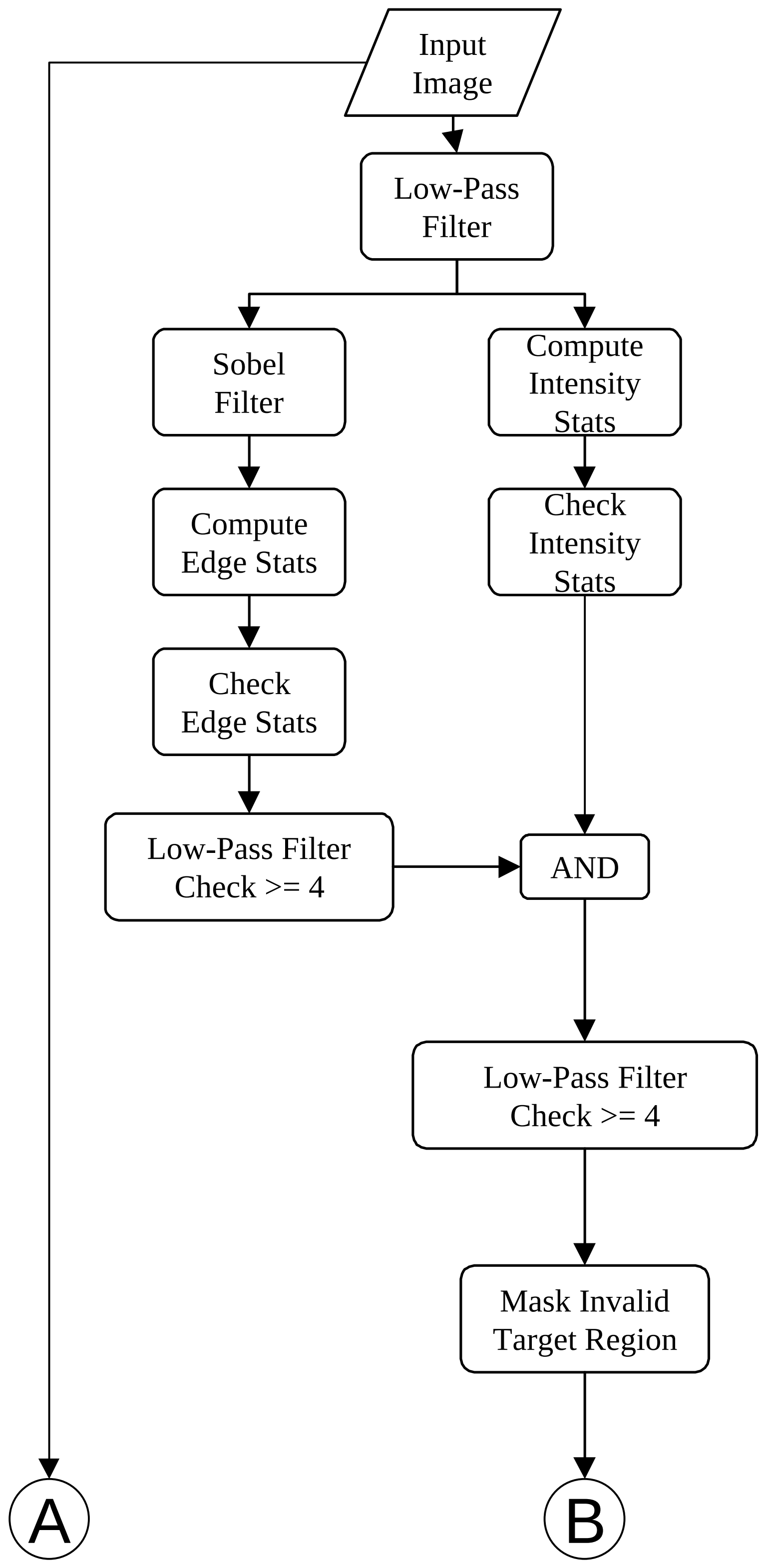


Figure III-2. *First Phase of START Algorithm.*

detection method chosen uses the *Sobel operators* to approximate the gradient. The Sobel operators are two 3 x 3 masks as shown here:

$$\text{Sobel X operator} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \text{Sobel Y operator} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The image is convolved separately with each of the two Sobel operators and then the output image is computed as the sum of the absolute values of the responses, which approximates the magnitude of the gradient response [37].

The statistics for the image after edge detection are then computed. The next block then marks all pixels more than four standard deviations above the mean as 1 and the remaining pixels as 0. This produces a pixel map of all pixels meeting the criterion of having a high gradient value and thus being associated with a transition between regions of greatly differing temperatures. This pixel map is then low-pass filtered, without scaling. This gives each output pixel a value from 0 to 9, representing the number of pixels in each 3 x 3 region that had the value 1 in the pixel map. Pixels with values from 0 to 3 are marked as 0 and pixels with values 4 and above are marked as 1. This creates a new pixel map, which indicates where clusters of pixels meeting the gradient criterion were located. This eliminates insignificant temperature changes or changes involving very small objects.

The gradient criterion pixel map and the intensity criterion pixel map are then logically AND'ed together. This produces a new pixel map with all pixels that meet both criteria being marked as ones and all other pixels marked as zeros. This map then has the clustering operation described previously applied. Next, regions along the edges of the image and in the immediate foreground are marked as zeros, since it is known that these are not valid regions for targets to appear in, based on the camera setup and geometry. A final target pixel map is thus generated and output at connector “B” of the flowchart in Figure III-2.

2) Second Phase

In the second phase of the algorithm, the target pixel map is used to identify target regions and draw frames around these regions. A flowchart of the second phase is shown in Figure III-3. The target pixel map enters the second phase flow chart through the connector labeled “B”. The next block determines whether there are any pixels in the target map with value 1; and if so, the column and row location of the first such target pixel. This information is passed to two more blocks. The first block marks a square frame centered around the first non-zero pixel found in the previous block. These frame pixels are marked on a blank frame map, which consists of all zeros. The second block masks all target pixels within 30 pixels of the first target pixel found by setting those pixels value to zero. This ensures that only one target is identified for every group of pixels corresponding to a particular vehicle in the FLIR image.

The masked target map is then passed to another block, which repeats the operation of finding the first non-zero pixel. This information is passed to blocks that repeat the frame marking and target masking operations. After six total repetitions of frame marking, the frame mask is complete. The frame is a binary image with pixel values of one everywhere a target frame should appear and pixel values of zero everywhere else. The frame map may contain from zero to six frames, depending on the number of target regions identified. A sample frame map can be seen in Figure III-1(b). This frame map corresponds to the input image shown in Figure III-1(a).

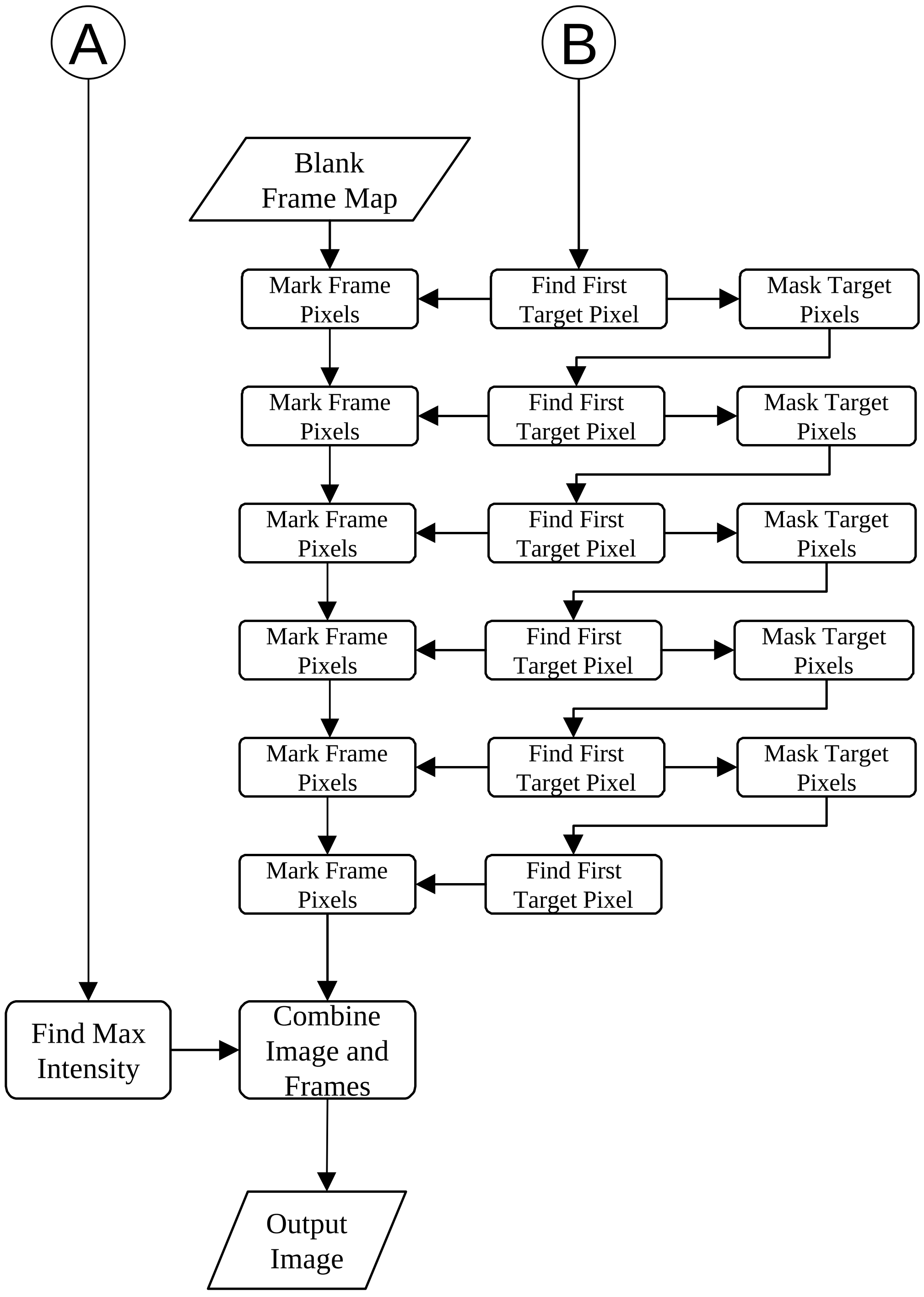


Figure III-3. *Second Phase of START Algorithm.*

The final step of the START algorithm is to combine the frame map and the original image. The original image enters the flowchart through the connector marked “A”. The maximum intensity value in the image is found. This value will be used as the value assigned to the frame pixels. By using the maximum value for the frames, the frame pixels will appear white, but without changing the apparent contrast of the image when displayed. The next block simply replaces every pixel in the original image for which the corresponding frame pixel is one with the maximum intensity value. The pixels in the original image corresponding to zero pixels in the frame map are unchanged. The resulting output image then consists of the original image, with square white frames marked around the identified targets. A sample output image, corresponding to the input image in Figure III-1(a), is shown in Figure III-1(c).

B. Wildforce Board

A commercially available CCM was chosen to implement the first application. Early research efforts often required the design of a new CCM board to meet the requirements of the research. As the field has matured, however, commercial companies have begun to produce hardware that is flexible enough to be used for a wide range of research topics as well as commercial or industrial applications. The Wildforce-XL board from Annapolis Micro Systems was chosen as the first architecture used in the CHAMPION project and was therefore the board available for the manual implementation presented in this paper. The Wildforce-XL board is based on technology licensed from the developers of Splash-2 [4] and like that CCM, uses Xilinx XC4000 series FPGAs. This section will briefly describe the structure of the Xilinx XC4000 series FPGA, as well as detailing the architecture of the Wildforce board itself.

1) Xilinx FPGAs

The Xilinx XC4000 series FPGAs have been widely used for developing CCMs. These FPGAs are in general use throughout the electronics industry, and are thus widely available and relatively inexpensive. There are also many commercial tools available for synthesizing designs for these FPGAs. Xilinx also has an extensive program for supporting university researchers with free or low-cost hardware and software. These facts, combined with their powerful and flexible design, explain the popularity of Xilinx XC4000 series FPGAs with configurable hardware developers.

The basic functional unit of the Xilinx FPGA is the Configurable Logic Block (CLB). A diagram of the CLB is shown in Figure III-4 (Some details relating to RAM functions and carry logic are not included in this diagram). Each CLB has five main components: two four-input function generators, one three-input function generator, and two storage elements. Each of the function generators can be configured to implement any possible function of three or four variables. The function generators can be also combined to produce functions of up to nine variables. The two four-input function generators receive their inputs from outside the CLB. The three input function generator can use the outputs of the four-input function generators for up to two of its inputs. The remaining input(s) must come from outside of the CLB. The four-input function generators can also be configured to be used as RAM, with up to 32 bits of RAM in each CLB.

In addition to the function generators, there are two configurable storage units in each CLB. These can be configured to act as D-type flip-flops or level sensitive latches. The inputs to the storage units can be configured to come from outside the CLB or from any of the function generators. When used as flip-flops, they can be configured to be clocked

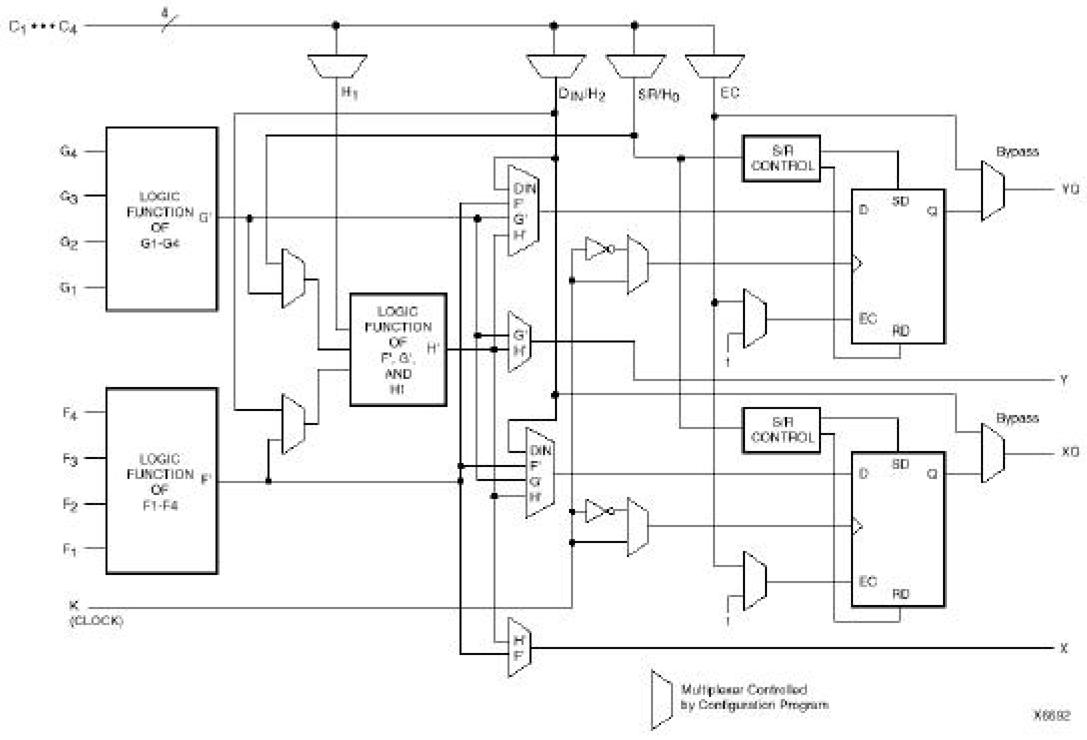


Figure III-4. *Xilinx 4000 Series FPGA Configurable Logic Block.*
Source: Xilinx, Inc, The Programmable Logic Data Book 1998. San Jose, CA, 1998. [38]

on either the rising or falling edges of the clock, and when used as latches they can be configured to be transparent for either high or low clock inputs. The storage elements also have configurable set/reset control and enables.

