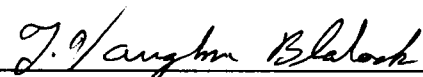


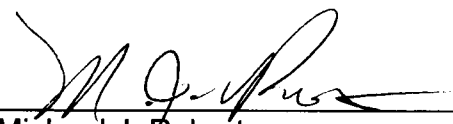
To the Graduate Council:

I am submitting herewith a thesis written by Stefan Weidmann "Fast Prototyping on Personal Computers." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.


Donald W. Bouldin, Major Professor

We have read this thesis
And recommend its acceptance:


T. Vaughn Blalock


Michael J. Roberts

Accepted for the Council


Associate Vice Chancellor and
Dean of The Graduate School

Fast Prototyping on Personal Computers

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Stefan Weidmann

May 1998

To Tatjana

- the major co-conspirator in my life...

ACKNOWLEDGMENTS

I would like to thank the members of my graduate committee at the University of Tennessee, Knoxville for their support and encouragement – especially Donald W. Bouldin. I am also grateful to Zellweger Uster for putting many important tools at my disposal.

ABSTRACT

Personal computers (PC) represent an attractive platform for many applications in the field of prototyping and commercial product development. They are readily available, inexpensive, and they offer a broad variety of software applications. These applications include very powerful development tools for software, printed circuit boards (PCB), and field programmable gate arrays (FPGA).

Numerous industrial applications require measurement of a quantity or communication with other units. In many cases custom designed acquisition boards or interface cards have to be developed. A big portion of such designs is repeated in the same or a similar form with every new implementation. For example, bus interfaces, software drivers, or communication mechanisms are often needed. With a modular design these parts can easily be extracted and reused in the next design.

Using this approach, an image acquisition system was entirely designed, simulated, implemented, and tested using PC based tools for the project presented here. For maximum flexibility, an FPGA was used to implement the logic. The FPGA code is downloaded from the PC in order to be able to easily change and update the logic in the final product.

The results achieved by this development are the image acquisition board on one side and a collection of modules on the other. These reusable units, such as the interface logic, the register bank in the FPGA, the download software, and the PC-side drivers, will speed up the next design. Moreover, the board itself has already been reused with a different FPGA code as an interface with a programmable power supply.

Contents

1 INTRODUCTION.....	1
1.1 MOTIVATION	1
1.1.1 Modularity and Reusability	2
1.1.2 Project Flow	2
1.2 APPLICATION EXAMPLE	4
1.2.1 Overview	4
1.2.2 Design Flow	6
2 DESIGN ENVIRONMENT	8
2.1 PLATFORM.....	8
2.2 TOOLS.....	9
2.2.1 Internet Information Retrieval.....	9
2.2.2 Schematic Capture and Circuit Board Editor.....	9
2.2.3 Protel AutoRoute.....	10
2.2.4 FPGA Design-Entry Tool.....	11
2.2.5 FPGA Core Tool	13
2.2.6 C Compiler.....	14
2.2.7 Documentation.....	14
3 SPECIFICATIONS	15
3.1 IMAGE ACQUISITION SYSTEM BLOCK DIAGRAM.....	16
3.2 ISA BUS INTERFACE FUNDAMENTALS.....	18
3.2.1 8-bit I/O Access.....	18
3.2.2 16-bit I/O Access.....	20
3.3 FPGA DOWNLOAD.....	21
3.4 SYNCHRONOUS DESIGN	22
4 IMPLEMENTATION	24
4.1 SCHEMATIC	24
4.2 PRINTED CIRCUIT BOARD (PCB).....	25
4.2.1 PCB Layout.....	25
4.2.2 Place and Route	27
4.3 INTERFACE PLD.....	28
4.3.1 Simulation	28
4.4 ANALOG INPUT.....	29
4.5 RESTORATION OF THE DC LEVEL.....	29
4.6 PHASE LOCKED LOOP (PLL).....	31
4.7 FPGA DESIGN.....	33
4.7.1 Overview.....	33
4.7.2 Host Interface.....	35
4.7.3 Serial Peripheral Interface (SPI).....	36
4.7.4 Acquisition Subsystem	39
4.7.5 Flash Trigger.....	43

