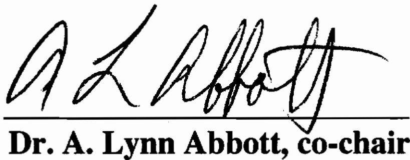
Design and Implementation of a Reconfigurable FPGA-Based Video Frame Grabber Board

By

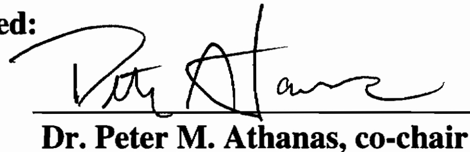
Jeffrey A. Nevits

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
In
Electrical Engineering

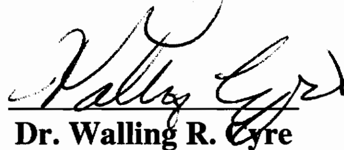
Approved:



Dr. A. Lynn Abbott, co-chair



Dr. Peter M. Athanas, co-chair



Dr. Walling R. Cyre

August 5, 1996

Blacksburg, Virginia

Keywords: FPGAs, Splash, Reconfigurable, Video, RS-170

c.2

LD
5655
V855
1996
N485
c.2

DESIGN AND IMPLEMENTATION OF A RECONFIGURABLE FPGA-BASED VIDEO FRAME GRABBER BOARD

By
Jeffrey A. Nevits

Dr. A. Lynn Abbott, co-chair
Dr. Peter M. Athanas, co-chair

Bradley Department of Electrical Engineering

ABSTRACT

This thesis describes the design and implementation of the JB1 reconfigurable video frame grabber board and its use in the Virginia Tech Splash system. The system utilizes the frame grabber board to provide the Splash-2 platform with real time digital images suitable for image processing. The board converts analog black and white video images (RS-170 format) into digital grey scale images of sizes up to 480 rows × 512 columns × 8 bits per pixel. The resulting images are then transferred to the Splash-2 platform in real time for subsequent processing. The board utilizes two Xilinx field programmable gate arrays (FPGAs) for implementation of different configurations. A software user interface has also been developed to control the operation of the board.

Acknowledgments

I would like to thank my committee members, Dr. Abbott, Dr. Athanas, and Dr. Cyre, for their guidance and patience throughout the entire project. Additional thanks goes to the Splash project team, especially Nabeel Shirazi and Brad Fross, for all of their help during the project and to Mr. Thomas Drayer for providing much of the technical assistance during the development of the board. And finally, a very special thanks to my parents for their continued support throughout this endeavor.

Table of Contents

1	**Introduction**		
	1.0	Motivation	1
	1.1	Project Overview	2
	1.2	Research Contribution	3
	1.3	Thesis Organization	4
2	**Background**		
	2.0	Introduction	5
	2.1	Splash Architecture	6
	2.2	Splash-2 Software Environment	8
	2.3	VTSplash	9
	2.4	Typical Applications	11
3	**JB1 Board Description**		
	3.0	Introduction	12
	3.1	Design Considerations	13
	3.2	Block Diagram	15
	3.3	PC-JB1 Interface	16
		3.3.1 PC Functions	16
		3.3.2 Interfacing to the PC	16
		3.3.3 Configuration Sequence for Xilinx FPGAs	21
	3.4	Video Camera Interface	26
		3.4.1 Introduction to RS-170 Video Format	26
		3.4.2 Video Digitizer Chip	28
		3.4.3 Video Interface Details	30
	3.5	Splash Interface	35
4	**JB1 Software**		
	4.0	Introduction	41
	4.1	Utility Function Blocks	42
	4.2	Grabber User's Guide	44
		4.2.1 Autoconfiguration	45
		4.2.2 Download Configuration to LCA	45
		4.2.3 Change Capture Parameters	45
		4.2.4 Continuously Capture Images	46
		4.2.5 Display Image	46
		4.2.6 Read/Write On-board Memory	46
		4.2.7 Change Clock Setup	47
		4.2.8 Reset Xilinx Chips	47

4.2.9 Enable/Disable Splash 47
4.2.10 Exit Program 48

5 **Future Enhancements**
5.0 Introduction 49
5.1 Alternate Configurations 50
5.2 Implementation of New Designs 51

6 **Conclusions** 53

7 **References** 54

8 **Appendix A:** Introduction to Xilinx FPGAs 56

9 **Appendix B:** JB1 Board Information 65

10 **Appendix C:** XC 3042 Design 73

11 **Appendix D:** XC 3090 Design 103

12 **Appendix E:** Grabber Device Driver Source Code Listing 122

List of Figures

Figure 1.1	VTSplash system overview	3
Figure 2.1	Splash-2 system architecture	6
Figure 2.2	VTSplash system	10
Figure 3.1	JB1 block diagram	15
Figure 3.2	PC I/O read cycle	17
Figure 3.3	PC I/O write cycle	18
Figure 3.4	PC-JB1 interface block diagram	19
Figure 3.5	Four step process for bitstream development	22
Figure 3.6	Configuration flowchart	25
Figure 3.7	Image interlacing	26
Figure 3.8	Representation of one RS-170 line	27
Figure 3.9	Image sizes for 4:3 and 1:1 aspect ratios	28
Figure 3.10	Line timing for 4:3 and 1:1 aspect ratios	28
Figure 3.11	Timing for even and odd fields	29
Figure 3.12	Video interface block diagram	30
Figure 3.13	Field digitization flowchart	31
Figure 3.14	Capture system block diagram	32
Figure 3.15	Splash interface block diagram	35
Figure 3.16	JB1 to Splash data transfer protocol	37
Figure 3.17	Transfer system block diagram	38

List of Tables

Table 3.1	I/O addresses used by frame grabber board	20
Table 3.2	Control port description	24
Table 3.3	XC 3042 internal port usage	35
Table 3.4	Programmable clock control register	36
Table 3.5	XC 3090 internal port usage	38
Table 3.6	Control register bit definitions	39

Chapter 1
Introduction

1.0 Motivation

Today's general purpose computers do not offer the performance needed to implement real time image processing applications. Their generic architectures lack the ability to transfer and process the large volumes of data in the time required for real time image processing. For systems which require the large throughput associated with image processing, application specific hardware is needed to achieve the desired performance. The need for a unique solution for each application requires the design of multiple systems, resulting in long design cycles and high costs. A different approach is to design one high speed platform which can be dynamically changed to match the application at hand. This solution offers the high performance needed while keeping the design cycle time relatively short for each application.

The Splash-2 system, designed by the Center for Computational Sciences (formerly the Supercomputing Research Center), in Bowie, Maryland, is an implementation of this custom computing approach [1]. The Splash-2 system is a programmable, reconfigurable parallel processing platform which can quickly be configured to optimize the hardware for the current application. Additional applications can be accommodated by changing the hardware to different configurations through software. Splash-2 offers the high performance needed for a wide range of applications, including many image processing tasks.

To perform real time processing of video streams, the Splash-2 system requires an external video frame grabber board to provide data to the processing board. A custom frame grabber was designed to provide the digitized data to the Splash-2 system in a variety of different formats and speeds to maximize the number of applications which could be performed.

1.1 Project Objective

The VTSplash team, composed of students and faculty from the Bradley Department of Electrical Engineering at Virginia Polytechnic Institute and State University, was formed to implement high speed image processing applications on real time video streams using the Splash-2 platform. A block diagram of the system is shown in Figure 1.1. A video camera produces black and white video images in the standard RS-170 format. These are digitized and captured using the video frame grabber board. The grey scale images are then transferred to the Splash-2 system where high speed image processing is performed. The resulting images are transferred to a video card and displayed on a monitor.

Figure 1.1: VTSplash system overview. The system captures, processes and displays real time black and white video images.

The purpose of this thesis is to describe the design and implementation of a video frame grabber board that interfaces an RS-170 video camera with the Splash-2 system. The interface allows video processing tasks to be performed in real time on a continuous video stream. Key requirements of the board were the ability to transfer data in real time and the ability to transfer images of different sizes.

1.2 Research Contributions

The contribution of this research to the VTSplash project was the design, implementation and integration of the video frame grabber board JB1 into the VTSplash system. The board converts RS-170 black and white video images into digital grey scale images and transfers the images to the Splash-2 platform via a handshaking protocol. For maximum flexibility, the board can be programmed for various image sizes up to 480 $\times$ 512 pixels and output rates up to 10 MHz. This flexibility allows for the processing of smaller images at slower rates for those applications which can not be performed in real time. The hardware can be easily adapted for new configurations through the use of on-board programmable logic devices. The software utility Grabber was developed to control the operation of the board.

1.3 Thesis Organization

This thesis documents the development of the JB1 video interface and its use in the VTSplash system. Chapter 2 introduces the Splash-2 system and the Virginia Tech Splash system used to perform the image processing applications. JB1, a component of the VTSplash system, is described in chapter 3. Chapter 4 describes the software needed to utilize the frame grabber board. Chapter 5 summarizes future work that would increase the flexibility of the board. The conclusions of the project are discussed in Chapter 6.

This thesis also contains five appendices. Appendix A provides a brief introduction to field programmable gate arrays. Appendix B contains the board level schematics, cable header definitions and a parts list for the board. Appendices C and D contain the schematics and state machines for the two Xilinx chips used on JB1. Appendix E contains the C source code for the device driver for the frame grabber board.

Chapter 2
Background

2.0 Introduction

The Virginia Tech Splash system, VTSplash, utilizes a parallel computing platform to perform real time video processing applications. The system hardware can be custom configured for a variety of different image processing tasks, resulting in high system performance. At the core of the system is the Splash-2 processor board. The unique reconfigurable architecture of the Splash-2 system is described in Section 2.1. The Splash-2 software environment, needed for controlling the Splash system, is described in Section 2.2. The Virginia Tech system, described in Section 2.3, was formed by augmenting the Splash-2 system with additional video hardware to make the system suitable for processing real time video images. Section 2.4 describes some of the video processing applications which have been implemented at Virginia Tech using the VTSplash system.

2.1 Splash-2 Architecture

Splash-2 is a flexible parallel processing platform based on field programmable gate arrays (FPGAs). The hardware can be reconfigured to match a given application, resulting in improved performance over general purpose computers. Figure 2.1 shows the Splash-2 system architecture.

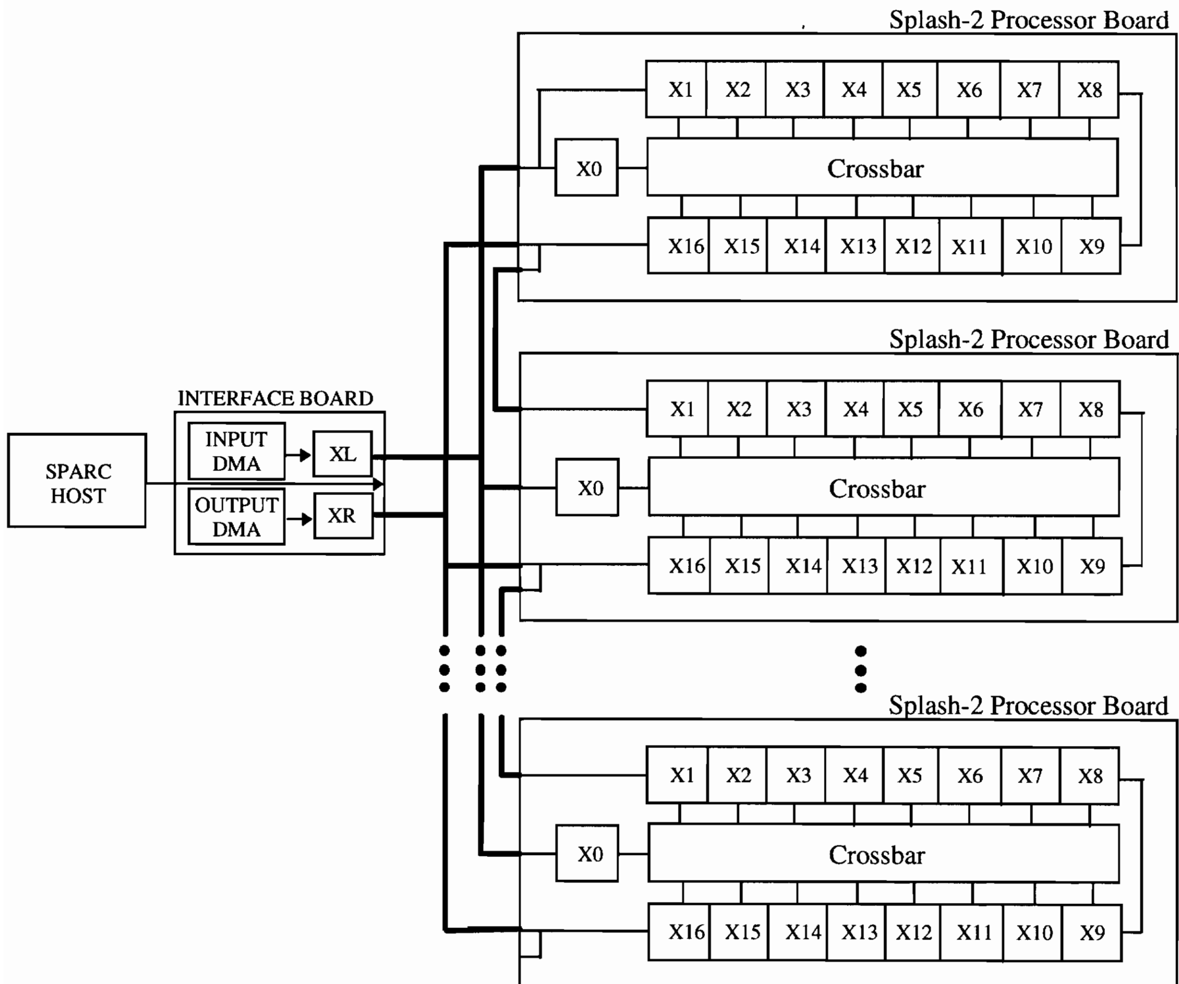


Figure 2.1: Splash-2 system architecture [1]. The Splash-2 system is composed of multiple processor boards, each containing 16 processing elements connected with a full crossbar.

The Splash-2 system consists of an interface board and up to sixteen processing boards hosted by a Sun SPARCstation 2. The host computer performs two main functions:

1. Programs the interface board configuration, and
2. Programs the individual processing elements on the processing boards.

The interface board provides the interface between the host computer and the processing boards. The SPARC Sbus is used for communication between the host and the interface board. Functions performed by the interface board include:

1. Data transfer between the host and the processor boards,
2. DMA transfers to and from the processor board memory,
3. Clock control for the processor boards,
4. Interrupt handling for the system,
5. Controlling access to the local memory on the processing boards, and
6. Controlling the Splash-2 system.

The interface board augments the 32-bit data from the host with an additional four bit tag before passing the data to the Splash-2 processing boards. Three DMA channels exist for high speed data transfers between the host and the Splash-2 system. Transfer rates up to 54 Mbytes per second are possible using burst transfers. The interface board utilizes two FPGAs to control the data streams to and from the host and processing boards. FIFOs are used with each channel to allow for a continuous stream of data to be fed into the processing boards. The system clock can be single-stepped, multiple-stepped or free running with frequencies ranging from 100 Hz to 40 MHz.

Interconnection between the interface board and the processing boards is achieved through the Futurebus+ backplane. The multiple processing boards are connected in a cascaded fashion as shown in Figure 2.1.

Each processing board is composed of sixteen Xilinx XC 4010 field programmable gate arrays connected in a bi-directional linear array. (Appendix A provides an overview of the Xilinx family of field programmable gate arrays.) The array provides an architecture that is well suited for pipelined operations. Each FPGA is a processing element which can be configured for one or more stages in the pipeline. Neighboring FPGAs in the array are connected with a 36-bit data bus.

Communication between non-neighboring Xilinx chips is accomplished though a full 16 $\times$ 16 bit bi-directional crossbar switch. The crossbar can store up to eight different configurations and can be dynamically reconfigured. Each FPGA has 1/2 Mbyte of SRAM to provide local data storage and is connected to its local memory through a 16-bit data bus, 18-bit address bus and two control strobes. The local memory may also be accessed by the host system. System hardware prevents the local processing elements and host from accessing memory simultaneously. An additional FPGA is used on each processor board to provide interfacing to the Splash system bus.

2.2 Splash-2 Software Environment

Prior to running an application on the Splash-2 system, the host, the interface board and the processing boards must be configured for the intended application [2].

The host must be configured through the use of library functions to regulate the data streams flowing to and from the Splash processing boards. Additional parameters must be set up for programming the processing boards. The programmable control devices on the interface board and the control elements on the processing boards are also configured by using functions from a pre-existing library.

Once the host and interface board are set up, the individual processing elements and the crossbars on the processing boards need to be configured. The FPGAs must be programmed with their configuration data. The configuration data is developed using a multi-step design process. After defining the application, the problem must be partitioned to fit into the individual processing elements. The hardware is then described using VHDL behavioral models. The model can then be simulated using the Splash simulator. Upon successful modeling and simulation, the logic can be synthesized, partitioned and then placed and routed into the processing elements through the use of automated tools. Upon completion of routing, timing information may be extracted and the design can be downloaded into the Xilinx chips.

2.3 VTSplash

In order to perform high speed image processing, the Splash system was augmented with additional hardware to form the VTSplash system. The completed system, shown in Figure 2.2, offers the power needed for the capturing, processing and displaying of real time video images.

The system consists of a black and white video camera, the Splash-2 system, an IBM PC compatible computer, the JB1 frame grabber board, several output video boards and a video monitor. Data is captured from the video camera by the frame grabber board and transferred to the Splash system via the Splash interface. Next, the data is transferred to the Splash processing board where the image processing algorithms are applied to the data. The data is then transferred out to several post-processing boards via the Splash interface. Finally, the resulting images are displayed on a video monitor.

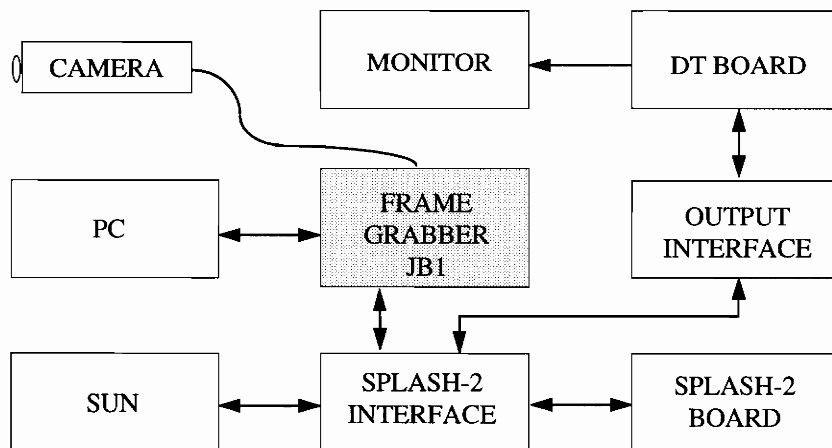


Figure 2.2: VTSplash system. The system contains equipment to capture, process, and display video images in real time.

The objective of this research was to design the video frame grabber card called JB1. The frame grabber card captures analog video images from the camera and transfers them to Splash-2 system in real time in a digital format suitable for image processing. The board is programmable by the user to provide maximum flexibility. Chapter 3 describes the hardware design of the board and chapter 4 describes the software for the board. Additional information is presented in the appendices.

2.4 Typical Applications

The Virginia Tech team utilized the VTSplash system to implement a group of real time image processing applications. Applications implemented include:

1. The Hough transform for the detection of straight lines in images [3,4,5,6],
2. Region detection and labeling in images [7],
3. The generation of Gaussian and Laplacian image pyramids [5,8],
4. 2-D Fast Fourier Transform [9,10],
5. Median and morphological filtering of images [11].

When multiple processing boards are used in the system, the various tasks can be pipelined together to perform more complex image processing applications.

Chapter 3
JB1 Board Description

3.0 Introduction

JB1 is a frame grabber board that captures video images from the video camera and transfers the data to the Splash system at real time rates. Key features of the board are:

- Real time capturing of RS-170 black and white video images,
- Programmable data transfer rate to Splash, using clock speeds up to 10 MHz,
- Programmable image size, up to 480×512 8-bit pixels, and
- Reconfigurable hardware.

The hardware was built as a wire-wrapped board that is hosted by an IBM PC compatible. The host provides power and control for the board. Through the use of two Xilinx field programmable gate arrays, the board is reconfigurable and programmable under software control by the host. This chapter describes the architecture of the board. The schematics

and state machines for the board are described in Appendices B, C and D. Chapter 4 describes the software necessary to use the board.

3.1 Design Considerations

To achieve the project goal of real time image processing, several fundamental design criteria were identified. The following items were initially identified as important issues which would shape the board design:

- Video format,
- Real time video transfer rate,
- Programmable clock,
- Programmable image size, and
- Expandability.

Each of these is now discussed in more detail. The common black and white video format, RS-170, was chosen for several reasons. First, the output from many video cameras is in the EIA standard RS-170 format [12]. Black and white data was chosen since only one channel of information needs to be digitized and stored, as opposed to the three channels needed for color. This reduces the complexity of the frame grabber board by reducing the amount of conversion hardware and memory needed.

The overall goal of the system is to perform real time video processing, so the board must be able to handle real time image transfers. Section 3.4 describes the throughput needed for real time processing. Additionally, the need for slower-than-real-time processing was identified for certain applications. Certain applications also require the use of images

smaller than the standard size. By making clock speeds and image size programmable from the PC, the frame grabber board offers maximum flexibility.

The size and speed of JB1 memory were dictated by the image size and the need for real time processing. The necessary logic was implemented on field programmable gate arrays. FPGAs, described in Appendix A, were used for their ability to be reconfigured and for their high density. By utilizing FPGAs, the board could be quickly reconfigured to meet the needs of different applications. The high density of the FPGAs would allow more complex designs to be implemented when needed. Additionally, the FPGAs offered a quicker design cycle than discrete logic.

JB1 needed a host computer to perform the necessary configuration functions before the frame grabber board could be used to capture video images. An IBM PC was chosen over other microprocessor based systems as the host computer for several reasons. The PC is a low-cost host that can perform all of the functions required to use the frame grabber board. Additionally, interfacing to the PC is straightforward. Finally, the PC offered a convenient method for verifying the correct operation of each stage of the frame grabber board.

The frame grabber board must have the ability to be expanded as additional applications for the system are developed. Specialized configurations which require more logic can then be implemented. The size of the FPGAs used in the design were selected to allow for additional features to be implemented.

3.2 Block Diagram

The block diagram for the JB1 frame grabber board is shown in Figure 3.1. A top-down approach was used to design the board. The block diagram depicts the four major components to the board: the PC interface, the video camera interface, the Splash interface and local memory. Each section was developed separately and then integrated into a single system.

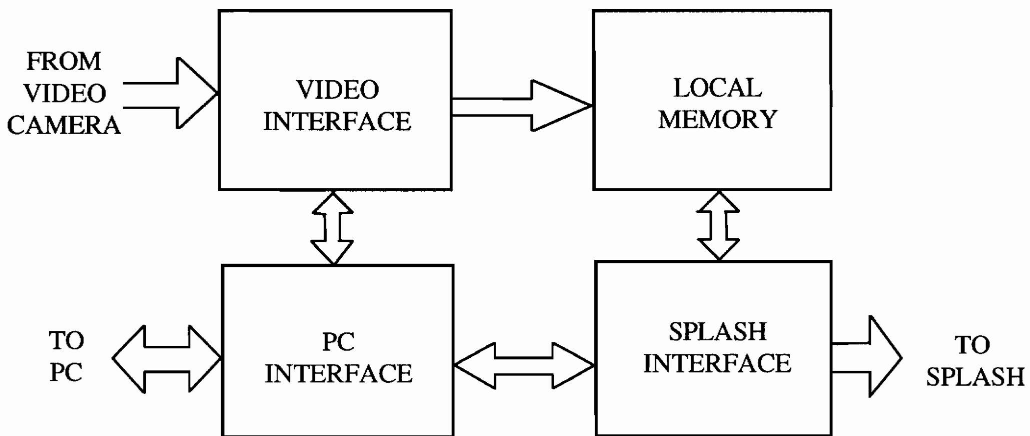


Figure 3.1: JB1 block diagram. The video image is digitized, stored in local memory and then transferred to Splash. A PC provides power and control to the board. The video interface and the Splash interface are each contained in separate field programmable gate arrays.

The PC interface allows the host computer to communicate with the frame grabber board. The host computer is used to provide power and to control the board. Various design configurations and parameters can be downloaded from the PC to the board through the PC bus. Section 3.3 discusses the interface.

The video camera interface captures data from the video camera and stores it in a local memory bank. After the bank is full, the Splash interface transfers the data in the bank to

the Splash system. At the same time, a second bank of memory is being filled with new camera data. In this manner, data can be sent to Splash in real time. Details of the two interfaces are given in Sections 3.4 and 3.5.

JB1 uses two clocks to synchronize the logic in the interfaces. The video interface uses the pixel clock to collect the video data. The Splash interface uses the transfer clock to transfer data to Splash. The two clocks are independent from each other.

3.3 PC-JB1 Interface

The PC host is used to control the frame grabber board through the use of a device driver written in C. The software, developed in conjunction with the hardware, is described in Chapter 4.

3.3.1 PC Functions

In addition to providing power, the PC performs three key functions:

1. Configures the FPGAs on the frame grabber board,
2. Sets the data transfer parameters, and
3. Verifies board functionality.

3.3.2 Interfacing to the PC

Interfacing to the IBM PC is accomplished through the ISA bus [13]. The following signals are necessary to interface to the ISA bus:

- CLK System clock.
- ALE Address latch enable. When active, address is valid.
- AEN Address enable. When active, a DMA cycle is in progress.
- A0-A19 Address lines. Only lines A0-A15 are used for an I/O cycle.
- D0-D7 Data lines.
- I/O_R* I/O read cycle strobe. When active, a read cycle is in progress.
- I/O_W* I/O write cycle strobe. When active, a write cycle is in progress.

All of the signals are available on the 62 pin system bus expansion slots of the PC. During I/O cycles, all of the signals except the data lines are driven by the PC. The data lines are driven by the PC during write cycles and by JB1 during read cycles. For more information on other signals on the ISA bus, consult [14] and [15]. Figure 3.2 shows the PC I/O read cycle. Figure 3.3 shows the PC I/O write cycle. The waveforms show the relative timing of the signals to each other. Exact timing of the signals is PC dependent.

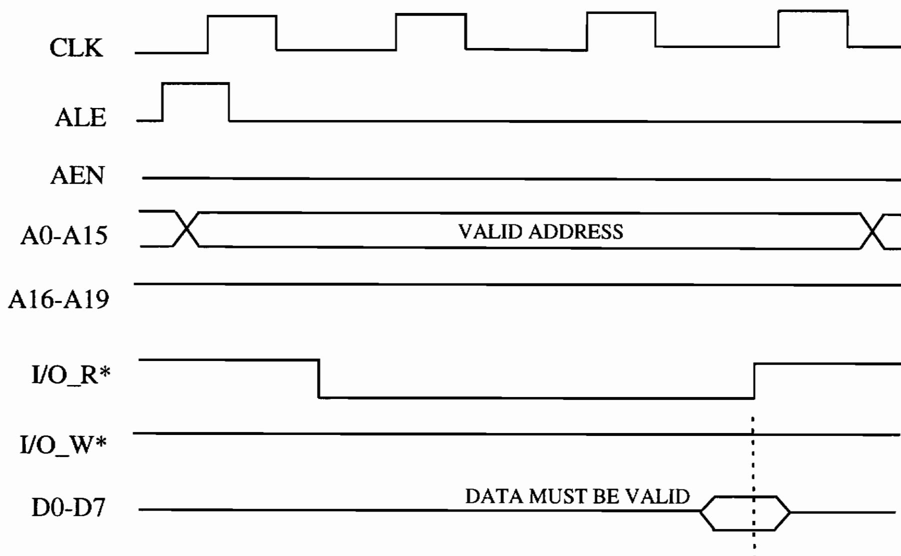


Figure 3.2: PC I/O read cycle. During a read cycle, data is transferred to the PC from an external card.

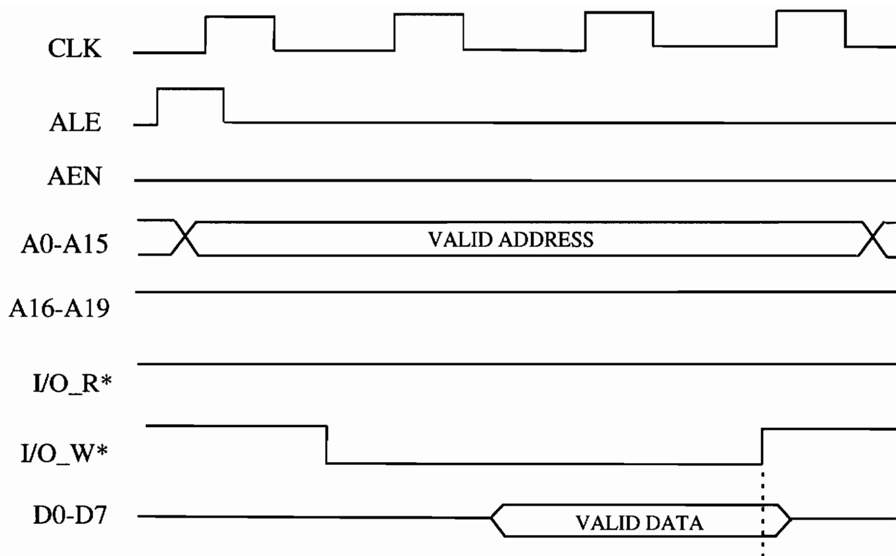


Figure 3.3: PC I/O write cycle. During a write cycle, data is transferred from the PC to an external card.

