5-2001

Automatic mapping of graphical programming applications to microelectronic technologies

Sze-Wei Ong

To the Graduate Council:

I am submitting herewith a dissertation written by Sze-Wei Ong entitled "Automatic mapping of graphical programming applications to microelectronic technologies." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Donald W. Bouldin, Major Professor

We have read this dissertation and recommend its acceptance:

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Sze-Wei Ong entitled "Automatic Mapping of Graphical Programming Applications to Microelectronic Technologies." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosopy, with a major in Electrical Engineering.

Donald W. Bouldin

Donald W. Bouldin, Major Professor

We have read this dissertation
and recommend its acceptance:

Accepted for the Council:

Interim Vice Provost and
Dean of The Graduate School

AUTOMATIC MAPPING OF GRAPHICAL PROGRAMMING APPLICATIONS TO MICROELECTRONIC TECHNOLOGIES

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Sze-Wei Ong

May, 2001

ACKNOWLEDGEMENT

I would first like to thank my advisor, Professor Donald W. Bouldin, for his support, guidance and encouragement during this research. Professor Bouldin introduced me to the area of VLSI design and provided me many helpful ideas in this research. The encouragement, valuable criticism, and technical advice from Professor Bouldin has sustained my progress in the research. I also thank my other committee members, Dr. Danny Newport, Dr. Michael A. Langston, Dr. Daniel B. Koch and Dr. Chandra Tan, for their guidance and useful suggestions. I would also like to acknowledge the Defense Advanced Research Projects Agency for providing the financial support for this research under grant F33615-97-C-1124. Finally, my sincere thanks go to Ee-Laine Cheong and my parents Yok-Ting Ong and Siew-Sin Chew for their love, patience, understanding, and support.

ABSTRACT

Adaptive computing systems (ACSs) and application-specific integrated circuits (ASICs) can serve as flexible hardware accelerators for applications in domains such as image processing and digital signal processing. However, the mapping of applications onto ACSs and ASICs using the traditional methods can take months for a hardware engineer to develop and debug. In this dissertation, a new approach for automatic mapping of software applications onto ACSs and ASICs has been developed, implemented and validated. This dissertation presents the design flow of the software environment called CHAMPION, which is being developed at the University of Tennessee. This environment permits high-level design entry using the Cantata graphical programming software from KRI. Using Cantata as the design entry, CHAMPION hides from the user the low-level details of the hardware architecture and the finer issues of application mapping onto the hardware. Validation of the CHAMPION environment was performed using multiple applications of moderate complexity. In one case, the application mapping time which required six weeks to perform manually took only six minutes for CHAMPION, yet comparable results were produced. Furthermore, the CHAMPION environment was constructed such that retargeting to a new adaptive computing system could be accomplished in just a few hours as opposed to weeks using manual methods. Thus, CHAMPION permits both ACSs and ASICs to be utilized by a wider audience and application development accomplished in less time.

TABLE OF CONTENTS

CHAPTER PAGE

CHAPTER PAGE

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

FIGURE **PAGE**

FIGURE PAGE

LIST OF TABLES

CHAPTER 1

Introduction

Graphical programming environments such as Khoros [25, 31] from KRI, LabVIEW from National Instruments, and Simulink from MathWorks, allow applications to be graphically represented as a set of functional blocks connected by signal paths. By insulating the application programmer from low-level or machine-dependent programming details, these environments allow faster and easier development of complex applications. Although complex applications can be easily developed with the help of these graphical programming environments, the execution times are often long due to large input data or computationally intensive operators in the applications. For many types of commercial and military applications, which require high throughput, these long execution times are simply unacceptable.

With advances in microelectronic technology, these complex applications, which are traditionally implemented in software, can now be implemented in hardware. Two main types of integrated circuits can be used to implement these applications. The first type of integrated circuit is the programmable logic devices such as Field Programmable Gate Array (FPGA) and Complex Programmable Logic Device (CPLD). Programmable logic devices contain circuitry that can be

configured by the user to implement a wide range of logic circuits. These chips include a collection of programmable switches that allow the internal cells to be reconfigured in many different ways. The switches are programmed by the end user, rather than during the fabrication of the chip. The programmable logic devices can dramatically reduce manufacturing turn-around time and cost of low volume manufacturing.

The programmable logic devices have a major drawback. The programmable switches in these devices consume valuable chip area and limit the speed of operation of the implemented circuits. To improve the circuit performance or reduce the chip area, the circuit can be implemented in a custom-designed chip. Such chips are intended for use in specific applications and therefore, are often called application-specific integrated circuits (ASICs). The main advantage of a custom-designed chip is that it usually leads to better performance since its design can be optimized for a specific task. However, the cost of designing such chips is usually much higher than that of programmable logic devices. But if the chips are used in a product that is sold in large quantities, the design cost is amortized over the large number of copies manufactured. Therefore, the cost per chip would be lower than that of programmable logic devices.

Although hardware can be used to improve the performance of complex applications, the lack of supportive design environments results in an unacceptably long turn-around time for leveraging the benefits of hardware technology. To reduce the design time significantly, it is necessary to develop mapping tools that

allow the designers to reduce the time required to move from specifications to hardware implementation.

In this dissertation, a new approach for automatic mapping of software applications onto ACSs and ASICs has been developed, implemented and validated. This dissertation presents the design flow of the software environment called CHAMPION, which is developed by the Microelectronic Systems Research Laboratory at the University of Tennessee. This software design environment takes graphical programming applications, and automatically map them onto ACS and ASIC (as shown in Figure 1.1). The graphical programming environment used in the CHAMPION project is Cantata by KRI. The target hardware architectures of the CHAMPION project are the programmable logic devices based computing system known as Adaptive Computing System (ACS) and Application-Specific Integrated Circuit (ASIC). Using CHAMPION, application designers can develop their applications using Cantata and implement them on an ACS or ASIC.

In this dissertation, the design flow of the software to hardware mapping in the CHAMPION project is presented. Relative to contemporary technology, the design flow developed and implemented in this dissertation:

- allows the functionality to be captured faster and more accurately using precompiled functions,
- produces synchronous circuit by synchronizing the design using delay buffers,
- uses linear programming to minimize the number of delay buffers used for

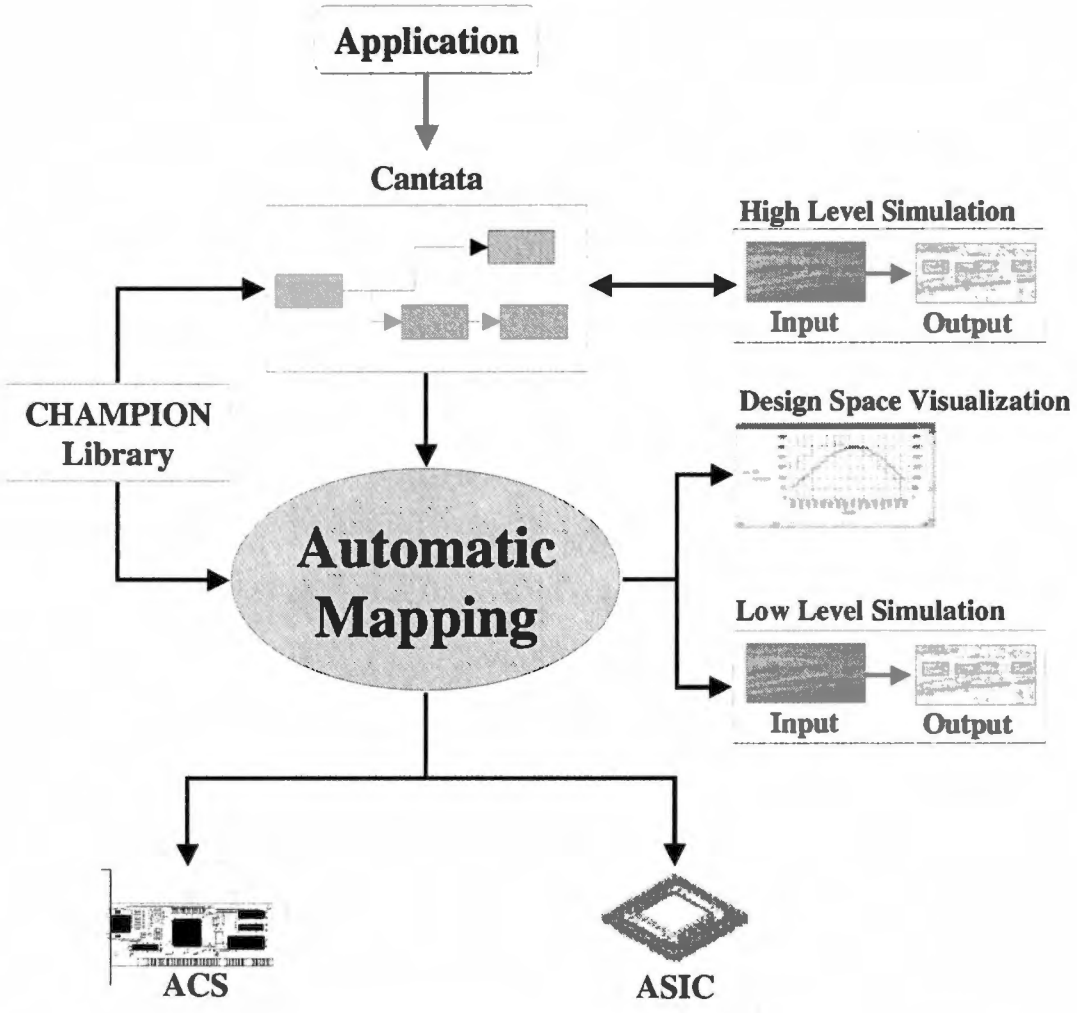


Figure 1.1: CHAMPION software design environment.

synchronizing the design

- partitions the design early instead of late to shorten the hardware mapping time
- allows the automatic mapping to be targeted to both ACS and ASIC.

This dissertation presents the algorithms, approaches, and mathematical techniques used in the design flow of the CHAMPION project. The main emphasis of the mechanisms used in this research is to shorten the mapping time while producing an implementation with performance which is compatible to the contemporary technology.

Several relatively complicated algorithms have been implemented using CHAMPION. A two criteria automatic target recognition algorithm was first implemented to validate the design flow. This algorithm was automatically as well as manually mapped onto ACS. It serves as a benchmark for determining the improvement in the designer productivity using CHAMPION. Three other algorithms have also been implemented to validate the design flow of CHAMPION. They are the neural network-based face detection algorithm [28] from the Carnegie Melon University (CMU), the Infrared Automatic Target Recognition (IR ATR) algorithm from the Army Night Vision Laboratory and a high pass filter for image processing applications.

1.1 Motivation

The designing process of ACS and ASIC presents many problems to the application programmers. The task of developing applications for ACS and ASIC requires considerable knowledge in digital logic, which is very different from the kind of algorithmic expression that arises in application programming. In addition, for ACS design, a sound knowledge of the programmable logic devices and the control and interfacing circuits on the board is required in order to take advantage of the ACS. Therefore, it remains a significant obstacle to the widespread adoption of ACS and ASIC by application programmers, who are generally unfamiliar with digital circuit design. Without supportive design tools such as CHAMPION to assist the hardware development, ACS and ASIC will go unused in many application domains.

In addition to the lack of hardware knowledge, the long and tedious process of developing ACS and ASIC also keeps application programmer from utilizing them. For instance, the implementation of an application on ACS requires considerable effort in generating, synchronizing, partitioning, and synthesizing the digital circuit. Significant effort is also required to resolve the issue of the intricate interactions between the hardware (ACS) and the software (host machine). Currently, the entire process of mapping an application onto an ACS requires months for a hardware engineer to complete. With the automatic mapping, more application designers will be able to achieve higher quality implementations in less time.

The market for consumer electronics is characterized by rapidly growing technology of hardware and rather short market windows. A key concept for coping with such requirements is the retargeting of system components on different hardware technologies. In the CHAMPION project, the design flow streamlines the process of implementing applications for new hardware technologies for designers who are concerned about their competitive position in adopting new hardware technology quickly.

1.2 Problem Statement

The problem of automatic mapping of graphical programming applications to ACS and ASIC involves translating a software application into a form that can be executed on both hardware technologies. Currently, there is a missing link between the graphical programming application and both technologies. The graphical programming applications are often developed by application programmers who lack the knowledge of hardware design while the hardware systems are often designed by engineers who are unfamiliar with the specific application domain.

A software environment which automatically maps graphical programming applications to ACS and ASIC will complete this missing link. This software environment allows application designers to perform hardware designs at a software level. To create a software design environment that provides an easier and faster way for application programmers to implement their applications on ACS and

ASIC, several issues have to be resolved:

1. How can the mapping be automated as much as possible?

2. How to shorten the mapping time while producing a high performance circuit?

3. What mechanisms should be used to handle the differences in data transfers and data sizes between the graphical programming applications and hardware circuits?

4. Which synchronization algorithms should be used to synchronize the circuit in the shortest possible time and to minimize the number of delay buffers used?

5. For ACS implementation, what partitioning method should be used to yield fast and optimized results?

6. For ASIC implementation, what compilation strategies should be used to shorten the time required to synthesize, optimize and generate the layout of the circuit?

7. What feedback estimates should be given to the user during the mapping process?

8. How can the design flow be as hardware independent as possible?

9. What is required to target a new technology?

1.3 Goals and Expected Contributions

In general, the main objective of this dissertation is to create an easier and faster way for application programmers to develop their applications for ACS and ASIC. A software design environment has been developed to map graphical programming applications automatically onto ACS or ASIC. This software design environment allows application programmers who lack the knowledge of hardware design to implement their applications onto ACS and ASIC. It also allows the designers to achieve higher quality implementations in less time. As a result, with the help of this design environment, both ACS and ASIC can be utilized by a wider audience and application development will be accomplished in less time.

CHAPTER 2

Background

A considerable body of literature exists on ACS, ASIC and their design flows. This chapter provides a brief description of ACS and ASIC, and covers the relevant aspects of the traditional design flows of ACS and ASIC.

The focus of the dissertation is upon improving hardware designer productivity by automating the mapping of graphical programming applications (GPAs) onto ACS and ASIC. Many of the tools that will be used to automate the mapping process involve handling the differences between the GPA and hardware implementation. Included in this chapter are sections containing material on the GPA used in this dissertation and the major differences between the GPA and hardware.

This chapter also includes a section describing the commercial software and other research programs that perform the mapping automation similar to that in this dissertation. The major differences between the approach taken by CHAMPION and that of commercial software and research programs are discussed.

2.1 Microelectronic Technologies

Historically, digital hardware has been divided into two main groups, general purpose processor and application specific hardware. General-purpose processor is a fixed architecture device which implements a pre-defined set of instructions. General-purpose processors commonly fall in one of two categories: microprocessors and digital signal processing (DSP) processors. Examples of microprocessors are Intel's Pentium family, SUN's UltraSparc family and Motorola/IBM's PowerPC family. DSP processors, on the other hand, include Texas Instruments' TMS320C6xxx family, Motorola's DSP560xx family and Analog Devices' TigerSHARC family. These processors execute programs stored in some internal or external memories by fetching their instructions, examining them and then executing them one after another. An organization of a simple bus-oriented microprocessor is shown in Figure 2.1. New programs can easily be loaded into memory as needed. The computation of any algorithm is determined by the software program, not the hardware. Because their instruction sets include very general operations such as arithmetic and logical operators, general-purpose processors can be programmed to perform any conceivable function. However, general-purpose processors are very slow at performing computational intensive functions. Execution of these functions requires the functions to be translated into the general-purpose instructions that are executed by the processors.

For application-specific hardware, an engineer designs all of the circuits specifically for an application. These hardware, which are often referred to as

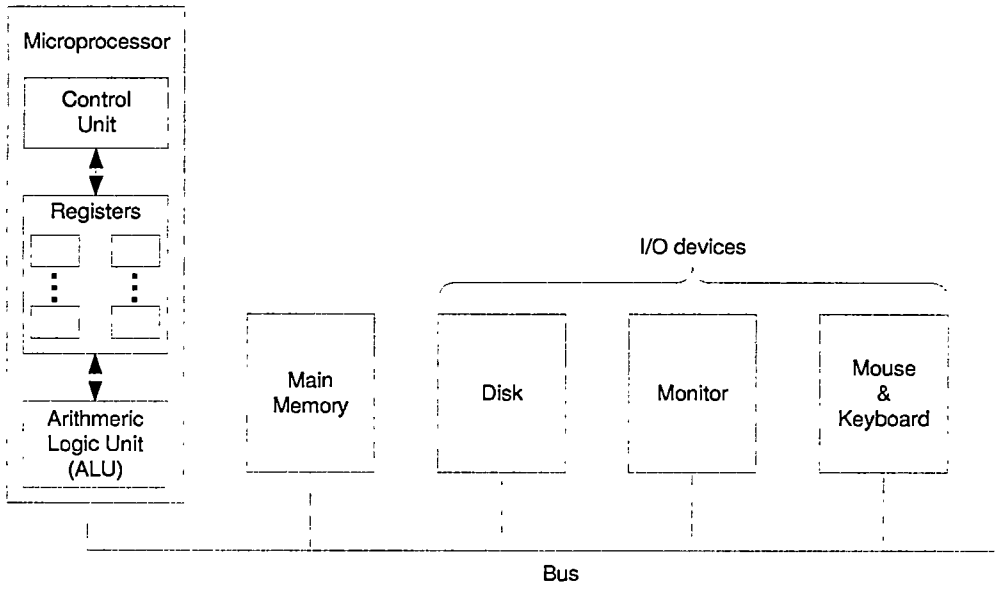


Figure 2.1: Organization of a simple bus-oriented microprocessor.

application-specific integrated circuits (ASICs), usually lead to better performance since they can be optimized for the specific application. However, an ASIC can only be designed to perform one particular application. If they are needed to perform a new function, then a new ASIC will have to be created. Another disadvantage of ASIC is its labor-intensive nature. It typically takes months for hardware engineers to design a new ASIC and have it fabricated. This labor-intensive nature of ASIC also translates it into a high cost and long time-to-market hardware. In spite of these drawbacks, application specific hardware is widely used whenever performance is of primary importance. By optimizing the hardware for a particular task, ASIC can often achieve computation speeds several orders of magnitude faster than general-purpose hardware.

In recent years, a new class of computing hardware, called adaptive computing system (ACS), has been increasingly gaining research interest. ACS has some of the advantages of both general-purpose and application-specific hardware. A typical ACS is composed of multiple programmable logic devices, memory elements and circuits for control and interface with a host computer. The architecture of the ACS board used in this research project is shown in Figure 2.2. The programmable logic devices, which are also known as processing elements (PEs), found in the ACS are commercially available Field Programmable Gate Arrays (FPGAs) or Complex Programmable Logic Devices (CPLDs). These PEs provide a relatively large number of programmable functional units and programmable interconnections. The functionality of the hardware is determined by how the functional units and interconnections 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. Different types of application can be implemented 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.

2.2 Design flows of ACS and ASIC

In this section, the entire synthesis-based ACS and ASIC design flows and methodologies are discussed. The main function of this section is to bring to the

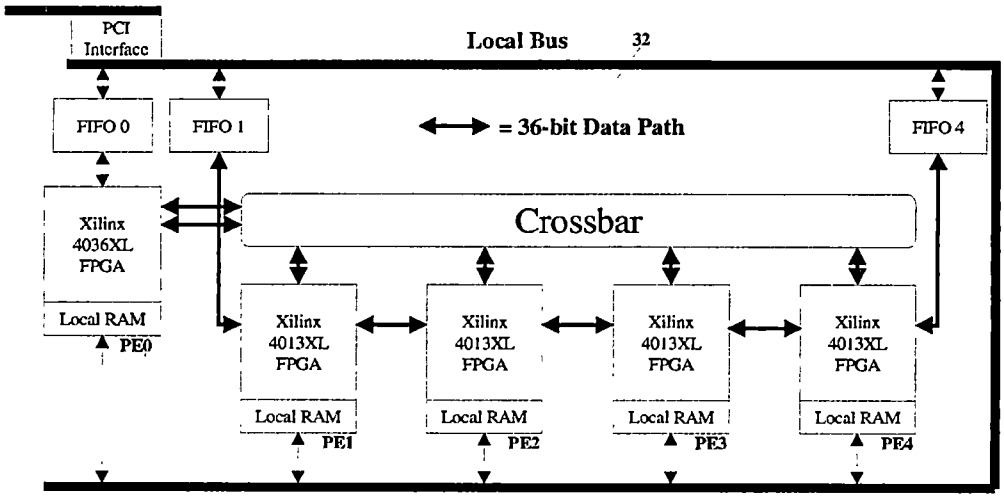


Figure 2.2: Architecture of the Wildforce board.

forefront different stages involved in ACS and ASIC design. It will provide the necessary background for understanding the design flow of CHAMPION.

2.2.1 Design Flow of ACS

The design flow for an ACS is depicted in Figure 2.3. To map an application to ACS, the designer must first define the hardware structure for the application. In the past, this was done using schematic capture, where the designer manually draws the schematics of the design using the components of a cell library. This process was time consuming and impractical for design reuse. To overcome this problem, schematic capture is increasingly being replaced with hardware description language (HDL). The two main HDLs in use today are VHDL and Verilog. For both schematic capture and HDL coding, a sound knowledge of the digital

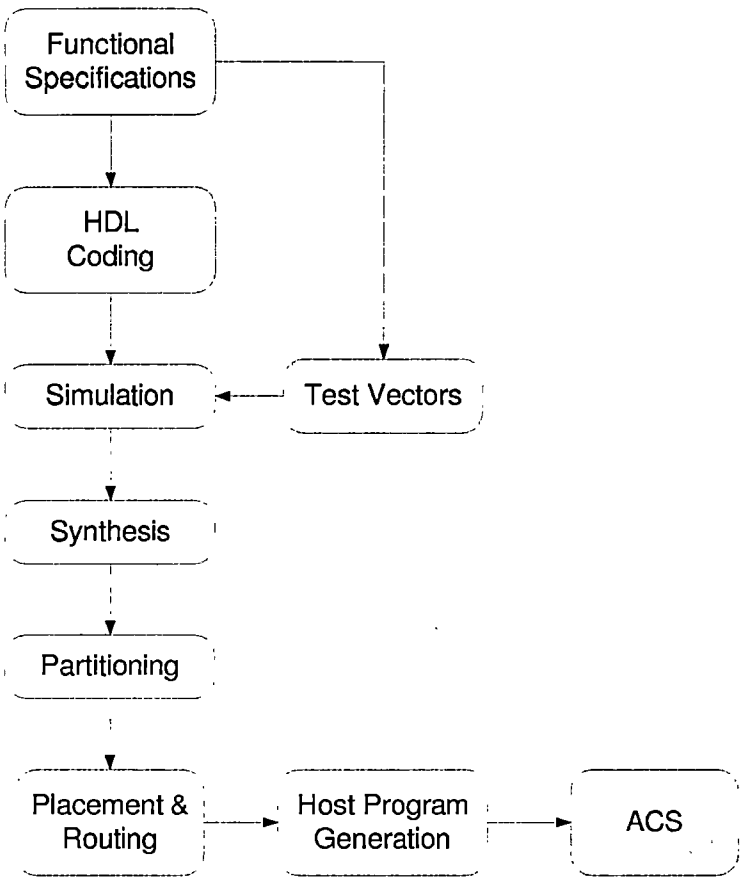


Figure 2.3: ACS design flow.

logic design is required.

After the design has been captured in schematic or HDL, it is essential to verify that the schematic or HDL code matches the required functionality, prior to synthesis. This step is commonly known as pre-synthesis behavioral simulation. If errors are found during the simulation, appropriate changes need to be made and the verification of the new design is repeated through simulation.

When the simulation results indicate that the design functions correctly, the design is then partitioned spatially. The spatial partitioning of the design is required to allow the design to be mapped onto multiple FPGAs available on the ACS. The partitioning process considers many factors such as, the size of the PEs, number of interconnections between partitions and number of memory elements in a partition. If the design does not fit in all the PEs available in the ACS board, then it must also be partitioned temporally, by allocating functional units to different configurations of the same PE.

Once the design is partitioned, pin assignment has to be performed on each partition to assign the I/O pins on each PEs to the schematic or HDL description of the partition. This is a long and tedious process since the designer has to manually assign all the input and output signals of each partition to the I/O pins on the corresponding PE. Each partition is then individually synthesized into a technology-dependent netlist. This netlist is specified in terms of the basic logic cell of the PE. For example, if the Xilinx XC4000 series FPGAs are used as the PEs on the ACS board, the netlist is specified in terms of Configurable Logic Block (CLB).

After the technology-dependent netlists for each partition has been generated, each logic cell specified in the netlists can be arranged on a layout surface of the PE. For ACS design, the main goal of the placement process is to arrange all the logic cells in such a way that the routing is feasible and all the critical nets are minimized. In ACS design, area minimization is normally not as important as it is

in the ASIC design. After placing the logic cells in that particular arrangement, the interconnections between logic cells are routed according to the specified netlist.

The placement and routing processes produce physical implementations for each partition of the design. These physical implementations are then translated into binary streams (commonly known as configuration file), which are used to program the PEs. A software program, which is normally written in C or C++, has to be generated to give instructions to the host computer on when and how to download the configuration file and data for the application to the ACS.

While commercial tools exist to help with parts of this design flow of ACS, it still requires a great deal of skill, knowledge of hardware design, time, and effort to design. These steep requirements have severely limited the potential users of this type of ACS and prevented its widespread acceptance.

2.2.2 Design Flow of ASIC

Figure 2.4 shows the sequence of steps to design an ASIC. The design process starts with the development of a hardware definition for the application. This is usually done with HDL. The functionality of the HDL is then verified against the initial specification. This can be done by assigning specific values to the input signals, performing simulation runs and viewing the output waveforms in a graphical simulation tool. An alternative is to write a testbench, which is an HDL block whose outputs provide the stimuli for the design to be simulated. In

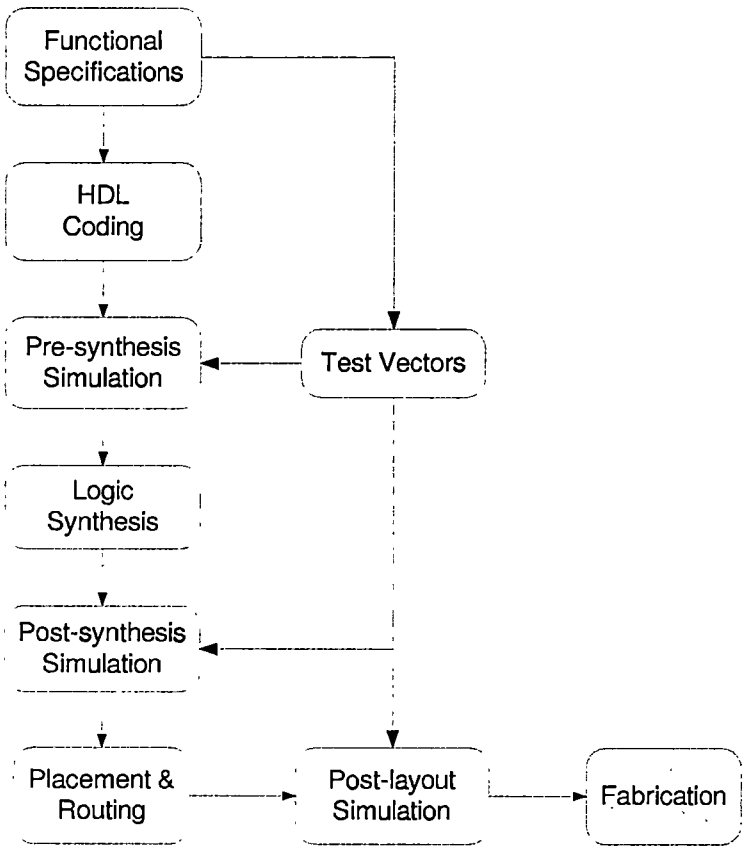


Figure 2.4: ASIC design flow.

ASIC design, testbench simulation is more commonly used compared to waveform simulation. Simulation using testbench simplifies the post-synthesis and post-layout simulation steps. The same testbench can be used for post-synthesis and post-layout simulation and the results can be compared. As a result, the testbench is used as the golden model for verifying the design at every level of abstraction

After the initial simulation, the HDL description of the design is then synthe-

sized into a netlist consisting of logic gates and their interconnections. The logic gates used in the netlist are obtained form a technology library provided by the ASIC manufacturer. The library defines the delay models, models for variations of temperature, voltage and manufacturing processes as well as the functionality of each gate. Notice that in the ACS design flow, the netlist obtained through the synthesis process is specified in terms of the basic logic block of the PE. But for ASIC design, the netlist is specified in terms of logic gates provided by the manufacturer.

This gate-level netlist is then simulated to verify the functionality of the design. This post-synthesis simulation is performed using the same specifications used during the pre-synthesis simulation. After verifying the design, the logic gates are placed on the layout of the chip. The main goal of placement in ASIC design is to find a minimum area arrangement for the gates that allows completion of interconnections between the gates, while meeting the performance constraints. This is typically done in two phases. In the first phase, an initial placement is generated. In the second phase, iterative improvements are made to the initial placement until the layout has a minimum area or best performance and conforms to design specifications. After placing the logic gates, the interconnections between logic gates are routed according to the specified netlist.

Another round of simulation can be performed after the placement and routing processes. The simulation at this level is commonly known as post-layout simulation. Post-layout simulation is performed mainly to verify that the design

meets the specified timing constraints. Once the design is verified, pin assignment is performed to connect the input and output signals of the design to the I/O pins of a chosen frame. After pin assignment, the physical layout of the design is ready to be sent off for fabrication.

2.2.3 Comparison between ACS and ASIC Design Flows

Comparing the ACS and ASIC design flows shown in Figure 2.3 and Figure 2.4, it can seen that the main difference between the two design processes is that ACS design requires the design to be partitioned before synthesis. To map a design onto ACS, the design has to be partitioned into smaller sub-designs, which can be fitted into the PEs. The partitioning process is performed based on constraints such as the size of the PE, the number of I/O pins on the PE, and the number of memory elements allowed in each partition. For ASIC design, no partitioning is required. The complete design is mapped onto a single chip.

The partitioning process in the ACS flow has caused simulation to be omitted at lower levels of abstraction. In ASIC design, simulations are performed at three different levels: pre-synthesis, post-synthesis and post-layout. The same set of specifications can used to simulate the design at these three levels since the design does not have to be partitioned into sub-designs. Once the design is partitioned, as in the case of ACS design, the initial specifications cannot be used to verify the design since each partition implements different functionality. In fact, it is almost impossible to simulate each partition since the functionality of each partition is

not known due to the fact that each partition is performed based on constraints, not according to the functionality of the partition.

The ACS design flows also requires the designer to generate a host program responsible for communications between the host system and the ACS board. To generate this host program, knowledge on the control and interface circuitry of the ACS board is required. The designer needs to understand how the control and interface circuitry on the ACS board works as well as the set of commands provided by the ACS manufacturer for using the circuitry.

2.3 Khoros Software Development Environment

Khoros is a software integration and development environment developed by Khoral Research Incorporated (KRI). It includes a suite of software development tools and a set of toolboxes containing over 300 functions. These functions 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 [31]. The functions can be used as stand alone programs from the command line, or as functions called by a C/C++ program.

All the functions in Khoros operate on data defined in three robust data models. The geometry data model allows for representation and storage of complex geometric structures. It also allows easy access and manipulation of the geometric structures. The color data model is designed for the storage of color

maps in a format that allows for easy transformation 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 [25, 26].

2.3.1 Cantata Graphical Programming Environment

While Khoros functions can be used as standalone commands, they are most widely used with the Khoros graphical programming environment called Cantata. Cantata allows users to develop their applications easily using the collection of functions in Khoros. Each function in the Khoros toolbox is represented on the screen by a small icon called a glyph. Graphical programs can be created by interconnecting these glyphs with data paths (as shown in Figure 2.5). Each 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 allows the user to set options for the operation of the glyph. Each of these objects corresponds directly to a parameter that can be passed on to each function on the command line.

Designed to act as an integrated software development environment, Khoros allows users to add new operators or functions to its collection of toolboxes. These new functions can be generated using C or C++ and installed in Cantata with the help of Khoros tools. Complex software wrappers that provide sophisticated

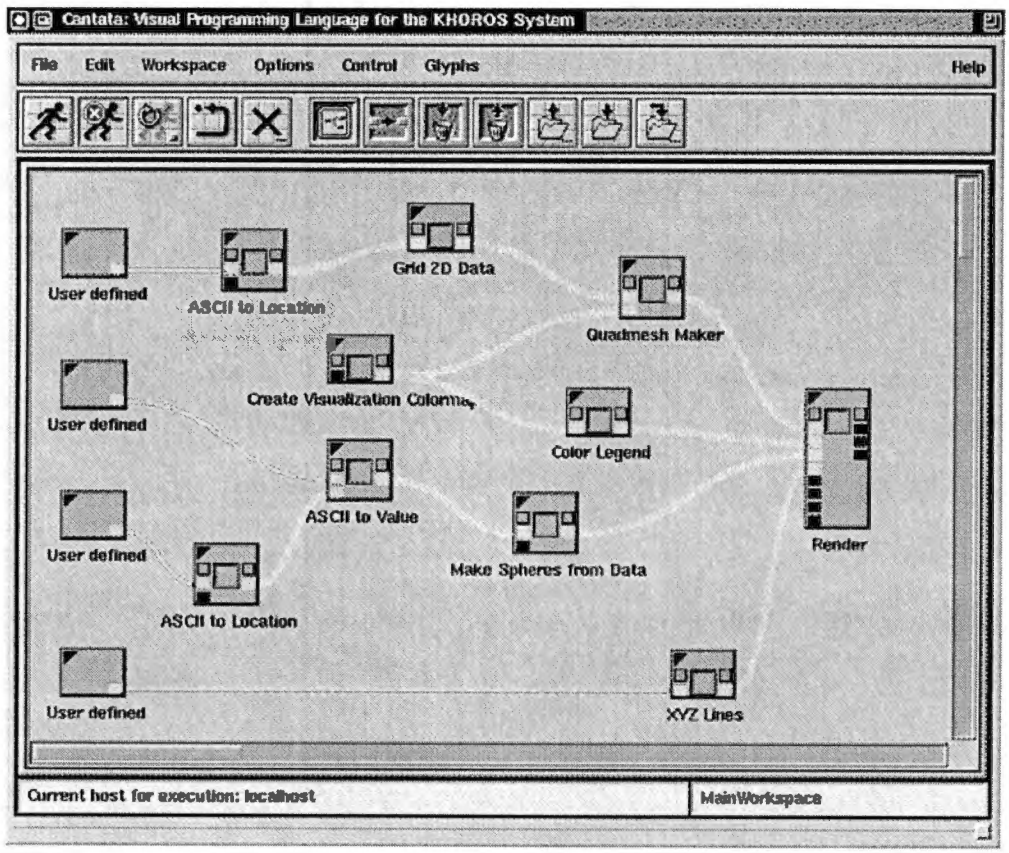


Figure 2.5: A sample Cantata workspace.

data handling are added to the glyph during the installation process.