There are 16 multiplexers in each CLB, six with four inputs and ten with two inputs. These multiplexers are configured to determine the functionality of the CLB. The total functionality of the CLB is determined by the setting for the multiplexers and the settings for the function generators, giving a great deal of flexibility and power in each CLB. The configuration process for the FPGA sets the control inputs to every multiplexer with bits of SRAM. The values in the function generators are also stored in SRAM. This means that the Xilinx FPGA must be reconfigured whenever the power supply to the FPGA is removed and restored.

Every Xilinx 4000 series FPGA contains a square matrix of CLBs, surrounded by a ring of Input/Output Blocks (IOBs). The IOB connect the CLBs to the pins of the FPGA. The IOBs are configurable and can be programmed for different electrical characteristics, use of registers, etc. Each FPGA has an extensive set of internal routing tracks, and many blocks of programmable switches which can form connections between the tracks and the CLBs and IOBs. Different FPGAs in the XC4000 series have different numbers of CLBs, and differ in the exact interconnections available internally and pins available externally.

Each FPGA configuration consists of individual settings for every CLB and IOB, and settings for every programmable interconnection point. The total configuration stream can reach nearly two million bits for the largest Xilinx FPGAs. In CCM applications, the configuration data is usually sent to the FPGA from a host computer. The data can be sent to the FPGA serially or in parallel, at either 1 MHz or 8 MHz clock rates. Even at the faster speed, configuring a large FPGA can take tens or hundreds of milliseconds. The Xilinx XC4000 series FPGA cannot be partially configured; that is, the entire FPGA must be reconfigured to change any of the device settings.

2) Board Architecture

Annapolis Microsystems manufactures several different types of CCM boards. The one used in this project is the Wildforce-XL board. It is a PCI-bus card, which uses five Xilinx XC4000XL FPGAs for processing elements. The specific version of the board used had 1 XC4036XL FPGA and 4 XC4013XL FPGAs available for processing. A comparison of the resources in each FPGA is shown in Table III-1. Other Xilinx FPGAs are used to implement a programmable crossbar.

Table III-1. *Resources Available in FPGAs Used on Wildforce-XL.*

	Logic Cells	CLB Matrix	Total CLBs	Number of Flip-Flops	Equivalent Gate Count
XC4013XL	1368	13 x 13	576	1536	10,000 – 30,000
XC4036XL	3078	36 x 36	1296	3168	22,000 – 65,000

Source: Xilinx, Inc, The Programmable Logic Data Book 1998. San Jose, CA, 1998. [38]

Annapolis Microsystems refers to the FPGAs on the Wildforce board as *processing elements*. The XC4036XL FPGA is called a *control processing element* and given the designation CPE0. It differs from the other FPGAs in that it is larger, and also in that it has control lines available for various resources on the board, such as the external I/O interface and crossbar configuration register, that are not available to the other FPGAs. The four XC4013XL FPGAs are given the designations PE1, PE2, PE3, and PE4. These four processing elements are connected together in a linear array by a 36-bit *systolic bus*. All five FPGAs can be connected by the 36-bit *crossbar*, which selectively allows connections between any of the processing elements. CPE0 can only connect to the other processing elements through the crossbar.

Each FPGA on the board has a small daughterboard associated with it, which can be populated with memory or a Digital Signal Processing (DSP) chip. Each of the FPGAs on the board used in this project had 32 KB of 32-bit SRAM on its daughterboard. These daughterboards have a dual-port memory controller such that both the FPGA and the host computer can access the SRAM.

The board also contains a PCI interface for communicating with the host computer, and several FIFO registers to facilitate data transfer across the PCI bus. Figure III-5 shows a simplified block diagram of the Wildforce board. Since there are many resources

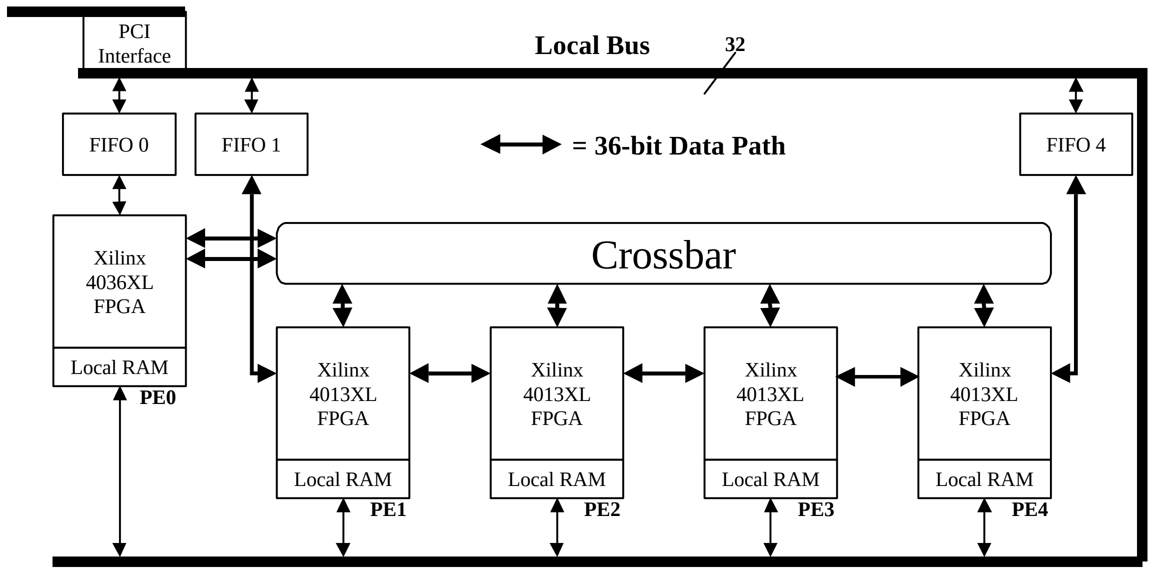


Figure III-5. *Basic Wildforce-XL Block Diagram.*

available on the Wildforce board and many configurations of the crossbar and other components, it was decided to use a constrained configuration of the board for the initial manual and automatic implementations. This reduces the problem complexity to a more manageable level. Eventually the constraints on the board utilization will be relaxed so that all of the power of the Wildforce-XL board will be available in the automated CHAMPION system.

The constrained configuration of the board used in this project did not use any of the FIFOs. All communication with the host was done through the SRAM associated with each processing element. The crossbar was used only to provide a 36-bit path from CPE0 to PE1. The connections between processing elements are normally bidirectional. For the constrained implementation, however, it was decided that the direction of all connections between processing elements would be fixed so that all signals would pass in one direction only. The board topology became a linear array, with all signals starting in CPE0 passing to PE1. No signals could run from PE1 back to CPE0. Similarly, all signals from PE1 ran to PE2, with no signals allowed to pass back from PE2 to PE1. A diagram showing the configuration of the Wildforce-XL board as used in this project is shown in Figure III-6.

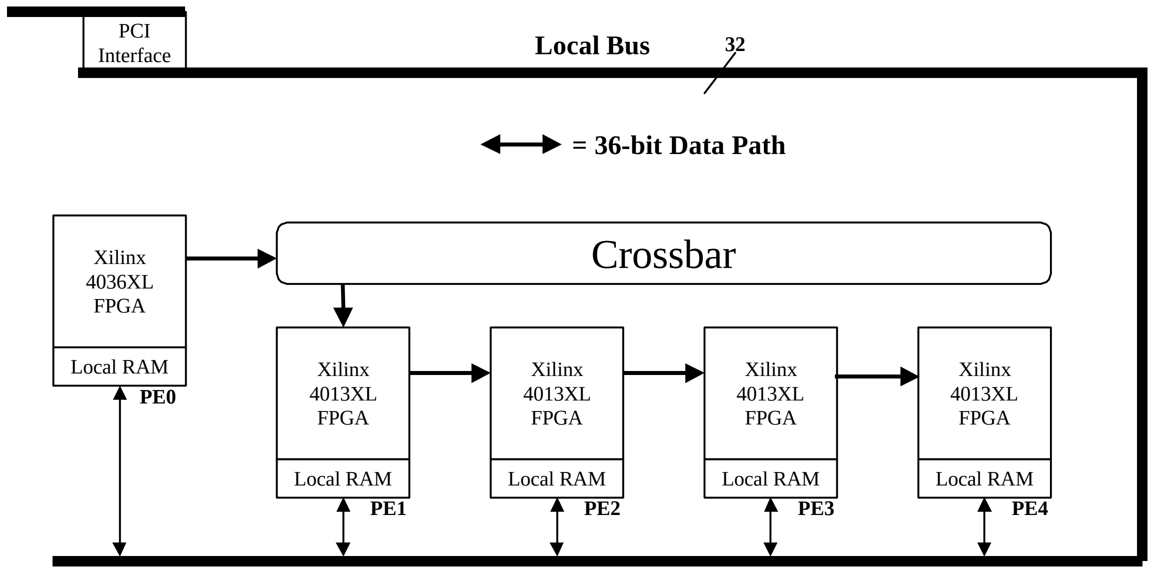


Figure III-6. *Wildforce-XL Board As Used.*

IV. IMPLEMENTATION

A. Cantata Implementations of START

1) Standard Cantata

The START algorithm was initially developed for a graduate class in digital image processing and was written in the MATLAB language. After the decision was made to adopt the START algorithm for this project, it first had to be implemented in Cantata. The first Cantata version used glyphs from the standard toolboxes supplied with the Khoros software package. It took approximately eight days to successfully implement the application in Cantata using the standard glyphs. The standard Cantata implementation took 57 seconds to process one 256 x 256 image from the set of test images.

Ninety-nine total glyphs were needed to implement the algorithm. Only 20 different glyphs were needed, but many of these glyphs were used repeatedly. A list of the glyphs used is shown in Table IV-1. Upon examination of the Cantata workspace, it was apparent that some of the glyphs used could be implemented directly in hardware and some could not be directly implemented in any practical manner. For instance, the glyphs classified as arithmetic in Table IV-1, addition and absolute value, are standard operators that can be easily implemented in hardware. The glyphs classified as Khoros specific, on the other hand, perform functions that are specific to the data structures used in Khoros. Since these complex data structures are not readily implementable in hardware, functions related to these data structures have no direct hardware equivalent. Determining which glyphs had hardware equivalents and which did not was crucial to the eventual implementation of the START algorithm in hardware.

Table IV-1. *Glyphs Used in Standard Cantata Implementation of START.*

Glyph Name	Glyph Type	Hardware Equivalent?	Times Used
Read File	File I/O	Yes	6
Save File	File I/O	Yes	1
Right Shift	Bitwise operator	Yes	1
AND	Bitwise operator	Yes	1
OR	Bitwise operator	Yes	6
NOT	Bitwise operator	Yes	1
> =	Comparison	Yes	4
>	Comparison	Yes	7
Absolute Value	Arithmetic	Yes	2
Add	Arithmetic	Yes	8
Inset	Geometric	No	1
Pad	Geometric	No	18
Statistics	Mathematical	No	9
Convolve	Mathematical	No	5
Convert Type	Data Handling	No	5
Switch	Data Handling	Yes	6
Constant	Data Handling	Yes	15
Copy from Value	Khoros Specific	No	1
Insert Segment	Khoros Specific	No	1
Unmask	Khoros Specific	No	1

Besides the addition and absolute value glyphs, other glyphs that were clearly readily implementable in hardware were the comparison operators and the bitwise operators. The file I/O operators would not actually be accessing file structures on a hard drive, but would be accessing data transferred to the CCM hardware by the host, and would thus perform an analogous function. They were therefore also considered to have a hardware equivalent. The switch glyph performed a function identical to that performed by a multiplexer, so it had a clear hardware equivalent. The constant glyph simply generates a constant value, which can be done in hardware by simply fixing wires to logical ones or zeros.

Many other glyphs had no clear hardware equivalent. The pad and inset glyphs performed geometric operations on a two-dimensional image. There was no directly equivalent way to do these two-dimensional operations in a hardware implementation where the images would be streamed through in one dimension. The statistics glyph presented a different problem. This glyph calls a Khoros routine that can perform over 25 different statistical functions, from simple functions such as mean, to much more complex functions such as skew and kurtosis. It would not be efficient to create a hardware module that performed all of these functions if only one or two were used in a given application. Since the START application only needed the mean and standard deviation functions, it made sense to have these be implemented separately, using other hardware glyphs, rather than by a monolithic statistics block capable of performing other operations. The convolve operator had a problem similar to the statistics glyph, in that its functionality was too broad. It could perform a convolution of one image with a mask of any size and with any values. This sort of flexibility can not be implemented efficiently in hardware. An efficient implementation of convolution would have to have a structure specified by the mask to be used, so a general-purpose convolution block had no direct hardware equivalent. The convert type glyph converts data from one data type to another, such as from floating point to integer. Since the data types would have to be managed quite differently in hardware than they are in software, there could be no glyph performing this exact function in hardware. As mentioned previously, the three glyphs that performed Khoros specific function also had no hardware equivalents.

The implementation of the START algorithm in standard Khoros revealed several problems. As just discussed, many of the glyphs used in standard Khoros have no exact hardware equivalent. The START algorithm would have to be implemented in Cantata again, using only glyphs that had hardware equivalents. However, the functionality of the glyphs that did not have hardware equivalents would have to be duplicated in some fashion that was compatible with glyphs that could be implemented. This required the creation of glyphs to perform new functions that would map directly to a hardware implementation.

2) Hardware Equivalent Cantata

After the problems with the standard Cantata implementation were identified, development of a new Cantata implementation that would be more compatible with hardware implementation was begun. Some portions of the original workspace contained only hardware compatible glyphs and could be retained, but in general, the algorithm had to be re-implemented from scratch. It took more than three weeks to develop the hardware equivalent Cantata workspace, including the time taken to develop the custom glyphs.

The hardware equivalent implementation took 17 minutes and 34 seconds to process one 256 x 256 image from the test image set. This is more than eleven times longer than the standard Cantata version. Not only did the hardware equivalent version have more glyphs than the standard version, but also some of the glyphs in the hardware equivalent were custom glyphs. Due to the way that Cantata integrates custom glyphs with the standard ones, custom glyphs are much slower.