To interface to the PC through the ISA bus, a board must decode the address lines and the I/O strobes. If the address on A0-A15 matches an address in the board's range and an I/O cycle is in progress, then the data may be read/written to/from the board. Note that the address enable, AEN, must be inactive to be a valid I/O cycle and not a DMA cycle.

The I/O address space of the IBM PC extends from 0x0000H to 0x03FFH and is accessible through the ISA bus of the PC. Locations 0x0000 through 0x01FFH are used by the base system. Locations 0x0200H through 0x03FFH are available for use by all of the cards found in the PC [13]. The standard cards, including the video card, serial cards, and disk controller cards leave gaps of the I/O space unused and available to the user.

Figure 3.4 shows the block diagram of the PC-JB1 interface. Appendix B contains the schematic for the interface. The main components of the interface are address decoding and buffering, which allow data to be transferred between the PC and JB1 using the ISA bus. Additional components consist of the control register and the clock register. The control register consists of an output port and an input port which are used for configuring the Xilinx chips and transferring data between the PC and JB1.

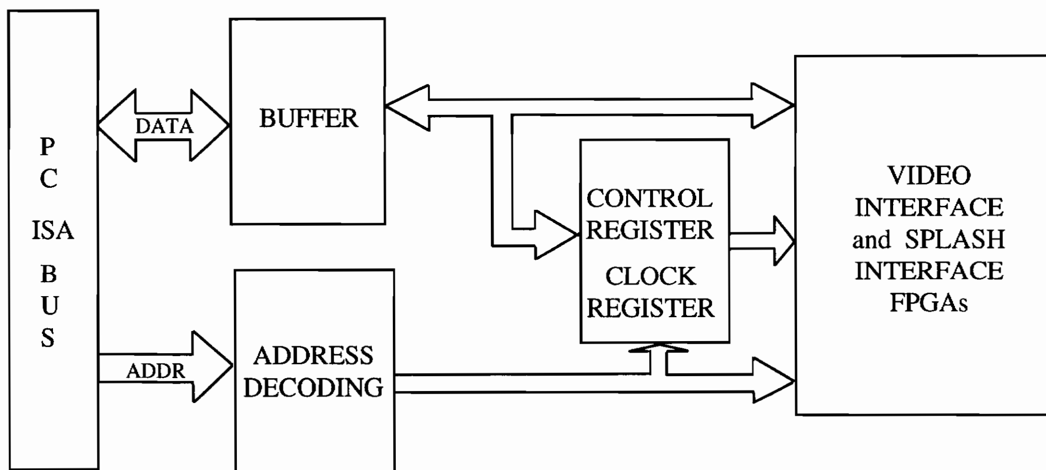


Figure 3.4: PC-JB1 interface block diagram. The interface allows data to be transferred between the PC and the FPGAs used in the video and Splash interfaces.

A PB751-AT prototype card, manufactured by Industrial Computer Source [16], was used as the basis for the interface. The card provides the address decoding and buffering for the board. Appendix B contains a schematic of the prototype card. Additional circuitry was added to provide additional decoding and data bus buffering from the PC bus to the Xilinx chips.

The frame grabber board is I/O mapped in the PC address space at locations 0x0300H-0x030FH. These locations were chosen to avoid any conflict with existing cards. The

sixteen port locations, defined in Table 3.1, provide enough locations to perform all of the needed functions.

Table 3.1: I/O addresses used by frame grabber board.

ADDRESS	USAGE
hex 300	Input port used for Xilinx chips
301	Latched output port used for configuring FPGAs
302	Programmable clock register
303	Reserved for expansion
304	XC 3042 programming port
305-307	Video capture parameter registers
308	XC 3090 programming port
30A	XC 3090 expansion
30B	XC 3090 data port
30C-30F	Image parameters for Splash data transfer

The input port 0x0300H is used to transfer information from the Xilinx chips to the PC during and after configuration. The output port 0x0301H is used to drive control pins on the Xilinx chip during configuration. The actual configuration data is sent to output ports 0x0304H and 0x0308H. The ports used for configuration are described in more detail in Section 3.3.3.

Port locations 0x0305H through 0x0307H are used to set the capture parameters for the video interface as described in Section 3.4.3. Port locations 0x0302H and 0x030AH through 0x030FH, used to set the data transfer rate and other parameters for the Splash interface, are described in more detail in Section 3.5.

When writing data to the internal registers of configured Xilinx chips, care must be taken to avoid the loss of data. Although both of the Xilinx chips have synchronous designs, the data must initially be latched asynchronously since the clocks used by the Xilinx chips are not synchronized with the PC system clock. The rising edge of I/O_W* should be used to latch the data onto the chip and then the data can be synchronized with the chip clock.

3.3.3 Configuration Sequence for Xilinx FPGAs

The JB1 frame grabber board uses two Xilinx field programmable gate arrays. An XC 3042 is used to interface the video digitizer to local memory. An XC 3090 is used to interface the PC and Splash to the local memory. As described in Appendix A, FPGAs must be configured upon power-up and when a new hardware configuration is desired. This programming consists of downloading a serial bitstream into the SRAM configuration memory of each Xilinx chip. The configuration contains the information needed to implement the desired logic in the FPGA. Creation of the bitstream and the configuration process is discussed in subsequent paragraphs and in Appendix A. The PC-JB1 interface described in Section 3.3.2 is used to transfer the configuration data from the PC to the Xilinx chips.

Development of the bitstream is a four step process. The flowchart in Figure 3.5 shows the necessary steps. Additional details are given in Appendix A.

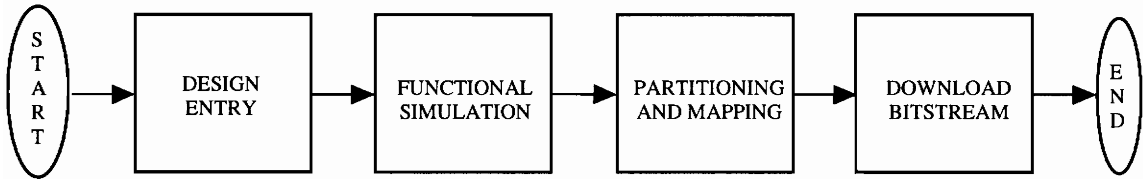


Figure 3.5: Four step process for bitstream development [17]. The resulting bitstream file is used to configure the FPGA with the desired logic function.

The first step is to specify the logic for the chip using one of the schematic capture programs which supports Xilinx chips. The schematic contains the logic which will be implemented using the chip. The components of the schematic are Xilinx primitives, pulled from a library of parts. Pin assignments are also defined at this time. Care should be taken to produce a pin assignment which results in a good data flow. Appendices C and D show the pin mappings for the XC 3042 and XC 3090 chips used on the frame grabber board.

Step two is to initiate a functional simulation of the design. At this time, no timing simulation may be performed since the actual routing of the chip has not yet been determined. After successful simulation, the design may be partitioned and mapped (step three) into the desired chip using Xilinx place and route software. The resulting file is the bitstream, ready for downloading. Additional information is also available for back-annotation to the schematic. Pin assignments determined by the automated software should be locked into the schematic so subsequent routings do not reassign the pins to new locations. The back annotated data also provides timing information based on actual placement and routing for accurate timing simulations.

The bitstream is downloaded to the logic devices during step four of the configuration process. The PC-JB1 interface described in Section 3.3.2 is used to transfer the bitstream

to the Xilinx chips. Several methods exist for transferring the data to the logic devices. The mode used is determined by the state of three mode pins on the Xilinx chip. For the FPGAs on JB1, the peripheral programming mode was chosen by hardwiring the mode pins to 000. In the peripheral mode, data is transferred byte-wise in a polled fashion from the host to the Xilinx chip, where it is converted back into a bitstream. The following signals from the Xilinx chip are used during the transfer process:

DONE/PROG* Bi-directional signal

- As an active low input to Xilinx chip, it clears the configuration memory.
- As an active high output from Xilinx chip, indicates configuration completed.

INIT* Bi-directional signal

- As an output, held low during initialization portion of configuration.
- As an input, can hold device in a wait state during initialization.

RDY/BUSY* Output signal

- Indicates when device is ready to accept another byte of configuration data.

Port locations 0x0300H and 0x0301H are used to write and read the control data to and from the Xilinx chip control pins DONE/PROG*, INIT* and RDY/BUSY*. The bits of each port are summarized in Table 3.2. Configuration data is written to the Xilinx chips from the PC by writing to port 0x0304H for the XC 3042 and to 0x0308H for the XC 3090. These ports are available for general use after the configuration process has been completed.

Table 3.2: Control port description.

PORT	USAGE
300H	Input port
	bit 0: RDY/BUSY* 3042
	bit 1: IMG_XFR
	bit 2: DONE/PROG* 3042
	bit 3: INIT* 3042
	bit 4: RDY/BUSY* 3090
	bit 5: IMG_RDY
	bit 6: DONE/PROG* 3090
	bit 7: INIT* 3090
301H	Latched output port
	bit 0: DONE/PROG* 3042 (open collector)
	bit 1: RESET* 3042
	bit 2: UNUSED
	bit 3: UNUSED
	bit 4: DONE/PROG* 3090 (open collector)
	bit 5: RESET* 3090
	bit 6: UNUSED
	bit 7: UNUSED

The programming sequence follows the flow chart shown in Figure 3.6. The first step in the process is to read the DONE/PROG* line to check if the device has been previously programmed to prevent accidentally overwriting a desired configuration. The next step is to begin the initialization process by driving the reset line active for two clock cycles. The INIT* line is then polled until it goes inactive, indicating the completion of the initialization process. Once complete, data is written out to the FPGA one byte at a time. The RDY/BUSY* line must be polled to determine when the FPGA is ready for another byte. Once all of the data has been written out, the DONE/PROG* line should be read to verify the configuration process was successful. All of the steps are controlled by the device driver program *grabber* listed in Appendix E and described in chapter 4.

For additional information on the peripheral programming mode, consult reference [17].

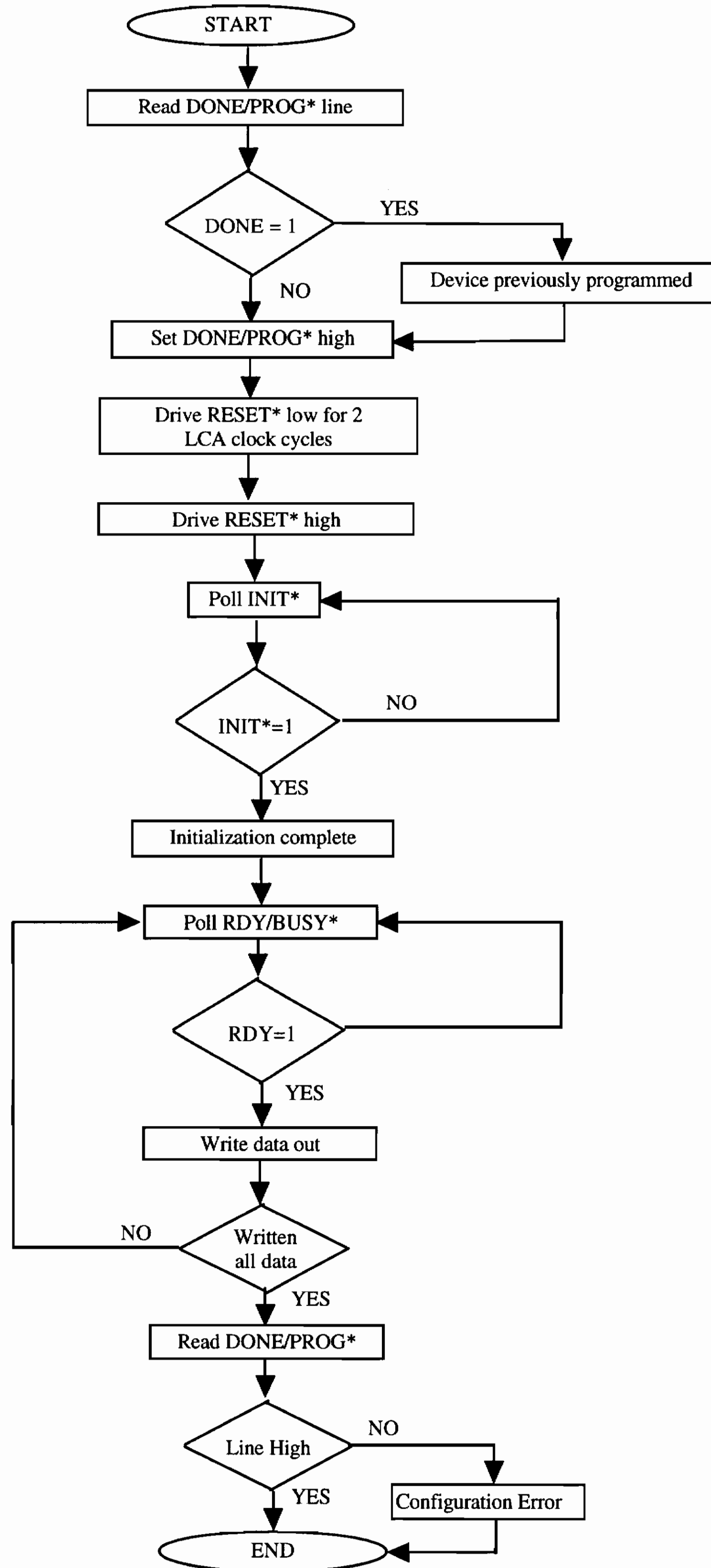


Figure 3.6: Configuration flowchart [17]. The peripheral mode uses a polled configuration sequence.

3.4 Video Camera Interface

The video camera interface must digitize the analog RS-170 video signal and store the resulting picture elements (pixels) in local memory. A commercially available hybrid video chip is used to convert the analog signal into digital pixels. Details of the video format are described in Section 3.4.1. Sections 3.4.2 and 3.4.3 describe the video chip and interface, respectively.

3.4.1 Introduction to RS-170 Video Format

Many video cameras in the United States produce video signals using the EIA RS-170 format [18]. In this format, a new frame is produced every 1/30th of a second (33 ms). Each frame is further divided into two fields: an even field and an odd field. As shown in Figure 3.7, the two fields are normally interlaced. Every other line belongs to one field; the remaining lines belong to the second field. Fields are generated alternately, with one field created every 1/60th of a second. The interlacing of the two fields helps eliminate flickering.

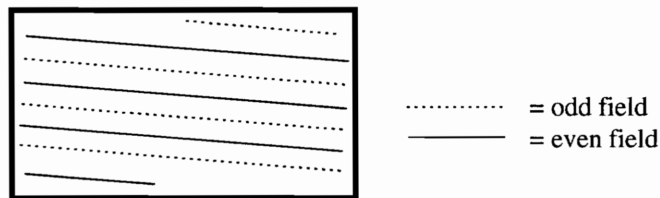


Figure 3.7: Image interlacing. The fields are alternately displayed in their entirety to reduce image flickering.

Each frame is composed of 525 video lines. Approximately 485 of the lines contain active video information and the remaining 40 lines are used for control. The 485 lines are

divided equally between the two fields. Control data must be included in the signal to differentiate between frames, fields and the 242.5 lines of each field. A single image line of the video is illustrated in Figure 3.8. This signal contains the data as well as the control and timing information.

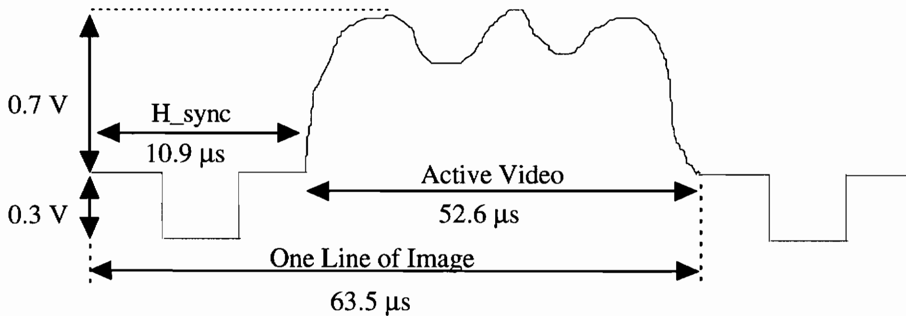


Figure 3.8: Representation of one RS-170 Line [19]. The Horizontal Sync pulse, H_sync, indicates the beginning of a line of video. The active video data follows the sync pulse and is contained in the 0.3 V to 1.0 V range.

The line frequency is 15.750 kHz (30 frames per second × 525 lines per frame). Each line has a duration of 63.5 microseconds. The beginning 10.9 μs of each line is used for horizontal blanking and does not contain valid image data. The remaining 52.6 μs of the line contains valid data.

The aspect ratio of an image refers to the ratio of the width of the image to the height of the image. The standard aspect ratio for RS-170 is 4:3. For a 4:3 aspect ratio, the entire 52.6 μs of the active video signal for a line is collected. For certain image processing applications, a 1:1 aspect ratio is preferred over the standard aspect ratio. This can be achieved by ignoring 13.6 μs of the actual line data to get a square image, resulting in only 39 μs of data being used per line. The image can be centered by dividing the 13.6 μs of dead time equally around the 39 μs of data being captured. Figures 3.9 and 3.10 show

both aspect ratios in terms of the number of pixels for a clock frequency of 9.828 MHz and in terms of time, respectively. Note that for the 1:1 aspect ratio, the number of pixels in a row is not equal to the number of pixels in a column since the pixels are not actually square. Both aspect ratios are supported by the JB1 frame grabber board.

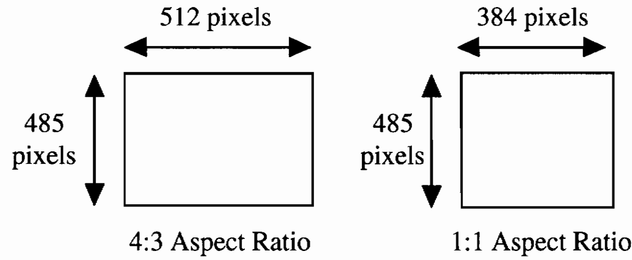


Figure 3.9: Image sizes for 4:3 and 1:1 aspect ratios. The 1:1 aspect ratio results in a smaller image.

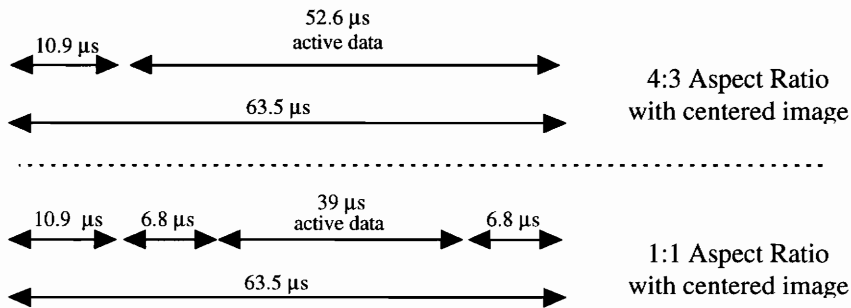


Figure 3.10: Line timing for 4:3 and 1:1 aspect ratios. The 1:1 image is centered by dividing the 13.6 µs of dead time equally between both sides of the active data.

3.4.2 Video Digitizer Chip

A video digitizer chip is used to convert the analog video signal into a stream of digital pixels. The AD9502 Video Digitizer chip from Analog Devices [20] is a hybrid circuit which contains all of the analog hardware needed to decode the video signal. The

resulting data stream is composed of 8-bit pixels, giving a gray scale image ranging in intensity from 0 (white) to 255 (black). Note that the values must be inverted to have the standard format of 0 representing black and 255 representing white.

The input to the chip is the RS-170 video signal. The outputs from the chip include a pixel clock, a vertical sync pulse V_sync, a horizontal sync pulse H_sync and the 8-bit video data stream. The pixel clock is used for timing and operates at a fixed frequency of 9.828 MHz. A slower clock frequency of 7.3 MHz was available but the number of pixels per line of video would have been greatly reduced (512 pixels for 9.828 MHz versus 384 pixels for a 7.3 MHz for an active line time of 52 μs). During the active video time, the pixel clock is used to distinguish the different pixels of the image.

The V_sync signal represents the start of a new field. The H_sync represents the start of a new line. The relative timing of the pulses is used to determine which field is being output. For the odd field, the first H_sync pulse occurs 64 μs after V_sync. For even fields, the first H_sync pulse occurs after 31 μs after V_sync. Figure 3.11 shows the relationships between the pulses for the even and odd fields.

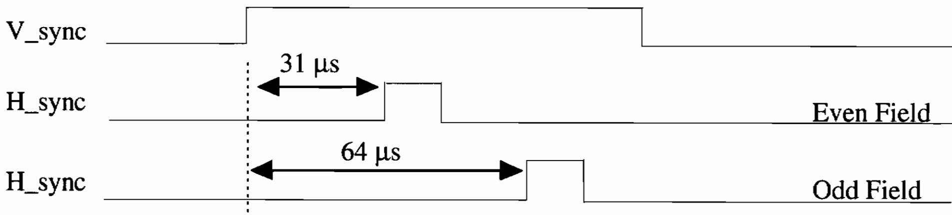


Figure 3.11: Timing for even and odd fields. The even and odd fields can be differentiated by the time between the V_sync pulse and the first H_sync pulse of the field.

3.4.3 Video Interface Details

The block diagram of the video interface is shown in Figure 3.12. There are two main sections to the interface: the video chip and the control logic. The video chip uses the video signal to produce the pixel stream, control signals V_sync and H_sync, and the pixel clock. The control logic decodes the V_sync and H_sync signals from the video chip and provides the control signals necessary to store the data in the local memory. The control logic consists of state machines synchronized to the video chip pixel clock.

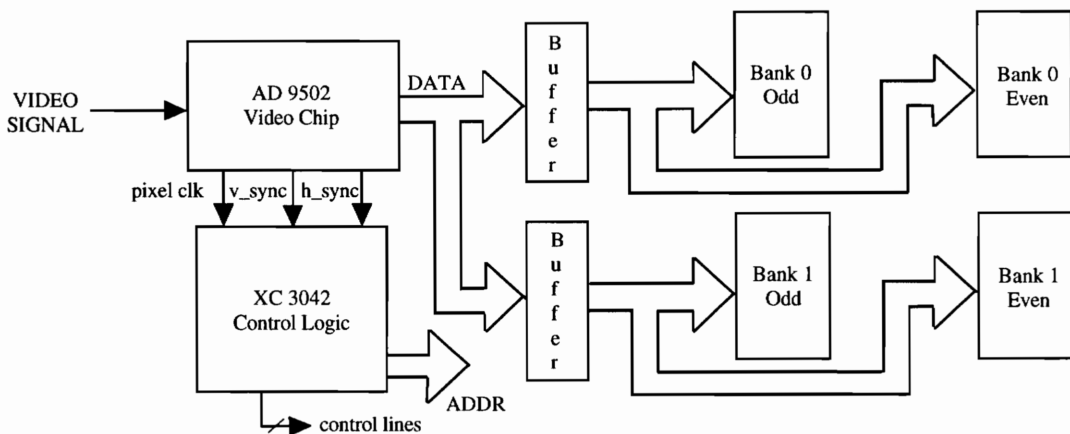


Figure 3.12: Video interface block diagram. The video interface consists of the video chip and the control logic and is connected to the local memory banks.

To achieve real time operation, an entire frame of data must be stored in 1/30th of a second. For an image of 485 pixels by 512 pixels, the minimum clock speed that can accomplish this is 7.5 MHz. The video chip pixel clock, running at 9.828 MHz, provides a suitable clock.

There are two memory banks, each consisting of two CMOS 128K × 8 bit static RAM devices. The memory chips, manufactured by Paradigm [21], have a 25 ns access time. The size of each bank of memory was based on the maximum image size. Since one field of the image is composed of a maximum 242.5 lines with 512 pixels per line, an entire field of either a 1:1 or 4:3 aspect ratio image can be stored in memory chip. An entire frame is then stored in one bank of memory. The fast access time allows real time data capture.

Interfacing to the video camera now becomes the task of capturing the data from the video chip. The process of capturing a frame is broken down into capturing two fields. The flowchart for capturing each field is shown in Figure 3.13.

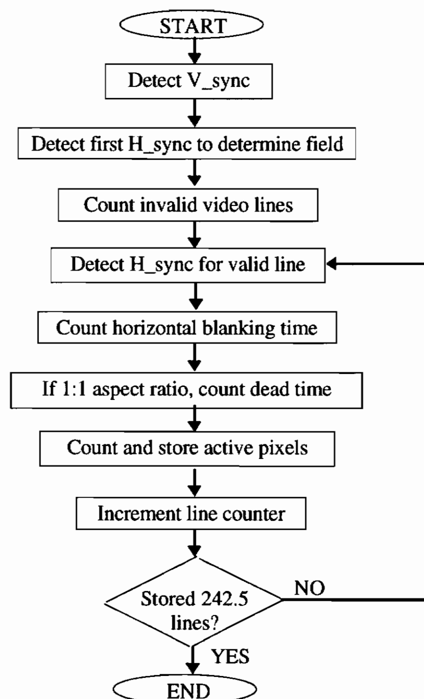


Figure 3.13: Field digitization flowchart [22]. Each line of active video for a field must be captured from the continuous video pixel stream.

The vertical sync pulse, horizontal sync pulse and the pixel clock are brought into the XC 3042. The data stream is connected directly to the memory banks through buffers enabled by the control signals generated in the XC 3042. The XC 3042 also generates the address signals for the memory devices.

The block diagram in Figure 3.14 shows the overall system used to implement the video capture system. The video interface is composed of many small sub-blocks. Information is passed between blocks through flags. Using a top-down design approach, each sub-system was independently designed and simulated before joining all of the parts together. The entire synchronous system was implemented on a single XC 3042 field programmable gate array using the pixel clock. Whereas many discrete components would be needed to implement the design, the FPGA provides a compact way of implementing all of the needed counters on a single chip. The XC 3042 was chosen based on a rough estimate of the number of flip-flops needed and the minimum speed required to implement the design.

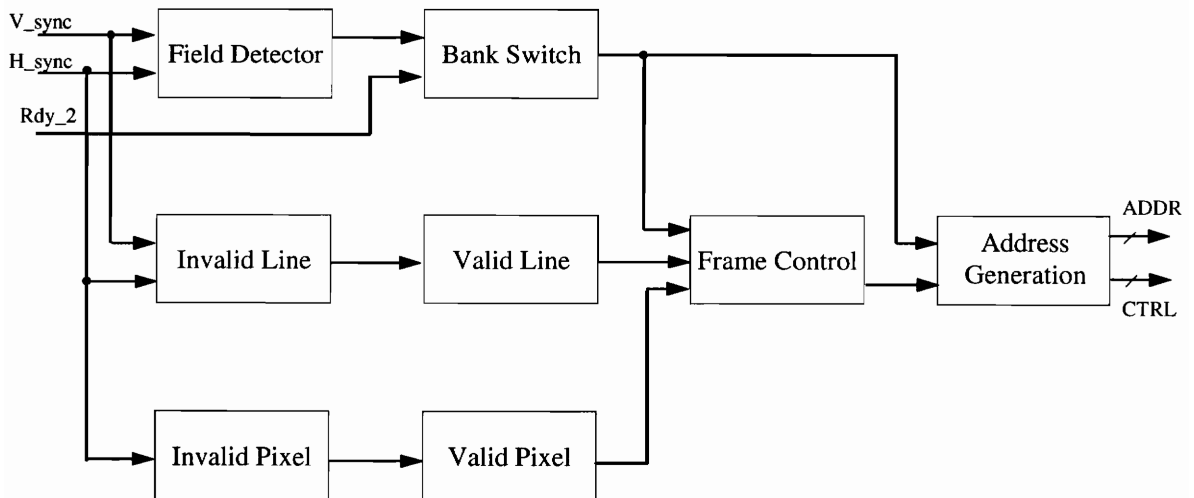


Figure 3.14: Capture system block diagram. The video interface broken down into individual state machines, represented here by rectangles. Each FSM performs one part of the capture process.

At the start of a field, the Field Detector determines which field of data is being presented. At the same time, the Bank Switch block switches the data path to an empty memory bank and resets the Address Generation block. The Invalid Line block then counts out the invalid image lines. Upon the completion of the invalid line count, the Valid Line block counts the number of valid lines. For each valid line, the Invalid Pixel block counts out the invalid data. Upon completion of the invalid data count, the Valid Pixel block starts counting out the valid pixel data. For each valid pixel, the Frame Control block enables the Address Generation block. The Address Generation block then creates the address and control strobes needed to store the valid data in memory. Each block is further described below.

