

To the Graduate Council:

I am submitting herewith a thesis written by Bernadeta Sriyanto entitled “Implementing Neural Network-Based Face Detection onto A Reconfigurable Computing System Using CHAMPION.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Donald W. Bouldin

Major Professor

We have read this thesis
and recommend its acceptance:

Dr. Danny Newport

Dr. Chandra Tan

Accepted for the Council:

Dr. Anne Mayhew

Vice Provost and
Dean of The Graduate Studies

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

**IMPLEMENTING NEURAL NETWORK-BASED FACE
DETECTION ONTO A RECONFIGURABLE COMPUTING
SYSTEM USING CHAMPION**

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Bernadeta Srijanto

August, 2002

Dedicated to
my parents, brother, and sister for their love and support

ACKNOWLEDGEMENT

I would like to thank my advisor, Professor Donald W. Bouldin, for his encouragement, support, and guidance which helped me to finally complete this work. I would also like to thank you, Dr. Bouldin for giving me the opportunity to be part of the Microelectronic Research group at the University of Tennessee, where I have learned a lot and have become more interested in the VLSI area. I also thank my committee members, Dr. Chandra Tan and Dr. Danny Newport for its assistance. I would also like to acknowledge the Defense Advance Research Projects Agency for its financial support of this research under grant F33615-97-C-1124. Special thanks to my parents, brother, and sister for their love, support, and patience during my time away from home to achieve my degree. Finally, I also thank Ms. Daisy Bolton, my UT mom, for her patience and encouragement and for always reminding me of what is important in my life.

ABSTRACT

From the innovation of the mechanical computers to the invention of semiconductors, there is Adaptive Computing System (ACS), which can be customized to suit users' specific applications. Automated system software is needed to accommodate application mapping onto an ACS as it takes an extensive amount of time to perform the process manually. CHAMPION is an automatic mapping tool developed at the University of Tennessee. Using Khoros Cantata workspace as its input, CHAMPION's goal is to improve designer productivity by 100 percent. The neural network section of face detection was one of multiple applications used as the challenge problem to CHAMPION. The face detection system was originally developed and written in floating point C programming language by Henry Rowley at the Carnegie Mellon University. Unfortunately, the neural network could not be realized in the Wildforce-XL board, which was the targeted ACS platform. Upon finishing this project, this thesis presents some challenges to improve CHAMPION specifically and automatic system in general.

TABLE OF CONTENTS

CHAPTER	PAGE
1. Introduction	1
1.1 Motivation	1
1.2 Research Goal	6
2. Background	10
2.1 CHAMPION	10
2.1.1 FPGA Design Process Flow	10
2.1.2 CHAMPION Motivation	12
2.1.3 CHAMPION Flow	14
2.1.4 CHAMPION Status	18
2.2 KHOROS	21
2.2.1 Cantata	22
2.2.2 Craftsman [4]	24
2.3 Neural Network	29
2.4 <i>A RT Library</i>	33
2.5 Face Detection	35
2.5.1 Localization and Pose Estimation	36
2.5.2 Pre-Processing	39
2.5.3 Neural Network Detection	42

CHAPTER	PAGE
2.5.4 Post-Processing	44
2.6 Wildforce-XL Board	48
2.6.1 Xilinx XC4000 Series FPGA	48
2.6.2 Wildforce-XL Board Architecture [1]	51
3. Methodology	54
3.1 Choosing Specific Routines of Face Detection System Provided by CMU	54
3.2 Floating Point Cantata Design	58
3.3 Fixed-Point Cantata Design	61
3.4 Glyphs Installation	66
3.4.1 VHDL Development	66
3.4.2 Using <i>Geninf</i>	71
3.5 Running CHAMPION	72
3.6 Host Code Generation	73
4. Implementation and Results	78
4.1 Floating Point Cantata Implementation	78
4.2 Fixed-Point Cantata Implementation	82
4.3 Fixed-Point Hardware Implementation	87
4.3.1 Hardware implementation first changes	87
4.3.2 Hardware implementation second changes	95
4.4 Result	101

CHAPTER	PAGE
4.4.1 Comparison between original C code, floating point Cantata, and fixed-point Cantata of the face detection system	102
4.4.2 Comparison between fixed-point Cantata implementation, hardware pre-layout and post-layout simulation	106
4.4.3 Design layouts on a Virtex-E XCV3200E device	112
5. Summary and Future Works	120
5.1 Performance	120
5.1.1 Design Performance	120
5.1.2 CHAMPION performance	122
5.2 Future Work	123
BIBLIOGRAPHY	125
VITA	128

LIST OF FIGURES

FIGURE	PAGE
1.1 Project goal to compare the results of different implementations. . . .	9
2.1 FPGA Design Flow.	13
2.2 CHAMPION Design Flow [15].	15
2.3 Example of truncating and padding glyphs insertion [15].	16
2.4 Example of delay buffer insertion [15].	17
2.5 CHAMPION Graphical User Interface.	19
2.6 Cantata Graphical Environment.	23
2.7 Example of Cantata Glyph: Multiplication program.	25
2.8 Craftsman User Interface.	27
2.9 Composer User Interface.	28
2.10 Multi Layer Perceptron Architecture.	31
2.11 Representation of One Neuron.	32
2.12 Face Detection algorithm adopted from Carnegie Mellon University [17].	37
2.13 Localization of each level of subsampling process [17].	38
2.14 Histogram Equalization [3].	41
2.15 Receptive field of 30×30 window detector [17].	43
2.16 Clean-up heuristic using <i>thresholding(4,2)</i> [17]	45
2.17 Xilinx XC4000 Architecture	49

FIGURE	PAGE
2.18 Wildforce-XL board block diagram. (a) Simplified block diagram of Wildforce-XL board (b) Wildforce-XL board as used in CHAMPION.	53
3.1 Face detection execution for each stage.	55
3.2 Output example of <i>im2_test</i>	56
3.3 The <i>im2_test</i> algorithm.	57
3.4 Floating point design.	59
3.5 <i>Umec</i> network receptive fields. (a) 10×10 -pixel receptive fields (b) 30×5 -pixel receptive fields (c) 5×30 -pixel receptive fields. . . .	62
3.6 New layout of <i>umec</i> network.	65
3.7 Block diagram of <i>umec</i> network fixed-point Cantata design.	67
3.8 Block diagram of <i>face17c</i> network fixed-point Cantata design.	68
3.9 Block diagram of <i>face18c</i> network fixed-point Cantata design.	69
3.10 Configuring CHAMPION environment.	74
3.11 Individual CHAMPION process.	74
3.12 CHAMPION process report window.	75
3.13 Host pseudo-code for the <i>umec</i> network.	77
4.1 Floating point Cantata workspace of <i>umec</i> network.	79
4.2 Floating point Cantata workspace of <i>face17c</i> network.	80
4.3 Floating point Cantata workspace of <i>face18c</i> network.	81
4.4 Partial data of files containing pixel values, weight values, and output in the floating point implementation.	82
4.5 Fixed-point Cantata workspace of <i>umec</i> network.	83

FIGURE	PAGE
4.6 Fixed-point Cantata workspace of <i>face17c</i> network.	84
4.7 Fixed-point Cantata workspace of <i>face18c</i> network.	85
4.8 Partial data of files containing pixel values, weight values, and output in the fixed-point implementation.	86
4.9 Data type Fix < 14, 10 >, 14-bit width and 10 bits of precision.	87
4.10 Partition violation.	89
4.11 <i>Umec</i> network sub-configurations.	90
4.12 Dividing one 3-input datamerge glyph into two 2-input glyphs.	91
4.13 Reducing and locating the multiplier and hyperbolic tangent function glyphs.	93
4.14 The use of register module.	94
4.15 New modified first configuration of <i>umec</i> network fixed-point Cantata.	96
4.16 New modified second configuration of <i>umec</i> network fixed-point Cantata.	97
4.17 The new flow of hardware implementation.	99
4.18 Xilinx Virtex-E Architecture. (a) One Virtex-E CLBs which contains four LCs arranged in two slices. (b) Detailed view of one slice.	100
4.19 <i>Umec</i> simulation timing diagram. (a) Pre-layout simulation. (b) Post- simulation.	113
4.20 <i>Face17c</i> simulation timing diagram. (a) Pre-layout simulation. (b) Post-simulation.	114
4.21 <i>Face18c</i> simulation timing diagram. (a) Pre-layout simulation. (b) Post-simulation.	115

FIGURE	PAGE
4.22 Umec network design layout on Virtex3200E.	117
4.23 Face17c network design layout on Virtex3200E.	118
4.24 Face18c network design layout on Virtex3200E.	119

LIST OF TABLES

TABLE	PAGE
2.1 Resources on XC4013XL and XC4036XL FPGAs.	51
3.1 The structure of different network used in <i>im2_test</i>	58
4.1 Resources on XC4013XL, XC4036XL and XCV3200E FPGAs.	101
4.2 Computation results on test image: albert.ppm	103
4.3 Computation results on test image: henry.ppm	104
4.4 Computation results on test image: couple2.ppm	105
4.5 Neural network computation results of first ten 30×30 window detectors <i>umec</i> network.	108
4.6 Neural network computation results of first ten 20×20 window detectors <i>face17c</i> network.	109
4.7 Neural network computation results of first ten 20×20 window detectors <i>face18c</i> network.	110
4.8 Truncation glyph altered the original values.	111
4.9 Timing information for pre-layout and post-layout simulation on design implementation on XCV3200E device.	116
4.10 Summary of resource utilization of each network on XCV3200E.	116

CHAPTER 1

Introduction

1.1 Motivation

Machines or computers were invented to help people do their jobs. Instead of doing some calculations of large numbers by hand, people can just enter the numbers into a calculator, and in a matter of seconds, a result is displayed. Instead of writing data or information in a book, a clerk can just store it in computer memory and have it filed by date so that it is easier to retrieve whenever it is needed. A computer can also be trained to detect a pattern provided by a satellite to recognize the type of surface of a planet.

Computers do what humans tell them to do. A program containing a set of instructions is written to tell computers to perform certain jobs. Examples include programs to do calculations such as addition, subtraction, multiplication and division, send data to a printer, recognize someone's voice, etc. Because of the language difference that exists between human and machine, those programs need to be translated or interpreted so that computers can recognize them. Once recognized, the instructions are carried out by the computer's electronic circuits, or the so-called computer hardware.

To make a computer execute faster, some programs written in software need to be built directly into hardware. In this case, neither translation nor interpretation is

involved. However, not every program should be implemented in hardware. Cost, efficiency, and flexibility of changes are other considerations whether or not an operation should be done in software or hardware. It is better to build hardware for operations that are used often, such as integer arithmetic functions.

Computer hardware primarily consists of three elements [14]. The first is a data operator. That is, sets of functions that perform operations on given inputs and then give results. The second element is a storage element, which is memory in which data inputs, outputs, and intermediate results of some operations are stored. The last element is wiring: connections to transfer data between operators, storage, and an operator and storage.

In the beginning, computers just performed simple operations. The Pascal's machine built in 1642 by Blaise Pascal as well as the Difference engine designed by Charles Babbage implemented addition and subtraction only. Both machines were mechanical. The Pascal's machine used a handle that was operated by hand. The Difference engine used copper plate to print its output, working like punch cards [20].

Also, computers in earlier times were big in size. Following the discovery of vacuum tubes, the electronic computer was produced in the 1940's. This computer, which was built during World War II, was called ENIAC (Electronic Numerical Integrator and Computer). Its weight was 30 tons, contained 18,000 vacuum tubes, 1,500 relays, 6,000 switches, and used about 140 kilowatts of power. The ENIACS was used as a mechanical calculator in the army at the time.

With the invention of transistor in 1948, followed by integrated circuit (IC) technol-

ogy in the 1960's, computer hardware today is smaller, faster, and cheaper. Small-scale integration (SSI) and medium-scale integration (MSI) contained up to 100 gates per chip [14]. They provided basic logic such as AND, OR, NOT, NAND, and NOR gates, flip-flops, registers, and counters. Those chips were mostly manufactured by Texas Instruments, named the TTL logic, and by National Semiconductor, named the 4000 series. To perform larger functions such as memory or a processor, these components had to be put together on a board and wired. In the 1970's, those larger functions could be packed into a single IC. There were about 10,000 gates per chip and was called large-scale integration (LSI). The technology improved with the development of very large-scale integration (VLSI). VLSI had about 100,000 gates per chip. An example was the 32-bit microprocessor Motorola 68000 family.

The ICs mentioned above are considered standard ICs. Standard ICs perform general applications; for example, mathematical calculations, signal and image processing, control systems, spreadsheet operations, etc. To carry out one of the tasks mentioned above a program is written and then translated or interpreted. This translation or interpretation process takes time and results in slowing down the performance. Because of this, in the 1980's an application specific integrated circuit (ASIC) was developed. ASIC are hardwired. The interconnections are fixed or customized for a special purpose. An ASIC for signal processing cannot be reprogrammed for a control system job and vice versa. Since a set of instructions is now run directly by hardware, an ASIC performs faster than a standard IC does. With the advance of technology in semiconductors, where more transistors can be held in a single space, an ASIC saves

space. An implementation with standard ICs that used 10 inches² of PCB board, for instance, could now be replaced by a single part. Another way to look at it, with the same of 10 inches², the board can carry more functionality than before.

Despite its better execution time and space saving, ASIC has a long time-to-market as its shortcoming. After an engineer designs an application to ASIC, the design is sent for manufacturing. Fabrication could take from weeks to months. Next, the chip is brought back to the designer for testing. Once it is made, the part cannot be changed. When there are flaws, or there is a need for modifications, the engineer has to redesign the application and send it for fabrication again. This process takes until the correct ASIC is completed and ready for production.

In the mid-1980's, Field Programmable Logic Arrays (FPGAs) were chosen as a solution to the longer time-to-market of ASIC [14]. In general, this type of hardware contains arrays of a large number of functional units and programmable interconnections between the units. The most used FPGAs are available from Xilinx, Altera, and Actel. For the chip to carry out a certain task, the interconnections as well as the functional units need to be configured. The configuration can be changed to perform different tasks. Reconfiguration is also possible to do modification of an existing application. No extra time is needed before the design is ready for use. At the same time, the designer is able to check and verify the application performance. The designer does not have to wait for weeks or months to receive the fabrication result from the manufacturer, as is the case with designing with an ASIC. In addition, an FPGA can also be used as ASIC prototyping because of its quick design and checking capability.

Once a design works correctly in an FPGA, it is ready to be sent to the manufacturer for ASIC fabrication.

Several FPGAs are connected together in a board to specially implement applications that do not fit in one FPGA. This kind of hardware is called an Adaptive Computing System (ACS) and is supported by additional components such as memory, external connectors, and PCI bus that assists communication between the board and a computer host. The Wildforce and Wildcard boards developed by Annapolis Inc. [1], the SLAAC-1 and SLAAC-2 by University of Southern California [5], are examples of ACS. All of them use Xilinx chips as their FPGAs.

To configure an ACS so it performs a specific function, certain process or steps are done using different sets of tools or software. Those steps include capturing the design of the function, synthesizing, simulating, partitioning, mapping, placement, and routing. Usually this process is quite tedious and time consuming. The need of a design automation tool is essential to reduce the hardware configuration time. CHAMPION is an automatic mapping tool developed at the University of Tennessee [16]. Its goal is to improve the designer productivity by 100 times. Several applications have been used as challenge problems of CHAMPION. The neural network-based face detection will be one of them.

Throughout history, researchers and scientists have found ways to develop technologies that help humans carry out their tasks in life. From the innovation of the mechanical computer to the invention of the semiconductors, there was ASIC which could perform tasks faster. To reduce the time-to-market as result of market demand,

FPGA has been developed. Today, FPGA technology has been used in a variety of fields such as telecommunications, special computer peripheral equipment, control systems, industrial instrumentation, etc. With application implemented onto hardware, the expectation is to have it execute faster than if it is run by a set of instructions or in software. The progress does not stop here. Design automatic tools are developed to make the process of hardware implementation even faster.

1.2 Research Goal

The purpose of this research done by the author is to implement certain operations of an application onto hardware. The operation chosen was neural network as part of Face Detection application. Henry Rowley from the Carnegie Melon University had developed an algorithm of this application for his PhD dissertation [17]. His algorithm was adopted and analyzed so it could be implemented onto FPGA type hardware.

To implement the application, an automatic mapping software tool was used. This software was called CHAMPION. It was developed by a research group at the University of Tennessee [16]. The reason to use this mapping tool instead of doing manual mapping is to reduce hardware configuration time. Since CHAMPION takes input from an image processing program called Khoros Cantata, the first thing that concerned the author was how to transform the given face detection code onto the Khoros Cantata workspace so that it imitated the behavior of hardware. The author's research task was also to develop libraries, which were needed in this application but did not exist in Khoros Cantata. Those libraries are called toolboxes. To help the process of

implementation using CHAMPION, each toolbox had its corresponding pre-compiled VHDL library. Accordingly, the author's job was also to write VHDL for the libraries, which were not available in the CHAMPION environment.

Using CHAMPION, the hardware was to be configured so that the neural network-based face detection application could be downloaded and executed. It was the author's intent to compare the execution time of running the application in hardware versus in software. In addition, the resulting implementation was analyzed to observe the accuracy of the result between hardware and software.

To achieve the goal mentioned above, the following statements have become the author's guidance for implementation:

1. Understanding and analyzing the neural network-based face detection algorithm.
2. How to design the network architecture so that it can be implemented in ACS without violating any constraints.
3. How to implement the network in floating point and fix point Khoros Cantata workspace.
4. How to implement the network workspace using the CHAMPION flow.
5. Comparing results between Cantata workspace and ACS in term of execution time and accuracy (shown in figure 1.1).

As this first chapter introduces the research done by the author, the next chapter gives background on the CHAMPION tool, Khoros Cantata, neural network, and the

Face Detection system. Chapter III discusses the methodology used in this project. Chapter IV presents implementations and results, followed by summary in Chapter V.

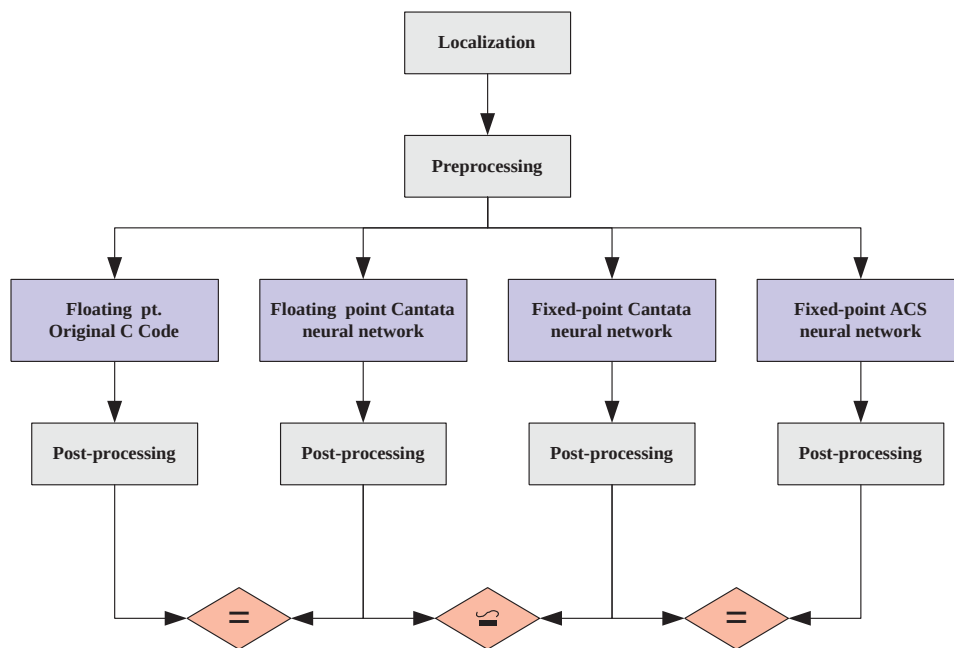


Figure 1.1: Project goal to compare the results of different implementations.

CHAPTER 2

Background

2.1 CHAMPION

2.1.1 FPGA Design Process Flow

Even though there are different types of FPGA technology, the design process flows are similar. They go through design capture, synthesis, pre-layout simulation, mapping, placement, routing, and post-layout simulation. After a design engineer defines specifications and comes up with an idea on how to implement an application, he needs to capture the design into a behavioral and/or structural description. A behavioral description specifies functions of a design. It is usually described using hardware description language (HDL) such as Verilog and VHSIC Hardware Description Language (VHDL). A structural description, meanwhile, defines components used and their interconnections. Although structural design can be described using HDL, it is commonly visualized using schematic diagrams.

Once a design is captured, it is translated into a netlist using a synthesis tool. A netlist is a description of circuits which carries out the behavioral design given. It contains logic cells that structure the design and interconnects information between those cells. The resultant netlist has usually been optimized. That is delays are

minimized. With the help of a synthesis tool, this optimization process, which could take hours to days if done by human effort, can be done in a few minutes. Creating a netlist can also be achieved for different hardware platforms without having to change the behavioral and/or structural design. The same captured design can be synthesized more than once by the tool using technology-specific libraries.

After synthesis, the design is ready for pre-layout simulation. In this step it is determined whether or not the design functions correctly. By using commercial tools, simulation generally is done by entering input vector to the circuit, then outputs are observed as well as its intermediate results. If the outputs are not as expected, the design may have to be recaptured and synthesized. Information about timing, such as delay which occurs in the logic cells, is also noticed to determine how fast the circuit executes its tasks.

The next step of the FPGA design process flow is mapping, continued by placement and routing. The mapping process takes logic cells listed in netlist and maps them physically on silicon. Placement deals with locating or arranging the logic blocks so that the design can use the spatial capacity of a particular FPGA architecture. Routing, which follows mapping and placement, does the wiring between the blocks, with the goal to minimize total interconnection length leading to reducing time delay. Since placement and routing involves thousands of gates and hundreds of nets, it is easier and more time saving to leave the job to automation software than using human effort. To verify if the design still meets its specifications after going through physical design stages, a post-layout simulation is performed. The entire flow of the FPGA design is

shown in figure 2.1.

2.1.2 CHAMPION Motivation

The process of implementing applications onto hardware as previously mentioned is very tedious, extensive, and time consuming. It could take several months for a design engineer to do manual mapping. Benjamin Levine did the manual mapping of Automatic Target Recognition (ATR) application on a multi FPGA platform in 250 hours [12]. In addition to the time consideration, a significant knowledge of design flow, digital logic designs, and different hardware technologies is required. It is difficult for one to implement a design if one is unfamiliar with the process starting from an idea to the actual performance.

To overcome this obstacle, it is beneficial to have a program or software, which can execute the design process flow automatically. One of such programs is currently being developed by Microelectronic System Research Laboratory at the University of Tennessee of Knoxville. This software is called CHAMPION. The goal of CHAMPION software is to perform automatic mapping, resulting in improvement of productivity by 100 times. CHAMPION implements applications that have been designed in Khoros Cantata environment. Khoros Cantata is a graphical programming software, which allows the programmer to develop an algorithm, and capture it in Khoros, then run it without having to write codes. More about Khoros Cantata will be discussed in the next section. CHAMPION provides a link that is missing between Khoros Cantata application and hardware. Another advantage of CHAMPION is that it is applicable to

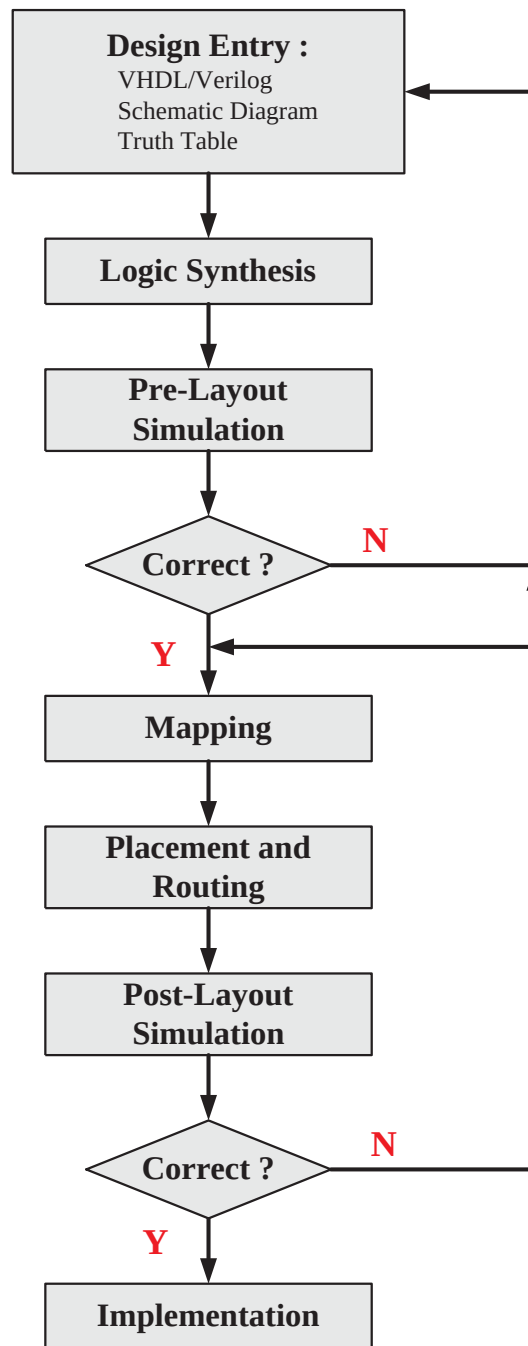


Figure 2.1: FPGA Design Flow.

most hardware technology. The application designer will have the option of choosing the hardware platform to obtain the expected results.

2.1.3 CHAMPION Flow

CHAMPION design flow (figure 2.2) consists of three main parts : library cell development, front-end design flow, and back-end design flow. New library cells are developed as needed for an application to be implemented. For example, library cells of addition, multiplication, and division are created for mean calculation purposes. Two types of library cells are required in the development process. The first is Khoros Cantata library cells, which are written in C language and commonly called the Cantata glyphs. Each Cantata glyph has its corresponding VHDL library, which has been synthesized and verified. The pre-compiled VHDL libraries are called the CHAMPION glyphs. After both Cantata glyphs and CHAMPION glyphs are developed, they are installed or added to the existing glyphs in CHAMPION environment.

The second component, the front-end flow, which was developed by Sze-Wei Ong [15], is responsible for converting Cantata workspace to the CHAMPION netlist, data width matching, and synchronization. When a CHAMPION user captures his algorithm in Cantata, he uses the existing Cantata glyphs and connects them together to carry out the design functionality. Once the expected result is obtained, the captured design is saved as a Cantata workspace. This workspace, which contains information about the glyphs used and their interconnections, is then translated into a CHAMPION netlist. In the translation process, the Cantata glyphs are replaced by CHAMPION

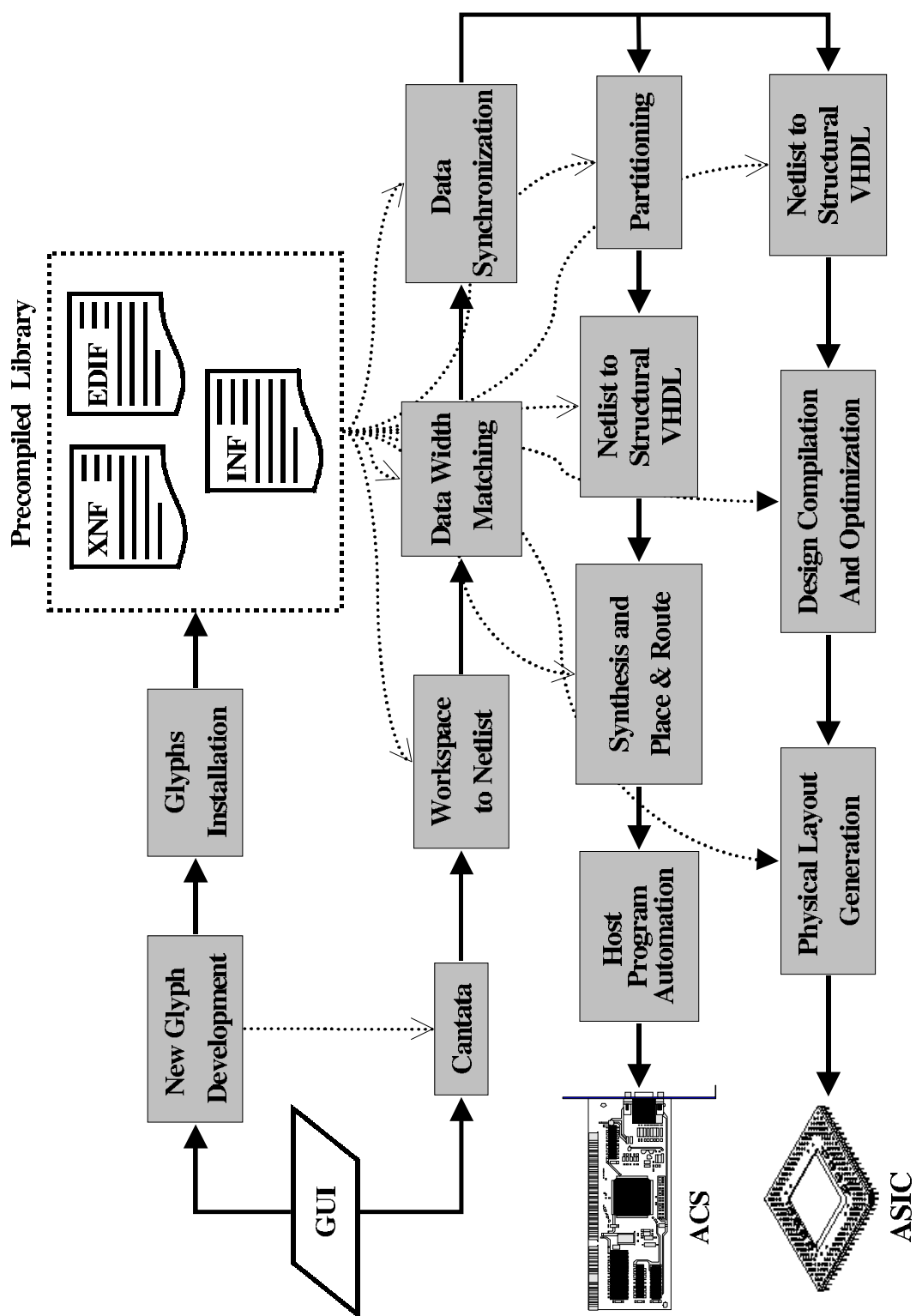


Figure 2.2: CHAMPION Design Flow [15].

