A Design Methodology for Creating Programmable Logic-based Real-time Image Processing Hardware

By
Thomas Hudson Drayer

Dissertation submitted to the Faculty of the Bradley Department of Electrical Engineering of Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering

Dr. Joseph G. Tront, Chairman
Dr. Richard W. Conners, Co-chairman
Dr. D. Earl Kline
Dr. A. Lynn Abbott
Dr. Charles E. Nunnally

January 24, 1997
Blacksburg, Virginia

Keywords: real-time image processing, FPGA-based computing, design methodology, reconfigurable computing, MORRPH, TRAVERSE

A Design Methodology for Creating Programmable Logic-based Real-time Image Processing Hardware

By

Thomas Hudson Drayer

Dr. Joseph G. Tront, Chairman

Electrical Engineering

(ABSTRACT)

A new design methodology that produces hardware solutions for performing real-time image processing is presented here. This design methodology provides significant advantages over traditional hardware design approaches by translating real-time image processing tasks into the gate-level resources of programmable logic-based hardware architectures. The use of programmable logic allows high-performance solutions to be realized with very efficient utilization of available logic and interconnection resources. These implementations provide comparable performance at a lower cost than other available programmable logic-based hardware architectures.

This new design methodology is based on two components: a programmable logic-based destination hardware architecture and a suite of development system software. The destination hardware architecture is a Custom Computing Machine (CCM) that contains multiple Field Programmable Gate Array (FPGA) chips. FPGA chips provide gate-level programmability for the hardware architecture. Sophisticated software development tools, called the TRAVERSE

development system software, are created to overcome the significant amount of time and expertise required to manually utilize this gate-level programmability. The new hardware architecture and development system software combine to establish a unique design methodology.

There are several distinct contributions provided by this dissertation. The new flexible MORRPH hardware architecture provides a more efficient solution for creating real-time image processing computing machines than current commercial hardware architectures. The TRAVERSE development system software is the first integrated development system specifically for creating real-time image processing designs with multiple FPGA-based CCMs. New standards and design conventions are defined specifically for creating solutions to low-level image processing tasks, using the MORRPH architecture for verification. The circuit partitioning and global routing programs of the TRAVERSE development system software enable automated translation of image processing designs into the resources of multiple FPGA chips in the hardware architecture. In a broad sense, the individual contributions of this dissertation combine to create a new design methodology that will change the way hardware solutions are created for real-time image processing in the future.

Acknowledgments

I would like to thank my committee chairman Dr. Joseph Tront in for his mentoring and intellectual feedback throughout the years. Similarly, my committee cochairman Dr. Richard Conners has contributed much to my education in the course of several research projects, for which he receives my most sincere appreciation. A special thanks to all the other committee members, Dr. Earl Kline, Dr. Charles Nunnally, and Dr. Lynn Abbott who devoted their time to the review of my work.

This research could not have been accomplished without the monetary and technical support of individuals such as Phil Araman of the United States Forest Service and Dr. Shiu Cheung of the Federal Aviation Administration. Thanks also to the individuals of Aristokraft Inc., such as Bill Fortney, Alan Smock, Tom Royce, and Greg Kroedel. Finally, thanks to the Bradley-Via family for their generous donations to the Electrical Engineering department of Virginia Tech. The academic freedom provided by the Bradley Fellowship is greatly appreciated.

Several students in the Spatial Data Analysis Lab of Virginia Tech have contributed to this work, the most important of which are Will King, Chase Wolfinger, Paul Lacasse, and members of the MORRPH design team. These individuals receive my gratitude for their efforts

on this project. Additional thanks to Xiang-Yu Xiao, Chong T. Ng, Dongping Zhu, Yuhua Cui, Qiang Lu, and all other students of the SDA Lab. A special thanks to Bob Lineberry for his wisdom and advice throughout the years.

Finally, I would like to thank my parents and sisters for their support and encouragement. I love them all very much and dedicate this work to them.

Table of Contents

Chapter 1: Introduction **1**

1.1 Overview/Motivation 1

1.2 Common Low-level Image Processing Operations 4

1.2.1 Transformations 5

1.2.2 Extractions 11

1.2.3 Image Processing Designs 13

1.3 Research Objectives 16

Chapter 2: Literature Review 20

2.1 Hardware Architectures 20

2.1.1 Custom ASIC Boards 21

2.1.2 Embedded Processors 24

2.1.3 FPGA-based Custom Computing Machines 26

2.1.3.1 The Virtual Computer 29

2.1.3.2 CHAMP 31

2.1.3.3 Splash 2 32

2.1.3.4 Spectrum Reconfigurable Computing Platform 34

2.1.3.5 Limitations of Current Hardware Architectures 35

2.2 Software Development Systems 36

2.3 Summary 40

Chapter 3: FPGA-based Destination Architecture 44

3.1 Incompletely Specified Architectures 46

3.2 MORRPH Design Philosophy 50

3.3 MORRPH-ISA Design Implementation 52

3.4 Hardware Design Summary 58

Chapter 4: Development System Software 60

4.1 Software Development System Design Philosophy 60

4.2 Design Entry 64

4.2.1 Image Processing Module Symbols 67

4.2.2 SUIT Bus Format 69

4.2.3 ZEE Bus Format 75

4.2.4 Variable and Constant Operand Values 76

4.2.5 Image Processing Module Schematics 79

4.2.6 Image Processing Designs 88

4.3 Verification 92

4.3.1 The PIXMAN program 95

4.3.2 The MDBUG program 103

4.4 Translation 112

4.4.1 Network Partitioning 113

4.4.1.1 State Search Algorithm 118

4.4.1.2 Heuristic Cost Function 140

4.4.2 Global Routing 145

4.4.3 Translation Implementation 148

Chapter 5: Experimental Results **161**

5.1 Justification 161

5.1.1 Evaluation Library Image Processing Modules 163

5.1.2 Single Module Image Processing Designs 166

5.1.3 Morphological Image Processing Design 171

5.1.4 Gaussian Pyramid Image Processing Design 174

5.1.5 Multiple Input Statistical Image Processing Design 179

5.2 Analysis of the Design Methodology 189

5.2.1 Efficiency of the Software Development System 191

5.2.2 Efficiency of the Hardware Architecture 197

5.2.3 Performance of the Basis Image Processing Designs 207

Chapter 6: Conclusions **212**

Appendix I: Format Definition for Architecture Configuration Files 217

Appendix II: Modules of the Evaluation Libraries 224

References 238

Vita 244

List of Figures

Figure 1.2-1. Classification of real-time image processing operations 5

Figure 1.2-2. Thresholding of an image 7

Figure 1.2-3. Common 3x3 window operator coefficients 9

Figure 1.2-4. Images processed using common 3x3 window operators 10

Figure 1.2-5. Histogram of pixel intensity values 12

Figure 1.2-6. Optical character recognition data flow graph for Example 1.2.1 14

Figure 1.2-7. Defect detection data flow graph for Example 1.2.2 15

Figure 2.1-1. Block diagram of the DT2856 board 22

Figure 2.1-2. Block diagram of a modular stage of the Aspex PIPE 23

Figure 2.1-3. Xilinx 4000 series FPGA structure 27

Figure 2.1-4. Virtual Computer Architecture 30

Figure 2.1-5. CHAMP system block diagram and PE architecture 31

Figure 2.1-6. Splash 2 processor array card architecture 33

Figure 2.1-7. Spectrum G800 RIC and XMOD block diagram 35

Figure 3.2-1. MORRPH processing element architecture 50

Figure 3.3-1. Block diagram of the MORRPH-ISA board 53

Figure 3.3-2. MORRPH-ISA adapter card 57

Figure 4.2-1. Symbol for the **OFFSET_A** image processing module 69

Figure 4.2-2. SUIT bus signal locations and bus commands 72

Figure 4.2-3. Sequence of SUIT bus values for example gray scale image 75

Figure 4.2-4. ZEE bus signal locations and bus commands 77

Figure 4.2-5. Schematic for module **OFFSET_A** 83

Figure 4.2-6. Schematic for module **XOFFSET_A** 84

Figure 4.2-7. Schematic for module **LUT_A** 86

Figure 4.2-8. Schematic for module **XLUT_A** 87

Figure 4.2-9. Image processing design with **OFFSET_A** image processing module 91

Figure 4.3-1. Input files, output files, and information flow of the PIXMAN program 96

Figure 4.3-2. Program options for the PIXMAN program 99

Figure 4.3-3. ASCII data files used by the PIXMAN program in Example 4.3.1 100

Figure 4.3-4. Command file created by Example 4.3.1 101

Figure 4.3-5. Simulation result waveforms for Example 4.3.1 102

Figure 4.3-6. Input files, output files, and information flow of the MDBUG program 103

Figure 4.3-7. Schematic for in-circuit emulation of the **OFFSET_A** module 109

Figure 4.3-8. Schematic for supplying and collecting data of in-circuit emulation 111

Figure 4.4-1. Average cutset size for 20 trials of bipartitioning 125

Figure 4.4-2. Average cutset size for 20 trials of 3-way partitioning 126

Figure 4.4-3. Average cutset size for 20 trials of 4-way partitioning 127

Figure 4.4-4. Average processing time for 20 trials of bipartitioning 128

Figure 4.4-5. Average processing time for 20 trials of 3-way partitioning 129

Figure 4.4-6. Average processing time for 20 trials of 3-way partitioning 130

Figure 4.4-7. Percent reduction in cutset size of the multi-step algorithm 133

Figure 4.4-8. Percent increase in processing time of the multi-step algorithm 134

Figure 4.4-9. Average cutset size for 20 trials of the simulated annealing algorithm 137

Figure 4.4-10. Standard deviation of cutset size for 20 trials of simulated annealing 138

Figure 4.4-11. Program flow and file extensions of the MTRANS program 149

Figure 5.1-1. Schematic for 3x3 Laplacian window operator image processing design 167

Figure 5.1-2. Schematic for 3x3 average window operator image processing design 168

Figure 5.1-3. Result images of single-module image processing designs 170

Figure 5.1-4. Schematic for morphological image processing design 172

Figure 5.1-5. Result image from morphological image processing design 174

Figure 5.1-6. Representation of a 4-level image pyramid 175

Figure 5.1-7. Schematic for 4-level Gaussian pyramid image processing design 176

Figure 5.1-8. Result images from the Gaussian pyramid image processing design 178

Figure 5.1-9. Sheet one of statistical image processing design 180

Figure 5.1-10. Sheet two of statistical image processing design 181

Figure 5.1-11. Sheet three of statistical image processing design 182

Figure 5.1-12. Input images for the statistical image processing design 186

Figure 5.1-13. Output images of the statistical image processing design 187

Figure 5.1-14. Output histograms of the statistical image processing design 188

Figure 5.2-1. Total FPGA resource requirements for basis designs 192

Figure 5.2-2. Overhead circuitry of the development system software for basis designs 195

Figure 5.2-3. Relationship between the number of SUIT bus channels and overhead 196

Figure 5.2-4. CLB resource utilization for basis image processing designs 200

Figure 5.2-5. FF resource utilization for basis image processing designs 201

Figure 5.2-6. IOB resource utilization for basis image processing designs 202

Figure 5.2-7. Percent utilization of MORRPH interconnection resources 205

Figure 5.2-8. Effect of FPGA speed grade on maximum operating clock frequency 209

Figure A1.1-1. Section definitions of the ACF file format 219

Figure A.1.1-2. Example file in ACF file format 224

List of Tables

Table 3.3-1. Processing element FPGA type considerations 54

Table 3.3-2. MORRPH-ISA I/O port locations 56

Table 4.1-1. Programs of the TRAVERSE development system software 63

Table 4.3-1. Command options of the MDBUG program 105

Table 4.4-1. Individual cost constraints used in the total cost function 141

Table 5.1-1. FPGA resource requirements of modules in evaluation libraries 164

Table 5.1-2. Performance of modules in evaluation libraries 165

Table 5.2-1. Compilation results for six image processing designs 190

Chapter 1. Introduction

1.1 Overview/Motivation

In the search for new applications of computer technology, the computer is utilized to perform tasks traditionally accomplished by humans. However, it is difficult to replace the sophisticated sensory input and processing of the human body with current computers. Vision is the most challenging of the five senses to emulate, due to the vast amount of data involved.

Specialized imaging sensors allow machine vision hardware to exceed the capabilities of human vision. Examples of this are infrared and X-ray imaging systems. Digital hardware can be used to collect, enhance, and display information from these specialized imaging sensors. Manual inspection systems use human intervention to interpret the obtained image data. In contrast, automated inspection systems use digital hardware to interpret the image data. These inspection systems are typically used to perform tasks such as object alignment, classification, and defect detection [1, 2, 3].

The most important advantages of automated inspection over manual inspection are speed and consistency. However, the large data throughput of real-time automated inspection systems requires the use of expensive, high performance computing hardware. Computer vision systems are, therefore, extremely expensive. The goal of this new design methodology is to create cost effective computing hardware for real-time image processing.

The input device for most current computer vision systems is the Charge-Coupled Device (CCD) array camera. Photons of light that strike the sensing elements of a CCD camera are converted to electrical charge which is stored by the sensing element. Over time, a charge accumulates in the sensing element that is proportional to the amount of light incident on the element. The amount of time for which the sensing elements are allowed to accumulate charge is called the *integration time*. At the end of the integration time all charge is removed from the sensing elements and used to generate an analog electrical signal that is proportional to the intensity of incident light on each sensing element. This analog signal is converted to a sequence of digital values that can be processed by a digital computer.

The CCD sensing devices are typically arranged in a one-dimensional linear (line scan cameras) or two-dimensional square (array cameras) sensing array, however some circular and rectangular array cameras have been commercially produced. A one-dimensional slice of image data, such as the data generated by a line scan camera, is called a *line*. Values from successive one-dimensional scans of a line scan camera may be appended together to create a two-dimensional array of values that represent the image, called a *frame*. Although the number

of sensing elements contained in CCD cameras varies widely, cameras with more than 250,000 elements are common.

Each point in the image plane is called a pixel element, ideally representing a unique region of the scene plane. For gray scale cameras, a single value is used to define the amount of light incident on the corresponding element in the sensing array over the integration period. This value is typically represented with either a 7-bit or 8-bit value. An 8-bit value provides 256 discrete values to represent the light intensity of each pixel location. Most color cameras use three values to represent each pixel location, of which each value corresponds to the amount of red, green, or blue light that is incident on a particular sensing element.

A frame rate of at least 30 frames/sec is required for video applications to eliminate any human perception of the subsampling of continuously moving objects (excluding aliasing effects). Subsampling appears as a "flickering" effect to human perception. Therefore, many common CCD array cameras operate at this rate, using an integration time of 3.33 ms/frame. A common 512X512 gray scale CCD camera operating at 30 frames/sec produces 7.5 Mbytes/sec of image data, when each pixel is converted to an 8-bit digital value.

A system that receives, processes, and outputs result data at or above the input data rate required for generating accurate results is defined as a *real-time* image processing system. Therefore, real-time image processing systems typically require a throughput of approximately 7.5 Mbytes/sec. Although this data rate is typical, many real-time vision applications may require significantly more or less throughput. A hardware processing system can only be defined as real-time within the context of a particular image processing task.

The processes involved in image processing can be divided into two classes: *high-level* and *low-level* operations. The low-level operations are defined as operations involving individual pixel element values. Processes that operate on the properties that have been extracted from the image are defined as high-level processes. Low-level processes are characterized by large amounts of input data and simple calculations. In contrast, high-level processes are characteristically more complicated calculations that are performed on a smaller amount of input data. The concentration of this dissertation is on hardware solutions for the low-level processes.

The goal of this research is to provide a new method for producing hardware and software for real-time image processing solutions. A design methodology is defined that is applicable to a wide range of low-level image processing tasks. This new design methodology creates efficient solutions for real-time image processing hardware.

1.2 Common Low-level Image Processing Operations

The purpose of this section is to illustrate the type of low-level operations that are typically performed in real-time image processing systems. Figure 1.2-1 illustrates the classifications used to define the low-level operations of the upcoming section. These operations are explained in the following subsections and used as examples in later chapters to validate the operation of the development system. This section is not intended to be a comprehensive list or survey, but the wide range of operators presented in this section does provide a representative

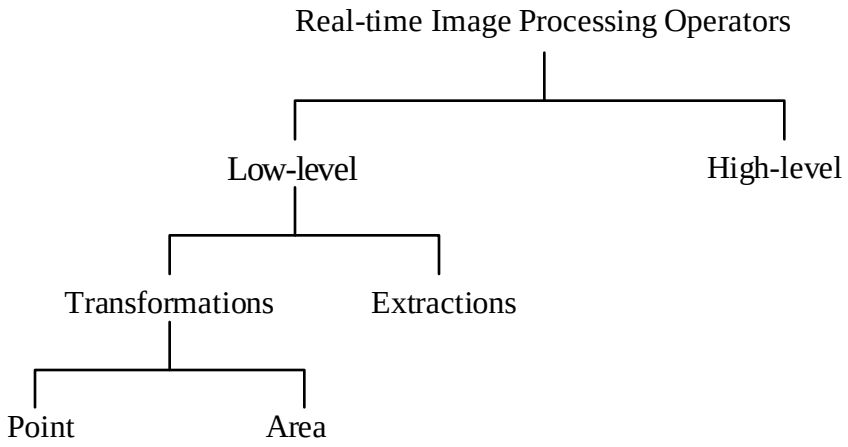


Figure 1.2-1. Classification of real-time image processing operations.

sample of common low-level image processing functions. For a more comprehensive list of image processing operators and techniques, the reader is referred to a text on image processing, such as [4], [5], or [6].

1.2.1 Transformations

Transformations are defined as operations for which both the input and output of the process is an image or sequence of images. Two types of transformation operators are pixel *point* and *area* (i.e., group, window, or mask) operators. Area operators use more than one pixel value of the input image in the calculation of a single output pixel value, while pixel point operators use a single input pixel to calculate an output pixel value.

Thresholding is a pixel point operation that performs a comparison on each pixel value to determine if the intensity value is above or below an input threshold variable T_{val}. The thresholding operation can be implemented in several ways. An output image J(r,c) can be created from the input image I(r,c) in which each pixel location is represented by a single bit value, called a *bit-mapped* image, in accordance with the following equation:

$$J(r,c) = \begin{cases} 1 & \text{iff } I(r,c) > T_{val} \\ 0 & \text{otherwise} \end{cases} \qquad (1\text{-}1)$$

A second thresholding method is to use Run-Length Encoding (RLE) for the resultant binary image, such that the output results are only a sequence of values that represent the locations of transitions from 0 to 1 or 1 to 0. Alternatively, the original image can be preserved by replacing all values below (or above) the threshold with a fixed value, such as zero. An example of this method of thresholding is illustrated in Figure 1.2-2. The original gray scale image is presented with the output image after all pixel values below a threshold of 125 are transformed to the value zero.

Scaling and *offset* operations are used to brighten or darken an image. The scaling operation multiplies each pixel intensity value by a fixed constant. Similarly, the offset operation adds a fixed constant to a each pixel intensity value. These operations are sometimes called histogram modification operations.

Another type of pixel point image transform operation is the *Look-Up-Table* or LUT operation. For each input pixel intensity value, a corresponding output value is defined and

(a) (b)

Figure 1.2-2. Thresholding of an image (a) original image (b) thresholded image.

stored in a table. Each input pixel intensity value is used to index into a table to retrieve the transformed value for the same pixel location in the output image. For 8-bit gray scale images, a table of only 256 values is required. However for 24-bit color images, a table of over 24 million values is required. The LUT is a powerful operation that can be used to implement a wide variety of pixel point operations (e.g., thresholding, scaling, and offset) when the output value is a function only of the input pixel intensity.

The output of some pixel point operations is a function of both the input intensity and the pixel location. *Light compensation* (i.e., shade compensation) is used to correct for non-uniform

lighting conditions. An *n*th order quadratic is commonly used, for which *n* quadratic coefficients are required for each pixel location.

The second type of transformation operations are pixel area operations. These operations use several pixels in the input image to calculate each output pixel value. Window operators use the neighboring pixels in an *m*x*n* window to calculate a single output pixel value. The *convolution* window operator calculates a weighted average of all the input pixel intensity values in the *m*x*n* window. Each of the input pixel intensity values are multiplied by a constant coefficient for their particular location in the window, then a sum of all the weighted values is calculated to produce the output pixel value. In some cases, the final sum is divided by a fixed value such that the sum of all the gains is unity.

Some common convolution coefficient values for a 3x3 window are shown in Figure 1.2-3 [4]. The average and Gaussian area operators remove image noise while the Sobel and Laplacian operations are useful for detecting edges in an image. The Gaussian window operator coefficients are an approximation to the following equation:

$$k(r,c) = \frac{1}{2\pi} e^{\frac{1}{2}\left(\frac{r^2}{\sigma^2}+\frac{c^2}{\sigma^2}\right)} \qquad \sigma = .5 \qquad (1\text{-}2)$$

Resultant images, using the input image of Figure 1.2-2, for each of the three window operators are shown in Figure 1.2-4. Since *m***n* multiplication operations and *m***n*-1 addition operations are required to calculate each output pixel value, a large number of operations must be performed even for small window operations. With the typical 7.5 Mbyte/sec input data rate, a 5x5 window operator requires a sustained throughput of over 375 Million Instructions Per Second (MIPS), well above the performance of typical low-cost personal computers.

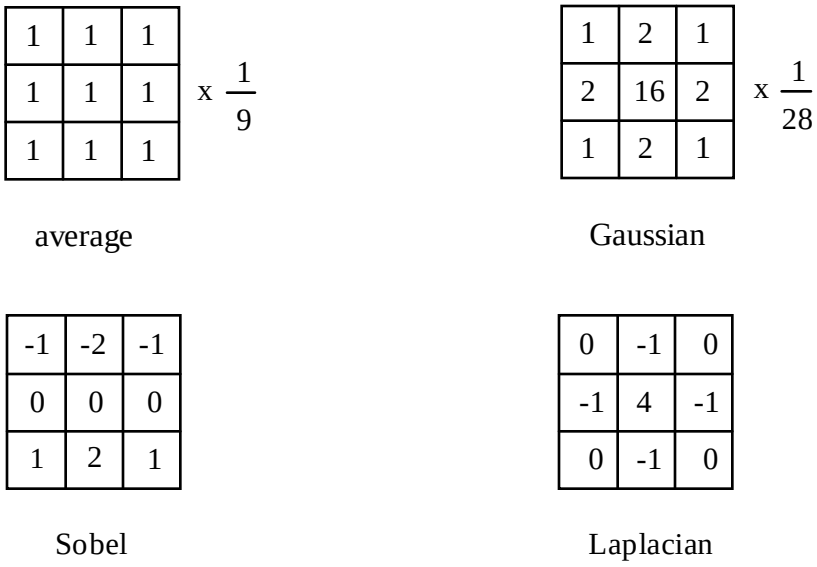


Figure 1.2-3. Common 3x3 window operator coefficients [4].

Morphological operations are performed on groups of pixels in bit-mapped images. The *erosion* operation is used to reduce the area of objects in an image. Conversely, *dilation* is used to expand the area of objects. The erosion (dilation) operation may be implemented with the logical AND (logical OR) function of all bit values within the defined window.

average

Gaussian

Sobel

Laplacian

Figure 1.2-4. Images processed using common 3x3 window operators.

A final class of image area transformation operations are *geometric* operations. Functions such as zooming, panning, warping, and rotation belong to this class of operations. These functions are particularly difficult to implement on pipelined computing architectures because of the sequence in which the input operands must be available and the order in which the output values must be calculated.

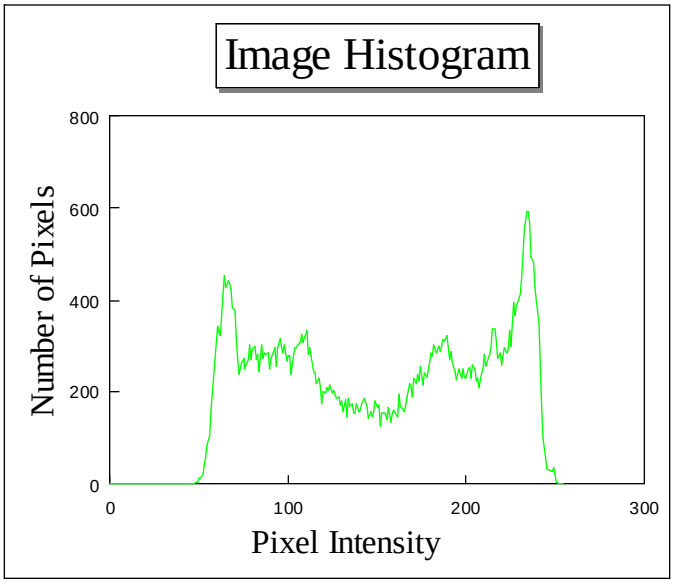
1.2.2 Extractions

Extraction operators calculate and output properties of the input image, without generating an output image. For example, a *histogram* counts the frequency of a particular pixel intensity value in an input image. The output of a histogram operation is a series of count values, one for each intensity value. An input gray scale image and a graph of the 256-value histogram is presented in Figure 1.2-5.

A second type of extraction processes are called *blob processing* operations. These operations determine the properties of subsections of an image. Typical properties to calculate are the area, center-of-mass, or average intensity of each defined subsection. Blob processing is common in the location of defects or identification of objects in an image scene.

Template matching is another common extraction operation. A fixed pattern or template is compared to regions of the input image, and a value corresponding to the "likeness-of-fit" is calculated. The template may be translated or rotated when trying to identify an object in the image.

(a)



(b)

Figure 1.2-5. Histogram of pixel intensity values (a) input image (b) histogram.

Extraction operators typically reduce the amount of data required for further processing. For example, the one-dimensional histogram of an 8-bit gray scale image represents only 256 output values. However, the three-dimensional histogram created from the three byte values of a 24-bit color image requires over 24 million (256x256x256) count values; significantly more data than the 750 kilobytes required to store a typical 512X512 pixel element, 24-bit color image.

1.2.3 Image Processing Designs

Sophisticated image processing systems seldom require only a single image processing operator. Typically, several low-level operations are performed on an input image before the high-level processing tasks are performed. A unique solution does not exist for a particular image processing task. Therefore, an *image processing design* is defined as combination of low-level image processing operations that are intended to provide one solution for a specific image processing task.

A data flow graph (i.e., dependency graph) is used to illustrate the combination of low-level operators and associated data dependencies involved in complicated image processing designs. Vertices of the data flow graph represent operations and edges represent data dependencies or transfers. The following examples illustrate how several operators may be combined to process an input image and the resultant data flow graphs for these image processing designs.

Example 1.2.1 Optical Character Recognition (OCR): An image processing design is desired to identify typed letters within a gray scale image. One solution follows:

1. First the input image is shade compensated, to provide a uniform lighting response.
2. Next, a threshold operation is performed to create a binary image.
3. Finally, the binary image is processed by 26 individual template matching operators, each of which calculates the likeness of fit for each individual letter.

The corresponding data flow graph for this image processing design is shown in Figure 1.2-6. After this low-level processing, high-level processes combine the data from each of the template matching operators to determine if a letter is present, and if so, its identity.

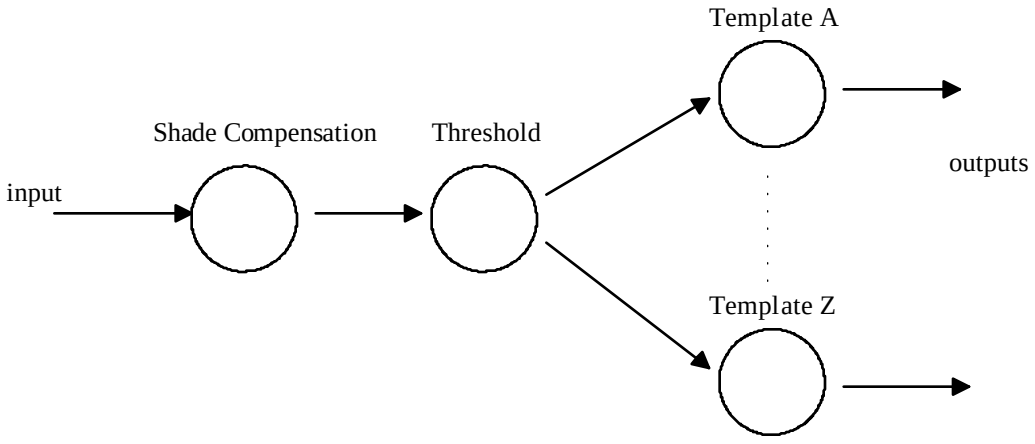


Figure 1.2-6. Optical character recognition data flow graph for Example 1.2.1.

Example 1.2.2 Defect Identification: An image processing design is desired to identify areas of defect within a single object. The input 24-bit full color image contains only the object and a saturated white background. Areas of defect are characterized by color values that do not appear in the object and always represent less than 50% of the object area. However, the specific color values that represent the object and any possible defect are not known. One solution follows:

1. First the input image is shade compensated, to provide a uniform lighting response.
2. Next, the image is smoothed using a 3x3 average window operator.
3. Then a threshold operation is performed to separate the object from its background.
4. Next, the 24 million full-color values are mapped to a pallet of 2,000 colors, using a LUT operation.
5. Finally, a 2,000 element histogram is calculated from only the pixel values of the object in the processed image.

The corresponding data flow graph for this image processing design is shown in Figure 1.2-7. The output histogram from these low-level processes is used by high-level processes to determine which colors represent defect areas. After the high-level processing, additional low-level processing may be performed using information obtained by the high-level analysis of the histogram data.

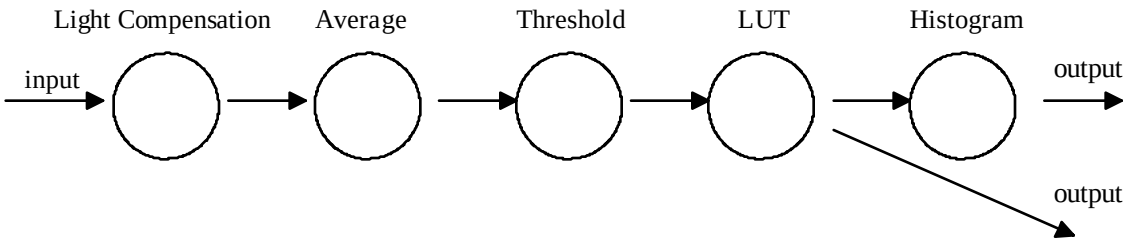


Figure 1.2-7. Defect detection data flow graph for Example 1.2.2.

1.3 Research Objectives

This introduction has described several low-level image processing operations and shown the data flow graph representation of more complicated image processing designs. A new design methodology that creates computational hardware for these types of image processing designs is created for this dissertation. The design methodology consists of two distinct components, an FPGA-based destination hardware architecture and a suite of development system software. Development system software translates the computer representation of an input image processing design into the gate level resources of the FPGA-based destination hardware architecture.

A new class of FPGA-based CCMs is defined for the destination hardware architecture of this design methodology. These computing machines have an incompletely specified architecture, allowing each processing element to contain a variety of chip types. The architecture is specified when the required chips for any particular real-time processing task are assembled on the printed circuit board. A novel FPGA-based CCM, called the MOdular Reprogrammable Real-time Processing Hardware or MORRPH, is designed and fabricated to provide a suitable destination hardware architecture for the verification of this new design methodology.

The gate-level logic resources provided by FPGAs require a new software development system that creates hardware solutions for real-time image processing tasks. A new software development system is created for this dissertation, called TRAVERSE. Several new standards

and design conventions are defined for the TRAVERSE development system software. For example, a new high-bandwidth bus interface standard is defined for the transfer of image and result data. This new bus interface is called the Synchronous Unidirectional Image Transfer or SUIT bus standard. A library of specialized systolic processing units are created using these design conventions to perform a range of low-level image processing tasks. Multiple modules are interconnected to create complex image processing designs that closely resemble data flow graphs. TRAVERSE automatically compiles these image processing designs into the gate-level resources of the destination hardware architecture.

Limitations of currently available commercial software require the suite of development system software to be implemented with a combination of existing software packages and custom software programs. For example, commercial software tools exist to translate an input design into the logic resources of a single FPGA chip. However, the translation of circuits into multiple FPGA chips is not supported by current commercial software because there are no satisfactory software tools for efficiently partitioning logic into the resources of multiple FPGA chips. An enhanced iterative improvement algorithm for partitioning the logic elements of electrical networks into several groups is developed. This new partitioning algorithm defines the multiple FPGA chip boundaries of the image processing design. A new state search algorithm and heuristic cost function allow this new partitioning algorithm to more consistently produce partitions that require fewer interconnections between FPGA chip boundaries than current partitioning algorithms. A new global routing program is also created to assign the interconnection and input/output nets of the partitioned image processing design to the physical

interconnection resources that exist in the destination hardware architecture. The result is a suite of commercially available, custom processing, and format translation programs that combine to provide the desired functionality.

The TRAVERSE software development system and MORRPH hardware architecture combine to provide the first image processing design methodology for FPGA-based CCMs. This new design methodology provides more efficient utilization of hardware resources, therefore creating computing solutions that are more cost-effective. The scalability of the MORRPH architecture and the flexibility of the TRAVERSE software allow the design methodology to provide efficient solutions to a wide range of complexity of image processing designs. Standards and conventions created for the design methodology provide image processing designs that are portable and maintainable. This dissertation will prove the propositions stated above by creating and verifying several representative image processing designs. Finally, the real-time performance of these image processing designs are shown to be adequate and methods are investigated for improving the performance of designs.

This chapter has provided a basis of image processing tasks and defined the significant contributions of this body of work. The next chapter presents the previous research into FPGA based CCMs and software development systems. Chapter 3 defines the new incompletely-specified FPGA-based MORRPH architecture and the MORRPH-ISA board. The software development system is described in Chapter 4. These two components, the hardware architecture and software development system, are used to solve six image processing tasks described in the beginning of Chapter 5. Chapter 5 concludes by using these six designs to

evaluate the performance of this design methodology. Overall conclusions are presented in Chapter 6

Chapter 2. Literature Review

This literature review investigates previous research work involving the two components of the design methodology, real-time hardware architectures for image processing and associated software development systems. Previous solutions and methods are identified. Previous work is used to define the research objectives of this dissertation.

2.1 Hardware Architectures

This section presents a survey of hardware architectures available for real-time image processing. The survey concentrates on cost-effective and commercially available hardware, as these provide the most reasonable alternatives for creating machine vision systems.

Traditional hardware systems that have been specifically designed for real-time image processing use Application Specific Integrated Circuits (ASICs) or embedded processor chips to

achieve the high performance required for low-level image processing functions. More recently, FPGA-based custom computing machines have been used for real-time image processing.

2.1.1 Custom ASIC Boards

The hardware design solution with the highest throughput for any particular image processing task is created by designing a custom ASIC chip to perform the required calculations. However, the process of creating a custom ASIC using gate array chips or VLSI design is very complicated and expensive. To justify the development costs, many units of the custom chip must be sold or the price per unit must be very high. Therefore, a small number of custom ASIC chips have been developed to solve particularly prevalent image processing tasks, such as the video compression and decompression chips used in the DVI chip set from the Intel Corporation [7].

There have also been several real-time image processing hardware platforms which utilize custom ASIC chips. One such board is the DT2856 line scan processor from the Data Translation Corporation [8]. This Industry Standard Architecture (ISA) bus adapter card performs several common line scan processing operations, such as light compensation (offset and gain correction), LUT, template matching, and RLE. A high-level architecture diagram for this board is shown in Figure 2.1-1. The board is capable of performing all its functions with an input pixel rate of up to 20 Mpixels/sec. Although fast and inexpensive, this board is only capable of performing the very limited set of operations for which it was designed.

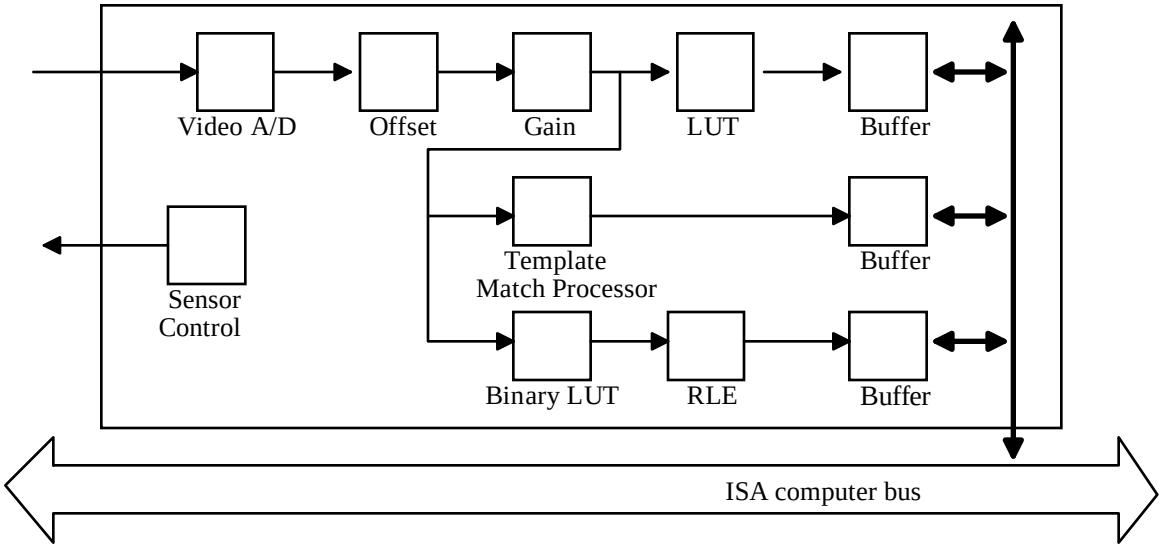


Figure 2.1-1. Block diagram of the DT2856 board [8].

There are also several examples of image processing boards and systems which are not limited to a simple set of low-level operations. Many of these boards, such as the Aspex PIPE [9] and Datacube's MaxVideo 200 [10], use multiple ASIC image processing chips using a pipelined architecture. Since both of these boards have similar architectures, only the Aspex PIPE is described in greater detail.

The Aspex PIPE is a parallel computer intended for real-time image applications. PIPE systems are configured with from one to eight Modular Processing Stages (MPSs). Each MPS contains the following functional elements:

- (2) image buffers
- (3) multifunction Arithmetic Logic Units (ALUs)
- (3) single valued LUTs
- (2) 3x3 or 9x1 convolution operators
- two valued LUT (TVF)
- crosspoint switches/multiplexers

The image buffers can be written to or read from by the host computer. Convolution operators are used for either arithmetic or Boolean operations. The TVF can be used for any function of two variables (e.g., multiplication, division, addition, min/max). A block diagram of a single MPS is illustrated in Figure 2.1-2. Multiple MPS boards are interconnected using the IN, OUT, and VIDEO busses shown in Figure 2.1-2.

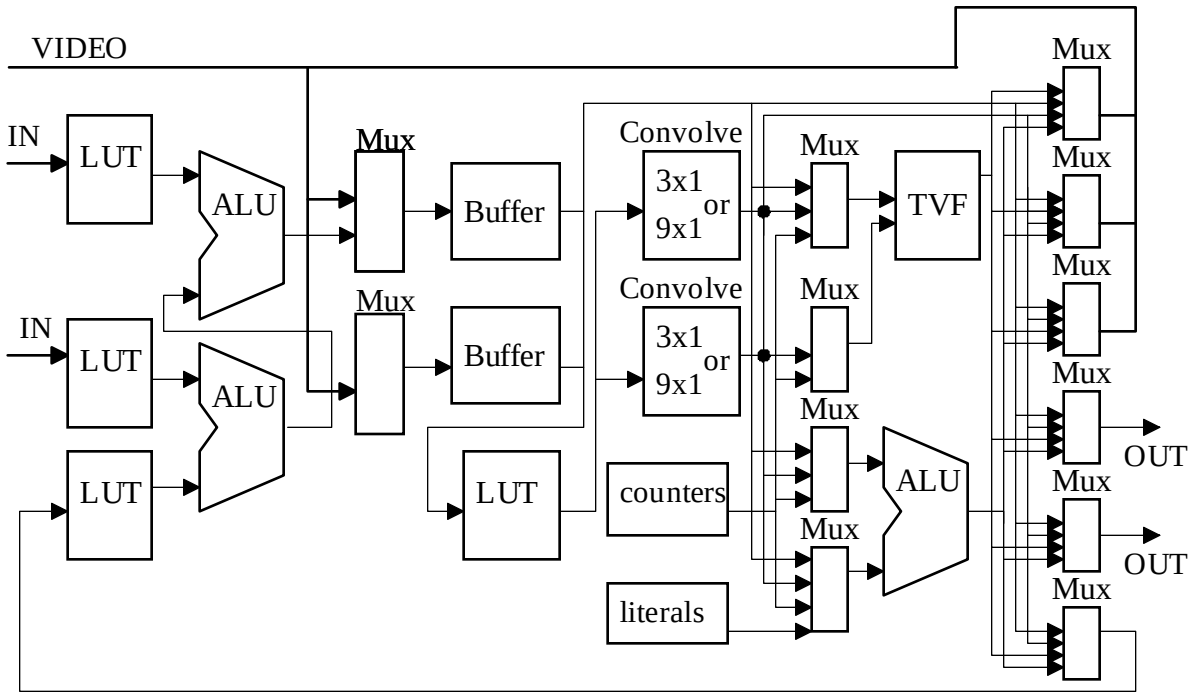


Figure 2.1-2. Block diagram of a modular processing stage of the Aspex PIPE [9].

The systems introduced in this section use pipelined processing units (see Figure 2.1-1 and Figure 2.1-2). A pipelined architecture is well-suited for many low-level image processing tasks. However, there are many processing tasks for which a pipelined processor is not optimal. To overcome this problem, the Aspex Corporation produces two additional boards to perform tasks not easily accomplished on the PIPE processing units. The first board performs histogramming and feature extraction. A second board is used for connected component labeling. The existence of these additional boards illustrates the limitations of ASIC pipelined processors. The inflexibility of an ASIC solution limits the type and number of image processing operations that can be performed by the hardware architecture.

2.1.2 Embedded Processors

The fixed functionality of custom ASIC chips contrasts directly with the flexibility offered by embedded processors. The high performance and low cost of current Digital Signal Processing (DSP) chips makes them particularly well-suited for use in image processing hardware. By reducing and simplifying the memory interface and processing unit, DSP chips are able to provide very high performance at a low cost. The cost of more expensive Reduced Instruction Set Chips (RISC) or Complex Instruction Set Chips (CISC) may sometimes be justified by more sophisticated development tools or increased performance.

Three image processing boards that use DSP chips are the DT2878 Advanced Processor from the Data Translation Corporation [8], the 4MEG VIDEO Model 12 from the Epix

Corporation [11], and the 3032 digital signal processing board from the Vision Modules Corporation [12]. Similarly, the Max860 processing board from the Datacube Corporation [10] uses an Intel i860 RISC microprocessor as the computational unit. The Eurocom 17 board from the Eltec Corporation [13] uses two Motorola 68040 CISC processor chips. The architectures of these boards are analogous to conventional embedded processor products, with the addition of very high I/O bandwidth capabilities. These boards all provide between 12 and 25 Mega Floating-Point Operations Per Second (MFLOPs) and approximately 70 MIPs of peak performance. However, these boards do not meet the real-time processing requirements of even a simple 5x5 convolution operation at common input pixel rates (shown to require over 375 MIPs in section 1.2.1).

The random memory access of an embedded processor allows the calculation of imaging functions such as geometric operations that are not easily accomplished by pipelined processors. Although these boards provide a significant performance increase over personal computers, many operations are not accomplished at common image data input rates. These architectures are not easily scaleable by adding more than a single embedded processor into the architecture to achieve the desired processing rates. Therefore, many of these embedded processor boards are considered "accelerator" boards, instead of real-time processing solutions.

2.1.3 FPGA-based Custom Computing Machines

FPGA chips provide an array of reconfigurable logic resources consisting of combinational logic functions, flip-flops, and interconnections. Functionality of current FPGA chips is established by programming Static Random Access Memory (SRAM) bits within the chip. This allows FPGA chips to be programmed after power is established to the chip and be reprogrammed for consecutive tasks an unlimited number of times.

The 4000 series of FPGA chips from the Xilinx Corporation are typical of current SRAM-based FPGA chips [14]. The structure of Xilinx 4000 series FPGA chips is illustrated in Figure 2.1-3. Sequential or combinational logic functions are created with Configurable Logic Blocks (CLBs) which are arranged in a 2-dimensional array. Each Xilinx 4000-series CLB contains four output connections, sixteen input connections, two 4-input function generators, and two flip-flops. SRAM bits are used to establish the logic functions implemented by each function generator and route signals through the CLB using multiplexers. Input/Output Blocks (IOBs) are distributed around the edge of the array for connection to package pins of the FPGA chip. Programmable interconnect resources are used to interconnect the individual logic functions implemented in each of the CLBs and IOBs. FPGA chips in the 4000 series are available in a variety of sizes, from 3,000 to 25,000 equivalent logic gates. Additionally, Xilinx 4000 series FPGA chips are available in a variety of speed grades (e.g., -3, -4, and -5, where -3 is the fastest). The speed grade number does not correspond to any physical attribute, it establishes a relative performance level between different FPGA chips.

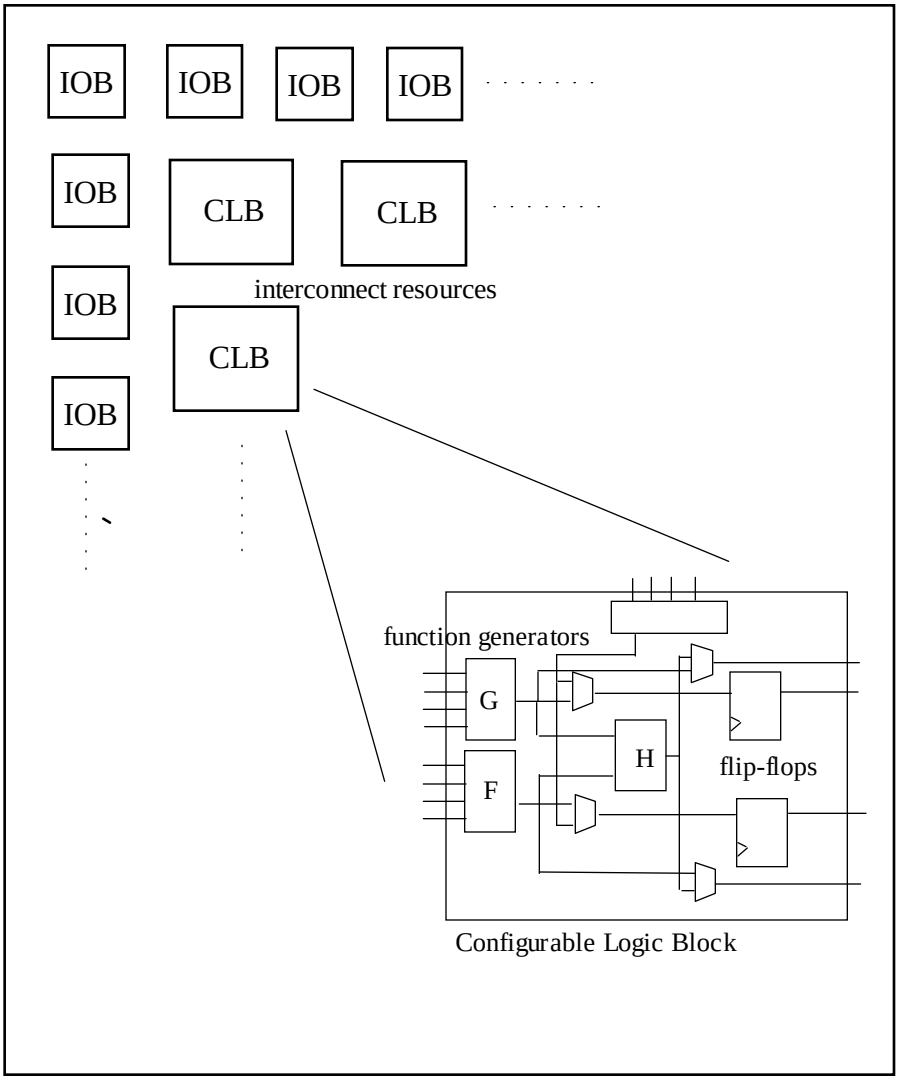


Figure 2.1-3. Xilinx 4000 series FPGA structure [14].

FPGA-based Custom Computing Machines (CCMs) use the reconfigurable logic provided by FPGA chips for computing elements. The logic resources provided by FPGA chips are used to create specialized processing units at the gate level for any desired application. FPGA-based architectures have been shown to provide an excellent hardware resource for low-level image

processing functions such as template matching [15], morphological operations [16], and window operators [17]. These applications involve the use of a single image processing operator.

Complicated designs that consist of several interconnected image processing functions have also been investigated. One such example is the infrared missile warning system presented by Box in [18] which processes two image data streams using spatial filtering and median filtering using an FPGA-based CCM. Another example of a significantly more complex FPGA-based design consisting of several interconnected low-level modules with different data formats is presented by Quenot in [19]. The Hough Transform, which consists of successive applications of low-level operations to an input image, is implemented on the FPGA-based Splash 2 architecture in [20].

Several FPGA-based custom computing machines have been developed specifically for real-time image processing. Early hardware architectures utilized a linear array of FPGAs [21], or a single FPGA integrated with a small SRAM [22]. However, the practicality of these approaches was limited by the FPGA's ability to perform arithmetic functions and store operands.

Recently, more sophisticated architectures have been proposed by Nozal [23], Rautiola [24], and Van den Bout [25] that integrate FPGA chips with standard memory, DSP, or Programmable Interconnect Chips (PIC) to overcome the limitations of implementing processing tasks on FPGA-based architectures that contain only FPGA chips. The most successful architectures combine FPGA, memory, and interconnect chips. Three such architectures are the Virtual Computer [26], CHAMP [18], and Splash 2 [15, 27]. These architectures couple SRAM with FPGA chips, then provide an interconnection scheme using PICs, crossbar switch elements,

or other FPGAs. These solutions represent large architectural solutions containing from 14 to 52 FPGAs with fixed support chip resources.

Several more cost-effective solutions have been developed for Personal Computer (PC) systems. A novel type of architecture is presented by Spectrum reconfigurable computing platforms from the Giga Operations Corporation [28]. From one to 16 Multi-Chip Modules (MCMs) containing FPGAs and RAM are assembled on a reconfigurable interface card. The BORG board [29] is an inexpensive ISA bus prototyping adapter card with four FPGAs, switches, displays, and a open grid area.

The Virtual Computer, CHAMP, Splash 2, and Giga Operations G800 architectures currently represent the most sophisticated FPGA-based Custom Computing Machines (CCMs). These three architectures are described in the following subsections.

2.1.3.1 The Virtual Computer

With fifty two Xilinx 4010 FPGA chips and twenty four ICUBE IQ160 PIC chips, the virtual computer board represents one of the largest FPGA-based custom computing machines. Each board also contains 8 MBytes of SRAM, 256 Kbytes of Dual-Port RAM (DPRAM), and three 64-bit I/O busses. The architecture of the Virtual Computer is shown in Figure 2.1-4.

Computations are performed in the 8x8 array of FPGA and PIC chips, called the Virtual Array. The basic processing unit of the Virtual Computer is defined as the Virtual Pipeline. Each Virtual Pipeline consists of two adjacent rows of FPGA, PIC, and dual-port RAM chips in the

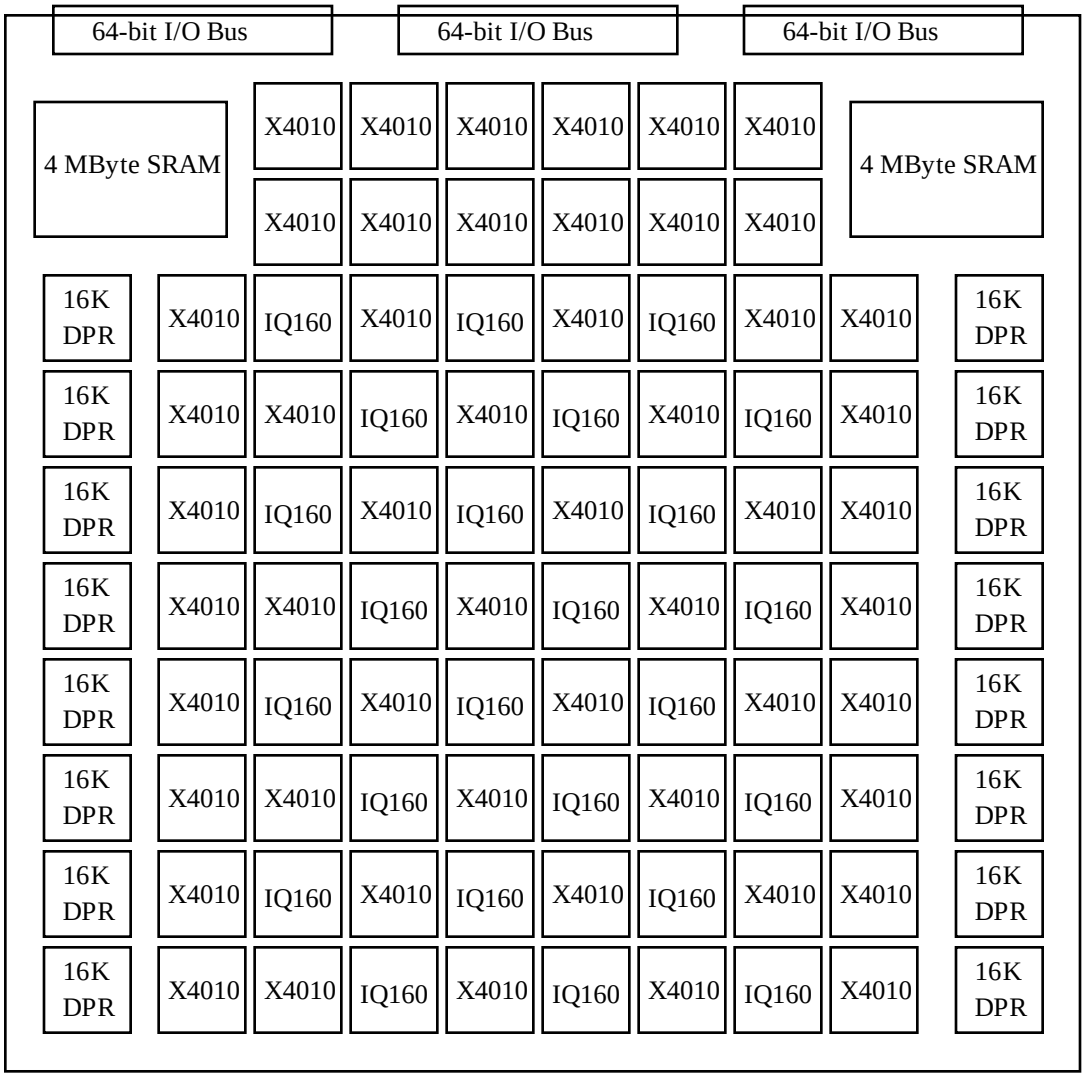


Figure 2.1-4. Virtual Computer Architecture [26].

