Design and implementation of a real-time morphological image processor prototype

Jens Rodenberg

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
DESIGN AND IMPLEMENTATION OF A
REAL-TIME MORPHOLOGICAL IMAGE
PROCESSOR PROTOTYPE

by

Jens Rodenberg

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by: ________________________________
Department Chairman - Dr. Roy S. Czernikowski

______________________________
Reader - Prof. George A. Brown

______________________________
Reader - Dr. Tony H. Chang

DEPARTMENT OF COMPUTER ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

JULY, 1994
THESIS RELEASE PERMISSION FORM
ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title: Design and Implementation of a Real-Time Morphological Image Processor Prototype.

I, Jens Rodenberg, hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Signature: 
Date: 7/28/94
Abstract

Morphology, the study of form and structure, is also a method used for processing images. Morphological image processing can be used for many purposes, including edge detection, shape recognition, smoothing, and enhancement of images. A prototype for a real-time Morphological Image Processor has been developed to process $512 \times 512$ extended 8-bit gray scale images, using a $7 \times 7$ extended 8-bit gray scale mask. This prototype processor was developed jointly with another M.S. thesis candidate, Jeffrey Hanzlik. Software was also developed to allow the user to conveniently use a personal computer to transfer images to and from the Morphological Image Processor prototype and to perform a variety of image processing operations. Field programmable gate arrays have been used for the prototype, and implementing the same architecture in VLSI will allow for real-time processing of images.
# Table of Contents

Abstract ............................................................................................................. i

Table of Contents .......................................................................................... ii

List of Figures ................................................................................................. iii

List of Tables ................................................................................................... v

Glossary of Terms ......................................................................................... vi

1. Introduction .............................................................................................. 1-1

2. Theory of Morphology ............................................................................. 2-1

3. Morphological Array Processor Architecture ........................................ 3-1
   3.1 PATC Architecture ........................................................................... 3-5
   3.2 SAC Architecture ........................................................................... 3-20
   3.3 PATC and SAC Architecture Comparisons ........................................ 3-25

4. Morphological Image Processor Architecture ....................................... 4-1

5. Hardware Design ..................................................................................... 5-1
   5.1 Timing ............................................................................................ 5-2
   5.2 Main Controller ................................................................................ 5-11
   5.3 Memory Controllers ........................................................................ 5-29

6. Software Design ...................................................................................... 6-1

7. Conclusions .............................................................................................. 7-1

Appendix A. Software User’s Manual ......................................................... A-1

Appendix B. Software Listing ...................................................................... B-1

Appendix C. Schematics .............................................................................. C-1

Appendix D. Bibliography ........................................................................... D-1
List of Figures

1-1. Morphological Image Processor Block Diagram ........................................... 1-2
2-1. Example of Minkowski Addition and Subtraction ........................................ 2-2
3-1. Image and Window Matrices ................................................................. 3-4
3-2. Morphological Array Processor Architecture ........................................... 3-5
3-3. Blanking Example (part 1) ................................................................. 3-8
3-4. Blanking Example (part 2) ................................................................. 3-9
3-5. Blanking Example (part 3) ................................................................. 3-10
3-6. Blanking Example (part 4) ................................................................. 3-11
3-7. Blanking Example (part 5) ................................................................. 3-12
3-8. Blanking Example (part 6) ................................................................. 3-13
3-9. Blanking Control Bits ........................................................................... 3-14
3-10. Example Window and Image .............................................................. 3-22
3-11. Data Flow of SAC Architecture (part 1) ............................................... 3-23
3-12. Data Flow of SAC Architecture (part 2) ............................................... 3-23
4-1. Block Diagram of the Morphological Image Processor ............................ 4-1
4-2. Block Diagram of Memories ............................................................... 4-2
4-3. Block Diagram of Processors ............................................................. 4-2
4-4. Morphological Image Processor Schematic ............................................ 4-3
5-1. System Timing Diagram ........................................................................ 5-8
5-2. Controller Schematic ............................................................................ 5-16
5-3. Mem_Select Schematic .......................................................................... 5-20
5-4. Bus_Select Schematic .......................................................................... 5-23
5-5. Start_Blank Schematic ........................................................................ 5-26
5-5. Blank.Counter Schematic ......................................................... 5-27
5-7. Memory Read Cycle ............................................................. 5-32
5-8. Memory Write Cycle ............................................................ 5-33
5-9. Mem_Control Schematic ......................................................... 5-35
5-10. Mem.Counter Schematic ....................................................... 5-36
A-1. Example of User Interface Screen Layout ............................... A-6
A-2. Main Menu- Other Operations ............................................. A-7
A-3. Run Operations Menu ........................................................... A-10
A-4. Instruction Operations Menu .............................................. A-11
A-5. Window Operations Menu .................................................. A-12
A-6. Modify Window Menu .......................................................... A-13
A-7. Image Transfer Menu ............................................................ A-14
C-1. Delay14 Schematic .............................................................. C-3
C-2. DLC4 Schematic ................................................................. C-4
C-3. DLC8 Schematic ................................................................. C-5
C-4. DLEC8 Schematic ............................................................... C-6
C-5. Mx2_18bit Schematic ......................................................... C-7
List of Tables

3-1. Binary Pixel Values and Their Decimal Equivalents ............................................. 3-1
3-2. Blanking Control Values ......................................................................................... 3-15
3-3. Complete Blanking Control Values ........................................................................ 3-16
4-1 Volume Adder Registers 0-4 .................................................................................... 4-5
4-2 ALU Constant Registers 5,7 .................................................................................. 4-5
4-3 ALU Instruction Registers 6,8 ................................................................................ 4-6
4-4 Controller Start Register B .................................................................................... 4-7
4-5 Controller Instruction and Status Register C .......................................................... 4-7
4-6 Controller Memory Select Register D ..................................................................... 4-7
4-7 Bus Interface Register E ........................................................................................ 4-8
4-8 Bus Interface Register F ........................................................................................ 4-8
5-1 Main Controller Inputs and Outputs ..................................................................... 5-11
5-2 Controller Instruction and Status Register C .......................................................... 5-12
5-3 Controller Memory Select Register D ..................................................................... 5-13
5-4 Mem_Control Inputs and Outputs .......................................................................... 5-29
5-5 Memory Switching Characteristics ....................................................................... 5-33
6-1 Pre-defined Image Processing Instructions ............................................................ 6-5
A-1 Pre-defined Image Processing Instructions ............................................................. A-18
# Glossary of Terms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>BLM</td>
<td>Behavioral Language Model</td>
</tr>
<tr>
<td>BPCR</td>
<td>Buffer for PC Reads</td>
</tr>
<tr>
<td>BPCW</td>
<td>Buffer for PC Writes</td>
</tr>
<tr>
<td>BPW</td>
<td>Buffer for Processor Writes</td>
</tr>
<tr>
<td>C</td>
<td>A programming language</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>M.S.</td>
<td>Master of Science</td>
</tr>
<tr>
<td>MAP</td>
<td>Morphological Array Processor- the processor of the MIP</td>
</tr>
<tr>
<td>MIP</td>
<td>Morphological Image Processor- the complete image processing system</td>
</tr>
<tr>
<td>PATC</td>
<td>Parallel Add, Tree Compare (an image processing architecture)</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>SAC</td>
<td>Serial Add/Compare (an image processing architecture)</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration or Very Large Scale Integrated circuit</td>
</tr>
</tbody>
</table>
1. Introduction

This thesis describes the author's contributions to the design and construction of a Morphological Image Processor (MIP). Morphology, as described in section 2, is a non-linear method of processing images that is somewhat similar to convolution but uses simpler mathematics, which will allow for simpler circuits to be used. Also involved in the project was Jeffrey Hanzlik, who will also present a related M.S. thesis describing his contributions. The author of this thesis was responsible for the design of the overall architecture, the main controller, the memory controllers, and simulation of those building blocks. Also, the author was responsible for writing the software, in both C and assembly language, to allow a Personal Computer (PC) to have complete control over the MIP. Jeffrey Hanzlik was responsible for the design of the Morphological Array Processor (MAP), which is the processing engine of the MIP, the bus interface, which interfaces the MIP to the PC's data bus, and the Arithmetic Logic Units (ALUs). Also, he was responsible for writing Behavioral Language Models (BLMs) to assist with the simulations and creating the board layout to allow the MIP to be placed on a printed circuit board. Both Jeffrey Hanzlik and the author of this thesis were responsible for the construction, debugging, and testing of the MIP.

The purpose of the project was to explore different architectures for the implementation of a MIP, using a $7 \times 7$ mask to operate on a $512 \times 512$ image, using 9-bit mask and image values, and to demonstrate the feasibility of the architecture. The project was primarily designed using Actel Field Programmable Gate Array (FPGA) chips (A1010A and A1020A chips). Due to the speed limitations of the chips, the MIP cannot run at real-time image rates (typically defined at 30 to 60 frames per second), however the architecture can easily support a real-time rate when the FPGA portion of the circuit is replaced with a faster technology. An expansion of the architecture will also allow for two or more MIPs to be cascaded to allow for a pipelined image processor arrangement.
Figure 1-1 shows the general block diagram of the complete system, which is designed to be operated by a PC. The MIP has been constructed on a printed circuit board, and has a ribbon cable with a card that plugs into one of the PC’s slots to provide access to the PC’s bus. Images, before or after processing, can be viewed using a frame grabber and monitor connected to the PC. Images can also be transferred to and from the PC using a disk, image scanner, printer, or network connection. On the board itself, the
bus interface is responsible for interfacing the rest of the board to the PC. The control unit is responsible for issuing control signals to the rest of the circuit. The processing unit consists of two ALUs and the Morphological Array Processor (MAP) and a volume adder, which collectively perform all the processing tasks. The board also contains four image buffers, each of which can store a complete $512 \times 512$ pixel image.

A user interface program has been written to provide a user-friendly interface to control the MIP. The program, written primarily in C and partially in assembly language, allows the user to control all the functions of the MIP from the PC. A set of the most common instructions (such as erode, dilate, add, etc.) has also been defined to simplify use of the MIP, and a user has the option of adding more commands that have not yet been defined. The program also allows the user to save and load instruction files, mask values, and images to and from a disk, and allows the user to view images on a frame grabber.

The design of the system was completed on HP/Apollo workstations running Mentor Graphics Corporation software. The software tools used include: Neted for schematic capture, Quicksim for simulation, Actel's ALS software for generating the FPGA fuse maps, and Boardstation for printed circuit board layout.
2. Theory of Morphology

Morphology, in general terms, is defined as the study of form and structure. When used in the context of image processing, it is the analysis of geometric structure, or texture. The two fundamental morphological operations are Minkowski addition and Minkowski subtraction. Minkowski addition is defined as

\[ A \oplus B = \bigcup_{b \in B} A + b \]

where \( A \) is the image, \( B \) is the structuring element (which may also be referred to as the patterning element, mask, or window), and the Minkowski sum is the union of the translation of the image \( A \) by all elements \( b \) belonging to \( B \). Similarly, Minkowski subtraction is defined as

\[ A \ominus B = \bigcap_{b \in B} A + b \]

with the Minkowski difference being the intersection of the translation of the image \( A \) by all elements \( b \) belonging to \( B \).

Since this thesis is based on two-dimensional images and two-dimensional structuring elements, both composed of gray-scale values, the discussion of morphology will primarily be concerned with those conditions. However, a binary example will be shown first to illustrate the effects of morphological image processing. Figure 2-1 shows an original image \((A)\), a structuring element \((B)\), a Minkowski addition using the structuring element to operate on the image \((A \oplus B)\), and a Minkowski subtraction \((A \ominus B)\). Note that the Minkowski addition expands the image by the radius of the structuring element, simulating the effect of the structuring element rolling around the outside of the image, due to the effect of the union operation. The Minkowski subtraction reduces the image by the radius of the structuring element, simulating the effect of the structuring element rolling around the inside of the image, due to the effect of the intersection operation.
Dilation and erosion are the two primary morphological operations that the Morphological Image Processor (MIP) prototype is capable of performing, and both are related to the fundamental morphological operations, Minkowski addition and Minkowski subtraction. Dilation is defined as the equivalent of Minkowski addition:

\[ \mathcal{D}(A,B) = A \oplus B \]

and erosion is defined as

\[ \mathcal{E}(A,B) = A \ominus B^\wedge \]

where \( B^\wedge \) is defined as the reflection of the structuring element \( B \) through its origin. In the case of the MIP, the structuring element is a \( 7 \times 7 \) array, with its origin in the center of the array. The reflection of the structuring element takes place in three dimensions (the third dimension is the amplitude of each of the elements), therefore the reflection is a 180° rotation of the \( 7 \times 7 \) array along with a negation of all the array's elements.

To determine the result of a dilation, the structuring element is passed over every pixel of the image, and each pixel of the structuring element is added to its corresponding pixel in the image. With gray scale morphology, the equivalent of the union operation is the maximum value of all the resulting additions, which replaces the image pixel
underneath the origin of the structuring element. The structuring element is then passed over the rest of the image, which each pass yielding a new result in the filtered image. Erosion works in much the same way, but the structuring element must be rotated 180° and all its elements must be negated, and the gray-scale equivalent of the intersection is the minimum value of the additions.

Another feature of the MIP is its ability to handle invalid pixels, also referred to as negative infinity values, as that is how they are mathematically interpreted. Invalid pixels will be referred to as * in this section, and as $-\infty$ in all the subsequent sections. When any value is added to or subtracted from an invalid pixel, the result will be an invalid pixel. When performing maximums and minimums, the invalid pixel is interpreted as negative infinity, therefore when a dilation (maximum) is being performed, the result can never be invalid unless both values being compared are invalid. When an erosion (minimum) is being performed, any comparison involving an invalid pixel will result in an invalid pixel. In the MIP, the primary purposes of invalid pixels are to declare the outside edges of images (beyond the valid pixels contained in the image) invalid, or to reduce the size of the structuring element. However, invalid pixels are permitted to show up anywhere within the image or the structuring element.

As an example of dilation, consider the following 4 × 4 gray-scale image on the left and the 3 × 3 structuring element on the right:

$$A = \begin{pmatrix} 2 & 3 & 1 & 2 \\ 1 & 0 & 3 & 0 \\ 2 & 4 & 5 & 1 \\ 0 & 4 & 3 & * \end{pmatrix} \quad B = \begin{pmatrix} 1 & 3 & 2 \\ 2 & 0 & 1 \\ 4 & 2 & 0 \end{pmatrix}$$

Placing the origin of the structuring element (where the value is 0) over the top left corner of the image (where the value is 2) and performing additions on all the overlapping values, the following result (in matrix form) is obtained:
Note that wherever there is no addition to be performed (the top and left sides), an invalid pixel is the result due to the invalid pixels outside the image.

By placing the origin of the structuring element over the rest of the pixels in the image and performing the additions, the following results (in matrix form) are obtained:

\[
\begin{bmatrix}
* & * & * \\
* & 2 & 4 \\
* & 3 & 0
\end{bmatrix}
\]

By computing the maximum value of each of the matrixes and placing the results back into an image, the following image is the result of a dilation:

\[
\mathcal{D}(A,B) = A \oplus B = \begin{bmatrix}
4 & 5 & 5 & 7 \\
5 & 6 & 8 & 9 \\
5 & 6 & 8 & 7 \\
6 & 7 & 8 & 6
\end{bmatrix}
\]

To perform an erosion on the same image using the same structuring element, the structuring element must first be reflected through its origin, resulting in the following structuring element:
Following the same procedure as with the dilation, but using $B^\wedge$ instead of $B$ as the structuring element, the following results (in matrix form) are obtained:

\[
B^\wedge = \begin{pmatrix}
0 & -2 & -4 \\
-1 & 0 & -2 \\
-2 & -3 & -1
\end{pmatrix}
\]

By computing the minimum value of each of the matrixes and placing the results back into an image, the following image is the result of a dilation:

\[
\epsilon(A,B) = A \ominus B^\wedge = \begin{pmatrix}
* & * & * & * \\
* & -3 & -2 & * \\
* & -2 & * & * \\
* & * & * & *
\end{pmatrix}
\]
3. Morphological Array Processor Architecture

To successfully perform a morphological operation, an image and a window composed of patterning elements are needed. The window is “slid” across the image such that the middle element of the window crosses over every pixel in the image. The image pixel that is under the center window element is the pixel that is being replaced in the filtered image; this pixel will be referred to as the target pixel. Operations are performed using all of the window values and the image values that lie directly beneath the window values. The resulting pixel value will take the place of the original target pixel value.

When the construction of a Morphological Image Processor prototype was proposed, it was decided that the image would be 512 × 512 pixels using a 9-bit gray scale pixel representation. The ninth bit is used to represent negative infinity and negative pixel values. The meaning of −∞ was discussed in the Morphological Theory section and since it was possible to use the extra bit to also represent negative pixel values, they were included in case a use is ever found for them. Table 3-1 shows the binary pixel values and their corresponding decimal values. The gray-scale values are represented using the binary two’s complement code. Since most image capture and processing systems can only deal with 8-bit gray scale images and have no concept of −∞ and negative pixel values, the 9-bit gray scale system will be referred to as an extended 8-bit gray scale system. The window was chosen to be a 7 × 7 array of patterning elements, also using the extended 8-bit gray scale pixel

<table>
<thead>
<tr>
<th>binary</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 0000</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 0001</td>
<td>1</td>
</tr>
<tr>
<td>0 0000 0010</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0 1111 1101</td>
<td>253</td>
</tr>
<tr>
<td>0 1111 1110</td>
<td>254</td>
</tr>
<tr>
<td>0 1111 1111</td>
<td>255</td>
</tr>
<tr>
<td>1 0000 0000</td>
<td>−∞</td>
</tr>
<tr>
<td>1 0000 0001</td>
<td>−255</td>
</tr>
<tr>
<td>1 0000 0010</td>
<td>−254</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1 1111 1101</td>
<td>−3</td>
</tr>
<tr>
<td>1 1111 1111</td>
<td>−2</td>
</tr>
<tr>
<td>1 1111 1111</td>
<td>−1</td>
</tr>
</tbody>
</table>

Table 3-1 Binary Pixel Values and Their Decimal Equivalents
representation. It is also possible to represent $3 \times 3$ and $5 \times 5$ patterning elements under appropriate conditions.

While choosing an architecture for the Morphological Image Processor prototype, there were a few goals that were desirable to achieve. One of these goals was to allow future revisions of the processor to operate at a real-time rate (30 to 60 images per second) using the same $512 \times 512$ pixel, extended 8-bit gray scale images and a $7 \times 7$ window. Implementing the design using VLSI is currently the best way to achieve the real-time rate, so a regular structure is desirable to facilitate a VLSI layout. Another goal was a simple control section to facilitate expansion to larger window sizes and larger images. Also, the processor should be designed such that it is easy to pipeline with identical processors and allow the inputs and outputs to be connected to real-time sources and destinations, possibly with some buffering to make the system compatible with interleaved scan line systems.

For the Morphological Image Processor prototype, a $512 \times 512$ image and a $7 \times 7$ window was used, so all examples and explanations will use these sizes unless otherwise noted. All of the concepts discussed in this section will work with any size image and window as long as appropriate adjustments are made. All index values will start at 0 and end at one less than the maximum value, with the first index value referring to the row and the second one referring to the column. For example, the top left pixel of a $512 \times 512$ image will be referred to as $X_{0,0}$, the top right pixel will be referred to as $X_{0,511}$, the bottom left pixel will be referred to as $X_{511,0}$, and the bottom right pixel will be referred to as $X_{511,511}$. The same notation will be used for window values.

As previously discussed in the Morphological Theory section, the morphological image processing operation is similar to convolution, with additions replacing the multiplications, and comparisons replacing the final additions. When a $512 \times 512$ image is operated on by the values of a $7 \times 7$ window, every pixel in the image along with its 48 neighbors are added to their corresponding window values (with the window centered
over the pixel being operated on). Then either the largest or smallest of those 49 sums, depending on the operation, is the resulting pixel value. The general mathematical expression follows:

\[
\text{compare} \left( X_{i+3,j+3} + W_{6,6}, X_{i+3,j+2} + W_{6,5} \ldots X_{i-3,j-3} + W_{0,0} \right)
\]

where the operands of the compare are 49 separate additions and the \textit{compare} is either the largest or smallest of the 49 results, depending on the desired morphological operation (maximum for a dilation, minimum for an erosion). In the equation, \( X \) is the input image, \( Y \) is the output image, \( W \) is the window matrix, \( i \) is the row index of the pixel being operated on (the target pixel), and \( j \) is the column index. A better indication of the 49 separate additions that are performed can be seen in figure 3-1. A 512 \( \times \) 512 image is shown with a 7 \( \times \) 7 subset of that image, with the subset denoting a portion of the image at any given time that will be added to the window matrix. The window matrix is shown to the right of the 7 \( \times \) 7 matrix, with \( X_{i,j} \) being the target pixel. The matrices are rotated 180° with respect to conventional matrix representation to be consistent with future references, so that the image pixels entering the window array can be shown entering in the top left corner. At the edges of an image, some of the values in the image matrix will not be valid (for example, when \( X_{0,0} \) is the target pixel, all pixels to the bottom and to the right in the matrix will have negative indices making them invalid). Depending on the specific architecture used, there are different ways to handle that situation, and will be discussed in the appropriate sections.

Two architectures were examined for the image processor. The first architecture examined, which utilizes adder and comparator blocks in a serial alignment, will be referred to as the SAC (Serial Add/Compare) architecture and is described in section 3.2. The next architecture examined, which was implemented in the Morphological Image Processor, utilizes a parallel adder array and a comparison tree and will be referred to as
the PATC (Parallel Add, Tree Compare) architecture. The PATC architecture is explained in the next section, and the two architectures are compared in section 3.3.
3.1 PATC Architecture

This section describes the architecture which was used in the Morphological Image Processor. It utilizes a parallel adder array followed by a comparison tree and is referred to as the PATC (Parallel Add, Tree Compare) architecture. The PATC architecture is very suitable for pipelining, with the input format being the same as the output format. Figure 3-2 shows the general block diagram of the architecture, which includes the MAP (Morphological Array Processor) and six external 512 x 9-bit FIFOs. Since this figure is meant to illustrate data flow, control signals have been omitted. $X$ refers to input image.

---

**Figure 3-2** Morphological Array Processor Architecture
values, \( Y \) refers to the resulting image, and \( W \) refers to window values. Both the window array and the adder array are rotated 180° with respect to the normal visual orientation of an array for imaging operations, with the window value \( W_{0,0} \) being in the lower right corner, \( W_{0,6} \) in the lower left corner, \( W_{6,0} \) in the upper right corner, and \( W_{6,6} \) in the upper left corner, as shown on the left side of figure 3-1. This orientation is used so that the data can be shown going into the top left of the array and coming out of the bottom right, which also coincides with the physical orientation of the processor array on the Morphological Image Processor prototype board. The FIFOs are external to the MAP, as that configuration will allow any image size to be used with the same MAP, as long as the right size FIFOs are used and the control circuitry is appropriately modified.

Before processing begins, the window elements are serially loaded into the 49 \( W \) registers contained in the adder blocks. The pixels of the first row of the 512 × 512 image are sent into the MAP from left to right, at a rate of one pixel per clock cycle, immediately followed by the next row, also being sent in from left to right, continuing until all the rows have been sent into the MAP. The pixels are also simultaneously being sent into the 512-stage FIFOs, which are being used as line delays. Each of the FIFOs output image data that is exactly one image row above the output of the previous FIFO. For example, when FIFO #6 is sending the first row of image data into the last row of adder blocks, FIFO #5 is sending the second row of image data into the second to last row of adder blocks, FIFO #4 is sending the third row of image data into the fifth row of adder blocks, and so on. This will cause a 7 × 7 array of image data to be in the adder array at any given time, which will allow 49 simultaneous additions with the window patterning elements.

At any given time during processing, the pixel that is being operated on (the target pixel) is in the middle of the 7 × 7 array of adder blocks. All pixel values in the array are being added to the window values residing in the same adder block with the exception of pixels that are blanked. Blanking occurs when the edges of the image are being operated on, and when unwanted pixels are in the 7 × 7 array. Pixels from the opposite edge, pixels
from the previous image, pixels from the next image, or unknown values when no image is
directly preceding or following the image being processed are classified as unwanted
pixels. When a row and/or column is blanked, that row and/or column is not used for
calculating the new target pixel value. For example, row blanking invalidates pixels from
the right edge of the image when the left edge is being processed, and column blanking
invalidates pixels from a previous image (or no image at all) when the first few rows of an
image are being processed.

Figures 3-3 through 3-8 show how the blanking is implemented, with the figures
showing $5 \times 5$ arrays and the image values contained in the corresponding adder blocks.
$5 \times 5$ arrays have been used in place of $7 \times 7$ arrays for space considerations. The first
and last rows and the first and last columns of a $7 \times 7$ array have been removed to create a
$5 \times 5$ array, and the difference in the blanking procedure will be discussed later in this
section. Each figure shows six consecutive cycles (in left to right, then top to bottom
order) which surround a critical moment in the blanking process. However, there is no
continuity between the figures. Shaded rows and/or columns indicate which rows and/or
columns are being blanked and the values in bold face indicates the target pixels. Any
pixel value with a `prev` or `next` indicates a pixel from the previous or next image,
respectively. Whenever a row and/or column needs to be blanked, the whole row and/or
column must get blanked, as all of the pixels in that row and/or column are not valid for
operating on the target pixel. In the example figures, each image is followed by another
image without any delay, so it can be seen that in a real-time application no delay between
images is needed. If desired, any delay greater than zero cycles can also be used, as any
value that is not in the current image is always blanked.
### Figure 3-3 Blanking Example (part 1)

<table>
<thead>
<tr>
<th>2.1</th>
<th>2.0</th>
<th>1.511</th>
<th>1.510</th>
<th>1.509</th>
<th>1.508</th>
<th>1.507</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>2.1</td>
<td>2.0</td>
<td>1.511</td>
<td>1.510</td>
<td>1.509</td>
<td>1.508</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.3</th>
<th>2.2</th>
<th>2.1</th>
<th>2.0</th>
<th>1.511</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>2.3</td>
<td>2.2</td>
<td>2.1</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.5</th>
<th>2.4</th>
<th>2.3</th>
<th>2.2</th>
<th>2.1</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>2.5</td>
<td>2.4</td>
<td>2.3</td>
<td>2.2</td>
<td>2.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prev. 511,1</th>
<th>prev. 511,0</th>
<th>prev. 510,511</th>
<th>prev. 510,510</th>
<th>prev. 510,509</th>
<th>prev. 510,508</th>
<th>prev. 510,507</th>
</tr>
</thead>
<tbody>
<tr>
<td>511,1</td>
<td>511,0</td>
<td>511,1</td>
<td>511,0</td>
<td>511,1</td>
<td>511,0</td>
<td>511,1</td>
</tr>
<tr>
<td>510,1</td>
<td>510,0</td>
<td>510,1</td>
<td>510,0</td>
<td>510,1</td>
<td>510,0</td>
<td>510,1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prev. 511,3</th>
<th>prev. 511,2</th>
<th>prev. 511,1</th>
<th>prev. 511,0</th>
<th>prev. 510,511</th>
<th>prev. 510,510</th>
<th>prev. 510,509</th>
<th>prev. 510,508</th>
<th>prev. 510,507</th>
</tr>
</thead>
<tbody>
<tr>
<td>511,3</td>
<td>511,2</td>
<td>511,1</td>
<td>511,0</td>
<td>510,511</td>
<td>510,510</td>
<td>510,509</td>
<td>510,508</td>
<td>510,507</td>
</tr>
<tr>
<td>510,3</td>
<td>510,2</td>
<td>510,1</td>
<td>510,0</td>
<td>509,511</td>
<td>509,510</td>
<td>509,509</td>
<td>509,508</td>
<td>509,507</td>
</tr>
</tbody>
</table>
Figure 3-4 Blanking Example (part 2)
Figure 3-5 Blanking Example (part 3)
Figure 3-6 Blanking Example (part 4)
Figure 3-7 Blanking Example (part 5)
Figure 3-8 Blanking Example (part 6)
To implement the blanking, control bits are needed to instruct each row and each column whether to blank or not. When a row and/or a column is blanked, the output of every adder in that row and/or column is set to $-\infty$. If a morphological erosion (minimum) is being performed, the $-\infty$ will propagate to the output, which is the desired effect for erosion. For a dilation (maximum), the $-\infty$ will have no effect on the output as $-\infty$ is the smallest defined value, which is also the desired effect. For a $7 \times 7$ window, no more than three rows and three columns can be blanked at any time, and the center row and center column of the adder array can never be blanked because the middle element of the array contains the target pixel. Figure 3-9 shows the bit numbers that will be used for both $5 \times 5$ and $7 \times 7$ windows to refer to the rows and/or columns that are to be blanked. The outer rows and columns have been removed from the $7 \times 7$ array to make the $5 \times 5$ array. In the array on the left side of figure 3-1, all elements with a row or column index offset of $\pm 3$ are the ones that have been removed from the $7 \times 7$ array.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{Blanking_Control_Bits.png}
\caption{Blanking Control Bits}
\end{figure}
Table 3-2 shows what the blanking control bits will be for all row and column index values of the target pixel. For a $7 \times 7$ window operating on a $512 \times 512$ image, no row blanking occurs between target pixel row index values of 3 and 508, as the rows in the adder array are far enough into the image to contain only valid pixels. Likewise, no column blanking occurs between target pixel column index values of 3 and 508, as the pixels from the opposite edge (when near either the left or right edge) are no longer in the adder array. As can be seen in the table, a $7 \times 7$ window needs one more blanking control bit for each edge than a $5 \times 5$ window does (bit 0 on the top and left edges and bit 5 on the bottom and right edges). Each one of those four extra control bits (two for row blanking and two for column blanking) will be blanked one more cycle than its neighboring bit. For example, when the target pixel is $X_{0,0}$, $X_{0,1}$, or $X_{0,2}$, column blanking control bit 5 will specify that its corresponding column should be blanked.

Table 3-3 shows more complete row and column blanking control bit values for a $7 \times 7$ window, with most of the critical areas of blanking directly corresponding to the $5 \times 5$ examples in figures 3-3 through 3-8. In the gaps between the left side and the right side of the image, no column blanking takes place and row blanking only occurs at the upper and lower edges of the image. In the gap between the upper and lower edges of the image, no row blanking takes place and column blanking continues as usual at the sides of the image.