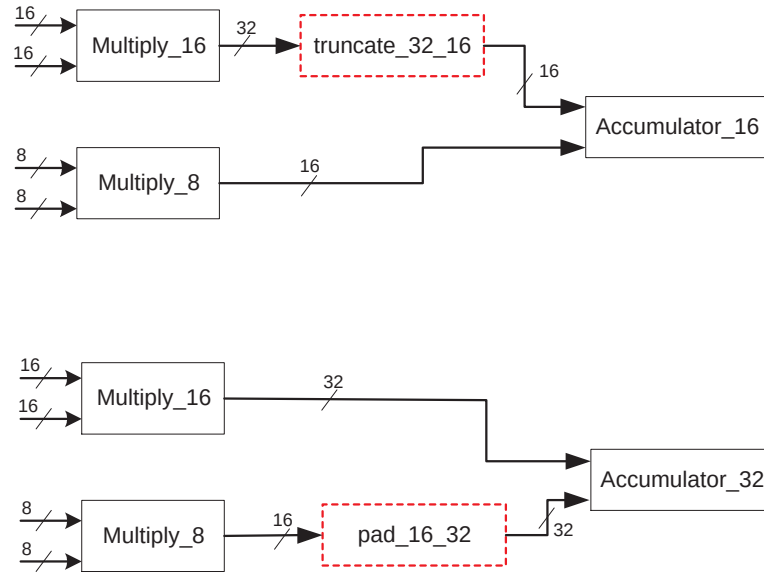


Figure 2.3: Example of truncating and padding glyphs insertion [15].

glyphs or nodes, and the their interconnections become nets between the nodes.

When there are two hardware glyphs cascaded together, there is a possibility of a mismatch number of bits between one and the other. The first mismatch occurs when the data width of a glyph output port is wider than that of the next glyph input port. To solve this problem, a truncating glyph is inserted between the two glyphs. The truncating glyph removes the extra bits from the first glyph. The second mismatch occurs when a glyph output port has fewer bits than the input of the next glyph. As opposed to truncating, a padding glyph is positioned between the two glyphs by adding 0's to the next input glyph. Figure 2.3 shows the insertion of truncating and padding glyphs.

In a Khoros application, program execution is data driven. A glyph begins its

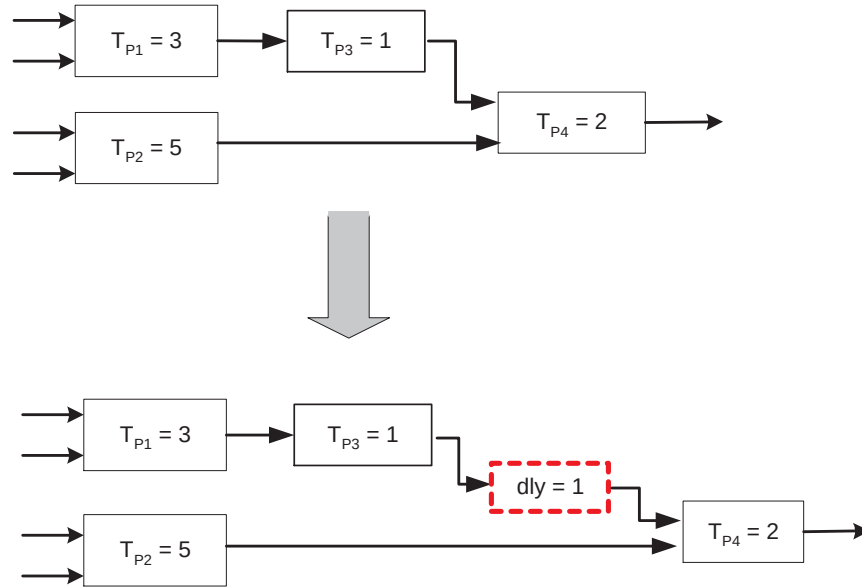


Figure 2.4: Example of delay buffer insertion [15].

operation when input data is available. In hardware applications, however, program execution is clock driven. Each hardware cell performs its process at each clock cycle despite the validity of the input data. A problem surfaces when a glyph has more than one input port. It is possible that one port has its input data available while the other ports are still waiting for input because it takes longer for the previous glyph to have the data result ready. Every time the glyph runs a process at each clock cycle, it takes valid data from the first input and invalid data from the other ports, resulting in incorrect data output. It is necessary for all inputs that arrive at the glyph concurrently to be valid. For this reason, data synchronization is obtained by inserting a delay buffer. Figure 2.4 is an example of the use of a delay buffer. The amount of time delay and the location of the buffer are calculated so that the total system delay is minimized.

After data width matching and synchronization, the CHAMPION netlist is ready to be mapped onto hardware. This process is performed in the last stage of CHAMPION flow, the back-end flow. The flow was developed by Nabil Kerkiz [10]. The processes of back-end flow include partitioning, synthesis, placement and routing, and host code development. In the case of a netlist that does not fit in one FPGA, the netlist is partitioned into several sub-netlists. This partitioning is done before the mapping process. Each sub-netlist is then converted to structural VHDL. Input and output ports are then added to those sub-netlists and merged with pre-compiled VHDL which has been mentioned in library cell development section earlier. After that, each sub-netlist is synthesized, followed by placement and routing process. These processes are carried out using commercial software tools.

To have a configuration file downloaded to the selected hardware, a host program is generated by CHAMPION. The host program calls functions, which are provided by specific board manufacturer, that allows communication between host workstation and the board. The host program also receives input data from the workstation and transmits the data to the board, and then obtains the output data from the board and sends them back to workstation.

2.1.4 CHAMPION Status

The CHAMPION software development has progressed where it is graphically available (figure 2.5). Instead of typing sets of commands, CHAMPION users are allowed to click different buttons to run particular processes within the flow. Running

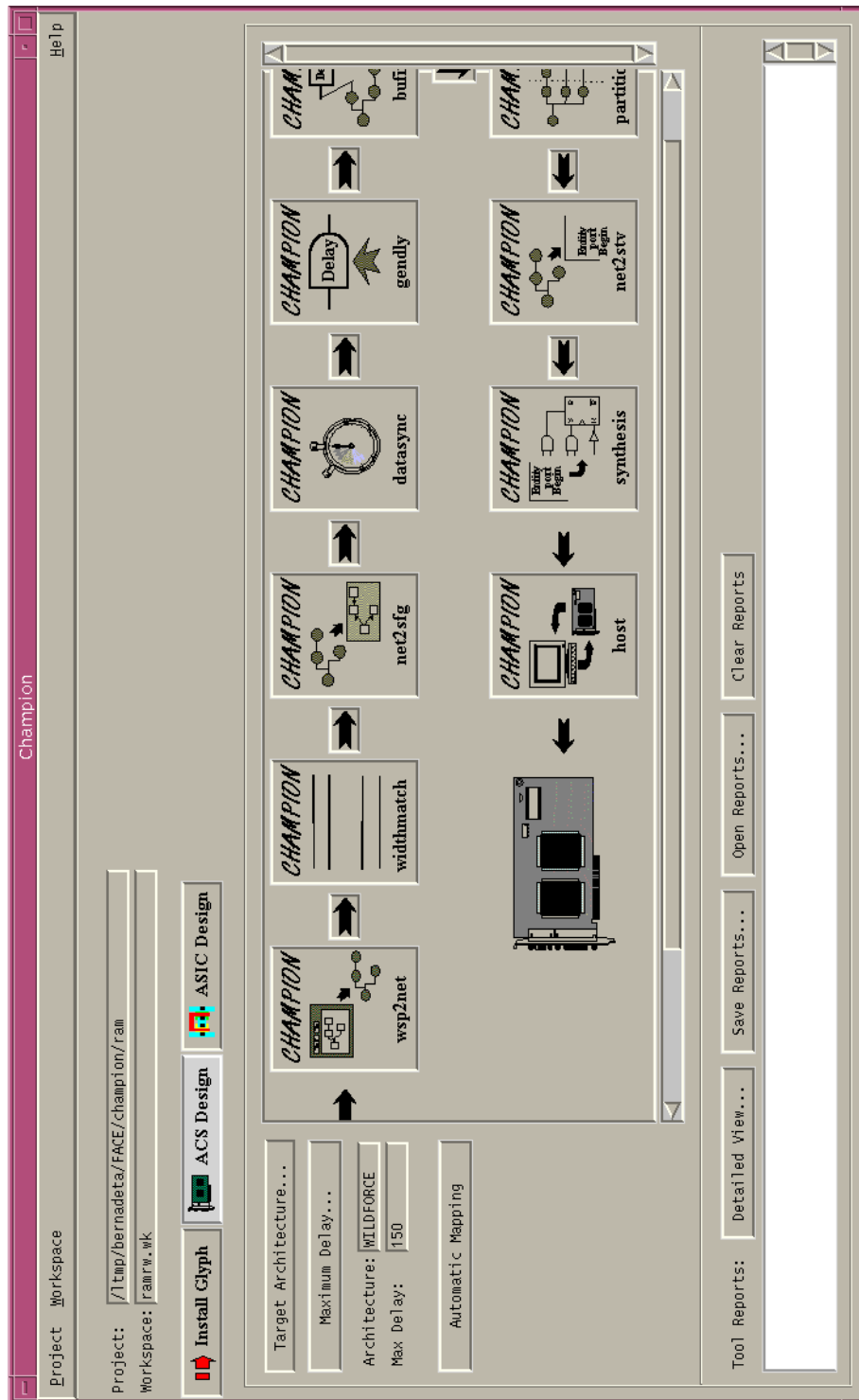


Figure 2.5: CHAMPION Graphical User Interface.

each process separately helps users during debugging. CHAMPION users also have the option to click on *Automatic Mapping* button to run the entire flow without stopping, unless errors occur. This saves the users time. In a graphical programming environment, users are able to visualize the entire process of design implementation. They can also observe the intermediate results of the flow. For example, by clicking the arrow after *gently* a list of delays generated automatically by CHAMPION appears. Furthermore, CHAMPION gives feedback to its users, such as process status, errors and warnings, timing information, and numbers containing data for analysis.

Another feature has been added to CHAMPION. It provides ASIC design flow so users are able to implement their applications on an ASIC, which can be sent for chip fabrication. The flow is similar except for the back end. Partitioning is not needed for an ASIC, and mapping is done directly to the target. Additionally, the design has to go through design optimization and physical layout generation.

Currently, CHAMPION has been used to implement various applications. They are a high pass filter, an Automatic Target Recognition algorithm, and Round-0 of the Army Night Vision Lab (NVL) algorithm. The hardware platform used is the Wildforce-XL from Annapolis Micro System. The next project is to implement these algorithms onto different hardware such as the Wildcard of Annapolis and the SLAAC board developed by the University of Southern California.

2.2 KHOROS

Khoros is a software integration and development environment developed by Khoros Research Incorporated [3]. Originated as a research project at the University of New Mexico, Khoros initially emphasized image processing and signal processing algorithms. Now, Khoros provides hundreds of programs that can be used in most applications. It also supports development tools for users to create and install new Khoros programs as needed. In addition, Khoros has the capability of handling a large amount of data and multi-dimensional data models up to 5D space.

Khoros users fall into two categories: end-users and application designers or developers. For end-users, Khoros provides hundreds of programs, which are classified into several operators, such as arithmetic, geometry, matrix, bit operations, statistic calculation, and image processing. Each program is stored in one container called a *Toolbox*, depending on its classification or operator. Using these available programs, users are able to do data manipulation, information processing, as well as data visualization. For a simple example, they can apply histogram equalization to an input image, then have its output image and the image size displayed in the Khoros environment. It is also possible to display the pixel value of the image and obtain the value of the mean, maximum, and minimum of those pixels.

Khoros' flexibility is very useful especially for application developers in order to expand existing Khoros software packages. The *Toolbox Programming Service* is a Khoros tool that allows developers to create a new toolbox or new program within

a toolbox needed for particular design. The toolbox programming service contains different routines or libraries. Examples of such routines are file read/write, information and data gathering, large data access, data manipulation, and usage of different widget sets to form graphical user interface. Once a new toolbox or program is created, it is installed so it can be accessed and used by end-users just as other Khoros original operators are.

2.2.1 Cantata

Cantata is a graphical programming environment within Khoros, which allows users to capture their designs or algorithms in data flow networks [21]. The algorithm data flow is visualized using a directed graph on Cantata workspace as displayed in figure 2.6. Each node, which is called a *glyph*, represents a program which either available in the Khoros system program or a new one created by developers. Meanwhile, each directed arc that connects two glyphs represents a data flow path between nodes. To implement an algorithm, a user locates glyphs corresponding to where the programs are used in the workspace, and then connects those glyphs to show the data flow direction.

To improve the design data flow model and simulation, Cantata provides control structures as in programming language like in C/C++ or in Fortran. The control structures are grouped into two categories: looping constructs and conditional constructs. Looping constructs allow users to apply a particular process to be repeated for a certain number times. Looping constructs are *count* and *while*, which are similar to *for-loop* and *do-while* statement respectively in C/C++ programming. For conditional con-

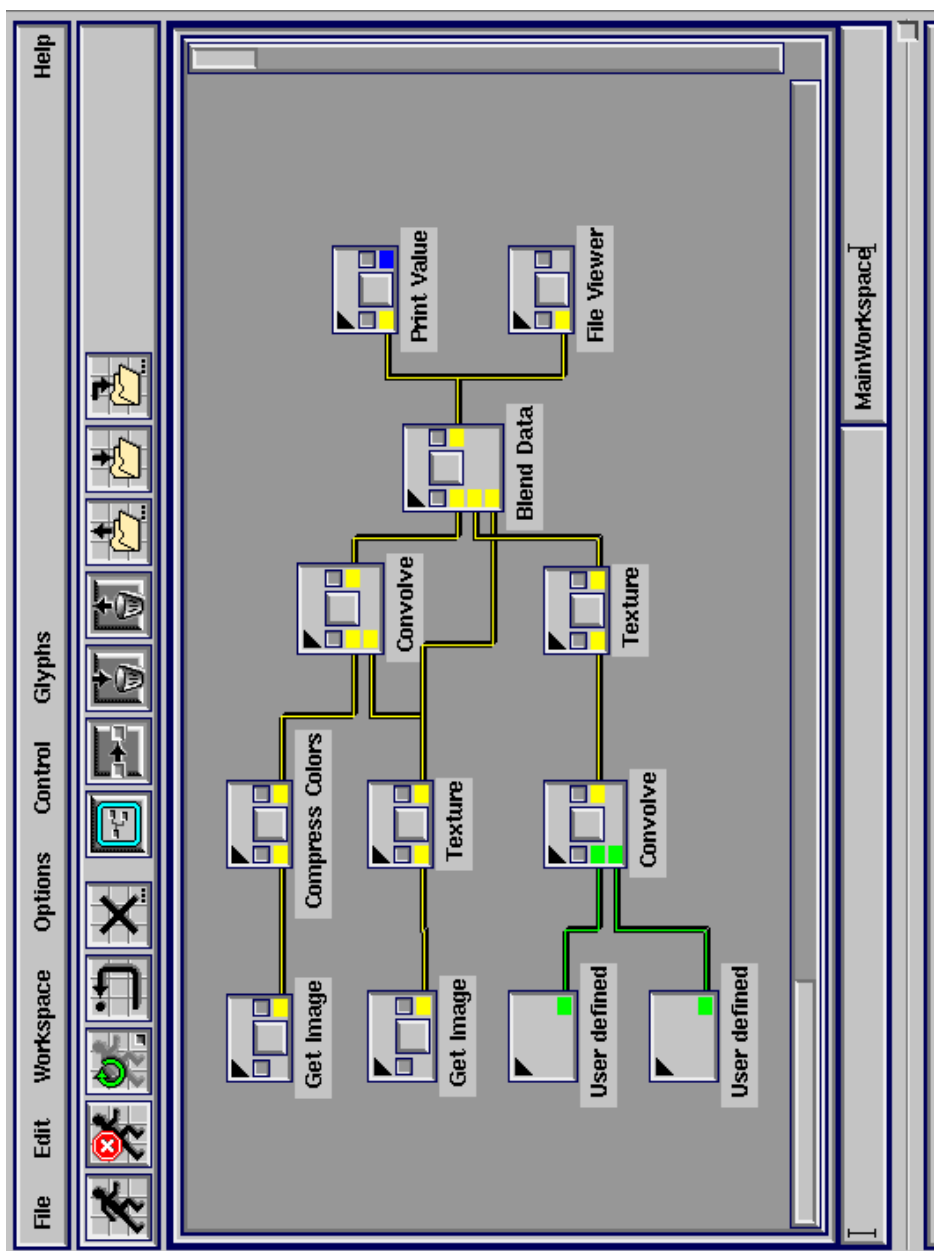


Figure 2.6: Cantata Graphical Environment.

structs, an iteration is executed within a loop until certain conditions are met. Included in conditional constructs are *if/else*, *break*, *continue*, *merge*, and *switch*. The *if/else*, *break*, and *continue* constructs operate analogously as in C/C++ programming. The *merge* construct combines two data paths into a single path data flow. Meanwhile, the *switch* construct allows data flow to choose one of two data inputs that meets some defined conditions.

As previously mentioned, in Cantata, each program is represented by one glyph. Each glyph contains several significant components (see figure 2.7). Those components include the input and output data connection node(s), the input and output control connection node(s), the Pane Access button, and the RUN button. Before a glyph is executed, input and output data connection nodes are in a yellow color. Once input data is available, the input data node becomes green. At this point, the glyph is ready to run. When the RUN button is clicked, the center of the glyph turns red showing that it is doing its job. After it is finished, the red color disappears and the output data node turns to green indicating that output data is available.

2.2.2 Craftsman [4]

Different Khoros tools and several steps are performed in developing new a toolbox and toolbox objects. First, there is a need to distinguish between a toolbox and a toolbox object. A toolbox is collection of programs and/or libraries. Each program or library is the one called toolbox object. For example, a toolbox of a two-operand arithmetic operator contains objects or programs such as multiplication, addition, subtraction, and

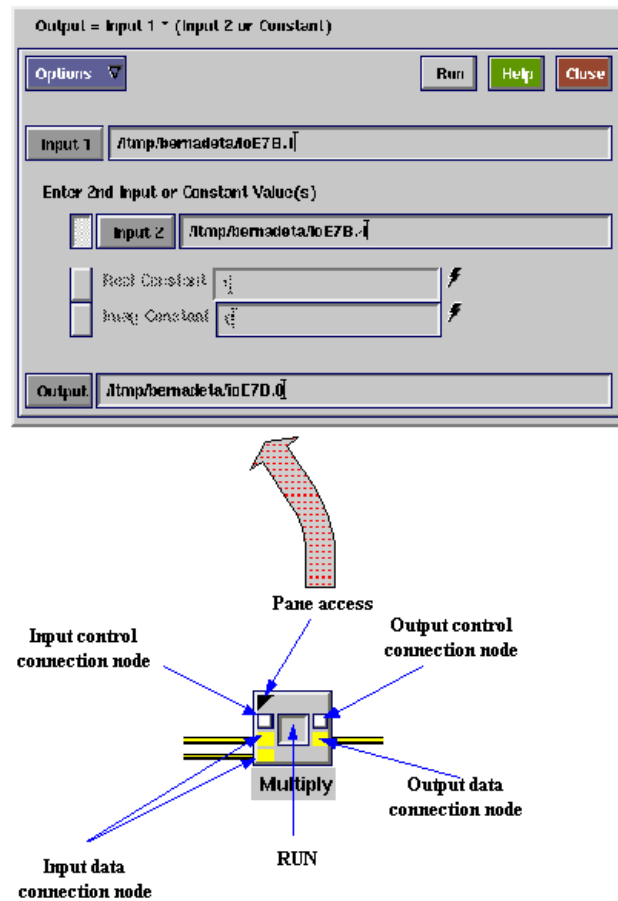


Figure 2.7: Example of Cantata Glyph: Multiplication program.

division. A toolbox needs to be created before developing the object. Three high-level Khoros tools needed in creating new toolbox are Craftsman, Composer, and Guise.

Craftsman is used to create, delete, copy, and change the attributes of a toolbox and toolbox objects. It is invoked via the command line. Two columns are displayed in the Craftsman user interface as seen in figure 2.8. The left column contains a list of existing toolboxes and the right one is a list of the toolbox objects of the selected or highlighted toolbox. The Toolbox Operations pull down menu on the left gives the developers an option to create new toolbox. When *Create Toolbox* is selected, a window appears and asks users to enter a toolbox name, path, title, status, and developers' name and email address. This information is used as the toolbox attributes.

To create the toolbox object or program within a toolbox, developers must first select that particular toolbox. Using the *Object Operations* pull down menu on the right, users then choose *Create Object*. The *Object Attributes* window is displayed for users to enter information about the new object being created such as object name and icon name. The icon name is shown with a corresponding glyph when the program is called in the Cantata workspace. For example, the object *kmul* for multiplication has the icon name Multiply. Besides having a particular object and an icon name as the object attributes, developers have the choice to write their code either in C, C++, or in script, also the choice of whether or not they want to install to object in Cantata.

Composer is invoked from the Craftsman object operations pull down menu by selecting *Edit Object*. Figure 2.9 shows the composer user interface. Composer is called the software object development environment because it is used to edit, manipulate, and

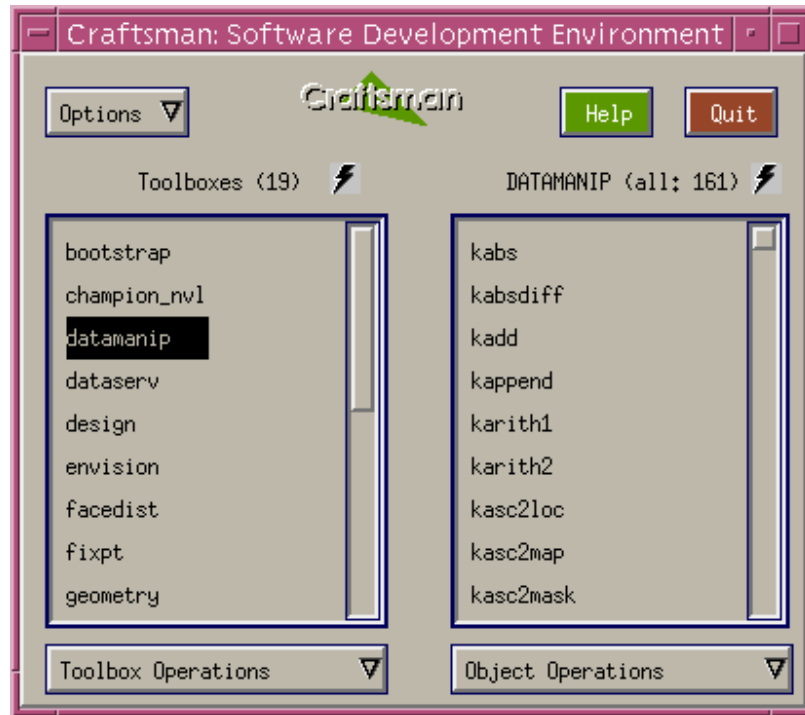


Figure 2.8: Craftsman User Interface.

compile existing software objects. The software object contains collections of files, called the file objects. Included in the file objects are source codes, documentation, user interface specifications (UIS), informational files, and miscellaneous files. The source codes consist of the code itself, the *include* files, and the *makefile*. Documentation is related to writing the manual page, and the UIS applies to object graphical user interface. Composer generates a template for each one of these files, which assists developers in integrating their programs with existing Khoros programs and libraries.

Before programmers start writing their code in a generated source code template,

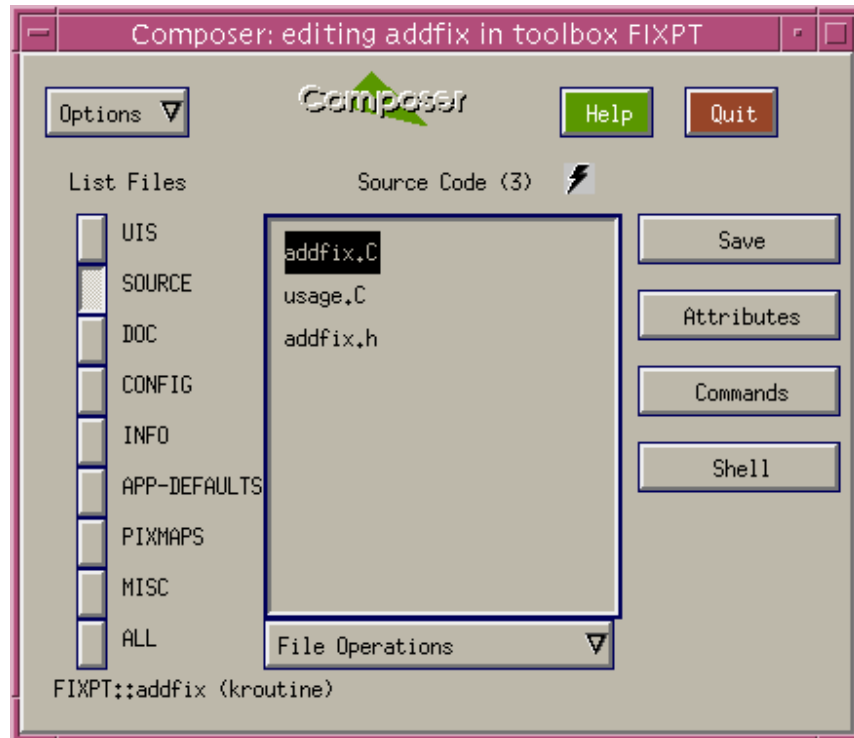


Figure 2.9: Composer User Interface.

they need to design the graphical user interface for the program. This dialog box will appear when the pane access button of its Cantata glyph is clicked as displayed in figure 2.7. The tool used to create GUI is called *Guise* which can be invoked from the *Composer File Operations* pull down menu by selecting *Guise*. *Guise* provides developers with several choices in designing dialog box such as:

- File variables : input and output file, stdin, stdout
- Various buttons : OK, Quit, Help buttons

- Simple variables : integer, floating, string variables
- List variables : circle, pull down menu, display list
- Other variables : flags, toggles, and logical selections.

During the development, programmers can preview the GUI using the *Preview* selection from the composer file operations pull down menu.

Khoros is a software environment that is useful to both end-users in performing various tasks such as data manipulation, information processing, and data visualization, as well as for application developers to create new programs, using Craftsman, Composer and Guise, which can be integrated with existing Khoros programs, to support certain applications. Cantata is the part of Khoros that allows users to capture an algorithm in a data flow network form in a workspace. Separate programs can be executed continuously or in parallel by connecting glyphs.

2.3 Neural Network

An Artificial Neural Network or so called *Neural Net* computing algorithm is inspired by the structure of the neural system in the human brain. The brain operates in parallel, allowing numerous processes to run at the same time. This is because human neural system contains billions of interconnected neural cells or so-called *neurons*. Each neuron is connected to other neurons via synapses. Through these synapses, information is transmitted from one neuron to others. This information transmission, which is called neurotransmission, is an electrochemical process. When

an electrical impulse is generated within a neuron, it stimulates chemical messengers called neurotransmitters, and releases them into its synapse. The neurotransmitters travel along the synapse to reach the next neuron, named the receiving neuron. In the receiving neuron, those messengers are transformed back into electrical impulse and so on. With such information transmission patterns, where neural cells transmit and receive neurotransmitters, the human brain is able to perform various complex functions.

A neural network model can be categorized into two types: single layer perceptron and multi layer perceptron. A single layer perceptron has only two layers, the input layer and the output layer. Each layer contains a certain number of neurons. The interconnection between two neurons of the two layers is similar to the synapse in brain. The input layer is usually used to hold the input data to the network. Thus, no operation is processed in neurons of this layer.