4.7.6 Phase Locked Loop (PLL).....	45
4.8 TIMING ANALYSIS	46
4.9 SOFTWARE.....	52
4.9.1 Drivers	52
4.9.2 Download Tips & Tricks	52
4.9.3 TIFF Format.....	53
5 RESULTS	56
6 SUMMARY AND CONCLUSION	62
7 OUTLOOK.....	63
7.1 REUSABLE MODULES	63
REFERENCES.....	65
APPENDICES.....	67
APPENDIX A: SCHEMATICS	68
APPENDIX B: PLD LISTING.....	71
APPENDIX C: VHDL LISTINGS.....	73
ROOT	73
HOST INTERFACE	78
FLASH TRIGGER.....	84
ACQUISITION SUBSYSTEM.....	86
SERIAL PERIPHERAL INTERFACE (SPI)	92
PHASE LOCKED LOOP (PLL)	93
APPENDIX D: C LISTING	95
APPENDIX E: SIMULATION COMMAND FILES	104
VITA.....	108

List of Figures

FIGURE 1: PROJECT FLOW	3
FIGURE 2: APPLICATION BLOCK DIAGRAM.....	5
FIGURE 3: DESIGN FLOW	7
FIGURE 4: IMAGE ACQUISITION SYSTEM BLOCK DIAGRAM.....	17
FIGURE 5: ISA 8 BIT I/O ACCESS	19
FIGURE 6: ISA 16 BIT I/O ACCESS	21
FIGURE 7: SYNCHRONIZED EDGE DETECTOR.....	23
FIGURE 8: THE PCB LAYOUT.....	26
FIGURE 9: DECODING PLD SIMULATION	29
FIGURE 10: LOW PASS FILTER CHARACTERISTIC	30
FIGURE 11: DC RESTORATION	31
FIGURE 12: PHASE LOCKED LOOP PRINCIPLE.....	32
FIGURE 13: PLL PHASE COMPARATOR	33
FIGURE 14: DESIGN HIERARCHY	34
FIGURE 15: SPI FINITE STATE MACHINE	37
FIGURE 16: SIMULATION OF THE SPI.....	38
FIGURE 17: ACQUISITION CONTROLLER FINITE STATE MACHINE	40
FIGURE 18: SYNCHRONIZED CLOCK MULTIPLEXER.....	42
FIGURE 19: ACQUISITION SIMULATION.....	43
FIGURE 20: FLASH TRIGGER FINITE STATE MACHINE	44
FIGURE 21: PLL FINITE STATE MACHINE.....	45
FIGURE 22: SIMULATION OF THE PLL	46
FIGURE 23: LUT WRITE ACCESS PRE-LAYOUT SIMULATION.....	50
FIGURE 24: LUT WRITE ACCESS POST-LAYOUT SIMULATION	51
FIGURE 25: LUT WRITE ACCESS TIMING ANALYSIS	51
FIGURE 26: FIRST PICTURE	57
FIGURE 27: POSITIVE IMAGE.....	58
FIGURE 28: NEGATIVE IMAGE (INVERTED LUT)	59
FIGURE 29: 'ZIGZAG' EFFECT	60
FIGURE 30: PICTURE OF THE IMAGE ACQUISITION BOARD.....	61

List of Abbreviations

AC	Alternating Current
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DC	Direct Current
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
LUT	Look Up Table
MMI	Man Machine Interface
OTP	One Time Programmable
P&R	Place and Route
PC	Personal Computer
PCB	Printed Circuit Board
PLD	Programmable Logic Device
PLL	Phase Locked Loop
RAM	Random Access Memory
ROM	Read Only Memory
SPI	Serial Peripheral Interface
TIFF	Tagged Image File Format
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XNF	Xilinx Netlist Format

1 INTRODUCTION

1.1 MOTIVATION

The task of developing new products in today's industry, and not only in the electronics industry, has changed dramatically over the past years or decades. On one hand product life cycles and therefore development times become shorter and shorter; on the other hand new technologies offer higher performance for lower cost like never before. Managing these faster changes and finding a way through the jungle of information are becoming the major challenges of new product development.