Virtual Array. Interconnections of the Virtual Array consist of a slightly augmented rectangular mesh of 32-bit interconnection busses.

Although the author of [26] does not state that the Virtual Computer is intended for real-time image processing, its high bandwidth I/O, large memory subsystems, and embedded pipeline architecture suggest it may be a good candidate.

2.1.3.2 Champ

The Configurable Hardware Algorithm Mappable Preprocessor or CHAMP board is specifically designed for real-time image pre-processing. Its architecture is shown in Figure 2.1-5. Each Processing Element (PE) contains two FPGAs and a 16Kx32 DPRAM which is shared by both the FPGAs, as shown in the insert of Figure 2.1-5. Two PEs are coupled to form a PE pair, with a dedicated sensor input and two-port memory. The current CHAMP board

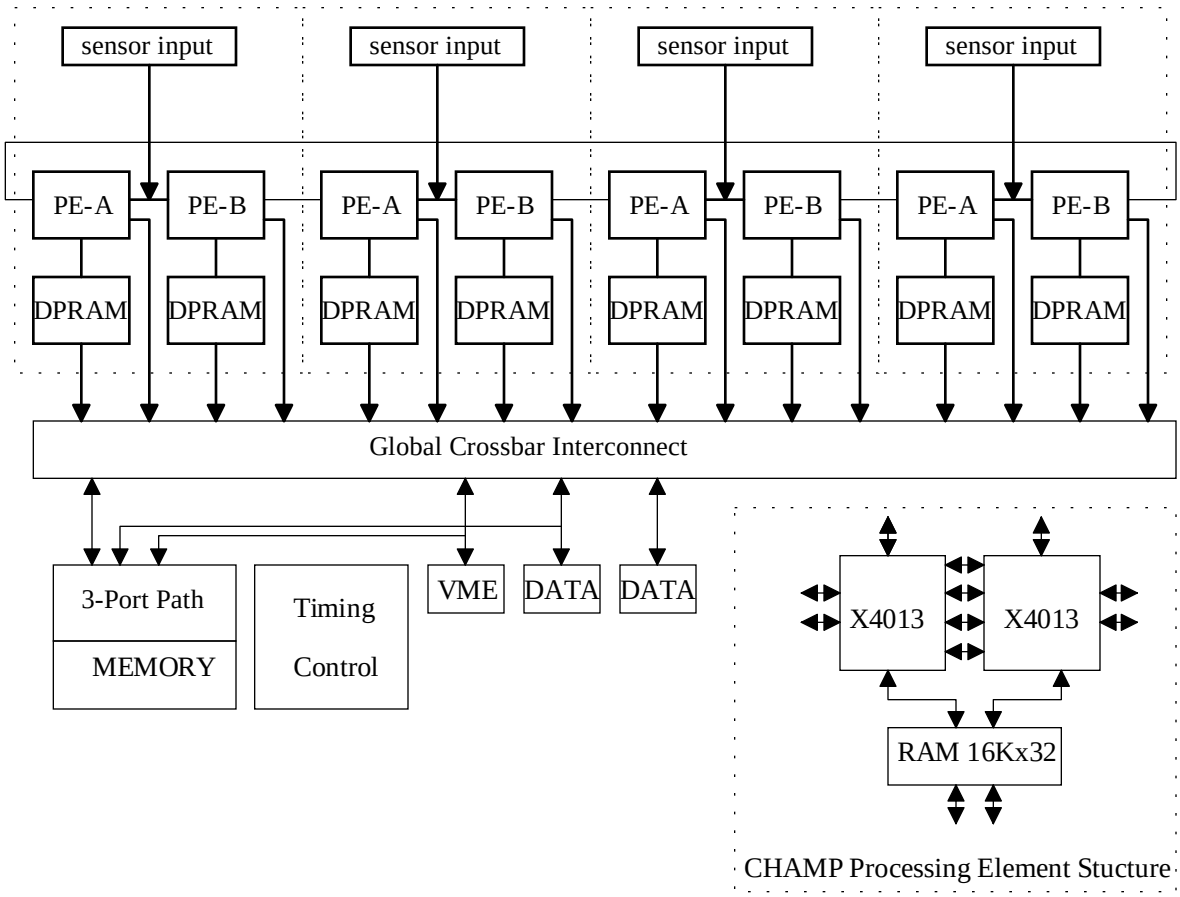


Figure 2.1-5. CHAMP system block diagram and PE architecture [18].

contains four of these PE pairs. The four PE pairs are interconnected with dedicated 16-bit interconnections as well as through the dual-port memories to a global crossbar interconnect.

The control strategy of the CHAMP architecture is of particular interest. All memory addresses are generated in the control logic hardware. This is done to simplify the task of partitioning the logic among the eight PEs, a process which is done manually for each design.

An infrared missile warning system is implemented on the CHAMP board by the Box in [18]. Input data is generated by 128x128 sensors with frame rates of up to 1000 frames/sec. A multiple path pipeline is used to solve the problem, represented as a data flow graph. The system requires a throughput of 16 MBytes/sec, making it one of the most powerful image processing architectures currently available.

2.1.3.3 Splash 2

Splash 2 is an attached parallel processor that utilizes FPGA chips as processing elements. Each Splash 2 system contains a Sun Sparcstation host, an interface board, and from one to sixteen Splash processing boards. The architecture of each Splash processing board is shown in Figure 2.1-6. The sixteen Xilinx 4010 FPGA chips form a linear array, as well as being fully interconnected by a crossbar switch. Five hundred Kbytes of RAM are tightly coupled to each FPGA. The final FPGA chip, X0, is used to control the configuration of the crossbar switch. Two ports are available for communication between separate Splash processing boards in a system. Two additional ports are used for communication with the interface board, providing

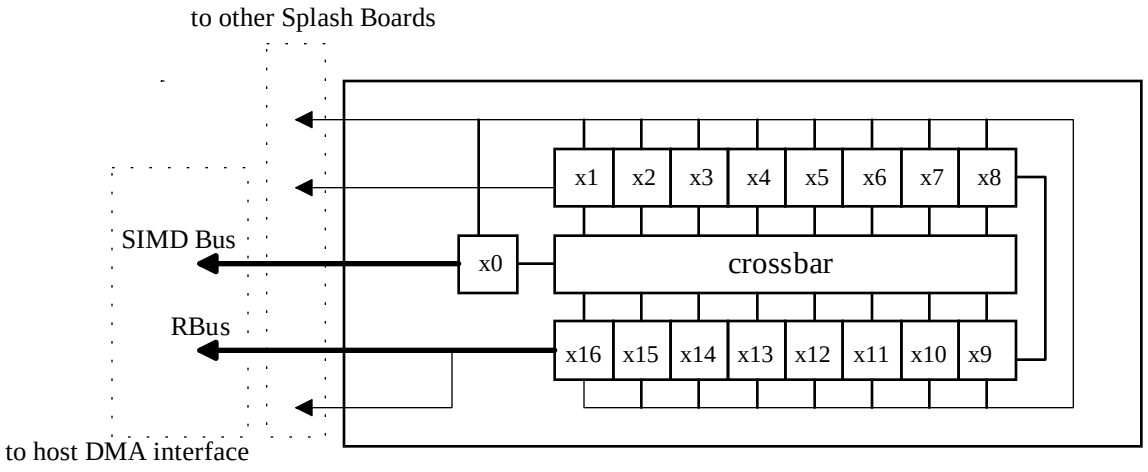


Figure 2.1-6. Splash 2 processor array card architecture [27].

high bandwidth Direct Memory Access (DMA) transfers to the Sun Sparcstation host machine.

The Splash 2 system can implement a variety of programming models, including single instruction multiple data (SIMD), one-dimensional pipelines, and higher dimension systolic models [27]. The interconnect resources provided by the crossbar of the Splash 2 architecture provides the flexibility required to implement these various programming models. High bandwidth provided by the DMA access to the Sparcstation host allows manipulation of very large amounts of data.

A large number of applications have been successfully implemented on the Splash 2 system [15, 17, 20, and 30]. A 3x3 Sobel window operator is implemented by Ratha in [17], requiring 13.89 msec of processing time for a 512x512 gray scale image. This is less than the maximum of 33 msec between the image frames generated by a camera operating at 30

frames/sec. The real-time processing capabilities of the Splash system have motivated much of the research into FPGA-based CCMs.

2.1.3.4 Spectrum Reconfigurable Computing Platform

The only hardware architecture with configurable hardware resources is the Spectrum reconfigurable computing platforms from the Giga Operations Corporation. These systems consist of a Reconfigurable Interface Card (RIC) and multiple FPGA-based XMOD processing units.

Two RIC cards are currently available, the G800 adapter card for the Video and Electronics Standards Association (VESA) local bus and the G900 adapter card for the Peripheral Component Interconnect (PCI) bus [31]. Four connectors are available on each RIC card for mounting XMOD processing units. Since XMOD processing units can be stacked up to four modules high, up to sixteen XMODs can be assembled onto a single RIC. A high bandwidth DMA channel is available on both of these RIC boards for quickly transferring data directly into the memory of the host computer. A block diagram of the G800 RIC and XMOD is shown in Figure 2.1-7.

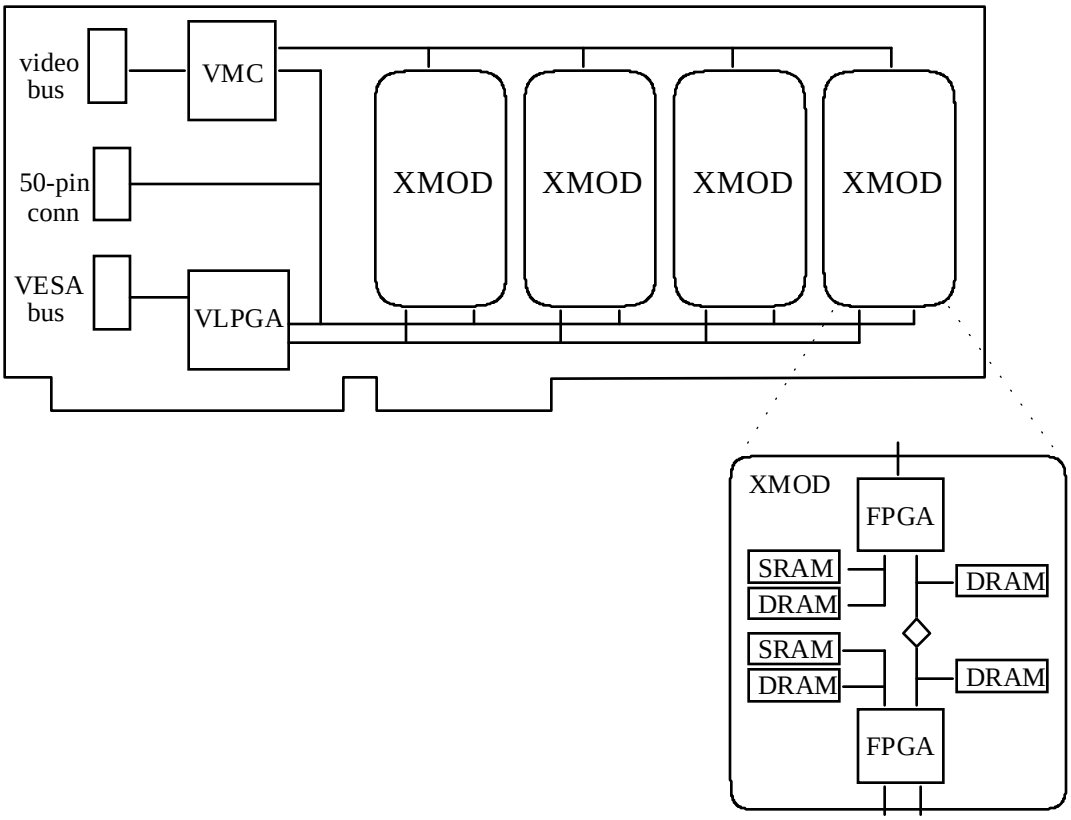


Figure 2.1-7. Spectrum G800 RIC and XMOD block diagram [28].

2.1.3.5 Limitations of Current Hardware Architectures

The Virtual Computer, CHAMP, and Splash 2 architectures represent very sophisticated, but large and expensive systems. Several thousand dollars are required to purchase the large number (from 16 to 52) of FPGAs required for any of these three systems [32]. Only the most complicated and compute intensive image processing applications are implemented using these architectures. Implementing smaller, simpler functions is completely cost ineffective. Although

some of the previous architectures in this section are board-level or module-level configurable, none are configurable at the chip-level, except for the BORG which is only a prototyping card. This lack of modularity is a significant disadvantage associated with all these architectures.

Another disadvantage of all the previous systems is that their architectures are completely defined. There is no method for embedding special purpose chips into the architecture of these FPGA-based CCMs. The only flexibility of these systems is in the interconnections of the FPGA chips, when PIC or crossbar chips are used.

A new incompletely-specified FPGA-based architecture is defined in this dissertation. The new architecture has chip-level modularity, unlike any of the previous FPGA-based CCMs. Additionally, IC chips or modules can be embedded into the architecture. These advantages provide a new incompletely-specified multiple FPGA-based architecture that overcomes the disadvantages of previous architectures defined in this section.

2.2 Software Development Systems

There are several software development systems specifically for creating image processing hardware solutions. All these development systems have three distinct processes: design entry, verification, and translation. However, there are significant differences between the implementations of these processes.

Design entry is typically accomplished by selecting one or more low-level operators from a library of existing functions. Some development systems, such as Global Lab from Data Translation Inc. [8], PXIPL from Epix [33], and iTOOLS from the Matrox Corporation [34] allow a single low-level operation or a linear sequence of operations to be performed on an input image. More sophisticated approaches such as the Advanced Imaging Tools (AIT) from Datacube Inc. [35] use a data flow representation for design entry. The data flow method of design entry provides much more flexibility in the type of designs that may be created.

After verification with off-line processing of stored images, the desired functionality is compiled into the resources of the destination architecture. Software is compiled, linked, and loaded into onboard memory for destination architectures with embedded processor chips. Development systems for ASIC boards create the desired functionality by enabling chip-level circuitry, programming data routing, and loading memory LUT values.

The gate-level resources of an FPGA-based custom computing machine require that the translation process of the development system be radically different from the previously mentioned systems. In this case, a functional description, textual or graphical, must be compiled into bitstreams that establish gate-level functionality in one or more FPGA chips. Logic of the input image processing design must be partitioned among the available FPGA chips. After partitioning, the nets of the design that interconnect logic in separate FPGA chips must be assigned to physical net connections on the destination architecture. Finally, an environment for compiling, downloading, and interactively verifying the image processing designs is essential.

One such system is the custom development system defined by Quenot in [19] which also uses a data flow graph for design entry. The development system translates designs into the register-level resources of a type of custom coarse-grained FPGA called a Field Programmable Operator Array (FPOA), but unfortunately cannot be compiled into commercially available FPGA chips. A library of low-level image processing modules are combined in a similar manner to program the multiple-FPGA CHAMP board [18], but without the benefit of an integrated design environment.

Many of the previously published FPGA-based image processing designs, such as [18] and [20], have been created without the benefit of a development system intended for image processing applications. Each image processing design has been created using the general-purpose development tools associated with the individual destination architectures and design entry software tools. This absence of a development system for image processing results in a longer design time and lack of portability to other designs.

Although not specifically designed for image processing, there are many useful and unique aspects of the Splash 2 development tools, specifically the debug environment. The Splash 2 development environment is one of the most sophisticated development systems available for an FPGA-based custom computing machine. Design entry is accomplished using VHSIC Hardware Description Language (VHDL). All functions that are to be mapped into the Splash 2 board are first created as VHDL descriptions. These designs are manually partitioned and compiled into a gate list using existing logic synthesis tools. The gate list is then compiled into the FPGA chip resources using existing automated placement and routing tools.

To verify designs are correct, complete VHDL descriptions for all the components of the Splash 2 system exist. ASCII files are used to define crossbar settings and initial memory configurations. Additionally, a C-code library is available to develop the host interface program to be debugged using the VHDL simulation tools defined above. Finally, an interactive symbolic debugger with clock control and register read features enables in-circuit emulation [36]. This debugger provides a valuable tool for identifying errors in the design process.

Most current FPGA-based CCMs use Xilinx FPGA chips, including the new FPGA-based architecture created for this dissertation. All development systems targeting Xilinx FPGA chips use the Xilinx Design Manager (XDM) software to compile image processing designs into the bitstreams required to program the individual FPGA chips. One significant limitation of the XDM software is its inability to partition input logic among multiple FPGA chips, a process called *circuit partitioning*. All current development systems use manual circuit partitioning.

After partitioning, nets crossing chip boundaries must be assigned to interconnection resources of the destination architecture, a process called *global routing*. This process is also performed manually by existing development systems. This dissertation describes software created for the development system to perform both circuit partitioning and global routing. This significant contribution provides the most fully automated design translation procedure currently available in a software development system for FPGA-based CCMs.

There have been many advances in the development systems for FPGA-based computing devices, but very few complete systems have been created. None of the existing systems were specifically intended for image processing. This dissertation describes the first complete software

development system created specifically for image processing using multiple FPGA-based CCMs, called TRAVERSE. This suite of software programs creates a fully automated design and debug environment. TRAVERSE uses the data flow graph representation used by conventional image processing development systems such as Datacube's AIT combined with sophisticated simulation and debug tools similar to the Splash 2 development environment.

2.3 Summary

Different low-level image processing operations have a wide variance in their processing requirements, as illustrated by the low-level processes defined in Section 1.2. The offset operation is ideally suited for a computation using pipelined architecture, but geometric operations such as rotation and warping are very difficult to implement with pipelined processors. Because of these differences, it is difficult to create a single ASIC processing units that can be used for a wide variety of low-level image processing tasks.

Embedded processors allow a wider range of applications to be performed. However, the processing rate of even a specialized embedded processor will always be less than an ASIC chip. When a single embedded processor does not provide enough computational performance, it is difficult to expand the number of embedded processors to achieve the additional performance required for real-time implementation.

FPGA-based CCMs provide a fast and flexible solution for low-level image processing tasks. The gate-level programmability of FPGA-based CCMs provides flexibility not found in ASIC chip implementations and the performance of specialized FPGA-based image processing designs exceeds the limited performance of embedded processor boards.

Current FPGA architectures begin to overcome the limitations of FPGA-based computing by embedding RAM and interconnect chips into their architectures. However, current FPGA-based CCMs are still very expensive and inflexible.

To overcome these limitations, a completely new class of incompletely-specified FPGA-based architectures has been defined by this dissertation. The new architecture allows chip-level modularity not provided by any other existing FPGA-based architecture. Additionally, the ability to embed unspecified chip types, a unique feature offered only by this new architecture, allows the limitations of FPGA chip implementation to be overcome in a cost-effective method.

A general-purpose FPGA-based CCM for real-time image and vector processing has been designed and fabricated for this dissertation, called the MOdular Reconfigurable Real-time Processing Hardware or MORRPH [37, 38, 39]. This FPGA-based processing unit uses multiple Xilinx FPGAs in an incompletely specified architecture to create a modular solution for image pre-processing. The MORRPH architecture is the first FPGA-based custom computing machine with an incompletely specified architecture.

The largest disadvantage of FPGA-based computing is the amount of expertise required to program the FPGA chips. Sophisticated software design tools are required to make

FPGA-based computing a practical design approach. These tools should be specifically intended for the class of design problems the development system is intended to solve.

This dissertation presents the first software development system specifically intended for creating image processing solutions using FPGA-based CCMs, called TRAVERSE. The TRAVERSE software creates real-time image processing solutions for an FPGA-based incompletely specified architecture, such as the MORRPH board. A data flow representation is used to provide the look and feel of other systems. However, the image processing designs are automatically compiled into the Xilinx FPGAs and support chips of an arbitrary architecture. New circuit partitioning and global routing software, not used by any other development system, provide a level of automation beyond any existing software development system for FPGA-based CCMs. This dramatically reduces the expertise and time required to program FPGA-based CCMs with image processing designs.

The contributions of this work extend beyond the new MORRPH hardware architecture and TRAVERSE development system software. Combined, these present a new method of creating hardware solutions for image processing tasks, a new design methodology. Previous image processing designs were created to implement a desired application on an existing hardware architecture. The fixed resources of conventional architectures constrain the possible solution space for the problem. Each architecture has fixed interconnection, I/O, and logic resources. These limitations are used by experienced digital design engineers to create a solution. This design technique creates a solution that is defined by the target architecture, instead of the algorithm itself.

A more efficient implementation is created by a hardware architecture that is configured with the appropriate resources for the image processing function it is intended to perform. The design methodology defined by this dissertation provides the software tools and hardware architecture required to create hardware solutions that are defined by the real-time process that is to be performed, instead of creating a solution to fit an existing hardware platform.

The remainder of this dissertation provides a method for creating and translating these image processing designs into a useful hardware solution. This new design methodology is the most significant contribution of this body of work.

Chapter 3. FPGA-based Destination Architecture

The disadvantages of current FPGA-based architectures, such as their lack of modularity and flexibility, were identified and discussed in the previous chapter. To overcome these disadvantages, this chapter describes a new class of incompletely specified FPGA-based architectures. This new MOdular and Reprogrammable Real-time Processing Hardware (MORRPH) architecture provides the modularity lacking in previous designs to provide a much more efficient and flexible architecture for implementing real-time low-level image processing designs.

The simulation and translation utilities of the development system software could be used as the only method of verification of this new FPGA-based hardware architecture. Simulation tools are available for verifying the functionality of image processing designs. Translation programs provide hardware utilization information. After translation, static timing tools and back-annotated simulations are used to determine the maximum clock frequency at which the designs will operate. This provides a theoretical validation for the new incompletely specified

MORRPH architecture. However, these software simulation and translation utilities provide only a limited verification of the new incompletely specified hardware architecture.

Implementation and manufacturability issues can only be addressed by creating physical hardware with the incompletely specified MORRPH architecture. Additionally, very large designs or image test vectors often require excessive computational resources for simulation and the static timing analysis software produces notoriously conservative estimates. A higher confidence in both the hardware architecture and the developed image processing designs is achieved after the designs are physically realized on an existing hardware platform. Finally, the need for such a device in the current marketplace motivates the creation of a physical implementation for the MORRPH architecture.

A specific implementation of the MORRPH architecture is created for this dissertation, called the MORRPH-ISA board. This ISA adapter card is created as a hardware platform for image processing design verification, debugging, and real-time operation. The performance of individual image processing modules or designs are verified using the clock frequencies required for real-time operation. This provides a high confidence in the correctness and performance of specific image processing designs and modules.

The following sections define the new incompletely specified MORRPH architecture and its benefits over previous architectures. Additionally, the MORRPH-ISA implementation of the architecture and its operation are described in detail. The detailed description of the architecture, its benefits, and its implementation should provide the reader with an appreciation for the

contribution of this new class of incompletely specified FPGA-based CCM architectures and how it will impact the way real-time image processing hardware will be developed in the future.

3.1 Incompletely Specified Architectures

The new incompletely specified architecture is created to overcome limitations of existing FPGA-based CCMs. Although several powerful FPGA-based architectures were presented in Section 2.1.3, none of these represent an efficient solution for a wide range of low-level image processing problems. This is due to the inflexibility in the amount and the type of chip-level resources included in these architectures combined with the wide variance in processing requirements of different image processing tasks.

The key to providing an efficient solution to any problem is to include exactly the required amount and type of resources. Including excess resources is wasteful and insufficient resources will not provide the required functionality. Since image processing algorithms vary significantly in computational complexity, the destination architecture must be assembled with the resources required for each problem to provide an efficient solution for a broad range of applications.

The traditional approach to creating digital hardware with exactly the required functionality is to create a unique Printed Circuit Board (PCB) for each separate processing task. A unique PCB provides very efficient hardware utilization. However, the time and

expertise required to design, fabricate, and debug printed circuit boards limits the practicality of this design approach.

The new solution created for this design methodology is to establish chip-level modularity on a PCB. Copper traces (or tracks) on the PCB are used to interconnect chip-level components, where chip-level components are defined as discrete Integrated Circuit (IC) chips and interconnection parts, including protected headers, cable connectors, and card-edge connectors. These interconnections are established when pins of the chip-level components are soldered to pads on the PCB, using either thru-hole or surface mount connections. In other architectures, the number and type of IC chips included on the board are fixed after the chips are assembled and soldered to the PCB. The new design philosophy is solder only empty sockets to the PCB whenever possible. This allows the number and type of chip-level resources to be determined when they are assembled onto the board by the end-user at a later time.

There are other design alternatives, such as the use of new board fabrication and interconnection techniques. The FPGA-based MCMs used by the Spectrum reconfigurable computing platform provides one example (previously defined in Section 2.1.3.4). From one to 16 of these MCMs with FPGAs and RAM are assembled onto a reconfigurable interface card. However, the type and size of the FPGA and RAM on the multi-chip modules are fixed after components are assembled and soldered to the module. No method exists to include other chip types into the architecture without fabricating a separate module using the existing interconnection scheme. This product provides board-level modularity, without chip-level modularity.

Industry standards are essential to successfully implement the chip-level modularity used by the new design methodology. For example, the location of pins used for programming the FPGA must be consistent if different FPGA chips are to be used in the same socket of an existing PCB. Although not formally established, FPGA manufacturers do place programming interface pins and power/ground pins in consistent locations for chips in the same "family" of FPGA chips. For example, all Xilinx 4000 series FPGAs with either 191 or 223 IC Pin Grid Array (PGA) package types have configuration, power, and ground pins all in the same individual pin locations. This allows an open PGA socket to accept any of a number of Xilinx FPGA chips with different logic densities, speed grades, or interconnection resources [14].

This type of chip-level modularity can be extended to non-FPGA chips. Both the directionality of the I/O pins and functionality of the interface logic in the FPGA chips are defined when the FPGA is programmed. This allows interconnections to be established by traces on the PCB between I/O pins of the FPGA chip sockets and empty sockets on the PCB. Only the type and number of chips required for the particular real-time image processing task are assembled by the end-user into the empty sockets on the PCB, when the non-FPGA chips are tightly-coupled to the programmable I/O pins of an FPGA socket.

The wide variety of chip package types that are currently available (e.g., PGA, DIP, SIPP, ZIGZAG, PLCC, LCC...) creates difficulty in the selection of a socket that accepts all chip types. Some socket adapters are available to convert between different chip and socket types, specifically to convert from 60-mil DIP and Zig-zag In-line Packages (ZIP) to 20-mil DIP spacing [40, 41].

Although the use of sockets allows much of the chip-level resources to be determined by the end-user, it is typically beneficial to define some functionality of the board. Specifically, some logic should be included on the PCB to provide for programming the FPGA chips after power is established to the board. This may be a single ROM chip or a sophisticated interface to a host computer.

The new design philosophy therefore represents a mix of undefined and completely defined logic resources and interconnections. This creates a new class of FPGA-based architectures, called *incompletely-specified architectures*. An incompletely specified architecture is defined as any architecture for which at least some of its chips or interconnections are not fixed by the architecture, but are determined by the operation that is to performed by the architecture.

Interconnections that exist on the printed circuit board should support the types of architectures that are typically used to solve problems for which the board is intended. For example, both pipelines and mesh architectures are commonly used to solve image processing problems. The interconnections of any incompletely specified architecture intended for image processing should support the translation of circuits with these structures. It is difficult to translate image processing designs onto a destination architecture when appropriate interconnection resources do not exist. Therefore, a new incompletely specified architecture is defined to solve low-level image processing tasks. The new architecture is called the Modular Reprogrammable Real-time Processing Hardware or MORRPH [39]. This new hardware architecture is defined in the next section.

3.2 MORRPH Design Philosophy

MORRPH is a general-purpose, FPGA-based processing unit intended for real-time 2-dimensional image processing. Other applications include 1-dimensional signal processing, 2-dimensional cellular automata problems, and 3-dimensional image processing. The SRAM-based FPGA chips allow the processing unit to be reconfigured an unlimited amount of times after power is established to the FPGA chips.

The MORRPH concept specifies a two-dimensional rectangular mesh of Processing Elements (PEs) called the *processing array* or MORRPH PE array, scaleable in both dimensions to an MxN array. The structure of each PE is illustrated in Figure 3.2-1. Four interconnection busses are used for communication with the processing elements to the North, South, East, and

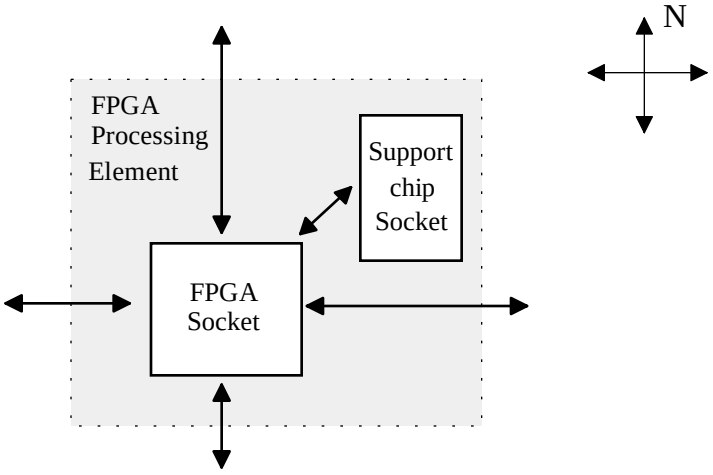


Figure 3.2-1. MORRPH processing element architecture.

West. The interconnection busses are all the same size, providing a regular architecture. A final bus, the support socket bus, connects the FPGA socket to the support chip sockets.

A single FPGA can be inserted into the FPGA socket of each PE. The SRAM-based FPGAs contain gate-level logic resources that can be reconfigured many times. Currently, several companies such as Altera, National Semiconductor, and Xilinx produce FPGA chips [42, 43, 14]. The selection of FPGA manufacturer is an implementation decision, not a feature of the architecture.

Support chip sockets provide enough space to insert several support chips. These storage or processing chips implement functionality that is not efficiently realized by the hardware resources of the FPGA chips. Several other multiple FPGA architectures defined in Section 2.1.3 include additional memory and/or switching elements. However, the type and number of these additional chips have been defined by each architecture.

The FPGAs in the MORRPH PE array are configured after power is established to the chips. This is usually accomplished by the host computer, when the card is an adapter card in a general-purpose computer. The host computer bus standard and host computer interface are implementation details not specified by the MORRPH architecture.

The incompletely specified architecture of the MORRPH board allows a hardware solution to be created in a chip-level modular manner on an existing platform, using only the required type and number of additional support chips. This allows the MORRPH architecture to adapt to the problem it is used to solve. The incompletely specified MORRPH architecture

represents one of the significant novel contributions of this work, providing a very efficient implementation for real-time processing.

3.3 MORRPH-ISA Design Implementation

A specific implementation of the MORRPH architecture defines the processing array size, FPGA type, and host computer interface. An Industry Standard Architecture (ISA) bus adapter card with a MORRPH processing array is created for this dissertation, called MORRPH-ISA [38]. The adapter card utilizes Xilinx 4000 series FPGA chips in a 3x2 array of PEs (see Figure 3.2-1). A system block diagram of the MORRPH-ISA adapter card is illustrated in Figure 3.3-1.

An ISA-bus interface is included on the MORRPH-ISA that uses port read and write cycles for both FPGA initialization and low-bandwidth communication after initialization. This bus access is slow and non-deterministic, and therefore not intended for real-time data transfer. Real-time image and result data is received and transmitted from the MORRPH-ISA board using the I/O busses at the edge of the processing array.

Three board I/O busses at the edge of the processing array are used to receive and transmit both image and result data. These are the busses on the top, left, and right of Figure 3.3-1. The width of the board I/O busses is the same as the North, South, East, and West interconnection busses of the individual PEs. Image data is input from these I/O busses,

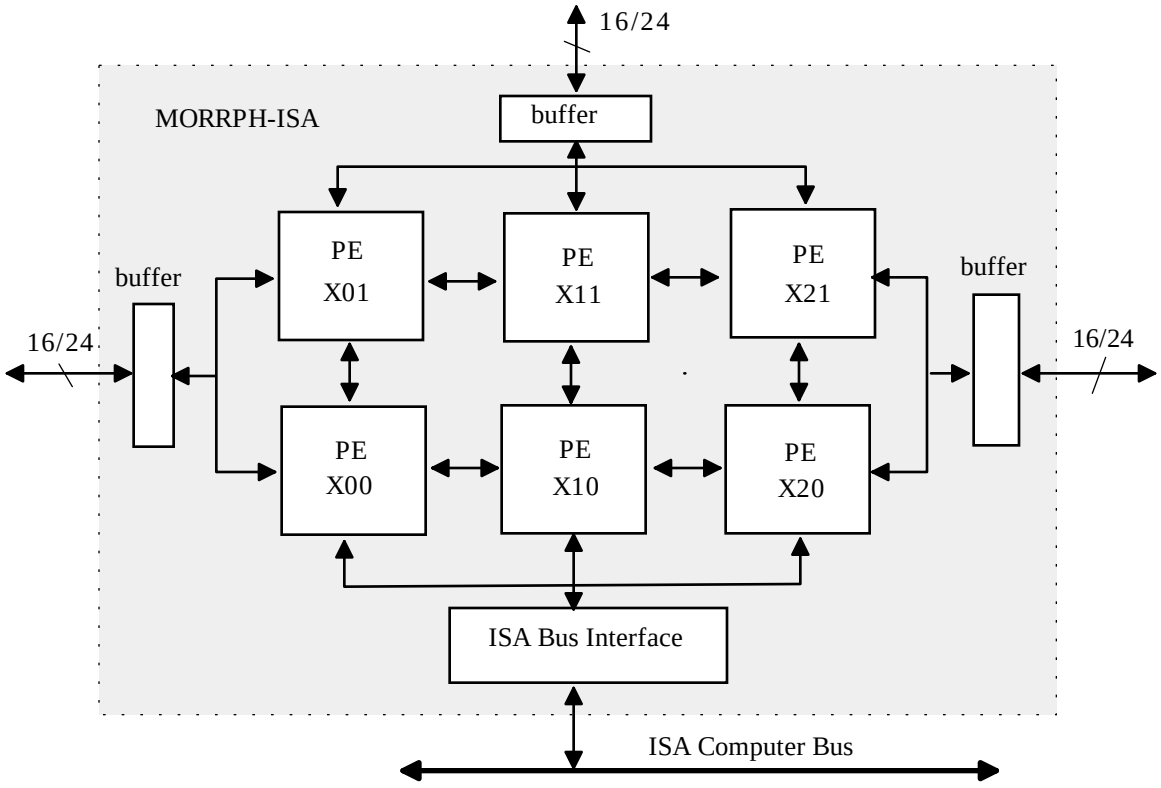


Figure 3.3-1. Block diagram of the MORRPH-ISA board.

processed by the PEs in the processing array, then result data is output on one of the three I/O busses all in real-time.

Direct connection of the board I/O busses to FPGA chips provides the ability to interface with a variety of digital transmission formats, such as the DT-Connect bus standard from the Data Translation corporation [44]. Additionally, the programmable interfaces allow several MORRPH-ISA boards to be interconnected to increase the available processing power.

The processing elements of the MORRPH-ISA board contain PGA sockets for the FPGA chips that accept 191-pin or 223-pin chip package types. This allows any one of the 4003H,

4005H, 4008, 4010, 4013, or 4025 Xilinx FPGA chips to be included as the FPGA in the processing element. Different speed grade options of the 4000 series Xilinx chips allows further flexibility in the type of FPGA to be used. Alternatively, if the PE is not required for interconnection, processing, or configuration of other PEs, the socket may be left empty.

The selection of FPGA type for the processing element impacts the design in several ways, as summarized by Table 3.3-1. Since the number of IOBs varies among the different FPGA types, the depth of the support chip busses is dependent upon the type of FPGA chip used in the processing element. The depth of the interconnect busses is limited by the smallest FPGA of the source and destination PEs. Higher speed FPGA chips allow the MORRPH-ISA adapter card to operate at higher clock rates. FPGA chips with a larger number of CLB resources allow larger designs to be translated into the MORRPH-ISA board, and smaller designs are more easily translated. A single MORRPH-ISA board can be configured with a wide range of FPGA logic resources, from 3,000 equivalent gates (a single 4003H FPGA chip costing $210.00) up to

Table 3.3-1. Processing element FPGA type considerations [15, 16].

FPGA type	Inter-Connection bus depth	Support Socket bus depth	CLB resources	Cost $ (-5 speed grade)
4003HPG191	16	80	100	$ 201.50
4005HPG223	24	80	196	$ 266.40
4008PG191	16	64	324	$ 392.85
4010PG191	16	80	400	$ 478.85
4013PG223	24	80	576	$ 848.85
4025PG223	24	80	1,024	$ 935.65

150,000 equivalent gates (six 4025 FPGA chips costing a total of $5,614.00). In general, the larger FPGA chips provide more logic and routing resources, but are significantly more expensive. As Xilinx introduces new FPGA chips which are backwards-compatible with the current Xilinx 4000 series, the number of FPGA possibilities will continue to grow.

Support sockets accept 30-mil Dual In-line Package (DIP), or Single In-line Package (SIP) chip and module types. Adapter sockets are available for other chip and module types, such as 60-mil DIP and Zig-zag In-line Packages (ZIP). This allows a wide variety of chip types to be inserted into the support sockets.

The I/O pads of Xilinx FPGA chips do not have adequate current source and sink capabilities to provide reliable power and ground for the support chips. I/O pads of the FPGA chip that are connected to power and ground pins of support chips should be driven to a high impedance state (i.e., tri-stated) during and after FPGA programming. Physical connections for power and ground pins are established by the end-user when the support chips are assembled onto the MORRPH-ISA board. This is accomplished by connecting jumper wires from the required support socket pins to power or ground terminal locations at the end of each support socket location.

Xilinx FPGA chips store the array configuration in SRAM bits distributed within the chip. This allows the chip to be reprogrammed at times other than powerup. Therefore, the board can be reprogrammed with different functionality an unlimited number of times. Programming of the FPGA chips is accomplished using I/O write cycles of the ISA bus [45]. The lowest FPGA in each column of Figure 3.3-1 is programmed using 8-bit parallel loading of the Asynchronous

Peripheral Mode defined by Xilinx [14]. Upper FPGAs are "daisy-chained" from the lowest FPGA and are programmed in a bit-serial manner using a single interconnection line. This allows columns of PEs to be independently programmed at any time after power-up. The maximum 500 Kbyte/sec bandwidth of the ISA bus requires only 40 msec to program a column containing two Xilinx 4010 FPGA chips.

All I/O read and writes to the MORRPH-ISA board are mapped into three successive I/O port locations, as defined in Table 3.3-2. The three ports are called the ***address_port***, ***data_port*** and ***program_port*** locations, at offset 0, 1, and 2, respectively. To program a column of PEs in the array, the column is first defined with I/O writes. An 8-bit value is created by setting only the appropriate bit position of the column to be programmed (i.e. the 8-bit value "00000100" for column two). This value is first written to the ***address_port***, then to the ***program_port*** to establish the value **PROGCOL[2:0]** (see Table 3.3-2). Finally, the binary value of the column is written to the **COL[1:0]** value in the ***address_port*** location. Successive writes to the ***data_port*** location are used to program all FPGAs in the defined column in the MORRPH processing array.

Table 3.3-2. MORRPH-ISA I/O port locations.

Port Name	Offset Address	Signal Definitions
address_port	0	**X, COL[1:0], ADDR[4:0]**
data_port	1	**DATA[7:0]**
program_port	2	**X,X,X,X,X,PROGCOL[2:0]**

The number of successive writes is sum of bytes required to program all FPGA chips in the column.

After initialization, the same port locations are used to access registers in the MORRPH processing array. Before any port access, the column (**COL[1:0]**) of the desired port and its address within the column (**ADDR[4:0]**) must be defined with an I/O write to the ***address_port*** location. The 5-bit address allows 32 port locations to be defined in each column. Data for the transfer is then read from or written to the 1-byte ***data_port*** location. The address decoding and port register logic must be programmed into the logic resources of the FPGA chips of the MORRPH processing array.

This section has described a physical implementation of the incompletely specified MORRPH architecture, the MORRPH-ISA board. Figure 3.3-2 shows a MORRPH-ISA adapter card configured with six Xilinx 4010 FPGA chips. This board is used to verify the functionality and performance of six image processing designs created in Chapter 5.

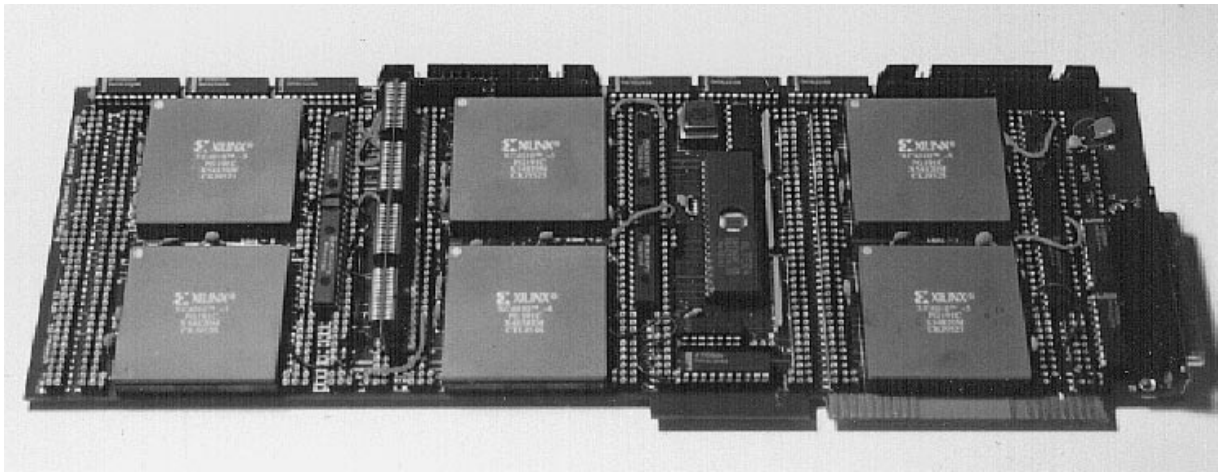


Figure 3.3-2. MORRPH-ISA adapter card.

3.4 Hardware Design Summary

A new class of incompletely-specified architectures are defined for the new design methodology. These custom computing machines provide a powerful but inexpensive solution for a broad class of real-time processing applications. Since the rectangular mesh architecture of the PE array is scaleable, different MORRPH processing units with larger or smaller array sizes can be created around different form factors and bus interfaces.

As an example implementation, the MORRPH-ISA board was created. This board contains a 3x2 array of MORRPH processing elements and the required circuitry for I/O, host bus interface, and FPGA configuration. The MORRPH-ISA board provides a processing resource that is more modular and less expensive than any other commercially available FPGA-based CCM.

The MORRPH-ISA board represents a significant computing resource for low-level image processing. Its three high bandwidth I/O connections and six FPGA sockets provide the ability to input, process, and output image data in real-time. These I/O connections also allow the interconnection of several MORRPH boards to solve larger problems.

The greatest disadvantage of the MORRPH-ISA board is its inability to collect data in real-time. The low-performance 500 Kbyte/sec ISA bus interface does not provide sufficient bandwidth for the real-time collection of data from even moderate performance cameras. A new Peripheral Component Interface (PCI) [31] adapter card, called MORRPH-PCI, is currently under development to overcome this limitation. The 133 Mbyte/sec PCI interface will provide

real-time collection of preprocessed data from the MORRPH-ISA board, as well as offering significant processing power in its own 2x2 MORRPH processing element array. However, even without the ability to collect image data in real-time, the current MORRPH-ISA board presents a powerful tool for the real-time processing of image data. Collection can be accomplished with existing data collection cards, such as the Micro Channel-based data collection card defined in [46].

The current MORRPH-ISA board provides an excellent vehicle for the verification of the new design methodology. Instead of relying entirely on the result of software simulation, the MORRPH-ISA board provides a hardware platform for verifying the functionality and performance of image processing designs and modules. This physical verification at real-time clock frequencies proves that the MORRPH architecture is valid and realizable. In addition to being used to verify the MORRPH architecture, the MORRPH-ISA board also represents a commercially viable product for real-time image processing. The definition of the MORRPH architecture and its implementation as MORRPH-ISA establish the first part of the design methodology, the hardware architecture. Next, the software development system is described.

Chapter 4. Development System Software

4.1 Software Development System Design Philosophy

The previous chapter described the FPGA-based MORRPH architecture, illustrating how it provides gate-level logic resources suitable for implementing real-time image processing tasks. This chapter defines the TRAVERSE development system software of the proposed design methodology. TRAVERSE consists of the design methods and software tools used to establish the functionality of gate-level logic resources provided by a multiple FPGA-based hardware architecture such as the MORRPH-ISA board. The MORRPH hardware architecture and TRAVERSE software development system combine to create the sophisticated design methodology created for this dissertation.

Previous literature describing the use of FPGA-based CCMs for image processing applications presented unique solutions created for specific image processing problems. For example, an image processing design that performs the Hough transform using the Splash 2 architecture is presented by Abbott in [20]. Similarly, a 5x5 convolution operator is implemented

on the Splash 2 architecture by Raza in [17]. Both these approaches represent solutions to particular image processing problems that cannot easily be extended or modified to solve other image processing problems. Additionally, a significant amount of gate-level design expertise is required to create and debug these image processing designs.

Some other attempts, such as Box in [18], use a more modular approach of combining low-level image processes from a library of modules. However, this is done without a formal definition of conventions, standards, and tools used in the design process. The most sophisticated development system for image processing is presented in [19] by Quenot. However, this approach is specific to a custom FPOA array, and cannot be used for FPGA-based CCMs.

The design philosophy used by TRAVERSE creates a library of processing modules for common image processing functions. Each module contains gate-level logic that can be compiled into the resources of commercially available FPGA chips. These specialized processing units are designed at the gate-level or register-level and are optimized for the specific function they are intended to perform. Individual modules are designed to be interconnected with other modules, creating complicated image processing designs represented by the data flow graphs of Section 1.2.3. After software verification, these image processing designs are compiled into the gate-level resources of an FPGA-based CCM.

Only the input/output characteristics and high-level symbolic representation of the individual modules are specified by TRAVERSE, the gate-level design techniques and entry method are not restricted to the methods presented in this dissertation. However, the new design methodology establishes the standard methods required for defining variables, transferring data,

and module representation. These standards allow software tools to be created which automate the design and implementation processes. Additionally, standard design entry methods allow complex image processing designs to be created by interconnecting modular components that have been independently constructed. Modules created for a specific application may be utilized by future image processing designs. The formal definition of these design standards provides a new method of creating image processing designs, representing a significant contribution of this body of work.

As stated in Section 2.2, the TRAVERSE development system requires three distinct processes; design entry, validation, and translation. Several software programs are used in each of these processes. These programs are illustrated in Table 4.1-1. To minimize the amount of custom software development required for the new design methodology, it is desirable to use existing software programs whenever possible. TRAVERSE takes advantage of currently available software tools, providing the desired functionality by integrating commercially available tools with custom software programs. Custom programs written by the author specifically for this development system are highlighted in Table 4.1-1.

Table 4.1-1 Programs of the TRAVERSE development system software.

Program Name	Task	Platform	Author/Vendor
DESIGN ENTRY			
Viewdraw	Creation of modules Creation of designs	UNIX/DOS UNIX/DOS	Viewlogic Viewlogic
TRANSLATION			
MTRANS	**Format translation** **Network partitioning** **Global routing**	**UNIX**	**Drayer**
XDM	Technology mapping Placement Routing	UNIX/DOS	Xilinx
VERIFICATION			
Viewsim	Functional simulation Timing simulation	UNIX/DOS	Viewlogic
XView	Image display	UNIX	John Bradley
PIXMAN	**File format translation**	**UNIX**	**Drayer**
XDelay	Timing analysis	UNIX/DOS	Xilinx
MDBUG	**In-circuit emulation**	**DOS**	**Drayer**
IDSP	Image Display	DOS	SDA lab

The development procedure begins with design entry, followed by verification with functional simulation. If functional errors are detected by the verification process, the procedure must return to the design entry process to correct each error. This procedure continues to the translation process after a correct functional simulation is obtained, followed by verification with

in-circuit emulation. Again, the procedure returns to design entry when errors are detected. Typically, the creation of an image processing design is an iterative process. The designer repeatedly returns to the design entry process to correct errors, improve performance, and modify functionality of the image processing design using feedback provided by the verification process.

Each of the three processes of TRAVERSE and their associated software programs are described in detail in the following sections. The programs and methods combine to establish the tightly integrated development system that is required for the new design methodology to be practical.

4.2 Design Entry

The design entry process creates a computer representation for the desired image processing task. Theoretically, this representation could be either textual or graphical. The ViewDraw schematic capture program from the Viewlogic Corporation [47] is used for this dissertation to create a graphical representation of the hardware schematic for an image processing design. Textual representation, such as VHDL code [48, 49], is an example of another possible method for design entry. However, the graphical schematic capture representation is preferred to textual representation, so VHDL is not used in this dissertation.

The design methodology starts with the generation of a data flow graph which represents

the physical processes of an image processing task. Individual processes of the data flow graphs presented in Section 1.2.3 are represented by the vertices of a directed graph. A circle is used to represent the process and edges of the graph represent communication between processes. The goal of this design entry process is to provide a method of creating hierarchical schematics which closely resemble the data flow graphs commonly used to represent low-level image processing tasks.

The Viewlogic representation of a circuit contains two components, symbols and schematics. The top-level schematic is an interconnection of symbols. Symbols are interconnected with nets and busses. Each symbol may represent an underlying schematic which is itself a hierarchical interconnection of other components. Symbols at the lowest level of the hierarchy (i.e., ones that do not contain underlying schematics) are called *primitives*.

TRAVERSE transforms the input data flow graph into a circuit that provides the desired functionality. Vertices are translated into logic circuits that implement the desired function and edges of the data flow graph are translated into bus connections between the subcircuits. A top-level Viewlogic schematic in which all the symbols are created as circles closely resembles a data flow graph and requires little translation. This top-level schematic containing an interconnection of image processing modules is defined to be an *image processing design*.

There are some differences between this type of schematic and a data flow graph. First, busses are not represented by directed edges. Second, interconnections between modules may only be made through pins attached to each symbol. Finally, some additional labeling is required to define bus widths and module names.

Design standards must be established if individually designed modules are to be interconnected into complex image processing designs. These standards can be as complicated as the protocol used to communicate between modules or as simple as the determination of the graphical size of symbols and text. Standards for the design methodology are provided throughout this and subsequent sections.

Hierarchical design techniques create several levels of detail for each low-level image processing module. Experienced digital hardware designers create gate-level schematics down to the lowest level of the hierarchy. These designers use the standard methods defined by this development system to produce libraries of low-level image processing modules that are arbitrarily interconnected. Libraries of modules allow complicated image processing designs to be created by individuals that are unskilled in gate-level digital hardware design. These individuals design at the highest level of the hierarchy, interconnecting modules and defining attribute values. Therefore, inexperienced designers may create complex designs using the complicated modules created by others.

The following sections define the standards and conventions used to design image processing modules. First, methods for creating symbols are established. Next, a synchronous bus standard is defined for transferring image and result data between modules. This bus standard is then extended to provide inter-board communications. A standard method for defining and modifying operand values required for the processes of individual modules is established. Finally, the methods required to create schematics for image processing modules are defined.

4.2.1 Image Processing Module Symbols

Symbols were defined previously to be circles, to closely resemble the vertices of a data flow graph. The addition of pins, labels, and attributes are required for the schematic representation of the image processing modules.