A multi layer perceptron is a neural network model that contains multiple layers. Each neuron in one layer receives its input from the neurons in previous layer and broadcasts its output to the neurons in the next layer. Figure 2.10 is an example of a neural network with three layers: input layer, one hidden layer, and output layer. Data are transmitted to neurons in the hidden layer from the input layer. An extra input called the *input bias* is added to the hidden layer. In fact, the input bias is added to each hidden layer and output layer. After some function is applied to each neuron in this layer, the result is sent to the neuron in the output layer. The same function is applied to the neuron in the output layer too. The output from this neuron is the output of the

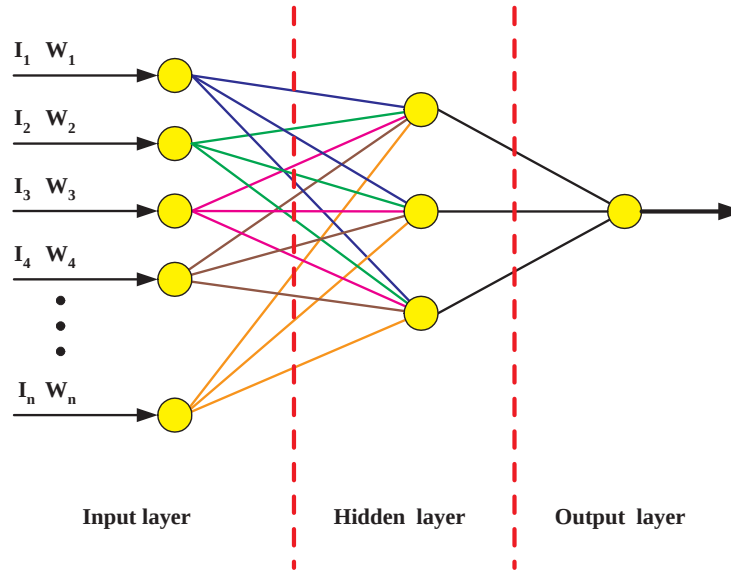


Figure 2.10: Multi Layer Perceptron Architecture.

network.

A neuron itself performs a particular process involving multiplication, accumulation, and applying an activation function, displayed in figure 2.11.

Input to each neuron is weighted by a certain number that is obtained from training the neural network. Each input is multiplied by its corresponding *weight* and then accumulated. The input bias value is usually assigned to one and has its own weight. The weighted sum of the neuron can be expressed as:

$$net = \sum_{i=1}^n I_i W_i$$

where I_i and W_i are the k^{th} input and weight respectively [11]. The *net* value is passed to the activation function. Output from the activation function is the actual neuron's

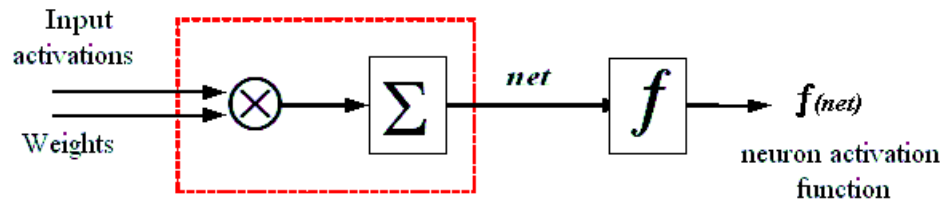


Figure 2.11: Representation of One Neuron.

output. The most common activation functions used are:

1. Threshold function

$$f(net) = \begin{cases} 0 & net < 0 \\ 1 & net \geq 0 \end{cases}$$

2. Sigmoid function

$$f(net) = \frac{1}{1 + e^{-net}}$$

In general, neural networks are used in pattern recognition or classification, image processing, function estimation, and process control as mentioned in Hammerstrom paper [9]. As an example of a classification system, is the Optical Character Recognition (OCR) by Sharp Corporation in Nara, Japan. The function of OCR is to read typeset of handwritten characters electronically. The system was trained to recognize more than 3000 characters, including Japanese and Romanese characters, and to process at least 200 characters per second at more than 99 percent accuracy. Othher existing systems recognize approximately 40 characters per second at 96 to 99 percent accuracy. Such speed is also supported by designing a programmable chip that executes the algorithm in parallel.

A neural network is used as a function estimator for financial forecasting. This application was developed at the NeuroForecasting Center, which was founded by the University College London and the London Business School. The system learns past market data to predict the next month bond market. The neural network is also trained to observe the effect of particular parameters. An example is how interest rates affect market behavior.

Other real world applications using a neural network are electrocardiograph and pap-smear systems from the Neuromedical System, Inc., process control by the Pavilion Technologies Inc., a check reader system called Magnetic-ink Character Recognition (MICR) from VeriFone, as well as fingerprints matching, underwater sounds identifier, and face recognition. In the military, neural networks have been used for target recognition, flight control and adaptive echo cancellation.

2.4 *A|RT Library*

Most applications running in software use floating point calculations. When they are implemented onto an ACS, the implementation is not effective because it takes large amounts of FPGA resources and run relatively slowly. To resolve this obstacle, fixed-point data representation is applied to replace the use of floating point number. However, as a consequence, the result of performance using fixed-point numbers is not as accurate as an implementation with floating point. Determining the number of bits is very important to reduce the error.

The A|RT Library, developed by Frontier Design, Inc., is a C++ linkable library that

can be integrated with existing user C or C++ to perform fixed-point applications [2]. The library contains a set of C++ classes including fixed-point data types and fixed-point operators. Another advantage of the A|RT Library is that limited knowledge of C++ is required, however, users can take control over the number of bits and precision used to represent data and its operations. Below are examples of A|RT Library usage:

- Format :
 - `Int< 4 > a = 2` is fixed-point integer format with value of 2 represented in 4 bits, that is 0010
 - `Fix< 8, 2 > b = 0.50` is signed fixed point number has the value of 0.25 represented 8-bit width and 2 bits after the imaginary binary point, displayed in binary as 00000010
- ASCII representation For example `Fix< 5, 2 > c = 0.5`
 - `c.dec() = 0.5`, a decimal representation
 - `c.bin() = 0bt000.10`, a binary 2s complement representation
 - `c.bpBin() = 00010`, a binary bit pattern representation
 - `c.hex() = 0xt0.8`, a hexadecimal 2s complement representation
- Data operations
 - `Fix< 8, 2 > a = 0.25`
 - `Fix< 8, 2 > b = 0.75`

- $\text{Fix} \langle 8, 2 \rangle c = a + b$
- Result : $c.\text{dec}() = 1.0$ or $c.\text{bpBin}() = 00000100$

The A|RT Library was used in this project for development of fixed-point Cantata glyphs.

2.5 Face Detection

Face detection is an important application that can be used for other applications especially in face recognition. The face recognition system has a task to identify one or more person in the image. Before the system can identify them, it needs to detect whether or not the image contains faces and the location of those faces. For a more enhanced application, face detection can also be part of face recognition in a video stream or movies. Below are several examples of face detection usage as listed also in [17]:

1. **Magic Morphin' Mirror:** The purpose of this application, which is conducted by Interval Corp. research, is to display distorted faces. An image is captured and the location of faces is determined. Then faces are twisted as if those faces were positioned in front of a fun house mirror [8].
2. **Name-It system:** Developed at Carnegie Melon University, the Name-It system (as part of the Informedia project) automatically labels each face found in a video stream such as in TV news or in movies. The system captures each frame of the stream, and applies the face detector algorithm. It also records times at which

particular faces appear. This Name-It system can be combined with speech recognition for closed caption application, such that detected faces and names are displayed at correct time [18].

3. **Security camera:** Today, security cameras are commonly used. A camera can be seen almost everywhere: in company buildings, libraries, supermarkets, and even in houses. The Justsystem Pittsburgh Research Center [19] and the Carnegie Mellon University have improved security camera products by adding face detection and recognition features. The new system records how many people appear in the camera and also recognizes the people captured by the camera.

This project adopted the face detection algorithm and routines that were developed by Henry A. Rowley at the Carnegie Mellon University [17]. The algorithm is based on a neural network and contains four main phases. They are localization and pose-estimation, pre-processing, neural network detection, and post-processing. The algorithm is displayed in figure 2.12.

2.5.1 Localization and Pose Estimation

Since the location of faces could be anywhere within an image, an exhaustive search was done using a 20×20 pixel window, displayed as a red box in figure 2.13. The detector was applied at every position of the image to make sure that the potential face detected was centered in the 20×20 pixel region. In case a face was larger than the 20×20 pixel window, the image was scaled down by a factor of 1.2. Image

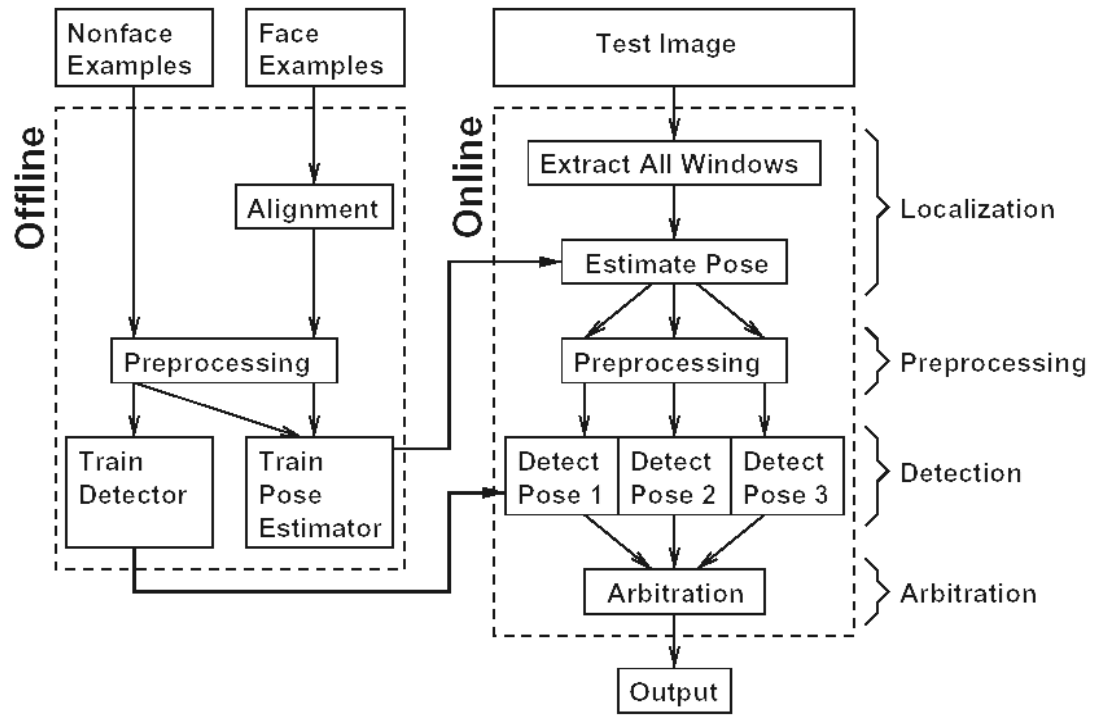


Figure 2.12: Face Detection algorithm adopted from Carnegie Mellon University [17].

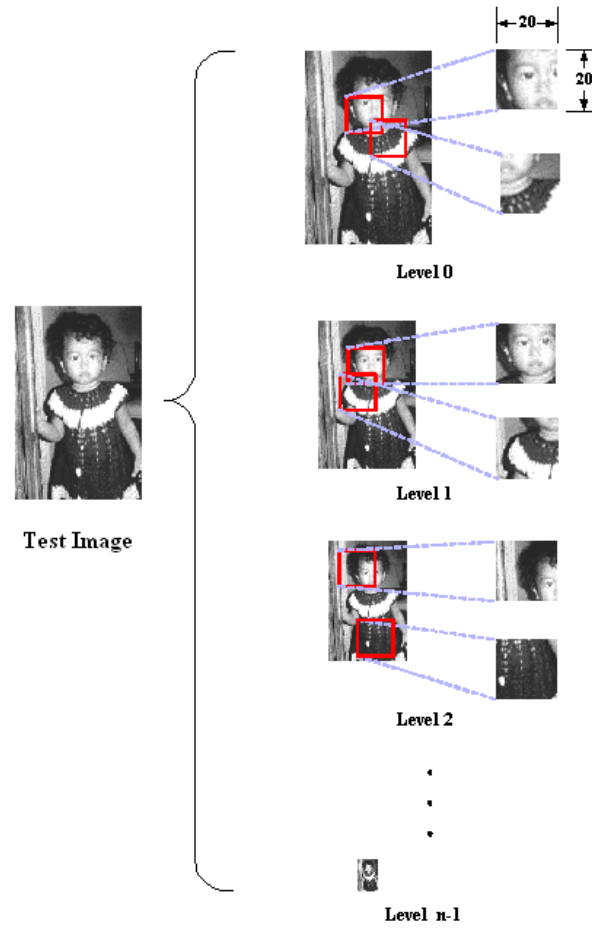


Figure 2.13: Localization of each level of subsampling process [17].

subsampling was done repeatedly as long as the size was still larger than 20×20 . Afterward, the 20×20 detector was applied to each scaled image. Each 20×20 pixel window then became the input of the neural network in this application after going through pre-processing step.

The drawback of using a detector window with a size of 20×20 pixels at every position of the image is that there were too many windows to be processed by the network. This slowed down the execution time. A 30×30 pixel detector was created to let the potential face candidate to be off-center by five pixels in any direction but the whole face still be seen. Using this detector, the window did not have to be moved over every pixel across the image. Instead, it could be moved every 10 pixels, resulting in decreasing the number of windows to be processed by the network.

The pose estimation process was applied mainly for a non-upright frontal face. The image was rotated, both in- and out-of-plane, so that the correct detector could be applied to the image. Then the rest of face detection algorithm is followed.

2.5.2 Pre-Processing

The pre-processing step includes histogram equalization and lighting correction. This step was performed to improve quality of images taken by a camera and to enhance network training and detection.

Histogram Equalization

An image histogram provides information on the number of pixels of each gray-level intensity [7]. The relationship displayed in figure 2.14. The horizontal axis denotes the gray-level value, and the vertical axis denotes the number of pixels. From the relationship between the gray-level intensity and the numbers of its pixels, the histogram tells about the image contrast. The objective of histogram equalization is to flatten the histogram. That is, ideally, the numbers of pixels in each gray-level value in the image are level. This can be done by redistributing the pixels uniformly over the image and thus, improving the brightness and contrast of the images.

Lighting Correction

Lighting correction was applied to remove extreme lighting conditions in an image. Because of the different structures of the human face, light is unevenly scattered when it hits the face from a camera. An example of bad lighting conditions is the extreme difference in intensity from the left side to the right side of a face. The correction was done by adding particular lighting models to the original image, in which the models were generated from the same image but under different lighting conditions such as a different angle of lighting. The new corrected image appeared as if its lighting source comes from the front of the face. The lighting correction was mostly used to remove a strong shadow on a face.

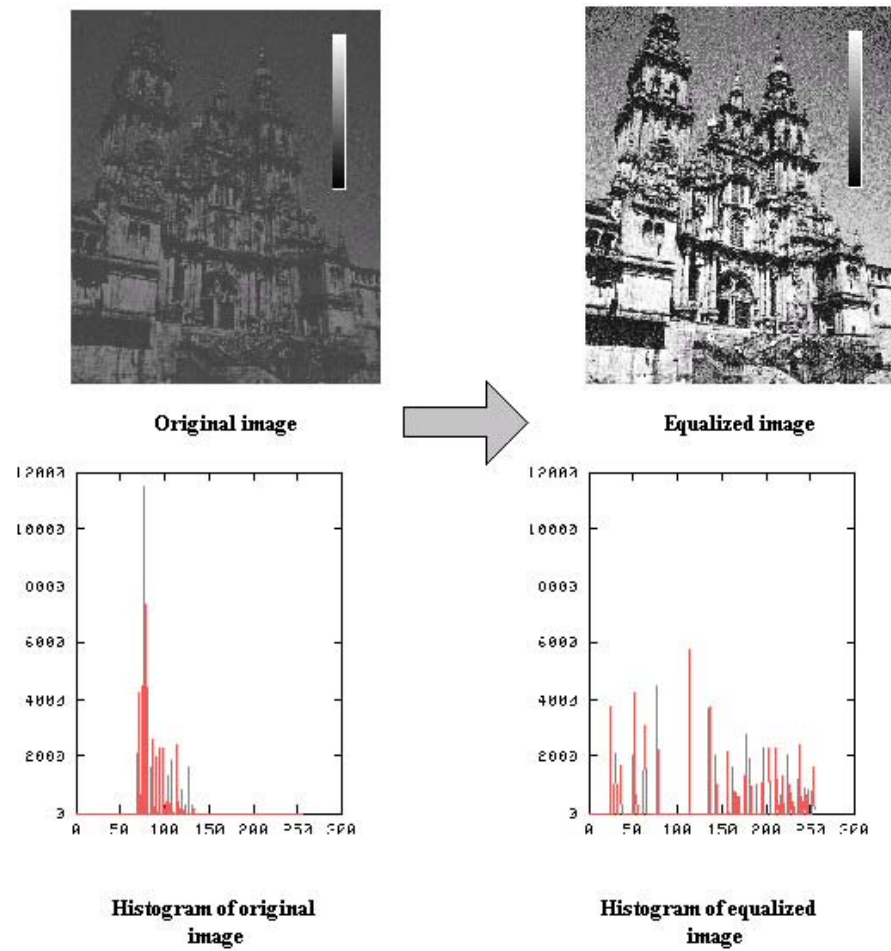


Figure 2.14: Histogram Equalization [3].

2.5.3 Neural Network Detection

After finishing the pre-processing step, the extracted window proceeds to the neural network stage. There was more than one set of neural networks used in this project, each of which has different structures. Output of individual networks had a real value with a range between -1 to +1 denoting if the window contains a face or not. The idea behind using multiple networks is to reduce the possibilities of missed detection and false detection.

Generally, all the networks used had similar systems. They were divided into groups of smaller regions called **receptive fields**.

Figure 2.15 shows an example of a 30×30 detector window. This network contained three groups of receptive fields. The first group had nine 10×10 pixel regions. They detected face features such as an eye, nose, and the corner of the mouth. The second group, had five 5×30 , that were used to detect features including a pair of eyes and mouth. The third group of receptive fields was five 30×5 , which represented the same features as of the second group. Each receptive field was connected to one or more hidden units, which were similar to neurons in the hidden layer. The value of each hidden unit was passed to the output layer for computation of the final network output. There was a possibility that there were other hidden layers before the output layer. In that case, the first layer hidden units had to pass their values to the next hidden layer. Then the outputs from the last layer hidden units were passed to the output layer.

The final output of each network was a real value in a range between -1 to +1, which is called *rating value*. A threshold was applied to each network so that if the

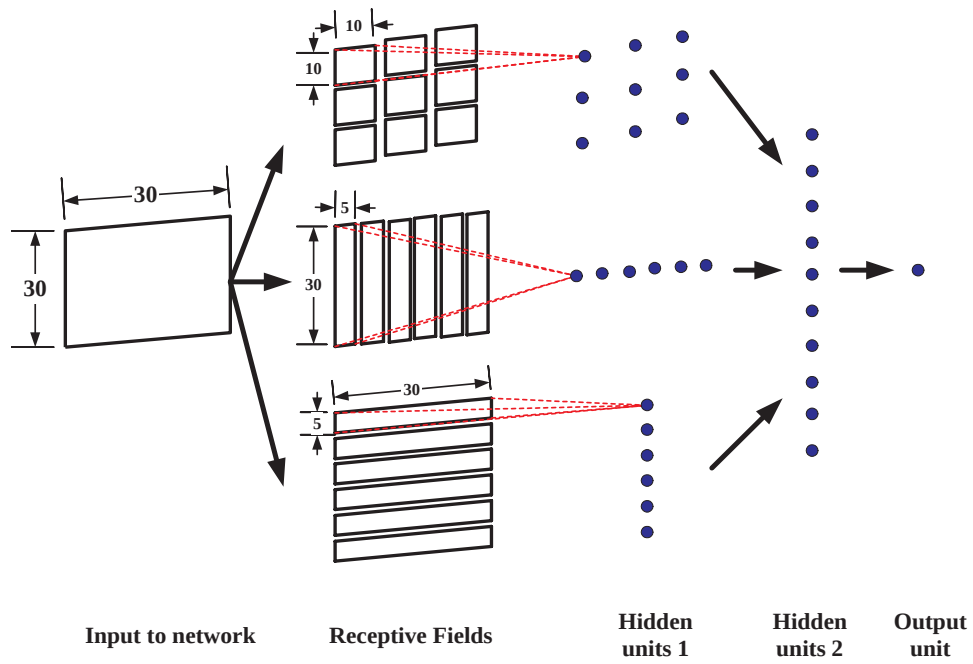


Figure 2.15: Receptive field of 30×30 window detector [17].

network output was greater than the threshold, the input window contains a face. On the other hand, there was no face found in the input window when the network output was less than the threshold value. When there were potential faces detected, the network displayed the information on the number of faces found, the face location within the image, and the rating value of each face.

2.5.4 Post-Processing

The post-processing step consists of two parts: individual network post-processing and multiple networks post-processing.

Individual Network post-processing: Clean-up Heuristic

The clean-up heuristic was mainly responsible for eliminating multiple detection of a single face. Since the neural network detection step was applied over an image on different scales and locations, there was a possibility that one face was detected more than once. For example, the same face was found one time for each scale level. To overcome this problem, a clean-up heuristic was applied to each network by collapsing overlapping detections. Rowley referred to the clean-up heuristic as *thresholding(size,level)*. *Size* represented the size of a particular region computed from the center point of detection, and *level* represented the number of detection in the specified *size*. The idea of *thresholding(size,level)* was that if the number of detections within the *size* was above threshold *level*, the location was classified as a face, and number of detections were reduced to one. Otherwise, if the detection numbers

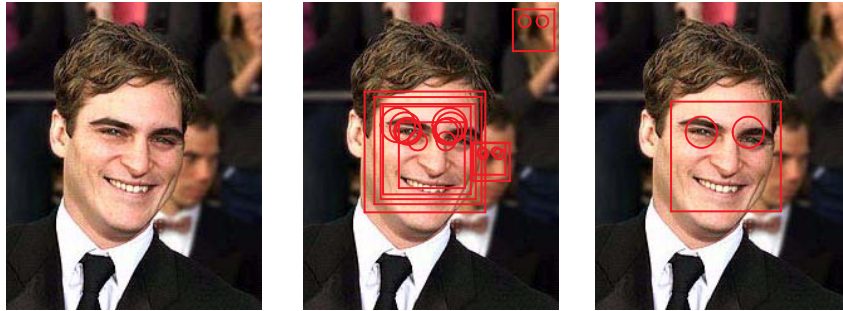


Figure 2.16: Clean-up heuristic using *thresholding(4,2)* [17]

were below threshold level, it was considered false, and detection was removed. In figure 2.16, a thresholding of (4,2) is applied. There are three face candidates, but only one is considered valid, the candidate that is detected more than twice within four pixels.

Multiple Network post-processing

This procedure can be applied before or after the clean-up process. The purpose is similar to the clean-up heuristic, except that arbitration is applied to multiple instead of individual networks. Each network was previously trained using the same algorithm and the same set of face examples. However, the random initial weights and initial non-face images were different for the individual network.

- **AND-ing outputs of two networks** If two networks detected a face at the same scale level and location, the result was a positive detection. Hence, AND-ing algorithm improved accuracy, but decreased detection rate since a face detected by only one network was removed.

- **OR-ing outputs of two networks** A detection was positive if either one or both of the networks detected a face at a particular scale and position. Opposite to AND-ing algorithm, OR-ing two network outputs increased the detection rate, while at the same time, the possibility of false detection increased.
- **Voting outputs of three networks** A detection was positive if at least two of the three networks gave positive results.
- **Arbitration neural network** A new neural network was introduced and trained here. A window of 3×3 pixels was applied to every location that gave a potential face on every image scale level of each network. The number of detections, which were found within the 3×3 pixel window, was used as input to the arbitration network. If output of the arbitration network was positive, a face was detected in the location where the 3×3 pixel window was applied. According to Rowley, this algorithm demonstrated better results than AND-ing and OR-ing but was more complicated in implementation.

This neural network-based face detection system was tested at Carnegie Mellon University. There were 130 images (called the *Upright Test Set*), which contained 507 faces and produced 83,099,211 20×20 pixel windows, used as test sets. Testing was done with different combinations of networks:

- One network with thresholding.
- One network without thresholding.
- Two networks with thresholding.

- Two networks with post-processing using AND.
- Two networks with post-processing using OR.
- Two networks with thresholding and post-processing using AND.
- Two networks with thresholding and post-processing using OR.
- Three networks with thresholding.
- Three networks with post-processing using voting.
- Three networks with post-processing using arbitration neural network.
- Three networks with thresholding and voting.
- Three networks with thresholding and arbitration neural network.

Implementing different composition of networks was used for comparisons in terms of detection rates, false detection rates, number of missed faces, and number of false detects. Recorded results showed that the range of detection rates was between 81.9% to 92.7% and the range of false detection rates was between 1/89546 to 1/10387401. That means, there was between one false detection per 89,546 windows to one false detection per 10,387,401 windows. The number of missed faces ranged between 37 to 97 out of 507 faces, and number of false detects were between 8 to 928. The results showed in [17] implies that the system using one network with thresholding decreased the number of false detections compared to the system with one network without thresholding. However, it also decreased the detection rate. In a system of

two or three networks, the number of thresholding and/or arbitration algorithm chosen were varied for the system to produce the desired output. Note that if a system could increase its detection rate, it would also increase its false detection rate, and vice versa.

2.6 Wildforce-XL Board

As mentioned in the introduction, FPGA technology has been used for implementation of various applications because of its flexibility. It can be reconfigured for specific functions. In this project, a commercially available Adaptive Computing System (ACS) board called Wildforce-XL was chosen. The Wildforce-XL board, which is provided by Annapolis Micro System, contains five Xilinx XC4000 series FPGA. This section will describe the architecture of both the Xilinx XC4000 series FPGA and the Wildforce-XL board.

2.6.1 Xilinx XC4000 Series FPGA

Besides its flexibility, the Xilinx FPGA is chosen because of its low unit cost. The XC4000 family has been developed with more advanced features compared to previous Xilinx FPGA family. One of XC4000's significant features is its ability to configure the device as memory, thus providing on-chip RAM. Another characteristic is increasing system speed. The system can run at clock rates of up to 80 MHz, and internal performance beyond 150 MHz. [6].

The Xilinx FPGA is composed by a matrix of the chip basic functional units called *Configurable Logic Block (CLB)*, which is surrounded by *Input/Output Blocks (IOB)*,

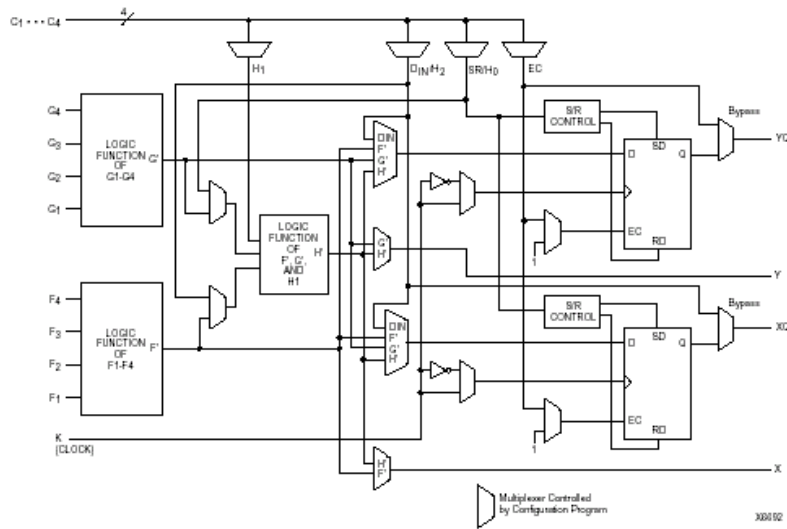


Figure 2.17: Xilinx XC4000 Architecture. Source: Xilinx, Inc., *The Programmable Logic Data Book 1998*.

and equipped with programmable interconnections. The CLB is where most of logic design is implemented. Each CLB of the Xilinx XC4000 series has thirteen inputs and four outputs, and consists of three function generators, two storage elements, and sixteen multiplexers. See figure 2.17.

The three function generators are divided into two four-input function generators and one three-input function generator. Each function generator can be used for implementing combinatorial logic function but only the two four-input function generators can be configured as memory lookup tables. The four-input function generators receive input signals from outside CLB, and their outputs can be passed directly to the next CLB, or used as inputs for the three-input function generators. For the three-input

function generator, two of its input signals are either from outside CLB or outputs of the four-input function generators. However, the last input has to be from outside CLB. Because of the available function generators, each CLB can be configured to implement two functions of four variables plus a function of three variables, or one function of nine variables. When used as memory, each CLB can be configured as 16×2 , 32×1 , or 16×1 array. This on-chip memory feature is one significant advantage of the XC4000 series FPGA.

The two storage elements operating as D-type flip-flops, can be used to store outputs resulted from the function generators or to data coming in from outside CLB. Output of each flip-flop is directed to outside the block. Both flip-flops are edge-triggered and have a common clock and clock enable. In addition, each flip-flop is accommodated with one Set/Reset control. The Set/Reset control determines the state of each flip-flop after configuration.