Personal computers meanwhile offer a very attractive and competitive platform for many development tasks. They offer performance comparable to work stations at a fraction of the price. The availability of cheap processing power along with more powerful software make PCs an affordable tool even for small companies.

The availability of the whole tool chain, from information retrieval, to development, to documentation under the same operating system with a standardized man machine interface (MMI), makes it much easier for one person to operate different tools. It took different tool specialists for drawing PCB layouts, developing FPGAs or software a couple of years ago. All these tasks can be performed by the development team or even by a single person now, eliminating a lot of time-consuming and error-susceptible overhead. The proximity to perfection of every step might be smaller this way but, especially in the prototype stage, a few gates more or an additional layer on a board do not hurt. Moreover the concentration of a wider span of problems on one person or team leads to a more system-oriented solution. The focus can be shifted from tool-specific knowledge to application-specific knowledge, making the job of the developer more interesting.

1.1.1 Modularity and Reusability

An aspect of growing importance is reusability. As product and design cycles become shorter and shorter it is more and more important to be able to quickly assemble new products from a stock of known and reliable components. It is not desirable and not necessary to reinvent the wheel every time.

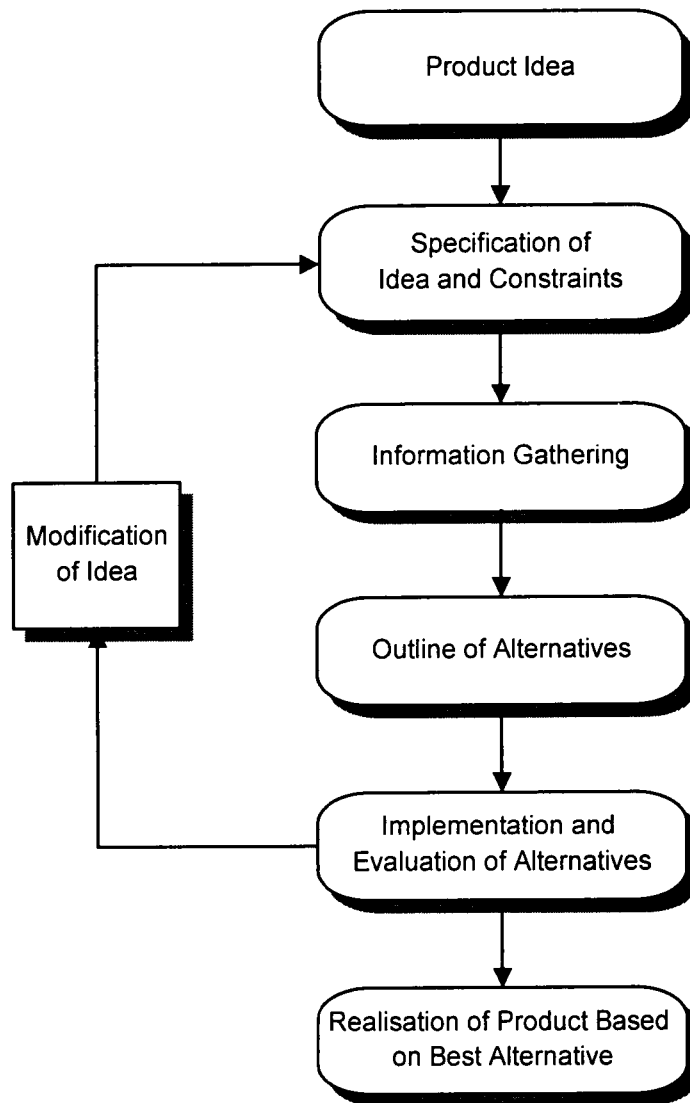
In order to reuse parts of a development, the single parts must be well defined and differentiated from each other. This is best achieved by building modules, which can be extracted from one design and then recombined with new modules to form a new system.

Especially when building prototypes the emphasis is on a quick development time rather than optimized cost. Therefore prototyping is especially suitable for a modular approach.