<table>
<thead>
<tr>
<th>row or column index</th>
<th>$7 \times 7$ blanking control bits</th>
<th>$5 \times 5$ blanking control bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 1 1 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 1 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0 0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>508</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>509</td>
<td>1 0 0 0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>510</td>
<td>1 1 0 0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>511</td>
<td>1 1 1 0 0 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

Table 3-2 Blanking Control Values
<table>
<thead>
<tr>
<th>target pixel (row, column)</th>
<th>row blanking bits</th>
<th>column blanking bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5</td>
<td>0 1 2 3 4 5</td>
</tr>
<tr>
<td>0,0</td>
<td>0 0 0 1 1 1</td>
<td>0 0 0 1 1 1</td>
</tr>
<tr>
<td>0,1</td>
<td>0 0 0 1 1 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>0,2</td>
<td>0 0 0 1 1 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>0,3</td>
<td>0 0 0 1 1 1</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0,508</td>
<td>0 0 0 1 1 1</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>0,509</td>
<td>0 0 0 1 1 1</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>0,510</td>
<td>0 0 0 1 1 1</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>0,511</td>
<td>0 0 0 1 1 1</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>1,0</td>
<td>0 0 0 0 1 1</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>1,1</td>
<td>0 0 0 0 1 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>1,2</td>
<td>0 0 0 0 1 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>1,3</td>
<td>0 0 0 0 1 1</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1,508</td>
<td>0 0 0 1 1 1</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>1,509</td>
<td>0 0 0 1 1 1</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>1,510</td>
<td>0 0 0 1 1 1</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>1,511</td>
<td>0 0 0 1 1 1</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>2,0</td>
<td>0 0 0 0 0 1</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>2,1</td>
<td>0 0 0 0 0 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>2,2</td>
<td>0 0 0 0 0 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>2,3</td>
<td>0 0 0 0 0 1</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,508</td>
<td>0 0 0 0 0 1</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>2,509</td>
<td>0 0 0 0 0 1</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>2,510</td>
<td>0 0 0 0 0 1</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>2,511</td>
<td>0 0 0 0 0 1</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>3,0</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>3,1</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>3,2</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>3,3</td>
<td>0 0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3-3 Complete Blanking Control Values
<table>
<thead>
<tr>
<th>target pixel (row,column)</th>
<th>row blanking bits</th>
<th>column blanking bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5</td>
<td>0 1 2 3 4 5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>508,508</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>508,509</td>
<td>0 0 0 0 0</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>508,510</td>
<td>0 0 0 0 0</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>508,511</td>
<td>0 0 0 0 0</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>509,0</td>
<td>1 0 0 0 0</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>509,1</td>
<td>1 0 0 0 0</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>509,2</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>509,3</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>509,508</td>
<td>1 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>509,509</td>
<td>1 0 0 0 0</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>509,510</td>
<td>1 0 0 0 0</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>509,511</td>
<td>1 0 0 0 0</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>510,0</td>
<td>1 1 0 0 0</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>510,1</td>
<td>1 1 0 0 0</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>510,2</td>
<td>1 1 0 0 0</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>510,3</td>
<td>1 1 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>510,508</td>
<td>1 1 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>510,509</td>
<td>1 1 0 0 0</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>510,510</td>
<td>1 1 0 0 0</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>510,511</td>
<td>1 1 0 0 0</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>511,0</td>
<td>1 1 1 0 0</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>511,1</td>
<td>1 1 1 0 0</td>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>511,2</td>
<td>1 1 1 0 0</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td>511,3</td>
<td>1 1 1 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>511,508</td>
<td>1 1 1 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>511,509</td>
<td>1 1 1 0 0</td>
<td>1 0 0 0 0</td>
</tr>
<tr>
<td>511,510</td>
<td>1 1 1 0 0</td>
<td>1 1 0 0 0</td>
</tr>
<tr>
<td>511,511</td>
<td>1 1 1 0 0</td>
<td>1 1 1 0 0</td>
</tr>
</tbody>
</table>

Table 3-3 Complete Blanking Control Values (cont’d)
After all the additions are performed in parallel, there are 49 results waiting to be compared against each other to determine which value is the largest or the smallest, depending on the operation. Since it is impractical to have a 49-input comparator, a comparison tree composed of 2-input comparators and synchronization registers is a much more feasible solution. Figure 3-2 shows the general block diagram of the Morphological Array Processor architecture, which includes the comparison tree. Each comparison stage consists of a comparator and an output holding register, with all of the output holding registers tied to a common clock, so it will take a minimum of 6 clock cycles to complete all of the comparisons needed to generate an output pixel. Since a comparison tree is being used and all comparators are properly clocked and synchronized, the tree will output one resulting pixel value per clock cycle. Each level of the comparison tree contains intermediate results for successive pixels. Further references to intermediate results of an operation on any input pixel $X_{i,j}$ will be denoted as $I_{i,j}$, with $I_{i,j}$ having the possibility of referring to numerous intermediate results.

For example, when the first pixel of the image ($X_{0,0}$) is in the target pixel position (the middle of the adder array), all 49 additions for that pixel are performed simultaneously. On the next clock cycle, the 49 results are sent into the first level of the comparison tree and the results are ready before the end of the clock cycle. Meanwhile, the second pixel of the image ($X_{0,1}$) is in the target pixel position and the 49 additions for that pixel are taking place. On the next cycle, the intermediate results of $X_{0,0}$ ($I_{0,0}$) are sent into the second level of the comparison tree, and $I_{0,1}$ is sent into the first level, and $X_{0,2}$ is now the target pixel. This will continue until $I_{0,0}$ is in the sixth level of the tree and $X_{0,6}$ is the target pixel. On the next cycle, the output of the final comparator in the tree will be the first output pixel of the morphological array processor, $Y_{0,0}$. The proposed tree in Figure 3-2 is only one possible way to implement it, and many other arrangements can be used as long as the intermediate results remain synchronized. For the Morphological Image Processor prototype, many more levels were needed, as the adders
in the MAP are split up among 21 FPGA chips, and the comparators are also contained in those 21 chips along with two additional chips to compute the final result. The specific configuration that was used in the Morphological Image Processor prototype is described in full detail in Jeffrey Hanzlik's M.S. thesis.
3.2 SAC Architecture

While deciding how to design the Morphological Image Processor, two different architectures were examined. The previous section describes the architecture that is currently in use with the image processor (the PATC architecture), and this section describes the first architecture that was presented, named the SAC (Serial Add/Compare) architecture because the adder/comparator blocks are aligned in a serial array. The next section compares the two architectures and explains why the PATC architecture was chosen over the SAC architecture.

As previously discussed, the Morphological Image Processor prototype computes the output pixels by sliding an $N \times N$ window over $N \times N$ areas of pixels in the image, performing $N^2$ additions of each window value with its corresponding image value, then $N^2-1$ comparisons to obtain either the absolute minimum (for erosion) or maximum (for dilation) value. This output pixel will be the new image pixel at the target pixel location, which corresponds to the center window value. The architecture implemented in the Morphological Image Processor prototype first performs all of the additions in parallel, then the comparisons are performed using a tree structure which will eventually yield one result. Each result is available at the output one pixel at a time in a raster scan format.

An alternate way to perform a morphological operation on the image is with a systolic array approach using a chain of processors, with registers placed between each processor to ensure proper timing. Each processor will perform an addition of a pixel value and its resident window value, and then a comparison between the result of the addition and an intermediate value from the last processor, which will be a partially computed minimum or maximum output value. All processors, with the exception of the last one, will also pass the input data values to the registers between the processors, so that the input data values can propagate unchanged through all of the processors. For this example, it is assumed that each processor can perform its computations within one clock
period and will hold its outputs (the intermediate value and the inputs) until the beginning of the next clock period. The first processor in the chain will either not perform a comparison or perform a comparison in which the value of the addition is the guaranteed output (the intermediate value which will be passed to the next processor), and the result of the comparison of the last processor in the chain will be the output pixel.

Figures 3-11 and 3-12 show an example of data flow for such a processing array, with each figure showing four consecutive cycles. In the example, a $3 \times 3$ window is shown operating on a $4 \times N$ image, as shown in figure 3-10. The first three rows in figures 3-11 and 3-12 show the input data from the image, and the fourth row is the output data, which will take on intermediate results before reaching the final processor. The rectangles represent the adder and comparator processor with the upper portion containing the input values and the lower portion containing the intermediate output value. The rounded rectangles represent a register between the processors. The output values will propagate from one processor to the next, while the input values will pass through a register between each processor. This allows the output value to meet each necessary input value at the correct processor. Each input data row contains three consecutive lines of the image, entered in a vertical scan format. The first line entered into each row is one line below the previous row's first line, with the first row getting its first line from outside the bounds of the image, using negative infinities as image values. The boundary around the whole image is negative infinity, therefore the last line and the first and last values of each line will also be negative infinities. Only one of the image data values, shown in the figure as boldface text, will be used per cycle. Figure 3-10 shows the top of the example image along with the negative infinity border and the example window, and also indicates the pixels that will be entered into the array for all three rows of the first pass. After the three rows along with the negative infinity border values have been entered into the processing array, a second pass of input data is entered into the processing array, originating three rows below the first pass. Figure 3-11 shows the beginning of the first
pass and figure 3-12 shows the end of the first pass and the beginning of the second pass, with the three negative infinities from the right edge of the first pass overlapping the three negative infinities from the left edge of the second pass, which can be done to decrease the delay between successive output rows to three (the width of the window) cycles.

**Figure 3-10 Example Window and Image**
Figure 3-11 Data Flow of SAC Architecture (part 1)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>2</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>3</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>4</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>5</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>6</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>7</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>8</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>9</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
<tr>
<td>10</td>
<td>X1.1</td>
<td>X2.1</td>
<td>X3.1</td>
<td>X4.1</td>
<td>X5.1</td>
<td>X6.1</td>
<td>X7.1</td>
<td>X8.1</td>
<td>X9.1</td>
<td>X1.0</td>
</tr>
</tbody>
</table>

Figure 3-12 Data Flow of SAC Architecture (part 2)
As can be seen in figures 3-11 and 3-12, at least two of the three input data values are equal at any given time. Therefore, the input data rows can be reduced from three to two by removing the middle input row, with each processor selecting the proper value before performing its processing operation. This will reduce the input bandwidth to two incoming data paths, regardless of the window size.

Several methods may be used to input two data paths into the processor array at once. If memory access is fast enough, two memory reads may be performed during each processing cycle. Otherwise, an image may be stored in two separate memories. While one data path is entering a pixel with an even row index, the other path is entering a pixel with an odd row index. An exception is when both paths are entering the same pixel, where only one memory read is necessary. Therefore all even rows may be stored in one memory unit, while all odd rows may be stored in another memory unit. The same is true for columns, so even and odd columns may be split into two different memories instead of the rows, depending on which is more convenient to implement. Alternately, a cache may be used to store pixels until they are needed again, as the pixels entering the top row (when they aren't the same as the pixels entering the bottom row) are the same pixels that had previously entered the bottom row. For a $W \times W$ window, the cache must store $W - 1$ rows of the image.

To implement the systolic array architecture to process a $512 \times 512$ image using a $7 \times 7$ window, a larger systolic array is needed. Each processor contains one window value, so 49 processors and 48 intermediate registers will be needed. The array can be built using seven input data paths, but as is the case with the $3 \times 3$ window, there will be at most two unique values in each of the seven rows. Those two unique values are guaranteed to be in either the first or last input row, therefore only two input data paths are needed, with a multiplexor selecting one of the two paths using a repeating pattern.
3.3 PATC and SAC Architecture Comparisons

This section compares the PATC (Parallel Add/Tree Compare) and SAC (Serial Add/Compare) architectures, which were both discussed individually in the previous sections. The comparisons will be made assuming a $7 \times 7$ window and a $512 \times 512$ image is used, though other window sizes and image sizes can be substituted. The PATC architecture, as it was implemented in the MIP, preloads the window values before starting an image processing operation, and it will be assumed that any implementation of the SAC architecture will do likewise.

**Processor arrangement:** In the SAC architecture, each processor consists of one adder and one comparator, where all 49 processors are in a serial alignment, with 48 intermediate registers between the processors. Both the window and image output data values propagate from one processor to the next, while the image input data values propagate through the processors and the intermediate registers, but also in a serial fashion. The PATC architecture has 49 adders arranged in a $7 \times 7$ array and a tree structure for the 48 comparators, with the 49 inputs to the comparator tree coming directly from each of the 49 adders. The window data values propagate through the first row of the array, then from the last adder of the first row to the first adder of the second row, and continue until they reach the last adder in the last row. The input image values enter each row of the array after passing through a FIFO, with the exception of the first row which gets its input directly from memory, which also feeds the input of the first FIFO. The output image values come directly from the final comparator in the comparison tree. The SAC architecture is clearly the more linear of the two architectures, making almost any implementation of the architecture easier, as the routing between processors is not only identical from one processor to the next but also simpler. The physical layout of the PATC architecture will most likely be more complex than the layout of the SAC architecture.
Memory access for input data: The PATC architecture accesses memory for input data in a non-interlaced raster scan format, allowing the memory address counter to count directly from 0 to $512^2 - 1$ using a simple 18-bit binary counter. The SAC architecture cannot access memory in such a linear fashion. Using the cache method to store data until it is needed again is the simplest memory access method for the SAC architecture, but the memory access will be fairly complex as an odd number of swaths, each containing an odd number of rows, is needed. Also, each swath is entered into the processing array one column at a time, making it more complex than entering complete rows into the array. Using either of the non-cache methods will require two such non-linear counters operating at the same time. The PATC architecture allows much simpler memory access than the SAC architecture does, which not only makes counter design much easier, but makes system debugging easier.

Memory access for output data: The output data of the PATC architecture is in the same format as the input data (non-interlaced raster scan format), therefore the same 18-bit counter circuit can be used for addressing the memory for the output data as is used for the input data. The SAC architecture will output data in a somewhat similar format as the input data, in 512 columns with a height of seven pixels for a $7 \times 7$ window and a $512 \times 512$ image. There will also be a delay of a minimum of seven pixels between each swath of seven rows. As is the case with input memory access, the PATC architecture allows much simpler memory access than the SAC architecture.

Pipelining: Since the input data of the PATC architecture is in the same format as the output data, the architecture is very suitable for pipelining multiple image processors, as the output of one image processor can be directly routed to the input of the next image processor. The SAC architecture, which has incompatible input and output data formats, cannot be pipelined so easily.
**Memory support:** The PATC architecture is required to have line delays to store complete rows of the image so the adder array can be operating on a 7 × 7 area of the image at all times. The size of the window determines the number of line delays needed (six are needed for a 7 × 7 window), and the width of the image determines how many stages the line delays need (FIFOs or shift registers of length 512 are needed for an image which is 512 pixels wide). If the SAC architecture is using a cache to simplify memory access, it needs the same amount of FIFO or shift register memory as the PATC architecture needs, except no taps are needed for each row. If no cache is being used, no memory support is needed, and the SAC architecture can use less circuitry than the PATC architecture at the expense of having more complex memory access. Otherwise, both architectures need the same amount of support memory, with the SAC architecture only needing one input and one output and the PATC memory needing one input and six outputs.

**Size of processing array:** Since the SAC architecture needs intermediate registers between the processors, 48 extra 9-bit registers are required which aren’t necessary for the PATC architecture.

**Real-time processing:** To facilitate real-time processing, an architecture should be able to be implemented with little or no memory to store images, using a real-time digital input (from a video camera, for example) and feeding the output to another device with real-time digital input (such as a video screen). The PATC architecture can read and write using a non-interlaced raster scan format, and can have as little as no delay between sequential images, which makes real-time processing quite feasible. The SAC architecture has input and output formats that are not only incompatible with each other, but very unlikely to be compatible with any real-time input or output formats. Also, there must be a delay between sequential images, and there are delays between adjacent swaths of output pixels, requiring the system to operate at a clock speed that is slightly faster than that of
the real-time source or destination, along with large buffers between the systems with different clock speeds. The PATC architecture has a clear advantage over the SAC architecture for allowing real-time processing of sequential images.

**3-dimensional image processing:** For the PATC architecture to be able to allow processing of a 3-D image using a 3-D window, much more memory is needed. Instead of the 1-D line delays that are necessary for 2-D processing, 2-D plane delays are needed for 3-D processing. For a $W \times W \times W$ window operating on an $N \times N \times N$ image, $W-1 \ N \times N$ FIFOs or shift registers are necessary, with each $N \times N$ FIFO or shift register needing $W-1$ taps every $N$ pixels, similar to the 2-D line delay. The SAC architecture, assuming no cache for input data memory access is used, is much more suitable for 3-D processing. Appropriate changes must be made to the memory access hardware and the processor array to facilitate 3-D processing, but the large amount of memory needed for the PATC architecture is not needed for the SAC architecture.

While both architectures have advantages over the other architecture, for most applications one architecture will be more suitable than the other architecture. The SAC architecture has a processor arrangement that is easier to route in hardware, no need for FIFOs or shift registers if no cache for easier memory access is used, and is more suitable for 3-D image processing. The PATC architecture has easier input and output memory access, allows pipelining of multiple image processors, and is very suitable for real-time image processing. Because the advantages of using the PATC architecture outweigh the disadvantages, it was decided that the PATC architecture would be used for designing the prototype for a real-time Morphological Image Processor.
4. Morphological Image Processor Architecture

Figure 4-1 Block Diagram of the Morphological Image Processor
Figure 4-2 Block Diagram of Memories

Figure 4-3 Block Diagram of Processors
Figure 4-4 Morphological Image Processor Schematic
A complete system, called the Morphological Image Processor (MIP), provides control and support circuitry that allows the Morphological Array Processor (MAP) to function. The system was designed to be placed on a printed circuit board and connected to a PC. A main controller is needed to provide control signals for the MAP and the rest of the system, memories and memory controllers are needed to store and use images on the board, ALUs and a volume adder increase the functionality of the MIP by allowing extra operations, a bus interface is necessary for interfacing the board to a PC, and tri-state buffers are needed to regulate access to the PC bus and on-board buses.

Figure 4-4 shows the top level schematic of the MIP, which will be discussed in this section. Figure 4-1 shows a more general block diagram of the MIP, with vague control signals but a clearly illustrated bus structure. Since the control signals and details of the inner workings of the building blocks will be discussed in other sections, this section will concentrate on the top level design and the data and address buses.

From figure 4-1, it can be seen that there are four major building blocks of the MIP, each represented by a rectangle. The Memories and Processors blocks are composed of multiple chips, with their block diagrams shown in figures 4-2 and 4-3, respectively. The Controller and Bus Interface blocks, along with the buffers BPCR (Buffer for PC Reads), BPCW (Buffer for PC Writes), and BPW (Buffer for Processor Writes), are single chips.

The block labeled Controller is the main controller that sends out control signals to the rest of the system. It also contains the registers that inform the PC of the status of each processing operation, such as indicating that a processing operation has been completed. The block labeled Memories contains the four memory controllers (named Mem Control), four memories (each capable of storing one image), and the eight tri-state buffers to allocate bus access for the X1 and X2 buses. The block labeled Bus Interface is the bus interface, which sends register read and write control signals to the rest of the system, allows the PC to access the on-board memory, and controls the tri-state buffers.
that allocate bus access for the PC data and memory data buses. The block labeled *Processors* is the processing engine of the system and includes the MAP, two ALUs, and the volume adder.

The three major operations that the MIP is capable of performing are writing images to the board, reading images from the board, and processing operations. Other operations, which are transparent to a user of the system, include register reads and writes. There are fourteen 8-bit registers on the board and tables 4-1 through 4-8 list those registers and their descriptions. The register numbers (0-8 and B-F) indicate the offset from the base I/O location of the registers, which are mapped into the lower bounds of the PC's addressable I/O space and can occupy one of four regions. The board's registers are currently addressable between 0300 to 030F (hex), and can be accessed by simple 8-bit I/O port reads and writes. Register numbers 9 and A are unused and available for future expansion.

<table>
<thead>
<tr>
<th>0-4: volume adder registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>write register:</td>
</tr>
<tr>
<td>0:  bit (0): 0 = sum of all X inputs, 1 = sum of all X inputs squared</td>
</tr>
<tr>
<td>read registers:</td>
</tr>
<tr>
<td>0:  bits (7-0) = volume adder results bits (7-0)</td>
</tr>
<tr>
<td>1:  bits (7-0) = volume adder results bits (15-8)</td>
</tr>
<tr>
<td>2:  bits (7-0) = volume adder results bits (23-16)</td>
</tr>
<tr>
<td>3:  bits (7-0) = volume adder results bits (31-24)</td>
</tr>
<tr>
<td>4:  bits (1,0) = volume adder results bits (33,32)</td>
</tr>
<tr>
<td>bit (7) = negative pixel in image flag: 0 = all positive, 1 = at least one negative</td>
</tr>
</tbody>
</table>

Table 4-1 Volume Adder Registers 0-4

<table>
<thead>
<tr>
<th>5: ALU1 constant register (bits 7-0) (write only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7: ALU2 constant (write) and image min/max (read) register (bits 7-0)</td>
</tr>
<tr>
<td>write access: bits (7-0) of the constants for ALU operations are written to these registers, with bit (8) written to the ALU operation registers</td>
</tr>
<tr>
<td>read access: for ALU2 only, bits (7-0) of the image minimum or maximum (min or max. determined by the ALU2 operation) are read from the register, with bit (8) read from the ALU2 operation register</td>
</tr>
</tbody>
</table>

Table 4-2 ALU Constant Registers 5,7
## 6: ALU1 Operation Register (write only)

<table>
<thead>
<tr>
<th>bits (7-4):</th>
<th>bits (3,2):</th>
<th>bits (3,2):</th>
<th>bits (3,2):</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4</td>
<td>bits (3,2) = 0,0</td>
<td>bits (3,2) = 0,1</td>
<td>bits (3,2) = 1,0</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>min (A,B)</td>
<td>min (C,B)</td>
<td>min (A,C)</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>max (A,B)</td>
<td>max (C,B)</td>
<td>max (A,C)</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>copy (A)</td>
<td>copy (C)</td>
<td>copy (A)</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>copy (B)</td>
<td>copy (B)</td>
<td>copy (C)</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>copy (A), min (A) †</td>
<td>copy (C), min (C) †</td>
<td>copy (A), min (A) †</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>copy (A), max (A) †</td>
<td>copy (C), max (C) †</td>
<td>copy (A), max (A) †</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>copy (A)</td>
<td>copy (C)</td>
<td>copy (A)</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>copy (B)</td>
<td>copy (B)</td>
<td>A–C</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>A–B</td>
<td>C–B</td>
<td>A–constant</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>A–constant</td>
<td>C–constant</td>
<td>A–constant</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>A+B</td>
<td>C+B</td>
<td>A+C</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>A+constant</td>
<td>C+constant</td>
<td>A+constant</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>A–B ‡</td>
<td>C–B ‡</td>
<td>A–C ‡</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>A–constant ‡</td>
<td>C–constant ‡</td>
<td>A–constant ‡</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>A+B</td>
<td>C+B</td>
<td>A+C</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>A+constant</td>
<td>C+constant</td>
<td>A+constant</td>
</tr>
</tbody>
</table>

### Notes:

- bits (3,2) = 1,1 is valid but has no unique or useful operations
- bit (1) is not used
- †: the min and max operations with only one operand determine the minimum or maximum value of the whole image passing through ALU2, and the result is stored in ALU2's read registers
- ‡: when the second operand is $-\infty$, the indicated subtractions will return the first operand, whereas the normal subtractions will return $-\infty$ when the second operand is $-\infty$

Table 4-3 ALU Instruction Registers 6,8
**B: process start register (REGS_STARTn) (write only)**

Any write to this register will issue a process start signal.

| Table 4-4 Controller Start Register B |

**C: process instruction and status register (REGS_PIIn)**

<table>
<thead>
<tr>
<th>bit (0): MAP MAX signal</th>
<th>0 = erosion (minimum)</th>
<th>1 = dilation (maximum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit (1): bus mode</td>
<td>0 = X2 bus → ALU1</td>
<td>1 = X2 bus → ALU2</td>
</tr>
<tr>
<td>bit (6): run status</td>
<td>0 = running</td>
<td>1 = OK to start next run</td>
</tr>
<tr>
<td>bit (7): instruction/window load status</td>
<td>0 = not OK to load next instruction or window</td>
<td>1 = OK to load next instruction and window</td>
</tr>
</tbody>
</table>

| Table 4-5 Controller Instruction and Status Register C |

**D: memory select register (REGS_MSn) (write only)**

<table>
<thead>
<tr>
<th>bit #:</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits (7-4) low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>mem0 → PC</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>mem1 → PC</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>mem2 → PC</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>mem3 → PC</td>
</tr>
<tr>
<td>all other combinations</td>
<td>s1</td>
<td>s0</td>
<td>mem3</td>
<td></td>
<td>s1</td>
<td>s0</td>
<td>mem2</td>
<td></td>
<td>s1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>mem → X1 bus</td>
<td></td>
<td>0</td>
<td>1</td>
<td>mem → X2 bus</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>examples:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>mem2 → PC</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>mem0 → X1 bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mem1 → X2 bus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y bus → mem2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Table 4-6 Controller Memory Select Register D |

---
Before performing any operations, the PC must inform the board where the on-board memories reside within the PC's address space. The ADDRCTRL register (register F, table 4-8) in the bus interface contains the MEMENABLE bit that defaults to not allowing window writes and memory access until that bit is set. This allows the user interface program to select the board's address space, which is also determined by writing to the ADDRCTRL register, before any board memory access by the PC is attempted. The board was designed such that all four on-board memories map into the same 512 Kbyte address space, using appropriate register writes to determine which memory is to be
accessed by the PC. The bus interface was designed to allow the on-board memories to reside anywhere within a 16 Mbyte address range, with the base address being any multiple of 512 Kbytes within that address range. The on-board memories currently use a base address of 8 Mbytes, which is in the extended memory range of the PC.

To write an image to the board memory from the PC, the PC must issue an appropriate value to the memory chip select bits of the SEGMENT register in the bus interface (register E, table 4-7), informing the bus interface which of the four memories is to be written to. This will send a control signal from the bus interface to the memory control chip associated with the selected memory, putting the memory control chip in a transparent mode, allowing the address and control signals from the PC bus and bus interface to pass through the memory control chip unaltered. When the PC issues a valid board memory address and a write signal, the bus interface will enable the tri-state buffer labeled BPCW (Buffer for PC Write), turning control of the memory data bus over to the PC's data bus. The bus interface also modifies the PC bus address by removing the base offset from the address, which will send an address ranging from 0 to $2^{18} - 1$ (the highest valid address of the memory chip) to the memory control chips. The PC will have complete control of the memory for write operations, and from the perspective of the PC, the on-board memory will be functionally equivalent to extended memory.

A memory read operation, sending an image from the board's memory to the PC, is very similar to the write operation. The same bus interface register is written to, which will select the proper memory and put its memory controller in a transparent mode. A valid address and a read signal will enable the tri-state buffer labeled BPCR (Buffer for PC Read), giving the X1 data bus control of the PC data bus. The memory select register (register D, figure 4-6) in the main controller chip is also loaded, which will select the proper tri-state buffer to get the data from the memory to the X1 data bus. The bus interface modifies the address the same way as for the write cycle, and the PC will have
complete control of the memory for read operations. From the perspective of the PC, the on-board memory will be functionally equivalent to extended memory.

The last major operation is an image processing operation. An image processing operation can involve one or both ALUs, the \textit{MAP}, and any combination of the ALUs and the \textit{MAP}. To select the processing operation, the proper registers must be written to. Both ALUs have instruction registers (registers 6 and 8, table 4-3), which can also be set to pass data through unaltered. The \textit{MAP} also passes through unaltered data, which is coincident with the processed data output, and can be used by selecting the proper \textit{ALU2} operation. The process instruction register in the main controller (register C, table 4-5) selects the \textit{MAP} operation (either erosion or dilation) and whether the X2 bus will be the input of \textit{ALU1} or \textit{ALU2}. The memory select register (register D, table 4-6) in the main controller is for selecting the input memory or memories and the output memory. The volume adder also has a register (register 0, table 4-1) to select between computing the volume or the sum of the squares. Whenever the \textit{MAP} is being used for processing, the window values must be entered into the \textit{MAP}. This is performed almost like memory write operations by using the PC to write to the extended memory area where the board is mapped to, but any valid on-board memory location may be written to, and a register write to the bus interface distinguishes between a memory write and a window write. A write to main controller's start register will start a processing operation. One memory will then send its output to the X1 bus, and another memory, if specified, will send its output to the X2 bus. The function of the X2 bus depends on the bus mode, which is written to the process instruction register (register C, table 4-5). If the bus mode is 0, the data on the X2 bus will be coincident with the data on the X1 bus and can only be used by \textit{ALU1}. If the bus mode is 1, the data on the X2 bus is coincident with the \textit{MAP}'s output data and can only be used by \textit{ALU2}. The user interface program will select the bus mode depending on the desired processing operation, making the operation of the bus mode transparent to a user of the system. The output of \textit{ALU2}, which is the final output of any
processing operation, will be put on the Y bus. The bus interface will select the tri-state buffer labeled \(BPW\) (Buffer for Processor Writes) to allow the Y bus to send its data to the memory data bus. The memory control chip associated with the memory designated to store the output will send write enable signals to allow writing to the memory.
5. Hardware Design

This section describes the hardware design of the Morphological Image Processor (MIP). The overall system, the main controller, and the memory controller were designed by the author of this thesis, therefore this chapter will concentrate on their design. The bus interface, the Morphological Array Processor (MAP), the ALUs, and the volume adder were designed by Jeffrey Hanzlik. The previous section briefly describes those building blocks, showing their purpose with respect to the whole system and how they interrelate with the other building blocks. A more detailed description of the bus interface, the MAP, the ALUs, and the volume adder can be found in Jeffrey Hanzlik’s thesis.

The design of the MIP was essentially a middle-out design, with the architecture of the MAP being determined first, and the rest of the system being designed based on the requirements of the MAP’s architecture and other features of the MIP. The top-level layout and timing of the complete system were then determined, and then the building blocks of the system, such as the controllers and processors, were designed to meet the timing requirements. The top level of the design was discussed in the previous chapter, and the lower levels will be discussed in this chapter. The overall timing of the system will be discussed first since it was the basis for designing all of the building blocks.
5.1 Timing

Figure 5-1 shows the general timing of the Morphological Image Processor. Note that the figure is continued on the following two pages. All signals shown are either pulses with a width of one clock cycle used for control or buses used to carry address and data values. The value shown for any address or data value is the pixel location expressed in row, column format (in decimal), since it is impractical to show actual pixel data values. All relevant timing is shown for one complete image pass, with the pixel locations and control pulses from that pass shown in boldface. Part of a possible previous and next image are also shown, respectively, before and after the complete image. Each cycle of the complete pass is numbered to provide decimal reference points. The timing diagrams show one image following the next with a delay of zero clock cycles, but any delay of more than zero clock cycles is permissible, since any stray data in the adder array of the MAP will be ignored due to the blanking control signals. All timing waveforms found on the timing diagram are described below:

CLK: The system clock.

CLK2: The system clock delayed by 30 ns, used for proper timing of the memory’s write enable signal.
START_MEM (X1): Originates from the Controller and goes to the Mem_Control chip controlling the memory selected to have its output be the A input of ALU1, using the X1_Bus to carry the image data. At the next rising clock edge after the signal is asserted, the memory address counter in each selected Mem_Control chip will start counting, and during the same clock cycle, that address (X1 Addr) will get sent to the memory’s address input and the memory will output data (X1 Data) onto the appropriate data bus. When the bus mode is 0 (both X1_Bus and X2_Bus are inputs of ALU1), this signal will also go to the Mem_Control chip controlling the memory selected to have its output put on the X2_Bus to be the B input of ALU1.

X1 Addr: The address generated by the Mem_Control chip controlling the memory designated to drive the X1_Bus. When the bus mode is 0, this address is also generated by the Mem_Control chip controlling the memory designated to drive the X2 Bus.

X1 Data: The data contained in the memory selected to drive the X1_Bus at the address generated by the Mem_Control chip. After the address is generated, the data is ready to be clocked into the input flip-flops of ALU1 at the next rising clock edge. When the bus mode is 0, the address is also used for the X2_Bus data.

INIT_ALU1: Instructs ALU1 to load its next instruction upon the next rising clock edge. This initializes ALU1 at the same time that the first valid image pixel (0,0) is its operand.

ALU1 Operand: The output of the input flip-flops of ALU1. This is the pixel that is being operated on by ALU1 during any given clock period. The operand pixel gets clocked into ALU1’s output flip-flops at the next rising clock edge.
**ALU1 Output**: The output data of the output flip-flops of ALU1, which gets clocked into the input of the MAP at the next rising clock edge.

**first MAP Operand**: The operand of the first adder in the MAP.

**Start_Blank.Counter**: Instructs the blank counter to start counting at the next rising clock edge.

**START_PROC**: Informs the MAP that the first valid image pixel will be in the target pixel position upon the next rising clock edge. The MAP uses this signal to latch the window values and the desired morphological operation for the next image processing operation.

**blank_counter count**: The address of the blank counter, which is used to generate the row and column blanking signals. The blank counter address is one clock cycle ahead of the target pixel because of the decoding scheme used, which is described in section 5-2.

**target pixel**: The pixel being operated on by the adder in the middle of the MAP, which also corresponds to the middle of the window.

**C1 operand**: The pixels in the first level of the comparison tree. All 49 adders in the MAP perform a simultaneous addition of a potential result, so there are 49 pixels that are inputs to the first comparison tree level.

**C13 operand**: The pixels in the last (13th) level of the comparison tree. The result of the comparison, which is the output value of the MAP, gets clocked into the output flip-flops of the MAP.
**MAP output**: The output data of the output flip-flops of the MAP.

**START_MEM (X2)**: Originates from the Controller and goes to the Mem_Control chip controlling the memory selected to have its output be the C input of ALU2, using X2_Bus to carry the image data. At the next rising clock edge after the signal is asserted, the memory address counter in the selected Mem_Control chip will start counting, and during the same clock cycle, that address (X2_Addr) will get sent to the memory’s address input and the memory will output data (X2_Data) onto the X2_Bus. This signal will only be issued at the time specified on the timing diagram when the bus mode is 1. When the bus mode is 0, START_MEM (X1) illustrates the timing for both X1_Bus and X2_Bus control signals.

**X2_Addr**: The address generated by the Mem_Control chip controlling the memory designated to drive the X2_Bus when the bus mode is 1. When the bus mode is 0, this address will not be generated.