The Bank Switch block coordinates the bank switching between the video interface and the Splash interface. Each interface accesses separate banks at the same time. While the video interface is filling one bank with the current frame being captured, the Splash interface is transferring the frame already stored in the other bank to Splash. After both interfaces complete their operations on their current frames, the two interfaces can switch banks. Both sections must be ready to switch memory banks before a switch can occur to prevent data from being lost or overwritten. The camera interface is ready to switch after completing the capture of an entire frame. The entire frame is captured when the Valid_Line flag goes inactive for the second field. The Splash interface is ready to switch after completing the transfer of a complete frame to Splash, indicated by Rdy_2 going active. The bank switching arbitration is performed through handshaking between the two interfaces. The Bank Switch block prevents memory contention between the two interfaces by allowing each interface access to only one bank of memory at a time.

Upon detecting a vertical sync pulse, indicating a new field, the first step is to determine which field of data is being presented. At the rising edge of V_sync, a counter is reset and enabled. The counter is disabled when the first H_sync pulse is detected. Approximately 624 pixel clocks (64 μs) will have passed if the field is odd; 312 pixel clocks (31 μs) if the field is even. The output from the Field Detection block is used to control to which memory chip in the enabled bank the data is written.

After detecting the start of a new field, the invalid video lines must be counted and ignored. The Invalid Line block performs this function by counting H_sync pulses. Upon completion of the count, the Valid Line block enables the Valid Line counter for 242 horizontal sync pulses. At the start of each new line, the Invalid Pixel block begins counting pixels. Depending on the aspect ratio, either 108 or 174 pixels will be counted and ignored. Upon completion of counting, the Valid Pixel block will begin counting and will set the Valid_Pixel flag. Both of the counters in these blocks are programmable via the ports listed in Table 3.3 to allow for various image sizes. Port location 0x0304H is used to determine the number of pixels which should be ignored for each line. This number includes the invalid pixels which occur before the valid pixels in a line and the pixels to be ignored for the 1:1 aspect ratio, if applicable. Port locations 0x0306H and 0x0307H are used to determine the number of pixels which should be collected for every line. The Frame Controller block uses the Valid_Line flag and the Valid_Pixel flag to create the Valid_Data flag. If the Valid_Data flag is active, the Address Generation block will be enabled. The Address Generation block provides the address and control strobes for the memories.

All of the state machines for the XC 3042 are listed in Appendix C.

Table 3.3: XC 3042 internal port usage.

PORT	USAGE
304	Configuration / control port
305	Invalid pixel mask register
306	Lower byte of valid pixel register
307	Upper byte of valid pixel register

3.5 Splash Interface

Referring again to Figure 3.1, the Splash interface transfers captured video images from local memory to the Splash system. Additionally, the interface allows the data to be transferred to and from the PC to verify hardware operation and to set transfer parameters. The logic for the Splash interface is contained in one XC 3090 FPGA. The block diagram for the interface is shown in Figure 3.15.

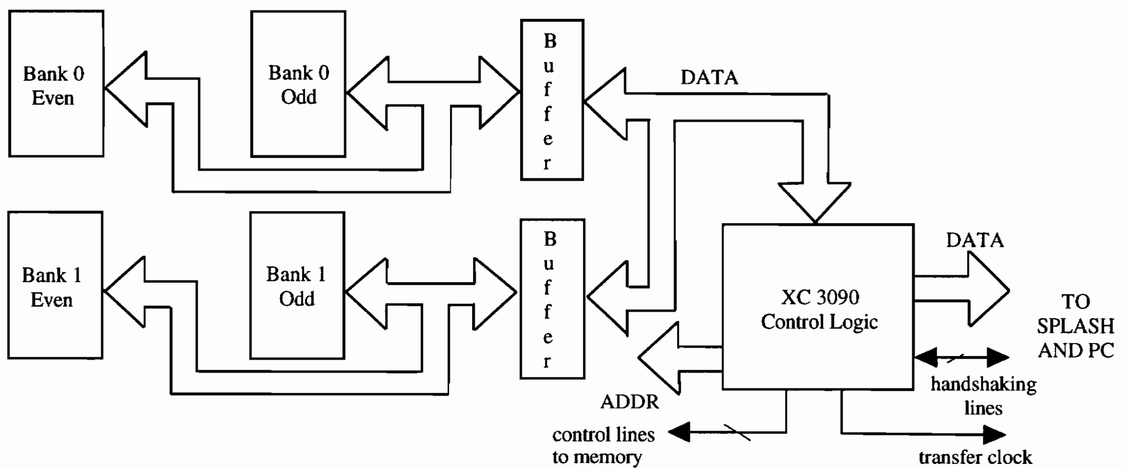


Figure 3.15: Splash interface block diagram. The interface consists of the control logic and is connected to the local memory banks. The interface transfers frames from the memory to the Splash system and the PC.

The entire Splash interface is a synchronous system with a clock independent of the PC and of the video capture system. The clock control register for the Splash transfer clock is programmable from the PC with an I/O address of 0x0302H. Table 3.4 identifies the individual bits of the clock control register.

Table 3.4: Programmable clock control register.

BIT	NAME	FUNCTION
0	B0	Least significant bit of clock divider
1	B1	Middle significant bit of clock divider
2	B2	Most significant bit of clock divider
3	EN	Clock enable (1 = on)

The Splash transfer clock can be varied from 0.078 MHz to 10 MHz using the following formula:

$$FREQ_{clk} = \frac{20}{2^{n+1}} \text{ MHz} \qquad \text{where } n = 4B_2 + 2B_1 + B_0$$

Running the clock at frequencies less than 10 MHz results in a non-real time system. Image frames will be lost as the video capture system must wait for the video transfer system to empty a memory bank. No new frames of data will be captured until the video transfer system finishes emptying the bank.

The frame grabber board is connected to Splash via a 50 pin cable. The cable is specified in Appendix B. The current data bus to Splash is 8 bits wide. The 50 pin cable will allow for expansion up to 36 bits. The remaining lines are used for control lines. To facilitate data transfers from JB1 to Splash, a transfer protocol was developed using handshaking. Figure 3.16 illustrates the protocol.

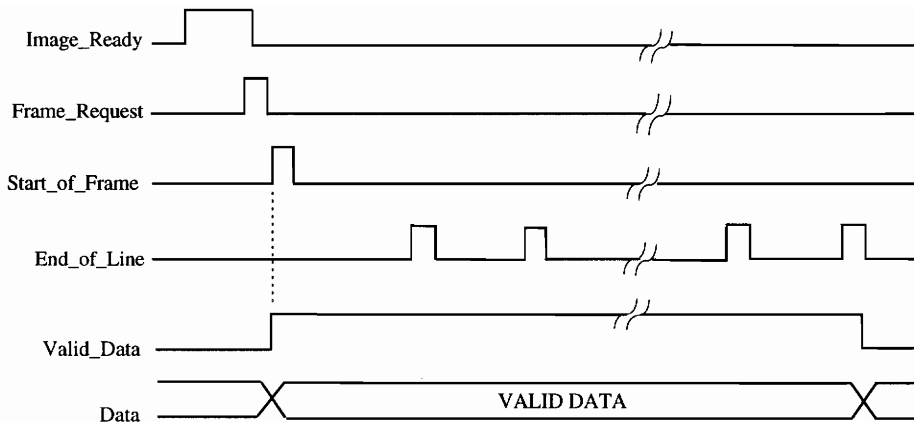


Figure 3.16: JB1 to Splash data transfer protocol. Frames are transferred from the frame grabber board to the Splash system using a handshaking protocol.

After a valid bank switch occurs, the Image_Ready flag in the XC 3090 is set high. Upon receiving a Frame_Request from Splash, the Image_Ready flag is set inactive. A Start_of_Frame pulse is then issued. The Valid_Data flag is also set high when the Start_of_Frame pulse is issued. A pixel of the valid image data will then be transferred at every Splash transfer clock cycle. The last pixel of a line is indicated by a simultaneous pulse on the End_of_Line signal. Once the entire image is transferred, the Valid_Data line goes inactive and the entire process starts again.

The logic that controls the interface comprises three main sections: the interface to Splash and the PC, the controller and the address generation unit. The block diagram for the XC 3090 logic is shown in Figure 3.17. The state machines for each piece of the interface are listed in Appendix D.

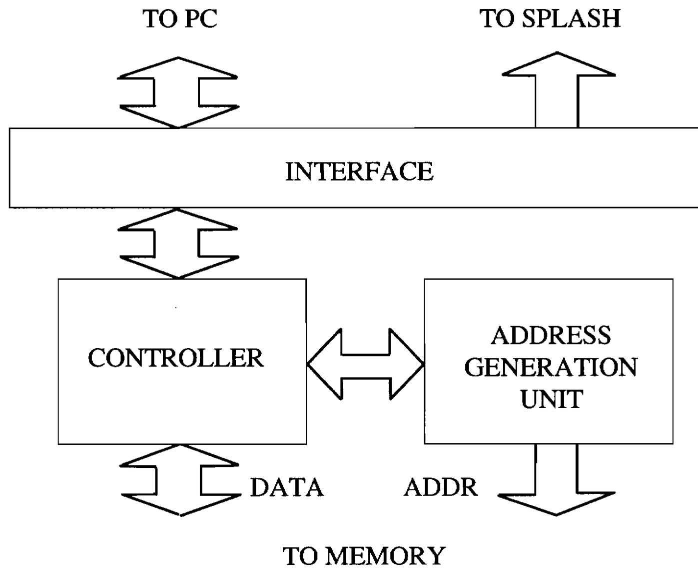


Figure 3.17: Transfer system block diagram. The system controls the transfer of data from the local memory to the Splash system or the PC host.

The interface portion connects the XC 3090 to the PC and to Splash. The interface consists of address decoders to write to local ports, data buffers and a control register. Internal ports can be written to by the PC as part of the PC's I/O address space. The ports are used to set transfer parameters and status bits. The port usage of the XC 3090 is shown in Table 3.5.

Table 3.5: XC 3090 internal port usage.

PORT	NAME	USAGE
Hex 308	PS0	Config / low byte number pixels per line
309	PS1	High byte number of pixels per line
30A	PS2	Reserved for expansion
30B	PS3	I/O address for local memory
30C	PS4	Number of lines per image to transfer
30D	PS5	Reserved for expansion
30E	PS6	Reserved for expansion
30F	PS7	Control port

The control port 0x030FH also allows the PC to access the local memory. Table 3.6 lists the function of the bits. The PC can read the current image in a particular bank of memory or fill a particular bank with any character. The PC transfer does not use the same protocol as the Splash transfer. No start of line or end of line indicator is used. Instead, only the start of frame bit is used. The software running on the PC must keep track of the data to determine when a new row is being started by counting pixels. A pixel is transferred every I/O read cycle to port location 0x030BH. Note that the data direction bit must be set to 1 to send images to Splash.

Table 3.6: Control register bit definitions.

BIT	FUNCTION
0	Data direction: 0 = write; 1 = read
1	Request for new image
2	Bank request
3	Chip reset
4-7	unused

The Controller block controls the data transfer process. Inputs to the block include the Splash transfer clock, a transfer request flag and handshaking signals from the XC 3042. The handshaking signals are used for bank switching. The block outputs are handshaking signals to the XC 3042 and frame control signals. Upon receiving an image request and having a valid image to transfer, a new frame pulse is generated and the image transfer flag is set high.

The Address Generation Unit (AGU) creates the addresses needed to access local memory. The same block is used for both writing and reading from memory. The AGU must be programmed for the desired number of pixels per line and lines per image by

writing to the ports listed in Table 3.5. The values written must match the parameters selected for the video capture interface or else the image will be skewed. The AGU works by accessing all of the locations for one line of the image in a row. Since the odd field is stored in one SRAM and the even field in a second SRAM, the whole image is obtained by swapping between the two memory chips every line.

Upon receiving a new frame pulse from the Controller, the Address Generation Unit resets all of its counters. At each clock pulse, the address is incremented by one location. The address data leaves the XC 3090 through two sets of buffers, each going to a bank of memory. Only one buffer is enabled at a time, allowing the other bank to be driven by the XC 3042. When the final count is reached, the AGU sets a flag to indicate to the controller that the data transfer is complete.

Chapter 4
JB1 Software

4.0 Introduction

The software utility *grabber*, written as part of the JB1 project, controls the operation of the frame grabber board. *Grabber*, a menu driven program written in C for the DOS environment, is listed in Appendix E. It performs all of the software functions needed to use the board. This utility includes a method for configuring the FPGAs with an initial configuration after power up, and at any time thereafter. The program is also used to set the board parameters under user control. Parameters include image size, Splash transfer clock frequency, memory fill characters and a Splash enable. Once the board is configured and the proper parameters are loaded, the board is free running and *grabber* can be exited without affecting the operation of JB1. The program also can be used to verify correct hardware operation by performing read and write operations to the local memory. Camera images may also be captured and displayed using the program. Section 4.1

describes the function blocks of the program. Section 4.2 is a brief user's manual for the program and describes all of the utility's options.

4.1 Utility Function Blocks

The utility is composed of sixteen function blocks. The functionality of each block is briefly described below. For additional information, consult the source code listing in Appendix E.

- **main**

  The main block calls the menu function block.

- **menu**

  Menu structure for the utility. All of the functions for configuring the board are called from this routine.

- **auto_config**

  Automatically configures the entire board with a set of default configurations and parameters. The Xilinx chips are configured by calling a second block named prog_xc. After configuring the FPGAs, the image size is set to 480 × 512 pixels and the Splash transfer clock frequency is set to 10 MHz. The image size uses 480 rows to prevent any invalid data bits from being transferred.

- **prog_xc**

  When the auto_config routine is invoked, it calls prog_xc to program the Xilinx chips. The default bitstream files (XC3042.pod and XC3090.pod) created from the logic synthesis software are downloaded to the two programmable logic devices. Data is transferred to the board in a polled fashion.

- **download**

  Configures the logic devices when the auto_config routine is not used. The bitstream file selected by the user is downloaded to the specified chip. This routine should be used when only one FPGA needs to be configured.

- **parameters**

  Allows the user to change the capture parameters. A 1:1 or 4:3 aspect ratio may be chosen by the user, corresponding to image sizes of 480 × 384 pixels and 480 × 512 pixels respectively. For the 1:1 case, some of the valid picture data of each line will be ignored as explained in Chapter 3. The user can select a centered or left justified slice of the full-size image.

- **capture**

  Captures a series of frames from the video camera and displays them on the PC's video monitor. The images will not be sequential images since the PC can not read the images fast enough from the frame grabber card. This routine provides verification of the video capture hardware.

- **graphics**

  Allows images previously captured by read_mem to be displayed on the monitor.

- **clk_setup**

  Sets the frequency of the programmable Splash transfer clock on JB1 to the clock frequency selected by the user by writing to the clock register on JB1.

- **memory**

  Performs read and write operations to the local JB1 memory from the host PC. The I/O operations are performed through two routines, read_mem and write_mem.

- **reset_xilinx**

  Resets the programmable devices after the configurations have been downloaded. Both Xilinx chips are driven into the reset state to clear all flip flops. The reset is then released and the board begins normal operation.

- **write_mem**

  Fills the local JB1 memory with a fill pattern for memory testing and as fill patterns for padding video images. Data is written one byte at a time. Upon completion of transfer, the number of bytes of data transferred is displayed.

- **read_mem**

  Reads a complete image from a bank of local JB1 memory and writes the data to a data file. Data is transferred one byte at a time.

- **get_key**

  Gets a value from the keyboard buffer and returns it to the calling routine.

- **file_open**

  Displays files with the selected extension and opens the file selected by the user.

- **dis_sp**

  Enables and disables the transfer of data to Splash.

4.2 Grabber User's Guide

The utility program is menu driven and may be executed from a DOS command line by typing "grabber" and then pressing return. Sections 4.2.1 through 4.2.10 provide instructions for each of the menu options.

4.2.1 Autoconfiguration

Select option 0 to perform an autoconfiguration of JB1. The routine will automatically program the Xilinx chips with a bitstream configuration which will permit data transfers to Splash. A default image size of 480 × 512 pixels with a Splash transfer clock frequency of 10 MHz will be selected. The Splash Enable bit, defined in table 3.5, will also be set active and the Xilinx chips will be reset. If any errors are encountered during the configuration process, an error message will be displayed.

4.2.2 Download Configuration to LCA

Option 1 allows the user to download a new configuration to either of the Xilinx chips at any time. After selecting this option, the user must select which Xilinx chip is to be configured. Next, a specific configuration file must be selected from a list of valid configuration files. All configuration files must have a .pod extension. The software will then configure the appropriate device. If any errors are encountered, an error message will be displayed.

4.2.3 Change Capture Parameters

Option 2 allows the user to change the image capture parameters by varying the aspect ratio. A 4:3 or 1:1 aspect ratio may be chosen by selecting the appropriate line from the sub-menu. The 4:3 aspect ratio will have an image size of 480 × 512 pixels whereas the 1:1 aspect ratio will have an image size of 480 × 384 pixels. If the 1:1 aspect ratio is chosen, only part of the possible image data will be used. In this case, the user must select

whether to use the leftmost part of the image or the center part of the image by selecting either left-justified or centered from the list of options.

4.2.4 Continuously Capture Images

Option 3 will initiate a sequence of frame transfers from the board to the PC's monitor. Since the transfer rate to the PC is not performed in real time, sequential images will not be displayed. Before selecting this option, transfers to Splash must be disabled using option 8 of the main menu as described below.

4.2.5 Display Image

Option 4 allows the user to display a previously captured image on the PC monitor. The user must select the image to be displayed from a list of files. Valid image files must have a .raw extension.

4.2.6 Read/Write On-board Memory

Option 5 allows the user to read or write to a particular bank of the local memory. A submenu will appear prompting the user to select whether to read or write to memory. After selecting the desired function, the user must select which bank is to be accessed. If the read option is selected, the data will be read and then stored in *temp.raw*. The file may then be viewed by selecting Option 4 from the main menu. If the write option is selected, the fill pattern must then be selected. The pattern may be all white, all black or a checkerboard pattern. The solid colors are useful for filling unused portions of the

memory with a known value before images are transferred to Splash. The checkerboard pattern is useful for verifying correct hardware operation. Before selecting this option, transfers to Splash must be disabled using Option 8 as described below.

4.2.7 Change Clock Setup

Option 6 is used to change the setup of the Splash transfer clock. The clock frequency may be varied by selecting Option 1 of the submenu. Available clock frequencies are 10 MHz, 5 MHz, 2.5 MHz, 1.25 MHz, 0.625 MHz, 0.3125 MHz, 0.156 MHz, and 0.078 MHz. The clock is used only to control the transfer rate of data to Splash as described in Section 3.5. Image frames are always collected at the fixed clock rate of 9.828 MHz. Note, however, if the clock rate is set to less than 10 MHz, frames will be lost since the system can no longer perform real time transfers to Splash.

4.2.8 Reset Xilinx Chips

Option 7 allows the user to reset the Xilinx chips. This step should be performed after downloading new configurations to the Xilinx chips to put the board in a known state.

4.2.9 Enable/Disable Splash

Option 8 allows the user to enable and disable data transfers to Splash. To transfer images to the PC, the Splash interface must be disabled by selecting Option 1 of the submenu. To allow data transfers to Splash, the interface must be enabled by selecting Option 2 of the submenu.

4.2.10 Exit Program

Selecting Option 9 exits the program and returns the user to the DOS prompt.

Chapter 5
Future Enhancements

5.0 Introduction

The JB1 frame grabber board can currently capture RS-170 format video images and transfer the images to Splash in real time. Additionally, the board can transfer the data to Splash at lower speeds, allowing slower image processing routines to be performed on non-real time data streams. The current configuration of JB1 transfers images to Splash using the protocol outlined in Section 3.5. Simple changes would make the board more versatile. With JB1's FPGA based architecture, additional designs are easy to implement. Section 5.1 discusses some potential modifications. Section 5.2 outlines the steps needed to implement the new designs.

5.1 Alternate Configurations

The current design transfers images to Splash in bursts of data. Handshaking between JB1 and Splash must occur to initiate the transfer of a frame. Once an image transfer is started, the complete image is transferred one pixel every Splash transfer clock cycle. Upon the completion of the image transfer, the handshaking must be repeated to begin the cycle again. This causes some dead time in between the transfer of frames. JB1 could be easily modified to perform the following:

1. Transfer several pixels per clock cycle.
2. Transfer pixels continuously without the handshaking.
3. Transfer only 1 field per frame.

Configuration 1 would allow for several pixels to be transferred per Splash clock cycle. Since each pixel is only 8 bits and the data bus from JB1 to Splash is 36 bits wide, up to four pixels could be transferred every Splash clock cycle. The local clock used on JB1 to fetch and organize the data would need to be faster (4 times faster for a 4 pixel wide transfer) than the Splash clock to allow enough time to fetch the data for each transfer from memory. This configuration would create a slower Splash clock, resulting in more time per cycle to process the data.

Configuration 2 would allow for a continuous flow of pixels from JB1 to Splash. Once the initial transfer is started, no cycles would be used for a handshaking process. Every cycle would have a pixel being transferred and the image processing algorithms would have an uninterrupted stream of data to process.

Configuration 3 would allow for the transfer of only one field per frame, reducing the amount of data which would need to be processed every image. Furthermore, the clock to Splash could run at half the rate and still have real time processing. This configuration would be useful for image processing algorithms which would take too much time to perform if all of the data were transferred.

The current design does not use all of the available resources on the Xilinx chips. On the XC 3090, the unused logic could be used to pre-process the images before they are sent to the Splash-2 system. Basic image processing tasks such as image thresholding could easily be performed as the data is transferred from the local memory to Splash. Instead of transferring the actual video data, the threshold data would be transferred for each pixel of the image.

5.2 Implementation of New Designs

All of the above configurations can easily be implemented without any physical changes to the hardware. The buffers needed for the expanded data bus are already located on the board and are wired into the connector and to reserved pins on the XC 3090. All of the hardware logic changes would be internal to the XC 3090 FPGA. The new logic configurations could be developed and downloaded into the chip using the configuration sequence outlined below.

1. Create schematic for new design using Xilinx primitives.
2. Simulate design.
3. Create netlist.

4. Place and route design using Xilinx XACT tools.
5. Create .mcs file.
6. Convert .mcs file to .pod file.
7. Download .pod file to the XC 3090 using the grabber utility.

Chapter 6
Conclusions

This thesis documents the design of the JB1 frame grabber board and its use in the VTSplash system. The board captures RS-170 format video images and transfers the digitized images to Splash in real time. The board is programmable to allow for a variety of image sizes to be transferred to Splash at a user selectable clock speed. The FPGA-based architecture of the board also allows for new configurations to be easily implemented as the needs of the system change. Several new configurations were suggested.

The VTSplash system has been used to perform image processing applications. The frame grabber board provides the system with a continuous flow of digitized video data, allowing for real time image processing to take place.

REFERENCES

1. D.A. Buell, J.M. Arnold, W.J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, Los Alamitos, California, 1996.

2. J.M. Arnold and M.A. McGarry, *Splash 2 Programmer's Manual*, Supercomputing Research Center, Bowie, Maryland, 1993.

3. P. Athanas and L. Abbott, "Real-Time Image Processing on a Custom Computing Platform," in *IEEE Computer*, February, 1995.

4. R. Elliot, "Hardware Implementation of a Straight Line Detector for Image Processing," project report, Virginia Tech, 1993.

5. L. Abbott, P. Athanas, R. Elliot, B. Fross, L. Chen, "Finding Lines and Building Pyramids with Splash-2," *IEEE Workshop on FPGAs for Custom Computing*, Napa, CA, April, 1994.

6. P. Athanas and L. Abbott, "Image Processing on a Custom Computing Platform," Fourth International Workshop on Field-Programmable Logic and Applications (FPL) '94, Prague, Czech Republic, September, 1994.

7. R. Rachakonda, "Region Detection and Labeling in Real-time Using a Custom Computing Platform," M.S. Thesis, Virginia Tech, 1994.

8. L. Chen, "Fast Generation of Gaussian and Laplacian Image Pyramids Using an FPGA-based Custom Computing Platform", M.S. Thesis, Virginia Tech, 1994.

9. N. Shirazi, "Implementation of a 2-D Fast Fourier Transform on an FPGA-based Computing Platform," M.S. Thesis, Virginia Tech, 1995.

10. N. Shirazi, P. Athanas, and L. Abbott, "Quantitative Analysis of Floating-Point Arithmetic on FPGA-based Custom Computing Machines," in *IEEE Symposium on FPGAs for Custom Computing*, Napa, CA, April, 1995.

11. A. Tarmaster, "Median and Morphological Filtering of Images in Real Time using an FPGA-based Custom Computing Platform," M.S. Thesis, Virginia Tech, 1994.

12. EIA Industrial Electronics Tentative Standard, No .1, November 7, 1977. Superseded by SMPTE 170M, 1994.

13. International Business Machines Corporation, *Technical References PC XT*, revised edition, 1983.

14. L.C. Eggebrecht, *Interfacing to the IBM Personal Computer*, second edition, SAMS, 1990.

15. E. Solari, *ISA & EISA: Theory and Operation*, Annabooks, San Diego, California, 1993.

16. Industrial Computer Source, *Model PB751-AT Reference Manual*, Manual NO. 00650-100-10A, San Diego, California, 1992.

17. Xilinx, Inc., *The Programmable Logic Data Book*, San Jose, California, 1993.

18. C. Poynton, *A Technical Introduction to Digital Video*, John Wiley and Sons, New York, 1996.

19. B. Jähne, *Digital Image Processing*, Springer-Verlag, New York, 1991.

20. Analog Devices, *Hybrid RS-170 Video Digitizer AD9502 Data Sheet*, Norwood, Massachusetts, 1989.

21. Paradigm, Inc., *1 Megabit Static Ram 128 K × 8-bit Data Sheet*, 1992.

22. Analog Devices, "The AD9502 Video Signal Digitizer and Its Application", *High-Speed Design Seminar Notes*, Norwood, Massachusetts, 1989.

Appendix A
Introduction to Xilinx FPGAs

This appendix contains a brief introduction to Xilinx field programmable gate arrays. Two Xilinx chips are used in the JB1 frame grabber board.

Introduction to FPGAs

Field programmable gate arrays are programmable logic devices which are configured by the user to implement a desired digital function. This is performed by downloading the configuration data into the device from a binary data file. This specifies the functionality of the device. Unlike EPROMs, no special voltages are needed to program the devices.

Xilinx FPGAs

Xilinx, Inc., produces CMOS reconfigurable field programmable gate arrays which support combinatorial and sequential designs. Reconfigurable chips allow for dynamic hardware changes and also for reduced design cycle time.

The Xilinx collection of FPGAs is divided into families based upon the number and types of on-chip resources and the operating speed. The various members within a family differ in terms of speed, resources, power consumption and packaging. The earliest family is the 2000 series of gate arrays. Later families include the 3000 and 4000 series of chips.

Xilinx Architecture

The structures of the Xilinx FPGAs are known as Logic Cell Arrays (LCAs) and are based upon the following four elements:

1. Configurable Logic Blocks
2. I/O Interface Blocks

3. Interconnection Resources

4. Internal Configuration Memory

A simplified architecture of the LCA is shown in Figure A.1. The Configurable Logic Blocks (CLBs) form a matrix in the interior of the device. The I/O Interface Blocks (IOBs) surround the CLBs. Interconnection resources are located between all of the CLBs and IOBs. Internal configuration SRAM, not shown in the diagram, is distributed throughout the LCA.

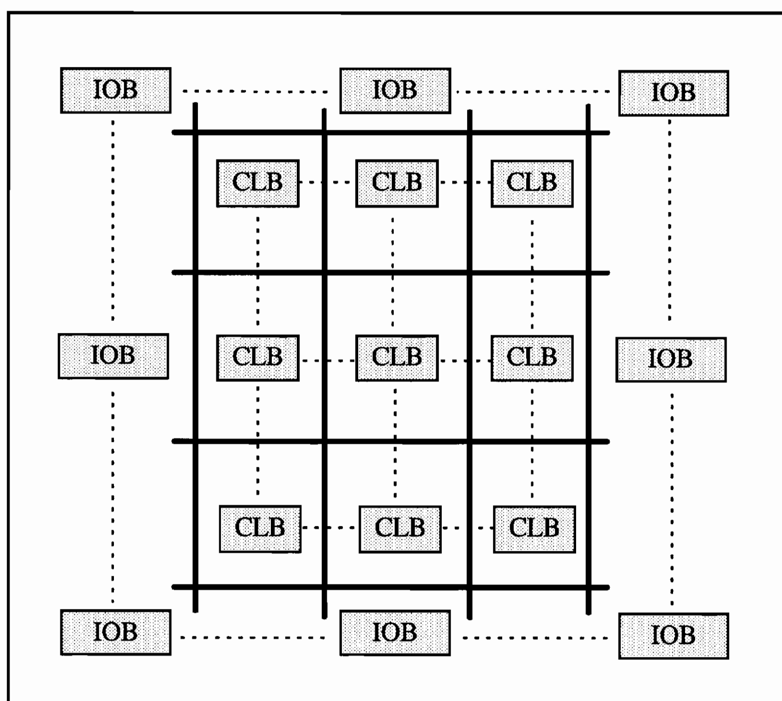


Figure A.1: Simplified Xilinx structure. The FPGA consists of a matrix of CLBs surrounded by a ring of IOBs with routing resources located throughout the chip.

