To the Graduate Council:

I am submitting herewith a thesis written by Senthilvelu Natarajan entitled "Development and Verification of Library Cells for Reconfigurable Logic." I have examined the final 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.

Donald W. Bouldin

Dr. Donald W. Bouldin, Major Professor

We have read this thesis
and recommend its acceptance:

Chandra Tan

Dayton Neyt

Accepted for the Council:

Lewis Winkel

Associate Vice Chancellor
and Dean of The Graduate School

DEVELOPMENT AND VERIFICATION OF LIBRARY CELLS FOR RECONFIGURABLE LOGIC

A Thesis

Preseneted for the

Master of Science

Degree

The University of Tennessee, Knoxville

Senthilvelu Natarajan

August 1999

This thesis is dedicated to my parents, Dr. and Mrs. Sundaram Natarajan.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Prof. Donald W. Bouldin for not only giving guidance through the development of thesis, but for also introducing me to the world of VLSI through his teachings. It is through his inspiring courses that I begin my interest in the field of digital microelectronics, and for that I am in his debt. I wish to also like to thank Dr. Chandra Tan and Dr. Danny Newport for serving on my thesis committee. I would also like to thank these two gentlemen for providing constant guidance throughout the course of developing this thesis. The work presented in this thesis is a part of the Champion project developed for the DARPA grant F33615-97-C1124. Finally, I wish to express my thanks to my mother and father for their wonderful support through all my endeavors.

ABSTRACT

Champion is an Adaptive Computing System being developed as a means to translate software-based image-processing applications into equivalent hardware implementations targeted for multi-FPGA (Field Programmable Gate Array) platforms. Using Cantata, an image-processing designer can develop a Khoros workspace to implement an algorithm by connecting Khoros glyphs together. Therefore, the objectives of Champion are to parse a Khoros workspace and automatically translate the design captured by the workspace into a form that can be executed on a multi-FPGA platform. A library of hardware glyphs was developed for the Champion research project to use for its translations. It was therefore necessary to ensure the same functionality between the Khoros glyphs and the hardware glyphs. Design criteria were established so that all hardware glyphs work together in a pipelined fashion. The hardware glyphs were developed in a parameterized manner written in VHDL (Very High Speed Integrated Circuit Hardware Description Language). The VHDL code was then synthesized using the Synplify synthesis tool to generate XNF (Xilinx Netlist File) files. Each glyph was simulated using Altera Max Plus II to insure proper functionality. After successful simulation of a hardware glyph was achieved, the glyph was executed on a Xilinx FPGA to verify the glyph's design. Finally, the verified glyphs were collected together into a library which characterizes the glyphs by size, delay, and I/O count.

TABLE OF CONTENTS

CHAPTER PAGE

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

FIGURE PAGE

CHAPTER 1

Introduction

1.1 Motivation

Champion is an Adaptive Computing System being developed by the Microelectronics Research Group at the University of Tennessee as a means to translate software-based image-processing applications into equivalent hardware implementations targeted for multi-FPGA platforms. The software development platform is the Khoros Cantata graphical programming environment. Using Cantata, an image-processing designer can develop a Khoros workspace to implement an algorithm by connecting Khoros glyphs together. These glyphs can be thought of as C subroutines, while the Cantata workspace can be viewed as a main C program that invokes these subroutines and controls the data flow from one subroutine to another. Therefore, the objectives of Champion are to parse a Khoros workspace and automatically translate the design captured by the workspace into a form that can be executed on a multi-FPGA platform. The first multi-FPGA platform chosen for Champion to target is a Wildforce board which contains five Xilinx FPGAs.

The development of a library of Khoros glyphs is described in this thesis for Champion to use for its translations. This library contains glyphs that are

referred to as hardware equivalent glyphs. A Khoros user who wishes to use Champion must develop his workspace using these hardware equivalent glyphs. A similar hardware glyph library of pre-compiled XNF files was also developed for the targeted Xilinx FPGA architecture. Champion will partition, place, and route these pre-compiled hardware glyphs as described in the Khoros workspace. To aid in the mapping procedure, an attempt was made to design the hardware equivalent glyph library and the hardware glyph library to be as similar as possible. By doing so, the mapping procedure will yield an almost one to one correspondence between the Khoros workspace design and the mapped hardware implementation. Some discrepancies will arise between the two platforms due to the differences between software execution versus hardware execution of a design.

It was therefore necessary to ensure the same functionality between the hardware equivalent glyphs and the hardware glyphs (XNF files). The first step towards achieving this goal was defining precisely a glyph's parameters and its explicit functionality. A hardware equivalent glyph was developed using a high level programming language such as C++. This developed glyph was then used to generate output vectors for a set of applied input test vectors. These input and output vectors together constitute a test bench that can be used for computer simulations of the hardware glyph.

Once a hardware equivalent glyph was completed, the hardware description language VHDL was used to model a hardware version of the glyph. By developing the hardware using VHDL, the same source code can be used to develop

hardware glyphs for other target architectures. Using CAD tools, this code was synthesized to generate a digital circuit, which was then targeted for the Xilinx FPGAs using place and route CAD software. The synthesized circuit was simulated with the previously generated test bench to ensure the validity of the hardware glyph. At this stage, the hardware glyph was ready for execution on the FPGA platform. Verification of the hardware execution entailed applying the same test bench to the hardware and observing its outputs. The hardware glyph passed this final verification stage when its results concured with those of the computer-aided simulations. Otherwise, the VHDL code was re-evaluated and modified as necessary to correct the execution failure. Once verified, the hardware glyph was characterized in terms of size, delay, I/O count, and functionality.

1.2 Development Flow

Many of the hardware glyphs needed for Champion's mapping procedure differed slightly in their combinational logic operations. Therefore, before the hardware glyphs are developed, several hardware kernels were developed which perform these combinational operations. These kernels were instantiated inside the VHDL files for the hardware glyphs appropriately. The next development step for the hardware glyphs was to establish a set of criteria of how the glyphs operate with one another and establish a means of data flow from glyph to glyph. To accommodate this, a set of control lines were established for means of classifying each pixel in a set of data that are passed from one hardware glyph to

another. This was necessary for the hardware glyphs to mimic the behavior of Khoros glyphs as well as to operate within a hardware environment. Finally, a set of hardware glyphs were developed, simulated, and executed on a Xilinx FPGA platform when applicable. The results from the simulations and executions were compared with expected results from the hardware equivalent glyphs. All glyphs were developed using VHDL template files. These template files contained VHDL generic statements to accommodate parameterized bit widths for input and output ports. Finally, two test circuits were developed which invoked various hardware glyphs to insure that the hardware glyphs worked together properly as intended by design. Also, insights about Champion's mapping procedure dealing with data synchronization and data bit width alignments were gained from investigating these two test circuits. Finally results, conclusions, and suggested future works are presented.

1.3 Purpose and Expected Contributions

The purpose of this thesis is four-fold. First, it serves as a means of documenting the development of the first set of hardware glyphs for use by Champion. Second, it establishes a set of design criteria for developing a hardware glyph. Third, it documents the procedure used in validating, verifying, and characterizing a hardware glyph. Finally, it serves as an investigation for developing design aspects for the Champion project. Therefore, the author's contributions to the Champion project was also four-fold. The first contribution to the project was

the first set of developed library cells that Champion will use during its mapping procedure. Also, it set the initial design criteria and established the methodology for validating through simulation and verifying through hardware execution future library cells for the Champion project. Finally, lessons learned from developing these cells and using them to create operating circuits aided in the design of the Champion software.

CHAPTER 2

Background

2.1 Champion

One of the main obstacles to the use of FPGA-based reconfigurable computing systems has been the difficulty inherent in implementing applications on this hardware [1]. Current systems require the user to be familiar with digital hardware design, hardware description languages, and device-specific tools such as compilers and place and route software. The Microelectronic Systems Research Lab at the University of Tennessee is developing a software development system, called Champion, for the automated mapping of applications in the Khoros Cantata graphical programming environment to FPGA-based reconfigurable computing hardware. Khoros is already widely used for algorithm development and simulation. Champion, will be used to map Khoros applications to a reconfigurable co-processor, speeding execution of the application and greatly increasing designer productivity. Since multiple hardware architectures can be targeted, Champion should also be able to map a completed algorithm to reconfigurable hardware to be used independently of a host computer.

The goal of this project is to automate the mapping of Khoros-based applications onto adaptive computing systems to improve designer productivity by 100

times [2]. Thus, more application designers will be able to achieve higher quality implementations in less time and adaptive computing systems will be utilized more effectively and by a wider audience. The approach taken is to build a library of pre-compiled primitives that can be used as building blocks of image processing applications. The Khoros flow graph will be automatically mapped onto the adaptive computing hardware by partitioning and floor-planning software that will be developed as part of the project. The library primitives in Khoros will be pre-compiles into FPGA configuration files and then characterized for area, delay, etc. in advance. This will not only accelerate the mapping process but will assist the designer in exploring candidate solutions before final implementation.

2.2 Cameron

A similar project, called Cameron, is being developed at the Computer Science Department at Colorado State University. The goal of the Cameron Project is to shift the programming paradigm of Adaptive Computing Systems (ACS) from hardware-centered to application-oriented, making them accessible to software engineers and portable across platforms [3]. This is achieved by developing an automated programming tool for Image Processing (IP) applications on Adaptive Computing Systems. This tool is seamlessly integrated into Khoros, a visual program development environment for IP applications. The approach consists of developing a subset of the C language, SA-C (Single Assignment C) that has been extended to support both IP applications and the mapping to ACS hardware;

an optimizing compiler for SA-C; a set of hardware modules implemented in VHDL and mapped to a variety of FPGA-based ACS platforms; and an optimized implementation of the IP components of the VSIPL standard library.

2.3 Reconfigurable Logic

Both these two projects utilize reconfigurable logic. The advantage of the reconfigurable hardware approach for digital signal processing implementation is that application flexibility is provided by allowing the actual hardware design to change rather than by adding software programmable hardware to a design [4]. That is, to change the application implemented by a reconfigurable hardware system, the system need only be reconfigured through the reloading of SRAM bits of an FPGA. This allows a completely new hardware design to be implemented. No external hardware needs to be added to the initial design to provide this flexibility.

2.4 Xilinx FPGA

The reconfigurable logic devices used by the Champion project are Xilinx FPGAs. the XC4000 series provide the benefits of custom CMOS VLSI, while avoiding the initial cost, long development cycle, and inherent risk of a conventional masked gate array [5]. XC4000 Series devices are implemented with a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs),

interconnected by a hierarchy of versatile routing resources, and surrounded by a perimeter of programmable Input/Output Blocks (IOBs). Because Xilinx FPGAs can be reprogrammed an unlimited number of times, they can be used in innovative designs where hardware is changed dynamically, or where hardware must be adapted to different user applications. FPGAs are ideal for shortening design and development cycles, and also offer a cost-effective solution for production rates will beyond 5,000 systems per month.

2.5 Wildforce ACS

Finally, the Adaptive Computing System used by Champion is a Wildforce board, which is developed by Annapolis Micro Systems. The Wildforce board contains one Control Processing Element and four Array Processing Elements [6]. Each Processing element is a high speed XC4000XL Series Xilinx FPGA. The Control Processing Element is a XC4036XL FPGA, and each Array Processing Element is a XC4013XL FPGA. A memory daughter board is attached to each processing element, and the board uses a high speed PCI bus interface to communicate with a host computer and can run at clock speeds of up to 50 MHz.

CHAPTER 3

Hardware Kernel Development

3.1 Hardware Kernels

Many of the hardware glyphs that need to be developed for use with Champion will differ from one another based on their combinational functionality. For example, the hardware glyph for addition will be the same as the hardware glyph for subtraction except for the circuitry that performs the respective arithmetic operation. Therefore, it is logical to first develop hardware kernels to perform these basic combinational operations that many of the hardware glyphs will require. These kernels can than be instantiated inside the VHDL source code for the hardware glyphs. By using this approach, much of the VHDL source code for the hardware glyphs can be re-used often.

Therefore, in this chapter, several hardware kernels were developed which will be used by the hardware glyphs to perform basic arithmetic operations. These kernels were either simulated or executed on an FPGA platform to insure proper functionality. However, these kernels were not tested extensively since the hardware glyphs in which these kernels were instantiated will be tested extensively. Two of the kernels developed in this chapter were used in developing hardware glyphs in Chapter 5.

3.2 Kadd

The kernel **kadd** adds two numbers of bit width *num_bits* with a result of *num_bits +1* bits. The inputs and outputs for **kadd** are all unsigned integers. By making the output one bit larger than the inputs, overflow errors are avoided. Therefore, all possible combinations of inputs will yield a correct output value. If a hardware glyph is designed to use fewer than *num_bits +1* bits to represent its result, then the most significant bits (MSB) or least significant bits (LSB) of *result* should be discarded as required by the hardware glyph. If the hardware glyph uses more than *num_bits +1* bits, then *result* should be zero-padded with leading zeros.

The RTL layout and simulation for the **kadd** kernel are shown in Figures 3.1 and 3.2, respectively. The values of all ports in the simulation are listed in decimal notation. The VHDL files for all hardware kernels developed are listed in the Appendix.

3.3 Ksub

The kernel **ksub** takes two inputs (a and b) with bit widths of *num_bits*. However, unlike the **kadd** kernel, the two inputs are signed integers using a two's complement representation with the most significant bit of each input being its sign bit. As in the case with the **kadd** kernel, the output port is also 1 bit larger than *num_bits* in order to in this case avoid both overflows and underflows.

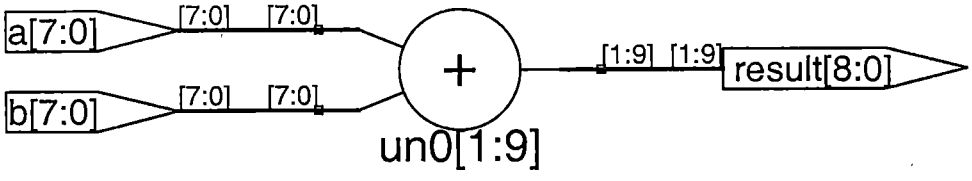


Figure 3.1: *RTL Layout for Eight Bit Kadd Kernel.*

Name:	500.0ns	1.0us	1.5us	2.0us	2.5us	3.0us	3.5us	4.0us	4.5us	5.0us	5.5
[I] a	162	3		52		197		0		255	
[I] b	179	241		33		7		0		255	
[O]result	341	244		85		204		0		510	

Figure 3.2: *Simulation Results of Eight Bit Kadd Kernel.*

Care must be taken to extend/discard the sign bit of the result when using a larger/smaller number bits to represent the result in a hardware glyph to avoid corrupting the computation. The RTL layout and a successful simulation for the **ksub** kernel are shown in Figures 3.3 and 3.4, respectively. The ports shown in Figure 3.4 are expressed in binary notation.

3.4 Kabs

The **kabs** kernel has only one input with *num_bits* bit width. This input is a signed integer. The MSB of this input is used as the sign bit with negative numbers expressed using the two's complement notation. The output has a bit width of *num_bits +1* to accommodate the most negative input value with *num_bits* bits. For example, if *num_bits* is eight, then the most negative number is -128. The absolute value of -128 is +128 which requires nine bits to be represented as a signed integer. The most significant bit of the output will always be zero since the absolute value of a number always yields a positive result. The extra bit is therefore mostly included in the kernel's design as a means to keep consistency when invoked inside a hardware glyph. For example, if the result was assumed to be the same bit width as the input and signed, then -128 would yield -128 for the eight bit case. However, with the extra MSB, there is no possibility for a sign error. The RTL layout and simulation for the **kabs** kernel are shown in Figures 3.5 and 3.6, respectively. The values listed for the ports *a* and *result* are both expressed as hexadecimal numbers.