Pins are added to the symbol to define connection points for input and output nets and busses. Each pin is assigned a unique label that identifies a corresponding net or bus in the underlying schematic of the symbol. Bus labels begin with a text name followed by numerical indices to determine the width and range of the bus. The indices are separated by a colon and enclosed in square brackets. For example, labels ***AA[15:0]*** and ***QQ[15:0]*** are used consistently in the example problems that follow for the labels of the 16-bit input and output bus pins. These labels are assigned to pins but are not typically displayed on the symbol. Although the labels may not be visible in the schematic, label values are required to identify corresponding busses in the underlying schematic.

The directionality of all pins must be established. This is done in two ways. First, a hidden attribute of either "PINTYPE=IN" or "PINTYPE=OUT" is attached to each pin. Additionally, a small arrow is added before each pin of a symbol to graphically indicate the directionality of the pin. This arrow removes ambiguity created by using busses instead of the directed edges of data flow graphs.

Attributes may be attached to a symbol or pin to provide additional information. For example, a 16-bit bus synchronous bus standard called the Synchronous Unidirectional Image

Transfer or SUIT bus standard is defined in the next section. This bus standard is used for transferring image and result data between modules. The attribute value "CLASS=SUIT" is attached to all pins that represent busses in the SUIT bus format.

Many image processing modules require input operands for their processing functions. The implementation of these operand values is discussed in detail later in Section 4.2.4. One method to define an operand is to attach a visible attribute to the symbol with the name of an underlying bus or net in the top-level schematic. Any visible attributes on the symbol are intended to allow the input of operand values or to alert designers of special functionality of the module.

The functionality of the image processing module is identified by the text name provided above the symbol circle. This corresponds to the file name used to retrieve the image processing module.

The following example is provided to illustrate the design of a symbol for an image processing module. This example is only partially useful, as some important labels and attributes are not visible on the symbol. Label and attribute values are not displayed when they do not provide information that is useful in the highest level schematic, the data flow graph representation. These invisible labels and attributes are identified in the accompanying text of the example.

Example 4.2.1 Symbol of an image processing module for adding an offset to all pixel values: An image processing module is desired to add a constant value to all pixel

values in an image. This module is given the name **OFFSET_A**. The Viewlogic symbol for the **OFFSET_A** module is shown below in Figure 4.2-1. The "OFFSET[7:0]=" and "CHAN[3:0]=" attributes allow these operand values (required for the processing functions of the **OFFSET_A** module) to be defined. The "PASS" attribute is discussed later in Section 4.2.5. The input bus pin on the left of the symbol has the hidden label ***AA[15:0]*** and the hidden attributes "PINTYPE=IN" and "CLASS=SUIT". Similarly, the output bus pin on the right of the symbol has the hidden label ***QQ[15:0]*** and the hidden attributes "PINTYPE=OUT" and "CLASS=SUIT".

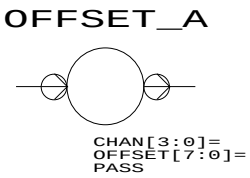


Figure 4.2-1. Symbol for the OFFSET_A image processing module.

4.2.2 SUIT Bus Format

A suitable interconnect bus should be flexible enough to transmit a variety of image or result data formats. Although some synchronous bus standards are available for transmitting data between board-level hardware using transmission cables, no suitable data formats for transmitting

image data on a single chip or board exist in the current literature. Therefore, a new synchronous bus protocol is established, the Synchronous Unidirectional Image Transfer or SUIT bus format [37].

The SUIT bus protocol has been created and refined using experience gained from creating the existing image processing modules and using knowledge obtained by studying other image bus standards. The DT-Connect bus standard from the Data Translation Corporation [44] provides bi-directional communication for data, but no specific signal definitions to make it particularly well-suited for transmitting image data. Two additional digital image transmission formats are represented by the TL2600 RGB color line scan camera format from the Pulnix Corporation [50] and the MAXbus standard from the Datacube Corporation [51]. Both of these bus standards have specific signal definitions that support the transfer of image data. However, these bus standards are intended for inter-system communication. For this design methodology, a bus standard is required for communication between separate modules which are implemented in the same FPGA chip or in separate FPGA chips on the same board. Therefore, these modules have access to a common global clock signal to synchronize the transfer of data.

The SUIT bus uses the global clock to synchronize the transfer of information between image processing modules. This eliminates the timing and circuitry overhead required to generate the handshaking signals associated with asynchronous data transfers.

It is desirable to estimate the maximum operating frequency of an image processing design by independently analyzing the propagation delays of all combinational paths within individual modules. The maximum operating frequency of the image processing design would be

limited to the frequency of the slowest module. However, SUIT bus connections between modules contain a propagation delay that is the sum of propagation delay in both the transmitting and receiving modules. To allow estimation of the maximum operating frequency of a design, the propagation delay in the receiving module is eliminated by immediately registering all input SUIT busses in the receiving module.

All flip-flop inputs in schematic of an image processing design must meet setup and hold times relative to the global clock signal. Required setup and hold times are defined by the FPGA chips used in the target architecture. For example, when an image processing design is translated into the logic resources of any -5 speed grade (see Section 2.1.3) FPGA in the Xilinx 4000 family of FPGA chips, the minimum setup and maximum hold times are 6 ns and 0 ns, respectively [14]. Output from the translation process allows these timing requirements to be verified for a translated image processing design (as indicated in Table 4.1-1).

Sixteen signals are defined for the SUIT bus, in addition to the global clock. Eight of the signals are data lines designated ***DATA[7:0]*** and the other eight signals are control lines designated ***DV***, ***CMD[2:0]***, and ***CSEL[3:0]***. The signal locations and definitions are illustrated in Figure 4.2-2. The functionality of these signals is defined in the following paragraphs.

The first four of the control signals are the channel select lines ***CSEL[3:0]***. These four lines define 16 independent channels for the time-multiplexed transmission of image data. Different channels may correspond to separate images or different spectral bands of the same image. Result data can be time-multiplexed with image data on available channels of the SUIT bus.

The data valid signal ***DV*** determines the function of the three command lines ***CMD[2:0],*** as shown in Figure 4.2-2. When the ***DV*** signal is low, the values of the three command lines define eight SUIT bus command cycles. During these bus cycles data is not transmitted on the eight data lines (the value is undefined). The ***marking*** command cycle is used when the bus is idle, denoting a clock period when no information is transmitted. The ***bus reset*** command cycle is used to reinitialize the modules, possibly in the case of an error or system reset. The value of the channel select signals is not defined for the ***marking*** and ***bus reset*** command cycles, but is required for all other commands.

Data on the SUIT bus may represent one, two, or three dimensional data. The ***line start***, ***frame start***, and ***sequence start*** command cycles indicate the start of a 1-dimensional line of

Location	Function
Q15	DV
Q14	CMD 2
Q13	CMD 1
Q12	CMD 0
Q11	CSEL 3
Q10	CSEL 2
Q9	CSEL 1
Q8	CSEL 0
Q7	DATA 7
Q6	DATA 6
Q5	DATA 5
Q4	DATA 4
Q3	DATA 3
Q2	DATA 2
Q1	DATA 1
Q0	DATA 0

SUIT bus commands:

If DV = 0	If DV = 1
000 - Marking	000 - 1 Byte data/Outside ROI
001 - Bus Reset	001 - 2 Byte data/Outside ROI
010 - Line Start	010 - 3 Byte data/Outside ROI
011 - Line End	011 - 4 Byte data/Outside ROI
100 - Frame Start	100 - 1 Byte data/Inside ROI
101 - Frame End	101 - 2 Byte data/Inside ROI
110 - Sequence Start	110 - 3 Byte data/Inside ROI
111 - Sequence End	111 - 4 Byte data/Inside ROI

Figure 4.2-2. SUIT bus signal locations and bus commands.

values, 2-dimensional frame of lines, or a 3-dimensional sequence of frames, respectively. After the entire line, frame, or sequence has been transmitted, a ***line end***, ***frame end***, or ***sequence end*** command is transmitted. These start and end commands are used to frame the data as it is transmitted on the SUIT bus. The channel select signals must be valid during the start and end commands to allow the independent channels to represent data of different dimensions and sizes.

Data values are transmitted on the bus when the ***DV*** signal is high. The ***DV*** signal is permitted to be high for only one clock cycle to transfer a single byte of data. During this clock cycle, the three command lines are used to define properties associated with the current data. The high command bit ***CMD[2]*** defines whether the value is contained within an arbitrary Region-Of-Interest (ROI). A separate channel can be used if more than two regions are required for a particular image processing function. The lower two command bits ***CMD[1:0]*** define the word size of the data that is transmitted. Data word sizes of one, two, three, or four bytes are supported by the SUIT bus. Additional bytes of larger data sizes are sequentially transmitted, in order of least significant to the most significant bytes.

One possible enhancement of the SUIT bus standard is to widen the data bus width from eight to sixteen signals. This doubles the available bandwidth of the SUIT bus at a fixed clock frequency. However, the design and verification of the image processing modules becomes more complex when two bytes of one byte word size data (representing two words) are transmitted using the same clock cycle. Therefore, the 24-bit SUIT bus standard allows only transfers of a single word (or part of a word) in one clock cycle of the SUIT bus. This larger SUIT bus should only be used when the data transmitted on the bus represents two or four byte word sizes and

additional SUIT bus bandwidth is needed. The 16-bit SUIT bus standard is used exclusively by this dissertation.

SUIT bus connections are added to a module by adding pins to the module's symbol. The top-level schematic must contain busses with labels that correspond to the labels of each of these pins. Since the SUIT bus is unidirectional, a graphic is added to each pin to clearly define the directionality of the bus connections. These pins and graphics are evident on the input and output SUIT bus connections of the **OFFSET_A** symbol shown in Figure 4.2-1.

Example 4.2.2 Sequence of SUIT bus values for test image: To illustrate the SUIT bus protocol, the sequence of SUIT bus commands and data values required to transmit a gray scale test image are shown in Figure 4.2-3. Data is transmitted on channel 15 (***CSEL[3:0]***=1111) using a single byte format with the entire image inside the ROI (***CMD[2:0]***=100 when ***DV***=1). To transmit this image, the data must be presented with all these command cycles and in this sequence using 19 cycles of the global clock. However, commands and data for other channels (and ***marking*** command cycles) may be randomly inserted into this sequence.

The current definition of the SUIT bus provides a flexible method for interfacing image processing modules, using a minimum number of signals. The 16-bit and 24-bit SUIT busses directly correspond to the 16-bit and 24-bit North, South, East, and West FPGA interconnection and I/O busses of the MORRPH architecture, defined in Figure 3.3-1. This is an example of how

cycle SUIT bus value

t0 01101111 xxxxxxxx - Sequence Start Chan 15
t1 01001111 xxxxxxxx - Frame Start Chan 15
t2 00101111 xxxxxxxx - Line Start Chan 15
t3 11001111 00000101 - Byte(0,0) = 05
t4 11001111 00000111 - Byte(0,1) = 07
t5 11001111 00000100 - Byte(0,2) = 04
t6 00111111 xxxxxxxx - Line End Chan 15
t7 00101111 xxxxxxxx - Line Start Chan 15
t8 11001111 00000100 - Byte(1,0) = 04
t9 11001111 11111111 - Byte(1,1) = 255
t10 11001111 00000011 - Byte(1,2) = 03
t11 00111111 xxxxxxxx - Line End Chan 15
t12 00101111 xxxxxxxx - Line Start Chan 15
t13 11001111 00000000 - Byte(2,0) = 0
t14 11001111 00001111 - Byte(2,1) = 15
t15 11001111 00001110 - Byte(2,2) = 14
t16 00111111 xxxxxxxx - Line End Chan 15
t17 01011111 xxxxxxxx - Frame End Chan 15
t18 01111111 xxxxxxxx - Sequence End Chan 15

05	**07**	**04**
04	**255**	**03**
00	**15**	**14**

Example gray scale image

Figure 4.2-3. Sequence of SUIT bus values for example gray scale image.

co-design of the development system software and hardware architecture is intended optimize performance of the design methodology.

4.2.3 ZEE Bus Format

The SUIT bus defined in the previous section is not suitable for inter-board communications when a common clock is not available to synchronize the transfers. To enable

communication between separate board-level devices with transmission cables in an image processing system, the SUIT bus is modified to include a clock signal. This new 16-bit synchronous unidirectional bus format is called the ZEE bus format.

The lowest order bit of the channel select lines ***CSEL[0]*** in the SUIT bus standard is replaced with a clock signal in the ZEE bus standard, reducing the number of transmission channels from 16 to eight. All other signals are defined as in the previous section. The added clock signal is used to synchronize the transfer of ZEE bus values. Sixteen signals that define the ZEE bus and their locations are defined in Figure 4.2-4.

Each high period of the ***CLOCK*** net defines a single cycle of the ZEE bus. Data must be stable on all fifteen other signal lines before, during, and after the active high time of the clock. There are no requirements on the duty cycle or regularity of the ***CLOCK*** signal. However, the minimum cycle time of the ZEE bus transfer is 100 ns and setup/hold times must be at least 20 ns. This provides a maximum transfer rate of 10 Mbytes/sec for a single ZEE bus operating at the minimum cycle time.

4.2.4 Variable and Constant Operand Values

Image processing modules typically require operands to perform the calculations that implement their desired functionality. These operands may range from a single bit-value to a large array of values. The manner in which the operands are implemented is dependent upon the number of bits required to define the operand value.

Location	Function
Q15	DV
Q14	CMD 2
Q13	CMD 1
Q12	CMD 0
Q11	CSEL 3
Q10	CSEL 2
Q9	CSEL 1
Q8	CLOCK
Q7	DATA 7
Q6	DATA 6
Q5	DATA 5
Q4	DATA 4
Q3	DATA 3
Q2	DATA 2
Q1	DATA 1
Q0	DATA 0

ZEE bus commands:

If DV = 0	If DV = 1
000 - Marking	000 - 1 Byte data/Outside ROI
001 - Bus Reset	001 - 2 Byte data/Outside ROI
010 - Line Start	010 - 3 Byte data/Outside ROI
011 - Line End	011 - 4 Byte data/Outside ROI
100 - Frame Start	100 - 1 Byte data/Inside ROI
101 - Frame End	101 - 2 Byte data/Inside ROI
110 - Sequence Start	110 - 3 Byte data/Inside ROI
111 - Sequence End	111 - 4 Byte data/Inside ROI

Figure 4.2-4. ZEE bus signal locations and bus commands.

Attributes are used to define operands when the value is represented by only a few bits or bytes. These attributes are attached to the symbol of the module without values, as shown in Figure 4.2-1. A corresponding bus or net must exist in the top-level schematic with a label name that corresponds to the defined attribute. For example, in Figure 4.2-1 the two attributes "CHAN[3:0]=″ and "OFFSET[7:0]=" on the **OFFSET_A** symbol correspond to busses with identical names and sizes in the top-level schematic.

The directionality of nets are defined by the symbol pins to which they are attached. Nets attached to any pins with the "PINTYPE=OUT" attribute are defined as output nets. Similarly,

nets attached only to pins with the "PINTYPE=IN" attribute are defined as input nets. Nets connected to any pins with the "PINTYPE=BI" are not permitted to be used for operands.

Operands are defined as constants when a value is assigned to the corresponding attribute. The underlying input bus in the top-level schematic is connected to power and ground signals within FPGA chips by the translation process to establish the correct operand value. For example, when the attribute "CHAN[3:0]=" of the **OFFSET_A** symbol is given the value "CHAN[3:0]=0001" in an image processing design, the high three bits ***CHAN[3:1]*** are connected to ground and the low order bit ***CHAN[0]*** is connected to power.

Operands without assigned attribute values are defined to be variables. These variables are implemented as read or write port locations, and may be accessed at any time by the host computer. Additional circuitry is automatically added to the schematic by the translation programs of TRAVERSE to implement the read port locations, write port locations, and host computer bus interface logic. This additional logic to implement port locations is compiled into resources of the FPGA chips .

Large arrays of variables must be implemented using memory resources; either internally on FPGA chips or in external memory chips. ROM memories are used for constants and RAM memories are used for variables. The logic interface is dependent upon the type, size, and manufacturer of the memory chips. Therefore, the method of interfacing and initializing memory chips is left to the individual designer.

The ability to create constant or variable operands allows flexibility in the implementation of operands used by modules. These two methods of implementing constant and variable operands provide an efficient implementation.

4.2.5 Image Processing Module Schematics

The methods of previous sections defined the representation (Viewlogic symbols) and I/O characteristics (SUIT bus standard) required to independently create low-level image processing modules. This section defines the methods used to create logic schematics which implement the desired functionality of each low-level image processing module.

A ViewDraw schematic contains circuit elements and interconnections combined to create an electrical circuit. This schematic is often a hierarchical representation of the circuit. The top-level of the hierarchy is important in the representation of image processing module schematics because it defines the input and output data busses and variable operands. Partitioning of logic between FPGA and support chips is also done in the top-level schematic.

All logic of a module that is to be compiled into the resources of FPGA chips in the destination architecture must be contained within a single circuit element of the top-level schematic. This circuit element is identified by attaching a visible attribute with the value "XILINX". The "XILINX" attribute definition is used by the automated translation tools defined later in Section 4.4. This circuit element is called the *primary module* of the top-level schematic.

All other circuit elements (i.e., modules without the "XILINX" attribute) are called *support modules*. Support modules are implemented with the external logic of support chips.

Each external support chip required for an image processing module must have a corresponding support module in the top-level schematic. All nets that interconnect one individual support module must be combined into a single bus. This bus must also contain net connections to power and ground pins of the support chip. The logic represented by a support module is not compiled into FPGA resources and is removed from the design by the translation process. However, the bus connections are not removed and are used by the automated chip placement procedure of the translation process.

Logic contained in the primary module of the top-level schematic is translated into FPGA resources. All primitive elements (i.e., components without underlying schematics) in the hierarchy of this module must contain only components from the Xilinx X4000 and MX4000 libraries. These libraries are supplied by the Xilinx corporation for compilation into its 4000 series FPGA chips. Alternatives to using these libraries include using the new unified libraries or hard macros in the HM4000 library, the use of XBLOX [14], and VHDL [48] translation. Implementing all these design entry methods is left for future investigation.

Elementary components such as gates and flip-flops are available in the X4000 and MX4000 libraries. Although more complicated components exist, the libraries are far from comprehensive. Complicated or odd-sized components not included in the X4000 or MX4000 libraries are created with components from these libraries and stored in the in the LIB project.

Several libraries of image processing modules are established by creating projects in the Viewlogic schematic design environment. The four separate data sizes defined in sections for the SUIT and ZEE busses (1-byte, 2-byte, 3-byte, or 4-byte) establish four projects containing image processing modules (SUIT2A, SUIT2B, SUIT2C, and SUIT2D, respectively). A fifth project (SUIT2) exists for image processing modules that are independent of data size.

The differences between data channels and data sizes of the SUIT bus create several design considerations. For example, most modules only process a single channel of the SUIT bus. All other channels may be transmitted on the output SUIT bus without modification. In many cases this is not convenient, and data from other channels is discarded. Either the attribute "BLOCK" or the attribute "PASS" is added to the symbol of a module to define if other channels of image data are blocked or passed. Additionally, most modules only process one of the four defined data sizes of the SUIT bus.

Techniques used to establish the functionality of individual modules are left to the designer. However, two examples follow to illustrate possible methods for the design of image processing module schematics. A simple schematic for the **OFFSET_A** of Example 4.2.1 is shown first in Example 4.2.3. Second, a more complicated schematic is presented in Example 4.2.4 to illustrate the use of support chips. This second image processing module uses an external RAM chip to implement an arbitrary LUT. Other modules are easily created by modifying and extending these examples.

Example 4.2.3 Schematic of an image processing module for adding an offset to all pixel values: This module is given the name **OFFSET_A** and added to the SUIT2A project. The circuitry required to add a programmable offset to each pixel value is easily implemented entirely with FPGA resources, so the only module in the top-level schematic is the primary module, **XOFFSET_A**. The top-level schematic of the **OFFSET_A** module is shown in Figure 4.2-5. Busses ***CHAN[3:0]*** and ***OFFSET[7:0]*** in this schematic correspond to identical attribute values on **OFFSET_A** symbol.

The underlying schematic for the primary module **XOFFSET_A** is shown in Figure 4.2-6. The control byte of the input SUIT bus is always passed to the control byte of the output SUIT bus, after a one clock cycle delay. Similarly, the data byte of the input SUIT bus is passed to the output SUIT bus, unless the input command byte value indicates 1-byte data on the channel defined for processing. In this case, the programmable offset is added to the data byte of the input SUIT bus and multiplexed onto the data byte of the output SUIT bus. Since all other channels are passed to the output SUIT bus without modification, the "PASS" attribute is added to the symbol of the **OFFSET_A** module.

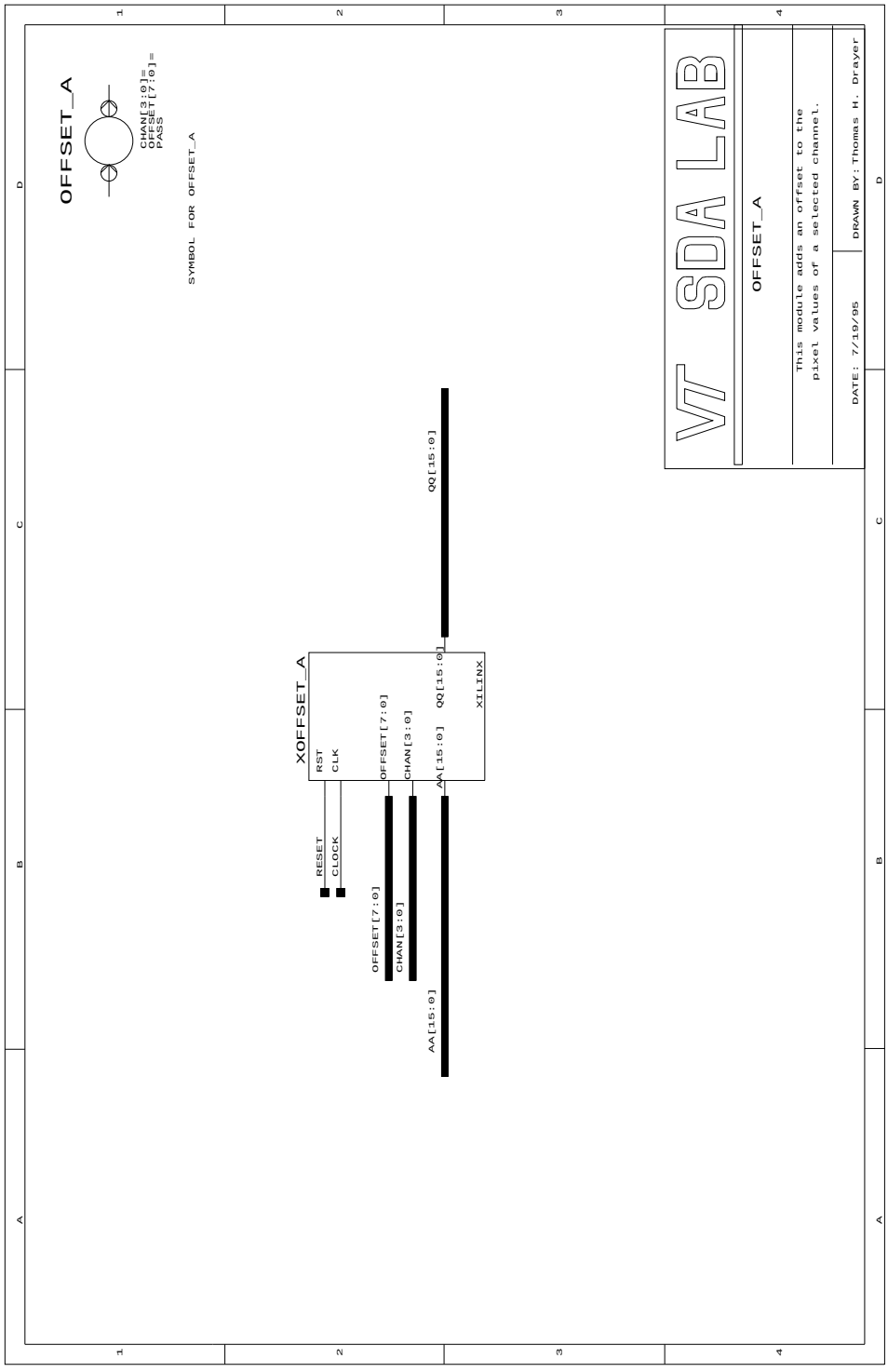


Figure 4.2-5. Schematic for module **OFFSET_A**.

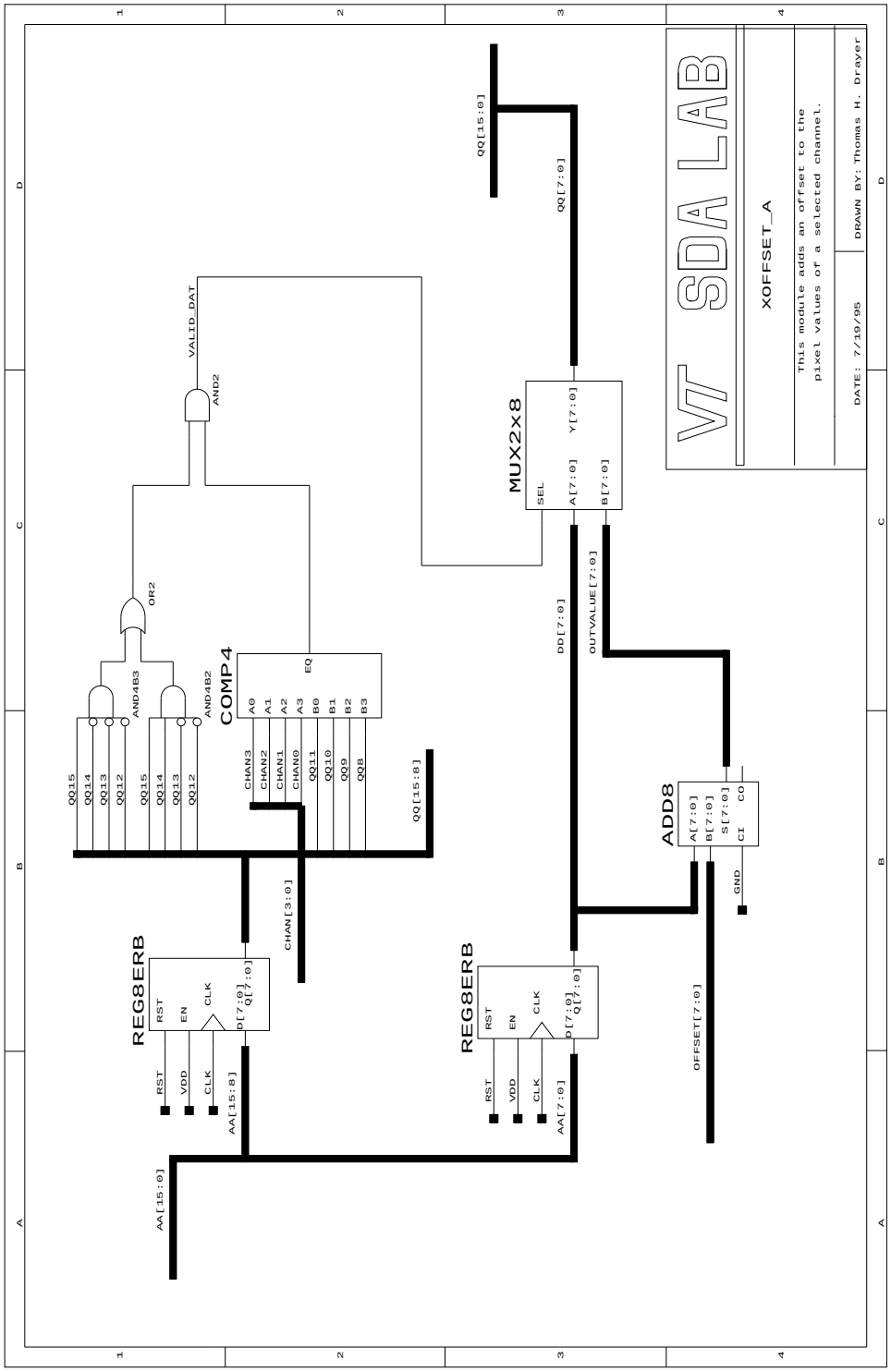


Figure 4.2-6. Schematic for module **XOFFSET_A**.

Example 4.2.4 Schematic of an image processing module for implementing an arbitrary LUT: This module is given the name **LUT_A** and added to the SUIT2A project. The module requires an external 6116 SRAM chip, so two modules are included in the top-level schematic, the primary module **XLUT_A** and the support module **M6116**. The top-level schematic of the **LUT** module is shown in Figure 4.2-7.

The underlying schematic for the primary module **XLUT_A** is shown in Figure 4.2-8. Values for the output SUIT data byte are retrieved from memory locations of the external RAM chip only when the input command byte values indicate 1-byte data on the channel defined for processing. The input data byte provides the lower bits of the address to the RAM chip, indexing into the appropriate location of the LUT. Input SUIT command bytes are always passed to the output SUIT bus with the appropriate data byte, after one clock cycle. This module passes other data channels unmodified (with a one clock cycle latency), so the "PASS" attribute is added to the module symbol.

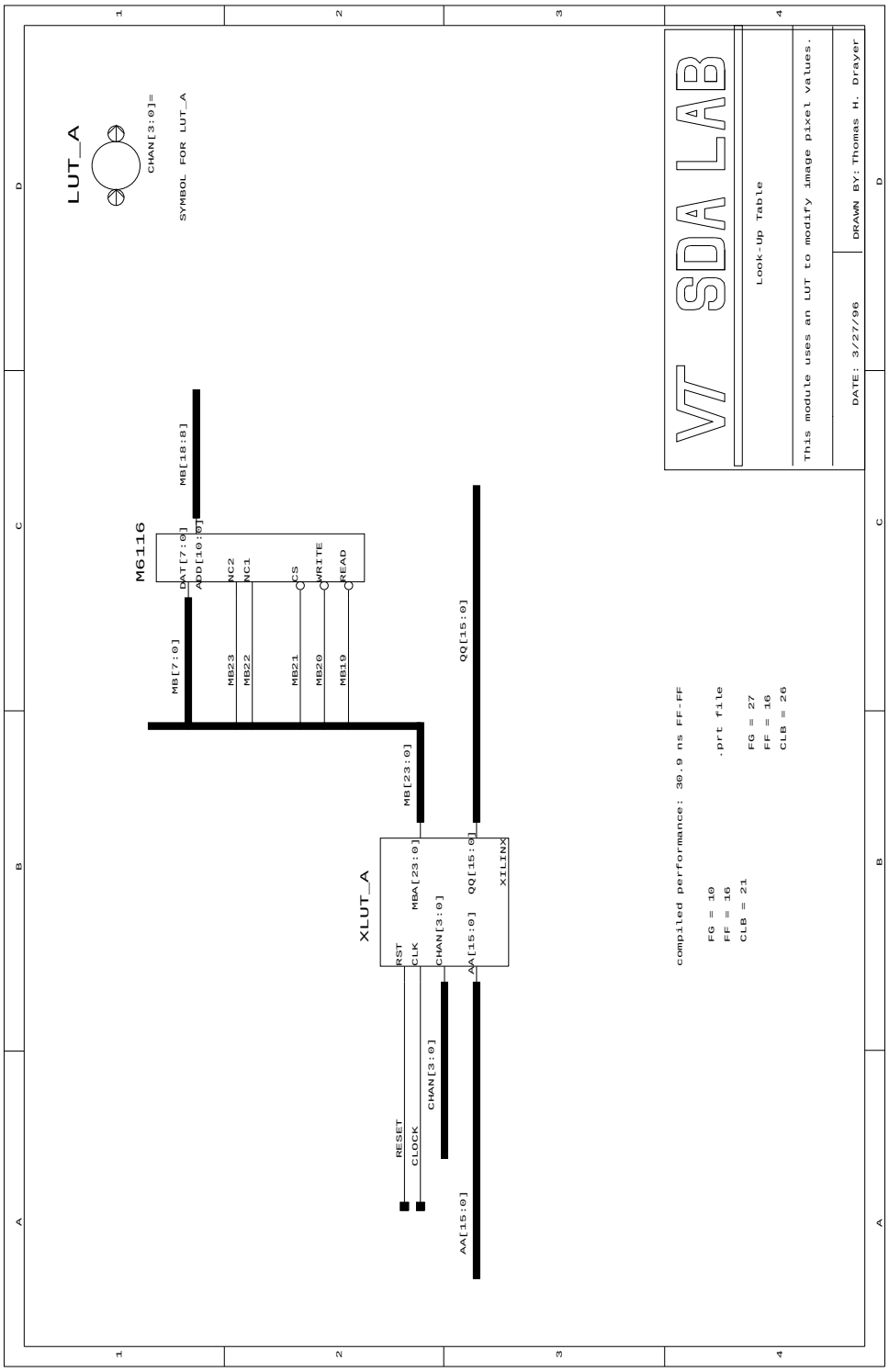


Figure 4.2-7. Schematic for module **LUT_A**.

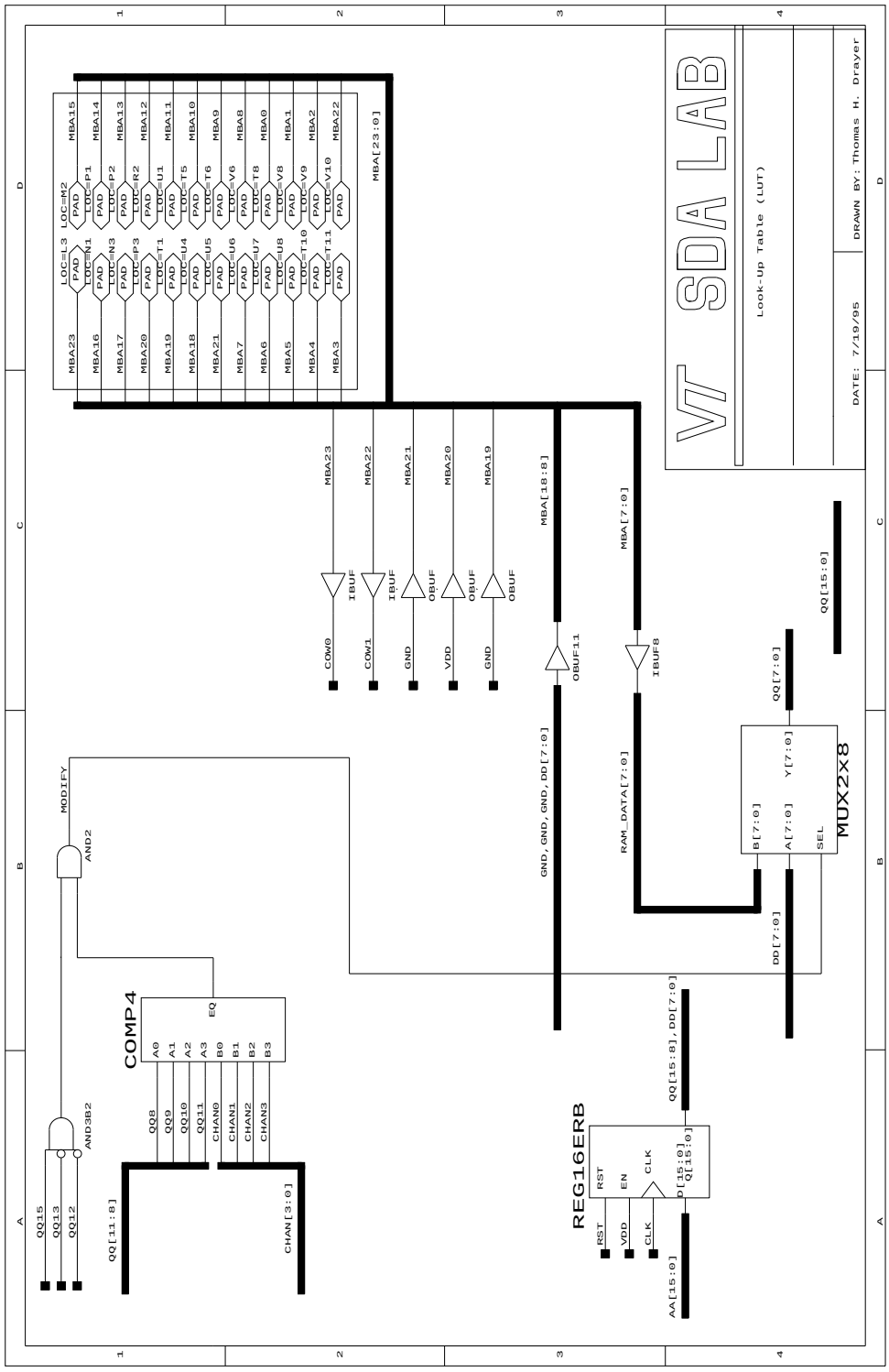


Figure 4.2-8. Schematic for module **XLUT_A**.

4.2.6 Image Processing Designs

Complex image processing designs are created by interconnecting two or more of the low-level image processing modules. The ViewDraw schematic capture program is used to interconnect several low-level modules in a 2-dimensional graphical design environment.

The schematic capture representation provided by the ViewDraw program is analogous to the data flow representation used by the development systems of [19] and [35]. Vertices of a data flow graph are replaced by module symbols and the directed edges of the data flow graphs are replaced by SUIT bus connections.

To create an image processing design, modules are added from the five existing libraries of image processing modules. When appropriate image processing modules do not exist in the library, new modules must be created. Usually, these new modules are easily created by extending or modifying the functionality of existing modules.

Additional overhead modules are often required to create an image processing design with several image processing modules. These additional overhead modules are required to define I/O pin locations, combine multiple SUIT busses, and control the flow of data on different channels.

Busses are added to the input and output pins of image processing modules to establish connectivity between individual modules. These busses are implemented with the SUIT bus protocol, so each bus is defined as a 16-bit bus when its label is assigned. User-assigned labels are required for all busses and modules in an image processing design.

Attribute values are assigned to operands that are defined as constant values to complete the image processing design. These defined attribute values are translated into power and ground connections by the automated translation software of the development system.

An example follows to illustrate a typical image processing design. The example image processing design receives data from the North I/O connection of the MORRPH-ISA board in the ZEE bus format, converts the data to SUIT bus format, adds a programmable offset to the image, and then transmits the result data out the East I/O connection of a MORRPH-ISA board after converting back to the ZEE format. This simple example image processing design is illustrative of the type of modules, interconnections, and attribute and label definitions required to create a complete image processing design.

Example 4.2.5 Image processing design to add a programmable offset to an input image: An image processing design that adds a programmable offset to the input image is shown in Figure 4.2-9. Only one module processes image data, the **OFFSET_A** module from the SUIT2A library introduced in previous examples. This module is given the label "PROCESS". Only data on channel zero is processed by this module because the *"CHAN[3:0]="* attribute is assigned the value "0000". All four nets of the bus ***CHAN[3:0]*** of the top-level schematic will be connected to ground to establish this constant operand. The attribute *"OFFSET[7:0]="* is not provided with a value and will be implemented as a variable. Additional circuitry will be added to define an 8-bit I/O write port location for the host computer.

All four additional modules are selected from the SUIT2 library. The **NORTHIN** and **EASTOUT** modules define the input and output FPGA pin connections used on the MORRPH board for the East and North I/O busses. The **ZEE2SUIT** and **SUIT2ZEE** modules convert data between the ZEE and SUIT bus protocols. All the busses and modules of the design are assigned unique labels as required by the design methodology.

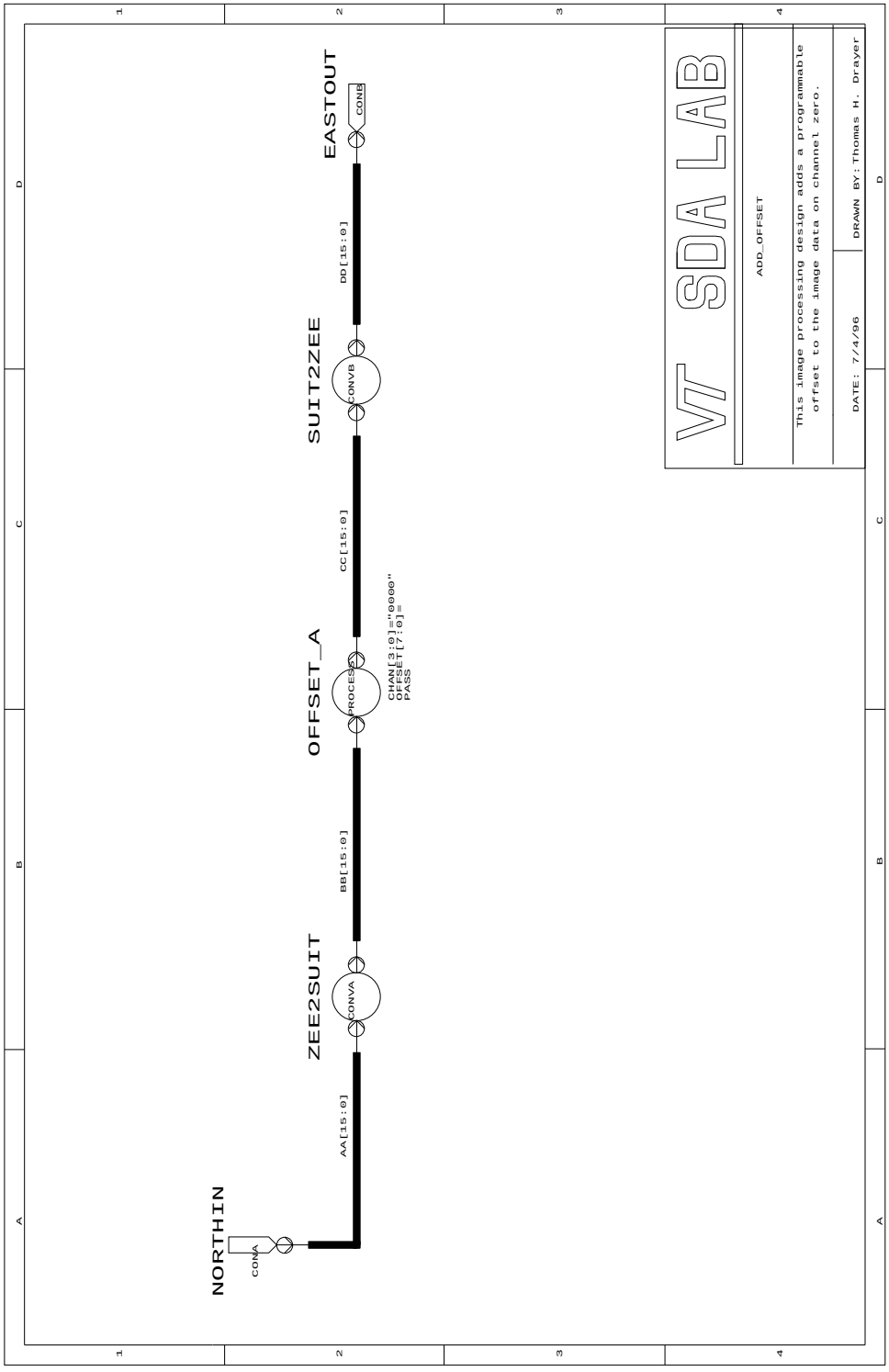


Figure 4.2-9. Image processing design with **OFFSET_A** image processing module.

4.3 Verification

Several different software programs are used to verify the functionality and performance of both the modules and the designs created using the design entry methods defined in the previous section. These programs, shown schematically in Figure 4.1-1 of Section 4.1, provide functional simulation, static timing analysis, and in-circuit emulation. Feedback from these verification programs is used to correct non-functional modules and increase the performance of working modules. Each of these verification processes provide a different perspective and added confidence in the functionality and performance of low-level image processing modules and designs.

The ViewSim program, which is part of the Powerview CAD software package from the Viewlogic corporation, provides software simulation capabilities for the development system. Functional simulation modules are available for all elements of the Xilinx X4000 and MX4000 libraries used to create image processing modules. However, propagation delays in the implemented logic are not known before logic elements are translated into hardware. The ViewSim program assigns a one nanosecond propagation time delay to each gate or flip-flop to allow for a functional simulation. This functional simulation does not address timing and propagation delay issues of the module. After a module is translated into FPGA resources, timing information can be back-annotated to the schematic to allow an accurate timing simulation. However, the back-annotated method of simulation is not currently used by the development system.

The processes used to verify a single image processing module should be similar to the verification processes for complicated image processing designs. However, image processing designs typically contain more logic than individual modules. This amount of logic often requires excessive computational power for the simulation of large designs. Additionally, the ***CLOCK*** and ***RESET*** nets are not available in an image processing design since they are only in the top-level and lower schematics of modules. Complete image processing designs are not simulated by the current software development system for these reasons. Future work should expand the capabilities of the development system software to allow simulation of complete image processing designs.

A significant amount of effort is required to manually create the input data waveforms required to simulate an image processing module with test image data. Similarly, extraction of image data from the simulation output is an equally difficult problem. The SUIT bus standard and regular method of defining operands allow an automated interface to the ViewSim program to be developed. This program, called PIXMAN, is developed by the author to provide a method for importing, exporting, and displaying test image data from a ViewSim simulation.

The Partition Place and Route (PPR) program, part of the XACT software from the Xilinx corporation, compiles a circuit into the resources of a single specified FPGA chip. The output report file contains the number of CLBs, IOBs, FGs, and FFs used to implement the design. These numbers should be viewed as estimates of the required amount of resources, as successive compilations with different seed values or compilations with other circuits may produce varying results.

The XDelay program, also part of the XACT software, analyzes all propagation paths of a compiled circuit. XDelay identifies the longest propagation delay of the compiled circuit to determine the maximum clock frequency at which the circuit will reliably operate. The XDelay program provides an important performance evaluation tool for individual modules.

In-circuit emulation is used to verify compiled image processing designs. The compiled circuit is loaded into the FPGAs of the destination architecture, known test values are provided, and output result data is collected. This output data is analyzed to verify correct operation at real-time clock frequencies. The rich set of possible interconnections of the MORRPH-ISA board allow one MORRPH-ISA board to provide known input data to a second MORRPH-ISA board configured with the compiled image processing design. The MORRPH-ISA board used to transmit data also collects output result data from the processing MORRPH-ISA board. In-circuit emulation is important to verify that image processing designs will operate reliably at predicted clock frequencies.

A second verification program called MDBUG has been created by the author specifically for providing support of in-circuit emulation with the MORRPH-ISA board. MDBUG automates the transmission, collection, and verification of test image data. Functions are provided to configure FPGAs on the board and modify port locations. MDBUG allows the functionality of a compiled image processing design to be verified with known test data at real-time clock frequencies.

The two new software programs created for this development system, PIXMAN and MDBUG, are discussed in the following sections. Together they dramatically reduce the amount of time and expertise required to design and debug new image processing modules and designs.

4.3.1 The PIXMAN program

The PIXMAN program was created by the author to automate the process of providing input to and retrieving results from the Viewlogic simulation of an image processing module. This process includes creating the input waveforms to drive the simulation and translating the output results. The programs associated with PIXMAN and the file conventions used for the transfer of image data are illustrated schematically in Figure 4.3-1.

The PIXMAN program allocates one data buffer to contain a sequence of 16-bit SUIT bus values. These 16-bit values correspond to the signal values on a SUIT bus at each rising edge of the clock. The control byte ***DV, CMD[2:0], CSEL[3:0]*** is stored first, followed by the data byte ***DATA[7:0]***. The data buffer stores either SUIT bus values to be used in a simulation or translated output results from a previous simulation.

A variety of data file formats are available to load or store the contents of the PIXMAN data buffer. Each different file format is identified by a unique file extension. Image files using the ELAS image format [52] have a ".IMG" file extension. This binary file format provides efficient storage of images. Two new file formats, defined in the following, are created for this

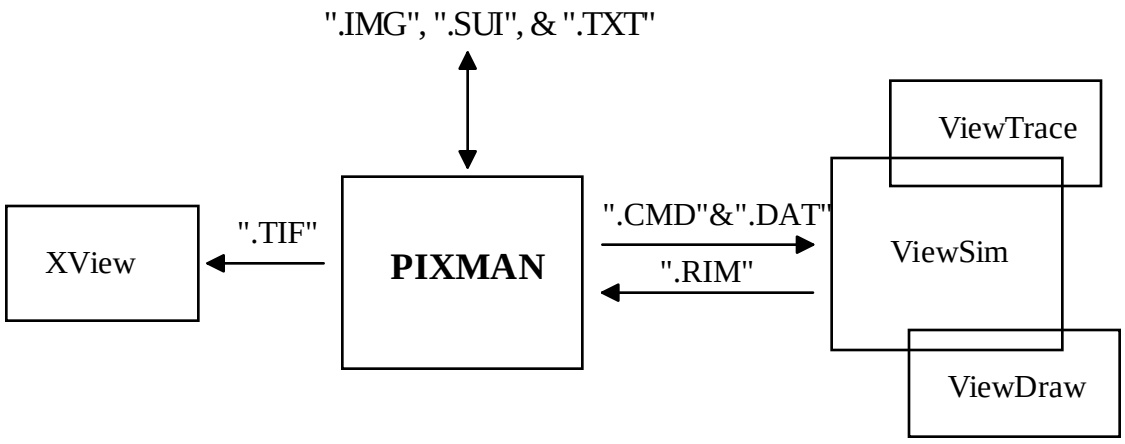


Figure 4.3-1. Input files, output files, and information flow of PIXMAN program.

dissertation to more closely correspond with the actual values on SUIT bus connections of the hardware.

The first new file format, identified with a ".SUI" file extension, contains both the control and data bytes of the SUIT bus in a binary file. Files stored in this format are called *binary SUIT files*. Successive values in the binary data file represent the 8-bit control values followed by the 8-bit data values of the SUIT bus, as the data is stored in the PIXMAN buffer. This method of storage provides a direct correspondence with the actual values transmitted in hardware and the data in the PIXMAN buffer. However, the binary SUIT files typically require more than two times the storage space required by the ELAS image files.

The second new file format, identified with a ".TXT" file extension, contains both the control and data bytes of the SUIT bus in an ASCII file. Files stored in this file format are called

ASCII SUIT files. Hexadecimal values of control and data bytes on the SUIT bus are stored, followed by an ASCII carriage return. This file format typically requires five times the amount of storage of an ELAS image file, but allows the values in the file to be easily displayed and edited using commonly available text editors.

Input data is accepted from any of the three file formats defined above and loaded into the PIXMAN data buffer. A *command file*, an ASCII file with a ".CMD" file extension, is created by the PIXMAN program to control a ViewSim simulation of the current image processing module using the data in the PIXMAN buffer as input test data for the simulation (see Figure 4.3-1). The command file is an ASCII text file as defined in the Powerview Users Guide [47]. The command file controls the ViewSim simulation by defining signal vectors, creating input signal waveforms, and writing output signal values to a file.

The command file defines the input and output SUIT bus connections and drives the input bus with the image data from the PIXMAN buffer. Since the large amount of image data may be unwieldy to include in the command file, it is input from a separate ASCII data file (with a ".DAT" file extension) called the *command data file*. Waveforms for the ***CLOCK*** and ***RESET*** nets are automatically added to the command file, but user input to the PIXMAN program is required to drive the input nets and busses of operand values. For example, user input is required to define the values driven on both the ***OFFSET[7:0]*** and ***CHAN[3:0]*** busses during a simulation of the **OFFSET_A** module shown previously in Figure 4.2-1.

Simulation results are observed in three ways. First, signal waveforms are graphically displayed using the ViewWave program, which is part of the Powerview CAD software package.

Second, the values on the output SUIT bus are stored to a *simulation result file*, with a ".RIM" file extension, at each rising edge of the clock input. Data from the simulation result file can be translated and loaded into the PIXMAN data buffer. Finally, when the ViewDraw program is active with the schematic of the corresponding image processing module used in a ViewSim simulation, the final value of each net is graphically displayed on the net in the ViewDraw window. This is useful for stopping the simulation at a predetermined time and observing values of all individual nets and busses.

Data in PIXMAN data buffer may be stored to a file or displayed by system console. All file formats defined above are supported for output file storage. The data buffer may be displayed using the XView program after intermediate file storage in the Tagged Image File Format (TIFF) format [53]. Alternatively, the data may be written in hexadecimal format to the console, when the amount of data is not large.

The PIXMAN program is written for UNIX systems using the Xwindows programming environment [54]. Similar functionality may be implemented on other operating systems (such as DOS or Windows) using a different Graphical User Interface (GUI). The input options for the PIXMAN program are shown in Figure 4.3-2.

The PIXMAN program provides a method for automatically generating test data and analyzing output data. This automation of the simulation process significantly reduces the effort required for software simulation of image processing modules. The following example is provided to illustrate the files created in the process of simulating an image processing module.

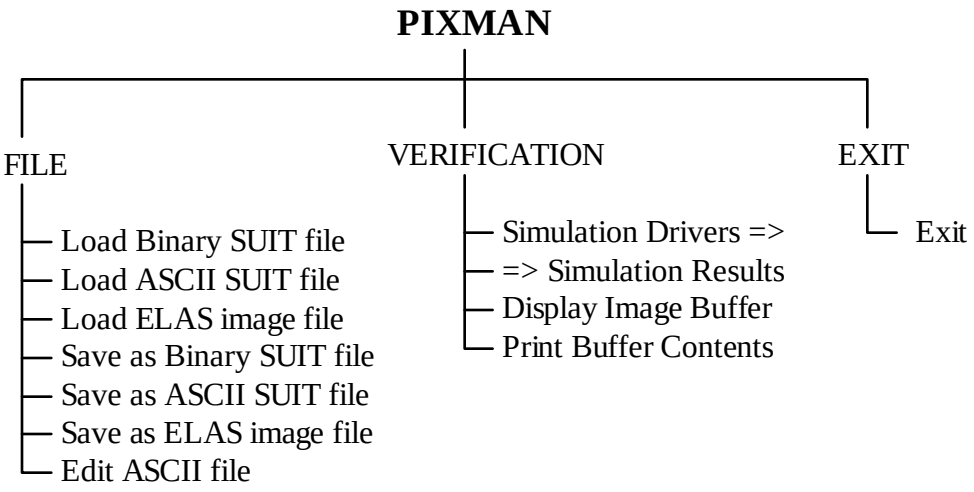


Figure 4.3-2. Program options for PIXMAN program.