The new implementation was identified as being *hardware equivalent Cantata*. It contained a mix of standard Cantata glyphs and glyphs that were designed especially for this project and were linked to C programs that performed the desired functions. The hardware equivalent Cantata version required a total of 305 glyphs, of 24 different types. The glyphs used are shown in Table IV-2. Many of the glyphs used are in the standard Cantata toolboxes, although not all of these were used in the standard Cantata implementation. This is because some of the functions that were performed by glyphs that did not have hardware equivalents could be implemented instead by other

Table IV-2. *Glyphs Used in Hardware Equivalent Cantata Implementation of START.*

Glyph Name	Glyph Source	In First Version ?	Times Used
RAM Read	Custom	No	1
RAM Write	Custom	No	1
Convolution Stream	Custom	No	4
Stream Max	Custom	No	7
Stream Sum	Custom	No	4
Row/Column Counter	Custom	No	7
Add	Khoros	Yes	62
Subtract	Khoros	No	24
Absolute Difference	Khoros	No	4
Right Shift	Khoros	Yes	15
Left Shift	Khoros	No	3
Square	Khoros	No	4
Square Root	Khoros	No	2
> =	Khoros	Yes	18
>	Khoros	Yes	12
<=	Khoros	No	12
<	Khoros	No	14
==	Khoros	No	24
AND	Khoros	Yes	50
NAND	Khoros	No	6
OR	Khoros	Yes	26
NOR	Khoros	No	1
Clip	Khoros	No	3
Constant	Khoros	Yes	1

standard Cantata glyphs, which did have hardware equivalents, or by combinations of these glyphs with custom glyphs.

All of the custom glyphs were needed because of differences in the way that image data is represented in Cantata and the way it is represented in hardware. In the standard Cantata implementation, the images are represented by two-dimensional arrays of pixel values. Since the hardware implementation would use one-dimensional streams of pixels, the mapping would be simplified if the hardware equivalent Cantata version used one-dimensional arrays of pixel values as well. This required new Cantata glyphs that operated on one-dimensional stream of pixel values, as well as glyphs to read and write these streams. These functions were written in C and used to create new Cantata glyphs using the development tools provided with the Khoros software package.

The hardware equivalent Cantata implementation was tested using the same suite of FLIR images used to test the standard Cantata implementation (These results will be discussed in Chapter V). Due to slight differences in the way some calculations were done, and the way data was represented, not all of the results were identical when compared pixel by pixel. However, the results were identical when the targets identified and missed were compared between the two versions. The hardware equivalent version had the same effective functionality as the standard Cantata version and so it was accepted as suitable for implementation in hardware.

B. Glyph Libraries

A major part of the development of the CHAMPION project is the creation of glyph libraries. A set of hardware-equivalent glyphs for use in the Cantata environment was needed for developing applications. These hardware equivalent glyphs were identified by determining which ones were needed for implementing the START algorithm. A corresponding set of hardware glyphs was also needed to implement the application on the CCM hardware. Further implementations of other applications will expand the libraries available for CHAMPION. Much of the hardware glyph development was done by another member of the CHAMPION team, Mr. Senthil Natarajan. A more complete account of this aspect of the research may be found in his M.S. thesis, submitted in August 1999, to the University of Tennessee in Knoxville.

1) Khoros Cantata Glyphs

The are many glyphs available in the toolboxes that come with Khoros. These range from very basic operations such as addition and subtraction, to very complicated ones such as kurtosis and fast Fourier transforms. All of the glyphs share common code that provides support for the various Khoros data structures and for the graphical user interface used in the Cantata environment. Built into each glyph is code to support data transfer from one glyph to another, handling of different data types and structures, and other low-level functions. The actual operations performed by the glyphs are performed by separate programs that can be called from the command line, as well as from Cantata. The glyphs thus represent fairly complex software constructs, which is part of what gives Cantata its power and flexibility.

Unfortunately, as previously discussed, many of the standard Khoros glyphs do not map very well to hardware. In addition, the complex software wrappers that provide the sophisticated data handling of the glyphs abstract away some important considerations for data transfer and synchronization that must be considered in hardware. The standard hardware glyphs are too “high-level” for completely trouble-free use in the CHAMPION system. By eliminating glyphs that cannot be implemented in hardware and creating new ones specifically designed for use in hardware implementations, and by making allowances for the different data handling characteristics of Cantata and hardware, the problems with the standard Cantata glyph libraries can be overcome.

2) Hardware Equivalent Glyphs

A set of glyphs that could be used in CHAMPION had to be chosen. These included standard Cantata glyphs that could be implemented in hardware, as well as custom glyphs developed specifically for CHAMPION. While not every existing Cantata workspace will have the potential to be mapped to hardware, the goal of the CHAMPION project is to be able to map to hardware any Cantata workspace which is entirely composed of hardware compatible glyphs. If the set of hardware compatible glyphs is large enough, them most, if not all, workspaces that can be implemented in Cantata can be implemented using hardware equivalent glyphs, and thus can be mapped to hardware by CHAMPION.

The glyphs chosen to be used in CHAMPION had to not only be implementable in hardware, but they also had to operate in Cantata in a manner equivalent to the way the hardware glyphs would operate. For this reason, the Cantata glyphs used in CHAMPION are called *hardware equivalent glyphs*. Many of the standard Cantata glyphs were considered to be hardware-equivalent, even though they had more functionality than the corresponding hardware. As long as the glyphs can produce the same outputs as hardware when presented with the same inputs, they are considered hardware equivalent, even if the Cantata version can handle a wider range of inputs than the hardware versions.

Glyphs for many of the standard functions needed, such as basic arithmetic and logical operators, could be supplied by the standard Cantata toolboxes. However, those operations that depend on the geometry of the data needed to be created especially for CHAMPION. This is because of the two-dimensional representation of images used in Cantata, as compared to the one-dimensional representation of images in hardware. Custom glyphs were created to perform functions on streams of pixel images. These functions were needed to perform the types of operations that are normally implemented with two-dimensional geometric operators in standard Cantata. All of the custom hardware equivalent glyphs are listed in Table IV-2.

3) Hardware Glyphs

A hardware glyph is a block of hardware that performs a certain specific function that corresponds directly to a hardware-equivalent glyph in the CHAMPION toolbox. A hardware glyph had to be developed for every hardware-equivalent glyph used in the START algorithm. The glyphs were developed in VHDL. This was done to allow for compatibility with CCMs other than the Wildforce. By using VHDL, the behavioral description of each glyph was independent of the specific FPGA that it would be implemented on. The VHDL description of each glyph was parameterizable, meaning that various characteristics of the glyphs, such as the number of bits in the input, were specified as generics that could be changed as needed. This meant that one VHDL template could be used to synthesize many different instances of specific glyphs. For instance, an addition glyph template could be used to synthesize adders for inputs of 8 bits, 9 bits, or however many input bits were needed for a specific application.

By using glyphs with characteristics that matched those needed by the application, a more efficient use of hardware could be made. Using an 8-bit adder when two 5-bit numbers are being added wastes nearly half of the resources used by the adder glyph. Each specific version of each glyph was synthesized and stored in a format specific to the hardware being used. Since Xilinx FPGAs were used on the Wildforce board, the specific glyph instances were synthesized into the Xilinx Netlist Format (XNF) and stored in a library. In CHAMPION, if an application needs an instance of a glyph that does not yet exist in the library, the VHDL template will used to create it. The new glyph will then be kept in the library in case it is needed in the future. In this way, most of the glyphs needed by applications can be pre-synthesized, speeding up the application mapping process. In the initial manual application, the library was created with only the glyphs needed to implement START. By implementing more applications, the library of pre-synthesized hardware glyphs will be enlarged.

All of the hardware glyphs were designed in such a way that any glyph can be connected with any other glyph, assuming that the number of bits in the inputs and outputs are equivalent. Special glyphs for padding and truncating bits can be used to interface glyphs that otherwise could not be connected because of differing data widths. In addition to data inputs and outputs, each hardware glyph had three standard control lines associated with every data stream being input to or output from the glyph. These control lines helped to emulate the data transfer characteristics of Cantata glyphs. The three control lines were called Stream Valid (SV), Pixel Valid (PV), and Data Valid (DV). Each hardware glyph handled these three control lines in a consistent way so that the meaning of the control lines was the same through out an entire hardware implementation.

Stream Valid was used to delimit the beginning and end of each stream of data. A single stream of data consisted of one unit of data that was to be considered as a whole, such as a single image. In Cantata, these units of data are transferred as one single data structure. Since in hardware, these units are transferred one data value at a time, a way of delimiting one stream from another was needed. The SV control line is normally low. When a new stream begins, the SV line associated with that stream goes at high at the first pixel of the stream and goes low after the last pixel. In other words, the SV line is high whenever a stream is present and low whenever it is not. As long as least one clock cycle is inserted between successive streams, the hardware glyphs could distinguish between them.

Pixel Valid was used to indicate if individual data values in a stream were actually part of an image, or if they were meaningless data inserted as a spacer by a glyph that could not produce a results every clock cycle. For instance, one method of performing convolution in hardware can only produce a value every nine clock cycles. In this case, eight out of every nine data will have the PV line set low; only the ninth data value, which corresponds to an actual pixel, will have the PV line set high. All of the data values will have the SV line set high, however, as there is only one stream of data present.

The Data Valid line is used to signal whether a particular data value contains data that is valid. These are data values that do represent valid pixels, so the PV line is high, but the value stored in that pixel location may not be data that should be used in further calculations. For instance, the pixels along the edges of an image that has been convolved with a mask contain values that are not valid, as they are the result of both the image and the zero-padded edges. These border pixels could skew statistical calculations involving the image, so they are marked by having the DV line associated with these pixels low, signaling that the data in these pixels is invalid. Table IV-3 shows the different possible combinations of control lines and their meanings.

Table IV-3. *Hardware Glyph Control Line Combinations and Meanings.*

SV	PV	DV	
0	X	X	If SV is low, there is not a data stream present. The values of PV and DV have no meaning in this case.
1	1	1	In this case, SV, PV, and DV are all high, indicating that a valid data stream is present, the current data represents a valid pixel location, and the current data value is valid.
1	1	0	Since DV is low, the current data value is not valid, but it is a valid pixel location and is part of a valid stream.
1	0	X	Since PV is low, the current value is not a valid pixel location. In this case, the data is always invalid, so DV line has no meaning. This data value is still part of a valid data stream.

C. Differences Between Cantata and Hardware

At first glance, an application in Khoros Cantata may seem very similar to a schematic for a hardware design. Each glyph can be seen to represent a functional block and the connections between glyphs can be seen to represent wires or buses connecting the functional units. Unfortunately, upon closer examination things are not quite that simple. The Khoros Cantata software takes care of many details that are not apparent from the graphical workspace. Operations such as converting between different data types, synchronizing data, and transferring blocks of data between glyphs are handled by software wrappers in each glyph. In a hardware design, these operations may need to be performed by specific hardware modules that have no corresponding glyphs in the Cantata workspace. Three different types of operations were found that were handled significantly differently in Cantata and in hardware. These operations and the way they are handled are discussed in the next three sections.

1) Data Transfer

In Cantata, data transfer from glyph to glyph is handled by the operating system. In the most common operating mode, the data output by one glyph is written out to a temporary file using system functions. The Cantata system records the name of the temporary file and passes it to the glyph(s) that will use that data as their input(s). These succeeding glyphs then use system functions to read in the data. The actual transfer method involves storage on a hard drive, and the entire file is transferred at one time. The data is stored in one of the Khoros data structures.

In the type of implementation that is suited for operation in FPGA-based hardware, data is transferred between glyphs one value at a time, rather than an entire block of data.

Instead of transferring an entire image, as in Cantata, only one pixel is transferred at each clock cycle in hardware. The transfer method requires registers to hold the data values as they are transferred between glyphs. In some cases, data must be stored in RAM as an intermediate type of storage between glyphs. Even if data is stored in RAM, though, data is still transferred one value at time.

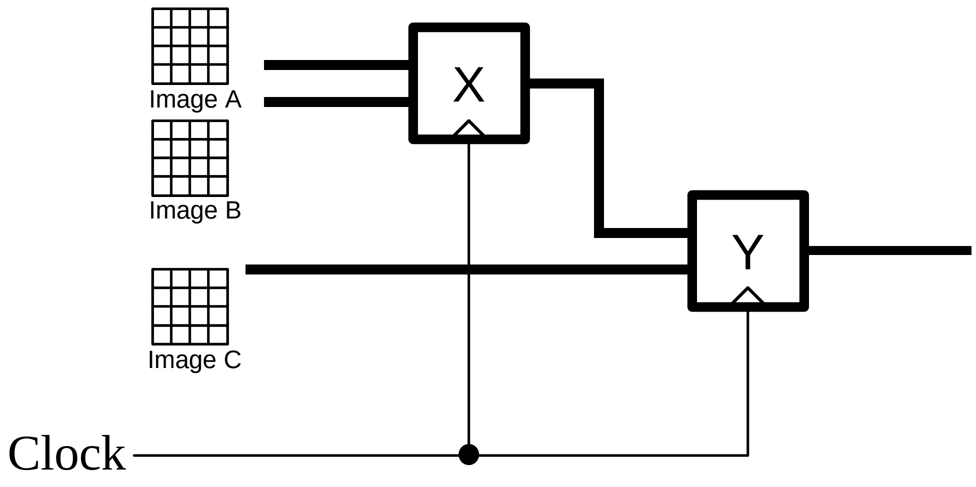
To prevent problems from occurring when mapping to hardware, this difference in data transfer must be accounted for. If a hardware glyph operates on each value in a stream individually, than this difference in data transfer will not matter. For instance, a hardware equivalent glyph that adds a constant to each value in a stream only needs to work with one value at a time. So the equivalent hardware glyph can work with one data value at a time as well and the mapping will be simple. A hardware equivalent glyph that needs every data value in a stream before it can produce a valid output, such as a glyph for finding the sum of all the pixels in an image, will not map to hardware easily, unless the hardware glyphs take the differences in data transfer methods into account. By creating hardware glyphs that do not bring the output SV line high until they have processed the entire input stream, the mapping from Cantata is again made simple.