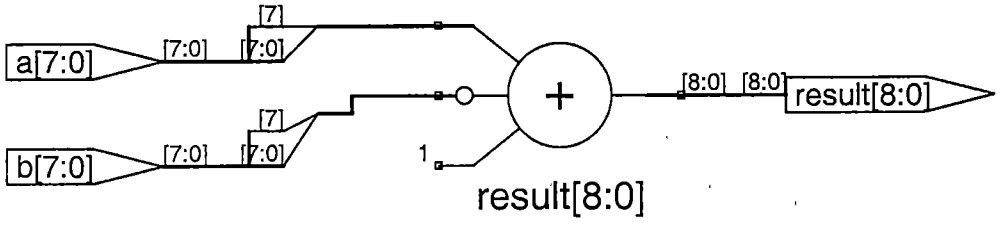


Figure 3.3: *RTL Layout for Eight Bit Ksub Kernel.*

Name.	500.0ns	1.0us	1.5us	2.0us	2.5us	3.0us	3.5us	4.0us	4.5us	5.0us	5.5us	6.0
[I] a	10000000		01111111		00000111		11000010		01010010		11000010	
[I] b	00000001		10000000		00110101		11000001		10000000		01111000	
[O]result	101111111		011111111		111010010		000000001		011010010		101001010	

Figure 3.4: *Simulation Results of Eight Bit Ksub Kernel.*

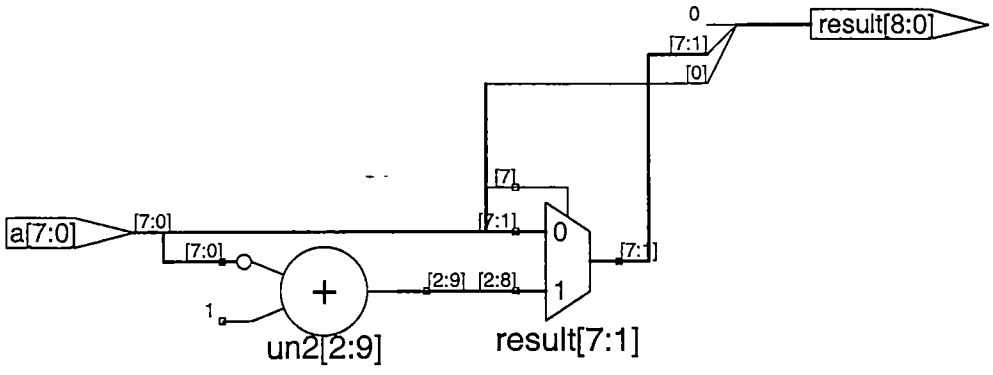


Figure 3.5: *RTL Layout for Eight Bit Kabs Kernel.*

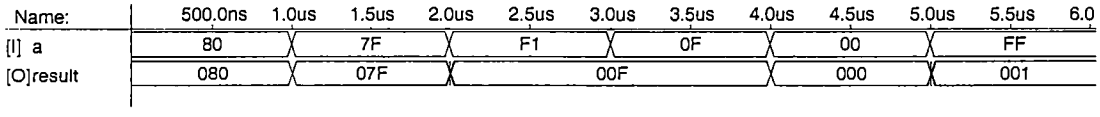


Figure 3.6: *Simulation Results of Eight Bit Kabs Kernel.*

3.5 Kabsdiff

The **kabsdiff** kernel has two signed input vectors of bit width *num_bits* with a result that is *num_bits +1* bits wide, which is also signed. The largest result for its eight bit case is | -128 -127 | = 255 The number 255 represented as a signed number needs nine bits. Again, all outputs will be positive values, yet the output is still represented using signed notation since the inputs to this kernel are signed. The RTL layout and simulation in hexadecimal notation for the **kabsdiff** kernel are shown in Figures 3.7 and 3.8, respectively.

3.6 Kdiv

The **kdiv** kernel returns the integer quotient of the division of an unsigned integer dividend by an unsigned integer divisor. The binary division process used for the **kdiv** kernel is similar to normal decimal division. Like decimal division, an iterative process is executed in which it is determined whether the divisor will "go into" manipulated values of the dividend [7].

Table 3.1 depicts the division process of two eight bit binary values. In this example 105 (0110 1001) is divided by 11 (0000 1011). After initialization, the process is basically an eight step loop since there are eight bits in the dividend. The resulting quotient is held in the final value of *a_var*. Appropriately, the final value for *a_var* is 9 (0000 1001). In the VHDL kernel source code, the values of *a*, *b*, and *result* are all *num_bits* wide. Figure 3.9 depicts the RTL layout for

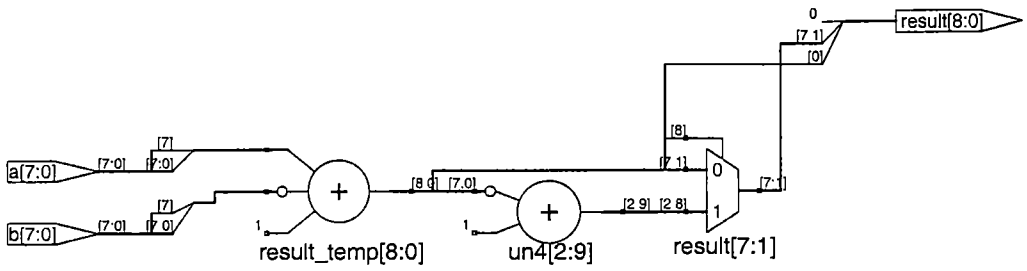


Figure 3.7: *RTL Layout for Eight Bit Kabsdiff Kernel.*

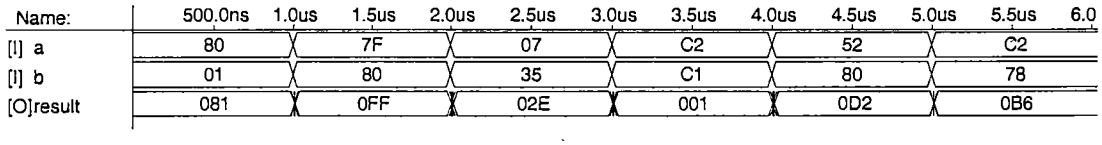


Figure 3.8: *Simulation Results of Eight Bit Kabsdiff Kernel.*

Table 3.1: *Illustration of the Binary Division Process.*

Step 1	Check to see if b = '0000 0000'. If so, then error = '1'. (b = '0000 1011', so division by zero does not occur in this example.)		
Step 2	Initialize variables	a_var = '0110 1001'	p = '0000 0000'
Step 3a	Begin 8 iteration loop: adjust p	a_var = '0110 1001'	p = '0000 0000'
Step 3b	p < b , ∴	a_var = '1101 0010'	p = '0000 0000'
Step 4a	Adjust p	a_var = '1101 0010'	p = '0000 0001'
Step 4b	p < b , ∴ adjust a_var	a_var = '1010 0100'	p = '0000 0001'
Step 5a	Adjust p	a_var = '1010 0100'	p = '0000 0011'
Step 5b	p < b , ∴ adjust a_var	a_var = '0100 1000'	p = '0000 0011'
Step 6a	Adjust p	a_var = '0100 1000'	p = '0000 0110'
Step 6b	p < b , ∴ adjust a_var	a_var = '1001 0000'	p = '0000 0110'
Step 7a	Adjust p	a_var = '1001 0000'	p = '0000 1101'
Step 7b	p ≥ b, ∴ adjust a_var & adjust p = p -b	a_var = '0010 0001'	p = '0000 0010'
Step 8a	Adjust p	a_var = '0010 0001'	p = '0000 0100'
Step 8b	p < b , ∴ adjust a_var	a_var = '0100 0010'	p = '0000 0100'
Step 9a	Adjust p	a_var = '0100 0010'	p = '0000 1000'
Step 9b	p < b , ∴ adjust a_var	a_var = '1000 0100'	p = '0000 1000'
Step 10a	Adjust p	a_var = '1000 0100'	p = '0001 0001'
Step 10b	p ≥ b, ∴ adjust a_var & adjust p = p -b	a_var = '0000 1001'	p = '0000 0110'

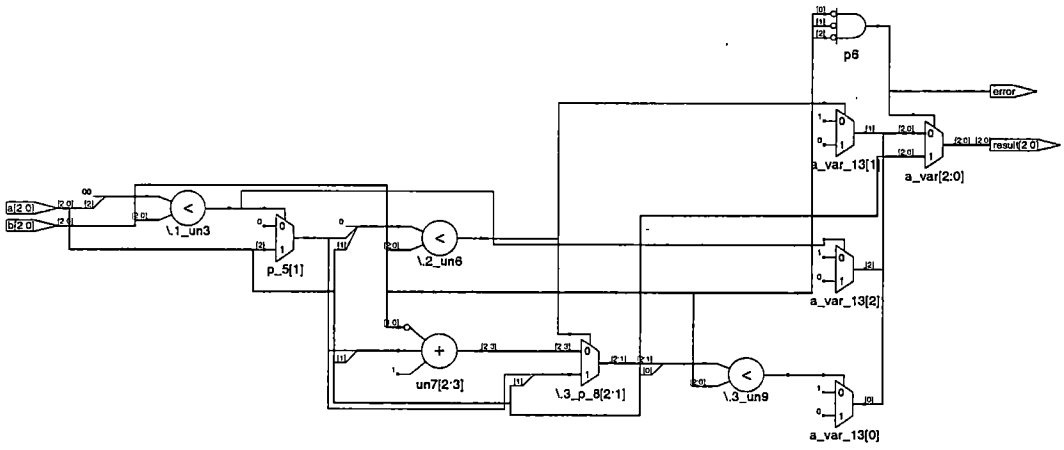


Figure 3.9: *RTL Layout for Three Bit Kdiv Kernel.*

Table 3.2: *Execution Results of Eight Bit Kdiv Kernel.*

	Data in value	Data out value
Execution 1	105 / 11 (01101001 / 00001011)	9 (00001001)
Execution 2	128 / 4 (10000000 / 00000100)	32 (00100000)
Execution 3	255/ 9 (11111111 / 00001001)	28 (00011100)
Execution 4	8 / 16 (00001000 / 00010000)	0 (00000000)
Execution 5	64/8 (01000000 / 00001000)	8 (00001000)

kdiv with *num_bits* set to three. Due to technical limitations which are beyond the scope of this thesis, **kdiv** is unable to be simulated using the Max Plus II software. Therefore, unlike previous kernels, the **kdiv** kernel with *num_bits* set to eight is executed on a Xilinx 4013XL FPGA to insure proper functionality. Five execution results are listed in Table 3.2 giving a modest level of assurance that the proper circuitry is synthesized. More adequate testing for functionality is done when **kdiv** is invoked inside a hardware glyph. This kernel has two outputs. The *error* output node is used to indicate that a division by zero error has occurred.

3.7 Kmod

The division process shown previously in Table 3.1 is also used to develop the **kmod** kernel. This kernel outputs the integer remainder in the division of two unsigned integer values. The final value of $p = 6$ (0110) in Table 3.1 is actually the remainder of 105/11, Figure 3.10 shows the RTL layout for the **kmod** kernel when *num_bits* is two.

Similar to the **kdiv** kernel, the **kmod** kernel is also unable to be simulated

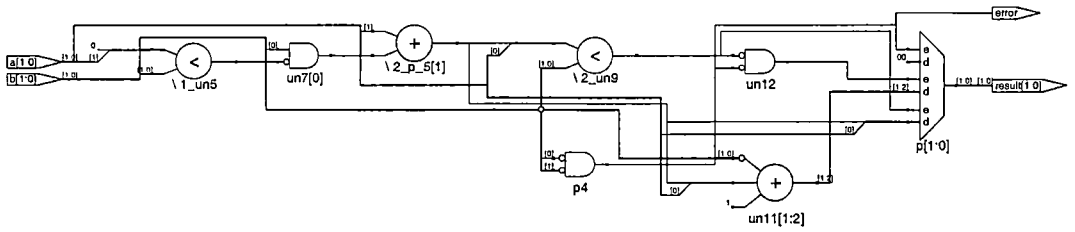


Figure 3.10: *RTL Layout for Two Bit Kmod Kernel.*

Table 3.3: *Execution Results of Eight Bit Kmod Kernel.*

	Data in value	**Data out value**
Execution 1	105 / 11 (01101001 / 00001011)	6 (00000110)
Execution 2	128 / 4 (10000000 / 00000100)	0 (00000000)
Execution 3	255/ 9 (11111111 / 00001001)	3 (00000011)
Execution 4	8 / 16 (00001000 / 00010000)	8 (00001000)
Execution 5	64/8 (01000000 / 00001000)	0 (00000000)

using the Max Plus II software. Therefore, an eight bit **kmod** kernel is also executed on the Xilinx FPGA to insure proper functionality. Five execution results for this kernel are shown in Table 3.3. As is the case with the **kdiv** kernel, the **kmod** kernel also has an *error* output node to indicate a division by zero.

3.8 Kpow

The final hardware kernel discussed is the kernel **kpow**, which is used for the exponential operation. The **kpow** kernel VHDL source code has one

std_logic_vector input *a* along with two VHDL generic assignments. The first generic assignment is *num_bits_a*, and the second is *pow* which determines what power input *a* will be taken to yield an output vector *result*. Figure 3.11 shows the **kpow** kernel with *num_bits_a* = 8 and *pow* = 4. The resulting vector will have a bit width of *pow* * *num_bits_a*. Similar to **kdiv** and **kmod**, the **kpow** kernel is also not able to be simulated. Therefore, Table 3.4 depicts the execution results of the **kpow** kernel with *num_bits_a* = 5 and *pow* = 6. Therefore, the output values are 30 bits wide (5*6).

3.9 Kernel Development

Seven hardware kernels were developed and either simulated with CAD tools or executed in a Xilinx 4013XL FPGA. In addition to these seven kernels, over twenty other combinational kernels were developed for use in designing hardware glyphs. When designing hardware kernels, proper bit sizes are always allocated to avoid errors such as overflows and underflows. The use of VHDL generics in the kernels were employed to ease the development of hardware glyphs with parameterized bit widths. All kernels were developed using unsigned integer values unless the combinational operation requires signed values such as the subtraction or negation operations.

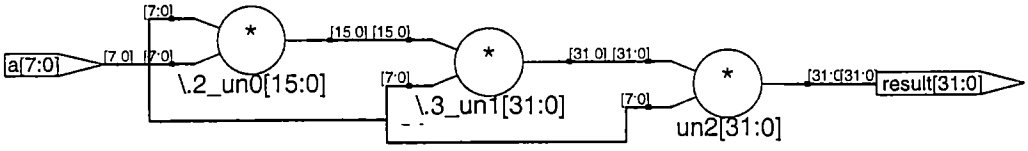


Figure 3.11: *RTL Layout for Eight Bit Kpow Kernel.*

Table 3.4: *Execution Results of Eight Bit Kpow Kernel.*

	Data in value	**Data out value**
Execution 1	7 (00111)	117,649 (000000000000011 100101110010001)
Execution 2	3 (00011)	729 (000000000000000 000001011011001)
Execution 3	18 (10010)	34,012,224 (000010000001101 111110001000000)
Execution 4	23 (10111)	148,035,889 (001000110100101 101100100110001)
Execution 5	31 (11111)	887,503,681 (11010011100100 11101101000001)

CHAPTER 4

Stream Representation of Images

4.1 Data Synchronization

Several basic combinational kernels were developed in the previous chapter. However, these hardware blocks cannot be used by themselves to replicate the behavior of a Khoros workspace. These kernels must be instantiated in higher VHDL components that operate in a sequential manner that is clock driven. It is important to keep in mind that the input data from the Khoros workspace represent two-dimensional images, and that the input data values will correspond to actual pixel values from these input images. The kernels were designed to operate on only a fixed number of pixels from their respective inputs. Therefore, it is necessary to translate the input images into a format that can be used as inputs to the developed kernels. The format used to accomplish this is to translate two-dimensional input images into one-dimensional data streams.