Example 4.3.1 Simulation of the OFFSET_A module using the PIXMAN program: The **OFFSET_A** module has been introduced in previous example problems. The test image of for Example 4.2.2 (shown in Figure 4.2-3) is used for input simulation data. This data is manually entered using a text editor to create the ASCII SUIT file shown in Figure 4.3-3a. After this data is loaded into the PIXMAN buffer it is used to create the command file and command data file for the simulation. User input defines the ***CHAN[3:0]*** bus to be driven with the value 0x0F and the ***OFFSET[7:0]*** bus to be driven with the value 0x03. The command file and command data file created by the PIXMAN program are shown in Figure 4.3-4 and Figure 4.3-3b, respectively. Simulation of the **OFFSET_A** module provides the output simulation result file shown in Figure 4.3-3c and the waveforms of the ViewWave display shown in Figure 4.3-5.

```
6F00
4F00
2F00
CF05
CF07
CF04
3F00
2F00
CF04
CFFF
CF03
3F00
2F00
CF00
CF0F
CF0E
3F00
5F00
7F00
```

(A) ASCII SUIT file

```
65NS=0000\H
50NS=6F00\H
50NS=4F00\H
50NS=2F00\H
50NS=CF05\H
50NS=CF07\H
50NS=CF04\H
50NS=3F00\H
50NS=2F00\H
50NS=CF04\H
50NS=CFFF\H
50NS=CF03\H
50NS=3F00\H
50NS=2F00\H
50NS=CF00\H
50NS=CF0F\H
50NS=CF0E\H
50NS=3F00\H
50NS=5F00\H
50NS=7F00\H
```

(B) command data file

```
Q=XXXX\H
Q=0000\H
Q=0000\H
Q=6F00\H
Q=4F00\H
Q=2F00\H
Q=CF08\H
Q=CF0A\H
Q=CF07\H
Q=3F00\H
Q=2F00\H
Q=CF07\H
Q=CF02\H
Q=CF06\H
Q=3F00\H
Q=2F00\H
Q=CF03\H
Q=CF12\H
Q=CF11\H
Q=3F00\H
Q=5F00\H
Q=7F00\H
```

(C) simulation result file

Figure 4.3-3. ASCII data files used by the PIXMAN program in Example 4.3.1.

```
| SUIT bus waveform generation
| version 2.2, August 1995
| modified 5/96,10/96
| written by: Thomas H. Drayer
restart
VECTOR AA AA[15:0]
VECTOR CHAN CHAN[3:0]
VECTOR OFFSET OFFSET[7:0]
VECTOR QQ QQ[15:0]
RADIX HEX AA CHAN OFFSET QQ
|clock signal
wfm CLOCK @0NS=0 (25NS=0 25NS=1)*621
wfm RESET @0NS=1 @110NS=0
wfm CHAN @0NS=f\H
wfm OFFSET @0NS=3\H
|image data
wfm AA < offset_a.dat
break CLOCK 1 do (display QQ>offset_a.rim)
wave offset_a.wve  AA CHAN CLOCK OFFSET QQ RESET
|time for first few bytes
sim 800NS
|time for entire image
|sim 31050NS
```

Figure 4.3-4. Command file created by Example 4.3.1.

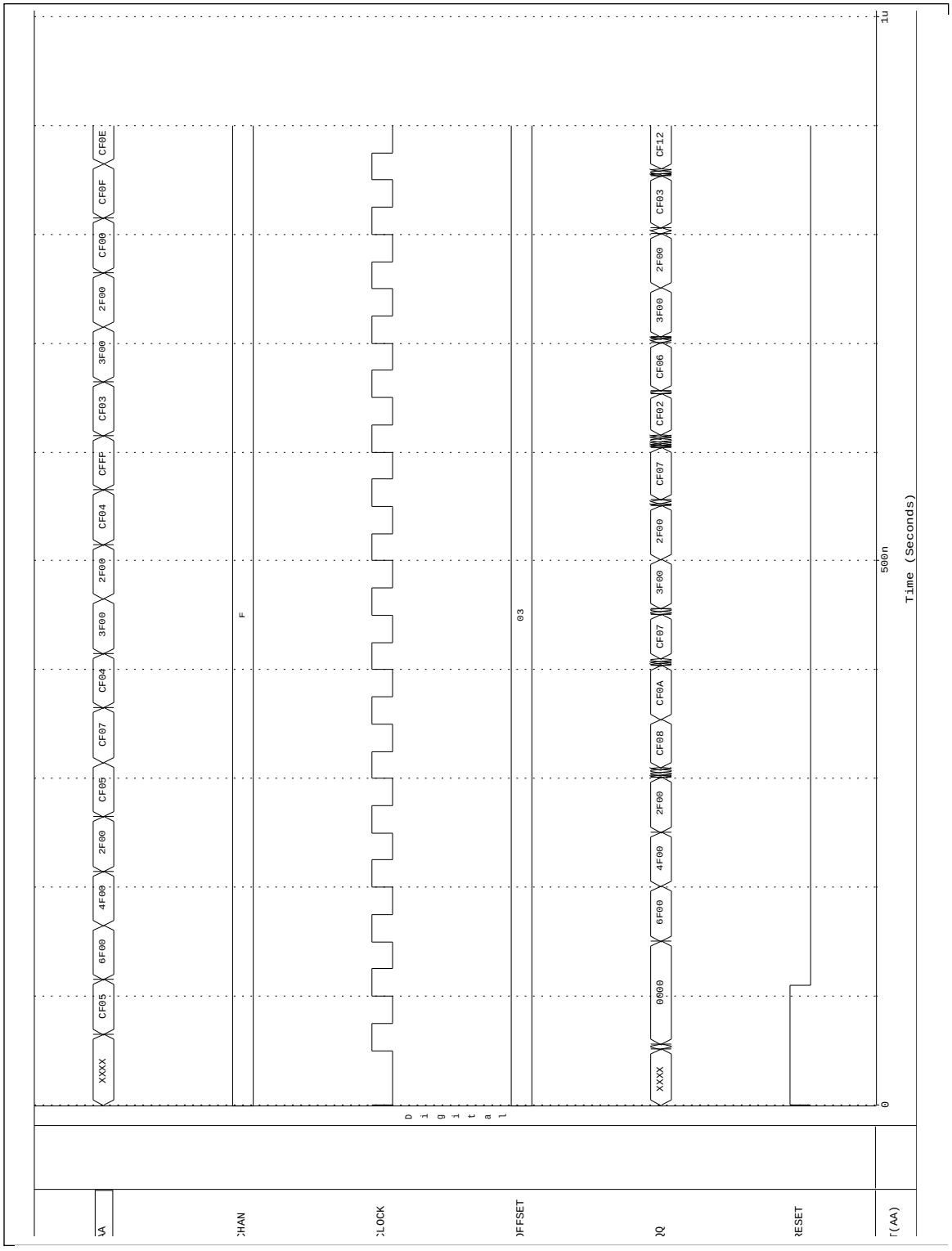


Figure 4.3-5. Simulation result waveforms for Example 4.3.1.

4.3.2 The MDBUG program

The MDBUG program has been created by the author to provide in-circuit emulation of image processing designs on the MORRPH-ISA board. In-circuit emulation supplies known test image data and collects output result data from MORRPH-ISA boards operating at the clock frequencies required for real-time operation. The programs associated with the MDBUG program and the file conventions used for the transfer of data are illustrated schematically in Figure 4.3-6.

All configuration information for an in-circuit emulation is contained in a single ASCII file, called an Architecture Configuration File or ACF file. All ACF files use the ".ACF" file extension. The ACF file is the primary output of the translation process.

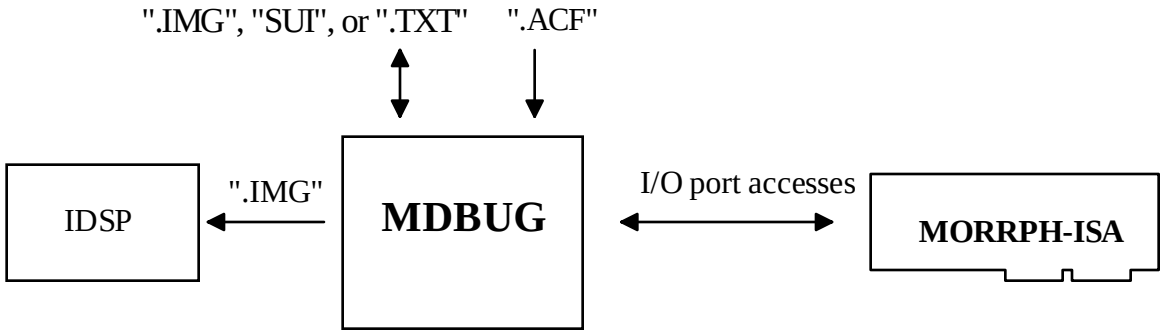


Figure 4.3-6. Input files, output files, and information flow of the MDBUG program.

ACF files are divided into three sections: the configuration, port, and bitstream sections. The configuration section of an ACF file defines the base I/O port address and the number, type, and location of FPGA chips in the MORRPH PE array. The location of modules used for transmitting and receiving emulation data and the default input and output test data file names are also defined in the configuration section. The port section of an ACF file defines the operand names, sizes, default values, and I/O port addresses used in the emulated image processing design. Finally, the bitstream section contains byte values used to program each of the FPGA columns in the MORRPH PE array. Byte values are stored in the ".MCS" file format defined by the Intel corporation [55]. A separate set of values is stored for each column, as the FPGAs in columns of the PE array are programmed in a daisy-chained manner. The format of ACF files has been created for this dissertation and is completely defined in Appendix I.

The MDBUG program is written in C and compiled for the DOS operating system. A Windows version of MDBUG is currently under development.

The MORRPH-ISA board is initialized using values defined in the configuration section of the specified ACF file (MORRPH.ACF is the default) when the MDBUG program begins execution. System memory is allocated for an input buffer that is loaded with test data. Similarly, memory is allocated for an output result buffer. Index pointers for both these buffers are initialized to the first location of each buffer. Port locations are identified and initialized with default values from the port section of the ACF file. The bitstream section of the ACF file is used to program the FPGAs of the MORRPH PE array. After these procedures, the MDBUG

program is ready to use input test data to emulate the real-time performance of an image processing design.

Eighteen command options are available for the MDBUG program. These command options are listed in Table 4.3-1. The functionality represented by these options is discussed in the following paragraphs.

Table 4.3-1. Command options of the MDBUG program.

Command Option Number	Option Description
Option 1	Re-initialize MORRPH configuration values
Option 2	Display MORRPH configuration values
Option 3	Display/modify port values
Option 4	Display design variables
Option 5	Transmit N SUIT values
Option 6	Transmit one SUIT value
Option 7	Transmit one SUIT line
Option 8	Transmit one SUIT frame
Option 9	Retrieve data from MORRPH board
Option 10	Display input and output buffer values
Option 11	Display input buffer image
Option 12	Display output buffer image
Option 13	Load/reload buffer from file
Option 14	Save input buffer to file
Option 15	Save output buffer to file
Option 16	Enable collection
Option 17	Disable collection
Option 18	Toggle verbose mode

The procedures defined above for program startup are executed when Option 1 is selected to re-initialize the MORRPH-ISA system configuration. Option 2 is used to display the current configuration values on the console.

Individual port locations that are implemented on the MORRPH-ISA board may be examined and modified using Option 3 of the MDBUG program. Current values of the registers are displayed in both hexadecimal and decimal format. Option 4 displays the operand names (bus and net names used in the ViewDraw schematic) of the current image processing design and their corresponding port addresses. These two options allow the operand variables of the emulated image processing design to be identified and modified.

The MDBUG program establishes two memory buffers, one for input and one for output. Input test data is accepted in any of the three formats (ELAS, binary SUIT, and ASCII SUIT file formats) defined in the previous section. The input buffer can be re-loaded with data at any time using Option 13. Similarly, data can be stored from the input or output buffer to a file in any of the three formats using Option 14 or Option 15, respectively. Data in the buffers can be printed to the console using Option 10 or displayed as an image with the IDSP program (Option 11 and Option 12). The IDSP program is a custom display program developed for internal use of the Spatial Data Analysis lab of Virginia Tech.

A special purpose image processing module called **ISA2SUIT**, located in the SUIT2 library, has been developed specifically for accepting input data from the host computer and transmitting this data on its output in the SUIT bus format. Test image data is supplied to the **ISA2SUIT** module on the MORRPH-ISA board using I/O port write cycles of the ISA bus. This

data could be manually entered using Option 6. However, this becomes a tedious task for even for small amounts of input test data. Options 5-8 of the MDBUG program automate this process by transmitting an arbitrary number of values, a single value, a single line, or a single frame of input test data on the MORRPH-ISA board in the SUIT bus format.

In a similar manner, a second special purpose module called **SUIT2ISA** has been created to collect image and result data from its input SUIT bus. The MDBUG program retrieves this data from the **SUIT2ISA** module and loads the data into its output buffer. Data is retrieved from the **SUIT2ISA** module automatically when Options 5-8 are used to transmit test data and manually when Option 9 is selected.

Both the **ISA2SUIT** and **SUIT2ISA** modules must be included in the image processing design that is loaded onto the MORRPH-ISA board to allow the transmission and collection of test image and result data for in-circuit emulation. The location of these modules in the MORRPH PE array for a compiled image processing design is defined in the configuration section of the ACF file.

The final three options of the MDBUG program enable/disable collection of the **SUIT2ISA** module and toggle a "verbose" operating mode. Collection is enabled and disabled automatically by the transmission Options 5-8 or manually with Option 16 and Option 17. The verbose mode of operation ignores most default values and repeatedly prompts the user for additional input.

Data cannot be supplied to or retrieved from the MORRPH-ISA board at the high data rates of some real-time applications. This is because of the low bandwidth provided by I/O port

read and write cycles of the ISA bus. Although data is not transmitted at the same input pixel rates as many real-time image processing applications, the systolic operation of the translated image processing design is emulated at the clock frequency required for real-time operation.

Improved data buffering and control of the **SUIT2ISA** and **ISA2SUIT** modules would allow data to be transmitted and collected at real-time data rates for short periods of time. This method provides higher confidence of the functionality and performance of an image processing design and is left for future work.

This design methodology relies heavily on verification of both image processing modules and designs. Since new specialized processing units are created for each desired task, there is a very real possibility that modules will not function properly or at the desired performance level. The interaction of individually created modules in an image processing design may also introduce errors in the output results. These new automated simulation and in-circuit emulation methods allow designs to be easily verified with detailed feedback of their functionality and performance.

Two methods are used to provide input data and collect result data from an in-circuit emulation. First, input modules (**ISA2SUIT**) and output modules (**SUIT2ISA**) may be added to the image processing design specifically to interface with the ISA bus. This method is typically used for verifying individual modules when additional resources for the **SUIT2ISA** and **ISA2SUIT** modules are readily available. A schematic for in-circuit emulation of the **OFFSET_A** module defined in previous examples is shown in Figure 4.3-7.

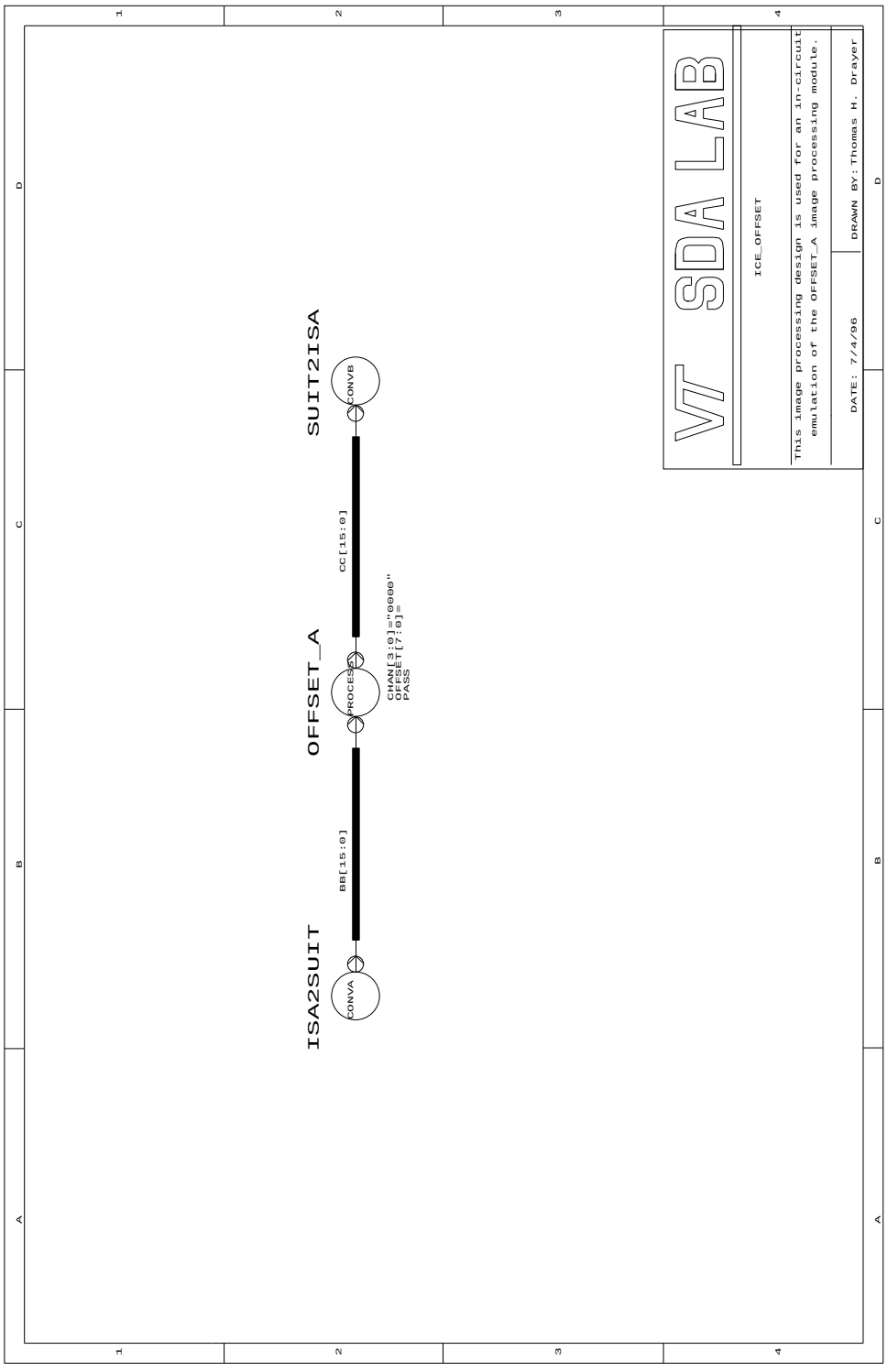


Figure 4.3-7. Schematic for in-circuit emulation of the **OFFSET_A** module.

In the second method, one MORRPH board can be used to supply and collect test data from a second MORRPH board. This method is commonly used for in-circuit emulation of complete image processing designs with several low-level image processing modules. Communication between MORRPH boards is accomplished with an appropriate bus standard, such as the ZEE bus format defined for this purpose in Section 4.2.3. Modules have been created to convert input (**ZEE2SUIT**) and output busses (**SUIT2ZEE**) between the SUIT and ZEE bus formats. The schematic created for Example 4.2.5 and shown in Figure 4.2-9 is appropriate for this method of in-circuit emulation. One additional schematic, shown in Figure 4.3-8, is required to program the MORRPH board that supplies input data and collects output image and result data.

The PIXMAN and MDBUG programs provide verification of image processing designs created with the design entry process. PIXMAN automates the process of simulation before image processing designs are compiled. After compilation, the MDBUG program verifies correct operation of the image processing design using in-circuit emulation. Together, these two programs provide a high confidence that the image processing design will operate with the desired functionality and performance.

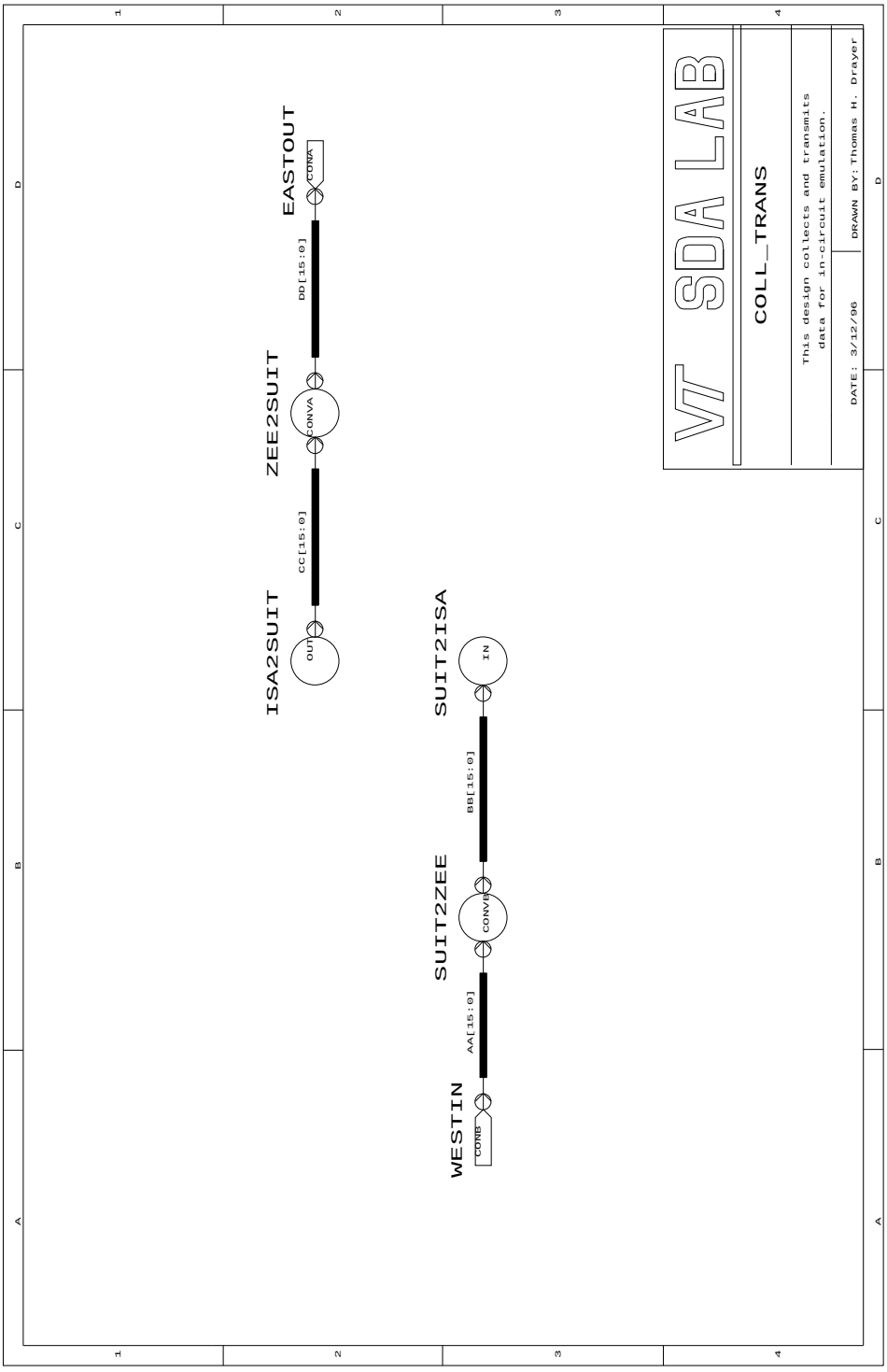


Figure 4.3-8. Schematic for supplying and collecting data of in-circuit emulation.

4.4 Translation

After a design has been created and partially verified with simulation, it is translated into the resources of the target architecture. This translation requires selecting and interconnecting discrete hardware resources available on the target architecture to create a circuit with the same functionality as the image processing design. Design translation is a highly automated procedure when the entire design can be compiled into a single FPGA chip. Commercially available software tools, such as the XACT software from the Xilinx Corporation, translate an input schematic or VHDL description of the circuit into the CLB, IOB, and interconnect resources of a target FPGA chip. Unfortunately, the existing software cannot automatically translate a single design into the logic of several FPGA chips. This capability is required for automatic translation of image processing designs into multiple-FPGA architectures, such as MORRPH-ISA.

Logic elements of an image processing design must be partitioned into the distinct groups that are implemented on individual FPGA chips when the logic elements cannot be translated into the resources of a single FPGA chip. This process of assigning logic elements to groups is called *network partitioning*. An appropriate partition must not exceed the logic, I/O, and interconnect resources of the destination FPGA-based hardware architecture.

Nets of a partitioned image processing design may interconnect logic components that have been placed in different FPGAs. These nets must be allocated to existing interconnection resources of the destination architecture. The assignment of these nets to the existing interconnection resources of the destination architecture is called *global routing*.

Since the feasibility of global routing is affected by each partition, it is logical to attempt to create a single procedure that performs both network partitioning and global routing. However, both network partitioning and global routing represent NP-hard problems. This complexity severely limits the possibility of combining the two algorithms and solving the two problems simultaneously. The approach taken by this development system is to implement both algorithms separately.

A new algorithm for network partitioning has been developed for this dissertation, and is described in detail in the next section. The following section describes a simple algorithm developed for global routing, and the final section describes the implementation of the integrated translation software programs created for this dissertation.

4.4.1 Network Partitioning

Network partitioning has received substantial attention in recent years. It is a prevalent problem associated with several common design automation tasks, such as the division of logic between FPGAs, PLDs, Multi-Chip Modules (MCMs), and floorplanning for VLSI layouts.

The image processing design to be partitioned can be represented by a hypergraph $H(V,E)$, where $V = \{ v_i | i=1,2,...n \}$ is the node set and $E = \{e_j | j=1,2,...m\}$ is the net set. Each net e_j in the net set of a hypergraph is a set of two or more elements from the node set V, while a each net of a graph is a set of exactly two elements from the node set. The discrete elements of a network may be single logic components or groups of logic components in a hierarchical design,

where each discrete element is represented by a node v_i in the node set V of the hypergraph. Each net e_j of the net set is a subset of V, as defined by the interconnections between discrete elements established by the corresponding net in the image processing design.

A K-way network partition $P(V)$ of the n elements of the circuit assigns each of the nodes in the node set into one of K subsets V_b such that $V = V_1 \cup V_2 \cup V_3 \cdots V_k$ and $V_1 \cap V_2 \cap V_3 \cdots V_k = \phi$. The K subsets of vertices are called the *groups* of a partition. The *cutset* $C(P)$ of a partition P is the minimal set of nets that create the independent hypergraphs $V_1, V_2, \ldots V_k$ when removed from the hypergraph $H(V,E)$. Stated alternatively, the cutset is the set of all nets that interconnect two or more nodes that are in separate groups of a particular partition. The number of nets in the cutset is defined as the size of the cutset $|C(P)|$.

If two elements of a network (nets, nodes, or groups) are interconnected, they are said to be *adjacent*. Most definitions of adjacency are obvious, however some of the less obvious definitions of adjacency follow. A node is adjacent to a group if it is included in the group. A net is adjacent to a group if is adjacent to at least one node in the group.

The *order* (cardinality) of a net $|e_j|$ is defined as the number of nodes interconnected by a net. Similarly, the order of a node $|v_i|$ is the number of nets that are adjacent to the node. The *span* of a graph element is defined as the set of all other graph elements which are adjacent to this element. The set of nodes which are adjacent to a net is defined as node span, or $span_V(e_i)$, of the net. Similarly, the set of groups which are adjacent to a net is defined as the group span, or $span_G(e_i)$, of the net. Note that the group span of a net may change for different partitions, while the node span of a net remains constant for a network.

Network partitioning divides the discrete elements of a circuit into several groups, to satisfy a defined set of constraints or to minimize a cost function. The constraints or cost function may involve the size of the cutset, the size of the groups, or other concerns. A heuristic cost function equal to the number of nets in the cutset is commonly used for circuit partitioning. When network partitioning is used to divide the logic of an image processing design into individual FPGA chips, these nets are implemented with the limited interconnection nets between FPGA chips on the printed circuit board of the destination hardware architecture.

All edges in a graph have order two ($|e_j|=2$), while nets in a hypergraph may have any order greater than one ($|e_j|>1$). This difference distinguishes graph partitioning from network partitioning. Therefore, K-way network partitioning assigns each of the n nodes of a circuit to one of the K groups of the partition. *Bipartitioning* is the special case of 2-way partitioning.

Graph and network partitioning algorithms have been divided into two distinct classes: constructive and refinement algorithms [56]. Constructive algorithms create a partition from the characteristics of the network. In contrast, refinement algorithms begin with an initial partition, and repeatedly attempt to improve the current partition.

A third class of algorithms, combinational algorithms, combine two or more algorithms from the previously defined classes. One combinational method uses the output of a constructive algorithm as the initial state for a refinement algorithm. This combination of algorithms is done either to create a better initial state (as in [57]) or to reduce the dimension of the partitioning problem [57, 58]. Another combinational method iterates between two different refinement algorithms, using the result of the previous algorithm as the initial state for the next [59, 60]. This

combinational method terminates when either algorithm does not provide an improvement over its initial state.

Three main types of constructive algorithms exist. First, clustering algorithms can be used to generate a partition as in [57, 61, 62]. Clustering algorithms generate partitions using heuristics that examine the local connectivity of a network. However, these bottom-up approaches lack global knowledge of the network being partitioned. A top-down clustering approach is presented in [63] that uses recursive bipartitioning. Errors that occur early in this type of algorithm can severely reduce the performance of the algorithm. Spectral methods defined in [64], [65], and [66] represent the second type of constructive algorithms. These methods construct matrices (i.e., adjacency, degree, and Laplacian matrices) that represent the connectivity of an input graph. The eigenvalues and eigenvectors of these matrices are used to generate a partition. The disadvantage of these approaches is that operations are performed on several nxn matrices, where n is the number of nodes to be partitioned, and an nxn matrix can become very large. Finally, Ford and Fulkerson [67] used a max-flow-min-cut algorithm to find the optimum partition in polynomial time. However, the optimal partition is found without respect to the size of the groups, and therefore the max-flow-min-cut algorithms typically create partitions in which the size of the groups are unbalanced [59]. For partitioning onto fixed FPGA architectures, the partition sizes are constrained by the amount of available resources in each FPGA chip, making the partition sizes an important constraint of the partitioning algorithm.

Refinement algorithms (e.g., iterative improvement algorithms) represent the second class of graph and network partitioning algorithms. Initial research began with the graph bipartitioning

problem. Kernighan and Lin (the K-L algorithm) provided an algorithm for the graph bipartitioning problem with complexity $O(n^2 \log n)$ in [68], and Schweikert and Kernighan extended the algorithm to accommodate networks in [69]. The K-L algorithm swaps two nodes from an initial partition P_1, one node from each of the two subsets V_1 and V_2, to create a new partition P_2. The two nodes are selected to provide the maximum decrease in a cost function, where the cost function is defined as the size of the cutset $|C(P_i)|$. A pass of the algorithm continues until all nodes have been moved, at partition $P_{n/2}$. Since all nodes have been swapped, the final partition is equal to the initial partition ($P_1 = P_{n/2}$). If a lower cost was found at one of the intermediate partitions P_i the steps of the algorithm are retraced to create partition P_i, otherwise the algorithm terminates. Fiduccia and Mattheyses (the F-M algorithm) refined the K-L algorithm, using techniques such as moving a single node at a time and sorting the cutset improvement values for each node in [70]. These improvements reduced the time complexity of the basic algorithm to $O(n \log n)$. The concept of a "level gain" was introduced by Krishnamurthy in [71]. This algorithm enhancement modified the simple gain metric associated with moving a node, such that gains from future moves could affect the cost of the current move. Sanchis expanded the work done by Krishnamurthy to K-way partitioning in [72]. Sanchis' work included modifying the level gains and state selection to partition into K groups, instead of two.

This previous work establishes the foundation for a new K-way partitioning algorithm developed by the author for assigning the logic of image processing designs to individual FPGA chips of the FPGA-based destination hardware architecture. The new refinement algorithm uses heuristic search techniques to find a suitable network partition. Two important procedures define

the functionality of most heuristic search algorithms, the state search algorithm and the heuristic cost function. These two topics are discussed in the next two sections.

4.4.1.1 State Search Algorithm

The transition of states in many heuristic search problems is restricted by a set of rules [73]. These rules define the possible state transitions. However, there are no restrictions on the possible transitions between states of the circuit partitioning problem. Furthermore, it is impractical to evaluate all possible state transitions at each step of a refinement algorithm. A subset of the possible state transitions are evaluated to identify the movement of a single node or several nodes that will create a new partition with lower cost at each step of the refinement algorithm. The selection of state transitions is critical to the performance of the algorithm.

All methods defined above select the state transitions to be evaluated based entirely upon the location of nodes in the partition. However, the interconnection of the network influences the selection of state transitions when nets are used to motivate the movement of nodes. Instead of simply selecting a node to move because of its location in the partition, a net is selected to be removed from the cutset. This net is removed from the cutset by moving the nodes adjacent to the net into the same group. Note that although the actual move is a transition of nodes, the selection of moves to be evaluated is motivated by identifying nets to be removed from the cutset. This method of identifying state transitions for evaluation is based on both the

interconnections of the network and node locations, and therefore creates an algorithm that produces better partitions.

An algorithm that analyzed moves based on the nets of a circuit was first proposed by Wei and Cheng in [59]. In their algorithm, two sets are introduced for each net-group combination. The *critical set* of a particular net e_j for group V_b is defined as the set of all nodes adjacent to e_j and contained within the same group V_b. The *complementary critical set* of a particular net e_j for group V_b is the set of all nodes that are adjacent to the net e_j but are not in the current group V_b. Using these two sets, two types of moves are considered. The first type moves all nodes in the complementary critical set into the current group. The second type moves the nodes of the critical set out of the current group and into one of the other groups.

A combinational algorithm is created by Wei and Cheng in [59] that iterates between the classical F-M algorithm and a new algorithm based on the removal of nets, using the output partition of one as the input to the next. The algorithm is terminated when either of the algorithms provides no further improvement. An implementation of the new algorithm by itself and comparisons with F-M algorithm are not addressed in [59], probably because a version of their new algorithm based exclusively on the movement of nets reportedly executed 12 times slower than the traditional F-M algorithm.

This dissertation creates a new method of state selection based on the removal of nets in the cutset of the current partition. Only nets in the cutset are considered for removal at any step of the algorithm. The first type of move defined above by Wei and Cheng, moving the complimentary critical set into the current group, removes the selected net from the cutset. The

second type of move will only remove a net from the cutset when the size of the group span of the current net is equal to two ($|span_G(e_j)| = 2$). However, this move is covered by the first type of move. Therefore, the algorithm created for this dissertation only evaluates new partitions created by movement of complimentary critical set into the current group for nets of the cutset. This new method of selecting moves for evaluation is the basis of a new network partitioning algorithm called the *net-move* algorithm.

> **Net-Move Algorithm**: For each iteration, all nets are selected sequentially to become the "current net" after the initial selection of a random starting net. Each current net is checked to determine if it is adjacent to nodes in more than one group of the current partition, when $|span_G(e_i)| > 1$ (e.g., if it is in the cutset). If so, each of the groups are sequentially selected to become the "current group". When the current net is adjacent to the current group, one move is evaluated for this net-group combination. The cost of the new partition created by moving all the nodes from the complimentary critical set of the current net into the current group is calculated. For each net in the cutset, if any one of the evaluated moves provides a new partition of lower cost, then the move that provides the lowest cost partition is taken. No moves are taken if all evaluated moves result in partitions with higher costs. For a complete iteration, all nets are checked in turn. The algorithm terminates when no moves were taken during the previous iteration.

A new C program, called NETPART, has been created to implement the net-move algorithm. Since the size of the cutset is the predominant cost metric in current literature, this cost function is used until the next section. However, using only the number of nets in the cutset as the cost function may lead to the trivial solution in which all nodes of the circuit are placed in one group.

Therefore, the cost function is modified to balance the groups. The size of each group is constrained by the cost function to be within 50% above or below the balanced group size of K/n nodes in each group.

The complexity of the algorithm may be affected by its software implementation, without a more precise definition than the one provided above. The NETPART program provides an implementation of the net-move algorithm with the following complexity:

$$O(m\eta[K\overline{|e|} + \overline{|span_G(e)|}\{(n + \overline{|e|} + m(2K + \overline{|e|})) + (m + 2K + n)\} + \lambda m(2K + \overline{|e|})]) \quad (4\text{-}1)$$

where η is the percentage of all nets that are in the cutset and λ is the percentage of nets in the cutset that provide at least one partition with lower cost. The equation above can be simplified as shown in the following:

$$O(m[K + \overline{|span_G(e)|}\{n + mK + m + K + n\} + mK]) = O(m[K + \overline{|span_G(e)|}\{n + mK\} + mK]) \quad (4\text{-}2)$$

Two assumptions allow Equation 4-2 to be reduced further. First, the average group span of a net is approximately proportional to the number of groups, and can be replaced by K. Second, it is difficult to increase the number of nodes in most practical circuits without also increasing the number of nets, therefore n can be assumed to be proportional to m. The converse is also true. These assumptions yield the following equation:

$$O(m[K + \overline{|span_G(e)|}\{n + mK\} + mK]) \cong O(m[K + K\{m + mK\} \cdot = O(m^2K^2) \quad (4\text{-}3)$$

The complexity of the algorithm therefore increases by the square of both the number of nets in the network and the square of the number of groups in the partition. For bipartitioning, the net-move algorithm's complexity of $O(n^2)$ is higher than the $O(n\log n)$ complexity of F-M algorithm. However, the F-M algorithm sorts gain values and it may be possible to do an analogous sorting with the net-move algorithm. Any investigation into the use of sorting is left for future work.

To determine the merit of the net-move algorithm, an analogous *node-move* algorithm is defined by this dissertation. Comparisons between these algorithms will be used to justify the use of the net-move algorithm.

> **Node-Move Algorithm:** In each iteration, all nodes are selected sequentially to be the "current node" after the initial selection of a random starting node. For each current node, all groups are sequentially selected to be the "current group". The cost of the partition created by moving the current node into the current group is determined. For each node in the node set, if any of the evaluated moves provide a new partition of lower cost, then the move that provides the lowest cost partition is taken. No moves are taken if all evaluated moves result in partitions with higher costs. For a complete iteration, moves for all nodes are sequentially evaluated. The algorithm terminates when no moves were taken during the previous iteration.

Minor modifications to the NETPART program are used to create a new program, called NODEPART, that implements the node-move algorithm. This new program uses the same cost function previously defined for the net-move algorithm.

The NODEPART program provides an implementation of the node-move algorithm with the following complexity:

$$O(n[K\overline{|e|} + \overline{|span_G(e)|}\{(n + m(2K + \overline{|e|})) + (m + 2K + n)\} + \xi m(2K + \overline{|e|})]), \qquad (4\text{-}4)$$

where ξ is the percentage of nodes that provide at least one partition with lower cost. Using the assumptions made for the net-move algorithm, the computational complexity of the node-move algorithm can be simplified as follows:

$$\begin{aligned} O(n[K + \overline{|span_G(e)|}\{(n + mK) + (m + K + n)\} + mK]) \\ \cong O(n^2K^2) \cong O(m^2K^2) \end{aligned} \qquad (4\text{-}5)$$

Equation 4-5 shows that the computational complexity of the node-move algorithm is the same as the complexity of the net-move algorithm.

One final algorithm has been implemented. This third program, called FMPART, implements the classic F-M algorithm, without gain sorting, to provide a comparison with one of the current refinement partitioning algorithms defined at the start of this section. The classic F-M algorithm is slightly modified for K-way partitioning, since the original algorithm is only defined for bipartitioning.

These three programs provide a method for evaluating the performance of the algorithms. Since all three programs use the same data structures, cost metric, and cell movement subroutines, these routines also provide an accurate comparison of the speed of execution for

each of the algorithms for circuits of the expected size. Performance of larger circuits must be predicted using the algorithm complexity.

These three programs have been tested on eight benchmark circuits provided by the Microelectronics Research Center in Research Triangle Park, North Carolina. Five of these are combinational circuits and the remaining three are sequential circuits. The average resultant size of the cutset for 20 bipartitioning trial runs on each circuit is graphically shown in Figure 4.4-1. Cutset size results for 3-way and 4-way partitioning are presented in Figure 4.4-2 and Figure 4.4-3. The average processing time for the 20 bipartitioning trial runs using each circuit is shown in Figure 4.4-4. Processing time results for 3-way and 4-way partitioning are shown in Figure 4.4-5 and Figure 4.4-6.

The data in the following graphs indicates that the node-move algorithm increases the cutset size by an average of 24.33% over the net-move algorithm, while requiring 88.3% more computation time. The F-M algorithm cutset sizes are comparable to the node-move model, but because gain values are not sorted by this implementation of the F-M algorithm it requires significantly more time for computation than either the node-move or net-move algorithms. For all circuits, the net-move algorithm provides better partitioning results while requiring less computation time. Therefore, the net-move model is the basis for the partitioning work that follows.

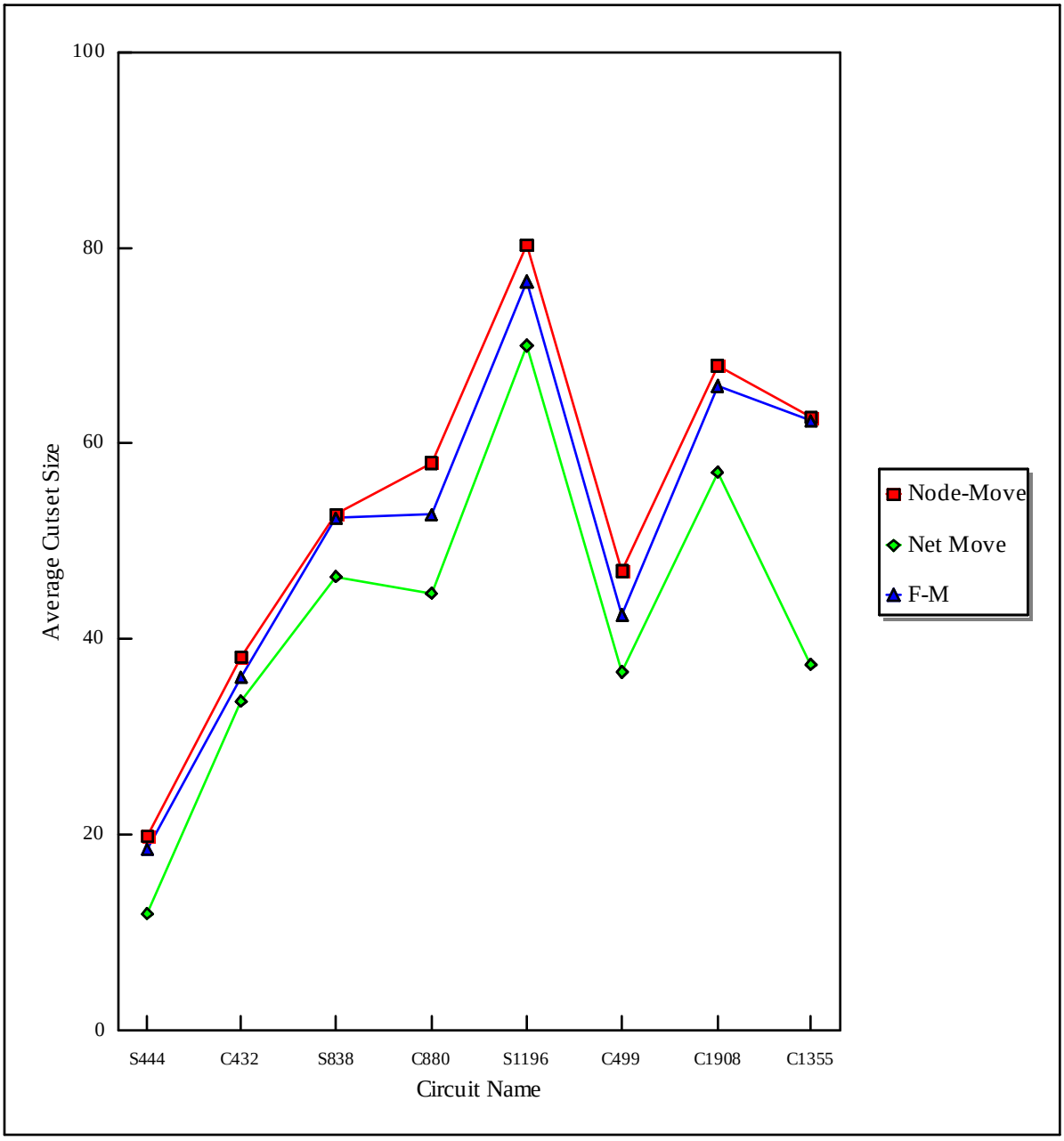


Figure 4.4-1. Average cutset size for 20 trials of bipartitioning.

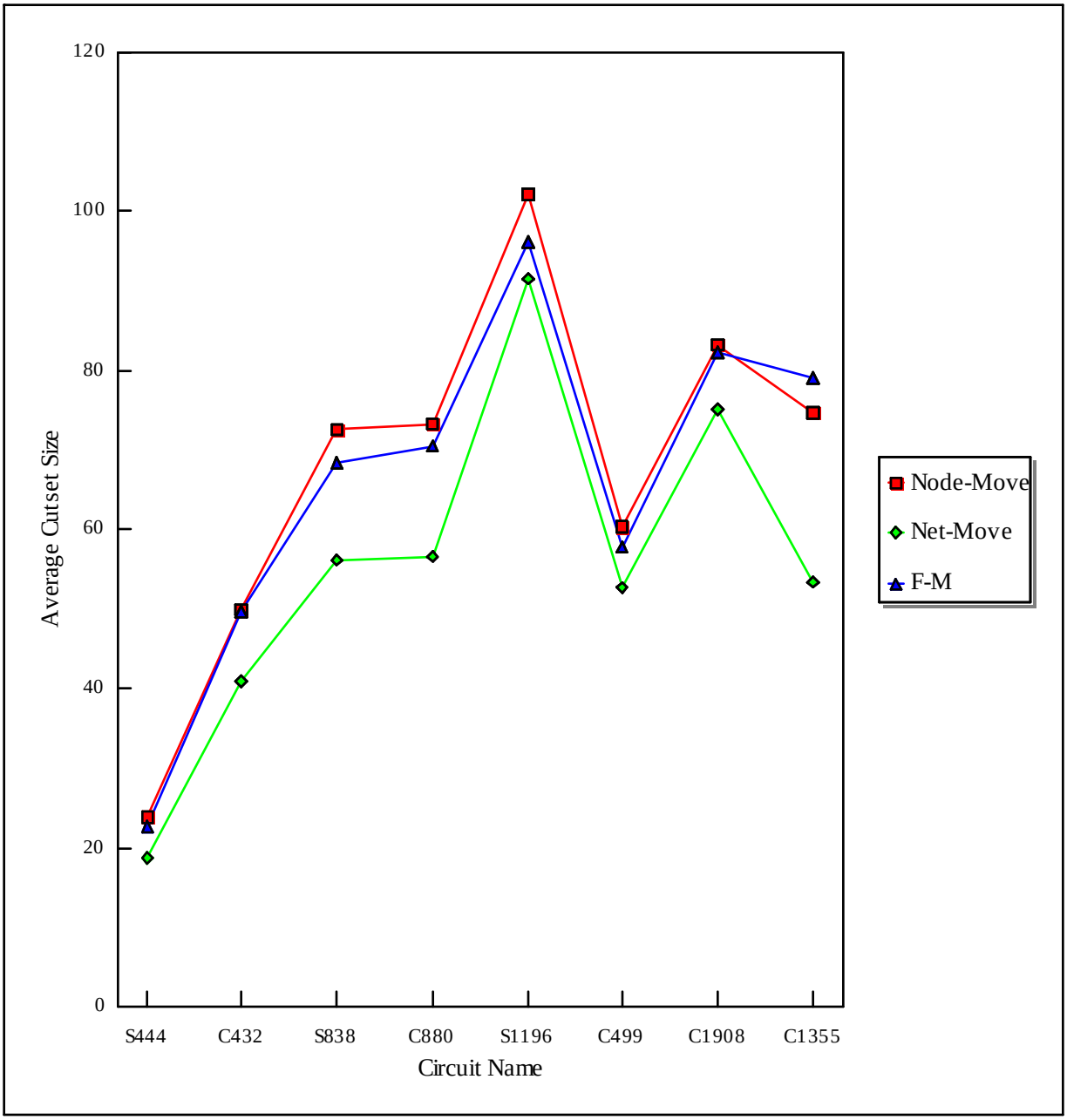


Figure 4.4-2. Average cutset size for 20 trials of 3-way partitioning.

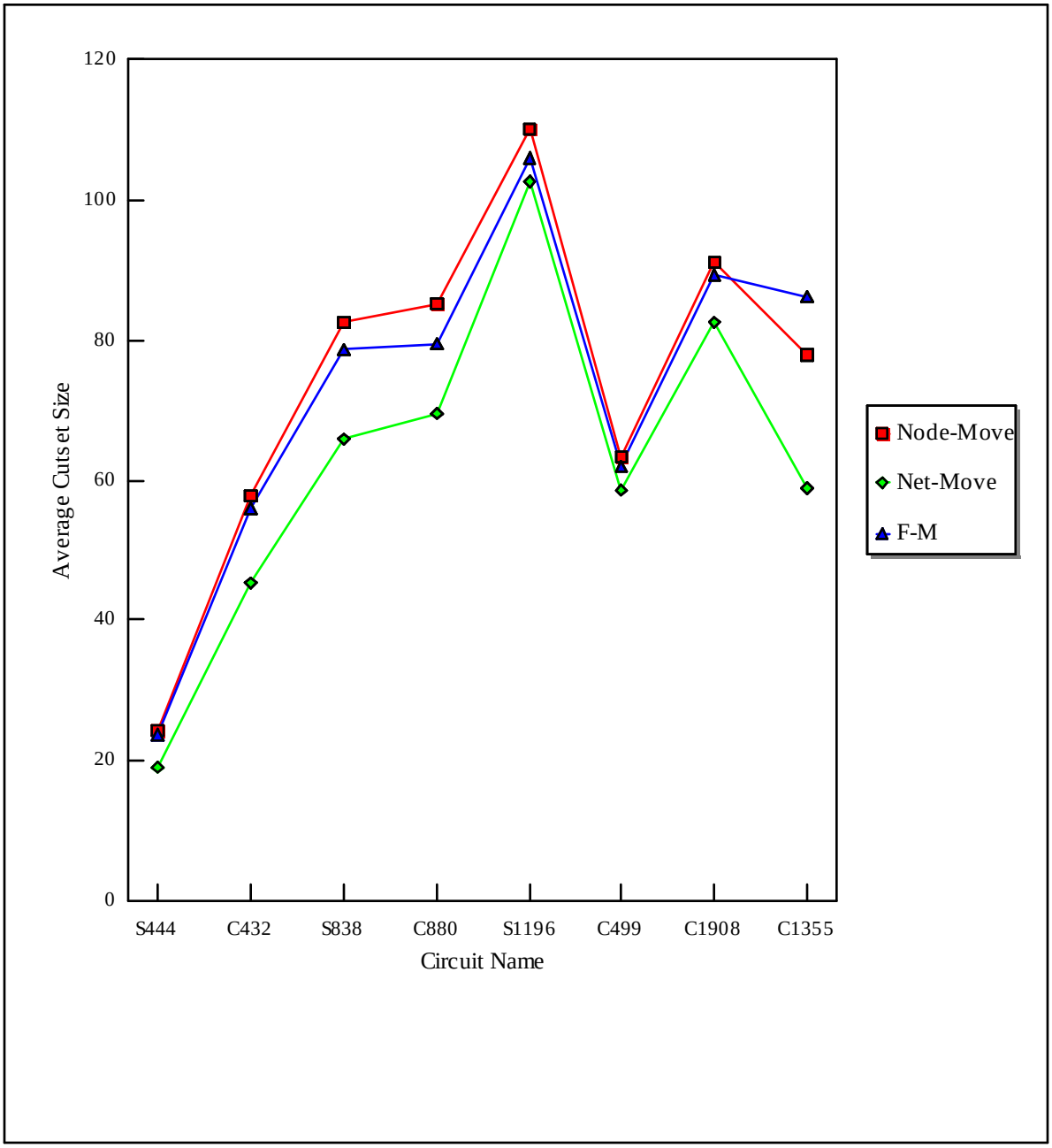


Figure 4.4-3. Average cutset size for 20 trials of 4-way partitioning.