Like most of the graphical programming environments, Cantata allows users to develop their application programs by simply interconnecting functional blocks. Once the applications are created, they can be executed in Cantata without having to compile their code. In addition, in Cantata, users do not have to worry about the details of how data is passed or where it is stored. Since Cantata uses the polymorphic data model, users do not have to be concerned about converting data from one data type to another. All these features allow users who may not be familiar with traditional programming methods to quickly implement and test their algorithms and ideas. Since Cantata can work on such wide ranges 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 physician who would like to apply an image processing algorithm onto a computed tomography (CT) scan image, but might otherwise be limited by a lack of programming ability, can use Cantata to apply the algorithm using real data. Cantata has become widely used partly because of its ability to isolate users from the underlying technology while still allowing them to exploit the power of the computing platforms they are using. By providing a high-level design environment for problem solving, Cantata increases the productivity of

both researchers and application developers, regardless of their programming experience.

2.4 Differences between Cantata and Hardware

At first glance, an application in Khoros Cantata may seem to be similar to a schematic of a hardware design. Each glyph in Cantata seems to be equivalent to a hardware cell and the connections between glyphs seem to be equivalent to the wires connecting the hardware cells. Unfortunately, upon closer examination things are not quite that simple. The Cantata programming environment handles 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 Cantata at the background during the execution of the program. In a hardware design, specific hardware modules have to be created to handle these operations. Three different types of operations that were handled significantly differently in Cantata and in hardware implementation of an application were found. The next three sections describe these operations and the way they are handled.

2.4.1 Data Transfer

In a Cantata implementation, data is transferred between glyphs through the use of temporarily files. The data output by one glyph is written out to a temporary file. Cantata will pass the filename of this temporarily file to any succeeding

glyphs that will use that data as their inputs. In other words, the input filename of any particular glyph is set automatically to be the same as the output of the preceding glyph. The succeeding glyphs can then read in the data for further processing. The user does not have to worry about the filenames for the inputs and outputs as they are chosen by Cantata. The actual transfer method involves storage on a hard drive, where the entire file is transferred all at once time. The data is stored in one of the three Khoros data structures.

In hardware implementation, data is transferred between hardware glyphs one value at a time rather than an entire block of data. Instead of transferring an entire image, as in Cantata, only a certain number of bits are transferred at each clock cycle in hardware, depending on the bus width. In addition, the hardware requires registers to hold the data values as they are being transferred between the hardware glyphs.

In order to map a Cantata application to hardware, this difference in data transfer must be accounted for. This data transfer difference will not affect the operation of a glyph if the Cantata glyph operates on the data one value at a time. For instance, a Cantata glyph that adds a constant to each value in a stream only needs to work with one value at a time. The corresponding hardware glyph can work with one data value at a time as well and the mapping of it will be simple. If a Cantata glyph performs some accumulation of operations in a stream before it produces a valid output, it will not map easily to hardware. An example of such glyph is an accumulator that finds the sum of all the pixels in an image.

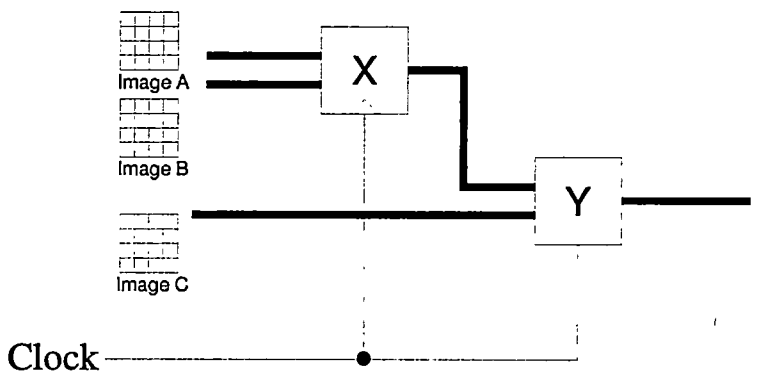
To map this type of glyph to hardware, the differences in data transfer methods must be taken into account.

2.4.2 Data Synchronization

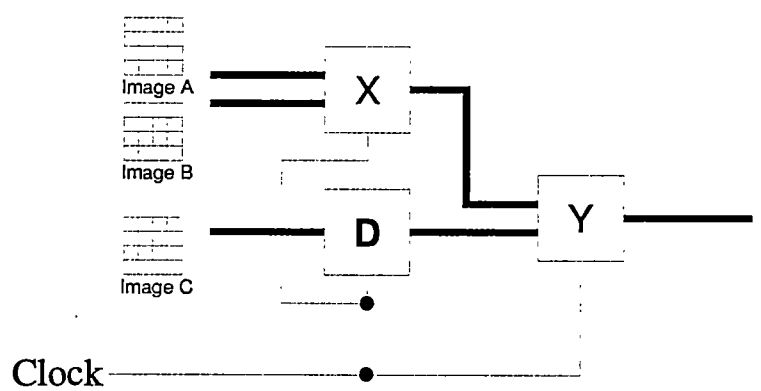
In a Cantata, executions of the programs are data driven. That is, each Cantata glyph will begin execution only when all its input data is available. With this, there are never any data synchronization errors in Cantata. However, hardware systems are clock driven. At each clock cycle, each hardware glyph will process whatever data is presented at its inputs.

Due to the difference in the processing time of each hardware glyph, data traveling over different concurrent paths may arrive at the inputs of a multi-input glyph at different times. To insure that each glyph generates the correct time-sequenced output, it is necessary that each glyph receive all its input data precisely at the same time.

In Figure 2.6 (a), a simple network for adding three images A, B and C, is shown. The two hardware glyphs, designated X and Y, are triggered by the same clock signal. Assuming that images A, B, and C all become available at the same time. In the first clock cycle, glyph X will add its two inputs, which are the first pixel from image A and the first pixel from image B. At the same time, glyph Y will also add its two inputs, which are the first pixel from image C and an invalid output from glyph X since glyph X is in the process of adding the its inputs. This causes the inputs to glyph Y to be out of sync, resulting in hardware results



(a)



(b)

Figure 2.6: Synchronization using delay buffers. (a) Unsynchronized glyphs and (b) glyphs synchronized by delay insertion.

that are different from those obtained in Cantata.

To fix this problem, a delay buffer, D, needs to be inserted at the input of glyph Y, as shown in Figure 2.6 (b). Now image C is delayed by 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. It is important to note that the delay glyphs inserted for data synchronization do not appear in the original Cantata workspace. Instead, they are identified and inserted during the mapping process.

2.4.3 Data Sizing

In Cantata, data size conversion is handled automatically by the glyphs. The polymorphic data model adopted by Cantata frees users from having worry about converting data from one size to another.

In hardware, the inputs of each glyph have fixed data width. Therefore, two unsigned five-bit numbers can be added together, but a five-bit number cannot be added to a six-bit number. If an input and an output of different sizes are connected together, one of them must be converted to match the other. Since there are so many different possible combinations of input and output sizes, it is impossible for the glyph to do this automatically. Part of the mapping process must therefore include a way of finding and fixing all the mismatches. To fix the mismatched data path, a hardware glyph can be inserted to perform the correct conversion. These conversion glyphs do not appear in the Cantata workspace.

Just like the delay glyphs inserted for data synchronization, they are identified and inserted during the mapping process.

2.5 Related Work

Several research projects and commercial software that perform similar mapping automation are being developed. In this section these research projects and commercial software are presented. The major differences between the approach taken by CHAMPION and that of the commercial software and research programs are discussed.

2.5.1 CAMERON Project: Colorado State University

Project Overview

At Colorado State University (CSU), a research project called CAMERON [7] is being developed. The CAMERON project develops compilers for mapping image-processing applications developed using Khoros onto ACS boards. New image processing operators in CAMERON are written in a high-level programming language called single-assignment C (SA-C). SA-C is a programming language based on C that has been developed at CSU and integrated in the Khoros graphical programming environment. It is created as a language well suited for image processing and amenable to compiler analysis and optimization.

SA-C is a subset of C with extensions for image processing. These extensions include parallel loops, true n-dimensional arrays of variable precision scalars,

and access mechanisms such as windowing. These extensions make it easy to implement image-processing algorithms [20].

SA-C is designed to exploit both coarse-grain and fine-grain parallelism as appropriate for ACS. SA-C eliminates recursion and pointers manipulation, and allows each variable to be assigned only once. The existence of recursion and pointers in a program complicates the analysis of data dependencies at compile-time. By eliminating recursion and pointers, SA-C makes data dependencies and access patterns clearer and therefore, allows the compiler to easily identify loop-level (coarse-grain) and instruction-level (fine-grain) parallelism. The reason for the single assignment restriction is that it establishes a one-to-one correspondence between variables in the program and wires in the resulting circuit. By allowing each variable to be assigned only once, the value of the variables will not change. Therefore, they do not need to be associated with memory. Instead, every operator is a sub-circuit, and the variables it operates on are the input wires. This will make the generation of VHDL circuits from SA-C easier. It will also make it easier for programmers to understand this mapping and to write SA-C programs that translate into efficient circuits [7].

The main goal of the CAMERON project is to allow low-level image processing algorithms to be written using SA-C inside the Khoros software development environment. These programs can then be manipulated as glyphs inside Cantata. The application generated using these SA-C glyphs can then be mapped onto on parallel architectures for execution in ACS.

The technology underlying the mapping approach taken by CAMERON project is dataflow analysis. The image processing glyphs written in SA-C can be directly translated into dataflow graphs (DFG). Once the program is transformed into a dataflow graph, dataflow optimizations such as loop merging, loop unrolling, and data blocking can be applied. Since SA-C is single assignment, a direct correspondence exists between dataflow graph components and FPGA hardware components. Therefore, optimized dataflow graphs can easily be mapped onto FPGA configurations.

The design flow of the mapping process of CAMERON project is shown in Figure 2.7. Image processing applications can be implemented in Cantata by interconnecting the SA-C glyphs. Once the application is developed using Cantata, all the glyphs are merged into a single SA-C program. The SA-C compiler is then used to optimize the SA-C program. The compiler first converts the entire SA-C program to an internal data flow form called Data Dependence and Control Flow (DDCF) graphs. Optimizations are then performed on the DDCF graphs. The optimizations performed includes traditional optimizations such as Common Subexpression Elimination, Constant Folding and Dead Code Elimination, and optimizations specifically designed for the FPGA such as Loop Unrolling, Function Inlining and Loop Nextification [6].

After the optimization process, the compiler translates the DDFG graph to DFG graph. The DFG graph is then partitioned into DFG for the FPGA and DFG for the host. The parallelizable loops in the DFG are assigned to the FPGA.

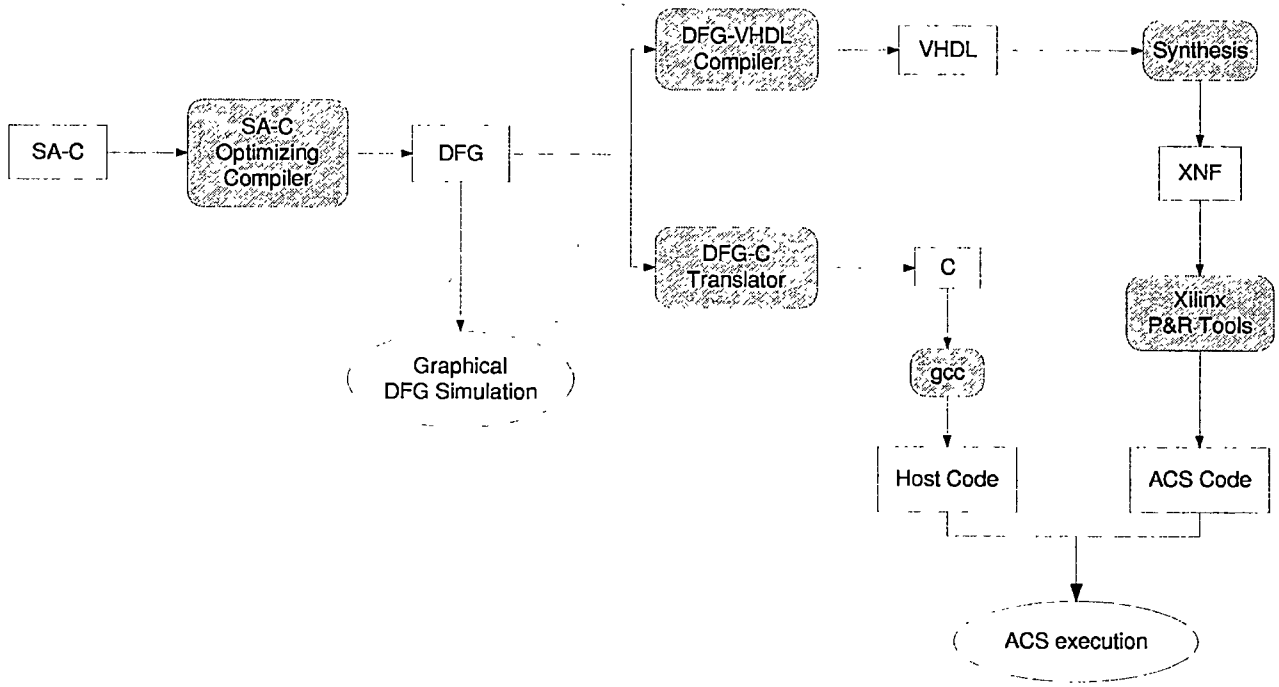


Figure 2.7: Design flow of CAMERON project.

The sequential code not inside a loop is assigned to the host code. Using these partition criteria, the main computational circuit that will be implemented in the FPGA is entirely combinational. As a result, pipelining cannot be used in the main computational circuit [6].

After partitioning, the DFG for the hardware is translated to VHDL. The process of translating the DFG to VHDL is divided into two main parts. The DFG is translated by converting each node into VHDL. For DFG nodes implementing simple operations, such as arithmetic or logical operations, there is a one-to-one correspondence between the nodes and VHDL statements. Therefore these nodes are directly translated into VHDL statements. For more complicated operations, such as array sum, the translator generates a connection to a predefined VHDL component. A library of such components allows the SA-C compiler to directly access hardware implementations for many complex operations.

The translated VHDL code is then processed by a commercial VHDL compiler and place and route tools. These tools produce the final FPGA configuration files that can be downloaded onto the ACS and executed. The DFG for the host is translated into C code and compiled using a C compiler into the host program. The host program is used to control the downloading of the FPGA configuration, the process of sending data to the ACS and the results reading process.

Comparison Between CHAMPION and CAMERON

Our research shares a "system-level" perspective with the CAMERON project. Both projects use Cantata as the entry point for application development and ACS as the hardware for application implementation. However, our research extends to support ASIC as an alternative hardware implementation since it provides some advantages over ACS.

CAMERON is more strongly oriented towards compilation-based approaches for implementing applications than our work. Most efforts are in the development of the SA-C language to support the programming for image processing and reconfigurable hardware, and the development of a SA-C compiler to optimize the code. The SA-C language is a subset of C with restrictions such as single assignment and the elimination of recursion and pointers manipulation. It is a language well suited for image processing application.

In our research work, we allow the user to use both fixed point C and C++ during glyph development. The fixed-point C and C++ are not restricted to image processing domain. They are well suited for more application domains. In addition, C++ also provides an object-oriented feature for programming. The only restriction that we imposed on C and C++ is the use of fixed-point number as opposed to floating point number. The only disadvantage of using fixed-point number is that a fixed-point number requires more data bits than a floating number to achieve the same degree of accuracy.

In the CAMERON project, the compiler exploits both coarse-grain and fine-

grain parallelism. The compiler identifies loop-level and instruction-level parallelism to increase the throughput of the application implemented in the ACS. However, the main computational circuit implemented in the FPGA is entirely combinational. This restricts the pipelining in the main computational circuit. Our research work improves the hardware throughput by mapping the Cantata application to sequential logic circuit instead of combinational circuit. Therefore, pipelining of the input data is possible in the entire circuit.

Perhaps the main difference between our research work and CAMERON is the time required to map a Cantata application to ACS. In our research, we perform synthesis and place/route on our library cells in advance. Thus, we have accurate information on the size and delay of each cell and only have to re-synthesize small net-lists that represent the collection of cells that fit in each FPGA. The CAMERON approach merges the VHDL code into a single, large file that must be fully re-synthesized and then partitioned at a finer grain than our approach. Hence, CHAMPION is presented with a smaller netlist and can be expected to execute in less time. Since the circuits produced by CHAMPION allow pipelining, the performance of the circuits is expected to be compatible to that of CAMERON, even though no optimization is performed on the circuit.

2.5.2 MATCH Project: Northwestern University

Project Overview

The objective of the MATCH (MATlab Compiler for Heterogeneous computing systems) [5] project at Northwestern University is to make it easier for users to develop efficient codes for heterogeneous computing systems. They are implementing and evaluating an experimental prototype of a software system that will take MATLAB descriptions of various embedded systems applications, and automatically map them onto a heterogeneous computing environment consisting FPGA arrays, embedded processors and DSP processors.

The prototype heterogeneous hardware system (shown in Figure 2.8) designed to work in the MATCH project consists of a controller (microprocessor running Solaris Operating system) and three types of computing resources: an ACS board, embedded processors, and DSP processors. A VME bus is used as the communication backbone for these computing resources.

This heterogeneous system combines the advantages of microprocessor based embedded systems, DSP processors and FPGA resources. The microprocessors and DSP processors are used to support the computations, which are not ideally suited for the FPGAs and the bulk of the functionality required to implement an algorithm. Example of functions implemented by these processors are control intensive algorithms, complex floating-point applications and computing tasks with large amount of code that is rarely executed. The reconfigurable logic is used to accelerate only the most critical computation kernels of the program.

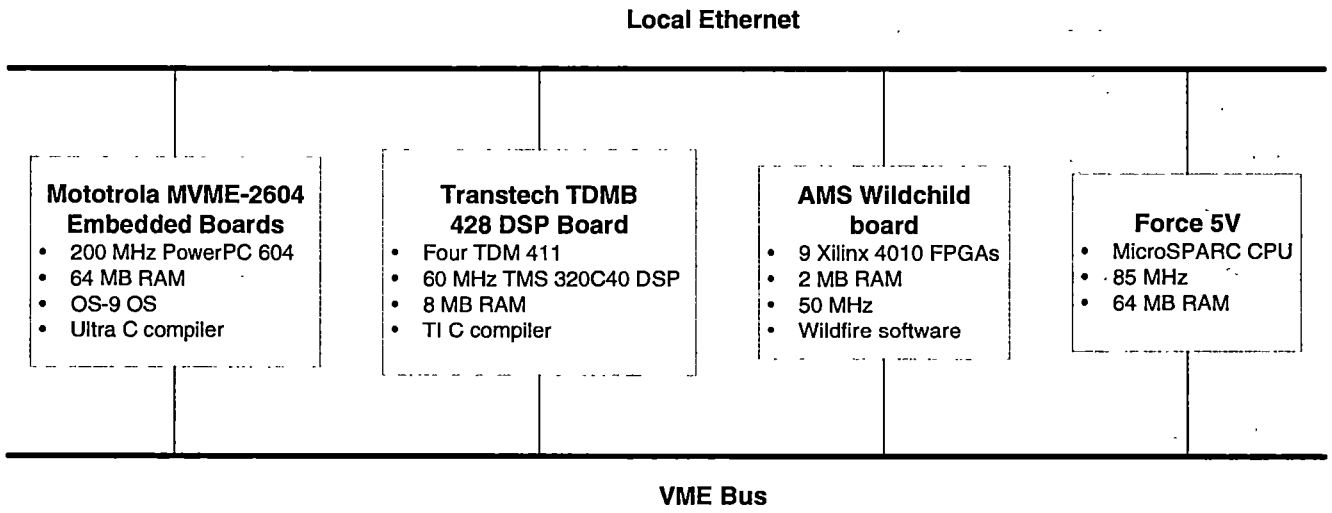


Figure 2.8: The heterogeneous hardware system used in the MATCH project [5].

The key issue that needs to be addressed in the MATCH project is how to map a MATLAB program on such a heterogeneous architecture without expecting the application programmer to get into the low level details of the architecture. A compiler is being developed to generate efficient code automatically for the heterogeneous target. An overview of the compiler is shown in Figure 2.9. The compiler parses the input MATLAB program based on a formal grammar and builds an abstract syntax tree (AST) [5]. The AST is then partitioned among the various components based on the set of predefined library functions. The nodes corresponding to the predefined library functions are assigned to the respective targets.

The ASTs for the FPGAs are then converted to register transfer level VHDL. Each user function is translated into a process in VHDL. Each scalar variable in MATLAB is translated into a variable in VHDL. Each array variable in MATLAB is stored in a RAM adjacent to the FPGA. The read or write functions of the memory corresponding the array variable are then generated. Control statements such as IF-THEN-ELSE constructs in MATLAB are translated into corresponding IF-THEN-ELSE constructs in VHDL. Assignment statements in MATLAB are translated into variable assignment statements in VHDL. Loop control statements are translated into a finite state machine. Once the translation is completed, logic synthesis and place and route tools will then be used to generate the FPGA binaries. The ASTs corresponding to the ACS host, embedded microprocessors and DSP processors are translated into equivalent C code.

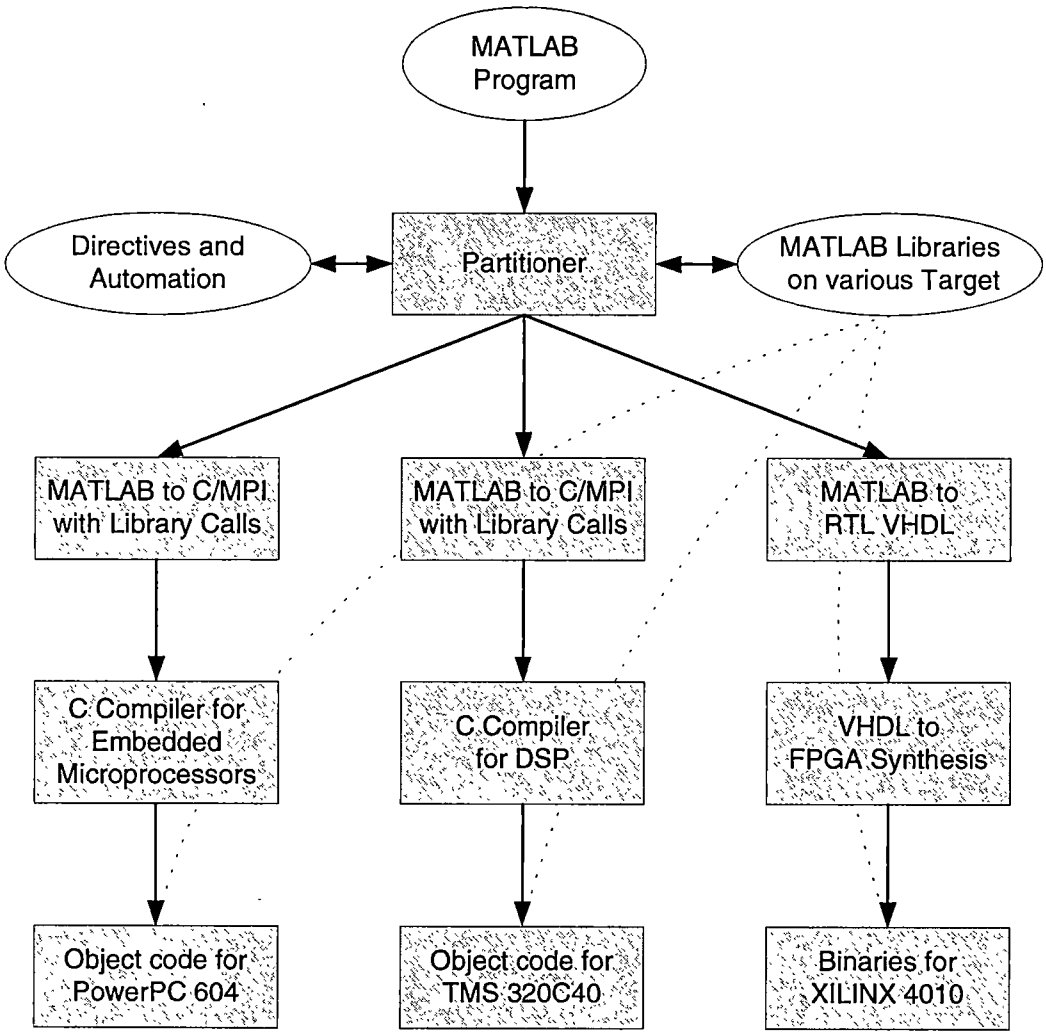


Figure 2.9: Match compiler [5].

Finally, these generated codes are compiled using the respective target compilers to generate the object codes for the processors. A main thread of control is automatically generated for the controller. The controller is used to make remote procedure calls to the microprocessors, DSP processors, and ACS.

Comparison Between CHAMPION and MATCH

The MATCH project uses MATLAB for application development. Similar to Cantata, MATLAB is a function-oriented language where most of the programs can be written using predefined functions. However, MATLAB programs are normally implemented as sequential codes since the language is designed to be executed on a single conventional processor. To take advantage of the hardware implementation, the parallelism within the MATLAB code must be exploited.

For graphical programming such as Cantata, application can easily be implemented in a parallel manner since the application is created in a form of data flow graph. The user can increase the "amount" of parallelism by interconnecting the glyphs in parallel. This will make the application highly parallel at the function-block level. The only remaining sequential part is the instruction-level code within the glyph. This instruction-level parallelism can be exploited similar to what is being done in the CAMERON project. However, this will require a more restricted language, such as SA-C, to enable the compiler to analyze the data dependency and therefore, extract the parallelism in the code.

The MATCH project uses a more complicated hardware architecture for the

mapping of application. A heterogeneous system consisting of FPGAs, microprocessors and DSP processors is used. The MATCH compiler is required to partition the MATLAB code to based on the type of computations. For example, control intensive algorithms, complex floating-point applications are not ideally suited for the FPGAs and should be implemented in the microprocessors or DSP processors. On the other hand, critical kernels which require fast computations should be implemented in FPGAs.

Similar to the CAMERON project, the main difference between our research work and MATCH will be the time required to map an application to hardware. In our research, we perform synthesis, placement and routing on our library cells in advance. Thus, we only need to re-synthesize small netlists that represent the interconnections of high level cells. The MATCH approach, however, translates the MATLAB code to VHDL code. The VHDL code must then be fully resynthesized and then partitioned at a finer grain than our approach. Hence, CHAMPION is presented with a smaller netlist and can be expected to execute in less time.

2.5.3 Commercial Software

There have been several major commercial efforts to automatically map high-level applications to hardware. The COSSAP tool from Synopsys allows application to be developed using a block diagram graphical language. The developed application is then translated into VHDL or Verilog and synthesized into hardware.

The Signal Processing Workbench (SPW) from Cadence also takes an application developed using a block diagram language and translates it to VHDL. The Renoir tool from Mentor Graphics Corporation allows an application to be developed in different graphical ways. The users can enter the design as either block diagrams, flow charts or state diagrams. The Renoir tool will then generates the corresponding behavioral VHDL or Verilog automatically. The Monet tool can then be used to convert the behavioral VHDL into register transfer level VHDL. The register transfer level VHDL can be synthesized using the Leonardo logic synthesis tool.

The CHAMPION software differs from all of the commercial tools above in that we perform synthesis, placement and routing on our library cells in advance. Thus, we only have to re-synthesize small netlists that represent the interconnections of high level cells and can be expected to execute in less time.

CHAPTER 3

Methodology

In order to automatically map a Cantata application onto ACS and ASIC, all the steps shown in the ACS and ASIC design flows (Figure 2.3 and 2.4 respectively) have to be automated. In addition, the different types of operations such as data transfer, synchronization and sizing that are handled differently in Cantata and in hardware have to be resolved. This chapter presents the algorithms, approaches, and mathematical techniques used in CHAMPION to automate the design flows and to handle the hardware and software differences.

3.1 Overview of the Design Flow of CHAMPION

The design flow of CHAMPION is illustrated in Figure 3.1. The entire design flow can be divided into four sub-flows (as shown in Figure 3.2):

- Glyph development flow
- Front-end flow
- ACS back-end flow
- ASIC back-end flow

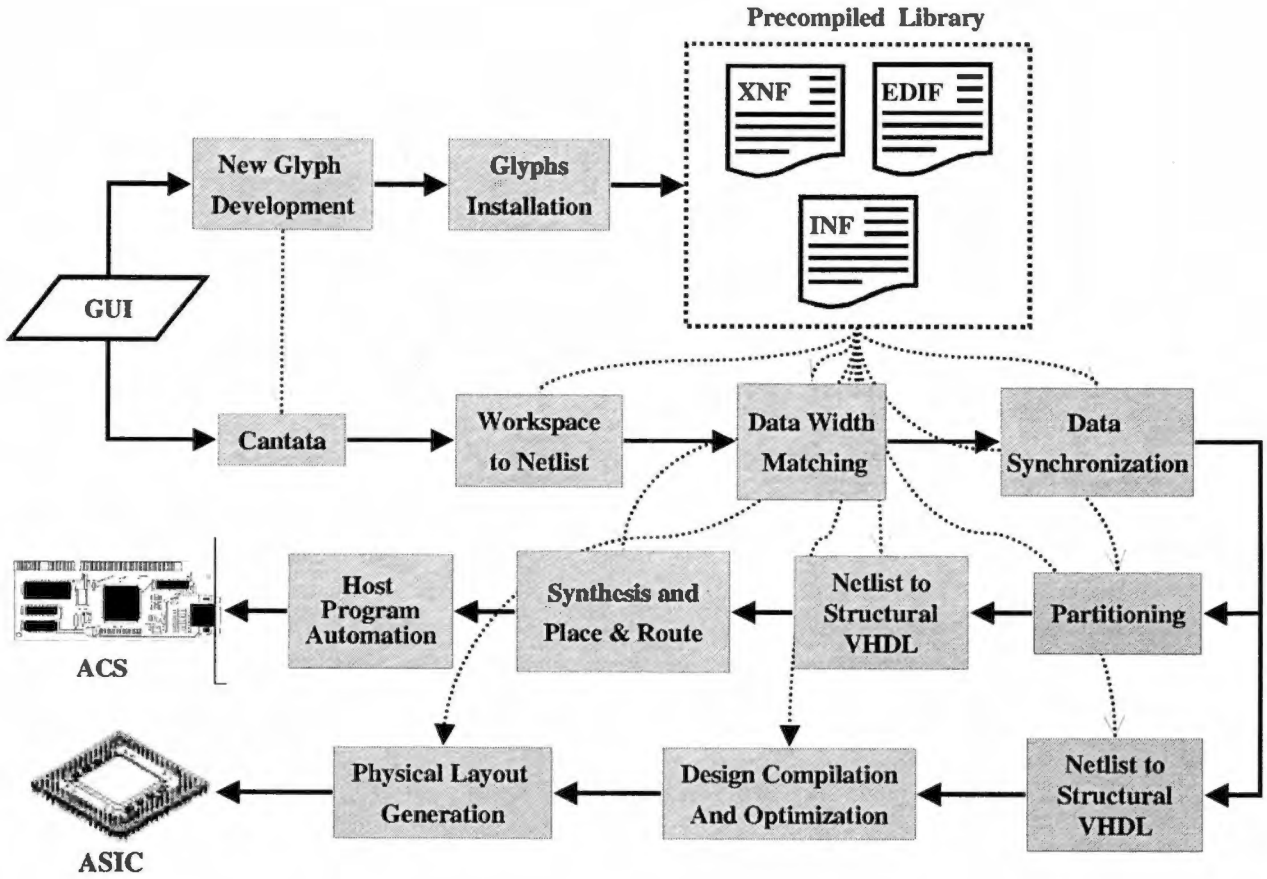


Figure 3.1: Design flow of CHAMPION.

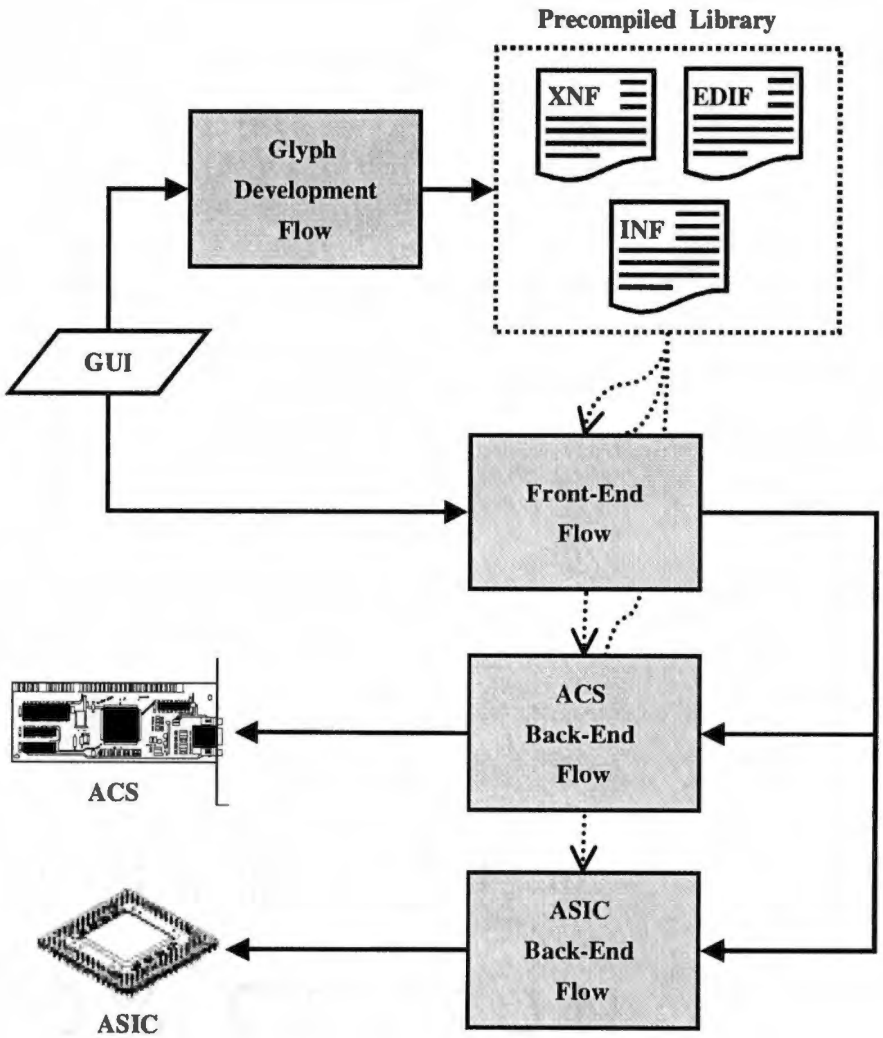


Figure 3.2: Four main flows in CHAMPION.