The Configurable Logic Blocks are used to implement the logic desired digital functionality As shown in Figure A.2, CLBs consist of combinatorial logic, flip flops and control logic. Inputs to the block include logic inputs A through E and Din, a clock enable EN, a reset line RST and the clock line CLK. The block has two logic outputs, X and Y, which can be latched or combinatorial. Configuration bits inside the CLB allow for a large variety of possible configurations. Combinational logic functions inside the CLB are implemented using a look-up table, allowing for the same propagation delay regardless of the logic function being implemented. The complexity and number of CLBs depends upon the particular family and device. CLBs can be interconnected together to implement more complex logic functions.

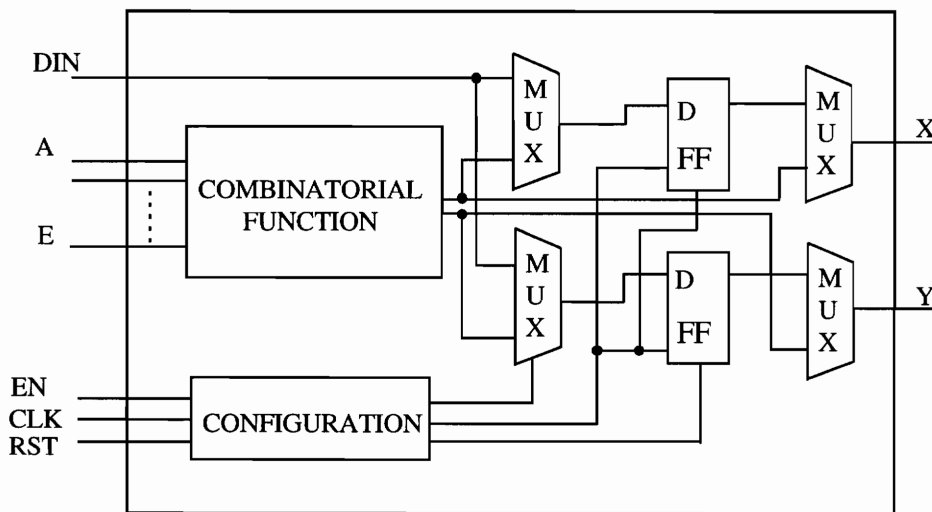


Figure A.2: Simplified block diagram of a CLB. The CLBs are used to implement the desired digital function. The outputs X and Y can either be a latched or combinatorial function of the inputs A through E and Din.

The I/O Interface Blocks provide the interface between the pins of the FPGA device and the internal logic circuitry. A simplified IOB is shown in Figure A.3. The output signal OUT, driven from the internal logic circuitry, can be directly passed through to the I/O pad or it can be latched using the chip clock CLK. The output signal can also three-stated. The input signal, derived from a source external to the chip, may be passed directly though the IOB or it may be latched in a flip flop located in the IOB. The signal is passed to the internal logic circuitry through DIR_IN if it is a direct signal and REG_IN if it is a latched signal. The inputs to the chip can be configured to be TTL or CMOS compatible and may use an optional internal pull-up resistor.

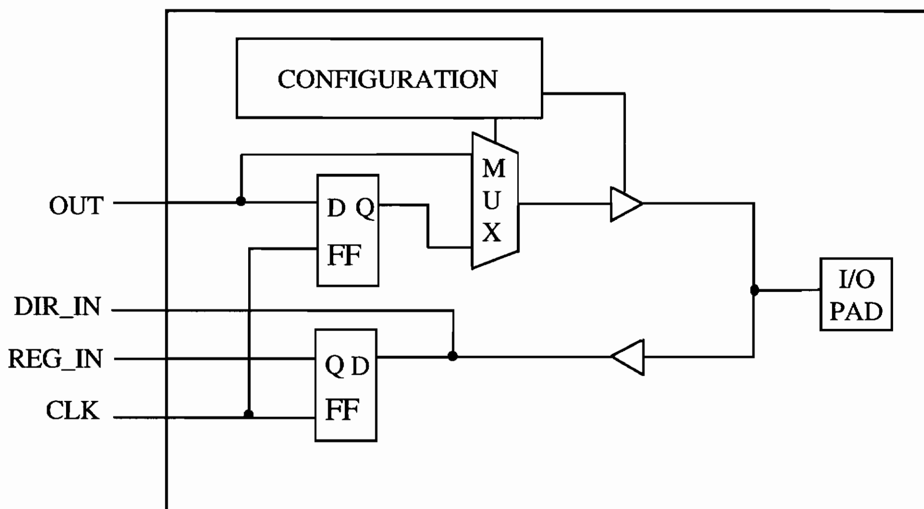


Figure A.3: Simplified block diagram of an IOB. The I/O Interface Block can be configured as either an input or an output used to connect the internal logic signals to the device pins.

The interconnection resources provide routing paths to connect CLBs to each other and to IOBs. The programmable routing resources consist of direct, general purpose and long lines. Direct lines are used for connecting adjacent CLBs or IOBs. General purpose resources are located between all of the rows and columns of CLBs and IOBs. The general purpose lines are internally connected through switches by the configuration data to allow for routing throughout the device. Long lines exist for connecting elements across the chip without the delays caused by the internal switches of the general purpose routing resources. Special clock lines exist for routing the clock to all parts of the chip with minimal delay regardless of loading. Additional interconnection resources distribute a global reset line throughout the chip.

The internal configuration memory is a large shift register used to hold the device configuration. Upon power-up, the LCA device must load a configuration specification into the internal configuration memory. Reconfiguration of the logic device is performed by downloading a new bit stream into the target device. Reconfiguration can be initiated at any time. The configuration memory can also be read back at any time.

Xilinx FPGA Design Cycle

The design cycle can be broken down into the following steps:

1. Design Entry
2. Functional Simulation

3. Partitioning and Mapping

4. Downloading

5. Design Verification

The design process begins with the design entry procedure, using schematic capture or VHDL modeling [17]. Various CAD packages which support Xilinx chips exist for both the PC and workstation environment. Schematics are created using parts from a pre-defined library of Xilinx primitives and macros. Primitives are the basic logic elements and macros are more complex logic elements composed of the primitives. Examples of macros include counters and shift registers. The use of macros allows for a hierarchical approach to be used.

Once the design is entered, it may be functionally simulated. At this time, no timing simulation can be performed since placement within the FPGA is not yet known. After the simulation is complete, the design must be compiled into a Xilinx netlist format (XNF) suitable for use with the Xilinx software.

The Xilinx software package XACT partitions the design and maps the logic to the target device. The operation can be performed automatically or interactively. Constraints may be given in the schematic to result in the best possible routing of critical nets. The end result of the process is a configuration specification known as a bitstream ready for

downloading. By back annotating the placement and routing information to the schematic, timing simulation can also be performed at this point.

Six modes exist for downloading the bitstream to the target device. Three master modes allow for the FPGA to control the configuration process. Two peripheral modes and a serial slave mode allow external devices to control the configuration process. For all modes, configuration is initiated using the RESET* line and is completed when the DONE/PROG* line goes high.

In the master parallel modes and master serial mode, the LCA automatically loads parallel or serial configuration data from an external PROM upon reset. For these modes, the FPGA uses its internal clock CCLK and generates the control signal DONE/PROG* for enabling the PROM. For the master parallel modes, the FPGA also generates the address data on A0-A10 in either a up or down fashion and retrieves data from the PROM on D0-D7. For the master serial mode, the PROM must be capable of automatically incrementing its internal address counter and the data is returned on DIN.

In the peripheral modes, the device is treated as a processor peripheral and the data is transferred byte-wise in a polled fashion from a host to the LCA on lines D0-D7. The RDY/BUSY* line of the FPGA indicates when the device is ready for another piece of data. The byte data is then internally serialized back into a bitstream either in a

synchronous or asynchronous manner. In the slave mode, serial configuration data is loaded into the device on DIN with a synchronized external clock source on CCLK.

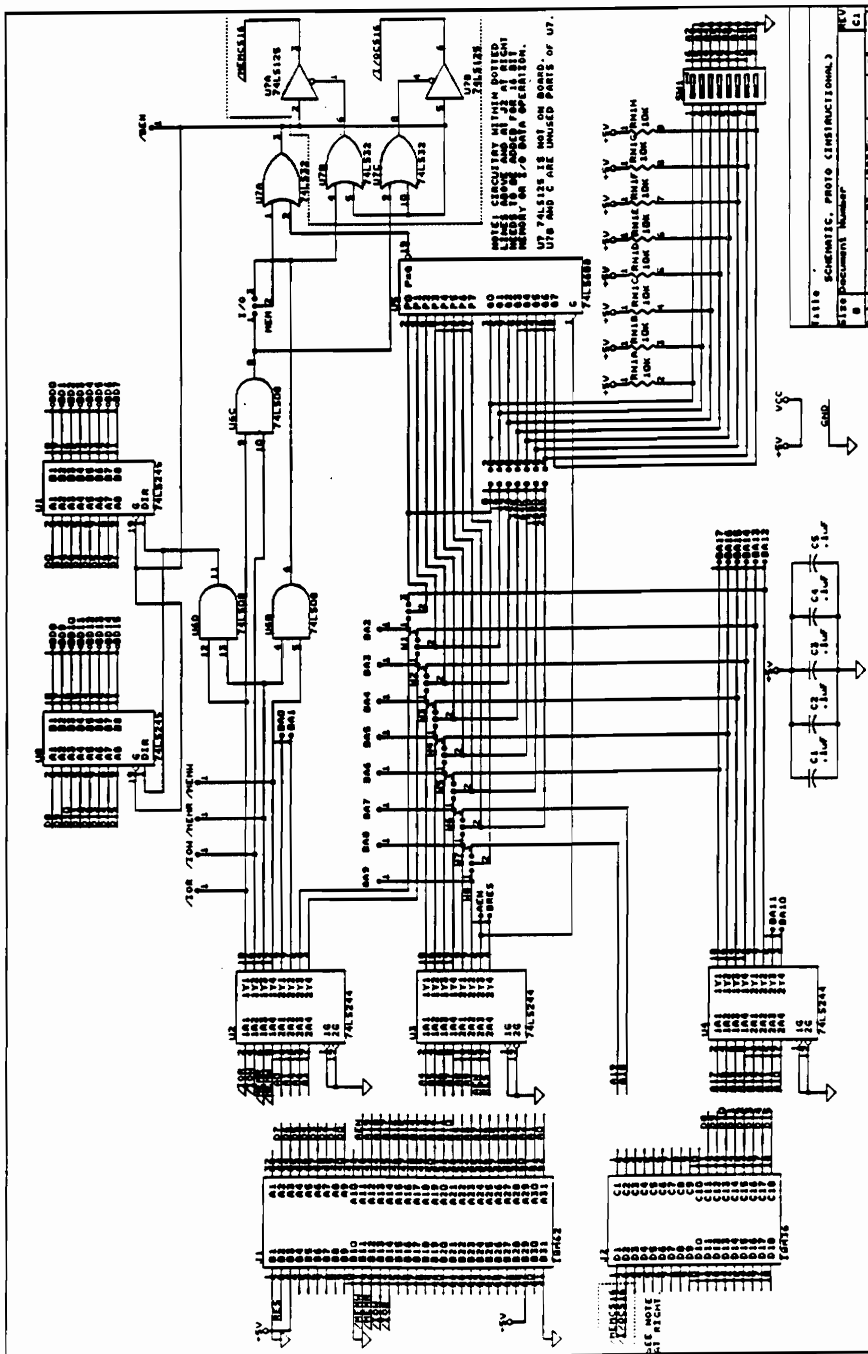
Multiple LCAs can be daisy-chained together to allow for easy configuration. The devices can be loaded with identical or unique configurations. For devices with identical data, the devices are wired in parallel, allowing them to be configured simultaneously. For device with different configurations, the DOUT of the one chip connected to the data in DIN of the next chip. Configuration of each device is then a serial process.

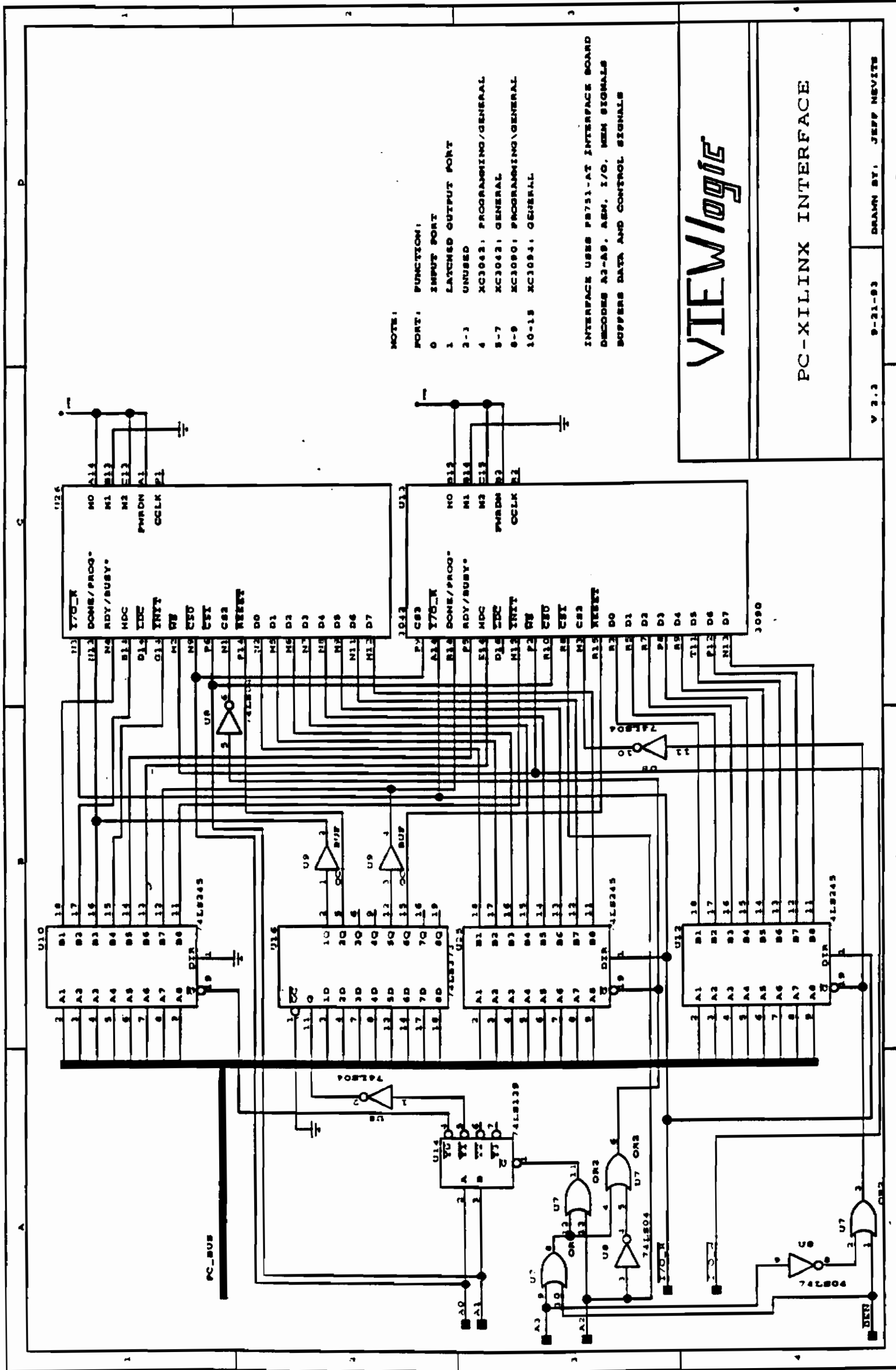
For more information on Xilinx field programmable gate arrays, consult [17].

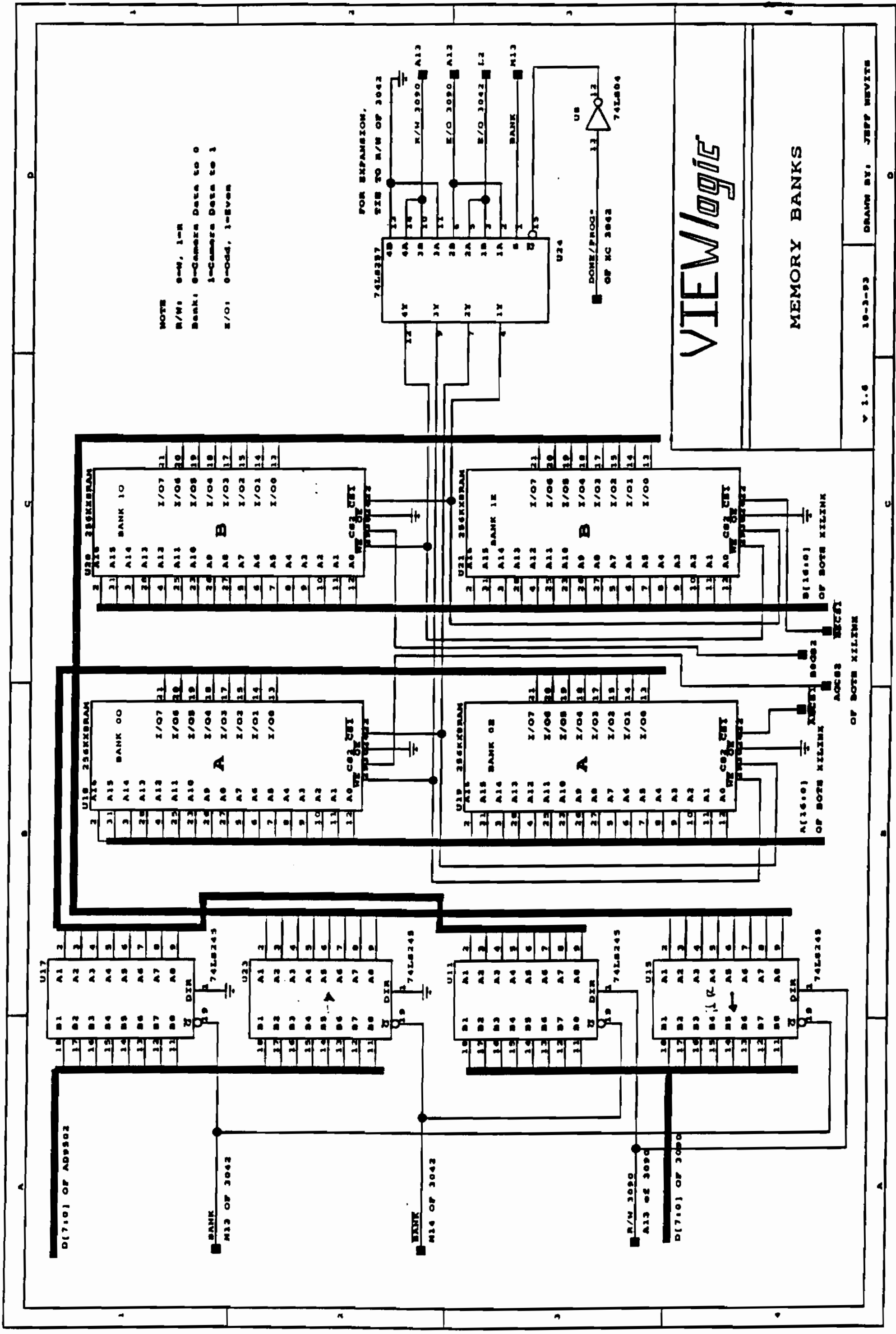
Appendix B
JB1 Board Information

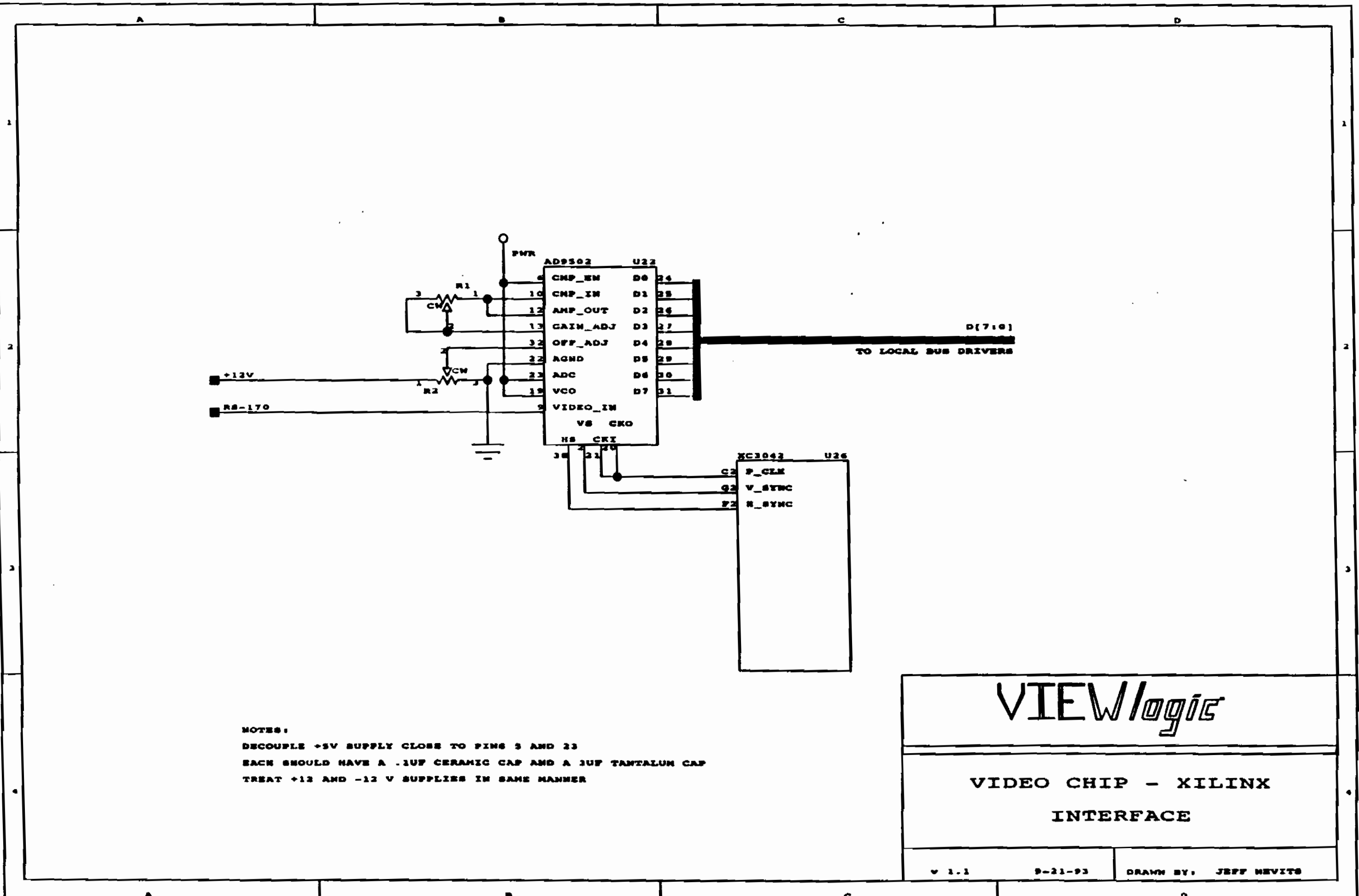
This appendix contains board level information for the JB1 frame grabber board. As described in Chapter 3, the following pages contain the ISA-bus prototype card schematic, the board level schematics, the cable definition and a parts list for JB1.

SCHEMATIC

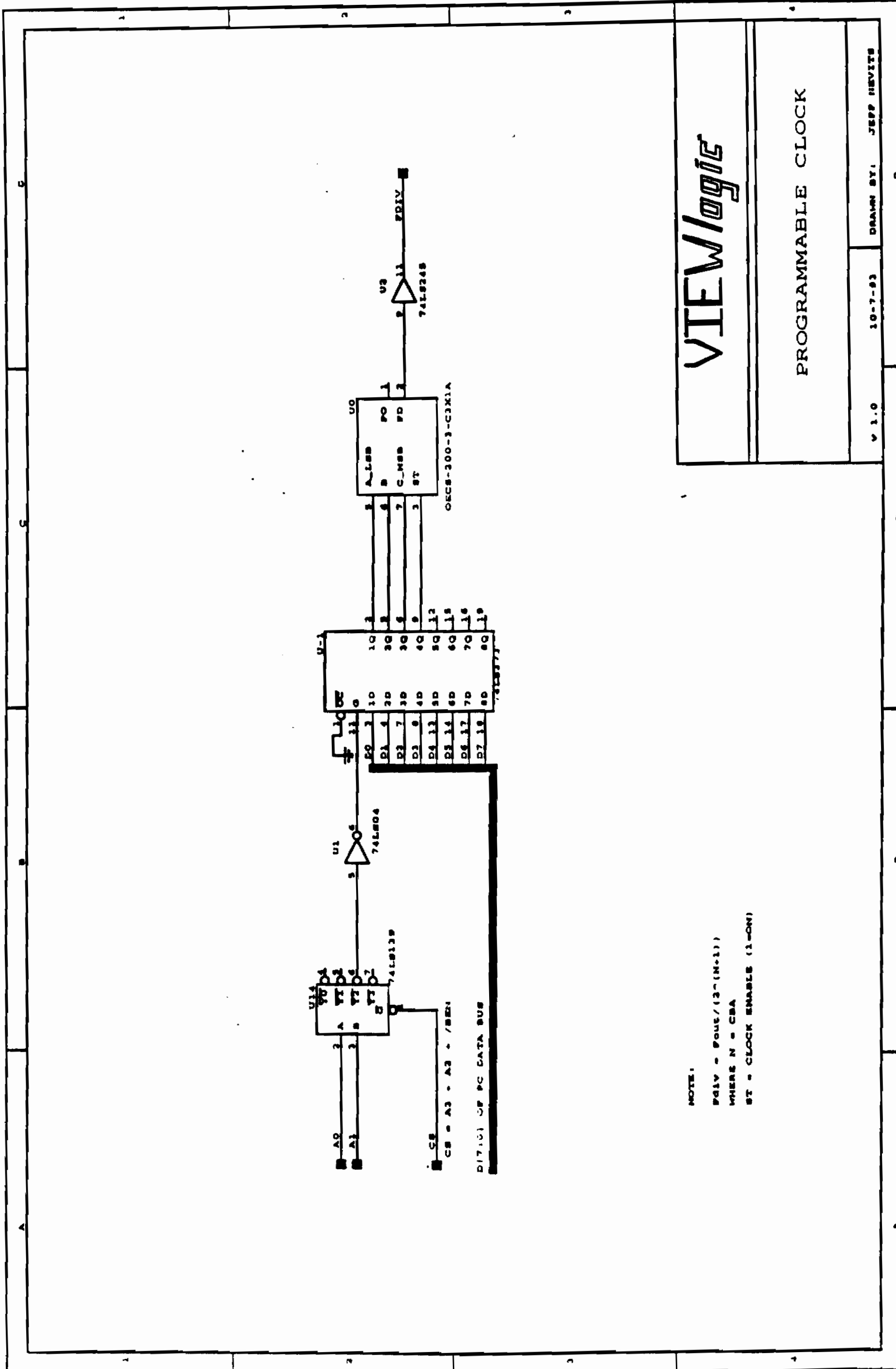



NOTE:
PORT: FUNCTION:
0 INPUT PORT
1 LATCHED OUTPUT PORT
2-3 UNUSED
4 XC3042: PROGRAMMING/GENERAL
5-7 XC3042: GENERAL
8 XC3090: PROGRAMMING\GENERAL
8-9 XC3090: GENERAL
10-15 XC3094: GENERAL
INTERFACE USES PS751-AT INTERFACE BOARD
DECODES A2-A9, AEN, I/O, MEM SIGNALS
BUFFERS DATA AND CONTROL SIGNALS
VIEWlogic
PC-XILINX INTERFACE
V 3.3
9-21-93
DRAWN BY: JEFF NEVITT
PC_BUS


VIEWlogic
MEMORY BANKS
DRAWN BY: JEFF NEVITT
10-3-93



PWR
AD9502
U22
CMP_EN
CMP_IN
AMP_OUT
GAIN_ADJ
OFF_ADJ
AGND
ADC
VCO
VIDEO_IN
VS
CKO
HS
CKI
D0
D1
D2
D3
D4
D5
D6
D7
R1
R2
CW
+12V
RS-170
D[7:0]
TO LOCAL BUS DRIVERS
XC3042
U26
P_CLK
V_SYNC
H_SYNC
NOTES:
DECOUPLE +5V SUPPLY CLOSE TO PINS 5 AND 23
EACH SHOULD HAVE A .1UF CERAMIC CAP AND A 1UF TANTALUM CAP
TREAT +12 AND -12 V SUPPLIES IN SAME MANNER
VIEWlogic
VIDEO CHIP - XILINX
INTERFACE
V 1.1
9-21-93
DRAWN BY: JEFF NEVITS


VIEWlogic
PROGRAMMABLE CLOCK

Cable Definition

JB1 is connected to Splash system through a 50 pin connector. The connector supports a 36 bit data bus as defined in Table B.1. Additional signals are used for controlling the data transfer process from JB1 to Splash.