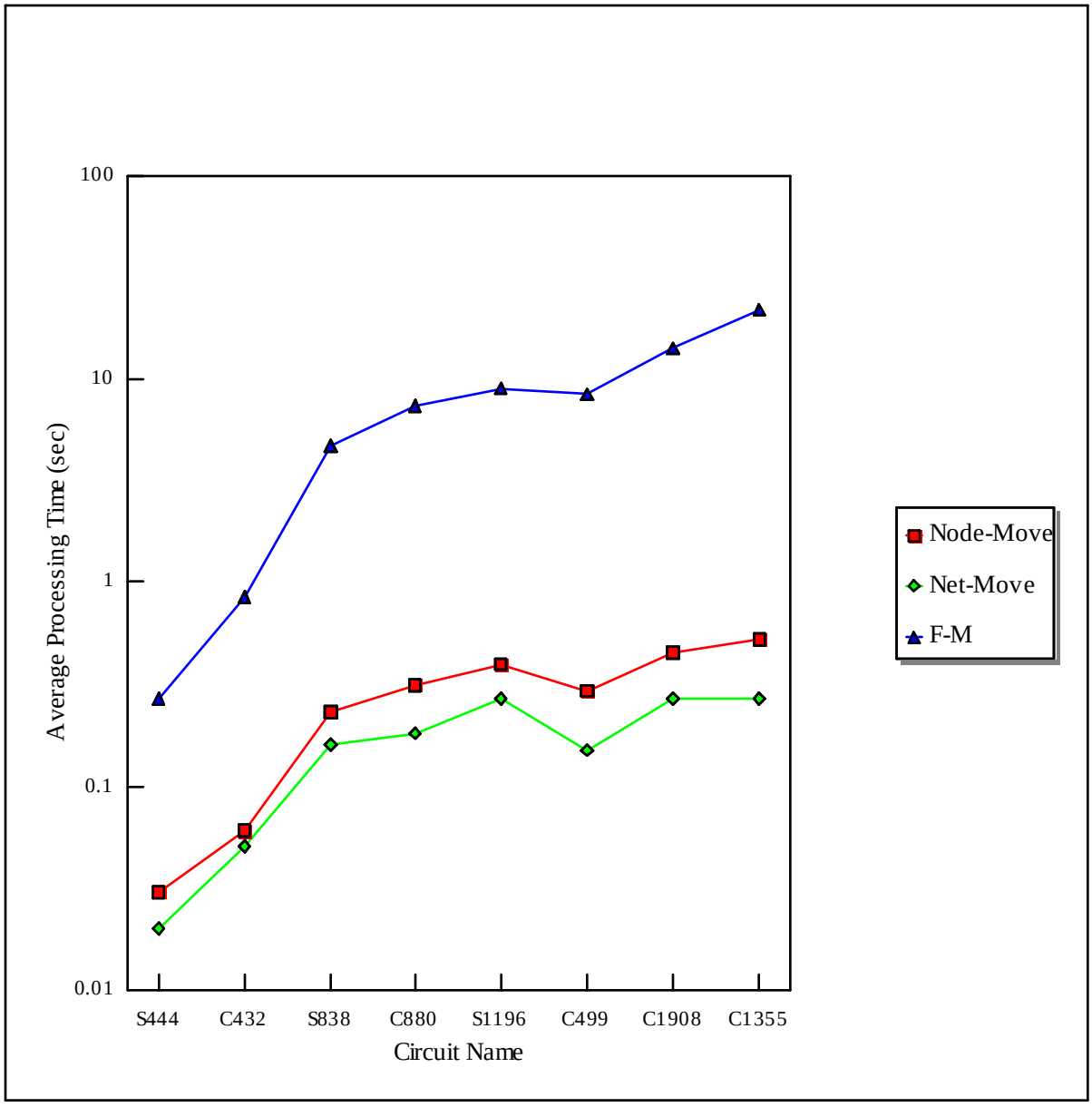


Figure 4.4-4. Average processing time for 20 trials of bipartitioning.

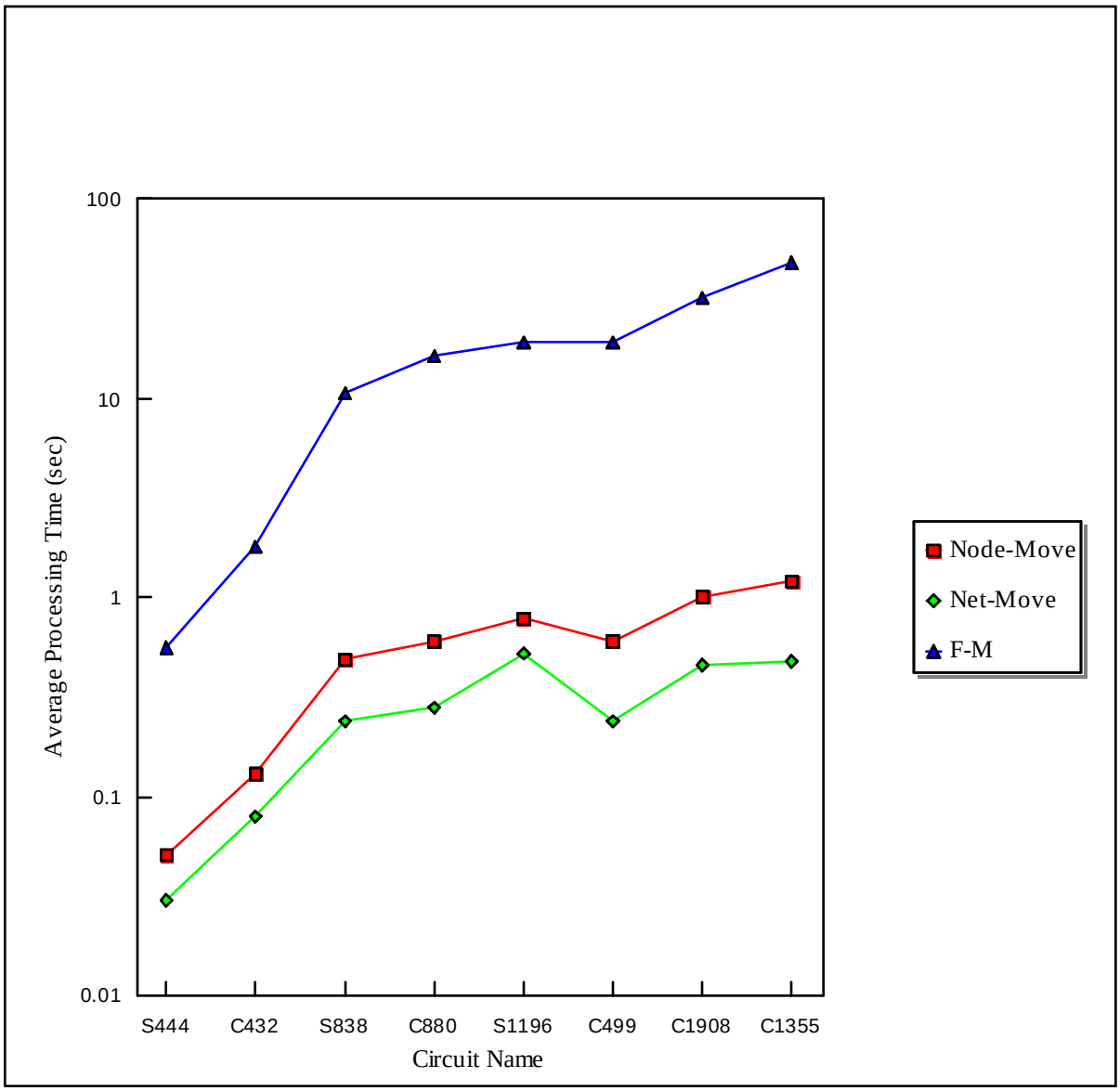


Figure 4.4-5. Average processing time for 20 trials of 3-way partitioning.

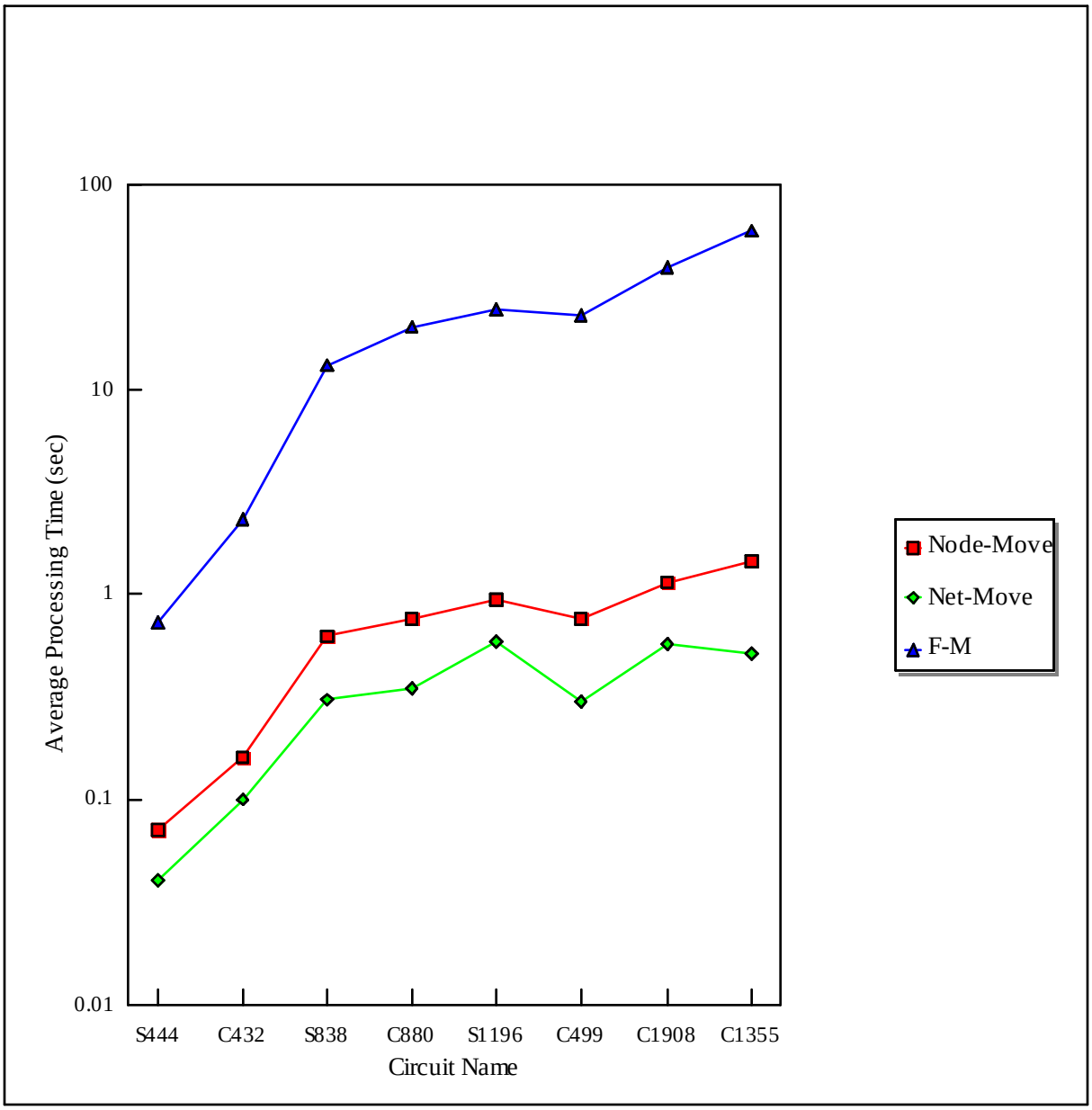


Figure 4.4-6. Average processing time for 20 trials of 4-way partitioning.

The net-move(node-move) algorithm may be significantly enhanced by considering the effect of future moves on the size of the cutset. Instead of using a look-ahead strategy to determine what benefit may be obtained from future moves, as in [72], multiple steps are analyzed for each net-group(node-group) combination. This new type of state search algorithm is defined as the multi-step algorithm in the following:

> **Multi-Step Algorithm:** Step 0 of the multi-step algorithm corresponds to any move taken by the net-move(node-move) algorithm defined above. However, this movement of nodes may add new nets in the cutset. In step 1 of the multi-step algorithm, all the nodes adjacent to the new nets added the cutset are moved to the current group, and the cost of the new partition is evaluated. The goal is to remove these new nets from the cutset. This step may again create new nets in the cutset. To remove these new nets, the procedure of step 1 is repeated for step 2, and so on for a fixed number of steps. This number of steps is defined as the S, the maximum number of steps or depth-of-search. For each net in the cutset(node), if any of the evaluated moves provides a new partition of lower cost, then the move that provides the lowest cost partition is taken. No moves are taken if all evaluated moves result in partitions with higher cost. For a complete iteration all nets(nodes) are checked in turn. The algorithm terminates when no moves were taken in the previous iteration.

A move in the net-move(node-move) algorithm was completely defined by each net-group(node-group) combination. In the new multi-step algorithm, a move is completely defined by the current net(node), current group, and the number of steps.

The additional steps of the multi-step algorithm affect the complexity of the algorithm. The current implementation of the multi-step algorithm has the following complexity:

$$O(mn[K\overline{|e|} + \overline{|span_G(e)|}\{(n + \overline{|e|} + m(2K + \overline{|e|})) + (m + 2K + n) + m \\ +S\{m\overline{|e|}^S(n + \overline{|e|} + m(2K + \overline{|e|})) + (m + 2K + n) + m\}\} + \lambda m(2K + \overline{|e|}) \quad (4\text{-}6)$$

Equation 4-6 can be simplified to the following:

$$O(m[K + \overline{|span_G(e)|}\{(n + mK) + (m + K + n) + m \\ +S\{m\overline{|e|}^S(n + mK) + (m + K + n) + m\}\} + \lambda mK]) \quad (4\text{-}7)$$

Applying the assumptions used previously, the complexity is further reduced to the following:

$$O(m[K\{(mK) + S\{m\overline{|e|}^S(mK)\}\}]) = O(m^2K^2 + m^3K^2S\overline{|e|}^S) \\ = O(m^3K^2S\overline{|e|}^S) \quad (4\text{-}8)$$

The equation above includes a term that is an exponential function of the maximum number of steps S. Therefore, even small increases in the depth-of-search will dramatically increase the processing time of the multi-step algorithm.

The program NETPART is modified to implement the net-move algorithm with the multi-step algorithm defined above. This new program is used to partition the eight benchmark circuits used previously. The percentage reduction in the average cutset size of 20 partitioning trials, using the multi-step algorithm instead of the single-step algorithm is presented in Figure 4.4-7. Figure 4.4-8 illustrates the average percent increase in processing time required by the additional steps of the multi-step algorithm.

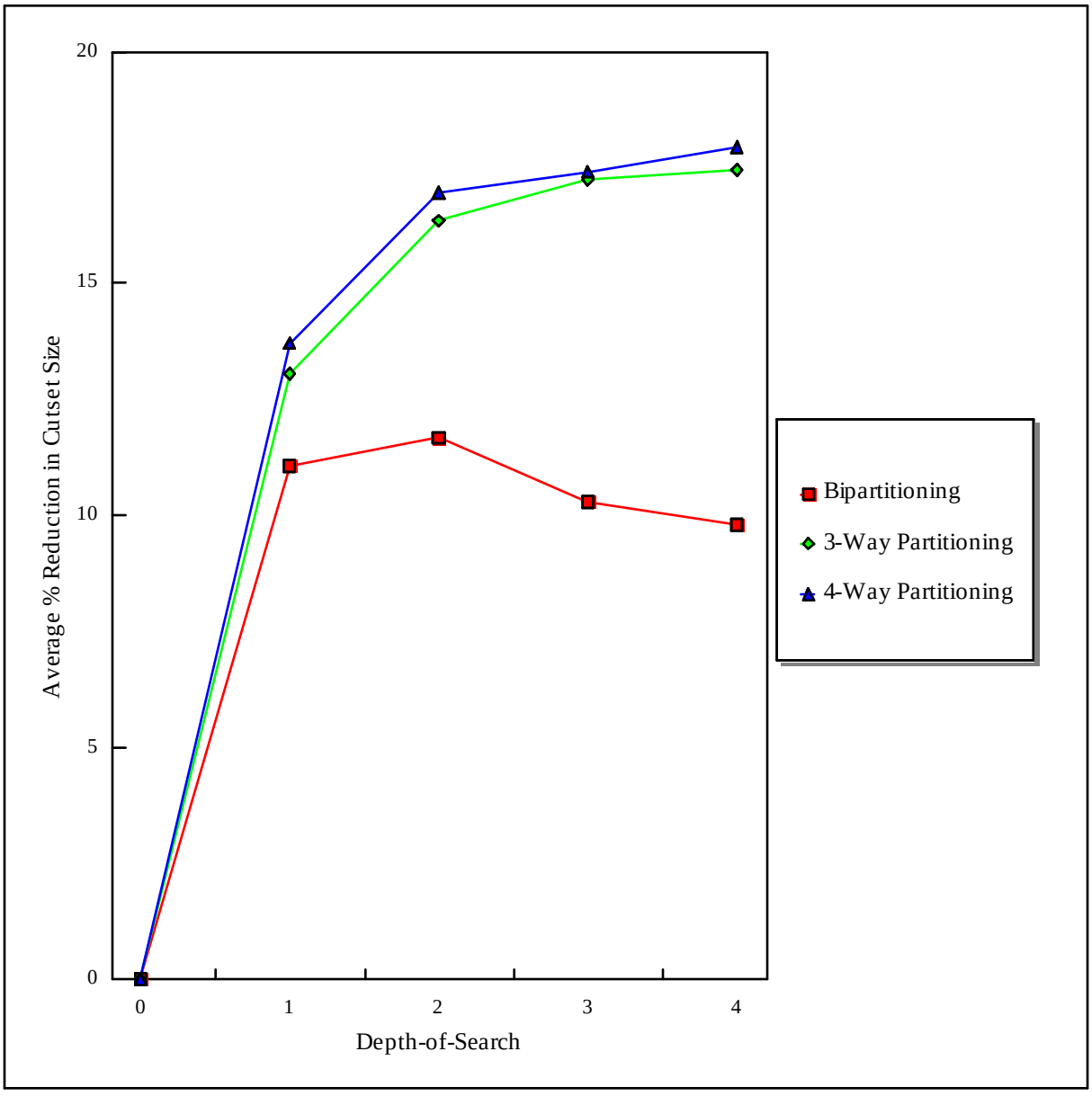


Figure 4.4-7. Percent reduction in cutset size of the multi-step algorithm.

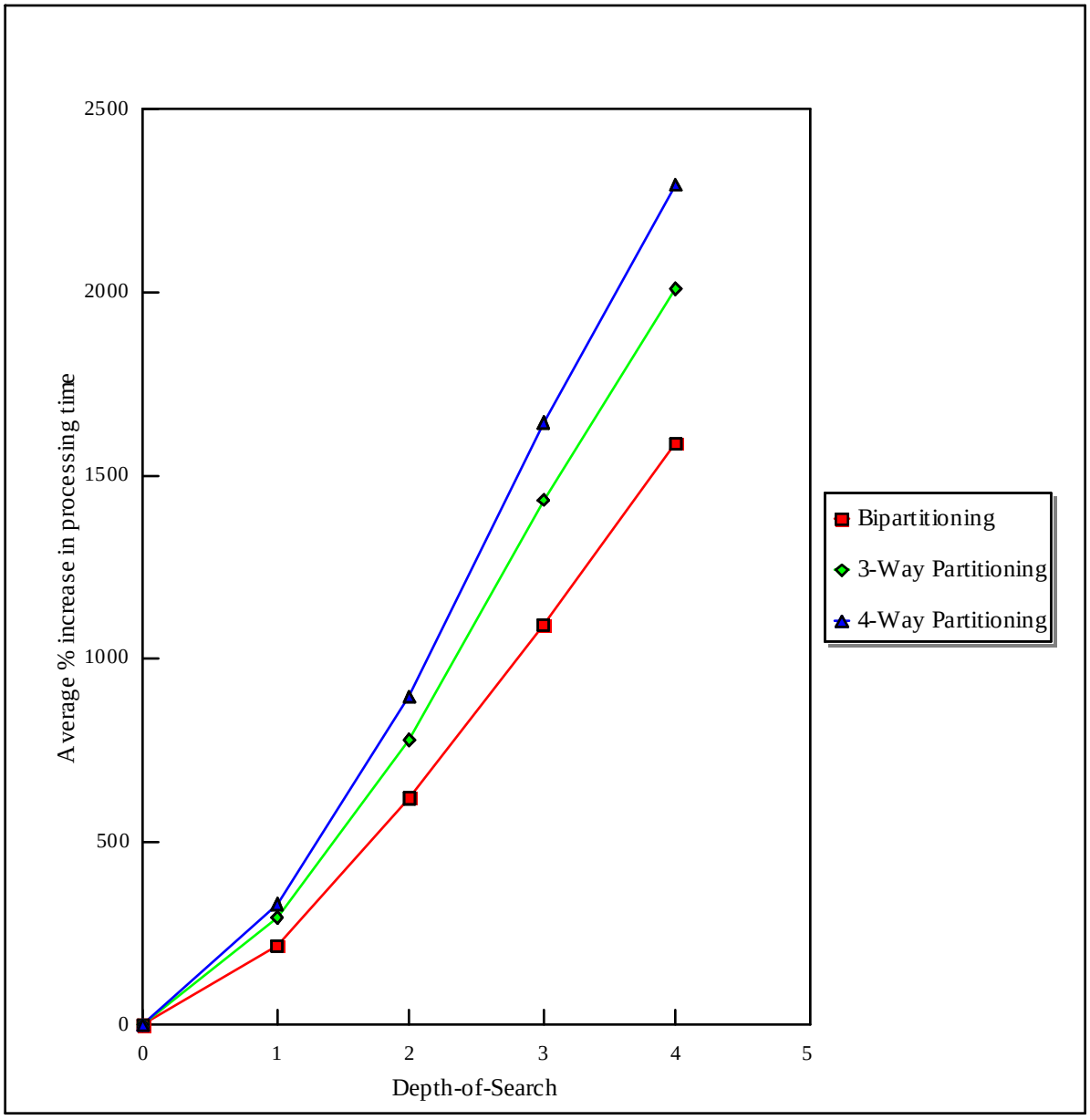


Figure 4.4-8. Percent increase in processing time of the multi-step algorithm.

It is evident from Figure 4.4-7 that a maximum step size of one ($S = 1$) provides an immediate improvement of between 11-14%, with a higher benefit achieved for partitioning onto a larger number of groups. Using a maximum step size of two ($S = 2$) provides additional improvement. Step sizes greater than two ($S > 2$) provide little improvement and may actually reduce the effectiveness of the multi-step algorithm. Figure 4.4-8 shows that the additional steps require a significant amount of additional processing time, as predicted by the exponential term of Equation 4-8. From these observations, a maximum step size of two is determined to be a reasonable default value and is used consistently in the work that follows.

As the refinement algorithms defined above search for a partition with the global minimum cutset size, the algorithms terminate when they encounter a local minimum in the solution space. Simulated annealing [74, 75] is incorporated into the multi-step algorithm to allow the algorithm to continue past local minimum.

Simulated annealing allows moves to be taken which provide an increase in the heuristic cost function. Moves that provide a decrease in the cost function are always taken. If a move provides an increase in the cost function, the probability that the move is selected is determined by the following function:

$$p = e^{-\frac{\Delta c}{t}} \tag{4-9}$$

In Equation 4-9 above, Δc is the change in the cost function and t is the current temperature. The current temperature is decreased by some percentage, the cooling rate, after each iteration of

the algorithm. This reduction in temperature causes the probability of selecting a more costly move to decrease exponentially as the algorithm progresses.

A small amount of additional processing is required to implement simulated annealing. However, this additional processing is an additive process of complexity $O(n)$ and therefore does not increase the complexity of the algorithm.

Two input parameters must be selected for any simulated annealing algorithm, the initial temperature and cooling rate. To test the algorithm, the initial temperature is selected to provide a 50% chance of selecting a move that adds three additional nets to the cutset during the first iteration. The selected cooling rated reduces the temperature by 3% for each iteration. These values were determined experimentally to be reasonable values.

The NETPART program is modified again, this time to incorporate simulated annealing into the multi-step partitioning algorithm. This new program is used to partition the previous eight benchmark circuits. Results from 20 trial runs of the simulated annealing algorithm are compared to the previous multi-step algorithm without simulated annealing, using a maximum step size of two for both algorithms. The average cutset size for bipartitioning is shown in Figure 4.4-9. Results for 3-way and 4-way partitioning are consistent. The standard deviation of the cutset size using both the simulated annealing algorithm and the multi-step algorithm for bipartitioning are shown in Figure 4.4-10. Again, the results for 3-way and 4-way partitioning are consistent.

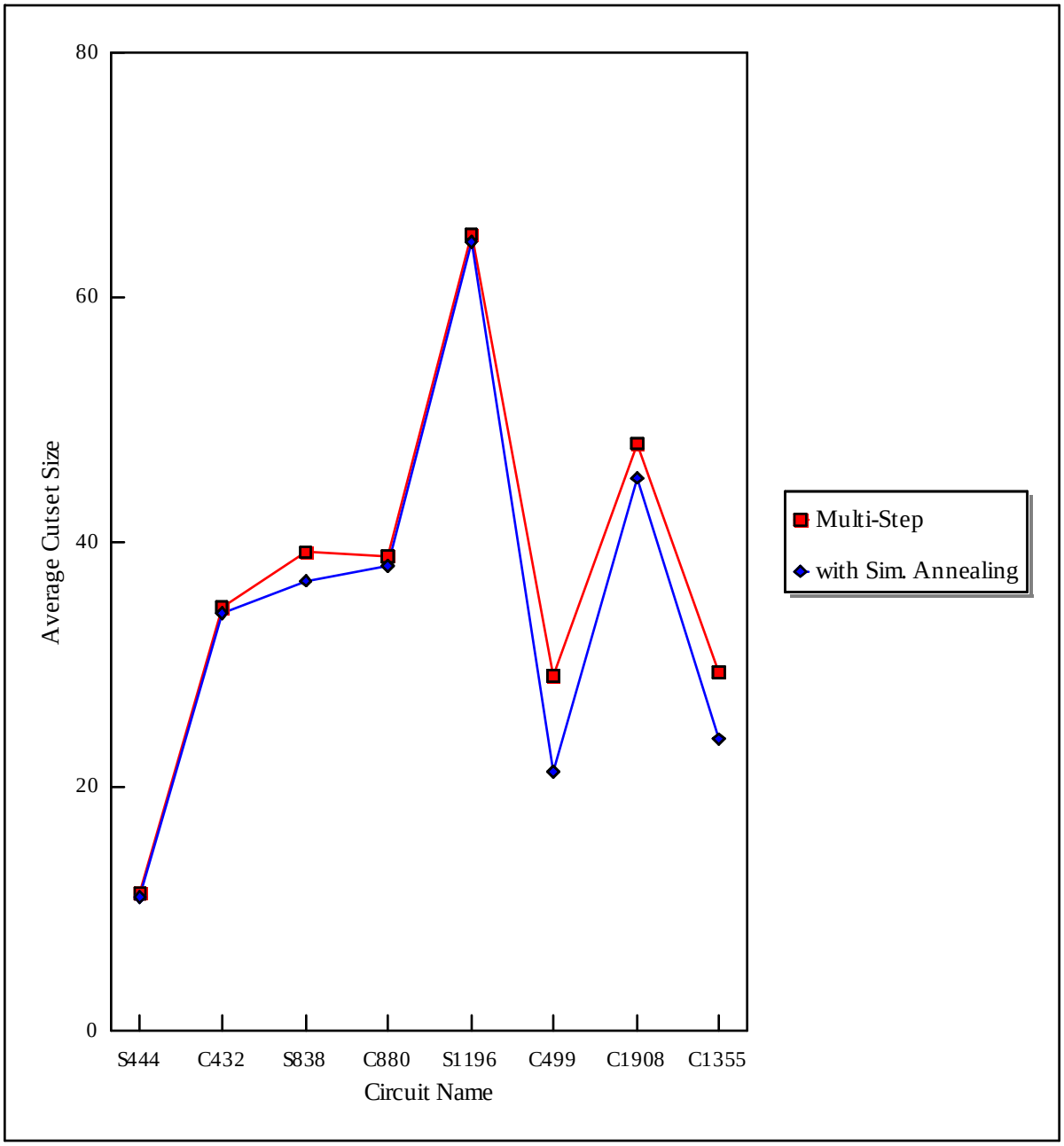


Figure 4.4-9. Average cutset size for 20 trials of the simulated annealing algorithm.

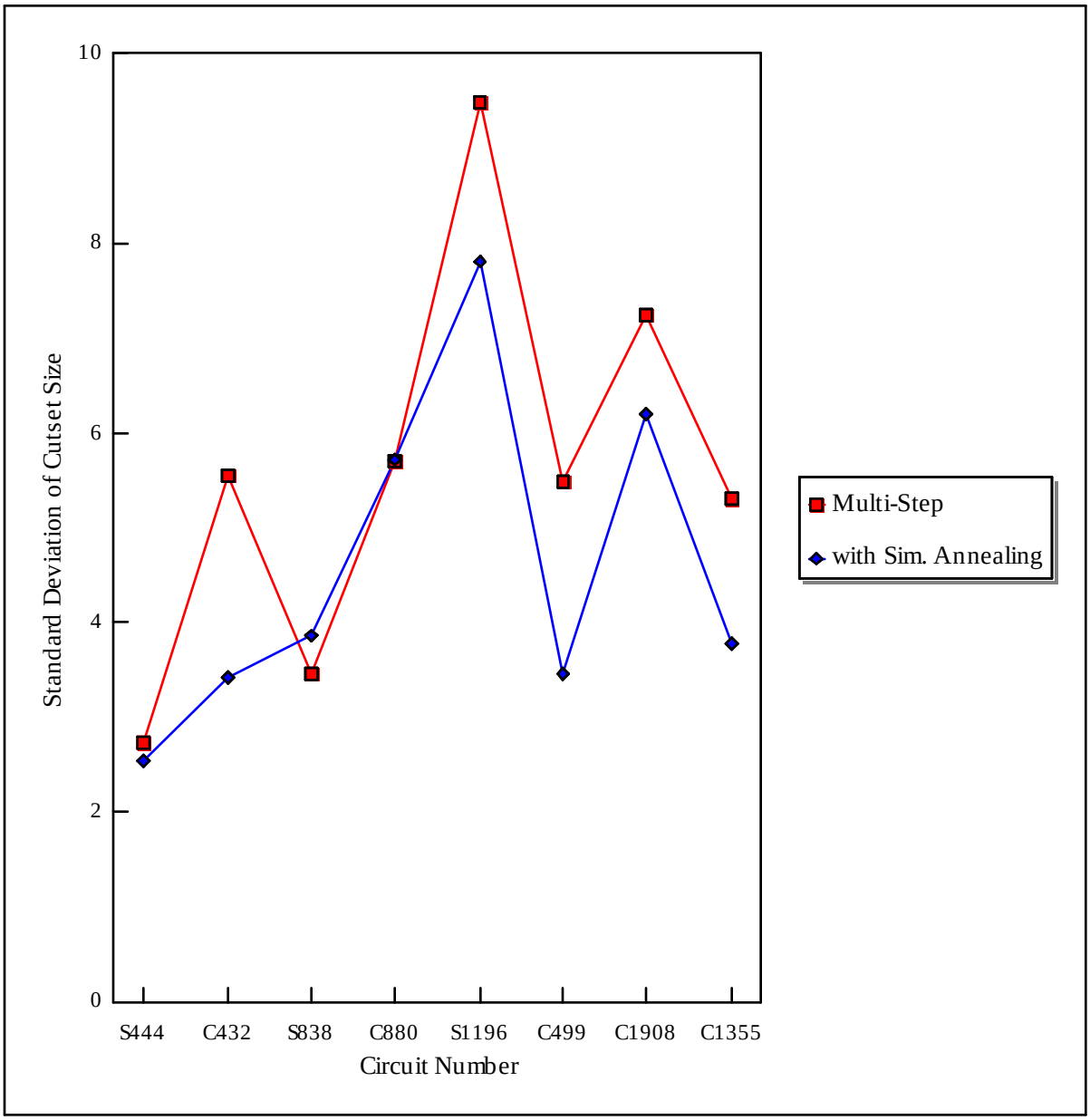


Figure 4.4-10. Standard deviation of cutset size for 20 trials of simulated annealing.

Figure 4.4-9 shows that the simulated annealing algorithm provides a 9% reduction in the average cutset size, compared to the multi-step algorithm. Similarly, the data in Figure 4.4-10 shows that the simulated annealing algorithm provided a more significant 15% decrease in the standard deviation of the average cutset sizes below the standard deviation of the multi-step algorithm.

The simulated annealing algorithm provides a consistent decrease in cutset size and a significant decrease in the standard deviation of the cutset size. Even though the simulated annealing algorithm requires some additional processing time, this additional processing time is justified by the reduction in cutset size and large gain in reliability (confidence) of the obtained solutions.

The net-move algorithm has been shown to provide superior performance by reducing the processing time and creating partitions with smaller cutset sizes. Enhancement of the net-move algorithm to the multi-step algorithm provides even higher performance, but with significant additional processing time for large maximum step sizes. However, it was shown that the greatest improvements in the algorithm are obtained by increasing the maximum step size to only one or two. Confidence in the quality of the solution is improved by incorporating simulated annealing into the algorithm, which provides a lower standard deviation in the final cost of partitions without affecting processing time. This new partitioning algorithm represents a significant improvement over existing partitioning algorithms and a major contribution of this dissertation.

4.4.1.2 Heuristic Cost Function

Refinement algorithms use a heuristic cost function to evaluate each new partition. In the partitioning algorithms defined above, the goal of the algorithm is to find the partition of a network with the smallest cutset while keeping the number of nodes balanced in all groups. Therefore, the heuristic cost function used previously summed the number of nets in the cutset and number of excess nodes in each group of the current partition.

The development system software requires a partitioning algorithm that creates K subcircuits from the original network, such that each of the K sub-circuits can be implemented on a single FPGA chip. Factors other than the size of the cutset and balanced group sizes should be considered to accurately estimate if a partition can be implemented on the destination architecture. Specifically, the distribution of interconnection and logic resources on the destination hardware architecture may not be balanced. Additionally, there may be more than one type of circuit element to be allocated within the FPGA chips. The heuristic cost function is modified to accommodate different types of FPGA resources and unbalanced availability of interconnections and FPGA resources in the destination hardware architecture.

A new heuristic cost function is defined for partitioning onto existing multiple FPGA-based hardware architectures. This new heuristic cost function, called the total cost function TC(P), evaluates five different individual cost constraints. These five individual constraints are defined in Table 4.4-1. When all the individual constraints are satisfied, the current partition should be able to be implemented on the destination hardware architecture. The

total cost function of a partition $TC(P)$ is the sum of four individual functions as defined in the following equation:

$$TC(P) = \sum_{0 \le i < 4} C_i(P) W_i \qquad (4\text{-}10)$$

where $C_i(P)$ is the individual cost function for constraint i and W_i is an integer weighting value for this individual cost constraint. The five individual cost functions evaluate the quality of the current partition based on one of five constraints defined in Table 4.4-1 for implementing the current partition on the destination hardware architecture. The five integer weighting values allow different cost considerations to contribute more or less to the total cost function. These weighting values provide a priority for each of the constraints to guide the evaluation of states for the partitioning algorithm.

Table 4.4-1. Individual cost contraints used in the total cost function.

#	Constraint
0	number of FGs in each group
1	number of FFs in each group
2	number of support chip net IOBs in each group
3	number of I/O net IOBs in each group
4	number of interconnect net IOBs in each group

The individual cost functions $C_i(P)$ each evaluate the ability of the destination hardware architecture to implement one of the individual constraints of Table 4.4-1. There is a one-to-one correspondence between the available resources and the group of the partition for which these resources are available. For example, if one group of the partition is to be implemented with a specific type of FPGA, the number of available FFs in that type of FPGA corresponds to the number of FFs available for that group. The constant $A_{i,b}$ is defined as the amount of a resource for constraint i that is <u>available</u> in the destination architecture to implement the subcircuit represented by group V_b of the current partition. Similarly, the variable $R_{i,b}(P)$ is defined as the amount of resource i that is <u>required</u> to implement the subcircuit represented by group V_b of the current partition.

The individual cost functions are defined to be equal to zero when the constraint is met for each group in the partition ($R_{i,b}(P) \le A_{i,b} \forall b, 0 \le b <$). Otherwise, the cost functions are defined to be proportional to the amount that the required resources exceed the available resources of the destination architecture. The individual cost functions are therefore defined as follows:

$$C_i(P) = \sum_{0 \le b < K} C_{i,b}(P) \tag{4-11}$$

$$C_{i,b}(P) = \left\{ \begin{array}{ll} 0 & \text{iff} \quad R_{i,b}(P) \le A_{i,b} \\ R_{i,b}(P) - A_{i,b} & \text{otherwise} \end{array} \right\} \tag{4-12}$$

These functions are used for the five constraints ($0 \le i < 4$) defined in Table 4.4-1. The complexity of all these individual cost functions is the same as the complexity of the heuristic

cost function used in the previous section, and therefore the total cost function does not affect the complexity of the algorithm.

The final three cost constraints are used to evaluate the possibility of globally routing each of the three types of system nets on destination hardware architecture. The individual cost functions defined above are easily calculated but are not very accurate. A more representative individual cost function for the last three individual cost functions is presented in the following.

Global routing of nets requires a representation of the both the circuit of the partitioned image processing design and the circuit of the destination hardware architecture. The individual nets in a network hypergraph are not associated with a particular node, as the was the case for the resources in the individual cost functions above, but with the interconnection of two or more nodes. To simplify the representation of nets, each net in the network hypergraph is decomposed into multiple edges with degree two. This simplification transforms the network into a graph.

The last individual cost function, the limited availability of interconnection nets, is improved in the following. Only nets in the cutset of the partitioned network need to be considered, since other nets are implemented with only FPGA interconnection resources. These nets of the cutset interconnect groups, instead of nodes. The individual cost functions for the constrained number of interconnection nets is defined above as $C_4(P)$. The number of global nets that are available in the destination architecture to implement the nets that interconnect group V_b and group V_c of the current partition is defined as the constant $A_{4,b\text{-}c}$ and the number of edges of degree two in the partitioned circuit between V_b and group V_c of the current partition is the variable $A_{4,b\text{-}c}(P)$. This new individual cost function is defined to be equal to zero when the constraint is met for each group in the partition ($R_{4,b-c}(P) \le A_{4,b-c} \forall b-c, 0 \le b < K : b < c < K$).

Otherwise, the cost function is proportional to the number of edges that exceed the interconnection resources of the destination architecture. This new individual cost function is defined as follows:

$$C_4(P) = \sum_{\substack{0 \le b < K \\ b < c < K}} C_{4,b-c}(P) \tag{4-13}$$

$$C_{4,_{b-c}}(P) = \left\{ \begin{array}{ll} 0 & \text{iff} \quad R_{4,b-c}(P) \le A_{4,b-c} \\ R_{4,b-c}(P) - A_{4,b-c} & \text{otherwise} \end{array} \right\} \tag{4-14}$$

This new individual cost function provides a better estimate of the cost associated with the global routing process.

This new individual cost function has a higher computational complexity than the previous individual cost functions. The computational complexity of this cost function is as follows:

$$O(K^2 + m(K + \overline{|e|} + K^2) + K^2) = O(mK^2) \tag{4-15}$$

This increases the computational complexity of the multi-step net-move algorithm defined in the previous section. Using this new individual cost function, the computational complexity of the multi-step net-move algorithm is increased to the following:

$$O(mKS|e|^s\{m^2K + mK^2\}) \tag{4-16}$$

A second term of $O(mK^2)$ has been added to the computation complexity of multi-step algorithm. This term only affects the processing time of the algorithm when the number of groups increases. The increased complexity of the multi-step algorithm may be justified when of a good estimate for the cost of global routing is required, but to save processing time the new individual cost function not currently used.

The total cost function defined above allows a variable number of individual constraints to affect the heuristic cost function. The number and type of individual cost functions can be adapted for a number of different partitioning problems. This adaptability of the heuristic cost function is a significant benefit of the use of a refinement algorithm. Five specific individual cost functions are developed for the particular problem of partitioning an input circuit onto the network of an FPGA-based hardware architecture. These cost functions provide realistic estimates of the ability to implement the input circuit on the destination architecture.

4.4.2 Global Routing

All nets in the cutset of the final partition must be implemented with existing interconnections of the destination architecture's printed circuit board. The process of global routing assigns the nets in the cutset of the final partition to one or more of the limited number of these available physical interconnections.

Two circuits are defined for the global routing problem, the partitioned logic circuit and the circuit of the destination hardware architecture. Each circuit is identified with either a

superscript L or A for the logic or architecture circuits, respectively. As in the previous section, the partitioned logic circuit is an represented by a hypergraph, in this case identified as the logic hypergraph $H^L(V^L,E^L)$ where $V^L = \{ v^L_i \mid i = 1,2, \ldots n^L \}$ is the node set and $E^L = \{e^L_j \mid j = 1,2,\ldots m^L\}$ is the net set. Similarly, the circuit of the destination hardware architecture is represented by the architecture hypergraph $H^A(V^A,E^A)$. Note that for the partition $P(H^L)$, the groups V^L_1, V^L_2, … V^L_k correspond to nodes in the node set V^A of the architecture hypergraph H^A.

An architecture net e^A_k is a suitable candidate for routing a logic net e^L_j if it interconnects the required components of the destination architecture ($span_g(e^L_j) \subseteq span_v(e^A_k)$) and it has not been previously allocated to another logic net. A sequential assignment algorithm is used to allocate the interconnections of the destination architecture to nets of the cutset. This algorithm, called the sequential routing algorithm follows:

> **Sequential Routing Algorithm**: Each net in the logic net set E^L is sequentially selected to be the "current logic net". If the current logic net is in the cutset, all nets in the architecture net set E^A that have not been previously allocated to a logic net are sequentially selected to be the "current architecture net". If the current architecture net is a suitable candidate for routing the current logic net (as defined above), the current architecture net is allocated to the current logic net. The algorithm terminates with success when all nets of the cutset are routed, or fails if a suitable candidate architecture net cannot be found for the current logic net.

This simple global routing algorithm is easily performed. The computational complexity of the sequential global routing algorithm is as follows:

$$O(m^L(K + \overline{|e^L|} + m^A(K + \overline{|e^A|} + K)) = O(m^L m^A K) \quad (4\text{-}17)$$

Although this global routing algorithm may fail to route a circuit, it has low computational complexity. This simple algorithm is used with the expectation that an improved, and probably more computationally complex, routing algorithm will be implemented at a later time.

The global routing algorithm defined above could be modified to reduce the probability of failure. One method allows the use of two or more architecture nets to create the connectivity of a single logic net. A second method routes signals through the logic of FPGAs. It may be possible to implement either of these methods without adding additional computational complexity to the global routing algorithm.

One final method of improvement is to implement a best-first routing strategy. The number of candidate architecture nets for each logic net is calculated in each step of the algorithm. The logic nets with the fewest candidates are routed first in each step, using the architecture nets with the smallest order. However, the best-first strategy will increase the computational complexity of the global routing algorithm. These methods are not implemented and are left for future work.

This section has provided the basic algorithms of network partitioning and global routing required for automatic translation of image processing designs. A new network partitioning algorithm with an improved state search algorithm and heuristic cost function has been developed. This new algorithm is shown to provide better partitioning results with less processing

time than traditional algorithms. A simple global routing algorithm is implemented to provide this functionality. The computational complexity of all the previously mentioned algorithms has been calculated and analyzed, to identify areas of possible future improvement.

4.4.3 Translation Implementation

A single program has been created to control the translation of an input image processing design into the logic of a multiple FPGA-based hardware architecture. This C program, called MTRANS, automatically partitions the logic, places support chips, routes interconnect system nets, and compiles the partitioned logic circuit into the resources of available FPGA chips. Several individual commercial software and custom programs written by the author combine to establish the desired functionality. All custom programs have been written in ANSI C. The input files, output files, and program flow of the MTRANS program are shown in Figure 4.4-11. Custom programs developed by the author are highlighted in this figure.

Two types of files are input to and output from the MTRANS program for each image processing design. First, a representation of the logic and interconnections (the network of interconnected image processing modules) created by the designer is required. The graphical entry of the Viewlogic schematic capture program Viewdraw creates a netlist in the "WIR/" subdirectory of a project, in the *wirelist* format of the Viewlogic Corporation [47]. Hierarchical design techniques require that several logic components from possibly separate projects be combined and interconnected to create a flattened wirelist of a single complete design. The

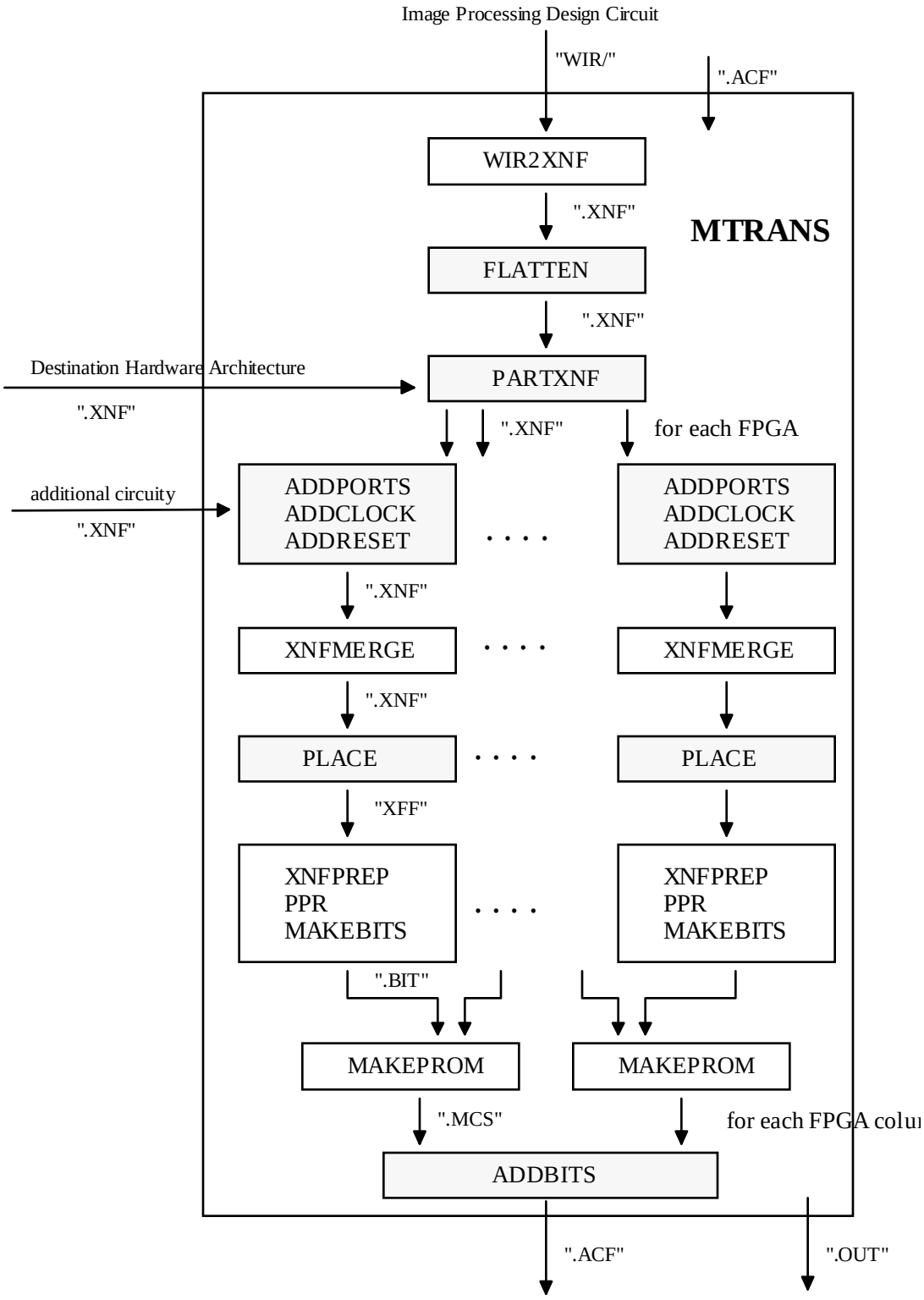


Figure 4.4-11. Program flow and file extensions of the MTRANS program.

MTRANS program must have access to wirelist files for all the individual logic components required to create a flattened wirelist of the input image processing design. Additional wirelist files are required for the port logic, clock logic, and reset logic that is automatically added by the MTRANS program.

The second type of file input to the MTRANS program is an ACF file, as mentioned previously in Section 4.3.2. This file contains information to control the translation of the image processing design into the resources of the destination architecture. It defines the destination hardware architecture, its current configuration, and the port addresses implemented in the input image processing design. The input ACF file is updated and modified to provide the primary output of the MTRANS program. Port locations are updated after partitioning in the port section. The MTRANS program creates bitstreams used to program the MORRPH PE array and adds them to the bitstream section of the output ACF file. The ACF file format has been developed by the author for this dissertation and a complete definition of the ACF file format is presented in Appendix I.

The second MTRANS output file type is the *translation report file* (with a ".OUT" file extension). This ASCII file contains text information from the translation process. Partitioning results, support chip locations, and the exit status of custom programs are included in this file.

The XACT software from the Xilinx Corporation creates the bitstreams required to program Xilinx FPGA chips in the hardware architecture. This software package utilizes an ASCII file format called the Xilinx Netlist Format or XNF file format [76] to represent circuits. Therefore, the input circuit must be translated from the wirelist format of the Viewlogic

corporation into the XNF file format of the Xilinx Corporation. The WIR2XNF program from the Viewlogic corporation performs this translation.

All development system programs written by the author process files in the XNF file format exclusively. These custom programs modify the circuit of the input image processing design by automatically editing files in the XNF format to add or remove circuit elements (i.e., modules and nets).

The first custom program used by the MTRANS program is the FLATTEN program. After the input wirelist of the image processing design is converted into the XNF file format by the WIR2XNF program, the FLATTEN program removes the highest level of the design hierarchy. Modules in the top-level schematic are replaced with their underlying schematics. This effectively flattens the hierarchical image processing design schematic down one level.

The FLATTEN program replaces the corresponding section of the XNF file for each component in the image processing design with the XNF representation of each component's sub-circuit, where each component is a module from one of the image processing libraries. Net connections between components in the high level schematic are translated into connections with components in each sub-circuit.

The labels for nets and logic elements in the flattened schematic must be modified to reflect the new design hierarchy. All circuit components have both a *relative label* and an *absolute label*. The relative label is used in the schematic to identify the component on its own level of the schematic. Absolute labels illustrate the location of the circuit component in the design hierarchy. An absolute label for a component in the lower levels of a design hierarchy

consists of the relative label of the component with a prefix of the absolute label of the parent element followed by the "\" character. Absolute labels are equal to relative labels for all components in the top level schematic. For example, a multiplexer at the top-level schematic of a hierarchical design may have "M2" as its absolute (and therefore relative) label. A NAND gate with the relative label "G1" used in the schematic of this multiplexer has the absolute label "M2\G1". When the schematic is flattened, the labels assigned to both nets and logic elements in the new first level of the schematic are given their absolute labels in the unflattened schematic to preserve the label naming conventions established by the previous design hierarchy.

The FLATTEN program performs three additional processes. First, the FLATTEN program removes logic that should not be compiled into the logic of FPGA chips. Second, individual ***CLOCK*** and ***RESET*** nets are merged. Finally, nets that implement constant operand values of the image processing module are translated into power and ground connections by the FLATTEN program.

Support chips are included in the top-level schematic of the image processing module to allow simulation of the module. These components are not compiled into the resources of FPGA chips and are removed from the flattened XNF file. Nets connected to support chips remain in the output XNF file, but the components themselves are removed. As defined previously in Section 4.2.5, all logic that is to be compiled into the resources of FPGA chips is contained in a single module. This module is identified with the attached attribute value "XILINX" on its symbol. All other modules represent logic that is implemented with support chips. Therefore,

only components in the top level schematic of an image processing module with the attribute value "XILINX" are included in the flattened output XNF file.

Individual ***CLOCK*** and ***RESET*** nets are included in the top level schematic of each image processing module. These nets are given their absolute labels when the circuit of an image processing design is flattened. The FLATTEN program merges all these individual nets by replacing them with the labels ***CLOCK*** and ***RESET***.

Variable and constant operand values are defined by net and bus connections in the highest level schematic of each module, as described previously in Section 4.2. When attribute values are provided at the highest level of the schematic that correspond to net or bus names of the flattened schematic, the flattening procedure connects these nets and busses to the corresponding power and ground nets required to establish the constant value defined by the attribute. This allows constant operand values to be provided for individual modules in an image processing design by simply defining an attribute value in the top level schematic.

The flattened design contains only logic that is to be compiled into FPGA chips. However, this logic may require the resources of several FPGA chips to be implemented. Therefore, the next processes of the translation utility are network partitioning and global routing. These processes combine to create K sub-circuits that are compiled into the individual FPGA chips available in the destination hardware architecture. A single C program, called PARTXNF, performs both of these processes for the translation program.

Both the input image processing design circuit and destination hardware architecture network are input to the PARTXNF program from files in the Xilinx XNF format. This input

allows the partitioning program to accept arbitrary circuits and destination architectures other than MORRPH-ISA. The input image processing design circuit is partitioned into K subcircuits represented by K different XNF files, one for each of the FPGAs in the destination hardware architecture.