In CHAMPION, Cantata is used as a function-oriented programming environment where all the application programs are developed using predefined functions or modules called glyphs. Currently, a set of library glyphs has been developed in the CHAMPION project. New library glyphs can be developed and added to the existing precompiled library if needed. A set of tools has been developed to automate the process of developing, verifying and installing the new glyphs in the CHAMPION library. This set of tools constitutes the glyph development flow.

Once the designer has all the required glyphs, the application can be generated by interconnecting these glyphs. The front-end flow is then used to transform the Cantata program into more of a hardware-like netlist. The Cantata program is first translated into a more graph-oriented database, preserving the original glyphs and their interconnections. Each interconnection is then checked to verify that the bit-widths of the connecting ports are the same. After matching the width of all the interconnections, data synchronization is performed. In CHAMPION, data synchronization is achieved by introducing delay buffers into the system. The synchronization software determines the lengths and locations of the delay buffers necessary to balance the various data paths. An optimization algorithm is employed to calculate a set of buffer lengths and insertion points that maximizes the amount of buffer sharing which therefore, minimizes the total number of delay buffers.

If the design is being mapped to ACS, the ACS back-end flow will be used. The ACS back-end flow first partitions the synchronized netlist at the glyph-level.

After partitioning, the internal data structure or format is translated into a structural VHDL representation. The required I/O ports for each of the sub-netlists are then added to the VHDL files. Next, the VHDL files are synthesized and merged with the pre-compiled VHDL components corresponding to the Cantata glyphs. Each sub-netlist is then placed and routed. A host program is generated to download the resulting configuration files to the corresponding programmable logic component on the ACS board.

If the design is being mapped to ASIC, the ASIC back-end flow will be used. No partitioning is required for the ASIC implementation. The synchronized netlist is translated into a VHDL representation. Next, the VHDL representation of the circuit is synthesized to a target technology selected by the user. A layout tool is then used to automatically generate the final layout of the ASIC based on the synthesized netlist.

3.2 Glyph Development Flow

Application programs can be constructed by interconnecting the glyphs contained in the CHAMPION library using Cantata. If certain glyphs needed for the application cannot be found in the CHAMPION library, the user can go through the glyph development flow to generate, verify and install these glyphs to the library.

3.2.1 Glyph Development and Verification

To incorporate a new glyph into the CHAMPION library, the designer must first develop the fixed-point C/C++ program for the glyph. The reason for using fixed-point arithmetic is to allow the C/C++ program to mimic hardware operations. For complex functions, the C/C++ program can be formed as a macro of lower-level functions.

For each of the C/C++ program developed, a corresponding VHDL code must be developed. The functionality of the VHDL code must be identical to that of the C/C++ program. A set of test vectors is used to simulate both the C/C++ and VHDL code. The simulation results are compared to verify that bitwise identical behavior is achieved. The steps for developing the new glyph are shown in Figure 3.3.

For the VHDL description, the functionality can be verified by:

- simulating the VHDL code using the commercial software such as Model Technology's ModelSim, or
- executing the synthesized VHDL code in the ACS.

Both the VHDL verification methods are shown in Figure 3.4.

To accelerate the glyph development process, the commercial software, A|RT Library and Builder [11] from the Frontier Design was integrated to the glyph development flow. The A|RT Library and Builder provide the ability to generate the VHDL description of the hardware directly from a C-code specification. The

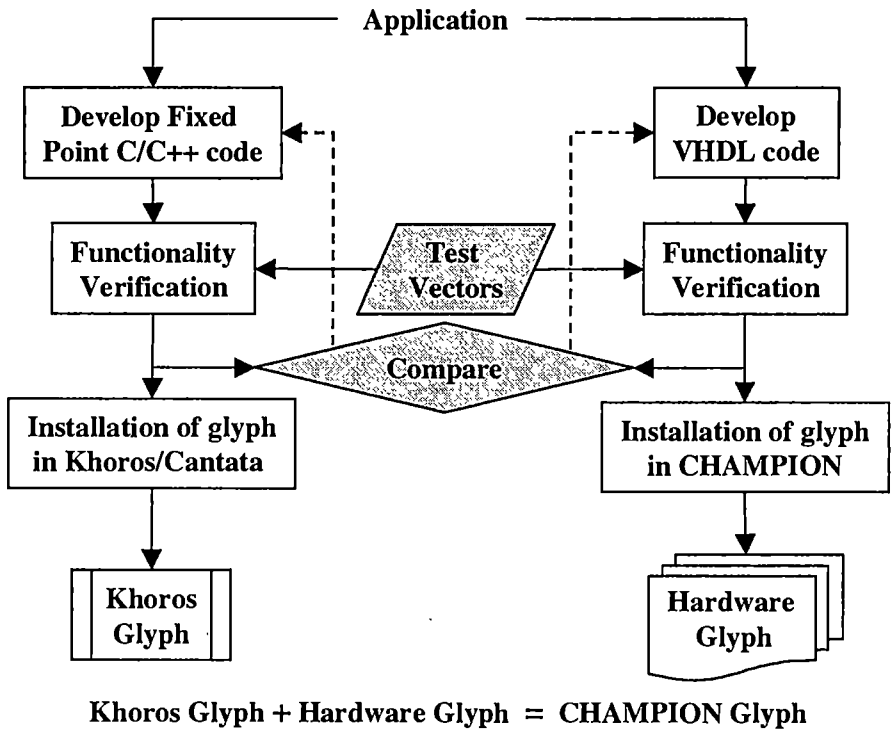


Figure 3.3: Steps for developing a new glyph.

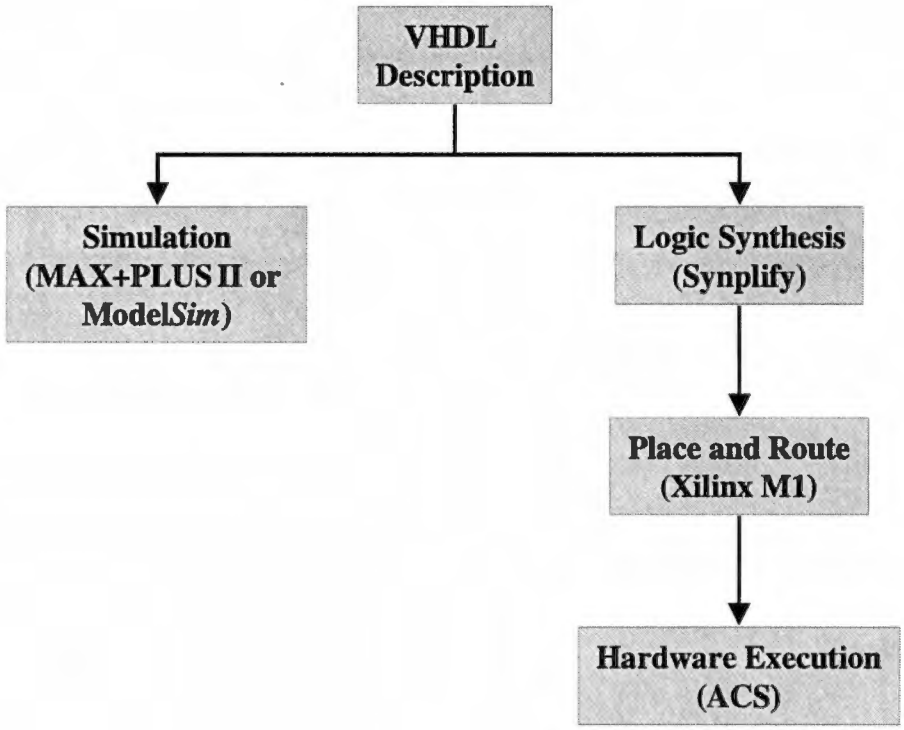


Figure 3.4: Glyph verification methods.

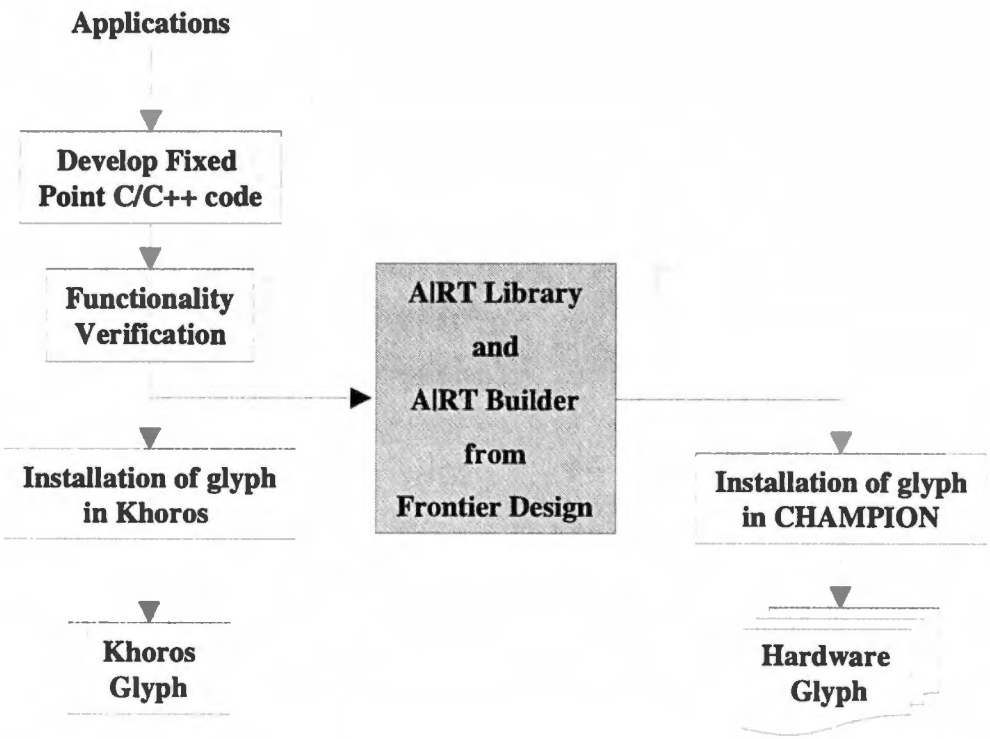


Figure 3.5: New glyph development using ART Library and Builder.

A|RT Library facilitates the development of fixed-point algorithms required for a hardware implementation. The A|RT Builder can then be used to directly and automatically convert the fixed-point C applications to VHDL. Therefore, the A|RT product line is able to increase the design productivity enormously. The user no longer has to go through the process of generating, simulating and verifying the VHDL code. The new steps for developing the glyph using A|RT Library and Builder are shown in Figure 3.5.

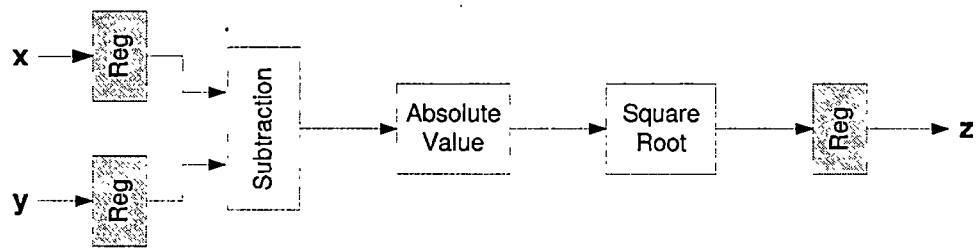
3.2.2 Glyph Installation

Once the functionalities are verified, the C/C++ program is converted to a Khoros glyph and installed in Cantata (as shown in Figure 3.5). The corresponding VHDL description is synthesized and converted to a hardware glyph. The hardware glyph is installed in the CHAMPION library. Together, the Khoros and hardware glyph constitutes a CHAMPION glyph.

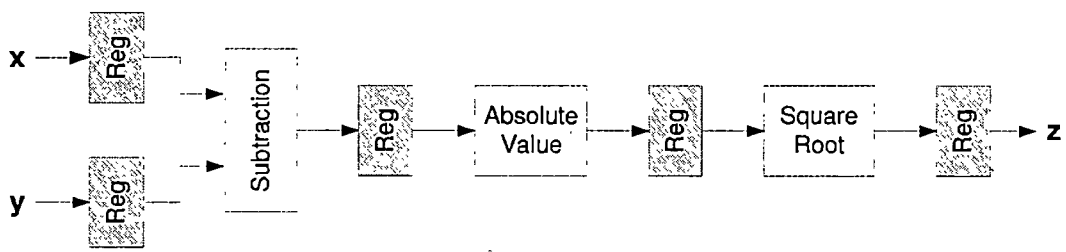
To install the C/C++ program in Cantata, the Khoros tools, Craftsman and Composer can be used. To install VHDL code into the CHAMPION library, a software tool was developed to automate the synthesis of the VHDL code using the commercial logic synthesis tool. Based on the ACS of ASIC architecture specified by the designer, the tool will generate the required technology-dependent netlist file (XNF or EDIF) and a glyph information file (INF file) for storing the size, latency and I/O data bit-widths of the hardware glyph. This information will be used during the data width matching, data synchronization and partitioning processes.

3.2.3 Pipelined Glyphs

Pipelining can greatly accelerate the operation of a circuit. The idea of pipelining can be easily explained using the following example. The circuit in Figure 3.6(a) is used to compute the square root of $\mid x - y \mid$. If this circuit is used to perform computation on a large set of input values, then x and y will represent two streams of numbers. The minimal clock period, T_{min}, required to



(a)



(b)

Figure 3.6: Datapath structures. (a) Nonpipelined structure and (b) pipelined structure.

ensure that valid output can be obtained is given by:

$$T_{min} = t_{reg} + \sum t_{gate} \tag{3.1}$$

where t_{reg} and t_{gate} are the propagation delays of the register and gates respectively. For the sake of simplicity, the registers are assumed to be ideal. That is, the setup and hold time for the registers are assumed to be zero. Assume that the propagation delay for the register, *subtract* gate and *abs* gate are 10 ns and the propagation delay for the *sqrt* gate is 30ns. Using Equation 3.1, the minimal clock period is found to be 70 ns. Therefore the maximum clock frequency for the circuit in Figure 3.6(a) can operate at is approximately 14.2 MHz.

If registers are added between the logic gates, as shown in Figure 3.6(b), the combinational circuit block will be partitioned into three sections. Each partition has a smaller propagation delay compared to the original circuit. This effectively reduces the value of the minimal clock period to:

$$T_{min} = t_{reg} + max(t_{gate}) \tag{3.2}$$

Since the maximum gate delay is equal to 30 ns, the minimal clock period is equal to 40 ns. Therefore, the maximum clock frequency for the circuit in Figure 3.6(b) is approximately 33.3 MHz. The pipelined circuit outperforms the original circuit by more than a factor of 2.

In addition to the improvement in clock frequency, the pipelining circuit also allow the computation of input data to spread over a number of clock periods, as shown in Table 3.1. The result for the first data set, (x_1, y_l), appears at the

Table 3.1: Pipelined computations.

Clock Period	Subtract	Absolute Value	Square Root
1	$x_1 + y_1$		
2	$x_2 + y_2$	$x_1 + y_1$	
3	$x_3 + y_3$	$x_2 + y_2$	$x_1 + y_1$
4	$x_4 + y_4$	$x_3 + y_3$	$x_2 + y_2$
5	$x_5 + y_5$	$x_4 + y_4$	$x_3 + y_3$

output after three clock-periods. At that time, the second data set, (x_2, y_2), has already gone through the subtraction and absolute value gates, and the third data sets, (x_3, y_3), has already gone through the subtraction gate. This assembly line type of computation can greatly improve the throughput of the circuit.

In CHAMPION, all the hardware glyphs are synchronous. Each CHAMPION glyph consists of combinational logic with registers to hold the input data as shown in Figure 3.7. As a result, the circuit produced by combining these glyphs is always a pipelined system.

3.2.4 Control Lines in CHAMPION Glyphs

In hardware implementation, data is transferred between hardware glyphs one value at a time rather than an entire block of data, as in Cantata. This difference in data transfer must be accounted for when mapping a Cantata application to hardware. The data transfer difference will not affect the operation of a glyph

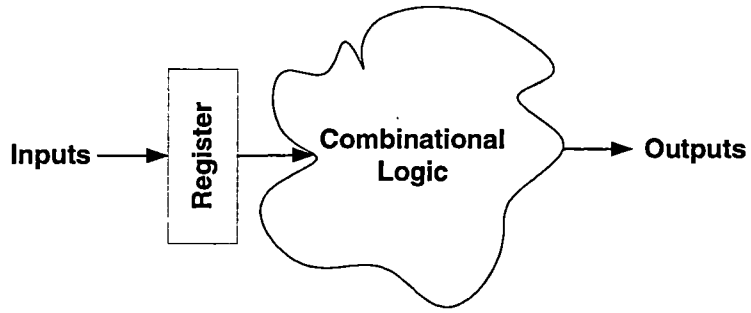


Figure 3.7: Structure of CHAMPION glyph.

if the Cantata glyph operates on the data one value at a time. For instance, a Cantata glyph that adds a constant to each value in a stream only needs to work with one value at a time. The corresponding hardware glyph can work with only one data value at a time as well and the mapping of it will be simple.

If a Cantata glyph performs some accumulation operations in a stream before it produces a valid output, it will not map easily to hardware. An example of such glyph is an accumulator that finds the sum of all the pixels in an image. To map this type of glyph to hardware, some control lines must be included in the glyph. Three control lines, stream valid, data valid and pixel valid, are included in every CHAMPION glyph. The functions of these control lines are as follows:

- Stream Valid Control Line: indicates the presence of an image stream to a hardware glyph.

- Pixel Valid Control line: indicates whether or not a particular data value within a stream represents an actual pixel in the image since the individual

pixels in a data stream may become separated from one another when it passed through certain hardware elements.

- Data Valid Control Line: indicates whether or not a pixel value has become corrupted through processes such as division by zero or the resulting border pixels from convolution

These control lines help guiding the hardware glyphs to synchronize and control operations on data streams. More details of these control lines can be found in [21].

3.3 Front-end Flow

Using Cantata, the designer can develop the application by interconnecting CHAMPION glyphs to form the Cantata workspace. Simulation, data analysis and visualization can be performed in Cantata. Once the desired functionality of the application is achieved, CHAMPION front-end converts the graphical program to a synchronized netlist with matching net width.

3.3.1 Converting Cantata Workspace to CHAMPION Netlist

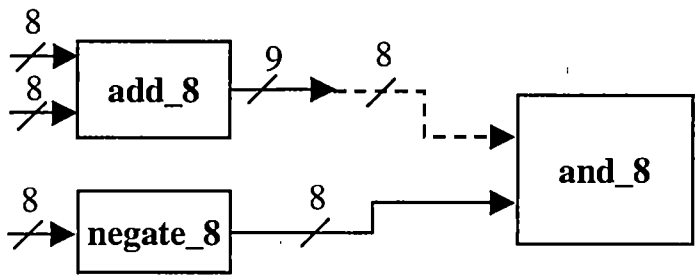
The first step in the front-end flow consists of translating the Cantata workspace into a more graph-oriented netlist format. The netlist format is a directed hypergraph where each glyph is represented as a node and the interconnections between glyphs are represented as directed hyperarcs. Based on the information

of the glyph, weights are assigned to the nodes and hyperarcs of the directed graph. The weights of the nodes correspond to the size in terms of the number of logic blocks, and the weights of the hyperarcs correspond to the net-width of the glyph interconnections. This netlist format simplifies the use of graph theories and network optimization theories during the data synchronization and partitioning process.

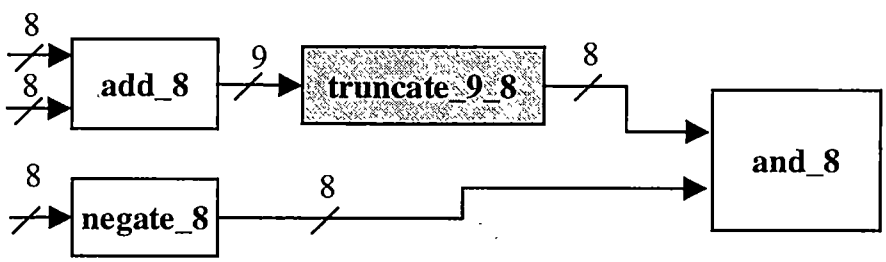
3.3.2 Data Width Matching

Since Cantata uses the polymorphic data model, the data size conversion is handled automatically by the Cantata glyphs. The users do not have to worry about the difference in sizes for the input and outputs of the Cantata glyphs. However, the inputs and outputs of each hardware glyph corresponding to the Cantata glyph have fixed data width. If an input and an output of different sizes are connected together, one of them must be converted to match the other.

Two types of data size mismatch can be found. In a hardware application, some functions may produce results that require fewer bits for their outputs than for their inputs. Consequently, glyphs cascaded into one another will progressively require a narrower data path. When one path of operations is connected to a parallel path, a mismatch in the number of bits for these inputs may occur. This mismatch is called a positive mismatch since the bit width of the net carrying the data is larger than the bit width of the net receiving the data. An example of a positively mismatched data path is shown in Figure 3.8(a). A software tool



(a)



(b)

Figure 3.8: Positive mismatch. (a) Example of a positively mismatched data path and (b) insertion of "truncating" glyph in positively mismatch data path.

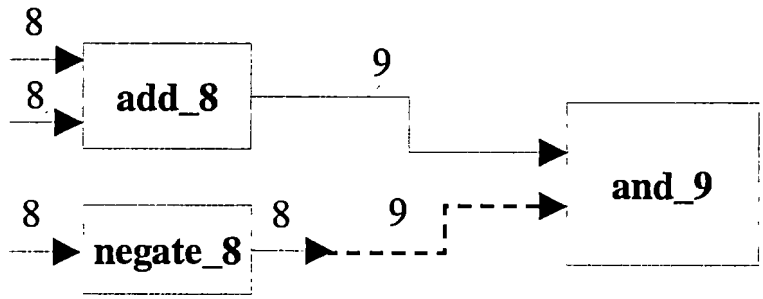
has to be developed to analyze each data path and truncates the additional bits when appropriate. The truncating process (shown in Figure 3.8(b)) is performed by inserting a "truncating" glyph at the mismatch data path. The "truncating" glyph will remove the additional data bits from the signal.

Similarly, some hardware functions may produce results that require more bits for their outputs than for their inputs. Glyphs cascaded into one another will progressively require a wider data path to avoid round off errors. Consequently, a negatively mismatched data path such as the one shown in Figure 3.9(a) may occur. In this case, a "padding" glyph (shown in Figure 3.9(b)) has to be inserted at the mismatched data path.

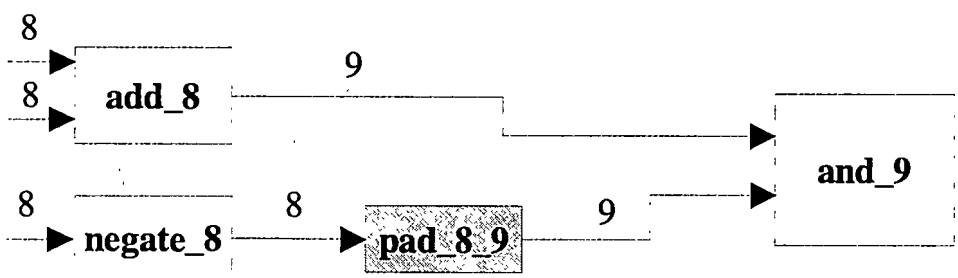
3.3.3 Data Synchronization

In the graphical programming environment such as Cantata, executions of the programs are data driven. That is, each Cantata glyph will begin execution only when all its input data is available. However, hardware systems are clock driven. At each clock cycle each hardware glyph will process whatever data is presented at its inputs.

Due to the difference in the processing time of each hardware glyph, data traveling over different concurrent paths may arrive at the inputs of a multi-input glyph at different times. To insure that each glyph generates the correct time-sequenced output, it is necessary that each glyph receive all its input data at precisely the same time. This requirement is often referred to as data synchro-



(a)



(b)

Figure 3.9: Negative mismatch. (a) An example of a negatively mismatched data path and (b) Insertion of "padding" glyph in positively mismatch data path.

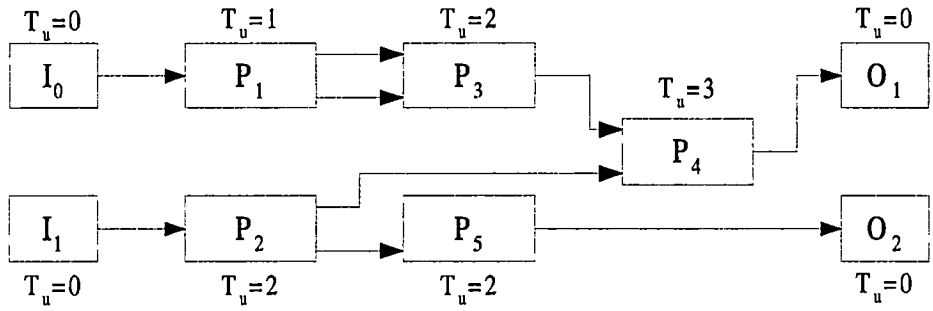


Figure 3.10: An unsynchronized digital system.

nization.

To illustrate the data synchronization problem, the digital system shown in Figure 3.10 will be used. In this system, there are two primary input modules, two primary output modules and five processing modules labeled P_1 through P_5. The processing time for each processing modules are labeled T_u. Note from the figure that the primary input and output modules have zero processing time. This is due to the fact that these modules are storage devices for the input and output data. No data processing will be performed in these modules.

We assume that all the input data are readily available in the input modules. Therefore, all input modules are time synchronized. We also require that all the outputs are made available at the same time. This is because if another digital system is connected to this system, these output modules will become the input modules for the new digital system. In addition, this requirement also allows us to break a large digital system into smaller subsystems for synchronization.

Examining this system, it is clear that the two input signals entering P_4 are

not synchronized; the lower signal arrives one time unit sooner than the upper signal. Also, the primary output, O_2, becomes available two-unit times earlier than O_1.

There are two main approaches to synchronize the various signals impinging on each processing module. The traditional approach is to use a control circuit and edge-triggered registers to synchronize the data. The registers are inserted at the input of each processing module (as shown in Figure 3.11). They are used to feed and hold the inputs to the modules. When all the input data have arrived at the register, the register is triggered by the control signal generated by the control circuit. The control circuit generates the signal only when all the signals have arrived at the inputs of the processing module. Therefore, the control circuit is designed based on the delay information of all the processing modules in the digital system.

When the register is triggered, it will feed all the input data to the processing module simultaneously. The output data of that particular module is then kept unchanged by holding the input data using the register. The input signal has to be held until the last module has finished processing the data. As a result, the digital system using this data synchronization approach has a pipelined time, which is equal to the processing time of the critical path (the data path having the highest processing time). Each time a new input signal is presented to the digital system, the signal has to be held unchanged for the length of the pipelined time before another new signal can be presented to the system.

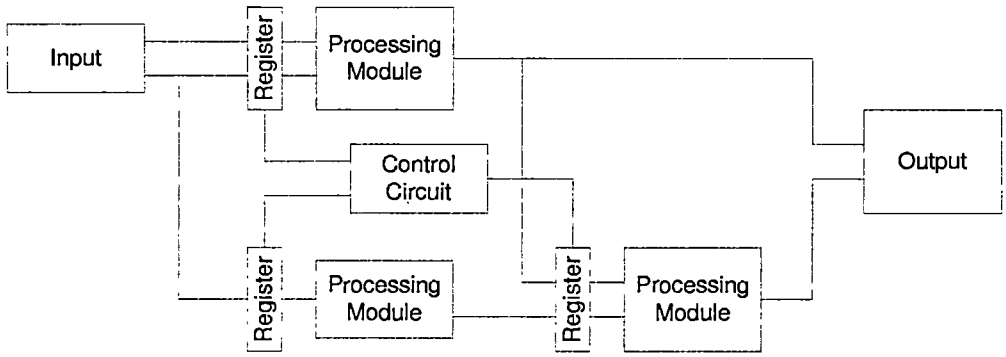


Figure 3.11: A digital system which uses clock-triggered registers to synchronize the data.

The high pipelined time of the synchronization approach using edge-triggered registers results in low throughput rate. To increase the throughput rate, data synchronization can be achieved by introducing delay buffers into the digital system. These delay buffers, are inserted at various locations to delay the signal on the data path which has lower processing time, and therefore, match it with the data path with higher processing time. For this synchronization approach, the pipelined time is zero. That is, the input signal can be presented to the system continuously without having to hold each new input signal, as in the case of the synchronization approach using edge-triggered registers. This feature is essential in many pipelined designs, which require high throughput rates.

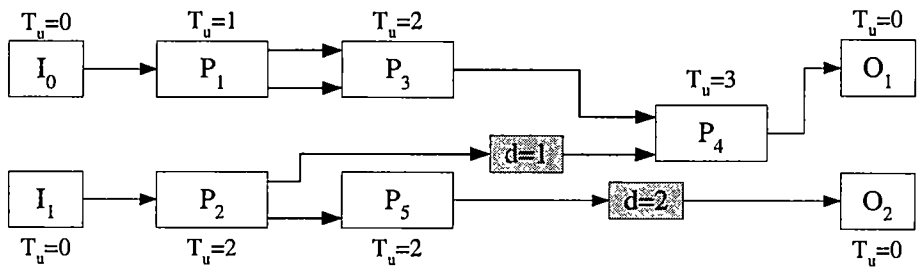
For the delay buffer insertion approach, synchronizing the system requires one to determine the lengths and locations of the delay buffers necessary to balance the various data paths. Straightforward methods for performing this task are not difficult to develop. For example, each multi-input processing module might be

examined to check if all the inputs have the same delay from the primary input modules. If all the inputs do not have the same delay, the input with the largest delay is identified. For those inputs with delays that are less than this maximum value, delay buffers must be inserted into each of these "early" input lines in order for all the inputs to the module to have equal processing delay from the primary input modules.

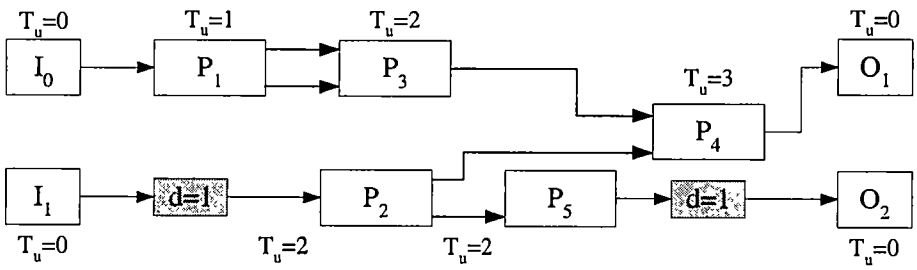
Applying this method to Figure 3.10, we will find that two delay buffers with a total of three unit time delays are required to synchronize the system (as shown in Figure 3.12(a)). One buffer with a unit time delay has to be inserted in the path between P_2 and P_4. Another buffer with a two-unit time delay has to be placed between P_5 and O_1.

These straightforward methods, however, do not necessarily provide the optimum solution in the sense that the total number of delay buffer units used is not necessarily a minimum. The delay buffers can always be moved forward or backward along the data path in order to achieve a maximal amount of delay buffer sharing. This can be seen in Figure 3.12 where inserting the delay buffer between I_1 and P_2 allows both I_1-P_4 and I_1-O_2 paths to share the delay. Therefore, a total of two unit time delays are required, compared to three unit time delays required in Figure 3.12(b).

Some optimization algorithms can be employed to calculate a set of buffer lengths and insertion points that maximizes the amount of buffer sharing and therefore, minimizing total length of the delay buffers required. In CHAMPION,



(a)



(b)

Figure 3.12: Synchronization approaches. (a) The result of synchronizing the system using the simple approach and (b) an optimum synchronization.

the algorithm developed by Hu [9] is adopted.

Research Work in Delay Buffer Minimization Problem

Many algorithms have been proposed in numerous publications to solve the problem of inserting and/or minimizing the number of delay buffers. In [15], Lee and Chang have proposed a systematic way of solving the buffer optimization problem based on integer linear programming (ILP) theory. Since no polynomial-time algorithm is available for general integer linear programming problems [24], a pseudo-polynomial-time algorithm is suggested by Lee and Chang. However, using a pseudo-polynomial-time algorithm, the computation time needed to solve a buffer minimization problem of a large network can still be excessive. Lee and Chang overcame this difficulty by using a path decomposition technique to partition large networks into smaller sub-networks. However, this decomposition method cannot guarantee that a given network is partitionable. In fact, non-partitionable networks can easily be found [9].

In [16], the authors assume that a given network has already been synchronized. That is, straightforward synchronization method is first used to insert the appropriate delay so that each module receives all its input data precisely at the same time. The task is then to rearrange these buffers to minimize the number of delay buffers in the already-synchronized pipelined system. The authors show that the buffer minimization problem for the already-synchronized system is the linear programming (LP) dual of a minimum-cost flow problem, which can

be solved in polynomial time. Thus, the entire synchronization process requires two separate steps: synchronization followed by minimization. The synchronization step requires a processing time which is proportional to the number of interconnections contained in the network, while the minimization step can be accomplished in polynomial time. As a result, the two-step approach proposed in [16] has an overall polynomial time complexity.

In [9], X. Hu et al. show that the synchronization and buffer minimization can be combined into a single step. That is, a single formulation can be used to solve the problem of minimizing the total buffer length for the synchronization of a system. The authors illustrate that the problem of synchronizing a pipelined system with minimum buffer stages is also the LP dual of a minimum-cost network-flow problem. Either polynomial time algorithms such as the ones in [23], and [14] or the simplex technique [19] can be used to solve the buffer minimization problem. A globally optimal solution is always guaranteed using the formulation proposed by [9]. In addition, the original formulation can also be extended to handle systems with hyperarcs and feedback loops.

Due to the advantages offered by the algorithm proposed by X. Hu et al., the formulation in [9] was adopted to solve the data synchronization problem. Slight modification to the formulation is required in order to use it for solving the data synchronization problem in the CHAMPION project. In the next few sections, the formulation in [9] is presented.

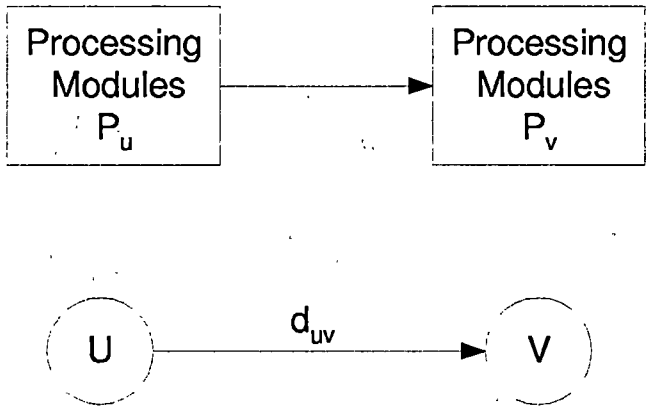


Figure 3.13: A block diagram of processing modules with their corresponding SFG.