Table B.1: Definition for Splash connector

PIN	DEFINITION	PIN	DEFINITION
1	D0	2	D35
3	D1	4	D34
5	D2	6	D33
7	D3	8	D32
9	D4	10	D31
11	D5	12	D30
13	D6	14	D29
15	D7	16	D28
17	D8	18	D27
19	D9	20	D26
21	D10	22	D25
23	D11	24	D24
25	unused	26	Gnd
27	Gnd	28	unused
29	D12	30	D23
31	D13	32	D22
33	D14	34	D21
35	D15	36	D20
37	D16	38	D19
39	D17	40	D18
41	Gnd	42	Valid Data
43	Gnd	44	Clk
45	Gnd	46	Start of Frame
47	Gnd	48	End of Frame
49	Gnd	50	Frame Request

Parts List

The JB1 board uses the following parts:

Reference Designator	Part
U1 - U6	SN74LS654N octal bus transceiver
U7	SN74LS32N quad 2 input OR gate
U8	SN74LS04N hex inverter
U9	SN7417N open collector buffer
U10-U12, U15, U17, U23, U25	SN74LS245N octal bus transceiver
U13	XC 3090 Xilinx FPGA
U14	SN74LS139N dual 2/4 decoder
U16	SN74LS373N octal D type latch
U18-U21	PDM40124S25TC Paradigm 128Kx8 SRAM
U22	AD9502 Analog Devices Hybrid Video Chip
U24	74AS257N tristate quad 2 input mux
U26	XC 3042 Xilinx FPGA
U27	20 MHz dual output oscillator

26 pin wire wrap connector
50 pin wire wrap connector
BNC coaxial connector
16 x 16 175 pin LIF socket
14 x 14 132 pin LIF socket
20 pin DIP wire wrap sockets
16 pin DIP wire wrap sockets
14 pin DIP wire wrap sockets
5 kΩ side adjust potentiometers
0.1 µf ceramic capacitors
0.001 µf ceramic capacitor
2.2 µf tantalum capacitor
PB751-AT Industrial Computer Source Prototype Board

Appendix C
XC 3042 Design

This appendix contains state machines, pinouts and schematics for the XC 3042 design. The XC 3042 is used on JB1 as the interface between the video chip and local memory as described in Section 3.4.

The field detection state machine shown in Figure C.1 and a counter are used to determine which field is being captured. The states are defined as follows.

A: Waiting for V_sync. Disable count and reset counter.
B: Detected V_sync. Enable count.
C: Detected H_sync. Disable count.
D: V_sync has gone inactive.

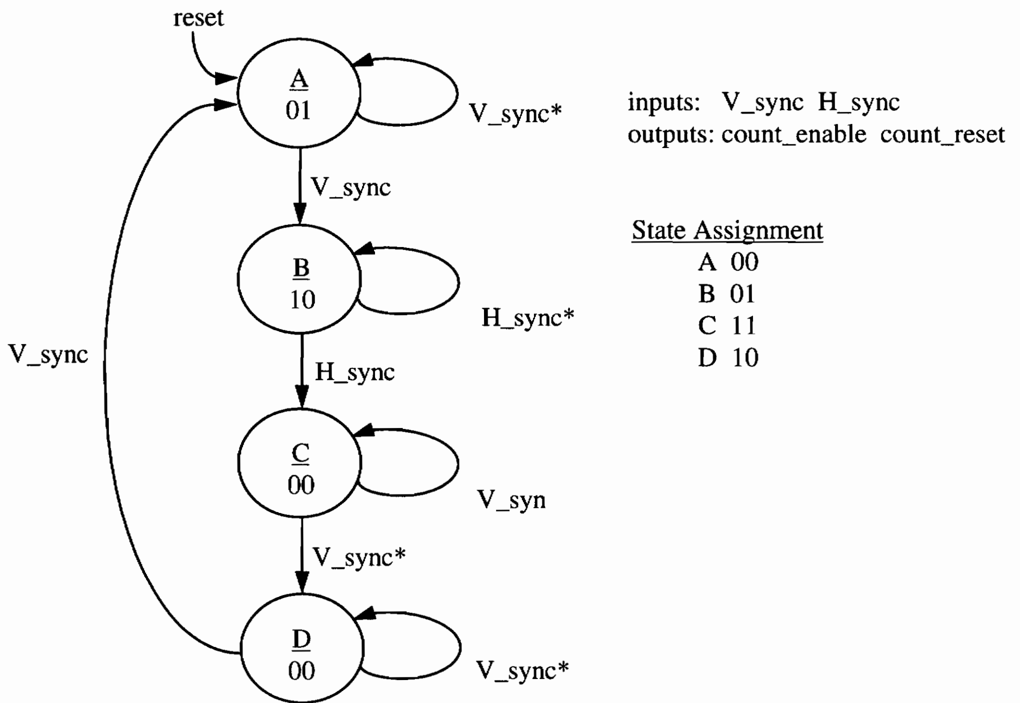


Figure C.1: Field detection state machine. The FSM is used to detect which field of data is being presented.

At the rising edge of V_sync, the counter is enabled. The counter is disabled at the rising edge of H_sync. Approximately 312 pulses will be counted if the field is even; 624 if the field is odd. The E/O flag is bit 9 of the counter. The field is even if the bit is zero and the field is odd if the bit is one.

The invalid line counter state machine shown in Figure C.2 and a counter are used to mask off the first eleven lines of a field since they do not contain active video data. The states are defined as follows.

A: Reset state. Waiting for V_sync. Reset counter.
B: Detected V_sync. Disable reset, enable counter and set invalid_line active.
C: Reached terminal count. Disable counter and set invalid_line flag inactive.
D: Dummy state.

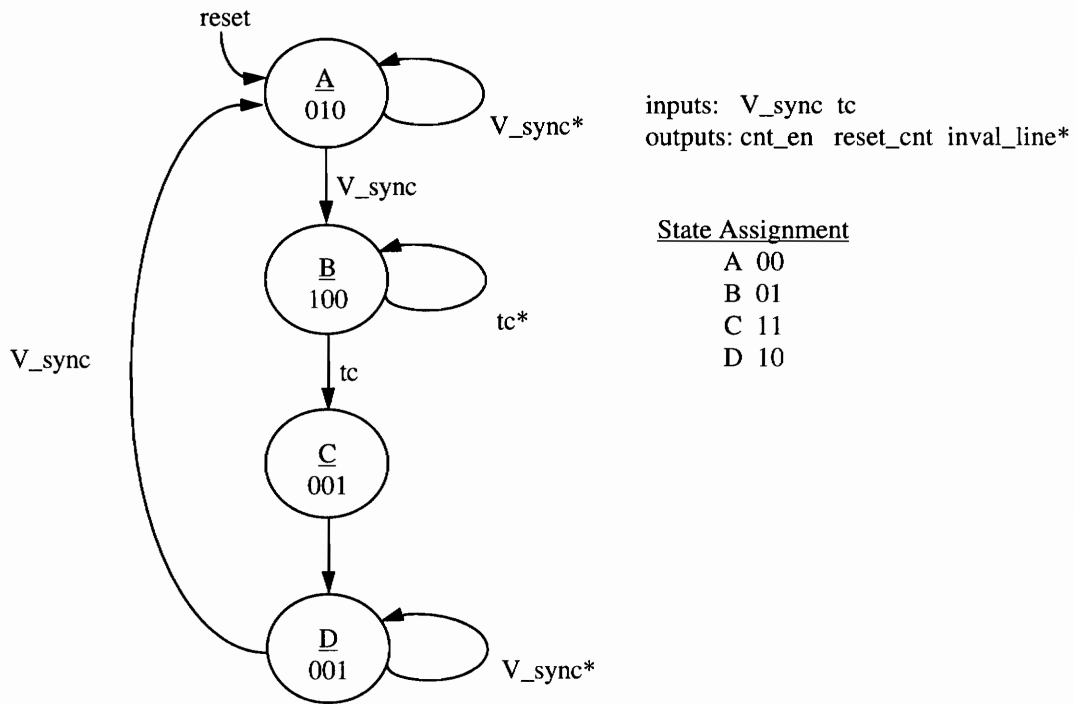


Figure C.2: Invalid line counter state machine. The FSM is used to mask out the invalid lines at the start of each field.

Upon detecting V_sync, invalid_line* is set active and the counter enable flag is set high. The enable flag is ANDed with H_sync pulses to gate the enable on the counter. Upon reaching the terminal count, the invalid_line flag is set inactive and the counting process is disabled.

The valid line counter state machine shown in Figure C.3 and a counter are used to determine when valid lines of video data are available. The states are defined as follows.

A: Reset state. Waiting for invalid_line* to go inactive.
B: Detected invalid_line* going high. Reset counter.
C: Disable counter reset. Enable counter and set valid_line flag active.
D: Reached terminal count. Disable counter and set valid_line flag inactive.

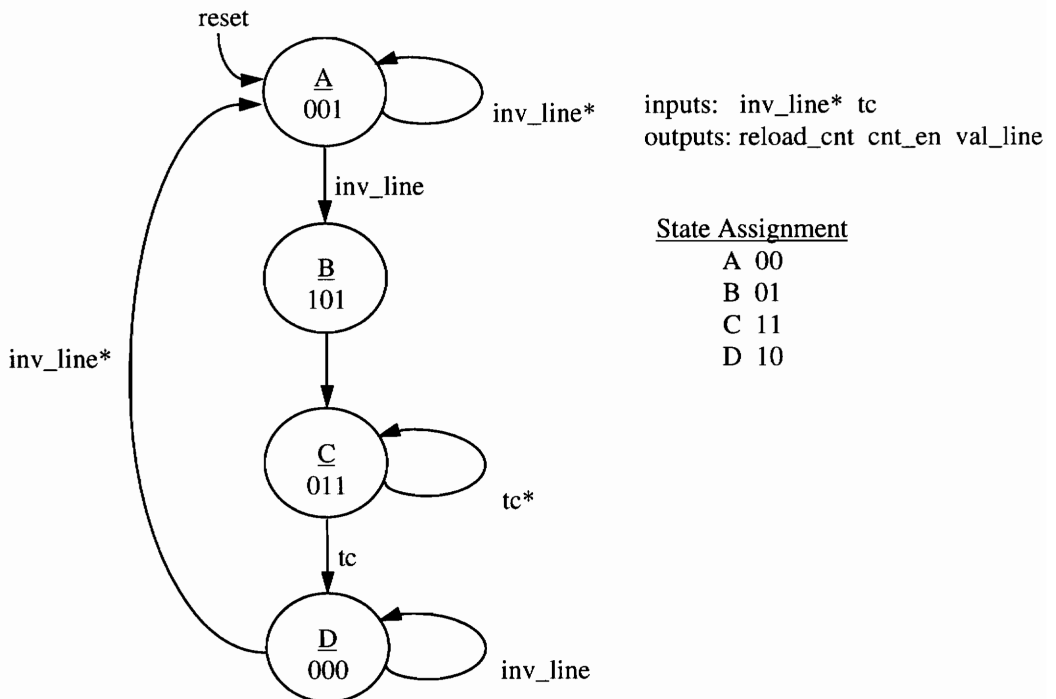


Figure C.3: Valid line counter state machine. The FSM is used to count the lines which contain active video data.

Upon detecting the invalid_line flag going inactive, the counter is enabled for 242 H_sync pulses. During this period, the valid_line flag is active. Upon reaching the terminal count, the valid_line flag is set inactive.

The invalid pixel counter state machine shown in Figure C.4 and a counter are used to mask off the invalid pixels of each line. The states are defined as follows.

A: Reset state. Waiting for H_sync. Reload counter.
B: Detected H_sync. Disable reset, enable counter and set invalid_pixel active.
C: Reached terminal count. Disable counter and set invalid_pixel flag active.
D: Dummy state. Waiting for next H_sync pulse.
E: Detected H_sync. Reload counter and return to state B.

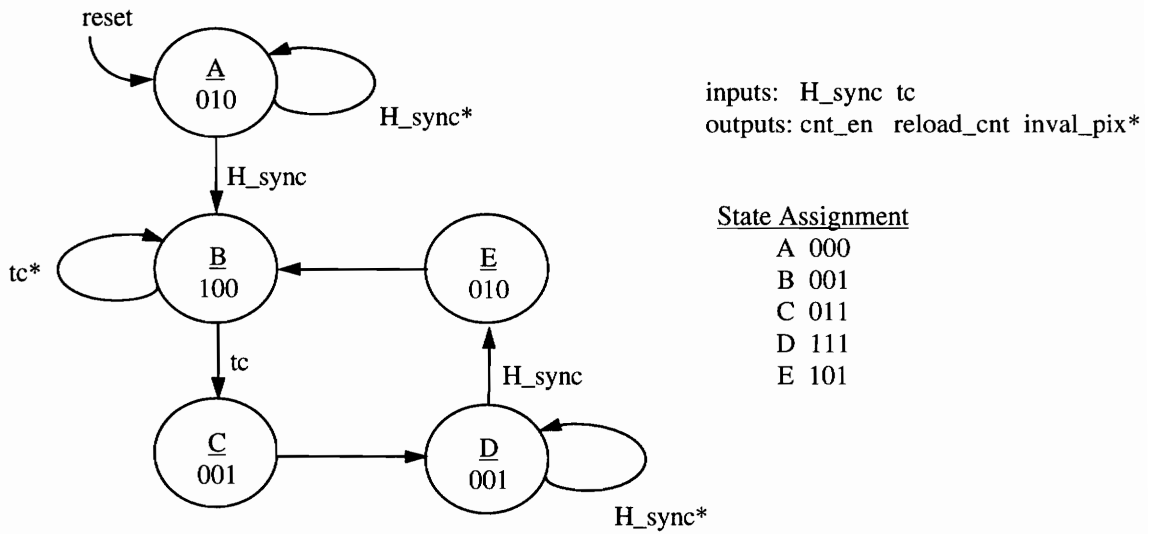


Figure C.4: Invalid pixel counter state machine. The FSM is used to mask out the invalid data at the start of each line.

The number of invalid pixels is variable, depending on the aspect ratio. Upon detecting a H_sync pulse, indicating a new line, the counter is loaded with a start value. The counter is then enabled and is incremented by each pixel clock. During this time, the invalid_pixel* flag is set active. Upon reaching terminal count, the invalid_pixel flag is set inactive.

The valid pixel counter state machine shown in Figure C.5 and a counter are used to count the valid data pixels of each line. The states are defined as follows.

A: Reset state. Waiting for inv_pix* to go inactive. Reload counter.
B: Inv_pix* has gone inactive. Disable reload, enable counter and set invalid_pixel high.
C: Reached terminal count. Disable counter and set valid_pixel flag inactive.
D: Dummy state. Waiting for inv_pix to go inactive.

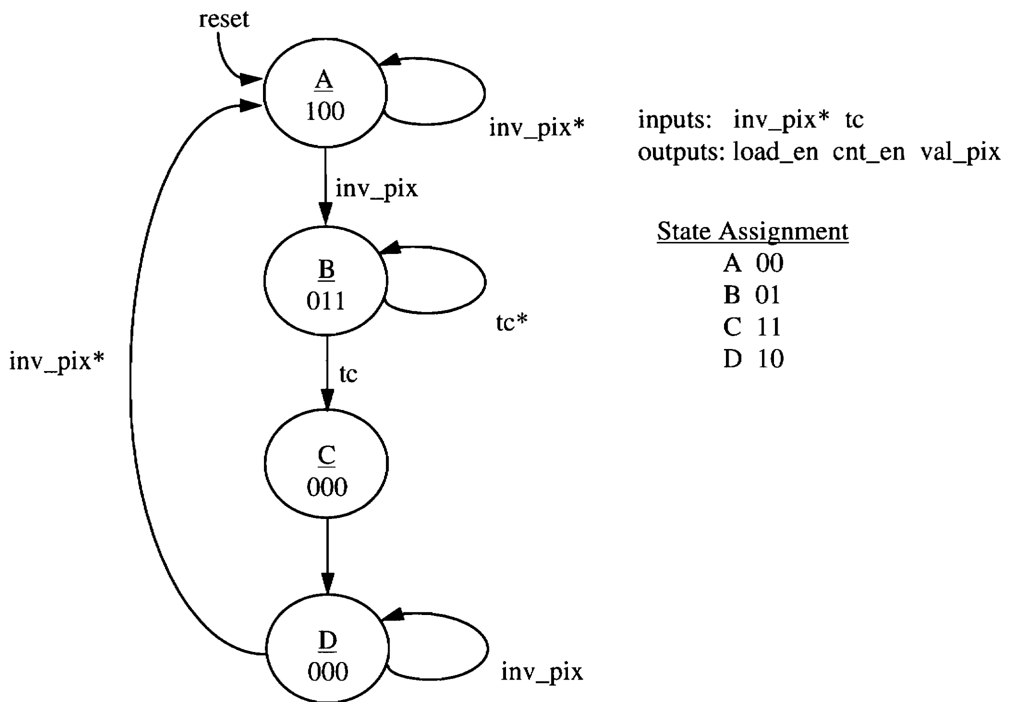


Figure C.5: Valid pixel counter state machine. The FSM is used to count the valid pixels for each line.

The number of valid pixels depends upon the aspect ratio and is programmable. Upon detecting the invalid_pixel* flag going inactive, the starting count value is loaded into the counter and the counter is enabled. The count value is incremented each pixel clock cycle

until the terminal count is reached. During this time, the valid_pixel flag is active. Upon reaching the terminal count, the valid_pixel flag is set inactive.

The frame controller state machine shown in Figure C.6 controls when data is actually stored in memory. The states are defined as follows.

A: Reset state. Wait for odd field.
B: Detected odd field. Wait until field is even or detect valid_signal.
C: Received valid signal. Enable valid_data flag.
D: Detected field change. Wait for valid signal.

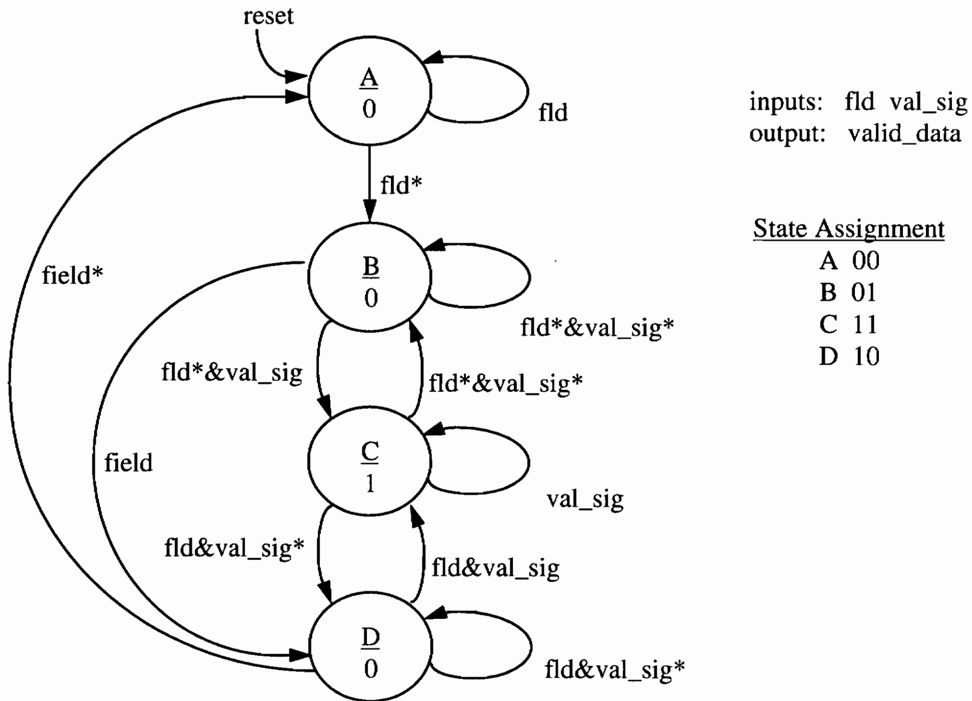


Figure C.6: Frame controller state machine. The FSM enables valid image data to be stored in memory.

The valid_signal is the logical "and" of valid_line, valid_pixel and get_frame. Data will only be stored when there is valid data. Storage must start with the first field. After

detecting the first field, the second field may start before valid_signal is active. If this occurs, a new frame must occur before storage can begin. If the valid_signal is detected first, then data can be stored and the valid_data flag is set active for both fields.

The address generation unit state machine shown in Figure C.7 is used to control the address lines and memory control lines. The states are defined as follows.

A: Reset state. Reload counters.
B: Detected V_sync.
C: Detected valid_data. Enable counters.
D: Capture complete. Disable counters.

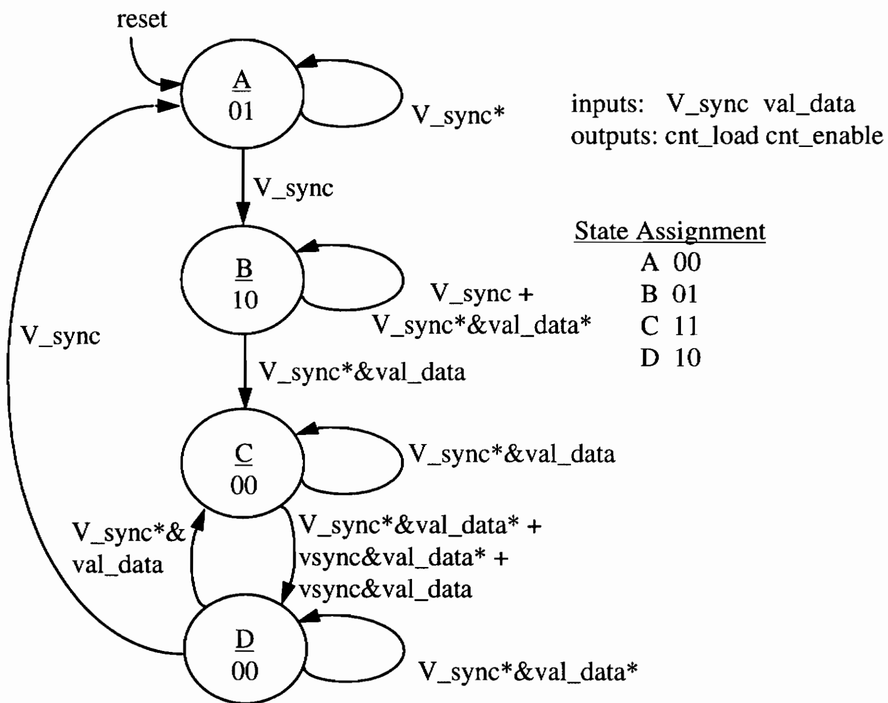


Figure C.7: Address generation unit state machine. The FSM controls the address counter and strobes for memory.

Upon detecting a new field, an internal counter is pre-loaded with a value that depends upon the image size being captured. Upon detecting the valid_data flag, the counter is enabled. During this time, each clock cycle increments the memory address for data storage. A write strobe is also generated for each clock cycle. Upon reaching terminal count, the counter enable and the address enables are disabled.

The image controller state machine shown in Figure C.8 determines when images may be captured. The states are defined as follows.

A: Reset state. Waiting for bank_toggle. Set outputs inactive.
B: Detected bank_toggle. Set get_frame active. Wait for second field.
C: Now have even field and valid_line low.
D: Dummy state. Now have even field and valid_line active.
E: Waiting for valid_line to go inactive.
F: Valid_line low so set done_collecting flag active and get_frame inactive.
G: Bank toggle occurred. Wait for first field.

After a bank toggle is completed, the get_frame flag is set to allow the capture of a frame. The flag will remain active until the end of the second field is detected. This is indicated by the second high to low transition of the valid_line flag. Upon completion of capture of the even field, the get_frame flag is set inactive and the done_collecting flag is set active.

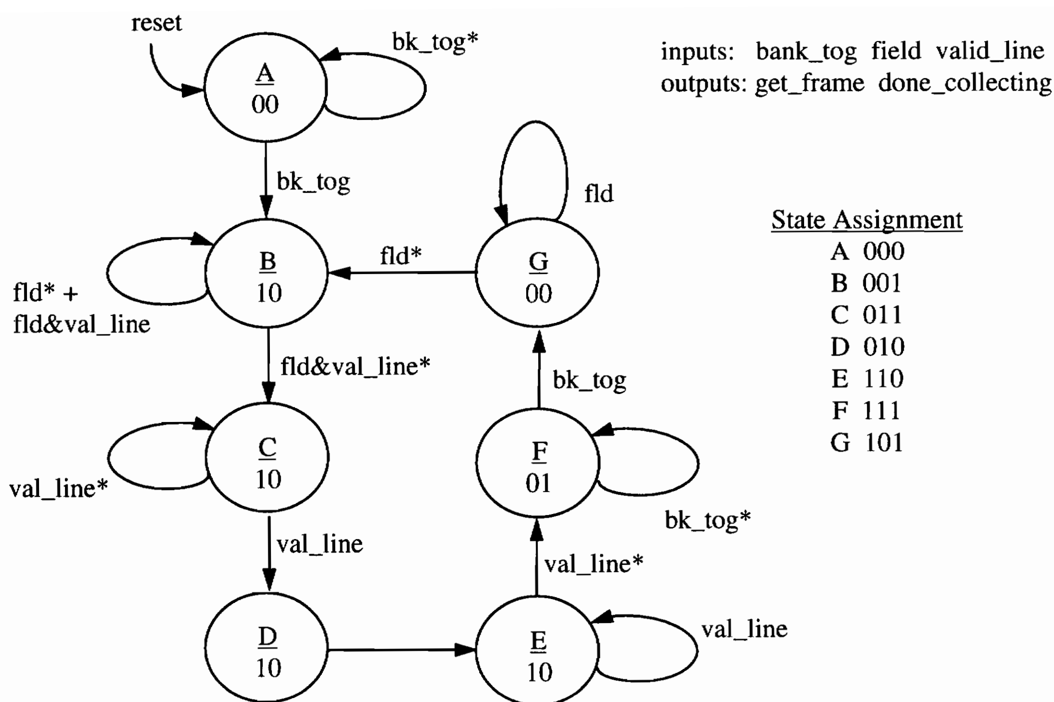


Figure C.8: Image controller state machine. The FSM is used to insure that capturing begins with the correct field.

The bank switch state machine shown in Figure C.9 controls the bank switching needed for real-time data transfer. The states are defined as follows.

A: Reset state. 3042 ready to switch banks so set rdy_1. Waiting for 3090 rdy1_ack.
B: Received 3090 acknowledgment. Waiting for rdy_2 from 3090 to go active.
C: Detected 3090 rdy_2 going active. Issue bank toggle.
D: Deassert rdy_1. Wait for 3090 to acknowledge by setting rdy1_ack low.
E: Received rdy1_ack. Now wait for rdy_2 to go inactive.
F: Detected rdy_2 going low. Acknowledge by setting rdy2_ack low.

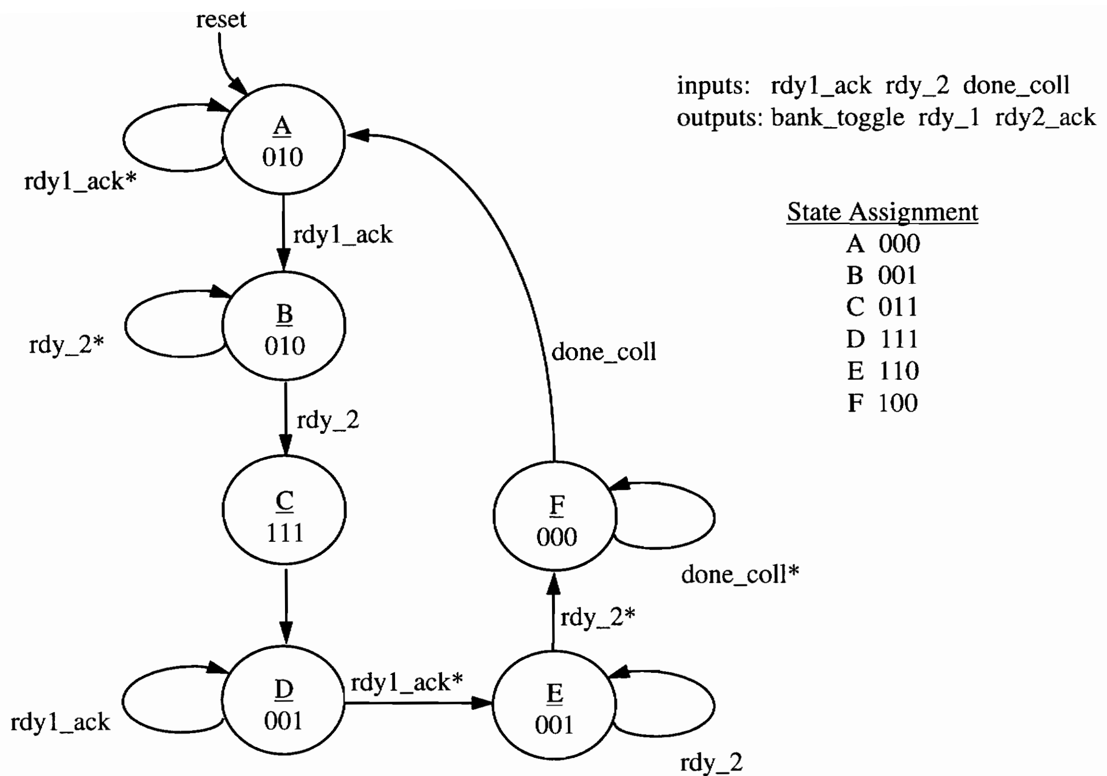


Figure C.9: Bank switch state machine. The FSM is used to control bank switching between the Xilinx chips.