**X2_Data**: The data contained in the memory selected to drive the X1_Bus at the address generated by the Mem_Control chip when the bus mode is 1. After the address is generated, the data is ready to be clocked into the input flip-flops of ALU1 at the next rising clock edge. When the bus mode is 0, this data will not be generated.

**INIT_ALU2**: Instructs ALU2 to load its next instruction upon the next rising clock edge. This initializes ALU2 at the same time that the first valid image pixel (0,0) is its operand.
**STOP_ALU2:** Informs \textit{ALU2} that at the next rising clock edge, the last valid pixel of the image being processed is on its output. This is used to capture the image minimum or maximum if that ALU operation was specified. This signal is also sent to the volume adder to indicate the end of the image.

**ALU2 Operand:** The output of the input flip-flops of \textit{ALU2}. This is the pixel that is being operated on by \textit{ALU2} during any given clock period. The operand pixel gets clocked into \textit{ALU2}'s output flip-flops at the next rising clock edge.

**ALU2 Output:** The output of the output flip-flops of \textit{ALU2}. The output pixel gets clocked into the input of the volume adder at the next rising clock edge, and this is also the final output which will be written into the memory selected to contain the output image.

**WRITE_MEM (Y):** Originates from the \textit{Controller} and goes to the \textit{Mem_Control} chip controlling the memory selected to receive the final image output of a processing operation. Starting at the first rising clock edge after the \textit{START_MEM} signal is sent to the same \textit{Mem_Control} chip, the chip will issue a \textit{WEn} (write enable) signal to its associated memory for every pixel of the image.

**START_MEM (Y):** Originates from the \textit{Controller} and goes to the \textit{Mem_Control} chip controlling the memory selected to receive the final image output of a processing operation. At the next rising clock edge, the memory address counter in the selected \textit{Mem_Control} chip will start counting, and during the same clock cycle, that address (\textit{Y Addr}) will get sent to the memory's address input. Also, the \textit{WEn_ENABLE} signal is asserted, which enables writing to the memory.
**Mem.Counter/DONE**: Internal to the *Mem_Control* chip, this signal will disable the WEn_ENABLE signal after the last pixel of the image, which allows only valid image pixels to be written to the destination memory.

**WEn_ENABLE**: Internal to the *Mem_Control* chip, this signal (when active) will allow the chip controlling the destination memory to issue WEn (write enable) signals to the memory.

**WEn**: Originates from the *Mem_Control* chip controlling the destination memory, and goes to the write enable input of that memory. This signal is an inversion of the CLK2 signal whenever the valid image output data is present at the memory’s data bus.

**Y Addr**: The address generated by the *Mem_Control* chip controlling the memory designated to receive the final output of the processing operation (from ALU2).
Figure 5-1  System Timing Diagram (part 1)
Figure 5-1  System Timing Diagram (part 2)
Figure 5-1 System Timing Diagram (part 3)
5.2 Main Controller

The Controller chip is in charge of controlling most other chips on the Morphological Image Processing board. Processing instructions are written directly to the Controller, and these instructions are used to generate the correct control signals. These control signals control the Mem_Control chips, the buffers at the outputs of the memories which drive the two data buses, the MAP, both ALUs, and the volume adder. Table 5-1 briefly describes all the inputs and outputs, with a more detailed description following the table. Subsequently, the schematic of the Controller will be shown and the generation of the signals will be explained.

<table>
<thead>
<tr>
<th>signal</th>
<th>I/O</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGS_PIn</td>
<td>I</td>
<td>processor instruction and status register (active low)</td>
</tr>
<tr>
<td>REGS_MSn</td>
<td>I</td>
<td>memory selection register (active low)</td>
</tr>
<tr>
<td>REGS_STARTn</td>
<td>I</td>
<td>start register (active low)</td>
</tr>
<tr>
<td>REGENW</td>
<td>I</td>
<td>enable write to registers</td>
</tr>
<tr>
<td>REGENR</td>
<td>I</td>
<td>enable read from registers</td>
</tr>
<tr>
<td>CLK</td>
<td>I</td>
<td>system clock</td>
</tr>
<tr>
<td>CLRn</td>
<td>I</td>
<td>global clear upon system power-up (active low)</td>
</tr>
<tr>
<td>SD(7:0)</td>
<td>I/O</td>
<td>data bus for register reads and writes</td>
</tr>
<tr>
<td>START_MEM(3:0)</td>
<td>O</td>
<td>start specified Mem_Control chip</td>
</tr>
<tr>
<td>WRITE_MEM(3:0)</td>
<td>O</td>
<td>instruct specified Mem_Control chip of write operation</td>
</tr>
<tr>
<td>X1_BUS_SELn(3:0)</td>
<td>O</td>
<td>assign specified memory to X1 Bus (active low)</td>
</tr>
<tr>
<td>X2_BUS_SELn(3:0)</td>
<td>O</td>
<td>assign specified memory to X2 Bus (active low)</td>
</tr>
<tr>
<td>INIT_ALU1</td>
<td>O</td>
<td>starts ALU1</td>
</tr>
<tr>
<td>MAX(2:0)</td>
<td>O</td>
<td>specify desired morphological operation</td>
</tr>
<tr>
<td>START_PROC(2:0)</td>
<td>O</td>
<td>instructs MAP to start processing</td>
</tr>
<tr>
<td>ROWBLNK(5:0)</td>
<td>O</td>
<td>instructs MAP to blank a particular row</td>
</tr>
<tr>
<td>COLBLNK(5:0)</td>
<td>O</td>
<td>instructs MAP to blank a particular column</td>
</tr>
<tr>
<td>INIT_ALU2</td>
<td>O</td>
<td>start ALU2</td>
</tr>
<tr>
<td>STOP_ALU2</td>
<td>O</td>
<td>stops ALU2</td>
</tr>
</tbody>
</table>

Table 5-1 Main Controller Inputs and Outputs
REGS_PIn: Selects the processor instruction/status register, which contains two write-only bits (the MAP and bus mode instructions), and two read-only bits (the process status bits). Bit 0, the MAP instruction bit, sends the MAX signal to the MAP, which will determine whether an erosion (MAX=0) or dilation (MAX=1) will be performed. Bit 1 is the bus mode selection which determines whether X2_Bus will be the input of ALU1 (bus mode = 0) or ALU2 (bus mode = 1). Bit 6 is the process status bit which gets turned on when the processor is ready to accept the next instruction. Bit 7 is the process status bit which gets turned on when the processor is done processing and prepared to accept another process start instruction. Table 5-2 describes the bits in greater detail.

| Bit (0): MAP MAX signal | 0 = erosion (minimum) | 1 = dilation (maximum) |
| Bit (1): bus mode | 0 = X2 bus → ALU1 | 1 = X2 bus → ALU2 |
| Bit (6): run status | 0 = running | 1 = OK to start next run |
| Bit (7): instruction/window load status | 0 = not OK to load next instruction or window | 1 = OK to load next instruction and window |

Table 5-2 Controller Instruction and Status Register C

REGS_MSn: Selects the memory select register, which is used to inform the Controller which memory or memories will contain the source image(s), which memory the output image will be written to, and which memory or memories will not be used. The Controller uses this information to route memory control signals to the appropriate Mem_Control chip, to select the bus buffers for processing operations, and allowing the host computer to read from on-board memory. Table 5-3 describes the memory select register bits.
REGS_STARTn: Selects the start register, which will start the processor when the REGENW signal is also active. The contents of the data bus are insignificant when issuing the REGS_STARTn signal.

REGENW: Enables the host computer to write to the selected register. The Controller has three writable registers: the processor instruction/status register, the memory select register, and the start register, selected by active REGS_PIn, REGS_MSn, and REGS_STARTn signals, respectively.

REGENR: Enables the host computer to read from the selected register. The Controller has one readable register: the process instruction/status register, which is selected by an active REGS_PIn signal.

SD(7:0): The eight-bit data bus for register reads and writes.
CLK: Main system clock.

CLRn: Clears all flip-flops upon system power-up.

START_MEM(3:0): Issues a start signal to the Mem_Control chip(s) specified by the active bit(s). This will instruct the selected Mem_Control to start its memory address counter.

WRITE_MEM(3:0): Instructs the Mem_Control chip specified by the active bit that its associated memory will be written to, starting when the START_MEM signal is issued. This will cause the selected Mem_Control to issue write enable signals to its memory chip during valid memory addresses.

X1_BUS_SELn(3:0): Assigns the memory specified by the active bit to the X1_Bus by enabling the buffer connected between the memory’s output and the bus.

X2_BUS_SELn(3:0): Assigns the memory specified by the active bit to the X2_Bus by enabling the buffer connected between the memory’s output and the bus.

INIT_ALU1: Instructs ALU1 to load its next instruction upon the next rising clock edge. This initializes ALU1 at the same time that the first valid image pixel is its operand.

MAX(2:0): Informs the MAP whether the next operation is to be an erosion (MAX=0) or a dilation (MAX=1). The three bits of the MAX(2:0) signal are identical, but are duplicated due to fan-out restrictions, as 23 of the MAP chips use this signal.
**START_PROC(2:0):** Informs the MAP that the first valid image pixel will be in the target pixel position upon the next rising clock edge. The MAP uses this signal to latch the window values and the desired morphological operation for next image processing operation. The three bits of the START_PROC(2:0) signal are identical, but are duplicated due to fan-out restrictions, as 23 of the MAP chips use this signal.

**ROWBLNK(5:0):** Informs the MAP which rows are to be blanked. Since the blanking signals select either the value of an addition or $-\infty$, and the result of the selection is not clocked into a flip-flop until the next rising clock edge, this signal is generated during the same clock cycle as its corresponding addition is being performed.

**COLBLNK(5:0):** Informs the MAP which columns are to be blanked. Since the blanking signals select either the value of an addition or $-\infty$, and the result of the selection is not clocked into a flip-flop until the next rising clock edge, this signal is generated during the same clock cycle as its corresponding addition is being performed.

**INIT_ALU2:** Instructs ALU2 to load its next instruction upon the next rising clock edge. This initializes ALU2 at the same time that the first valid image pixel is its operand.

**STOP_ALU2:** Informs ALU2 that its operand upon the next rising clock edge will no longer be a valid pixel for the current processing operation.

The Controller was designed such that it properly issues the control signals for the rest of the system, according to the timing requirements described in section 5-1. The schematic for the Controller is shown in figure 5-2. The description of how all the Controller’s outputs are generated will roughly follow the order they are generated in, as shown in the timing diagram in figure 5-1.
Figure 5-2 Controller Schematic
Upon power-up of the system (or if the board’s reset button is pushed at any time), a CLRn signal is issued, which will clear all the Controller’s flip-flops. The four buffers, labeled CLRDEL0 through CLRDEL3, ensure that the CLRn signal is asserted for at least approximately 40 ns (each buffer has a delay of about 10 ns). This will prevent glitches in the CLRn signal to issue any false clear signals. The clear signal is duplicated five times to satisfy fan-out restrictions of the gates.

Before any processing operations may be performed, processor instruction and memory selection information must be written to the Controller by the host computer, using the REGS_Pln and REGS_MSn signals in conjunction with the REGENW signal and the SD bus. The process instruction register is described in table 5-2. Whenever the REGENW (register enable for write) and the REGS_Pln (processor instruction) are both asserted, bit 0 of the SD bus is written to the latch labeled MAXR. The output of that latch (the MAX signal) is then available to the MAP, which will latch it in when issued the START_PROC signal from the Controller. Three MAX signals are sent to the MAP to satisfy the fan-out restrictions of the gate array chip. When REGENW and the REGS_MSn (memory select) signals are both asserted, bit 1 of the SD bus is written to the latch labeled BMR1, which will contain the bus mode bit, as described in table 5-2. Memory selection information, as described in table 5-3, must also be written to the Controller before a processing operation may be performed. Whenever the REGENW and REGS_MSn (memory select) are asserted, the contents of the SD bus are written to the 8-bit latch labeled MSREG.

To start a processing operation, the host computer informs the bus interface chip to issue a start signal to the Controller in the form of a register write. When the REGENW and REGS_STARTn signals are both asserted, the output of the latch labeled ST1 will be high, with the value of the SD bus being irrelevant. That output is fed through a chain of two flip-flops, labeled ST2 and ST3, which prohibits metastability problems. The outputs of those two flip-flops are fed into the multiplexed flip-flop (ST4),
which generates the Controller's internal start signal (START_X1). The multiplexed flip-flop allows the START_X1 signal to have a duration of one clock cycle, regardless of the length of the register write cycle. The START_X1 signal consists of two identical signals to satisfy the fan-out restrictions of the gates in the gate array chip.

When the START_X1 signal is issued, the bus mode select bit is latched into the latch labeled BMR2. The bus mode, as described in section 4, selects the function of the X2 Bus (when the bus mode is 0, the X2 Bus is fed into ALU1 coincident with the X1 Bus data being fed into ALU1, and when the bus mode is 1, the X2 Bus is fed into ALU2 coincident with the MAP's output being fed into ALU2). If the bus mode is 0, the output of the latch labeled BMR2 becomes the Start_X2 signal by passing through the multiplexor labeled STX2SEL, which will allow the Controller to issue identical START_MEM signals to the two memories designated to control the X1 and X2 Buses. If the bus mode is 1, the Start_X2 signal is delayed such that the resulting START_MEM signal for the X2 Bus matches the timing requirements shown in figure 5-1.

The block labeled MEM_SELECT, with the inner details shown in figure 5-3, issues the WRITE_MEM and START_MEM signals to the Mem_Control chips controlling the selected memories at the appropriate times. When the Start_X1 signal is asserted, the memory instruction register value is latched into the 8-bit latch labeled DLC8. As shown in table 5-3, the 8 bits are divided into 2 bits per memory, with a binary value of 00 indicating the output of the memory is to be put on the X1 Bus, 01 indicating the output of the memory is to be put on the X2 Bus, 10 indicating that the Y bus (the output of a processing operation) is to be sent to the memory, and 11 indicating that the memory is not to be used in that processing operation. The 2-bit values for each memory are used to select the START_MEM signals for each memory controller using the multiplexors labeled MUX1 through MUX4. The multiplexors will pass through the proper memory start signal (Start_X1, Start_X2, Start_Y, or none) to each memory. For example, if the 8-bit memory select register value is 11100100, Mem_Control 0 will
be issued the Start_X1 signal. Mem_Control 1 will be issued the Start_X2 signal, Mem_Control 2 will be issued the Start_Y signal, and no signal will be issued to Mem_Control 3. The MEM_SELECT block also issues the WRITE_MEM signal to the Mem_Control chip controlling the memory selected to store the output of a processing operation. Whenever the 2-bit value for a memory is 10 (the same memory that receives the START_Y signal), the WRITE_MEM signal is asserted, using the AND gates, until the Start_X1 signal for the next processing operation is issued. The WRITE_MEM signal informs the selected Mem_Control chip to issue write enable signals to its memory at the appropriate times, as explained in section 5.3.
Figure 5-3 Mem_Select Schematic
The block labeled BUS_SELECT, with the inner details shown in figure 5-4, controls the buffers between the memories and the X1 and X2 Buses, allowing each bus to only be written to by the selected memory. As can be seen in the system schematic in figure 4-2 and the block diagram of the memories in figure 4-2, all four memories have two buffers between their outputs and the buses (one for the X1 Bus and one for the X2 Bus). The Controller sends the X1_BUS_SELn and X2_BUS_SELn signals, generated by the BUS_SELECT block, to those buffers. When the bus mode is 0, both of the appropriate buffer enable signals must be set when the X1 data value has an address of 0,0 in time to allow ALU1 to use the values on the buses as its operand(s) by the next rising clock edge. When the bus mode is 1, the X1 bus enable signals are the same as for bus mode 0, but the X2 bus enable signals must coincide with the X2 data entering ALU2 for the duration of one image pass. The buffer enable signals do not need to be cleared unless another processing operation requires a different memory and bus configuration, in which case the next processing operation will set the bus select signals as needed. The X1_BUS_SELn and X2_BUS_SELn signals need to be set one clock cycle after the Start_X1 and Start_X2 signals, respectively. When the bus mode is 0, both the Start_X1 signal and the Start_X2 signal coincide with the START_MEM (X1) signal shown in figure 5-1, and when the bus mode is 1, the Start_X1 signal coincides with the START_MEM (X1) signal and the Start_X2 signal coincides with the START_MEM (X2) signal. In the BUS_SELECT schematic, it can be seen that the Start_X1 and Start_X2 signals set the flip-flops labeled FF1 and FF2 at the next rising clock edge, which in turn latches the inputs of the 4-bit latches labeled LAT1 and LAT2. As shown in table 5-3, each memory has a two-bit control value, with 00 indicating that the output of the memory should be sent to the X1 Bus and 01 indicating that the output of the memory should be sent to the X2 bus. Since a memory instruction for a valid processing operation can only contain exactly one control value of 00 and a maximum of one control value of 01, the output of only one of the AND gates labeled AND2 through
AND5 can be high, and the output of a maximum of one of the AND gates labeled AND6 through AND9 can be high. The outputs of all the AND gates are passed through the 4-bit latches and the following multiplexors or inverters directly to the buffer enable signals (X1_BUS_SEL[n] and X2_BUS_SEL[n]), with the correct buffers for each bus being selected.

Another function of the BUS_SELECT block is to allow the PC to read from the on-board memory using the X1 Bus. As shown in table 5-3, bits 4 through 7 of the memory instruction register must be low, which will select the B inputs of the multiplexors labeled MX1 through MX4 using the AND gates labeled AND4, AND5, and AND1. This will send bits 0 through 3 to the buffers between memories 0 through 3 and the X1 Bus, which will enable the proper buffer to allow the PC to read from the selected memory. The buffer enable signals bypass the 4-bit latch labeled LAT1 so as not to be dependent on the Start_X1 signal, which is only issued during processing operations.
As can be seen in the system timing diagram shown in figure 5-1, the INIT_ALU1 signal needs to be issued one clock cycle after the Start_X1 signal, which is accomplished by using the flip-flop labeled IA1.

After the INIT_ALU1 signal is issued, the Controller must issue the control signals for the MAP, which are the START_PROC, ROWBLNK (row blanking), and COLBLNK (column blanking) signals. However, the MAP does not need the control signals until the first pixel of the image has reached the target pixel position in the MAP, and in the case of two consecutive image processing operations with little or no delay between the images, the Controller must finish issuing control signals for the previous image while the next image is already being loaded into the MAP. To account for the delay between the Start_X1 signal (coincident with the START_MEM (X1) signal in the system timing diagram), the Start Blank block (labeled ST_BL) is utilized. Its purpose is to issue the DONE output (shown as StartBlankCounter in the system timing diagram) 1541 clock cycles after the START_MEM (X1) signal is issued. When the StartBlankCounter signal is issued, the blank counter (labeled BL_CNT in the Controller schematic), starts counting and issues the appropriate ROWBLNK and COLBLNK signals to the MAP. The 1541 clock cycle count is derived from the following: 1536 clock cycles for the first pixel to propagate through the first three FIFO chips (each with a delay of 512 cycles), two cycles to pass through ALU1, four cycles to reach the target pixel position upon entering the MAP, then subtract one cycle since the blank counter address is one cycle ahead of the address of the target pixel (as can be seen in the system timing diagram). The counting is performed by the 11-bit counter found in the Start Blank block, as shown in figure 5-5. The operation of the counter is explained in Appendix C. When the count reaches 1540 in decimal (which is 1541 clock cycles since the count starts at 0), the count_done signal at the output of the AND gate labeled U35 is set. Since the done_enable signal at the output of the flip-flop labeled U21 is set by the START pulse, the AND gate will output the DONE signal for as long as the
count_done signal is high, which is one clock cycle. The count_done signal also disables the done_enable signal using the multiplexor labeled U20 and the flip-flop labeled U21, to disallow any false DONE signals to be generated after the 11-bit counter resets back to 0 and reaches 1540 (decimal) again.

One clock cycle after the Start_Blank.Counter signal is issued, the flip-flop labeled DOA issues the START_PROC signal to the MAP, as shown in the system timing diagram. The START_MEM (X2) signal (Start_X2 in the Controller schematic) needs to be issued 14 clock cycles after the START_PROC signal, which is accomplished by the 14-cycle delay block (containing 14 flip-flops) labeled DELAY14A. The flip-flop labeled D15A issues the INIT_ALU2 signal one clock cycle thereafter, and the flip-flop labeled D16A issues the START_MEM (Y) signal (Start_Y in the Controller schematic) one cycle later, with all the timing signals meeting the system timing requirements.

The Start_Blank.Counter signal also starts the address counter in the Blank Counter block (labeled BL_CNT in the Controller schematic). The schematic is shown in figure 5-6 and explained in greater detail in Appendix C. The address count is decoded and fed into small state machines, which issue the blanking signals (as described in section 3.1). The state machines were used as opposed to straight address decoding so that the blanking signals will be available as soon as possible after the rising clock edge, since the MAP uses the blanking signals during the same clock cycle. Since the state machines take one clock cycle to issue the blanking signals, the address of the blank counter must be one value behind the address of the target pixel, as shown in the system timing diagram.

When the blank counter has completed counting the addresses for the image, it issues a DONE signal that is exactly 262,144 (5122) cycles after the blank counter start signal was issued. That signal then propagates through 16 flip-flops (labeled DOB, DELAY14B, and D15B) becoming the STOP_ALU2 signal, which is exactly 262,144 cycles after the INIT_ALU2 signal was issued, as required by ALU2.
Figure 5-5 Start_Blank Schematic
Figure 5-6 Blank Counter Schematic
Before the host computer can issue another instruction to the MIP or write new window values to the MAP, the host computer must poll the instruction load ready bit of the Controller's processor instruction register. Once the INIT_ALU2 signal is issued, any instructions (except for the process start instruction) and window writes will not affect the current processing operation. The Controller's internal start signal (Start_X1) will reset the flip-flop labeled PSR1, indicating that new instructions or window values may not be loaded yet. The flip-flop will remain reset until the INIT_ALU2 signal is issued. The register will then be indicating that new instructions and window values may be loaded, until the next start signal resets the flip-flop. The flip-flop is preset upon power-up to allow the first instruction to be written.

The host computer must also poll the start ready bit (the flip-flop labeled PSR2) of the Controller's process instruction register to determine if a new process start signal may be issued. It operates the same way as the instruction load ready register bit, but is set by the STOP_ALU2 signal. The host computer may issue another process start signal without affecting any other processing operations once the bit is set. The flip-flop is also preset upon power-up to allow the first start signal to be issued.
5.3 Memory Controllers

To provide sufficient on-board memory but not use excessive board space, it was decided that the Morphological Image Processor prototype board would support four memories, each capable of containing one image. Each memory obtains its address and control signals from its associated memory control chip, named Mem_Control. The Mem_Control chip accepts an 18-bit address from the bus interface and control signals from both the main controller and the bus interface. The control signals and address from the bus interface are used to allow the host computer to access on-board image memory, and the control signals from the master controller are used to perform image processing operations using images stored in the on-board memory.

<table>
<thead>
<tr>
<th>signal</th>
<th>I/O</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC_ADDR(17:0)</td>
<td>I</td>
<td>18-bit address from host computer</td>
</tr>
<tr>
<td>PC_CS</td>
<td>I</td>
<td>chip selection from host computer</td>
</tr>
<tr>
<td>PC_RD</td>
<td>I</td>
<td>read enable from host computer</td>
</tr>
<tr>
<td>START_MEM</td>
<td>I</td>
<td>starts internal memory address counter</td>
</tr>
<tr>
<td>WRITE_MEM</td>
<td>I</td>
<td>enables write to associated memory</td>
</tr>
<tr>
<td>CLK</td>
<td>I</td>
<td>system clock</td>
</tr>
<tr>
<td>CLK2</td>
<td>I</td>
<td>second clock used for proper write enable timing</td>
</tr>
<tr>
<td>CLRn</td>
<td>I</td>
<td>global clear upon system power-up (active low)</td>
</tr>
<tr>
<td>MEM_ADDR(17:0)</td>
<td>O</td>
<td>18-bit image memory address</td>
</tr>
<tr>
<td>WEn</td>
<td>O</td>
<td>write enable for image memory (active low)</td>
</tr>
</tbody>
</table>

Table 5-4 Mem_Control Inputs and Outputs

Table 5-4 briefly describes the inputs and outputs of each Mem_Control chip. PC_ADDR(17:0) is an 18-bit address used by the host computer to access on-board memory, which occurs when the PC_CS signal (from the bus interface) to a memory's associated Mem_Control goes high. The PC_RD signal, also from the bus interface but common to all four Mem_Control chips, determines whether the host computer will perform a memory read or memory write using the selected memory. The PC_CS and
PC_RD signals originate from the host computer as register writes to the bus interface. Both the START_MEM and WRITE_MEM signals are issued from the master controller to control the internal address counter and write enable signal of each Mem_Control chip. The START_MEM signal is a one-cycle pulse which starts the address counter on the next rising CLK edge. If the WRITE_MEM signal is also high, the memory will be written to by issuing a WEₙ signal for each pixel, otherwise it will be read from. The CLK signal is the main board clock, and the CLK2 signal is the CLK signal delayed by approximately 30 ns used for proper timing of the WEₙ (write enable) signal to the memory. It was decided to keep the memories selected at all times, therefore no memory chip select signal is being used. The CLRₙ signal is taken from the global reset signal to clear the Mem_Control chips upon board power-up.

Both the MEM_ADDR(17:0) and the WEₙ signals are sent from a Mem_Control chip to its associated memory, with the origin of those signals being either the host computer or the Mem_Control chip, depending on the status of the PC_CS signal. When the PC_CS signal is low, MEM_ADDR(17:0) and WEₙ are internally generated in the Mem_Control chip, and when PC_CS is high, MEM_ADDR(17:0) and WEₙ are equivalent to the inputs PC_ADDR(17:0) and PC_RD, respectively, which allows the memories to be accessed by the host computer. MEM_ADDR(17:0) is the address used for memory access, and WEₙ is the write enable signal which determines whether a memory read or write will take place.

Whenever the host computer needs to read from or write to the on-board memory, it issues the appropriate signals, which are described in Jeff Hanzlik’s thesis, to the bus interface. The bus interface generates signals to enable the appropriate data bus buffer (WR_ENₙ and RD_ENₙ), a chip select signal (PC_CS) for every Mem_Control chip, one read/write signal (PC_WEn) used by all four Mem_Control chips, and the address (PC_ADDR). Whenever a PC_CS signal for one of the Mem_Control chips is high, that chip allows the host computer to access its associated memory. An enabled PC_CS
signal will select the PC_ADDR and PC_WEn signals as opposed to the internally generated memory address and write enable signals. This gives the host computer complete control of the selected memory.

The other mode of operation of a Mem_Control chip (when the PC_CS signal is low) is used for image processing operations. One or two memories can be read from to provide input data, and the output is written to another memory. The Controller chip is in charge of informing each Mem_Control chip whether it will be used for reading or writing (using WRITE_MEM) and when to start the address counter (using START_MEM). X1_Bus is always used to carry data from one memory to ALU1. When a second memory is being used as an image source, data can be sent to either ALU1 or ALU2 using X2_Bus. The bus arbitration is performed by the Controller by sending the X1_BUS_SELn(3:0) and X2_BUS_SELn(3:0) enable signals, as described in section 5.2, to the buffers between the memories and the buses.

To ensure proper operation of the memories during an image processing operation, the timing conditions of the memories must be met. When a memory read is being performed, the data from the memory must reach ALU1 before the next rising CLK edge. Using a logic analyzer with a resolution of 10 ns, a valid address was measured as occurring between 20 ns and 30 ns after a rising CLK edge. According to the memory specifications (see figure 5-7 and table 5-5), the data will be valid at a maximum of 30 ns after the address is valid. The average time between a rising CLK edge and valid data on X1_Bus was measured with the logic analyzer as 51 ns. All average times were measured by averaging the results of 1000 processing operations. With the 200 ns clock period, the data is valid for almost 150 ns before the next rising CLK edge, which exceeds the 5 ns minimum setup time for the input flip-flops of ALU1. The START_MEM signals and address values in figure 5-1 show how a memory read operation relates to the timing of the rest of the system.
The timing conditions of the memories must also be met for proper operation of image writes during image processing operations. The most crucial timing for writing to memory is the proper assertion of the WEn (write enable) signal. According to the memory specifications (see figure 5-8 and table 5-5), WEn can not be asserted until at least 5 ns after the address is valid, which is an average of 20 ns to 30 ns after the rising edge of CLK. To create an accurately timed WEn signal, a delay chip is used, which generates the CLK2 signal 30 ns behind CLK. During the memory write cycle, Mem_Control will generate the WEn signal by inverting CLK2. Due to the gate and I/O delay in the Mem_Control FPGA chip, the WEn signal will be asserted at least 20 ns after the rising CLK2 edge, which ensures that WEn will not be issued until approximately 20 ns after the address is valid, meeting the memory’s minimum 5 ns address set-up time. The actual timing of the signals was measured using a logic analyzer, with the results shown in table 5-5. Figure 5-1 shows the general timing for the memory write operation, with all of the control pulses and address and data values shown in boldface.

![Figure 5-7 Memory Read Cycle](image-url)
Figure 5-8 Memory Write Cycle

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Cypress 1540PF-30</th>
<th>Measured (1000 samples)</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>READ CYCLE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( t_{RC} )</td>
<td>Read Cycle Time</td>
<td>30</td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>( t_{AA} )</td>
<td>Address to Data Valid</td>
<td></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>( t_{OHA} )</td>
<td>Data Hold from Address Change</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WRITE CYCLE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( t_{WC} )</td>
<td>Write Cycle Time</td>
<td>30</td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>( t_{AW} )</td>
<td>Address Set-Up to Write End</td>
<td>20</td>
<td>110</td>
<td>116</td>
</tr>
<tr>
<td>( t_{HA} )</td>
<td>Address Hold from Write End</td>
<td>4</td>
<td>80</td>
<td>81</td>
</tr>
<tr>
<td>( t_{SA} )</td>
<td>Address Set-Up from Write Start</td>
<td>5</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>( t_{PWE} )</td>
<td>WE Pulse Width</td>
<td>20</td>
<td>90</td>
<td>96</td>
</tr>
<tr>
<td>( t_{SD} )</td>
<td>Data Set-Up to Write End</td>
<td>20</td>
<td>100</td>
<td>109</td>
</tr>
<tr>
<td>( t_{HD} )</td>
<td>Data Hold from Write End</td>
<td>5</td>
<td>70</td>
<td>75</td>
</tr>
</tbody>
</table>

Table 5-5 Memory Switching Characteristics

The schematic for the `Mem_Control` chip is shown in figure 5-9. When the host computer needs to access the on-board memory, it sets the PC_CS signal, which will select the B inputs of the multiplexors labeled U2 and U7. This will pass the address from the PC through to the `Mem_Control`'s associated memory and also pass the PC's
memory read signal through to the WEn signal, giving complete control of the memory to the PC.

The rest of the Mem_Control's circuitry is devoted to controlling the on-board memory during processing operations, when the PC_CS signal is low. For both read and write operations, a START_MEM signal from the Controller will indicate when to start the memory address counter (labeled U1). Figure 5-10 shows the schematic of the memory address counter, and a brief explanation of how it works is also in Appendix C. The memory address will be available at the first rising CLK edge after the START_MEM signal is issued. Since the multiplexor labeled U2 is selecting the A input, the memory address is sent to the Mem_Control's associated memory.

For write operations, the multiplexors labeled U3 and U4, the flip-flop (U5), the NAND gate (U6), and the DONE output of the memory address counter (U1) serve the purpose of controlling the WEn signal going to the memory. Whenever the WRITE_MEM and START_MEM signals are both high, the WEn_ENABLE signal is enabled, which will issue the proper WEn signals after NANDed with the CLK2 signal. The WEn_ENABLE signal will retain its state until the DONE signal is issued or another START_MEM signal is issued. The normal mode of operation is for the DONE signal to be issued, which occurs two pixels before the end of the image, which will cause the last WEn signal to occur while writing the last pixel of the image. The Mem_Control chip, along with the rest of the system, was designed to allow a START_MEM signal to start an operation while another operation is in progress, which will abort the operation in progress. However, the software was designed to make sure an operation has been completed before starting another one.
Figure 5-9 Mem_Control Schematic
Figure 5-10 Mem.Counter Schematic
6. Software Design

Since the Morphological Image Processor (MIP) was designed to be operated by a PC, it was necessary to write software to control the MIP from a PC. The software is responsible for providing a friendly user interface, which allows the user to utilize all the MIP's functions. The software must also send the appropriate control signals from the PC to MIP, which are determined based on the user's desired operation(s).