2) Data Synchronization

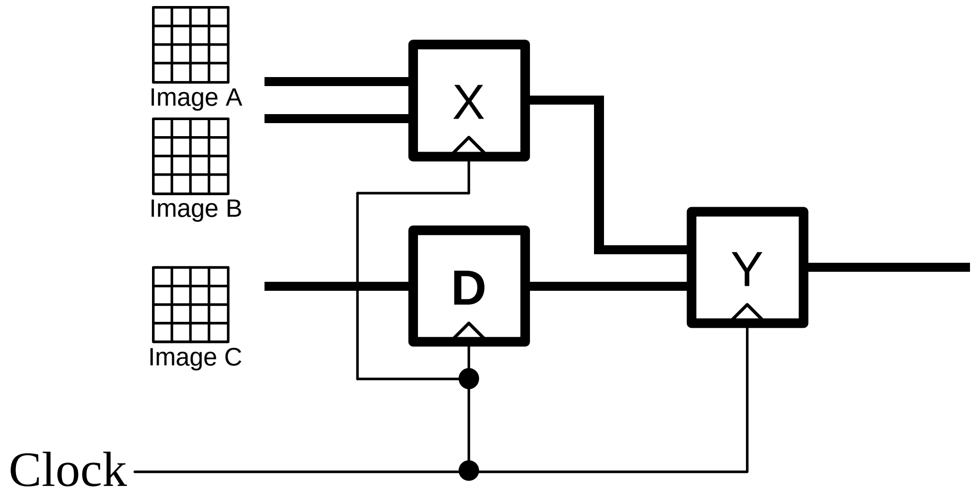
In Cantata, each glyph is a separate C program that is called by the main routine. These separate glyph routines are called whenever there is new data for all of their inputs. If only some of the data needed is present, than the main routine will wait to call the glyph routine until the missing data is also present. In this way, there are never any data synchronization errors in Cantata. In the hardware implementation, however, data must be synchronized because all of the hardware is synchronous and processes whatever data is at the inputs at every rising clock edge.

In Figure IV-1(a), a simple network of two hardware glyphs, designated X and Y, is shown. Both glyphs run off the same clock signal, as do all of the hardware glyphs in every FPGA. If images A, B, and C all become available at the same time, then their SV lines will go high on the same clock cycle. When both input SV lines to glyph X go high, it can add the two inputs, which are the first pixel from image A and the first pixel from image B. On the next clock cycle, the SV line at the output of glyph X will go high, since it can output valid data. However, at the same time that the SV lines for images A and B go high, the SV line for image C goes high as well. One of input SV lines for glyph Y is high, since it is the image C SV line, but the other input SV line for glyph Y is the output SV line for glyph X, and it will not go high for another clock cycle. This means that the inputs to glyph Y are out of sync, and the results in hardware will not be the same as the results obtained in Cantata.

To fix this problem, a delay buffer, D, needs to be inserted before one of the inputs to glyph Y, as shown in Figure IV-1(b). Now the SV line for image C is delayed one clock cycle before it reaches glyph Y, synchronizing it with the output of glyph X. The data synchronization for all glyphs with more than one input must be checked and fixed throughout the entire application. If the correction factor needed is too large, than it will be impractical to insert delay buffers to synchronize the data. Instead, the data must be stored in RAM and synchronized by correct timing of the RAM read operations. It is important to note that the glyphs inserted for data synchronization do not appear in the original Cantata workspace. Instead, they are identified and inserted during the mapping process, whether the mapping is done manually or automatically.



(a) Unsynchronized Glyphs.



(b) Glyphs Synchronized by Delay Insertion.

Figure IV-1. *Data Synchronization Problem and Solution.*

3) Data Conversion / Operator Sizing

In Cantata, data conversion is handled automatically by the software. If the inputs to a glyph are of different types, the mismatched data will by recast, using the standard rules of the C language, adapted to prevent data loss. For instance, if an integer is added to a floating-point number, the result will be a floating-point number. If a double precision number is added to single precision number, the result will be double precision. This is easy to do in software, since every operator can work on any valid data type. For instance, one addition program can easily be made to work on any kind of input data.

As discussed earlier, all of the hardware glyphs are parameterized for the number of input bits. There are also different glyphs for signed and unsigned numbers, and when floating point support is added to CHAMPION, there will be different glyphs for floating point numbers. All of the glyphs are designed with the assumption that the inputs will all be of the same type. So two unsigned five-bit numbers can be added together, but a five-bit number cannot be added to a six-bit number, and an unsigned number cannot be added to a signed number. If the inputs to a glyph are of different types, one of the inputs must be converted to match the other. Since there are so many different possible combinations of input types, there is no way for the glyph to do this automatically. Part of the mapping process must therefore include checking to see that all of the inputs are of the same type. If not, than one or more inputs must be converted to correct the mismatch. This is done by inserting a hardware glyph to perform the correct conversion. These conversion glyphs do not appear in the Cantata workspace. Just like the glyphs inserted for data synchronization, they are not identified and inserted until the mapping process.

In addition to ensuring that data is converted to consistent types at the inputs of a glyph, the glyph itself must also be of the correct size for the data being input. The Cantata workspace will have the same adder glyph every place that addition is to be performed, regardless of the type of data being added. In hardware, the addition glyph selected must be chosen to match the type of numbers being added. If two five-bit unsigned numbers are being added, an adder glyph for five-bit unsigned numbers must be chosen from the library of pre-synthesized hardware glyphs. If that specific adder glyph is not already in the library, it can be created using the VHDL template for adder glyphs. In this way, the correct glyph for the data can always be made available. The information as to the specific adder being used does not exist in the Cantata workspace. The mapping process must determine which specific hardware glyph is needed, based on the data being presented to it.

D. Mapping Procedure

Once the glyph libraries had been completed and the differences between Cantata and hardware were understood, the development of the mapping process could begin. A simple example, actually a portion of START, will be used to illustrate the steps involved in mapping from Cantata to hardware. The same techniques demonstrated with the example were also used to map the entire START application.

1) Workspace Conversion

The first step in the process was to convert the Cantata workspace into a graph form that could be manipulated more easily. The example hardware equivalent Cantata workspace is shown in Figure IV-2. This workspace performs a simple low-pass filtering

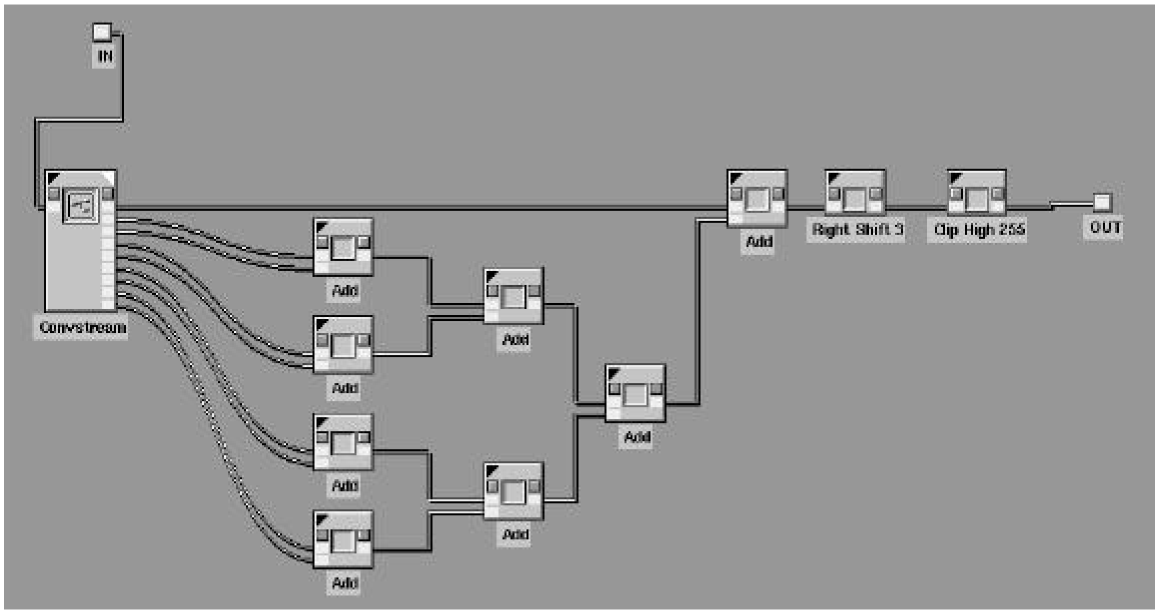


Figure IV-2. *Example Hardware Equivalent Cantata Workspace – Low Pass Filter.*

operation. A one-dimensional stream of pixel values, representing a two-dimensional image enters the workspace through the terminal labeled “IN.” The convstream glyph produces nine data streams from the input. For each pixel in the input stream, nine pixels are output from the convstream glyph, representing the eight neighboring pixels, and the original pixel itself. An entire stream representing each of the neighboring pixels is generated by the glyph, with zeros inserted for any pixel neighbor which falls outside the original image. These pixel neighbor streams, along with the original stream, can be used to implement a convolution with a 3 x 3 mask. In the example, all of the streams are added together, and the result is divided by eight. This is equivalent to convolution with the mask shown below:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \frac{1}{8}$$

This is the same low-pass filter operation described in section III.A.1. The divide by eight is implemented by a right shift of three places, and the output is clipped to a value

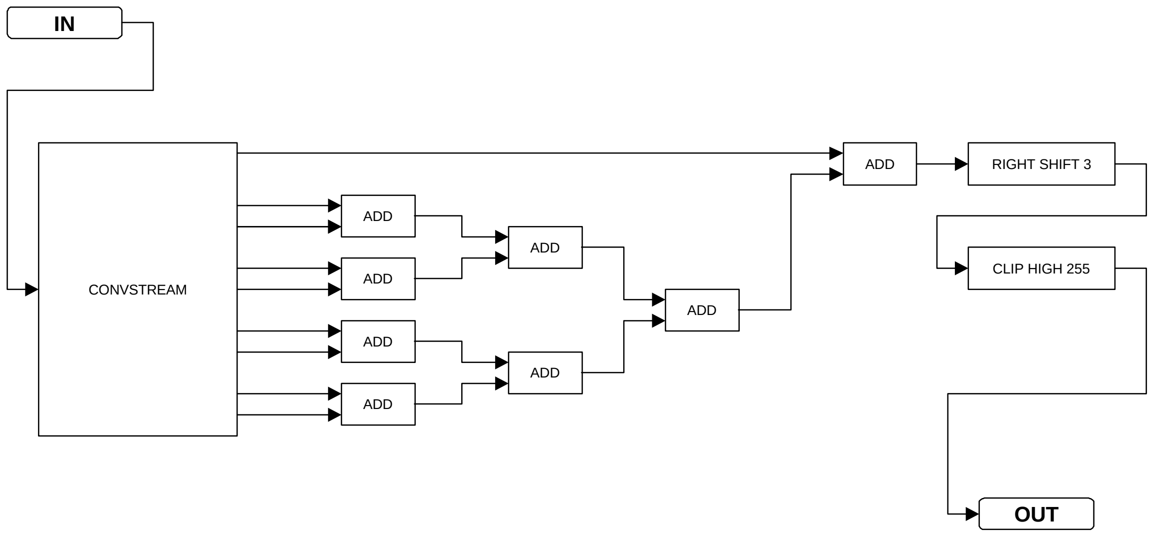


Figure IV-3. *Graph Form of Low-Pass Filter Workspace.*

of 255, so that the data remains 8-bits wide. The equivalent graph form of this workspace is shown in Figure IV-3. This graph form consists of a set of vertices, which correspond to glyphs, and a set of edges, which correspond to the connections between glyphs. The graph form is widely used for design automation and can easily be stored and manipulated by computer, making it ideal for the automated system, CHAMPION. For the manual implementation, the graph form was simply a convenient representation that made the mapping process easier to visualize. The graph form was produced manually by simply drawing the Cantata workspace as seen on the workstation screen. In CHAMPION, the workspace data files will be automatically converted to a proprietary data structure

2) Data Sizing and Glyph Selection

The graph form is missing much of the information necessary to complete the mapping process. The vertices are identified by the type of operation each one represents, but there is no information associated with the edges. This is not a problem in Cantata, since all data is treated the same way. In hardware, however, the number of bits wide each connection is must be known. This information must be determined before the mapping process can continue.

The information about the vertices is also not complete. The type of glyph is known (e.g., addition, clip, etc.), but the exact version of the glyph is not known. As discussed previously, it is not enough to know that an addition glyph is needed in hardware; the input bit width of the glyph (and possibly other parameters) must be known as well. The process of filling in this information for both the vertices and edges must proceed simultaneously, since the inputs to the glyphs determine which specific glyph is needed, and the output of the glyph determines the width of the input to the next glyph.

The bit width of the data entering the workspace must be supplied by the user. Other information, such as the size of the image being processed, may also be required of the user. For the convstream glyph, both the image size and input size must be supplied. Once these two pieces of information are known, the correct version of the convstream glyph, convstream_8_256_256 can be selected from the library, and the new designation associated with that vertex of the graph. Each hardware glyph has an information file associated with it that contains, among other things, information about the size of the outputs of the glyphs. These information files are referred to as INF files, due to the file extension used to designate them. By referring to the INF file for the convstream_8_256_256 glyph, it can be determined that all of its outputs are 8 bits wide. This information is then associated with the nine edges that represent the outputs of the glyph.

Next, the four adders to the immediate right of the convstream glyph can be specified. Since their inputs are 8 bits wide, the add_8 glyph is selected from the library, and that designation is added to the corresponding vertices. The INF file indicates that the output of the cdd_8 glyph is 9 bits wide, so that information is associated with the appropriate edges. This process is repeated for the next two adders, which are to the immediate right of the four add_8 glyphs, and for the next adder to the right of those. The appropriate edges are specified using information from the INF files. All of the vertex and edge information determined so far can be seen in Figure IV-4. At this point, a problem arises. The last adder gets one of its inputs from the convstream_8_256_256 glyph and the other from the add_10 glyph to its right. These two inputs are supplied by edges with different widths. Since all of the hardware glyphs have been designed to have all of their inputs be the same width, a glyph must be inserted to convert one of the inputs to be the same as the other input. This can be done by inserting a pad_high_8_11 along the edge running from the convstream_8_256_256 glyph to the last adder. This glyph simply adds three zeros to the left of the most significant bit of the input, producing an 11-bit output.

The pad_high_8_11 glyph does not exist in the Cantata workspace; it must be added during the mapping process to ensure that the data is in the correct form. Although this glyph does not have an equivalent in Cantata, it is otherwise the same as the other hardware glyphs. It exists in pre-compiled, XNF form, it has an associated INF file, and it handles the control lines correctly. Its function is very simple, however, and when the entire design is synthesized, it will be represented by a wiring directive telling the FPGA place and route software to connect three of the inputs of the adder to ground, representing logical zero. By treating it the same as any other glyph, though, the mapping process is simplified, since all vertices can be treated identically.

Once the pad glyph is added, the last adder can be specified and the annotation process continued. The shift glyph shifts the data, but it does not change the number of bits.