A bank switch occurs when both the 3042 and the 3090 are ready to switch. A full handshaking protocol is used between the chips for communication. The 3042 indicates that is ready to switch banks by setting rdy_1 active. When the 3090 is ready to switch, it sets the rdy_2 line active. The 3090 acknowledges the 3042 is ready to switch by setting rdy1_ack active. After receiving the 3090 acknowledgment, the 3042 acknowledges the 3090 is ready to switch by setting the rdy2_ack line active and a bank switch occurs. Rdy_1 is then deasserted by the 3042. The 3090 will acknowledge by setting rdy1_ack low and then setting rdy_2 low. The 3042 will acknowledge the rdy_2 line going low by

setting rdy2_ack low. The bank switch is now complete. Another bank switch can not occur until the 3042 collects a frame. The handshaking is shown in the Figure C.10.

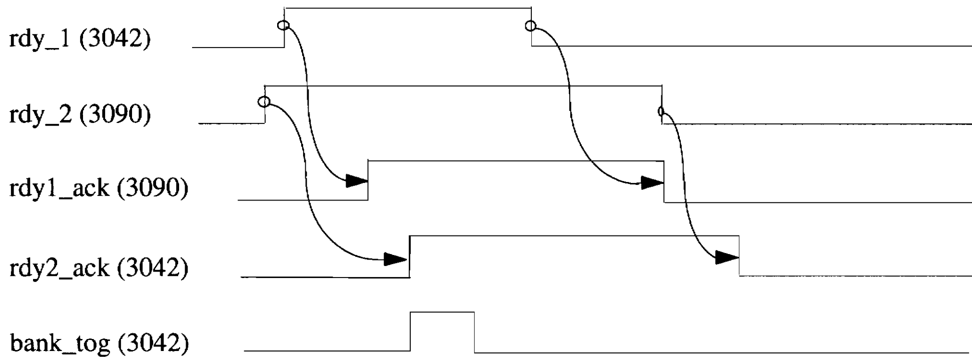


Figure C.10: Bank switching handshaking. Both FPGAs must be ready to switch banks before the switch can occur.

XC 3042 Pinout

The XC 3042 used on JB1 is a 132 pin device arranged in a pin grid array. The pins of the XC 3042 are divided into three groups: permanently dedicated pins, dual function pins and generic I/O pins. The permanently dedicated are used to support basic chip functions and can not be used by the design. The dual function pins are used to configure the device. After configuration, these pins may be used by the design. In Table C.1, the dual function pins contain two names. The first is the name of the signal used during configuration and the second, in parentheses, is the name of the signal after configuration. The generic I/O pins become defined after configuration. For a complete detailed list of the dedicated pins and the dual function pins, see [17].

Table C.1: XC 3042 pin mappings.

PIN	NAME	NOTES
C8, D12, G12, M11, M8, M4, G3, D3	Vcc	Dedicated power pins
C4, C7, C11, H12, L12, M7, H3, L3	GND	Dedicated ground pins
A1	PWRDWN*	Dedicated power down pin
P14	RESET*	Dedicated reset line
P1	CCLK	Dedicated configuration clock
N13	DONE/PROG*	Dedicated programming pin
A14, B13	M0, M1	Dedicated programming mode pins
C13	M2	Mode pin. Unused after configuration.
N4	RDY/BUSY*	Programming pin. After configuration, unused input to PC
N9	CS0* (A0)	Programming chip select. After configuration, PC address bit A0
P6	CS1* (A1)	Programming chip select. After configuration, PC address bit A1
N1	CS2 (3042_Sel)	Programming chip select. After configuration, 3042 select line
M2	WS* (I/O_W*)	Programming write strobe. After configuration, PC I/O write strobe
B14	HDC	High during configuration. Unused after configuration
D14	LDC	Low during configuration. Unused after configuration
A8	ADD2	PC address bit A2
A9	ADD3	PC address bit A3
D13	BS*	Board Select
G14	INIT*	Initialization pin. Unused after configuration
N2, M5, M6, N7, N8, M9, N11, M12	PD0-PD7	Configuration data bus. After configuration, data bus to PC
N3	I/O_R*	PC I/O read strobe
C2	P_CLK	Pixel Clock from AD9502
G2	V_SYNC	Vertical Sync from AD9502
F2	H_SYNC	Horizontal Sync from AD9502
K3	RDY_1	Bank Switch Handshaking Line to

		3090
J3	RDY_2	Bank switch handshaking line from XC 3090
L1	RDY1_ACK	Bank switch handshaking line from XC 3090
J2	RDY2_ACK	Bank switch handshaking line to XC 3090
M13	BANK	Bank Flag
M14	BANK*	Bank Flag complement
L2	E/O	Field Indicator
K1	AECS1*	Memory Bank 0, Odd chip select
H2	AOCS2	Memory Bank 0, Even chip select
F3	BECS1*	Memory bank 1, Odd chip select
H1	BOCS2	Memory bank 1, Even chip select
H13, G13, K13, C9, C10, E12, C12	AA0-AA7	Bank 0 address lines A0-A7
E3, B4, D1, B3, B5, C5, A6, B8, B6	AA8-AA16	Bank 0 address lines A8-A16
J13, F14, J12, B9, A10, B11, C14, B12	BA0-BA7	Bank 1 address lines A0-A7
B1, P2, D2, B2, C6, A4, B7, A7, A5	BA8-BA16	Bank 1 address lines A8-A16

XILINX

XC 3042 SCHEMATIC

Sheet 1 of 1

v 3.0 | 12-1-93 | DRAWN BY: JEFF NEVITS

IR8B

8 BIT INPUT REGISTER

V 2.0 | 11-21-93 | DRAWN BY: JEFF NEVITS

D0 D1 D2 D3 D4 D5 D6 D7 CLK2 CLK

BD0 BD1 BD2 BD3 BD4 BD5 BD6 BD7

XILINX
AOB
ADDRESS OUTPUT BUFFER
V 1.0
DRAWN BY: JEFF NEVITS
A0 A1 A2 A3 A4 A5 A6 A7 A8
B0 B1 B2 B3 B4 B5 B6 B7 B8
A9 A10 A11 A12 A13 A14 A15 A16
B9 B10 B11 B12 B13 B14 B15 B16
OBUFT
FAST
EN

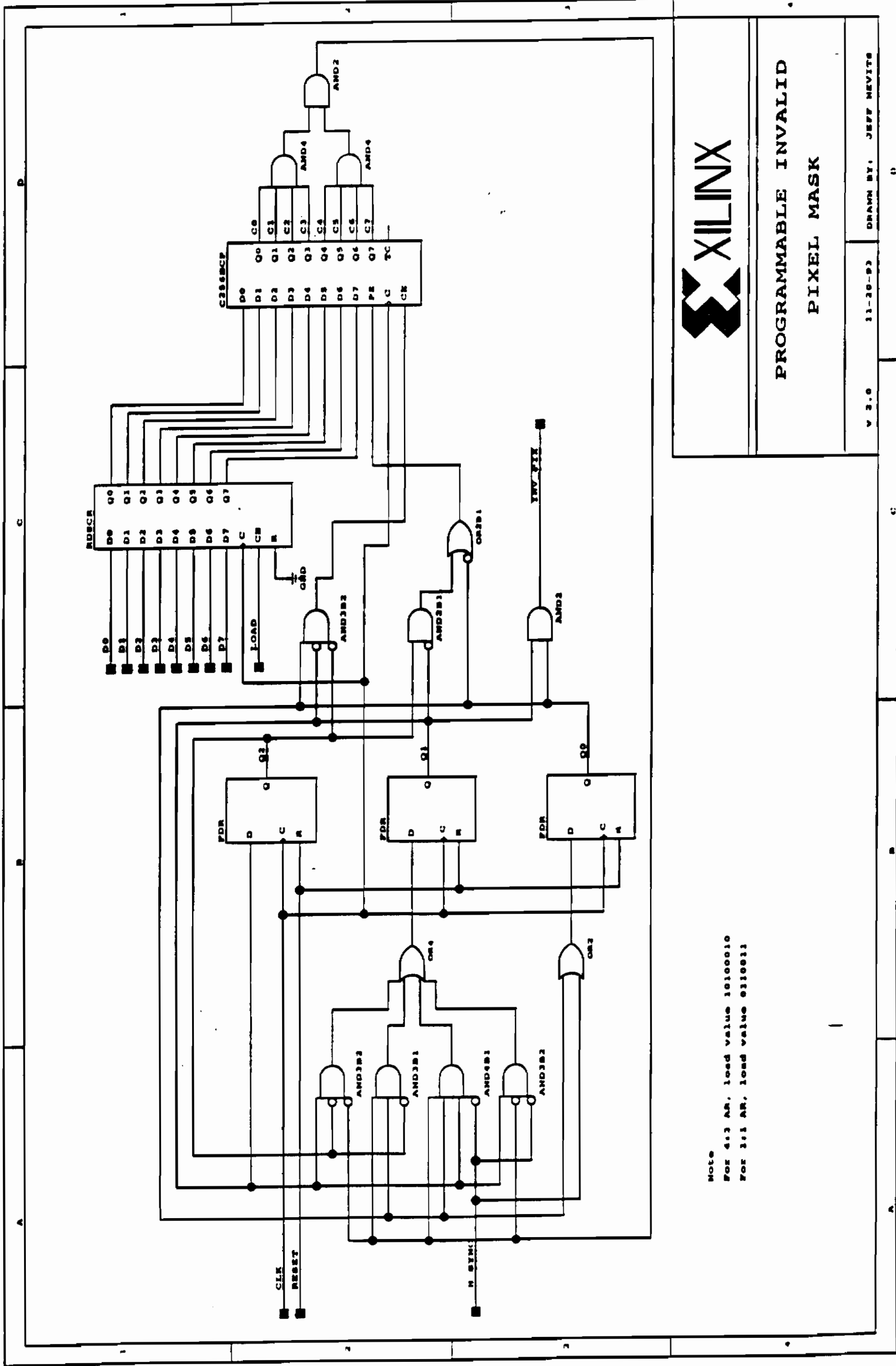
PROGRAMMABLE VALID
PIXEL COUNTER

V 3.0 11-20-93 DRAWN BY: JEFF NEVITT

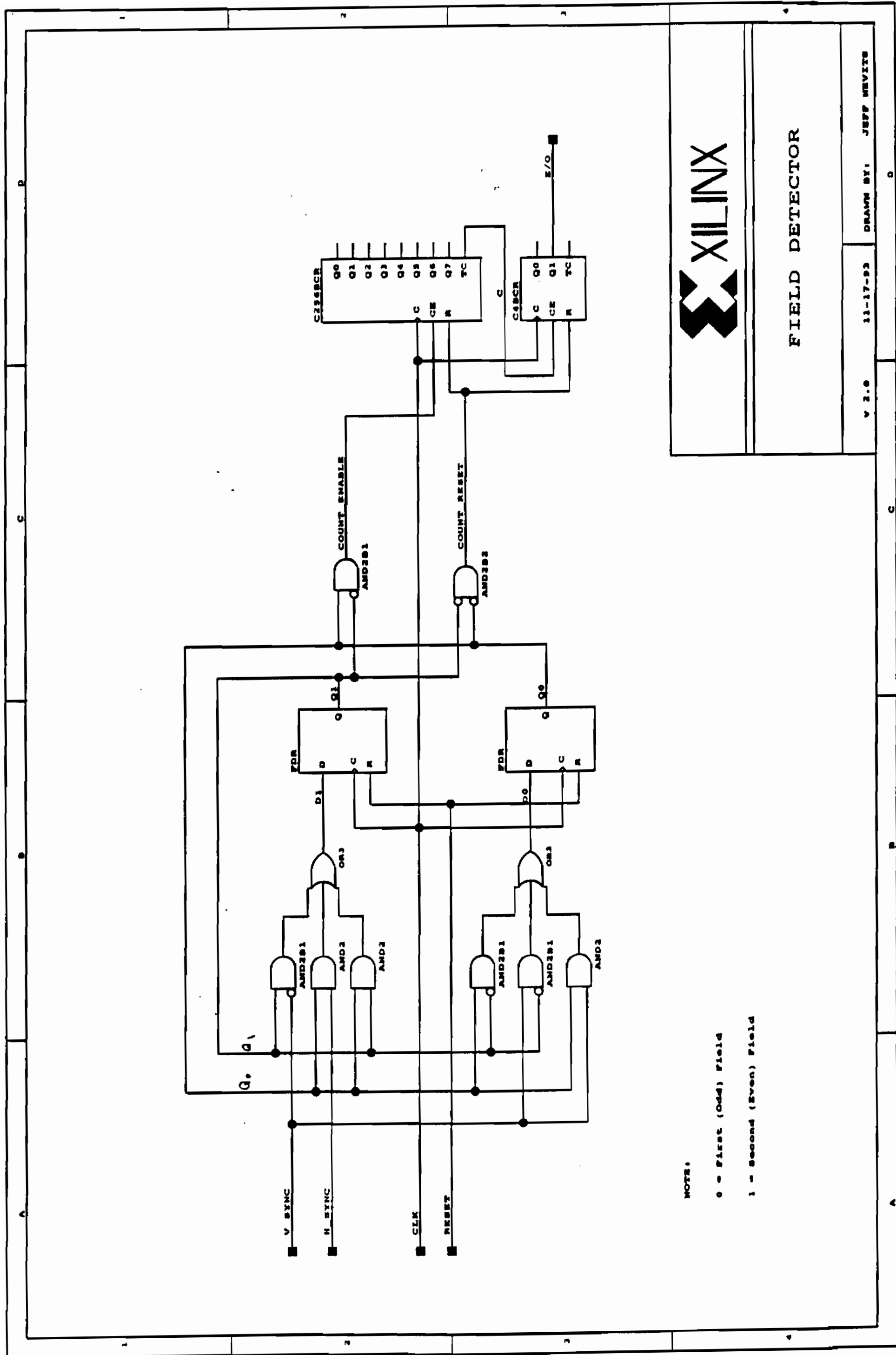
NOTE:

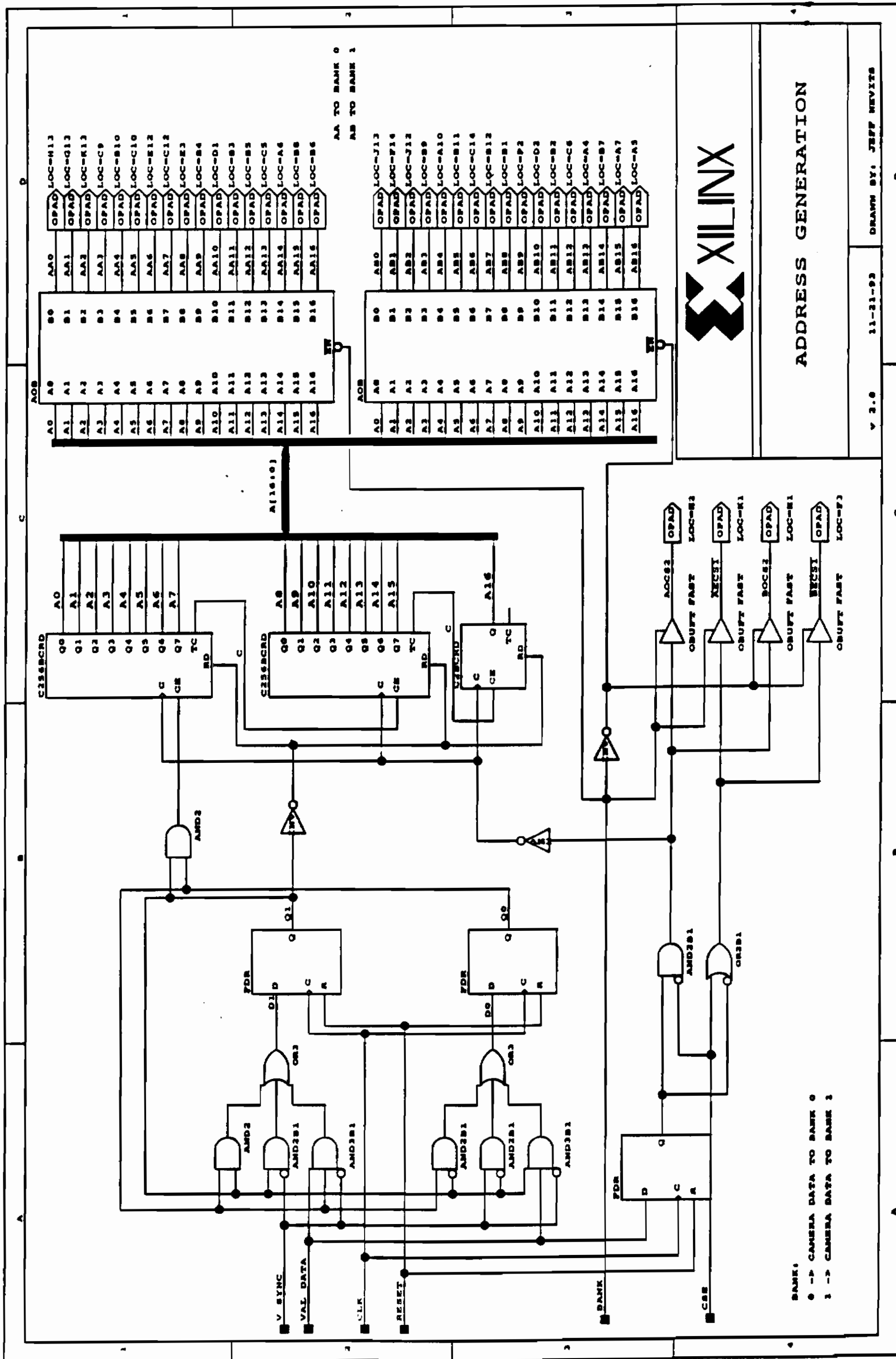
For 4:3 AR, Load Value = 0111111111

For 1:1 AR, Load Value = 1001111111


XILINX
PROGRAMMABLE INVALID
PIXEL MASK
DRAWN BY: JEFF NEVITT
CLK
RESET
LOAD
INV_PIX

FRAME CONTROLLER


XILINX
FIELD DETECTOR
V 2.0
11-17-93
DRAWN BY: JEFF NEVITS
V SYNC
H SYNC
CLK
RESET
E/O
COUNT_ENABLE
COUNT_RESET
C256BCR
C4BCR
FDR
AND2B1
AND2B2
AND2
OR3
NOTE:
0 = First (Odd) Field
1 = Second (Even) Field


XILINX
ADDRESS GENERATION

BANK CONTROLLER

sheet 1 of 2

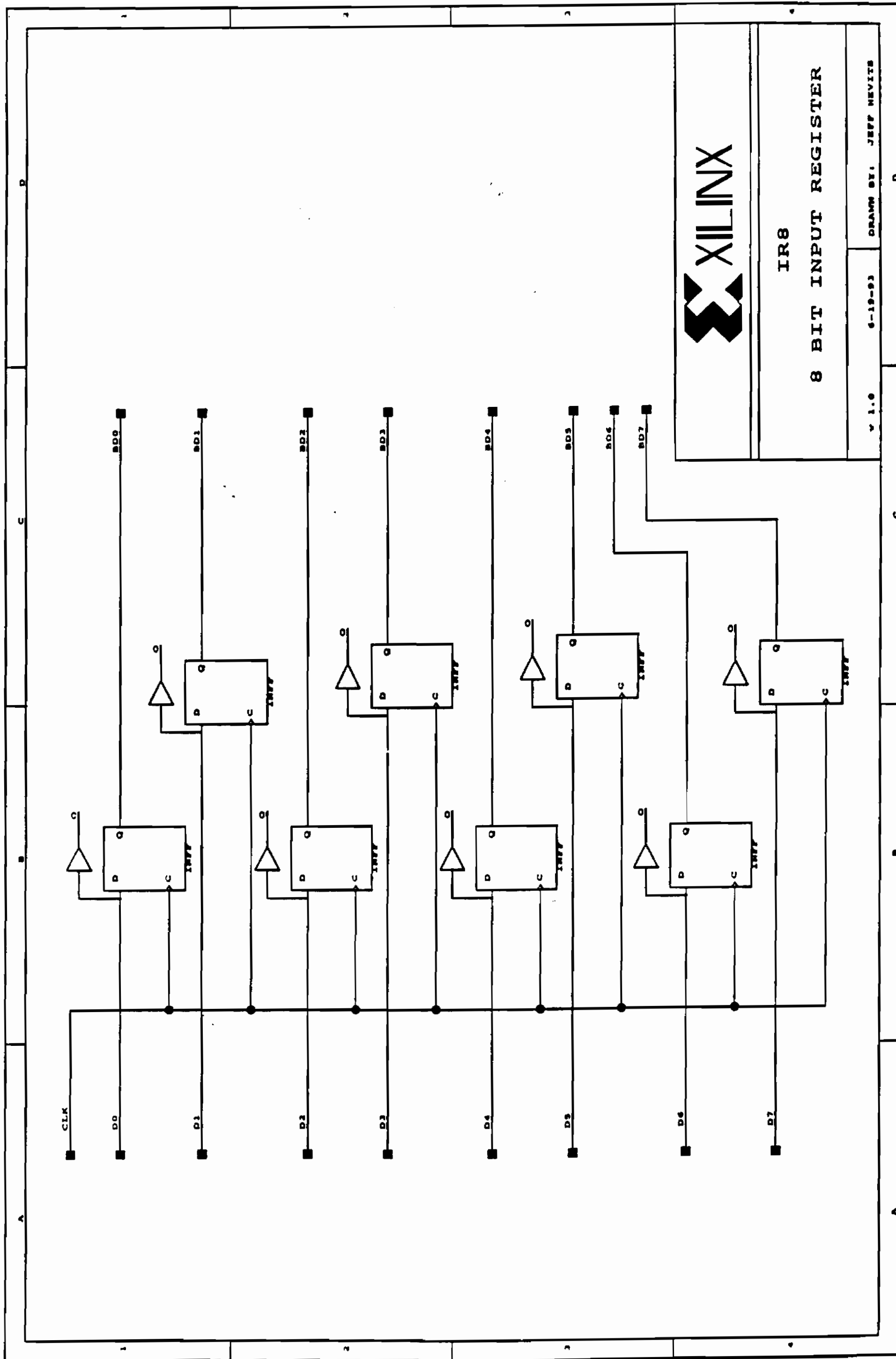
v 2.0 11-20-93 DRAWN BY: JEFF NEVITS

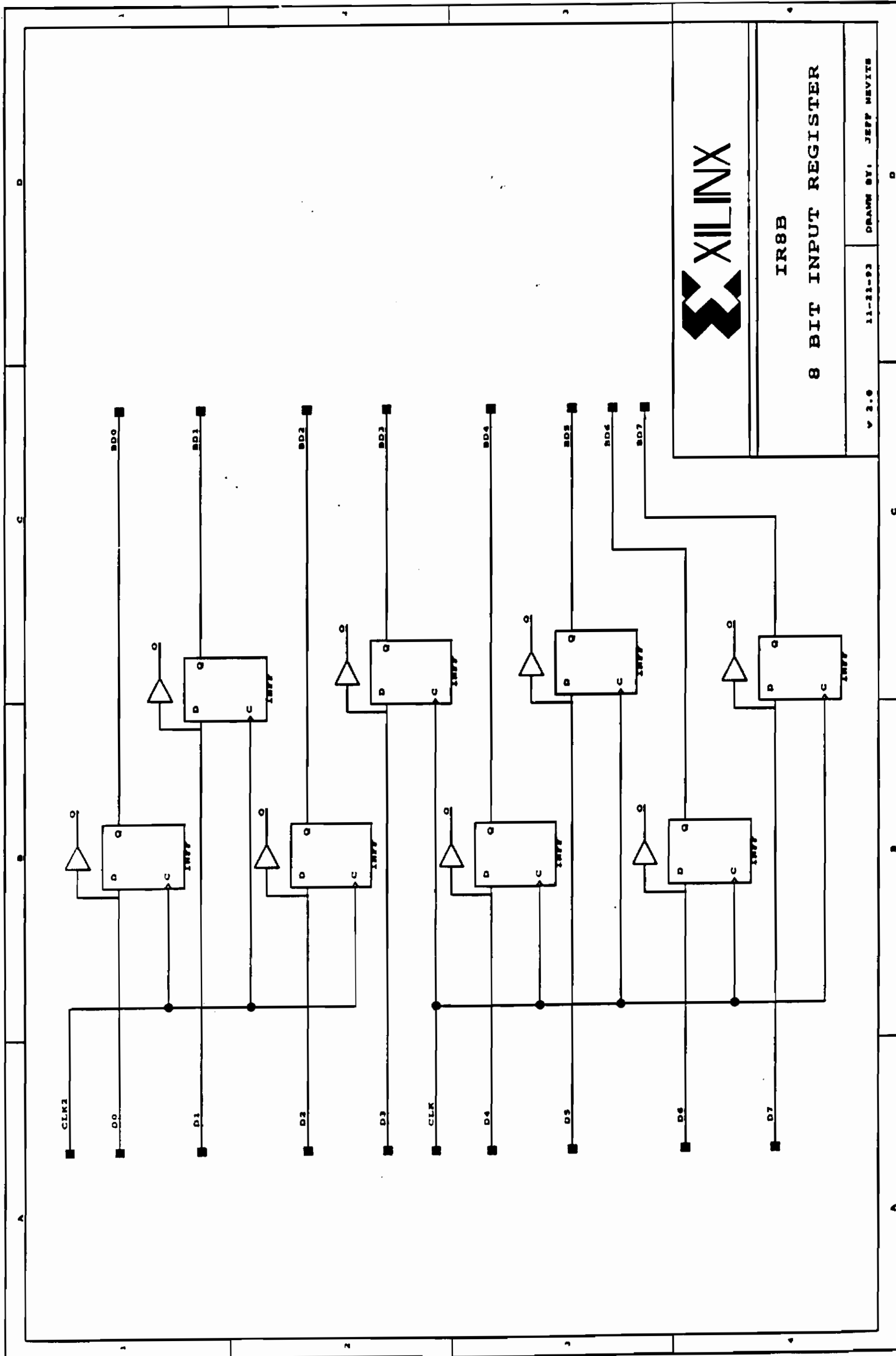
BANK CONTROLLER

sheet 2 of 2 image control

v 1.5 11-30-93 DRAWN BY: JEFF NEVITS

FIELD, VAL LINE, BANK TOG, CLK, RESET, GET FR, DONE


XILINX
IR8
8 BIT INPUT REGISTER


XILINX
IR8B
8 BIT INPUT REGISTER
11-21-93
DRAWN BY: JEFF NEVITS
V 2.0
BD0
BD1
BD2
BD3
BD4
BD5
BD6
BD7
CLK2
CLK
D0
D1
D2
D3
D4
D5
D6
D7

EDGE DETECTOR

V 1.2 | 11/6/93 | DRAWN BY: JEFF NEVITS

Q

AND2

FD

FD

AND2B1

OR2

AND2B2

AND2B1

D

CLK

IR2

2 BIT INPUT REGISTER

v 1.0 | 6-19-93 | DRAWN BY: JEFF NEVITS

D
CLK
R
OR2
AND2B1
FDR
FDR
AND2
S

XILINX
ONE SHOT
version b: synchronous 1-shot
v 2.1
DRAWN BY: JEFF NEVITS

AOB
ADDRESS OUTPUT BUFFER

Appendix D
XC 3090 Design

This appendix contains state machines, pinouts and schematics for the XC 3090 design. The XC 3090 is used on JB1 as the interface between the local memory and Splash and the ISA bus as described in Section 3.5.

The controller state machine shown in Figure D.1 interfaces with the bank switch state machine of the XC 3042. The states are defined as follows.

A: Reset state. Waiting until 3042 is ready to switch banks. Rdy_2 is set active.
B: Detected rdy_1. Set rdy1_ack high to acknowledge. Wait for 3042 to acknowledge.
C: Received rdy2_ack from 3042. Wait for rdy_1 to go low to indicate switch complete.
D: Detected rdy_1 low. Acknowledge by setting rdy1_ack low. Also set rdy_2 low.
E: Received low rdy2_ack . Set img_rdy high.
F: Dummy state. Wait for img_req or rdy_1.
G: Received img_req. Set img_rdy low, img_xfer high and generate new frame pulse.
H: Frame transfer proceeding. Wait for tc to go active to mark end of transfer.