The software is written primarily in C, with a few functions written in assembly language. Since "standard" C has inadequate screen display functions for this particular application, Turbo C functions not included in the standard definition of C were used. If any modifications or additions to the user interface program are made, the new program must be compiled on a Turbo C (or compatible) compiler, or proper modifications to the program must be made.

The listing of the user interface program is in Appendix B. Every function in the program has function description comments preceding the function. The function descriptions include the function name, the purpose of the function, a list of the inputs to the function and their purposes, a list of the output of the function and its purpose, and a list of global variables that may be changed by the function. To describe what is occurring throughout the program, comments have been added to any statement whose purpose is not obvious. Also, all variable names (except for indexing variables such as x, y, i, or j), have been given descriptive names, as have all the functions. Due to the software essentially being self-documenting, an extensive description of the software will not be offered in this section, but rather within the code itself.

To provide for organized files and to increase the speed of compilation, the program has been broken up into multiple C and assembly language files. All C files have an extension of ".C" and an extension of ".ASM" has been used for all assembly language files. The following files make up the user interface program:
**DEFINE.C:** This file includes all predefined values and constants, which are commonly defined in C using the "#define" statement. Some of the defined values may be changed by the user, including the maximum number of instructions and masks the user can load, path names to default directories (for storing instruction files, mask files, images, etc.), and default file extensions. There are some other defined values that should not be changed by the user, such as functions that have been renamed to facilitate easier use of the functions, the constant definition for negative infinity, and other definitions such as TRUE or FALSE.

**GLOBAL.C:** This file includes all global variable definitions. Global variables have been used when they need to be accessed by multiple functions, many contained in different files, reducing the number of function calls with an excessive number of arguments. They have also been used to simplify data transfer between the C and assembly language functions.

**IMGXFER.C:** This file contains all the functions used to transfer images, and some of their associated functions. Images can be contained on the MIP board itself, in the PC's extended memory, in a frame grabber, or on a disk, and may be transferred to and from any of those storage areas.

**INSTR.C:** This file contains all the functions that allow the user to list, modify, load, and save instruction files. The MIP has its own set of image processing instructions, which are defined in a text file. A user can create a group of these instructions to carry out certain image processing tasks, and then can save them for later use.
MASK.C: This file contains all the functions that allow the user to list, modify, load, and save masks. Every morphological image processing function needs a mask to operate on the image, and these functions allow a user to manipulate the masks.

PROTO.C: This file contains all function prototype statements, which define the name of the function, and typecasts the inputs and outputs of the functions. Prototyping is necessary for most C functions.

RUN.C: This file contains all the functions for running the image processing instructions. The processing instruction commands are parsed in this file, and the resulting information is used for the register writes to the MIP.

TYPEDEF.C: This file contains all the type definitions of the structures and enumerations used throughout the user interface software.

USERINT.C: This file contains the function "main()" (required by C to be the main function from which all the other functions are called) and a few support functions. Included in the main function is the top level user interface menu. All the lower level user interface menus are included in more appropriate files (i.e., the mask menu is in MASK.C).

UTIL.C: This file contains all the utility functions, such as input and conversion routines.

IMAGES.ASM: This file contains the plot pixel and get pixel functions, which plots pixels to and gets pixels from the frame grabber.
MEMXFER.ASM: This file contains the functions to transfer memory between conventional memory and extended memory. Since the extended memory cannot be reached using C function calls, an MS-DOS function call must be used. Extended memory is used for extra memory to store images, and is also where the MIP's memory (for image reads and writes and mask writes) resides.

A majority of the software deals with user interfacing, such as menus and prompts. Since the user interface program itself is explained in the software user's manual (Appendix A), the code is self-documenting, and the writing of such software is beyond the scope of this thesis, it will not be explained in this section. However, the format of the image processing instructions, the parsing of the instructions, and writing the register values to the board will be discussed.

The list of predefined instructions is shown in table 6-1. All the instructions have a unique name; input, output, mask, and constant values contained within parentheses; ALU1 and ALU2 operations; and whether the MAP should perform a maximum or minimum operation. The commands exist in a file read in by the user interface program. All basic commands, along with a few more complex commands, have been included in the file. A user has the option of adding more instructions as long as they adhere to the proper format, which is described in the software user's manual. All the processing of commands occurs within the RUN.C program.
<table>
<thead>
<tr>
<th>predefined instruction</th>
<th>ALU1 operation</th>
<th>MAP operation (max)</th>
<th>ALU2 operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy(Q=I)</td>
<td>20</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>dilate(Q=I;W)</td>
<td>20</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>erode(Q=I;W)</td>
<td>20</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>min(Q=I,J)</td>
<td>00</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>max(Q=I,J)</td>
<td>10</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>add(Q=I,J)</td>
<td>A0</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>sub(Q=I,J)</td>
<td>80</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>sub2(Q=I,J)</td>
<td>C0</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>add_const(Q=I;k)</td>
<td>B0</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>sub_const(Q=I;k)</td>
<td>90</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>sub2_const(Q=I;k)</td>
<td>D0</td>
<td>x</td>
<td>30</td>
</tr>
<tr>
<td>min_dilate(Q=I,J,W)</td>
<td>00</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>max_dilate(Q=I,J,W)</td>
<td>10</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>add_dilate(Q=I,J,W)</td>
<td>A0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>sub_dilate(Q=I,J,W)</td>
<td>80</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>sub2_dilate(Q=I,J,W)</td>
<td>C0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>add_const_dilate(Q=I;k;W)</td>
<td>B0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>sub_const_dilate(Q=I;k;W)</td>
<td>90</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>sub2_const_dilate(Q=I;k;W)</td>
<td>D0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>min_erode(Q=I,J,W)</td>
<td>00</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>max_erode(Q=I,J,W)</td>
<td>10</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>add_erode(Q=I,J,W)</td>
<td>A0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>sub_erode(Q=I,J,W)</td>
<td>80</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>sub2_erode(Q=I,J,W)</td>
<td>C0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>add_const_erode(Q=I;k;W)</td>
<td>B0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>sub_const_erode(Q=I;k;W)</td>
<td>90</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>sub2_const_erode(Q=I;k;W)</td>
<td>D0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>dilate_min(Q=I;W,J)</td>
<td>20</td>
<td>1</td>
<td>08</td>
</tr>
<tr>
<td>dilate_max(Q=I;W,J)</td>
<td>20</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>dilate_add(Q=I;W,J)</td>
<td>20</td>
<td>1</td>
<td>A8</td>
</tr>
<tr>
<td>dilate_sub(Q=I;W,J)</td>
<td>20</td>
<td>1</td>
<td>88</td>
</tr>
<tr>
<td>dilate_sub2(Q=I;W,J)</td>
<td>20</td>
<td>1</td>
<td>C8</td>
</tr>
<tr>
<td>dilate_add_const(Q=I;W;k)</td>
<td>20</td>
<td>1</td>
<td>B0</td>
</tr>
<tr>
<td>dilate_sub_const(Q=I;W;k)</td>
<td>20</td>
<td>1</td>
<td>90</td>
</tr>
<tr>
<td>dilate_sub2_const(Q=I;W;k)</td>
<td>20</td>
<td>1</td>
<td>D0</td>
</tr>
<tr>
<td>erode_min(Q=I;W,J)</td>
<td>20</td>
<td>0</td>
<td>08</td>
</tr>
<tr>
<td>erode_max(Q=I;W,J)</td>
<td>20</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>erode_add(Q=I;W,J)</td>
<td>20</td>
<td>0</td>
<td>A8</td>
</tr>
<tr>
<td>erode_sub(Q=I;W,J)</td>
<td>20</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>erode_sub2(Q=I;W,J)</td>
<td>20</td>
<td>0</td>
<td>C8</td>
</tr>
<tr>
<td>erode_add_const(Q=I;W;k)</td>
<td>20</td>
<td>0</td>
<td>B0</td>
</tr>
<tr>
<td>erode_sub_const(Q=I;W;k)</td>
<td>20</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>erode_sub2_const(Q=I;W;k)</td>
<td>20</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>find_min(Q=I)</td>
<td>x</td>
<td>x</td>
<td>44</td>
</tr>
<tr>
<td>find_max(Q=I)</td>
<td>x</td>
<td>x</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 6-1  Predefined Image Processing Instructions
A user of the MIP executes image processing instructions by issuing one or more of the predefined instructions. If a user attempts to execute a non-existent instruction or includes incorrect fields within the parentheses, the user interface program will issue an error message. The software checks for numerous errors that the user may attempt, such as invalid images, sending the output to one of the input images, or invalid mask values. The predefined instruction file includes the required format for the fields within the parentheses, however, those fields are optional within the predefined instruction file.

To successfully parse an instruction issued by a user, the software searches the predefined instruction list until a match is found. When a match is found, all values between the delimiters are used to determine the proper instruction to issue to the MIP. The delimiters used are a left parenthesis, equal sign, comma, semicolon, and colon. Any value found after a left parenthesis is determined to be the output memory of an image processing operation. Valid inputs for this field are "m0", "m1", "m2", or "m3", indicating which on-board memory to send the output to, with the "m" being optional. Any value after the equal sign is the first input memory (the one sent to the X1 bus), with valid values being the same as the output memory. The same inputs are also valid for any value found after a comma, which will be the second input image (sent to the X2 bus). Any value found after a semicolon specifies the mask to be used by the MIP, with valid values for that field being "wX" or "X", where X is the window number as loaded in the user interface program. File names for masks are also permissible if surrounded by double quotations. For erosions, the window must be reflected through its origin (rotating the window 180° and negating all the values). The software performs the reflection just before writing the window to the board, therefore it does not need to be done manually by the user.

The last delimiter is the colon, which specifies that the following value will be an ALU constant. If only one constant is specified, the same value will be sent to both ALUs and will be ignored by the ALU not performing an operation with a constant. If both
operations are performing operations involving constants, the value after the first colon will be sent to ALU1 and the value after the second colon will be sent to ALU2.

The values that are derived from the instructions are translated to the proper register values, as described by the comments within the RUN.C program. For example, the input and output memory values are used to determine what value will be issued to the memory select register in the Controller chip. The predefined instruction file also includes the ALU1 and ALU2 operations (in hexadecimal form) and whether the MAP should perform a maximum or minimum operation (if the value is 1, perform a maximum operation, otherwise minimum). If the value of any of those fields is “x”, a “don’t care” situation is indicated and the value defaults to 0. Those commands need no processing before being sent to the appropriate registers, with the exception of the Controller's process instruction register, which requires the bus mode and MAP instruction. The MAP instruction is taken directly from the "max" signal specified in the predefined instruction file, and the bus mode is derived from the ALU2 operation. If the operation specifies to use the third input of ALU2 (which is connected to the X2 bus), then the bus mode will be 1, which delays the second image to coincide with the MAP's output.

After writing all the appropriate values to the registers and sending the mask values to the MIP, a start instruction is issued, also using a register write. The user interface program will then poll the process status register to determine if another instruction may be issued, and if there are more instructions, the software will recalculate the new register values and issue another start instruction.
7. Results

Several examples will be shown to demonstrate the functionality of the Morphological Image Processor. All examples will use the following structuring element:

\[
\begin{array}{cccccccc}
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-\infty & -\infty & 0 & 0 & 0 & -\infty & -\infty \\
-\infty & -\infty & 0 & 0 & 0 & -\infty & -\infty \\
-\infty & -\infty & 0 & 0 & 0 & -\infty & -\infty \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
\end{array}
\]

The first example will demonstrate an edge detection operation. The first step of an edge detection operation is to perform a dilation, which will expand the image. The original operation is then subtracted from the dilated image, which will leave only the expanded portion of the image. The remaining portion is the result of the edge detection operation. The original image is shown in figure 7-1, the dilation of the original image is shown in figure 7-2, and the result of the edge detection is shown in figure 7-3.
Figure 7-1  Original Sample Image

Figure 7-2  Dilation of the Sample Image
The next example demonstrates removal of "pepper" noise, which in this case is random groups of pixels with a value of zero (black). To remove such noise, a closing operation is used, which is a dilation followed by an erosion of the dilation. The dilation removes all noise that is the same size of or smaller than the structuring element, but also expands the image in the process. The erosion will then restore the original size of the image. The original image is shown in figure 7-1, the image with the "pepper" noise added is shown in figure 7-4, the dilation of that image is shown in figure 7-5, and the result (an erosion of the dilation) is shown in figure 7-6.
Figure 7-4 Sample Image with “Pepper” Noise

Figure 7-5 Dilation of the “Pepper” Noise Sample Image
Figure 7-6 Example of “Pepper” Noise Removal

The next example demonstrates removal of “salt” noise, which in this case is random groups of pixels with a value of 255 (white). To remove such noise, an opening operation is used, which is an erosion followed by a dilation of the erosion. The erosion removes all noise that is the same size of or smaller than the structuring element, but also reduces the size of the image in the process. The dilation will then restore the original size of the image. The original image is shown in figure 7-1, the image with the “salt” noise added is shown in figure 7-7, the erosion of that image is shown in figure 7-8, and the result (a dilation of the erosion) is shown in figure 7-9.
Figure 7-7 Sample Image with "Salt" Noise

Figure 7-8 Erosion of the "Salt" Noise Sample Image
Figure 7-9 Example of "Salt" Noise Removal
8. Conclusions

This thesis has shown that it is possible to design and build a functional prototype for a real-time Morphological Image Processor (MIP). While the prototype is currently incapable of performing real-time image processing operations (typically defined as 30 to 60 frames per second), replacing chips with faster equivalents will allow the MIP to operate at real-time rates. The design of the system does not need to be modified to increase the processing speed. Also, pipelining is possible by connecting two or more identical image processors to one another, with some slight design modifications. The Controller chip needs another output and another input for pipelined start and end signals. The pipelined start signal will feed directly to the Start_X1 signal in the Controller chip after being ORed with the output of the existing start signal circuitry. The pipelined end signal, which will feed directly to the pipelined start input of the next processor in the chain, can be taken directly from the signal in the Controller chip labeled PL_START_NEXT.

The Morphological Array Processor (MAP) and all the control circuitry is currently operating at a clock rate of 1.25 MHz. There is a 5 MHz clock on the board that is used to control the FIFOs, and that clock is divided by four to provide the rest of the system with a 1.25 MHz clock signal. If the MIP were constantly processing images with no delay between each image (which also allows the latency time of each image processing operation after the initial image to be ignored), the MIP would be capable of processing 4.77 images per second. Using a function generator in the place of the clock chip, a clock rate of nearly 20 MHz (with the MAP operating at 5 MHz) has been successfully used. If the MIP were constantly processing images with no delay between each image, the MIP would be capable of processing 19.07 images per second. For a VLSI implementation (or any technology faster than the gate arrays used), a clock rate of 7.86432 MHz (31.45728 MHz for the FIFOs) is required to allow the MIP to process
images at a real-time rate of 30 images per second. The FIFOs currently on the board have a maximum operating speed of 50 MHz, and it should not be difficult (using fast enough chips) to design the rest of the system to operate at 7.86432 MHz.

The Morphological Image Processor architecture has also been designed such that changes to the size of an image or mask are possible. To increase the size of an image, the only changes that need to be made are to the FIFOs that act as the line delays (each FIFO or shift register must be able to store a complete row of the image), and all the counters need to be modified. For example, if a 1024 x 1024 image is to be used instead of a 512 x 512 image, 1024 values need to be stored in each FIFO, all 18-bit counters need to be changed to 20-bit counters, the Start_Blank block in the Controller chip needs to count to a different value, and the blanking control circuitry in the Blank Counter block of the Controller chip needs to be slightly modified. No architectural changes to the array processor are required. However, since a 1024 x 1024 image is four times as large as a 512 x 512 image, the time required for each processing operation will increase by a factor of four, unless the clock speed is increased by a factor of four.

To increase the size of a mask, a larger processing array needs to be designed, more line delays are needed, and some of the timing delays in the Controller chip need to be changed to accommodate the larger mask. For example, if a 9 x 9 mask is to be used instead of a 7 x 7 mask, a 9 x 9 adder array is required along with a comparison tree to compare all 81 resulting values. Also, one extra FIFO needs to be added, along with appropriate changes to the Controller chip to account for the extra FIFO.

Another goal of this thesis was to examine different architectures for implementing the image processor. The two architectures examined were discussed in section 3, with the comparisons between the architectures discussed in section 3.3. As noted in that section, the architecture used was determined to be the most suitable for the implementation of the MIP. While there was the disadvantage of requiring six FIFOs, the
advantages (such as simpler data paths and standard binary counters) outweighed the disadvantages.

For future upgrades of the MIP, it is suggested to change the use of negative infinities. When performing erosions, the mask must be rotated and negated. When negated, a negative infinity becomes 255 (the maximum positive value), with the desired result being that the 255 added to any image value will be greater than or equal to any of the other additions. Since the erosion operation selects the minimum of all the added values, 255 added to any positive image value can never be selected unless all values are 255. This will produce the desired result in all cases of images with all positive pixel values, however, this is not guaranteed to work with images containing negative pixel values. The author of this thesis suggests that the negation of a negative infinity should be mathematically equivalent to a positive infinity, possibly using the term "invalid pixel" to refer to both negative and positive infinities. An invalid pixel would still be represented by -256 in two's complement notation, but would not be negated if performing an erosion. The comparators in the array processor would then be modified to not select the result of any addition that had an invalid pixel as a mask value, for both erosions and dilations (unless all mask values are invalid).

The system was so large that the CAD system could not handle a gate-level simulation of the complete system. Behavioral Language Models (BLMs) were then used to model each building block of the system, and the complete system was then simulated using the BLMs. Each BLM was also simulated individually and the results were compared to the actual simulation results of the BLM's corresponding building block (which the CAD system was able to simulate). It may have been more desirable to use FPGAs that do not permanently program a chip. Such FPGAs commonly use a temporary connection to a computer to download the desired functions of the FPGA during testing, then use a PROM connected to the FPGA for the permanent version. Design modifications could then have been made without programming a new chip. Also, since
the complete simulations consumed large amounts of time (several hours for one image processing operation, using the BLMs), partial simulations used to determine chip layouts on the printed circuit board could have been performed. A logic analyzer could then have been used to monitor all the desired signals, then completion of lower level design work (such as slight modifications of timing signals) could have been performed before programming the FPGAs.

While no design for testability features were added to the design, there were many instances when design for testability would have been quite useful. Since most chips were pushing the limit of both pin and gate count, adding design for testability features would have increased the chip count of the MIP, however, debug time should have been reduced.

Many problems were encountered while debugging the board. Some of the problems could have been avoided by specifically designing the board to avoid such problems, such as the design for testability issues. Other problems, such as non-operational FIFOs, would have been more difficult to avoid. The non-operational FIFOs were replaced with FIFOs from another company. Also, the PLCC sockets for the FPGAs caused many problems, such as poor connections between the sockets and the FPGAs. The poor connections were also made worse by removing the FPGAs from their sockets. The connections were improved by bending the socket pins toward the pins of the FPGA, but only after extensive debugging to find the faulty connections. One way to avoid that problem would have been to use a more reliable method of connecting the chips, most likely at the expense of increased cost and decreased ease of placing and removing chips.

This thesis was an excellent learning opportunity. It encompassed all aspects of a group effort in designing, constructing, and debugging the hardware, and offered experience in writing the software interface. While the CAD system had its shortcomings, it proved as a valuable method of verifying that the hardware was properly designed before laying out the printed circuit board.
Appendix A. Software User’s Manual

Introduction:

The Morphological Image Processor (MIP) board has been designed to run from an IBM-compatible PC. A frame grabber and monitor connected to the PC allow images (before and after processing) to be viewed on the monitor. A special set of instructions, specifically designed for use with the MIP, allow a user to run a wide variety of image processing operations. A program, called “MIP.EXE,” has been written to allow a user to have complete control of the MIP by using the PC. This user’s manual describes the program and explains how to use it.

System Requirements:

- IBM-compatible PC with at least 64K available RAM and one open board slot
- Morphological Image Processing board and 5V power supply
- No more than 15M extended memory
- The PC’s extended memory may not be reserved for other purposes
- Frame grabber and monitor for viewing images
- Turbo C or Turbo C++ compiler if software modifications are desired

Installation:

To install the MIP's user interface software, first create a directory on the hard drive for the program. It is recommended to use a directory called "MIP" created from the root directory, so directory default values will not need to be changed. All file names and directories in this user’s guide will be referred to by their default and/or recommended names. Place the MIP software disk into the disk drive, and from the newly created
directory, type "B:INSTALL" if the installation disk is in the B drive, otherwise specify
the proper drive. The following files will be placed in the MIP directory:

**MIP.EXE**: The executable file for the user interface software.

**MIP.PRI**: The project file for the software used by a Turbo C++ compiler, which should
be used as the project file if modifying the program.

**IMAGES.OBJ**: The object file created from assembling the IMAGES.ASM assembly
language file. Used only when modifying the program, and if IMAGES.ASM is modified,
a new IMAGES.OBJ should be created.

**MEMXFER.OBJ**: The object file created from assembling the MEMXFER.ASM
assembly language file. Used only when modifying the program, and if MEMXFER.ASM
is modified, a new IMAGES.OBJ should be created.

The following directories will be created from within the MIP directory:

**INSTR**: This directory contains the instruction files, with each file containing one or more
of the pre-defined image processing instructions.

**WINDOWS**: This directory contains the window files, with each file containing the 49
values contained in a structuring element.

**STARTUPS**: This directory contains the default startup file (STARTUP.DEF), which
loads an example instruction file and window. More startup files may be placed in this
directory, with the files being loaded when specified as a command line argument when
invoking the MIP EXE file. The first line of the file specifies an instruction file, and any lines thereafter specify a window number (in decimal), followed by the window file (on the same line, with a space between the number and the file).

**CFILES:** This directory contains the C files that are part of the user interface program.

**ASMFILES:** This directory contains the assembly language files that are part of the user interface program.

**INCLUDES:** This directory contains the DEF2600.INC file, which is necessary to operate the frame grabber.

**DEFINES:** This directory contains the system configuration files. The following files are contained in this directory:

**CONFIG.DEF:** Contains the register and board memory base address definitions, in hexadecimal notation. The first number has valid values of 300, 310, 320, or 330, with the default being 300 when the board's switches 1 and 2 are set to the "on" position. The second number, on the next line, specifies the board's memory base address. The default value is 800000 (8 Meg), which is acceptable for any PC with 8 Megs of memory. The board's memory may not be mapped into any existing memory, so that value will need to be raised if the PC has more than 8 Megs of memory. Any value up to F80000, in half megabyte increments (80000), is valid. Both of those values may be changed if done properly.

**DEFAULT.DEF:** Specifies the default locations of certain directories, which will be used in the user interface program. The first line specifies the default location
of the directory for storing image processing instruction files, which is recommended to be the "\MIP\INSTR\" directory. The second line specifies the default directory for storing windows (structuring elements), which is recommended to be the "\MIP\WINDOWS\" directory. The third line specifies the default directory where the images are stored, which is recommended to be the "\IMAGES\" directory. The last line specifies the default location of the startup directory for storing startup files (which loads both instruction files and window files when starting the program), which is recommended to be the "\MIP\STARTUP\" directory. Any of these values may be changed to other directories if the original format is maintained.

**INSTRUCT.DEF**: Lists all the pre-defined image processing instructions, along with the format for using each instruction and codes used by the user interface program to issue instructions to the MIP. More instructions may be added if they adhere to the format of the existing pre-defined instructions.

**FRGRAB.DEF**: Defines the number of image buffers in the frame grabber, and the top left pixel locations of each of those buffers (all numbers are decimal). The first number in the file specifies the number of frame grabber buffers (each image is 512 x 512 pixels), and each line thereafter specifies the top left pixel value of each image, one for each buffer, and numbered in order they appear in the file, starting with 1. The default number of buffers is 8, with the top left pixel locations being (0,0), (512,0), (0,512), (512,512), (1024,0), (1536,0), (1024,512), (1536,0), (1024, 512), and (1536,512), in that order. The number of buffers and/or the locations may be changed if the original format is retained.
**Running the program:**

To run the MIP user interface program, type "MIP" from the MIP directory. This will start the program using the startup file MIP\STARTUPS\STARTUP.DEF, and will load the instructions and windows specified within the file. To specify another startup file contained within the MIP\STARTUP directory, type "MIP <startup_file>" where <startup_file> is the name of a startup file, without the .DEF extension. To start the program without any preloaded instructions or windows, use any invalid or non-existent file name as the argument (such as "MIP -").

**Layout of the screen:**

The screen is divided into three sections: the bottom of the screen is devoted to displaying instructions and the name of the currently loaded instruction file. The top right of the screen normally displays a 7×7 window, but is also used to display the pixel values of a section of the frame grabber, when specified to do so. The rest of the screen is devoted to menus and user inputs. Figure A-1 shows an example of the user interface's screen.
### Conventions of the program:

- All single-character multiple-choice options are presented in parentheses.
- All default values are presented in brackets, with <Enter> selecting the default value.
- The backspace and <Delete> keys can be used to erase characters during string inputs.
- The <Esc> key may be used to abort any input and return to the previous menu.
- All values must be entered in decimal notation.
- All hexadecimal values are viewed in nine-bit two's complement notation.
- Some menu options may not appear on the screen when they are disabled.
- All structuring elements used for erosions are rotated 180° degrees and negated by the software, therefore it is not necessary to perform manual rotation and negation.
Storing images:

The MIP's user interface program allows a user to store images before and after processing. Images may be stored in the following four ways:

1. In any of the MIP's four on-board memories, numbered 0 through 3.
2. In any of the frame grabber's image buffers. The number of buffers available depends on the size of the frame grabber, and are numbered from 0 to one less than the number of buffers available.
3. In a file in the .IMG format.
4. In any of the PC's extended memory buffers. The number of buffers available depends on the amount of extended memory (each image buffer needs half of a megabyte of extended memory, and are numbered from 0 to one less than the number of buffers available).

The main menu:

The main menu is the top level menu of the MIP's user interface program, and is the first menu displayed when starting the program. An example of the whole screen with the main menu displayed is shown in figure A-1. Due to the large number of instructions available from the main menu, it is broken down into two menus, with the "(o) other operations" option used to switch between the two. An example of the menu displaying the other operations is shown in figure A-2. Since the window and instruction areas do not change, only the menu itself is shown, as will be the case with the remaining menus. The main menu options are:
**run operations**: Go to the run operations menu, which allows the user to run pre-defined image processing instructions.

**← or →**: Views the previous or next window, respectively, in the window viewport.

**instruction operations**: Go to the instruction operations menu, which allows the user to enter pre-defined image processing instructions and load and save instruction files.

**↑ or ↓**: When there are more instructions than can fit on the screen, the viewed instructions are scrolled up or down (respectively).

**window operations**: Go to the window operations menu, which allows the user to enter window values and load and save windows.

**transfer image**: Go to the image transfer operations menu, which allows the user to copy images to and from disk, the frame grabber, the PC’s extended memory, and/or the MIP’s memory.

**new startup file**: Prompts the user to enter a new startup file, which will load a new instruction file and a new window or windows.

**other operations**: Switches between the two sub-menus of the main menu.

**change base**: Toggles the current base between decimal and hexadecimal notation.

**clear frame grabber**: Issues a prompt to select one or all of the frame grabbers buffer, then clears the selected frame grabber buffer(s).
(v) view pixel values of image: Prompts for a frame grabber image buffer to view, then displays a portion of the selected buffer on the screen. The arrow keys may then be used to move the viewport one pixel at a time, or pressing <Ctrl> in conjunction with the arrow keys to move the viewport 10 pixels at a time. Also, “n” or “p” will view the next frame or previous frame, respectively. The pixel locations are displayed in green around the viewed pixels, with the pixel locations being relative to the top left corner of the complete frame grabber, regardless of which buffer is being viewed.

(g) get pixel value: Prompts for a pixel location to read its value from the frame grabber, with the location being relative to the top left corner of the complete frame grabber.

(p) plot pixel: Prompts for a pixel location and value to plot to the frame grabber, with the location being relative to the top left corner of the complete frame grabber. The previous pixel value is also displayed, and the user is prompted whether to replace the pixel or not.

(d) draw wedge over frame grabber images: Prompts the user whether to draw a wedge or not, and draws the wedge if instructed to do so. The wedge is a group of vertical lines, where each line has a gray scale value one higher than the previous line. The wedge starts with a value of zero on the left side and wraps back to zero after the maximum value is reached. The whole frame grabber will be cleared when issuing this instruction. To draw a wedge over only one of the frame grabber, use the image transfer operation.

(q) quit: Quits the MEP user interface program after being prompted to quit or not.
The run operations menu:

The run operations menu is used to run pre-defined image processing instructions. An example of the menu is shown in figure A-3. The options available are:

(r) run: Run all of the instructions currently loaded.

(o) run one instruction: Run only the instruction specified, after being prompted for which instruction to run.

(s) step through instructions: Run all of the instructions, but with a “press any key” prompt before each instruction.

(p) partial run: Runs only the instructions specified for the partial run, which are selected using the (f) and (l) options.

(f) select first instruction (partial run): Selects the first instruction to be run when issuing the partial run command.

(l) select last instruction (partial run): Selects the last instruction to be run when issuing the partial run command.

(q) quit run instructions: Returns to the previous menu.
The instruction operations menu:

The instruction operations menu is used to manipulate the currently loaded predefined image processing instructions, and to load and save instruction files. Instruction files consist of one of more pre-defined image processing instructions, as shown in figure A-1. An example of the instruction operations menu is shown in figure A-4. The options available are:

(l) load instruction file: Loads a new instruction file, first prompting for the directory to look in, then prompting for the file name.

(s) save instruction file: Save the current instructions to a file, first prompting for the directory to save the file in, then prompting for the file name. If the file exists, “a” aborts the save and “r” replaces the existing file.

(i) insert instruction: Inserts a new instruction into the list of current instructions, first prompting for the position of the new instruction, then prompting for the new instruction.

(m) modify instruction: Prompts for which instruction to replace, then replaces that instruction with the newly entered instruction.

(d) delete instruction: Prompts for which instruction to delete (“0” may be entered to delete all instructions), then deletes the specified instruction(s).

(r) run: Go to the run operations menu, which allows the user to run the instructions.
(q) quit instruction operations: Returns to the main menu.

The window operations menu:

The window operations menu is used to manipulate windows (structuring elements), and to load and save windows. The windows are numbered from 1 to the maximum number, which defaults to 50. Any number of windows, not necessarily in order, may be loaded at any given time. A unity window is always defined as window #0, though it does not appear in prompts. An example of the window operations menu is shown in figure A-5. The available options are:

(b) change base: Toggles the current base between decimal and hexadecimal notation.

(l) load window: Loads a new window, first prompting for the directory to look in, then prompting for the file name.

(s) save window: Save the current window to a file, first prompting for the directory to save the file in, then prompting for the file name. If the file exists, “a” aborts the save and “r” replaces the existing file.

(c) copy window: Copies one window to another window, prompting for source and destination windows.

Figure A-5 Window Operations Menu

Window operations:
(b) change base {current base: decimal}
(l) load window  (s) save window
(c) copy window
(m) modify current window
(d) delete current window
(v) view different window
(q) quit window operations
Enter your choice:
(m) modify current window: Goes to the modify window menu, which allows the user to modify any or all of the window values.

(d) delete current window: Deletes the current window after a prompt.

(v) view different window: Views another window after being prompted for which window to view.

(q) quit window operations: Returns to the main menu.

The modify window menu:

The modify window menu is accessible from the window operations menu, and allows one or more window values to be changed. An example of the menu is shown in figure A-6. The available options are:

(c) change one value: Changes one window value, using the arrow keys to select one value and highlighting the selected value. The selected value may be changed by using “+” or “−” to increment or decrement the value or by pressing <Enter> to enter a new value.

(a) add constant: Adds a constant to every window value, after being prompted for the constant.

(s) subtract constant: Subtracts a constant from every window value, after being prompted for the constant.
(m) multiply by constant: multiplies every window value by a constant, after being prompted for the constant.

(d) divide by constant: Divides every window value by a constant, after being prompted for the constant.

(t) transpose quadrant: Prompts for a quadrant to transpose, and copies the selected quadrant to the other three quadrants after performing the necessary rotations of the quadrant.

(q) quit modify window: Returns to the window operations menu.

The image transfer operations menu:

The image transfer operations menu is used to transfer images from one storage location to another. The source image is entered first, followed by the destination image. An example of the menu is shown in figure A-7. The following options are presented for the source and destination images:

(0)-(3) board memories: Selects one of the MIP's on-board memories as the source or destination image.