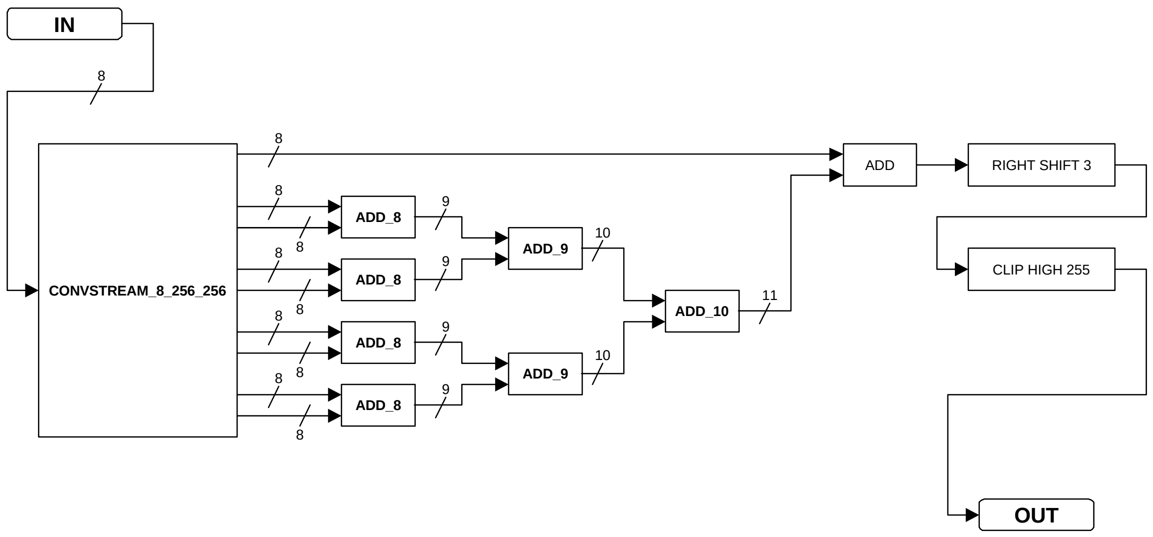


Figure IV-4. *Data Alignment Problem in Low-Pass Filter Graph.*

Since we want the output to be that same width as the input, a truncate_high_12_8 glyph must be added before the output. This is another data conversion glyph, like the pad glyph. It has no corresponding glyph in the Cantata workspace. It simply truncates the four most significant bits of the input. Since the inputs have been clipped to 255, we know that the four high bits must be zero, so no information is lost. The data graph with complete annotation for glyph type and connection width is shown in Figure IV-5.

3) Data Synchronization

The next step in the mapping process is data synchronization. Each glyph takes a certain number of clock cycles from the time it is presented with a valid data stream to output a valid data stream. This value is the latency of the glyph. It can be determined from the VHDL template and is stored in the INF file for the glyph. Each edge is given a time value. The edge at the output of a glyph is given a time value equal to the input plus the latency of the glyph. By assigning the input to the graph time value T = 0, and

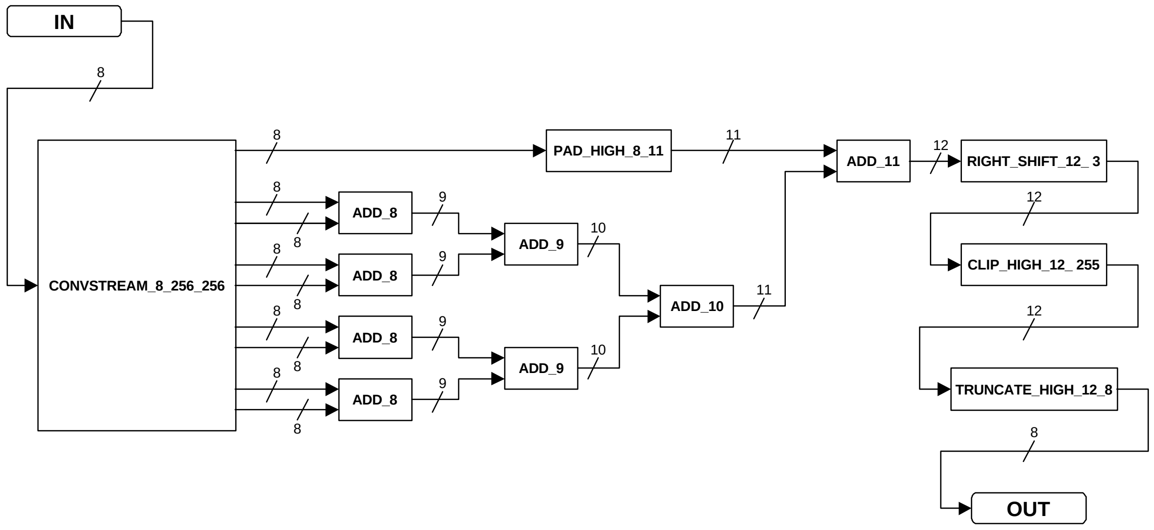


Figure IV-5. *Low-Pass Filter Graph with Data Widths and Glyph Types Annotated.*

adding the latencies of each glyph in each term, data synchronization errors can be found and corrected.

If the input is of our example workspace is given the time value T = 0, we must next look at the latency of the Convstream_8_256_256 glyph. Its INF file indicates that the latency is 257 clock cycles. This means that all of the outputs will be given the time value T = 0 + 257 = 257. All of the adder glyphs have latency 1, so the outputs of all of the add_8 glyphs will have time value T = 257 + 1 = 258. This process can be continued in the same manner for the add_9, add_10, and pad_high_8_11 glyphs. The graph with time annotations at this point can be seen in Figure IV-6. Each glyph has been annotated with its latency.

There is a problem at the inputs to the add_11 glyph. One input has time value T = 257 and one input has time value T = 260. Since the time values are not equal, the data will not be synchronized. To synchronize the data, a delay of D clock cycles must be inserted before the input with the lowest time value, where D is the difference between

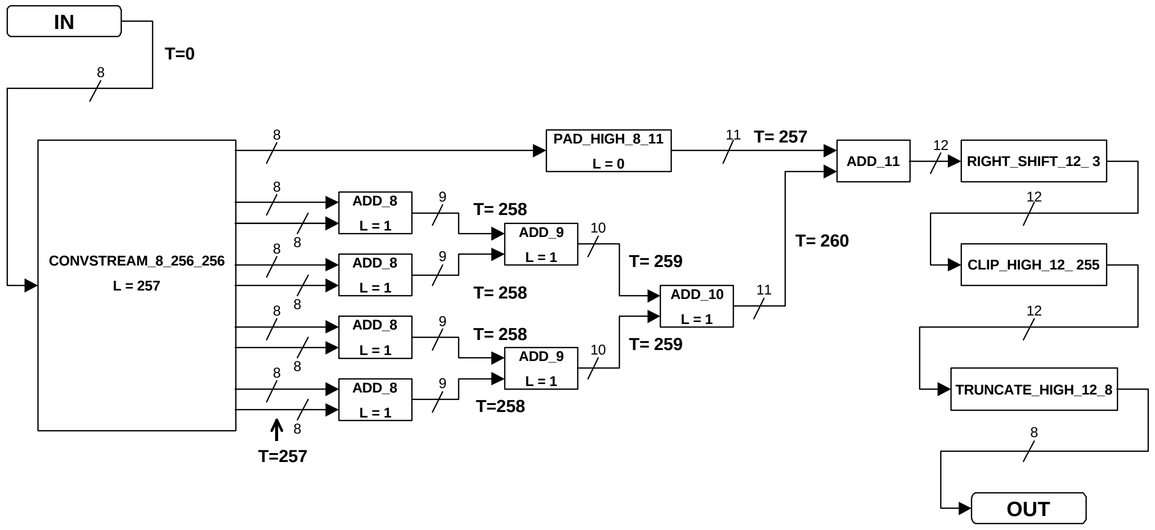


Figure IV-6. *Partial Time Annotation for Low-pass Filter Graph.*

the two time values. In this case, D = 260 – 257 = 3 clock cycles. The delay can actually be added in either of two places. It can be added before *or* after the pad_high_8_11. In either case, the data synchronization problem will be corrected. If it is added before the pad_high_8_11 glyph, the delay will have to buffer data that is 8 bits wide. If the delay is added after the pad_high_8_11 glyph, it will have to buffer data that is 11 bits wide. Since either position will work, it makes sense to use the position that will use less hardware resources. It will require more hardware to buffer 11 bit data than to buffer 8 bit data, so the delay glyph should be added before the pad_high_11_8 glyph. Now that the data size and delay values are known, the correct delay glyph can be specified. The delay_3_8 glyph can now be added to workspace. The time value annotation process continues through the reminder of the graph. Even though there are no more glyphs with more than one input, and thus no potential for data synchronization errors, the latency for the entire graph should be computed in case it is used as part of another, larger graph. The completely annotated graph, which also shows the location of the delay glyph, can be seen in Figure IV-7.

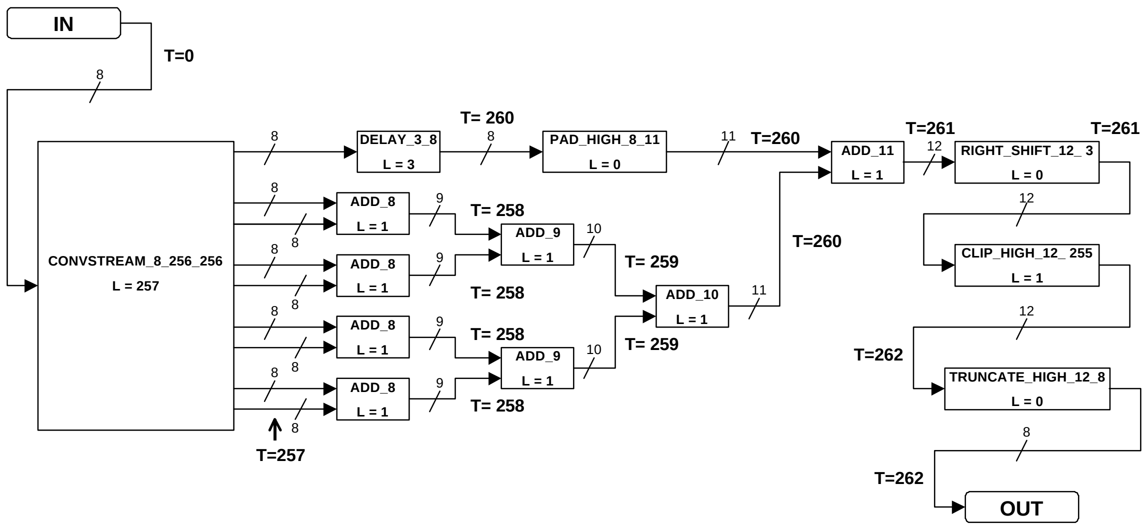


Figure IV-7. *Low-pass Filter Graph with Complete Time Annotation.*

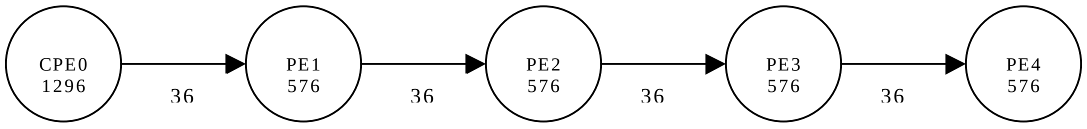
4) Partitioning

a) Background

At this point in the mapping process, the application graph is essentially complete. The original Cantata workspace has been converted into a graph form, and the graph has been annotated with the necessary information to specify the glyphs and the connections between them. The differences between Cantata and hardware have been accounted for by inserting glyphs to take care of data conversion and data synchronization. Now the application must be partitioned to fit the available hardware.

This partitioning takes the form of mapping the vertices and edges of the application graph onto the vertices and edges of the hardware graph. The hardware graph is a set of vertices representing the hardware resources and a set of directed edges representing the connections between them. Each vertex has a number representing the amount of hardware resources available in the vertex, measured in CLBs. Each edge has a number representing the number of available one-bit connections. The hardware graph for the

Wildforce-XL board as used in this project is a simplified version of Figure III-6 and can be represented simply as shown below. If a different CCM is to be used, or a different



configuration of the Wildforce-XL board, than the hardware graph would be changed correspondingly. In this way the partitioning process is not specific to the CCM architecture used.

Assigning the glyphs of the application graph to the resources on the hardware is called spatial partitioing. If the application graph does not fit on the hardware graph, than the CCM board must be reconfigured. This produces, in effect, another identical hardware graph onto which the remainder of the application graph can be mapped. This reconfiguration process can be repeated as many times as is necessary for the entire application graph to be mapped to hardware. This mapping to different board configurations is called temporal partitioning, since the board configurations are not present simultaneously; they exist at different points in time.

b) Constraints

As the glyphs of the applications graph are mapped to the hardware graph, several constraints must be observed. First, the total number of CLBs used by the glyphs mapped into each FPGA must not exceed the number of CLBs available in the FPGA. The number of CLBs available in each FPGA is dependent on the board architecture and is known in advance. This information is part of the hardware graph. The number of CLBs used by each glyph is also known in advance. When the individual glyphs are synthesized, the synthesis tool determines the number of CLBs used by the glyph and this inrofmation is stored in the INF file. As each glyph is added to an FPGA, a running total of the number of CLBs occupied must be kept to ensure that the capacity of the FPGA is not exceeded.

The second constraint is that the number of edge connections in each direction is not exceeded. As each glyph is assigned to an FPGA, connections from the glyph to glyphs in other FPGAs must be made through the edges connecting the vertices of the hardware graph. These edges have capacities, in terms of the number of bits that they can carry from vertex to another, in each direction. In the hardware graph for the Wildforce-XL as used in this project, the capacity of the connection from one FPGA to the next is 36 bits in one direction and zero bits in the other direction. As each glyph is added to an FPGA, the connection capacity used by connections to this glyph must be calculated and checked to ensure that the available connection capacity is not exceeded.

The third constraint is imposed by the architecture of the Wildforce-XL board. Each FPGA on the board has one bank of associated SRAM. There is a dual-port memory controller for each bank of SRAM. One port of the controller is used by the host interface and one port is available to hardware in the FPGA. Since only one memeory access port is available to the glyphs in the FPGA, the SRAM cannot be read and written to at the same time. Given the design of the glyphs, only one glyph can access the RAM in a given FPGA. This means that only one glyph that requires FPGA access can be placed in each FPGA.

Reconfiguration imposes another consideration on partitioning. Whenever the board is reconfigured, intermediate data streams must be stored in RAM while the reconfiguration takes place. Then the intermediate data must be read out of RAM once the newly configured board start running again. This means that a RAM write glyph and a RAM read glyph must be inserted into the application graph whenever a reconfiguration takes place. These glyphs take up resources and may affect the partitioning. Since these glyphs must access RAM, this also means that there cannot be any other glyphs that need to access RAM in the same FPGA, which can also affect the partition.

c) Partitioing Algorithm

There are numerous methods of partitioning a graph. There are many papers addressing various partitioning algorithms. An overview of these, with an extensive list of references can be found in [39]. The ones most commonly used are iterative improvement methods that optimize the partition for some cost function. These can often be computationally intensive. For the manual implementation, all that was desired was one useable partition. Rather than use an automated partitioning method, a simple partitioning heuristic was developed. A simple method was possible since the Wildforce-XL board was being used in a simple linear array configuration. If a more complicated configuration had been used, then a more complex partitioning method would likely have been needed to produce an acceptable partition.

Partitioning was done at a macro level first. A macro is a set of hardware glyphs that performs a specific function and that can be considered as one unit. The low-pass filter used as an example earlier in this chapter is an example of a macro. Using macros simplified the partitioning and synthesis of the application. A macro can be considered to be a large glyph for partitioning purposes, but a macro can be broken down into individual glyphs as needed to implement an efficient partitioning.