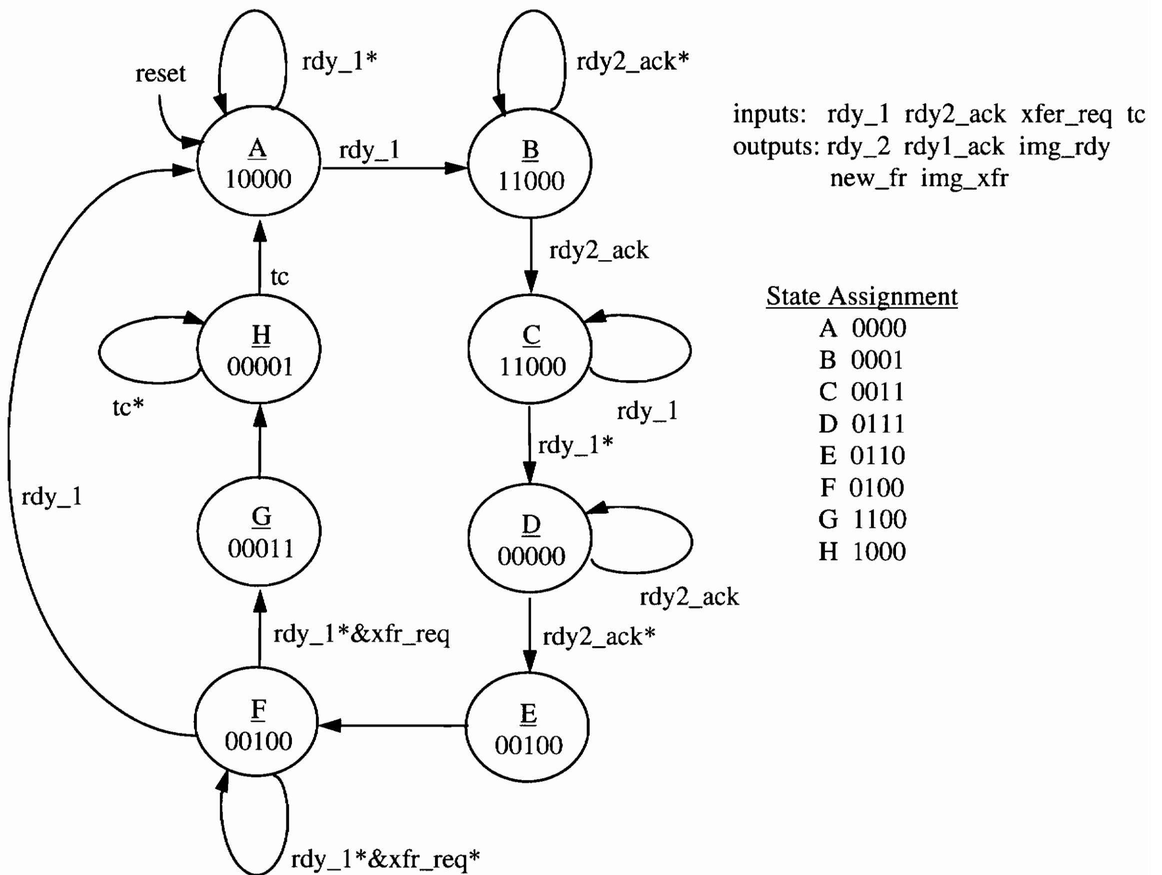


Figure D.1: Controller state machine. The FSM controls the XC 3090 handshaking needed for bankswitching.

Upon reset, the controller the 3090 is ready to switch banks and is waiting for the 3042 to indicate that it is ready to switch banks. The process uses the handshaking protocol outlined in the bank switch state machine in Appendix B. The handshaking uses states A through F. There are two possible paths out of state F. The first path is taken if an image transfer is requested before the XC 3042 requests another bank switch. In this case, the image ready flag is set low, the image transfer flag is set high and a new frame pulse is generated. The controller will then remain in state H until the image transfer is complete. Upon completion, the 3090 is ready to switch banks. The second path is taken when the XC 3042 is ready to switch banks before an image transfer is requested. If this happens, a bank switch will occur and a frame will be lost. In this manner, the latest frame will always be available, even if the XC 3090 is not transferring at a real-time rate.

XC 3090 Pinout

The XC 3090 used on JB1 is a 175 pin device arranged in a pin grid array. The pins of the XC 3090 are divided into three groups: permanently dedicated pins, dual function pins and generic I/O pins. The permanently dedicated are used to support basic chip functions and can not be used by the design. The dual function pins are used to configure the device. After configuration, these pins may be used by the design. In Table D.1, the dual function pins contain two names. The first is the name of the signal used during configuration and the second, in parentheses, is the name of the signal after configuration. The generic I/O

pins become defined after configuration. For a complete detailed list of the dedicated pins and the dual function pins, see [17].

Table D.1: XC 3090 pin mappings.

PIN	NAME	NOTES
D9, D14, H14, P14, N9, P3, H3, D3	Vcc	Dedicated power pins
D8, 14, J14, N14, N8, N3, J3, C3	GND	Dedicated ground pins
B2	PWRDWN*	Dedicated power down pin
R15	RESET*	Dedicated reset line
R2	CCLK	Dedicated configuration clock
R14	DONE/PROG*	Dedicated programming pin
B15, B14	M0, M1	Dedicated programming mode pins
C15	M2	Mode pin. Unused after configuration.
P5	RDY/BUSY*	Programming pin. After configuration, unused input to PC
R10	CS0* (A1)	Programming chip select. After configuration, PC address bit A1
R8	CS1* (A2)	Programming chip select. After configuration, PC address bit A2
M3	CS2 (3090_Sel)	Programming chip select. After configuration, 3090 select line
P9	CS3 (A0)	Programming chip select. After configuration, PC address bit A0
P2	WS* (I/O_W*)	Programming write strobe. After configuration, PC I/O write strobe
E14	HDC	High during configuration. Unused after configuration
D16	LDC	Low during configuration. Unused after configuration
N7	ADD3	PC address bit A3
P7	BS*	Board Select
H15	INIT*	Initialization pin. Unused after configuration

R3, R5, R7, P8, R9, T11, P12, N13	PD0-PD7	Configuration data bus. After configuration, data bus to PC
A14	I/O_R*	PC I/O read strobe
D5	CLK	Clock from clock generator
K3	RDY_1	Bank Switch Handshaking Line from XC 3042
L3	RDY_2	Bank switch handshaking line to XC 3042
P1	RDY1_ACK	Bank switch handshaking line from XC 3042
N1	RDY2_ACK	Bank switch handshaking line to XC 3042
G3	BANK	Bank Flag
A12	E/O	Field Indicator
D10	AECS1*	Memory Bank 0, Odd chip select
D7	AOCS2	Memory Bank 0, Even chip select
D12	BECS1*	Memory bank 1, Odd chip select
D11	BOCS2	Memory bank 1, Even chip select
A13	R/W*	Memory read/write strobe
J1, H1, G1, F1, E1, D1, C1, B4	AA0-A7	Bank 0 address lines A0-A7
B5, B6, B7, B8, B9, B10, B11, B12, B13	AA8-AA16	Bank 0 address lines A8-A16
J2, H2, G2, F2, E2, D2, C2, C4	BA0-BA7	Bank 1 address lines A0-A7
B8, C5, C6, C7, C8, C(, C10, C11, C12, C13	BA8-BA16	Bank 1 address lines A8-A16
A4, A5, A6, A7, A8, A9, A10, A11	D0-D7	Memory data bus
N2	IM_RDY	Image Ready Flag
R1	IMG_XFR	Image Being Transferred Flag
B16, C16, D13, E16, F16, G16, H16, J16, K16, L16, M16, N16	S0-S11	Splash data bus S0-S11
P16, R16, P15, N15, M15, L15, K15, J15, G15, F15, E15, D15	S12-S23	Splash data bus S12-S23
F14, G14, K14, L14, M14, T14, R13, P13, N12, R12, T12, R11	S25-S35	Splash data bus S25-S35
P11, N11, N10, P10	HS0-HS3	Splash Handshaking Lines

PC-XILINX-MEMORY INTERFACE

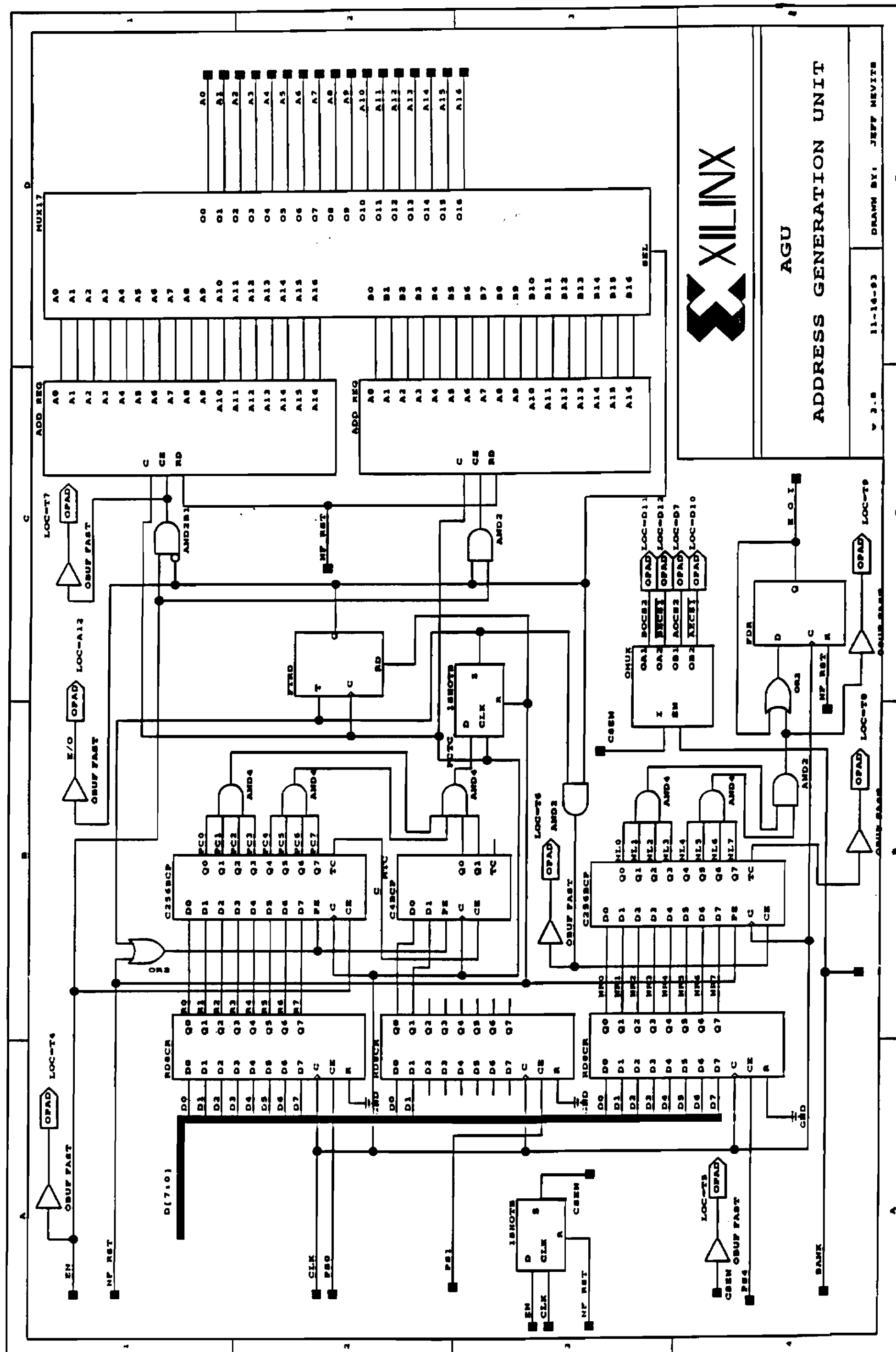
XC 3090 SHEET 1 OF 1

11-11-93

DRAWN BY: JEFF NEVITT

V 2.5

XILINX
PORT DECODING
V 1.0
11-8-93
DRAWN BY: JEFF NEVITS


XILINX
AGU
ADDRESS GENERATION UNIT

CONTROL 2

V 1.0 | 11-30-93 | DRAWN BY: JEFF NEVITS

D
CLK
R
FDR
D
C
R
Q
OR2
FDR
D
C
R
Q
AND2B1
AND2
E

ONE SHOT

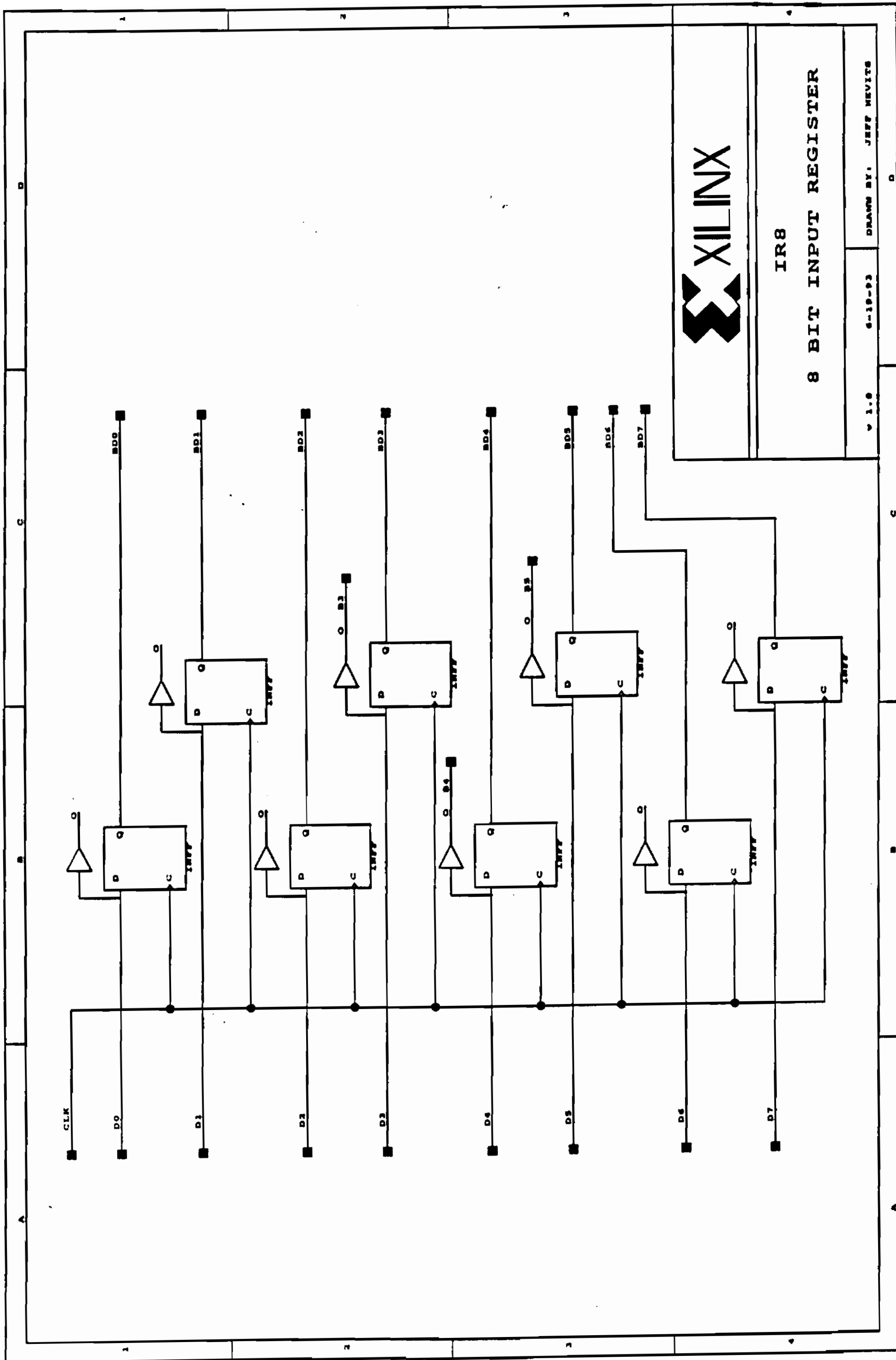
version b: synchronous 1-shot

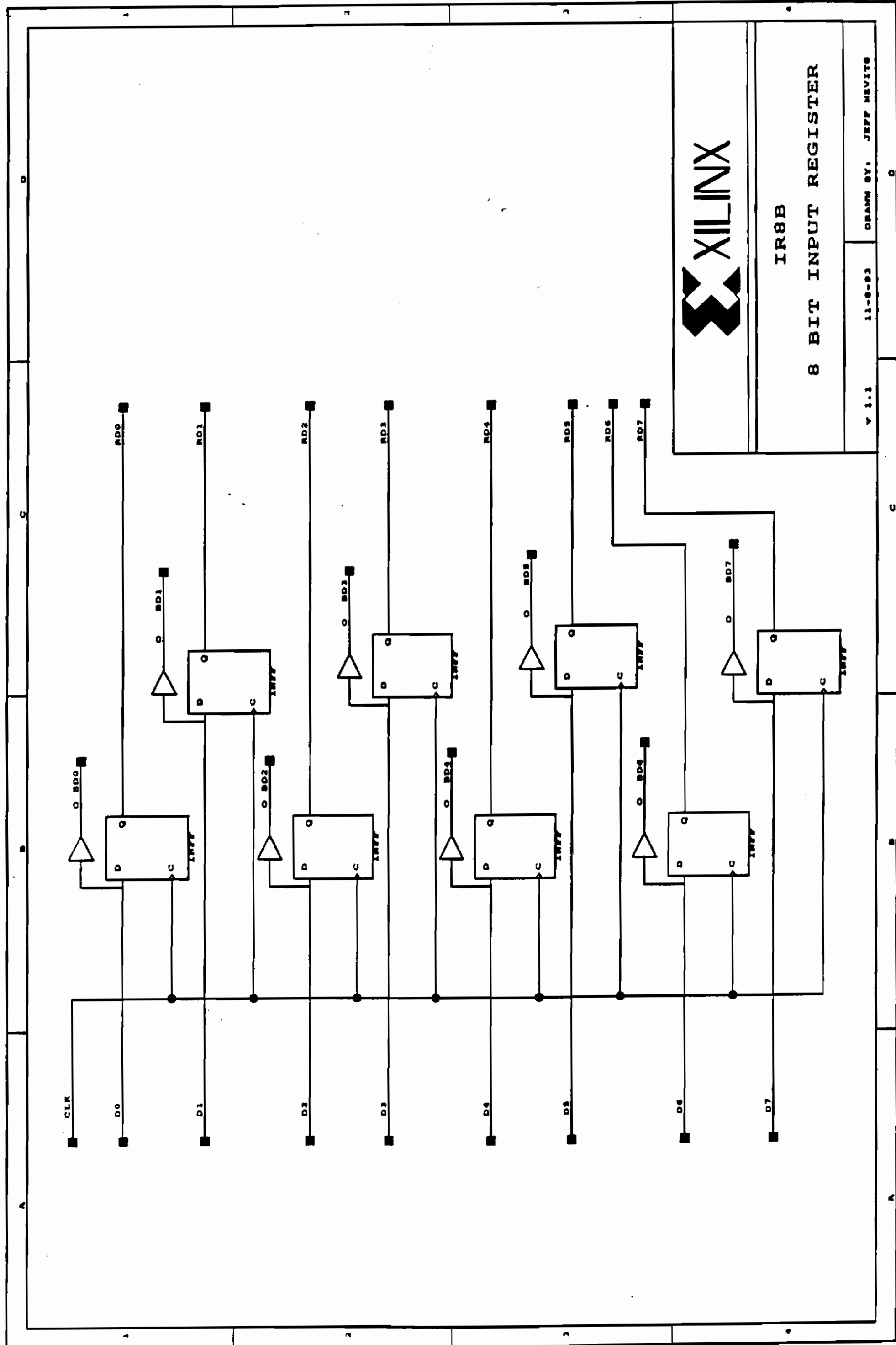
v 2.1 8-25-93 DRAWN BY: JEFF NEVITS

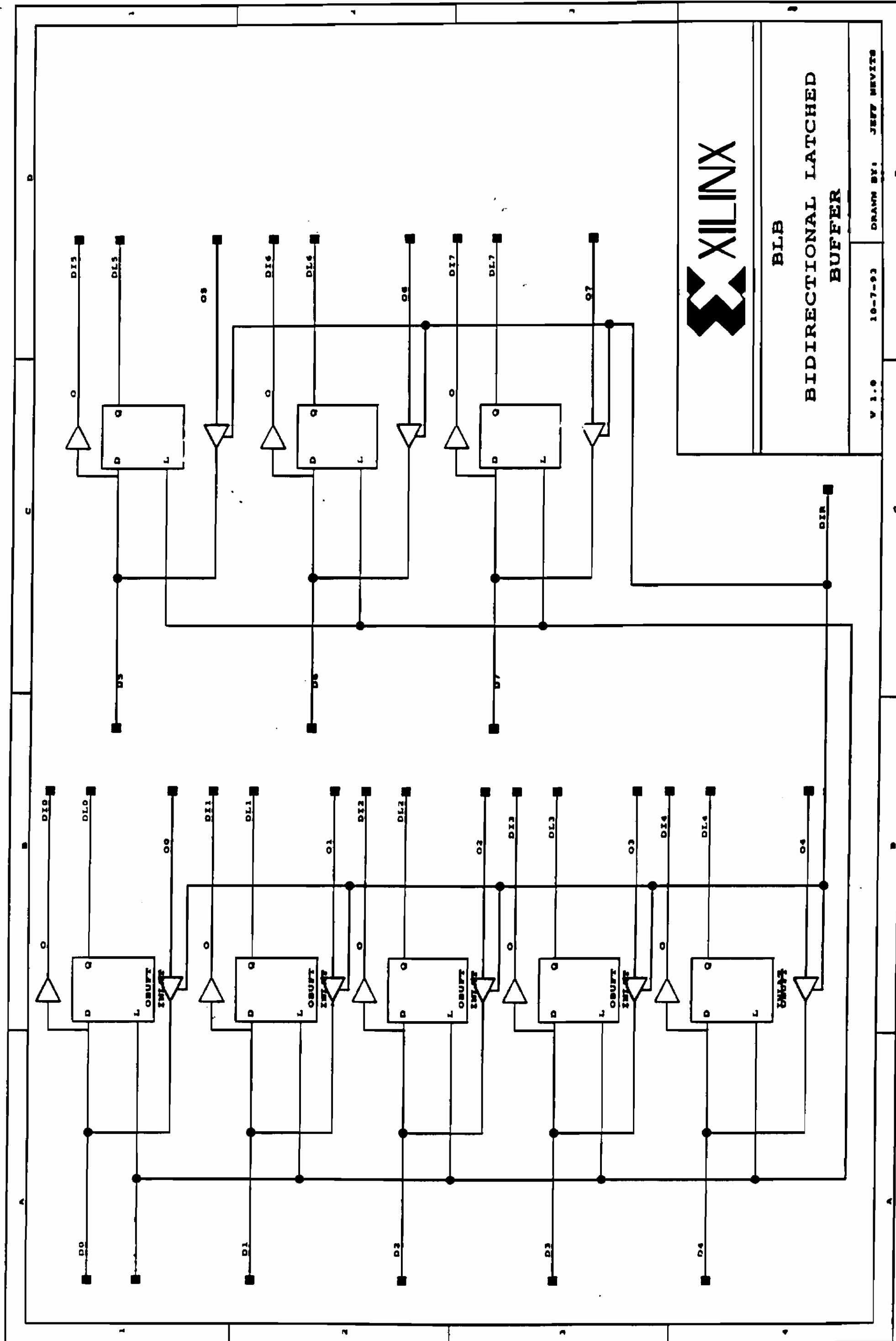
EDGE DETECTOR

v 1.3 11/6/93 DRAWN BY: JEFF NEVITS

AND3B1
FDR
D
C
R
Q
AND2
Q
AND2B2
OR2
AND2B1
D
RST
CLK


XILINX
IR8
8 BIT INPUT REGISTER


XILINX
IR8B
8 BIT INPUT REGISTER


XILINX
BLB
BIDIRECTIONAL LATCHED
BUFFER

BRB

BIDIRECTIONAL REGISTERED BUFFER

V 1.0 | 10-7-93 | DRAWN BY: JEFF NEVITT

OMUX

2-BIT OUTPUT MUX

DRAWN BY:

EN
I
OA1
OA2
OB1
OB2
OBUFT
FAST

XILINX
ADDRESS REGISTER
V 3.5
12-7-93
DRAWN BY: JEFF NEVITS
C256BCP
C256BCP
C2BCP
VCC
A0 A1 A2 A3 A4 A5 A6 A7
A8 A9 A10 A11 A12 A13 A14 A15
A16
RD
C
CE