Multiplexers setting in each CLB define the functionality of the CLB. For example, four two-input multiplexers at the top row in figure 2.17 determines which control signals will be used for CLB operations. When performing function implementation, the multiplexers assign the following signals: enable clock, Set/Reset signal or the first input of a three-input function generator, Direct In signal or the second input of a three-input function generator, and third input of three-input function generator. On the other hand, when CLB is configured as memory, the multiplexers drive the enable clock, write enable, and data inputs to the four-input function generators.

The array of CLBs in every Xilinx XC4000 family has a parameter of Input/Output

Blocks (IOBs), which provide interface between internal logic and external devices. Each IOB is connected to one FPGA pin, and can be configured as input, output, or bi-directional signals. In addition, each Xilinx XC4000 series is supported by programmable internal interconnections. The internal interconnection resources consist of the routings themselves and programmable switching matrices. The switching matrices allow connections between different routings, CLBs, and IOBs.

2.6.2 Wildforce-XL Board Architecture [1]

The Wildforce-XL board contains five of the Xilinx XC4000-XL FPGAs connected together as shown in figure 2.18(a). The first FPGA is an XC4036XL and is called CPE0. The other four of the FPGAs labeled PE1, PE2, PE3, and PE4 are XC4013XL. Differences between the two types of chips are described in table 2.1.

There are several features for the Wildforce-XL board. However, only related features to this project are discussed in this section as described in figure 2.18(b).

As implied by its name, Control Processing Element, the CPE0 holds controls over

Table 2.1: Resources on XC4013XL and XC4036XL FPGAs [6].

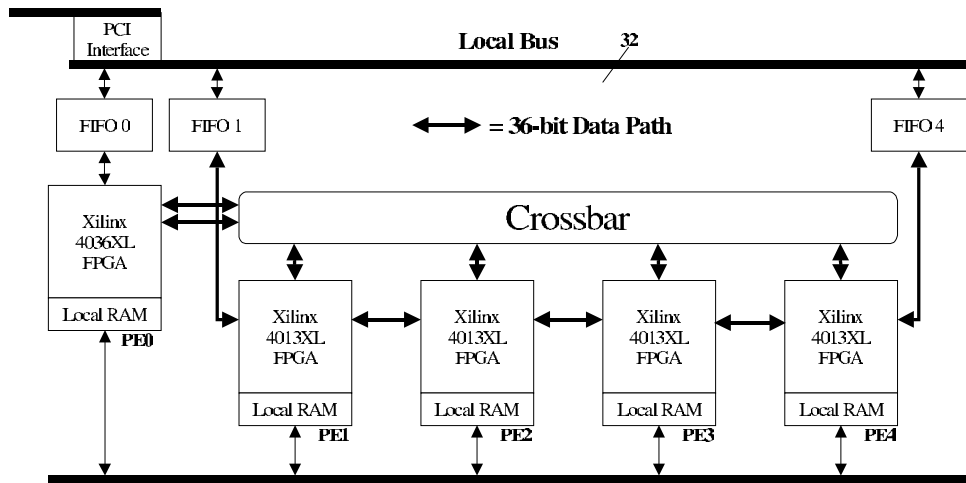
	Logic Cells	CLB Matrix	Total CLBs	Number of Flip-Flops	Equivalent Gate Count
XC4013XL	1368	24×24	576	1536	10,000 - 30,000
XC4036XL	3078	36×36	1296	3168	22,000 - 65,000

some functionality on the board. One example is the use of the board I/O port: certain line signals have to be set so that the board sends and receives data to and from outside the board via SIMD, Single Instruction Multiple Data, connector.

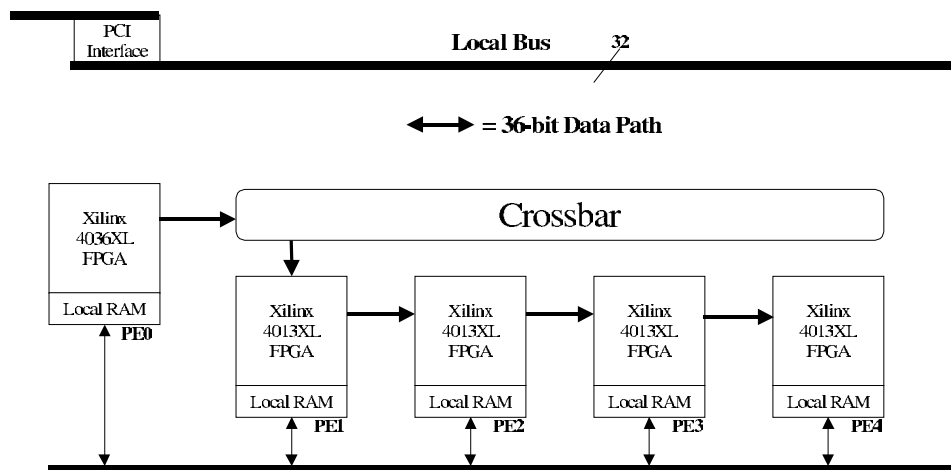
The CPE0 is connected to the other PEs by a crossbar. The 36-bit wide crossbar also allows each PE to selectively talk to any PE. Besides the crossbar, there are also connections for each PE to communicate with its neighboring PE. It is called the systolic bus, and it is 36 bits wide. Both the crossbar and the systolic bus are bi-directional. However, in this project, they are configured for one direction only from left to right. For example, signals and data could be transferred from CPE0 to PE1 or from PE1 to PE2, but signals and data from PE1 or PE2 could not travel CPE0 or PE1 respectively.

The board is connected to the host computer using a PCI interface. A host program, which is written in C, is needed to download the board configuration bit file. It is also used to transmit input data from the host computer to the board and to receive output data from the board to the host computer.

The input and output data to and from the board can be stored in the Static RAM, which is available on the daughter board of each PE. The 32K by 32 SRAM is dual-ported; it can be accessed by both the PE and the host.



(a)



(b)

Figure 2.18: Wildforce-XL (a) Simplified block diagram of Wildforce-XL board (b)

Wildforce-XL as used in CHAMPION.

CHAPTER 3

Methodology

This project is focused on the neural network section instead of the entire face detection application to be implemented onto hardware, while the localization, preprocessing, and post-processing stages were done in a host computer (shown in figure 3.1). The algorithm and coding developed by Henry Rowley were taken through choosing different routines to carry out the face detection application, implementing the neural network part in Khoros Cantata environment by using the floating point and fixed-point data type, utilizing the software CHAMPION to map the designs from Cantata fixed-point to the Wildforce board, and then executing the application on the board. The implementations results were compared afterwards as shown in figure 1.1.

3.1 Choosing Specific Routines of Face Detection System Provided by CMU

Henry Rowley developed different routines for each stage included in his face detection algorithm. For example, in the preprocessing stage, there was a routine, which read any black and white or color images separately, and one that read various angles of any faces located within an image, whether the face was up straight or tilted. The histogram equalization and lighting correction routines were also provided in this first stage. In the neural network computation stage, the main routines were to calculate

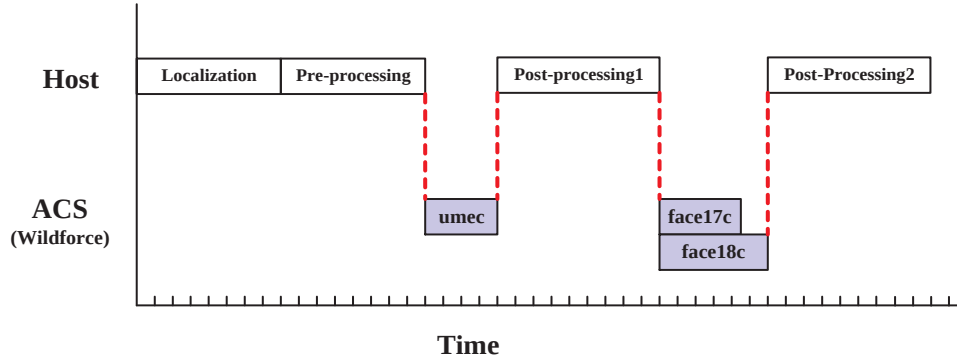


Figure 3.1: Face detection execution for each stage.

the network output given the pixel values of an input image and their corresponding weights. Rowley also provided the information of those networks in terms of which nodes were interconnected. The last stage, which was the post-processing stage, contained procedures to remove overlapping faces detected in the same location and different algorithms for arbitration such as AND, OR, or voting.

To carry out the face detection algorithm, these routines from each stage were put together in an executable program. In fact, there were several programs written by Rowley as examples. Examples are *im_test* which read a color image, converted it to grayscale, and located all faces; *imorient_test* which also converted color image to grayscale, and then detected any faces from different angles; *imgray_test* which read a grayscale image only but had the same functionality as the *im_test*. There was *track_test* which worked similar to the *imgray_test* and which could also locate eyes. One executable program chosen by the author was called *im2_test*. It was a detector which detected upfront faces and read color images as its input. When it found all face

```

% ./im2_test ../test/henry.ppm
Processing image...
Candidate found:
candidate rating: 1 location: (75,75)-(135,135)
verifier rating 1: 0.999922 location: (54,68)-(126,140)
verifier rating 2: 0.999972 location: (54,68)-(126,140)
Candidate found:
candidate rating: 0.99992 location: (62,87)-(112,137)
verifier rating 1: 0.995982 location: (57,75)-(117,135)
verifier rating 2: 0.998604 location: (57,75)-(117,135)
Candidate found:
candidate rating: 0.997902 location: (45,75)-(105,135)
verifier rating 1: 0.999922 location: (54,68)-(126,140)
verifier rating 2: 0.999972 location: (54,68)-(126,140)

```

Figure 3.2: Output example of *im2_test*.

candidates, it displayed the location of the faces and their network ratings. In addition, the program applied two verifiers. Thus, it used three networks. Similarly, the ratings and locations found by the verifiers were printed out. Figure 3.2 presents result of *im2_test*, while the algorithm is shown in figure 3.3.

As mentioned previously, the *im2_test* program utilized a three set networks. The first one was the main network, referred to as the *umec* network. The second and third networks, which were used as the verifiers for the first network, were called *face17c* and *face18c* respectively. The *umec* network used a 30×30 pixel window detector, resulting in 900 pixels as its input. It had two hidden layers and one output layer. The verifiers used a 20×20 pixel window detector, thus having 400 pixels as its input with only one hidden layer and one output layer. In addition, the two verifier networks were arbitrated using the AND-ing algorithm in the post-processing stage. More detailed information on the networks' structure were shown in table 3.1.

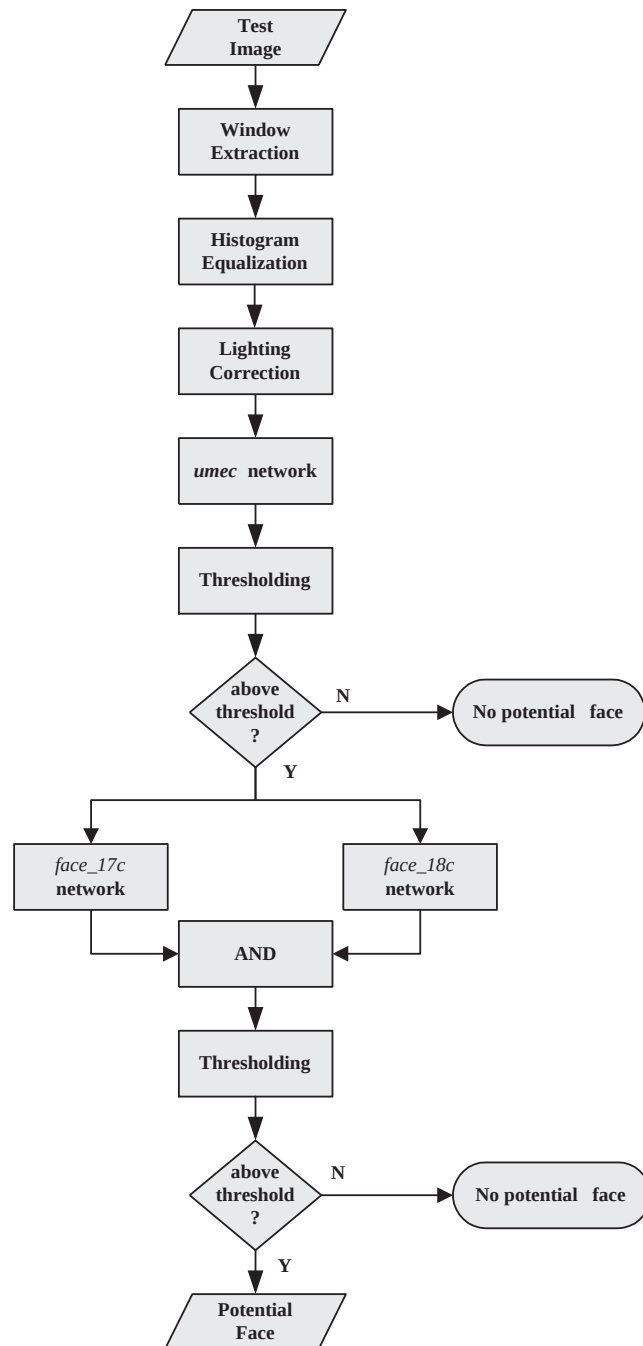


Figure 3.3: The *im2_test* algorithm.

Table 3.1: The structure of different network used in *im2_test*.

Layer Type	umec	face17c	face18c
Neurons in input layer	900	400	400
Neurons in hidden layer1	42	52	78
Neurons in hidden layer2	10	not available	not available
Neurons in output layer	1	1	1
Number of connections	5883	2905	4357

3.2 Floating Point Cantata Design

As the goal of this thesis is to implement a neural network onto the Wildforce board, only the neural network part of the face detection was focused on and was developed in the Cantata environment. It was carried out with the following glyphs: *input stream data* glyph, *weight extract* glyph, *hidden layer computation*, *output layer computation*, *activation function* glyph, and *saved detection* glyph. All glyphs except the activation function were created by using Craftsman, another Khoros graphical user interface program that is specifically provided for glyphs generation. For the activation function, the hyperbolic tangent (*tanh*) glyph available within the Khoros package was used. The flow diagram of this implementation is displayed in figure 3.4.

The *stream data* module received its input from the preprocessing stage including the histogram equalization and lighting correction. Depending on the type of network,

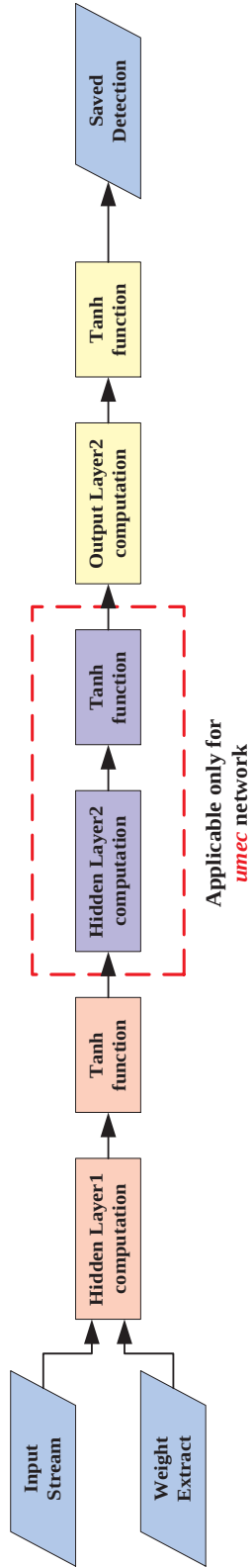


Figure 3.4: Floating point design.

the number of inputs could be 900 from the 30×30 pixel window detector of the *umec* network or 400 from the 20×20 pixel window detector of *face17c* and *face18c* networks. The *weight extract* glyphs, meanwhile, read weight data that were stored in separate files for different networks. These weight files were provided by Henry Rowley as a result of neural network training. Unfortunately, the amount of data for pixel input and weight input were not equal. The number of pixels was determined by the number of neurons in the input layer, while the number of weights was defined by the number of network connections. Because of this circumstance, the multiplication glyph available within the Khoros package could not be used. This led to creating the *layer computation* glyphs, which included multiplication and accumulation. The glyphs were divided into two parts: *hidden layer computation* glyph and *output layer computation* glyph. The layer computation procedure read the given information on neuron interconnections that specified which neurons were interconnected. Using this specification, the author was able to match the pixel value of a certain node with its corresponding weight.

Due to several differences in the three network structures, workspace for each network was implemented separately. However, each of them used the same glyphs: *weight extract*, *input stream data*, activation function (*tanh*), and *saved detection* glyph. They were distinct in the *hidden layer* and the *output layer computation* glyphs.

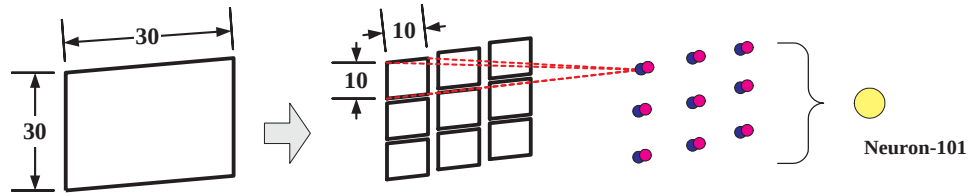
3.3 Fixed-Point Cantata Design

The fixed-point Cantata was similar in idea but different in implementation from the floating point Cantata. Both implemented the neural network element of the face detection, but the fixed-point Cantata design was more detailed because of the resemblance to the hardware modules being downloaded to the Wildforce board. Essentially, it flattened out the glyphs in the floating point design such that the size of each module was not bigger than the available resources in an FPGA, otherwise it would give partitioning problems. In addition, all routines in the fixed-point Cantata glyphs were written in C with an additional library available in the A|RT Builder commercial software package.

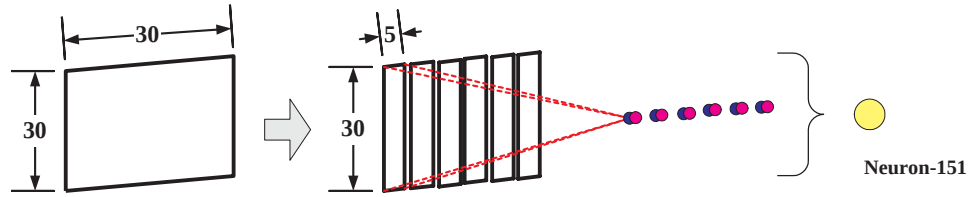
As one neuron is composed of a multiplier, an accumulator, and an activation function, the network designs mainly consisted of glyphs with those functions. The input to the multiplication module was a stream of pixel data and weight data. The stream was arranged so that the data for both inputs were the same, to complicate the process less; hence, providing a faster performance. The outputs from the multiplication process were streamed out to the accumulator then passed to the hyperbolic tangent function.

Taking the *umec* network as an example, it had 42 neurons in the first hidden layer. For efficiency in drawing the flow in the Cantata environment, and to reduce the number of resources to be used, those neurons were reduced in visualization to three neurons according to their corresponding receptive fields as illustrated in figure 3.5. There were the following:

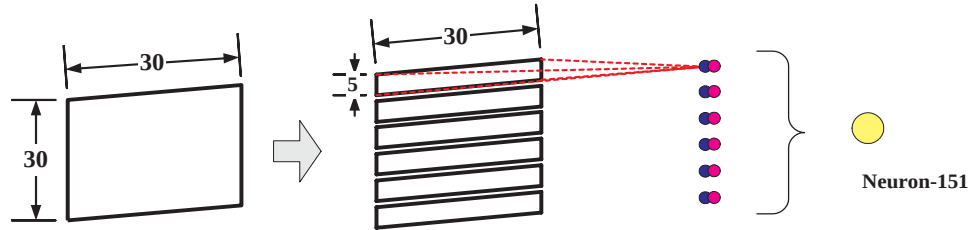
1. First neuron group. It had its inputs from receptive fields of size 10×10 pixels.



(a)



(b)



(c)

Figure 3.5: *Umec* network receptive fields. (a) 10×10 -pixel receptive fields. (b) 30×5 -pixel receptive fields. (c) 5×30 -pixel receptive fields.

From a 30×30 -pixel window detector, there were nine of such receptive fields. Each of them was connected to two hidden units in the first layer. Therefore, this new first neuron originally contained 18 old neurons, and it had 101 input data, which came from 100 pixels plus the input bias. The new neuron of this group was called **Neuron_101**.

2. Second neuron group. It received its inputs from receptive fields of size 30×5 pixels. There were six, each of which was also connected to two hidden units. Thus, the new neuron of this second group contained 12 old neurons and had 151 input data, which was built from 150 pixels plus the input bias and called **Neuron_151**.
3. Third neuron group. It was similar to the group of second neuron, except the size of the receptive fields was 5×30 pixels instead of 30×5 . This new neuron also contained 12 old neurons. It had 151 input data and it was also referred to **Neuron_151**.

After calculations in the first hidden layer, which involved multiplications, accumulations, and activation function processes, the results were transmitted to the second hidden layer. The second hidden layer contained 10 neurons, each of which received its inputs from outputs of every neuron in the first hidden layer. As in the first hidden layer, there was a reduction in number of neurons from 10 to 1. The new neuron was referred to as **Neuron_43** since the number of input data was 43 (42 from neurons in first hidden layer plus the input bias). The result from the second layer was streamed

to the output layer. There was only one neuron in the last layer, receiving 11 input data (10 from the second hidden layer plus the input bias). This neuron in the output layer was named **Neuron_11** and carried out the process of multiplication, accumulation, and the activation function as in the previous layers. Its outcome would be sent to the post-processing stage, which in this case was done by software instead of being carried out on the Wildforce board as shown in figure 3.6.

Several modules were added to the neuron glyphs mentioned previously. The modules were *RAM Read/Write*, *dataselect*, and *datamerge* glyph.

1. The *RAM Read* glyph was created to store input data before streaming out to each neuron in the first hidden layer. Meanwhile, the *RAM Write* glyph was developed to store the output of neuron in the output layer.
2. The *dataselect* glyph purpose was to separate the input data for particular neurons. For example in the *umec* network, there were 5,883 values, in which the first 1,818 values belonged to the first neuron in the first hidden layer and the next two 1,812 values were sent to the second and third neuron separately. The next 430 values were transmitted to neurons in second hidden layer and the last 11 values were sent to the output layer.
3. To convert data entering a glyph in parallels then output them serially, the *datamerge* glyph was created. This process occurred in the first hidden layer. The *datamerge* glyph received its inputs from the three neurons simultaneously, and had to pass them to the next layer in sequence starting from the first neuron, followed by the second and the third neuron.

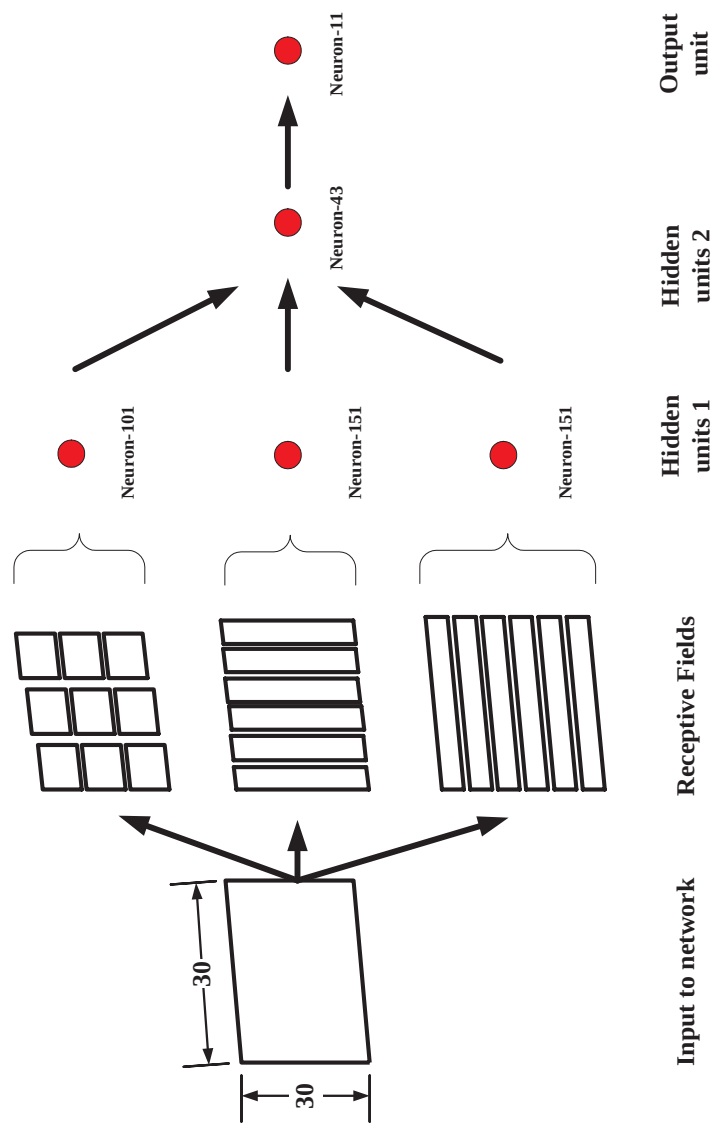


Figure 3.6: New layout of *umec* network.

4. To add the input bias in one layer for the next layer computation, additional glyphs were created, referred to as *layerx_to_layery*. For example, *layer1_to_layer2* glyph was used to add the input bias to layer one output data before being passed to the second layer.

The thorough block diagram of the fixed-point Cantata implementation of *umec* network is shown in figure 3.7.

The other two network implementation, *face17c* and *face18c*, were done in similar ways as in the *umec* network explained earlier. The exception was that there was no second hidden layer in those verifier networks. See figure 3.8 and figure 3.9.

3.4 Glyphs Installation

As the implementation of the neural network section of the face detection in the fixed-point Cantata was used as input to CHAMPION, the next step was to develop modules that corresponded to each glyph created in fixed-point Cantata. These modules were called the hardware modules, which were also referred to as CHAMPION glyphs, were written in VHDL. After each of the CHAMPION glyphs were synthesized and simulated for accuracy, they were installed in CHAMPION by using a certain program called *Geninf*.

3.4.1 VHDL Development

The VHDL code to create CHAMPION glyphs must obey certain criteria. First, the number of input and output ports of a module should be equal to that of the

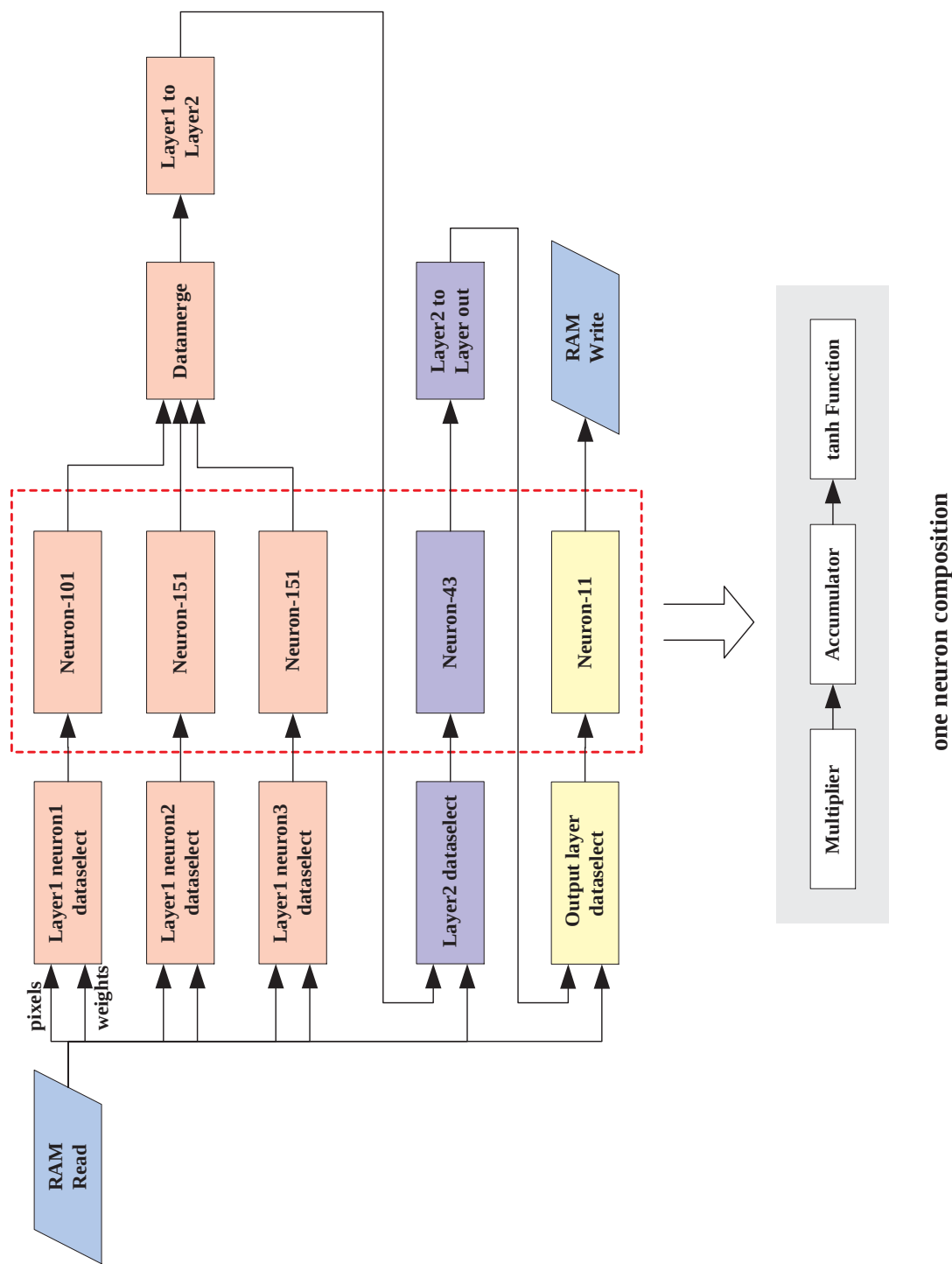


Figure 3.7: Block diagram of *umec* network fixed-point Cantata design.