Figure 4.1 illustrates how two streams serve as inputs to Filter 1 and Filter2. The outputs from these two filters are then fed into Filter3. This process is clearly sequential in nature and represents what most Khoros workspaces will entail. In this fictitious example, Filter 1 finishes processing its input data long before Filter 2. Therefore, the two output streams must be synchronized together to feed as

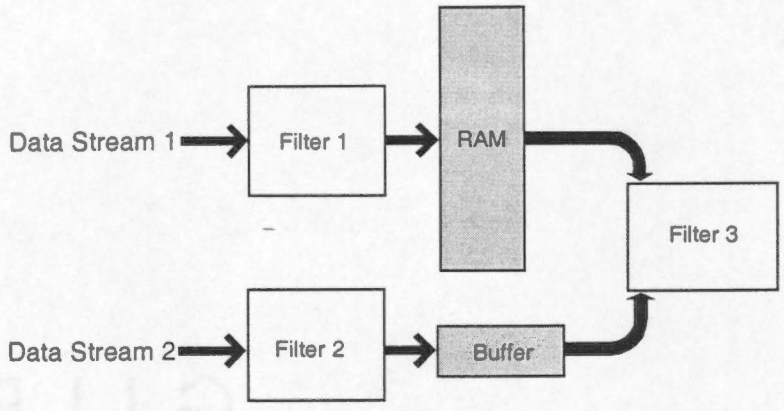


Figure 4.1: *Synchronization of Modules.*

inputs to Filter 3. One way to accomplish this is to store the output of Filter 1 in RAM. When Filter 2 is finished, its data may need to traverse through a small buffer to synchronize itself with the other stream due to the delay caused by accessing the RAM for the first stream. In this chapter, methods were developed to help aid with such data synchronization issues as well as to instantiate the previously developed purely combinational kernels inside modules that operate in a sequential nature.

4.2 Control Lines

Two-dimensional images are translated into a one-dimensional stream as shown in Figure 4.2. The stream format is generated by taking each pixel from the original image one at a time, starting with the upper-left most pixel and

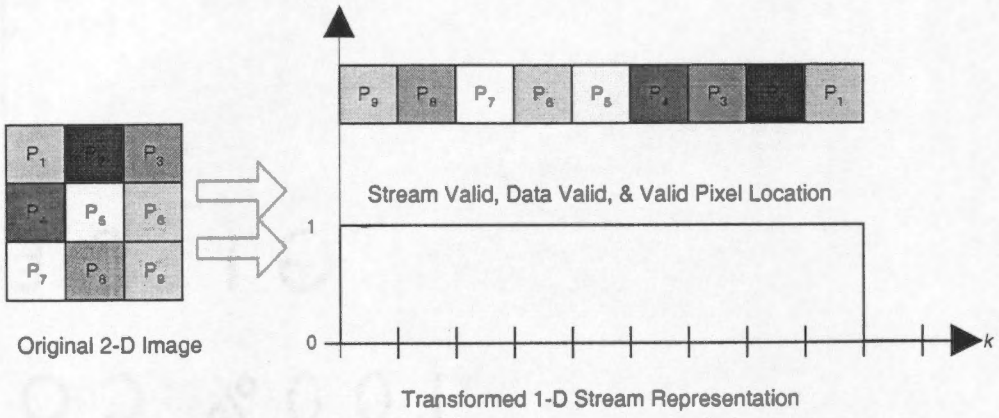


Figure 4.2: *Stream Representation of Two Dimensional Image.*

continuing down the first row of pixels. Once all the pixels from the first row are exhausted, the pixels from the second row are used, again starting with the left most pixel from this row. The procedure is repeated until all rows from the original image are exhausted. The image transfer procedure to the Wildforce board will be accomplished by the host code software for the board, which is beyond the scope of this thesis. What will actually take place in hardware, is that the input pixel will be stored in RAM, in precisely the same order as in the input image. A hardware RAM read module will be used to access these pixels and generate this stream of data.

Along with generating the stream of data from the pixels stored in RAM, the RAM read module must also generate the three control signals shown in Figure 4.2. Various operations, such as convolution, may disrupt a stream's continuity or corrupt its pixel values. These three control signals are designed

to help guide the hardware glyphs synchronize and control operations on data streams.

4.2.1 Stream Valid Control Line

The *Stream Valid* control line will be used by Champion to indicate when a data stream is present at the input to a hardware glyph. Figure 4.3 illustrates two data streams traversing down a path. The first image stream has been broken into two separate pieces while the second stream remains in tact. The cause for the first stream being broken is due to a hardware glyph that produces output values in a non-continuous fashion. For example, a hardware glyph may generate a cyclic pattern of four output values in a row followed by a period with no output value. Between the blocks of values the glyph's output may either be null or simply corrupt data depending on the hardware glyph. However, the *Stream Valid* control line must remain high indicating that the data stream is not yet complete. Only after all output pixel values for a given stream have been generated by the hardware glyph will the *Stream Valid* control line be cleared.

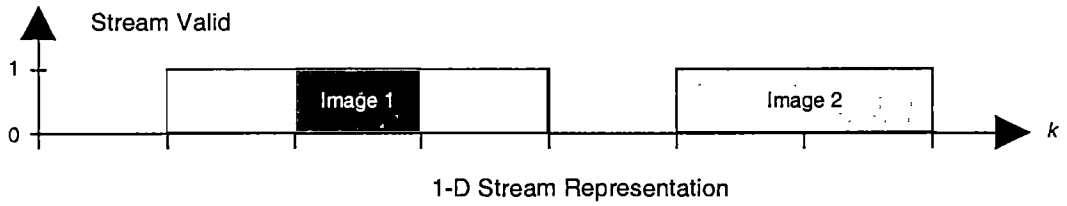


Figure 4.3: *Stream Valid Control Line.*

4.2.2 Data Valid Control Line

The second control line associated with a data stream is the *Data Valid* control line. The *Data Valid* control line will be used to determine whether or not data has become corrupted due to an improper calculation. Figure 4.4 shows a data stream that has nine pixel values. However, pixels 1, 2, 8, and 9 have become corrupted due to an improper calculation. Therefore, the *Data Valid* control line is set low during the periods of these pixels. The line remains set only during periods which contain pixel values that have not been corrupted.

Therefore, this line will be used to determine whether a pixel value should be used or avoided in future calculations. Corrupted pixel values are not simply discarded away since they still need to be used when reconstructing an image from the data stream or other geometry related processes. Border pixels generated from a convolution process or division by zero are two examples of when data pixels can become corrupt. Such pixels do not need to be used further along the pipeline of glyphs when doing statistical calculations on the pixel values of the

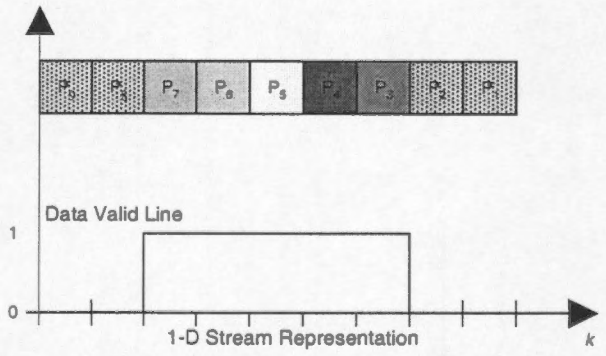


Figure 4.4: *Data Valid Control Line.*

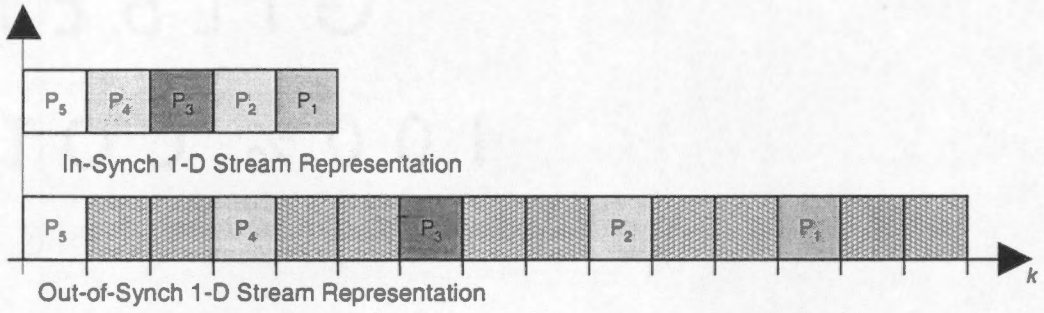


Figure 4.5: *Valid Pixel Control Line Example.*

stream. However, they are not simply thrown out since they do represent the output pixels of the convolution or the division, and need to be present when converting the stream back to an image.

4.2.3 Valid Pixel Location Control Line

The final control line of the hardware glyphs to be used by Champion is the *Valid Pixel Location* control line. As mention previously, individual pixels within a given data stream may become separated from one another when passed through certain hardware elements. The *Valid Pixel Location* control line will be used to indicate which periods of a stream represent actual pixel values. Figure 4.5 shows five pixels which form one continuous or in-synch data stream and the same five pixels after they have become separated or out-of-synch with one another. The *Valid Pixel Location* control line will stay high only during the periods which contain the five pixels in the out-of-synch data stream. Since all periods having

a clear *Valid Pixel* Location bit have no physical meaning to an image, the pixel values during these periods may be discarded when a stream is written to RAM or converted back to an image. Although a data stream is out-of-synch, it may still be able to be used in this format due do the symmetry of many image processing algorithms.

4.3 Simple Example of Control Line Use

Figure 4.6 shows how a data stream is used in conjunction with its control lines in a simple mask convolution. For simplicity, the mask is set to a size of three by three with all values being zero except the center pixel which has a value of one. The original two-dimensional image is first converted into its one-dimensional data stream representation along with its associated three control lines. This data stream is first passed through a hardware glyph called "Stream to Pixel Neighbor Streams" to generate the proper input streams for the convolution hardware glyph. The resulting output is shown as a two-dimensional image. Notice that the border pixels from the convolution no longer represent actual pixel values. The one-dimensional output data stream shown in Figure 4.6 illustrates the second row of the image. In this stream, the *Stream Valid* line stays high during the whole time. The *Data Valid* line only goes high during the two center pixels which have remained uncorrupted after the convolution. Finally, The *Valid Pixel Location* goes high whenever a period represents an actual geometric pixel regardless of whether that pixel's data is corrupt or not.

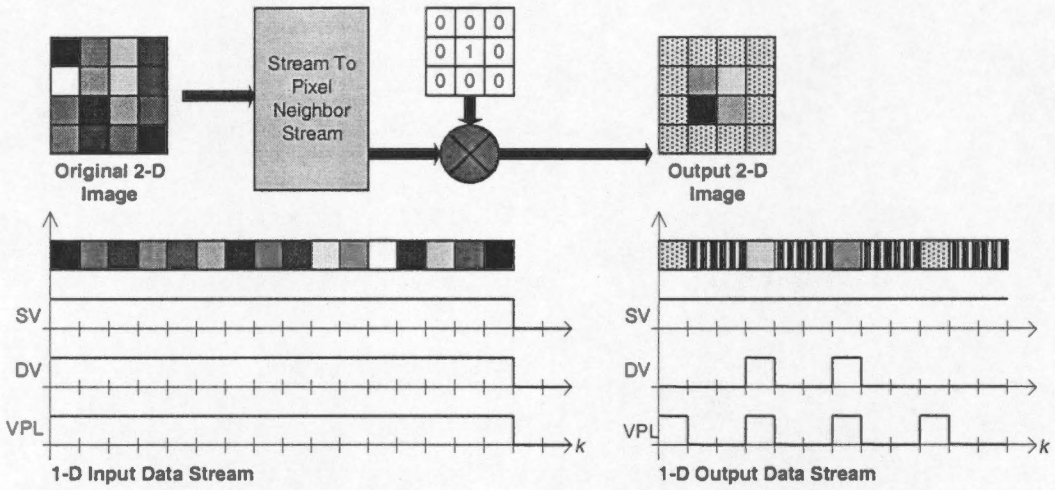
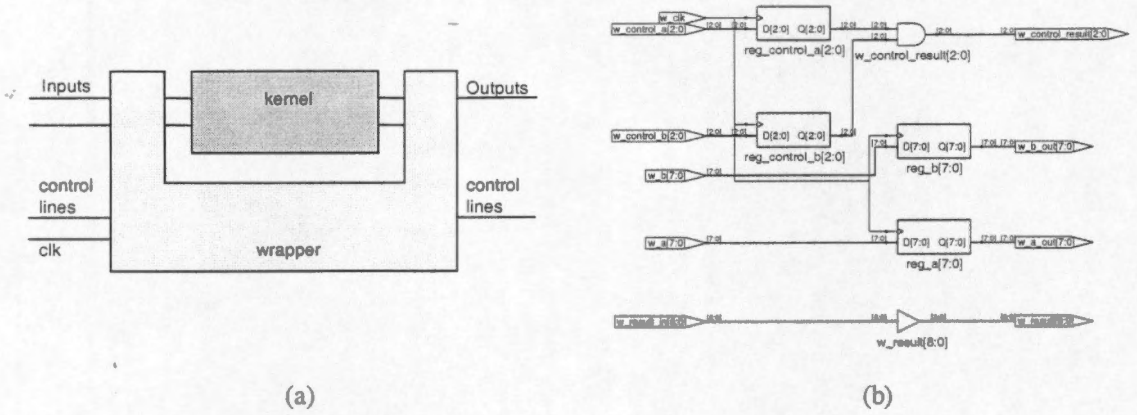


Figure 4.6: *Simple Convolution Example.*



(a) (b)

Figure 4.7: *Hardware Glyph (a) Wrapper and Kernel. (b) Internal Wrapper Logic.*

4.4 Sequential Wrapper Logic

The basic idea in generating hardware glyphs for Champion to use is to instantiate the developed kernels within a defined sequential mode of operation so that they mimic the behavior of native Khoros glyphs. Figure 4.7(a) shows how a hardware kernel is surrounded by a wrapper hardware block to provide the data handling needed by each hardware glyph. For most simple kernel operations, the control logic lines need to simply be logically and-ed together. However, for more complex kernels, the control lines outputs will be governed by the design requirements of that particular hardware glyph.

Figure 4.7(b) shows the logic needed inside the most basic wrapper block. In this example, all inputs are first latched for each clock cycle. The latched data values are then transferred to the kernel to perform the glyph's calculation. The result from the purely combinational logic housed inside the kernel will then be transferred back to the wrapper to be placed on the output lines of the hardware glyph.

The hardware glyphs developed in the next chapter are developed by joining together a hardware kernel with an appropriate sequential wrapper to form a hardware glyph. The approach taken is to simply instantiate the hardware kernel and define the wrapper logic within the hardware glyph's VHDL source code. Since the hardware kernels are often similar in their port declarations, this approach promotes the re-use of VHDL source code with only slight modification for many of the hardware glyphs.