(f) frame grabber: Selects one of the frame grabber buffers as the source or destination image, issuing a prompt to select which buffer to use.
(e) extended memories: Selects one of the PC's extended memory buffers as the source or destination image, issuing a prompt to select which buffer to use.

(d) disk: Selects a file on the PC’s disk drive as the source or destination image, issuing a prompt to enter the file to use. When writing to a file and the file already exists, a prompt will be issued to either abort the operation or write over the existing file.

(w) wedge: Selects a wedge as the source image. A wedge may not be selected as a destination image.

(q) quit image transfer operations: Returns to the main menu.

Specifying image processing instructions:

To run image processing instruction from the MIP's user interface program, the pre-defined instructions must be properly used. Table A-1 shows the pre-defined instructions, along with the format for the use of the instruction and a description of the instruction. The expressions contained within the parentheses show the required format of each instruction, with Q always designating the destination image, I always designating the source image (or the first source image if two are being used), and J always designating the second source image. The window is always designated by W, and a constant is always designated by k. It is necessary to substitute each of those values with images, windows, and constants.

Valid replacements for Q, I, and J are “m0”, “m1”, “m2”, or “m3”, where each of those values indicates one of the MIP’s on-board memories. The “m” is optional, allowing “0”, “1”, “2”, or “3” to be used. All image processing commands may only use the on-board memories, except for the copy command. The copy command also accepts
"fX", "eX", or a file name (with the complete path to the file) enclosed in double quotation marks. "fX" is used to copy a frame grabber buffer, and X denotes one of the valid frame grabber buffer numbers. "eX" is used to copy an extended memory buffer, and X denotes one of the valid extended memory buffer numbers.

Valid replacements for W are "wX" or just "X" since the "w" is optional, where X is the number of a window that has been loaded from the program. A file name (with a complete path to the file) enclosed in double quotation marks may also be used.

Valid replacements for k are "kX" or just "X" since the "k" is optional, where X is any decimal value within the range of -256 and 255, with -256 representing negative infinity.

Comments may be added at the end of any line, using a "#" character to indicate that a comment will follow. Everything up to the end of the line will be ignored.

**Example instructions:**

- `copy(m1=m0)` copies on-board memory #0 to memory #1
- `copy(2=3)` copies on-board memory #3 to memory #2
- `copy(l=f0)` copies frame grabber buffer #0 to on-board memory #1
- `copy(e1="\images\i.img")` copies image stored in file to extended memory #1
- `dilate(m3=m0;wl)` dilates on-board memory #0 using window #1, and places the result in on-board memory #3
- `erode(2=1;5)` erodes on-board memory #1 using window #5, and places the result in on-board memory #2
sub(m2=1.0) subtracts on-board memory #0 from on-board memory #1, and places the result in on-board memory #2

add_const(m0=m3:15) adds 15 to on-board memory #3, and places the result in on-board memory #0

sub_const_erode(l=0:15;1) subtracts 15 (decimal) from on-board memory #0, then dilates the result using window #1 and places the final result in on-board memory #1
<table>
<thead>
<tr>
<th>pre-defined instruction</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy(Q=I)</td>
<td>Copies one image to another</td>
</tr>
<tr>
<td>dilate(Q=I,W)</td>
<td>Performs a dilation operation</td>
</tr>
<tr>
<td>erode(Q=I,W)</td>
<td>Performs an erosion operation</td>
</tr>
<tr>
<td>min(Q=I,J)</td>
<td>Performs a pixel by pixel minimum operation on two images</td>
</tr>
<tr>
<td>max(Q=I,J)</td>
<td>Performs a pixel by pixel maximum operation on two images</td>
</tr>
<tr>
<td>add(Q=I,J)</td>
<td>Adds two images</td>
</tr>
<tr>
<td>sub(Q=I,J)</td>
<td>Subtracts two images (result is $-\infty$ when subtracting $-\infty$)</td>
</tr>
<tr>
<td>sub2(Q=I,J)</td>
<td>Subtracts two images (result is first operand when subtracting $-\infty$)</td>
</tr>
<tr>
<td>add_const(Q=I;k)</td>
<td>Adds constant to image</td>
</tr>
<tr>
<td>sub_const(Q=I;k)</td>
<td>Subtracts constant from image (see sub)</td>
</tr>
<tr>
<td>sub2_const(Q=I;k)</td>
<td>Subtracts constant from image (see sub2)</td>
</tr>
<tr>
<td>min_dilate(Q=I,J;W)</td>
<td>Performs a minimum between two images, then dilates the result</td>
</tr>
<tr>
<td>max_dilate(Q=I,J;W)</td>
<td>Performs a maximum between two images, then dilates the result</td>
</tr>
<tr>
<td>add_dilate(Q=I,J;W)</td>
<td>Adds two images, then dilates the result</td>
</tr>
<tr>
<td>sub_dilate(Q=I,J;W)</td>
<td>Subtracts two images, then dilates the result (see sub)</td>
</tr>
<tr>
<td>sub2_dilate(Q=I,J;W)</td>
<td>Subtracts two images, then dilates the result (see sub2)</td>
</tr>
<tr>
<td>add_const_dilate(Q=I;k;W)</td>
<td>Adds a constant to an image, then dilates the result</td>
</tr>
<tr>
<td>sub_const_dilate(Q=I;k;W)</td>
<td>Subtracts a constant from an image, then dilates the result (see sub)</td>
</tr>
<tr>
<td>sub2_const_dilate(Q=I;k;W)</td>
<td>Subtracts a constant from an image, then dilates the result (see sub2)</td>
</tr>
<tr>
<td>min_erode(Q=I,J,W)</td>
<td>Performs a minimum between two images, then erodes the result</td>
</tr>
<tr>
<td>max_erode(Q=I,J,W)</td>
<td>Performs a maximum between two images, then erodes the result</td>
</tr>
<tr>
<td>add_erode(Q=I,J,W)</td>
<td>Adds two images, then erodes the result</td>
</tr>
<tr>
<td>sub_erode(Q=I,J,W)</td>
<td>Subtracts two images, then erodes the result (see sub)</td>
</tr>
<tr>
<td>sub2_erode(Q=I,J,W)</td>
<td>Subtracts two images, then erodes the result (see sub2)</td>
</tr>
<tr>
<td>add_const_erode(Q=I;k,W)</td>
<td>Adds a constant to an image, then erodes the result</td>
</tr>
<tr>
<td>sub_const_erode(Q=I;k,W)</td>
<td>Subtracts a constant from an image, then erodes the result (see sub)</td>
</tr>
<tr>
<td>sub2_const_erode(Q=I;k,W)</td>
<td>Subtracts a constant from an image, then erodes the result (see sub2)</td>
</tr>
<tr>
<td>dilate_min(Q=I;W,J)</td>
<td>Dilates an image, then a minimum between the result and another image</td>
</tr>
<tr>
<td>dilate_max(Q=I;W,J)</td>
<td>Dilates an image, then a maximum between the result and another image</td>
</tr>
<tr>
<td>dilate_add(Q=I;W,J)</td>
<td>Dilates an image, then adds another image</td>
</tr>
<tr>
<td>dilate_sub(Q=I;W,J)</td>
<td>Dilates an image, then subtracts another image (see sub)</td>
</tr>
<tr>
<td>dilate_sub2(Q=I;W,J)</td>
<td>Dilates an image, then subtracts another image (see sub2)</td>
</tr>
<tr>
<td>dilate_add_const(Q=I;W;k)</td>
<td>Dilates an image, then adds a constant</td>
</tr>
<tr>
<td>dilate_sub_const(Q=I;W;k)</td>
<td>Dilates an image, then subtracts a constant (see sub)</td>
</tr>
<tr>
<td>dilate_sub2_const(Q=I;W;k)</td>
<td>Dilates an image, then subtracts a constant (see sub2)</td>
</tr>
<tr>
<td>erode_min(Q=I;W,J)</td>
<td>Erodes an image, then a minimum between the result and another image</td>
</tr>
<tr>
<td>erode_max(Q=I;W,J)</td>
<td>Erodes an image, then a maximum between the result and another image</td>
</tr>
<tr>
<td>erode_add(Q=I;W,J)</td>
<td>Erodes an image, then adds another image</td>
</tr>
<tr>
<td>erode_sub(Q=I;W,J)</td>
<td>Erodes an image, then subtracts another image (see sub)</td>
</tr>
<tr>
<td>erode_sub2(Q=I;W,J)</td>
<td>Erodes an image, then subtracts another image (see sub2)</td>
</tr>
<tr>
<td>erode_add_const(Q=I;W;k)</td>
<td>Erodes an image, then adds a constant</td>
</tr>
<tr>
<td>erode_sub_const(Q=I;W;k)</td>
<td>Erodes an image, then subtracts a constant (see sub)</td>
</tr>
<tr>
<td>erode_sub2_const(Q=I;W;k)</td>
<td>Erodes an image, then subtracts a constant (see sub2)</td>
</tr>
<tr>
<td>find_min(Q=I)</td>
<td>Finds the absolute minimum value of an image</td>
</tr>
<tr>
<td>find_max(Q=I)</td>
<td>Finds the absolute maximum value of an image</td>
</tr>
</tbody>
</table>

Table A-1 Pre-defined Image Processing Instructions
Appendix B. Software Listing

```c
#include <stdio.h>
#include <conio.h>
#include "cf1e\define.c"
#include "cf1e\typedef.c"
#include "cf1e\fileglobal.c"
#include "cf1e\proto.c"

/********************************************************************************
 * name: print_borders *
 * purpose: prints borders (using extended ASCII graphics characters) on screen
 * inputs: none
 * output: none
 * globals: none
 ********************************************************************************/

void print_borders()
{
  int i;
  normal_video(); clrscr();
  for (i=1; i<LEFT_MASK_BORDER; i++)
    putch(205); /* prints line under mask window */
  gotoxy(LEFT_MASK_BORDER, 0); putch(205); /* prints line under mask window */
  putch(205); /* prints lower left corner of mask window */
  for (i=1; i<LEFT_MASK_BORDER; i++)
    putch(205); /* prints line to left of mask window */
  gotoxy(1, TOP_INSTRUCTION_BORDER);
  for (i=1; i<LEFT_MASK_BORDER; i++)
    putch(205); /* prints line above instruction window */
}

/********************************************************************************
 * name: print_main_menu *
 * purpose: prints main menus (main menu and other menu)
 * inputs: instr_count: number of instructions
 *         other_menu: TRUE if other menu should be printed, FALSE otherwise
 * output: none
 * globals: none
 ********************************************************************************/

void print_main_menu(int instr_count, int other_menu)
{
  menu_window(); clrscr();
  if (other_menu == FALSE) /* print main menu */
    if (instr_count) cputs("(c) run operations\n\n");
    if (instr_count > 8) cprintf("(i) view prev/next window\n\n", instr_count);
    cputs("(n) new startup file\n\n");
    else /* print other menu (selected by the other operations option) */
    cprintf("(b) change base (current base: \$)\n", Base);
    cputs("(d) draw wedge over frame grabber\n");
    cputs("(x) quit\n");
    color_green(cputs("Enter your choice: "); normal_video());
}

/********************************************************************************
 * name: main *
 * purpose: Morphological Image Processor software- main routine- loads startup and configuration information and prints main menu *
 * inputs: argc: number of arguments from command line *
 *         argv: pointer to array of arguments from command line *
 * output: none
 * globals: PrefDef_Instr_Count, Instr.Current, Ext_Menu_Count, Base,
 ********************************************************************************/

void main(int argc, char **argv)
{
  int x, y, pixel, buf, ch, instr_count = 0, current_mask = 0, other_menu = FALSE, another_startup = FALSE;
```

B-1
config.

buffers

purposes

file

get_extended_memory_size(); /* gets size of extended memory inmegs. places result in Ext_Mem_Count = Ext_Mem * 2 */

if ((temp_file = fopen(DEFAULT_DIRECTORIES_FILENAME, "r")) != NULL) /* reads default directories from config. file */
{
    fscanf(temp_file, "%s", Default_Instr_Directory);
    fscanf(temp_file, "%s", Default_Window_Directory);
    fscanf(temp_file, "%s", Default_Image_Directory);
    fscanf(temp_file, "%s", Default_Startup_Directory);
    fclose(temp_file);
}

if ((temp_file = fopen(CONFIG_FILENAME, "r")) != NULL) /* reads board register location and base address from config. file */
{
    fscanf(temp_file, "%x", &Reg_Loc);
    fscanf(temp_file, "%x", &board_base_address);
    Board_Loc_High = (int)(board_base_address / 8x1000L); /* convert bytes to megabytes */
    fclose(temp_file);
}

else
    print_error("\n\nWARNING: register configuration file not found; press any key: ");

if (register_read(Osc) == 0xFF) print_error("\n\nWARNING: board not found; press any key: ");
/* if the board is connected, powered up, and working, this register should never read FF */

if ((temp_file = fopen(FR_GRAB_DEF_FILE_NAME, "r")) != NULL) /* read number of frame grabber buffers from config. file */
{
    if (fscanf(temp_file, "%li", &Fr_Grab.buffers) > 0) /* read origins of frame grabber buffers */
        fscanf(temp_file, "%li", &Fr_Grab.x_origin, &Fr_Grab.y_origin);
    fclose(temp_file);
}
else
    print_error("\n\nWARNING: frame grabber info file not found; press any key: ");

load_startup_file: /* location label for goto command- loads new startup file */

if (another_startup)
    strcpy(startup_filename, open_file(STARTUP, "r", Default_Startup_Directory, EMPTY_STRING)); /* 'n' was pressed from main menu- */
/* loads new startup file */
else if (argc < 2) strcpy(startup_filename, STARTUP_FILENAME); /* no command line arguments- loads default startup file */

else /* load specified startup file from command line argument */
{
    if (temp_file = fopen(startup_filename, "r")) != NULL) /* look for startup file */
    {
        if (fscanf(temp_file, "%s", instr_filename) > 0)
            instr_count = load_instruction_file(-1, instr_filename); /* load instruction file if specified */
        else
            while (fscanf(temp_file, "%li", &current_mask) > 0)
                load_mask(&current_mask); /* load mask(s) if specified */
        fclose(temp_file);
    }

    current_mask = 0;
    if (another_startup)
        Mask[0].valid = FALSE;
    else
        current_mask = MAX_MASKS; current_mask++ /* sets current mask for viewing purposes */

    do
    {
        print_instr_filename(instr_count); /* prints instruction filename on bottom of screen */
        if (read_instr_def_filename = read_instr_def_file()); /* reads pre-defined MIP instruction file */
        define Unity_mask(); /* defines the unity mask (mask 0) */
    }

    if (other_menu = FALSE) /* checks main menu options */
        switch (ch)
        {
            case 'I': /* instr_count = instr_menu(instr_count); break; */
                goto to instruction menu

            case UP: if (instr_count >= instr.current_count)
                scroll_up(=Instr.current); /* scrolls up one instruction, if possible */
;
            case DOWN: if (instr_count <= instr.current+8)
                scroll_down(+Instr.current); /* scrolls down one instruction, if possible */
;
            case LEFT: do if (Mask_Loaded && (~current_mask == 0)) current_mask = MAX_MASKS-1;
        }
while ((Mask[current_mask].valid != NULL) && (current_mask <= MAX_MAKES) && (!current_mask == MAX_MAKES)) current_mask = ++current_mask;

while (Mask[current_mask].valid != NULL)
{
    switch (current_mask)
    {
    case RIGHT: do if (Mask_loaded & (current_mask == MAX_MAKES)) current_mask = 1;
        view_mask(Mask[current_mask].valid, FALSE); break; /* views previous mask, if possible */
    case LEFT: do if (Mask_loaded & (current_mask == 0)) current_mask = MAX_MAKES;
        view_mask(Mask[current_mask].valid, FALSE); break; /* views next mask, if possible */
    case 'u': current_mask = Mask_menu(current_mask); break; /* goes to mask menu */
    case 'r': if (Instr_count <= instr_count) view_mask(Mask[current_mask].valid, FALSE); break; /* goes to run menu */
    case 'e': image_transfer(); break; /* goes to image transfer menu */
    case 'o': other_menu = !other_menu; break; /* selects other main menu */
    case 'n': another_startup = TRUE; goto load_startup_file; /* loads a new startup file */
    case 'q': input_window(); color.purple; cprintf("Are you sure you want to quit? "); if (yes_prompt(TRUE) != TRUE) ch = 0;
        normal_video(); clrscr(); break; /* quits program */
        default:
        if (ch == 'A') ++(ch <= 'Z')
        {
            input_window();
            print_error("Warning: CAPS LOCK may be on; press any key: ");
            break; /* does nothing for all other inputs, but warns that CAPS LOCK may be on */
        }
        else
            switch (ch)
            {
                case 'B': Base = Base: view_mask(Mask[current_mask].valid, FALSE); break; /* toggles default base between hex and decimal */
            case 'C': clrscr(); color lt.blue; cprintf("Clear frame grabber buffer");
            case 'V': clrscr(); color lt.blue; cprintf("View Image \n\nImage (get_frame_grab_num(TRUE, TRUE));
        view_mask(Mask[current_mask].valid, FALSE); break; /* views some pixels of frame grabber image in mask window */
    case 'O': other_menu = !other_menu; break; /* selects the main menu */
    case 'D': input_window(); cprintf("Draw wedge (\n\nRed color. ");
        cputs("WARNING: All frame grabber images will be cleared"; normal_video();
        gotoxy(12,1); if (yes_prompt(FALSE))
            {cputs("Y");
                spawnl(P_WAIT, "/fr_grab/wedge_fb", NULL); /* draws wedge on frame grabber */
            }
            clrscr();
            break; /* clears frame-grabber window */
    case 'G': input_window(); color.purple; /* gets single pixel value from frame grabber */
        cputs("Note: All pixel locations are relative to (0,0) of frame grabber buffer 0");
        normal_video();
        cputs("\n\nEnter X coordinate: "; x = integer_input(FALSE, 0); clrscr();
        cprintf("Pixel location: (%d,%d) \n\nPixel value: "; x, y, get(x,y));
        color.green; cputs("\n\nPlease any key: "; normal_video(); getch(); clrscr();
        break; /* plots single pixel value to frame grabber */
    case 'P': input_window(); color.purple; /* plots single pixel value to frame grabber */
    normal_video();
        cputs("\n\nEnter X coordinate: "; x = integer_input(FALSE, 0);
        cputs("\n\nEnter Y coordinate: "; y = integer_input(FALSE, 0);
        cputs("\n\nEnter mask value: "; pixel = integer_input(FALSE, 0); clrscr();
        cprintf("Pixel location: (%d,%d) \n\nReplace old pixel value (%d) with new pixel value (%d)? "; x,
        y, get(x,y));
        if (yes_prompt(TRUE) plot(x, y, pixel);
        clrscr();
        break; /* plots single pixel value to frame grabber */
    case 'Q': input_window(); color.purple; cprintf("Are you sure you want to quit? "); if (yes_prompt(TRUE) != TRUE) ch = 0;
        normal_video(); clrscr(); break; /* quits program */
    case UP: if (intr_count <= instr_count) scroll_up(--instr_count); /*scrolls up one instruction, if possible */
            break; /* goes to run menu */
        case DOWN: if (intr_count <= instr_count) scroll_down(instr_count); /*scrolls down one instruction, if possible */
            break; /* goes to run menu */
        case LEFT: do if (Mask_loaded & (current_mask == 0)) current_mask = MAX_MAKES;
        view_mask(Mask[current_mask].valid, FALSE); break; /* views previous mask, if possible */
        case RIGHT: do if (Mask_loaded & (MAX_MAKES)) current_mask = 1;
        while (Mask[current_mask].valid, FALSE); break; /* views next mask, if possible */
        view_mask(Mask[current_mask].valid, FALSE); break; /* goes to run menu */
        default:
        if (ch == 'A') ++(ch <= 'Z')
        {
            input_window();
            print_error("Warning: CAPS LOCK may be on; press any key: ");
            break; /* does nothing for all other inputs, but warns that CAPS LOCK may be on */
        }
    }
    while (ch != 'q');
    normal_video(); /* resets screen before quitting */
    window(1,1,80,25);
    clrscr();
    exit(1);
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <math.h>
#include "cf\define.c"
#include "cfiles\typedef.c"
#include "cfiles\proto.c"

extern directory_type Dir[];
extern FILE *Disk;

int Use_Arrows_For_Input = FALSE;

int from_twos_complement(int num)
    { return (((num > 255) && (num < 512)) ? num - 512 : num); /* converts from two's complement, if necessary */
    }

int to_two_complement(int num)
    { return (((num < 0) && (num >= NEG_INF)) ? num + 512 : num); /* converts to two's complement, if necessary */
    }

int fix_overflow(int mask_val)
    { if (mask_val > 255) return 255;
      if (mask_val < NEG_INF) return -255;
      return mask_val;
    }

void menu_window()
    { window(1,1,LEFT_MASK_BORDER-1,BOTTOM_MENU_BORDER-1); }

void mask_window()
    { window(LEFT_MASK_BORDER-1,1,80,BOTTOM_MASK_BORDER-1); }

void input_window()
    { window(1,BOTTOM_MASK_BORDER-1,80, TOP_INSTR_BORDER-1); }

void instr_window(int bottom)
    { window(1, TOP_INSTR_BORDER-1,80,bottom); }

void bottom_line_window()
    { window(1,25,80,25); }

void comment_window()
    { window(2,2,LEFT_MASK_BORDER-2,BOTTOM_MENU_BORDER-2); }

int convert_hex_string_to_int(char string[5])
    { int hex_val;
      sscanf(string, "0x", &hex_val);
      return hex_val;
    }
```c
/*
 * name: check_input
 * purpose: checks input character for validity
 * inputs: ch: character to check
 * output: TRUE if character is a valid input, FALSE otherwise
 * globals: none
 *******************************************************************************/

int check_input(int ch, enum input_types input_type, int str_length)
    if (input_type == integer) return isdigit(ch); /* return TRUE if checking for integer and the character is an integer */
    else if (input_type == signed_integer) return ((str_length > (ch == '.') || isdigit(ch))); /* return TRUE if checking for signed integer and the character is an integer or '-' */
    else return isprint(ch); /* last remaining choice is string- returns TRUE if the character is printable */

/*
 * name: input
 * purpose: string or integer input routine; reads input one character at a time, allowing for <backspace> or <delete> to delete characters, <esc> to abort, and <enter> to terminate or return default value if no input
 * inputs: max_length: maximum length of user input
 * input_type: integer, signed_integer, or string
 * default_string: output if <enter> is pressed with no input
 */
```
/* must be "" if no default value is to be used 
* output: pointer to string that was read; if aborted, returns "".
* if no string read, returns default_string 
* globals: none
*/

strptr

input(int max_length, enum input_types input_type, file_name_type default_string)
{
    int ch = getch();
    switch (ch) {
        case ESC : if (Use_Arrows_For_Input = TRUE) str[0] = ESC; str[1] = NULL_CHAR;
                       /* if arrows were used to read directory, the escape needs to be passed to the calling function */
                       else str[0] = str[1] = NULL_CHAR; done = TRUE; break;
        case ENTER : if (is_empty(str)) /* if compiled string is empty, return default */
                                  return (strcpy(str, default_string);
unityf("Le", default_string);
    done = TRUE;
    break;
        case 0 : ch = getch(); /* extended ASCII character was pressed, and another character is waiting in the"
           input buffer */
           if (Use_Arrows_For_Input = FALSE) /* if using arrows for reading directory... */
           if (ch == UP) (str[0] = 24; str[1] = NULL_CHAR; done = TRUE; break) /* up arrow value needs to
            be passed to calling function */
           if (ch == DOWN) (str[0] = 25; str[1] = NULL_CHAR; done = TRUE; break) /* down arrow value
            needs to be passed to calling function */
           if (ch == DELETE) break; /* DELETE key has same function as BACKSPACE, therefore don't break */
    case BACK : if (is_empty(str)) /* if there is something to be deleted in the compiled string... */
              (str[strlen(str)-1] = NULL; /* move the string terminator back one character */
              if (wherey() == 1) gotoxy(xmax, wherey()-1); /* if at 1st character position, move to end of
              previous line */
              else gotoxy(wherey()-1, wherey()); /* otherwise move back one character */
              clear(); /* clear deleted character from screen */
            break;
    default : if ((strlen(str) < max_length) && check_input(ch, input_type, strlen(str))) /* check to see if
                   it's OK to add another character to string */
                   strcat(str, strptr(ch)); /* concatenate character to string */
                        sprintf("%c", ch); /* print concatenated character on screen */
                       xmax = max(xmax, wherey()); /* keeps track of xmax (rightmost character), used for deleting
                       characters if input wraps to next line */
                    break;
    }
    while (!done)
        return str;
    }

 understanding

 /***************************************************************************/
 * name: integer_input
 * purpose: inputs an integer, with an optional default value
 * inputs: use_default: if TRUE, use default_num as default input
 * default_num: default value to be used (if use_default is TRUE)
 * output: integer entered, 0 if aborted
 * globals: none
 /***************************************************************************/

 int integer_input(int use_default, int default_num)
 {
     char dummy[5];
    if (use_default)
    return atoi(input(4, integer, ltoa(default_num, dummy, 10)));
            else return atoi(input(6, integer, ""));
    }

 /***************************************************************************/
 * name: prompt
 * purpose: inputs a character with cursor at specified coordinates
 * inputs: xpos, ypos: x and y coordinates (in menu window)
 * output: first key pressed
 * globals: none
 /***************************************************************************/

 int prompt(int xpos, int ypos)
 {
     (menu_window()); gotoxy(xpos, ypos); return getch();
    }

 /***************************************************************************/
 * name: yes_pprompt
 * purpose: generic prompt for a "yes" or "no" question
 * inputs: default value: YES, NO, FALSE for no default
 * output: TRUE if first keypress is 'y' or 'Y', ABORT if esc was pressed,
 * FALSE otherwise
 * globals: none
 /***************************************************************************/

 int yes_pprompt(int default_val)
 {
     char ch;
     cputs(" (y/n) ");
        if (default_val == YES) cputs("y ");

if (default_val == NO) puts("\n");
while (TRUE)
{ ch = yorchr();
  if (tolower(ch) == 'y' || !default_val || (ch == ENTER)) return TRUE;
  if (tolower(ch) == 'n' || (default_val == NO && ch == ENTER)) return FALSE;
  if (ch == ESC) return ABORT;
  
  /**************************************************************
  * name: open_file  
  * purpose: prompts user for a file name and tries to open that file; if  
  * trying to open a write file and the file exists, the user  
  * will be prompted to abort or replace file, also allows user to  
  * use arrow keys to read a directory and select a file in that  
  * directory  
  * inputs: type: type of file to open, used in prompt (max length 11)  
  * mode: file open mode for fopen command (max length 3)  
  * output: pointer to name of file opened, "" if unsuccessful  
  * globals: Disk is a pointer to the opened stream, NULL if unsuccessful  
  **************************************************************/
open_file(int type, char mode[4], file_name_type default_directory, file_name_type default_name)
{ char type_string[12];
  char default_extension[5] = EMPTY_STRING;
  file_name_type temp_file_name;
  int l, i = 0, dir_entry = 0, max_dir_entries, new_directory, get_filename = FALSE, used_arrows = FALSE;
  char temp_string[REALY_LONG_STRING_LENGTH];
  FILE *disk;
  
  if (strstr(default_name, '\\') != NULL)
  { default_name = strstr(default_name, '\\') + 1;  
    /* remove path of default filename so default directory is used instead */
    
    switch (type)  
    { case SOURCE  
        : strcpy(type_string, "Source"); break;
      case DESTINATION  
        : strcpy(type_string, "Destination"); break;
      case INSTRUCTION  
        : strcpy(type_string, "Instruction"); break;
      case WINDOW  
        : strcpy(type_string, "Window"); break;
      case STARTUP  
        : strcpy(type_string, "Startup"); break;
      }  
    do
    { input_window(); clrscr();
      color(purple);
      printf("Enter string to search: ");
      if (is_empty(default_directory))  
      { /* prints default directory, if it exists */
        printf(\"\n\");  
        printf("Enter default directory\n");  
      }  
      normal_video(); gotoxy(1,1);
      printf("Enter file name directory: ");
      strcpy(default_directory, temp_string);  
      
      if (get_filename)
      {  
        if (strstr(type_string, "\n\")) putch('\n');  
        /* prints "\n" at the end of the directory if not specified by the user */
        ypos1 = ypos2 = wherey();  
        printf("Enter file name: ");
        temp_string[0] = 1;  
        if (strstr(type_string, "\n\")) putch('\n');  
        /* saves cursor position */
        new_directory = FALSE;  
      }  
      else if (strstr(type_string, "\n\"))  
      { /* prints default extension if in read mode */
        printf(\"\n\");  
      }  
      else
      { /* prints default extension */
        printf(\"\n\");  
      }
      normal_video(); gotoxy(1,1);
      cprintf("Enter file name: ");
      if (strstr(type_string, "\n\")) printf(\"\n\");  
      putchar('\n');  
      if (is_empty(default_name))  
      { /* prints default filename if one exists */
        printf(\"\n\");  
      }  
      else
      { /* print default extension */
        printf(\"\n\");  
      }
      printf("Enter file name directory: ");
      strcpy(default_directory, temp_string);
      
      normal_video(); gotoxy(1,1);
      printf("Enter file name: ");
      if (strstr(type_string, "\n\")) printf(\"\n\");  
      putchar('\n');  
      if (is_empty(default_name))  
      { /* print default filename if one exists */
        printf(\"\n\");  
      }  
      else
      { /* print default extension */
        printf(\"\n\");  
      }
      normal_video(); gotoxy(1,1);
      cprintf("Enter file name: ");
      if (strstr(type_string, "\n\")) printf(\"\n\");  
      putchar('\n');  
      if (is_empty(default_name))  
      { /* print default filename if one exists */
        printf(\"\n\");  
      }  
      else
      { /* print default extension */
        printf(\"\n\");  
      }
    }  
  }  
  else
  {  
    if (used_arrows == TRUE)  
    { /* the up/down arrows (for reading from a directory) have been used... */
      strcpy(temp_file_name, EMPTY_STRING);  
      ch = getch();
      if (ch == 0) ch = getch();  
      /* read again if an extended ASCII character has been pressed */
      switch (ch)
      { case ENTER  
        : break;  
        /* the file has been chosen */
      case UP:
        temp_file_name[0] = 25; temp_file_name[1] = NULL_CHAR; break;
        /* read the previous file from the directory */
      case DOWN:  
        temp_file_name[0] = 25; temp_file_name[1] = NULL_CHAR; break;
        /* read the next file from the directory */
      case BACK:
        case DELETE:
          gotoxy(xpos1, ypos1); clrscr(); used_arrows = FALSE; break;  
        /* aborts the read from directory function */
      case ESC  
        : break;  
        default
        : temp_file_name[0] = 26; temp_file_name[1] = NULL_CHAR; break;  
        /* specifies no operation */
    }  
  }  
  Use_Arrows_For_Input = TRUE;  
  /* allows arrows to be use to read from a directory */
if (used_arrows == FALSE) strcpy(temp_file_name, input(MAX_FILE_NAME_LENGTH, string, default_name)); /* reads the filename */