Problem Formulation

To solve the buffer minimization problem using the algorithm proposed in [9], the pipelined system has to be represented using a signal flow graph (SFG). In the SFG, each processing module is represented as a node and each data path between the modules is represented as a directed edge. The weight of the edge directed from node u (corresponding to processing module P_u) to node v (corresponding to processing module P_v) is an unknown delay variable d_{uv} (as shown in Figure 3.13). This delay variable d_{uv} corresponds to the delay buffer which has to be inserted between P_u and P_v for synchronization. It will be determined during the synchronization process (solving of the delay buffer minimization problem). The value of d_{uv} computed during the synchronization process equals the size of the delay buffer required to be inserted between node u and node v.

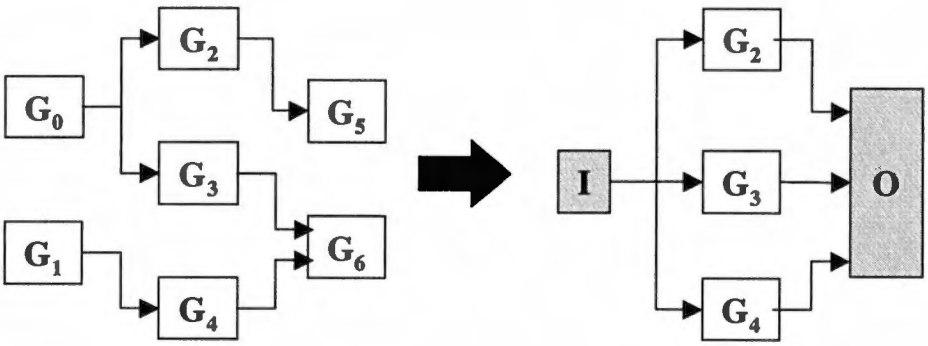


Figure 3.14: Insertion of input and output nodes in the SFG.

Besides the nodes and edges that represent the modules of a system, two virtual nodes are introduced. One is termed the primary input node, i, while the other is called the primary output node, o. All of the input modules of the digital system will be combined and represented using the primary input node, i. Similarly, all the output modules of the digital system will be combined and represented using the primary output node, o. Therefore, in the SFG, all input signals to the system originate at node i, and all outputs from the system terminate at node o (as shown in Figure 3.14). Both these special nodes are assumed to consume zero processing time.

Based on the rules stated above, a SFG representation can be constructed for any given digital system. For instance, for the digital system shown in Figure 3.10, the corresponding SFG is shown in Figure 3.15. The synchronizing problem is then reduced to assigning optimal values to the delay variables in the SFG.

To synchronize a system, all input signals to any given module must arrive

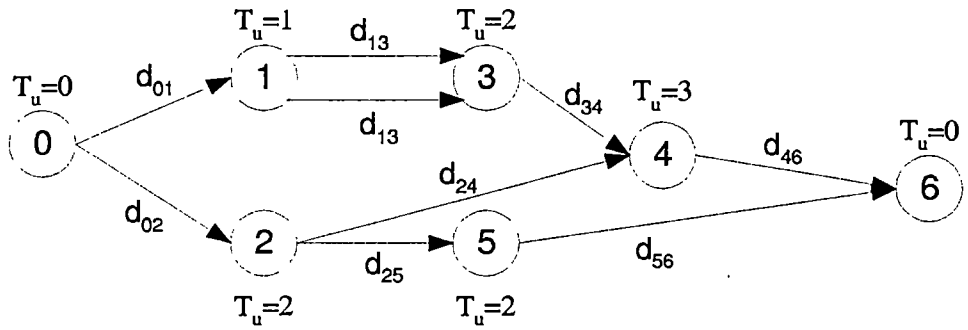


Figure 3.15: The SFG representation of the digital system shown in Figure 3.10.

at the same time. In other words, the accumulated delays along all the distinct paths from the primary input node i to a particular SFG node should be equal. The accumulated delay along a particular data path is simply the sum of all the weights of the edges and the processing time, T, of all the processing modules on the path from i to u in the SFG. Denoting the j^{th} path between node i and u as $P_j(u)$ and the total delay along $P_j(u)$ as $D_j(u)$, it follows that $D_j(u)$ equals the sum of the delays associated with all the edges along $P_j(u)$, i.e., it can be expressed as:

$$D_j(u) = \sum_{(x,y)\in P_j(u)} T_x + d_{xy} \tag{3.3}$$

where (x, y) represents an edge from node x to node y.

The synchronization problem can now be defined as assigning a value to each delay variable, d_{xy}, such that the values of $D_j(u)$ for $j = 1, 2, ...k_u$ are identical for each and every u. Here, k_u denotes the number of distinct paths from the primary input to processing modules P_u. Thus, the task of synchronizing a SFG may be seen to be equivalent to assigning values of D_u to all graph nodes and

values d_{uv} to all graph edges such that

$$D_v - D_u = (T_u + d_{uv}) \tag{3.4}$$

holds for each and every pair of nodes, u and v, that are connected by an edge. The synchronization-minimization problem can then be expressed as an ILP:

$$\begin{aligned} \text{Minimize} \quad & \sum_{(u,v)} d_{uv} \\ \text{Subject to :} \quad & -D_u + D_v - d_{uv} = T_u \\ & u \in V \\ & v \in V \\ & (u,v) \in E \\ & D_u \text{ integer} \\ & d_{uv} \geq 0, \text{ integer} \end{aligned} \tag{3.5}$$

where E and V represent the sets of all edges and nodes in the SFG, respectively, and (u,v) denotes a directed edge from node u to node v. The variables D_u and d_{uv} are determined based on the known processing time delay of each node, T_u. Equations 3.5 form a standard description of an integer linear programming problem. For the SFG shown in Figure 3.15, the integer linear programming

problem can be formulated as follows:

$$
\begin{aligned}
\text{Minimize} \quad & d_{01} + d_{02} + d_{13} + d_{13} + d_{24} + d_{25} + d_{34} + d_{46} + d_{56} \\
\text{Subject to :} \quad & -D_0 + D_1 - d_{01} = 0 \\
& -D_0 + D_2 - d_{02} = 0 \\
& -D_0 + D_2 - d_{13} = 1 \\
& -D_0 + D_2 - d_{24} = 2 \\
& -D_0 + D_2 - d_{25} = 2 \\
& -D_0 + D_2 - d_{34} = 2 \\
& -D_0 + D_2 - d_{46} = 3 \\
& -D_0 + D_2 - d_{56} = 2
\end{aligned}
\tag{3.6}
$$

Solving the Synchronization-Minimization Problem.

Several algorithms can be used to solve the integer linear programming problem. However, none of those algorithms has polynomial time complexity unless sub-optimal solutions are acceptable [9]. As a result, finding an optimum delay buffer set using the general ILP approach may be very time-consuming if the SFG consists of a large number of nodes and edges.

In [9], the author avoids solving the ILP using the following steps:

- Form the dual of the ILP.
- Convert the dual into a minimum cost network flow (MCNF) problem.
- Show that the constraint matrix of the MCNF problem is totally unimodular.

- Show that for MCNF problem with totally unimodular constraint matrix, the integer constraint can be relaxed.

To avoid solving the ILP, the dual of the formulation given in Equations 3.5 is first constructed. Each constraint equality leads to a dual search variable, z_{uv}. To illustrate the process of forming the dual, the ILP expressed by Equation 3.6 will be used. Equation 3.6 can be expressed in a matrix form as shown in Table 3.2. The dual of the ILP problem can then be formulated as shown in Table 3.3.

The dual shown in Table 3.3 can be represented using the following equations:

$$\begin{aligned} &\text{Maximize}: && \sum_{(u,v)} T_u \cdot z_{uv} \\ &\text{Subject to}: && \sum_{(u,w)\in E} z_{uw} - \sum_{(w,v)\in E} z_{wv} = 0 \quad \text{for } w \in V \\ & && -z_{uv} \leq 1 \end{aligned} \tag{3.7}$$

where E and V are the sets of all edges and nodes in the SFG. Note that the constraints in Equation 3.7 can be expressed in a matrix-vector form, and the resulting matrix is the transpose of the constraint matrix in Equation 3.6. This confirms the validity of the dual set given by Equation 3.7.

The ILP given by Equation 3.7 can be transformed into a minimum-cost flow problem by introducing the new variable x_{uv}, defined as $x_{uv} = z_{uv} + 1$. Substituting x_{uv} into Equation 3.7, the following minimum-cost flow problem can be obtained:

$$\begin{aligned} &\text{Minimize}: && \sum_{(u,v)} (-T_u) \cdot x_{uv} \\ &\text{Subject to}: && \sum_{(u,w)\in E} x_{uw} - \sum_{(w,v)\in E} x_{wv} = S_w \text{ for } w \in V \\ & && x_{uv} \geq 0 \end{aligned} \tag{3.8}$$

Table 3.2: Equation 3.6 in matrix form.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	d_{01}	d_{02}	d_{13a}	d_{13b}	d_{24}	d_{25}	d_{34}	d_{46}	d_{56}		
Minimize	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1		
Subject to:	-1	1	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	=	0
	-1	0	1	0	0	0	0	0	-1	0	0	0	0	0	0	0	=	0
	0	-1	0	1	0	0	0	0	0	-1	0	0	0	0	0	0	=	1
	0	-1	0	1	0	0	0	0	0	0	-1	0	0	0	0	0	=	1
	0	0	-1	0	1	0	0	0	0	0	0	-1	0	0	0	0	=	2
	0	0	-1	0	0	1	0	0	0	0	0	0	-1	0	0	0	=	2
	0	0	0	-1	1	0	0	0	0	0	0	0	0	-1	0	0	=	2
	0	0	0	0	-1	0	1	0	0	0	0	0	0	0	-1	0	=	3
	0	0	0	0	0	-1	1	0	0	0	0	0	0	0	0	-1	=	2

Table 3.3: Dual of the ILP expressed by Equation 3.6.

	z_{01}	z_{02}	z_{13a}	z_{13b}	z_{24}	z_{25}	z_{34}	z_{46}	z_{56}		
Maximize	0	0	1	1	2	2	2	3	2		
Subject to	- 1	- 1	0	0	0	0	0	0	0	=	0
	1	0	- 1	- 1	0	0	0	0	0	=	0
	0	1	0	0	- 1	- 1	0	0	0	=	0
	0	0	1	1	0	0	- 1	0	0	=	0
	0	0	0	0	1	0	1	- 1	0	=	0
	0	0	0	0	0	1	0	0	- 1	=	0
	0	0	0	0	0	0	0	1	1	=	0
	- 1	0	0	0	0	0	0	0	0	$\leq$	1
	0	- 1	0	0	0	0	0	0	0	$\leq$	1
	0	0	- 1	0	0	0	0	0	0	$\leq$	1
	0	0	0	- 1	0	0	0	0	0	$\leq$	1
	0	0	0	0	- 1	0	0	0	0	$\leq$	1
	0	0	0	0	0	- 1	0	0	0	$\leq$	1
	0	0	0	0	0	0	- 1	0	0	$\leq$	1
	0	0	0	0	0	0	0	- 1	0	$\leq$	1
	0	0	0	0	0	0	0	0	- 1	$\leq$	1

where S_w represents the difference between the number of edges coming into the node w and the number of edges leaving w. Equation 3.8 describes a minimum-cost flow problem with $-T_u$ as the "cost" of edge (u, v) and S_w as the "supply" to node w. For the dual in Table 3.3, the MCNF problem is given by:

$$\begin{aligned}
\text{Minimize}: \quad & -x_{13a} - x_{13b} - 2x_{24} - 2x_{25} - 2x_{34} - 3x_{46} - 2x_{56} \\
\text{Subject to}: \quad & -x_{01} - x_{02} = -2 \\
& -x_{01} - x_{13a} - x_{13b} = -1 \\
& -x_{02} - x_{24} - x_{25} = -1 \\
& -x_{13a} + x_{13b} - x_{34} = 1 \\
& -x_{34} + x_{24} - x_{46} = 1 \\
& -x_{25} - x_{56} = 0 \\
& -x_{46} + x_{56} = 2
\end{aligned} \tag{3.9}$$

To show that the integer constraint of the MCNF can be relaxed, we have to first prove that the constraint matrix of the MCNF problem is totally unimodular (TU). A matrix $A_{p\times q}$ is said to be TU if the determinant of each square submatrix of A is equal to 1, 0 or -1. That is, if $B_{n\times n}$ with $n < \min(p, q) \in A_{p\times q}$, and $det(B) \in \{0, -1, 1\}$, then A is TU. The definition of total unimodularity does not show us how to prove that a matrix is TU. To demonstrate that a matrix is TU, we will need the following theorems.

Theorem 3.1 *[22] If a matrix A is TU, the transpose of A and (A,I) are also TU.*

Theorem 3.2 *[29] Suppose that A is a matrix with elements equal to 0, +1,*

-1, and such that each column has at most two nonzero elements. A is TU if and only if its rows can be divided into two disjoint sets I_1 and I_2 satisfying the following: If the two nonzero elements of a column have the same sign, one of them is in I_1 and the other is in I_2. If the two nonzero elements of a column have the opposite sign, both of them are in I_1 or both of them are in I_2.

Using these two theorems, we will first show that the constraint matrix of the example MCNF problem in Equation 3.9 is TU. We will then extend the proof to include the MCNF problem expressed by Equations 3.7. Expressing the MCNF problem in matrix form, we get the matrix in Table 3.4.

The constraint matrix, A, shown in Table 3.4 can be expressed as:

$$A = \begin{bmatrix} B \\ I \end{bmatrix} \tag{3.10}$$

(14) where B is given by

$$B = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

and I is a 9×9 identity matrix. From Theorem 3.1 , we know that to show that A is TU, we only need to show that B is TU. Since matrix B satisfies the

Table 3.4: The MCNF problem in matrix from.

$$A = \begin{bmatrix}
-1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

condition Theorem 3.2, it can easily be seen that the constraint matrix, A, of the MCNF problem expressed by Equations 3.8 can always be expressed in the form

$$A = \begin{bmatrix} B \\ I \end{bmatrix}$$

where I is a $n \times n$ identity matrix and n is the number of edges in the SFG. Since each column of B represents a directed edge x_{uv} in the SFG, there will always be two elements, one +1 and one -1, in each column. The -1 will be included in the constraint equation of node u, and +1 will be included in the constraint equation of node v. The matrix B will always satisfy the condition of Theorem 3.2.

Since the constraint matrix of the ILP in Equations 3.5 can be written as the transpose of the constraint matrix of the MCNF problem in Equations 3.8, the conclusion that the constraint matrix of the ILP in Equations 3.5 is also TU follows directly after applying basic linear algebra theorems found in [18].

Since the constraint matrix satisfies the definition of unimodularity, the ILP problem in Equations 3.5 and the MCNF problem in Equations 3.8 can be reduced to a LP problem based on the following theorem:

Theorem 3.3 *[8]: Let A be any $m \times n$ matrix of integers and b be any vector of m integers. Then the following two statements are equivalent.*

- $P(A,b) = P_I(A,b)$ *for all* $b \in Z^m$.
- A *is totally unimodular.*

with $P(A,b) = \{x \in R^n : Ax \leq b, x \geq 0\}$ *and* $P_I(A,b) = conv\{P(A,b) \cap Z^n\}$.

The buffer synchronization-minimization problems can then be solved using linear programming algorithm such as simplex method. With this, the integer solutions are always guaranteed.

Methods for Solving the Buffer Minimization Problem

Both the ILP problems in Equations 3.5 and Equations 3.8 can be solved by relaxing the integer constraint. Therefore, we have two methods for solving the buffer minimization problem as shown in Figure 3.16. Method 1 consists of forming the ILP expressed by Equations 3.5 and solving the ILP with the integer constraint relaxed, and Method 2 consists of forming the MCNF problem expressed by Equations 3.8, solving the MCNF problem with the integer constraint relaxed and converting the x_{uv} in the MCNF to d_{uv}.

Although Method 2 involves more steps than Method 1, Method 2 may consume less time since much efficient algorithms such as the one proposed by [23] can be used to solve MCNF problem. The solution of the MCNF problem can then be used to obtain the solution of the ILP in Equations 3.5 based on the following theorem:

Theorem 3.4 *Complementary Slackness Theorem Let x^* and w^* be any feasible solutions to the primal and dual problems. Then they are respectively optimal if and only if*

$$(c_j - w^* a_j)x_j^* = 0, \quad j = 1, \ldots, n$$

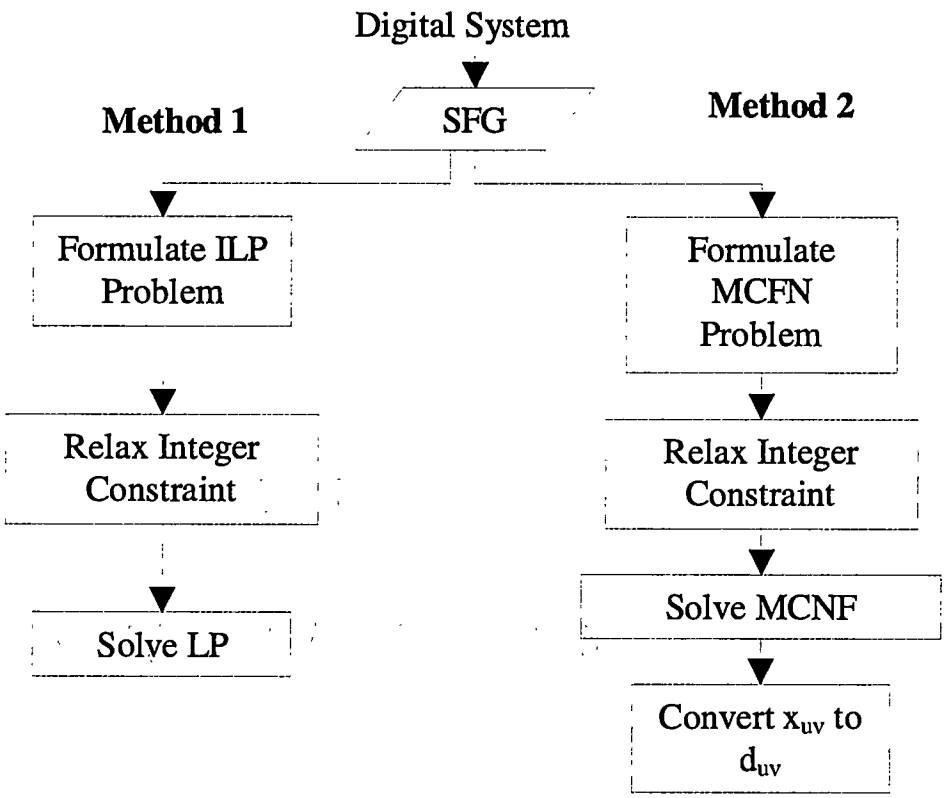


Figure 3.16: Two methods for data synchronization.

and

$$w_i^*(a^i x^* - b_i) = 0, \quad i = 1, \ldots, m$$

This theorem can be used to relate MCNF problem to the ILP problem in Equations 3.5. It indicates that at least one of the two terms in each expression must be zero. In particular,

$$x_j^* > 0 \Rightarrow w^* a_j = c_j$$

$$w^* a_j < c_j \Rightarrow x_j^* = 0$$

$$w_i^* > 0 \Rightarrow \mathrm{a}^{\mathrm{i}} x^* = b_i$$

$$\mathrm{a}^{\mathrm{i}} x^* > b_i \Rightarrow w_i^* = 0$$

Hence at optimality, if a variable in one problem is positive, then the corresponding constraint in the other problem must be tight. If a constraint in one problem is not tight, then the corresponding variable in the other problem must be zero. To show how this theorem can be used to convert the solution of the MCNF problem in Equations 3.8 to the solution of the ILP in Equations 3.5, we will use the example MCNF problem in Equations 3.9. Solving Equations 3.9, we get

$$x_{01} = 2, x_{02} = 0, x_{13a} = 3, x_{13b} = 0, x_{24} = 1, x_{25} = 0, x_{34} = 2, x_{46} = 2, x_{56} = 0$$

Using the equation, we have

$$z_{01} = 1, z_{02} = -1, z_{13a} = 2, z_{13b} = -1, z_{24} = 0, z_{25} = -1, z_{34} = 1, z_{46} = 1, z_{56} = -1$$

Substituting these values of z_{uv} into the equations in Table 3.3, we find that Equations in row 8, 10, 12, 14 and 15 in Table 3.3 are not tight. Therefore, the corresponding primal variables are zero. That is,

$$d_{01} = d_{13a} = d_{24} = d_{34} = d_{46} = 0$$

Substituting 0 for these d_{uv} in Table 3.2 and also notice that $D_0 = 0$, we have the following simultaneous equations:

$$D_1 = 0$$

$$D_2 - d_{02} = 0$$

$$-D_1 + D_3 = 1$$

$$-D_1 + D_3 - d_{13b} = 1$$

$$-D_2 + D_4 = 2$$

$$-D_2 + D_5 - d_{25} = 2$$

$$-D_3 + D_4 = 2$$

$$-D_4 + D_6 = 3$$

$$-D_5 + D_6 - d_{56} = 2$$

Solving these simultaneous equations, we get

$$\begin{aligned} D_1 &= 0 \\ D_2 &= 1 \\ D_3 &= 1 \\ D_4 &= 3 \\ D_6 &= 6 \\ d_{02} &= 1 \\ d_{13b} &= 0 \\ d_{25} + d_{56} &= 1 \end{aligned} \tag{3.11}$$

Equation 3.11 indicates that a delay buffer of one unit delay can be inserted at either edge (2,5) or edge (5,6) to obtain an optimal solution. This is due to the fact that edge (2,5) and edge (5,6) are on the same path. Inserting the unit delay buffer in either location will give the same effect to the synchronization of the system. As a result, we have two sets of solutions that give the same $\sum d_{uv}$, which is equal to 2. Using Simplex algorithm to solve the ILP problem in Table 3.2 with the integer constraint being relaxed will give us the following solution:

$D_0 = 0, D_1 = 0, D_2 = 1, D_3 = 1, D_4 = 3, D_5 = 4, D_6 = 6$

$d_{01} = 0, d_{02} = 1, d_{13a} = 0, d_{13b} = 0, d_{24} = 0, d_{25} = 1, d_{34} = 0, d_{46} = 0, d_{56} = 0$

This solution is consistent with what we found from the MCNF method.

System Containing Hyperarcs

An output connected to more than one input is called a hyperarc (shown in Figure 3.17. If hyperarcs exist in the digital system, special representations of these

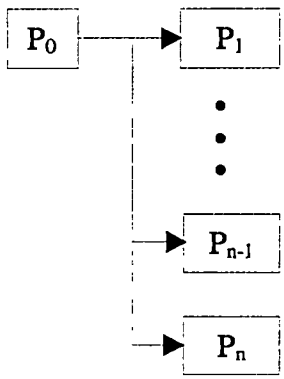


Figure 3.17: Hyperarc.

nets need to be considered so that the above procedure can be applied to these situations. One easy solution is to treat the hyperarc as having multiple ordinary outputs and constructs the SFG as shown in Figure 3.18(b). The second representation can be obtained by inserting a virtual node, V1, with zero processing time as Figure 3.18(c) shows. Another representation, which is proposed in [23], is depicted in Figure 3.18(d). This representation uses a binary tree structure, which systematically introduces virtual node to the SFG.

Figure 3.19 shows an example of the delay buffer required for the three different types of SFG representations in Figure 3.18. In the example, a total of 12 units of delay buffers are required to synchronize the hyperarc represented using the SFG representation shown in Figure 3.18(b). With the insertion of a virtual node, the total number of delay buffers can be reduced to 6 as shown in Figure 3.19(b). If the binary structure is used, the total number of delay buffers can be reduced to 5 as shown in Figure 3.19(c). This example demonstrates that

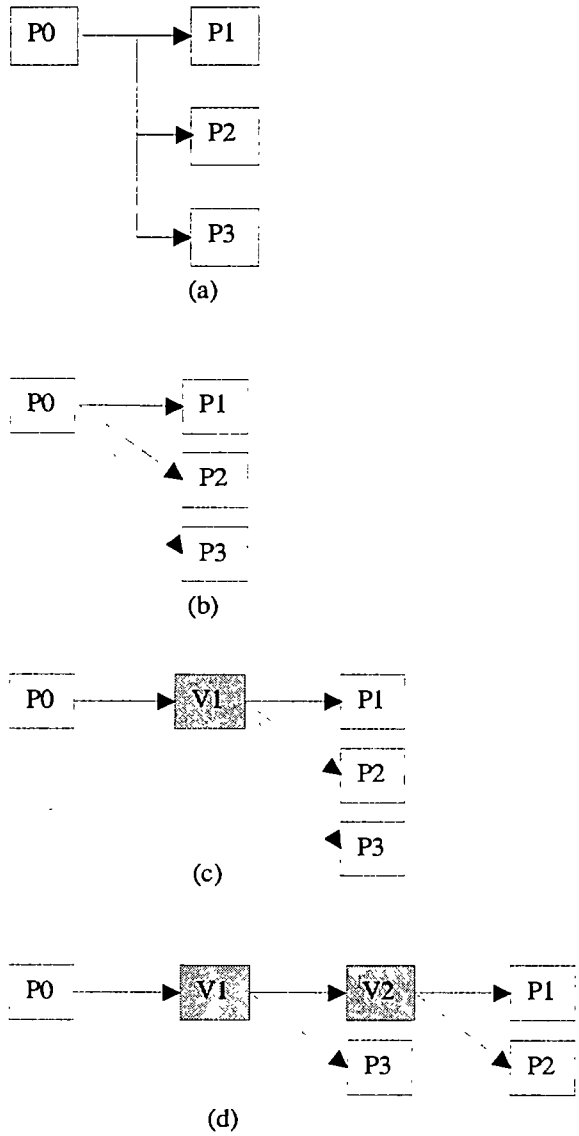


Figure 3.18: Different representations of a hyperarc.

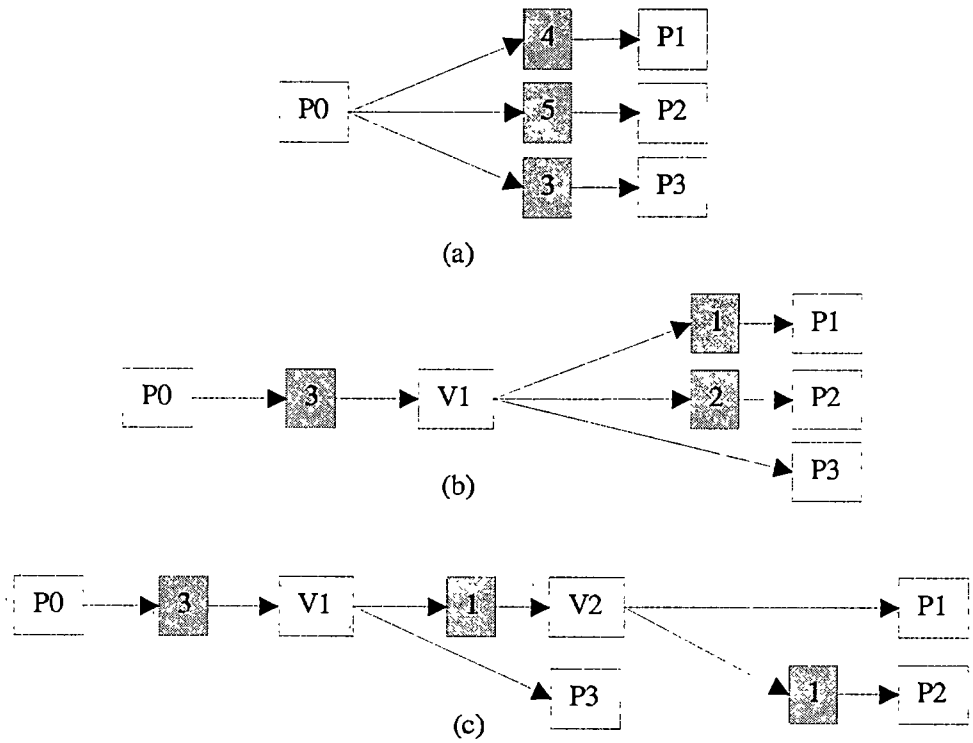


Figure 3.19: Delays of the hyperarc.

some of the delay buffers can be shared along the hyperarc through the insertion of the virtual node. The use of binary tree structure in representing the hyperarc can greatly exploit the buffer sharing property. To fully exploit the buffer sharing property, an algorithm must be used to arrange the processing nodes in the hyperarc. Different arrangements of the processing nodes driven by a hyperarc may vary the total number of buffers along the hyperarc because each arrangement may cause different length delay buffer to be shared among the nodes driven by the hyperarc.

3.4 ACS Back-end Flow

Once, all the data paths are matched and synchronized, the ACS back-end flow can be used to map the netlist onto the ACS.

3.4.1 Partitioning

In ACS, which is composed of reconfigurable devices such as FPGA, each of the reconfigurable devices has a certain capacity in terms of logic blocks. If the netlist does not fit in a single reconfigurable device, the netlist has to be partitioned into sub-netlists. One of the main problems in partitioning is complexity. The research in partitioning theory has seen many algorithms with good results even with today's design complexity. The main disadvantage of these approaches is that they are based on gate-level net-lists, thus requiring hours to execute.

In CHAMPION, we drastically reduced the complexity by keeping the struc-

tural information of the netlist. The partitioning is performed at the glyph-level. This yields very low netlist complexity (hundreds vs. tens of thousands). Therefore, partitioning is performed with netlists containing hundreds of nodes instead of tens of thousands, and the partitioning process has a very short runtime (seconds vs. hours). Another advantage of performing partitioning at the glyph level is that the functional flow information is preserved. Thus, debugging and simulation of the system are facilitated even after the partitioning.

To further reduce the complexity of the circuit, we configure the programmable logic components and their interconnects in the ACS board into a linear array. With this topology, the partitioning operates on an order that proceeds in a forward-only direction. A new recursive partitioning method based on topological ordering and levelization (RPL) [12] has been developed to take advantage of this topology and further reduces the partitioning time.

For our design flow, the partitioning problem is based on the following constraints: capacity per partition, number of I/O pins per partition, RAM access, and temporal partitioning. The first two constraints are used to meet the limitations of the programmable logic components. The third constraint deals with the memory access for each the programmable logic component. The architectures of some of the ACSs require that a fixed number of local RAMs are available to each FPGA for data writing and data reading. Therefore, a partition can contain only a certain number of RAM access modules. The fourth constraint deals with temporal partitioning of the ACS board. If the entire application cannot fit in

one board configuration, then multiple configurations of the board are necessary and storage of intermediate results between board configurations is needed. In this case, one pair of RAM-access modules must be added to each configuration.

To solve the partitioning problem, three different approaches were investigated in our research. In the first and second approaches, we implemented two existing algorithms: a hierarchical partitioning method based on topological ordering (HP) [30] and a recursive algorithm based on the Fiduccia and Mattheyses bipartitioning heuristic (RP) [13]. Some modifications have been made on these algorithms to take advantage of the acyclic nature of our net-list, and to handle the RAM access constraint and the temporal partitioning constraint. A new recursive partitioning method based on topological ordering and levelization (RPL) [12] was also introduced. In addition to handling the partitioning constraints, the new approach efficiently addresses the problem of minimizing the number of FPGAs used and the amount of computation, thereby overcoming the weaknesses of the HP and RP algorithms.

3.4.2 Netlist to Structural VHDL, Synthesis, and Placement & Routing

After partitioning, the graph-based netlist format is translated into a structural VHDL representation. The required I/O ports for each of the sub-netlists are then added to the VHDL files. The resulting VHDL files are synthesized and merged with the pre-compiled VHDL components corresponding to the Cantata

glyphs. Each sub-netlist is then placed and routed.

3.4.3 Host Program Generation

The final step in the ACS design flow is the generation of the host program. A CHAMPION software tool will generate the host program to download the configuration file to the corresponding programmable logic component on the ACS. A set of function calls to communicate with the board is provided by the manufacture of the ACS. The host program will initialize the ACS board and download the programming bit files using these function calls. The host program will also read the input data from the host workstation, send the data to the ACS and write the output back to the host workstation.

3.5 ASIC Back-end Flow

3.5.1 Netlist to Structural VHDL

To implement the design in ASIC, the entire graph-based netlist format is first translated into structural VHDL. No partitioning is required in this process since the entire design will be implemented in a single semiconductor chip.

3.5.2 Design Compilation and Optimization

Once the VHDL description of the design is obtained, the VHDL description is synthesized and optimized to achieve the optimal gate-level for the design. The synthesis and optimization process is performed using Synopsys Design Compiler.

A software tool was developed to automate the synthesis and optimization process. Two compilation strategies have been implemented in CHAMPION:

- Top-down compile method.
- Bottom-up compile method.

The top-down compilation strategy compiles the VHDL source by reading the entire design at once. All the glyphs in the design are then grouped and flattened. The flattened design is then synthesized and optimized. The top-down compile strategy provides better results since the optimization is performed across the entire design. It provides a push-button approach since the inter-glyph dependencies are handled automatically through the flattening of the design. However, this approach is extremely memory intensive. It also requires a longer compile time compared to the bottom-up approach.

The second compilation approach is performed bottom-up. The compilation process begins at the lowest level and ascends to the topmost level of the design. After synthesizing and optimizing each sub-design, the sub-design is preserved from being modified or replaced during subsequent optimizations. This approach allows the optimization process to be performed on one sub-design at a time instead of the entire design at once. Thus, less memory resources and compile time are required for this approach. However, the quality of the results obtained from this approach is not as good as that of the top-down approach since the optimization is performed locally. That is, the optimization is only performed to one sub-design at a time.

Depending on the compilation approach used, two different types of gate-level netlists are produced after the compilation process. If the top-down approach is used, the resulting gate-level netlist consists of a flattened design. If the bottom-up approach is used, the resulting netlist is hierarchical.

3.5.3 Physical Layout Generation

The final step in the back-end flow of ASIC is the generation of the physical layout. The layout generation is performed using the commercial software EPOCH. The steps for generating the physical layout using EPOCH are shown in Figure 3.20. The EPOCH tool reads the gate-level netlist and generates the standard and datapath cells corresponding to the named parts in the technology library specified by the user. These cells are then placed in a manner that optimizes for density, route minimization and timing. The placement is performed bottom up, starting at the lowest level and ascending to the highest level composites. Global and detail routing on the cells are then performed using the routing channels created during placement.

After the routing process, the sizes of the buffers are adjusted based on the capacitive loads of the cells and routes. Once the appropriate size is chosen, the cells affected are regenerated with the newly sized buffers. The power consumption at the different power nodes on the chip is then calculated and the appropriate widths to the power rails are assigned. The placement of the cells is adjusted to accommodate cells that have become larger due to upsizing of buffers.