4.5 Stream Processing Modules

One final variation of hardware glyphs are those which require memory and are therefore inherently sequential by nature. These glyphs therefore do not require developing an underlying kernel to govern their operation. Many of these types of glyphs will operate on an entire stream of data before generating a single output. Figure 4.8 illustrates the hardware glyph Stream Max, which outputs the maximum value of an input stream of pixels. This module must compare all input values of a data stream before generating an output. Although its output will

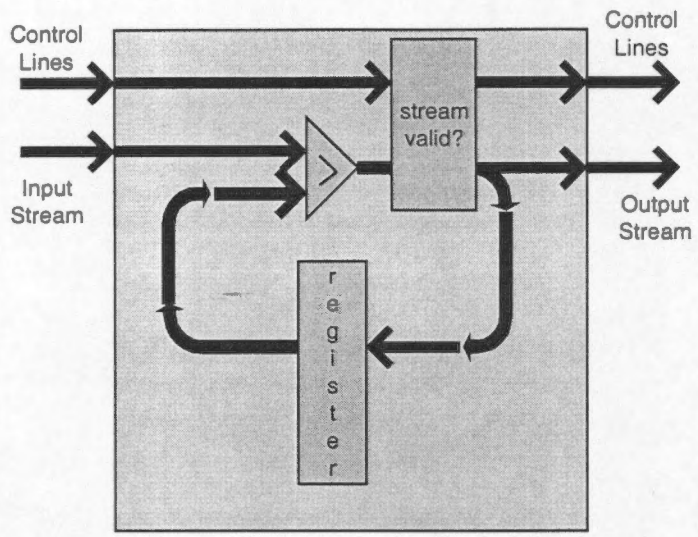


Figure 4.8: *Stream Processing Module: Stream Max.*

technically be a stream, the stream will simply be the same data value repeated over and over again. This output stream will only discontinue when a new input stream is applied to the glyph. The *Stream Valid* control line is used by the hardware glyph to determine when the input stream is over so that it can output a value. The hardware glyph will also use the *Valid Pixel Location* and *Data Valid* control lines to ignore corrupted or meaningless data.

4.6 Hardware Glyph Framework

In this chapter, the framework for developing hardware glyphs was established. Two-dimensional images from the Khoros environment were first translated into one-dimensional streams to be processed in the hardware environment.

Khoros glyphs operate inside a Cantata workspace in a synchronized manner. The insertion of delay elements by Champion and the development of three control lines served as a means to help synchronize the processing of data by hardware glyphs. These design criteria were used in the next chapter to develop the first set of hardware glyphs to be used by the Champion software.

CHAPTER 5

Hardware Glyph Development

5.1 Pre-compiled Glyphs

The purpose for developing these hardware glyphs is for the use of Champion. Champion's goal is to substantially decrease the time needed to map a software design developed in Khoros into a hardware implementation of that design mapped over a multi-FPGA platform. One aspect for the decrease in mapping time is that Champion will use a pre-developed hardware library in its execution.

Champion's initial library will contain the hardware glyphs developed in this chapter. Therefore, it was necessary to insure proper functionality for each pre-compiled hardware glyph. Champion will use several attributes including size, I/O count, and delay from the hardware glyphs in its library during the mapping process. Algorithms developed in Khoros required the hardware equivalent glyphs to be parameterizable to accommodate various bit widths. For the library of pre-complied hardware glyphs to be complete, different versions of the each hardware glyph were necessary which differ only in bit size. Therefore, all hardware glyphs were developed using parameterized VHDL templates to accommodate this necessity efficiently. These templates were modified using the

hardware glyph's parameters and then synthesized to generate XNF files. These XNF files are what will be stored as the library of pre-compiled hardware glyphs.

Figure 5.1 shows the methodology for developing a hardware glyph. The first step in this process was to clearly establish the glyph's functionality. Second, a hardware equivalent glyph was developed using a high level programming language such as C++. This hardware equivalent glyph was supplied with a set of input test vectors (an input image) to generate a set of output test vectors. Then, a hardware glyph was developed in VHDL to have the same functionality as the hardware equivalent glyph. This VHDL code was synthesized and simulated with the same test vectors used previously. Finally, the hardware glyph was placed and routed for a Xilinx FPGA platform. Once again, the same set of test vectors were used to insure proper functionality of the hardware glyph on the hardware device. If the hardware simulation or execution did not generate the same output test vectors generated by the hardware equivalent glyph, then the VHDL source code for the hardware glyph or the source code for the hardware equivalent glyph were modified to correct the discrepancy.

5.2 Hardware Glyph Kadd

The first VHDL hardware glyph template developed is called champion_kadd_#g_num_input_bits#. This VHDL template adds two input streams of the same bit width together to generate an output stream which is one bit larger than the input streams. The attribute #g_num_input_bits# determines the bit

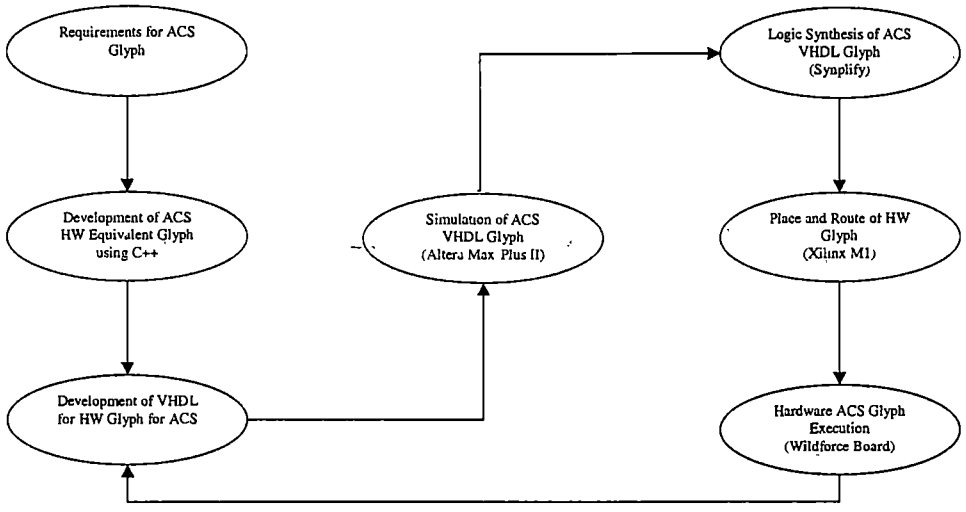


Figure 5.1: *Hardware glyph design methodology.*

widths of the two input streams. The output stream is one bit larger to avoid any overflows from the addition. Both input streams and the output stream are unsigned integers. The addition of the two streams is accomplished by invoking the **kadd** VHDL kernel developed in Chapter3. The template could easily have been developed without invoking a kernel and simply performing the addition within the VHDL template file. However, this approach was not taken in an attempt to keep the VHDL template files consistent. By doing this, many other VHDL hardware glyph templates were developed rapidly by simply replacing the **kadd** kernel with a different combinational kernel and changing file names and generics appropriately. The source code for this template and all other source code mentioned is listed in the Appendix.

The first hardware glyph developed from this template was champion_kadd_8.

The RTL layout for this hardware glyph is shown in Figure 5.2. The inputs to this hardware glyph, as well as most other hardware glyphs, are latched during the rising edge of each clock pulse. These latched values supply the input to the **kadd** kernel. The result from the current clock cycle will be present on the output g_result during the rising edge of the next clock cycle. The output control lines for this glyph is simply the result of and-ing the two input control lines.

Figure 5.3 illustrates the simulation of the RTL layout of champion_kadd_8. Two input streams are applied to the hardware glyph at the ports *g_a* and *g_b* at t = 8.0us. The values for these two ports are shown in hexadecimal notation. From close inspection of Figure 5.3, it clear that the input values are not yet applied during the rising clock edge at t = 6.0us. Therefore, it is the values held at t = 8.0us which are the first non-zero values to be latched as inputs. Soon after t = 8.0us, the resulting value is placed on the port *g_result*, which is also shown in hexadecimal notation. However, this output value would not be latched by a following glyph until the next rising clock edge, which is at t = 10.0us. Therefore this glyph has a one clock cycle latency, since each clock cycle in this simulation is 2.0us.

Two 32-kilo-pixel image files were generated using random pixels with a range of 0-255. These two images are shown in Figure 5.4a and Figure 5.4b. These two random images were added together in Khoros using an addition hardware equivalent glyph to generate the image shown in Figure 5.4c. The same two input image files were used in the simulation shown previously in Figure 5.3. The full

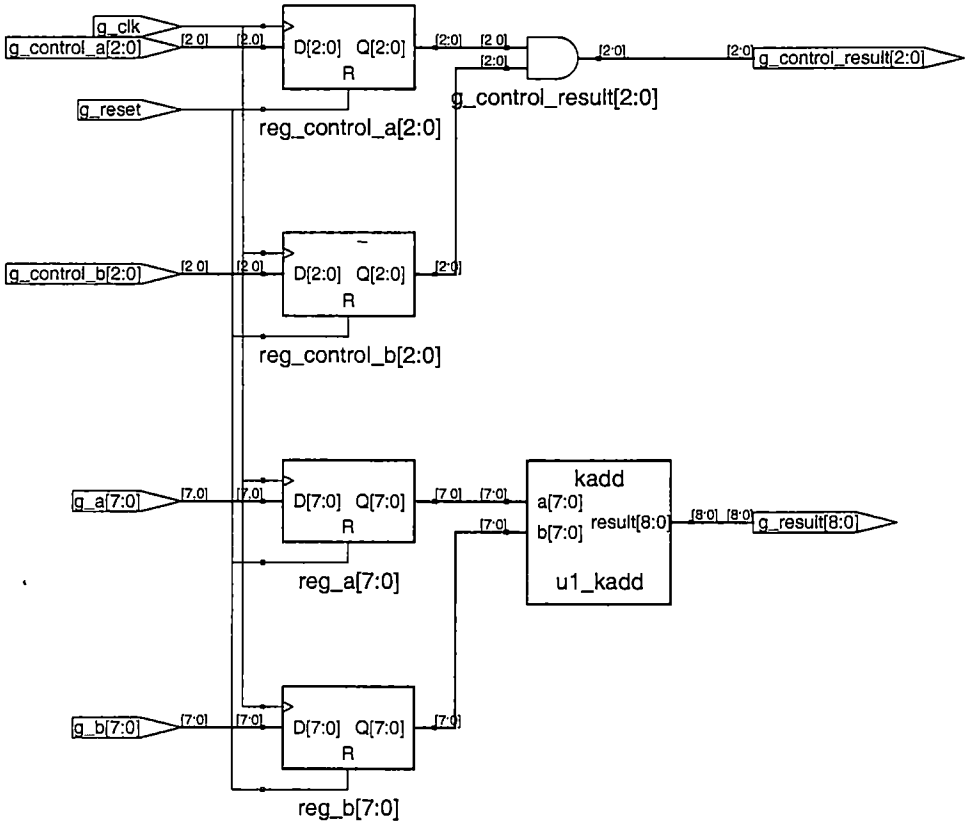


Figure 5.2: *RTL Layout for champion_kadd_8.*

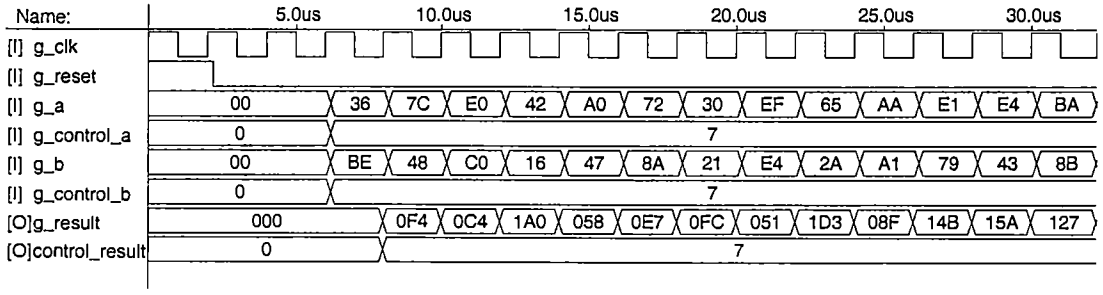


Figure 5.3: *Simulation of champion_kadd_8 RTL Layout.*

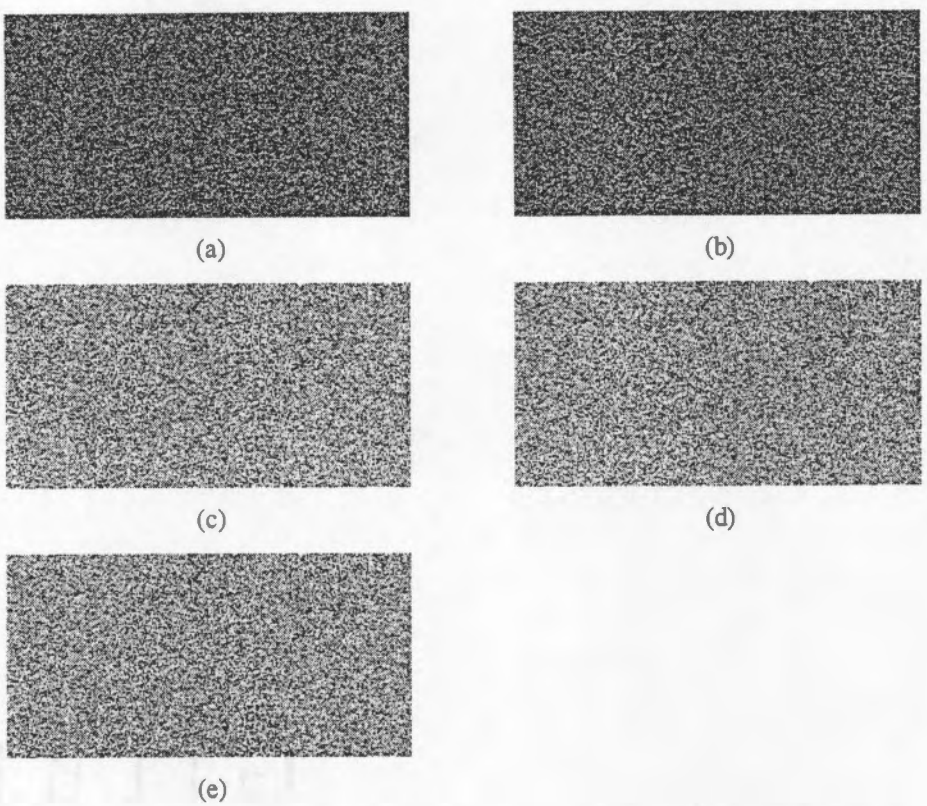


Figure 5.4: *Testing of Champion Kadd Hardware Glyph. (a) Input Image 1. (b) Input Image 2. (c) Resulting Khoros Image. (d) Resulting Simulation Image. (e) Resulting Execution Image.*

resulting stream from the port *g_result* from the simulation in Figure 5.3 is shown in Figure 5.4d. Finally, the hardware glyph champion_kadd_8 was executed on the Wildforce board. Again, the same two input images were applied as inputs to the hardware glyph. The execution results are shown in Figure 5.4e. The images in Figure 5.4c, Figure 5.4d, and Figure 5.4d are identical to one another pixel for pixel. It was therefore concluded that the VHDL hardware glyph template is designed successfully to meet the its design criteria. Attributes from this hardware glyph and all hardware glyphs developed are listed at the end of this chapter.