Starting at the source node of the graph, the object was to pack in as many hardware glyphs as possible in to the first FPGA. If a glyph could not fit in the first FPGA, it was placed in the next FPGA to the right, as seen in the hardware graph on page 64. Similarly, if placing a glyph in one FPGA violated the connection constraint or the RAM access constraint, it was placed in the next FPGA to the right. If a glyph did not fit in this next FPGA, it was placed in the one after that, as always working from left to right. Each glyph was taken in turn and in placed in the first FPGA it could fit in without violating any of the constraints. Once a glyph was placed, it was not moved again. Once as much of the application as possible was placed on the hardware graph, a reconfiguration was inserted and the remaining portion of the application graph was placed on the new hardware graph. This process was repeated until all of the glyphs had been assigned to a particular configuration of a specific FPGA.

5) Synthesis

Once partitioning was complete, it was necessary to create the programming files to actually implement the desired hardware in the FPGAs on the Wildforce-XL board. The programming files are created by first producing a VHDL file representing the hardware desired for each FPGA. Since the glyphs themselves are pre-synthesized, it is only necessary to identify the glyphs used, the connections between glyphs, and the connections between glyphs and the other FPGAs. This can be done by using structural VHDL. All of the behavioral information is in the pre-synthesized XNF files for each glyph.

There are several other VHDL files supplied by the manufacturer of the Wildforce-XL that specify the board architecture, internal interface logic for each FPGA, and global signals present on the board. These files are combined by the synthesis tool with the one file created for the specific application to produce an XNF file describing the hardware to be implemented in the FPGA. One of these XNF files is required for each configuration of each FPGA.

The Xilinx place and route software is then used to map the hardware description in the XNF file to the specific resources available in each FPGA. This results in a programming file which specifies the configuration of all the function generators and storage units in the CLBs, as well as the configuration of all of the programmable interconnections in the FPGA and the configuration of the IOBs. This programming file can then be downloaded to the FPGA to specify its behavior. One programming file is needed for every configuration of each FPGA.

6) Host Code Generation

The Wildforce-XL board used for this project was installed in a Sun workstation. The workstation acts as a host for the Wildforce-XL board, and a program running on the host takes care of certain functions necessary to enable the Wildforce to be used. The manufacturer of the Wildforce-XL board provides a driver and set of function calls to communicate with the board. These function calls must be used to create the host program, a C program created and compiled by the user. This host program must initialize the Wildforce board and download the necessary programming files for each FPGA. The host program also stores data to the SRAM on the Wildforce and reads result data back from the SRAM. The host program reads image files from the workstation hard drive to be used as input to the application, and writes the application results back to the hard drive as another image file. In some cases, the formatting of the data must be changed when it is moved from to SRAM to disk or vice versa.

Each application will require a different host program, depending on the number of reconfigurations, where data is transferred inside the application, and a number of other factors. A small set of operations should be sufficient for most, if not all host programs, however. In order to make it possible to generate this host code automatically, a modular library was created by the author specifically for CHAMPION. This library contains all of the functions that would be needed to create a host program. By calling these functions in series, a working host program can be created. While this was done manually for this project, it is hoped that the modular nature of this library will enable the development of an automatic host code generator as part of the CHAMPION system.

A pseudo-code representation of the host program used for running the START algorithm on the Wildforce-XL is shown in Figure IV-8. Each line of pseudo-code corresponds to a function call or small section of standardized code. First, the input data is read from the workstation hard drive and formatted for storage in SRAM. Communications are established between the board and the host computer, and then the Wildforce-XL board and the crossbar are initialized. For each configuration of the board, the board must be setup for configuration, and the programming files for that configuration must be downloaded. Any data needed is written to SRAM, and then execution is started. When the hardware is finished, it signals the host program using a hardware interrupt. The host program then reads any necessary data from SRAM and proceeds to the next configuration, if any. When the final configuration is complete, the results can be written out to workstation hard drive.

```
MAIN {

read in input image from disk;
format input image for storage in SRAM;
open session with Wildforce board;
initialize Wildforce board;
configure crossbar;

// First Configuration
setup board for configuration;
download first set of programming files to FPGAs;
write input data to Wildforce SRAM;
signal board to begin execution;
wait for interrupt signaling completion of execution;
read intermediate data from Wildforce SRAM;

// Second Configuration
setup board for configuration;
download second set of programming files to FPGAs;
write intermediate data back to Wildforce SRAM;
signal board to begin execution;
wait for interrupt signaling completion of execution;
read intermediate data from Wildforce SRAM;

// Third Configuration
setup board for configuration;
download third set of programming files to FPGAs;
write intermediate data back to Wildforce SRAM;
signal board to begin execution;
wait for interrupt signaling completion of execution;
read intermediate data from Wildforce SRAM;

// Fourth Configuration
setup board for configuration;
download fourth set of programming files to FPGAs;
write intermediate data back to Wildforce SRAM;
signal board to begin execution;
wait for interrupt signaling completion of execution;
read results from Wildforce SRAM;

format result data for disk storage;
write out result image to disk;
close Wildforce board;

}
```

Figure IV-8. *Pseudo-Code for START Host Program.*

V. RESULTS

The entire START application was successfully run on the Wildforce board. Four board configurations were required, an initial configuration and three reconfigurations. Details of these configurations, the results of the running the application and timing data will be presented in this chapter.

A. Configurations

The hardware equivalent Cantata workspace for the START application was converted into graph form, as discussed in section IV.D.1. The glyph selection and connection sizing discussed in section IV.D.2 was then applied to the graph. Finally, the data synchronization procedure described in section IV.D.3 was applied. The final application graph was then partitioned onto the Wildforce-XL graph using the algorithm discussed in IV.D.4

Due to the partitioning constraints presented in section IV.D.4.b, the entire application could not fit in one configuration of the Wildforce board. In total, four configurations were required. Only two configurations would have been necessary if the number of CLBs needed was the only consideration. However, the constraint preventing the placement of two hardware glyphs requiring RAM access in the same FPGA made it impossible to use fewer than four configurations. The later portions of the START algorithm require frequent RAM access and relatively less computation, so the constraint on RAM access dominated the glyph placement.

Due to the repetition of the label function, and the necessity of beginning each configuration with a RAM read and ending with a RAM write, some of the FPGAs ended up with the same glyphs from one configuration to the next. This meant that not every FPGA needed to be reprogrammed every time. Figure V-1 shows which FPGAs were programmed for each board configuration.

A close examination of the resources used in each configuration illustrates how much the RAM access constraint dominated the placement process. Table V-1 shows how many of each hardware resource were used in each FPGA configuration. The total resources used by any of the FPGAs did not exceed 57%. This number was not higher due to the use of CLBs as the only measure of hardware resources. Keeping track of the

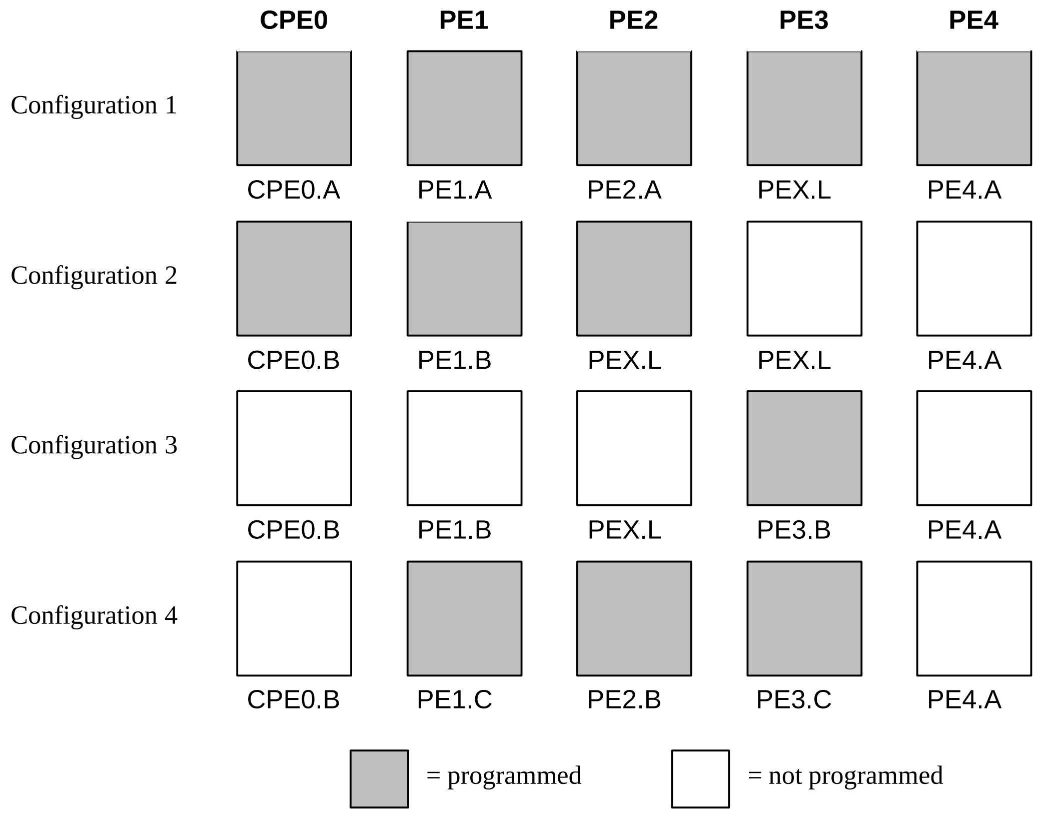


Figure V-1. *FPGA Programming for each Board Configuration.*

Table V-1. *Hardware Resource Usage for Each FPGA Configuration.*

FPGA Config.	Calculated CLB Usage	Actual CLB Usage	Flip-Flop and Latch Usage	3-Input FG Usage/ 4-Input FG Usage	I/O Pin Usage	Total FPGA Resource Usage %
CPE0.A	1159/1296 (89 %)	1296/1296 (100 %)	1853/2592 (71 %)	242/1296 (18 %)/ 1601/2592 (61 %)	33/36 (92 %)	3696/6480 (57 %)
PE1.A	452/576 (78 %)	566/576 (98 %)	600/1152 (52 %)	114/576 (19 %) 691/1152 (59 %)	26/36 (72 %)	1405/2880 (49 %)
PE2.A	488/576 (85 %)	548/576 (95 %)	800/1152 (65 %)	84/576 (14 %) 559/1152 (48 %)	4/36 (11 %)	1443/2880 (50 %)
PEX.L	374/576 (65 %)	502/576 (87 %)	604/1152 (51 %)	53/576 (9 %) 386/1152 (33 %)	8/36 (22 %)	1043/2880 (36 %)
PE4.A	52/576 (9 %)	75/576 (13 %)	108/1152 (13 %)	11/576 (1 %) 51/1152 (4 %)	N/A	170/2880 (6 %)
CPE0.B	32/1296 (2 %)	53/1296 (4 %)	45/2592 (1 %)	10/1296 (1 %) 79/2592 (3 %)	5/36 (14 %)	134/6480 (2 %)
PE1.B	374/576 (65 %)	495/576 (85 %)	600/1152 (51 %)	53/576 (9 %) 374/1152 (32 %)	8/36 (22 %)	1027/2880 (36 %)
PE2.B	63/576 (11 %)	90/576 (15 %)	115/1152 (9 %)	30/576 (5 %) 99/1152 (8 %)	26/36 (72 %)	244/2880 (8 %)
PE3.B	0/576 (0 %)	6/576 (1 %)	2/1152 (1 %)	0/576 (0 %) 12/1152 (1 %)	8/36 (22 %)	14/2880 (1 %)
PE1.C	117/576 (20 %)	75/576 (13 %)	79/1152 (6 %)	12/576 (2 %) 100/1152 (8 %)	26/36 (72 %)	191/2880 (7%)
PE3.C	97/576 (17 %)	119/576 (20 %)	152/1152 (13 %)	30/576 (5 %) 121/1152 (10 %)	11/36 (31 %)	303/2880 (1 %)

number of each type of function generators and the storage units separately would have made the resource tracking more exact, at the expense of making partitioning more complex. For CPE0, it can be seen that the calculated CLB usage was 89%, but the actual CLB usage was only 57%. This is because the CLB usage figures from the synthesis tool do not reflect the fact one glyph may use more of a specific hardware resource than another may. To take an extreme case, two glyphs could both be identified as using 100 CLBs, with one using all of the function generators in 100 CLBs and one using all of the storage units in 100 CLBs. The partitioning algorithm would assume that 200 CLBs would be needed for these two glyphs, when they could possibly be placed in only 100 CLBs, by sharing the resources in those CLBs. This fact does not account for the very low resource utilization in the later FPGA configurations, such as CPE0.B and PE2.B. These were so low due to the RAM access constraint only allowed a very few glyphs to be placed in each FPGA.

The first board configuration is shown in Figure V-2. The number of I/O lines used by each signal leaving the FPGA is shown next to each signal. The number of CLBs used in each FPGA is shown in the corner of the block representing the FPGA. Most of the glyphs are shown as macros, to reduce the complexity of the graph. The actual partitioning was done on an individual glyph level wherever necessary. Since this is the first configuration and thus the beginning of the application, the input image must be read in before anything else can be done. This means it must be read in CPE0. This is done by a RAM read glyph, which is represented by the block labeled “Input Image” in the figure. The first processing operation, low-pass filtering, was implemented by a macro, also placed in CPE0. Two other large macros, for Sobel filtering and for computing the intensity statistics, fit into CPE0. No other macros or individual glyphs could fit into CPE0, as it was found that either the resource constraint, the I/O constraint, or the RAM access constraint would be violated by the addition of any more

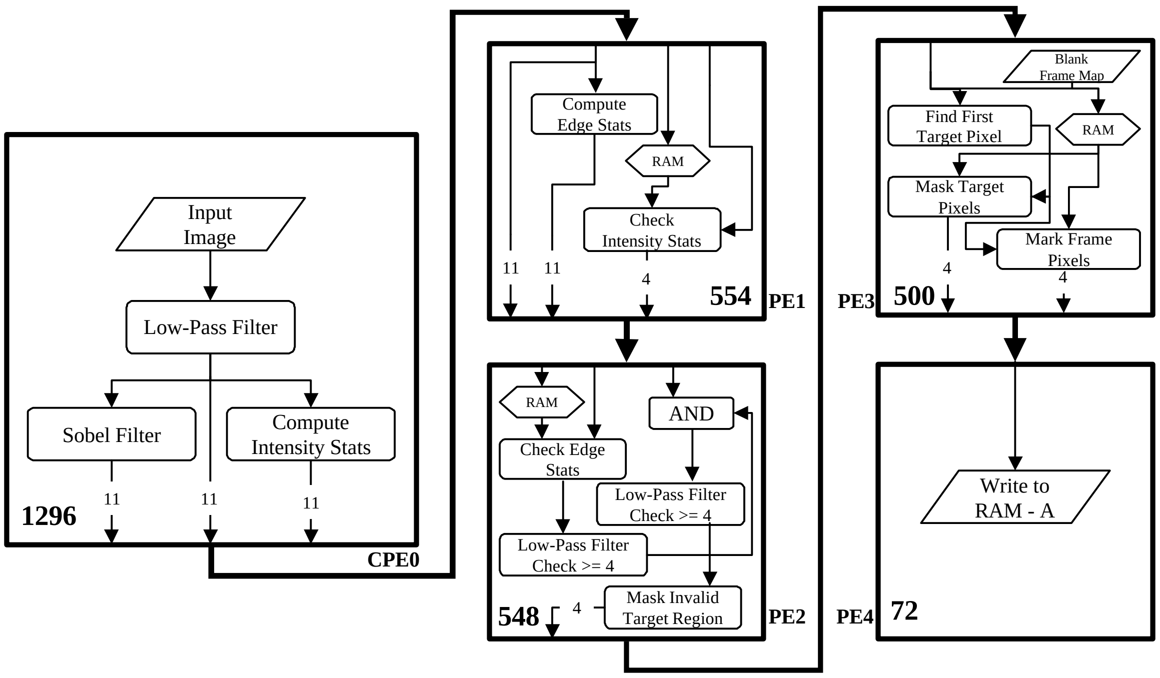


Figure V-2. *First Board Configuration.*

glyphs. Once it was placed and routed, all 1296 CLBs of CPE0 were used, although the actual resource usage was only 57%. Of the 36 I/O lines to PE1, 33 were used.