1.1.2 Project Flow

A typical project flow is shown in Figure 1. Several steps have to be taken, from the idea to the sale of a product. The major issue of developing is to decide what the best solution for a new product is. Therefore we have to create ideas, validate, implement, evaluate, and modify them until we have an appropriate solution.

The more accurate our information is, the better we can make the right decision. It is known from experience that the sooner a weak point is detected and eliminated, the less its correction costs. It is a matter of hours or days to modify the specifications of a new product if a weak point is detected. But it is a major effort to correct a product inadequacy if there are already several thousands units shipped and have to be called back. This scenario may lead to major crises or even bankruptcy.

**Figure 1: Project Flow**

This is the point where fast prototyping can be used to gather a lot of valuable information of a new application in a stage where changes can easily be made and the associated costs and impacts are low.

1.2 APPLICATION EXAMPLE

1.2.1 Overview

The application for this fast prototyping procedure consists of an image acquiring system (frame grabber). The new product is an online quality control system based on image processing. Since the final product is supposed to be fabricated in large numbers and therefore a low price is important, the development of 'homemade' hardware consisting of an acquisition system and a processor is desired.

The plan was to evaluate the basic image-processing algorithm on a PC with an off-the-shelf frame grabber board. The prototype was designed to be able to operate in a PC as well as in conjunction with a controller evaluation board.

The prototype and the final product will be operated with off-the-shelf cameras. The initial idea to build one's own camera was given up very quickly since cameras are mass produced and therefore relatively cheap and their quality better than their reputation.

Figure 2 shows the system setup for the prototype. The key features of the board are:

- Two host interfaces: PC and controller board
- Camera interface
- Trigger output for a flash
- 256 Kbytes frame memory
- 16-bit-wide memory access