5.3 Hardware Glyph Kdiv

The second hardware glyph template developed was the hardware glyph for division called champion_kdiv_#g_num_input_bits#. The RTL layout for champion_kdiv_8 is shown in Figure 5.5. This glyph invokes the **kdiv** kernel developed in Chapter 3 to perform its combinational calculation. The resulting control line differs from the previous hardware glyph since the division operation can yield a divide by zero error. If the dividend *g_b* is zero, the *error* output from the **kdiv** kernel is asserted high. With this case, the Data Valid line must be cleared to indicate corrupted data in the output image stream. As mentioned in Chapter 3, the **kdiv** kernel was not able to be simulated due to technical limitations. Therefore, this hardware glyph which uses this kernel was not able to be simulated.

Figure 5.6a and Figure 5.6b illustrate two 32-kilo-pixel images. The last six

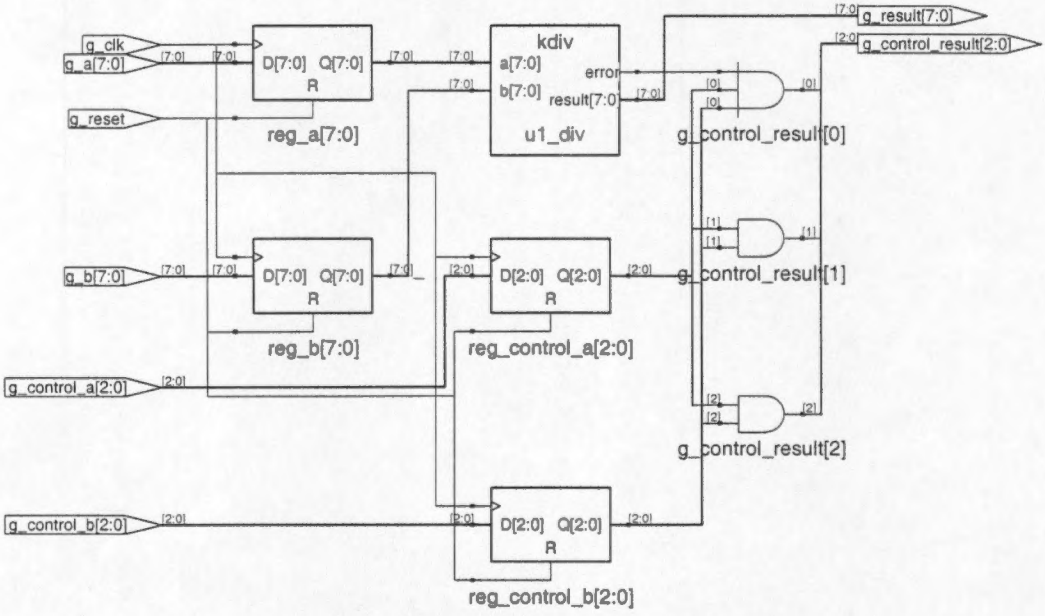


Figure 5.5: *RTL Layout for champion_kdiv_8.*

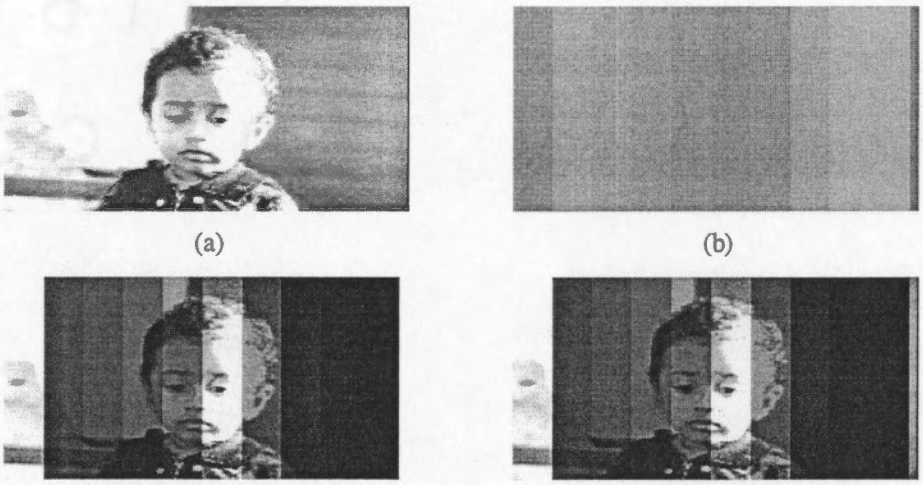


Figure 5.6: *Testing of Champion Kdiv Hardware Glyph. (a) Input Image 1. (b) Input Image 2. (c) Resulting Khoros Image. (d) Resulting Execution Image.*

columns of Figure 5.6b have zeros for their pixel values. Figure 5.6c is the resulting image generated by Khoros from dividing Figure 5.6a by Figure 5.6b. The last six columns of this image all have pixel values of 255 due to division by zero. Figure 5.6d shows the result from implementing champion_kdiv_8 using the same two input image files. The pixels from the first 250 columns are identical to the first 250 columns of Figure 5.6c. The last six columns of Figure 5.6d differ from Figure 5.6c, but they also have their associated Data Valid pixels cleared, implying corrupted data in these columns. Therefore, the template champion_kdiv_#g_num_input_bits# was assumed to be accurate in its description.

5.4 Stream Processing Glyphs

So far the hardware glyphs developed required no memory of previous pixels. Therefore, combinational kernels were developed and used in conjunction with registers to form a sequential circuit. In this manner, every input pixel generated an output pixel or value. However, some hardware glyphs require memory, are inherently sequential in nature, and use all pixel values of an input stream before generating a result. Often, the result of such hardware glyphs is no longer an image stream, but instead represents some information of the input stream such as its sum or maximum value. These types of hardware glyphs are referred to as stream processing glyphs. The major difference between stream processing glyphs and the glyphs developed previously is that they output values only after

the entire input stream is completed. These output values are held until a new stream is applied at the input. Therefore, state machines are required to develop these glyphs.

5.5 Hardware Glyph Stream Sum

The first stream processing hardware glyph template developed was champion_stream_sum_#g_num_input_bits#_#g_num_output_bits#. Hardware glyphs were synthesized from this template for specific number of input and output bits. The number of output bits for this hardware glyph is expressed by Equation 5.1.

$$\#\ of\ output\ bits = log_2(\#\ of\ inputs\ pixels) + \#\ of\ input\ bits \qquad (5.1)$$

The RTL layout for champion_stream_sum_8_23 is shown in Figure 5.7. When the glyph's reset line is asserted, all output lines are cleared. Once the reset line goes low, the glyph moves into its normal mode of operation. The glyph waits for an input stream. Once the input stream is completed, the glyph outputs the stream's sum and the appropriate control lines. The glyph then holds the value of the sum until a new input stream is applied. Once applied, the output of the glyph returns low until the new stream is completed. Figure 5.8 shows the final clock cycles of a simulation with an input stream corresponding to the image shown previously in Figure 5.6a. After a latency of the number of input pixels plus two additional clock cycles, the glyph outputs the sum of 5,556,910. This value obtained in simulation of the hardware glyph was also obtained from

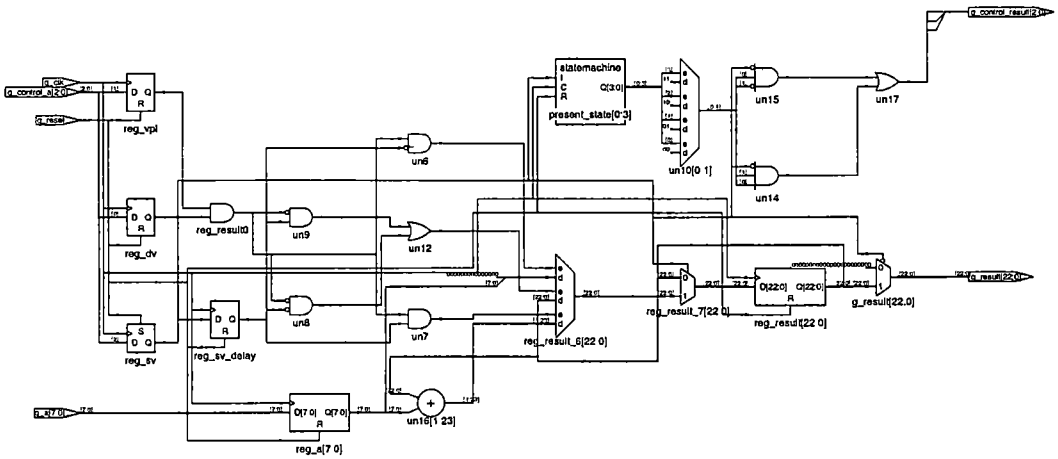


Figure 5.7: *RTL Layout for champion_stream_sum_8_23.*

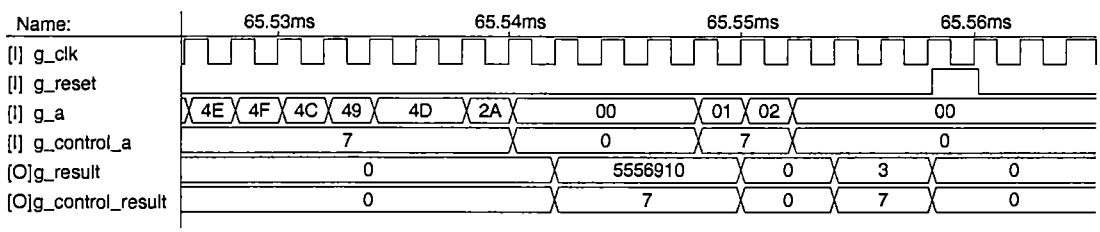


Figure 5.8: *Simulation of champion_stream_sum_8_23 RTL Layout.*

a Khoros hardware equivalent glyph and from executing the hardware glyph on the Wildforce board, indicating that the VHDL hardware glyph template champion_stream_sum_#g_num_input_bits#_#g_num_output_bits# was correct in its design.

5.6 Hardware Glyph Stream Max

The hardware glyph for stream max is similar to that of stream sum except that the glyph outputs the input stream's maximum value, the row index for the maximum value, and the column value for the maximum value. Therefore, the hardware glyph template file has parameters for the number of bits needed to represent the input pixels, the number of bits needed to represent the number of rows, the number of bits needed to represent the number of columns, and also the number of columns in the input image. The VHDL template developed is called champion_stream_max_#g_num_input_bits#_#g_num_r_bits#_#g_num_c_bits#_#g_num_cols#. The RTL layout for champion_stream_max_8_7_8_256 is shown in Figure 5.9. If the input stream has two or more pixels with the same maximum value, the hardware glyph outputs the row and column indices for the first occurrence of that pixel value. The order for this search starts with the first row, then the second, and so forth. Figure 5.10 shows the final few clock cycles of the RTL simulation of champion_stream_max_8_7_8_256. The hardware glyph outputs the input streams maximum value, its row index, and its column index three clock cycles after the input stream is complete. Therefore, this hardware

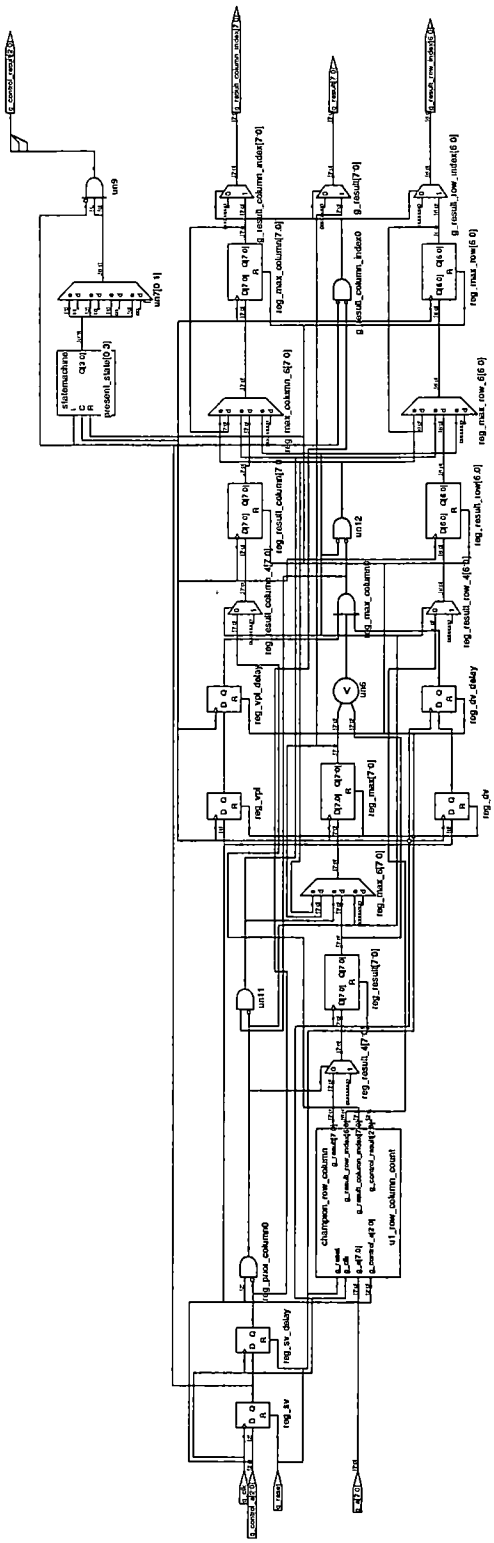


Figure 5.9: *RTL Layout for champion_stream_max_8_7_8_256.*

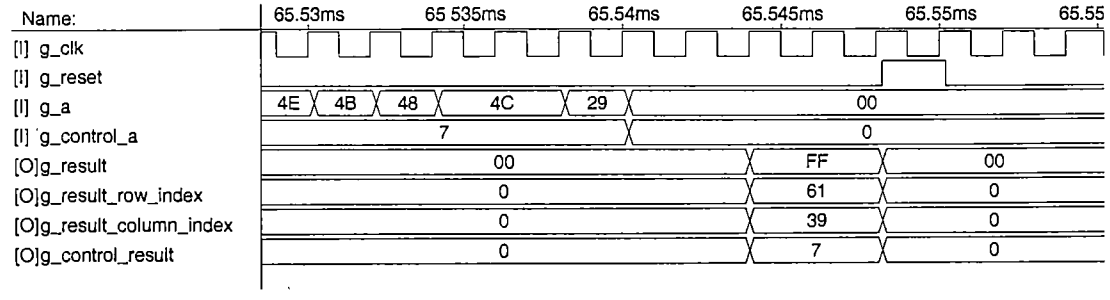


Figure 5.10: *Simulation of champion_stream_max_8_7_8_256 RTL Layout.*

glyph template has a latency of three clock cycles plus the number of input pixels in the stream. The same output data was obtained from executing a hardware equivalent glyph in Khoros and from executing the hardware glyph on the Wildforce board.

5.7 Compile-time Glyphs

Some hardware equivalent glyphs have one input stream which is operated on using a constant. An example of this is a hardware glyph which adds a constant to each input pixel. It is not feasible to generate a hardware glyph for each possible constant that a user of Champion may require. Therefore, VHDL template files are created for compile-time glyphs. When a user invokes a compile-time hardware glyph in Cantata, Champion must first substitute the constant into the VHDL template file and then synthesize the hardware glyph.

The newly synthesized XNF file is saved in a user library, rather than in the pre-compiled library, so that it can be used in future designs. This option requires the user of Champion to have the necessary synthesis as well as place and route tools to develop a user library.