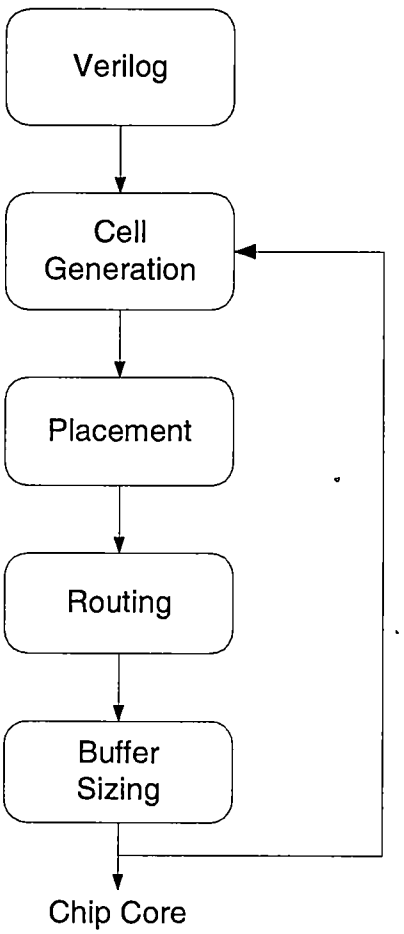


Figure 3.20: The design flow of Epoch tool.

The cells are then rerouted with the correct power rail sizes. The final output is a finished design consisting of the core of the ASIC.

As in the synthesis process, two approaches to the physical layout generation are implemented. If the gate-level netlists provided by the synthesis process is a flattened netlist, the top-down approach will be used. The entire netlist is treated as a single design and the steps in Figure 3.20 are performed. Since the placement, routing and buffer sizing are performed for the entire design, this approach requires more memory resources and longer compile time compared to the bottom-up approach

If the gate-level netlist is hierarchical, the bottom-up approach is used. The entire layout process in Figure 3.20 are performed for each sub-design, starting at the lowest level and ascending to the topmost level. After generating the layout for each sub-design, the layout of the sub-design is preserved from being modified or replaced during subsequent layout processes. This approach allows the layout process to be performed on one sub-design at a time instead of the entire design at once. Thus, less memory resources and compile time are required for this approach. However, the area of the core generated using this approach is usually larger than that of the top-down approach.

CHAPTER 4

Implementation

The design flow (shown in Figure 4.1) for automatic mapping of Cantata applications onto ACS and ASIC has been implemented in this dissertation. A complete design environment that allows the users to specify, simulate, debug and implement their applications on ACS and ASIC has been developed. This design environment, named CHAMPION, consists of a set of software tools that implements all the steps of the design flow in Figure 3.1. For this design environment, the Wildforce board from Annapolis Micro Systems [1] was chosen as the ACS board for the ACS flow. For the ASIC implementation, the Hewlett Packard CMOS processes HP26G was used.

This chapter describes all the software tools in CHAMPION. Sections 4.1 and 4.2 provide brief descriptions on the Wildforce board and the HP26G technology. Section 4.3 describes the two different user interfaces developed for the software design environment. In Section 4.4, the software tools developed for the glyph development flow are presented. The software tools implementing the front-end flow are then described in Section 4.5. Finally, Sections 4.6 and 4.7 provide details of the tools implemented for the ACS and ASIC back-end flow respectively.

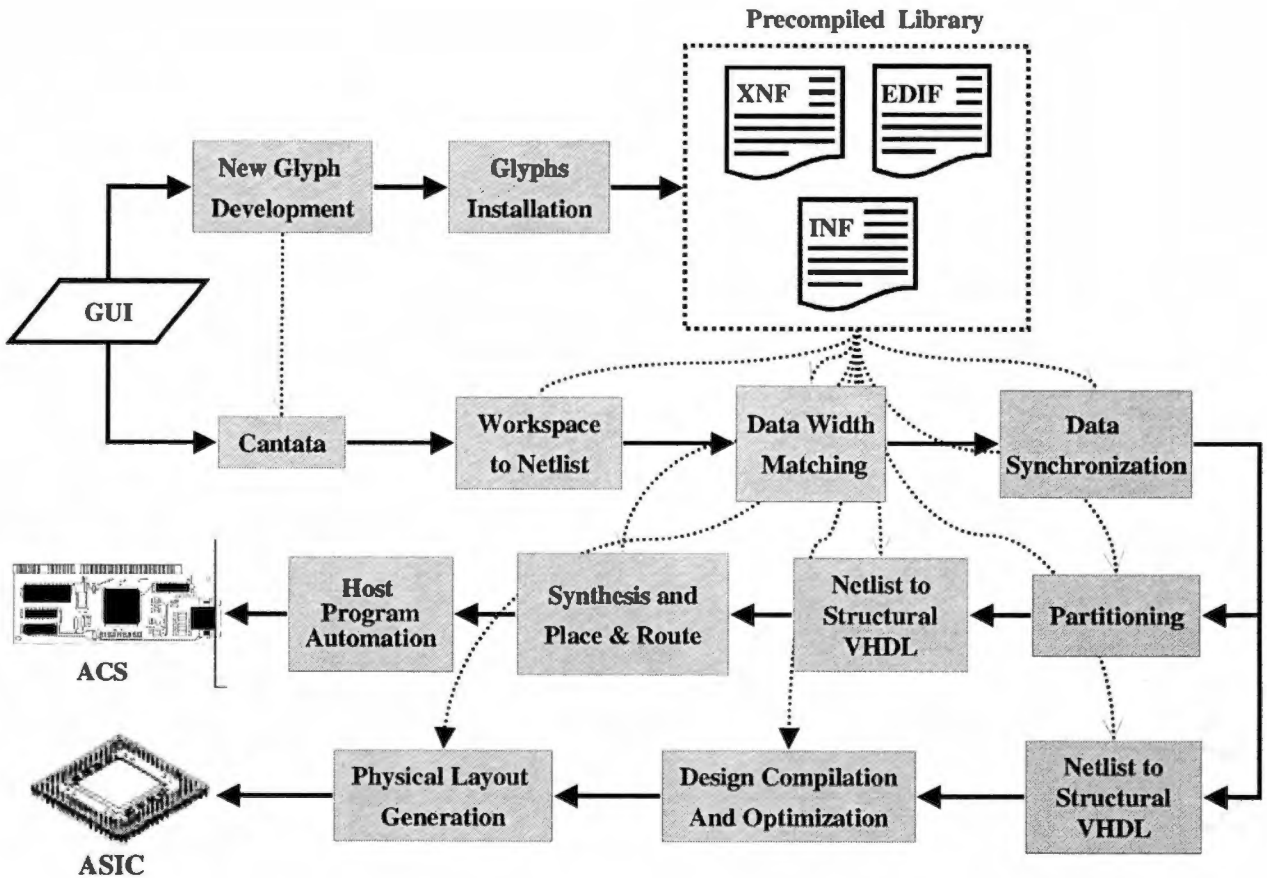


Figure 4.1: Design flow of CHAMPION.

Table 4.1: Resources on XC4013XL and XC4036XL FPGAs. *Source: Xilinx, Inc, The Programmable Logic Data Book 1998.*

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

4.1 Wildforce Board

The Wildforce board from Annapolis Micro Systems (AMS) was chosen as the first ACS board to be used in the CHAMPION project. It is a PCI-bus card, which contains 4 Xilinx XC4000 family chips for computational purposes. These four FPGAs are the XC4013XL FPGAs and are referred to as *processing elements* (PEs) by AMS. The four XC4013XL FPGAs are designated as PE1, PE2, PE3 and PE4. A fifth Xilinx XC4000 chip for communicating with the host computer is also included in the board. It is a XC4036XL FPGA and is referred to as *control processing element.* The XC4036XL is designated as CPE0. Table 4.1 shows the resources on the XC4013XL and XC4036XL FPGAs.

CPE0 has more than twice the amount of CLBs compared to the other FPGAs. It also includes control lines for various resources on the board, such as the external I/O interface and crossbar configuration register. These control lines are not available to the other four FPGAs.

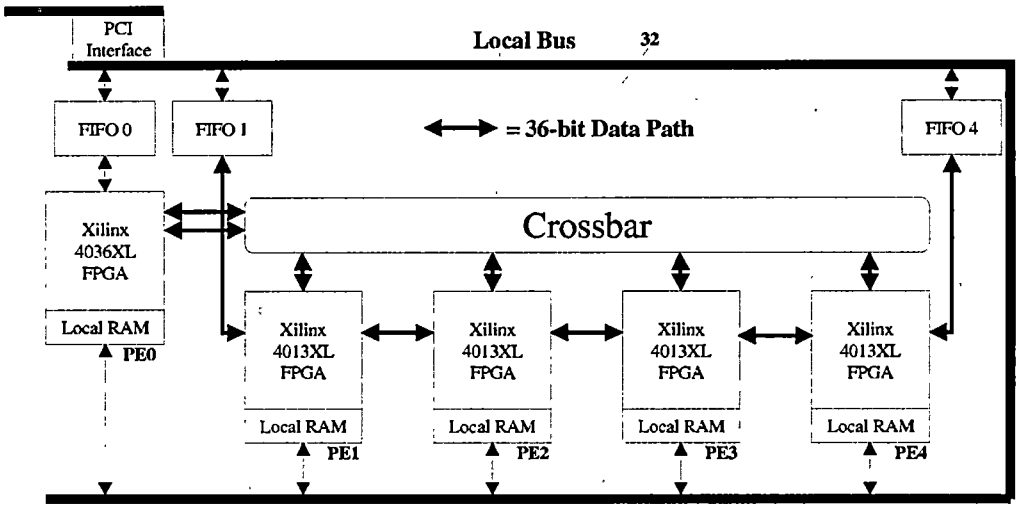


Figure 4.2: A simplified block diagram of the Wildforce board.

Each FPGA on the board has a small daughterboard, which allows memory or a digital signal-processing chip to interface with the FPGA. Each of the FPGAs on the Wildforce used in CHAMPION has 32 KByte of 32-bit SRAM on its daughterboard. A dual-port memory controller is included in the daughterboards to allow both the FPGA and the host computer to access the SRAM.

A simplified block diagram of the Wildforce board is shown in Figure 4.2. The four PEs are connected together in a linear array by a 36-bit systolic bus. All five FPGAs can communicate with each other through a 36-bit crossbar, which selectively allows connections between any of the processing elements. Since CPE0 is not connected to the systolic bus, it can only connect to the processing elements through the crossbar.

To reduce the complexity of the Wildforce board to a more manageable level, a

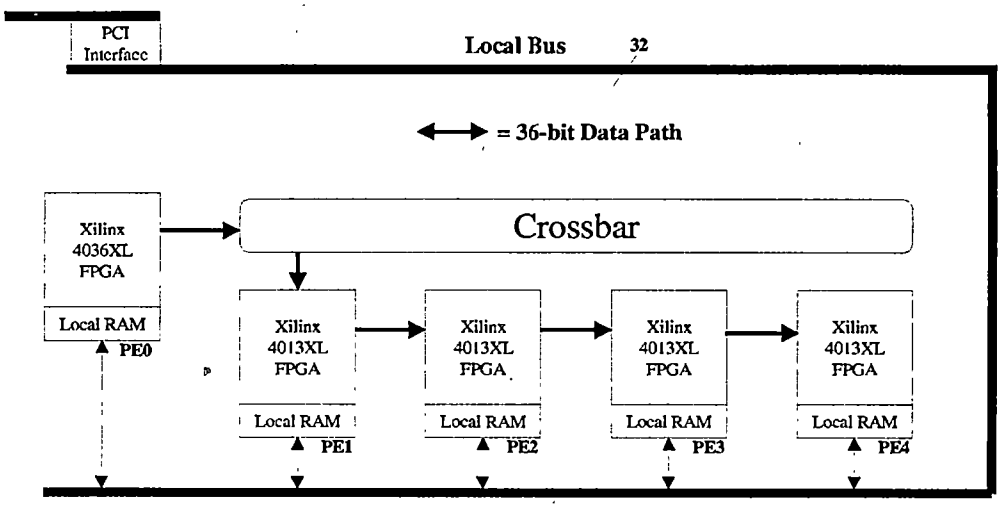


Figure 4.3: Wildforce board as used in CHAMPION.

constrained configuration of the board was used in CHAMPION. This constrained configuration of the board did not use any of the FIFOs avaliable on the board. Communication between the host computer and the board was performed through the SRAM associated with each PE. The PEs and their interconnects in the ACS board was configured into a linear array and the direction of all connections between the PEs was fixed in one direction only. With this configuration, all the signals started in CPE0 and traveled to PE1. No signals could travel from PE1 back to CPE0. Similarly, no signals were allowed to travel from PE2 to PE1. The signals were only allowed to travel from PE1 to PE2. The configuration of the Wildforce board used in CHAMPION is shown in Figure 4.3.

AMS provides a set of VHDL templates that define the signals used in the Wildforce board. These modifiable templates were developed for the Synplify

tool from Synplicity. Thus, in the CAHMPION ACS flow, Synplify was used to synthesize all the hardware glyphs and designs to be mapped onto the Wildforce board.

4.2 HP26G

In the CHAMPION ASIC flow, the physical layouts of the cores were generated in the Hewlett Packard HP26G technology. The HP26G is a 0.8 um CMOS process. The HP26G process uses 3-metal layers for routing. The designs implemented in the HP26G technology use a 5 Volt power supply.

4.3 CHAMPION Graphical User Interface and Command Line User Interface

CHAMPION supports both the graphical user interface (GUI) and command line user interface (CLUI). If a workstation with the X Window System is used, the CHAMPION GUI can be used to execute all the tools in CHAMPION. On the other hand, if a workstation without the X Window System is used, the CLUI can be used to execute all the CHAMPION tools, except for tools for viewing schematics of the designs.

Figure 4.4 shows the CHAMPION GUI. The CHAMPION GUI provides a menu-driven interface to all the CHAMPION tools for the ACS and ASIC mapping. The GUI frees the user from having to learn and use the command line

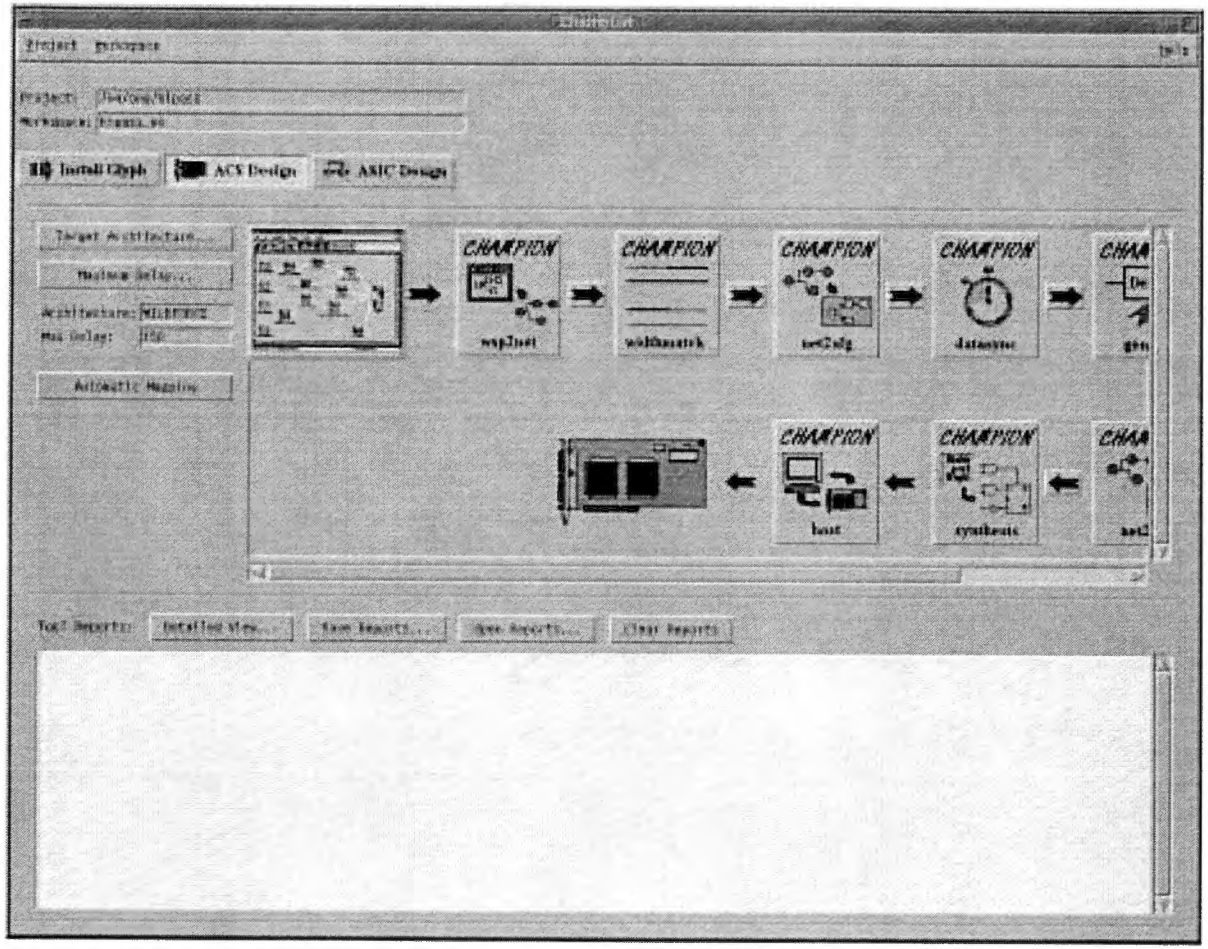


Figure 4.4: CHAMPION graphical user interface.

language. It allows the user to setup up the CHAMPION project, specify the ACS hardware architecture or ASIC technology, and execute all the CHAMPION tools. In addition, the GUI also facilitates the data transfer from one CHAMPION tool to another.

All the tools in the CHAMPION design environment can also be invoked at the UNIX command prompt. At the UNIX command prompt, the user can enter commands composed of command names, arguments, and variable values to perform the ACS and ASIC mapping. The CLUI allows shell script to be used to facilitate the ACS and ASIC mapping.

The user can use either the CHAMPION GUI or CLUI to perform the ACS and ASIC mapping. In fact, the user can use both interfaces, moving between them, depending on which interface is most convenient or informative for a certain task. For debugging a design, the CHAMPION GUI is better than CLUI because it can be used to view the schematic and other results after the execution of each tool. However, if the user is proficient with CHAMPION, CLUI is often found to be sufficient and easier to use. For example, the user can create script files to be executed repeatedly from the UNIX command prompt, modifying values during each cycle to optimize a design. In this case, the GUI is only required to view the schematics or tool reports.

4.4 Glyph Development Flow

A set of library glyphs has been developed in the CHAMPION project. New library glyphs can be developed and added to the CHAMPION library as needed. A set of software tools can be used to facilitate the process of developing, verifying and installing the new glyphs in the CHAMPION library. This set of tools constitutes the glyph development flow.

4.4.1 Glyph Development Tools

To incorporate a new glyph into the CHAMPION library, two versions of the glyph, a fixed-point C/C++ version and a VHDL version, are required. To ensure the same functionality between the C/C++ version and the VHDL version, the glyph parameters and its functionality must first be defined. Based on these predefined parameters and functionality, the fixed-point C/C++ program for the glyph can be generated.

Once the C/C++ program is developed, a hardware version of the C/C++ program must be developed using VHDL. The functionality of the VHDL code must be identical to that of the C/C++ program. This can be achieved by using the C/C++ program to generate a set of output vectors for a set of applied input test vectors. These input and output vectors constitute a test bench that is used to verify that bitwise identical behavior is achieved between the C/C++ program and the VHDL code.

To accelerate the glyph development process, the commercial software, A|RT

Library and Builder [11] from the Frontier Design has been integrated to the glyph development flow. The A|RT Library and Builder free users from having to generate the VHDL description of the hardware. They provide users with the ability to generate the VHDL code directly from a C-code specification.

The A|RT Library is a C++ library that allows the user to add fixed-point arithmetic to C or C++ programs. A|RT Library consists of a set of C++ classes (as shown in Figure 4.5) with the characteristics of fixed-point data types and a set of operators for these fixed-point data types. The fixed-point classes model the intended behavior in a bit-precise way, including overflow and quantization effects. Therefore, the user can take full control over the number of bits and precision that is used to represent the data, regardless of the compiler and computing platform. Once the C/C++ program has been developed, the A|RT Builder can be used to convert the fixed-point C/C++ applications to synthesizable VHDL.

4.4.2 Glyph Installation Tools

To complete the glyph development process, the C/C++ code and the corresponding VHDL codes have to be installed in Cantata and CHAMPION precompiled library respectively. Several tools can be used to facilitate the process of installing the C/C++ and VHDL codes. To install the C/C++ program in Cantata, the Khoros tools, *Craftsman* and *Composer* [2] can be used. The *Craftsman* tool allows the creation and deletion of glyphs for Cantata. It also allows the users to modify the glyph attributes such as the description, category, subcat-

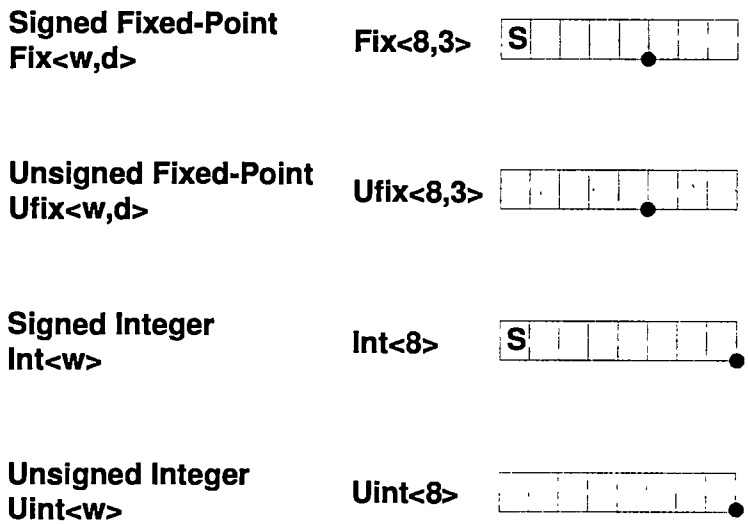


Figure 4.5: Art Library classes [4].

egory and icon name. When creating a new glyph for Cantata, the *Craftsman* tool generates a complex C/C++ wrapper that provides the means for handling operations such as data type conversion, data synchronization and data transfer between glyphs. The C/C++ code for the glyph can then be inserted in this wrapper with the help of the Composer tool. Once the C/C++ code has been inserted in the wrapper, the Composer tool can be used to compile the code. During the compilation, the Composer tool automatically generates an icon for the glyph in Cantata. Applications can be developed in Cantata by interconnecting this icon with other icons in Cantata.

To automate the process of installing the hardware glyph in the CHAMPION library, a software tool, called *Geninf*, was developed. *Geninf* automates the process of synthesizing the VHDL code and generating the INF file. It first

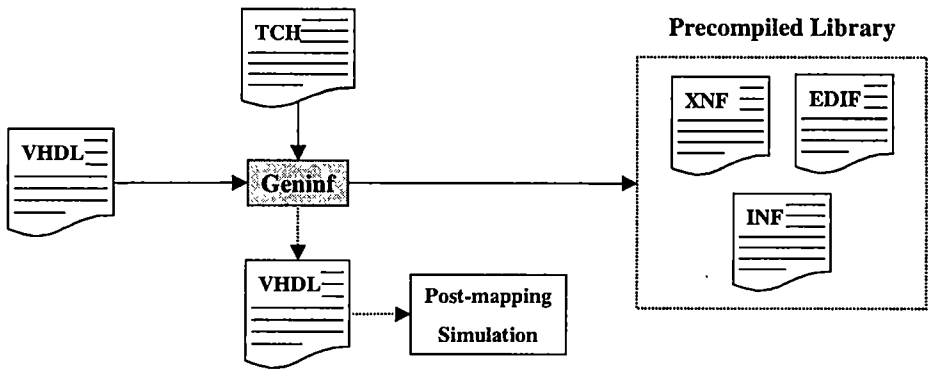


Figure 4.6: Glyph installation process using *Geninf*.

generates a Tool Command Language (TCL) description of the hardware based on the hardware architecture specified in the technology file (TCH file). The TCL file is then executed to synthesize the VHDL code and to obtain the required technology-dependent netlist file (XNF, EDIF, AHDL, DSL or QDIF formats). The synthesis is performed using the commercial logic synthesis tool, Synplify. Based on the output of the synthesis, *Geninf* will then generate an INF file for storing the size, latency and I/O data bit-widths of the hardware glyph. This information will be used during the data width matching, data synchronization and partitioning processes. The generated INF and technology-dependent netlist file will then be stored in the CHAMPION precompiled library to complete the glyph installation process. The hardware glyph installation process using *Geninf* is shown in Figure 4.6.

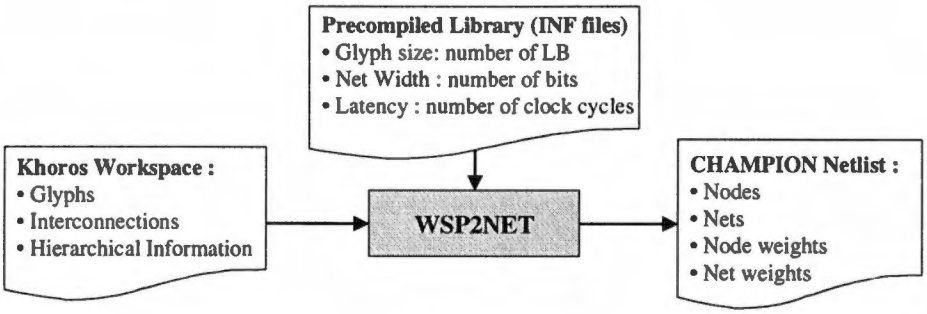


Figure 4.7: Conversion of Cantata workspace to CHAMPION netlist.

4.5 Front-End Flow

4.5.1 Conversion of Cantata Workspace to CHAMPION Netlist

The first step in the mapping process is to convert the Cantata workspace to CHAMPION netlist. A software tool, named *Wsp2net*, was developed to perform this task. *Wsp2net* (shown in Figure 4.7) converts the Cantata workspace into a more graph-oriented netlist format. The netlist is a directed hypergraph where each glyph is represented as a node and the interconnections between glyphs are represented as directed hyperarcs. Based on the information from the INF file, weights are assigned to the nodes and hyperarcs of the directed graph. The weights of the nodes correspond to the size in terms of the number of logic blocks, and the weights of the hyperarcs correspond to the net-width of the glyph interconnections.

4.5.2 Data Width Matching

A CHAMPION tool, called *Widthmatch*, was developed to perform the data width matching. *Widthmatch* tracks and matches all the positively and negatively mismatched data paths. If a positively mismatched data path is found, *Widthmatch* generates a "truncating" glyph and inserts it in the data path. In order to generate the "truncating" glyph, *Widthmatch* first generates the VHDL file for the glyph. The VHDL file for a sample "truncating" glyph is shown in Figure 4.8(a). This glyph truncates a 9-bit signal to an 8-bit signal. Once the VHDL file is generated, *Widthmatch* invokes *Geninf* to synthesize and install the "truncating" glyph in the CHAMPION library. The hardware architecture of the *truncate_9_8* glyph in Figure 4.8(a) is shown in Figure 4.8(b).

If a negatively mismatched data path is found, *Widthmatch* generates a "padding" glyph and inserts it in the data path. *Widthmatch* generates the VHDL file for the glyph, and invokes *Geninf* to synthesize and install the "padding" glyph in the CHAMPION library. The VHDL file for a sample "padding" glyph is shown in Figure 4.9(a). This glyph pads an 8-bit signal to a 9-bit signal. The hardware architecture of the *pad_9_8* glyph in Figure 4.9(a) is shown in Figure 4.9(b).

4.5.3 Data Synchronization

To synchronize the CHAMPION netlist, a set of tools was developed. A CHAMPION tool, *Net2sfg*, was developed to first convert the netlist to signal flow graph

```
-- Filename : truncate_9_8.vhd
-- Generated by : Widthmatch
-- Date : Mon Mar 26 17:11:14 2001

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity truncate_9_8 is
 generic (g_num_input_bits: positive:= 9;
          g_num_output_bits: positive:= 8);
 port ( i: in std_logic_vector(g_num_input_bits -1 downto 0);
        o: out std_logic_vector(g_num_output_bits -1 downto 0));
end truncate_9_8;

architecture behave of truncate_9_8 is
begin
  o <= i(g_num_output_bits -1 downto 0);
end;
```

(a)

(b)

Figure 4.8: Example of truncating glyph. (a) VHDL file for a truncate_9_8 glyph and (b) the corresponding hardware architecture.

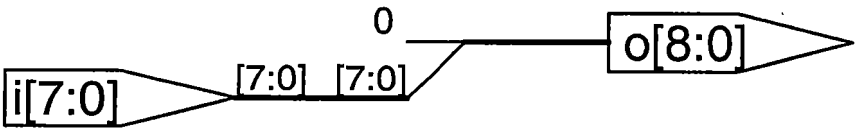
```
-- Filename : pad_8_9.vhd
-- Generated by : Widthmatch
-- Date : Mon Mar 26 17:11:14 2001

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity pad_8_9 is
  generic (g_num_input_bits: positive:= 8;
           g_num_output_bits: positive:= 9);
  port (i: in std_logic_vector(g_num_input_bits-1 downto 0);
        o: out std_logic_vector(g_num_output_bits-1 downto 0));
end pad_8_9;

architecture behave of pad_8_9 is
begin
  o(g_num_input_bits -1 downto 0) <= i(g_num_input_bits -1 downto 0);
  o(g_num_output_bits -1 downto g_num_input_bits) <= "0" ;
end;
```

(a)



(b)

Figure 4.9: Example of padding glyph. (a) VHDL file for a pad_8_9 glyph and (b) the corresponding hardware architecture.

Table 4.2: Comparison of execution time of Simplex algorithm.

LP problem	Time required to solve the LP Problem
Highpass filter(16 nodes, 23 nets)	1 second
ATR(135 nodes, 321 nets)	2 seconds

(SFG). Once the netlist is converted to SFG, the CHAMPION tool, *Lp_form*, can be used to formulate the linear programming (LP) problem for the delay buffer minimization problem. The LP problem can then be solved using the non-commercial linear programming software, *Lp_solve*, written by Michel Berkelaar from Eindhoven University of Technology. *Lp_solve* implements the Simplex algorithm, which is known to be very efficient for solving large LP problems. This can be seen from Table 4.2, where the times required for Simplex algorithm to solve two different LP problems are shown.

After solving the LP problem, the delay buffers that are required to synchronize the SFG are known. These delay buffers may not exist in the CHAMPION precompiled library. Therefore, a CHAMPION tool, *Gendly*, has been developed to generate the glyphs for those delay buffers that cannot be found in the CHAMPION precompiled library. Two versions of VHDL files are generated for each delay buffer. The first version of VHDL file is generated for the Wildforce board. This version of VHDL uses the pipelined delay element generated by the Xilinx CORE Generator. The Xilinx CORE Generator is a design tool that delivers

parameterizable cores optimized for Xilinx FPGAs. The delay cores generated using the CORE Generator deliver high levels of performance and area efficiency since the CORE Generator leverages Xilinx FPGA architectural advantages such as look-up tables (LUTs), distributed RAM, segmented routing, floorplanning and power utilization information. A sample VHDL file for a 4-bit, 5-cycle delay glyph is shown in Figure 4.10. In the VHDL file, the *dly_4_5* is a pipelined delay component generated using the CORE Generator. The hardware architecture for this VHDL file is show in Figure 4.11.

Since the first version of VHDL file contains Xilinx technology dependent components, it cannot be used for ASIC implementation. Therefore, a different version of VHDL file is required. The second version of VHDL is generated using generic VHDL. Therefore, it is technology independent and can be implemented onto any chosen CMOS process. A sample VHDL file for a 4-bit, 5-cycle delay glyph for the ASIC implementation is shown in Figure 4.12. The corresponding hardware architecture is shown in Figure 4.13. From Figure 4.13, it can be seen that the 5-cycle delay is implemented using five D flip-flops instead of using the delay component generated using CORE Generator.

Once these delay glyphs are created and installed in the CHAMPION library, they are inserted into the SFG using the CHAMPION tools, *Bufins*. *Bufins* inserts these delay glyphs in the SFG and then converts the SFG back to CHAMPION netlist. This completes the entire data synchronization process, which is shown in Figure 4.14.

```
-- Filename   : delay_4_5.vhd
-- Created by : Gendly
-- Date       : Mon Mar 12 17:22:38 2001

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity delay_4_5 is
generic(g_num_input_bits: integer := 4);

port (g_clk : in std_logic;
      i : in std_logic_vector( (g_num_input_bits - 1) downto 0);
      o : out std_logic_vector( (g_num_input_bits - 1) downto 0));
end delay_4_5;

architecture struct of delay_4_5 is

 attribute black_box:boolean;
 component dly_4_5
  port (din  : in std_logic_vector( (g_num_input_bits - 1) downto 0);
        c    : in std_logic;
        ce   : in std_logic;
        dout : out std_logic_vector( (g_num_input_bits - 1) downto 0));
 end component;
 attribute black_box of dly_4_5: component is true;

begin

 dly: dly_4_5
  port map (din  => i,
            c    => g_clk,
            ce   => '1',
            dout => o );
end struct;
```

Figure 4.10: Delay glyph using components from Xilinx Core Generator.

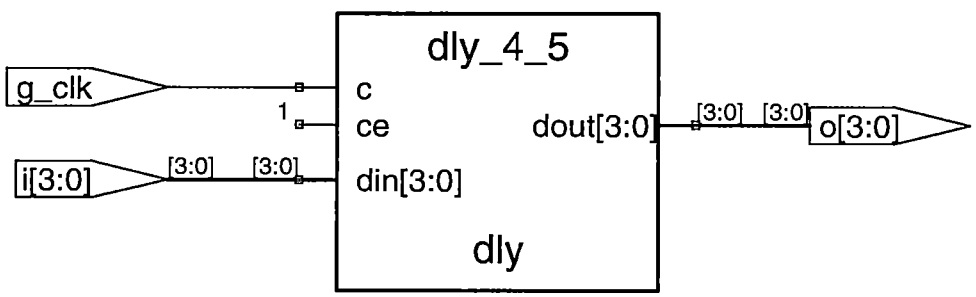


Figure 4.11: Hardware architecture of the VHDL file in Figure 4.10.

```
-- Filename     : delay_4_5.vhd
-- Generated by : Net2stv
-- Date         : Mon Mar 12 17:22:38 2001

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity delay_4_5 is
generic( g_num_input_bits : integer := 4 ;
         g_delay : integer := 4 ) ;

port   ( g_reset, g_clk : in std_logic;
          i : in std_logic_vector( (g_num_input_bits - 1) downto 0 );
         o : out std_logic_vector( (g_num_input_bits - 1) downto 0 )
       );
end delay_4_5 ;


architecture struct of delay_4_5 is

  type sarray is array(g_delay downto 0) of std_logic_vector( (g_num_input_bits-1)
downto 0);
  signal s:sarray;

begin
  process(g_clk)
  begin
    if(g_reset='1') then
      o<= "0000" ;
      for k in 0 to g_delay loop
        s(k) <= "0000" ;
      end loop ;
    elsif (g_clk'event and g_clk='1') then
      for k in 1 to g_delay loop
          s(k) <= s(k-1);
      end loop;
      s(0) <= i;
      o <= s(g_delay-1);
    end if;
  end process;

end struct;
```

Figure 4.12: Delay glyph for ASIC implementation.