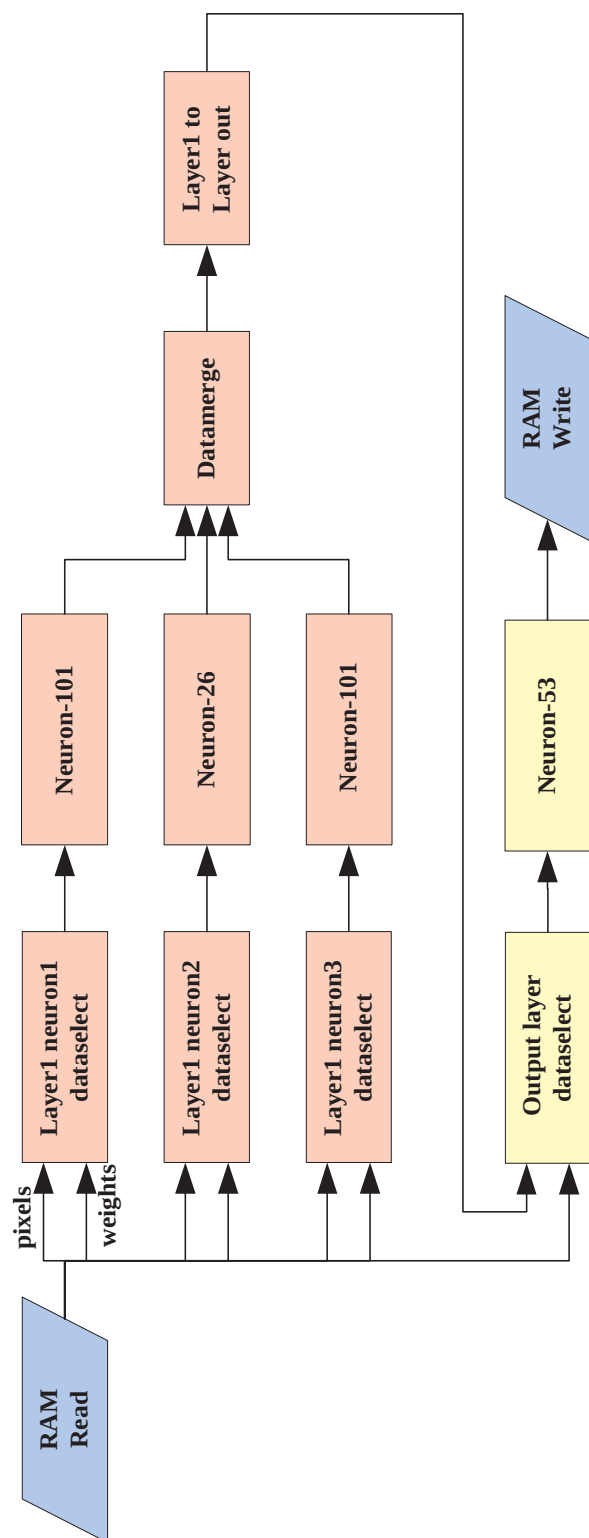


Figure 3.8: Block diagram of *face17c* network fixed-point Cantata design.

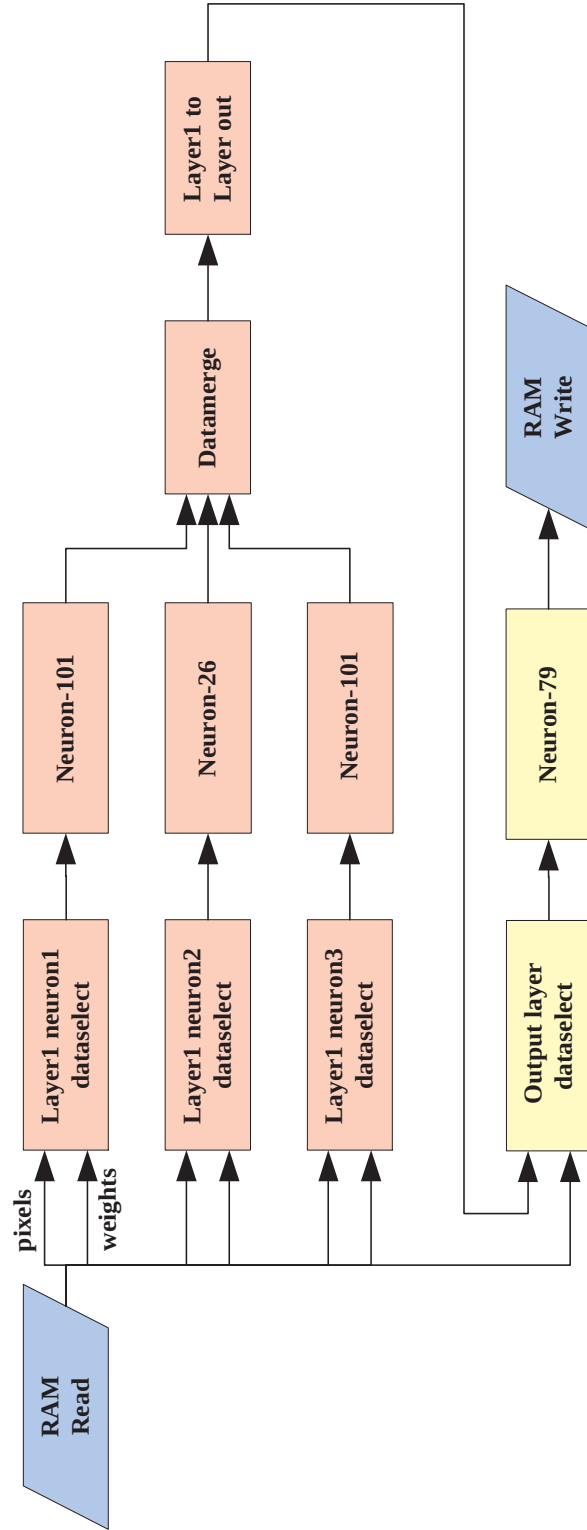


Figure 3.9: Block diagram of *face18c* network fixed-point Cantata design.

corresponding fixed-point Cantata glyph. Also, the name of those ports should be identical between both glyphs. For example, the multiplication glyph for both fixed-point Cantata and CHAMPION contained two input ports named **g_control_a** and **g_a**, and two output ports named **g_control_result** and **g_result**.

Second, three control lines were introduced in CHAMPION glyphs: Stream Valid (SV), Pixel Valid (PV), and Data Valid (DV). The Stream Valid was a control line, which indicated the beginning and end of data stream. A glyph would recognize that a stream of valid data entering the input port when the SV was set to high. As soon as the SV switched to low, the glyph would acknowledge that the last data was the end of the stream. Similarly, after a glyph processes its task, its results were emitted to the next glyph through the output port the same way as the input data entering the glyph. That is, the SV line of the output data was set to high when the first result of the stream was sent out. As the last result was parsed out as well, the SV line was then set back to low. The purpose of Pixel Valid was to inform glyphs whether a pixel value being processed was valid or not. When PV was high, it indicated that pixel was valid, on the other hand, if PV was low then the pixel was invalid. In this project, the PV line was not used since data being processed by glyphs were all data value, but not image pixel value. Even though the data were generated from an image, it had been through a pre-processing stage. The last control line, the Data Valid, was used to suggest glyphs if the data being processed were valid or not. It worked similarly with the Pixel Valid. When DV was set high, its corresponding data was valid, and it was invalid when DV was set to low. Accordingly, after a glyph finished its task, the DV line was used to

inform the next glyph of the validity of its result data. Details about the control lines can be found in [13].

Before the CHAMPION glyphs were installed, they were verified. Each VHDL code was synthesized and was simulated using some commercial software. *Max+plus* from Altera was used for both synthesis and simulation. Another software, the Synplicity's *Synplify*, was used for synthesis only. Its results were simulated using *ModelSim* from Model Technology. Particular glyphs such as the *RAM Read/Write* were tested by downloading and executing them directly onto the Wildforce board.

3.4.2 Using *Geninf*

Geninf was a CHAMPION tool whose purpose was to install CHAMPION glyphs written in VHDL. The installation was defined by an automatic process of synthesizing and generating INF files for individual glyphs. The synthesis was done using Synplify software by calling a batch file for a specific module that was created by *Geninf*. The synthesis resulted in a technology-dependent netlist including XNF or EDIF format. In addition, a report file was generated, which contained some information such as the glyph's size, its number of input and output ports, and the ports bit width. The information was stored in an INF file for each glyph. One type of information in the INF file that was not available from the synthesis process was the clock cycle latency. In this case, it had to be completed manually and was obtained from timing information which resulted from its simulation.

3.5 Running CHAMPION

After the neural network algorithm was implemented in fixed-point Cantata and the corresponding hardware glyphs were installed, the application was ready to be mapped to the Wildforce board using CHAMPION. It went through the front-end and back-end flows as mentioned in Chapter II. The process could be carried out automatically from the first step in the front-end, that is, workspace to netlist conversion, until the last step of the back-end or the place and route phase. It could also be done step by step separately.

Before running CHAMPION, users need to take a few steps. The first step done before invoking the CHAMPION window was to set the CHAMPION environment by calling a Unix shell script *.champion*. This included setting up a path of the executable files and commercial tools used by CHAMPION.

The CHAMPION GUI as shown in figure 2.5 was invoked simply by typing *Champion* at the users unix prompt. After the CHAMPION window appeared, the second step was to fill several parameters needed by CHAMPION:

1. Specify the application to be mapped onto an ACS board. This was done by either creating a new project or opening an existing one. By entering the project name, the directory path of the application was set. The directory included all designs that were captured in Cantata, the VHDL libraries, as well as any intermediate and final results.
2. Select the fixed-point Cantata workspace of the design.

3. Choose the ACS platform where applications were to be mapped: the Wildforce, Wildcard, or the SLAAC board.

Figure 3.10 shows part of CHAMPION GUI where users had to do those steps just mentioned.

Next, CHAMPION was ready for design mapping. This mapping could be done by clicking the *Automatic Mapping* button. Unless there were errors, the CHAMPION flow was carried out automatically from one process to the next. When the process was to be executed step by step individually, it could be done by selecting a particular process block. Its intermediate result could be observed by clicking the arrow located at the right of the process block, as seen in figure 3.11.

The status window was at the bottom of the CHAMPION GUI. It displayed information for each process during execution and after. Such information included execution time, process completion notification, errors report if any, and intermediate results. The status window is shown in figure 3.12.

3.6 Host Code Generation

The place and route phase in CHAMPION generated bit files, which were used to configure the Wildforce board. To download those files onto the board, it required a host program. The host code was written in C, and it called different subroutines or libraries provided by the board manufacturer. These libraries included essential functions to run the application, such as opening and closing the board, downloading

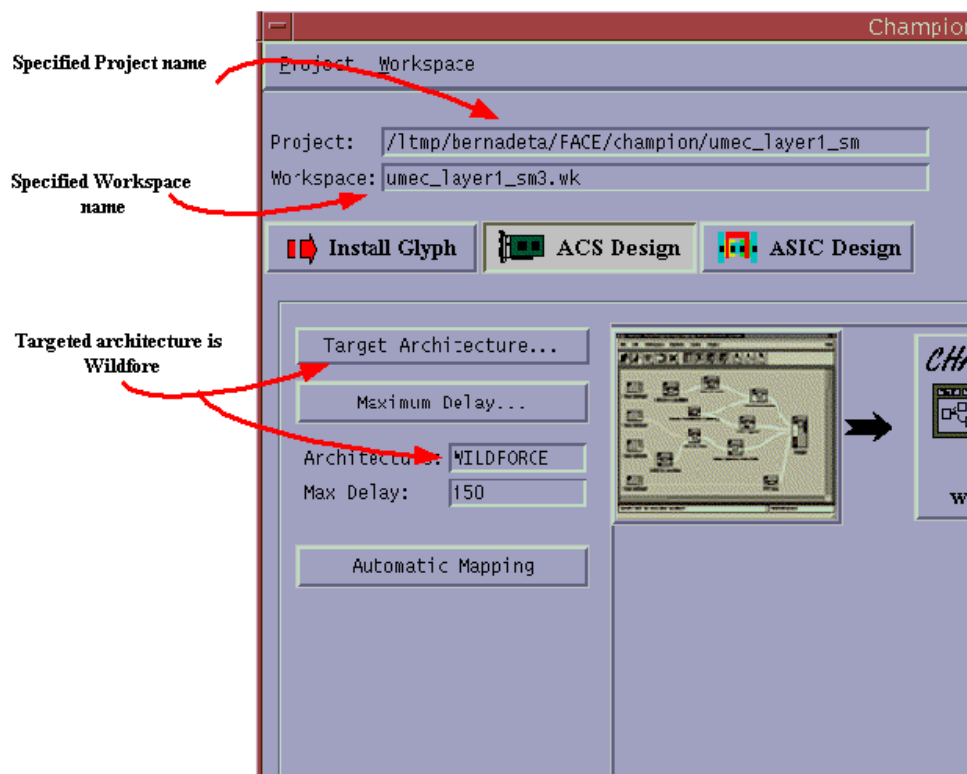


Figure 3.10: Configuring CHAMPION environment.

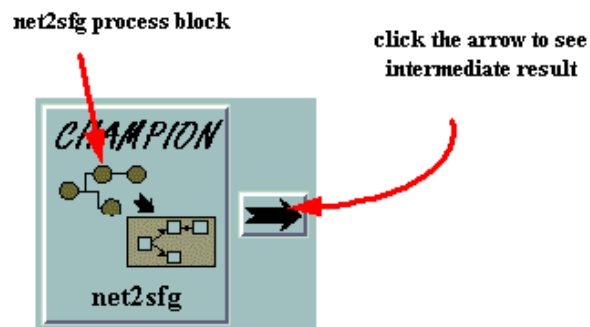


Figure 3.11: Individual CHAMPION process.

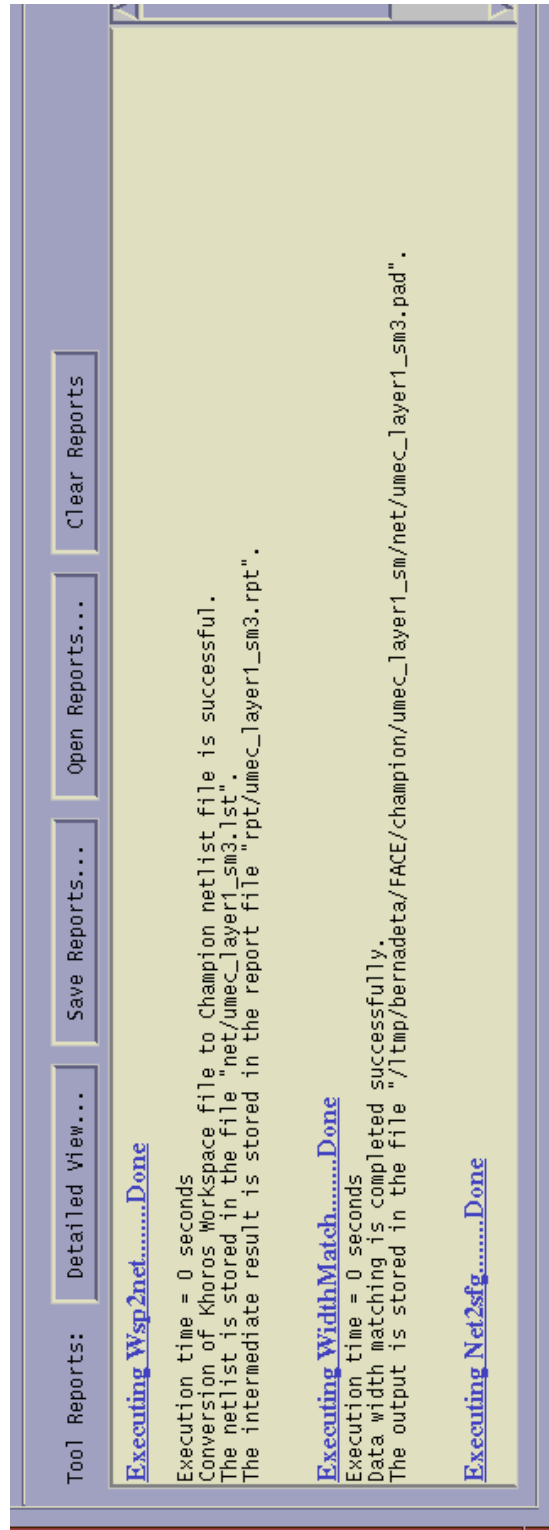


Figure 3.12: CHAMPION process report window.

the bit files, downloading the input data into memory, starting and stopping the clock, as well as reading execution results from memory. The host program was also responsible for transmitting data between the host machine and the board memory. Figure 3.13 shows the pseudo-code of the host code.

Instead of downloading all the pixel and weight data of one image and executing them at once, only one window detector at one time was processed. This is because of the limited capacity of the board memory, which was only 32K, while the size of data of one image could be a minimum of 3Mbytes. There were 5,883 pairs of pixel and weight data for the *umec* network. After one window frame being carried out in the Wildforce board, its result was stored in a file. Then, the next window detector or the next 5,883 values of data were transferred to the board and the board execution was performed again, whose result was appended to the storage file and so on. A similar process was done for the other two network verifiers. In this case, they had 2,905 and 4,357 data sent to the board at one time for the *face17c* and *face18c* network respectively.

MAIN {

```
read in input data from disk
open and initialize the Wildforce board
configure crossbar

setup board for configuration
download PE images
write data to Wildforce SRAM
signal board to begin execution
wait for interrupt signaling completion of execution
read results from Wildforce SRAM

write out result to disk
close Wildforce board

}
```

Figure 3.13: Host pseudo-code for the *umec* network.

CHAPTER 4

Implementation and Results

This section will show the realizations of each methodology mentioned in the previous chapter including several changes that were made in the hardware implementation. Outputs of each implementation were compared to each other, using the original face detection floating point C code written by Henry Rowley for standard results. Only three images were chosen for testing purposes due to lengthy timing in executions. In addition, the Wildforce board was no longer used as the target platform due to several problems that were faced during the implementation process.

4.1 Floating Point Cantata Implementation

Implementing the floating point block diagram in Chapter III, the Cantata workspace for *umec*, *face17c*, and *face18c* are displayed in figure 4.1, 4.2, and 4.3.

Input to the workspace was separated into two categories: pixel data and weight data. The neural network pixel data were obtained from the face detection pre-processing stage. They were stored under the filename *NetFloatData.txt* and were used as input for the *stream data* glyph. The weight data, provided by Rowley, were grouped and saved separately according to the network; they were called *umec_w.wet*, *face17c_w.wet*, and *face18c_w.wet* and inputted to the *weight extract* glyph. The com-

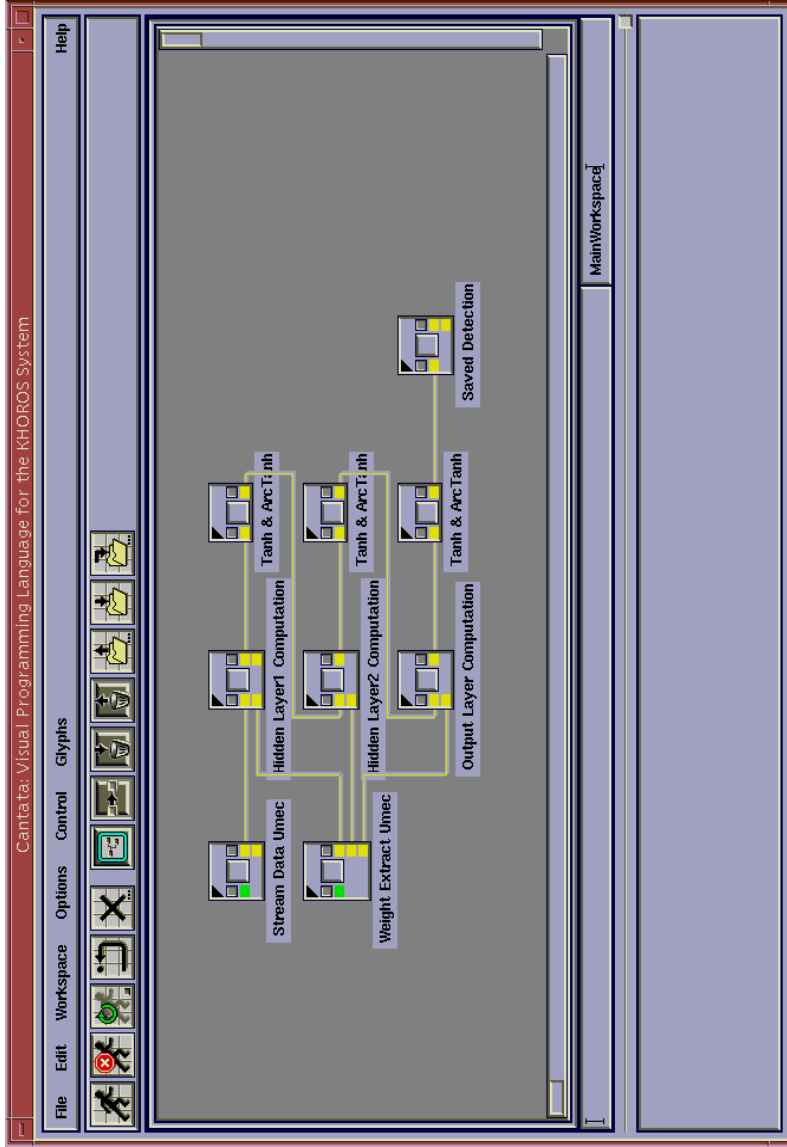


Figure 4.1: Floating point Cantata workspace of *umec* network.

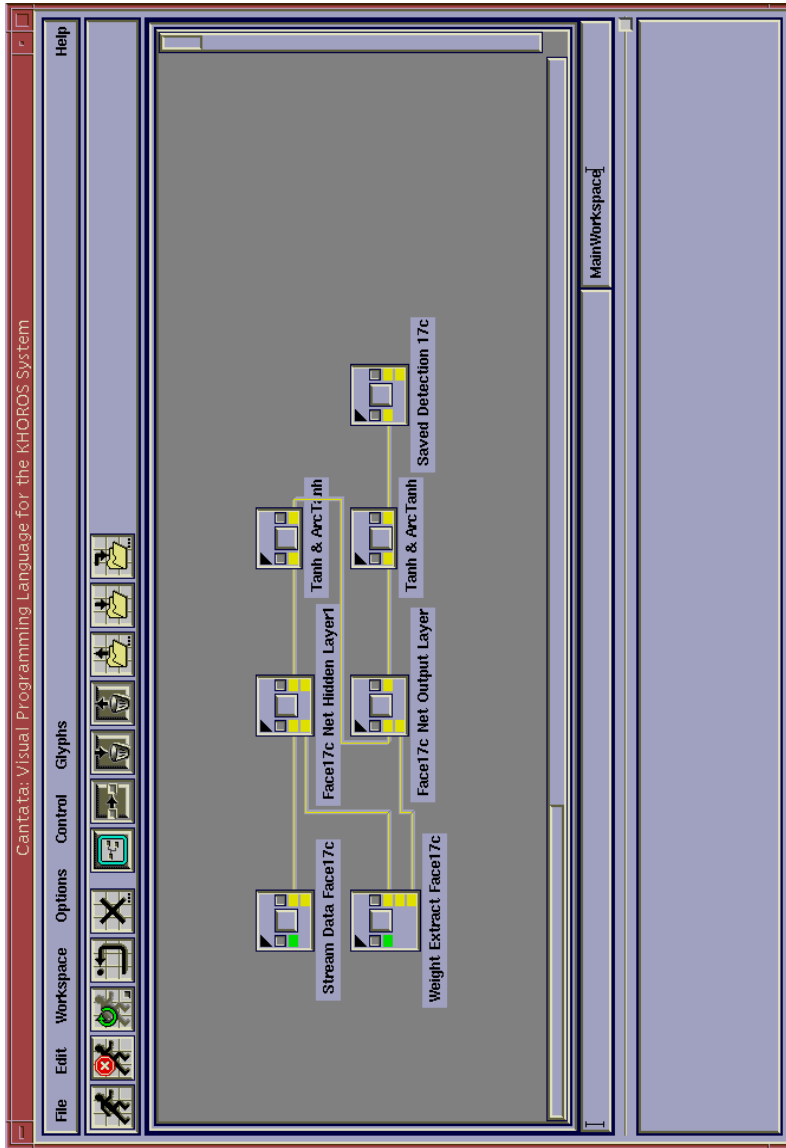


Figure 4.2: Floating point Cantata workspace of *face17c* network.

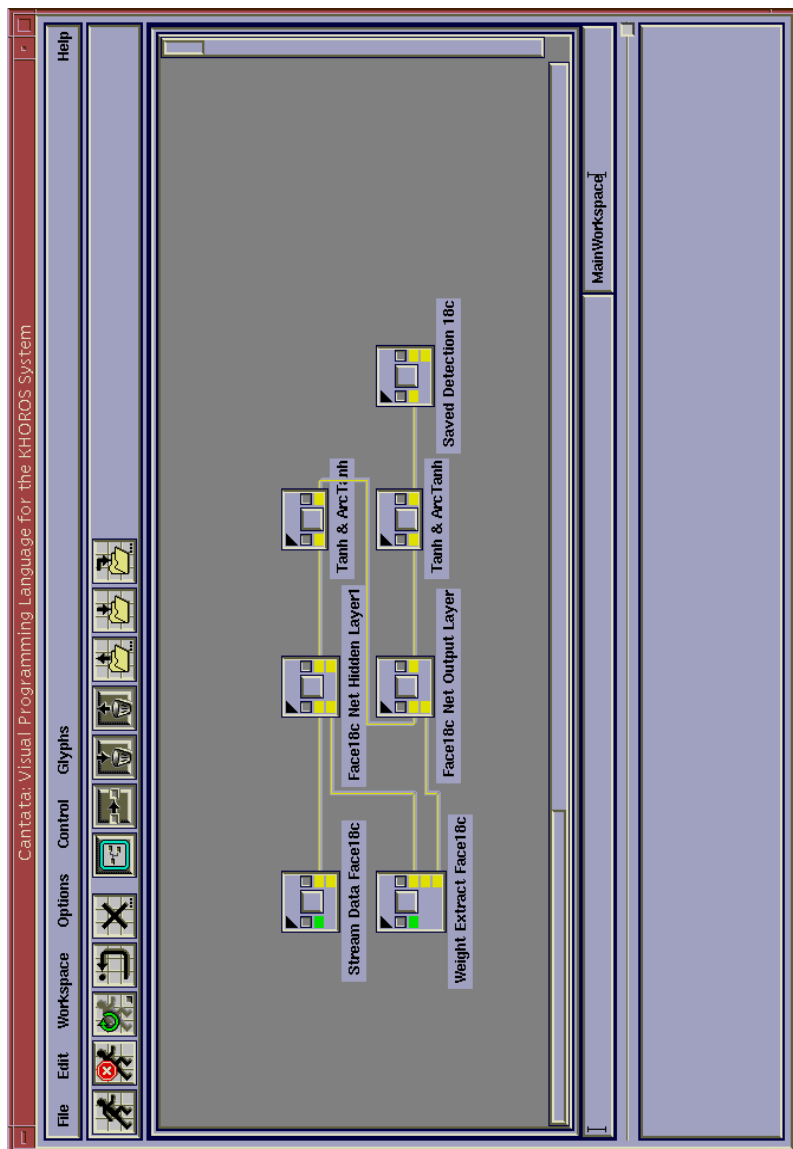


Figure 4.3: Floating point Cantata workspace of *face18c* network.

Net1FloatData.txt	umec_w.wet	SavedDetection1Float.txt
-0.425556	2.799808e-01	-0.999999
-0.453333	1.593433e-01	0.614326
-0.467778	6.375033e-02	-1.000000
-0.467778	-7.918153e-02	-0.999988
-0.434444	2.921627e-03	-1.000000
-0.400000	-2.020814e-01	-1.000000
-0.367778	-3.060105e-01	-1.000000
-0.362222	-2.993781e-01	-0.999996
-0.375556	-7.310571e-02	-0.999292
-0.375556	-1.024048e-01	-0.850911

Figure 4.4: Partial data of files containing pixel values, weight values, and output in the floating point implementation.

putation results were saved under SavedDetectionFloat.txt and were retrieved from the output port of the *saved detection* glyph. These numbers were then passed to the post-processing stage of the face detection system to locate some faces inside a test image. Figure 4.4 shows pixel and weight values as well as the neural network calculation results processed in the *umec* network.