5.8 Compile-time Hardware Glyph Kadd

The first compile-time hardware glyph template developed was champion_kadd_#g_num_input_bits#_#g_constant#. The parameters for the number of input bits and the constant are substituted with values of 8 and 17, respectively. Figure 5.11 shows the RTL layout for this hardware glyph, champion_kadd_8_17. This hardware glyph uses the **kadd** kernel. The constant value is generated only when the input stream has its stream valid, valid pixel location, and data valid lines asserted high. Figure 5.12 compares the results between Khoros, RTL simulation, and Wildforce execution of this hardware glyph. In this case, Figure 5.12a shows the original input image used for all three cases. When compared pixel by pixel, each of the three output images shown in Figures 5.12b, c, and d are identical. When a Khoros user requires the use of the hardware equivalent glyph champion_kadd with a constant, Champion will modify the VHDL template file to insert the value of the constant, in this case 17, for the parameter #g_constant#. Therefore, for a user of Champion to use compile-time hardware glyphs, he must have synthesis and place & route tools to create a user library of glyphs.

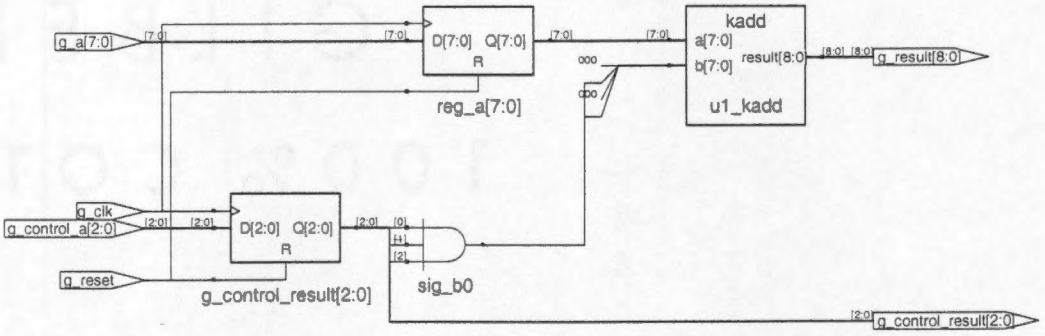


Figure 5.11: *RTL Layout for champion_kadd_8_17.*

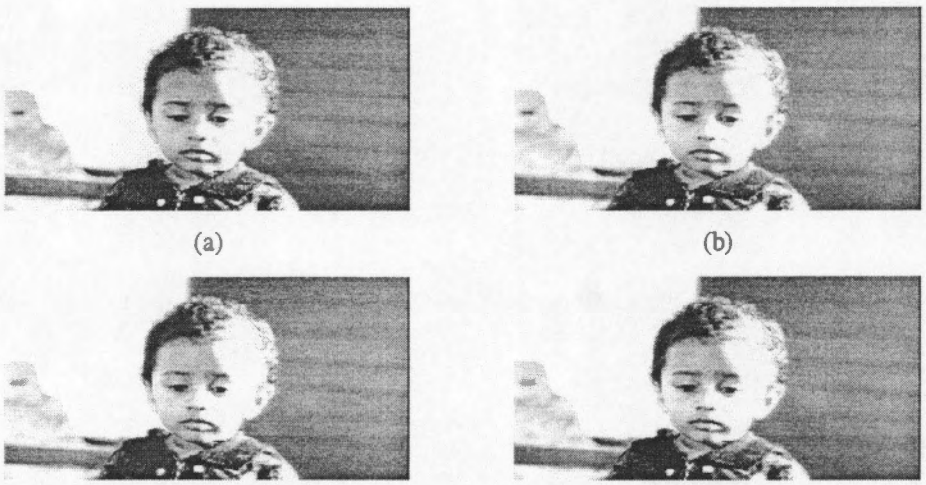


Figure 5.12: *Testing of Champion Kadd Compile-time Hardware Glyph. (a) Input Image. (b)Resulting Khoros Image. (c) Resulting Simulation Image. (d) Resulting Execution Image.*

5.9 Hardware Glyph Development Status

Table 5.1 shows the resulting attributes from the five hardware glyphs developed in this chapter. Champion will use these attributes for its partitioning algorithms and for data synchronization of Khoros workspaces. Over twenty other hardware glyph templates have been developed in addition to these first five to server as the first set of library cells for Champion to use. Other cells will be developed in VHDL, validated through simulation, verified through execution, and characterized by their attributes using the same procedures and design criteria as were used to design these first five library cells described in this chapter.

Table 5.1: *Hardware Glyph Attributes*

Hardware Glyph	Frequency (MHz)	Inputs	Outputs	CLB Utilization	Latency (# of clock cycles)
kadd_8	94.3	g_a 8 ,g_b 8	g_result 9	11	1
kdiv_16	10.6	g_a 16 ,g_b 16	g_result 16	31	1
stream_sum_8_23	62.1	g_a 8	g_result 23	36	32,770
stream_max_8_7_8_256	50.5	g_a 8	g_result 8 g_row 7 g_column 8	49	32,771
kadd_8_17	68.8	g_a 8	g_result 9	6	1

CHAPTER 6

Hardware Glyph Library

6.1 Library Files

The VHDL template files were used to generate XNF files for each hardware glyph. These synthesized XNF files were targeted to a specific hardware architecture. The XNF files generated in Chapter 5 were all targeted for Xilinx 4013XL FPGAs. These XNF files when grouped together were what actually constitute the library cells for Champion to use in its mapping and partitioning algorithms. Along with each XNF file generated, a simple text file was written to document each library cell. Each text file contained information pertaining to the maximum clock speed, number of CLBs, I/O names and sizes, and latency for its associated XNF file. Champion will parse these simple text files to determine how to generate a structural VHDL file which will connect the hardware glyphs together as dictated by a Khoros workspace and the target multi-FPGA platform.

6.2 Manual Mapping of Hardware Glyphs

The purpose for developing hardware glyphs is for use by Champion to map these glyphs from a Khoros workspace to an ACS environment. Champion will

map these glyphs automatically together using a structural VHDL file. To accomplish this task, Champion must extract the entity declarations from the VHDL files of the hardware glyphs used in the Khoros workspace. It must then generate a structural VHDL file invoking each hardware glyph entity as a black box component. By instantiating the hardware glyphs as black boxes, the hardware glyphs will not be re-synthesized during the mapping procedure. Instead, only the high level structural VHDL file is synthesized to generate an XNF file which connects the hardware glyphs (which are just XNF files themselves) together. In this chapter, two examples were investigated by mapping the glyphs together manually. This manual process not only gave insight into the requirements for Champion's automatic mapping process, but it insured that the glyphs worked together as intended by design.

6.3 Combining Hardware Glyphs

The first example is shown in Figure 6.1. The three addition hardware glyphs used in this figure were mapped together manually using a structural VHDL file. The circuit developed adds four input image streams together. The images on the input lines w and x as well as y and z are added together first by the components *u1* and *u2*, respectively. These intermediate results are then added together one clock cycle later by the component *u3*. Figure 6.2 shows a brief simulation of four input image streams each containing four pixels. The port *final_result* in the simulation represents the output of the circuit, which indeed corresponds to the

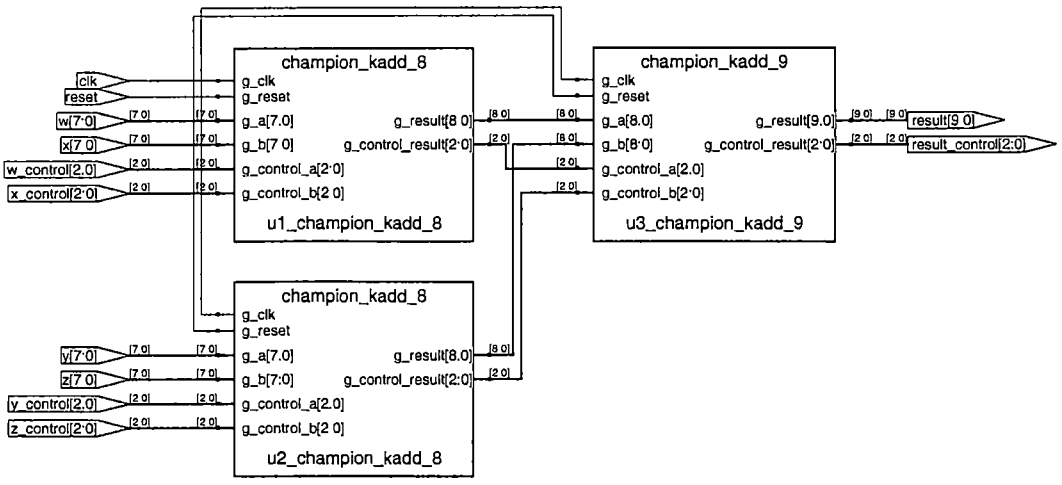


Figure 6.1: *Example 1: Addition Hardware Glyph Layout.*

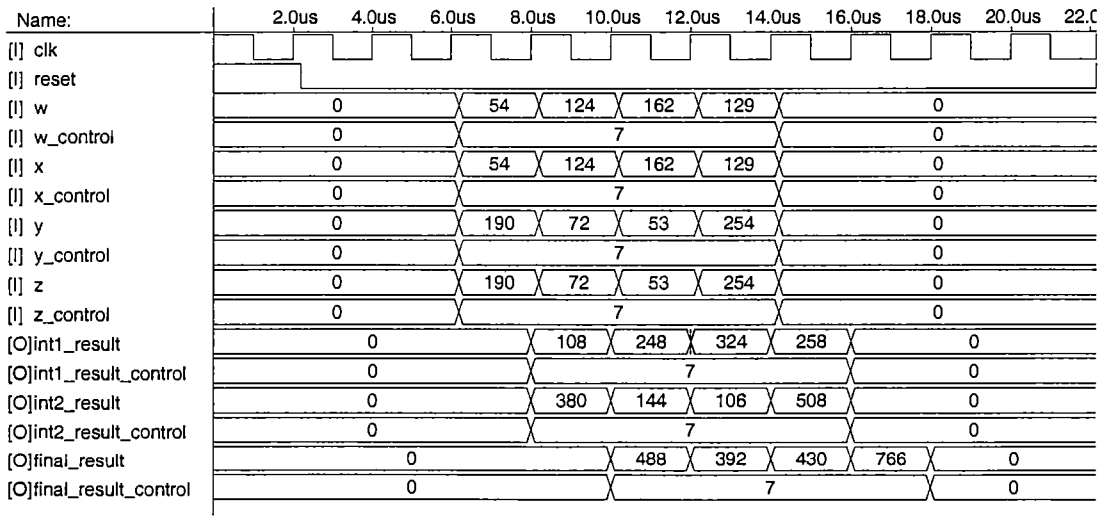


Figure 6.2: *Simulation Results of Example 1.*

addition of the four input streams, after a two clock cycle delay.

The second example is shown in Figure 6.3. The four hardware glyphs used in this circuit are also chained together using a structural VHDL file. A 32 kilo-pixel input image is applied to this circuit for simulation. The final few clock cycles of this simulation is shown in Figure 6.4. The component *u1* holds its value at its output until a new input stream is applied. Therefore, once the input stream is finished, it outputs a value of 5,556,910 and holds it at its output. This number is right shifted fifteen places by *u2* to yield 169. Component *u3* adds one to this input to generate a stream of 170's. Finally, component *u4* shifts the stream of 170's by one binary digit to generate an output of 85. This value, which is the input image's mean value, is held at the output during the simulation until the reset line is asserted high for a clock cycle.

From the simulation diagrams of these two examples, it is clear that the glyphs do work according to their intended design. However, it is clear from these two examples that the hardware glyphs used were generated specifically to match each others input and output bit widths. Also, both circuits did not have any latency/synchronization issues to deal with. However, a Khoros workspace file may be generated where hardware glyphs are not synchronized and/or do not have proper bit alignment.

In the first example, if the inputs to the first addition component *u1* were seven bits wide, then its output would be eight bits wide. However, the addition component *u3* is expecting its inputs to be nine bits wide. Therefore a zero

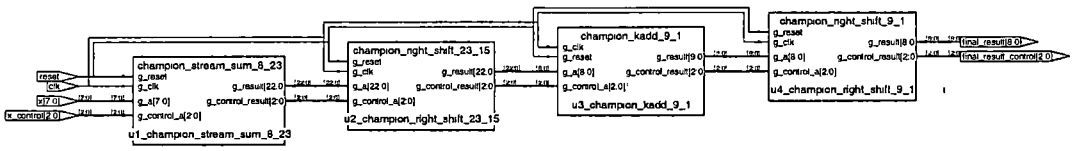


Figure 6.3: *Example 2: Mean Calculation Hardware Glyph Layout.*

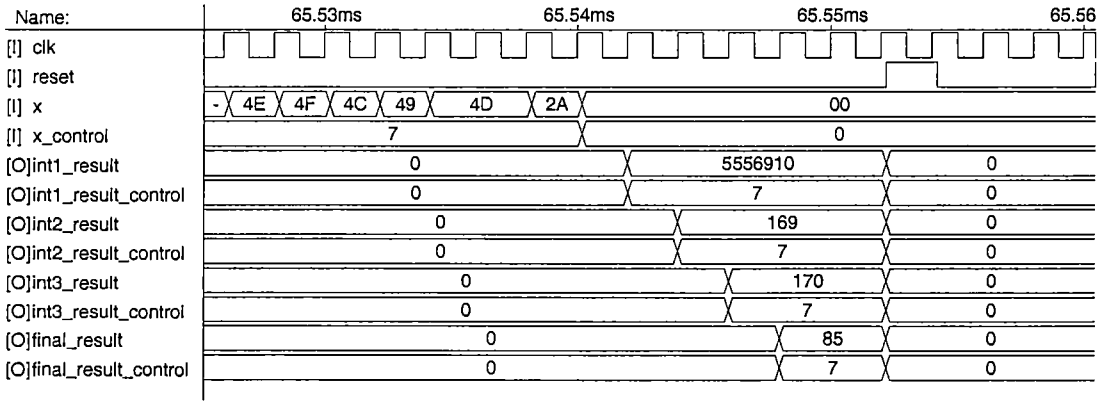


Figure 6.4: *Simulation Results of Example 2.*

padding element would need to be inserted between these two components to properly align their interconnection. In the opposite case, the if input to *u1* were nine bits, then its output would be ten bits wide. If the Khoros workspace called for component *u3* to have only a nine bit input, then a truncating element would need to be inserted to either drop the MSB, drop the LSB, or saturate the output value of component *u1*.

Also, in these two examples, there were no synchronization glitches to deal with. However, there can arise a situation where data will not arrive from two different sources at the same time to a hardware glyph. Therefore, before a structural VHDL can be generated by Champion, a check must be done on the Khoros translated netlist to determine if all data is properly synchronized to arrive to each hardware glyph at the proper times based on the latencies of the hardware glyphs used in the Khoros workspace. If this not the case, delay elements will need to be inserted by Champion to accommodate the required synchronization of data. For small discrepancies in synchronization, registers may be used to delay the arrival of a stream of data to a hardware glyph. For large differences in synchronization, the data stream will need to be stored in RAM until required by the next hardware glyph in the design's pipeline.

CHAPTER 7

Summary, Conclusion, and Future Work

7.1 Summary

The purpose of developing hardware glyphs is for use by Champion to map designs developed in a software environment to a hardware platform. The methodology for developing a hardware glyph is a three-fold process entailing the description, simulation, and execution of each hardware glyph. For the description stage, the VHDL hardware description language was used to develop the hardware glyphs. Using CAD tools, the code was synthesized to generate a Xilinx Netlist File (XNF) for each glyph. During the simulation stage, each glyph's VHDL code was simulated using test vectors generated by an associated hardware equivalent glyph to ensure proper design functionality. This hardware equivalent glyph was developed using a high level programming language such as C++. Finally, during the execution stage, synthesized glyphs were downloaded into an AMS Wildforce Board to executed on a Xilinx 4013-XL FPGA. Execution results were compared to the expected output values obtained during simulation.