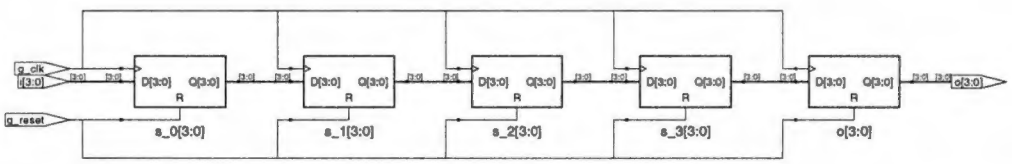


Figure 4.13: Hardware architecture of the VHDL file in Figure 4.12.

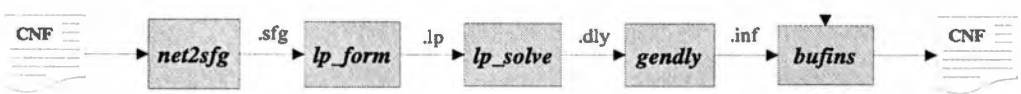


Figure 4.14: Data Synchronization process.

4.6 ACS Back-End Flow

The software tools for the ACS back-end flow was implemented by N. Kerkiz in [12]. This section provides brief descriptions of these tools. More information about the tools can be found in [12]

4.6.1 Partitioning

Three partitioning algorithms were implemented for the CHAMPION software design environment. The first algorithm is a hierarchical partitioning method based on topological ordering. The second algorithm is a recursive algorithm based on the Fiduccia and Mattheyses bipartitioning heuristic. The third algorithm is an extension of the first two algorithms to handle the constraints imposed by ACS. This new algorithm is a recursive partitioning method based on topological ordering and levelization. Details of the three partitioning algorithms implemented in CHAMPION can be found in [12].

4.6.2 Netlist to Structural VHDL, Synthesis, and Place & Route

A CHAMPION tool was also developed to translate the graph-based netlist format into structural VHDL after the partitioning process. The tool takes the resulting files and passes them through the synthesis tool, which adds the required I/O ports and merges the synthesized netlist with the pre-compiled VHDL components. Once the structural VHDL files have been synthesized, the resulting netlist files are placed and routed separately.

4.6.3 Host Program Generation

The final step in the CHAMPION design flow is the generation of the host program. A CHAMPION software tool was developed to generate the host program, which downloads the configuration file to the corresponding FPGA on the Wildforce board. The host program generated by the tool can only perform simple data transfer and movement. The host program uses the set of function calls provided by Annapolis Micro Systems to initialize the ACS board and download the programming bit files using these function calls. Once the programming bit files have been downloaded, the host program reads the input data from the host workstation, sends the data to the ACS and writes output back to the host workstation.

4.7 ASIC Back-end Flow

To implement the ASIC back-end flow, a set of software tools has been developed in this dissertation. This section describes the implementation of these tools.

4.7.1 Netlist to Structural VHDL

To implement the design in ASIC, the entire graph-based netlist format is first translated into a structural VHDL description of the netlist. A CHAMPION tool, *Net2stv*, has been developed to perform this translation. Net2stv translates the CHAMPION netlist into a single structural VHDL file, which can be implemented

as a single semiconductor chip.

4.7.2 Design Compilation and Optimization

Once the structural VHDL description of the design is obtained, the VHDL description can be synthesized and optimized using the CHAMPION tool, *Stvcomp*. *Stvcomp* generates the required command to synthesize and optimize the design using Synopsys Design Compiler. Both the top-down and bottom-up compilation strategies have been implemented in *Stvcomp*.

4.7.3 Physical Layout Generation

The final step in the back-end flow of ASIC is the generation of the physical layout. The CHAMPION tool, *Laygen*, automates the physical layout generation process. *Laygen* generates the required commands to perform both the top-down and bottom-up layout generation using the commercial software EPOCH.

CHAPTER 5

Experimental Results

Four applications have been used to verify the design flow of CHAMPION. A high-pass filter for image processing applications was first implemented in both ACS and ASIC using CHAMPION. To determine the improvement in mapping time for the ACS design flow, a two criteria automatic target recognition algorithm, called START, was used. This algorithm was automatically as well as manually mapped onto ACS. It serves as a benchmark for comparing the manual and automatic mapping time for the ACS design. Two other algorithms:

- the infrared automatic target recognition (IR ATR) algorithm from the Army Night Vision Lab (NVL) and
- the neural network-based face detection algorithm [27, 28] from the Carnegie Melon University (CMU)

were also implemented to test the ACS and ASIC design flow. Table 5.1 shows the sizes of the four applications in terms of glyphs and interconnections.

In this chapter, the results of the ACS and ASIC implementations of the four applications are presented. Section 5.1 shows the implementation results of the high-pass filter. Since the high-pass filter consists of a small set of glyphs and interconnections, the netlist of the high-pass filter produced by each CHAMPION

Table 5.1: Sizes of the four applications implemented.

Applications	Number of Glyphs	Number of Nets
High-Pass	14	42
IR ATR	45	71
Face Detection	58	118
START	93	226

tool is presented. Sections 5.2, 5.3 and 5.4 present the results of the IR ATR, Face Detection and START algorithms, respectively. Due to the sizes of the netlists, only the mapping time, execution time, resources reports and physical layouts of these three applications are presented.

5.1 High Pass Filter

5.1.1 Overview

High frequency components in an image correspond to edges and other sharp details in an image. To emphasize these fine details in an image, a high-pass filter can be used. The high-pass filter attenuates or eliminates low frequency components, which are responsible for the slowly varying characteristics in an image. Therefore, the net result of high-pass filtering is the reduction of low frequency details and a correspondingly apparent sharpening of edges and other

sharp details in the image. High-pass filters are often used to sharpen an image blurred by atmospheric seeing or poor focus.

In this dissertation, a basic 3×3 high-pass filter was implemented. For this filter, the value of a pixel in the output image depends on the value of the pixel in the same position in the input image and the eight pixels surrounding it (neighborhood of a pixel). For each pixel in the input image, the pixel and its neighboring pixels are multiplied by a set of coefficients called the mask. The output pixel value is then obtained by summing all the products and finding the absolute value of the sum.

The mask for the high-pass filter implemented in this dissertation is shown below.

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

To simplify the hardware implementation, $-\frac{1}{8}$ is used as the coefficients for the neighboring pixels, instead of $-\frac{1}{9}$ as in typical high-pass filter. The division by eight can be achieved by simply shifting the binary number 3-bit to the right. If $-\frac{1}{9}$ is used as the coefficients, a much more complicated circuit will have to be used to implement the division.

5.1.2 Implementation Results

In order to implement the high-pass filter, a set of CHAMPION glyphs was first developed. Both C/C++ and VHDL codes were generated for each glyph. The

C/C++ codes were installed in Cantata and the VHDL codes were synthesized and installed in the CHAMPION library using the CHAMPION tool, Geninf.

After developing all the required glyphs, the high-pass filter was implemented in Cantata by interconnecting the glyphs. Figure 5.1 shows the Cantata workspace for the high-pass filter. Simulations were performed in Cantata to verify that the desired functionality of the high-pass filter was achieved.

Once the Cantata workspace for the high-pass filter was developed and verified, the CHAMPION front-end flow was used to transform the Cantata program into a hardware netlist. The Cantata program was first translated into CHAMPION netlist (as shown in Figure 5.2), preserving the original glyphs and their interconnections. *Widthmatch* was then used to match all the data paths in the netlist. Two mismatched data paths were found in the netlist. In the first mismatched data path, an 8-bit output port is connected to an 11-bit input port. Therefore, a "padding" glyph labeled *pad_8_11* was automatically generated by *Widthmatch* and inserted into the netlist to fix the data path. This glyph appends three zeros to the 8-bit output port so that the output port has the same bit width as the port connected to it. In the second mismatched data path, a 12-bit output port is connected to an 8-bit input port. Therefore, a "truncating" glyph is required to fix this mismatch. *Widthmatch* generated a glyph labeled *truncate_12_8* and inserted it into the netlist. This glyph truncates four bits from the 12-bit output port so that the output port has the same bit width as the port connected to it. Figure 5.3 shows the netlist after the data width matching

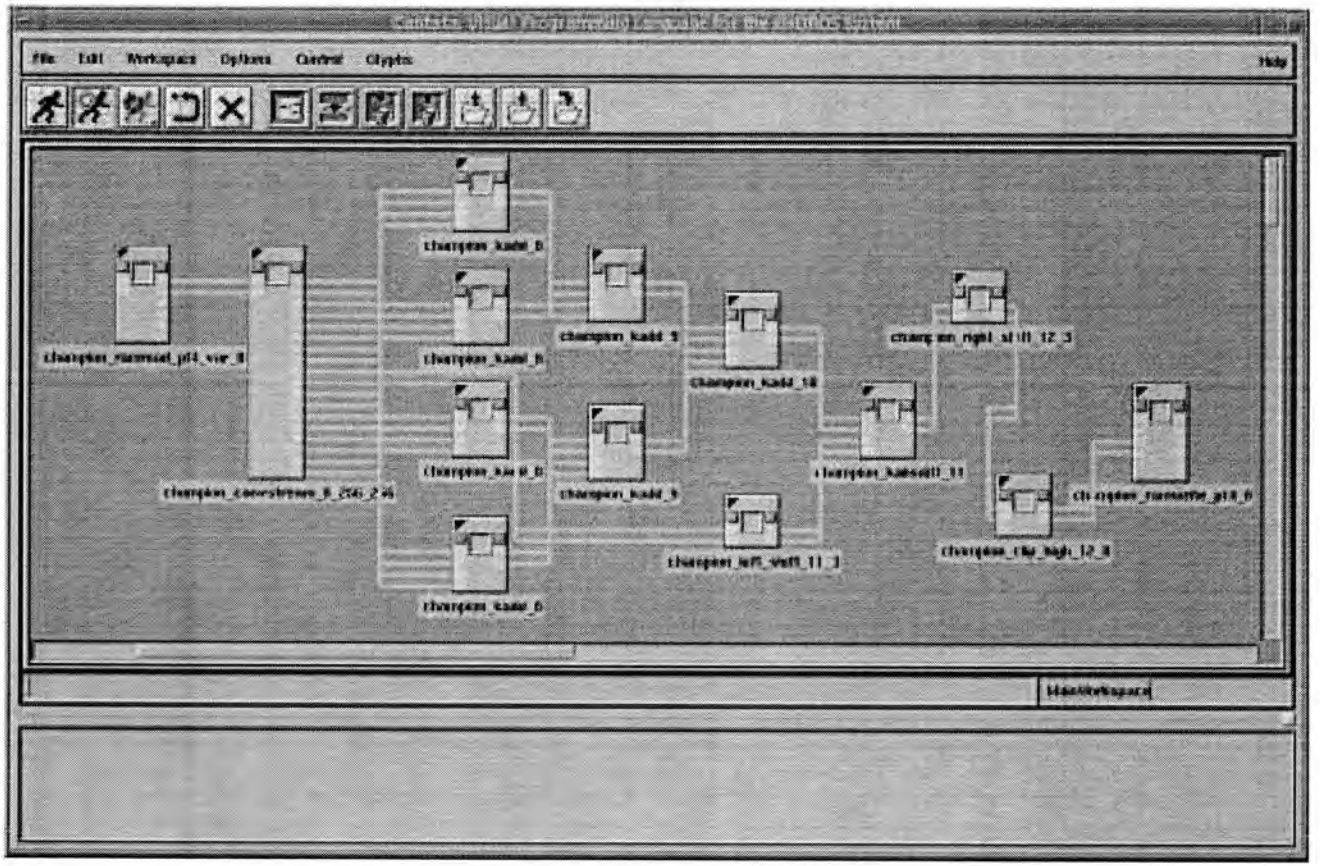


Figure 5.1: Cantata Workspace for the high-pass filter.

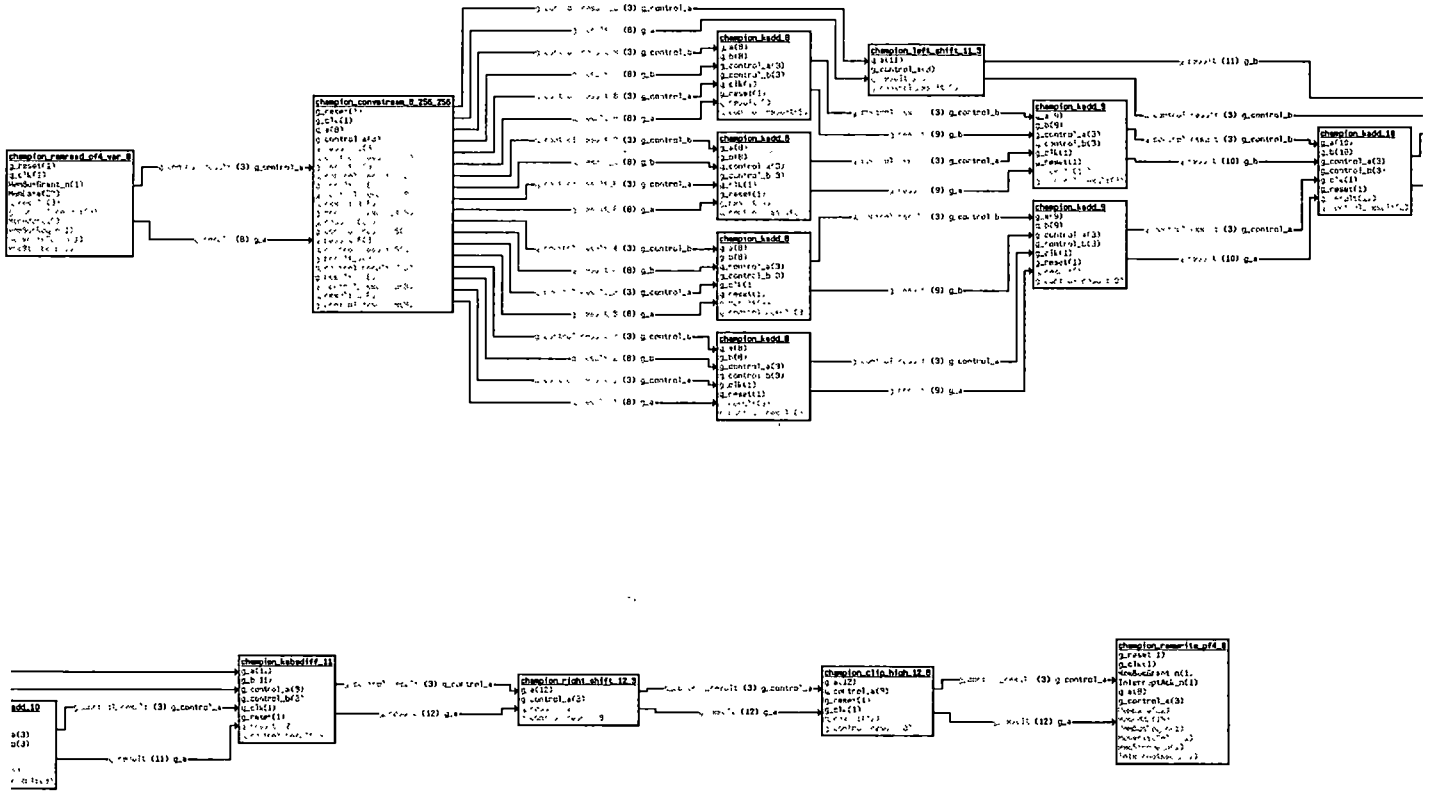


Figure 5.2: High-pass filter in CHAMPION netlist.

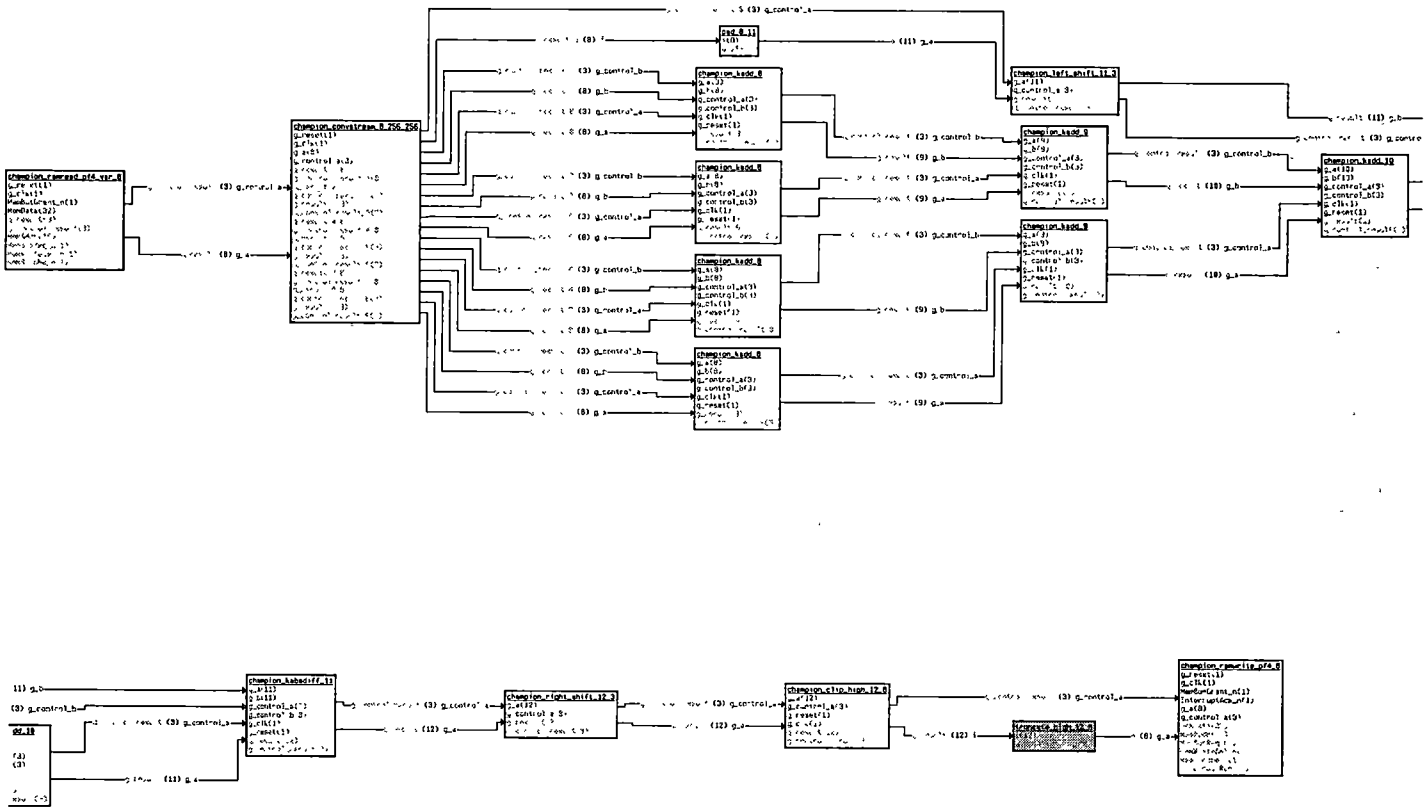


Figure 5.3: High-pass filter in after data width matching.

process.

After matching the width of all the interconnections, data synchronization was performed. The result obtained from the data synchronization is shown in Figure 5.4. Two delay buffers were introduced into the netlist. The CHAMPION synchronization tools generated the VHDL files for an 8-bit, 3-cycle delay glyph and a 3-bit, 3-cycle delay glyph. These two VHDL files were automatically synthesized and inserted into the netlist. After examining all the data paths in the netlist obtained after the synchronization process, it can be found that all the data paths have a latency of 520 clock cycles. Since all the data paths have the same latency, the netlist is thus synchronized.

To map the synchronized design onto the Wildforce board, ACS back-end flow was used. The netlist was partitioned into two sub-netlists as shown in Figures 5.5 and 5.6 . The entire design can actually fit into one single FPGA in the Wildforce board. However, due to the existence of two RAM modules in the netlist, the design has to be partitioned into two sub-netlists. As Figure 5.6 shows, the second partition contains only one glyph, that is, the glyph for writing the result to the RAM. The resources used by the partitions are listed in Table 5.2. Table 5.2 shows that the total number of glyphs and nets of the two partitions are more than the values shown in Table 5.1. This is due to the fact that four new glyphs have been inserted into the netlist during the data width matching and data synchronization processes.

After partitioning, the CHAMPION netlist was translated into a structural

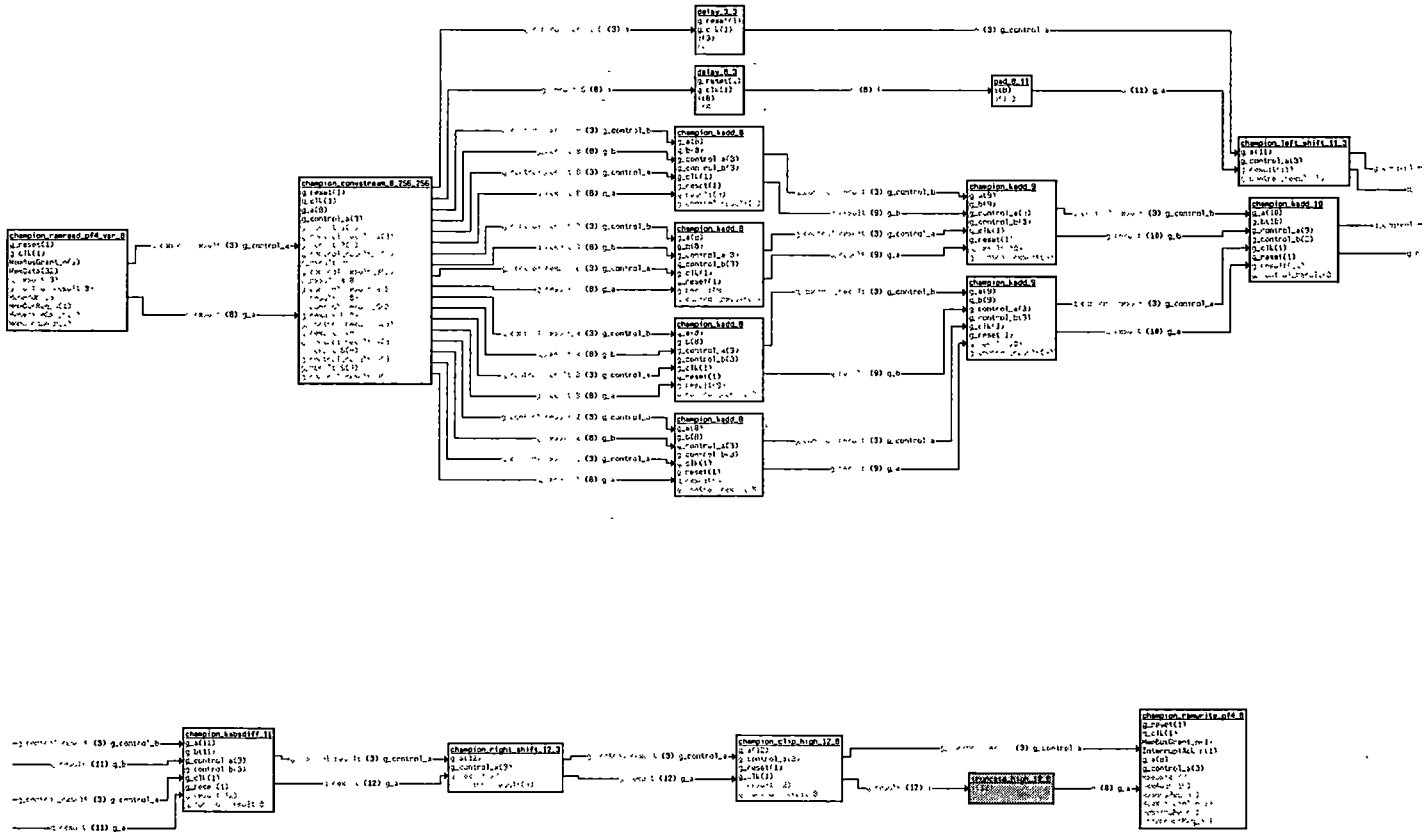


Figure 5.4: High-pass filter after data synchronization.

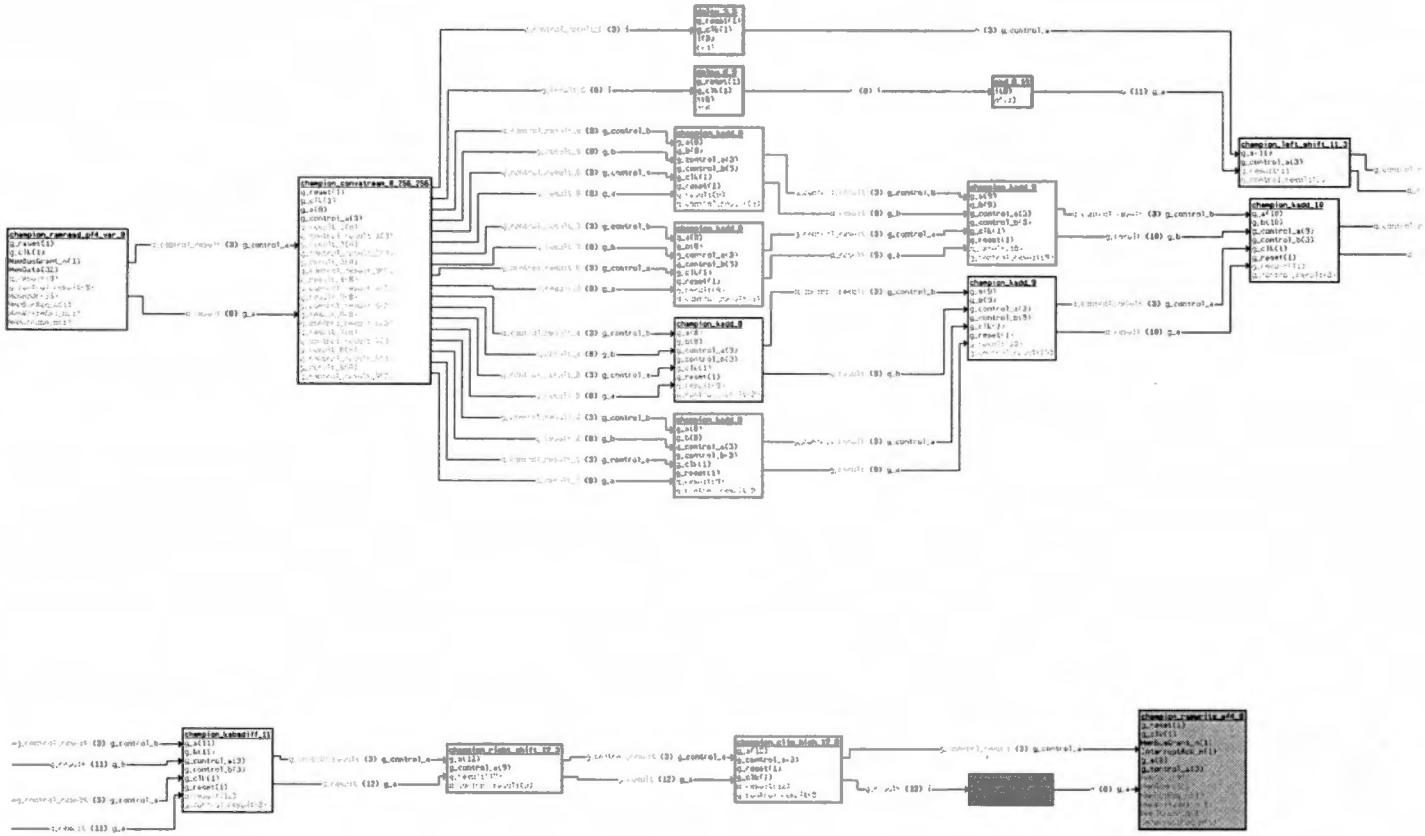


Figure 5.5: First partition of high-pass filter.

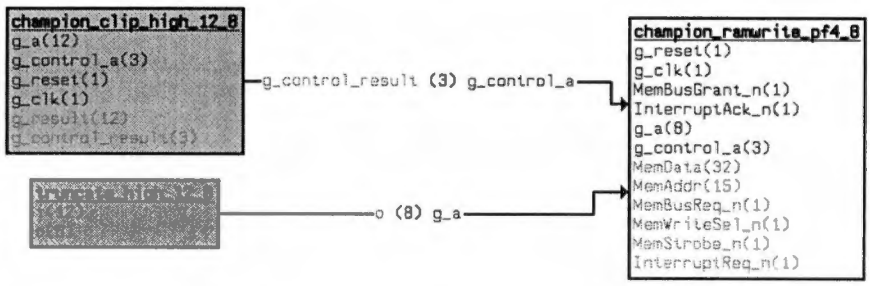


Figure 5.6: Second partition of the high-pass filter.

Table 5.2: Partitions of high-pass filter.

Partition	Number of Glyphs	Number of Nets	Number of RAM Glyphs	Number of CLB Used
1	17	46	1	476 of 1296 (36%)
2	1	2	1	75 of 576 (13%)

VHDL representation. The required I/O ports for each of the sub-netlists were then added to the VHDL files and synthesized using the Synplify tool. Xilinx M1 tools were then used to merge the synthesized netlist with the pre-compiled VHDL components corresponding to the glyphs, and to place and route the netlist. A host program was automatically generated to download the resulting configuration files to the Xilinx FPGAs on the Wildforce board.

Table 5.3 shows the time required to map the Cantata workspace for the high-pass filter onto the Wildforce board. The entire mapping process took 39 minutes and 18 seconds. From Table 5.3, it can be seen that most of the mapping time (96.73%) was spent on the placement and routing, which were performed using the Xilinx software. The CHAMPION tools were able to perform data width matching, data synchronization, partitioning and netlist conversions in approximately 77 seconds.

Table 5.4 compares the execution time of the high-pass filter in Cantata and on the Wildforce board. A 256×256 image was used during the execution. The Wildforce execution is about 4 times faster than the Cantata execution. The execution time on the Wildforce can be broken down into board configuration time, data transfer time and the actual hardware execution time. The time to process an image is greatly dominated by the time needed to configure the board. The actual time to process one image is only 3 milliseconds, which is only 0.08% of the total execution time. If the configuration time could be eliminated, the hardware implementation would be over 4100 times faster than the Cantata

Table 5.3: ACS mapping time of high-pass filter.

Mapping Tools	Mapping Time (seconds)	Percentage
Wsp2net	1	0.04%
Widthmatch	38	1.61%
Datasync	35	1.48%
Partition	2	0.08%
Net2stv	1	0.04%
Syn, P & R	2281	96.73%
Total	2358	100%

Table 5.4: Execution time of high-pass filter.

	Cantata (seconds)	Wildforce (seconds)
Board Configuration	—	1.049
Data Transfer	1.9	2.669
Data Processing	12.3	0.003
Total	14.2	3.721

implementation.

The high-pass filter was also mapped to ASIC using the HP26G technology. The ASIC back-end flow was used to first translate the synchronized netlist into a VHDL representation. The VHDL representation of the design was then compiled using both the top-down and bottom-up approaches. Physical layouts for the design compiled using both approaches were then generated. The physical layout for the top-down approach is shown in Figure 5.7. From the figure, it can be seen that the entire design is flattened and no signs of glyphs can be observed. The entire mapping process using the top-down approach took 6 hours 40 minutes and 15 seconds. The size of the core generated using this approach measures 13.7 mm^2.

The physical layout for the bottom-up approach is shown in Figure 5.8. Figure 5.8 shows that the layout is composed of blocks of a smaller layout. These

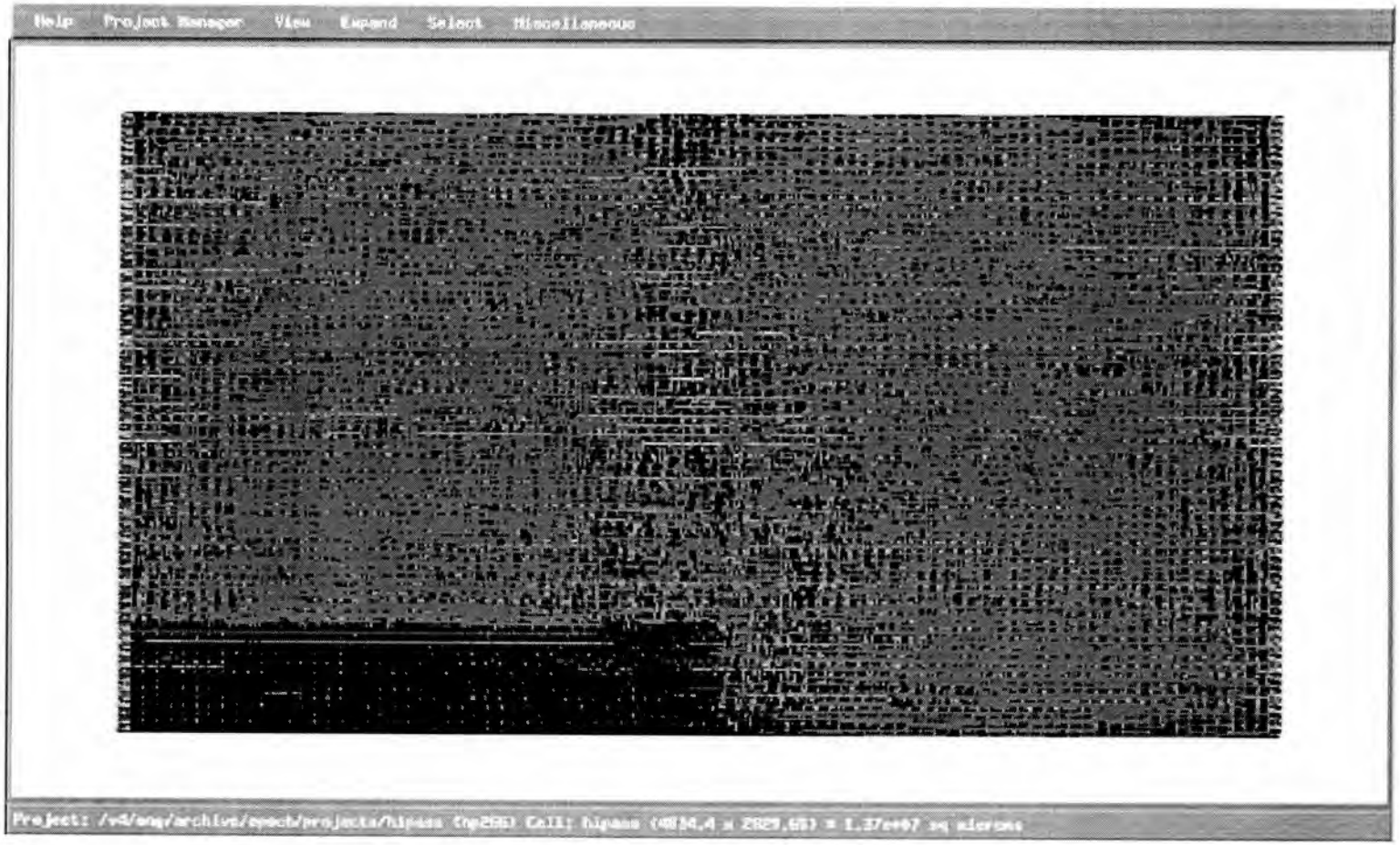


Figure 5.7: The core of the high-pass filter generated using the top-down approach.