AOB

ADDRESS OUTPUT BUFFER

V 1.0 6-18-93 DRAWN BY: JEFF NEVITS

MUX17

17 2-1 MULTIPLEXERS

DRAWN BY: JEFF NEVITS

Appendix E

Grabber Source Code Listing

This appendix contains the C source code for the JB1 device driver, *Grabber*. This is used to configure the JB1 frame grabber board for operation. The software runs on a PC AT computer with a Paradise video card running DOS version 3.3 or higher. The code was compiled using Borland's C++ compiler version 3.0.

```
/*******************************************************************
* PROGRAM: GRABBER.C
* AUTHOR: JEFF NEVITS              COPYRIGHT 1993 VIRGINIA TECH
*
* JB1 DEVICE DRIVER: THIS PROGRAM CONTROLS THE SETUP AND USE
* OF THE JB1 BOARD.
*
* I/O PORTS USED BY BOARD: 0x0300 - 0x030F
*
* PORT DESCRIPTIONS:
*    300: GENERAL INPUT PORT
*         BIT 0: RDY/BUSY* 3042
*             1: IMG_XFR
*             2: DONE/PROG* 3042
*             3: INIT* 3042
*             4: RDY/BUSY* 3090
*             5: IMG_RDY
*             6: DONE/PROG* 3090
*             7: INIT* 3090
*    301: GENERAL LATCHED OUTPUT PORT
*         BIT 0: DONE/PROG* 3042 (OPEN COLLECTOR)
*             1: RESET* 3042
*             2: UNUSED
*             3: UNUSED
*             4: DONE/PROG* 3090 (OPEN COLLECTOR)
*             5: RESET* 3090
*             6: UNUSED
*             7: UNUSED
*    302: PROGRAMMABLE CLOCK REGISTER
*    303: FOR EXPANSION
*    304: XC3042 PROGRAMMING\CONTROL PORT AFTER PROGRAMMING
*    305: XC3042 INVALID PIXEL MASK REGISTER
*    306: XC3042 LOWER BYTE OF VALID PIXEL REGISTER
*    307: XC3042 UPPER BYTE OF VALID PIXEL REGISTER
*    308: XC3090 PROGRAMMING\NUM_PIXELS_LO PER LINE AFTER CNFG
*    309: XC3090 PROGRAMMING\NUM_PIXELS_HI PER LINE AFTER CONFG
*    30A: XC3090 FOR EXPANSION
*    30B: XC3090 DATA PORT TO MEMORY
*    30C: XC3090 NUM_LINES PER IMAGE
*    30D  XC3090 FOR EXPANSION
*    30E: XC3090 FOR EXPANSION
*    30F: XC3090 CONTROL PORT
*******************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <dos.h>
#include <dir.h>
#include <math.h>
#include "pvga.h"

#define INREG        0x0300
#define OUTREG       0x0301
#define SCLK         0x0302
#define PROG_3042    0x0304
#define CON_42       0x0304
#define INVAL_PM     0x0305
#define VAL_PR_LB    0x0306
#define VAL_PR_HB    0x0307
#define PROG_3090    0x0308
#define N_PIX_LO     0x0308
#define N_PIX_HI     0x0309
#define DATA_PORT    0x030B
#define NUM_LINES    0x030C
#define CON_90       0x030F

/* Function prototypes */
void menu(int *choice, int *img_size),
   auto_config(void),
   prog_xc(char name[], int port, int reset, int prog, int rdy_mask,int D_P_msk, int
      in_mask,char dev[]),
   capture(int *img_size),
   down_load(void),
   parameters(int *img_size),
   graphics(int *img_size),
   pause(void),
   get_key(int low, int high, int *choice),
   file_open(char ext[],FILE **stream),
   memory(int *img_size),
   clock_setup(void),
   write_mem(int *img_size),
   read_mem(void),
   reset_xilinx(void);

int gprintf(int *xloc, int *yloc, char *fmt, ... );
```

```
/****************************************************************
*
*  MAIN FUNCTION
*
*  THIS FUNCTION CALLS THE MAIN MENU FOR THE DRIVER.
*
****************************************************************/

int main()
   {
      int     choice = 0,
              img_size=512;

      while (choice!=56)
      {
         menu(&choice,&img_size);
      }
      return(0);
   }

/****************************************************************
*
*  MENU
*
*  THIS FUNCTION DISPLAYS THE AVAILABLE OPTIONS AND CALLS
*  THE APPROPRIATE SUB-ROUTINES.
*
****************************************************************/

void menu(int *choice, int *img_size)
   {
      int     value;

      clrscr();
      printf("\n\n\n\t\t     FRAME GRABBER DEVICE DRIVER\n\n");
      printf("\n\t\t   0. Automatically configure board");
      printf("\n\t\t   1. Download Configuration to LCA \n\t\t");
      printf("   2. Change Capture Parameters \n\t\t   3.");
      printf(" Continuously capture images\n\t\t   4.");
      printf(" Display image \n\t\t   5. Read/Write on-board memory");
      printf("\n\t\t   6. Change clock setup\n\t\t   7. Reset Xilinx");
      printf("\n\t\t   8. Exit program");
      get_key(48,56,&value);
```

```
    choice = value;
    switch (*choice)
        {
        case 48: auto_config();
            break;
        case 49: down_load();
            break;
        case 50: parameters(img_size);
            break;
        case 51: capture(img_size);
            break;
        case 52: graphics(img_size);
            break;
        case 53: memory(img_size);
            break;
        case 54: clock_setup();
            break;
        case 55: reset_xilinx();
            break;
        case 56: break;
        }
  }

/*****************************************************************
*
*  AUTOCONFIGURE
*
*  THIS ROUTINE AUMATICALLY CONFIGURES THE FRAME GRABBER
*  BOARD USING DEFAULT CONFIGURATIONS AND SETTINGS.
*
*****************************************************************/

void auto_config(void)
  {
    int   prog_port,
          reset,
          prog,
          ready_mask,
          D_P_mask,
          init_mask,
          junk;
    char  dev[8],
          name[16];
```

```
    clscr();
    printf("\n\n\t\t\t   AUTO_CONFIGURATION");
    /* 3042 settings */
    strcpy(dev,"XC 3042");
    prog_port  = PROG_3042;            /* set address to XC3042*/
    reset      = 0x31;                 /* reset value for 3042 */
    prog       = 0x32;                 /* prog* value for 3042 */
    ready_mask = 0x01;                 /* mask for bit 0 */
    D_P_mask   = 0x04;                 /* mask for bit 2 */
    init_mask  = 0x08;                 /* mask for bit 3 */
    strcpy(name,"XC3042.POD");
    prog_xc(name,prog_port,reset,prog,ready_mask,D_P_mask,init_mask,dev);
    /* 3090 settings */
    strcpy(dev,"XC 3090");
    prog_port  = PROG_3090;            /* set address to XC3090*/
    reset      = 0x13;                 /* reset value for 3090 */
    prog       = 0x23;                 /* prog* value for 3090 */
    ready_mask = 0x10;                 /* mask for bit 4 */
    D_P_mask   = 0x40;                 /* mask for bit 6 */
    init_mask  = 0x80;                 /* mask for bit 7 */
    strcpy(name,"XC3090.POD");
    prog_xc(name,prog_port,reset,prog,ready_mask,D_P_mask,init_mask,dev);
    get_key(1,128,&junk);
  }

/*********************************************************************
*
*  PROG_XC
*
*  THIS ROUTINE PROGRAMS THE XILINX CHIPS.
*
*********************************************************************?

void prog_xc(char name[],port,rst,prog,rdy_msk,D_P_msk,in_msk,char dev[])
  {
    int     status,
            D_P,
            ready,
            init;
    char  a[3];
    FILE *stream;
```

```
stream= fopen(name,"r");
if (stream==NULL)
   printf("\n\n\t\tFILE ERROR! CAN'T OPEN FILE");
else
   {
   status = inportb(INREG);
   D_P = status & D_P_msk;
   if (D_P==D_P_msk)
      {                          /* device has previously been programmed */
      printf("\n\t\t   Device previously programmed ");
      outportb(OUTREG,prog);     /* assert prog* */
      }
   printf("\n\t\t   Resetting device %s .....",dev);
   outportb(OUTREG,rst);         /* drive reset low & prog* high */
   outportb(OUTREG,0x33);        /* drive reset high */
   init = 0;
   while (init==0)
      {
      status= inportb(INREG);    /* read from input port */
      init = status&in_msk;      /* mask bits to get init value */
                                 /* wait for init & clear states to finish */
      }
   printf("device cleared");
   printf("\n\t\t   Programming %s.....",dev);
   if (stream != NULL)
      {
      while (fgets(a,3,stream) != NULL)
         {
         strcpy(base,"0x");
         strncat(base,a,3);
         value= strtol(base,NULL,0);
         outportb(port,value) ;  /* write out data */
         ready = 0 ;
         while (ready==0)
            {                    /* wait for RDY/BUSY* line to go high */
            status=inportb(INREG);
            ready = status & rdy_msk;
            }
         }
      printf("finished");
      outportb(OUTREG,0xff);
      }
   fclose(stream);
```

```
            delay(100);
            status = inportb(INREG);
            D_P = status & D_P_msk;
        if (D_P==D_P_msk)
            printf("\n\n\t\t    Device successfully programmed");
        else
            printf("\n\n\t\t    PROGRAMMING ERROR! D_P is still low");
        printf("\n\n\t\t\t        Hit any key...");
        get_key(1,128,&choice2);
        }
    }

/*****************************************************************
*
*  DOWN_LOAD
*
*  THIS ROUTINE OPENS THE .POD FILE CREATED BY MAKEPODF AND
*  DOWNLOADS THE VALID CONFIGURATION DATA TO AN LCA DEVICE.
*
*  TO PROGRAM THE XC 3042, DATA IS WRITTEN TO PORT 0x304.
*  TO PROGRAM THE XC 3090, DATA IS WRITTEN TO PORT 0x308 OR 0x309.
*  (DURING THE PROGRAMMING PROCESS, THESE TWO ADDRESSES ARE
*  IDENTICAL.  AFTER PROGRAMMING IS COMPLETE, THE TWO ADDRESSES
*  ARE FOR TWO DIFFERENT PORTS.)
*
*****************************************************************/

void down_load(void)
    {
        char  a[3],
              base[5],
              device[8],
              ext[4] = "pod";
        unsigned char status;
        int   choice=0,
              choice2,
              D_P,
              D_P_mask,
              HDC,
              HDC_mask,
              i,
              init,
              init_mask,
```

```
        prog,
        prog_port,
        ready,
        ready_mask,
        reset,
        value;
FILE *stream;

while (choice!=51)
{
    clrscr();
    printf("\n\t\t        DOWNLOAD CONFIGURATION TO LCA\n");
    printf("\n\n\t\t\t   1. Configure XC 3042\n\t\t\t   2. Configure ");
    printf("XC 3090 \n\t\t\t   3. Return to main menu");
    get_key(49,51,&choice);
    if (choice!=51)
        {
        if (choice==49)
            {
            strcpy(device,"XC 3042");      /* set-up for 3042 configuration */
            prog_port  = PROG_3042;        /* set address to XC3042*/
            reset      = 0x31;             /* reset value for 3042 */
            prog       = 0x32;             /* prog* value for 3042 */
            ready_mask = 0x01;             /* mask for bit 0 */
            D_P_mask   = 0x04;             /* mask for bit 2 */
            init_mask  = 0x08;             /* mask for bit 3 */
            }
        else
            {
            strcpy(device,"XC 3090");      /* set-up for 3090 configuration */
            prog_port  = PROG_3090;        /* set address to XC3090*/
            reset      = 0x13;             /* reset value for 3090 */
            prog       = 0x23;             /* prog* value for 3090 */
            ready_mask = 0x10;             /* mask for bit 4 */
            D_P_mask   = 0x40;             /* mask for bit 6 */
            init_mask  = 0x80;             /* mask for bit 7 */
            }
        file_open(ext,&stream);
        status = inportb(INREG);
        D_P = status & D_P_mask;
        if (D_P==D_P_mask)
            { /* device has previously been programmed */
            printf("\n\t\t   Device previously programmed ");
```

```
        outportb(OUTREG,prog);      /* assert prog* */
        }
    printf("\n\t\t   Resetting device %s .....",device);
    outportb(OUTREG,reset);         /* drive reset low & prog* high */
    outportb(OUTREG,0x33);          /* drive reset high */
    init = 0;
    while (init==0)
        {                           /* wait for init & clear states to finish */
        status= inportb(INREG);     /* read from input port */
        init = status&init_mask;    /* mask bits to get init value */
        }
    printf("device cleared");
    printf("\n\t\t   Programming %s.....",device);
    if (stream != NULL)
        {
        while (fgets(a,3,stream) != NULL)
            {
            strcpy(base,"0x");
            strncat(base,a,3);
            value= strtol(base,NULL,0);
            outportb(prog_port,value) ; /* write out data */
            ready = 0 ;
            while (ready==0)
                {                   /* wait for RDY/BUSY* line to go high */
                status=inportb(INREG);
                ready = status & ready_mask;
                }
            }
        printf("finished");
        outportb(OUTREG,0xff);
        }
  fclose(stream);
  delay(100);
  status = inportb(INREG);
  D_P = status & D_P_mask;
  if (D_P==D_P_mask)
    printf("\n\n\t\t   Device successfully programmed");
  else
    printf("\n\n\t\t   PROGRAMMING ERROR! D_P is still low");
  printf("\n\n\t\t\t      Hit any key...");
  get_key(1,128,&choice2);
  }
}
```

```
   }

/*********************************************************************
*  CAPTURE PARAMETERS
*
*  THIS ROUTINE ALLOWS THE USER TO CHANGE THE PARAMETERS USED
*  TO CAPTUREVIDEO IMAGES.  AVAILABLE ASPECT RATIOS FOR THE
*  IMAGE ARE 1:1 AND 4:3.  FOR 1:1 IMAGES, PART OF THE IMAGE IS NOT
*  COLLECTED.  THE VIDEO DATA COLLECTED MAY BE FROM THE LEFT
*  PORTION OF THE IMAGE OR THE CENTER PORTION OF THE IMAGE.
*
*********************************************************************/



void parameters(int *img_size)
   {
      int     choice =0,
              choice2=0,
              data_lb=0,
              data_hb=0,
              delay  =0;

      while (choice!=51)
         {
         clrscr();
         printf("\n\n\n\t\t\t   CAPTURE PARAMETERS\n");
         printf("\n\n\t\t\t1. Change aspect ratio \n\t\t\t2.");
         printf(" Change window for 1:1 AR\n");
         printf("\t\t\t3. Return to main menu");
         get_key(49,51,&choice);
         switch (choice)
            {
            case 49:printf("\n\n\t\t\t1. 1:1 aspect ratio");
               printf("\n\t\t\t2. 4:3 aspect ratio");
               get_key(49,50,&choice2);
               if (choice2==50)
                  {
                  *img_size = 512;          /* 512 for 4:3 aspect ratio */
                  data_lb = 0x00;           /* set low byte */
                  data_hb = 0x02;           /* set high byte */
                  printf("\n\n\t\t\tAspect ratio set to 4:3.\n");
                  printf("\n\t\t\tImage size: 480 x 512\n\n");
```

```
			}
		else
			{
			*img_size = 384;		/* 384 for 1:1 aspect ratio */
			data_lb = 0x080;		/* set low byte */
			data_hb = 0x002;		/* set high byte */
			printf("\n\n\t\t\tAspect ratio set to 1:1.\n");
			printf("\n\t\t\tImage size: 480 x 384\n\n");
			}
		delay = 0xa9;	/* 0xaa 4:3: full image; 1:1: left just.*/
		outportb(VAL_PR_LB,data_lb);
		outportb(VAL_PR_HB,data_hb);
		outportb(INVAL_PM,delay);
		printf("\n\t\t\tData written to ports\n");
		printf("\n\t\t\t   Press any key...");
		get_key(1,128,&choice2);
		break;
	case 50:if (*img_size==384)
			{
			printf("\n\n\t\t\t1. Centered\n\t\t\t2. Left justified");
			get_key(49,50,&choice2);
			if (choice2==49)
				{
				delay = 0x063;
				printf("\n\n\n\t\t\tWindow is centered.\n");
				}
			else
				{
					delay = 0x0a9;
			printf("\n\n\n\t\t\tWindow is left justified.\n");
				}
				outportb(INVAL_PM,delay);
			printf("\n\t\t\tData written to ports");
			}
			else
				printf("\n\n\t\t   Windowing only available for 1:1 AR!");
			printf("\n\n\n\t\t\t   Press any key...");
			get_key(1,128,&choice2);
			break;
	case 51: break;
	}
    }
}
```

```
/*********************************************************************
*
*  IMAGE CAPTURE
*
*  THIS ROUTINE CONTINUOUSLY CAPTURES VIDEO FRAMES AND
*  UPDATES THE SCREEN.
*
*  IMAGE SIZE:
*     4:3 AR: 512 x 480
*     1:1 AR: 384 x 480
*
*********************************************************************/
void capture(int *img_size)
   {
      int     color,
              done,
              err_cd,
              img_count=0,
              index,
              reg,
              status,
              status1,
              status2,
              xcoor,
              ycoor,
              val_mem;
      long int   count,
                 lcount;

      clrscr();
      img_count=0;
      graphmode();
      dac(0,0,0,0);
      dac(255,63,63,63);
      smear(0,255);
      reg=6;
      while (img_count<15)
         {
         ++img_count;
         outportb(CON_90,reg);              /* image request */
         status2=0;
         err_cd = 0;
```

```
        lcount =0;
        while (status2==0 & err_cd==0)
            {
            ++ lcount;
            if (lcount==25000) err_cd=1;
            status = inportb(INREG);       /* polling for Img_Xfr flag */
            status1= status &32;
            status2= status&2;
            }
        if (status1==0 & status2==2)
            {                              /* Transfer ready to proceed */
            outportb(CON_90,1);            /*  Direction set for reading */
            done=2;
            count = 0;
            xcoor=0;
            ycoor=0;
            while(done!=0)
                {
                status = inportb(INREG);
                done =status&2;
                if ((done!=0) && (count<256000))
                    {
                    ++index;
                    ++xcoor;
                    if (xcoor>*img_size-1)  /* >*img_size-1 */
                        {
                        xcoor =0;
                        ++ycoor;
                        }
                    color=inportb(DATA_PORT)^255;
                    if (ycoor<480)plot(xcoor,ycoor,color);
                    }
                }
            }
        }
    textmode();
}
```

```
/*********************************************************************
*
*  GRAPHICS CONTROL
*
*  THIS ROUTINE DISPLAYS AN IMAGE FILE FROM DISK.
*
*********************************************************************/

void graphics(int *img_size)
  {
     char  ext[4]="raw";
     FILE *stream;
     long  index=-1;
     int   color,
           field,
           i,
           junk,
           junk2,
           xcoor=0,
           ycoor=0;

     clrscr();
     printf("\n\n\t\t\t    DISPLAY IMAGE");
     file_open(ext,&stream);              /* open file */
     graphmode();                         /* set display to graphics mode */
     dac(0,0,0,0);                        /* set-up greyscale shades */
     dac(255,63,63,63);
     smear(0,255);
     xcoor=0;
     ycoor=0;
     field=0;
     while (!feof(stream))
        {
        ++index;
        ++xcoor;
        if (xcoor>*img_size-1)
           {
           if (field==0)field =1;
           else field=0;
           xcoor =0;
           ++ycoor;
           }
        color=getc(stream);
```

```
        if (ycoor<480)plot(xcoor,ycoor,color);     /* plot valid pixels */
        }
    get_key(1,128,&junk);
    fclose(stream);
    textmode();                                  /* Return the system to text mode   */
    get_key(1,128,&junk); */
  }

/*****************************************************************
*
*  CLOCK_SETUP
*
*  THIS FUNCTION PROGRAMS THE SPLASH CLOCK.
*  CLOCK LOCATION: I/O ADDRESS 302H.
*     BIT 0:      LSB OF CLOCK DIVIDER
*     BIT 1:      MIDDLE BIT
*     BIT 2:      MSB
*     BIT 3:      CLOCK ENABLE (1=ON)
*     BITS 4-7:   UNUSED
*
*  BASE FREQUENCY:         F0 = 20 MHz
*  DIVIDED FREQUENCY:      Fdiv = 20/(2^(N+1))
*
*****************************************************************/

void  clock_setup(void)
  {
    static int enable=0;
    float  freq,
           value2;
    int    choice =0 ,
           choice2 = 9,
           port_val,
           value1;
    char   status[10],
           status2[10];

    while (choice!=51)
        {
        clrscr();
        printf("\n\n\n\t\t\t   CLOCK SETUP");
        printf("\n\n\t\t\t1. Change Clock Frequency \n\t\t\t2.");
        printf(" Enable/Disable Clock\n\t\t\t3. Return to main menu");
```

```
        get_key(49,51,&choice);
        switch (choice)
            {
            case 49: printf("\n\n\t\t\tSelect Frequency:");
                printf("\n\n\t\t\t1. 10 MHz\n\t\t\t2. ");
                printf("5 MHz\n\t\t\t3. 2.5 MHz\n\t\t\t");
                printf("4. 1.25 MHz\n\t\t\t5. 0.625 MHz\n");
                printf("\t\t\t6. 0.3125 MHz\n\t\t\t7. 0.156 MHz");
                printf("\n\t\t\t8. 0.78 MHz");
                get_key(49,56,&choice2);
                port_val= choice2-49;
                value2 = 20/(pow(2.0,(port_val+1)));
                port_val=port_val+enable;
                outportb(SCLK,port_val);
                printf("\n\n\t\t\tCLOCK SET TO ");
                printf("%5.3f MHz",value2);
                printf("\n\n\t\t\t    Press any key...");
                get_key(1,128,&choice2);
                break;
            case 50: printf("\n\n\t\t\t1. Disable Clock\n\t\t\t");
                printf("2. Enable Clock");
                get_key(49,50,&choice2);
                if (choice2==49)
                    {
                    port_val=port_val & 247;
                    enable=0;
                    strcpy(status2,"DISABLED")
                    }
                else
                    {
                    port_val=port_val | 8;
                    enable=8;
                    strcpy(status2,"ENABLED");
                    }
                outportb(SCLK,port_val);
                printf("\n\n\t\t\t CLOCK %s",status2);
                printf("\n\n\t\t\t   Hit any key...");
                get_key(1,128,&choice2);
                break;
            case 51: break;
            }
        }
}
```

```
/*****************************************************************
*
*  MEMORY
*
*  THIS ROUTINE PROVIDES TESTING OF THE SYSTEM MEMORY
*  THROUGH THE XC3090.
*
*  THE MEMORY IS I/O MAPPED AT LOCATION 30BH.
*
*  READ/WRITE PROCEDURE:
*     1. LOAD NUMBER OF PIXELS PER LINE AND LINES PER IMAGE
*        COUNTERS
*     2. POLL BIT 5 OF INPUT PORT (300H) UNTIL IMAGE_READY FLAG
*        IS ACTIVE HIGH
*     3. WRITE TO BIT 0 OF PORT 309H TO SET DIRECTION (0=WRITE,
*        1=READ)
*     4. WRITE A 1 TO BIT 1 OF PORT 309H TO REQUEST IMAGE
*     5. POLL BIT 5 OF INPUT PORT UNTIL IMAGE_READY FLAG IS
*        INACTIVE
*     6. RESET BIT 1 OF PORT 309H TO 0.
*     7. FOR EACH PIXEL, CHECK BIT 1 TO SEE IF IMAGE_TRANSFER IS
*        HIGH.  IF SO, PERFORM READ/WRITE OPERATION TO PORT 30BH.
*
*****************************************************************/

void memory(int *img_size)
  {
    int    choice = 0;
    int    hb,
           lb,
           tmp;

    while (choice!=51)
      {
      clrscr();
      printf("\n\n\t\t\t   MEMORY READ/WRITE\n");
      reset_xilinx();*/
      printf("\n\n\t\t\tSelect but one:\n");
      printf("\n\t\t\t1. Write to memory \n\t\t\t2. Read from memory");
      printf("\n\t\t\t3. Return to main menu");
      get_key(49,51,&choice);
      if (choice==49 || choice==50)
        {
```

```
            hb = (0x0ff00 & (1023 - *img_size)) >> 8;
            lb = (0x000ff & (1023 - *img_size));
            outportb(N_PIX_LO,lb);
            outportb(N_PIX_HI,hb);
            outportb(NUM_LINES,12);
            switch(choice)
              {
              case 49:write_mem(img_size);
                break;
              case 50: read_mem();
                break;
              case 51: break;
              }
            }
         }
    }

/*******************************************************************
*
*  RESET_XILINX
*
*  THIS ROUTINE  RESETS BOTH XILINX CHIPS BY TOGGLING THE
*  FPGA RESET LINES LOCATED AT PORTS 304H AND 30FH.
*
*******************************************************************/
void reset_xilinx(void)
    {
      int    choice,
             junk,
             status,
             status1,
             status2;

      clrscr();
      printf("\n\n\t\t\t     XILINX RESET\n\n\t\t\t");
      printf("Resetting all internal logic\n");
      outportb(CON_42,1);                  /* reset 3042 */
      printf("\n\t\t\t3042 internal reset set active");
      outportb(CON_90,128);                /* reset 3090 */
      printf("\n\t\t\t3090 internal reset set active");
      printf("\n\t\t\tHit any key to release 3090 reset");
      get_key(1,128,&choice);
      outportb(CON_90,0);                  /* release reset */
```

```
        printf("\n\t\t\t3090 internal reset released and req set low");
        printf("\n\t\t\tHit any key to release 3042 reset");
        get_key(1,128,&choice);
        outportb(CON_42,0);
        printf("\n\t\t\t3042 internal reset released");
        printf("\n\n\t\t\tBoth Xilinx chips reset\n\n\t\t\t    Hit any key..");
        get_key(1,128,&junk);
    }

/**********************************************************************
*
*   WRITE_MEM
*
*   THIS ROUTINE WRITES A FILL PATTERN TO THE LOCAL JB1
*   MEMORY BANKS.  NOTE FOR FILL VALUE THAT WHITE IS
*   REPRESENTED BY 0 AND BLACK BY 255.  DONE TO REMAIN
*   COMPATIBLE WITH THE AD 9502 WHICH INVERTS THE IMAGE
*   COLORS.
*
**********************************************************************/

void write_mem(int *img_size)
    {
        int     choice2,
                done,
                err_cd,
                fill_val,
                i,
                pattern,
                pix_count,
                status,
                status1,
                status2;
        long int    count,
                    lcount;

        clrscr();
        printf("\n\t\t\t   MEMORY WRITE");
        printf("\n\n\t\t\tSelect fill value:\n");
        printf("\n\t\t\t1. white (0)\n\t\t\t2. black (255)");
        printf("\n\t\t\t3. checkerboard");
        get_key(49,51,&choice2);
        switch (choice2)
```

```
    {
    case 49: fill_val =0;
        pattern = 0  ;                          /* pattern = solid white*/
        break;
    case 50: fill_val=255;
        pattern = 0;                            /* pattern = solid black*/
        break;
    case 51: fill_val=255;
        pattern = 1;                            /* pattern = checkerboard*/
         break;
    }
outportb(CON_90,6);                             /* image request for bank1 */
printf("\n\n\t\t\tImage transfer requested\n\t\t\tPolling for");
printf(" Img_Xfr flag");
status2=0;
err_cd = 0;
lcount =0;
while (status2==0 & err_cd==0)
    {
    ++ lcount;
    if (lcount==25000) err_cd=1;
    status = inportb(INREG);
    status1= status &32;
    status2= status&2;
    }
if (status1 ==32) printf ("\n\t\t\tImg_Rdy flag active");
else printf("\n\t\t\tImg_Rdy flag inactive");
if (status2 ==2) printf ("\n\t\t\tImg_Xfr flag active");
else printf("\n\t\t\tImg_Xfr flag inactive");
if (err_cd!=0) printf("\n\t\t\tTime out error on polling");
if (status1==0 & status2==2)
    {
    printf("\n\t\t\tTransfer ready to proceed");
    outportb(CON_90,0);
    printf("\n\t\t\tDirection set for writing.");
    printf("\n\t\t\tImg_req set inactive\n");
    done=2;
    count = 0;
    pix_count=0;
    i=0;
    printf("\n\t\t\tData Transfer in progress");
    while(done!=0)
        {
```

```
            status = inportb(INREG);
            done =status&2;
            if (done!=0)
                {
                ++count;
                ++pix_count;
                if (pattern==1)
                    {
                    ++i;
                    if ((i%32)==0) fill_val =fill_val^255;
                    if (i==*img_size*32)
                        {
                        fill_val=fill_val^255;
                        i=0;
                        }
                    }
                    outportb(DATA_PORT,fill_val);
                }
            }
        printf("\n\n\t\t\t%li pieces of data transferred\n",count);
        }
    printf("\n\t\t\tHit any key");
    get_key(1,128,&choice2);
  }

/*******************************************************************
*
*  READ_MEM
*
*  THIS ROUTINE READS A BANK OF MEMORY FROM JB1 AND STORES
*  THE DATA IN FILE TEMP.RAW.
*
*******************************************************************/

void read_mem(void)
  {
    FILE     *stream;
    int      choice,
             choice2,
             done,
             err_cd,
             reg,
             status,
```

```
        status1,
        status2,
        xcoor,
        ycoor,
        val_mem;
long int   count,
           lcount;
char   junk;

clrscr();
stream=fopen("temp.raw","wb");
printf("\n\t\t\t   MEMORY READ");
printf("\n\n\t\t\tSelect bank:\n\n\t\t\t0) Bank_0  1) Bank_1");
get_key(48,49,&choice);
if (choice==48) reg = 2;                    /* bank 0 */
else  reg=6;                                /* bank 1 */
printf("\n\n\t\t\tDumping data to temp.raw\n");
outportb(CON_90,reg);                       /* image request */
printf("\n\n\t\t\tImage transfer requested\n\t\t\tPolling for");
printf(" Img_Xfr flag");
status2=0;
err_cd = 0;
lcount =0;
while (status2==0 & err_cd==0)
    {
    ++ lcount;
    if (lcount==25000) err_cd=1;
    status = inportb(INREG);
    status1= status &32;
    status2= status&2;
    }
if (status1 ==32) printf ("\n\t\t\tImg_Rdy flag active");
else printf("\n\t\t\tImg_Rdy flag inactive");
if (status2 ==2) printf ("\n\t\t\tImg_Xfr flag active");
else printf("\n\t\t\tImg_Xfr flag inactive");
if (err_cd!=0) printf("\n\t\t\tTime out error on polling");
if (status1==0 & status2==2)
    {
    printf("\n\t\t\tTransfer ready to proceed");
    outportb(CON_90,1);
    printf("\n\t\t\tDirection set for reading");
    printf("\n\t\t\tImg_req set inactive\n");
    done=2;
```

```
            count = 0;
            printf("\n\t\t\tData Transfer in progress");
            while(done!=0)
                {
                status = inportb(INREG);
                done =status&2;
                if ((done!=0) && (count<256000))
                    {
                    ++count;
                    junk = inportb(DATA_PORT)^255;   /* complement value! */
                    fputc(junk,stream);
                    }
                if(count==256000)
                    {
                    printf("\n\n\t\t\tError: DONE not reached");
                    break;
                    }
                }
            fclose(stream);
            printf("\n\n\t\t\t%li pieces of data transferred\n",count);
            }
        printf("\n\t\t\tHit any key");
        get_key(1,128,&choice2);
    }

/*******************************************************************
*
*  GET_KEY
*
*  THIS ROUTINE GETS A KEYSTROKE FROM KEYBOARD.
*
*******************************************************************/

void get_key(int low, int high, int *choice)
    {
        *choice = getch();
        while (*choice<low || *choice >high)
            {
            *choice = getch();
            }
    }
```

```
/*****************************************************************
*
*  FILE_OPEN
*
*  THIS ROUTINE DISPLAYS ALL RELEVANT DIRECTORY FILES AND
*  OPENS THE FILE SELECTED BY THE USER.
*
*****************************************************************/

void file_open(char ext[],FILE **stream)
   {
      char      file_list[20][16],
                file_name[16],
                mask[7];
      int       done,
                index=0,
                choice;
      struct  ffblk ffblk;

      strcpy(mask,"*.");
       strupr(ext);
      strncat(mask,ext,4);
      done = findfirst(mask,&ffblk,0);
      if (done==-1)
         printf("\n\n\t\t\tNO .%s FILES FOUND",ext);
      else
         {
         printf("\n\n\t\t\t    SELECT FILE:\n");
         while (!done)
            {
            ++index;
            strcpy(file_list[index],ffblk.ff_name);
            printf("\n\t\t\t   %i. %s",index,ffblk.ff_name);
            done = findnext(&ffblk);
            }
         get_key(49,index+48,&choice);
         choice =choice-48;
         strcpy(file_name,file_list[choice]);
         printf("\n\n\t\t\t    Opening file %s\n",file_name);
         *stream=fopen(file_name,"rb");
         if (*stream==NULL) printf("\n\t\t\t   FILE ERROR!");
         }
   }
```

Vita

Jeffrey Nevits was born in New Hartford, New York on June 27, 1969. After attending high school in his hometown, he enrolled at Virginia Polytechnic Institute and State University in the fall of 1987. In the spring of 1991, Jeffrey received a Bachelor of Science degree from the Bradley Department of Electrical Engineering at Virginia Tech. He continued his studies at Virginia Tech by enrolling in the graduate program. At the start of 1994, Jeff began working as an electronics engineer with GE Fanuc Automation, Inc., in Charlottesville, Virginia while he continued working on his degree. Mr. Nevits should receive his MSEE in the summer of 1996.

Jeffrey A. Nevits