Using this methodology, five hardware glyph templates were developed in VHDL using generics for parameterized bit widths. All templates were written with adequate output bit sizes to avoid overflows and underflows. The VHDL

source code was written with integer inputs and outputs using unsigned logic. The RTL netlist for each glyph is simulated with Altera Max Plus II. Also, each glyph was synthesized using Synplify to generate an XNF file for a 4013-XL FPGA. These XNF files were instantiated for execution on a Wildforce board with the execution results compared to simulation results.

The development of hardware glyphs was mostly governed by the differences between Khoros and an FPGA hardware platform. For example, some Khoros glyphs have multiple functions for the user to choose from. Data between these glyphs are stored on a host system as temp files. Cantata handles data movement between these glyphs. Each glyph begins its execution only after all its inputs have been written to a host file system when running inside a Cantata workspace. A single Khoros glyph can handle all input dimension sizes. Finally, these glyphs can handle inputs of any data type. Conversely, for hardware execution, each hardware glyph has only one specific function. Data is transferred from hardware glyphs sequentially one pixel at a time per clock cycle. Synchronization of data arrival to hardware glyphs is necessary through the insertion of delay elements by Champion. Also, some hardware glyphs were developed for specific image sizes. Finally, each hardware glyph was developed for specific I/O bit widths.

In addition to testing the hardware glyphs individually, the glyphs were used to develop test circuits to ensure that they work together as intended. This process employs the use of a structural VHDL file to instantiate pre-compiled hardware glyph entities to synthesize a high level XNF file. This XNF file consists

of only the interconnects between each hardware glyph. The structural VHDL file uses Black Box Attributes for each hardware glyph to avoid re-synthesizing the hardware glyphs. Since the hardware glyphs were already synthesized, this synthesis process was accomplished quickly. Xilinx M1 Place and Route Tools were used to merge the top level XNF file with the XNF files of each hardware glyph employed. This process was investigated as a means to develop a procedure for the automatic mapping of a design to an FPGA platform by Champion.

The hardware glyphs developed in this thesis will server as the first set of library cells for Champion to use during its partitioning and mapping procedures. Also, the development of these library cells set the initial design criteria and establishes the methodology for validating through CAD simulation and verifying through hardware execution future library cells for the Champion project. Finally, lessons learned from developing these cells and using them to create operating circuits aid in the design of the Champion software.

7.2 Conclusion

It is concluded that the adaptive computing system Champion will greatly decrease the time for a design engineer to map an algorithm from a software prototype to a multi-FPGA implementation of the algorithm. One major cause for this decrease in mapping was due to the use of pre-compiled library cells by Champion in this mapping procedure. Since the library cells are pre-compiled, the user can also gain valuable estimated knowledge of a design's hardware implementa-

tion and execution metrics before actually mapping the design to a multi-FPGA platform. Such estimates will be based on the latency, maximum clock speed, usage of CLBs, and number of I/O ports for each hardware glyph that is used in a design. Based on these parameters for the glyphs used in a design, Champion will be able to give its users estimates for information such as the number of FPGAs and FPGA reconfigurations needed for a given hardware platform, as well as the maximum circuit operating clock speed. For a manual mapping of a design without the use of primitives from a pre-compiled library, such estimates are not available since the hardware building blocks are not yet developed. Also when mapping a design manually, development time is needed to develop the necessary hardware circuits for a design. However, Champion's use of a pre-compiled library therefore eliminates this hardware development time from the design cycle.

7.3 Future Work

For this library of hardware glyphs to be useful, they were modeled in Khoros using hardware equivalent glyphs. This way, a designer can develop an algorithm using these hardware equivalent glyphs in a high level of abstraction without making hardware implementation decisions. Therefore, a developer can design an algorithm using these hardware equivalent glyphs just as if they were native Khoros glyphs. Upon design completion, the user can invoke Champion to map the Cantata workspace which uses hardware equivalent glyphs to a hardware implementation of the design using the actual hardware glyphs developed in this

thesis. Therefore, it is important to not only develop the hardware glyphs, but to also adequately develop the hardware equivalent glyphs. This task is quite necessary since the user will input his design in a Cantata workspace using a hardware equivalent glyph toolbox. Therefore, the development of a hardware equivalent toolbox is necessary that can be accessed through the Cantata graphical user interface. Another aspect that can be explored is the development of hardware glyphs for data types other than unsigned integers. Finally, macro-level hardware glyphs for more intricate image processing routine must be developed from the basic hardware glyphs developed in this thesis project.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Benjamin Levine, Senthil Natarajan, Chandra Tan, Danny Newport, Don Bouldin, "Mapping of an Automated Target Recognition Application from a Graphical Software Environment to FPGA-based Reconfigurable Hardware", 1999.

[2] D.W. Bouldin, "CHAMPION: A Software Design Environment for Adaptive Computing Systems", http://www.microsys6.engr.utk.edu/ bouldin/darpa/.

[3] "Cameron: Optimized Compilation of Visual Programs for Image Processing on Adaptive Computing Systems", http://www.cs.colstate.edu/cameron/.

[4] Zijun Shen, "A Development Environment for Reconfigurable Computing", Master's thesis, University of Tennessee, Knoxville, 1996.

[5] Xilinx, "Xilinx: The Programmable Logic Data Book", 1998.

[6] Annapolis Micro Systems, Inc., "Wildforce Reference Manual", 1998.

[7] "Binary Division", http://www.txdirect.net/users/uswatson/division.htm.

APPENDIX

Appendix

```
-- kadd.vhd
-- by: senthil natarajan
--
-- description: inputs two unsigned std_logic_vectors and returns
--              a std_logic_vector that is one bit larger than the
--              input vectors


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity kadd is
        generic (num_bits : positive);
        port ( a,b:      in std_logic_vector (num_bits -1 downto 0);
               result:     out std_logic_vector(num_bits downto 0) );

end kadd;

architecture behave of kadd is
   signal a_ext, b_ext : std_logic_vector(num_bits downto 0);
begin
   a_ext <= '0' & a;
   b_ext <= '0' & b;
   result <= a_ext + b_ext;
end;



-------------------------------------------------------
-- ksub.vhd
-- by: senthil natarajan
--
-- description:
-- inputs two signed std_logic_vectors with a bit width of
-- 'num_bits' and returns a signed std_logic_vector with
-- bit width of 'num_bits +1'.
-------------------------------------------------------

library ieee;
```

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

entity ksub is
 generic (num_bits: positive:= 8);
 port (a,b: in std_logic_vector(num_bits -1 downto 0);
       result: out std_logic_vector(num_bits downto 0));
end ksub;

architecture behave of ksub is
 signal a_ext, b_ext: std_logic_vector(num_bits downto 0);
begin
 a_ext <= a(num_bits-1) & a;
 b_ext <= b(num_bits-1) & b;
 result <= a_ext - b_ext;
end;

------------------------------------------------------
-- kabs.vhd
-- by: senthil natarajan
--
-- description:
-- inputs a signed std_logic_vectors and returns
-- a signed std_logic_vector that is one bit larger
-- than the input vector. It is one bit larger to
-- accomidate -128 => +128 (8 bit example).
------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity kabs is
 generic (num_bits: positive:= 8);
 port (a: in std_logic_vector (num_bits -1 downto 0);
       result: out std_logic_vector(num_bits downto 0) );
end kabs;

architecture behave of kabs is
begin
 process (a)
  variable a_sig: std_logic_vector(num_bits downto 0);
```

```
 begin
  if (a(num_bits -1) = '1') then
   a_sig := '1' & a;
   result <= not(a_sig) + 1;
  else
   result <= '0' & a;
  end if;
 end process;
end;


------------------------------------------------------------------
-- kabsdiff.vhd
-- by: senthil natarajan
--
-- description:
-- inputs two signed std_logic_vectors with bit widths 'num_bits'
-- and returns an signed std_logic_vector that is one bits larger
-- than the input vectors, 'num_bits +1'.
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity kabsdiff is
 generic (num_bits: positive:= 8);
 port (a,b: in std_logic_vector(num_bits -1 downto 0);
       result: out std_logic_vector(num_bits downto 0) );
end kabsdiff;

architecture behave of kabsdiff is
 signal a_ext, b_ext, result_temp :
        std_logic_vector(num_bits downto 0);
begin
 a_ext <= a(num_bits-1) & a;
 b_ext <= b(num_bits-1) & b;
 result_temp <= a_ext - b_ext;
 process (result_temp)
 begin
  if (result_temp(num_bits) = '1') then
   result <= not(result_temp) + 1;
  else
   result <= result_temp;
```

```
  end if;
 end process;
end;


--------------------------------------------------
-- kdiv.vhd
-- Senthil Natarajan
--
-- description:
-- inputs two unsigned std_logic_vectors of bit widths
-- 'num_bits' and returns a std_logic_vector with the
-- same bit width that is the result of a/b.
--------------------------------------------------

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

entity kdiv is
 generic (num_bits: positive:= 3);
 port (a,b: in std_logic_vector(num_bits -1 downto 0);
       error: out std_logic;
       result: out std_logic_vector (num_bits -1 downto 0));
end kdiv;

architecture behave of kdiv is
begin
 process (a,b)
  variable a_var,b_var,b_neg_var,p:
           std_logic_vector (num_bits -1 downto 0);
  variable sign: std_logic;
 begin
  error <= '0';
  a_var := a;
  b_var := b;
  p := (others => '0');
  b_neg_var := not b_var + '1';
  if (b_var = 0) then
   error <= '1';
   result <= (others => '0');
  else
   for i in 1 to num_bits loop
```

```
    p := p(num_bits -2 downto 0) & a_var(num_bits -1);
    if (p >= b_var) then
     p := p + b_neg_var;
     a_var := a_var(num_bits -2 downto 0) & '1';
    else
     a_var := a_var(num_bits -2 downto 0) & '0';
    end if;
   end loop;
  end if;
  result <= a_var;
 end process;
end behave;


--------------------------------------------------
-- kmod.vhd
-- Senthil Natarajan
--
-- description:
-- inputs two unsigned std_logic_vectors of bit widths
-- 'num_bits' and returns a std_logic_vector with the
-- same bit width that is the remainder of a/b.
--------------------------------------------------

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

entity kmod is
 generic (num_bits: positive:= 2);
 port (a,b: in std_logic_vector(num_bits -1 downto 0);
       error: out std_logic;
       result: out std_logic_vector (num_bits -1 downto 0));
end kmod;

architecture behave of kmod is
begin
 process (a,b)
  variable a_var,b_var,b_neg_var,p:
           std_logic_vector (num_bits -1 downto 0);
  variable sign: std_logic;
 begin
  error <= '0';
```

```
  a_var := a;
  b_var := b;
  p := (others => '0');
  b_neg_var := not b_var + '1';
  if (b_var = 0) then
   error <= '1';
   result <= (others => '0');
  else
   for i in 1 to num_bits loop
    p := p(num_bits -2 downto 0) & a_var(num_bits -1);
    if (p >= b_var) then
     p := p + b_neg_var;
     a_var := a_var(num_bits -2 downto 0) & '1';
    else
     a_var := a_var(num_bits -2 downto 0) & '0';
    end if;
   end loop;
  end if;
  result <= p;
 end process;
end behave;


-------------------------------------------------------------------
-- kpow.vhd
-- by: senthil natarajan
--
-- description: inputs an unsigned std_logic_vectors with a generic
-- bit width of 'num_bits_a'. The result is the input vector taken
-- to the generic 'pow'-power. The result is an unsigned
-- vector of length pow * num_bits_a.
-------------------------------------------------------------------

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

entity kpow is
 generic (num_bits_a : positive:= 8;
          pow: positive:= 4);
 port (a: in std_logic_vector(num_bits_a -1 downto 0);
       result: out std_logic_vector(pow * num_bits_a -1 downto 0));
end kpow;
```

```
architecture behave of kpow is
 constant size_a : integer := a'length;
begin
 process(a)
  variable r,q : std_logic_vector(2048 downto 0);
  -- Synplify will not compile this code unless the variables
  -- r and q have a high width.
  -- It has a problem invoking the right size multipliers when
  -- the vectors are to small.
 begin
  r(size_a -1 downto 0) := a;
  r(31 downto size_a) := (others => '0');
  q := r;
  for i in 2 to pow loop
   r((size_a*(2**(i -1)))-1 downto 0) :=
      r((size_a*(2**(i-2)))-1 downto 0) *
      q((size_a* (2** (i-2))) -1 downto 0);
  end loop;
  result <= r(pow * num_bits_a -1 downto 0);
 end process;
end;


------------------------------------------------------------------
-- champion_kadd_#g_num_input_bits#.vhd
-- by: senthil natarajan
--
-- description:
-- hardware equivalent for glyph kadd with two inputs
--  of #g_num_input_bits# bits.
-- result is #g_num_input_bits# +1.
------------------------------------------------------------------

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

entity champion_kadd_#g_num_input_bits# is
 generic (g_num_input_bits: positive:= #g_num_input_bits#;
          g_num_control_bits: positive:= 3);
 port (g_a,g_b: in std_logic_vector(g_num_input_bits -1 downto 0);
       g_control_a, g_control_b: in
          std_logic_vector(g_num_control_bits -1 downto 0);
```

```
       g_clk, g_reset: in std_logic;
       g_result: out std_logic_vector(g_num_input_bits downto 0);
       g_control_result: out
          std_logic_vector(g_num_control_bits -1 downto 0));
end champion_kadd_#g_num_input_bits#;

architecture behave of champion_kadd_#g_num_input_bits# is
 component kadd is
  generic (num_bits: positive);
  port (a,b: in std_logic_vector (num_bits -1 downto 0);
        result: out std_logic_vector(num_bits downto 0) );
 end component;
 signal reg_a, reg_b:
        std_logic_vector(g_num_input_bits -1 downto 0);
 signal reg_control_a,reg_control_b:
        std_logic_vector(g_num_control_bits -1 downto 0);
begin
  process (g_clk, g_reset)
 begin
  if g_reset = '1' then
   reg_a <= (others => '0');
   reg_b <= (others => '0');
   reg_control_a <= (others => '0');
   reg_control_b <= (others => '0');
  elsif rising_edge(g_clk) then
   reg_a <= g_a;
   reg_b <= g_b;
   reg_control_a <= g_control_a;
   reg_control_b <= g_control_b;
  end if;
 end process;
 g_control_result <= reg_control_a and reg_control_b;
 u1_kadd: kadd generic map(num_bits => g_num_input_bits)
              port map(a => reg_a, b => reg_b, result=> g_result);
end;


------------------------------------------------------------------
-- champion_kdiv_#g_num_input_bits#.vhd
-- by: senthil natarajan
--
-- description:
-- hardware equivalent for glyph kdiv with two
-- inputs of #g_num_input_bits# bits.
```

```
-- result is #g_num_input_bits# also #g_num_input_bits# bits.
------------------------------------------------------------