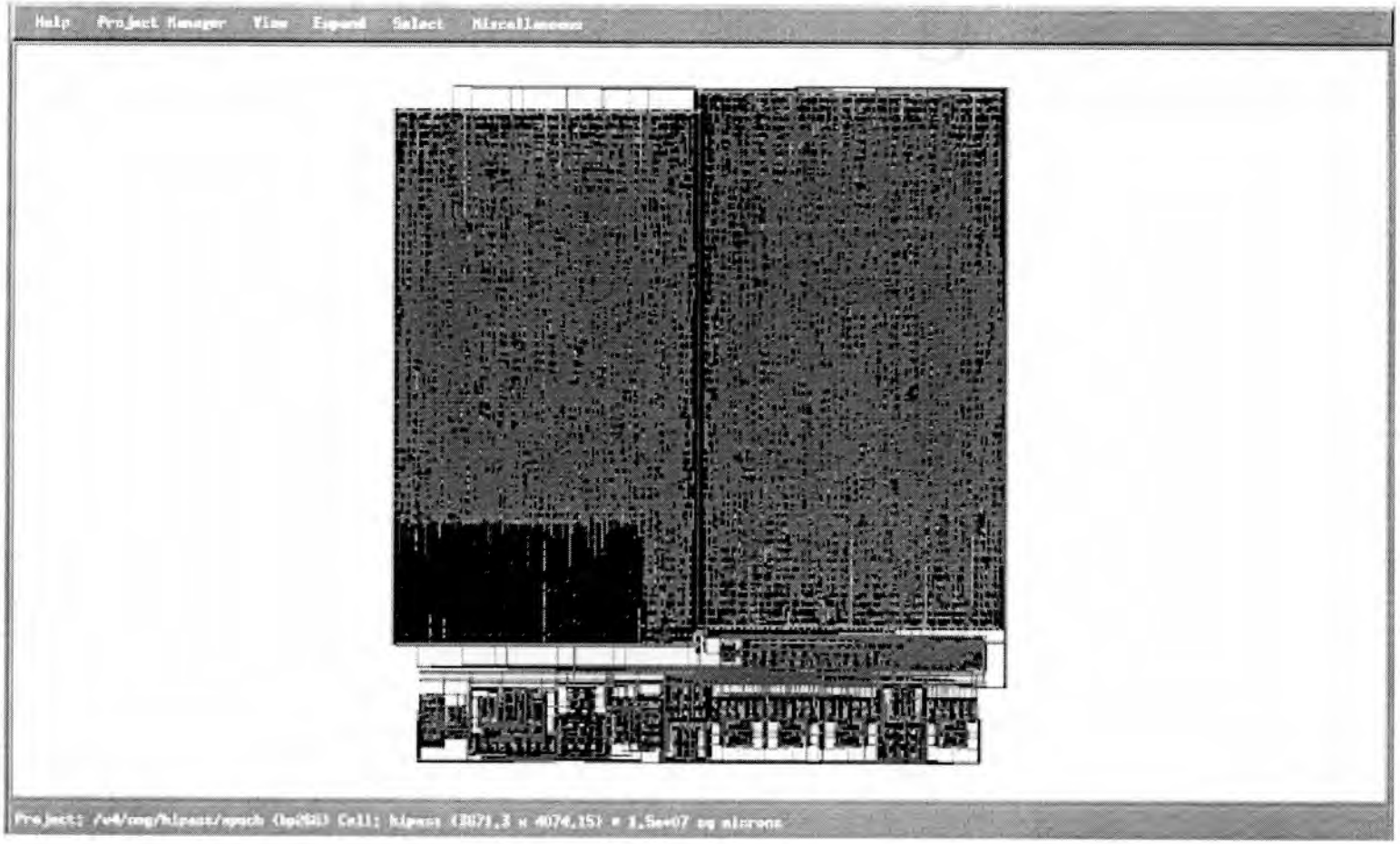


Figure 5.8: The core of the high-pass filter generated using the bottom-up approach.

Table 5.5: ASIC implementation results of the high-pass filter.

	Core Area (mm^2)	Mapping Time (seconds)
Top-down	13.7	24015
Bottom-up	15.0	2958
Difference	1.3	21057

blocks of layout are the glyphs in the design. The entire mapping process using the bottom-up approach took 49 minutes and 18 seconds. The size of the core generated using this approach is 15.0 mm^2.

A comparison between the results generated using the two approaches is shown in Table 5.5. Compared to the core generated using the top-down approach, the core generated using the bottom-up approach is 1.32 mm^2 or 9.5% larger. For the core generated using the bottom-up approach, approximately 85% of the layout area is used for one single glyph (*champion_convstream_8_8_256* in Figure 5.2). Not much area is wasted due to the placement problems. As a result, the size of the layout generated using the bottom-up approach is compatible to that of the top-down approach. However, the bottom-up approach only took about 1/8 of the mapping time required by the top-down approach.

5.2 IR ATR algorithm from the Army Night Vision Laboratory

5.2.1 Overview of the Algorithm

One of the key problems in military applications is the ability to locate and identify military targets such as tanks, trucks, and other mobile weaponry. The collection capacity of the military imagery systems is far higher than the rate at which humans can visually review the images. Over the years, many algorithms have been developed to automatically locate and identify military targets. These algorithms, which are known as Automatic Target Recognition (ATR) algorithms, require both high throughput rate and low latency. Therefore, they are best implemented in pipelined hardware such as those produced by CHAMPION.

The ATR algorithm implemented in CHAMPION is used to locate and identify ground vehicles in a single infrared image frame. It is one of the research challenge problems identified by the ACS program of the U.S. Defense Advanced Research Projects Agency (DARPA). The analysis in this ATR algorithm is based on techniques known as template matching. The algorithm consists of many conceptually simple steps for matching an image area and a template pair of target and background. Decision tree is used in the algorithm to improve the computational efficiency.

Figure 5.9 gives an overview of the algorithm. The algorithm is composed of five major steps, named Round 0 to 5. In Round 0, six template pairs are used to process the entire 480 x 640 input image to identify and locate the regions

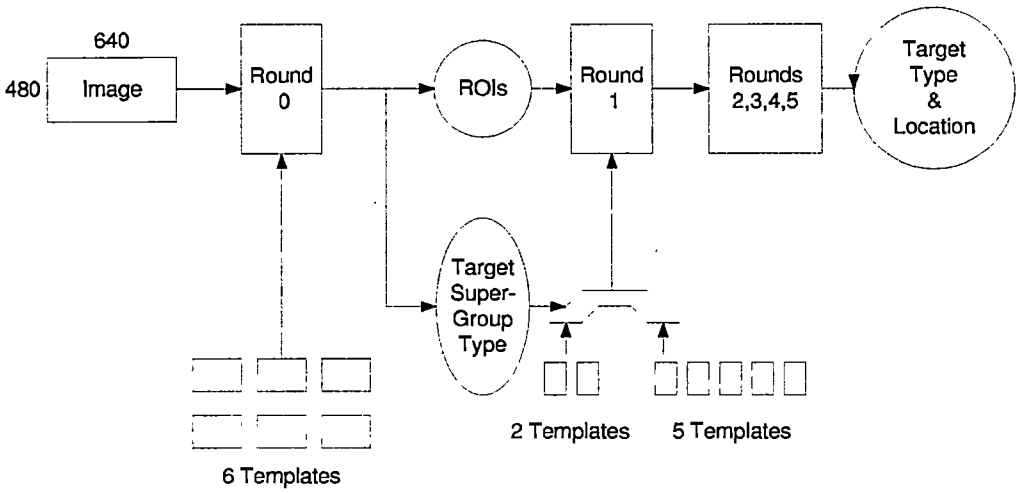


Figure 5.9: The IR ATR algorithm.

of interests (ROIs). An ROI is composed of image pixels whose surrounding area meets a certain criterion. These image pixels are candidates for further investigation in Round 1 to 5. In a typical infrared image, no more than 20% of the image pixels become ROIs [10].

All the ROIs identified in Round 0 are passed to Round 1 for further hypothesis testing and classification into target groups. Based on which target group an ROI belongs to, either two or five templates are used to process these ROIs. If the ROIs pass the hypothesis testing in Round 1, further testing are performed on these ROIs in the remaining 4 rounds. The output of the algorithm is the pixels which pass the testing in all 6 rounds, and their identified target types. Because of the pruning process, Round 0 is the most computational intensive step [10]. Therefore, Round 0 was targeted for hardware acceleration using the

Wildforce board and ASIC. Details of Round 0 can be found in [10].

5.2.2 Implementation Results

To implement Round 0 of the IR ATR algorithm, another new set of CHAMPION glyphs was developed. For each new glyph, both C/C++ and VHDL codes were generated and installed in Cantata and the precompiled library respectively. Once all the required glyphs were developed, a Cantata workspace for Round 0 was developed. The Cantata workspace was executed to verify its functionality.

After developing and verifying the workspace, it was mapped onto the Wildforce board using CHAMPION. The CHAMPION partitioning tool partitioned the design into two sub-netlists as shown in Table 5.6. As in the case of the high-pass filter, the entire Round 0 design could actually fit into one single FPGA in the Wildforce board. However, due to the existence of two RAM modules in the netlist, the design had to be partitioned into two sub-netlists. The second sub-netlist contains only one glyph, that is, the glyph for writing the result to RAM.

Table 5.7 shows the time required to map the Round 0 workspace onto the Wildforce board. The entire mapping process took 57 minutes and 8 seconds. From Table 5.7, it can be seen that most of the mapping time (99.82%) was spent on the placement and routing processes, which were performed using the Xilinx software. The CHAMPION tools were able to perform data width matching, data synchronization, partitioning and netlist conversions in approximately 6 seconds.

Table 5.6: Partitions of Round 0.

Partition	Number of Glyphs	Number of Nets	Number of RAM Glyphs	Number of CLB Used
1	44	71	1	920 of 1296 (70%)
2	1	2	1	47 of 576 (8%)

Table 5.7: ACS mapping time of Round 0.

Mapping Tools	Mapping Time (seconds)	Percentage
Wsp2net	1	0.03%
Widthmatch	1	0.03%
Datasync	2	0.06%
Partition	1	0.03%
Net2stv	1	0.03%
Syn, P & R	3422	99.82%
Total	3428	100%

Table 5.8: Execution time of Round 0.

	Cantata (seconds)	Wildforce (seconds)
Board Configuration	—	0.807
Data Transfer	72	153.128
Data Processing	15604	0.269
Total	15676	154.20

Table 5.8 compares the execution time of Round 0 in Cantata and on the Wildforce board. A 640 × 480 image was used during the execution. The Wildforce execution is about 100 times faster than the Cantata execution. The time to process an image is greatly dominated by the data transfer time. The actual time to process one image is only 0.2694 seconds, which is only 0.17% of the total execution time. Comparing the data processing time in Cantata and Wildforce, the Wildforce implementation is about 58,000 times faster then the Cantata implementation.

Round 0 was also mapped to ASIC using the HP26G technology. The physical layouts for the design were generated using both the top-down and bottom-up approaches. The physical layout for the top-down approach is shown in Figure 5.10. The entire mapping process using the top-down approach took 1 hour 17 minutes and 27 seconds. The size of the core generated using this approach measures

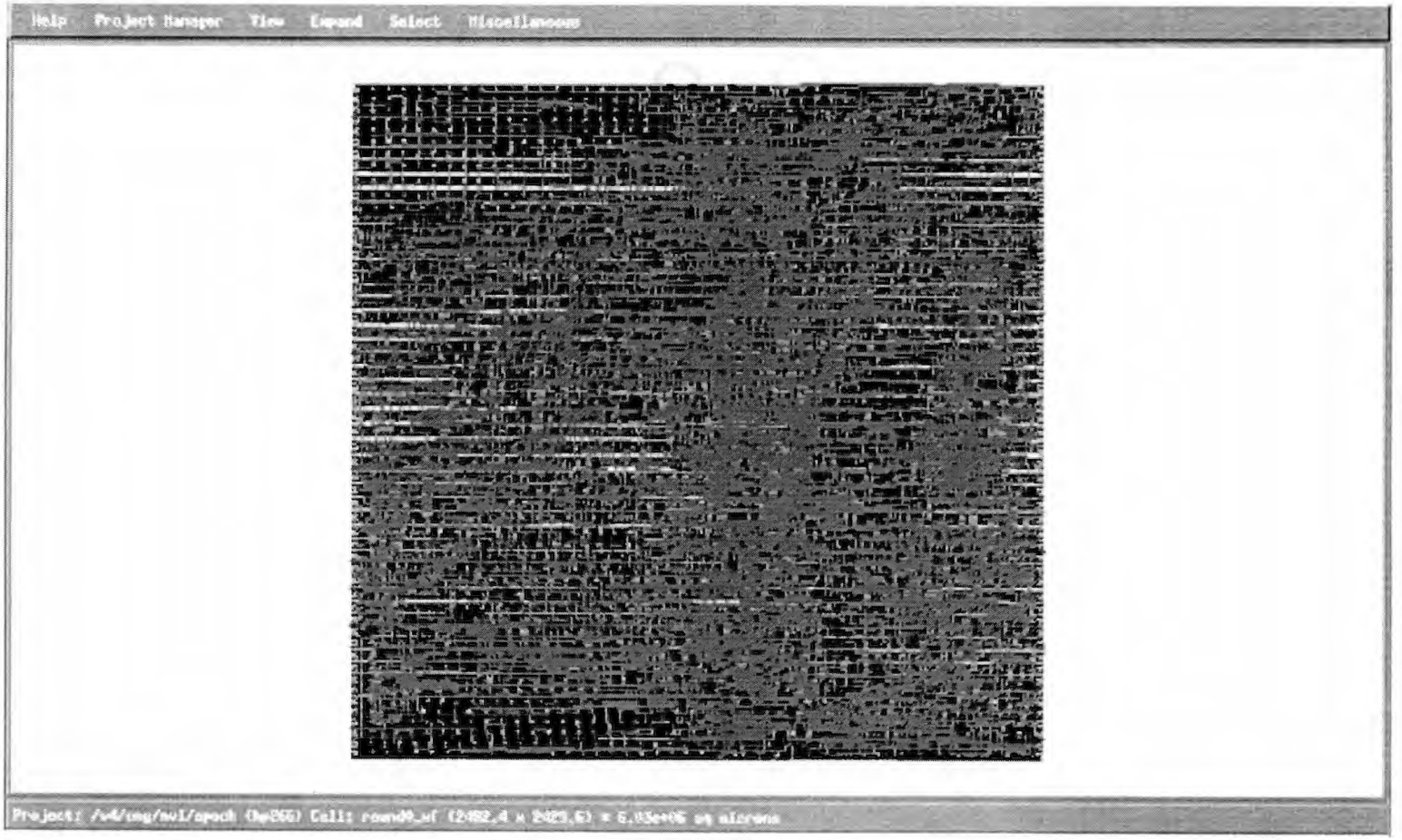


Figure 5.10: The core of the Round 0 generated using the top-down approach.

Table 5.9: ASIC implementation results of Round 0.

	Core Area (mm^2)	Mapping Time (seconds)
Top-down	6.0	4647
Bottom-up	12.1	990
Difference	6.1	3657

6.0 mm^2. Figure 5.11 shows the physical layout generated using the bottom-up approach. For this approach, the entire mapping process took 16 minutes and 30 seconds. The size of the core generated using this approach is 12.1 mm^2.

A comparison between the results generated using the two approaches is shown in Table 5.9 Compared to the core generated using the top-down approach, the core generated using the bottom-up approach is double in size. The large increase in size is due mainly to the placement of the glyphs and the area used for routing. The bottom-up approach took about 1/5 of the mapping time required by the top-down approach.

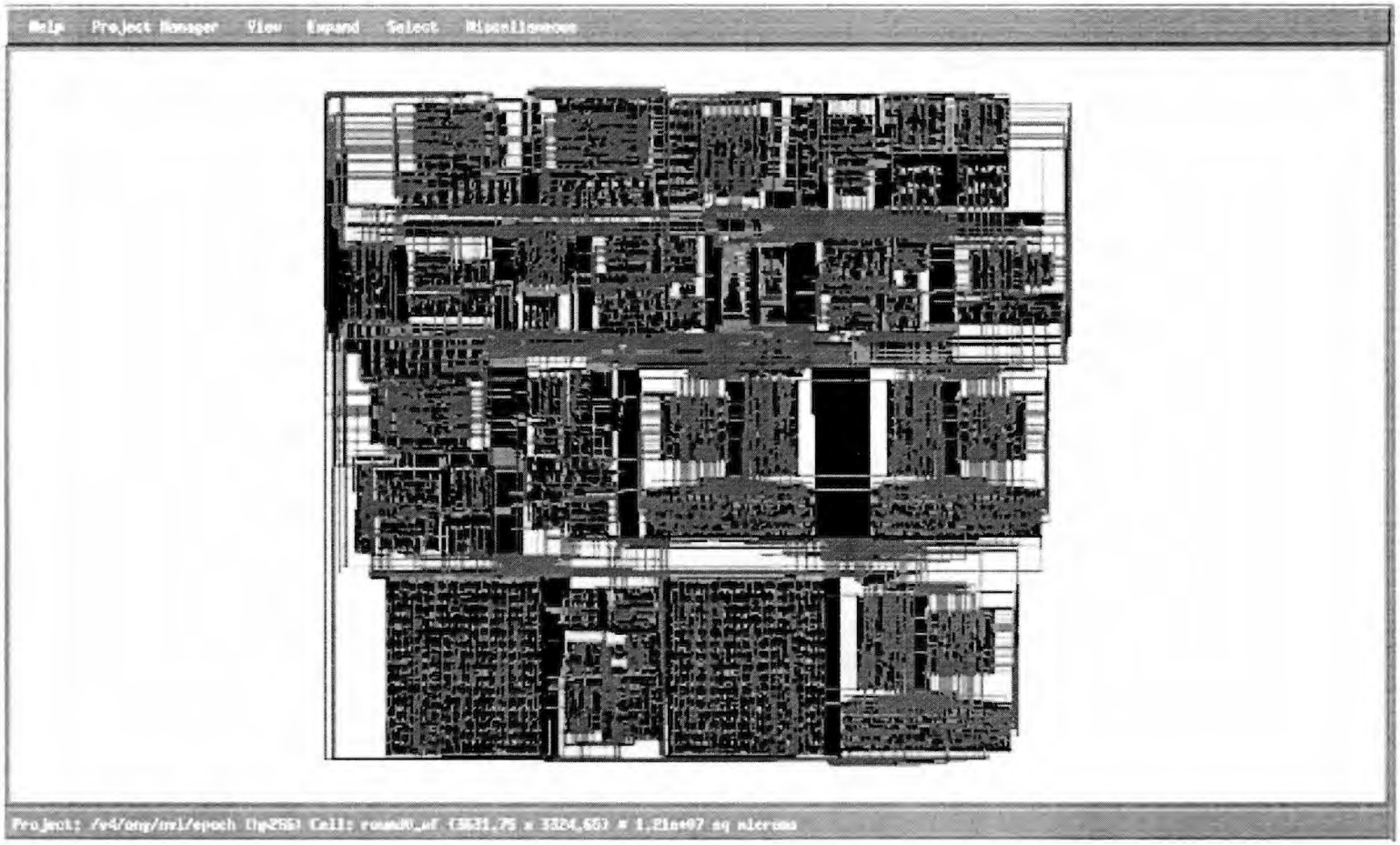


Figure 5.11: The core of the Round 0 generated using the bottom-up approach.

5.3 Face Detection Algorithm

5.3.1 Overview of the Algorithm

The third application implemented in CHAMPION is the neural networks used in the face detection algorithm in [27]. The basic algorithm used for face detection is shown in Figure 5.12. To detect faces in different sizes, the input image is repeatedly reduced in size by subsampling. This image pyramid is shown on the left of Figure 5.12. Each 20×20 pixel window of each level of the pyramid is then preprocessed with standard algorithms such as histogram equalization and lighting correction to improve the overall brightness and contrast in the images. The preprocessed window is then broken down into smaller pieces of four 10×10 pixel regions, sixteen 5×5 pixel regions, and six overlapping 20×5 pixel regions. These pixel regions are passed to three neural networks, which generate single, real-valued outputs, signifying the presence or absence of a face. The output from each network is then merged using an arbitrator to eliminate overlapping detections.

In this research project, CHAMPION was used to map the three neural networks in Figure 5.12 to hardware. The preprocessing steps such as size reduction, histogram equalization, and lighting correction are done in the host. The three neural networks are named Umec, Face17c and Face18c.

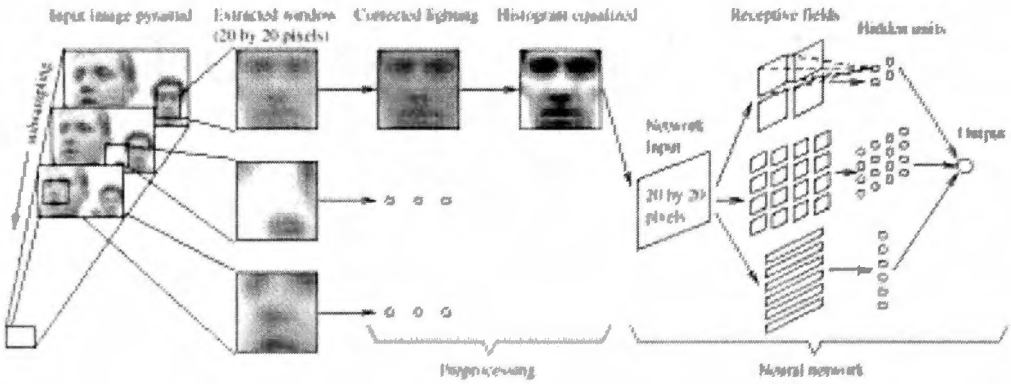


Figure 5.12: The basic algorithm used for face detection [27].

5.3.2 Implementation Results

To implement the three neural networks, a new set of CHAMPION glyphs was developed. Three Cantata workspaces for the three neural networks were developed using the existing CHAMPION glyphs (developed for the high-pass and IR ATR applications) and the newly developed glyphs. The Cantata workspaces were simulated and their functionalities were verified.

Once the Cantata workspaces were developed and verified, CHAMPION tools were used to map the workspaces onto the Wildforce board. During the mapping process, it was found that the Umec network could not be partitioned for the FPGAs on the Wildforce board. No partition with I/O connections of less than 32 bits can be found. Since the interconnection between each FPGA in the Wildforce board is 32-bit, Umec could not be implemented onto the Wildforce board. To solve this problem, the Umec network was manually partitioned into

Table 5.10: ACS mapping time of Round 0.

Networks	Mapping Time (Seconds)					
	Wsp2net	Widthmatch	Datasync	Partition	Net2stv	S,P&R
Layer 1	0	1	1	1	1	1684
Layer 2	0	0	1	1	1	735
Face17c	0	1	1	1	1	1442
Face18c	1	0	1	1	1	1538

two smaller networks called Umec Layer 1 and Umec Layer 2. In these two networks, some of the interconnections were multiplexed to reduce the I/O.

All four neural networks were successfully mapped onto the Wildforce board using CHAMPION. Table 5.10 lists the time for mapping the four workspaces onto the Wildforce board.

The neural networks were also mapped to HP26G ASIC using CHAMPION. The physical layouts for the design were generated using both the top-down and bottom-up approaches. For Umec Layer 1, the physical layout generated using the top-down approach is shown in Figure 5.13. The entire mapping process using the top-down approach took 28 minutes and 8 seconds. The size of the core generated using this approach measures 5.88 mm^2. The physical layout of Umec Layer 1 generated using the bottom-up approach is shown in Figure 5.14. The entire mapping process using the bottom-up approach took 12 minutes and

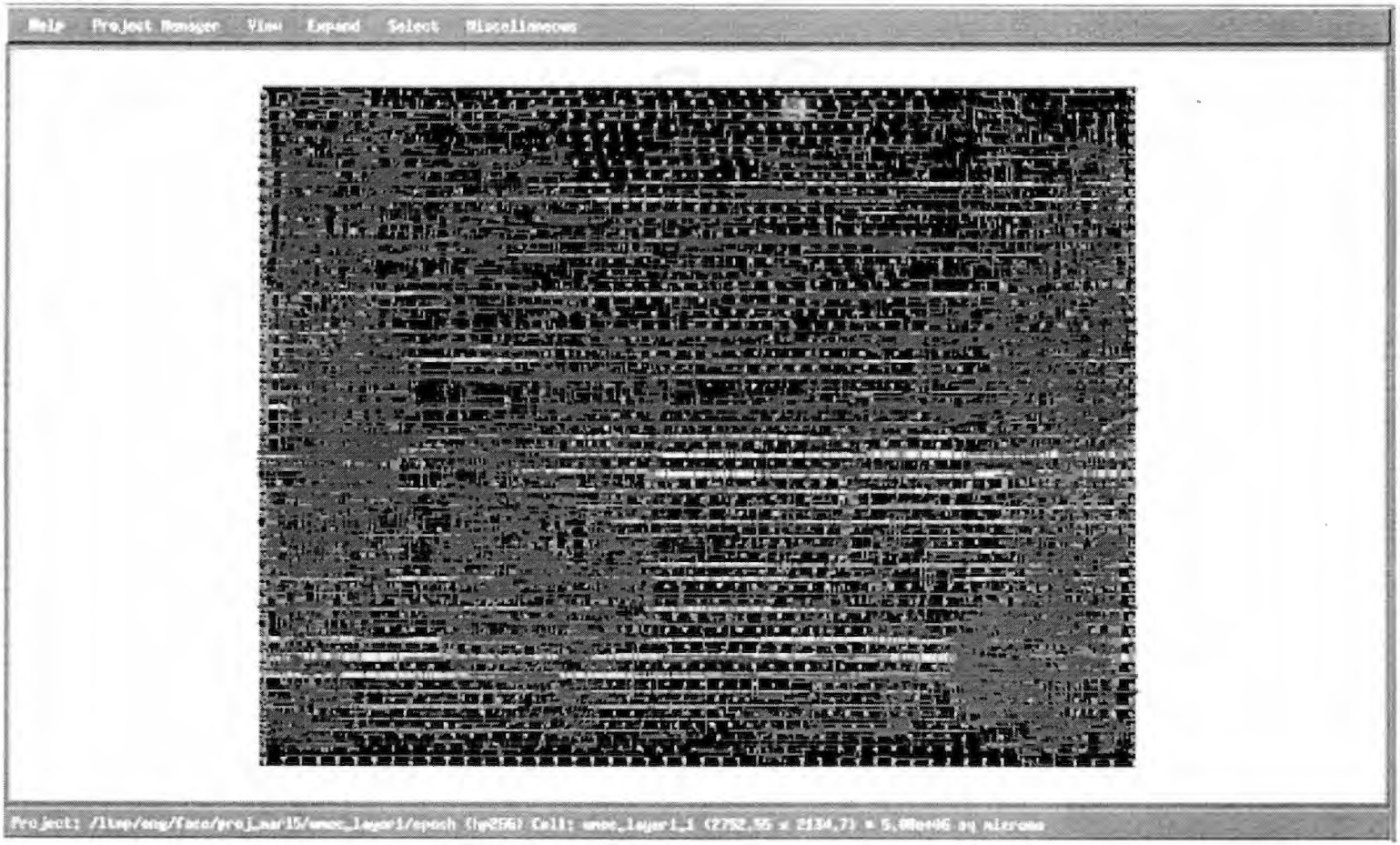


Figure 5.13: The core of Umec Layer 1 generated using the top-down approach.

Figure 5.14: The core of Umec Layer 1 generated using the bottom-up approach.

Table 5.11: ASIC implementation results of the neural networks.

Network	Mapping Time (Seconds)		Core Area (mm^2)	
	Top-down	Bottom-up	Top-down	Bottom-up
Layer 1	4819	4211	5.88	7.69
Layer 2	1909	2351	3.43	6.33
Face17c	30818	6027	11.7	14.5
Face18c	35448	6674	12.6	15.3

18 seconds. The size of the core generated using this approach is 7.69 mm^2.

Figures 5.15 to 5.20 show the top-down and bottom-up layouts of Umec Layer 2, Face17c and Face18c, respectively. The corresponding mapping time for these layouts are shown in Table 5.11. As Table 5.11 shows, for Umec Layer 2, the mapping time for the top-down approach is actually shorter than the mapping time for the bottom-up approach. The size of the core generated using the top-down approach is also much smaller than the bottom-up approach. In other words, the top-down approach produced a much smaller layout in a shorter time compared to the bottom-up approach.

A hypothesis for why this happens is as follows: For the bottom-up approach, the layout of each glyph was first generated. Therefore, the dimension and shape of the layout of the individual glyph are fixed. During the placement of these glyphs, blocks are positioned on a layout surface in such a way that no two

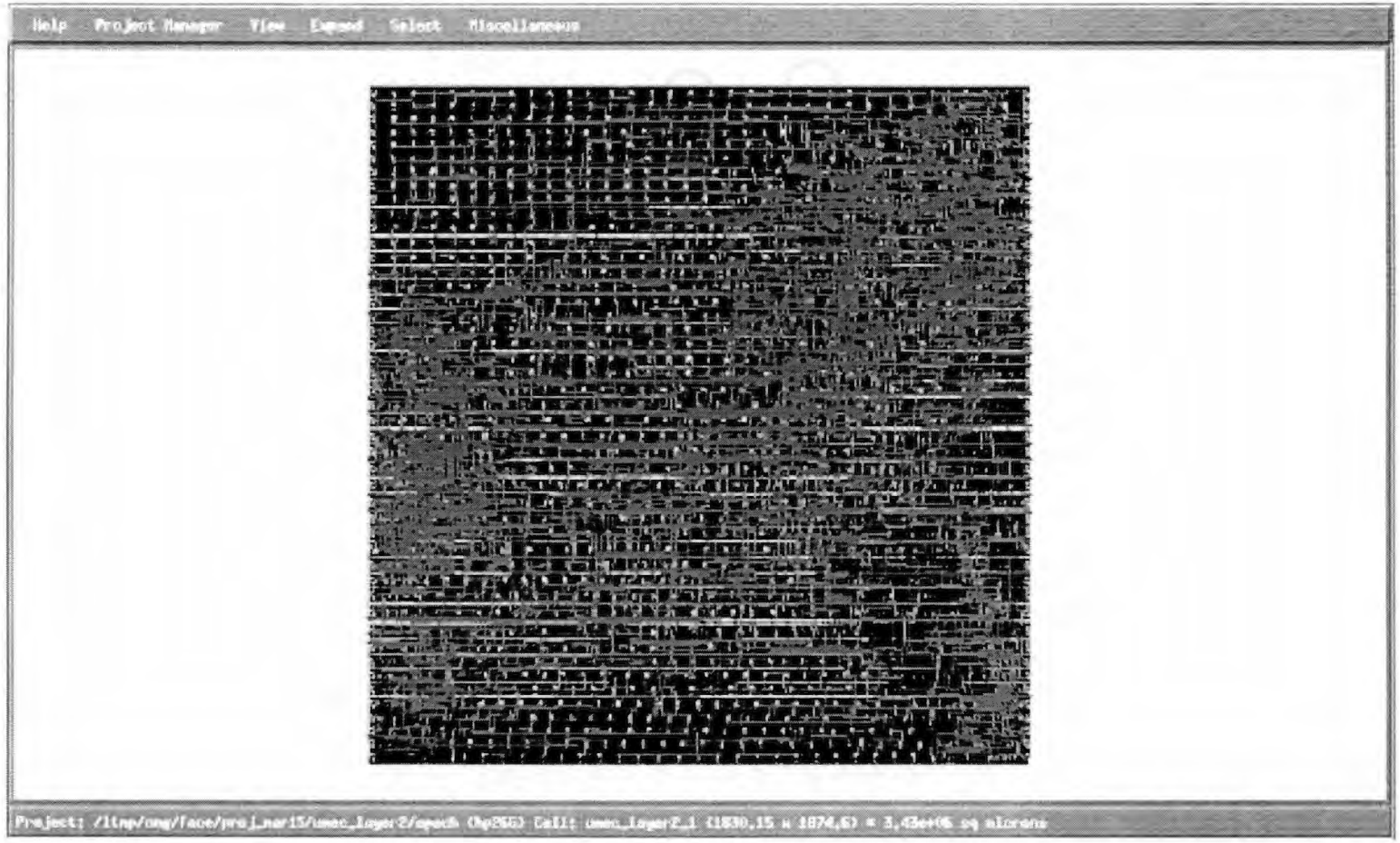


Figure 5.15: The core of Umec Layer 2 generated using the top-down approach.