4.2 Fixed-Point Cantata Implementation

The Cantata workspace realizations of the fixed-point block diagram discussed in the Methodology chapter are shown in figure 4.5, 4.6, and 4.7 for *umec*, *face17c*, and *face18c* networks.

The input data were stored differently from the floating point implementation. Instead of having two separate files as inputs to the workspace, the pixel and weight data were combined in one file since in the hardware implementation they were intended

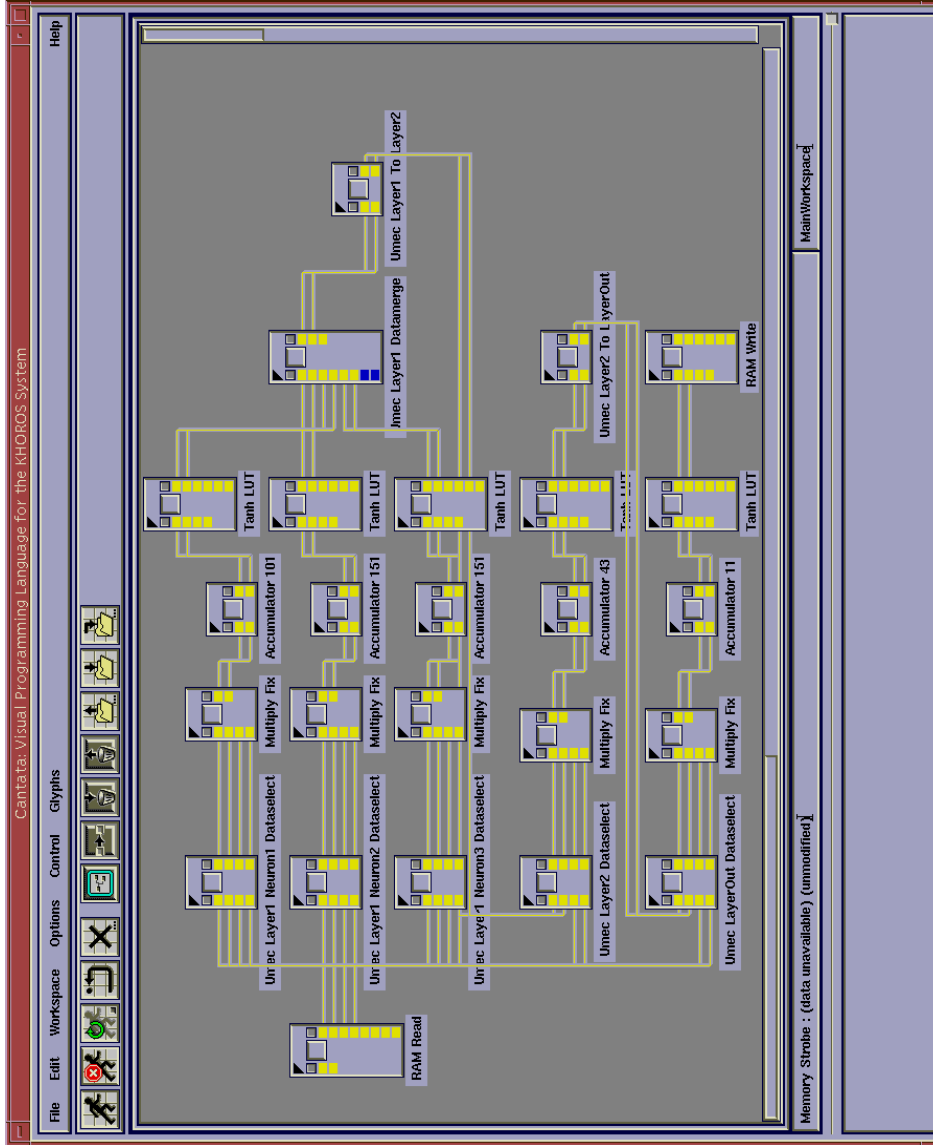


Figure 4.5: Fixed-point Cantata workspace of *umec* network.

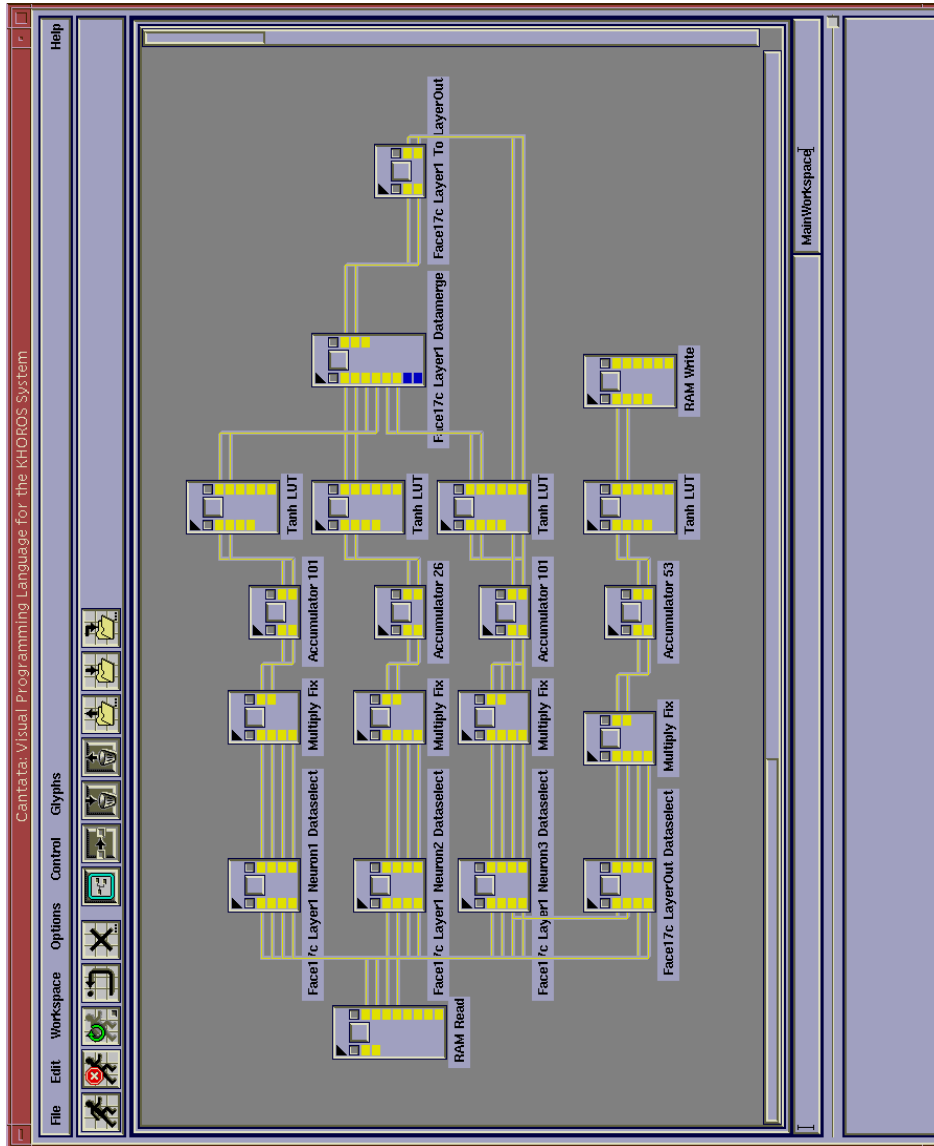


Figure 4.6: Fixed-point Cantata workspace of *face17c* network.

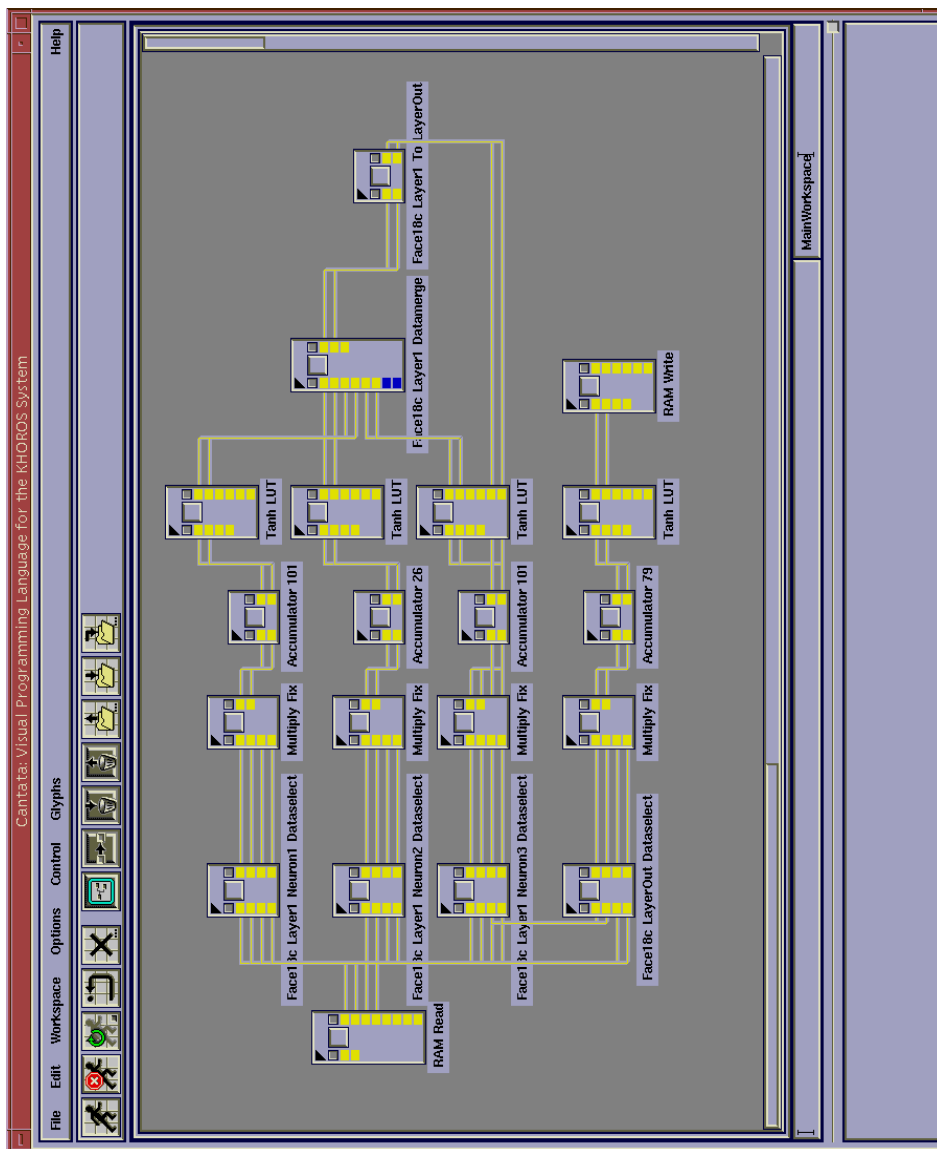


Figure 4.7: Fixed-point Cantata workspace of *face18c* network.

Net1FixData.txt		SavedDetection1Fix.txt
-0.822265625	0.0185546875	-1
-0.822265625	0.0556640625	-1
1	0.9091796875	0.9619140625
-0.822265625	0.2275390625	0.6943359375
-0.587890625	0.0869140625	-1
-0.4345703125	0.0068359375	-1
-0.240234375	0.208984375	-1
-0.0517578125	0.1171875	-1
0.0009765625	0.064453125	-0.990234375
-0.087890625	0.271484375	0.9990234375


data


weight

Figure 4.8: Partial data of files containing pixel values, weight values, and output in the fixed-point implementation.

to be downloaded in pairs onto the Wildforce SRAM. The input file, named NetFixData.txt, was streamed to the *RAM Read* glyph in the workspace. The output data, which were obtained from the *RAM Write* glyph, were saved under SavedDetection file and to be processed in the post-processing phase for face detection. Figure 4.8 represents partial data examples of input and output values. The results of this implementation are presented in the result section.

As mentioned in Chapter II, in creating fixed-point Cantata glyphs, an extra library was added besides the existing Cantata libraries. By calling the fixed-point library, a specific data type was defined as needed in this project. Since the Wildforce FPGA SRAM width was 36 bits, it was decided to use only 34 bits. Each pixel and weight data occupied 17 bits, where 14 bits was taken up for the data values and the remaining 3 bits were filled by the control lines. Only the 14 bits of data line used the fixed-point library as seen in figure 4.9. The fixed-point data type was declared as **Fix**< 14, 10 > ,

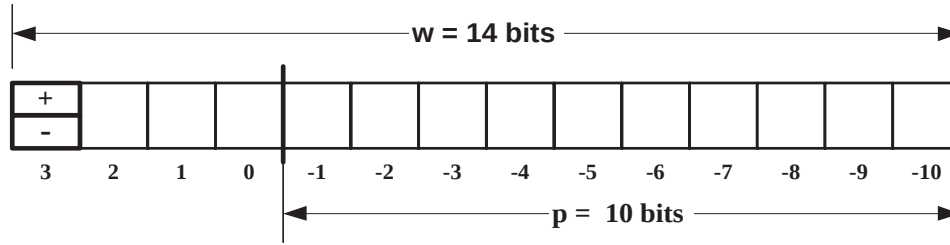


Figure 4.9: Data type **Fix**< 14, 10 >, 14-bit width and 10 bits of precision.

in which fourteen was the total number of binary digits, and ten was the fractional length or the precision, that is, number of binary digits to the right of the imaginary binary points. The most significant digit represented sign of the number.

4.3 Fixed-Point Hardware Implementation

While implementing each network workspace onto the ACS through CHAMPION, two major changes had to be applied. The first change was due to partition problems. The second change involved altering the targeted hardware platform. Instead of implementing designs onto the Wildforce board, they were mapped on a Xilinx FPGA Virtex-E chip. In addition, its results were obtained from pre-layout and post-layout simulations instead of from the actual hardware execution.

4.3.1 Hardware implementation first changes

Before the entire workspace of each network was processed by CHAMPION, a small number of input data were used for testing purposes. For instance, for *umec* network,

instead of using 5,883 pairs of pixels and weight data, it only 60 were used. With fewer data, CHAMPION completed the front-end stages successfully. However, a problem surfaced when CHAMPION partitioned the workspace. The *umec* network workspace had violated one constraint. The bandwidth of some cuts, where partitions occurred, exceeded the bit width between processing elements, which is 36 bits. Figure 4.10 describes this violation in the *umec* network. In the partition violation A, there were three netlists connected between output ports of three neurons in the first hidden layer and input port of the *datamerge* glyph. Each netlist was 17 bits wide, which gave 51 bits at this particular partition. A similar explanation applies to the partition violation B.

The first step to overcome this problem was to split the *umec* network into two configurations or two separate workspaces. This was done to fix partition violation B. The first workspace contained the first hidden layer computation, while the second configuration had the second hidden layer and output layer calculations as demonstrated in figure 4.11.

The second solution, which solves violation A, was to divide the 3-input *datamerge* glyph in the first hidden layer into two 2-input *datamerge* glyphs, named *datamerge_12* and *datamerge_123* as displayed in figure 4.12. The *datamerge_12* combined output data of the first and second neurons. Next, its result was then cascaded with the third neuron data using *datamerge_123* glyph.

Some modifications were also made to reduce the number of resources used in the configuration. Those changes were applied in the first hidden layer. One of the

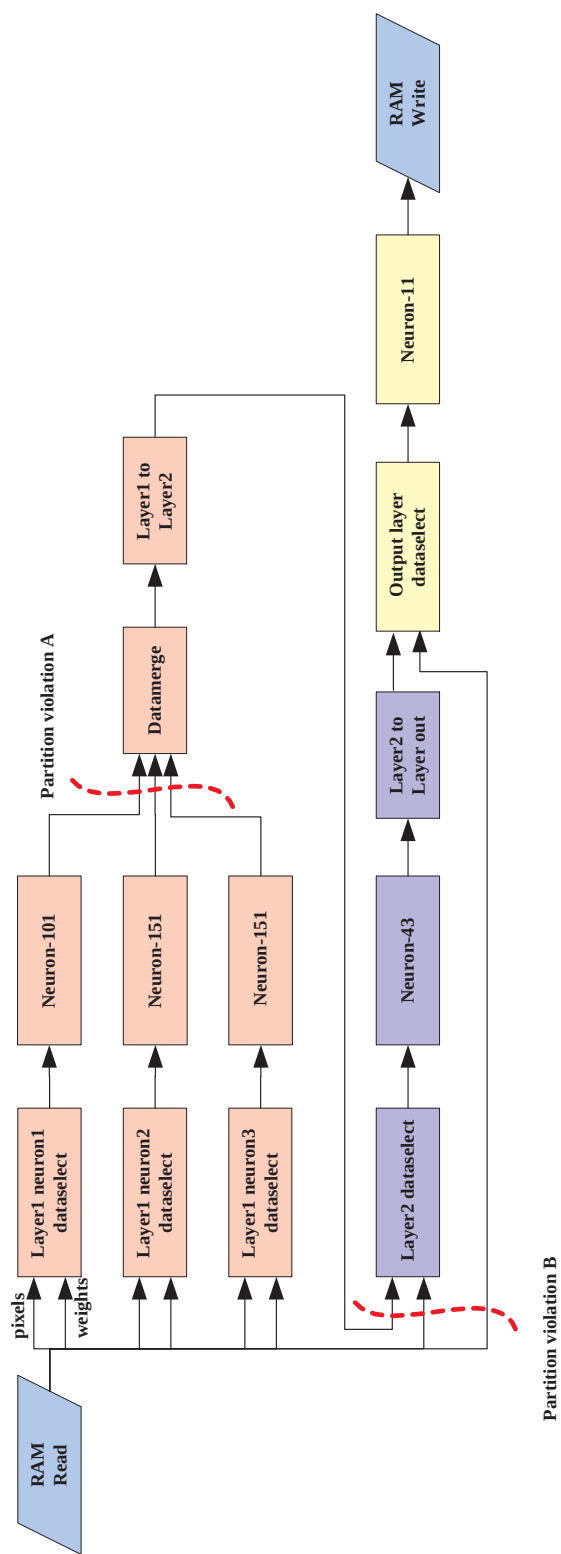


Figure 4.10: Partition violation.

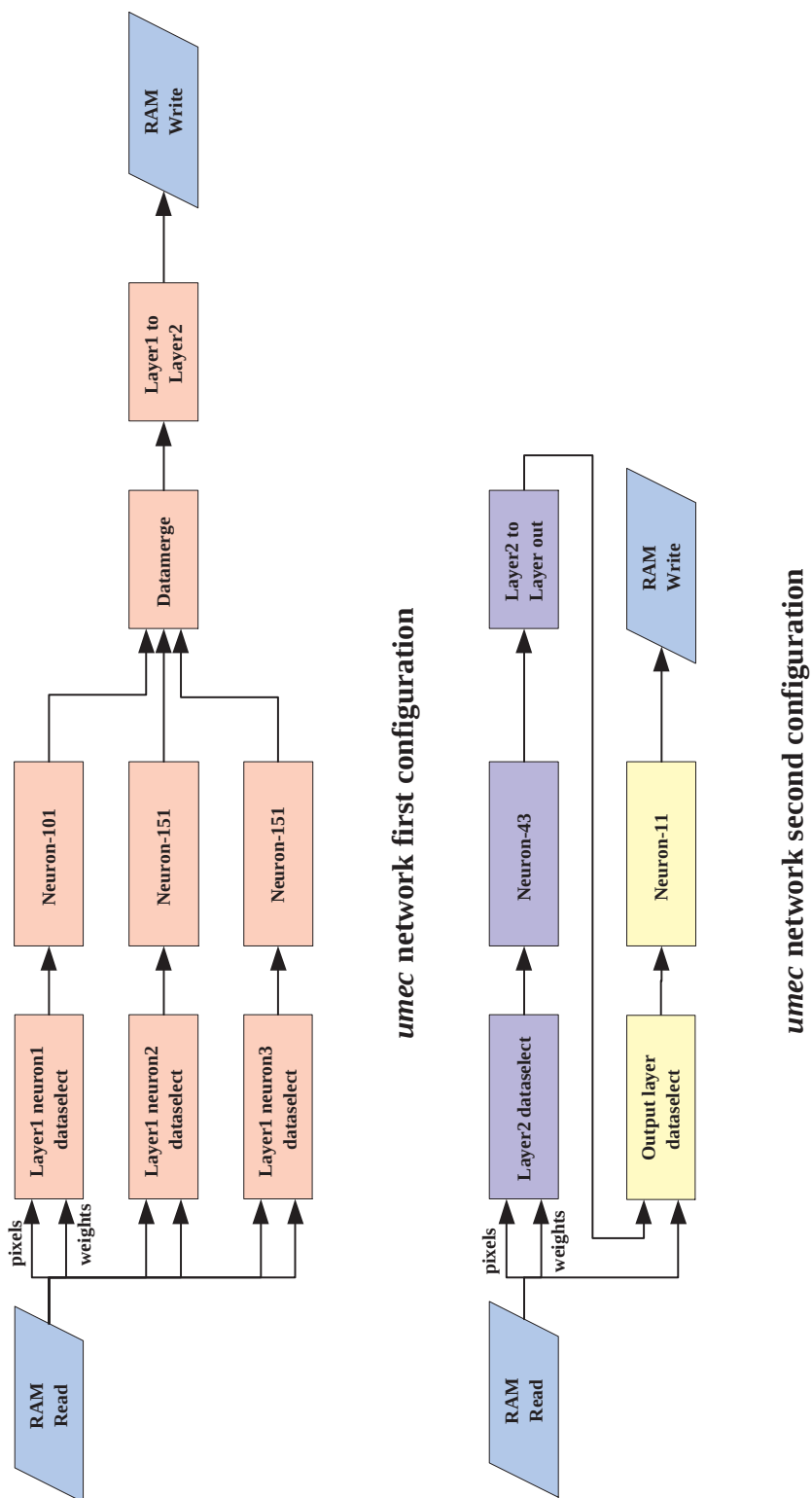


Figure 4.11: *UmeC* network sub-configurations.

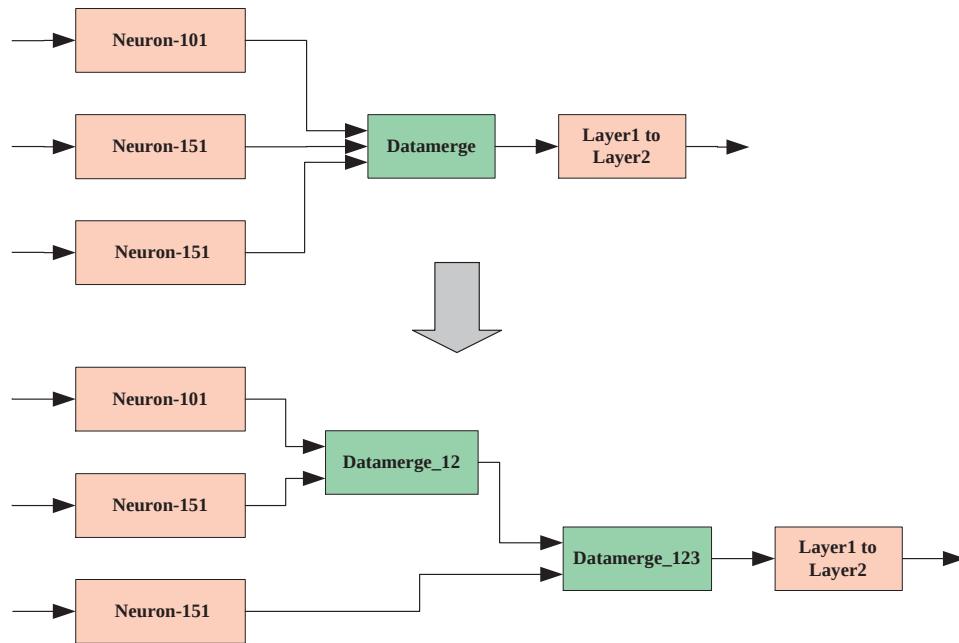


Figure 4.12: Dividing one 3-input datamerge glyph into two 2-input glyphs.

changes was to reduce the number of multipliers from three to one. One multiplier occupied 212 CLBs, hence the configuration saved 424 CLBs (2×212 CLBs). The new multiplier was then positioned after the *RAM Read* glyph. That is, the pixel data and their corresponding weights were multiplied before, instead of after, being separated into different neurons. A similar change was made for the activation function that was implemented using the FPGA RAM Read. Using the RAM as a lookup table, the RAM address was the input to the glyph, while data stored in the memory were the hyperbolic tangent values. As there were three tanh RAM Read glyphs in the first configuration, which took up three RAM hardware, the author decided to cut the number of tanh nodes to one. The new node was placed after the *datamerge_123* glyph. Thus, the activation function was applied after, instead of before, data from the three neurons were passed serially to the next layer. The new modified configuration with the reduction number of multipliers and tanh glyphs is presented in figure 4.13.

An additional glyph, called *register*, was created in this implementation and it was located after each accumulator. The purpose of this register glyph was to remove any intermediate accumulations and to transfer each final result to the next glyph without any delay, which occurred during the accumulation process. Figure 4.14 should explain this idea clearly using the example of *accumulator_11* and *register_3*. The total amount of input data is 33, and they are accumulated after every 11 numbers, resulting in three final outputs. There are 10 clock cycle delays between any two final accumulations. By storing each final accumulation in a specified buffer, the *register_3* will send only those final results continuously.

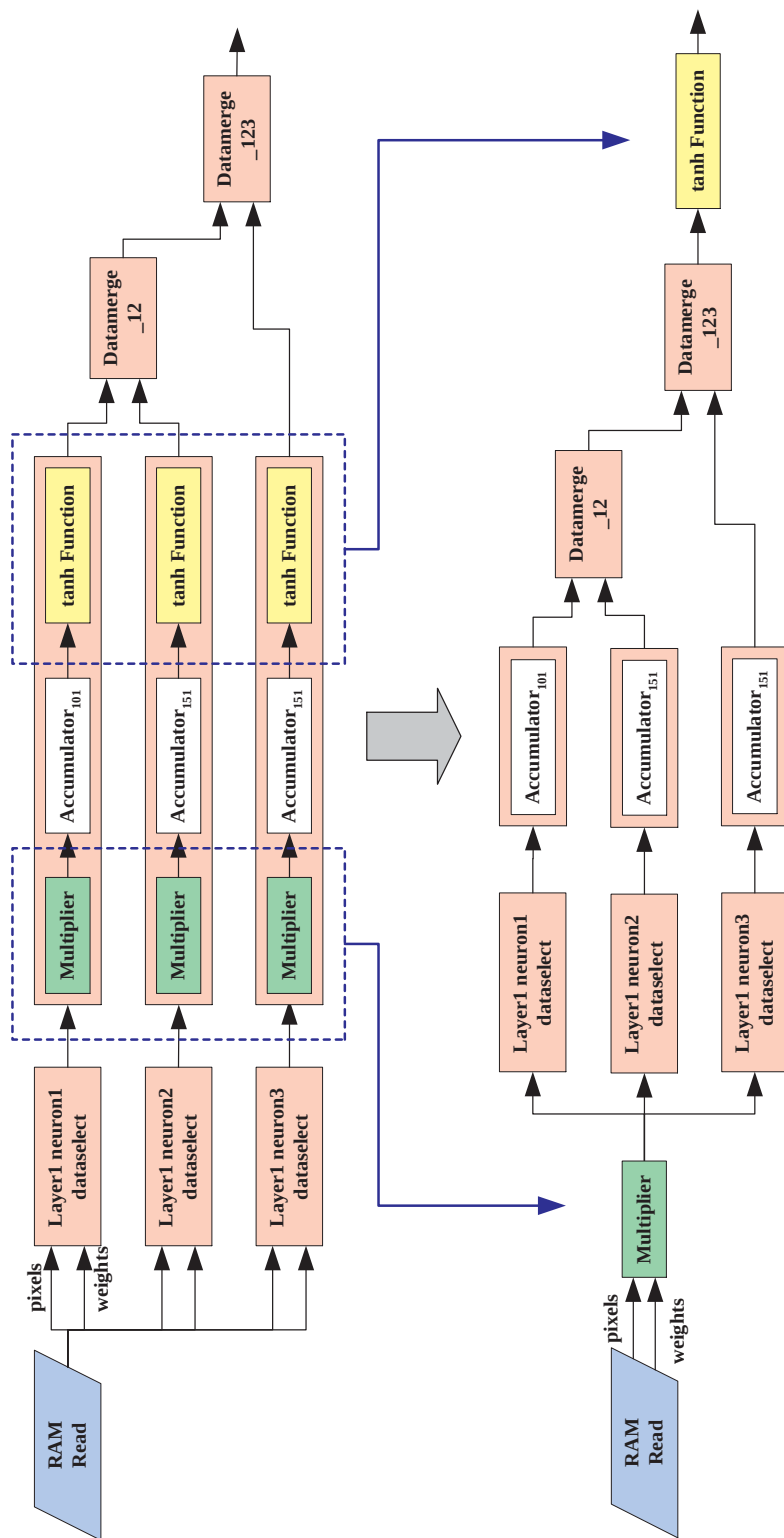


Figure 4.13: Reducing and locating the multiplier and hyperbolic tangent function glyphs.