Use_Arrows_For_Input = FALSE;
if (temp_file_name[0] == ESC) clrsck(); return EMPTY_STRING; /* returns empty string if abortd */
if (is_empty(temp_file_name) ) /* if no filename exists yet... */
(clrsck();
  if ([apos] == 'POS2) 46 (used_arrows = TRUE) ) /* return filename from use of arrows */
  strcat(temp_file_name, Dir(dir_entry).filename); /* concatenates directory and filename... */
  strcat(temp_file_name, ".");
  strcat(temp_file_name, Dir(dir_entry).extension);
  break;
}
else return EMPTY_STRING; /* no filename exists */
/*
if ([temp_file_name[0] = 0] || (temp_file_name[0] = 25)) /* arrows have been used to read from directory...
(used_arrows = TRUE);
if ([new_directory == TRUE) ) /* if reading from a new directory... */
(new_directory = FALSE; /* specify that it's no longer a new directory, so it doesn't keep on getting read */
  color_green(); epts("reading directory..."); normal_video();
  if ([directory[strlen(directory)] - 1] == NULL_CHAR; /* removes extra \ */
     strcat(directory_command, "; dir "); /* concatenates system directory read command... */
     strcat(directory_command, directory);  
     strcat(directory_command, " > temp.dir"); /* used to pipe the directory read into a temporary file */
     system(directory_command);
     while((fgets(temp_string, MAX_LONG_STRING_LENGTH - 1, disk)) 
       if (is_empty(temp_string) ) get_filename = TRUE; /* a blank line exists between the directory info and
the filenames, */
          /* so once the blank line is found, filenames will follow */
  }
  max_dir_entries = 1; /* set max_dir_entries to the number of filenames found */
  for ([i=0; i <= max_dir_entries; i++ ) /* move the filenames into an array... */
    for ([x=0; x<; x++]
      if ([dir[i].filename[0] == SPACE] Dir[i].filename = NULL_CHAR;
      if ([x=0; x<; x++)
      if ([dir[i].extension[0] == SPACE] Dir[i].extension = NULL_CHAR;
    }
  }
else if ([temp_file_name[0] = 0] || (temp_file_name[0] = 25) ) dir_entry--; /* if up arrow is pressed, go up on file */
else if ([temp_file_name[0] = 25) dir_entry++; /* if down arrow is pressed, go down one file */
  if (dir_entry < 0 ) dir_entry = max_dir_entries - 1; /* if reading up beyond first file, reset to last file */
  if (dir_entry >= max_dir_entries) dir_entry = 0; /* if reading down beyond last file, reset to first file */
  gotoxy(xpos, ypos1);
  sprintf("%s.k2", Dir[dir_entry].filename, Dir[dir_entry].extension); /* prints current filename */
  ypos2 = wherey(); ypos2 += 3;
  if (clrline() ) /* clears to end of line, in case the filename is shorter than the previously displayed one */
    break;
  if ([Disk] Disk = fopen(temp_file_name, mode); /* open the file in the specified mode (read or write */
    if (!Disk) exit(1); /* if the file isn't found, */
    try again after adding the default extension */
    if (Disk ) print_error("Error: file not found, hit <esc> to abort or any other key to try again ");
    return EMPTY_STRING; /* if the file still isn't found, inform user and return */
    while([Disk];
      clrsck();
      return temp_file_name; /* returns the opened filename */
    )
  }
/**
 * name: print_comment
 * purpose: prints image file comments to screen
 * inputs: size: size of comment, or -1 if printing comment entry prompt
 *          'comment': comment text
 * output: none
 * globals: none
 */

void print_comment(int size, char *comment)
{
    int i, control_char = FALSE, comment_entry = FALSE;
    char ch;
    if (size == -1) /* comment entry mode */
    { size = 255; /* set comment size to maximum */
        comment_entry = TRUE;
    }
    putch(218); /* print borders on screen... */
    for (i = 0; i < 17 - 3 * comment_entry; i++) putch(196);
    if (comment_entry) cputs("enter ");
    cputs("comment:");
    for (i = 25 + 3 * comment_entry; i < LEFT_MASK_BORDER-2; i++)
        putch(196);
    putch(191);
    for (i = 0; i < size; i++)
        if (whereX() == LEFT_MASK_BORDER-1)
            putch(179);
        if (whereY() == 1)
            putch(179);
        if (comment_entry)
            putch(* (comment + i));
        else
            switch (ch = *(comment + i)) /* check for non-printable characters */
            { case 0 : case 7 : case 8 : case 9 : case 10 : case 13 :
                control_char = TRUE; /* if non-printable character, print in pink */
                color_pink();
                putch(ch == 0 ? 219 : ch + 'A' - 1);
                normal_video();
                break;
            default : putch(ch); /* print if printable character */
                }
    gotoxy(LEFT_MASK_BORDER-1, whereY());
    putch(179);
    putch(192);
    for (i = 0; i < LEFT_MASK_BORDER-2; i++)
        putch(196);
    putch(217);
    if (control_char == TRUE)
        { gotoxy(3, whereY() - 1);
            color_pink();
            cputs("(unprintable control chars highlighted)");
            normal_video();
        }
}

/**
 * name: isnumber
 * purpose: checks if a string is a number
 * inputs: string: string to check
 * output: TRUE if input is a number, FALSE otherwise
 * globals: none
 */

int isnumber(char_string)
{
    int i;
    if (is_empty(string)) return FALSE;
    for (i = 0; i < strlen(string); i++)
        if (isdigit(string[i])) return FALSE;
    return TRUE;
}

/**
 * name: convert_bytes_to_int
 * purpose: converts two bytes (low and high order) to an integer
 * inputs: high, low: high and low order bytes
 * output: integer value of input word
 * globals: none
 */

int convert_bytes_to_int(unsigned char high, unsigned char low)
{return(high * 0x100 + low);
}

/**
 * name: fread_chars_to_int
 * purpose: reads in two bytes (chars) from previously opened file (Disk),
 *           LOB first, then HOB, returns integer equivalent
 * inputs: none
 * output: integer equivalent of bytes read
 * globals: none
 */

unsigned int fread_chars_to_int()
{unsigned char val_low, val_high;
*****/
\begin{verbatim}
val_low = getc(Disk); val_high = getc(Disk);
return convert_bytes_to_int(val_high, val_low);

 /******************************************************************************************************************
* name:  fwrite_int_to_chars  
* purpose: converts input word into bytes (chars) and writes to previously opened file (Disk), LOB first, then HOB
* inputs: val: word to write into bytes (chars)  
* output: none  
* globals: none  
******************************************************************************************************************/

void fwrite_int_to_chars(unsigned int val)
{putc(val & Oxff, Disk);
 putc(val >> 8, Disk);
}
\end{verbatim}
```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <conio.h>
#include "cf\typedef.c"
#include <dos.h>
#include "cf\proto.c"

extern mask_type Mask[];
extern instr_type Predef_Instr[ ];
extern int Predef_Instr_Count, Reg_Loc, Board_Loc_High;
extern predef__instr_type Predef_Instr[];
extern fr_grab_info_type Fr_Grab;
extern FILE *Disk;

void print_instr_predef_file_not_found()
{color_red();
cprintf("WARNING: instruction definition file 'is' not found.", INSTRUCTION_PREDEFINE_FILE);
normal_video();}

void print_instr_predef_error(int i)
{color_red();
cprintf("ERROR: predefined instruction 'is' not valid.", Predef_Instr[i].instr_name);
normal_video();}

void print_instr_error(int instr_num, long_string error_string)
{input_window(); color_red();
cputs("ERROR: "); color_pink();
cprintf(" is: ", instr_num );
cprintf(" is:", instr_num );
cprintf(" is:", instr_num );
cputs("press any key:"); normal_video();
gets(); clrscrO; getchO; input_window();
}

int read_instr_def_file()
{char alul_op[5], bus_mode[5], max[5], alu2_op[5], dummy[200];
  int i=0;
  FILE *disk = fopen(INSTRUCTION_PREDEFINE_FILE, "r"); /* opens predefined instruction file */
  if (!disk) {print_instr_predef_file_not_found(); return 0;} /* file not found */
  fgets(dummy, 199, disk); fgets(dummy, 199, disk); fgets(dummy, 199, disk); /* the first three lines of the file aren't predefined instructions */
  while(fscanf(disk, "%s is %s is", Predef_Instr[i].instr_name, alul_op, max, alu2_op) > 0)
    /* reads the instruction name, ALU1 operation, MAP operation, and ALU2 operation, in that order */
    if (tolower(alu1_op[0]) == 'x') Predef_Instr[i].alu1_op = 0; /* if "don't care" for ALU1 operation, default to 0 */
```

# register_write(int val)

```c
int register_write(int val)
{
    int ch = 0;
    do {
        if (inportb(Reg_Loc + 0x0C) == val) return TRUE; /* returns TRUE if polled bit is high */
        if (kbhit()) ch = getch(); /* reads character if a key is pressed */
    } while (ch != ESC); /* continues polling register until <Esc> has been pressed */
    return FALSE;
}
```

# register_read(int reg_offset)

```c
int register_read(int reg_offset)
{
    return (int) inportb(Reg_Loc + reg_offset); /* reads from the specified 8-bit register */
}
```

# register_write(int reg_offset, int val)

```c
void register_write(int reg_offset, int val)
(outportb(Reg_Loc + reg_offset, (unsigned char) val)); /* writes to the specified 8-bit register */
```
/*
 * globals: none
 */

strptr

find_string(long_string, int start_delimiter, long_string end_delimiters, int direction)
strptr ptr;
long_string temp_string; /* use a temporary string because so as not to modify the original string */
strcpy(temp_string, string);

if ((direction == FORWARD) ptr = strchr(temp_string, start_delimiter); /* find start delimiter, search from

beginning of string */
else ptr = strchr(temp_string, start_delimiter); /* or find start delimiter searching from end of string (used for

finding the ALU2 constants) */
if (*ptr) return EMPTY_STRING; /* if not found, return EMPTY_STRING */

/*ptr */ = strchrn(ptr, end_delimiters) + 1 = NULL_CHAR; /* set the character following the first end delimiter found to NULL_CHAR, which terminates the string */

/*ptr */ = return *ptr; /* return the string found, incrementing the pointer so as not to include the start delimiter in the returned string */
}

/*******************************************************************/

/*
 * name: register_write
 * purpose: writes the mask to the board
 * inputs: mask_num: number of mask to write
 * reflect_mask: if true, rotate mask 180 degrees and negate
 * output: none
 * globals: none
 */

void

write_mask(int mask_num, int reflect_mask)

int i, j, mask_value;
if (reflect_mask == FALSE) /* if not reflecting, write normal values */

for (j=0; j<mask_SIZE; j++)

for (i=0; i<mask_SIZE; i++)

write_mask_to_board(2, int(Mask_num).val[i][j]); /* calls assembly language function to write the mask

value */

else /* reflect the mask (rotate 180 degrees and negate all values) used for erasing */

for (j=mask_SIZE-1; j>0; j--) /* loop backwards */

for (i=mask_SIZE-1; i>0; i--)

(mask_value = int(Mask_num).val[i][j][j](i));

if (mask_value == NEG_INF) mask_value = 255; /* if negative infinity, set to maximum positive value */

else mask_value = -mask_value; /* otherwise negate pixel */

write_mask_to_board(2, mask_value); /* calls assembly language function to write the mask value */

}/*
if (is_ext_mem(ext_mem_num)) xl = ext_mem_num; /* checks if valid */
else if (tolower(xl_string[0]) == ';') /* if the source image is a file (indicated by being enclosed by double quotes) */
{ /* skip file name */
  xl = strlen(xl_string) - 2; /* only scan to end of file name */
  x_string = NULL_CHAR; /* remove closing double quote */
  xl = 9; /* 9 is the file code */
  Disk = fopen(x_file, "r"); /* open file */
  if (!Disk) print_error("file not found, hit <esc> to abort or any other key to try again");
  read_lps_header(); /* reads the header information */
}

/** decode X2 (between ']' and next delimiter) **/;
strcpy(x2_string, find_string(instr, '\', '[', '"', ':;,=)'/*, FORWARDS*/); /* finds X2 */
if (is_empty(x2_string)) x2 = -1; /* no value for X2 specified */
else if (is_number(x2_string)) /* X2 must now be a number... */
  x2 = atol(x2_string); /* convert to integer */
else if (strtok(x2_string) != NULL) x2 = -2; /* checks if X2 is valid, x2 = -2 (invalid) if not */
else x2 = -2; /* x2 is invalid */

/** decode Y (between ']' and next delimiter) **/;
y = -2; /* assume Y bus not valid */
strcpy(y_string, find_string(instr, '\', '"', ':;,=)'/*, FORWARDS*/); /* finds Y */
if (tolower(y_string[0]) == '\') strcpy(y_string, strtok(y_string, "", "m")); /* Y is a board memory, remove the 'm' */
if (it's optional) y = atoi(y_string); /* convert to integer */
else if (is_number(y_string)) /* if Y is a number... */
  y = atoi(y_string); /* convert to integer */
else if (strtok(y_string) != NULL) y = -2; /* return -2 if Y is invalid */
else if (copy) /* copy instruction has been specified */
  if (tolower(y_string[0]) == '\') /* if the destination image is the frame grabber... */
    if (is_number(y_string)) /* y = tree + 1; */
      y = atoi(y_string); /* convert to frame grabber code */
    else if (is_board_mem(y) y = fr_grab_num; /* checks if valid */
  else if (tolower(y_string[0]) == '"') /* if the destination image is an extended memory... */
    if (is_ext_mem(y) y = ext_mem_num; /* checks if valid */
  else if (strtok(y_string) == NULL) /* if the destination image is a file (indicated by being enclosed by double quotes)... */
  { /* for [100: <extension string>] 1**1 y_file[i] = y_string[i] + 1; */
    copy_file_name y_string[strlen(y_string) - 2] = NULL_CHAR; /* remove closing double quote */
    y = 9; /* 9 is the file code */
  }

/** determine if memory mapping must occur **/;
mem_swap = ((x1 > 3) && (y > 0)) || ((x1 > 0) && (y > 3)); /* memory swapping occurs if not copying from one board memory to another */

if (mem_swap)
{ if (is_file(xl)) /* if source is a file */
  if (is_ext_mem(xl) copy_file_to_ext_mem(y - 100, 8); /* if destination is an extended memory */
    if (is_board_mem(y) copy_file_to_board_mem(y, 8); /* if destination is a board memory */
  else if (is_file(xl) copy_file_to_file(xl - 10, y - 100); /* if destination is a frame grabber */
    if (is_ext_mem(y) copy_file_to_ext_mem(xl - 10, y - 100); /* if destination is an extended memory */
    if (is_board_mem(y) copy_file_to_board_mem(xl - 100, y); /* if destination is a board memory */
  else if (is_file(xl) copy_file_to_file(xl - 100, 8, FALSE); /* if destination is a file */
    if (is_ext_mem(xl) copy_ext_mem_to_ext_mem(xl - 100, y - 100); /* if destination is a frame grabber */
    if (is_board_mem(xl) copy_ext_mem_to_board_mem(xl - 100, y); /* if destination is a board memory */
  else if (is_ext_mem(xl) copy_ext_mem_to_ext_mem(xl - 10, y - 100); /* if destination is a frame grabber */
    if (is_board_mem(xl) copy_ext_mem_to_board_mem(xl - 100, y); /* if destination is a board memory */
  else if (is_ext_mem(xl) copy_board_mem_to_ext_mem(xl - 100, y); /* if destination is a frame grabber */
    if (is_board_mem(xl) copy_board_mem_to_board_mem(xl - 100, y); /* if destination is a board memory */
  else /* instruction is purely a board instruction (and not off-board image transfers) */
  { /* decode mask [between ']' and next delimiter] */
    strcpy(mask_string, find_string(instr, '\', '"', ':;,=)'/*, FORWARDS*/); /* finds the mask number */
    if (tolower(mask_string[0]) == '\') strcpy(mask_string, strtok(mask_string, "", "m")); /* remove the optional 'm' if it exists */
    if (is_empty(mask_string)) mask_num = -1; /* no mask value found */
    else if (isdigit(mask_string[0]) /* it looks valid so far... */
    { mask_num = atoi(mask_string); /* convert to integer */
      if (mask_num == MAX_MASKS) mask_num = -2; /* mask is invalid */
    }
    else /* mask is a file name */
      for (i=0; i<strlen(mask_string); i++) Mask[TEMP_MASK_NUM].filename[i] = mask_string[i + 1]; /* copy file name */
  }
}

B-14
if it
/***/
if (mask_num == 3) // mask file was not found or invalid */

}  /* decode ALU1 constant (between first ';' and next delimiter) */
strncpy(alul_k_string, find_string(instr, ';', ';;', FORWARDS)); /* finds ALU1 constant (starts from beginning)
if (tolower(alul_k_string[0]) == 'k') strncopy(alul_k_string, strtonum(alul_k_string, 'k')); // remove optional 'k'
if (is_empty(alul_k_string)) alul_const = convert_int_string_to_int(alul_k_string); /* if ALU1 constant exists,
convert to integer */
else alul_const = -1; /* ALU1 constant doesn't exist */

}  /* decode ALU2 constant (between last ';' and next delimiter) */
strncpy(alul_k_string, find_string(instr, ';', ';;', BACKWARDS)); /* finds ALU2 constant (starts from end)
if (tolower(alul_k_string[0]) == 'k') strncopy(alul_k_string, strtonum(alul_k_string, 'k')); // remove optional 'k'
if it exists */
if (is_empty(alul_k_string)) alul2_const = convert_int_string_to_int(alul_k_string); /* if ALU2 constant exists,
convert to integer */
else alul2_const = -1; /* ALU2 constant doesn't exist */

}  /* decode bus mode (if 3rd input of ALU2 is used, bus mode is 1) */
bus_mode = ((Predef_Instr[predef_instr_num].alu2_op & 0x0C) > 0);

}  /* decode memory selection: memX = 0 if X1, 1 if X2, 2 if Y, 3 otherwise */
mem0 = 3 - 3 * (x1 == 0) - 2 * (x2 == 0) - 1 * (y == 0);
mem1 = 3 - 3 * (x1 == 1) - 2 * (x2 == 1) - 1 * (y == 1);
mem2 = 3 - 3 * (x1 == 2) - 2 * (x2 == 2) - 1 * (y == 2);
mem3 = 3 - 3 * (x1 == 3) - 2 * (x2 == 3) - 1 * (y == 3);

mem_select_reg = mem0*0x00 + mem1*0x00 + mem2*0x00 + mem3*0x00; /* combine into memory select register value */

}  /* check for errors */
if ((x1 == -2) || (x2 == -2)) print_instr_error(instr_num, "has an invalid source");
else if (x1 == -2) print_instr_error(instr_num, "has an invalid destination");
else if (mask_num == -2) print_instr_error(instr_num, "has an invalid window number");
else if (mask_num == -3) print_instr_error(instr_num, "has an invalid window filename");
else if (mask_num > 0) && (mask[mask_num].valid == 0)
print_instr_error(instr_num, "has no window loaded");
else if (x1 < 0) && (x2 < 0) print_instr_error(instr_num, "has invalid sources");
else if ((x1 == y) || (x2 == y)) print_instr_error(instr_num, "has identical source and destination");
else if (((alu2_op & 0x00) == 0x90) && (alu2_const == -1)) print_instr_error(instr_num, "has no ALU2 constant");
else if (alu2_op & 0x00) == 0x90) && (alu2_const == -1)) print_instr_error(instr_num, "has no ALU2 constant");
else error = FALSE; /* no error has been found */

}  /* remove high-order bit from ALU constants and incorporate into ALU opcodes */
if (alu1_const > 0)
alul1_const = (alu1_const >> 8);
alul2_const = 0;
else alul1_const = 0;
alul1_const = 0;
alul1_const = 0;

}  /* check to see if OK to issue next instruction and window */
if (register_poll(INSTR_LOAD_READY) == FALSE) return FALSE;

}  /* print and run */
if (error)
{  /* writes to ADUCTRL register */
write_to_ADJCTRL(register, 0x1);
write_to_SEGMNT(register, 0x2);
}

}  /* print and run */
if (mask_num > 0)
write_mask_reg = Predef_Instr[predef_instr_num].max]; /* if valid mask, write to board */
write_to_ALU1_CONST(register, 0x5, alul1_const); /* writes to ALU1 constant register */
write_to_ALU2_CONST(register, 0x8, alul2_const); /* writes to ALU2 constant register */
write_to_ALU1_OPR(register, 0x7, alul1_op); /* writes to ALU1 operation register */
write_to_ALU2_OPR(register, 0x8, alul2_op); /* writes to ALU2 operation register */
write_to_ADDRCTRL(register, 0x0, mem_select_reg); /* write to memory select register */
write_to_ADDRCTRL(register, 0x0, FALSE); /* waits until ready to start */
write_to_ADDRCTRL(register, 0x0, 0); /* three start signals to take care of metastability problem */

}  /* slight delay, so as not to read the register before it's ready... may not be necessary if reading volume first */

}  /* starts operation register */
write_to_ADDRCTRL(register, 0x0, FALSE); /* holds until ready to start */
write_to_ADDRCTRL(register, 0x0, 0); /* three start signals to take care of metastability problem */

}  /* slight delay, so as not to read the register before it's ready... may not be necessary if reading volume first */

if (register_poll(START_READY) == FALSE) return FALSE;

}  /* get volume from previous run */
for (offset=0; offset<5; offset++)
volume[offset] = register_read(offset); /* reads volume adder registers */
volume[4] = 0x00;

return TRUE; /* instruction was run successfully */

}  /* get volume from previous run */

void run(int first_instr, int last_instr)
{
    int i, predef_instr_match;
    if (first_instr > last_instr) /* if trying to run from a higher number to a lower number... */
    { input_window(); print_error("ERROR: first instruction is after last instruction; press any key: ");
    } else if (first_instr == 0) /* everything looks OK so far... */
    { for (i = first_instr; i <= last_instr; i++) /* loop from first to last instruction */
        if (i < predef_instr_match || instr_comp(inst_num[i])) >= 0) /* if the instruction was found... */
            if (decode_and_run_instr(i, predef_instr_match) == FALSE) return; /* return if successful */
            else print_instr_error(i, "is not a valid instruction"); /* the instruction wasn't found */
    }
}

int get_instr_num(int instr_count, char msg_string[7], int default_num)
{
    char default_num_string[4] = EMPTY_STRING, instr_num_string[4];
    if (default_num) itoa(default_num, default_num_string, 10);
    color_green(); cprintf("%rEnter %s instruction number: ", msg_string, normal_video);
    if (is_empty(string(instr_num_string, input(1, integer, default_num_string)))
        return default_num;
    instr_num = atoi(instr_num_string);
    if (instr_num < 1) // [inst_num > instr_count]
        input_window(); print_error("ERROR: invalid instruction number; press any key: ");
        return default_num;
    return instr_num;
}

void run_menu(int instr_count)
{
    static int first_instr, last_instr=1000;
    if (first_instr > instr_count) first_instr = instr_count;
    if (last_instr > instr_count) last_instr = instr_count;
    do
    { menu_window(); clrscr();
        color_lt_blue(); cprintf("%rRun operations:
"%r"

"normal_video();
        cputs("[r] run");
        if (instr_count != 1) /* only display the following options if there is more than 1 instruction */
            cprintf("%rAll [1 - %d]%r\n", instr_count);
            cputs("%r[0] run one instruction\n");
            cputs("%r[p] partial run\n");
            cputs("%r[S] select first instruction\n");
            cputs("%r[L] select last instruction\n");
        cputs("%r[q] quit run instructions\n");
        color_green(); cputs("Enter your choice: "); normal_video();
        if (instr_count != 1) ch = input_char("rospflq23456789");
        else ch = input_char("rq");
        switch(ch)
        { case 'r': run(1, instr_count); break; /* run all instructions */
            case '0': instr_num = get_instr_num(instr_count, "", 0); run(instr_num, instr_num); break; /* run one instruction */
            case 'p': for (i = 1; i <= instr_count; i++) /* step through instructions */
                (input_window(); cprintf("Press any key to run instruction\n"); getch()); clrscr();
                menu_window();
                run(i, 1); /* run the current instruction */
            break;
            case 's': for (i = 1; i <= instr_count; i++) /* step through instructions */
                (input_window(); cprintf("Press any key to run instruction\n"); getch()); clrscr();
                menu_window();
                run(i, 1); /* run the current instruction */
            break;
            case 'p': run(first_instr, last_instr); break; /* partial run */
            case 's': first_instr = get_instr_num(instr_count, "first", first_instr); break; /* select first instruction for partial run */
            case 'l': last_instr = get_instr_num(instr_count, "last", last_instr); break; /* select last instruction for partial run */
            default: if (ch == 'q') & (ch < '9') /* instruction numbers 1 through 9 may be run by pressing the keys */ 1-9 */
                { sound(460); delay(5); nosound(); /* alert user that run is starting */
                    run(ch - '0', 0); ch = '0'; /* run the desired instruction */
            } /*
sound(810); delay(5); nosound(); /* alert the user that the run is done */

while (ch != 'q');
#include <stdio.h>
#include <string.h>
#include <conio.h>

#include "files/define.c"
#include "files/typesdef.c"
#include "files/typedef.c"
#include "files/proto.c"

extern instr_type instr;
extern FILE *Disk;

/**
 * purpose: print instruction
 * inputs: instr_num: instruction to print
 * output: none
 * globals: none
 */
void print_instr(int instr_num)
{
  printf("%d
", instr_num); 
}

/**
 * purpose: print instruction file and number of instructions to screen
 * inputs: instr_count
 * output: none
 * globals: none
 */
void print_instr_filename(int instr_count)
{
  (bottom_line_window()); clrscr();
  if (instr_count) 
  (color_lt_blue());
  if (!is_emptyInstr.filename)) printf("filename: ", instr.filename);
  printf("(Instr. instructions)", instr_count, instr_count==1 ? "s" : "s");
  normal_video();
}

/**
 * purpose: displays instructions on screen (as many as can fit)
 * inputs: current_instr: first instruction to display
 * instr_count: number of instructions
 * output: none
 * globals: none
 */
void show_instr(int current_instr, int instr_count)
{
  int i;
  instr_window(24); clrscr(); /* goes to instruction window and clears the screen */
  for (i=current_instr; i<current_instr+8 && i<instr_count; ++i) /* loops from current instruction to 8 instructions */
  if (/* instruction or last instruction, whichever comes */
  first */
  print_instr(i); 
}

/**
 * purpose: opens instruction file as specified by user
 * inputs: old_instr_count: return value if no new instr. file loaded
 * temp_file_name: file name to load
 * output: number of instructions
 * globals: instr.Instr[]: new instructions, if successful
 * Instr.filename: new instruction file name, if successful
 */

int load_instruction_file(int old_instr_count, file_name_type temp_file_name)
{
  int instr_num;
  char temp_string[REALLY_LONG_STRING_LENGTH];
  if ((old_instr_count == -1) || !is_empty(strlen(temp_file_name, open_file(INSTRUCTION, "r", Default_instr_Directory, EMPTY_STRING))))
  if (old_instr_count == -1)
    if (!Disk) return 0;
  instr_window(24); clrscr();
  while(!gets(temp_string, REALLY_LONG_STRING_LENGTH - 1, Disk)) /* read instruction from file */

B-18
void 
save_instruction_file(int instr_count)
{
    int i;
    input_window();
    strcpy (Instr.filename, open_file(INSTRUCTION, "w", Default_Instr_Directory, Instr.filename)); /* opens file */
    if (!is_empty(Instr.filename)) /* if OK to write to file... */
        for (i=0; i<instr_count; i++)
            fprintf(Disk, "%s", Instr.instr[i]); /* write instructions to file */
    fclose(Disk);
}

void 
scroll_up(int instr_num)
{
    print_instr(Instr.window(23); insline());
    gotoxy(1, instr_num);
}

void 
scroll_down(int instr_num)
{
    gotoxy(1, instr_num + 7);
}

int 
delete_instr(int instr_count)
{
    int instr_num, i;
    long_string temp;
    char temp_string[MAX_STRING_LENGTH] = *strchr(temp_string, 10) = NULL_CHAR;
    strcpy(Instr.instr[instr_num], temp_string);
    if (i <= instr_num)
        bottom_line_window(); clrscr();
        print_error("Warning: file found but no instructions loaded; press any key: ");
    show_instr(0, instr_num);
    strcpy(Instr.filename, temp_file_name);
    return instr_num;
} else return old_instr_count;

******************************************************************************

name: save_instruction_file
purpose: saves current instructions to file, allows user to abort if file already exists
input: instr_count: number of instructions to save
output: none
globals: Disk is pointer to instruction file name
Instr.filename is instruction file name
******************************************************************************

void 
save_instruction_file(int instr_count)
{
    int i;
    input_window();
    strcpy (Instr.filename, open_file(INSTRUCTION, "w", Default_Instr_Directory, Instr.filename)); /* opens file */
    if (!is_empty(Instr.filename)) /* if OK to write to file... */
        for (i=0; i<instr_count; i++)
            fprintf(Disk, "%s", Instr.instr[i]); /* write instructions to file */
    fclose(Disk);
}

******************************************************************************

name: scroll_up, scroll_down
purpose: scroll the instruction window up or down (if necessary)
input: instr_num: number of first instruction to be displayed
output: none
Globals: none
******************************************************************************

void 
scroll_up(int instr_num)
{
    print_instr(Instr.window(23); insline());
    gotoxy(1, instr_num);
}

void 
scroll_down(int instr_num)
{
    gotoxy(1, instr_num + 7);
}

******************************************************************************

name: delete_instr
purpose: deletes instruction(s) as specified by user
input: instr_count: number of instructions
output: number of instructions
globals: Instr.current: new current instruction
Instr.instr[i]: new instructions
Instr.filename: "" if all instructions deleted
******************************************************************************

int 
delete_instr(int instr_count)
{
    int instr_num, i;
    long_string temp;
    char temp_string[MAX_STRING_LENGTH] = *strchr(temp_string, 10) = NULL_CHAR;
    input_window(); clrscr();
    cputs("Press any key to continue "); normal_video(); gotoxy(1,1);
    cputs("Delete which instruction? ");
    instr_num = atoi(strcpy(temp_string, input(1, integer, EMPTY_STRING))); clrscr(); /* enter instruction to delete */
    if (!is_empty(temp_string) && (instr_num < 0)) return instr_count; /* nothing deleted */
    if (instr_num > instr_count) print_error("ERROR: out of range; press any key: ");
    else
        if ((instr_num) /* delete all instructions... */
            !is_empty(Instr.filename) show_instr(Instr.current, instr_count); normal_video(); input_window(); /* show all instructions in pink */
        else /* delete one instruction */
            if (yes_prompt(NO) == YES)
                instr_count = Instr.current = 0; /* delete all instructions */
            strcpy(Instr.filename, EMPTY_STRING); /* delete instruction file name */
    }
}

if (yes_prompt(YES))
    
/*
*/
(if ((instr_count = instr.current) && (instr.current) instr.current -=; /* delete instruction */
   for (instr_num=1; instr_count; instr_count -=) strcopy(instr.instr[i], instr.instr[i+1]); /* move the next
   instructions up one */
   clrscr();
   show_instr((instr.current, instr_count)); /* show new instruction list */
   return instr_count;
   }
}


/**************************************************************************/
* name: modify_insr
* purpose: modifies an instruction specified by user
* inputs: instr_to_modify: number of instr. to modify, 0 if not known
* instr_count: number of instructions
* output: none
* globals: Instr.instr[instr_to_modify] is new instruction
***************************************************************************/
void
modify_insr(int instr_to_modify, int instr_count)
{

}


/**************************************************************************/
* name: insert_instruction
* purpose: inserts an instruction into current instruction array
* inputs: instr_count: number of instructions
* output: new instruction count
* globals: Instr.instr[instr_to_modify] has new instruction inserted
* Instr.current will change if necessary
***************************************************************************/
int
insert_instruction(int instr_count)
{