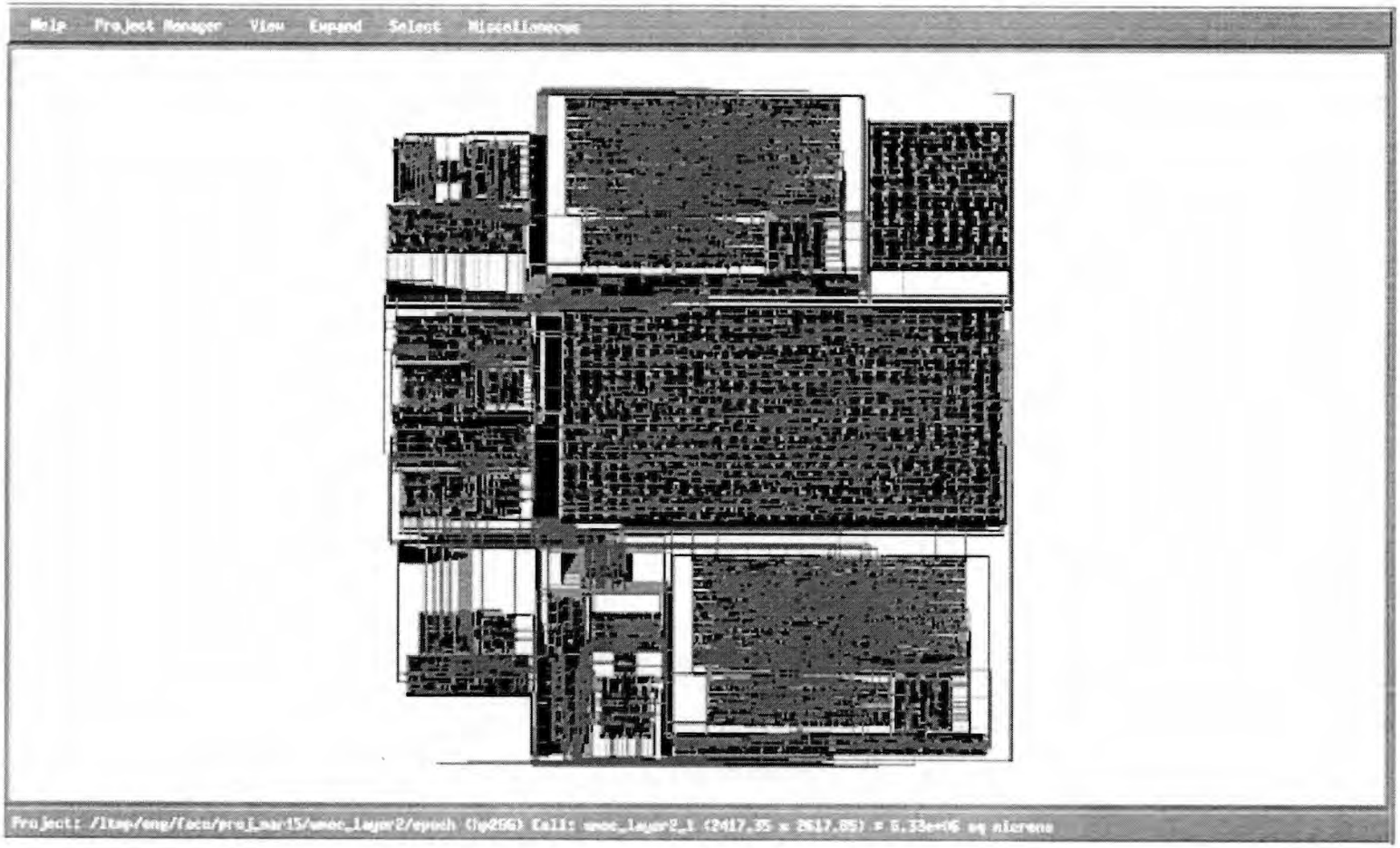


Figure 5.16: The core of Umec Layer 2 generated using the bottom-up approach

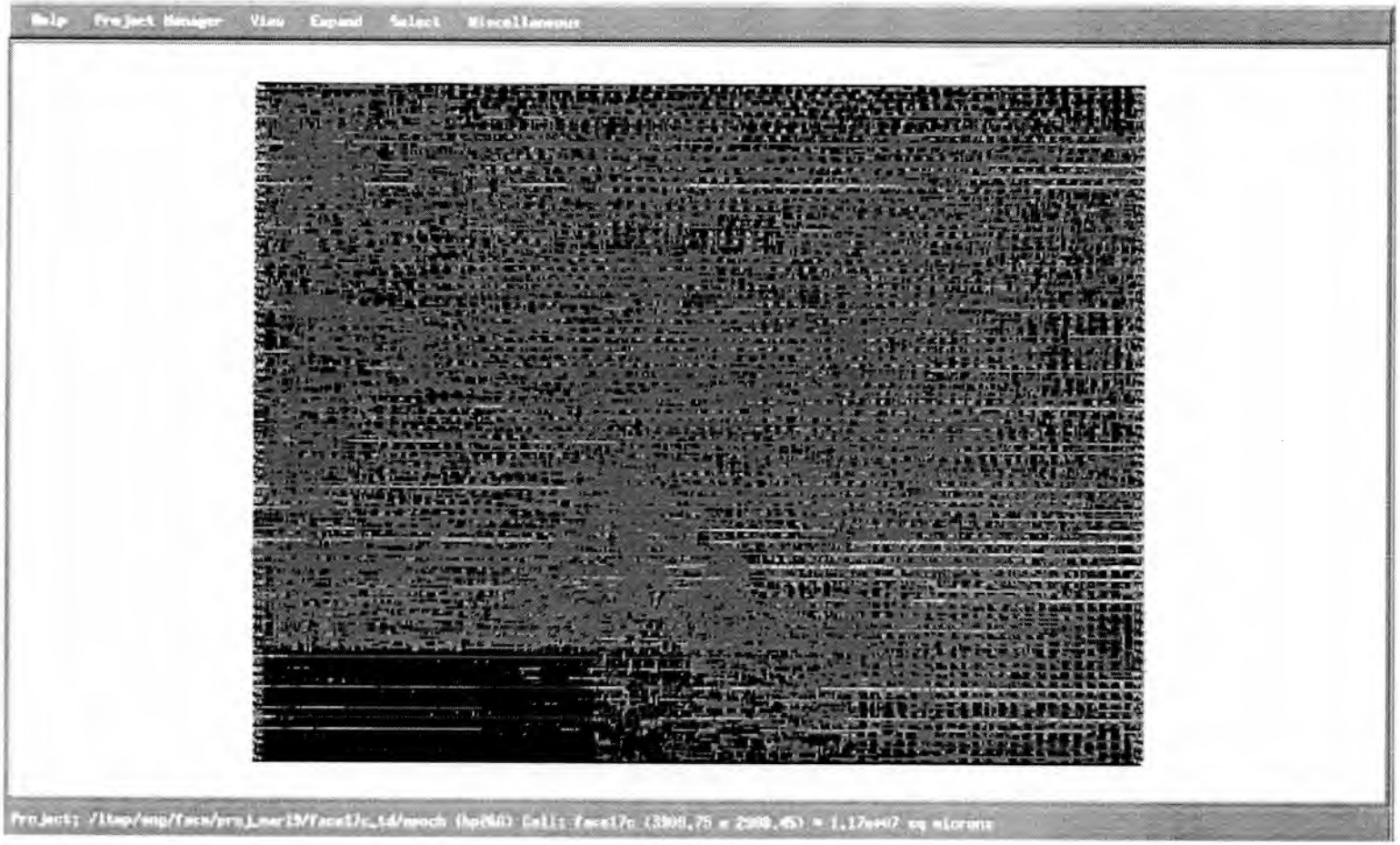


Figure 5.17: The core of Face17c generated using the top-down approach.

Figure 5.18: The core of Face17c generated using the bottom-up approach.

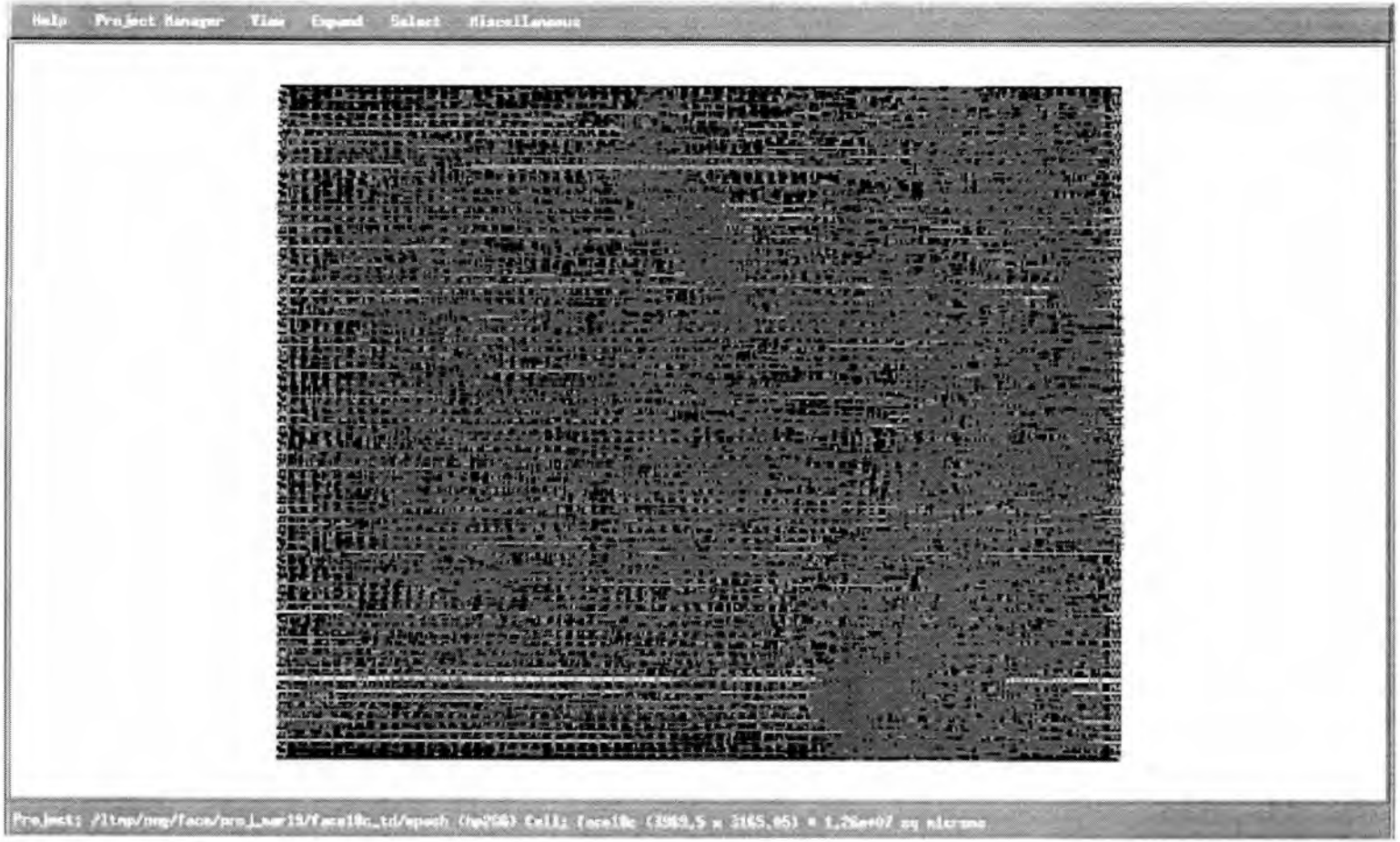


Figure 5.19: The core of Face18c generated using the top-down approach.

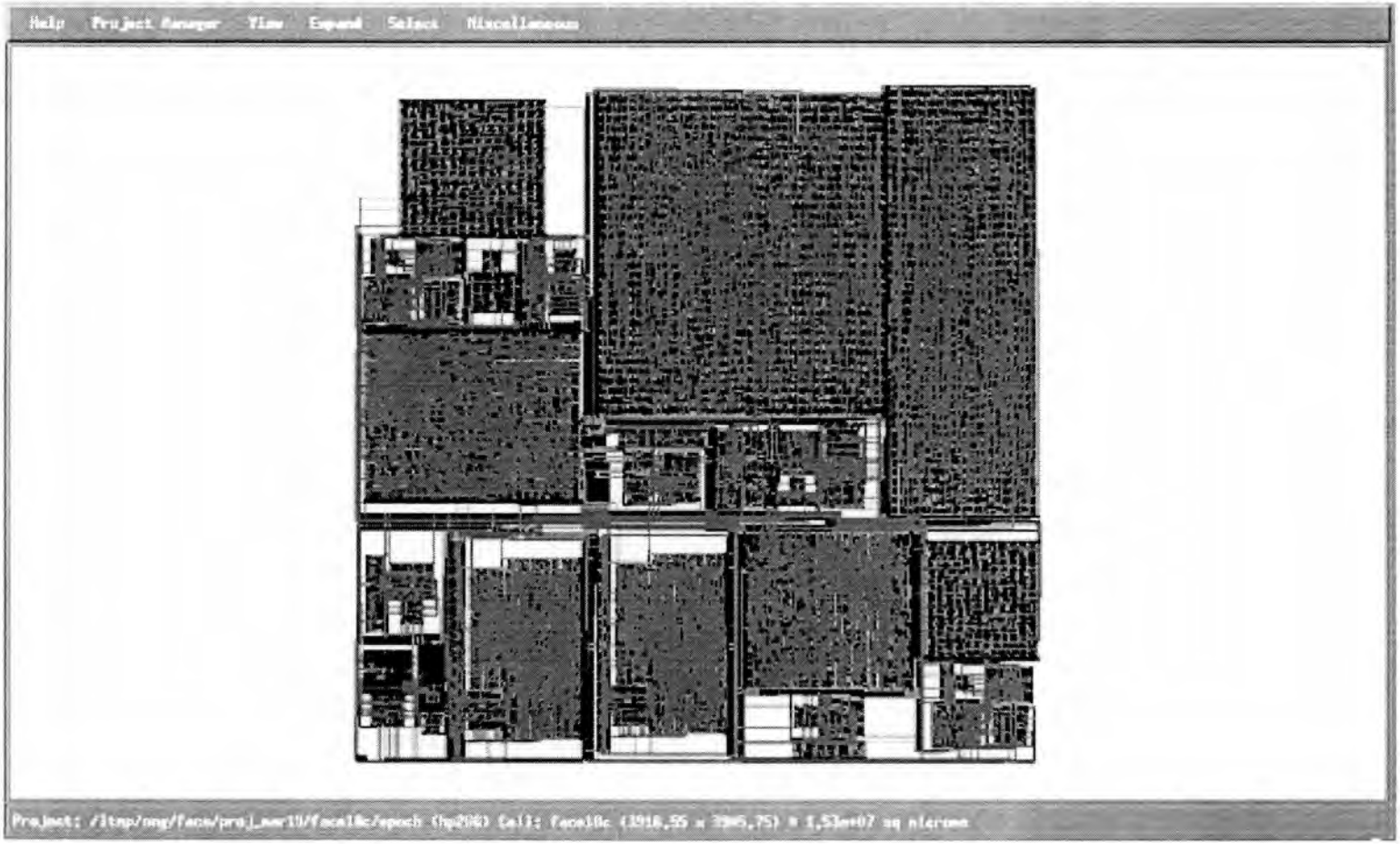


Figure 5.20: The core of Face18c generated using the bottom-up approach.

blocks are overlapping and the total area of the layout is minimized. For Umec Layer 2, the glyphs are shaped in such a way that there is no "good" optimum arrangement that can be achieved. As depicted in Figure 5.16, the glyphs cannot possibly be arranged in a way that no excessive empty space can be found. As a result, the bottom-up approach produced a layout with a much larger area. The long placement time might be due to the fact that the placement tool took many iterations to achieve a placement with a layout area which is below a certain threshold value. This threshold value is set at the beginning of the placement process. It may be set to be a certain percentage larger than the sum of the area of the individual glyphs. It is possible that the placement tool was never able to produce a layout with an area smaller than the threshold. In this case, the resulting placement is the best placement that the tool achieved once the tool reached its maximum number of iterations. Thus, the bottom-up approach took a longer time to generate the layout of the design.

For Face 17c and Face 18c, the layouts generated using the bottom-up approach are less than 25% larger than the layouts generated using the top-down approach. The large increase in size is due mainly to the placement of the glyphs and the area used for routing. The bottom-up approach took about 1/5 of the time required by the top-down approach to complete the mapping.

5.4 START Algorithm

5.4.1 Overview of the Algorithm

The fourth algorithm was implemented mainly to serve as a benchmark for measuring the CHAMPION performance improvement in mapping time. An ATR application developed by B. Levine [17] was chosen. The algorithm was named Simple, Two-criterion, Automatic Recognition of Targets, or START. The START algorithm uses a statistical approach to search Forward-Looking InfraRed (FLIR) images for regions where a target may exist. The START algorithm does not positively locate a target, nor does it identify the type of target. It is used to cue human analysts to regions of interest, reducing the time required for them to review each image. It identifies regions where there is a high probability that a target is present.

The regions of interest identified by the START algorithm have two characteristics. First, they must be hotter than the surrounding. In a FLIR image, hotter pixels are brighter. Therefore, a region of interest must contain pixels which are brighter. The second characteristic of a region of interest is that it exhibits sharp temperature gradients. This is due to the fact that military vehicles are likely to exhibit sharp temperature changes, either between the vehicle and the surrounding terrain, or between different components of the vehicle, such as the exhaust system and the chassis. As a result, to qualify as a region of interest, they must contain bright pixels and pixels that exhibit large temperature gradients. More

details of the START algorithm can be found in [17].

5.4.2 Implementation Results

As described earlier, START algorithm was implemented mainly to serve as a benchmark for determining the improvement in mapping time for the ACS flow. It was first manually mapped to the Wildforce board in [17]. It took over 250 hours to complete the manual mapping, not including the time spent for developing the required glyphs and performing the synthesis and placement/routing of the design. That is, the 250 hours were required for performing the data width matching, data synchronization, partitioning and generation of the structural VHDL netlist.

To measure the improvement in mapping time of CHAMPION, CHAMPION was used to map the START algorithm. The 250-hour manual process was performed automatically by CHAMPION in 5 minutes and 23 seconds, demonstrating a productivity improvement of over 2,000 times.

The execution time of START algorithm in Cantata and on the Wildforce board are shown in Table 5.12. Table 5.12 shows that the manual Wildforce implementation has a faster execution time than the Wildforce implementation achieved using CHAMPION. The main difference between the two implementations is in the time required to reconfigure the board. During the manual mapping performed in [17], the START algorithm is partitioned in a way that some of the partitions can be reused in the next configuration. Therefore, the reconfiguration

Table 5.12: Execution time of START [12].

	Cantata (seconds)	Wildforce (Manual) (seconds)	Wildforce (CHAMPION) (seconds)
Board Configuration	—	5.125	9.147
Data Movement	3.4	0.583	0.735
Data Processing	1050.6	0.010	0.010
Total	1054	5.718	9.892

time is reduced. However, in the automatic mapping, the partitioning tool did not perform partitioning in such a manner. As a result, the reconfiguration time for the CHAMPION implementation is higher. However, compared to the Cantata implementation, the execution time of the CHAMPION implementation is 184 times faster.

Effort was also spent on mapping the START algorithm onto HP26G layout. However, due to the size of the design, the commercial tool, Epoch, was not able to route the layouts generated using both the top-down and bottom-up approach. The physical layouts for the top-down and bottom-up approaches are shown in Figures 5.21 and 5.22 respectively. These figures show the layout generated after the placement process. Since Epoch was unable to route both layouts, no routing can be observed in the layout. The mapping process using

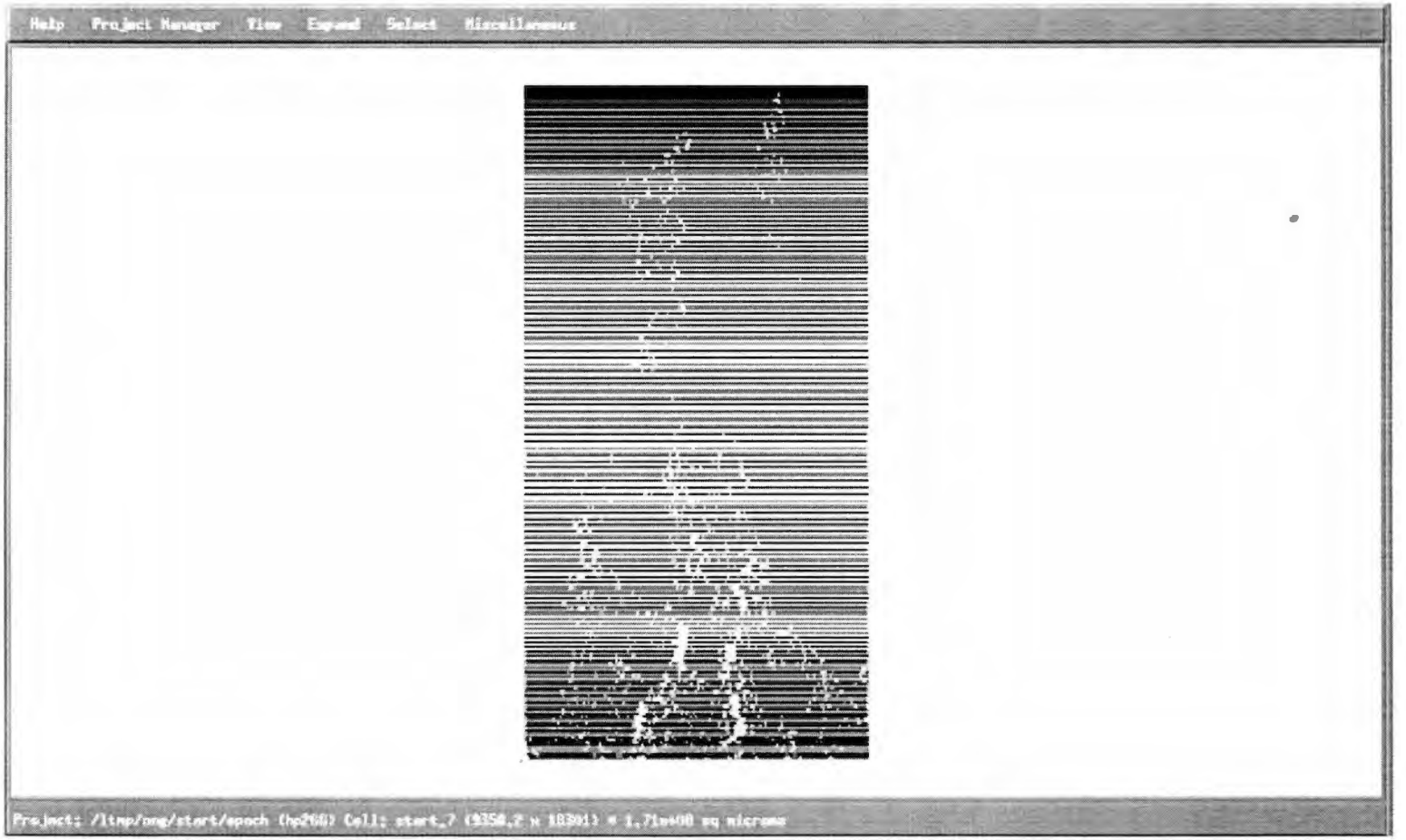


Figure 5.21: The core of START generated using the top-down approach.

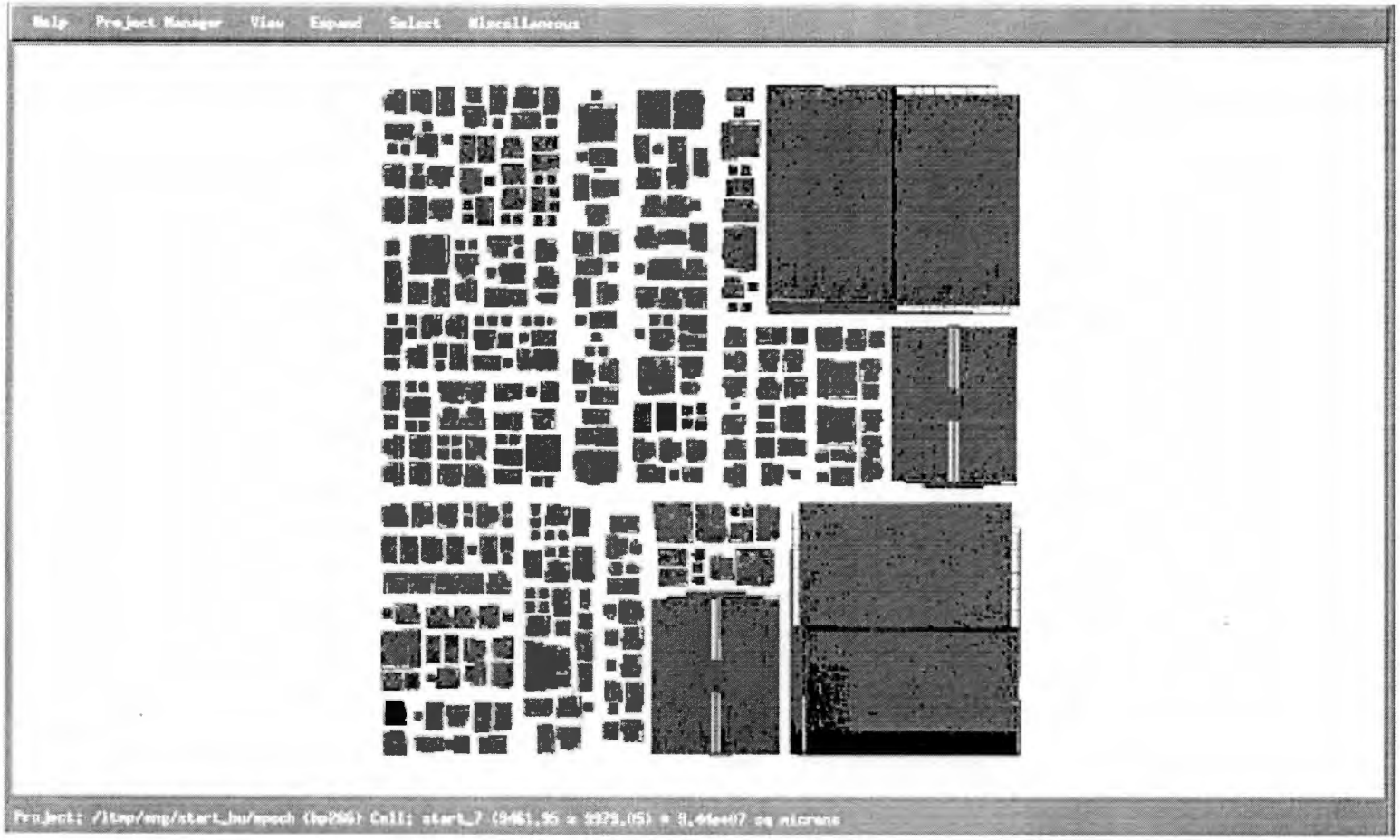


Figure 5.22: The core of START generated using the bottom-up approach.

Table 5.13: ASIC implementation results of the START algorithm.

	Core Area (mm^2)	Mapping Time (seconds)
Top-down	6.0	4647
Bottom-up	12.1	990
Difference	6.1	3657

the top-down approach took 33 hours 48 minutes and 17 seconds. The size of the core (without routing) generated using this approach measures 171.10 mm^2. For the bottom-up approach, the mapping time was 14 hours 15 minutes and 15 seconds. The size of the core without routing is 94.41 mm^2. The results for the ASIC implementation are summarized in Table 5.13.

CHAPTER 6

Summary and Future Work

With advances in microelectronic technology, complex applications, which are traditionally implemented in software, can now be implemented in hardware such as ACS and ASIC. As a result, hardware can serve as flexible accelerators for the time-consuming and computationally intensive applications. However, the lack of supportive design environments results in an unacceptably long turn-around time for leveraging the benefits of hardware technology. To significantly reduce the turn-around time, it is necessary to develop the mapping tools that allow the designers to capture an application faster as well as reducing the time necessary to move from specification to hardware implementation.

This objective has been achieved by the CHAMPION software design environment, which provides automatic mapping of applications in the Cantata graphical programming environment to ACS and ASIC. CHAMPION is a software design environment developed by the Microelectronic Systems Research Laboratory at the University of Tennessee. It is a design environment intended for mapping Cantata applications onto ASIC and ACS in multiple platforms. Three ACS platforms are used in the CHAMPION research projects. They are the Wildforce board and Wildcard developed by AMS [1] and the SLAAC board [3] developed

by the University of Southern California.

This dissertation developed and implemented most parts of the software design environment for mapping Cantata applications onto Wildforce board and ASIC in HP26G technology. The main contributions of this dissertation includes:

- detailing the design flow of the CHAMPION software design environment,
- developing the graphical user interface for CHAMPION,
- revising all CHAMPION glyphs for implementation in both Wildforce and ASIC,
- developing and implementing the glyph development flow,
- developing and implementing the front-end flow,
- developing and implementing the ASIC back-end flow, and
- verifying CHAMPION using various applications.

In general, this dissertation created an easier and faster way for application programmers to develop their applications for ACS and ASIC. A design flow was developed and implemented to automatically map Cantata applications onto Wildforce board and ASIC in HP26G technology. The primary strength of the design flow developed in this dissertation includes:

- allowing the functionality to be captured faster and more accurately using precompiled functions,

- demonstrating a productivity improvement of 2000x over manual methods (5 minutes vs. 250 hours),
- producing synchronous circuit by synchronizing the design using delay buffers,
- using linear programming to minimize the number of delay buffers used for synchronizing the design,
- partitioning netlist at glyph-level instead of gate-level to shorten the hardware mapping time, and
- allowing a new ACS architecture to be adopted to the design flow easily.

It is particularly significant that the design flow can easily adopt a new ACS board. To adopt a new ACS board to the design flow, the changes that need to be made are:

- replacement of the Xilinx placement and routing tools with the placement and routing tools for the FPGA on the new ACS, and
- generation of new host programs for interface between the host system and the ACS.

As a result, one possibility of extending this dissertation is to include other ACS architectures, which is being performed during the writing of this dissertation. To retarget a previously captured application to the new ACS, a high-level description of the new board, which is composed of information such as the number

of FPGAs in the ACS, the FPGA sizes and I/O, the RAMs available on the ACS, can be used to guide the CHAMPION tools such as the glyph installation and partitioning tools during the mapping process. With this retargeting compatibility, application designers can determine the ACS architecture that best matches the application.

This retargeting capability also allows an application designer to exploit the rapid advances in FPGA offerings. For example, as soon as a new ACS board is announced, the board-specific data file can be generated and the mapping/partitioning performed by CHAMPION. Upon arrival of the new board, the design can be downloaded and executed. The common situation of waiting months for the arrival of a new board until the application can be manually retargeted would be avoided.

Another possibility of extending this dissertation is to permit the design flow to accept inputs from other graphical programming environments such as LabVIEW from National Instruments and/or Simulink from MathWorks. To include these programming environments, the only component in the design flow that needs to be modified is the front-end translator that converts the Cantata workspace into a CHAMPION netlist. The rest of the steps in the design flow will remain the same. With this extension, CHAMPION can be utilized by a wider audience and more software applications can be implemented in ACS and ASIC in less time.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] *Annapolis Micro Systems.* http://www.annapmicro.com.

[2] *Khoros Pro User's Guide.* Khoral Research Inc., Albuquerque, NM.

[3] *Systems Level Applications of Adaptive Computing (SLAAC).* http://www.east.isi.edu/projects/SLAAC.

[4] *A|RT Library User's and Reference Documentation.* Frontier Design Inc., September 1999.

[5] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems. In *Int. Symp. on FPGA Custom Computing Machines*, Napa Valley, CA, April 2000.

[6] B. Draper, W. Najjar, W. Bohm, J. Hammes, R. Rinker, C. Ross, M. Chawathe, and J. Bins. Compiling and Optimizing Image Processing Algorithms for FPGA's. In *Int. Workshop on Computer Architecture for Machine Performance*, pages 11–13, Padova, Italy, September 2000.

[7] J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper, and R. Beveridge. Cameron: High Level Language Compilation for Reconfigurable Systems. In *PACT'99 Conference on Parallel Architectures and Compilation Techniques*, pages 12–16, Newport Beach, CA, October 1999.

[8] A. J. Hoffman and J. B. Kruskal. Integral boundary points of convex polyhedra. In H. W. Kuhn, editor, *Linear Inequalities and Related Systems*, pages 223–46. Princeton University Press, Princeton, N.J., 1956.

[9] X. Hu, S. C. Bass, and R. G. Harber. Minimizing the Number of Delay Buffers in the Synchronization of Pipelined Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(12):1441–1449, December 1994.

[10] J. Jean, X. Liang, B. Drozd, K. Tomko, and Y. Wang. Automatic Target Recognition with Dynamic Reconfiguration. In *http://citeseer.nj.nec.com/250639.html*, July 1999.

[11] D. Johnson. Architectural Synthesis from Behavioral C Code to Implementation in a Xilinx FPGA. In *http://www.frontierd.com/.* Frontier Design Inc.

[12] N. Kerkiz. *Development and Experimental Evaluation of Partitioning Algorithms for Adaptive Computing Systems.* PhD thesis, University of Tennessee, Knoxville, TN, December 2000.

[13] R. Kuznar and F. Brglez. PROP: A Recursive Paradigm for Area-Efficient and Performance Oriented Partitioning of Large FPGA Netlists. In *International Conference on Computer-Aided Design*, pages 644–649, November 1995.

[14] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

[15] C. S. G. Lee and P. R. Chang. A Decomposition Approach for Balancing Large-Scale Acyclic Data Flow Graphs. *IEEE Trans. Comput.*, 39(1):34–46, January 1990.

[16] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry By Retiming. In *Proc. 3rd Caltech Conf. Very Large Scale Integration*, pages 87–116, Pasadena, CA, March 1983.

[17] B. Levine. A system for the implementation of image processing algorithms on configurable computing hardware. Master's thesis, University of Tennessee, Knoxville, TN, August 1999.

[18] A. H. Lightstone. *Fundamental of Linear Algebra.* Meredith Corporation, New York, New York, 1969.

[19] K. Murty. *Linear and Combinatorial Programming.* John Wiley & Sons, Inc., New York, 1976.

[20] W. Najjar, B. Draper, A. Bohm, , and R. Beveridge. The Cameron Project: High-Level Programming of Image Processing Applications on Reconfigurable Computing Machines. In *PACT'98 Workshop on Reconfigurable Computing*, Paris, France, October 1998.

[21] S. Natarajan. Development and verification of library cells for reconfigurable logic. Master's thesis, University of Tennessee, Knoxville, TN, August 1999.

[22] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., New York, 1988.

[23] J. B. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. In *Proc. 20th ACM Symposium on the Theory of Computation*, pages 377–387, May 1988.

[24] C. H. Paradimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[25] J. R. Rasure and S. Kubica. The KHOROS Application Development Environment. *World Scientific*, 1994.

[26] J. R. Rasure and C. S. Williams. An Integrated Data Flow Visual Language and Software Development Environment. *Visual Languages and Computing*, 2:217–246, 1991.

[27] H. Rowley. *Neural Network-Based Face Detection.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1999.

[28] H. A. Rowley, S. Baluja, and T. Kanade. Neural Network-Based Face Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):23–38, January 1998.

[29] J. F. Shapiro. *Mathematical Programming: Structures and Algorithms.* John Wiley & Sons, Inc., New York, 1979.

[30] B. Stanley. *Hierarchical Multiway Partitioning Strategy with Hardware Emulator Architecture Intelligence.* PhD thesis, Georgia Institute of Technology, 1997.

[31] M. Young, D. Argiro, and S. Kubica. Cantata: Visual Programming Environment for the Khoros System. *Computer Graphics*, 29(2):22–24, May 1995.

VITA

Sze Wei Ong was born on May 17, 1974 in Segamat, Johor, Malaysia. After receiving his primary and secondary education in Segamat, he studied at Damansara Utama College, in Kuala Lumpur, Malaysia. In August 1993, he transferred to University of Tennessee at Knoxville where he received his Bachelor of Science degree in Electrical and Computer Engineering in December 1995. Upon completion of his undergraduate education, he accepted a graduate research assistantship at the University of Tennessee. He received his Master of Science degree in Electrical Engineering from the University of Tennessee in August 1997. He then began working toward his Doctor of Philosophy degree. While pursuing his doctorate degree, he was a teaching assistant for the Department of Electrical and Computer Engineering and a research assistant for the Communications, Information, and Signal Processing Laboratory and the Microelectronic Systems Research Laboratory. He received his Doctor of Philosophy degree in Electrical Engineering in May 2001.