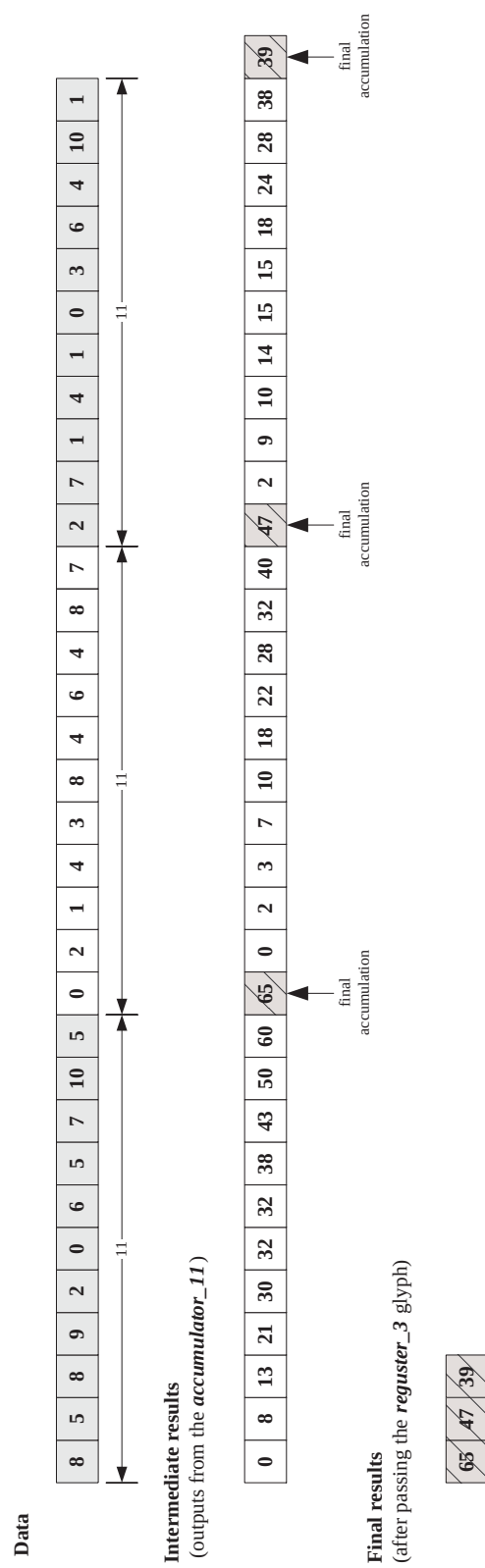


Figure 4.14: The use of register module.

With all the changes made, the new configuration of *umec* network is shown in figure 4.15 for the first workspace and figure 4.16 for the second workspace.

4.3.2 Hardware implementation second changes

With the new modified configurations and fewer inputs mentioned in the first changes, CHAMPION had successfully partitioned each workspace. Even the first workspace could be mapped and executed on the Wildforce board. The next task was continued by using the correct number of input data lines for each design. Unfortunately, CHAMPION was unable to complete the delay generation step. This was because in the CHAMPION ACS flow, the delay was generated by COREGEN. COREGEN was only able to create a delay up to 600 clock cycles. Each neural network, however, needed one or more delay glyphs between 800 to 1800 clock cycles.

One alternative solution was to cascade several glyphs with smaller clock cycle delays. For example, to create a delay glyph of 1200, two glyphs of 600 delays could be used. However, CHAMPION was not capable of carrying out such a solution. Also, those glyphs could take up a large number of resources in the Wildforce board. For instance, one processing element contains 576 CLBs, while the 1200 delay glyph requires 2×257 CLBs because a 600 clock cycle delay 14 bits wide occupies 257 CLBs. This circumstance could trigger the partition problem since the total number of CLBs for one workspace is more than the available resources. Furthermore, CHAMPION is unable to partition into multiple configurations, in which case this must be done manually.

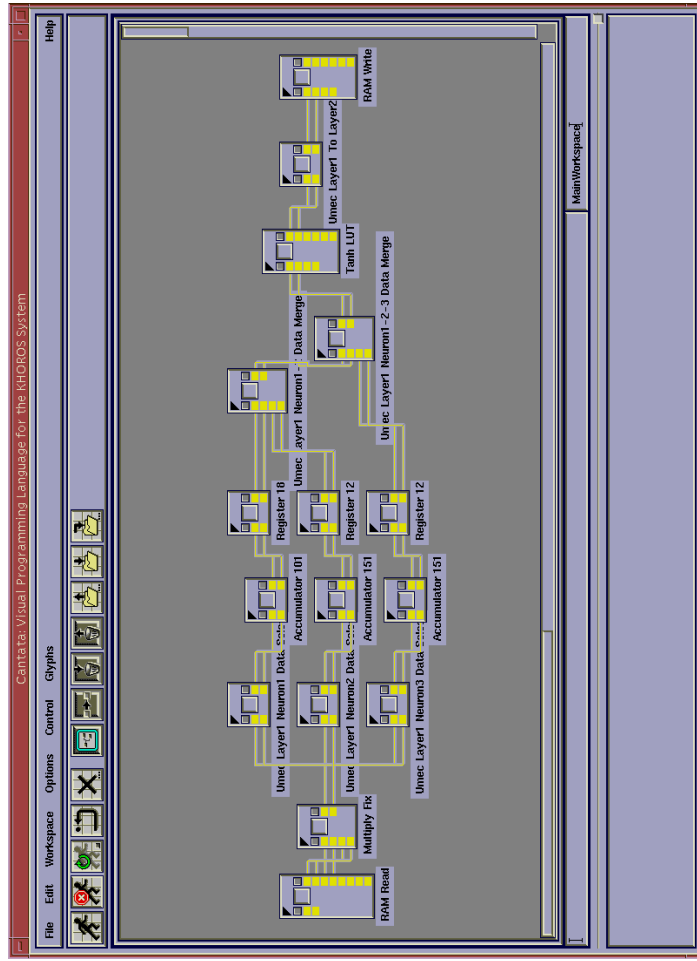


Figure 4.15: New modified first configuration of *umec* network fixed-point Cantata.

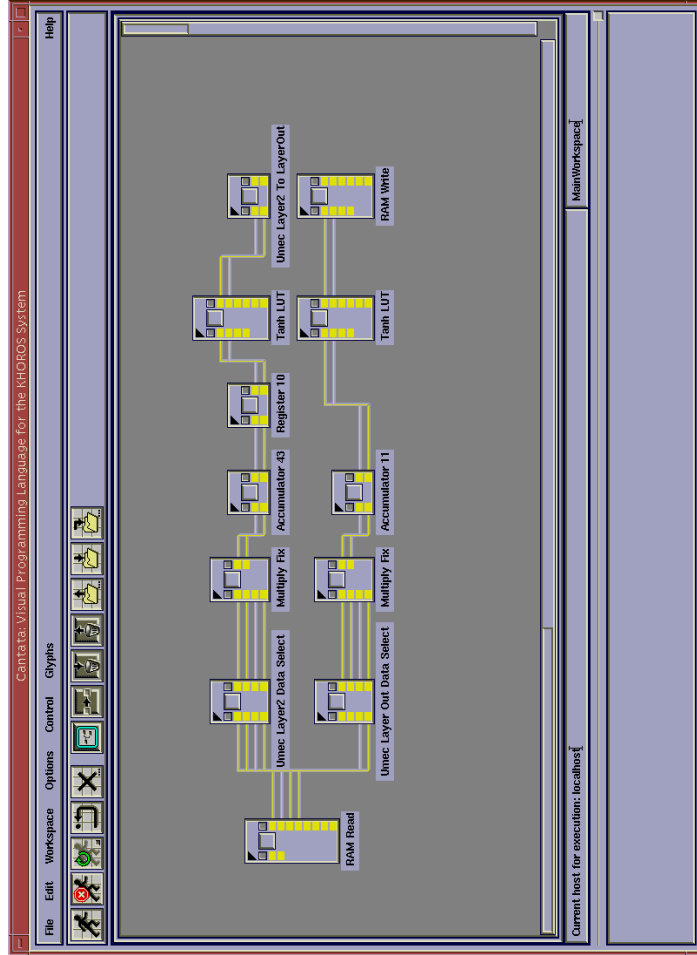


Figure 4.16: New modified second configuration of *umec* network fixed-point Cantata.

Because of the problems above, the hardware implementation of the networks were changed. Instead of mapping them onto the Wildforce, each was implemented onto a Virtex chip, which provided more resources. The CHAMPION software was still used for the front end, that is, to convert designs from the Cantata workspace to the CHAMPION netlist, to match data bit widths, and to synchronize data. The partitioning was removed from this flow. Thus, it was preferable to follow the CHAMPION ASIC flow until generating structural VHDL, instead of the CHAMPION ACS flow. Besides, the CHAMPION ASIC flow does not use COREGEN to generate a delay glyph. Once the structural VHDL file was created, it was then mapped utilizing commercial software. Mentor Graphic's Leonardo Spectrum was used for synthesis, Xilinx's ISE (Integrated Synthesis Environment) Series 4.1i for placement and route, and ModelSim from Model Technology for pre-layout and post-layout simulations. Figure 4.17 shows the new flow of the hardware implementation.

The designs were successfully mapped onto a Virtex-E FPGA XCV3200E device. It contained 104×156 arrays of CLBs, while the Wildforce board consists of one XC4036XL and four XC4013XL's forming a total of 3,600 CLBs. Table 4.1 shows this comparison. In addition, one CLB on Virtex-E contains 4.5 basic blocks that were called Logic Cells (LC). The four LCs were arranged in two similar slices, as seen in figure 4.18(a). The other half of the LC logic was used to combine the function generators from each LC to make up five or six input functions. One basic LC, displayed in figure 4.18(b), has one 4-input function generator, one carry logic, and one storage element. More details on the Virtex-E family can be obtained from [6].

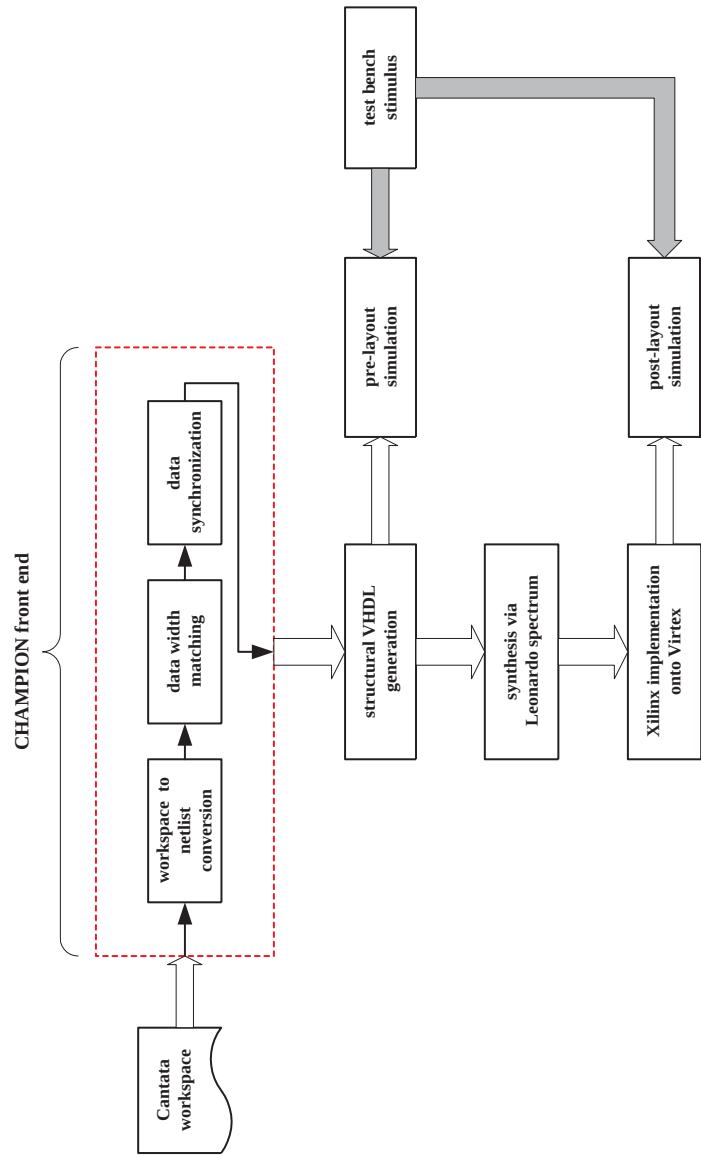
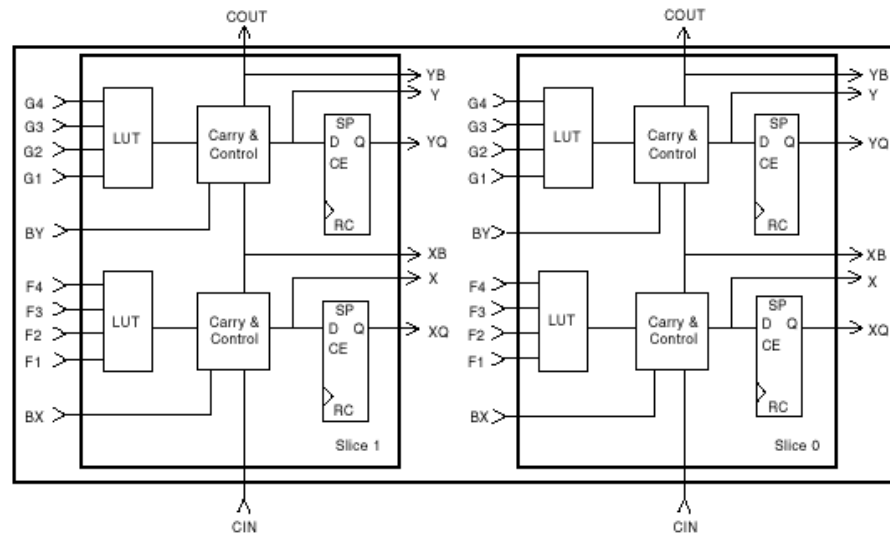
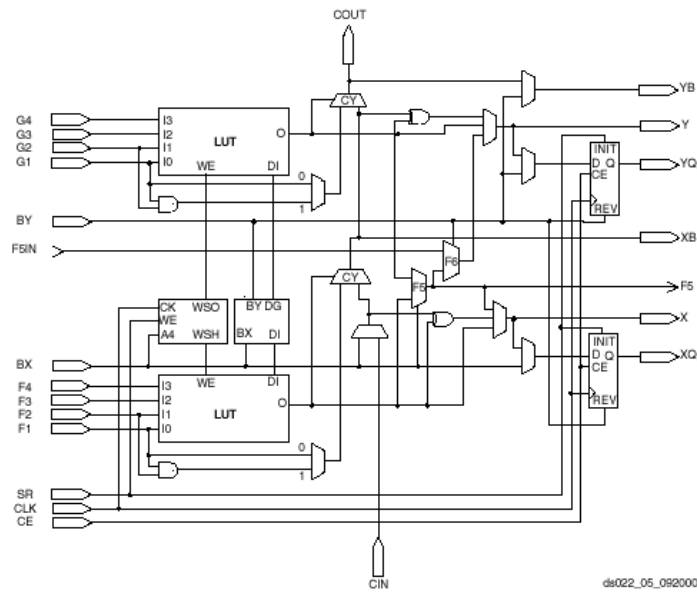


Figure 4.17: The new flow of hardware implementation.



(a)



(b)

Figure 4.18: Xilinx Virtex-E Architecture [6]. (a) One Virtex-E CLBs which contains four LCs arranged in two slices. (b) Detailed view of one slice.

Table 4.1: Resources on XC4013XL, XC4036XL and XCV3200E FPGAs.

	Logic Cells	CLB Matrix	Total CLBs	Equivalent Gate Count
XC4013XL	1368	24×24	576	10,000 - 30,000
XC4036XL	3078	36×36	1296	22,000 - 65,000
XCV3200E	73008	104×156	16224	4,074,387

Another change made was that the modified *umec* network, which contains two sub-designs, was converted back to its original design with a single configuration. The configuration should fit in one XCV3200E device which would make it simpler in the layout and simulation process. The results shown in later sections will display simulations of each network on the device, for the first few window detectors. These numbers were then compared to those resulting from the fixed-point Cantata implementation. The result section will also show the chip layout of each network.

4.4 Result

Each workspace has been successfully executed on floating point and fixed-point Cantata, and was simulated on the Virtex-E XCV3200E chip. This section is divided into three sub-sections. The first compares the outcomes between the original C code of the face detection developed by Henry Rowley, the floating point Cantata, and the

fixed-point Cantata methods. The second part displays comparisons between outputs of fixed-point Cantata execution, simulation results of pre-layout as well as post-layout design implementation on the Virtex chip. The last sub-section will show the layout capture of each neural network on the chip with additional information of resource usages.

4.4.1 Comparison between original C code, floating point Cantata, and fixed-point Cantata of the face detection system

Table 4.2, table 4.3, table 4.4 summarize the results of each test image, which includes each neural network rating, the face location if found, and the processing time. For the first and second test images, the original C code detected correct number and location of the faces found within the images. However, it missed one face on the last image (couple2.ppm), which could be attributed to the reduction from its original size, from 600×406 pixels to 217×147 pixels. Nevertheless, the goal here was to evaluate the results between different methodologies. As expected, the outcomes resulting from the floating point Cantata implementation were identical with those from the face detection C code including the number of faces detected, their locations and ratings. They should match because all the glyphs in Cantata were written in C and all variables declared within the functions used floating point data types that were similar to those in the original C code.

Meanwhile, data resulting from the fixed-point Cantata were different from the other two procedures, due to the use of fixed-point data type. The rounded numbers in

Table 4.2: Computation results on test image: albert.ppm

Test image 1 : <i>albert.ppm</i> ; size: 160×110 pixels				
Network		Original code from CMU	Floating Point Cantata	Fixed-Point Cantata
<i>Umec</i>	Rating	0.933243	0.933243	0.951172
	Location	(62,12)-(112,62)	(62,12)-(112,62)	(62,12)-(112,62)
	Execution time	4.20 seconds	11 minutes 28 seconds	59 minutes 63 seconds
<i>Face17c</i>	Rating	0.999749	0.999749	0.999023
	Location	(60,19)-(101,60)	(60,19)-(101,60)	(60,19)-(101,60)
	Execution time	22.70 seconds	12 minutes 15 seconds	1 hour 32 minutes
<i>Face18c</i>	Rating	0.999296	0.999296	0.999023
	Location	(60,19)-(101,60)	(60,19)-(101,60)	(60,19)-(101,60)
	Execution time	22.70 seconds	15 minutes 17 seconds	2 hours 20 minutes

Table 4.3: Computation results on test image: henry.ppm

Test image 2 : <i>henry.ppm</i> ; size: 209×200 pixels				
Network		Original code from CMU	Floating Point Cantata	Fixed-Point Cantata
<i>Umec</i>	Rating	1	1	0.999023
	Location	(75,75)-(135,135)	(75,75)-(135,135)	(75,75)-(135,135)
	Execution time	11.10 seconds	28 minutes	3 hours 50 minutes
<i>Face17c</i>	Rating	0.999922	0.999922	0.999023
	Location	(54,68)-(126,140)	(54,68)-(126,140)	(50,68)-(122,140)
	Execution time	141.0 seconds	1 hour 33 minutes	11 hour 12 minutes
<i>Face18c</i>	Rating	0.999922	0.999922	0.999023
	Location	(54,68)-(126,140)	(54,68)-(126,140)	(50,54)-(122,136)
	Execution time	141.0 seconds	2 hours 6 minutes	16 hours 4 minutes

Table 4.4: Computation results on test image: couple2.ppm

Test image 2 : couple2.ppm; size: 124×147 pixels				
Network		Original code from CMU	Floating Point Cantata	Fixed-Point Cantata
<i>Umec</i>	Rating	1	1	0.999023
	Location	(55,45)-(75,65)	(55,45)-(75,65)	(54,42)-(78,66)
	Execution time	8.60 seconds	51 minutes	3 hours 17 minutes
<i>Face17c</i>	Rating	0.999957	0.999957	0.999023
	Location	(55,43)-(75,63)	(55,43)-(75,63)	(55,43)-(75,63)
	Execution time	128.80 seconds	1 hour 23 minutes	16 hour 29 minutes
<i>Face18c</i>	Rating	0.999839	0.999839	0.999023
	Location	(56,44)-(76,64)	(56,44)-(76,64)	(55,43)-(75,63)
	Execution time	128.80 seconds	1 hour 50 minutes	16 hours 54 minutes

fixed-point data type contributed to less accurate results. However, the differences were not much. The network ratings, for example, differed within a range of 0.09% to 1.92%. This small amount of tolerance led to slight variance of the face location represented in the x-y coordinate. The results could be made more accurate by increasing the number of precision bits.

In terms of processing time, the Cantata in general took more time in executing the neural network stage. This fact could be attributed to the way Khoros Cantata handles its glyph. For example, each glyph saved its outputs to one or more intermediate files before passing them to the next glyph. Also the fixed-point Cantata implementation had the longest processing time, because of the input data was treated as characters. For every glyph, the functions **fscanf** and **fprintf** were used to read and write those numbers line per line. Unfortunately, the smallest test image, that is, the albert.ppm, contained 3,188,586 lines stored in the NetFixData.txt file.

4.4.2 Comparison between fixed-point Cantata implementation, hardware pre-layout and post-layout simulation

Due to the larger amount of time to execute the post-layout simulation, the author decided to process only the first ten of window detectors for each neural network included in this project. The umec network, for example, took about 12 hours to perform a post-layout simulation of these ten 30×30 window detectors, of which each window detector contained 5,883 pairs of pixel and weight data. The input data to be evaluated for each network was obtained from the pre-processing stage done in the

fixed-point Cantata implementation (saved in NetFixData.txt), using the albert.ppm as the test image. Results from each pre-layout and post-layout simulation were compared to the first ten numbers in SavedDetectionFix.txt of the fixed-point Cantata implementation. The comparisons are displayed in table 4.5, table 4.6 and table 4.7 for each network *umec*, *face17c* and *face18c*.

Unfortunately, the neural network output from the hardware implementation simulations disagrees with those from the fixed-point Cantata. The differences were mainly contributed by the truncation glyph. Table 4.8 is an example that shows the difference before and after the truncation process following a multiplier. The first two columns show the numbers to be multiplied. They are **Fix**< 14, 10 > data type. Results of the multiplication, which are 28 bits wide with 20 bits of precision, are in the fourth column. Validation of these values are supported by the numbers in the third column, which are the result of multiplication done in fixed-point C using the A|RT library. After passing through the *truncate_high_28_14* glyph, the truncated results are listed in the last column. The values have changed because the *truncate_high_28_14* process eliminates the 14 most significant digits. This truncation glyph will work correctly without altering the result if the numbers to be processed are positive integers.

The execution time for the hardware implementation cannot be successfully obtained in this realization, except to determine it from a timing diagram of the pre-layout and post-layout simulation. With a clock rate of 20Mhz or 50 nanoseconds, the timing diagrams are shown in figure 4.19, figure 4.20, figure 4.21 and were summarized in table 4.9. The numbers shown denote the time needed to process one window detector.

Table 4.5: Neural network computation results of first ten 30×30 window detectors *umec* network.

Window number	Fixed-point Cantata Implementation		Pre-layout simulation		Post-layout simulation	
	bin	dec	bin	dec	bin	dec
1	1111000000000000	-1	0001000000000000	1	0001000000000000	1
2	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
3	11110000010111	-0.9775390625	0001000000000000	1	0001000000000000	1
4	1111000000000000	-1	0001000000000000	1	0001000000000000	1
5	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
6	1111000000000000	-1	0000100100000000	0.5625	0000100100000000	0.5625
7	1111000000000000	-1	1111100100000000	-0.4375	1111100100000000	-0.4375
8	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
9	1111000000000000	-1	0000111000000000	0.875	0000111000000000	0.875
10	1111000000000000	-1	0001000000000000	1	0001000000000000	1

Table 4.6: Neural network computation results of first ten 20×20 window detectors *face17c* network.

Window number	Fixed-point Cantata Implementation		Pre-layout simulation		Post-layout simulation	
	bin	dec	bin	dec	bin	dec
1	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
2	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
3	1111000000000000	-1	0001000000000000	1	0001000000000000	1
4	1111000000000000	-1	0001000000000000	1	0001000000000000	1
5	1111000000000000	-1	0001000000000000	1	0001000000000000	1
6	1111000000000010	-0.998046875	0001000000000000	1	0001000000000000	1
7	1111000000000100	-0.99609375	1111000000000000	-1	1111000000000000	-1
8	1111000000000000	-1	0001000000000000	1	0001000000000000	1
9	1111000000000000	-1	0001000000000000	1	0001000000000000	1
10	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1

Table 4.7: Neural network computation results of first ten 20×20 window detectors *face18c* network.

Window number	Fixed-point Cantata Implementation		Pre-layout simulation		Post-layout simulation	
	bin	dec	bin	dec	bin	dec
1	1111000000000000	-1	1111010100000000	-0.6875	1111010100000000	-0.6875
2	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
3	1111000000000000	-1	0001000000000000	1	0001000000000000	1
4	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
5	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1
6	111100000000100	-0.99609375	0001000000000000	1	0001000000000000	1
7	11110000001011	-0.9892578125	0001000000000000	1	0001000000000000	1
8	11110000000011	-0.9970703125	1111111000000000	-0.125	1111111000000000	-0.125
9	1111000000000000	-1	0001000000000000	1	0001000000000000	1
10	1111000000000000	-1	1111000000000000	-1	1111000000000000	-1

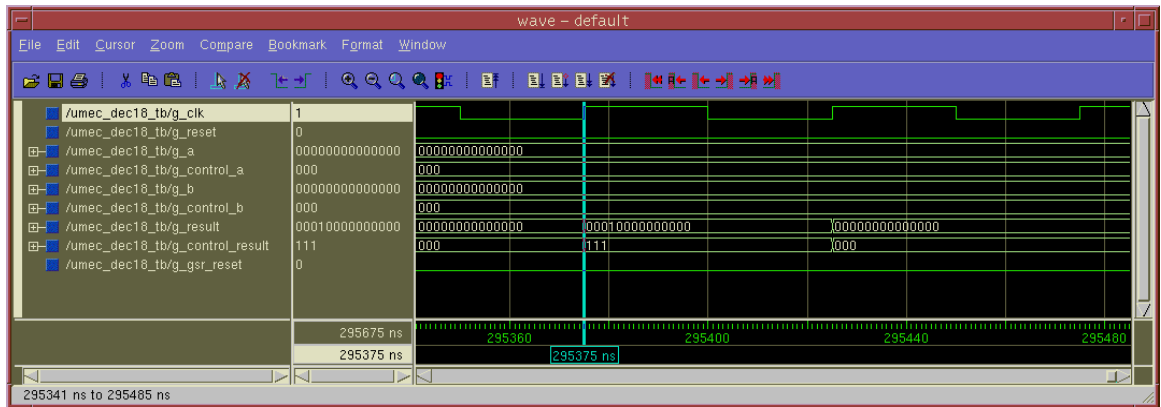
Table 4.8: Truncation glyph altered the original values.

Data1	Data2	Multiplication results		
		Fixed-point C with A RT library	Hardware glyph before truncation	Hardware glyph after truncation
0.099609375	-0.05859375	-0.00583648681640625	-0.00583648681640625	0
-0.1630859375	-0.017578125	0.0028667449951171875	0.0028667449951171875	-5.9765625
-0.837890625	-0.02734375	0.02291107177734375	0.02291107177734375	2.935546875
-0.84375	0.076171875	-0.06427001953125	-0.06427001953125	7.4609375
-0.3251953125	0.1533203125	-0.04985904693603515625	-0.04985904693603515625	-1.8125
-0.390625	0.2119140625	-0.0827789306640625	-0.0827789306640625	-3.0556640625
-0.59765625	0.263671875	-0.15758514404296875	-0.15758514404296875	-4.765625
-0.5537109375	0.0595703125	-0.03298473358154296875	-0.03298473358154296875	-1.3671875
-0.140625	-0.0556640625	0.0078277587890625	0.0078277587890625	-1.7763671875
-0.0537109375	-0.087890625	0.0047206878662109375	0.0047206878662109375	-7.984375

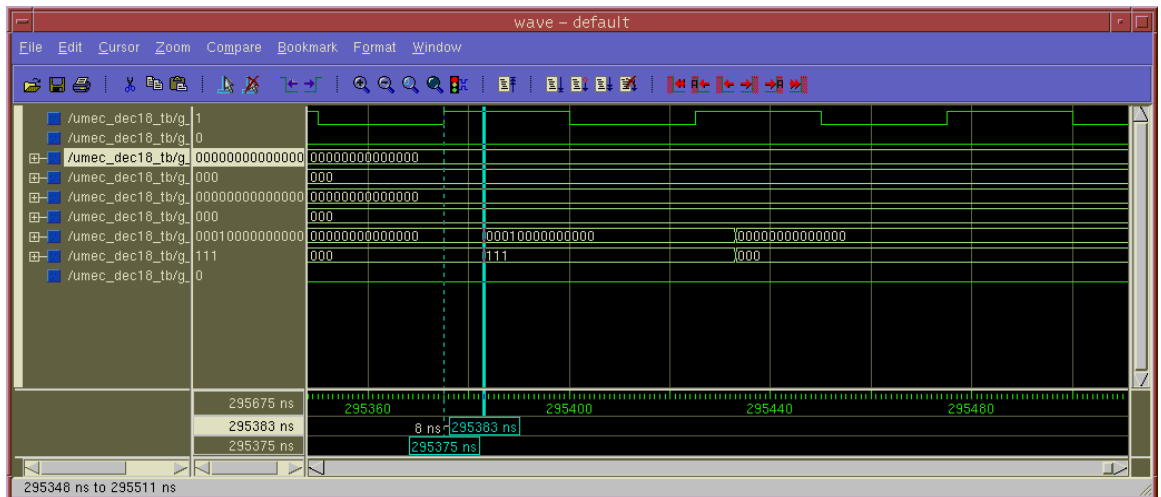
Therefore, to process 10 window detectors, for instance, it would take 10 times the amount of time to process one window. However, this timing cannot be used for comparison with the previous implementations (the floating point and fixed-point Cantata) because other timing information such as, device configuration time, data downloading time, and output reading time, should also be included.