The incompletely specified MORRPH architecture creates ambiguity in the amount of logic and interconnection resources that are available in each PE of the MORRPH array, since a variety of FPGA chip types may be inserted into each PE in the MORRPH array or the sockets may remain empty (see Table 3.3-1). Since this information is required for the heuristic cost function of the partitioning algorithm, the number and type of FPGAs included in the MORRPH array is determined before partitioning. Typically, the largest desired FPGA for each PE location is defined in the configuration section of the ACF file. The resources allocated to each FPGA chip are analyzed after partitioning to determine if smaller FPGA chips may be used.

The flattened schematic of an image processing design is input to the PARTXNF program. Therefore, partitioning is done at the module level and all the logic of a module must be placed in a single FPGA. This coarse-grained partitioning reduces the number of elements to be partitioned and therefore allows the use of more computationally complex partitioning algorithms.

The PARTXNF program uses the multi-step net-move algorithm with simulated annealing and the five constraint total cost function defined in previous sections. However, some limitations of the partitioning program were experimentally observed. These limitations required improvement of the partitioning algorithm as described in the following.

The net-move algorithm provides excellent results when partitioning to obtain circuits with a minimum cutset size but does not find moves required to distribute FPGA logic resources. This is because the net-move state search algorithm is optimized to find and remove nets from the cutset, not the satisfy FPGA resource constraints. However, the node-move state search algorithm does find many of these moves. Therefore, a new combinational algorithm is implemented that iterates between the net-move and node-move algorithms. Each pass of the algorithm consists of a full pass of the multi-step net-move algorithm followed by a full pass of the multi-step node-move algorithm. The algorithm terminates when the total cost function is reduced to zero. If no improvement is found by either the net-move or node-move algorithms in a current pass and the cost is greater than zero, a new random starting point is selected and the partitioning algorithm restarts. The addition of the node-move algorithm slows the execution time of the partitioning algorithm, but does not change the computational complexity of the algorithm. This additional computation time is justified by the increased probability of finding an acceptable partition.

The actual amount of FPGA resources (e.g., CLBs, IOBs, FGs, and FFs) required to implement the logic of each image processing module is not known before the image processing design is compiled. This information is required to calculate the FPGA resource constraints of the heuristic cost function. Therefore, an estimate of the logic required by each module in an image processing design is created. This is accomplished by compiling only the image processing module into the largest FPGA chip available and parsing the information from the output report file of the Xilinx PPR program.

A new ASCII file format is created, called the Part Resource File or PRF file, that contains the estimate of the number of CLB, IOB, FG, and FF resources required to implement an image processing module. The PRF file format is trivial and only contains four ASCII strings. To illustrate the PRF file format the corresponding file of the **OFFSET_A** module (used in previous examples) is provided below:

```
CLBS 25
IOBS 0
FGS 35
FFS 16
```

PRF files also exist for each of the FPGA types, to determine the amount of logic and interconnect resources represented by different FPGA chips. All PRF files are contained in a "PRT/" subdirectory of the current project. If any modules in the flattened design do not have a corresponding PRF file in this directory, they are independently compiled and a PRF file is automatically generated by the partitioning software.

The PARTXNF program also performs global routing, using the algorithm defined in the previous section. New nets, pads, and flip-flops are automatically added for each system interconnection net used to implement nets in the cutset of the partitioned image processing design. An output flip-flop, pad, and net connections are added to the XNF file of the source FPGA. Similarly, an input flip-flop, pad, and net connections are added to each of the destination FPGAs. Since the image processing design is partitioned at the module level, all nets in the cutset are SUIT bus connections and may be registered at chip boundaries without affecting the performance of the image processing design.

The PARTXNF program creates K output files in the XNF file format, one for each of the K subcircuits created by the partitioning algorithm. Each of these XNF files are now individually modified and compiled into the resources of individual FPGA chips by the MTRANS program.

Clock, reset, and port circuitry is added to each of the output XNF files after partitioning. The ***CLOCK*** and ***RESET*** net labels are reserved names of the design methodology (see Section 4.2.5). Both the ADDCLOCK and ADDRESET programs search the XNF files for the net labels ***CLOCK*** and ***RESET***, respectively. The logic represented by the **XCLOCK** schematic is automatically added to the XNF file by the ADDCLOCK program to drive all ***CLOCK*** nets. Currently, all ***RESET*** nets are connected to ground by the ADDRESET program, as the FPGA chip logic is reset after programming.

Port locations allow dynamic operand values to be supplied to or read from the modules of an image processing design. The logic required to implement port locations is automatically added to the XNF files of the image processing design by the ADDPORTS program. The port section of the ACF file (see Appendix I) defines the location and configuration of all I/O ports. Additional logic to implement each 8-bit read or write port location is automatically added by connecting either the **XRPORT** or **XWPORT** modules to desired nets in the output XNF file. A single port interface module, called **XPORTS** is added to all FPGAs to interface the ports with the I/O bus of the host computer. This additional logic interfaces with a common 24-bit *port communication bus* that interconnects the logic of all individual port locations and the port interface module within each FPGA chip.

Although the development system has been created to function for any incompletely-specified FPGA-based hardware architecture, the implementation of ports is dependent on the interface with the host computer. The MORRPH-ISA board loads specific interface logic into the FPGA chips of the MORRPH PE array to interface with the ISA bus of the host computer. This logic is contained in the **XPORTS** module and uses eight data bits, five address bits, ***READ***, ***WRITE***, and ***CS*** signals. The **XPORTS** module may be modified to implement interfaces to other busses.

Only the FPGAs at the bottom of the MORRPH processing array are connected to the ISA interface. To implement ports in the upper rows of the processing array, an additional module, called **XPORTUP**, is added to all FPGAs except FPGAs in the top row of the PE array. This module interfaces the port communication bus signals with the North-South interconnection busses of the MORRPH PE array.

Each individual port location contains a 5-bit attribute value that is compared to the 5-bit address of the ISA bus interface. The 5-bit address value of each individual port location is defined in the ACF file. Although the use of separate chip selects for each column of the MORRPH PE array allow 32 read and write ports for each column, addresses for ports must be defined before partitioning. To avoid conflicts, only 32 read and write port addresses are defined for image processing designs that are translated into the resources of a MORRPH-ISA board. The **ADDPORTS** program updates the ACF file to identify the FPGA in which each port is located. Future upgrades with automatic port placement after partitioning will allow up to 32 ports in each column.

The PLACE program identifies and resolves conflicts in support chip net allocation. Pad locations for support socket nets are defined when a module is created in the design entry process. These default locations identify the pin locations for each type of support chip by defining a default chip placement location. However, the location of support chips is determined by the partitioning function of the PARTXNF program and conflicts may occur when more than one support chip is included in a single FPGA.

Two rows of 40-pin headers separated by 300 mils are connected to I/O pins of each FPGA in the MORRPH-ISA architecture. The pins are numbered consecutively, from the top of the printed circuit board. A second row of pin headers are next to both rows, to establish support chip connections to power and ground with wire jumpers. A separate net on the MORRPH-ISA board connects each of the 80 pin header locations to a single pin of the associated FPGA chip.

Support chips are given a default location in the support sockets when added to the schematic of an image processing module. This default location is used when no conflicts exist after partitioning the image processing design logic. When a conflict occurs, the support chips associated with a particular FPGA are sequentially placed in descending support socket locations, with the first chip placed at the top of the socket. A positive or negative offset is added to the default pin locations to place the chip. The use of default chip positions minimizes the movement of support chips in the destination architecture.

All modifications to the XNF files have been completed at this point in the translation process. The individual circuits of each XNF file represent the actual logic that is compiled for each FPGA chip. The automated translation programs XNFPREP, PPR, and MAKEBITS, of the

Xilinx XACT development tools, are used to compile these individual XNF files. A separate bitstream file is created to program each of the individual FPGA chips. Since all FPGAs in a column are daisy-chained for programming, the MAKEPROM combines the bitstreams from all FPGAs in a column into a single ASCII file in the MCS file format [55]. The MCS files for each column are added to the ACF file by the MTRANS program.

Board assembly is the last step of the translation process. This procedure involves assembling the appropriate FPGA and support chips into the MORRPH PE array. If the same type of FPGA chips specified in the configuration section of the ACF file are desired for operation they may be directly assembled into the MORRPH PE array. The FPGA configuration data added to the bitstream section of the ACF file is only valid for the specific types of FPGA chips defined before partitioning. However, the logic resources allocated to each FPGA are analyzed to determine when it is possible to use smaller FPGA chips. Support chips are placed in the locations specified by the output of the PLACE program, and jumper wires are added for power and ground connections.

The MTRANS program provides a fully automated process for the translation of image processing designs. Board assembly is the only manual process required. Since board assembly is not required for completely-specified FPGA-based hardware architectures, this development system provides a fully automated translation utility for these architectures. The level of automation provided by this development system is greater than any commercial system for image processing with FPGA-based CCMs and therefore represents a significant contribution of this dissertation.

Chapter 5. Experimental Results

5.1 Justification

The goal of this dissertation has been to establish a new design methodology for creating real-time image processing hardware. Previous chapters have defined the new design methodology and its current implementation. This chapter proves the viability of the new design methodology by using the hardware architecture and development system software of this dissertation to create hardware solutions for several real-time image processing problems.

Twenty-eight low-level image processing modules have been created for this dissertation. The functionality of these modules is defined in Appendix II. Each module is located in one of the five libraries (SUIT2, SUIT2A, SUIT2B, SUIT2C, or SUIT2D) defined in Section 4.2.5. These libraries are called the *evaluation libraries* and represent the minimal amount of design effort required to illustrate and evaluate the new design methodology. A continuation of this design

effort is required to develop these modules into a comprehensive library of low-level image processing modules.

Evaluation library modules are used to create six individual image processing designs. These individual designs have been selected to show a range of complexity, representative of the wide range of possible image processing applications. Designs are presented in order of increasing complexity and are called the *basis image processing designs*.

An image processing design is created for each of six individual problems by interconnecting existing components from the libraries of image processing modules defined in Appendix II. The functionality and performance of these designs are verified using the in-circuit emulation capabilities of the development system software. Each basis image processing design transmits and receives data using the ZEE bus format defined in Section 4.2.3 for inter-board communications.

The first three basis image processing designs include only a single module for processing. Four other modules are included in these simple designs to identify I/O interconnections and for inter-board communications. Several modules are used for processing in the final three basis image processing designs.

The basis image processing designs provide a method for evaluating the new design methodology created for this dissertation. They are representative of the type of low-level image processing functionality the development system is intended to provide. The efficiency and performance of both the software development system and hardware architecture are quantitatively evaluated using the six basis image processing designs.

5.1.1 Evaluation Library Image Processing Modules

Each of the 28 modules in the evaluation libraries has been independently created, compiled, and verified using tools of the development system software. Modules are created to be easily understood and to provide reliable operation. The development of these modules establishes a tutorial for the creation of new libraries of higher performance modules.

FPGA logic resources are a primary concern of the new design methodology, as this information is used by the automated circuit partitioning software. Resource requirements of all modules in the evaluation libraries are shown in Table 5.1-1. This information is obtained by compiling each module into a single Xilinx 4010PG191-5 FPGA chip and manually examining the output report file of the Xilinx PPR program.

The compilation described above also provides the maximum operating frequency of the image processing modules. However, many modules require multiple clock cycles to process each pixel value. Additionally, modules in the SUIT2B, SUIT2C, and SUIT2D libraries process multiple-byte data sizes. The Mbytes/sec and Mpixel/sec processing rates of each module, when compiled into a Xilinx 4010PG191-5 FPGA chip, are provided in Table 5.1-2.

It is apparent from Table 5.1-1 that the amount of logic and I/O resources required by all of the evaluation modules is less than the amount currently available in individual FPGA chips. Table 5.1-2 shows that a wide range of performance, from 1 - 30 Mpix/sec, is provided by these modules. These modules are used in the next section to create complete image processing designs.

Table 5.1-1. FPGA resource requirements of modules in evaluation libraries.

Module Name	Library	FGs	FFs	IOBs	CLBs
MULTIPLEX2	SUIT2	108	100	0	125
MULTIPLEX4	SUIT2	181	168	0	197
SUIT2ZEE	SUIT2	121	148	0	153
ZEE2SUIT	SUIT2	7	34	0	37
BLOCK_CHAN	SUIT2	9	16	0	24
INC_CHAN	SUIT2	4	17	0	19
REGINT2BW	SUIT2	4	8	0	12
HISTOGRAM_A	SUIT2A	171	75	72	150
SHADE_A	SUIT2A	183	121	48	225
FIELD_VIEW_A	SUIT2A	53	36	0	55
OFFSET_A	SUIT2A	15	15	0	24
EROSION_A	SUIT2A	154	52	0	133
DILATION_A	SUIT2A	154	52	0	130
THRESHOLD_A	SUIT2A	8	16	0	17
BOUNDARY_A	SUIT2A	216	142	24	214
LAPLACE_3x3_A	SUIT2A	161	118	48	170
AVERAGE_3x3_A	SUIT2A	331	217	48	328
GAUSSIAN_3x3_A	SUIT2A	320	178	48	332
HALF_ROW_A	SUIT2A	17	18	0	29
HALF_COL_A	SUIT2A	14	18	0	24
LUT_A	SUIT2A	10	16	24	21
FIELDVIEW_C	SUIT2C	57	38	0	54
SHADE_C	SUIT2C	182	91	48	203
HALF_COL2_C	SUIT2C	62	45	0	67
THRESHOLD_C	SUIT2C	108	51	0	91
BOUNDARY_C	SUIT2C	222	142	24	214
AVERAGE_CHAN_C	SUIT2C	176	57	0	158
SEPARATE_CHAN_D	SUIT2D	24	27	0	32

Table 5.1-2. Performance of modules in evaluation libraries.

Module Name	Library	Speed(MHz)	clks/byte	Mbytes/sec	Mpix/sec
MULTIPLEX2	SUIT2	30.3	1	30.3	n/a
MULTIPLEX4	SUIT2	22.17	1	22.17	n/a
SUIT2ZEE	SUIT2	39.53	4	9.88	n/a
ZEE2SUIT	SUIT2	47.62	3	15.87	n/a
BLOCK_CHAN	SUIT2	38.46	1	38.46	n/a
INC_CHAN	SUIT2	48.54	1	48.54	n/a
REGINT2BW	SUIT2	42.19	1	42.19	n/a
HISTOGRAM_A	SUIT2A	21.88	4	5.47	5.47
SHADE_A	SUIT2A	9.17	1	9.17	9.17
FIELD_VIEW_A	SUIT2A	18.83	1	18.83	18.83
OFFSET_A	SUIT2A	16.84	1	16.84	16.84
EROSION_A	SUIT2A	21.74	2	10.87	10.87
DILATION_A	SUIT2A	20.96	2	10.48	10.48
THRESHOLD_A	SUIT2A	28.33	1	28.33	28.33
BOUNDARY_A	SUIT2A	14.04	4	3.51	3.51
LAPLACE_3x3_A	SUIT2A	11.64	2	5.82	5.82
AVERAGE_3x3_A	SUIT2A	10.62	3	3.54	3.54
GAUSSIAN_3x3_A	SUIT2A	9.44	2	4.72	4.72
HALF_ROW_A	SUIT2A	21.69	1	21.69	21.69
HALF_COL_A	SUIT2A	25	1	25	25
LUT_A	SUIT2A	32.36	1	32.36	32.36
FIELDVIEW_C	SUIT2C	18.9	1	18.9	18.9
SHADE_C	SUIT2C	9.13	1	9.13	3.04
HALF_COL2_C	SUIT2C	21.69	1	21.69	7.23
THRESHOLD_C	SUIT2C	8.53	1	8.53	2.84
BOUNDARY_C	SUIT2C	13.62	4	3.41	1.14
AVERAGE_CHAN_C	SUIT2C	9.47	1	9.47	3.16
SEPARATE_CHAN_D	SUIT2D	26.39	1.33	19.79	4.95

5.1.2 Single Module Image Processing Designs

The first three basis image processing designs are presented in this section, each with a single module for processing. The first basis image processing design adds a programmable offset to an input gray scale image using the **OFFSET_A** image processing module. The second basis image processing design uses a 3x3 Laplacian window operator for edge detection using the **LAPLACIAN_3x3_A** image processing module. Finally, the last basis image processing design uses the 3x3 average window operator to perform a low-pass filter using the **AVERAGE_3x3_A** module. These functions are described in Section 1.2 and the modules are defined in Appendix II.

The offset operation has been used frequently in examples of this text. An image processing design that adds a programmable offset to the input image was introduced in Example 4.2-5 (see Figure 4.2-9). This image processing design uses the **OFFSET_A** module to process gray scale image data on channel 0 of the SUIT bus.

Input data is obtained from the North I/O bus of the MORRPH PE array using the **NORTHIN** module and converted to the SUIT bus format with the **ZEE2SUIT** module. After processing, the result data is re-converted to the ZEE bus format by the **SUIT2ZEE** module and output on the East I/O bus of the MORRPH PE array using the **EASTOUT** module.

The edge detection and low-pass image processing designs are created by replacing the **OFFSET_A** module of Figure 4.2-9 with either the **LAPLACIAN_3x3_A** module or the **AVERAGE_3x3_A** module. These image processing designs are shown in Figure 5.1-1 and Figure 5.1-2.

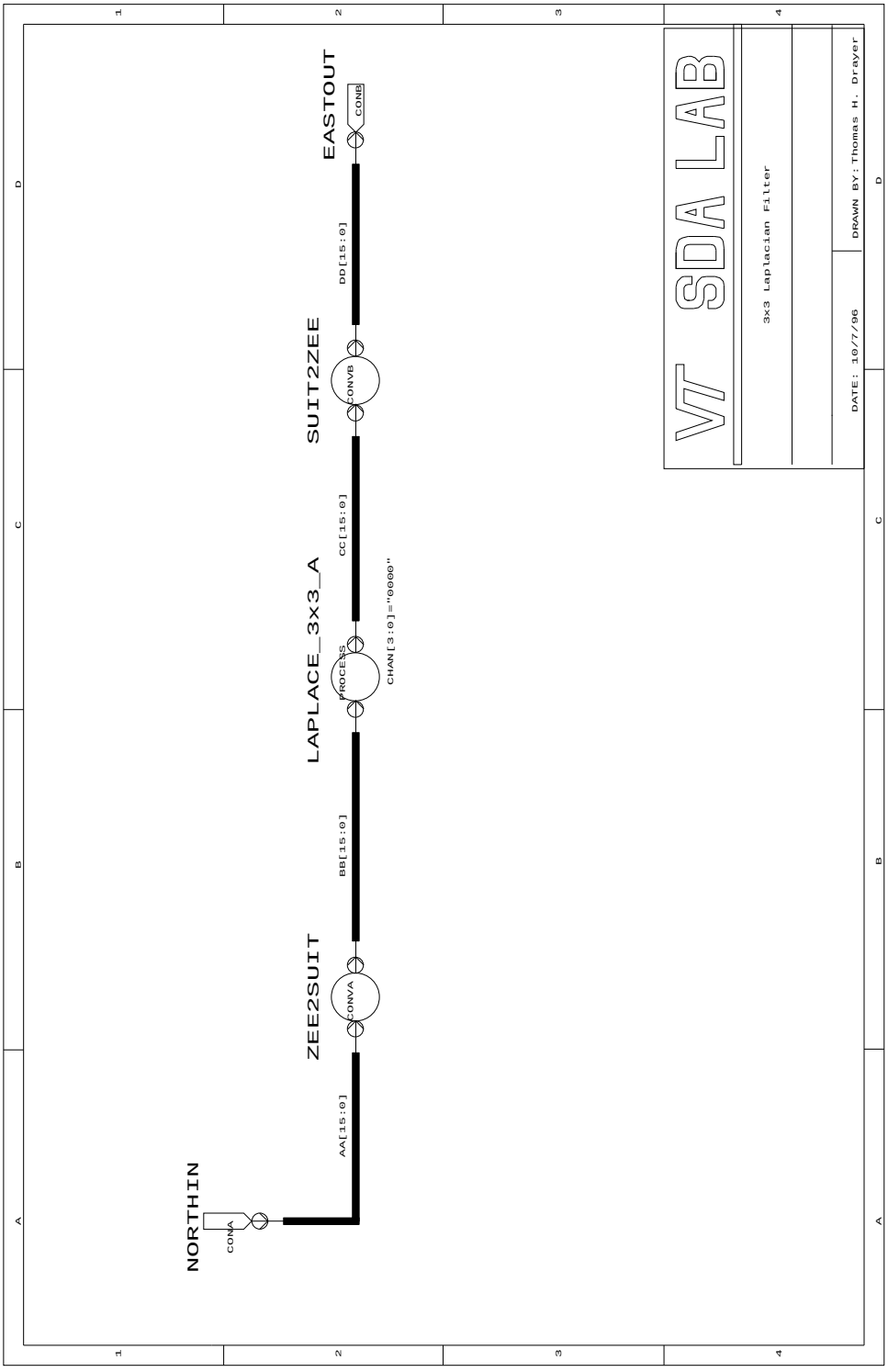


Figure 5.1-1. Schematic for 3x3 Laplacian window operator image processing design.

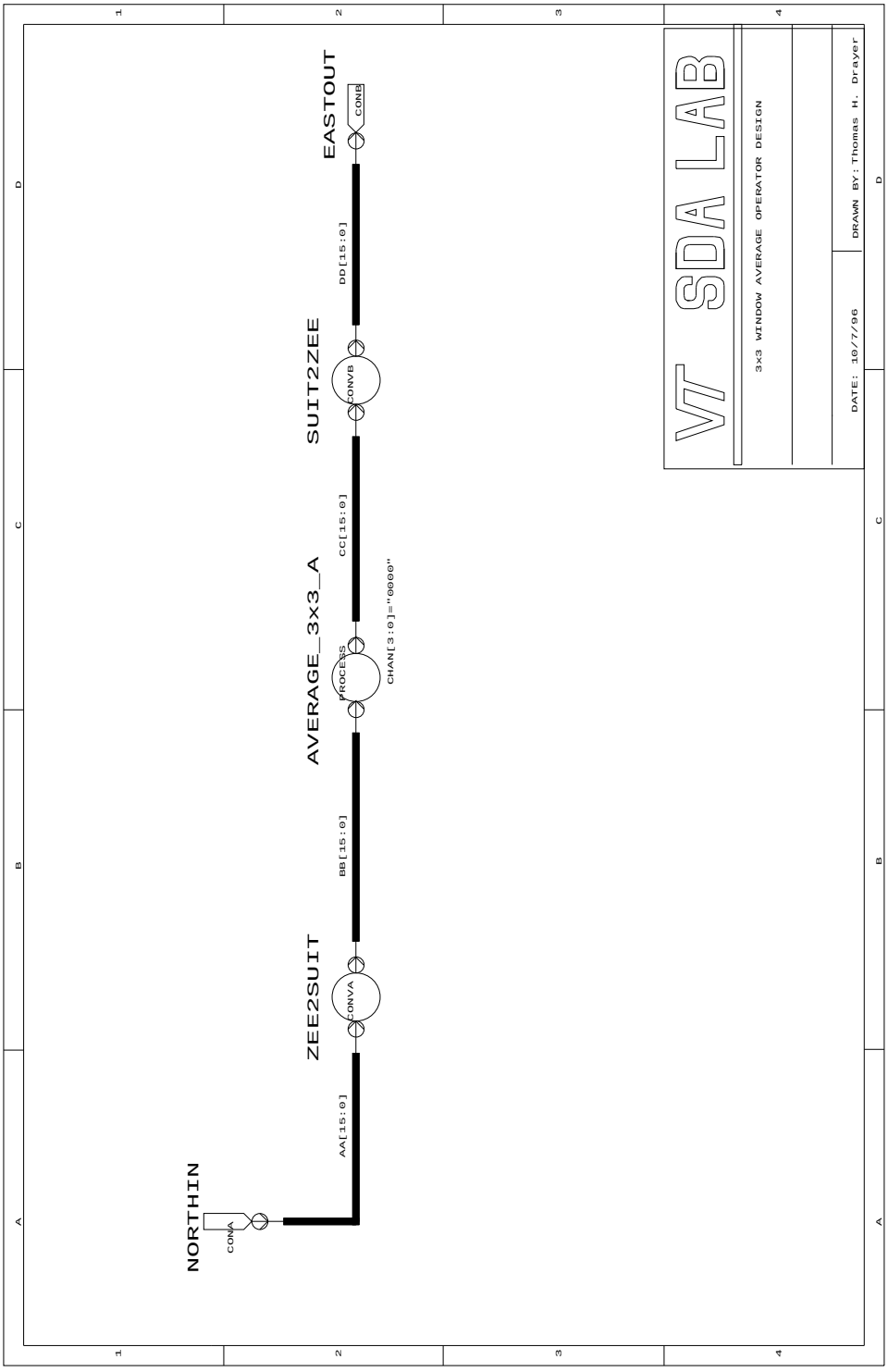


Figure 5.1-2. Schematic for 3x3 average window operator image processing design.

All three of these basis image processing designs use two FPGA chips, in locations X20 and X21 (see Figure 3.3-1) of the MORRPH processing array. The offset and Laplacian image processing designs are implemented using two Xilinx 4003HPG223 FPGA chips with 3000 equivalent gates. The average image processing design also uses a 4003HPG223 FPGA chip in X20 but requires a 5000 equivalent gate 4005HPG223 FPGA chip in location X21. The offset image processing design requires no support chips, but the Laplacian and average image processing designs both require two 2Kx8 SRAM chips for intermediate storage of image data.

After chip assembly on a MORRPH-ISA board, individual designs are verified. The MDBUG program performs in-circuit emulation to verify the functionality of the compiled designs.

Output result images for all three basis image processing designs are shown in Figure 5.1-3. A programmable offset value of 80 was used to create the offset image of Figure 5.1-3b. Since the overflow condition is ignored by the **OFFSET_A** module, output values above 255 wrap around zero. Effects of ignoring the overflow condition are evident in Figure 5.1-3b. The Laplacian and average image processing designs provide the desired edge detection and low-pass filtering operations. The top and left edge initial conditions are ignored by the 3x3 window operators, which is also evident in Figure 5.1-3c and Figure 5.1-3d. These images were collected after processing by the MORRPH-ISA board using the compiled designs of this section to verify the correct operation of all three image processing designs.

a) original

b) offset

c) 3x3 Laplace

d) 3x3 average

Figure 5.1-3. Result images of single-module image processing designs.

5.1.3 Morphological Image Processing Design

Two important morphological operators are erosion and dilation. Both of these operations are performed on bit-mapped images. The erosion function can be implemented using the logical AND of all bits in a window of the input bit-mapped image. Similarly, the dilation function can be implemented using the logical OR of all bits in a window of the input image. These operators are typically performed in succession, to preserve the original size of regions in the input bit-mapped image.

A fourth basis image processing design created for morphological operations of gray scale images is presented in Figure 5.1-4. After conversion to the SUIT bus format, the **THRESHOLD_A** module uses a programmable threshold to create a bitmapped image using the ROI bit of the SUIT bus. This bitmapped image is an estimate of the location of regions within the image.

The erosion and dilation processes are applied sequentially by the **EROSION_A** and **DILATION_A** image processing modules. The erosion-dilation operation removes small isolated regions from the image. This is followed by the dilation-erosion operation to merge adjacent regions and fill small voids within regions.

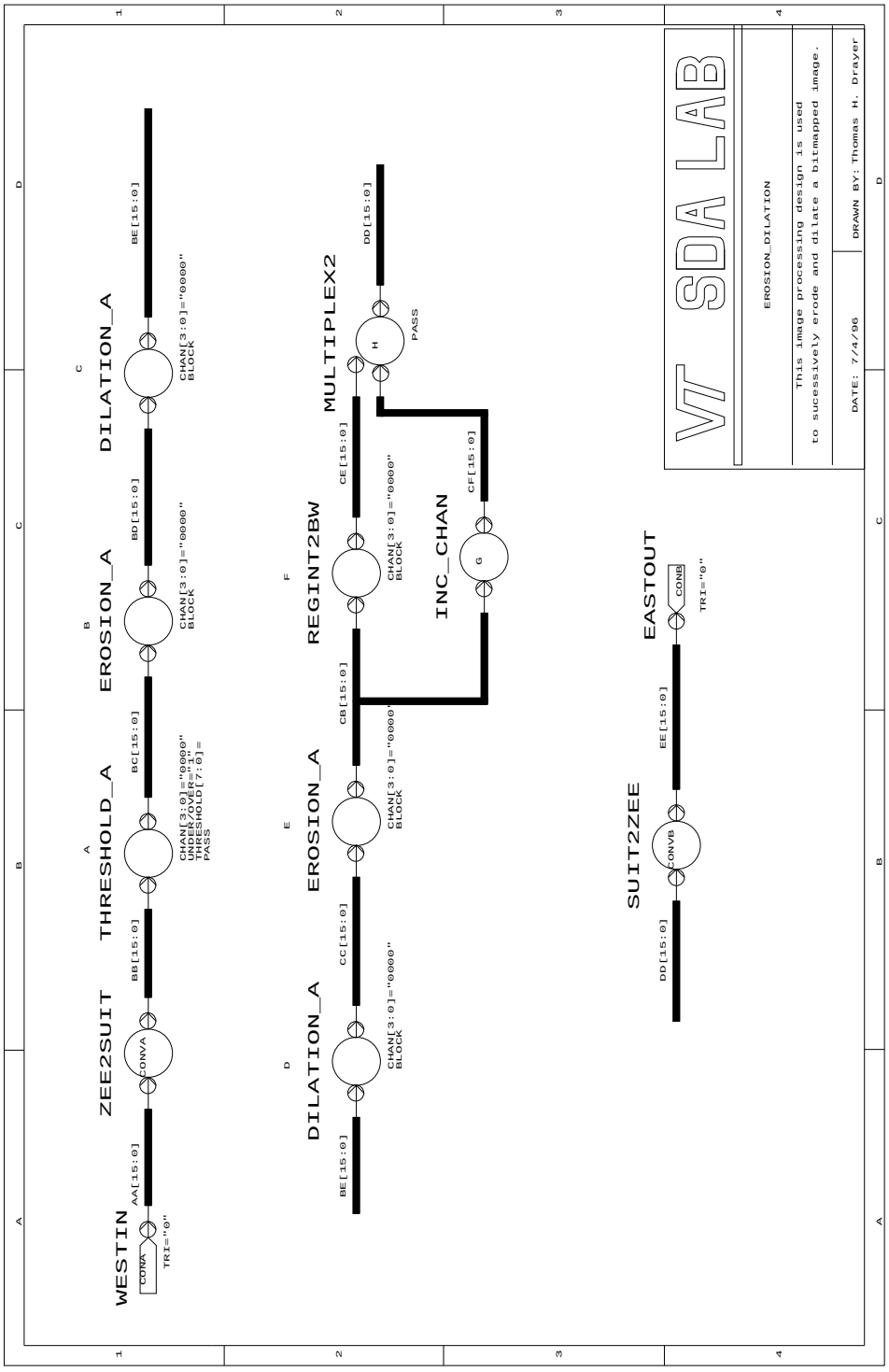
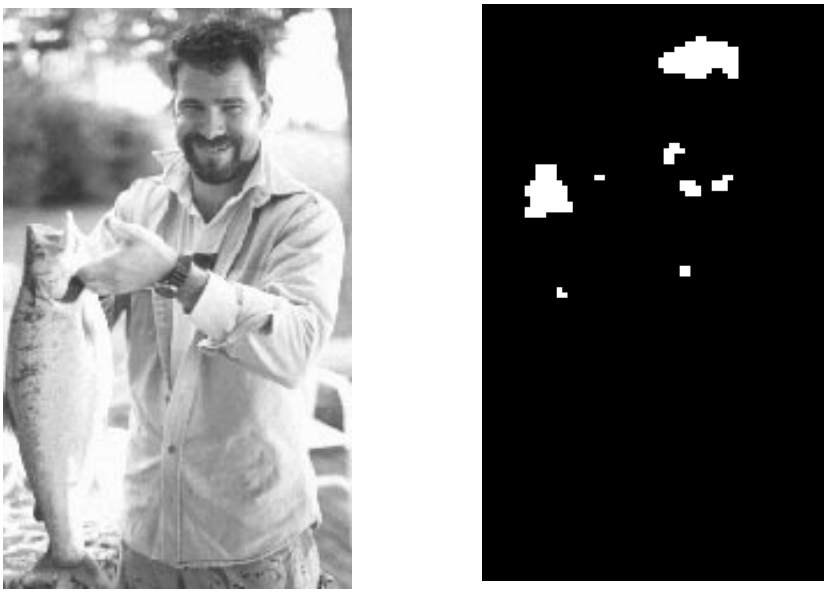


Figure 5.1-4. Schematic for morphological image processing design.

The ROI bit of the SUIT bus must be transformed into a separate image for collection and display. The **REGINT2BW** module creates a bitmapped image on channel 0 of its output SUIT bus from the ROI bit of the input bus. The original image is moved from channel 0 to channel 1 by the **INC_CHAN** module and combined with the bit-mapped image onto a single output bus by the **MULTIPLEX2** module

Data is obtained from the North I/O bus, processed with FPGAs X20 (a 5000 equivalent gate 4005HPG223) and X21 (an 8000 equivalent gate 4008PG191) of the MORRPH array, and output on the East I/O bus. No support chips are required for this design. The **SUIT2ZEE** and **ZEE2SUIT** modules are used for inter-board communication as before. This design is verified as before with the in-circuit emulation provided by the MDBUG program.

The original input and output result images are presented in Figure 5.1-5. All input pixel values below the programmable threshold value of 80 are set in the bitmapped image. This bitmapped image is used by the morphological erosion and dilation operations. Morphological processing of the bitmapped image finds several dark regions in the original image, corresponding the fish's gills, the subject's beard and hair, and a group of trees in the background.



a) original b) after morphological processing

Figure 5.1-5. Result images from morphological image processing design.

5.1.4 Gaussian Pyramid Image Processing Design

An image pyramid is a sequence of images in which the sample density and resolution of successive images is reduced by a constant factor. An illustration of a 4-level image pyramid is shown in Figure 5.1-6. Each successive image is generated with two operations. First, a low-pass convolution reduces the image resolution. A Gaussian window operator is used for the low-pass convolution operator to create a Gaussian image pyramid. Subsampling is then used to reduce the sample density. The varying resolutions provided by an image pyramid are useful for identifying image features that may be not be evident at higher or lower resolutions.

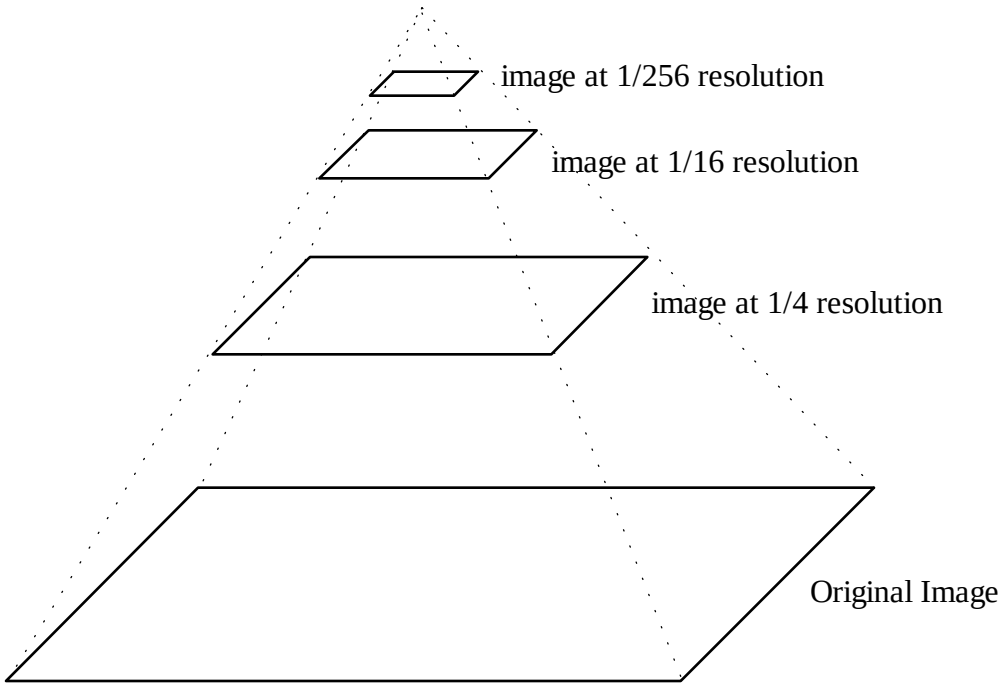


Figure 5.1-6. Representation of 4-level image pyramid.

The new design methodology is used to create and verify a fifth basis image processing design that creates a 4-level image pyramid of an input gray scale image. The bottom image in the pyramid is the original input image. Three new images are created at successively lower sample densities and resolutions. The image processing design created to implement this function is shown in Figure 5.1-7.

A single module, **GAUSSIAN_3x3_A**, is used to reduce the resolution of images in the pyramid. This module implements the 3x3 Gaussian window operator defined in Section 1.2 and shown in Figure 1.2-3. Two image processing modules combine to reduce the sample density of

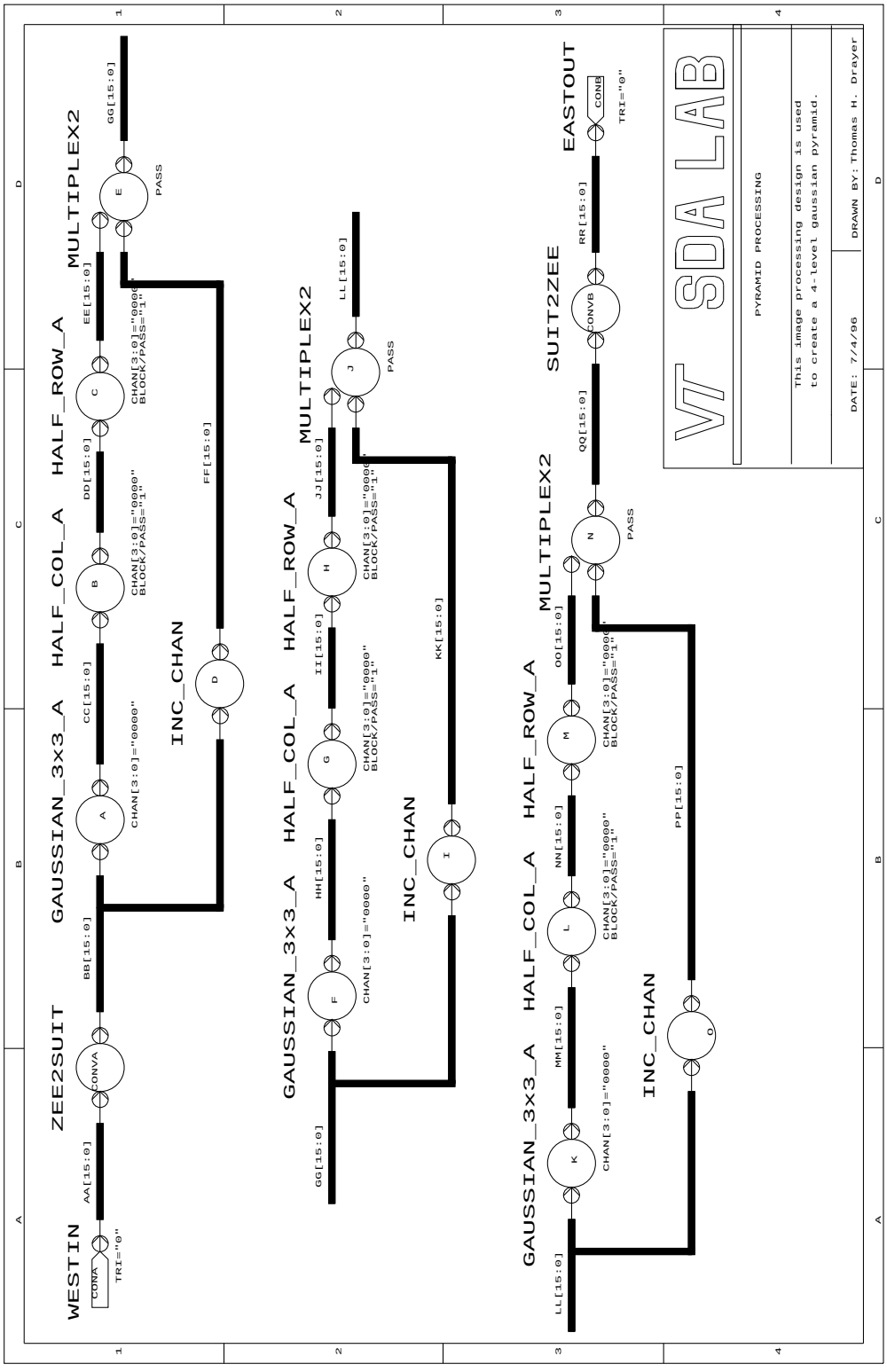


Figure 5.1-7. Schematic for 4-level Gaussian pyramid image processing design.

the input image. The first module called **HALF_COL_A** creates an output image with half the number of columns in the input image. Similarly, a second module called **HALF_ROW_A** creates an output image with half the rows of the original image. Successive processing of an image with these two modules produces an output image with half the number of columns and rows in the original image, with one fourth the number of pixels in the original image.

The three transformation modules defined above replace the input image with the processed image data on their output SUIT busses, using the channel defined by the "CHAN[3:0]=" attribute for processing. Information on all other channels is discarded. This is evident from the "BLOCK" attribute attached to each of the three processing symbols in Figure 5.1-7.

A separate path is provided to preserve the input image data for output from the image processing design. This data multiplexed in with processed data using the **MULTIPLEX2** module. Since the processed data uses the same channel as the input image data, all higher resolution images are moved to the next channel by the **INC_CHAN** module. At the output bus, the highest resolution image has been moved to channel 3 and lower resolution images are available on channels in descending order down to channel 0.

Image data is input from the East I/O bus, processed by the three lower FPGAs in the MORRPH array (X00, X10, and X20), and output on the West I/O bus. Each PE requires an 8000 equivalent gate Xilinx 4008PG191 FPGA chip and two 8Kx8 SRAM support chips for intermediate image data storage. The **SUIT2ZEE** and **ZEE2SUIT** modules are used for inter-board communication as before.

Result images obtained from the in-circuit emulation are presented in Figure 5.1-8. The decreasing image resolution and sample density expected for pyramid processing function are both evident in Figure 5.1-8.

Figure 5.1-8. Result images from the Gaussian pyramid image processing design.

5.1.5 Multiple Input Statistical Image Processing Design

Statistical image processing calculates the properties of pixel values within defined regions of an image. After the region of an object is located in the input image, a histogram of the pixel values in this region is typically calculated. This histogram is used by high-level image processing routines to identify the object represented by the region or further segment the image into sub-regions.

The final basis image processing design for statistical image processing is intended for use in a machine vision system to identify defects in rough-cut lumber [77]. Image data is collected from three separate input imaging technologies: transmission X-ray, laser profile, and color image data. A programmable field-of-view is desired for each of the image technologies. Both X-ray and color image data must be shade corrected after collection. The boundary of a board is located in each of the three input imaging technologies. Using this boundary information, histograms of pixel values in the board region are created from the X-ray and color image data. All three processed images and the two histograms are required outputs of this image processing design. The image processing design created to implement this multiple input statistical image processing function is shown in Figure 5.1-9, Figure 5.1-10, and Figure 5.1-11.

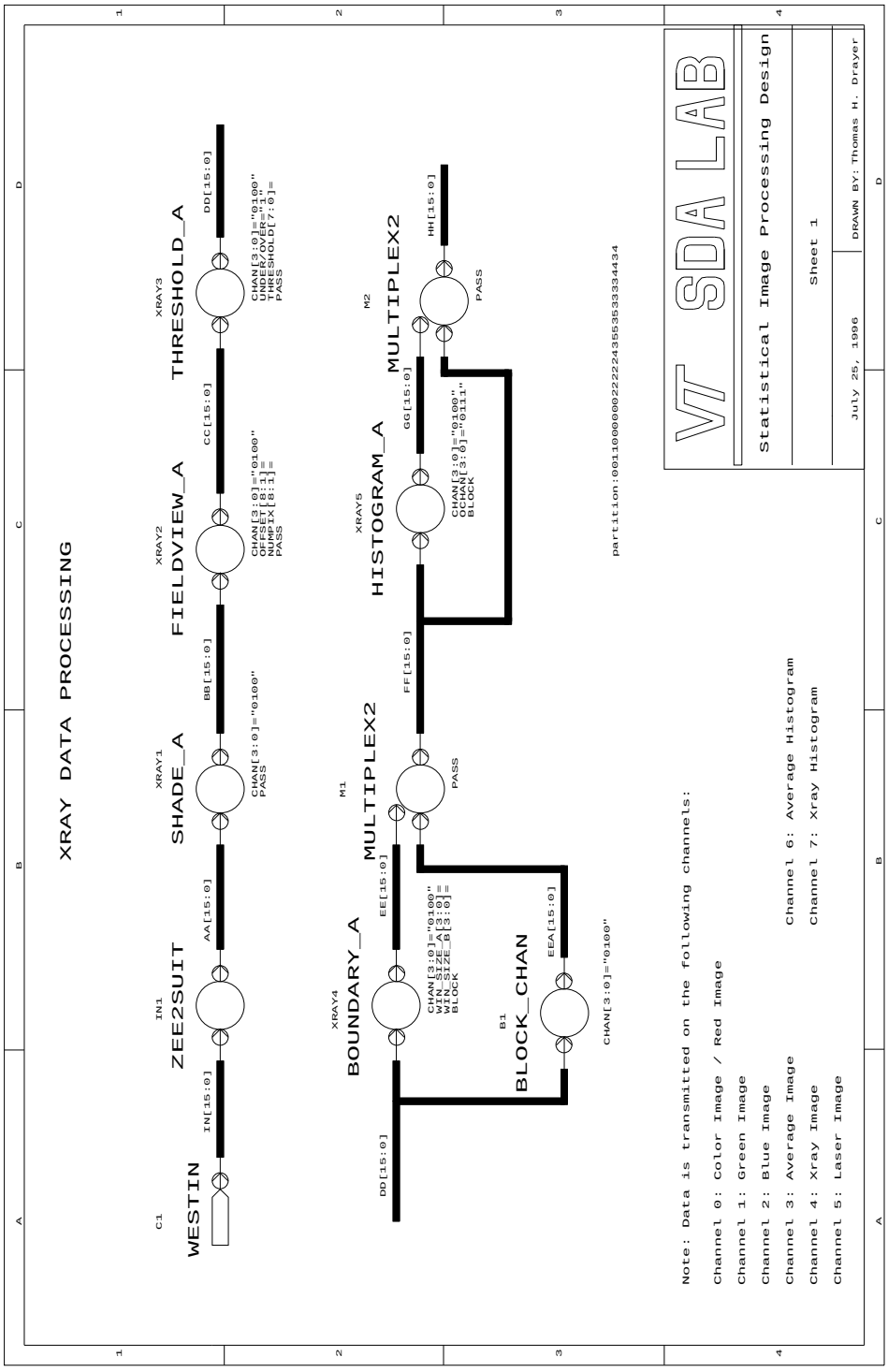


Figure 5.1-9. Sheet one of statistical image processing design.

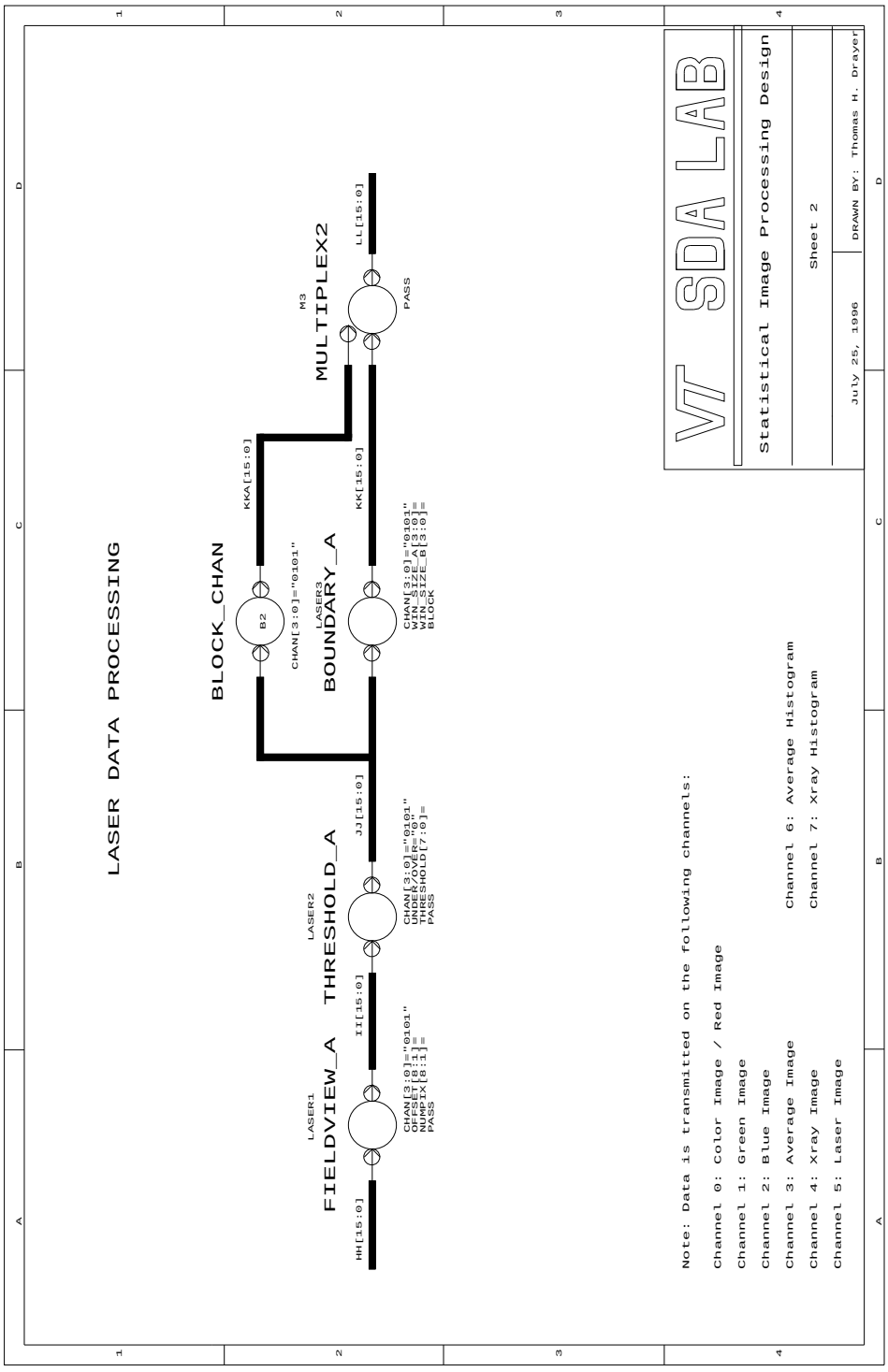


Figure 5.1-10. Sheet two of statistical image processing design.

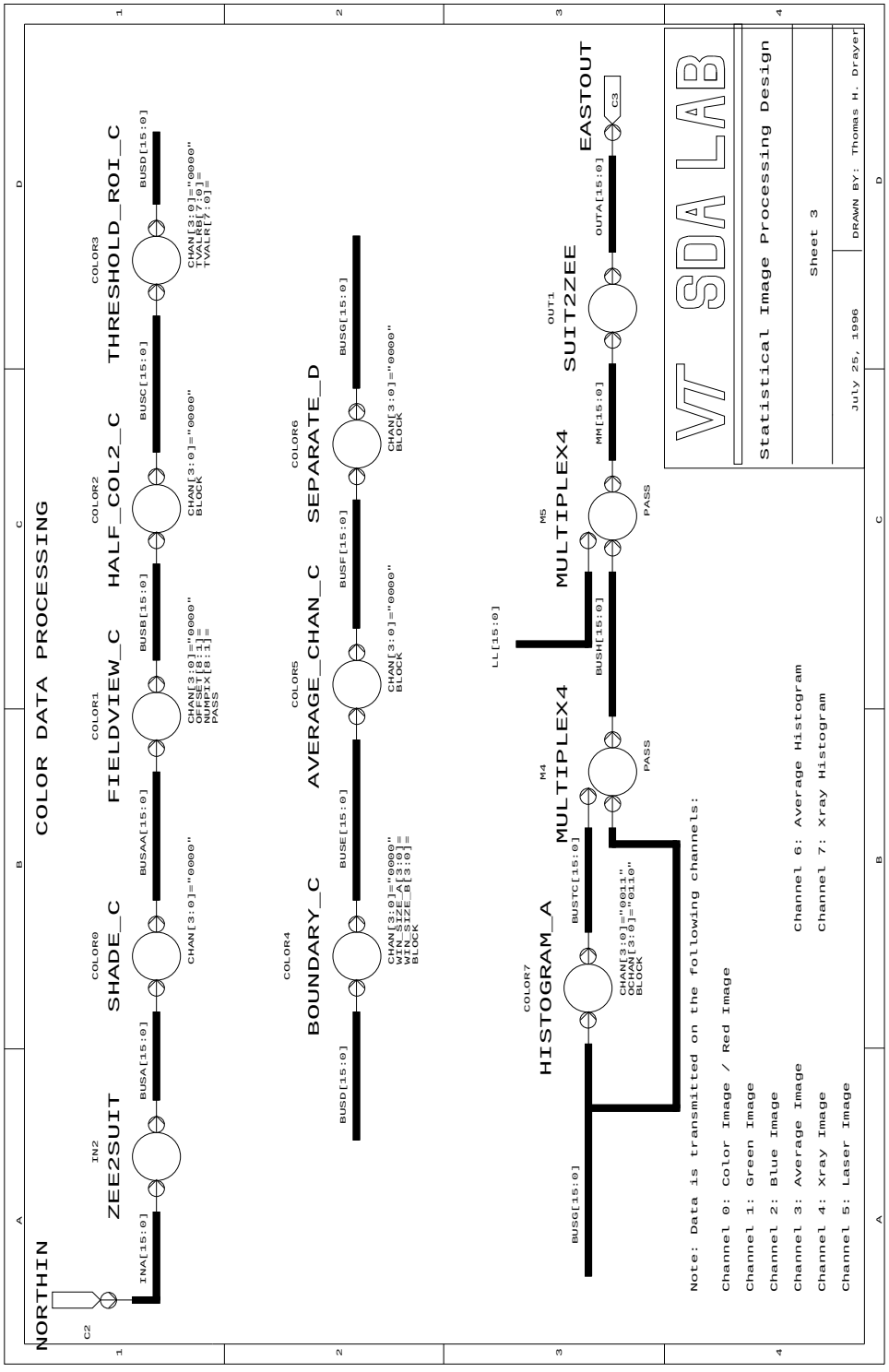


Figure 5.1-11. Sheet three of statistical image processing design.