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

entity champion_kdiv_#g_num_input_bits# is
 generic (g_num_input_bits: positive:= #g_num_input_bits#;
          g_num_control_bits: positive:= 3);
 port (g_a,g_b: in std_logic_vector(g_num_input_bits -1 downto 0);
       g_control_a, g_control_b: in
            std_logic_vector(g_num_control_bits -1 downto 0);
       g_clk, g_reset: in std_logic;
       g_result: out std_logic_vector(g_num_input_bits -1 downto 0);
       g_control_result: out
            std_logic_vector(g_num_control_bits -1 downto 0));
end champion_kdiv_#g_num_input_bits#;

architecture behave of champion_kdiv_#g_num_input_bits# is
component kdiv is
 generic (num_bits: positive:= 3);
 port (a,b: in std_logic_vector(num_bits -1 downto 0);
       error: out std_logic;
       result: out std_logic_vector (num_bits -1 downto 0));
end component;
 signal reg_a, reg_b:
        std_logic_vector(g_num_input_bits -1 downto 0);
 signal reg_control_a,reg_control_b:
        std_logic_vector(g_num_control_bits -1 downto 0);
 signal sig_error: std_logic;
begin
  process (g_clk, g_reset)
 begin
  if g_reset = '1' then
   reg_a <= (others => '0');
   reg_b <= (others => '0');
   reg_control_a <= (others => '0');
   reg_control_b <= (others => '0');
  elsif rising_edge(g_clk) then
   reg_a <= g_a;
   reg_b <= g_b;
   reg_control_a <= g_control_a;
```

```
   reg_control_b <= g_control_b;
  end if;
 end process;

 g_control_result(2) <= reg_control_a(2) and reg_control_b(2);
 g_control_result(1) <= reg_control_a(1) and reg_control_b(1);
 g_control_result(0) <= reg_control_a(0) and reg_control_b(0)
                        and (not sig_error);
 u1_div: kdiv generic map(num_bits => g_num_input_bits)
              port map(a => reg_a, b => reg_b,
                       error => sig_error, result=> g_result);
end;


------------------------------------------------------------------
-- champion_stream_sum_#g_num_input_bits#_#g_num_output_bits#.vhd
-- by: senthil natarajan
--
-- Description: Calculates the total value of
-- a stream and holds its output until
-- a new stream is applied at the input.
------------------------------------------------------------------

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

entity champion_stream_sum_#g_num_input_bits#_#g_num_output_bits# is
 generic (g_num_input_bits: positive:= #g_num_input_bits#;
          g_num_output_bits: positive:= #g_num_output_bits#;
          g_num_control_bits: positive:= 3);
 port (g_a: in std_logic_vector(g_num_input_bits -1 downto 0);
       g_control_a: in
          std_logic_vector(g_num_control_bits -1 downto 0);
       g_reset: in std_logic;
       g_clk: in std_logic;
       g_result: out
          std_logic_vector(g_num_output_bits -1 downto 0);
       g_control_result: out
          std_logic_vector(g_num_control_bits -1 downto 0));
end champion_stream_sum_#g_num_input_bits#_#g_num_output_bits#;

architecture behave of
```

```
 champion_stream_sum_#g_num_input_bits#_#g_num_output_bits# is
 signal reg_a: std_logic_vector(g_num_input_bits -1 downto 0);
 signal reg_result: std_logic_vector(g_num_output_bits -1 downto 0);
 signal reg_init: std_logic_vector(g_num_output_bits -1 downto 0);
 signal reg_control_a: std_logic_vector(2 downto 0);
 signal reg_sv: std_logic;
 signal reg_vpl: std_logic;
 signal reg_dv: std_logic;
 signal reg_sv_delay: std_logic;
 type states
   is(state_init, state_no_input_yet, state_input, state_no_input);
 signal present_state: states;
 signal next_State: states;

begin
 reg_init <= (others => '0');
 process (g_clk, g_reset)
 begin
  if g_reset = '1' then
   present_state <= state_init;
   reg_a <= (others => '0');
   reg_result <= (others => '0');
   reg_control_a <= (others => '0');
   reg_sv_delay <=  '0';
   reg_sv <= '1';
   reg_vpl <= '0';
   reg_dv <= '0';
  elsif rising_edge(g_clk) then
   present_state <= next_state;
   reg_a <= g_a;
   reg_sv <= g_control_a(2);
   reg_vpl <= g_control_a(1);
   reg_dv <= g_control_a(0);
   reg_control_a <= g_control_a;
   reg_sv_delay <= reg_sv;
   case reg_sv is
   when '1' =>
    if reg_sv_delay = '0' then
     if reg_vpl = '1' and reg_dv = '1' then
      reg_result <= reg_init + reg_a;
     end if;
    else
     if reg_vpl = '1' and reg_dv = '1' then
      reg_result <= reg_result + reg_a;
```

```
     end if;
    end if;
   when others =>
   end case;
  end if;
 end process;
 process (reg_sv)
 begin
  case reg_sv is
  when '1' =>
   g_result <= (others => '0');
  when others =>
   g_result <= reg_result;
  end case;
 end process;
 process(present_state, reg_sv)
 begin
  case present_state is
   when state_init =>
g_control_result <= (others => '0');
next_state <= state_no_input_yet;
   when state_no_input_yet =>
    g_control_result <= (others => '0');
if reg_sv = '0' then
     next_state <= state_no_input_yet;
    else
 next_state <= state_input;
end if;
   when state_input =>
if reg_sv = '1' then
     next_state <= state_input;
 g_control_result <= (others => '0');
else
 next_state <= state_no_input;
 g_control_result <= (others => '1');
    end if;
   when state_no_input =>
    if reg_sv = '1' then
     next_state <= state_input;
     g_control_result <= (others => '0');
else
 next_state <= state_no_input;
 g_control_result <= (others => '1');
end if;
```

```
   end case;
 end process;
end;


-------------------------------------------------------------------
-- champion_stream_max_#g_num_input_bits#_
--   #g_num_r_bits#_#g_num_c_bits#_#g_num_col#.vhd
-- by: senthil natarajan
--
-- Description: Calculates the first maximum value of
-- a stream and holds its output until
-- a new stream is applied at the input.
-------------------------------------------------------------------

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

entity champion_stream_max_#g_num_input_bits#_
       #g_num_r_bits#_#g_num_c_bits#_#g_num_col# is
 generic (g_num_input_bits: positive:= #g_num_input_bits#;
          g_num_r_bits: positive:= #g_num_r_bits#;
          g_num_c_bits: positive:= #g_num_c_bits#;
          g_num_col: positive:= #g_num_col#;
          g_num_control_bits: positive:= 3);
 port (g_a: in std_logic_vector(g_num_input_bits -1 downto 0);
       g_control_a: in
         std_logic_vector(g_num_control_bits -1 downto 0);
       g_reset: in std_logic;
       g_clk: in std_logic;
       g_result: out std_logic_vector(g_num_input_bits -1 downto 0);
       g_result_row_index: out
         std_logic_vector(g_num_r_bits -1 downto 0);
       g_result_column_index: out
         std_logic_vector(g_num_c_bits -1 downto 0);
       g_control_result: out
         std_logic_vector(g_num_control_bits -1 downto 0));
end champion_stream_max_#g_num_input_bits#_
    #g_num_r_bits#_#g_num_c_bits#_#g_num_col#;

architecture behave of champion_stream_max_#g_num_input_bits#_
                       #g_num_r_bits#_#g_num_c_bits#_#g_num_col# is
```

```
component champion_row_column is
 generic (g_num_input_bits: positive;
    g_num_r_bits: positive;
          g_num_c_bits: integer;
          g_num_col: integer;
          g_num_control_bits: positive);
 port (g_a: in std_logic_vector(g_num_input_bits -1 downto 0);
       g_control_a: in
         std_logic_vector(g_num_control_bits -1 downto 0);
       g_reset: in std_logic;
       g_clk: in std_logic;
       g_result: out std_logic_vector(g_num_input_bits -1 downto 0);
       g_result_row_index: out
         std_logic_vector(g_num_r_bits -1 downto 0);
       g_result_column_index: out
         std_logic_vector(g_num_c_bits -1 downto 0);
       g_control_result: out
         std_logic_vector(g_num_control_bits -1 downto 0));
end component;
 signal reg_result:
  std_logic_vector(g_num_input_bits -1 downto 0);
 signal reg_result_row:
  std_logic_vector(g_num_r_bits -1 downto 0);
 signal reg_result_column:
  std_logic_vector(g_num_c_bits -1 downto 0);
 signal reg_prior:
  std_logic_vector(g_num_input_bits -1 downto 0);
 signal reg_prior_row:
  std_logic_vector(g_num_r_bits -1 downto 0);
 signal reg_prior_column:
  std_logic_vector(g_num_c_bits -1 downto 0);
 signal sig_result:
  std_logic_vector(g_num_input_bits -1 downto 0);
 signal sig_result_row:
  std_logic_vector(g_num_r_bits -1 downto 0);
 signal sig_result_column:
  std_logic_vector(g_num_c_bits -1 downto 0);
 signal sig_control_result:
  std_logic_vector(g_num_control_bits -1 downto 0);
 signal reg_sv: std_logic;
 signal reg_sv_delay: std_logic;
 signal reg_vpl: std_logic;
 signal reg_vpl_delay: std_logic;
 signal reg_dv: std_logic;
```

```
 signal reg_dv_delay: std_logic;
 type states
  is(state_init, state_no_input_yet, state_input, state_no_input);
 signal present_state: states;
 signal next_state: states;
begin
u1_row_column_count: champion_row_column
     generic map (g_num_input_bits => g_num_input_bits,
      g_num_r_bits => g_num_r_bits, g_num_c_bits => g_num_c_bits,
      g_num_col => g_num_col, g_num_control_bits =>
      g_num_control_bits)
     port map(g_a => g_a, g_control_a => g_control_a,
      g_reset => g_reset, g_clk => g_clk, g_result => sig_result,
      g_result_row_index => sig_result_row,
      g_result_column_index => sig_result_column,
      g_control_result => sig_control_result);
 process(present_state)
 begin
  case present_state is
   when state_init =>
    next_state <= state_no_input_yet;
    g_control_result <= (others => '0');
   when state_no_input_yet =>
    if reg_sv = '0' then
 next_state <= state_no_input_yet;
else
 next_state <= state_input;
    end if;
    g_control_result <= (others => '0');
   when state_input =>
    if reg_sv = '0' then
 next_state <= state_no_input;
else
 next_state <= state_input;
    end if;
    g_control_result <= (others => '0');
   when state_no_input =>
    if reg_sv = '1' then
 next_state <= state_input;
    g_control_result <= (others => '0');
else
 next_state <= state_no_input;
     g_control_result <= (others => '1');
    end if;
```

```
  end case;
end process;
process (g_clk, g_reset, g_control_a, reg_sv, reg_sv_delay)
 variable reg_max: std_logic_vector(g_num_input_bits -1 downto 0);
 variable reg_max_row: std_logic_vector(g_num_r_bits -1 downto 0);
 variable reg_max_column:
  std_logic_vector(g_num_c_bits -1 downto 0);
begin
 if reg_sv = '0' and reg_sv_delay = '0' then
  g_result <= reg_max;
  g_result_row_index <= reg_max_row;
  g_result_column_index <= reg_max_column;
 else
  g_result <= (others => '0');
  g_result_row_index <= (others => '0');
  g_result_column_index <= (others => '0');
 end if;
 if g_reset = '1' then
  present_state <= state_init;
  reg_sv <= '0';
  reg_sv_delay <= '0';
  reg_vpl <= '0';
  reg_vpl_delay <= '0';
  reg_dv <= '0';
  reg_dv_delay <= '0';
  reg_result <= (others => '0');
  reg_result_row <= (others => '0');
  reg_result_column <= (others => '0');
  reg_max := (others => '0');
  reg_max_column := (others => '0');
  reg_max_row := (others => '0');
  reg_prior <= (others => '0');
  reg_prior_row <= (others => '0');
  reg_prior_column <= (others => '0');
 elsif rising_edge(g_clk) then
  present_state <= next_state;
  reg_sv <= g_control_a(2);
  reg_sv_delay <= reg_sv;
  reg_vpl <= g_control_a(1);
  reg_vpl_delay <= reg_vpl;
  reg_dv <= g_control_a(0);
  reg_dv_delay <= reg_dv;
  if g_control_a(2) ='1' and reg_sv_delay = '0' then
   reg_result <= (others => '0');
```

```
    reg_result_row <= (others => '0');
    reg_result_column <= (others => '0');
    reg_max := (others => '0');
    reg_max_column := (others => '0');
    reg_max_row := (others => '0');
    reg_prior <= (others => '0');
    reg_prior_row <= (others => '0');
    reg_prior_column <= (others => '0');
   else
    reg_result <= sig_result;
    reg_result_row <= sig_result_row;
    reg_result_column <= sig_result_column;
    if (reg_result > reg_prior) and (reg_vpl_delay ='1')
        and (reg_dv_delay ='1') then
     reg_max := reg_result;
 reg_max_row := reg_result_row;
 reg_max_column := reg_result_column;
    else
     reg_max := reg_prior;
 reg_max_row := reg_prior_row;
 reg_max_column := reg_prior_column;
    end if;
    reg_prior <= reg_max;
    reg_prior_row <= reg_max_row;
    reg_prior_column <= reg_max_column;
   end if;
  end if;
 end process;
end;


-------------------------------------------------------------------
-- champion_kadd_#g_num_input_bits#_#g_constant#.vhd
-- by: senthil natarajan
--
-- Description:
-- Hardware equivalent for glyph kadd
-- with one input of #g_num_input_bits# bits.
-- One input is a constant that is determined at compile-time.
-- The result is #g_num_input_bits# +1 bits. Because of the use
-- of a compile-time constant, #g_constant#, this
-- hardware equivalent
-- glyph must be synthesized at compile-time.
-------------------------------------------------------------------
```

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

entity champion_kadd_#g_num_input_bits#_#g_constant# is
 generic (g_num_input_bits: positive:= #g_num_input_bits#;
          g_constant: positive:= #g_constant#;
  g_num_control_bits: positive:= 3);
 port (g_a: in std_logic_vector(g_num_input_bits -1 downto 0);
       g_control_a: in
         std_logic_vector(g_num_control_bits -1 downto 0);
       g_clk, g_reset: in std_logic;
       g_result: out std_logic_vector(g_num_input_bits downto 0);
       g_control_result: out
         std_logic_vector(g_num_control_bits -1 downto 0));
end champion_kadd_#g_num_input_bits#_#g_constant#;

architecture behave of champion_kadd_#g_num_input_bits#_#g_constant#
 is
 component kadd is
  generic (num_bits: positive);
  port (a,b: in std_logic_vector (g_num_input_bits -1 downto 0);
        result: out std_logic_vector(g_num_input_bits downto 0) );
 end component;
 signal reg_a: std_logic_vector(g_num_input_bits -1 downto 0);
 signal reg_control_a:
  std_logic_vector(g_num_control_bits -1 downto 0);
 signal sig_b: std_logic_vector(g_num_input_bits -1 downto 0);
 signal sig_b_init: std_logic_vector(g_num_input_bits -1 downto 0);
begin
 sig_b_init <= (others => '0');
 u1_kadd: kadd generic map(num_bits => g_num_input_bits)
          port map(a => reg_a, b => sig_b, result=> g_result);
 process (g_clk, g_reset,sig_b_init)
 begin
  if g_reset = '1' then
   reg_a <= (others => '0');
   reg_control_a <= (others => '0');
  elsif rising_edge(g_clk) then
   reg_a <= g_a;
   reg_control_a <= g_control_a;
  end if;
```

```
   if reg_control_a = "111" then
     sig_b <= sig_b_init + g_constant;
    else
     sig_b <= (others => '0');
    end if;
  end process;

  g_control_result <= reg_control_a;
end;
```

VITA

Senthil Natarajan was born in Coimbatore, India on October 4, 1974, and moved to North America in 1977. After beoming a US citizen in 1989, he finished his high school education at Cookeville High School in Cookeville Tennessee and entered the University of Tennessee for his undergraduate studies in Electrical Engineering. Upon completion of his Bachelor of Science degree in EE in 1997, he then entered the graduate program also at UT. While there, he worked for the university as a graduate research assistant. In August of 1999, he earned a Master of Science degree in Electrical Engineering.