PE1 could hold two macros, for computing the edge statistics and checking the intensity statistics, as well as one RAM buffer. Its calculated CLB usage was 78%, although the actual resource usage was only 49%. PE2 and PE3 also reached fairly high levels of calculated CLB usage, 85% and 65% respectively, although the actual resource usage was much lower. The I/O usage was also well within the constraints. Since at least one reconfiguration was needed, a RAM write was needed and was inserted into the application graph. It had to go into PE4, but because of the RAM access constraint, no other glyphs could be placed in PE4 with it, as any possible glyphs also needed RAM access. The calculated CLB usage of was thus only 13%. This is the first example of how the RAM access constraint prevented efficient usage of all of the FPGA resources.

The second board configuration is shown in Figure V-3. The RAM access constraint prevented the placement of the RAM buffer in CPE0, and the first glyph in the Find First Target Pixel macro had too many I/O lines for placement in CPE0. Consequently, the only glyph that could be placed in CPE0 was the RAM read glyph. This meant that only 2% of the hardware resources in the FPGA were used. PE1, PE2, and PE3 were utilized to a much higher degree, although the actual resource utilization remained low. PE4 had the same problem as in the previous configuration and was also underutilized.

The third board configuration is shown in Figure V-4. The same RAM access problems

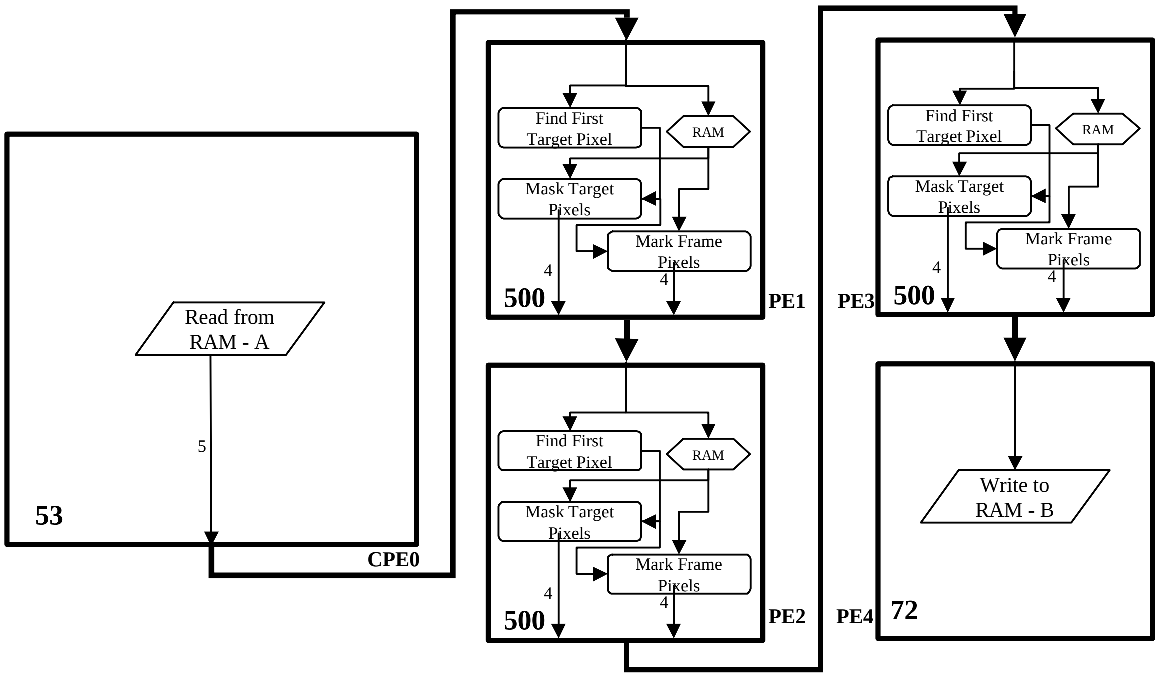


Figure V-3. *Second Board Configuration.*

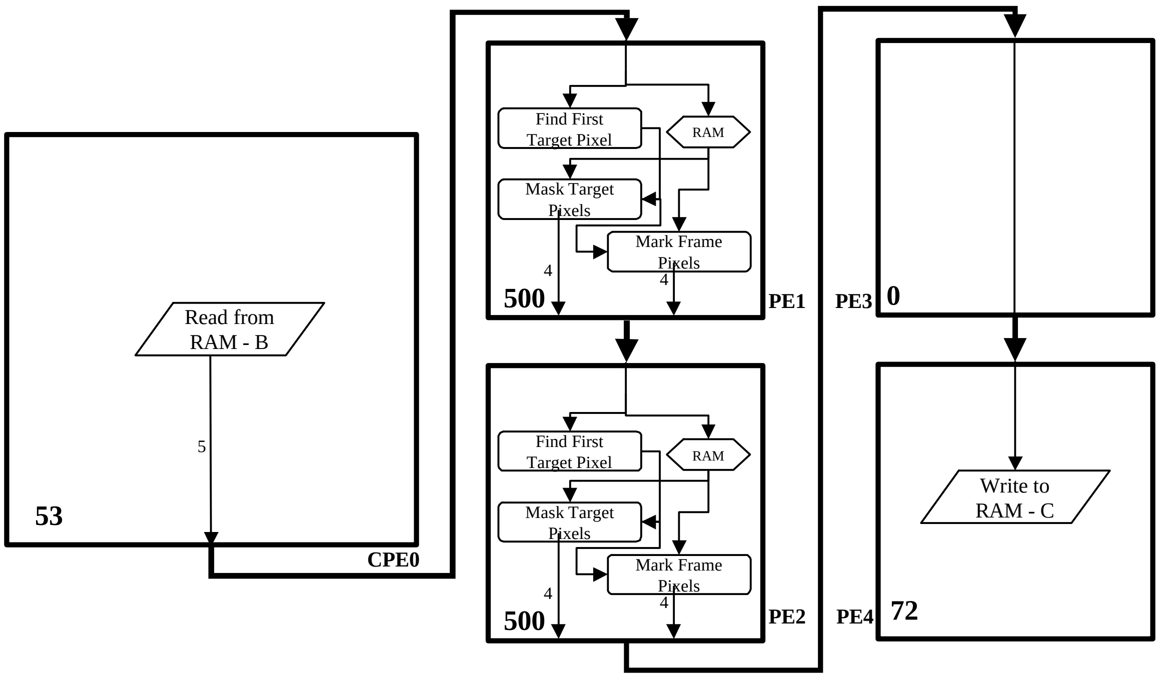


Figure V-4. *Third Board Configuration*.

were also found in CPE0 and PE4 as were found in the second configuration. Due to the high number of RAM accesses required in the remainder of the application graph, nothing could be placed in PE3, except for the RAM write glyph put in PE4. It was placed in PE4 to be consistent with the other configurations, simplifying the host code.

The fourth, and final, configuration is shown in Figure V-5. Due to the large number of RAM access glyphs, as compared to other glyphs, the resource utilization for this configuration was very low. The aggregate resource utilization for the five FPGAs in the fourth configuration was only 6%. While the START algorithm had perhaps an unusually high number of operations requiring RAM access, the low resource usage engendered by the RAM access constraint is a concern that will have to be addressed in the CHAMPION system.

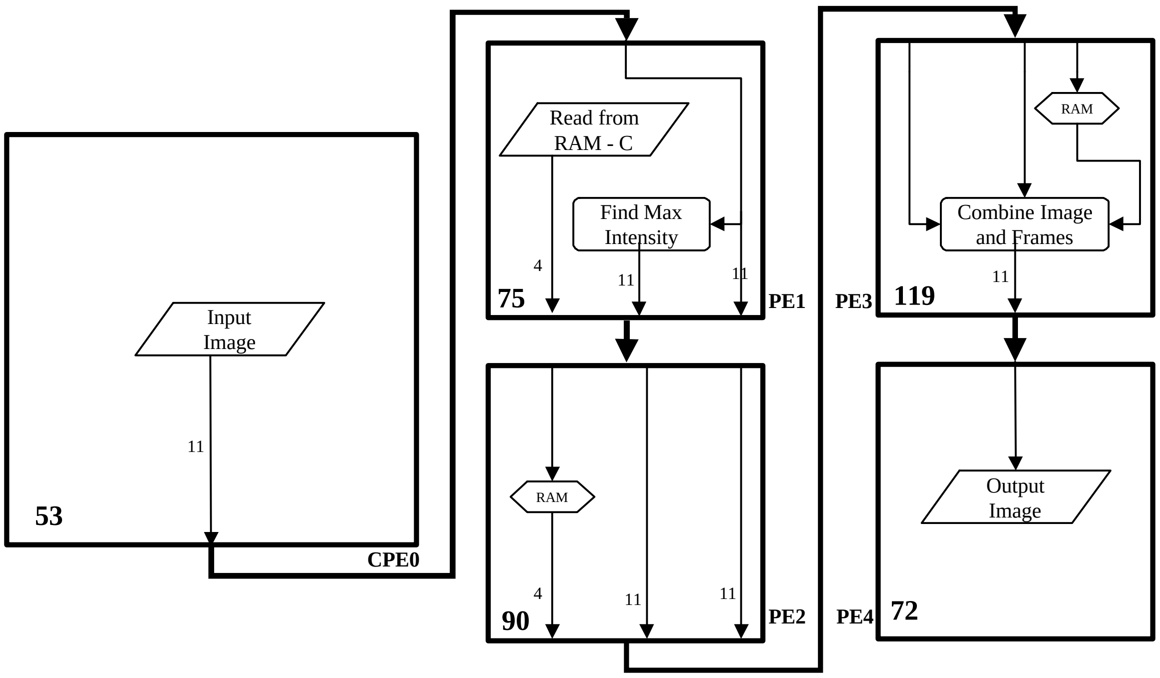


Figure V-5. *Fourth Board Configuration*.

B. Application Performance

1) Target Recognition

The entire test set of 40 images was run through the START application on the Wildforce board. The targets identified, the targets missed, and the number of false detections were recorded for each test image and compared to the results obtained by running the same test images through the hardware-equivalent Cantata implementation. The same results were obtained from the hardware implementation as were obtained from the hardware-equivalent Cantata implementation. These results are shown in Table V-2. The implementation would not be considered successful had the same results not been obtained in each implementation, so this was an important result.

Table V-2. *Results from Cantata and Hardware Implementations of START.*

Test Image #	Original Image Filename	Targets Detected	Targets Present	False Detections	Targets Detected	Targets Present	False Detections
		Cantata Implementation			Hardware Implementation		
1	nov10903.xv	3	4	0	3	4	0
2	nov10955.xv	2	4	0	2	4	0
3	nov11000.xv	1	3	0	1	3	0
4	nov11205.xv	3	4	0	3	4	0
5	nov11230.xv	3	4	0	3	4	0
6	nov11326.xv	2	3	0	2	3	0
7	nov11329.xv	2	4	0	2	4	0
8	nov11348.xv	3	4	0	3	4	0
9	nov11350.xv	3	3	0	3	3	0
10	nov11442.xv	3	3	0	3	3	0
11	nov11445.xv	4	4	0	4	4	0
12	nov31001.xv	1	1	0	1	1	0
13	nov31003.xv	2	1	0	2	1	0
14	nov31007.xv	2	2	0	2	2	0
15	nov31025.xv	1	1	0	1	1	0
16	nov31102.xv	1	1	0	1	1	0
17	nov31110.xv	1	1	0	1	1	0
18	nov31112.xv	0	1	0	0	1	0
19	nov31115.xv	1	1	0	1	1	0
20	nov31300.xv	1	1	0	1	1	0
21	nov31301.xv	1	1	0	1	1	0
22	nov31304.xv	1	1	0	1	1	0
23	nov31337.xv	1	1	0	1	1	0
24	nov31339.xv	3	2	0	3	2	0
25	nov31342.xv	1	1	0	1	1	0
26	nov31345.xv	3	2	0	3	2	0
27	nov31447.xv	1	1	0	1	1	0
28	nov31452.xv	0	1	0	0	1	0
29	nov31530.xv	1	1	0	1	1	0
30	nov31535.xv	1	1	0	1	1	0
31	nov31545.xv	1	1	0	1	1	0
32	nov31546.xv	1	1	0	1	1	0
33	nov31550.xv	0	0	0	0	0	0
34	nov31553.xv	0	1	0	0	1	0
35	nov31602.xv	1	1	0	1	1	0
36	nov31604.xv	1	1	0	1	1	0
37	nov40857.xv	0	1	3	0	1	3
38	nov40903.xv	1	1	0	1	1	0
39	nov40905.xv	2	2	0	2	2	0
40	nov40907.xv	2	2	0	2	2	0

2) Processing Time

The other test of the hardware implementation was how much faster it could process images than the Cantata implementation. The host program was written to include code to measure the time taken by each part of the process of running START on the Wildforce-XL board, broken down into Wildforce-XL configuration time, data transfer time (from the host RAM to and from the local RAM), and the actual hardware execution time. The remaining time consisted of the Wildforce-XL setup time and the time to actually run the host program. Timing varied according to server load, but the average of 50 runs gave the following values:

Total time to process one image	:	6770 ms
Board configuration time	:	5159 ms
Host code run time + Wildforce setup time	:	1544 ms
Data transfer time	:	34 ms
Hardware execution time	:	33 ms

These results are also shown graphically in Figure V-6. The time to process one image is greatly dominated by the time needed to configure the board. The actual time to process one image is only 33 milliseconds, as compared to the nearly seven seconds needed for the entire execution. The hardware implementation was still more than eight times faster than the standard Khoros implementation and 156 times faster than the hardware equivalent Cantata implementation. However, if the reconfiguration time could be eliminated, the hardware implementation would be 36 times faster than the standard Khoros implementation and over six hundred times faster than the hardware equivalent Cantata implementation. Finally, if the hardware was set up to process

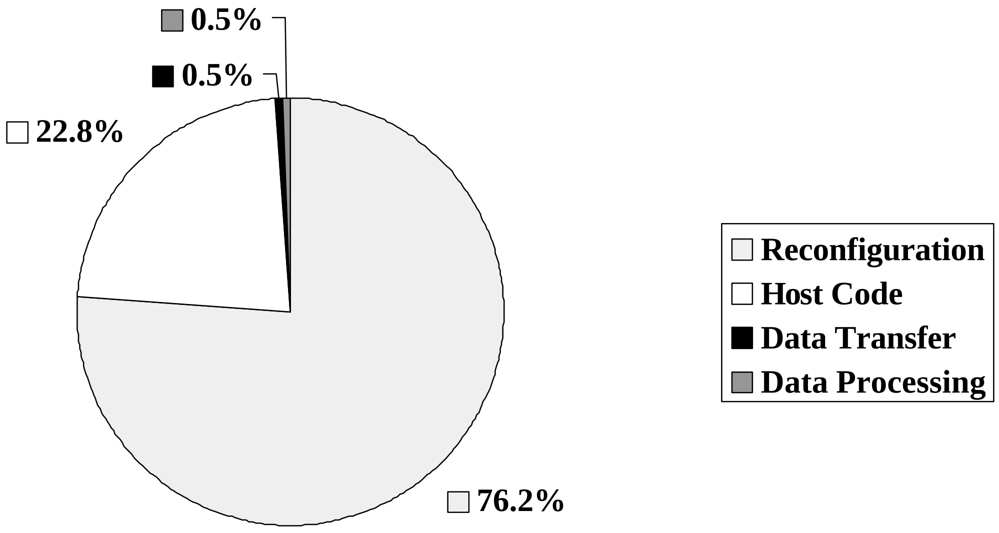


Figure V-6. *Breakdown of Image Processing Time for Hardware Implementation.*

images sequentially, with no setup time or configuration time necessary, than the hardware implementation would be over 1700 times faster than the standard Khoros implementation and over 32 thousand times faster than the hardware equivalent Cantata implementation.