Prototype Block Diagram

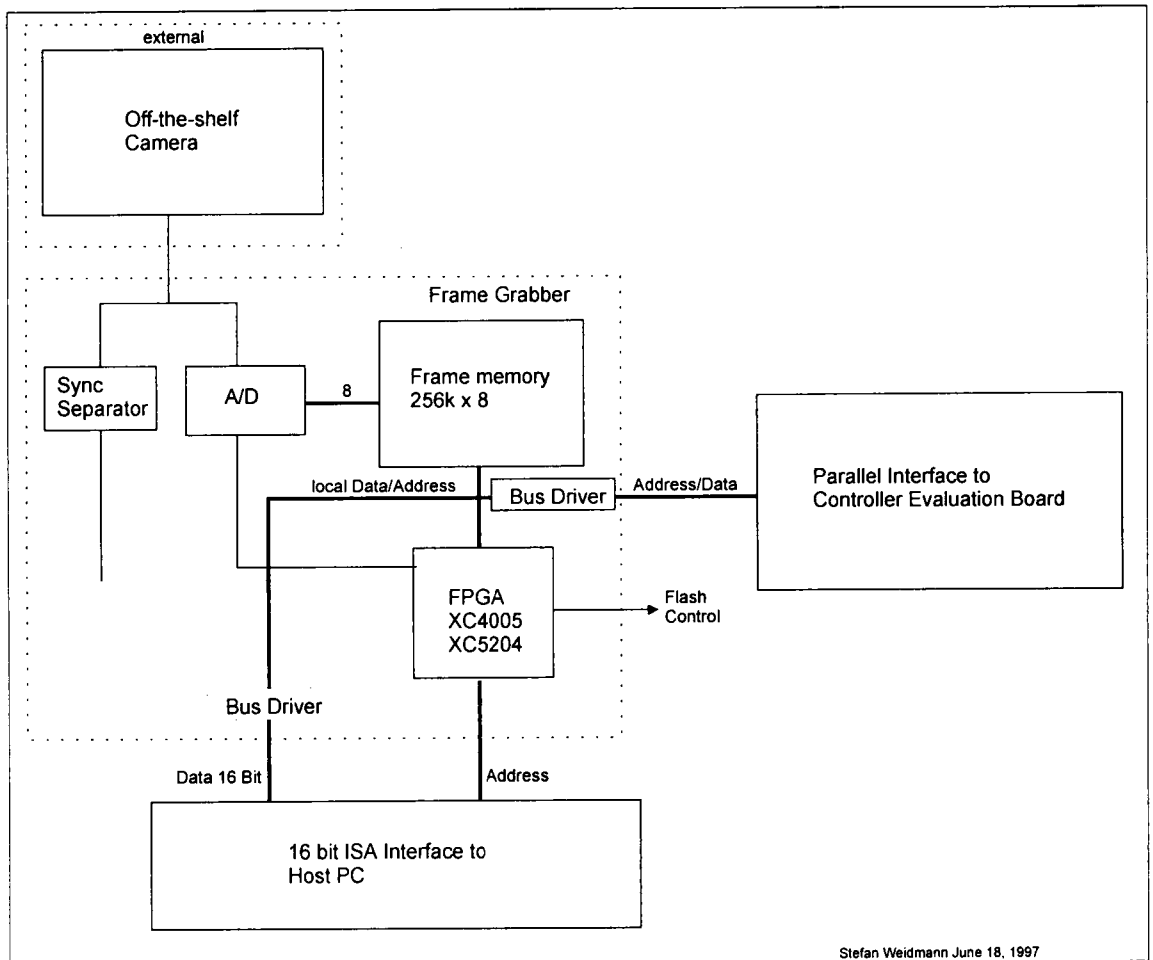


Figure 2: Application Block Diagram

1.2.2 Design Flow

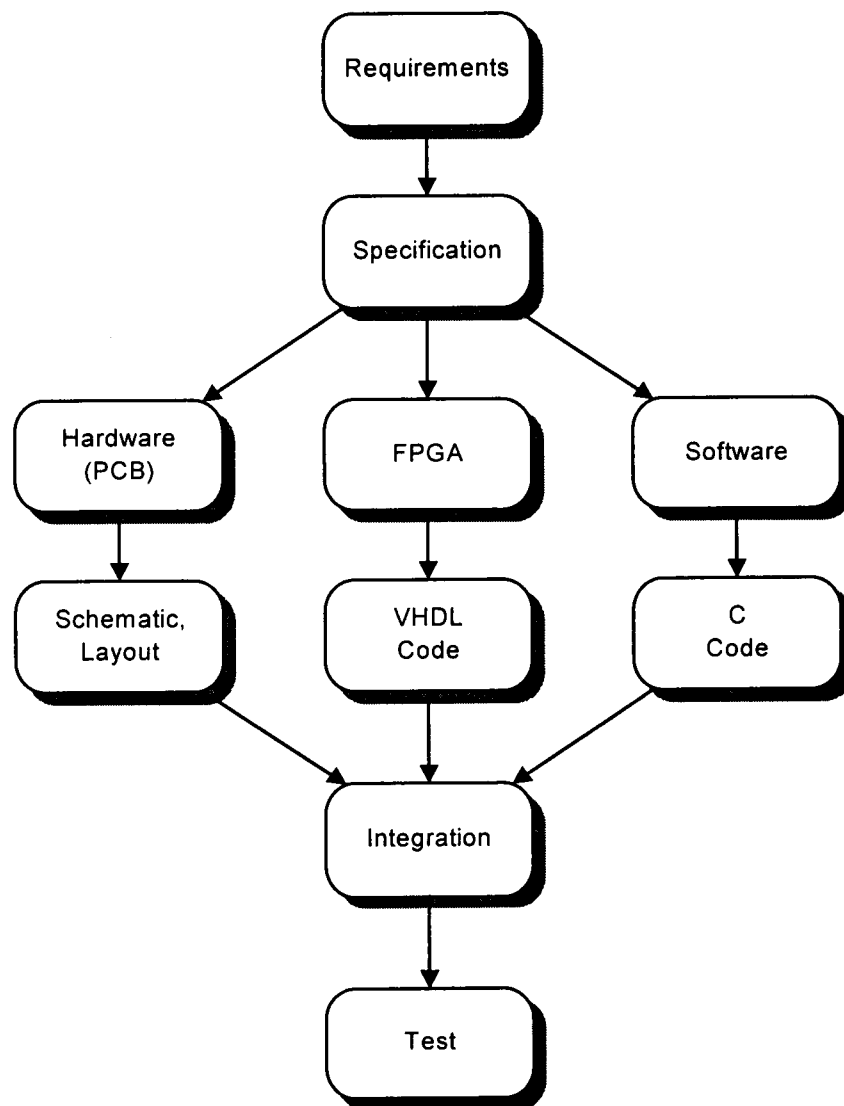
Figure 3 shows the design flow for this prototype. The first two and the last two steps can basically be applied for any design work. The middle part, the implementation, is design specific. For this design the design step consists of the following tasks:

- Drawing schematic and layout
- Writing and testing the FPGA code
- Writing and testing the host C code

These tasks can be performed more or less independently from each other whereas the first two and the last two steps have a strong interdependence. If a whole team is involved, the implementation step can be split and performed in parallel. In my case a single person was in charge of all tasks so the tasks had to be performed serially. But even in this case it was possible to take advantage of the parallelization of certain tasks. After the PCB layout was finished I had to order the boards. Since I knew I needed time to implement the VHDL and C code, I could give the PCB manufacturer more time and thus obtain a much better price compared to a super-fast prototype service.

The integration phase is very critical and important. If a team is involved all team members are required to participate. In this stage it usually becomes apparent how thorough the previous steps were performed. Any omission from earlier steps becomes obvious.

A lot of the risk of the integration phase can be taken away if known and already-tested modules are used. The case where two established and known modules are combined with a new one is much easier to handle than the case of having three unknowns. It was sometimes difficult in this project to locate an error since all parts were new and untested. There is no alternative to going back one step and reviewing and retesting all involved components in such a case.

**Figure 3: Design Flow**

2 DESIGN ENVIRONMENT

Tools are not the solution for every problem but they are crucial in the design cycle. The best ideas are worth little if they can not be implemented, and the latest and most expensive tools do not help if their use is so complicated that a normal person can not handle them.

A consistent tool chain under the same graphical environment helps in many ways to speed up the design cycle. One point is the data interchange, which is easier and more efficient if all programs run under the same environment. Another aspect is handling the tools. The learning process for a new program is much quicker if the user is familiar with the operating philosophy and the program follows the same style guide of other programs. Once you know how to handle a word processor it is fairly easy to become familiar with another one.

2.1 PLATFORM

The platform used for the implementation of the whole project is an IBM PC compatible running under Windows '95. The equipment is more or less main stream and is what you can find in every computer store in the lower or middle price segment. The only thing worth spending a little more money on is the monitor. Since the applications are all graphics oriented, and especially the PCB layout requires detailed graphical work, a screen resolution of at least 768 by 1024 is required; more can not do any harm. The refresh rate should be 72Hz non-interlaced or higher in order to allow longer work periods.

2.2 TOOLS

This chapter gives an overview of the tool chain used in this project. It is not my intention to give a complete description of all programs involved. For detailed information please refer to the corresponding user's guide. There are certainly more utilities involved, especially editors, that are not listed explicitly.

2.2.1 Internet Information Retrieval

A very useful source for information retrieval is the Internet. Currently all major companies have WEB sites with the latest data sheets available. Getting data books can sometimes take weeks, while downloading from the WEB site takes minutes, and very often the electronic data sheets are newer than the books.

I think the times when the Internet was considered a toy or a pastime are definitely over. No company, which is seriously interested in having an up-to-date development group, can afford not to have access to 'The Net.'

2.2.2 Schematic Capture and Circuit Board Editor

The software manufacturer Protel provides several tools under the same client/server environment:

- Schematic editor
- Schematic library editor
- PCB editor
- PCB library editor
- Text editor

- PLD compiler
- PLD simulator

The shell offers some common features like setup, design hierarchy, etc. But the individual tools are used like stand-alone programs. The schematic and PCB tools are closely related, whereas the PLD compiler and simulator form a group themselves. The PLD tool does not support schematic entry. The input is being done in a high level description language, which is fairly easy to handle, especially since the target devices are small. Included in the PLD package is also a functional simulator with a graphical front end.