X-ray and laser profile image data are input from the West I/O bus of the MORRPH-ISA board. These two image channels are time-multiplexed on a single input bus in the ZEE bus format. X-ray image data is presented on channel 4 and laser profile image data is on channel 5.

The X-ray image data on channel 4 is shade corrected with the **SHADE_A** module and its field-of-view is limited by the **FIELDVIEW_A** module. An estimate of the boundary of a board in the input image is identified by the **THRESHOLD_A** module. This estimated region is used to determine the boundaries of the board by the **BOUNDARY_A** module.

The **BOUNDARY_A** module has the attribute "BLOCK" while all previous modules used for X-ray processing have the attribute "PASS". Therefore, the laser profile image data must be passed around the **BOUNDARY_A** module and multiplexed with the processed X-ray image data. Image data on channel 4 (X-ray image without boundary processing) is removed from this feed-forward path by the **BLOCK_CHAN** module. In a similar manner, the X-ray and laser profile image data are both passed around the **HISTOGRAM_A** module.

Laser profile image data on channel 5 is processed similarly to X-ray image data. However, laser profile image data does not require shade compensation and a histogram of laser profile pixel values is not needed. Only the **FIELDVIEW_A**, **THRESHOLD_A**, and **BOUNDARY_A** modules are required to process laser profile image data. The **BLOCK_CHAN** and **MULTIPLEX2** modules are included as shown previously for X-ray image processing to pass X-ray image data around the **BOUNDARY_A** module and block laser profile image data that is replaced with data from the B**OUNDARY_A** module.

Color image data is input on channel 0 from the North connection of the MORRPH-ISA board and processed in the same manner as the X-ray image data. However, since color image data contains three bytes of data for each pixel (i.e., red, green, and blue) the **SHADE_C**, **THRESHOLD_C**, and **BOUNDARY_C** modules from the SUIT2C module library are used. Input color images have twice the number of columns as the other two image technologies due to the physical sensors and imaging geometry, so the **HALF_COL2_C** module is used to reduce the spatial dimension of the color image data.

A histogram of the 3-byte color image data represents over 24 million values. To reduce the size of the color image histogram, the average of the red, green, and blue values for each pixel is calculated by the **AVERAGE_CHAN_C** module. This module creates a 4-byte data structure on its output bus. The **SEPARATE_CHAN_D** module creates four independent 1-byte image channels from the input 4-byte image. The average channel data (channel 3) is used to create a 256-value histogram using the **HISTOGRAM_A** module.

Processed image data from all three imaging modalities is combined onto a single output SUIT bus by the **MULTIPLEX4** module. This bus is converted to the ZEE bus format and output on the East I/O bus of the MORRPH-ISA board.

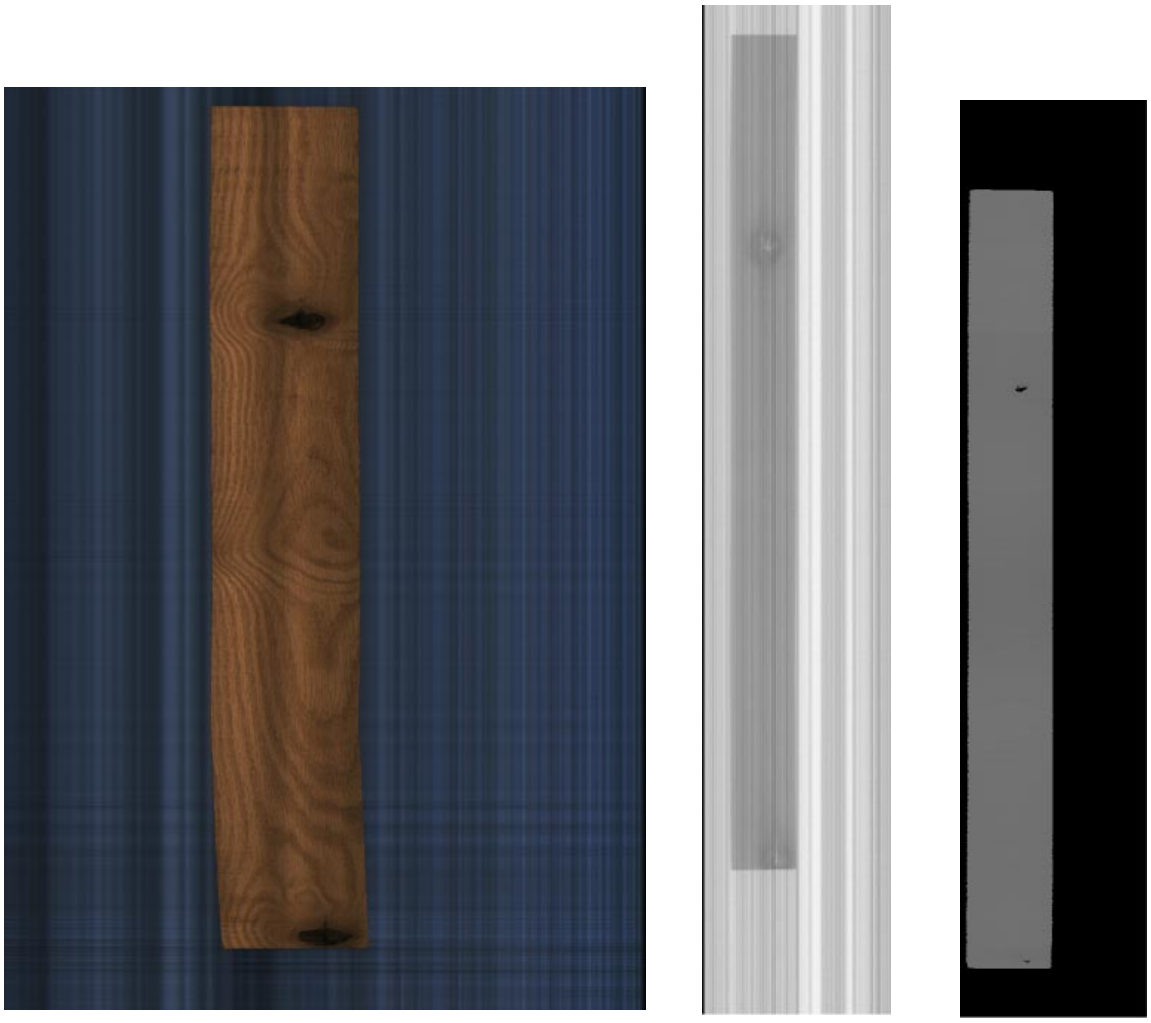
All six available FPGA sockets are used to implement the statistical image processing design. Two 8000 equivalent gate Xilinx 4008PG191 FPGA chips are required for PE locations X01 and X20 in this design, all other PEs contain 10000 equivalent gate 4010PG191 FPGA chips.

This design is verified as before with the in-circuit emulation provided by the MDBUG program. The original input images are shown in Figure 5.1-12 and result processed images are shown in Figure 5.1-13. Histograms of the X-ray and color image data are presented in Figure 5.1-14.

Vertical lines caused by non-uniform sources and sensors are evident in the X-ray and color input images of Figure 5.1-12. These lines are removed by shading compensation and are not evident in the output result images of Figure 5.1-13. The FOV operation reduces the vertical dimension of all output result images. Additionally, the spatial dimension of the output result color image has been reduced to 1/2 the dimension of the input color image to correspond with the spatial resolution of other image technologies. Leading and lagging edges of the board are determined by the boundary operation as indicated by the horizontal cropping of all three result images.

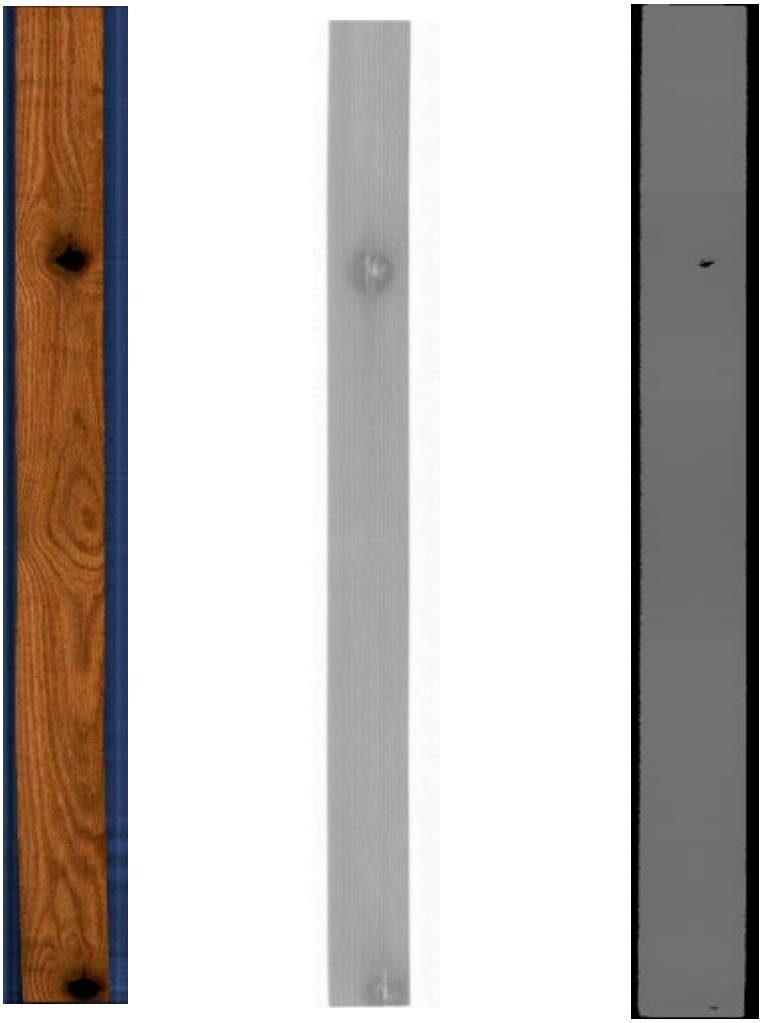
The histograms of Figure 5-1-14 show that the pixel values of the board region are brighter in the X-ray image than in the color image data. The distribution of values indicates that the color image values provide a larger variance than the X-ray pixel values. Pixel values that correspond to the two knots of the board are indicated by plateaus at the lower end of both histograms.

This image and histogram data is provided to high-level algorithms for further processing. The high-level algorithms could identify the plateaus in the histogram to locate pixel values in the result images that correspond to defect areas such as the two knots. The rich information provided by three image technologies is used to identify the type of each defect in the board.



a) Color b) X-ray c) Laser

Figure 5.1-12. Input images for the statistical image processing design.



a) Color b) X-ray c) Laser

Figure 5.1-14. Output images of the statistical image processing design.

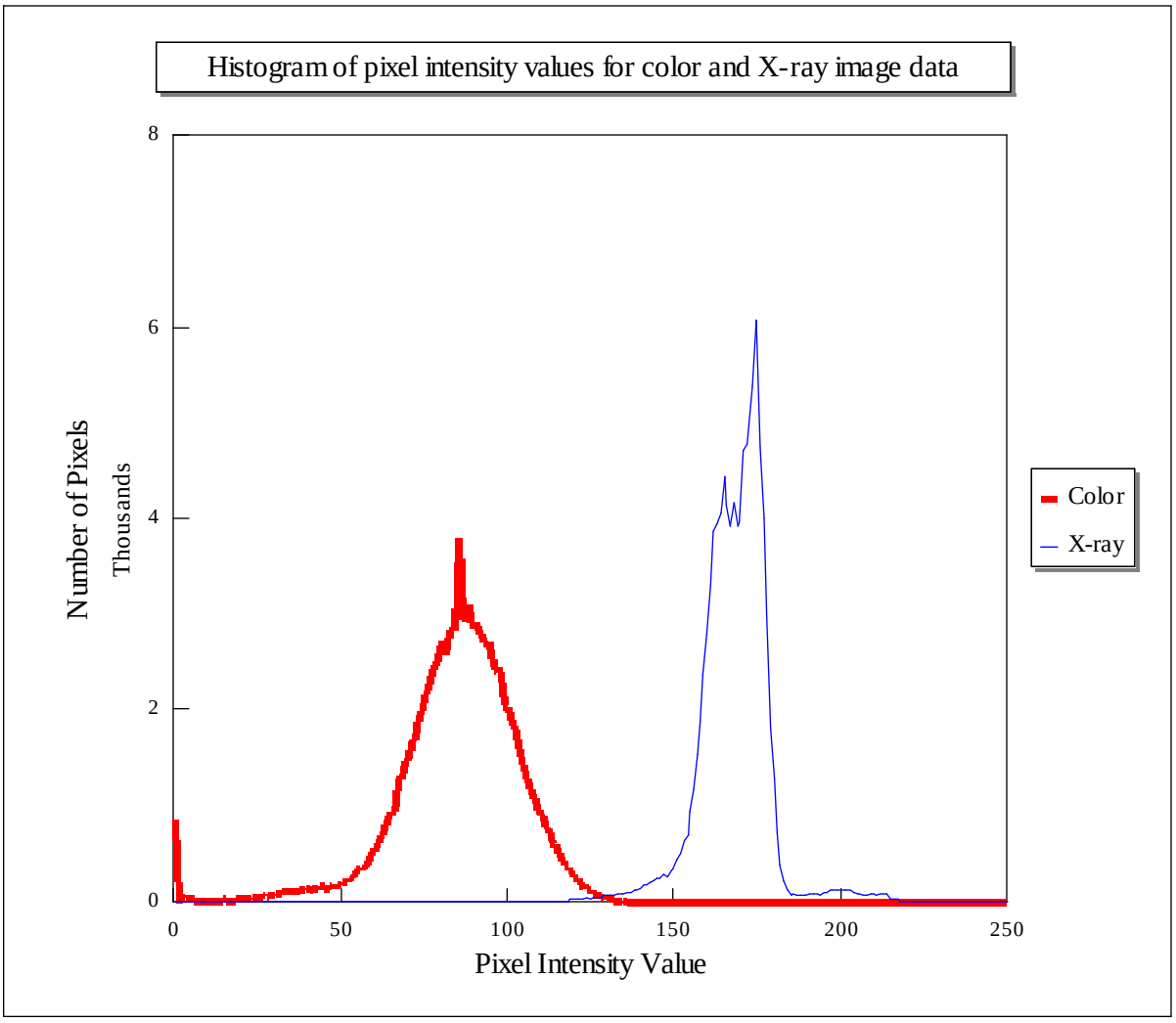


Figure 5.1-15. Output histograms of the statistical image processing design.

5.2 Analysis of the Design Methodology

All of the six basis image processing designs defined in the previous section have been translated into the resources of the MORRPH-ISA board. Compilation output from the translation process is used in this section to determine the efficiency and performance of the new design methodology described in this dissertation.

Efficiency is determined by analyzing the utilization of three essential FPGA resources, FFs, FGs, and IOBs. This information is obtained from the corresponding output report files of the Xilinx PPR program when each of the basis image processing designs is translated into the resources of the MORRPH-ISA board. The amount of each of these three types of FPGA resources used by the six basis image processing designs is presented in Table 5.2-1.

Circuits are partitioned such that the FPGA resources required to implement the logic assigned to each group does not exceed the amount provided by a corresponding FPGA of the destination hardware architecture. As mentioned previously in Section 4.4.3, image processing designs are partitioned by the development system software of this design methodology in a manner that does not exceed the amount of resources available in the largest desired FPGA chip to be used in each PE. Therefore, the FPGA resource requirements of Table 5.2-1 are used after partitioning to determine the smallest size of FPGA to be assembled into each PE of the MORRPH array for a specific image processing design.

Table 5.2-1 Compilation results for six image processing designs.

OFFSET	PE	FPGA	FGs	FFs	IOBs
	X20	4003HPG191	123	147	67
	X21	4003HPG191	31	55	50
		total	154	202	117
LAPLACE	PE	FPGA	FGs	FFs	IOBs
	X20	4003HPG191	123	147	67
	X21	4003HPG191	167	150	80
		total	290	297	147
AVERAGE	PE	FPGA	FGs	FFs	IOBs
	X20	4003HPG191	123	147	67
	X21	4005HPG223	336	249	80
		total	459	396	147
MORPHOLOGICAL	PE	FPGA	FGs	FFs	IOBs
	X20	4005HPG223	381	316	68
	X21	4008PG191	490	206	51
		total	871	522	119
PYRAMID	PE	FPGA	FGs	FFs	IOBs
	X00	4008PG191	450	351	80
	X10	4008PG191	443	320	77
	X20	4008PG191	556	460	80
		total	1,449	1,131	237
STATISTICAL	PE	FPGA	FGs	FFs	IOBs
	X00	4010PG191	638	487	134
	X10	4005HPG223	283	177	131
	X20	4008PG191	440	340	87
	X01	4010PG191	683	428	117
	X11	4008PG191	559	333	108
	X21	4005HPG223	288	305	50
		total	2,891	2,070	627

A plot of the total amount of the three FPGA resources required by each of the basis image processing designs is provided in Figure 5.2-1. This figure illustrates the ordering by size of the six basis image processing designs. The image processing designs are ordered to increase in the required amount of FGs, FFs, and IOBs. IOB requirements of the morphological image processing design provide the largest variation from this trend because this design does not use any support chips. This ordering of designs is used to predict trends in FPGA resource utilization based on the size of the image processing designs in the remainder of this section.

The two components of this new design methodology are the hardware architecture and development system software. Six image processing designs defined in the previous section provide a basis for individually evaluating the efficiency and performance of both components of the design methodology. The amount of overhead circuitry added to an image processing design by the development system software is calculated first, followed by an evaluation of the utilization of FPGA resources in the hardware architecture. Finally, an analysis of the obtained and expected pixel processing rates is presented.

5.2.1. Efficiency of the Software Development System

Standard methods of circuit design entry provide the benefits of maintainability, accessibility, and portability. However, the most efficient implementation for any specific processing task is created without the restrictions imposed by standard design methods. Additional circuitry is required to implement designs using standard methods, this circuitry

represents an overhead cost associated with the use of design entry standards. This overhead is minimized by the use of appropriate and flexible design entry standards.

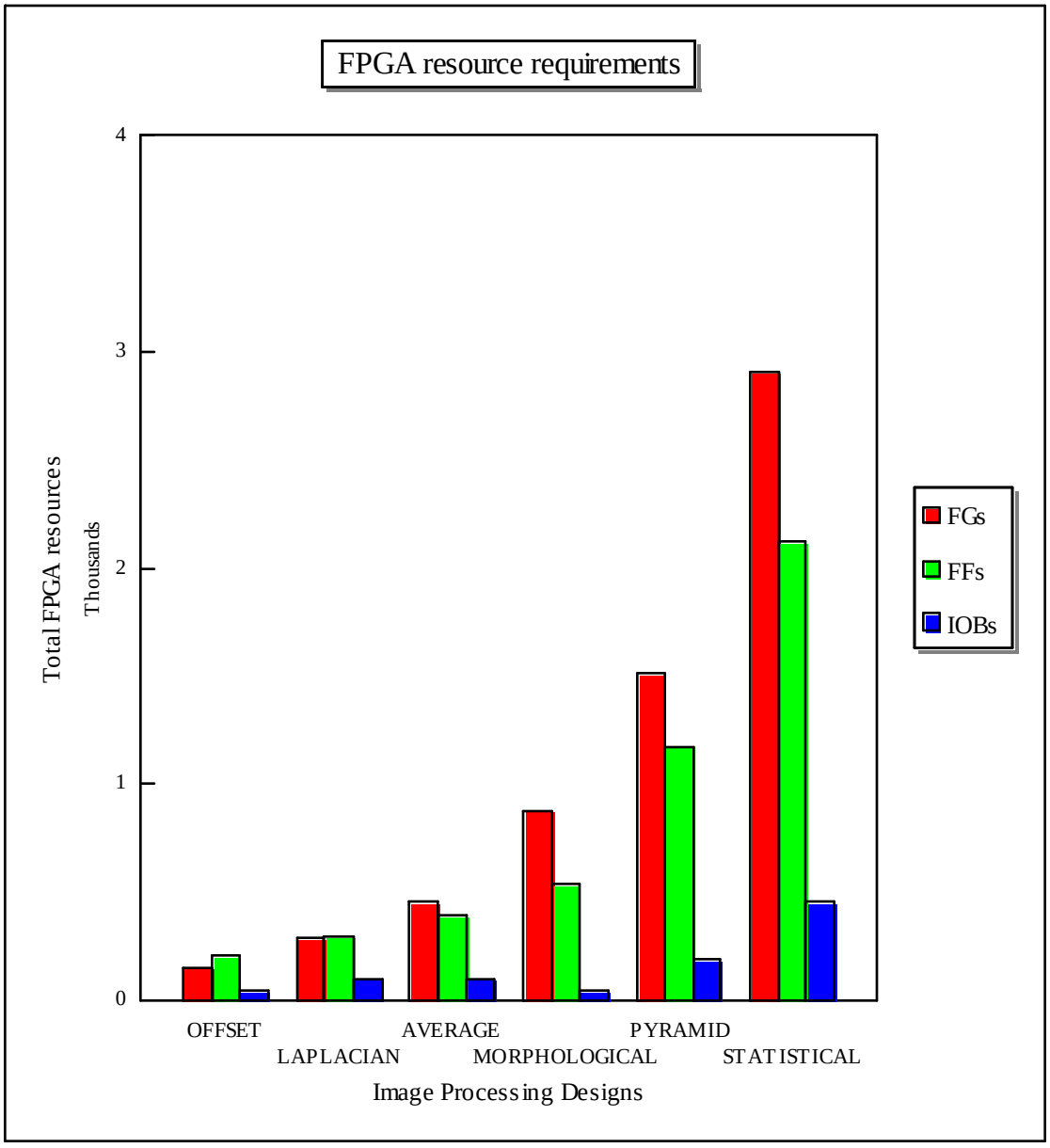


Figure 5.2-1. Total FPGA resource requirements for basis designs.

The new software development system described in this dissertation established a standard method for independently creating low-level image processing modules that are easily interconnected in complicated image processing designs. The SUIT bus standard defines a regular method for the exchange of image and result information. This design entry method allows complicated image processing designs to be easily created, but requires overhead circuitry to allow independently designed modules to be arbitrarily interconnected.

The synchronous transfer of data and commands on the SUIT bus defines a systolic flow of information in image processing designs. Overhead circuitry is required to control this flow of SUIT data and commands. The multiplexing of multiple channels of image and result data allows the extension of image processing designs beyond simple linear pipelined architectures. However, this also requires additional circuitry in order to avoid data collisions. These two sources of additional logic contribute the majority of overhead associated with the design entry method.

Several types of logic within individual modules can be directly attributed to overhead of the design entry methods. First, a transformation module that requires several clock cycles to calculate an output result from an input pixel value delays the 4-bit input commands until the processed data is available at the output bus. This process synchronizes the bus commands with the processed data. Four FFs are required for each additional clock cycle required to calculate the output results.

Additional overhead logic is included within modules to transmit multiple-byte values on the 1-byte data bus of the SUIT bus. Higher order bytes of data are transmitted on successive

clock cycles when the 2-byte, 3-byte, or 4-byte data sizes are used with a 16-bit SUIT bus. This serial transmission of bytes requires an input serial-to-parallel conversion and an output parallel-to-serial conversion. All data must be clocked at the input to a module, so the storage requirements of the input bus do not add to the overhead circuitry. However, higher order bytes must wait for transmission on the output bus when an n-byte result is calculated, requiring (n-1)*8 additional FFs for intermediate storage in a module with an n-byte output data operand size. Since all these bytes are multiplexed on the same output bus, a 8-bit n-to-1 multiplexer is also required for the output SUIT bus connection.

Several modules are included in image processing designs only to control the flow of information on the SUIT bus. These modules are the **MULTIPLEX*n***, **INC_CHAN**, and **BLOCK_CHAN** modules of the SUIT2 library. All logic required to implement these modules represents an overhead cost associated with the design entry methods of this dissertation.

These factors represent the majority of additional overhead logic required to create a design with the design entry methods of the development system software instead of creating a special purpose circuit. Some of this logic may still be required for a special purpose circuit, so these calculations are considered an upper bound on the overhead cost.

All of the individual modules have been manually examined and overhead circuitry has been identified. These values are used to calculate an estimate of the overhead circuitry required by the regular method of interconnecting modules. The data is presented graphically in Figure 5.2-2.

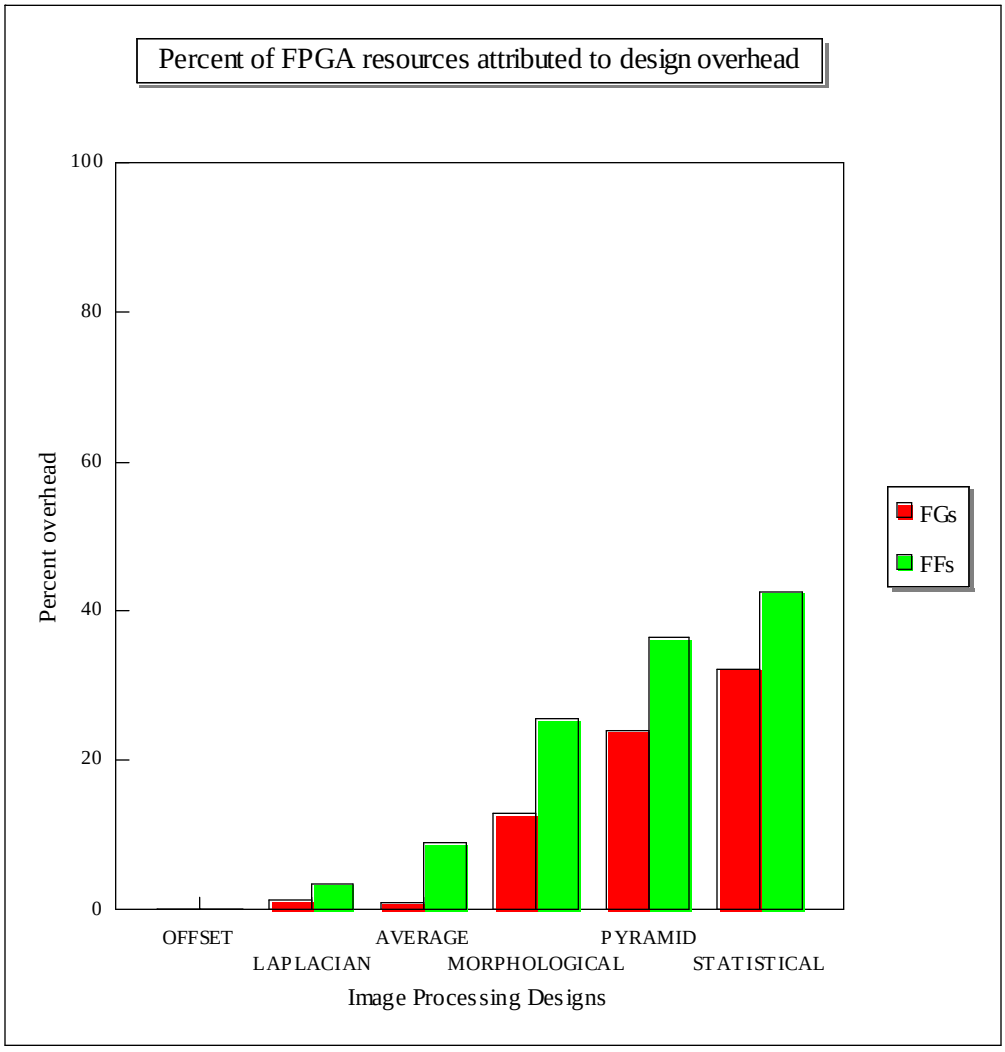


Figure 5.2-2. Overhead circuitry of the development system software for basis designs.

The first three image processing designs are simple linear pipelines that only use one channel of the SUIT bus. These three image processing designs do not contain a significant amount of logic overhead (less than 10%). However, the morphological, pyramid, and statistical

image processing designs transmit image and result data on two, four, and eight channels of the SUIT bus, respectively. The additional SUIT bus channels used by these more complicated image processing designs increases the required overhead circuitry considerably. This relationship between the number of SUIT bus channels and the overhead circuitry is shown graphically in Figure 5.2-3. These trends indicate that the overhead requirements of FFs and FGs may increase to as much as 50% and 40%, respectively, when all 16 channels of the SUIT bus are used.

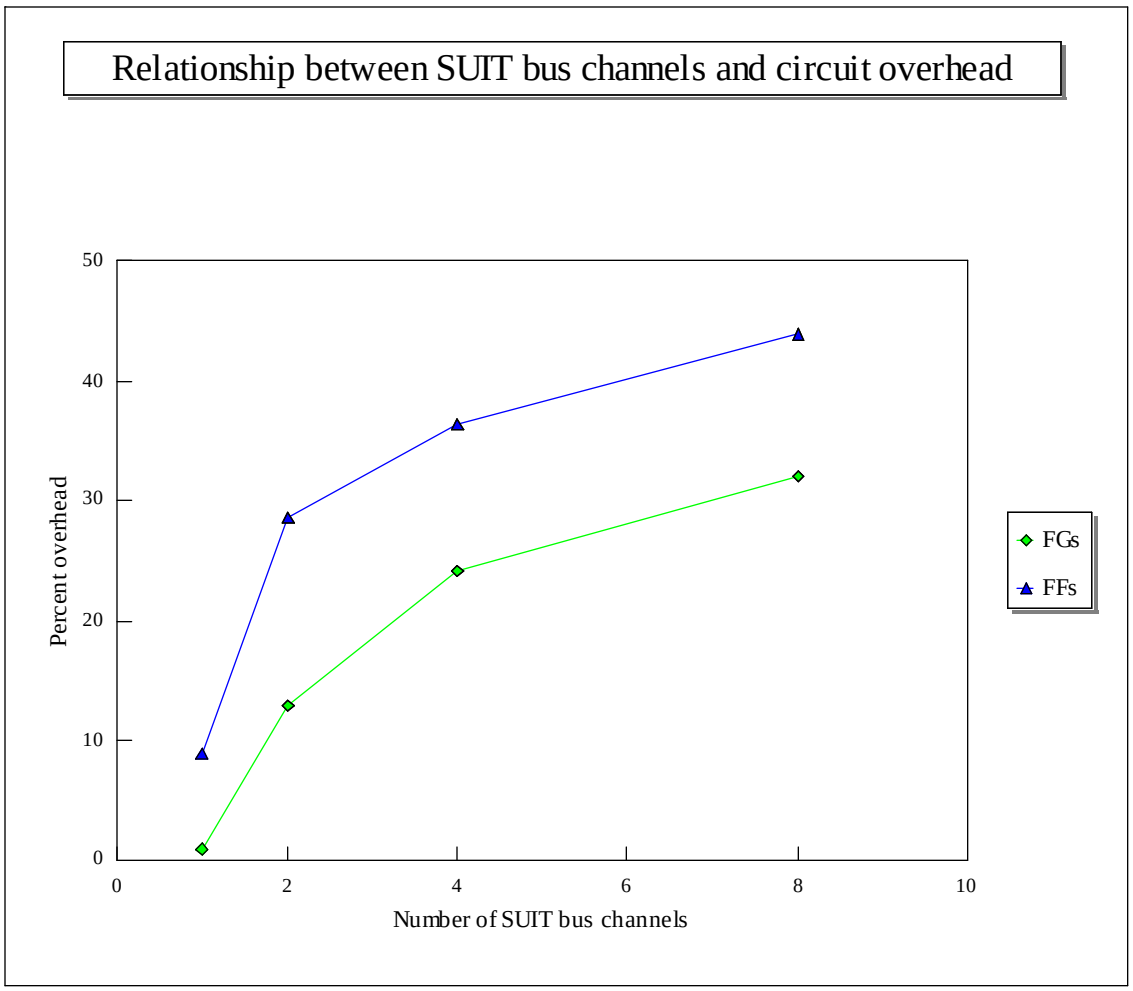


Figure 5.2-3. Relationship between the number of SUIT bus channels and overhead.

This overhead may be reduced by more efficient implementation of the **MULTIPLEX*n*** modules, which are a significant source of overhead. For example, a **MULTIPLEX2** module requires almost as many FFs as both the **EROSION** and **DILATION** modules combined. Additionally, analysis of the queuing theory associated with the systolic flow of data in designs may be used to determine the appropriate size of input buffers on modules such as **MULTIPLEXn**. Further investigation of both these methods is left for future work.

5.2.2. Efficiency of the Hardware Architecture

The second component of the design methodology is the MORRPH hardware architecture. The modular nature of this incompletely specified hardware architecture should provide efficient utilization of the available FPGA resources. This is verified by comparing the FPGA resource utilization of the MORRPH-ISA board with each of the FPGA-based CCMs defined in Section 2.1.3, using the basis image processing designs of the previous section.

FPGA resource utilization is calculated by dividing the total compiled resources of an image processing design by the sum of available resources provided by FPGA chips on the destination hardware architecture. The total compiled resources of each image processing design and the types of FPGA chips required for the MORRPH-ISA board were shown previously in Table 5.2-1. The sum of individual resources provided by FPGA chips on completely specified hardware architectures, such as Splash and CHAMP, are easily calculated from the number and type of FPGA chips in each architecture.

Since the circuits are not compiled onto each of the individual hardware architectures, it is required to verify that the resources of these other FPGA-based hardware architectures are adequate to implement each of the six image processing designs. Three types of resources are required to implement each of the designs on a specific architecture. There must be appropriate support chips, sufficient FPGA resources, and adequate interconnections for the circuit to be compiled into a specific hardware architecture.

The six image processing designs intentionally contain only small SRAM support chips, since SRAM memory chips are available in all four of the other hardware architectures under consideration. FPGA resource requirements provided in Table 5.2-1 are used to verify the second constraint. Finally, the interconnection resources are not verified completely, since there is significant variation in the implementation of FPGA interconnections among the different hardware architectures, such as the use of crossbar switches and PIC chips. The number of available IOBs in each FPGA of the destination architecture is used to verify this final constraint.

Other than the MORRPH, the Giga Operations G800 board represents the only user-configurable FPGA-based hardware architecture. Support chip, FPGA resource, and interconnect requirements of each design are used to determine the number of multi-chip modules required to implement each design. Each multi-chip module contains two Xilinx 4010 FPGA chips. New types of multi-chip modules with different size FPGA chips are recently available from Giga Operations for the G800 board, but these new modules are not considered in this analysis.

The total compiled resources of an image processing design is divided by the sum of individual resources provided by FPGA chips on the destination hardware architecture to determine the FPGA resource utilization. The utilization of available FGs, FFs, and IOBs for the six image processing designs on five FPGA-based CCMs is presented in Figure 5.2-4, Figure 5.2-5, and Figure 5.2-6.

It is evident from these figures that the MORRPH-ISA board provides significantly more efficient utilization of FGs, FFs, and IOBs for all the image processing designs under consideration. The modularity of the Giga Operations board provides more utilization than the completely specified hardware architectures, but generally only about half the utilization of the MORRPH-ISA board.

Completely specified hardware architectures can only increase or decrease the available FPGA resources by adding or removing entire processing boards from the host computer. As the image processing designs become larger, the utilization of FF and FG resources by the completely specified hardware architectures increases. However, the upper limit of utilization will be designs that correspond to the exact amount of resources available on a single board (or an integer multiple thereof). An image processing design that requires slightly more logic than provided on one board will require two boards to implement, reducing the efficiency by fifty percent. The chip-level configurability provided by the MORRPH board will clearly be more efficient than the board-level or module-level configurability provided by other hardware architectures over a large range of image processing designs.

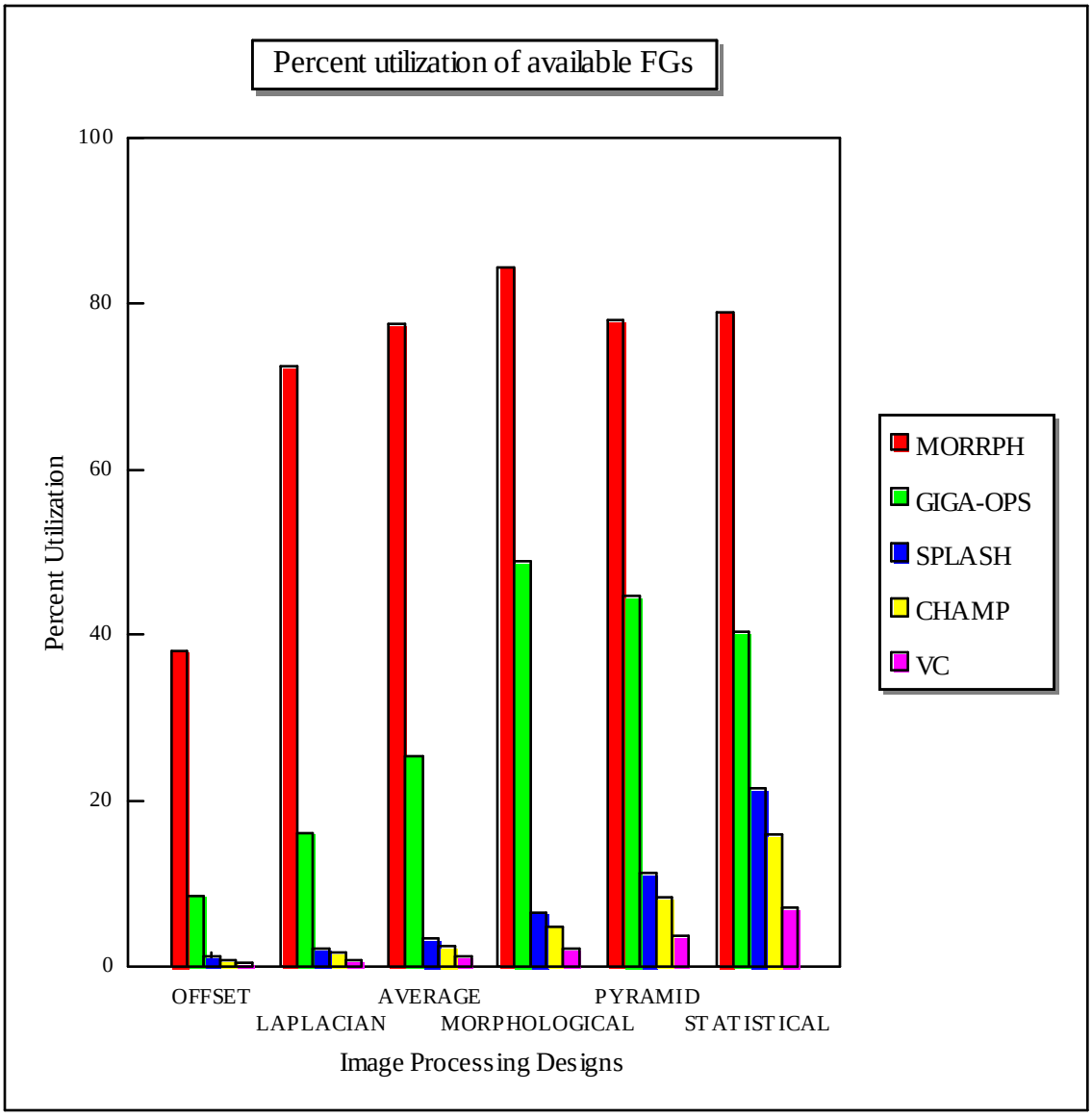


Figure 5.2-4. CLB resource utilization for basis image processing designs.

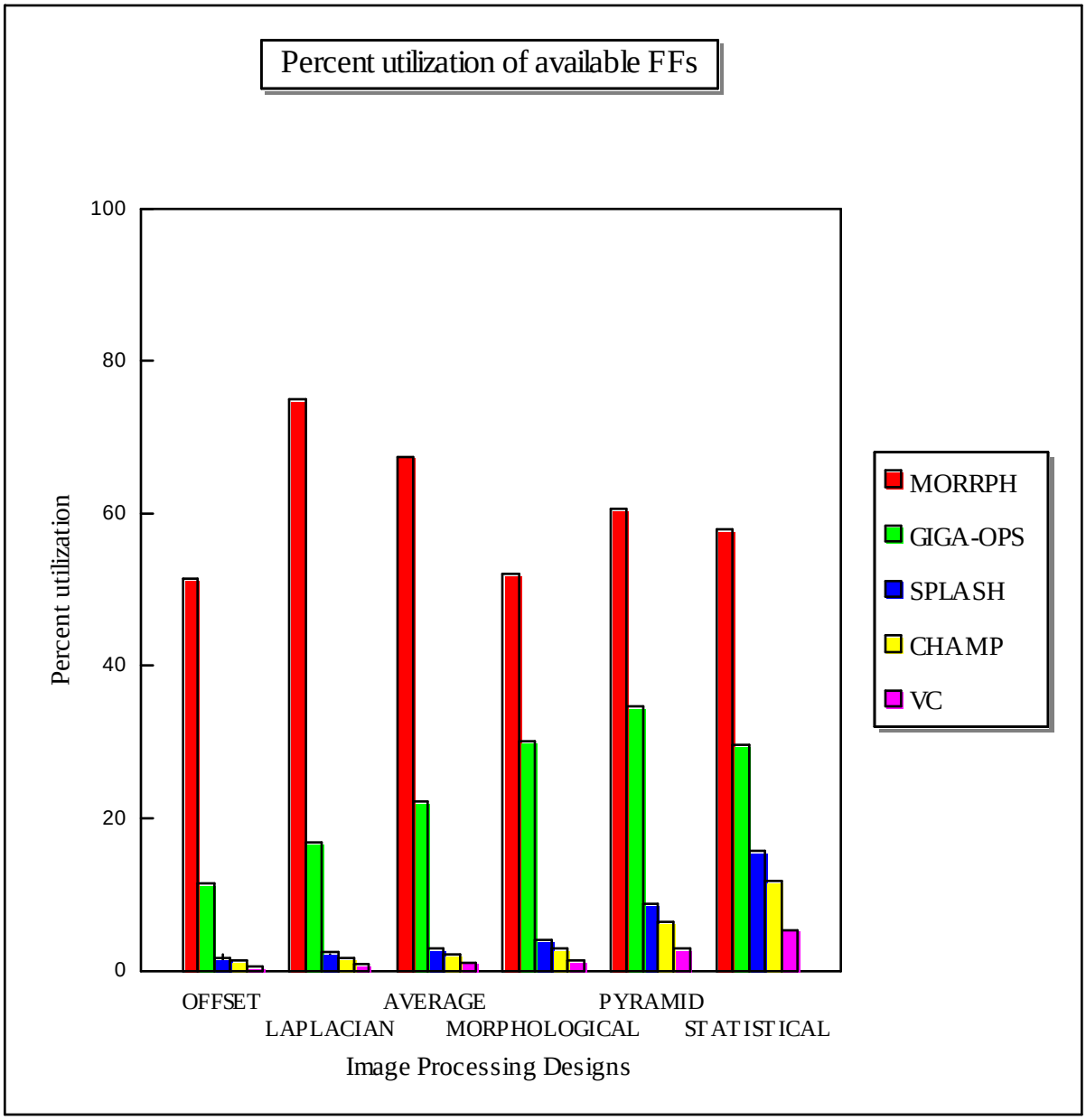


Figure 5.2-5. FF resource utilization for basis image processing designs.

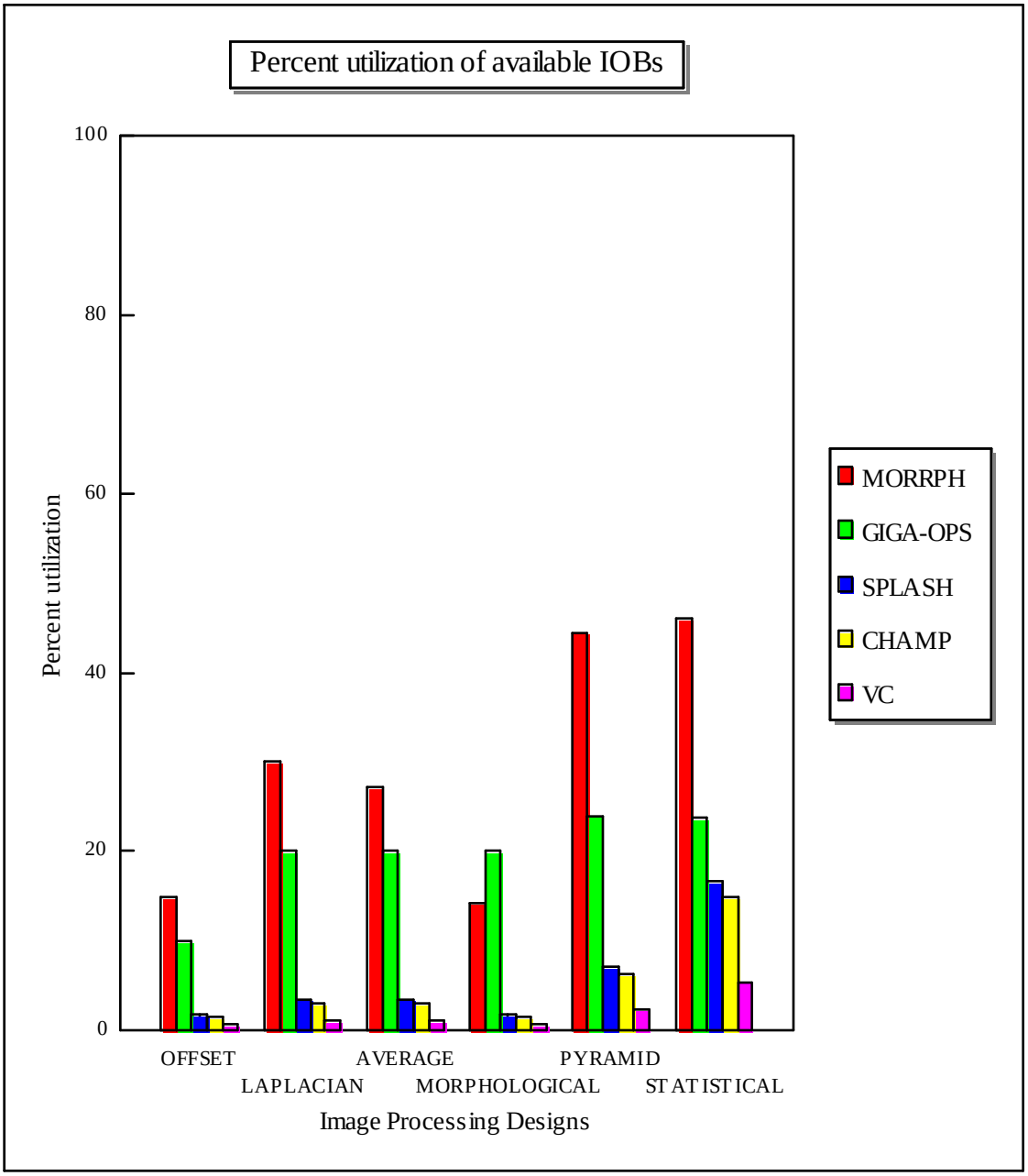


Figure 5.2-6. IOB resource utilization for basis image processing designs.

All image processing designs except the small offset design yield a FG utilization of between 70-85 percent when compiled into the MORRPH-ISA board, as shown in Figure 5.2-4. The reason for the approximate 40 percent utilization of the offset design is that at least two FPGAs are required to input and output data from the 3x2 PE array of the MORRPH-ISA board. Therefore, image processing designs that are limited only by FPGA resources are expected to yield 70-85% utilization, unless constrained by another variable such as the interconnection resources of the MORRPH architecture.

Utilization of FFs is shown in Figure 5.2-5 to be approximately 50-75 percent for basis image processing designs. This is consistently lower than the utilization of FGs because the ratio of required FGs to FFs is higher for the six image processing designs than the fixed 1:1 ratio of FGs to FFs in all Xilinx 4000 series FPGA chips. Therefore, image processing designs are more likely constrained by available FG resources than FF resources. This important result indicates that consideration of FFs in the cost metric of the partitioning software may not be required for most image processing designs.

The 15-45 percent utilization of IOBs shown in Figure 5.2-6 is the lowest utilization of all three types of FPGA resources. Although these numbers are useful for comparison with other architectures and utilization of available resources, the incorrect conclusion that additional interconnection resources are readily available in the MORRPH architecture may be obtained from these numbers. IOBs may be connected to nets used by other FPGAs or may be connected to nets used for interconnection to FPGAs not present in the MORRPH PE array. Therefore, the

utilization of available nets in the MORRPH architecture is presented in Figure 5.2-7 instead of the utilization of IOBs in the individual FPGAs shown in Figure 5.2-6.

Since IOBs are physically dedicated to interconnections of the MORRPH printed circuit board, all IOBs cannot be viewed as interchangeable (like FGs and FFs have been). There are three types of nets in the MORRPH architecture: interconnection nets, I/O nets, and support socket nets. The utilization of each of these types of nets is graphed independently with the total utilization in Figure 5.2-7.

The total utilization of nets in the MORRPH architecture shows the same trends as the IOB usage in Figure 5.2-6. However, the utilization is almost double. Both interconnect and I/O nets show consistently high utilization for all image processing designs. The previous conclusion that the variance in IOB utilization is caused by low utilization of support socket interconnections is clearly substantiated by Figure 5.2-7. Since support socket nets show the lowest utilization, it is reasonable to conclude that some of these IOBs should have been allocated to I/O or interconnect nets. However, additional I/O and interconnect nets are available when larger FPGA chips are used and in many cases the utilization of support chip nets of individual FPGA chips in a multiple FPGA design is very high.

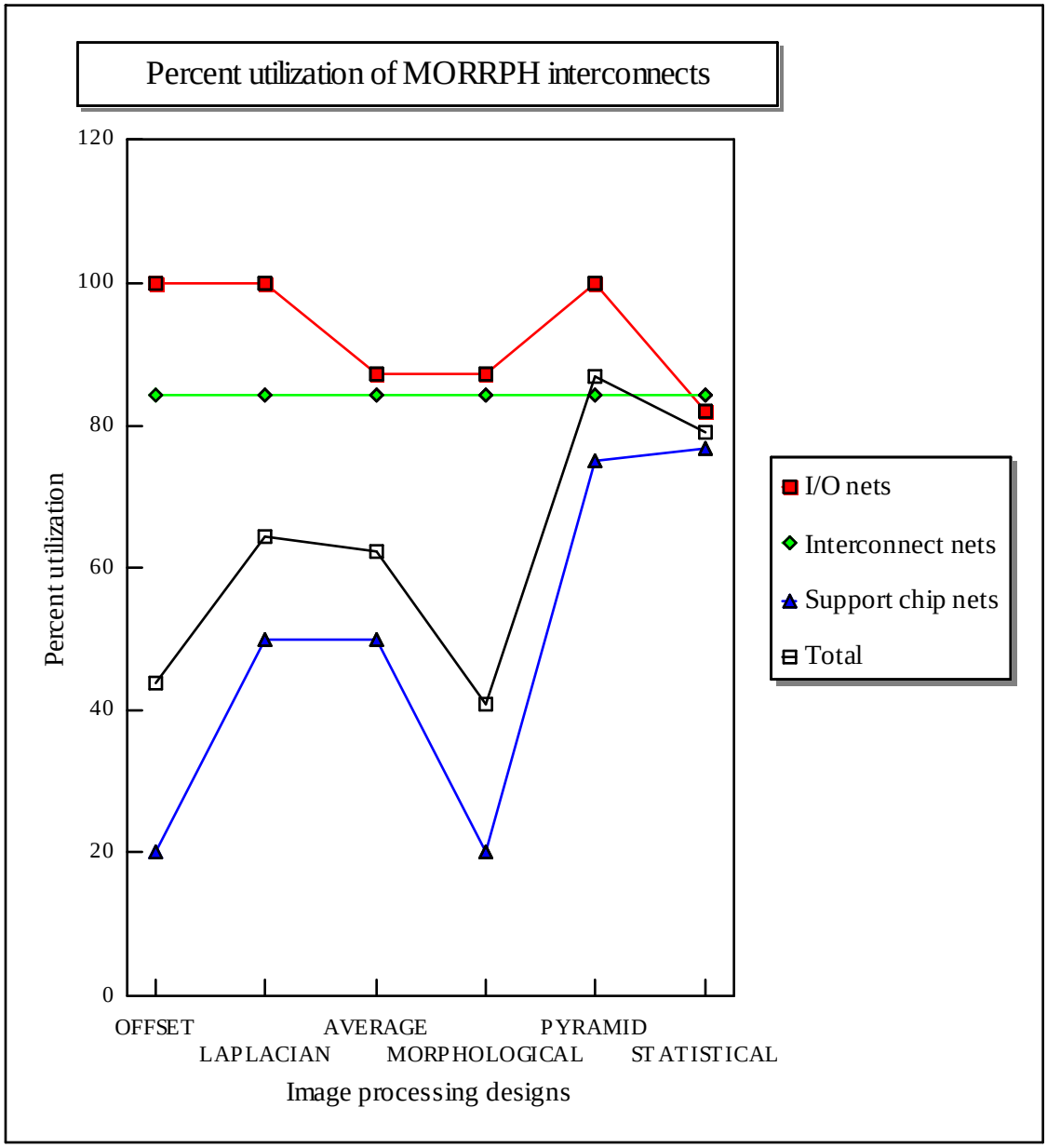


Figure 5.2-7. Percent utilization of MORRPH interconnection resources.