VI. CONCLUSIONS

The START algorithm was successfully implemented on the Wildforce board using the hardware glyphs and the implementation strategy developed for this project. This is important not only in the context of this project, but in the larger context of the CHAMPION project as well. This success validates the design choices that were made as to the basic structure of the hardware glyphs, as well as the techniques used to perform the mapping. This will help form a foundation for the development of the automated system. Some performance issues were found, related to the high time cost of reconfiguration and the constraints placed on partitioning by the RAM access constraint. The direction of future work on the CHAMPION project can be guided by some of the results of this research. These conclusions will be discussed in greater depth in the remainder of this chapter.

A. Glyph Libraries

The hardware equivalent glyph libraries were sufficient to implement a relatively complex application in Cantata. Future application implementations will likely require the addition of more glyphs to the glyph libraries, thus expanding the range of applications that can be executed in Cantata while maintaining hardware compatibility. The hardware glyphs were shown to perform the same on the CCM as the corresponding hardware equivalent glyphs did in Cantata. This shows that despite the differences in execution between Cantata and hardware, the use of glyphs in the two implementations can be made to be congruent.

The hardware glyph design was shown to be satisfactory for use in CHAMPION system. The concept of using parameterizable VHDL templates to produce the specific pre-synthesized glyphs needed proved to be a viable approach to the creation of hardware glyphs. The control lines and other glyph interface details worked as designed, allowing glyphs to be easily and automatically connected together without consideration of the specific glyphs being connected.

One problem that arose with the hardware equivalent Cantata workspaces is that they were much slower to execute than standard Cantata workspaces. A small part of the problem was due to the larger number of glyphs needed in the hardware equivalent workspaces. This may be alleviated by identifying common macros and compiling them into single Cantata glyphs, which will run much faster than a collection of smaller glyphs connected to perform the same function. This macro approach would also make application development easier. Identifying useful macros will require careful study of the application domain, however. The main reason that the hardware equivalent workspaces were so much slower is related to the way that Cantata handles custom glyphs. This problem is supposedly remedied in the newest version of Khoros and it is hoped that updating to the new version will provide a substantial increase in speed for the hardware equivalent workspaces.

One limitation of the current glyph libraries is that there is very little support for control structures. The control in the START application is very limited, and it is inherent in the data flow itself. To implement a wide range of applications, a method of handling complex control structures in hardware must be designed, and a set of corresponding hardware equivalent glyphs developed for Cantata. The implementation of explicit control structures may require modifications of the mapping process as well.

B. Mapping Procedure

The manual mapping procedure worked, although it was very tedious and time-consuming. It took over 250 hours to complete the manual mapping, not including the time spent developing glyphs and mapping procedures. All of the procedures in mapping are, by design, well suited to implementation by computer program, so automating the mapping process should be feasible. Some portions of the mapping process may prove to require some human guidance, however, so complete automation may not be achievable.

There are more sophisticated algorithms for data synchronization that can minimize the number of delay buffers added and reduce the hardware complexity. While these are too complex to be implemented manually, it may be worthwhile to include these in the automated system. Similarly, they are much more robust partitioning algorithms that will work with a much wider range of hardware architectures and produce better results than the simple heuristic method used here. An automated partitioning method should probably track all of the FPGA resources separately in order to achieve better hardware utilization than can be obtained by tracking CLBs alone.

One portion of the automated mapping process that may require much more investigation is the generation of host code. While the manual creation of host was done in a modular manner to aid in the automation of the process, the exact mechanism by which the requirements of the host code will be determined is not clear to this author. Portions of the host code generation may have to be guided to some extent by the user. Different boards may have very different host interfaces and thus very different host code requirements, making automation even more difficult.

C. Performance

The performance of the hardware implementation was not as good as had been hoped. This was due largely to the amount of time needed for reconfiguring the Wildforce-XL board. This is a common problem in CCM architectures, although the Wildforce-XL has a particularly long reconfiguration time. There are two ways to reduce the total reconfiguration time; either the time per reconfiguration must be reduced or the number of reconfigurations must be lessened. The time per reconfiguration is determined by the architecture, so that cannot be changed except by using a different CCM. The only way to reduce the reconfiguration time is thus to minimize the number of reconfigurations needed.

Since reconfiguration time is a very large percentage of the total image processing time, reducing the number of reconfigurations would have a substantial impact on overall performance. Better partitioning could reduce the number of FPGAs needed, but the limiting factor was RAM access. The START algorithm had 13 glyphs that required RAM access. With five FPGAs available per configuration, that means that at least 3 total configurations would be needed to limit the number of RAM access glyphs per FPGA to one. Each configuration after the first requires two additional RAM access glyphs, so a total of 17 FPGAs are required to implement START. This means that despite the very low resource utilization, the four total configurations used are the minimum needed for implementing START.

Clearly some way of having more than one RAM access glyph per FPGA would be desirable. This will difficult to do on the Wildforce-XL board without combining multiple RAM access glyphs into one memory controller module, which would require some complicated graph transformations and require data scheduling, making the mapping process much more complex.

D. Future Work

Many challenges await the developers of CHAMPION. The glyph libraries must be expanded to allow the implementation of a wider range of applications. The creation of macros will have to be studied to determine which macros are needed and how best to implement their use into the mapping process. This may also be guided by the implementation of new applications. The ability to utilize more complex control structures will also likely be needed. This will require additional hardware development and potentially changes in the mapping procedure.

The mapping process must be generalized for other CCM architectures and FPGA types. New, more flexible, partitioning algorithms must be implemented and the partitioning process should be optimized to make more efficient use of hardware resources. A method of automatically generating host code may also be needed. However, even partial automation of the mapping process will greatly increase designer productivity, although partial automation, as opposed to complete automation, will limit the pool of potential users.

To increase the performance of applications on the Wildforce and similar CCMs, a way around the RAM access constraint must be found. It is not clear what form the solution to this problem may take. Additional performance gains may be realized if a way can be found to exploit the implicit parallelism of applications, thus leveraging one of the most important strengths of CCMs.

Despite the many challenges remaining, the manual implementation of the START application validated many of the basic components and concepts behind the CHAMPION project, including the glyph library structure and design, as well as the basic mapping and partitioning procedures. It is hoped that the research presented herein will help provide some of the underpinnings for future successes by the CHAMPION team at the University of Tennessee, and perhaps in some small way, other researchers working on related problems.

REFERENCES

REFERENCES

[1] S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, D.E. Thomas, "Managing Pipeline-Reconfigurable FPGAs," in *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, Feb. 1998.

[2] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, “Architecture Design of Reconfigurable Pipelined Datapaths,” in *Proceeding of the 20th Anniversary Conference on Advanced research in VLSI*, Atlanta, GA, 1999.

[3] S. Guccione, “List of FPGA-based Computing Machines,” <http://www.io.com/~guccione/HW_list.html>

[4] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine.* Los Alamitos, CA: IEEE Computer Society Press, 1996.

[5] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, “Programmable Active memories: Reconfigurable Systems Come of Age,” *IEEE Trans. On VLSI Systems*, vol. 4, no. 1, pp. 56-69, March 1996.

[6] S. Casselman, “Virtual Computing and the Virtual Computer,” in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 43-48, Napa, CA, 1993.

[7] C. Rupp, M. Landguth, T. Gaverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, “The NAPA Adaptive Processing Architecture,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1999.

[8] J. Hauser and J. Wawrzynek, “GARP: A MIPS Processor with a Reconfigurable Coprocessor,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12-21, Napa, CA, 1997.

[9] S. Hauck, T. Fry, M. Hosler, and J. Kao, “The Chimaera Reconfigurable Functional Unit,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87-96, Napa, CA, 1997.

[10] M. Piacentino, G. van der Wal, and M. Hansen, “Reconfigurable Elements for a Video Pipeline Processor,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1999.

[11] H. Keding, F. Hürtgen, M. Willems, and M. Coors, “Transformation of Floating-Point into Fixed-Point Algorithms by Interpolation Applying a Statistical Approach,” 9th International Conference on Signal Processing Applications & Technology, Toronto, Canada, 1999.

[12]M. Leong, M. Yeung, C. Yeung, C. Fu, P. Heng, and P. Leong, “Automatic Floating to Fixed Point Translation and its Application to Post-Rendering 3D Warping,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1999.

[13] M. Rencher and B. Hutchings, "Automated Target Recognition on SPLASH 2," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 192-210, Napa, CA, 1997.

[14] S. Singh and R. Slous, "Accelerating Adobe PhotoShop with Reconfigurable Logic," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1998.

[15] W. Luk, T. Lee, J. Rice, P. Cheung, and N. Shirazi, "Reconfigurable Computing for Augmented Reality," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1999.

[16] T. Miyazaki, T. Murooka, M.Katayama, and A. Takahara, "Transmutable Telecom Systems and its Application," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1999.

[17] J. McHenry, P. Dowd, F. Pellegrino, T. Carrozzi, and W. Cocks, "An FPGA-Based Coprocessor for ATM Firewalls," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 30-39, Napa, CA, 1997.

[18] Z. Luo, M. Martonosi, and P. Ashar, "An Edge-Endpoint-Based Configurable Hardware Architecture for VLSI CAD Layout Design Rule Checking," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1999.

[19] P. Chan and M. Schlag, "Acceleration of an FPGA Router," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175-181, Napa, CA, 1997.

[20] M. Abramovici and P. Menon, "Fault Simulation on Reconfigurable Hardware," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 182-189, Napa, CA, 1997.

[21] S.Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[22] D. Clark and B. Hutchings, "Supporting FPGA Microprocessors Through Retargetable Software Tools," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 195-203, Napa, CA, 1996.

[23] A. Duncan, D. Hendry, and P. Gray, "An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1998.

[24] M. Gokhale and E. Gomersall, "High Level Compilation for Fine Grained FPGAs," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 165-173, Napa, CA, 1997.

[25] T. Isshiki and W. Dai, "Bit-Serial Pipeline Synthesis for Multi-FPGA Systems with C++ Design Capture," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 38-47, Napa, CA, 1996.

[26] C. Iseli and E. Sanchez, “A C++ Compiler for FPGA Custom Execution Units Synthesis,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, 1995.

[27] J. Peterson, R. O’Connor, and P. Athanas, “Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 178-187, Napa, CA, 1996.

[28] P. Bellows and B. Hutchings, "JHDL-An HDL for Reconfigurable Systems,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1998.

[29] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek, “Object Oriented Circuit-Generators in Java,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1998.

[30] A. Wenban and G. Brown, “A Software Development System for FPGA-Based Data Acquisition Systems,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 28-37, Napa, CA, 1996.

[31] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, “A CAD Suite for High-Performance FPGA Design,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1999.

[32] R. Woods, S. Ludwig, J. Heron, D. Trainor, and S. Gehring, in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pages 155-164, Napa, CA, 1997.

[33] W. Luk, S. Guo, N. Shirazi, and N. Zhuang, “A Framework for Developing Parameterized FPGA Libraries,” in *Field-Programmable Logic, Smart Applications, New Paradigms, and Compilers*, pages 24-33, R.W. Hartenstein and M. Glesner (eds.), Springer, 1996.

[34] O. Mencer, M. Morf, and M. Flynn, “PAM-Blox: High Performance FPGA Design for Adaptive Computing,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1998.

[35] T. Drayer, “A Design Methodology for Creating Programmable Logic-based real-time Image Processing Hardware,” Ph.D. Dissertation, Virginia Polytechnic Institute, January, 1997.

[36] Colorado State University Computer Vision Group, Fort Carson RSTA Data Collection, <http://www.cs.colostate.edu/~vision/ft_carson/>

[37] R. Gonzalez and R. Woods, *Digital Image Processing*. Reading, MA: Addison-Wesley, 1992.

[38] Xilinx, Inc., *The Programmable Logic Data Book 1998*. San Jose, CA, 1998.

[39] S. Hauck and G. Borriello, "An Evaluation of Bipartitioning Techniques," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 16, No. 8, pp. 849-866, August 1997.

VITA

Ben Levine was born in Rochester, New York on August 25th, 1966. He lived in New York and New England until 1980, when his family moved to Greene County, Tennessee. He graduated from Greeneville High School in Greeneville, Tennessee in May of 1984. He attended Georgia Tech in Atlanta, Georgia for one year, before deciding to take a brief break from school. This brief break stretched to more than six years, during which time he was in the U.S. Army and later worked as a field service technician for Baxter Scientific Products in and around Boston, Massachusetts.

He returned to Tennessee to attend the University of Tennessee in Knoxville. In August of 1996, he married Ruth Lindeman, of Apple Valley, Minnesota. He received his Bachelor of Science degree in electrical engineering in May of 1997. He entered the graduate program in electrical engineering at UT in August of 1997 and worked as a teaching assistant for two semesters before beginning work as a research assistant for Dr. Don Bouldin. He has completed all of the requirements for the Master of Science degree in electrical engineering and that degree will be awarded by UT in December of 1999.

Ben Levine and his wife moved to Pittsburgh Pennsylvania, in June of 1999, where he will be attending Carnegie Mellon University and pursuing his Ph.D. in electrical engineering. He is currently working as a research assistant for Dr. Herman Schmit of CMU.