}


void print_instr_menu(int instr_count)
{
    clrscr();
    cputs("Instruction operations:\n");
    cputs("(l) load instruction file\n");
    cputs("(i) insert instruction\n");
    cputs("(m) modify instruction\n");
    cputs("(d) delete instruction\n");
    cputs("(r) run\n");
    cputs("(q) quit instruction operations\n");
    color_green(); cputs("Enter your choice: "); normal_video();
}

int instr_menu(int instr_count)
{
    clrscr();
    ch = getch();
    switch(ch)
    {
    case 'l' : instr_count = load_instruction_file(instr_count); print_instr_filename(instr_count); break;
    case 'i' : instr_count = insert_instruction(instr_count); print_instr_filename(instr_count); break;
    case 'm' : instr_count = modify_instruction(instr_count); print_instr_filename(instr_count); break;
    case 'd' : instr_count = delete_instruction(instr_count); print_instr_filename(instr_count); break;
    case 'r' : instr_count = run_menu(instr_count); break;
    case 'q' : break;
    default : break;
    }
    return(instr_count);
}
```c
#include <stdio.h>
#include "files/stdio.c"
#include "files/types.h"
#include "files/protos.c"

extern mask_type Mask[];
extern int Mask_loaded;
extern enum bases Base;
extern file_name_type Default_Window_Directory;
extern FILE *Disk;

enum mask_ops {add, sub, mult, div};

/**************************************************************************
 sensitive: none
 inputs: none
 output: TRUE if weight is valid, FALSE otherwise
 globals: none

int mask_output(int val, int mode)
{
    int valid_weight = (val > NEG_INF) && (val <= 255); /* checks to see if the weight is valid */
    if (valid_weight) {
        cprintf("%4d is \%s \%s
", val, color_green("valid", "yellow"), color_red("weight", "red"));
    } else {
        cprintf("\%s \%s\%s
", color_red("invalid", "red"), color_green("weight", "green"), color_red("weight", "red"));
    }
}

/**************************************************************************
 sensitive: none
 inputs: none
 output: TRUE if weight is valid, FALSE otherwise
 globals: none

int mask_input(enum bases mask_base)
{
    int input_value;
    scanf("%d", &input_value); /* read mask value from disk */
    return from_two_complement(input_value);
}

/**************************************************************************
 sensitive: none
 inputs: none
 output: TRUE if weight is valid, FALSE otherwise
 globals: none

int get_mask_num(char mask_to_copy[], int default_mask)
{
    int mask_num;
    char default_mask_string[10];
    char mask_string[10];
    if (default_mask) {
        mask_string = default_mask_string; /* changes the default mask integer to a string for use with the input */
        color_green(0); cprintf("\%s \%s
", "\%s", color_green("source", "black"), color_red("mask", "red"));
        color_green(0); cprintf("\%s
", "\%s", color_green("source", "black"), color_red("mask", "red"));
        mask_num = atoi(input[0], input[1], input[2]); /* inputs the mask number */
        if (mask_num > 0) return mask_num; /* returns the mask number */
    }
    return mask_num;
}

/**************************************************************************
 sensitive: none
 inputs: none
 output: TRUE if weight is valid, FALSE otherwise
 globals: none

void define UNITY
{
    for (y = 0; y < MASK_SIZE; y++)
```
load_mask(int mask_num)
    [int x, y; temp;
    long_string mask_base_string;
    enum bases mask_base = dec;
    if (mask_num < 0) /* load mask for during startup */
        // mask_num = positive mask number */
        if (!Disk = fopen(Mask[mask_num].filename, "r");) return 0; /* open stream using specified startup file name */
        else if (mask_num == TEMP_MASK_NUM)
            if (!Disk = fopen(Mask[TEMP_MASK_NUM].filename, "r");) return 0; /* open the mask file specified in an image processing instruction */
            else if (!Disk = fopen(Mask[mask_num].filename, "r");) return 0; /* open the mask file specified in an image processing instruction */
        return 0; /* return unsuccessful if invalid mask number of file */
        fscanf(Disk, "%s", mask_base_string); /* first field may specify mask base */
        if (mask_base_string[0] == 'h') mask_base = hex; /* mask base is hexadecimal if first letter of first field is 'h' */
        for (y=0; y<MASK_SIZE; y++)
            for (x=0; x<mask_base_string[0]; x++)
                if (!x && y && isalpha(mask_base_string[0])) /* if the first field wasn't a mask base specification, it was the first mask value */
                    Mask[mask_num].val[y][x] = atoi(mask_base_string); /* convert the non-existent mask base string to the first mask value */
                    else Mask[mask_num].val[y][x] = mask_input[mask_base]; /* otherwise read the mask values from the file */
        fclose(Disk);
        Mask[mask_num].valid = Mask_Loaded = TRUE; /* declare the mask as being valid */
        Mask[mask_num].modified = FALSE; /* declare the mask as being unmodified */
        return mask_num;
    *
    save_mask(int mask_num)
    [int x, y;
    file name type mask_file_name;
    inputwindow();
    if (!Disk = fopen(Mask[mask_num].filename, open_file(WINDOW, "w", Default_Window_Directory, EMPTY_STRING)))
        // opens file for writing */
        fprintf(Disk, "decimal\n"); /* prints first field specifying decimal mask values */
        for (y=0; y<MASK_SIZE; y++)
            for (x=0; x<mask_num; x++)
                fprintf(Disk, "%d %s", Mask[mask_num].val[y][x], x == MASK_SIZE-1 ? "n": ""); /* writes values to disk, skipping to next line at the end of every row */
        Mask[mask_num].modified = FALSE; /* mask is no longer modified */
        putCTRLZ(Disk);
        fclose(Disk);
    }
    *
    view_mask(int mask_num, int reflect_mask)
    [int x, y, invalid_weights;
    mask_window(); clrscr();
    if ([mask_num < 0] || [mask_num >= MAX_MASKS]) Mask[mask_num].valid return; /* return if invalid mask specified */
color_it_blue();
cprintf("window #4: \l1\n", mask_num, Base == dec ? "decimal" : "hex"); normal_video(); /* print base (hex or
decimal) */
if (reflect_mask == FALSE) /* view mask non-reflected (normal use) */
    else /* view reflected mask - this mask is used for erosions */
        if (Reflect_mask) return;
    for (x=0; x<MASK_SIZE; x++)
        for (y=0; y<MASK_SIZE; y++)
            if (input_window(Mask[mask_num].val[y][x], x == 0))
                inval_weight_count if invalid */
else /* view reflected mask - this mask is used for erosions */
    for (y=0; y<MASK_SIZE; y++)
        for (x=0; x<MASK_SIZE; x++)
            if (mask_output(Mask[mask_num].val[y][x], x == 0))
                inval_weight_count if invalid */
    }
if (inval_weight_count % 0)
    inval_weight_count % 0
else
    inval_weight_count;

void copy_mask(int dest_mask, int source_mask)
(
    int x, y;
    gotoxy(1,0); delline(); delline();
    if (Mask[source_mask].valid && source_mask>=0 && dest_mask>0) /* check if valid */
        cprintf("copy window %d to window %d ?",source_mask, dest_mask);
    if (yes_prompt(YES))
        for (y=0; y<MASK_SIZE; y++)
            for (x=0; x<MASK_SIZE; x++)
                Mask[dest_mask].val[y][x] = Mask[source_mask].val[y][x]; /* copy mask */
        Mask[dest_mask].modified = FALSE; /* specify mask as being unmodified */
    }
else if (source_mask>0 && dest_mask>0) /* if invalid, print error message */
        print_error("source window isn't valid; press any key");

void show_mask_value(int mask_num, int x2, int y2, int x1, int y1)
(
    mask_window(); gotoxy(x1+5, y1+3); /* move cursor to position of formerly highlighted mask */
    mask_output(Mask[mask_num].val[y][x], 1); /* redraw formerly highlighted mask value using normal color */
    if (x2 == 0)
        gotoxy(y2+5, y2+3); /* move cursor to position of next mask value to highlight */
    mask_output(Mask[mask_num].val[y2][x2], 2); /* highlight the mask value */

    normal_video(); menu_window();
)

void change_mask_value()
(
    name: change_mask_value
    purpose: prompts user for a new single mask value for specified mask,
    and changes that value
    inputs: mask_num number of mask to change
    x, y: x and y indices of specified mask
    output: none
    globals: Mask[mask_num].val is changed if successful
    Mask[mask_num].modified is TRUE if successful
)

void menu_window();
```c
change_mask_value(int mask_num, int x, int y)
    char new_mask_value[5];
    cputs("Enter new value: ");
    if (!is_empty(strcmp(new_mask_value, input(4, signed_integer, EMPTY_STRING))) /* inputs new mask value */
        Mask[mask_num].val[y][x] = atoi(new_mask_value); /* converts new mask value string to integer */
    show_mask_value(mask_num, x, y); /* displays the new mask value */
    Mask[mask_num].modified = TRUE; /* declares mask as having been modified */
    gotoxy(1,3); /* clears the screen */
}
else
delline(); /* deletes the prompt */
}

void
clear_all()
{
    menu_window(); gotoxy(1,1);
    mask_output(Mask[mask_num].val[0][0], 2); normal_video(); /* highlights the first mask */
    menu_window(); clear();
    color(3,7); cputs("Mask selection menu: "); normal_video();
    cputs("Select an operation to perform ");
    cputs("(q) to quit: "); xpos=wherex(); ypos=wherey();
    do
    { ch = prompt(xpos, ypos);
        switch (ch)
        {
        case UP: if (y>0) show_mask_value(mask_num, x, y--; x, y); break; /* move up one value */
        case DOWN: if (y<6) show_mask_value(mask_num, x, y++; x, y); break; /* move down one value */
        case LEFT: if (x>0) show_mask_value(mask_num, x--; x, y, y); break; /* move left one value */
        case RIGHT: if (x<6) show_mask_value(mask_num, ++x, x, y, y); break; /* move right one value */
        case ENTER: change_mask_value(mask_num, x, y); break; /* change the selected mask */
        case 'r' : Mask[mask_num].val[y][x]--; show_mask_value(mask_num, x, y, y); Mask[mask_num].modified = TRUE; break; /* decrement selected mask value */
        case 't' : Mask[mask_num].val[y][x]++; show_mask_value(mask_num, x, y, y); Mask[mask_num].modified = TRUE; break; /* increment selected mask value */
        } while (ch != 'q' && (ch != ESC));
    }
}

void
clear_all()
{
    menu_window(); gotoxy(1,1);
    mask_output(Mask[mask_num].val[0][0], 2); normal_video(); /* highlights the first mask */
    menu_window(); clear();
    color(3,7); cputs("Mask selection menu: "); normal_video();
    cputs("Select an operation to perform ");
    cputs("(q) to quit: "); xpos=wherex(); ypos=wherey();
    do
    { ch = prompt(xpos, ypos);
        switch (ch)
        {
        case UP: if (y>0) show_mask_value(mask_num, x, y--; x, y); break; /* move up one value */
        case DOWN: if (y<6) show_mask_value(mask_num, x, y++; x, y); break; /* move down one value */
        case LEFT: if (x>0) show_mask_value(mask_num, x--; x, y, y); break; /* move left one value */
        case RIGHT: if (x<6) show_mask_value(mask_num, ++x, x, y, y); break; /* move right one value */
        case ENTER: change_mask_value(mask_num, x, y); break; /* change the selected mask */
        case 'r' : Mask[mask_num].val[y][x]--; show_mask_value(mask_num, x, y, y); Mask[mask_num].modified = TRUE; break; /* decrement selected mask value */
        case 't' : Mask[mask_num].val[y][x]++; show_mask_value(mask_num, x, y, y); Mask[mask_num].modified = TRUE; break; /* increment selected mask value */
        } while (ch != 'q' && (ch != ESC));
    }

void
transpose_quad(int mask_num)
{
    char constant_string[5];
    switch (op) /* defines what is to be printed, depending on the operation */
    {
    case ADD: strcpy(operation, "addition"); break;
    case SUB: strcpy(operation, "subtraction"); break;
    case MUL: strcpy(operation, "multiplication"); break;
    case DIV: strcpy(operation, "division"); break;
    }
    cprintf("Enter constant for ": operation);
    if (is_empty(constant_string)) return; /* returns if aborted */
    constant = atoi(constant_string);
    if (constant < 0 || constant > MAX_UINT) gotoxy(1,3); /* clears the screen */
    if (constant < 0) cputs("Number must be positive. ");
    for (y=0; y<MAX_SIZE; y++)
    for (x=0; x<MAX_SIZE; x++)
    switch (op)
    {
    case ADD: Mask[mask_num].val[y][x] += constant; break;
    case SUB: Mask[mask_num].val[y][x] -= constant; break;
    case MUL: Mask[mask_num].val[y][x] *= constant; break;
    case DIV: Mask[mask_num].val[y][x] /= constant; break;
    }
    view_mask(mask_num, FALSE);
    Mask[mask_num].modified = TRUE;
}
```
void
transpose_quadrant(int mask_num)
{
int i, j, ch, left=0, top=0;
clear();
color_lt_blue(); cputs("Transpose quadrant:\n\n"); normal_video();
cputs("<Space> to select quadrant to transpose:\n\n");
cputs("<Enter> to transpose selected quadrant:\n\n");
color_green(); cputs("Enter your choice: "); normal_video();
do
{
mask_window();
for (i=left; i<left+4; i++)
for (j=top; j<top+4; j++)
{
  gotoxy(i+5, j+3);
  mask_output[Mask[mask_num].val[i][j]]; /* highlight the selected quadrant */
}
}
menu_window(); gotoxy(20,7);
ch = getch();
if (ch == SPACE)
{
mask_window();
for (i=left; i<left+4; i++)
for (j=top; j<top+4; j++)
{
  gotoxy(i+5, j+3);
  mask_output[Mask[mask_num].val[i][j]][0]; /* unhighlight the previously selected quadrant */
}
}
else if ((left == 3) && (top == 0))
{
  change_mask(add,mask_num);
  change_mask(sub,mask_num);
  change_mask(div,mask_num);
  change_mask(mult,mask_num);
  if ((left == 0) && (top == 0))
    change_mask(trans,mask_num);
  else if ((left == 3) && (top == 3))
    change_mask(trans,mask_num);
  else if ((left == 0) && (top == 3))
    change_mask(trans,mask_num);
}
while (ch != 'q')
{
  ch = getch();
  cputs("\n\n");
}
}

int
modify_mask(int mask_num)
{
enum mask_ops op;
do
{
clear();
color_lt_blue(); cprintf("Modify window #id: ", mask_num); normal_video();
cputs("\n\n[c] choose one value\n\n[a] add constant\n\n[s] subtract constant\n\n[m] multiply by constant\n\n[d] divide by constant\n\n[t] transpose quadrant\n\n[q] quit modify window\n\n");
color_green(); cputs("\nEnter your choice: "); normal_video();
ch = getch();
cputs("\n\n");
switch (ch)
{
case 'c'
  : select_mask_value(mask_num); break;
case 'a'
  : change_mask(add,mask_num); break;
case 's'
  : change_mask(sub,mask_num); break;
case 'm'
  : change_mask(mult,mask_num); break;
case 'd'
  : change_mask(div,mask_num); break;
case 't'
  : transpose_quadrant(mask_num); break;
case LEFT:
  : if (!mask_num) goto LEFT; /* view previous mask */;
  else if (Mask[mask_num].valid)
    goto LEFT; /* repeat until a valid mask is found */;
  view_mask(mask_num, FALSE); break;
case RIGHT:
  : do if (!mask_num) goto RIGHT; /* view next mask */;
  else if (Mask[mask_num].valid)
    goto RIGHT; /* repeat until a valid mask is found */;
  view_mask(mask_num, FALSE); break;
}
while (ch != 'q');
}
return mask_num;

int delete_mask(int mask_num)
{
  int ch, temp_mask = mask_num;
  if (Mask[mask_num].modified)
  {
    switch (ch)
    {
      case 'b': Base = Base; view_mask(current_mask, FALSE); break; /* toggle base (hex/decimal) */
      case 'l': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", 1)); /* load mask 1 if none loaded... */
      case 's': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", current_mask)); /* otherwise load current mask */
      case 'm': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", current_mask)); /* otherwise load current mask */
      case 'd': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", current_mask)); /* otherwise load current mask */
      case 'g': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", current_mask)); /* otherwise load current mask */
      case 'c': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", current_mask)); /* otherwise load current mask */
      case 't': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", current_mask)); /* otherwise load current mask */
      case 'w': if (current_mask == 0) temp_mask = load_mask(get_mask_num("", current_mask)); /* otherwise load current mask */
    }
  }
  return temp_mask;
}

int main()
{
  int mask_num;
  return 0;
}
if (Mask[temp].valid && temp_mask > 0) current_mask = temp_mask;
view_mask(current_mask, FALSE); break;
case 'r': if (current_mask) view_mask(current_mask, TRUE); break; /* reflect mask (used for erosions) */
case LEFT: do if (Mask_Loaded && (--current_mask == 0)) current_mask = MAX_MASKS-1; /* view previous mask */
while (!Mask[current_mask].valid);
view_mask(current_mask, FALSE); break;
case RIGHT: do if (Mask_Loaded && (++current_mask == MAX_MASKS)) current_mask = 1; /* view next mask */
while (!Mask[current_mask].valid);
view_mask(current_mask, FALSE); break;
} while (ch != 'q');
return(current_mask);
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cfiles\define.c"
#include <process.h>
#include "cfiles\typedef.c"
int
extern intl6 Address_Register;
int8
extern int Bits;
extern file_name_type
extern header_type
extern intl6
extern unsigned char
FILE
file_name_type
Temp_File_Name;
define
#define
#define
#define
in_file,
FILE
*****
int
inputs:
*
purpose: reads
*
*
name: read_ips_header
output:
*
globals:

void
* globals:
purpose contains
name
program.

int
return 32;
}

/****************************
* name: read_ips_header
* purpose: reads the header information from an image file
* inputs: none
* output: TRUE if successful, FALSE otherwise
* globals: Header
*****************************************************************************/
int
read_ips_header()
{
unsigned char val_low, val_high;
menu_window(); clrscr();
Header.title[0] = get(Disk); Header.title[1] = get(Disk); /* get header title information */
if (Header.title[0] == 'I' && Header.title[1] == 'H')
{ cputs("ERROR reading IPS file!"); return FALSE; /* signal error if the header title is invalid */
Header.comment_size = fread_chars_to_int(); /* read the header comment size */
Header.x_size = fread_chars_to_int(); /* read the width of the image */
Header.y_size = fread_chars_to_int(); /* read the height of the image */
Header.x_origin = fread_chars_to_int(); /* read the X origin of the image */
Header.y_origin = fread_chars_to_int(); /* read the Y origin of the image */
Header.format = fread_chars_to_int(); /* read the format information of the image */
for (i=0; i<63; i++)
get(Disk); /* read reserved data */
for (i=0; i<Header.comment_size; i++)
{ Header.comment[i] = get(Disk); /* read the comment */
cprintf("image size: %d x %d\n", Header.x_size, Header.y_size); /* print the image size */
cprintf("upper left corner: %d, %d\n", Header.x_origin, Header.y_origin); /* print the image origin */
if (Header.format == 2) Bits = 9;
else Bits = 8;
cprintf("bits: %d\n", Bits);
return TRUE;
}

/****************************************************************************
* name: print_header_info
* purpose: prints the header information to the screen
* inputs: no_print_comment: print comment entry prompt if TRUE
* output: none
* globals: none
*****************************************************************************/
void
print_header_info(int no_print_comment)
{ menu_window(); clrscr();
cprintf("image size: %d x %d", Header.x_size, Header.y_size); /* prints the image size */
cprintf("origin: (%d,%d)\n", Header.x_origin, Header.y_origin); /* prints the image origin */
if (no_print_comment) print_comment(-1, ""); /* print header comment entry prompt */
else if (Header.comment_size == 0) cputs("no comment"); /* there is no header comment */
else print_comment(Header.comment_size, Header.comment); /* print the header comment */
}
void write_ips_header() {
    /*************************************************************************/
    void change_header_info();
    /*************************************************************************/

    int i, loop = TRUE;
    unsigned char val_low, val_high;

    Header.x_size = IMAGE_SIZE; /* define image size */
    Header.y_size = IMAGE_SIZE;
    Header.x_origin = 0; /* define image origin */
    Header.y_origin = 0;
    Header.format = 0; /* define header format */

    while (loop) {
        print_header_info(FALSE); /* prints the header information to the screen */
        input_window();
        cpwts("Make any changes to header info?");
        switch(yes_prompt(NO))
        { /* case TRUE : change_header_info(): break: */
            case ABORT: return FALSE; /* aborts */
            default : loop = FALSE; /* loops until all header information is satisfactory */
        }
    }

    putc('I', Disk); putc('M', Disk);
    fwrite_int_to_chars(Header.comment_size);
    fwrite_int_to_chars(Header.x_size);
    fwrite_int_to_chars(Header.y_size);
    fwrite_int_to_chars(Header.x_origin);
    fwrite_int_to_chars(Header.y_origin);
    fwrite_int_to_chars(Header.format);
    for (i=0; i<Header.x_size; i++)
        putc0(Disk); /* write reserved data */
    for (i=0; i<Header.comment_size; i++)
        putc(Header.comment[i], Disk); /* write comment */
    return TRUE;
}

/*************************************************************************/
name: copy buffer to file
purpose: copies a partial image buffer to a file
inputs: bits: number of bits
        chop: truncate to 8 bits if TRUE, otherwise set to 0 if negative
output: none
globals: none
*************************************************************************/
void copy_buffer_to_file(int bits, int chop)
{ /* if (bits=8) */ /* use 8-bit format */
    if (chop) /* chop off msb to get 8 bits */
        for (i=0; i<BUFFER_SIZE; i++) fputc(Buffer[i], Disk); /* copy buffer to disk */
    else /* set =0 values to 0 to get 8 bits */
        for (i=0; i<BUFFER_SIZE; i++)
            if (Buffer[i]>255) fputc(0, Disk);
            else fputc(Buffer[i], Disk);
    else /* use 16 bits for 9-bit format */
for (i=0; i<BUFFER_SIZE; i++)
    ifputc(Buffer[i] > 255, Disk); /* copy byte to file */
    ifputc(Buffer[i] & 0xff, Disk); /* copy not to file */
}

void
copy_file_to_buffer(int bits, int initialize)
{static int x, y;
static int ymax;
unsigned char msb, lsbyte;
if (initialize == 0) y = 0; /* initializes Y value to zero, used when starting a new copy operation */
ymax = BUFFER_SIZE / 512 + y; /* defines maximum Y value */
if (bits == 8) /* read 8-bit format */
    for (; y < ymax; y++)
        for (x = 0; x < IMAGE_SIZE; x++, i++)
            if (Header.x_size <= x) Buffer[i] = 0; /* read 0s if between the bounds of a non-standard */
            /* size image and a standard size image */
    else ifputc(Disk); /* normal read mode */
    else if (y < ymax)
        for (x = 0; x < IMAGE_SIZE; x++)
            if (Header.x_size <= x) Buffer[i] = 0; /* read 0s if between the bounds of a non-standard */
            /* size image and a standard size image */
    else /* normal read mode */
        (lsbyte = getc(Disk)); /* read lsbyte */
        msb = getc(Disk); /* read msb */
        Buffer[i] = lsbyte + ((msb == 0) ? 0 : 256); /* combine lsbyte and MSB */
}

void

copy_buffer_to_fr_grab(int *xstart, int *ystart)
{register int l, x = *xstart, y = *ystart, xend = *xstart + 512;
    for (i = 0; i < BUFFER_SIZE; i++)
        if (x == xend) /* if x reached the maximum value, reset x and increment y */
            {x = *xstart;
             y++;
        }
        Pixel_Value = Buffer[i]; /* get the pixel value from the image buffer */
        Address_Register = convert_to_address_register(x, y); /* converts X and Y coordinates to address register used by the frame grabber */
        Page_Number = convert_to_page_number(y); /* converts Y coordinate to page number used by the frame grabber */
        plot_pixel(); /* plots the pixel to the frame grabber */
}

void

copy_fr_grab_to_buffer(int *xstart, int *ystart)
{register int l, x = *xstart, y = *ystart, xend = *xstart + 512;
    for (i = 0; i < BUFFER_SIZE; i++)
        if (x == xend) /* if x reached the maximum value, reset x and increment y */
            {x = *xstart;
             y++;
        }
        Address_Register = convert_to_address_register(x, y); /* converts X and Y coordinates to address register used by the frame grabber */
        Page_Number = convert_to_page_number(y); /* converts Y coordinate to page number used by the frame grabber */
        get_pixel(); /* gets the pixel value from the frame grabber */
        Buffer[i] = Pixel_Value; /* copies the pixel value to the buffer */
}

void

copy_ext_mem_to_file
{register int l, x = *xstart, y = *ystart, xend = *xstart + 512;
    for (i = 0; i < BUFFER_SIZE; i++)
        if (x == xend) /* if x reached the maximum value, reset x and increment y */
            {x = *xstart;
             y++;
        }
        Address_Register = convert_to_address_register(x, y); /* converts X and Y coordinates to address register used by the frame grabber */
        Page_Number = convert_to_page_number(y); /* converts Y coordinate to page number used by the frame grabber */
        get_pixel(); /* gets the pixel value from the frame grabber */
        Buffer[i] = Pixel_Value; /* copies the pixel value to the buffer */
}
void copy_ext_mem_to_file(int mem_num, int base_offset, int chop);  
if (write_ips_header()) return;
Ext_Mem_Base_High = 0x10 + mem_num * 0x08;  
/ * defines the extended memory base address */
for (base_offset=0; base_offset < BUFFERS_PER_IMAGE2; base_offset++, Ext_Mem_Base_High++)
for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x100)
{putc('.', Disk);  
// prints periods to screen as it's copying */
copy_file_to_buffer(bits, initialize++);  
/* first copy file to buffer */
copy_buffer_to_ext_mem();  
/* then copy buffer to extended memory */
}
fclose(Disk);
cprintf("\n");
}

/********************************************
* name: copy_file_to_ext_mem
* purpose: copies an image file to the extended memory
* inputs: mem_num: extended memory buffer number
*          bits: number of bits
* output: none
* globals: Ext_Mem_Base_High, Ext_Mem_Base_Mid
*********************************************/
void copy_file_to_ext_mem(int mem_num, int bits)
{register int base_offset;
int initialize=0;
Ext_Mem_Base_High = 0x10 + mem_num * 0x08;  
/ * defines the extended memory base address */
for (base_offset=0; base_offset < BUFFERS_PER_IMAGE2; base_offset++, Ext_Mem_Base_High++)
for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x100)
{putc('.', Disk);  
// prints periods to screen as it's copying */
copy_file_to_buffer(bits, initialize++);  
/* first copy file to buffer */
copy_buffer_to_ext_mem();  
/* then copy buffer to extended memory */
}
fclose(Disk);
cprintf("\n");
}

/********************************************
* name: copy_file_to_board_mem
* purpose: copies an image file to the board memory
* inputs: mem_num: board memory number
*          bits: number of bits
* output: none
* globals: Ext_Mem_Base_High, Ext_Mem_Base_Mid
*********************************************/
void copy_file_to_board_mem(int mem_num, int bits)
{register int base_offset;
int initialize=0;
Ext_Mem_Base_High = Board_Loc_High;  
/ * defines the extended memory base address */
register_write(0XE, mem_num * 0x08);  
/ * set up the board's registers for a memory write operation */
register_write(0xF, 0x60 : (Board_Loc_High >> 3));
for (base_offset=0; base_offset < BUFFERS_PER_IMAGE2; base_offset++, Ext_Mem_Base_High++)
for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x100)
{putc('.', Disk);  
// prints periods to screen as it's copying */
copy_file_to_buffer(bits, initialize++);  
/* first copy file to buffer */
copy_buffer_to_ext_mem();  
/* then copy buffer to board memory */
}
fclose(Disk);
cprintf("\n");
}

/********************************************
* name: copy_ext_mem_to_fr_grab
* purpose: copies an extended memory to the frame grabber
* inputs: mem_num: extended memory buffer number
*          frame_grab_num: frame grabber buffer number
* output: none
* globals: Ext_Mem_Base_High, Ext_Mem_Base_Mid
*********************************************/
void copy_ext_mem_to_fr_grab(int mem_num, int frame_grab_num)
{int base_offset, xstart = Fr_Grab.x_origin[frame_grab_num], ystart = Fr_Grab.y_origin(frame_grab_num);
Ext_Mem_Base_High = 0x10 + mem_num * 0x08;  
/ * defines the extended memory base address */
for (base_offset=0; base_offset < BUFFERS_PER_IMAGE2; base_offset++, Ext_Mem_Base_High++)
for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x100)
{putc('.', Disk);  
// prints periods to screen as it's copying */
copy_ext_mem_to_buffer();  
/* first copy extended memory to buffer */
copy_buffer_to_ext_mem();  
/* then copy buffer to the frame grabber */
ystart += 0x16;  
/* increment y by the Y dimension of the buffer */
}
cprintf("\n");
}
```c
void copy_wedge_to_fr_grab(int fr_grab_num) {
    int i, base_offset, xstart = Fr_Grab.x_origin[fr_grab_num], ystart = Fr_Grab.y_origin[fr_grab_num];
    Ext_Mem_Base_High = 0x10 + mem_num * 0x08; /* defines the extended memory base address */
    Ext_Mem_Base_Mid = 0x08;
    for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++)
        for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x40)
            for (Ext_Mem_Base_High=0; Ext_Mem_Base_High < 0x10; Ext_Mem_Base_High += 0x100)
                putcht('.'); /* prints periods to screen as it's copying */
    copy_buffer_to_ext_mem(); /* first copy frame grabber image to buffer */
    ystart += 16; /* increment y by the Y dimension of the buffer */
    puts("\n");
}

void copy_ext_mem_to_ext_mem(int source_mem_num, int dest_mem_num) {
    int base_offset, source_ext_mem_base_high = 0x10 + source_mem_num * 0x08, dest_ext_mem_base_high = 0x10 + dest_mem_num * 0x08;
    for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, source_ext_mem_base_high++,
        dest_ext_mem_base_high++)
        for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x40)
            putcht('.'); /* prints periods to screen as it's copying */
    copy_buffer_to_ext_mem(); /* first copy extended memory to buffer */
    Ext_Mem_Base_High = dest_ext_mem_base_high; /* defines the extended memory base address */
    copy_buffer_to_ext_mem(); /* then copy buffer to extended memory */
    puts("\n");
}

void copy_wedge_to_ext_mem(int dest_mem_num) {
    int i, base_offset, dest_ext_mem_base_high = 0x10 + dest_mem_num * 0x08;
    for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, dest_ext_mem_base_high++)
        for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x40)
            putcht('.'); /* prints periods to screen as it's copying */
    copy_buffer_to_ext_mem(); /* copies buffer to extended memory */
    puts("\n");
}

void copy_wedge_to_board_mem(int mem_num) {
    int base_offset, mem_num = board_mem_num;
    for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++)
        for (Ext_Mem_Base_Mid=0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid += 0x40)
            putcht('.'); /* prints periods to screen as it's copying */
    copy_to_board_mem(); /* copies buffer to extended memory */
    puts("\n");
}
```

void copy_fr_grab_to_board_mem(int fr_grab_num, int mem_num)
{
  int base_offset, xstart = Fr_Grab.x_origin[fr_grab_num], ystart = Fr_Grab.y_origin[fr_grab_num];
  Ext_Mem_Base_High = Board_Loc_High; /* defines the extended memory base address */
  register_write(0xE, mem_num * 0x08); /* set up the board's registers for a memory write operation */
  register_write(0xF, 0x60 | (Board_Loc_High >> 3));
  for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, Ext_Mem_Base_High++)
  
    for (fr_grab_num=0; fr_grab_num < BUFFERS_PER_IMAGE/2; fr_grab_num++)
    
      /* prints periods to screen as it's copying */
      
      printf("\r\n");

    for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, Ext_Mem_Base_High++)
      printf("\r\n");

  cprintf("\r\n");
}

/*****************************************************/

void copy_ext_mem_to_board_mem(int source_mem_num, int dest_mem_num)
{
  register_write(0xE, dest_mem_num * 0x08); /* set up the board's registers for a memory write operation */
  register_write(0xF, 0x40 | (Board_Loc_High >> 3));
  for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, source_mem_num, dest_mem_num++)
    printf("\r\n");

  cprintf("\r\n");

  */

}

/*****************************************************/

void copy_wedge_to_board_mem(int source_mem_num, int dest_mem_num)
{
  register_write(0xE, dest_mem_num * 0x08); /* set up the board's registers for a memory write operation */
  register_write(0xF, 0x40 | (Board_Loc_High >> 3));
  for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, source_mem_num, dest_mem_num++)
    printf("\r\n");

  cprintf("\r\n");

  
}

/*****************************************************/

void copy_board_mem_to_ext_mem(int mem_num, int fr_grab_num)
{
  register_write(0xD, reg_val << mem_num); /* set up the board's registers for a memory read operation */
  register_write(0xF, 0x40 | (Board_Loc_High >> 3));
  for (Ext_Mem_Base_High = Board_Loc_High; /* defines the extended memory base address */
    register_write(0xD, mem_num * 0x08); /* set up the board's registers for a memory read operation */
    register_write(0xF, 0x40 | (Board_Loc_High >> 3));
    for (base_offset=0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, Ext_Mem_Base_High++)
      printf("\r\n");

  cprintf("\r\n");