4.4.3 Design layouts on a Virtex-E XCV3200E device

The layouts of each design are shown in figure 4.22, figure 4.23, and figure 4.24, with their resource utilizations summarized in table 4.10. As expected, the *umec* network occupied most of the available resources compared to the other two networks since it has two hidden layers and one output layer. The *face17c* network, meanwhile, used the least resources as it only has one hidden layer and contains the least number of connections between neurons.

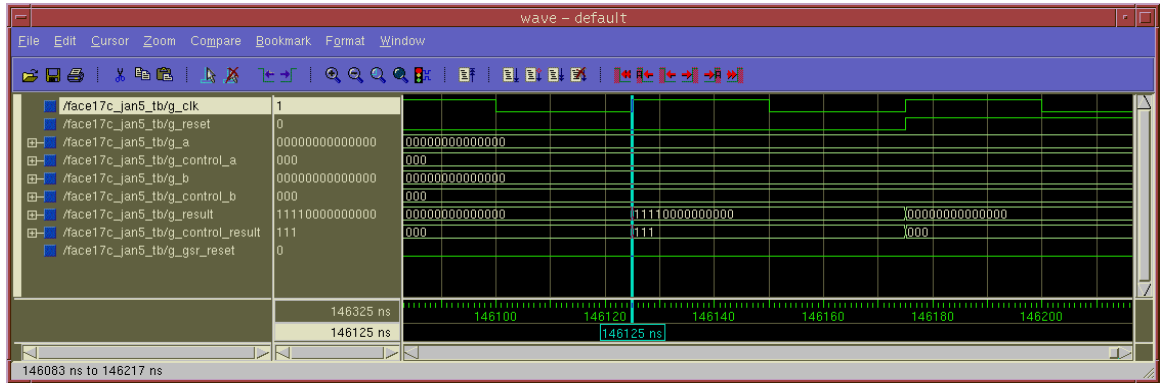


(a)

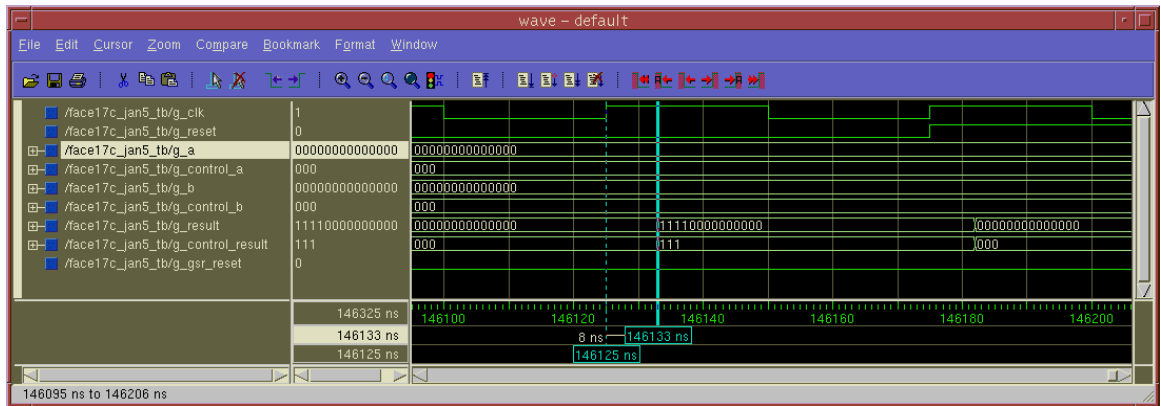


(b)

Figure 4.19: Umec simulation timing diagram. (a) Pre-layout simulation. (b) Post-layout simulation.

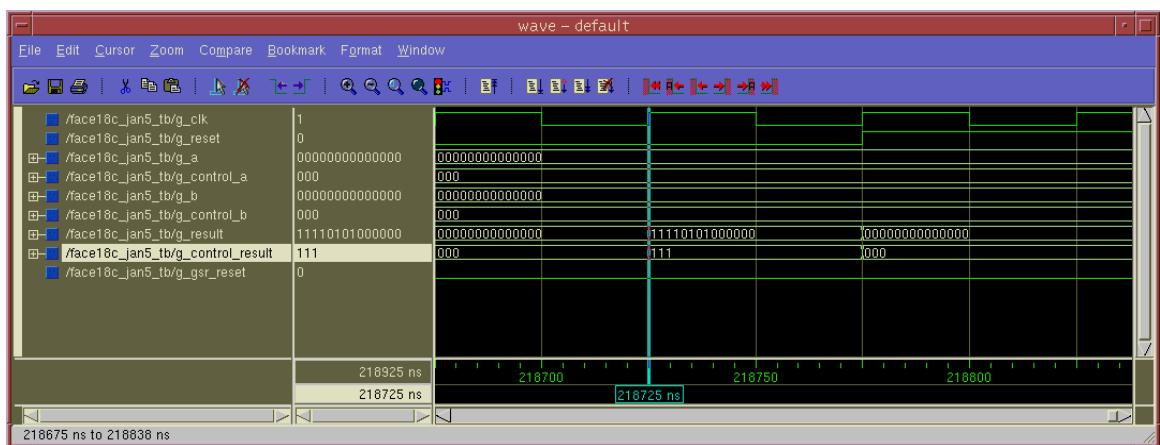


(a)

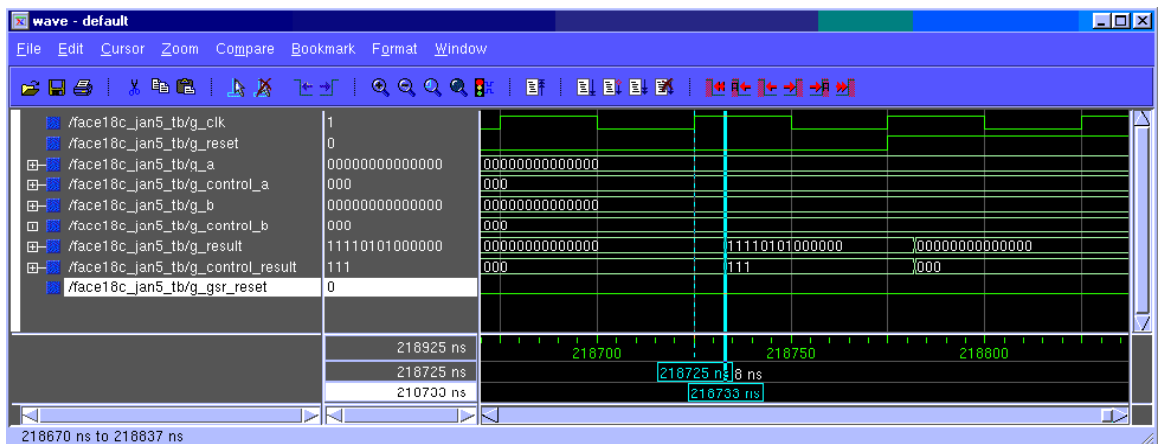


(b)

Figure 4.20: Face17c simulation timing diagram. (a) Pre-layout simulation. (b) Post-layout simulation.



(a)



(b)

Figure 4.21: Face18c simulation timing diagram. (a) Pre-layout simulation. (b) Post-layout simulation.

Table 4.9: Timing information for pre-layout and post-layout simulation on design implementation on XCV3200E device.

Network	Pre-layout timing	Post-layout timing
<i>umec network</i>	295,375 ns	295,383 ns
<i>face17c network</i>	146,125 ns	146,133 ns
<i>face18c network</i>	218,725 ns	218,733 ns

Table 4.10: Summary of resource utilization of each network on XCV3200E.

Neural Network	Slice Usage	Slice Flip Flop Usage	4-Input LUT	Equivalent Gate Count
<i>umec</i>	27,888/32,448	51,140/64,896	2,777/64,896	430,274
	85%	78%	4%	
<i>face17c</i>	15,914/32,448	23,287/64,896	2,316/64,896	243,496
	49%	43%	3%	
<i>face18c</i>	23,437/32,448	42,111/64,896	2,812/64,896	357,377
	72%	64%	4%	

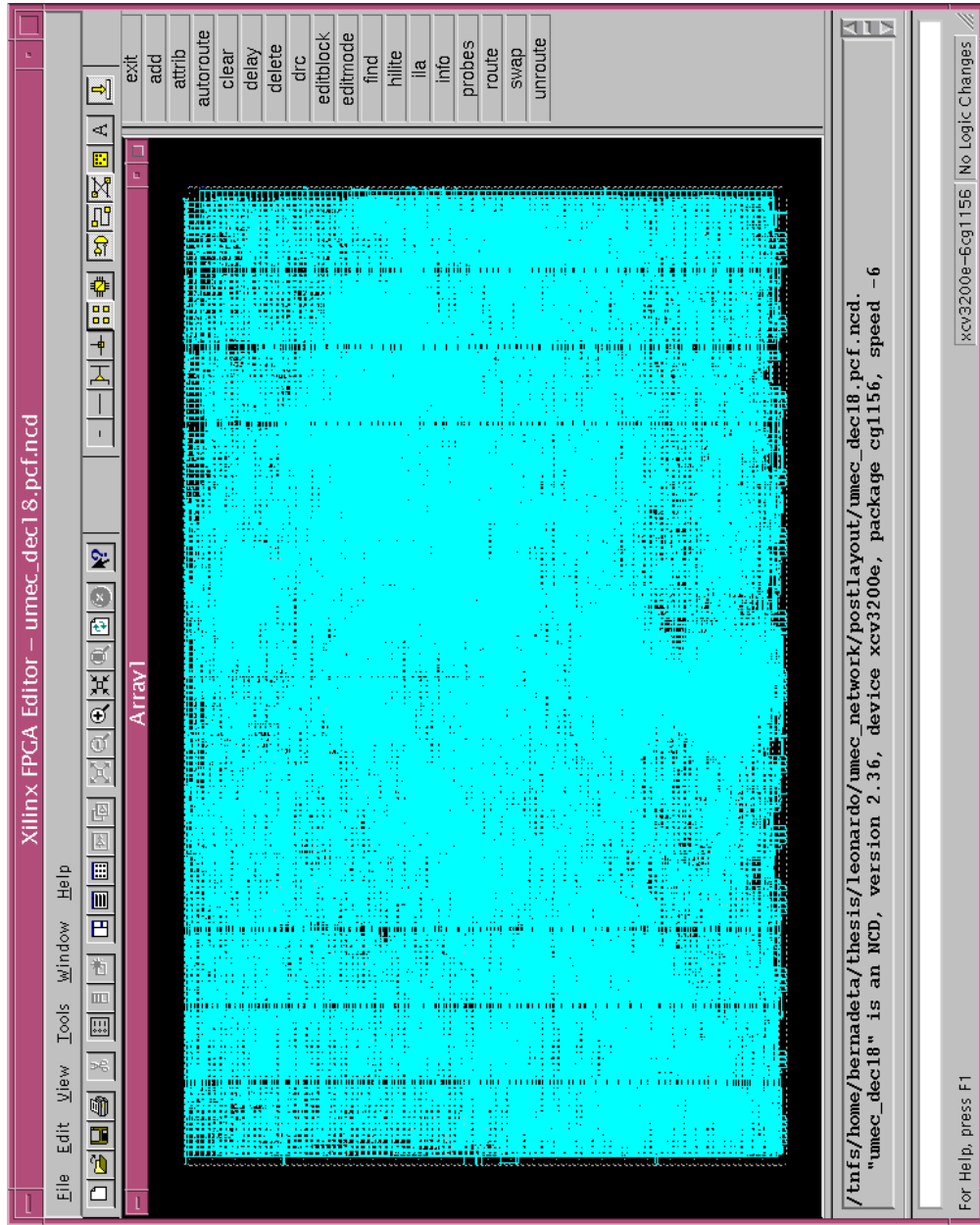


Figure 4.22: UmeC network design layout on Virtex3200E.

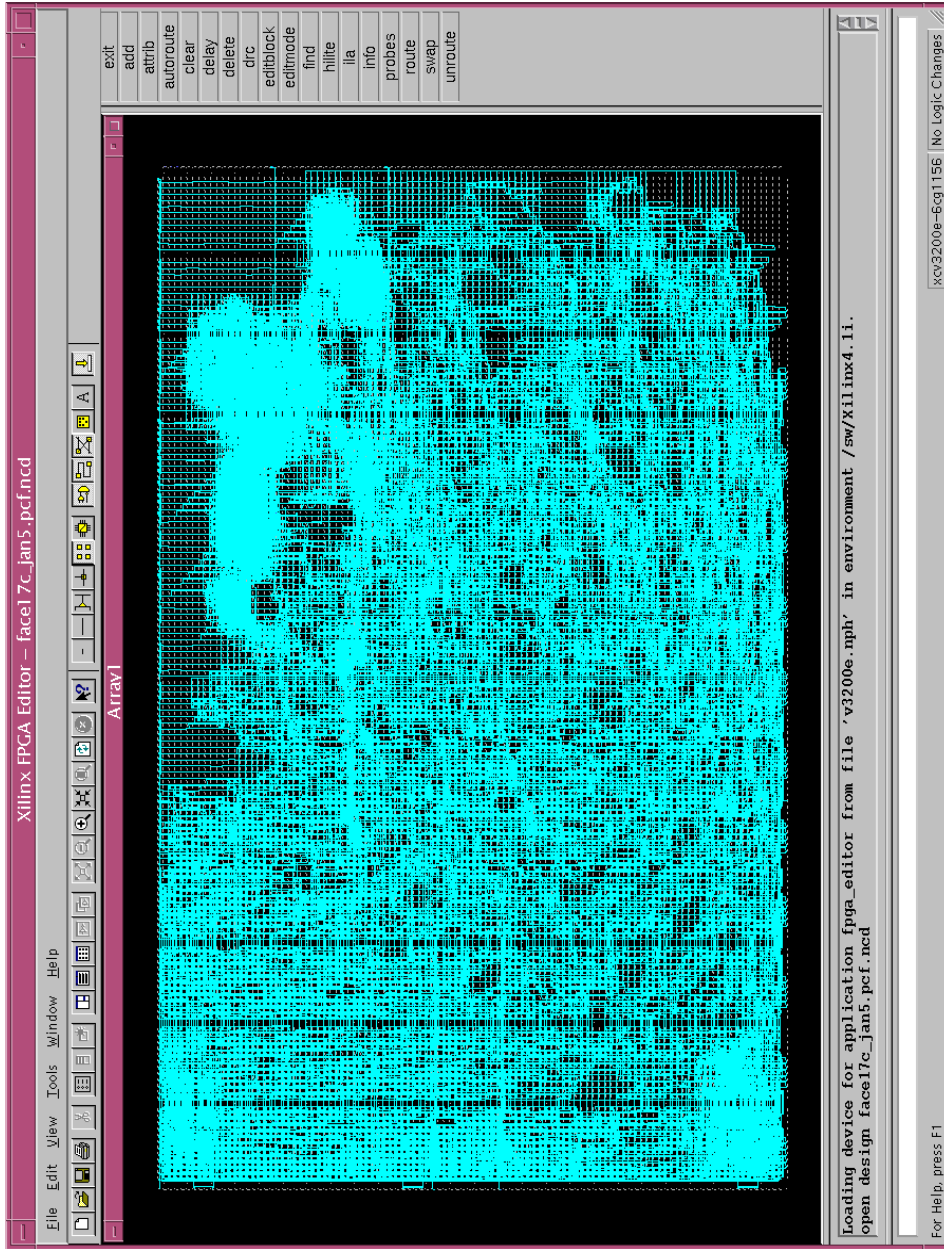


Figure 4.23: Face17c network design layout on Virtex3200E.

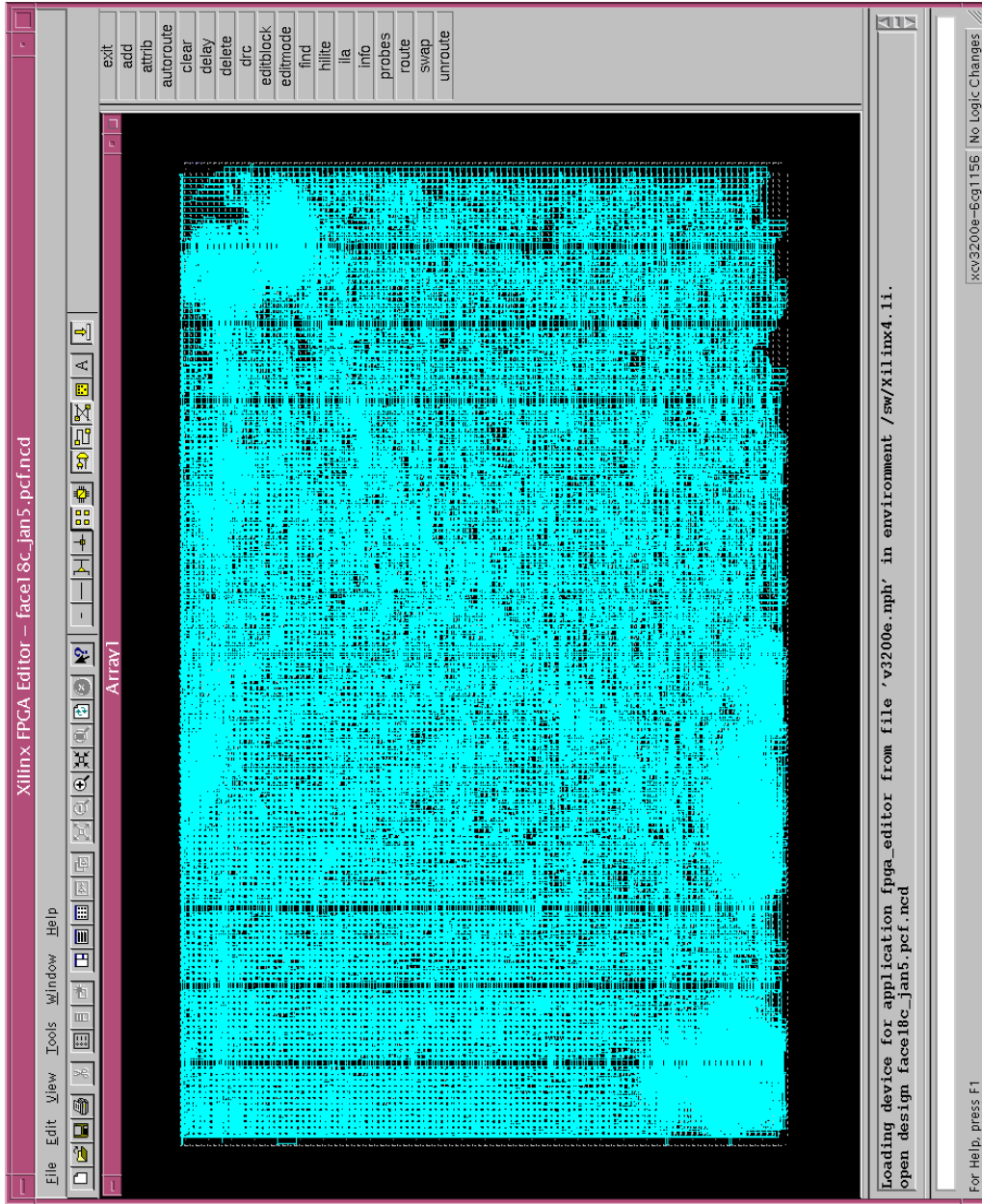


Figure 4.24: Face18c network design layout on Virtex3200E.

CHAPTER 5

Summary and Future Works

The flexibility offered by FPGAs helped in working on design changes many times. With the available an ACS platform, a design, which has been modified, can be tested within a few minutes or hours. The neural network designs, which were part of the face detection system, had gone through several changes because of unanticipated problems that occurred during the hardware implementation process. Two out of three proposed procedures were successfully finished. While the last methodology, which was the main goal in this project, was unable to be completed. Some commercial software environments had to be utilized to replace a few steps in CHAMPION that did not perform their tasks as expected. Upon finishing this project, a few suggestions were proposed to improve CHAMPION performance.

5.1 Performance

5.1.1 Design Performance

In the beginning of this project, the neural network stage was extracted from the face detection system. The system contained four main stages: localization and pose estimation, pre-processing including histogram equalization and lighting correction,

neural network, and post-processing. The system contained three neural networks whose structures were different in terms of the number of layers, number of neurons and their interconnections. The results displayed were the face candidates found within an image, their locations, and their ratings.

A workspace was designed for each network based on its neuron interconnection information and was successfully executed in the floating point and fixed-point Cantata environment. The floating point Cantata results and the original C code output were the same. The floating point Cantata glyphs were flattened in the fixed-point Cantata workspace so that each glyph would have a corresponding hardware module. The results almost equal those of the original C and floating point Cantata workspace. The difference in results was due to rounded numbers used in the fixed-point data type. The A|RT library was very useful in the fixed-point Cantata implementation process, in which the users were able to change the number of precision bits as needed.

Executing the applications in Cantata took longer because of the way Khoros handles processes within a glyph. A glyph had to save its results in one or more intermediate files before being sent to the next glyph. The fixed-point glyph, in particular, was the slowest. This drawback can be attributed to the way A|RT library reads/writes fixed-point data from/to a file line per line. This is inefficient when there is a large number of data to be processed.

Unfortunately, the neural networks' fixed-point workspace could not be mapped and executed on the Wildforce board as planned because of problems faced in CHAMPION and because of the large number of resources needed for each workspace. By using

available commercial synthesis, place and route tools such as Leonardo Spectrum and Xilinx ISE, this incomplete goal was replaced by mapping each workspace onto a Virtex-E XCV3200E device and running design simulations using the ModelSim software. Due to the considerable amount of time in simulating all input data, only the first 10 window detectors were processed. The results were not as expected since they were not equal to those executed in the fixed-point Cantata workspace. The problem was located in the truncation glyph. The glyph was originally designed for the truncation process of a positive integer data type only. Thus, it needed to be changed to accommodate signed numbers. Even though it can be determined from the simulation timing diagram how long it takes to produce one output of one window detector, depending on the clock rate, the total execution time of hardware implementation could not be obtained. Other timing information such as board configuration and data transfer time needed to be considered.

5.1.2 CHAMPION performance

CHAMPION performed the mapping of the workspace onto the users selected ACS platform. Upon learning on how to use CHAMPION, the author discovered the following shortcomings:

- CHAMPION failed to partition a large design that needed multiple configurations. This difficulty can affect the designers' efforts as they redesign their workspace, or to partition it into several configurations, which is very time consuming. From the experience encountered in this project, it took several months

to redesign the neural networks to fit CHAMPION tasks.

- CHAMPION delay generation could not create a delay glyph that exceeded maximum delay provided by COREGEN.
- CHAMPION truncation glyph changed the result values.

Despite the above problems, CHAMPION provided an alternative, that is, the CHAMPION ASIC flow. Although this flow was actually designed to implement an application using ASIC technology, part of this flow can be utilized up to the structural VHDL generation.

5.2 Future Work

Even though, in the end, the neural network designed in fixed-point Cantata workspace could not be mapped and executed in the Wildforce board, several possible projects can be identified to achieve this goal:

1. Improving CHAMPION

- Add automatic partitioning into multiple configurations. As it was mentioned in the shortcoming list, CHAMPION can be improved to partition more than one configuration when an application requires more resources than the board capacity.
- Improve CHAMPION's ability to generate longer delays. Since CHAMPION ACS' delay is generated by COREGEN, which allows a maximum

of 600 clock cycle delays, one alternative solution is for CHAMPION to be able to cascade several glyphs with smaller delays to form a delay of more than 600 clock cycles.

- Flexibility of truncation glyph. The CHAMPION truncation glyph can be modified to accommodate not only positive integer numbers, but also signed fraction numbers, by eliminating or truncating the correct bits.
- Provide larger ACS boards. A possibility to avoid partitioning problems and multiple configurations is to target a large application onto boards that offer more resources.

2. Improving designs.

While waiting for CHAMPION improvement, some changes can be made on the neural network designs. That is, manually divide the existing fixed-point Cantata workspace into several configurations, each of which fits into the Wildforce board. An example is to include only one neuron in one configuration for the *umec* network; hence, resulting in five sub-configurations.

After improving the network designs and resolving any CHAMPION problems, hopefully the entire face detection system, not just the neural network stage, can be implemented on a larger board.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] *Annapolis Micro Systems*. <http://www.annapmicro.com>.
- [2] *A|RT Library User's and Reference Documentation*. Frontier Design Inc., September 1999.
- [3] *Khoros Pro User's Guide*. Khoral Research Inc., Albuquerque, NM.
- [4] *Khoros Toolbox Programming*. Khoral Research Inc., Albuquerque, NM.
- [5] *Systems Level Applications of Adaptive Computing (SLAAC)*. <http://www.east.isi.edu/projects/SLAAC>.
- [6] *Xilinx Programmable Logic Data Book*. Xilinx, Inc., 1998.
- [7] G. J. Awcock and R. Thomas. *Applied Image Processing*. pages 106–108, McGraw-Hill, Inc., New York, NY, 1996.
- [8] T. Darell, G. Gordon, J. Woodfill, and M. Harville. A Virtual Mirror Interface using Real-time Robust Face Tracking. In *Third International Conference in Face and Gesture Recognition*, Nara, Japan, April 14-16, 1998.
- [9] D. Hammerstrom. Neural Network at Work. *IEEE Spectrum*, pages 26–32, June 1993.
- [10] N. Kerkiz. *Development and Experimental Evaluation of Partitioning Algorithms for Adaptive Computing Systems*. PhD dissertation, University of Tennessee, Knoxville, TN, December 2000.
- [11] M. Minsky and S. Papert. *Perceptron*. MIT Press, Cambridge, MA, 1969.
- [12] 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.
- [13] S. Natarajan. *Development and verification of library cells for reconfigurable logic*. Master's thesis, University of Tennessee, Knoxville, TN, August 1999.
- [14] J. V. Oldfield and R. C. Dorf. *Field Programmable Gate Array: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital System*. pages 1–3, 6–12, 53–69, John Wiley & Sons, Inc., New York, NY, 1995.
- [15] S. W. Ong. *Automatic Mapping of Graphical Programming Application to Micro-electronic Technologies*. PhD dissertation, University of Tennessee, Knoxville, TN, May 2001.

- [16] S. W. Ong, O. Kerkiz, C. Tan, M. Langston, D. Newport and D. Bouldin. Automatic Mapping of Multiple Applications to Multiple Adaptive Computing Systems. In *IEEE Symposium on Field-programmable Custom Computing Machines (FCCM)*, Rohnert, CA, April 30 2001.
- [17] H. Rowley. *Neural Network-Based Face Detection*. PhD dissertation, Carnegie Mellon University, Pittsburgh, PA, May 1999.
- [18] S. Satoh and T. Kanade. *Name-It: Association of Face and Name in Video*. CMU-CS-960205, Carnegie Mellon University, Pittsburgh, PA, December 1996.
- [19] R. Sukthankar and R. Stockton. Argus: An Automated Multiagent Visitor Identification System. In *Proceeding of the AAAI*, 1999.
- [20] A. S. Tanenbaum *Structured Computer Organization*. pages 13–27, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990
- [21] M. Young, D. Argiro, and S. Kubica. Cantata: Visual Programming Environment for the Khoros System. *Computer Graphics*, 29(2):22–24, May 1995.

VITA

Bernadeta Srijanto was born on December 27th, 1973 in Kuala Lumpur, Malaysia. In 1979, She and her family moved to Salatiga, Indonesia, where she finished her high school. In 1991 she continued her study at Gadjah Mada University in Yogyakarta. A year later she was awarded the Science and Technology Scholarship from the government of Indonesia to pursue her degree abroad. Arrived in Knoxville in August of 1993, she received her Bachelor of Science in electrical engineering from the University of Tennessee in May of 1998. She entered the graduate program in electrical engineering in August of 1998. While pursuing her master degree, she was a teaching assistant for the Department of Electrical and Computer Engineering for two years before beginning work as a research assistant for Microelectronics System Research Laboratory. She will receive her Master of Science degree in electrical engineering in August 2002.