Finally, the utilization of support chips could be compared with utilization of other hardware architectures. However, this utilization should be 100% for the MORRPH PE array, when the appropriate type of support chips are available. This is because only the number and type of support chips required for each particular application are assembled on the MORRPH board. The low cost of SRAM support chips significantly reduces the importance of a comparison with the utilization of SRAM in other hardware architectures. Additionally, the undetermined nature of support chips allows solutions to be achieved using other types of chips or multi-chip modules that cannot be included in other completely defined architectures, the most significant benefit of support sockets.

5.2.3. Performance of the Basis Image Processing Designs

Performance of an image processing design is determined by the maximum rate at which pixel values may be processed by the design. The maximum pixel processing rate of an individual module is calculated by dividing its maximum operating clock frequency by the number of clock cycles required to process each pixel value. For an image processing design, the maximum pixel processing rate is calculated by dividing the maximum operating frequency of the slowest module by the maximum number of cycles required to process a pixel value by any module in the design. Therefore, the performance of image processing designs can be improved by increasing the operating frequency of the slowest modules in the design or by reducing the maximum number of clock cycles required to process a pixel value.

The basis image processing modules of Appendix II are designed to operate with four cycles/pixel or less. Most modules that require additional clock cycles to process single pixel values have asynchronous interfaces with support chips (**SHADE_A**, **BOUNDARY_A**, **HISTOGRAM_A** ...) or inter-board I/O interconnections (**ZEE2SUIT** and **SUIT2ZEE**). These asynchronous interfaces are difficult to design for consistent operation, due to the variable propagation and routing delays introduced by a the random placement and routing methods of the translation software. Additional design effort may reduce the number of clock cycles required by these asynchronous circuits, but require more design effort and sophisticated translation techniques.

The maximum operating clock frequency of an image processing module is affected by several factors, including the design of individual modules, speed and size of FPGA chips, and compilation techniques. Each of these factors can be used to improve the performance of existing modules.

There are many trade-offs in the design of an individual image processing module. For example, a hardware multiply circuit may be designed to use a minimal amount of FPGA resources, require fewer clock cycles, or operate at a higher clock frequency. One method to increase the maximum operating clock frequency is to pipeline the design of individual modules. This reduces propagation delay through combinational circuitry but increases the latency of modules and requires additional FFs for intermediate operand storage. However, this is a viable option because it was shown previously in this section that most current designs are limited by CLB usage, not available FFs.

The second method of increasing the maximum operating clock frequency of an image processing design is to use faster or larger FPGA chips. For example, the Xilinx 4010PG191 FPGA chip is currently available in five speed grades (-3, -4, -5, -6, and -10, where -3 is the fastest). The maximum clock frequencies provided in Table 5.1-2 were obtained by compiling each evaluation module in Appendix II into a 4010PG191-5 FPGA chip. The center value -5 speed grade is used so that performance increase or decrease of faster or slower parts may be more reliably estimated. The **HISTOGRAM_A**, **EROSION_A**, and **AVERAGE_3x3_A** modules have been compiled into each of the available speed grades and the average change in maximum clock frequency is shown in Figure 5.2-8. This plot is useful for estimating the appropriate speed grade FPGA using compilation results from the -5 speed grade parts.

The clock frequency may be reduced when FPGA chips are heavily utilized. Routing of signals within heavily utilized FPGA chips becomes more complicated, resulting in longer propagation delays for internal combinational signals. This performance degradation is easily observed by comparing the compiled frequency of FPGAs in the image processing design with the minimum of the estimated frequency of all individual modules contained within the FPGA (shown for evaluation modules in Table 5.1-2). Larger FPGA chips are used only when the performance is observed to be degraded in heavily utilized FPGA chips.

Faster and larger FPGA chips are more expensive. The modular nature of the MORRPH board allows the PE array to be configured with the appropriate size and speed grades required to implement the desired task at the required processing rate. Using only the required size and speed FPGA chips provides a cost-effective solution.

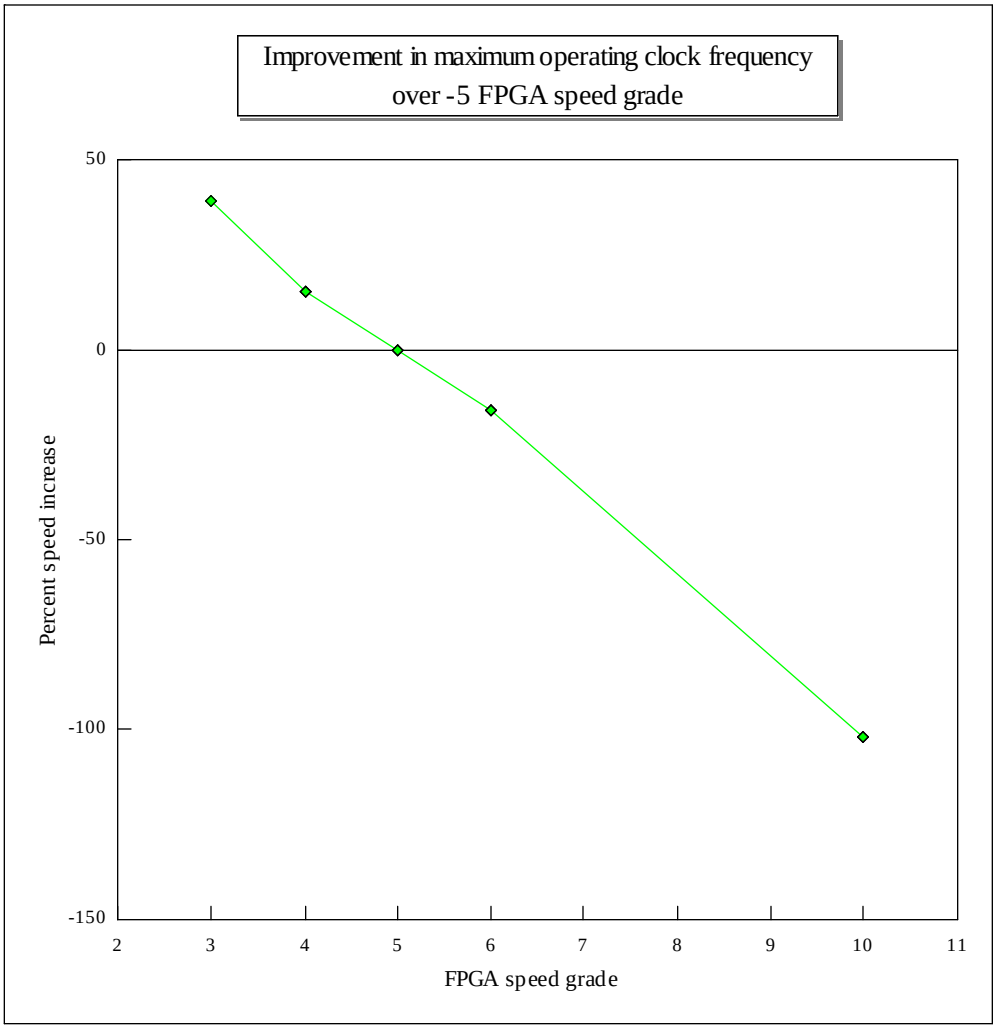


Figure 5.2-8. Effect of FPGA speed grade on maximum operating clock frequency.

Finally, performance of individual modules may be improved by using more sophisticated translation techniques. All modules have been compiled with default place and route effort of the Xilinx XACT software tools. These automated translation tools from Xilinx can use more effort in placement and routing to get faster designs, at the expense of longer compilation times. Timing constraints can be used to control placement and routing effort to obtain a targeted

performance level. Routing priorities may be attached to individual nets in order to have nets with long propagation delays routed first. None of these methods are currently used by the development system software, and are suggested as future performance enhancements.

Individual evaluation modules of Appendix II were designed with the minimum of effort for correct operation. The slowest compiled clock frequency is 9.1 MHz and at most four clock cycles are required (see Table 5.1-2). This provides a minimum processing rate of 2.275 MPixels/sec when a design contains both these limitations, possibly in two different modules.

This level of performance is adequate for many image processing applications. However, many applications require additional speed, such as the 7.5 Mbytes/sec required for typical 512x512 gray scale image data generated at 30 frames/sec. Additional performance is achieved using any of the techniques described above. A combination of using the fastest available FPGA parts and redesigned modules that require only two clock cycles per pixel immediately provides a pixel processing rate over 7.5 Mbytes/sec for all the individual evaluation modules.

The maximum operating frequency estimates in Table 5.1-2 are very conservative. These estimates use the maximum propagation delays through each CLB and routing resource of the compiled FPGA chip. It is expected that compiled circuits will operate reliably at considerably higher clock frequencies than the conservative estimates provided by the Xilinx software tools.

All basis image processing designs were compiled on a 200 MHz DEC Alpha workstation with 190 Mbytes of RAM [78]. Compile times ranged from 12 minutes for the offset design up to almost three hours for the statistical image processing design. The long compile times are a

characteristic problem associated with development of multiple FPGA computing solutions and motivates the creation of the TRAVERSE software simulation tools. Any future modification to the translation procedure intended to increase performance that results in significantly longer compilation times should be a user selectable option.

This section has validated the design methodology by quantitatively analyzing both the MORRPH hardware architecture and TRAVERSE software development system. Design entry methods of the TRAVERSE were shown not to require excessive overhead circuitry when the number of utilized SUIT bus channels is small. Similarly, the modular MORRPH architecture provides more efficient use of FPGA hardware than all other evaluated hardware architectures. The high utilization of different types of nets in the architecture justify the allocation of available IOB between interconnection, I/O, and support socket nets. Finally, the performance of image processing designs is provided with suggestions for increasing the processing speed if required.

Chapter 6. Conclusions

The goal of this dissertation has been to provide a new design methodology for creating real-time image processing hardware that overcomes the limitations of current solutions. This goal is achieved by creating a new class of incompletely specified FPGA-based architectures and an associated software development system.

This new incompletely specified MORRPH architecture provides the modularity lacking existing FPGA-based CCMs. Since the size and speed grade of FPGA chips are determined by each application, the MORRPH architecture provides a much more efficient and flexible architecture for implementing a wide range of real-time low-level image processing designs.

Open support sockets provide even greater utility to the MORRPH architecture. Embedding support chips into the processing array overcomes restrictions imposed by completely specified hardware architectures. These support chip sockets allow solutions to be achieved with the MORRPH architecture that are impossible with the fixed resources of other architectures.

The new incompletely specified MORRPH architecture represents a significant contribution of this work, even influencing the way commercial FPGA hardware architectures

are designed. Several commercial FPGA-based architectures are already providing modular and configurable features to similar to the MORRPH architecture.

The incompletely specified MORRPH architecture changes the way real-time image processing designs are created. Instead of creating a circuit to accommodate the limitations of a completely specified architecture, the hardware architecture is configured with the resources required by the intended application. This method of creating real-time image processing solutions has been shown to be efficient in its use of available hardware resources.

A specific example of the incompletely specified MORRPH hardware architecture is created for this dissertation, an FPGA-based real-time processing board for the ISA bus called MORRPH-ISA. This physical implementation of the MORRPH architecture provides a vehicle for absolute verification of both the MORRPH hardware architecture and image processing designs created with the TRAVERSE development system software.

The TRAVERSE development system software was created to overcome the difficulty associated with creating, verifying, and translating the gate-level circuits required for FPGA-based CCMs. Gate-level FPGA resources require a significant amount of design expertise to create custom solutions for individual problems. To make the design methodology practical, the processes of design entry, verification, and translation are simplified and automated by the TRAVERSE development system software.

Standards in design entry process allow independent development of low-level image processing modules that can be combined into complicated designs that represent data flow graphs. These standards in design entry also allow the creation of automated tools for

verification and translation. Additional benefits of portability, accessibility, and maintainability are also provided by these design standards.

The logic of an image processing design must be compiled into the resources of a multiple FPGA-based destination hardware architecture. Custom programs written by the author perform the essential processes of network partitioning and global routing. Another significant contribution of this dissertation is the new state search algorithm and heuristic cost function created for the network partitioning algorithm.

Both software simulation and in-circuit emulation are used for design verification by the TRAVERSE development system software. Standards in the design entry methods enabled the creation of automated verification tools. These tools reduce the amount of effort required to generate test vectors and eliminates the need for custom verification programs.

These tools combine to make TRAVERSE the most highly automated development system available for the creation, translation, and verification of image processing designs into an FPGA-based CCM, another of the significant contributions of this dissertation. The design and verification time for simple image processing tasks has been reduced by TRAVERSE from a few days down to a few hours.

Six image processing designs were created as a basis for evaluation of the hardware architecture and software development system. Basis image processing designs are representative of the types of low-level image processing tasks the design methodology is intended to solve.

Basis image processing designs are used to analyze the amount of overhead circuitry required for use of the design entry standards of the development system software. Results indicate that the upper bound of overhead circuitry approaches 50% of the total logic in some image processing designs. Excessive overhead logic exists in image processing designs that use a large number of image channels. Therefore, it is important to only preserve image data on channels that are required for future processing.

The basis designs are also used to compare the MORRPH-ISA architecture to four other commercial FPGA-based CCMs. MORRPH-ISA is shown to provide significantly greater utilization of its FPGA resources, approximately twice the utilization of the closest architecture. The utilization of system nets and FPGA resources are balanced, proving the suitability of the mesh interconnections of the PE array and the allocation of FPGA IOBs between support socket and interconnection system nets.

Evaluation modules were created without attempting to optimize their performance. Initial design effort produced modules capable of processing from 3.4 - 48.5 Mbtes/sec. Additional performance can be obtained by compiling into faster FPGA chips, using more sophisticated compilation techniques, or redesign of slow components. These modules are suitable for many real-time image processing tasks, but more high-performance modules are required to create a comprehensive library of modules for low-level image processing.

The new design methodology created for this dissertation represents the first automated method of creating image processing designs for an FPGA-based CCMs. Basis image processing designs have proven the feasibility of the design methodology and shown that the obtained

solutions have a high utilization of hardware resources. Efficient utilization of resources translates into cost-effective solutions. This success in solving a number of industrially interesting image processing problems has already begun to impact the way FPGA-based image processing hardware is created in industry, and the ease of use provided by the TRAVERSE software will make FPGA-based processing applicable to a larger set of image processing problems.

Appendix I. Format Definition for Architecture Configuration Files

This appendix defines a configuration file format for programming and communication with multiple FPGA-based CCMs. The goal of this file format is to provide a single medium for the exchange of information between the distinct processes involved in the creation and verification of a working design. This file format was developed by the author specifically for the development system software of this dissertation, but is general enough to be used with other hardware architectures and software development systems.

Version 2.1 of the ACF file format is supported by the translation and in-circuit emulation programs (MTRANS and MDBUG) created for this dissertation. It is recommended that custom real-time applications programs use this format as well.

The Architecture Configuration File or ACF file format contains all the required information for configuration, communication, and programming of a multiple FPGA-based CCM. This ASCII text file format contains up to three sections. The first section, called the configuration section, contains information required to identify the configuration of the hardware architecture. The functionality and location of implemented I/O port locations are defined in the next section, called the port section. The final section, called the bitstream section, contains the actual bitstreams required to program FPGAs in the destination hardware architecture.

Each line in the ACF format, called a *record*, is terminated by an ASCII carriage return and line feed (CR and LF). A maximum of 200 ASCII characters (including the CR and LF) are allowed in each record. Each record is comprised of several *fields*, where fields are separated by a comma. Spaces and tabs between fields are ignored. For example, any field that begins with a semicolon is defined to be a comment field. *Empty records* only contain only the CR and LF characters and *comment records* begin with a comment field. Empty records and comment records may be inserted anywhere within an ACF file

Special records called *section framing records* are used to open and close the three sections defined above. The format of fields used in section framing records is shown in the following:

Record:

<section type>_SECTION

section type = string defining name of section

function: open defined section.

example: BITSTREAM_SECTION *or* PORT_SECTION

Record:

END_<section type>_SECTION

section type - string defining name of the section

function: close defined section

example: END_CONFIGURATION_SECTION *or* END_PORT_SECTION

Between these framing records, information for the section is contained in *data records*. The end of an ACF file is defined with the special "EOF" framing record. The sectional format of ACF files is presented below in Figure A1.1-1.

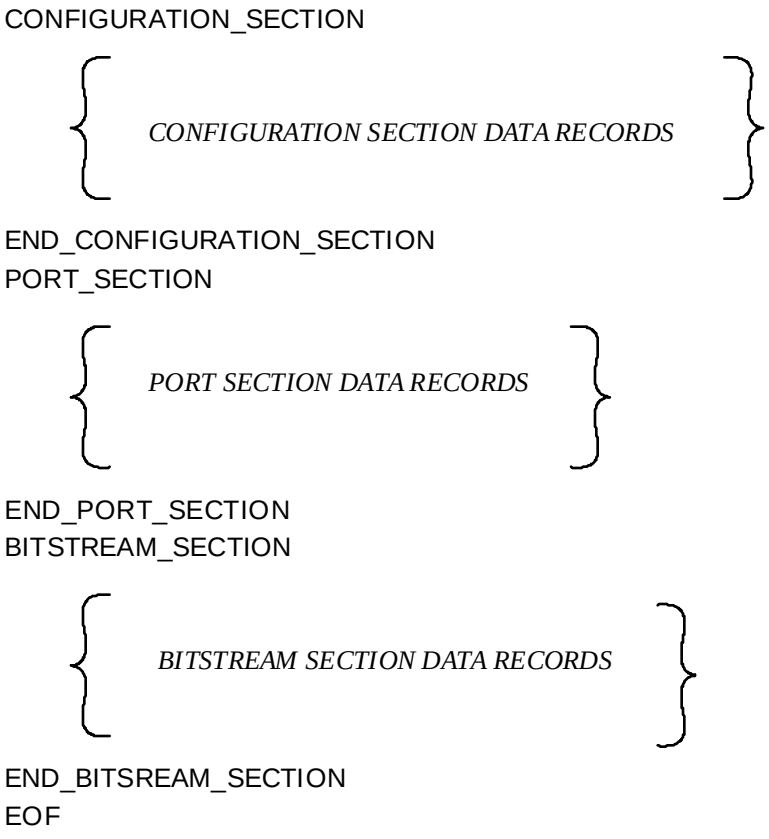


Figure A1.1-1. Section definitions of the ACF file format.

Several types of data records may be contained in the configuration section. These data records define the location and type of FPGA chips and base I/O address of the destination hardware architecture. The format of these data records follows:

Record:

FPGA_TYPE, COL=<value1>, ROW=<value2>, TYPE=<value3>

value1 = integer column number

value2 = integer row number

value3 = string FPGA type

function: define type of FPGA and location

example: FPGA_TYPE, COL=0, ROW=1, TYPE=4010PG191-5

Record:

IO_ADDRESS, ADDR=<value>

value = integer base I/O address

function: define base I/O address of destination hardware architecture

example: IO_ADDRESS, ADDR=300

Records with additional information may be defined for new versions of the ACF file format.

The port section defines all port locations used for communication with the destination hardware architecture. Both the offset address, configuration, and function of each port is defined. Three type of 8-bit ports are defined: read (RPORTB), write (WPORTB), and read/write ports (RWPORTB). Similarly, 16-bit ports are called RPORTW, WPORTW, and

RWPORTW. Each port location is defined with three types of records, two types of *port framing records* and one type of *port data record*. These records appear sequentially in an ACF in the order shown in the following:

Record:

<port type>, ADDR=<value1>, COL=<value2>, ROW=<value3>, INIT=<value4>

port type = string of port types defined above (RPORTB, WPORTB ...)

value1 = integer I/O offset address

value2 = integer number of FPGA column location

value3 = integer number of FPGA row location

value4 = integer number of initial value (only for write ports)

function: open defined port location

example: WPORTB, ADDR=0, ROW=2, COL=1, INIT=45

Record:

PIN, <value1>, <value2>

value1 = integer value of bit position (0-7 for byte ports)

value2 = string value of corresponding net name in design

function = define bit positions that are used and their design net connections

example: PIN, 0, AA/INVALUE0

Record:

END_<port type>

port type = string of port types defined above (RPORTB, WPORTB ...)

function: close defined port location

example: END_WPORTB

The size and location (and possibly initial value for write ports) of each port is defined by the first port framing record defined above. Connections of each bit position are then individually specified using the port data records, unused bit positions are omitted. Finally, the port location is closed with the second port framing record.

The bitstream section contains the data values used to program FPGAs in the destination hardware architecture. Whenever FPGAs are daisy-chained for programming, they define a column in the destination hardware architecture. A single bitstream is included to program all FPGAs in a column. As with ports and sections, each bitstream is opened and closed with one of two types of *bitstream framing records*. A single type of data record contains values for programming a column of FPGAs. This data record contains configuration data in the MCS file format defined by the Intel Corporation. Each of these records are used in the order they are presented in the following:

Record:

MCS_FILE, COL=<value1>

value1 = integer value of column of FPGAs to program with bitstream

function = open definition of a bitstream and indicate column to be programmed

example: MCS_FILE, COL=1

Record:

:<value>

value = numerical data in the Intel MCS file format

function = all MCS file records begin with a colon

Record:

END_MCS_FILE

function = close definition of bitstream

example: END_MCS_FILE

A leading colon indicates data in the MCS file format. If this data is extracted to an MCS file, the leading colon is part of the MCS file format and should be included in the output file.

This completely defines Version 2.1 of the ACF format. An example ACF file (with abbreviated MCS bitstream) is shown in Figure A1.1-2. The three sections of the ACF file contain all the information required for communication, programming, and configuration of a multiple FPGA-based CCM.

```
CONFIGURATION_SECTION
IO_ADDRESS, ADDR=300
FPGA_TYPE, COL=2, ROW=0, TYPE=4010PG191-5
FPGA_TYPE, COL=2, ROW=1, TYPE=4010PG191-5
END_CONFIGURATION_SECTION
PORT_SECTION
; this is the only port location, for the offset value
WPORTB, ADDR=0, COL=2, ROW=0, INIT=23
PIN, 7, AA/OFFSET7
PIN, 6, AA/OFFSET6
PIN, 5, AA/OFFSET5
PIN, 4, AA/OFFSET4
PIN, 3, AA/OFFSET3
PIN, 2, AA/OFFSET2
PIN, 1, AA/OFFSET1
PIN, 0, AA/OFFSET0
END_WPORTB
END_PORT_SECTION
BITSTREAM_SECTION
MCS_FILE, COL=2
:020000020000FC
    .
    .
    .
    .
:023FF
END_MCS_FILE
END_BITSTREAM_SECTION
EOF
```

Figure A1.1-2. Example file in ACF format.

Appendix II. Modules of the Evaluation Libraries

This appendix defines the functionality of modules contained in the evaluation libraries. A description the inputs, outputs, support chips, latency, and a functional description of each of the evaluation modules are presented in the following. Evaluation modules are divided into the five individual libraries (i.e., SUIT2, SUIT2A, SUIT2B, SUIT2C, and SUIT2D) used to organize the modules. There are no 2-byte processing modules, so the SUIT2B library is empty. This appendix is intended to provide a reference for the functionality of individual modules and enough information for the interested reader to reproduce functionally equivalent modules, if desired.

LIBRARY 1: SUIT2

Module Name: **MULTIPLEX2**
Library: SUIT2
Input busses: ***AA[15:0], BB[15:0]***
Output busses: ***QQ[15:0]***
Input operands: none
Output operands: none
Support chips: none
Latency: three cycles (minimum)

Functional Description: The two input SUIT busses ***AA*** and ***BB*** are multiplexed onto the single output SUIT bus ***QQ***. When data is available on either of the input busses, it is input to a 2-word deep queue. The queue for the ***AA*** bus has service priority over the ***BB*** bus queue. When data is not available in either of the two queues, a marking bus cycle is transmitted on SUIT bus output ***QQ***.

Module Name: **MULTIPLEX4**
Library: SUIT2
Input busses: ***AA[15:0], BB[15:0]***
Output busses: ***QQ[15:0]***
Input operands: none
Output operands: none
Support chips: none
Latency: five cycles (minimum)

Functional Description: The two input SUIT busses ***AA*** and ***BB*** are multiplexed onto the single output SUIT bus ***QQ***. When data is available on either of the input busses, it is input to a 4-word deep queue. The queue for the ***AA*** bus has priority over the ***BB*** bus queue. When data is not available in either of the two queues, a marking bus cycle is transmitted on SUIT bus output ***QQ***.

Module Name: **SUIT2ZEE**
Library: SUIT2
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: none
Output operands: none
Support chips: none
Latency: not defined

Functional Description: The input bus ***AA*** is converted from the SUIT bus standard to the output bus ***QQ*** in the ZEE bus standard. The required clock signal for the ZEE bus is generated from the system clock, and is 1/4 of the frequency of the system clock. The high order bit of the channel select lines is discarded, as this signal is used for the output clock. Non-marking SUIT bus cycles are loaded into a 4-word deep queue. Since only one SUIT value is transmitted every four clock cycles, the continuous input should not exceed one non-marking SUIT cycle every four clock cycles. A maximum of four consecutive non-marking cycles are allowed, when the queue is empty.

Module Name: **ZEE2SUIT**
Library: SUIT2
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: none
Output operands: none
Support chips: none
Latency: not defined

Functional Description: The input bus ***AA*** is converted from the ZEE bus standard to the output bus ***QQ*** in the SUIT bus standard. Rising edges are detected on the input clock signal from the ZEE bus and a single non-marking cycle is generated on the output SUIT bus. All other bus cycles are marking cycles. The system clock must be at least four times the frequency of the clock signal on the input ZEE bus. Only output SUIT bus channels 0-7 are used, because the ZEE bus only accommodates eight channels.

Module Name: **BLOCK_CHAN**
Library: SUIT2
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: The transmission channel of all non-marking cycles of the input SUIT bus ***AA[11:8]*** is compared with the input operand ***CHAN***. If these two values match, the SUIT bus output ***QQ*** is a marking cycle. Otherwise, the output SUIT bus ***QQ*** is equal to the input bus ***AA***.

Module Name: **INC_CHAN**
Library: SUIT2
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: none
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: The transmission channel of the input SUIT bus ***AA[11:8]*** is incremented and then the entire 16-bit SUIT bus value is transmitted on the output SUIT bus ***QQ***. Channel 15 is incremented to channel 0.

Module Name: **REGINT2BW**
Library: SUIT2
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***

Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module creates a bitmapped image from the ROI bit of the input SUIT bus. If the ROI bit is "1" the output SUIT data bus is set to "FF". Otherwise, the output SUIT data bus is set to "00". The bitmapped image effectively replaces data on the selected SUIT transmission channel, all other channels are blocked.

Module Name: **SUIT2ISA**
Library: SUIT2
Input busses: ***AA[15:0]***
Output busses:
Input operands: ***GO, NEXT***
Output operands: ***SUIT_BUS_OUTPUT[15:8], SUIT_BUS_OUTPUT[7:0], COUNT[12:0]***
Support chips: two M6264 8Kx8 SRAM chips
Latency: not defined

Functional Description: This module stores data from all non-marking cycles of the input SUIT bus ***AA*** into an 8Kx16 FIFO buffer, when the ***GO*** signal is set. When the ***GO*** signal is cleared, data can be retrieved from the buffer. The ***COUNT*** operand value is read to determine the number of non-marking SUIT values written to the buffer while ***GO*** was set. To retrieve data from the buffer, high and low bytes are sequentially read from the ***SUIT_BUS_OUTPUT*** operands. The ***NEXT*** signal is toggled to retrieve the next stored SUIT value from the buffer. Setting the ***GO*** signal clears the count value.

Module Name: **ISA2SUIT**
Library: SUIT2
Input busses: none
Output busses: ***QQ[15:0]***
Input operands: ***SUIT_BUS_INPUT[15:8], SUIT_BUS_INPUT[7:0], TRANS_SUIT_VALUE***
Output operands: none
Support chips: none
Latency: not defined

Functional Description: This module normally outputs marking cycles on the output SUIT bus ***QQ***. To transmit a 16-bit SUIT bus value on ***QQ***, first the high (command) and low (data) bytes must be written to the **SUIT_BUS_INPUT** high and low byte operands. The ***TRANS_SUIT_VALUE*** signal is toggled to transmit the value. The rising edge of ***TRANS_SUIT_VALUE*** initiates the transmission.

LIBRARY 2: SUIT2A

Module Name: **HISTOGRAM_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***, ***OCHAN[3:0]***, ***COLL/TRANS***
Output operands: none
Support chips: three M6116 1Kx8 SRAM chips
Latency: not defined

Functional Description: This module creates an array of 24-bit values that count the number of times each of the 256 possible pixel intensity values occur in the input image, called a histogram. The histogram is generated from the selected input SUIT transmission channel ***CHAN*** and output on the selected SUIT transmission channel ***OCHAN***. Count values are stored in the support RAM chips. The histogram is transmitted on the falling edge of the ***COLL/TRANS*** signal. Transmission of the histogram also resets the count values for the next histogram. All input SUIT channels are blocked from the output SUIT bus ***QQ***.

Module Name: **SHADE_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: two M6116 1Kx8 SRAM chips
Latency: four cycles

Functional Description: This module performs the shade compensation operation defined in Section 1.2.1, using the 1st order quadratic (e.g., linear) equation $y_i = m_i x_i + b_i$. The 10-bit m_i and 6-bit b_i operands for each pixel location are stored in the support SRAM chips. Only image data on the SUIT transmission channel selected by ***CHAN*** is shade compensated. All other channels are passed unmodified to the output SUIT bus ***QQ*** with the same latency.

Module Name: **FIELD_VIEW_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0], OFFSET[8:1], NUMPIX[8:1]***
Output operands: none
Support chips: none

Latency: one cycle

Functional Description: This module limits the field-of-view for the selected SUIT transmission channel defined by the operand ***CHAN***. All other SUIT bus transmission channels are passed to the output SUIT bus ***QQ*** unmodified. The field-of-view of the selected image is limited with two values, ***OFFSET*** and ***NUMPIX.*** Both of these 8-bit input operands are used to create a 9-bit value in which the least significant bit is "0", creating even values from 0-510. All pixels before ***OFFSET*** are ignored and only ***NUMPIX*** values are transmitted after ***OFFSET*** input values are received. Both count values are reset after each new line start SUIT bus command.

Module Name: **OFFSET_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0], OFFSET[8:1]***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module adds a programmable offset to each pixel of the SUIT transmission channel defined by the ***CHAN*** input operand. See the schematic of Example 4.2.3. Image values from the input SUIT bus ***AA*** are modified using the equation $y_i = x_i +$ ***OFFSET***, and transmitted on the output SUIT bus ***QQ*** on the same channel. All other channels are passed unmodified to the output SUIT bus ***QQ*** with the same latency.

Module Name: **EROSION_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: none
Latency: two cycles

Functional Description: This module performs the erosion operation defined in Section 1.2.1. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are passed to the output SUIT bus ***QQ*** unmodified. A logical AND operation is performed on the ROI bit of a 3x3 window to perform the erosion operation. The output ROI bit is modified accordingly.

Module Name: **DILATION_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: none
Latency: two cycles

Functional Description: This module performs the dilation operation defined in Section 1.2.1. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are passed to the output SUIT bus ***QQ*** unmodified. A logical OR operation is performed on the ROI bit of a 3x3 window to perform the erosion operation. The output ROI bit is modified accordingly.

Module Name: **THRESHOLD_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0], THRESHOLD[7:0], UNDER/OVER***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module performs the threshold operation defined in Section 1.2.1. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are passed to the output SUIT bus ***QQ*** unmodified. When ***UNDER/OVER*** signal is "1", the ROI bit of the output SUIT bus ***QQ*** is set when the input pixel value is under the operand value ***THRESHOLD*** and the ROI bit is cleared otherwise. When ***UNDER/OVER*** signal is "0", the ROI bit of the output SUIT bus ***QQ*** is set when the input pixel value is over the operand value ***THRESHOLD*** and the ROI bit is cleared otherwise.

Module Name: **BOUNDARY_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0], WIN_SIZE_A[3:0], WIN_SIZE_B[3:0]***
Output operands: none
Support chips: one M6116 1Kx8 SRAM chip
Latency: one line

Functional Description: This module finds the horizontal and vertical boundaries of an object in the image of the input SUIT bus ***AA***. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are passed to the output SUIT bus ***QQ*** unmodified. Input SUIT image data is assumed to be generated by a line scan camera, and frame start and frame end SUIT bus commands are therefore not present. The horizontal boundaries are marked with SUIT frame start and frame end commands. A frame start command is generated when the first line with ***WIN_SIZE_A*** consecutive input pixels that have the ROI bit set. A frame end command is transmitted on the output SUIT bus ***QQ*** when ***WIN_SIZE_B*** consecutive lines do not have any region with at least ***WIN_SIZE_A*** consecutive pixels with the ROI bit set. Vertical boundaries are identified by modifying the ROI bit of the output SUIT bus. The output ROI bit is set beginning with the first occurrence of at least ***WIN_SIZE_A*** consecutive input pixels that have the ROI bit set. The output ROI bit remains set until the last pixel in a line that is part of at least ***WIN_SIZE_A*** consecutive input pixels that have the ROI bit set.

Module Name: **LAPLACE_3X3_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: two M6116 1Kx8 SRAM chips
Latency: three cycles

Functional Description: This module performs the Laplace 3x3 window operator defined in Figure 1.2-3. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are blocked from the output SUIT bus ***QQ***.

Module Name: **AVERAGE_3X3_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: two M6116 1Kx8 SRAM chips
Latency: eight cycles

Functional Description: This module performs the average 3x3 window operator defined in Figure 1.2-3. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are blocked from the output SUIT bus ***QQ***.

Module Name: **GAUSSIAN_3X3_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: two M6116 1Kx8 SRAM chips
Latency: five cycles

Functional Description: This module performs the Gaussian 3x3 window operator defined in Figure 1.2-3. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are blocked from the output SUIT bus ***QQ***.

Module Name: **HALF_ROW_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module removes all even rows from an image. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed and all other channels are blocked from the output SUIT bus ***QQ*** when the ***BLOCK/PASS*** signal is set. When the ***BLOCK/PASS*** signal is cleared, all information on other channels is transmitted on the output SUIT bus ***QQ***.

Module Name: **HALF_COL_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0], BLOCK/PASS***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module removes all even columns from an image. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed and all other channels are blocked from the output SUIT bus ***QQ*** when the ***BLOCK/PASS*** signal is set.

When the ***BLOCK/PASS*** signal is cleared, all information on other channels is transmitted on the output SUIT bus ***QQ***.

Module Name: **LUT_A**
Library: SUIT2A
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: one M6116 1Kx8 SRAM chip
Latency: one cycle

Functional Description: This module uses an support chip RAM to implement an arbitrary LUT. See the schematic of Example 4.2.4. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed and all other channels are passed to the output SUIT bus ***QQ*** unmodified.

LIBRARY 3: SUIT2B

no modules were created for this library.

LIBRARY 4: SUIT2C

Module Name: **FIELDVIEW_C**
Library: SUIT2C
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0], OFFSET[8:1], NUMPIX[8:1]***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module limits the field-of-view for the selected SUIT transmission channel defined by the operand ***CHAN***. All other SUIT bus transmission channels are passed to the output SUIT bus ***QQ*** unmodified. The field-of-view of the selected image is limited with two values, ***OFFSET*** and ***NUMPIX.*** Both of these 8-bit input operands are used to create a 9-bit

value in which the least significant bit is "0", creating even values from 0-510. All pixels before ***OFFSET*** are ignored and only ***NUMPIX*** values are transmitted after ***OFFSET*** input

Module Name: **SHADE_C**
Library: SUIT2C
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: two M6116 1Kx8 SRAM chips
Latency: four cycles

Functional Description: This module performs the shade compensation operation defined in Section 1.2.1, using the 1st order quadratic (e.g., linear) equation $y_{i,c} = m_{i,c} x_{i,c} + b_{i,c}$. Although this module is a 3-byte processing module, each of the three bytes are processed independently, as indicated by the "c" subscript of the variables in the linear equation above. The 10-bit $m_{i,c}$ and 6-bit $b_{i,c}$ operands for each pixel location are stored in the support SRAM chips. Only image data on the SUIT transmission channel selected by ***CHAN*** is shade compensated. All other channels are passed unmodified to the output SUIT bus ***QQ*** with the same latency.

Module Name: **HALF_COL2_C**
Library: SUIT2C
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module removes half of all columns from an image. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed and all other channels are blocked from the output SUIT bus ***QQ.*** For each odd column in the image, the red value of the pixel is compared to the red value of the previous even column. The red, green, and blue values are output on the output SUIT bus ***QQ*** for the pixel with the lowest red value, either the odd pixel or the even pixel.

Module Name: **THRESHOLD_C**
Library: SUIT2C
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***

Input operands: ***CHAN[3:0], TVALR[7:0]***, ***TVALRB[7:0]***
Output operands: none
Support chips: none
Latency: variable

Functional Description: This module performs a threshold operation to modify the output ROI bit. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are blocked from the output SUIT bus ***QQ***. The ROI bit of the output SUIT bus ***QQ*** is set when either the red pixel value is greater than the operand value ***TVALR*** or when the operand value ***TVALRB*** signal is greater than the red pixel value minus the blue pixel value. The ROI bit is cleared otherwise.

Module Name: **BOUNDARY_C**
Library: SUIT2C
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0], WIN_SIZE_A[3:0]***, ***WIN_SIZE_B[3:0]***
Output operands: none
Support chips: one M6116 1Kx8 SRAM chip
Latency: one line

Functional Description: This module finds the horizontal and vertical boundaries of an object in the image of the input SUIT bus ***AA***. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are passed to the output SUIT bus ***QQ*** unmodified. Input SUIT image data is assumed to be generated by a line scan camera, and frame start and frame end SUIT bus commands are therefore not present. The horizontal boundaries are marked with SUIT frame start and frame end commands. A frame start command is generated when the first line with ***WIN_SIZE_A*** consecutive input pixels that have the ROI bit set. A frame end command is transmitted on the output SUIT bus ***QQ*** when ***WIN_SIZE_B*** consecutive lines do not have any region with at least ***WIN_SIZE_A*** consecutive pixels with the ROI bit set. Vertical boundaries are identified by modifying the ROI bit of the output SUIT bus. The output ROI bit is set beginning with the first occurrence of at least ***WIN_SIZE_A*** consecutive input pixels that have the ROI bit set. The output ROI bit remains set until the last pixel in a line that is part of at least ***WIN_SIZE_A*** consecutive input pixels that have the ROI bit set.

Module Name: **AVERAGE_CHAN_C**
Library: SUIT2C
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***

Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module calculates the average of the three values for each pixel. The average value is added to the data structure, creating a 4-byte data on the output SUIT bus ***QQ***. The image on the SUIT channel equal to the value defined by the ***CHAN*** input operand is processed, all other channels are blocked from the output SUIT bus ***QQ***.

LIBRARY 5: SUIT2D

Module Name: **SEPARATE_CHAN_D**
Library: SUIT2D
Input busses: ***AA[15:0]***
Output busses: ***QQ[15:0]***
Input operands: ***CHAN[3:0]***
Output operands: none
Support chips: none
Latency: one cycle

Functional Description: This module takes the 4-byte input data structure defined by the ***CHAN*** input operand and outputs the data as four independent 1-byte data structures. The four 1-byte data structures are transmitted on channels 0-3 of the output SUIT bus ***QQ***. All other channels are blocked from the output SUIT bus ***QQ***.

References

[1] Bidlack, C., and Trivedi, M., "An Integrated Vision System for Object Identification and Localization Using Three-Dimensional Geometric Models," Proceedings of Applications of Artificial Intelligence IX, April 1991, pp. 270-280.

[2] Feather, T., Guan, L., et.al., "CAXSS - An Intelligent Threat Detection System," Proceedings of Applications of Signal and Image Processing in Explosives Detection Systems, SPIE Vol. 1824, 1992, pp. 153-161

[3] Conners, R. Ng, C., et. al., "Computer Vision Hardware System for Automating Rough Mills of Furniture Plants," Proceedings of SPIE, Applications of Artificial Intelligence VIII, April 1990, pp. 777-787.

[4] Haralick, R.M. and Shapiro, L.G., Computer and Robot Vision, Addison-Wesley, 1993, pp. 1-483.

[5] Jain, R., Kasturi, R., and Schunck, B., Machine Vision, McGraw-Hill, Inc., 1995, pp. 1-481.

[6] Phillips, D., Image Processing in C, R&D Publications, Inc., Lawrence KS, 1994, pp. 1-694.

[7] Intel DVI chipset

[8] Data Translation, Inc, "1993 Product Handbook," Vol. 2, No. 1, 1993, pp. 20-109.

[9] Aspex, Inc., "The PIPE: The Supercomputer for Real-Time Image Processing," Sales Literature, Aspex Inc., 536 Broadway, New York, NY 10012, 1991.

[10] Datacube, Inc., *High Performance Imaging*, Volume 8, No. 1 and 2, 1996.

[11] Epix Vision, "4MEG VIDEO Model 12 Pricing and Upgrade Offer," Vol. 2, No. 4, October 1993, pp. 4-6.

[12] Vision Modules, Inc. "3032 Digital Signal Processing Board," Sales Literature, Vision Modules, Inc., 1550 La Pradera Drive, Campbell, CA 95008, 1993.

[13] Eltec, Inc., "Product Summary," sales correspondence, 1994.

[14] Xilinx Corporation, *The Programmable Logic Data Book*, 1994.

[15] Arnold, J.M., Buell, D.A., Hoang, D.T., Pryor, D.V., Shirazi, N., and Thistle, M.R., "The Splash 2 Processor and Applications," IEEE Proceedings of the 1993 International Conference on Computer Design: VLSI in Computers & Processors, 1993, pp. 482-485.

[16] Djunatan, M., Mengko,T., "A Programmable Real-Time Systolic Processor Architecture for Image Morphological Operations, Binary Template Matching and Min/Max Filtering," 1991 IEEE International Symposium on Circuits and Systems (1 of 5) Apr. 1991, pp. 65-68.

[17] Ratha, N.K., Jain, A.K, Rover,D.T., "Convolution on Splash 2", Proceedings of FCCM 95, April 1995, pp. 204-213.

[18] Box, B., "Field Programmable Gate Array Based Reconfigurable Preprocessor," in Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa CA, April 1994, pp. 40-48.

[19] Quenot, G.M., Kraljic,J.S., Serot, J., Zavidovique, B. "A Reconfigurable Compute Engine for Real-time Vision Automata Prototyping," FCCM 94, pp. 91 - 100.

[20] Abbott, A.L., Athanas, P.M., Chen, L., and Elliott, R.L., "Finding Lines and Building Pyramids with Splash 2," IEEE Workshop on FPGAs for Custom Computing Macines, Apr. 1994, pp. 155-163.

[21] Chan S.C, Ngai H.O., Ho, K.L., "A Programmable Image Processing System using FPGAs," Intl. Journal of Electronics, 1993, Vol. 75, No. 4, pp. 725-730.

[22] Monaghan, S. "A Gate Level Reconfigurable Monte Carlo Processor," Journal of VLSI Signal Processing, Vol. 6, 1993, pp. 139-153.

[23] Nozal, L., Lorenzo, S., Boucho Olivereira, R.M., Shaban Mohamed, M., "A New Vision System: Programmable Logic Devices and Digital Signal Processor Architecture (PLD+DSP)," Proceedings of the 1991 Int. Conference on Industrial Electronics, Control, and Instrumentation, 1991, pp. 2014-2018.

[24] Rautiola, K., Pehkonen, K., Stahle, L., Jokitalo, P., "Design of a TMS320C40 Signal Processor and Programmable Logic Based Prototyping Environment of Real-time Machine

Vision Architectures," Proceeding of the 19th EUROMICRO Symposium on Microprocessing and Microprogramming, V. 38, n. 1-5, Sept. 1993, pp. 663-668.

[25] Van den Bout, D.E., "The Anyboard: Programming and Enhancements," Proceedings of FCCM 93, Napa CA., April 1993, pp.68-78.

[26] Casselman,S., "Virtual Computing and the Virtual Computer," in Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa CA, April 1993, pp. 43-48.

[27] Arnold,J.M., Buell, D.A., and Davis, E.G. "Splash 2," in Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, 1992, pp. 316-322.

[28] Giga Operations Corp., *G-800 Technical Summary*, Sales Literature, 1994.

[29] Chan, P.K., "A Field-Programmable Prototyping Board: XC4000 BORG User's Guide," UCSRC-CRL-94-18, April 1994.

[30] Shirazi, N. "Quantitative Analysis of Floating Point Arithmetic on FPGA-based Custom Computing Machines", Proceedings of FCCM '95, April 1995, to be published.

[31] PCI Special Interest Group, "PCI Local Bus Specification," Revision 2.0, April 30, 1993.

[32] Xilinx Corporation, *Components Price List*, Sales Literature, October 3, 1994.

[33] Epix, Inc, "SVOBJ & PXIPL for Silicon Video Mux," Sales Literature, June 1991.

[34] Matrox, Inc, "Matrox iTOOLS," Sales Literature, 1994.

[35] King, D. "Diary of a Field Applications Engineer", High Performance Imaging, Datacube, Inc. Winter 1995.

[36] Arnold, J.M., "The Splash 2 Software Environment," Proceedings of FCCM '93, Napa CA, April 1993, pp.88-93.

[37] Drayer,T.H. King, W.E., Tront, J.G., and Conners, R.W., "Using Multiple FPGA Architectures for Real-time Processing of Low-level Machine Vision Functions," Proceedings of IECON '95, to be published in Nov. '95.

[38] Drayer, T.H., Tront, J.G., King, W.E., and Conners, R.W., "A Modular and Reprogrammable Real-time Processing Hardware, MORRPH," Proceedings of FCCM '95, Napa, CA, April 1995, pp. 11-19.

[39] Drayer, T.H., Tront, J.G. King, W.E., and Conners, R.W., "MORRPH: A MOdular and Reprogrammable Real-time Processing Hardware," Proceedings of ISIE '95, July 1995, to be published.

[40] Newark, Inc " sales catalogue", 1995.

[41] Digikey, Inc "sales catalogue", 1995.

[42] Altera Corporation, *FLEX 8000 Handbook*, May 1994.

[43] National Semiconductor, *sales catalogue*, 1995.

[44] Data Translation, Inc., *DT-Connect II Bus Specification*, 1992.

[45] Eggebrecht, L.C., Interfacing to the IBM Personal Computer, Second Edition, Macmillan Computer Publishing, 1991, pp. 74-297.

[46] Drayer, T.H., et. al., "A High-performance Micro Channel Interface for Real-time Industrial Image Processing Applications," Proceedings of IECON '94, Bologna, Italy, September 1994, pp. 884-889.

[47] Viewlogic, Inc., *Powerview Release Notes*, 1992.

[48] Thomas, D., and Moorby, P., "The Verilog Hardware Descriptive Language," Kulwer Academic Publishers, Boston, 1991.

[49] Synopsis, Inc., "4000 Series Interface using FPGA compiler, " February 1993.

[50] Pulnix America Corporation, *TL-2600 RGB Line Scan Camera Operating Instructions*, 1987.

[51] Datacube, Inc., *MaxVideo MAXbus Specification*, 1988.

[52] National Aeronautics and Space Administration - John C. Stennis Space Center Science and Technology Laboratory, ELAS, Science and Technology Laboratory Applications Software Programmer Reference, Volume I, Report No. 183, May 1989, pp. 1-4, 20-22.

[53] Aldus Corporation, "TIFF, Revision 6.0," Aldus developers desk, (206-628-6593) June 3, 1992, pp.1-117.

[54] Barkakati, N., X Window System Programming, second edition, SAMS publishing, 1994, pp. 1-169.

[55] Intel Corporation, "MCS File format"

[56] Krishnamurthy, B., "An Improved Min-Cut Algorithm For Partitioning VLSI Networks," *IEEE Transactions on Computers*, Vol. C-33, No. 5, May 1984, pp. 438-446.

[57] Bui, T., Heigham, C., Jones, C., and Leighton, T., "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms," in Proc. of ACM/IEEE Design Automation Conference, 1989, pp. 775-778.

[58] Roy, K., and Sechen, C., "A Timing Driven N-Way Chip and Multi-Chip Partitioner," in IEEE/ACM International Conference on CAD-93, Santa Clara, CA, November 1993, pp. 240-247.

[59] Yeh, C.W., Cheng, C.K., Lin, T.Y., "A General Purpose Multiple Way Partitioning Algorithm," 28th ACM/IEEE Design Automation Conference, 1991, pp. 421-426.

[60] Yeh, C.W., Cheng, C.K., Lin, T.Y., "A General Purpose Multiple Way Partitioning Algorithm," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13, No. 12, December 1994, pp. 1480-1487.

[61] Saucier, G., Brasen, D., Hiol, J.P., "Partitioning with Cone Structures," IEEE/ACM International Conference on CAD-93, Santa Clara CA., November 1993, pp. 236-239.

[62] Murgai, R., Brayton, R., and Sangiovanni-Vincentelli, A., "On Clustering for Minimum Delay/Area," 1991 IEEE Conference on CAD, 1993, pp. 6-9.

[63] Cheng, C., Wei, Y., "An Improved Two-Way Partitioning Algorithm with Stable Performance," *IEEE Transactions on Computer-Aided Design*, Vol. 10, No. 12, December 1991.

[64] Charney, H., Plato, D., "Efficient Partitioning of Components," Proceedings of the Fifth Annual SHARE-ACM-IEEE 1968 Design Automation Conference, Washington D.C., July 1968, pp. 16.1-16.16.

[65] Hagen, L., Khang, A., "New Spectral Methods for Ratio Cut Partitioning and Clustering," *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 9, September 1992, pp. 1074-1085.

[66] Chan, P., Schlag, D., and Zien, J., "Spectral K-Way Ratio Cut Partitioning and Clustering," Proceedings of the 30th ACM/IEEE Design Automation Conference, July 1993, pp. 749-754.

[67] Ford, L., and Fulkerson, D., *Flows in Networks*, Princeton, NJ: Princeton University Press, 1962.

[68] Kernighan, B., and Lin, S., "An Efficient Heuristic Procedure for Partitioning of Graphs," Bell Systems Tech. Journal, Vol. 49, February 1970, pp. 291-307.

[69] Schweikert and Kernighan, "A Proper Model for the Partitioning of Electric Circuits," in Proceedings of 9th Design Automation Workshop, June 1970, pp. 57-62.

[70] Fiduccia, C., and Mattheyses, R., "A Linear-Time Heuristic for Improving Network Partitions," in Proceedings of 19th Design Automation Conference, 1982, pp. 175-181.

[71] Krishnamurthy, B., "An Improved Min-Cut Algorithm for Partitioning VLSI Networks," *IEEE Transactions on Computers*, Vol. C-33, No. 5, May 1984, pp. 438-446.

[72] Sanchis, L., "Multiple-Way Network Partitioning," in IEEE Transactions on Computers, 38, 1989, pp. 62-81.

[73] Rich, E., and Knight, K., Artificial Intelligence, Second ed., McGraw-Hill, 1992.

[74] Kirkpatrick, S., et.al., "Optimization by simulated annealing," *Science*, Vol. 220, May 1983, pp.671-680.

[75] Bui, T.N., "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms," in Proc. of ACM/IEEE Design Automation Conference, 1989, pp. 775-778.

[76] Xilinx, Inc, The XNF file format," 1994.

[77] Conners, R. Cho, T., et. al., "A Machine Vision System for Automatically Grading Hardwood Lumber," Industrial Methodology, 1992, pp. 317-341.

[78] Digital Electronic Corporation, *DEC 3000 product literature*, 1995.

Vita

Thomas Hudson Drayer was born on December 14, 1963 at the US naval hospital in Annapolis, Maryland. He graduated from Brunswick High School, located within Brunswick, Maryland, in June of 1982, where he was president of his senior class. He received a Bachelor of Science degree in Electrical Engineering from Virginia Polytechnic Institute and State University in December of 1987, followed by a Masters of Science in Electrical Engineering in December of 1991, also from Virginia Polytechnic Institute and State University. He received a Bradley Fellowship from the department of Electrical Engineering at Virginia Polytechnic Institute and State University to pursue his doctorate degree, and this work is a direct result of that fellowship.