  
}
/* global variables */

void copy_board_mem_to_ext_mem(int source_mem_num, int dest_mem_num)
{
    int reg_val = 1, base_offset, source_ext_mem_base_high = Board_Loc_High, dest_ext_mem_base_high = 0x10 + dest_mem_num;
    register_write(0x0D, reg_val << source_mem_num); /* set up the board's registers for a memory read operation */
    register_write(0x0E, source_mem_num << 0x08);
    register_write(0x0F, 0x10 | (Board_Loc_High >> 3));
    for (base_offset = 0; base_offset < BUFFERS_PER_IMAGE/2; base_offset++, source_ext_mem_base_high++)
    {
        for (Ext_Mem_Base_Mid = 0; Ext_Mem_Base_Mid < 0x100; Ext_Mem_Base_Mid = 0x00)
        {
            if (Mem_Select_Reg == 0x03 + source_mem_num << source_mem_num);
                /* sets up the memory selection register */
            mem_select_reg = (0x03 << source_mem_num << source_mem_num);
                /* sets up the memory selection register */
            register_write(0x0F, 0x10 | (Board_Loc_High >> 3));
                /* set up the board's registers for the copy operation */
            register_write(0x05, 0x00);
                /* inputs: source_mem_num: board memory number (source) */
            register_write(0x06, 0x00);
                /* dest_mem_num: board memory number (destination) */
            register_write(0x07, 0x00);
            register_write(0x08, 0x00);
                /* output: none */
            register_write(0x0C, mem_select_reg);
                /* globals: none */
            register_write(0x0E, 0x20);
            register_write(0x0B, 0x00);
        }
        /* purpose: copies one board memory to another one */
        /* inputs: source_mem_num: board memory number (source) */
        /* dest_mem_num: board memory number (destination) */
        /* output: none */
        /* globals: none */
    }
}

void copy_board_mem_to_board_mem(int source_mem_num, int dest_mem_num)
{
    int mem_select_reg = 0xFF;
    mem_select_reg = (0x03 << source_mem_num << source_mem_num);
        /* sets up the memory selection register */
    mem_select_reg = (0x01 << dest_mem_num << dest_mem_num);
        /* set up the board's registers for the copy operation */
    register_write(0x05, 0x00);
        /* purpose: plots pixel to frame grabber at specified location */
    register_write(0x06, 0x00);
        /* inputs: x,y: X and Y coordinates of pixel to plot */
    register_write(0x07, 0x00);
        /* p: pixel value to plot */
    register_write(0x08, 0x00);
        /* output: none */
    register_write(0x0C, mem_select_reg);
        /* globals: none */
    register_write(0x0E, 0x20);
    register_write(0x0B, 0x00);
}

void plot(int x, int y, int p)
{
    int Address_Register = convert_to_address_register(x, y);
        /* converts X and Y coordinates to address register used by the frame grabber */
    int Page_Number = convert_to_page_number(y);
        /* converts Y coordinate to page number used by the frame grabber */
    Pixel_Value = p;
        /* sets the pixel value */
    plot_pixel(); /* calls assembly language plot pixel function, with location and pixel value passed in as global variables */
}

int get(int x, int y)
{
    int Address_Register = convert_to_address_register(x, y);
        /* converts X and Y coordinates to address register used by the frame grabber */
    int Page_Number = convert_to_page_number(y);
        /* converts Y coordinate to page number used by the frame grabber */
    get_pixel(); /* calls assembly language get pixel function, with location and pixel value passed in and out as global variables */
    return Pixel_Value;
}

void print_image_transfer_menu(int image)
{
    int i;
    menu_window(); clrscr();
    color.lt_blue(); sputs("Image transfer operations:\n\n"; normal_video());
    sputs"(0)-(3) board memories\n\n";
}

B-35
if (Ext_Nom_Count) cprintf(" (e) extended memories (0-4d)\n", Ext_Nom_Count - 1); /* prints option if extended memory has been detected, */
    / * includes number of extended memories detected */
if (Fr_Grab_buffers) cprintf(" (f) frame grabber (0-4d):\n", Fr_Grab_buffers - 1); /* prints option if frame-
grabber buffers are available, */
    / * includes number of available frame grabbers */

    cprintf(" (g) disk:\n");
if (image == SOURCE) cprintf(" (h) wedge\n"); /* prints option only if prompting for source image */
    cprintf(" (q) quit image transfer operations");

void print_fr_grab_number
    (int frame, int frame_to_highlight)
    { /* (frame = 0\n = frame_to_highlight) */
        cprintf(" k\n", frame); /* print the frame grabber number */
        normal_video();
    }

void print_fr_grab_info
    (int frame, int frame_to_highlight)
    { /* frame_to_highlight */
        for (y=0; y<2; y++)
            for (x=0; x<4; x++)
                buf_num[y][x] = '-'; /* initialize all 8 frame grabber buffers to '-' which indicates an inactive frame grabber */
        for (i=0; i<Fr_Grab_buffers; i++)
            buf_num[Fr_Grab_y_origin[i]/IMAGE_SIZE][Fr_Grab_x_origin[i]/IMAGE_SIZE] = i + '0'; /* number the frame grabber numbers according to values */
            / * specified in the frame grabber configuration file */

cprintf("#\n"); /* print the top border */
    for (y=0; y<2; y++)
        for (x=0; x<4; x++)
            buf_num[y][x] = '-'; /* initialize all 8 frame grabber buffers to '-' which indicates an inactive frame grabber */

    for (i=0; i<Fr_Grab_buffers; i++)
        buf_num[Fr_Grab_y_origin[i]/IMAGE_SIZE][Fr_Grab_x_origin[i]/IMAGE_SIZE] = i + '0'; /* number the frame grabber numbers according to values */

    cprintf("#\n"); /* print the middle border */
    else if (y==1 && x==3)
        cprintf("#\n"); /* print the bottom border */
    else if (x==2)
        cprintf("#\n"); /* print the side border */
}

void get_frame_grabber_numbers(int include_all, int use_default)
    { /* include all used only when clearing frame grabbers, adds a prompt to clear all frame grabbers */
        use_default = use_default frame grabber number */
        for (i=0; i<Fr_Grab_buffers; i++)
            buf_num[Fr_Grab_y_origin[i]/IMAGE_SIZE][Fr_Grab_x_origin[i]/IMAGE_SIZE] = i + '0'; /* number the frame grabber numbers according to values */
            / * specified in the frame grabber configuration file */

cprintf("#\n"); /* print the top border */
    for (y=0; y<2; y++)
        for (x=0; x<4; x++)
            buf_num[y][x] = '-'; /* initialize all 8 frame grabber buffers to '-' which indicates an inactive frame grabber */

    for (i=0; i<Fr_Grab_buffers; i++)
        buf_num[Fr_Grab_y_origin[i]/IMAGE_SIZE][Fr_Grab_x_origin[i]/IMAGE_SIZE] = i + '0'; /* number the frame grabber numbers according to values */

    cprintf("#\n"); /* print the middle border */
    else if (y==1 && x==3)
        cprintf("#\n"); /* print the bottom border */
    else if (x==2)
        cprintf("#\n"); /* print the side border */
    }
buff_num
were
selected
*/
Default_Image_Directory,
EMPTY_STRING)
memory number +
number
memory
information */
*
*
copy_file_to_fr_grab(int
buf)
name *
purpose: prompts for an extended memory number *
inputs: none *
output: selected extended memory number if valid, -999 if invalid *
globals: none *
*****************************************/
int get_ext_mem_num()
{char temp_string[13];
color_green();
cprintf("\n\nEnter extended memory number (0-4d): ", Ext_Mem_Count - 1); /* prints the prompt, indicating available extended memories */
normal_video();
strcpy(temp_string, input(2, integer, EMPTY_STRING)); /* places user input in temp_string */
if (is_empty(temp_string)) return -999; /* returns -999 if invalid extended memory number was selected */
else return atoi(temp_string); /* otherwise returns the extended memory value selected */
}
////////////////////////////////////////////////////////////////="/**************************************************t****t***i*l**li,il***i*
name: get_image_to_transfer *
purpose: prompts for the image to copy *
inputs: image: SOURCE if source image, DESTINATION if destination image *
output: 0, 1, 2, or 3 if on-board memories *
  8 if wedge *
  9 if file *
  10 + frame grabber number if frame grabber *
  100 + extended memory number if extended memory *
  ~ if aborted or invalid *
globals: none *
*********************************************************************************
int get_image_to_transfer(int image)
{int ch, ch2, image_to_transfer = -1;
color_green();
cprintf("\n\nEnter choice for the image: ", image == SOURCE ? "source" : "destination"); normal_video();
/*prints prompt, specifies source or destination */
ch = input_char("0123defwq");
if (ch == ENTER) || (ch == ESCI) return -1; /* returns unsuccessful value if aborted */
switch (ch)
{case 'f': if (Fr_Grab.buffers)
{for (image = SOURCE)
{ch2 = image_to_transfer = get_frame_grab_num(FALSE, FALSE) + 10; /* image to copy is frame grabber number + 10 if source image */
else image_to_transfer = 10; /* returns 10 if destination image, actual number gets entered later */
break;
}
if (Ext_Mem_Count) image_to_transfer = get_ext_mem_num() + 100; break; /* image to copy is extended memory number + 100 */

d = image_to_transfer = 9; /* image to copy is 9, indicating disk */
if (is_empty(temp_file_Name, open_file(image, image == SOURCE ? "rb" : "wb", Default_Image_Directory, EMPTY_STRING)))
return -1; /* returns unsuccessful value if not able to read or write file */
break;

if (image == SOURCE) image_to_transfer = 8; break; /* image to copy is 8, indicating a wedge */
case '0';
case '1';
case '2';
image_to_transfer = ch = '0'; break; /* image to copy is 0, 1, 2, or 3, indicating the on-board memories */
}
return image_to_transfer;
}
/*************************************************************************
name: copy_file_to_fr_grab *
purpose: copies an image file to the frame grabber *
inputs: buf: frame grabber buffer number *
output: none *
globals: Address_Register, Page_Number, Pixel_Value *
*************************************************************************/
void copy_file_to_fr_grab(int buf)
```c
register int n, y;
int x_max  = Header.x_size + Fr_Grab.x_origin(buf); /* defines maximum X coordinate */
int y_max  = Header.y_size + Fr_Grab.y_origin(buf); /* defines maximum Y coordinate */
for (y = Fr_Grab.y_origin(buf); y < y_max; y++) { /* loops through range of Y coordinates */
  for (x = Fr_Grab.x_origin(buf); x < x_max; x++) { /* loops through range of X coordinates */
    if ((y + y_offset) == y_dest) { /* puts character through range of X coordinates */
      unsigned int xdest = image[ysource][x]; /* displays period on screen as it's copying */
      unsigned int ydest = Address_Register->convert_to_address_register(xdest, ydest); /* converts X and Y coordinates to address register used by the frame grabber */
      Page_Number = convert_to_page_number(y); /* converts Y coordinate to page number used by the frame grabber */
      plot_pixel(); /* plots pixel on frame grabber */
    }
    ydest = Address_Register->convert_to_address_register(xdest, ydest); /* writes pixel value to the image file */
    fclose(Disk); /* close the image file */
  }
  fclose(Disk); /* close the image file */
}
```
```c
int is_file(int image)
{return (image == 9); /* returns TRUE if image is a file */}

int is_fr_grab(int image)
{return (image == 10) && (image < 10 + Fr_Grab.buffers); /* returns TRUE if image is a frame grabber buffer */}

int is_ext_mem(int image)
{return ((image >= 100) && (image < 100 + Ext_Mem_Count)); /* returns TRUE if image is an extended memory buffer */}

/*********************************************************************************

/* name: copy_file_to_file */
/* purpose: prints what type of image has been selected */
/* inputs: image: image number (using format defined in */
/* get_image_to_transfer function) */
/* file_name: file name if image is a file */
/* output: none */
/* globals: none */
*/

void
print_image_type(int image, file_name_type file_name)
{color_yellow();
 if (is_file(image)) printf("%s", file_name);
 else if (is_wedge(image)) cputs("wedge");
 else if (is_board_mem(image)) printf("board memory Id", image);
 else if (is_fr_grab(image)) printf("frame grabber buffer Id", image - 10);
 else if (is_ext_mem(image)) printf("extended memory Id", image - 100);
 normal_video();
}

/*********************************************************************************

/* name: clear_fr_grab */
/* purpose: clears a frame grabber buffer (after a yes/no prompt), */
/* or all if specified */
/* inputs: fr_grab_num: number of the buffer to clear, MAX_FR_GRAB_NUMBERS */
/* if clearing all */
/* output: none */
/* globals: Address_Register, Page_Number, Pixel_Value */
*/

void
clear_fr_grab(int fr_grab_num)
{register int x, y;
 int x_max, y_max;
 if (fr_grab_num < 0) return; /* returns if invalid frame grabber number specified */
 if (fr_grab_num == MAX_FR_GRAB_NUMBERS) cputs("\n\nclear all buffers");
 else cprintf("\n\nclear frame grabber Id ", fr_grab_num);
 if (yes_prompt(YES))
 cprintf("\n\nclearing ...");
 cputs("clearing...");
 if (fr_grab_num == MAX_FR_GRAB_NUMBERS) spawnl(P_WAIT, "/fr_grab/clear_fb", NULL); /* issues frame grabber clear command */
 else /* if not clearing the whole frame grabber... */
 for (y = Fr_Grab.y_origin[fr_grab_num]; y < max; y++) /* loops through range of Y coordinates */
 for (x = Fr_Grab.x_origin[fr_grab_num]; x < max; x++) /* loops through range of X coordinates */
 plot_pixel(); /* clears the pixels by plotting values of 0 */

clear();

```

void view_image(int frame, int next)
{
  if (next) frame++; /* if incrementing, add 1 */
  else frame--; /* else subtract 1 */
  if (frame < 0) return Fr_Grab_buffers - 1; /* return the maximum frame grabber buffer number if wrapped below 0 */
  else if (frame > Fr_Grab_buffers - 1) return 0; /* return 0 if wrapped above the maximum frame grabber buffer number */
  return frame; /* otherwise return the modified frame grabber buffer number */
}

void view_image(int frame)
{
  int ch, x, y, xinit, yinit, xpos, ypos, newframe = TRUE;
  if (frame < 0) return;
  do
    /*
     * name: view_image
     * purpose: views part of a frame grabber image in a viewport on the screen
     * inputs: number of frame grabber buffer to view
     * output: none
     * globals: Address_Register, Page_Number
     *------------------------------------------------------------------------*/
    if (newframe)
      /*
       * name: next_frame
       * purpose: increases or decreases the frame grabber buffer number
       * inputs: frame: current frame grabber buffer number
       * output: the incremented or decremented frame grabber buffer number
       * globals: none
       *------------------------------------------------------------------------*/
      newframe = FALSE;
    gotoxy(6,2);
    for (x=xinit; x<=xinit+6; x++)
      for (y=yinit; y<=yinit+8; y++)
        mask_output(x, y); /* prints X locations to screen */
    cprintf("Enter your choice: ");
    mask_window();
    cputs("Move viewport up ten pixels");
    for (x=xinit; x<=xinit+6; x++)
      for (y=yinit; y<=yinit+8; y++)
        mask_output(x, y); /* prints Y locations to screen */
    Page_Number = convert_to_page_number(y); /* converts Y coordinate to page number used by the frame grabber */
    x = xinit;
    y = yinit;
    (Address_Register = convert_to_address_register(x,y)); /* converts X and Y coordinates to address register used by the frame grabber */
    x = xinit;
    y = yinit;
    get_pixel();
    mask_output(Pixel_Value, (x = xinit + 5) && (y = yinit+6)); /* plots the pixel values to the mask window on the screen */
  while (ch != 'q' && ch != ESC);
  /*
   */
}

/*
* name: image_transfer
* purpose: copies an image from one location to another
* inputs: none
* output: none
* globals: none
* *------------------------------------------------------------------------*/

void image_transfer()
{
  int source_image, dest_image, use_default_fr_grab_num = FALSE;
  char source_file[128], dest_file[128];
  if ((source_image = get_image_to_transfer(SOURCE)) < 0) return; /* returns if invalid */
  /*
   */
  /*
   */
if (is_file(source_image)) strcpy(source_file, Temp_File_Name);  /* defines the source file name if the source is a file */
input_window();
cputs("source image: ");
print_image_type(source_image, source_file);  /* prints what type of image the source is */
print_image_transfer_menu(DESTINATION);
if ((dest_image = get_image_to_transfer(DESTINATION)) < 0)
{input_window(); clrscr(); return;  /* returns if invalid or aborted */}
if (is_file(dest_image)) strcpy(dest_file, Temp_File_Name);  /* defines the destination file name if the destination is a file */
if (is_fr_grab(dest_image))
{if(is_file(source_image))
 {read_ips_header();  /* reads the image file header information */
 use_default_fr_grab_num = TRUE;
} else clrscr();
if ((dest_image = get_frame_grab_num(FALSE, use_default_fr_grab_num) + 10) < 0)
{input_window(); clrscr(); return;  /* returns if invalid or aborted */
}
if ((is_ext_mem(dest_image) || is_board_mem(dest_image)) && is_file(source_image))
{read_ips_header();  /* reads the image file header information */
input_window(); cputs("copying image...");
print_image_type(dest_image, dest_file); cputs(" to ");
print_image_type(source_image, source_file); cputs(" from ");
clrscr();
cputs("copying image...");
if(is_file(source_image))
{if(is_file(dest_image))
 {copy_file_to_file(source_file, dest_file);
 if(is_fr_grab(dest_image)) copy_file_to_fr_grab(dest_image = 0);
 if(is_ext_mem(dest_image)) copy_file_to_ext_mem(dest_image = TRUE);
 if(is_board_mem(dest_image)) copy_file_to_board_mem(dest_image, Bits);
}
if(is_fr_grab(source_image))
{if(is_file(dest_image))
 {copy_fr_grab_to_file(source_image = 10);
 if(is_fr_grab(dest_image)) copy_fr_grab_to_fr_grab(source_image, dest_image = 0);
 if(is_ext_mem(dest_image)) copy_fr_grab_to_ext_mem(source_image, dest_image = 100, Bits);
 if(is_board_mem(dest_image)) copy_fr_grab_to_board_mem(source_image, dest_image, Bits);
}
if(is_ext_mem(source_image))
{if(is_file(source_image))
 {copy_ext_mem_to_file(source_image, 0, FALSE);
 if(is_fr_grab(source_image)) copy_ext_mem_to_fr_grab(source_image, dest_image = 0);
 if(is_ext_mem(dest_image)) copy_ext_mem_to_ext_mem(source_image, dest_image = 100, Bits);
 if(is_board_mem(dest_image)) copy_ext_mem_to_board_mem(source_image, dest_image, Bits);
}
if(is_board_mem(source_image))
{if(is_file(source_image)) /* copy_board_mem_to_file(source_image, 8, FALSE);*/ cputs("not implemented yet");
 if(is_fr_grab(source_image)) copy_board_mem_to_fr_grab(source_image, dest_image = 0);
 if(is_ext_mem(dest_image)) copy_board_mem_to_ext_mem(source_image, dest_image = 100);
 if(is_board_mem(dest_image)) copy_board_mem_to_board_mem(source_image, dest_image, Bits);
}
if(is_wedge(source_image))
{if(is_file(dest_image)) copy_wedge_to_fr_grab(dest_image = TRUE);
 if(is_ext_mem(dest_image)) copy_wedge_to_ext_mem(dest_image = 0);
 if(is_board_mem(dest_image)) copy_wedge_to_board_mem(dest_image, Bits);
}
clrscr();
}
The following defines may be changed when making modifications to the user interface program, however, care should be taken to not change the format of any of the defines, or incorrectly change the value.

#define DEFAULT_BASE "dec" /* default base upon program startup, valid values: "dec" or "hex" */
#define MASK_SIZE 7 /* mask size, should be left at 7 to be compatible with the KIF */
#define MAX_INSTRUCTIONS 50 /* maximum number of instructions allowed at any given time */
#define MAX_MASKS 50 /* maximum number of masks allowed at any given time */
#define MAX_PREDEF_INSTR 150 /* maximum number of instructions allowed in the predefined instruction file */
#define MAX_FR_GRAB_BUFFERS 9 /* maximum number of frame grabber buffers */
#define IMAGE_SIZE 512 /* width and height of square image */
#define BUFFERS_PER_IMAGE 16 /* number of buffers per image, this value should always be (IMAGE_SIZE * 2) / (BUFFER_SIZE * 2) */
#define INSTRUCTION_PREDEFINE_FILE "defines\instruct.def" /* default pathname to the predefined instruction file */
#define DEFAULT_DIRECTORIES_FILE "defines\default.def" /* default pathname to the file containing default directory information */
#define FR_GRAB_DEF_FILE "defines\fgrab.def" /* default pathname to the file containing frame grabber information */
#define CONFIG_FILE "defines\config.def" /* default file containing board configuration information */
#define STARTUP_FILE "defines\startup.def" /* default startup file when running program with no command line arguments */
#define DEF_IMAGE_EXT ".img" /* default extension for image files */
#define DEF_INSTR_EXT ".i" /* default extension for instruction files */
#define DEF_WINDOW_EXT ".win" /* default extension for mask files */
#define DEF_STARTUP_EXT ".def" /* default extension for startup files */

The following defines are specific for a 2048 x 1024 frame grabber, and convert x and y coordinates into the address register and page number values, as specified by the frame grabber manual. If the frame grabber attaches to the PC, the frame grabber manual for the new conversion equations, and modify the following defines.

#define convert_to_address_register(x,y) ((y << 11) + x)
#define convert_to_page_number(y) (y >> 5)

The following defines should not be changed when making modifications to the user interface program, unless it is absolutely necessary.

#define strlen(s1,s2) /* define a new function for checking equality of strings */
#define is_empty(s1) /* define a new function to check if a string is empty */
#define normal_video() /* define a new function to print in standard text color */
#define color_black() /* define a new function to print black text */
#define color_blue() /* define a new function to print blue text */
#define color_green() /* define a new function to print green text */
#define color_red() /* define a new function to print red text */
#define color_purple() /* define a new function to print purple text */
#define color_yellow() /* define a new function to print yellow text */
#define reverse_background() /* define a new function to reverse the background color */
#define normal_background() /* define a new function to restore the background color */
#define TRUE 1 /* define the numeric value for TRUE */
#define FALSE 0 /* define the numeric value for FALSE */
#define NO 2 /* define the numeric value for NO */
#define YES 1 /* define the numeric value for YES */
#define ABOUT -1 /* define the numeric value for ABOUT */
#define BACKWARDS 0 /* define the numeric value for BACKWARDS */
#define FORWARDS 1 /* define the numeric value for FORWARDS */
#define SOURCE 0 /* define the numeric value for SOURCE */
#define DESTINATION 1 /* define the numeric value for DESTINATION */
#define INSTRUCTION 2 /* define the numeric value for INSTRUCTION */
int Base = DEFAULT_BASE, Predef_Instr_Count, Mask_Loaded = FALSE, Ext_Mem_Count, Reg_Loc, Bits;
int8 Page_Number, Pixel_Value, Ext_Mem, Block_Copy_Status;
int16 Board_Loc_High, Address_Register, Ext_Mem_Base_High, Ext_Mem_Base_Mid, Buffer[BUFFER_SIZE];
file_name_type Default_Instr_Directory, Default_Window_Directory, Default_Image_Directory, Default_Startup_Directory;

instr_type instr;
predef_instr_type Predef_Instr[MAX_PREDEF_INSTR];
mask_type Mask[MAX_MASKS + 1];
fr_grab_info_type Fr_Grab;
header_type Header;
directory_type Dir[MAX_FILES_IN_DIRECTORY];

FILE *Disk;
typedef unsigned int int16; /* 16-bit integer definition */
typedef unsigned char int8; /* 8-bit integer definition */
typedef char file_name_type[MAX_FILE_NAME_LENGTH + 1]; /* file name string definition */
typedef char *strptr; /* string pointer definition */

typedef struct
  {long_string instr[MAX_INSTRUCTIONS];
   file_name_type filename;
   int current, modified;
  } instr_type; /* instruction type definition */

typedef struct
  {int val[MASK_SIZE][MASK_SIZE], valid, modified;
   file_name_type filename;
  } mask_type; /* mask type definition */

typedef struct
  {long_string instr_name;
   int alu1_op, max, alu2_op;
  } predef_instr_type; /* predefined instruction type definition */

typedef struct
  {unsigned int x_size, y_size, x_origin, y_origin, format, comment_size;
   unsigned char title2[2], comment[256];
  } header_type; /* IPS header type definition */

typedef struct
  {int buffers;
   int x_origin[MAX_FR_GRAB_BUFFERS], y_origin[MAX_FR_GRAB_BUFFERS];
  } fr_grab_info_type; /* frame grabber configuration type definition */

typedef struct
  {char filename[9];
   char extension[4];
  } directory_type; /* file name type definition */

enum input_types {string, integer, signed_integer}; /* specify the input type enumeration */
enum bases {dec, hex}; /* specify the base enumeration */
Morphologic Image Processor User Interface Software

program name: IMAGES.ASM

; purpose: contains the plot pixel and get pixel routines

.286c

DOSSES

include includes\def2600.inc

.MODEL small

.STACK 200h

.DATA

EXTERN _Page_Number:WORD
EXTERN _Picel_Value:BYTE

.CODE

PUBLIC _plot_pixel
PUBLIC _get_pixel

_plot_pixel PROC
    push es
    mov eax,reg_reg
    mov es,ax

    push ds
    mov dh,Page_Number
    mov ds,ax

    mov BYTE PTR ds:2h,01h

    mov ds,ax

    pop ds

    mov ax,mem_reg
    mov es,ax

    mov bx,Address_Register
    mov dl,Pixel_Value

    pop es

    mov al,dl

    mov ax,reg_reg
    mov es,ax

    mov dh,Page_Number
    mov ds,ax

    mov BYTE PTR ds:2h,01h

    mov ds,ax

    pop ds

    mov ax,mem_reg
    mov es,ax

    mov bx,Address_Register
    mov dl,Pixel_Value

    pop es

    ret

_plot_pixel ENDP

_get_pixel PROC
    push es
    mov eax,reg_reg
    mov es,ax

    push ds
    mov dh,Page_Number
    mov ds,ax

    mov BYTE PTR ds:2h,01h

    mov ds,ax

    pop ds

    mov ax,mem_reg
    mov es,ax

    mov bx,Address_Register
    mov dl,Pixel_Value

    pop es

    ret

_get_pixel ENDP

END
Worphoiogical Image Processor User Interface Software

program name: MQ-fiCFHK.-.C

purpose: contains all extended memory transfer routines

DOSSEG

.MODEL small
.STACK 100h

.DATA
EXTRN _Ext_Mem:BYTE, _Block_Copy_Status:BYTE
EXTRN _Board_Loc_High:WORD, _Ext_Mem_Base:WORD, _Ext_Mem_Base_Mid:WORD
EXTRN _Buffer:WORD:4000h
bmdt db 30h dup (0)

.words_to_copy dw 2000h
segment_length dw 03fffh ;at least words_to_copy*2-1

.CODE
PUBLIC _write_mask_to_board
PUBLIC _get_extended_memory_size
PUBLIC _copy_ext_mem_to_buffer
PUBLIC _copy_buffer_to_ext_mem

_get_extended_memory_size PROC
mov ah,88h
int 15h
shr ah,1
shr ah,1
mov ext_mem,ah
ret
_get_extended_memory_size ENDP

_write_mask_to_board PROC
push bp
mov bp,sp
mov cx,[bp+4] ;move size of mask array into CX
mov ax,[bp+6] ;move address of mask array into AX
push si
push ds
push es
mov dx,Board_Loc_High
mov bx,seg ax
mov ds,bx
mov bx,offset ax
mov ax,0
mov si,seg bmdt
mov es,si
mov si,offset bmdt
mov ea:[si+10h],cx
mov ea:[si+18h],cx
mov byte ptr es:[si+15h],93h
mov byte ptr es:[si+16h],93h
mov es:[si+1ah],ax
mov es:[si+1ch],dl
mov ax,ds
mov dx,16
mul dx
add ax,bx
adc dx,0
mov es:[si+12h],ax
mov es:[si+14h],di
shr cx,1
mov ah,87h
int 15h
mov _block_copy_status,ah
pop es
pop ds
pop si
pop bp
ret
_write_mask_to_board ENDP

_copy_ext_mem_to_buffer PROC
push si
push ds
push es
call copy_setup
call getblk
call setup
pop es
pop ds
pop si
ret
_copy_ext_mem_to_buffer ENDP

_copy_buffer_to_ext_mem PROC
push si

B-47
push ds
push es
call copy_setup
call putblk
pop es
pop ds
pop si
ret

_COPY_BUFFER_TO_EXT_MEM ENDP

COPY_SETUP PROC near
    mov dx, _EXT_MEM_BASE_HIGH
    mov ax, _EXT_MEM_BASE_MID
    mov ah, al
    mov bx, seg _Buffer
    mov ds, bx
    mov bx, offset _Buffer
    mov si, seg bmdt
    mov es, si
    mov si, offset bmdt
    mov cx, segment_length
    mov es:[si+10h].cx
    mov byte ptr es:[si+15h], 03h
    mov byte ptr es:[si+16h], 03h
    ret
COPY_SETUP ENDP

GETBLK PROC near
    mov es:[si+12h], ax
    mov es:[si+14h], dl
    mov ax, ds
    mov dx, 16
    mul dx
    add ax, bx
    adc dx, 0
    mov es:[si+1ah], ax
    mov es:[si+1ch], dl
    mov cx, words_to_copy
    mov ah, 87h
    int 15h
    mov _block_copy_status, ah
    ret
GETBLK ENDP

PUTBLK PROC near
    mov es:[si+1ah], ax
    mov es:[si+1ch], dl
    mov ax, ds
    mov dx, 16
    mul dx
    add ax, bx
    adc dx, 0
    mov es:[si+12h], ax
    mov es:[si+14h], dl
    mov cx, words_to_copy
    mov ah, 87h
    int 15h
    mov _block_copy_status, ah
    ret
PUTBLK ENDP

END
Appendix C. Schematics

This appendix contains all schematics not shown in previous sections and describes schematics not explained in previous sections. The top level schematic was shown in section 4. All main controller schematics, except for delay14, DLC4, DLC8, and DLEC8, were shown in section 5.2. All memory controller schematics except for mx2_18bit were shown in section 5.3. The delay14 block, as shown in figure C-1, consists of 14 daisy-chained flip-flops, which delays a single cycle pulse by 14 clock cycles. The DLC4 and DLC8 blocks, as shown in figures C-2 and C-3 respectively, provide 4-bit and 8-bit versions of Actel’s DLC block, which is a gated latch with a clear signal. The DLEC8 block, shown in figure C-4, provides an 8-bit gated latch with clear and enable signals. The mx2_18bit block, shown in figure C-5, provides an 18-bit version of Actel’s MX2 multiplexor.

The Blank.Counter and Mem.Counter schematics were both shown in section 5 (figures 5-6 and 5-10, respectively), and will be described here. Both the Blank.Counter and Mem.Counter blocks use the same counting scheme to implement an 18-bit counter. The counters are composed of 18 flip-flops (each outputting one bit of the count) split into two groups of nine. Each group of nine operates in a similar fashion, with the upper nine bits being enabled when the lower nine bits have reached their maximum count. To reset the count back to zero, a START or CLRn signal is issued, which selects the B inputs of all the flip-flops at the next rising clock edge. When the START or CLRn signal is no longer present, the counter will begin counting to a maximum value of $2^{18} - 1$, increasing the count by one at every rising clock edge. Each nine bit group of flip-flops operates using a standard method of counting, with the lowest bit being inverted at every clock edge. The rest of the flip-flops have their own outputs exclusive ORed with the AND of the previous bits and returned to the inputs, which serves the purpose of only toggling a bit one clock cycle after all the previous bits are
high. When the count of the lower nine bits reaches its maximum value, an enable signal is sent to the upper nine bits, which operate the same as the lower nine bits with the exception of the enable signal. A done signal is issued when all 18 bits are high. The Start_Blank block also counts using the same method as the Mem.Counter and Blank.Counter blocks, but some of the bits have been removed, and the done signal is issued as described in section 5.2.

The Blank.Counter block is also responsible for issuing the row and column blanking signals (ROWBLNK and COLBLNK, respectively). The required control signals are described in tables 3-2 and 3-3. Since the ROWBLNK and COLBLNK signals are issued using identical circuitry, with the ROWBLNK signal using the upper nine bits and the COLBLNK signal using the lower nine bits, only the COLBLNK signals will be explained. The flip-flops labeled U65 through U70 are responsible for the column blanking bits 0 through 5, respectively. When the column value is 0 (the lower nine bits are all low), the AND gate labeled U71 is high, which will set the COLBLNK bits 3 through 5 high at the next rising clock edge. During the next few rising clock edges, a low value will propagate through the three flip-flops, following the pattern described in table 3-2. After the three bits are low, they will remain low until the next time the column is 0. COLBLNK bits 0 through 2 are generated almost the same way, but the bits are normally low, and a high value is propagated through to generate the signals as described in table 3-2. Since all the ROWBLNK and COLBLNK signals are generated one clock cycle after their corresponding addresses are generated, the address of the Blank.Counter block must always be one clock cycle ahead of the address of the target pixel, as explained in section 5.3 and as shown in the system timing diagram (figure 5-1).
Figure C-1 Delay14 Schematic
Figure C-2 DLC4 Schematic
Figure C-3  DLC8 Schematic
Figure C-4 DLEC8 Schematic
Figure C-5 Mx2_18bit Schematic
Appendix D. Bibliography