The whole package makes a very sophisticated impression and runs without major problems. One of the reasons for this is the fact that Protel migrated from DOS to Windows a long time ago and therefore had the time to 'learn the lesson.'

A feature of the PCB layout software worth mentioning is the 'Board wizard.' This tool generates boards for various standardized computer buses such as ISA, PCMCIA, PCI, SUN, VME and others.

2.2.3 Protel AutoRoute

The Protel PCB base package contains a rip-up and retry router. The router is fairly simple and only capable of automatically routing small designs.

But Protel also offers a shape based auto router as a stand-alone tool. The results of this tool are impressive. The router is capable of routing complicated boards with a high rate of success and good results.

Since it is a separate product (originally from a different company), the interface to the

client/server tools is not as consistent as the interfaces between them. But after figuring out the right configuration the tool does a very good job.

2.2.4 FPGA Design-Entry Tool

The FPGA tool selected is the one provided by XILINX, the FPGA manufacturer. There are different tools available under the same environment, similar to the Protel client/server. The whole package, called 'Foundation Series,' is divided into two major blocks:

- Design-entry tool
- Place and Route (P&R) tool

The design-entry tool is provided by the XILINX alliance partner ALDEC. This package is comparable to the WORKVIEW package we currently use at the University of Tennessee in Knoxville. It contains the following items:

- Schematic entry tool
- VHDL and ABEL editor and compiler
- Finite state machine (FSM) editor
- Functional simulation
- Timing (post-layout) simulation

ALDEC also offers the same environment with interfaces to other FPGA vendors, like ACTEL or LUCENT. This gives the end user a certain degree of independence regarding the choice of the FPGA manufacturer.

The P&R tool set comes from XILINX. It is actually the same set XILINX provides for

years. Most programs are integrated in a Windows shell to configure, combine, and start them.

2.2.4.1 VHDL Editor and Compiler

The VHDL editor is a very handy tool. It combines a language sensitive editor with online help, compiler launch capability, and a syntax error display in the source code.

Language sensitivity means that the editor distinguishes between reserved key words, user defined names, and comments. Accordingly, they are displayed in different colors.

The online help is a very quick way to find specific VHDL statements and syntax rules, especially for less experienced VHDL users.

The compiler allows different user defined setup options, such as optimization effort, optimization for place, speed, or both. The compiler can also detect special functions, like counters, adders, multiplexers, etc. and integrate pre-defined and optimized soft-macros. XILINX calls these macros XBLOX.

It is very important to be able to control these options. Optimization benefit and used time are trade-offs. For prototyping it is desirable to have a quick turn-around time at the expense of a larger chip, whereas the final product may be cheaper due to an optimized and therefore smaller logic.

2.2.4.2 Finite State Machine (FSM) Editor

State machines ('bubble diagrams') are a very powerful way of implementing control structures. State machines can be carried out with every design tool, be it schematic or behavioral language, but the graphical entry allows the visualization of the structure very well.

It is very useful to know how to implement a state machine in VHDL in order to review

the resulting code from the FSM editor. This way one can check if the generated code meets the expectations. After playing around with the tool for a couple of hours and checking out the effects of different settings, the FSM editor becomes a very powerful instrument.

Several modules in this project, from simple to complex, were realized with the FSM editor.

2.2.4.3 Simulator

The included simulator is capable of simulating both functional and timing behavior. For the latter it is necessary to back-annotate the netlist with the actual delay information. The simulator can not process the VHDL code. It converts the compiled XNF netlist to its own format in order to perform a simulation.

The stimuli are defined in a command file. The commands form a subset of the workview command set. This is a very popular format and makes data interchange easy.

2.2.5 FPGA Core Tool

The FPGA tools of the Foundation development kit are the same as the ones used under the UNIX or DOS operating system. The integration in a common shell makes the use more comfortable. But certain operations, such as back-annotation, still have to be executed from the command line interpreter (DOS shell).

2.2.5.1 Place and Route

The place and route software is executed from within the shell. This allows the definition of project specific parameters, such as constraints file, part type, speed grade, etc.

2.2.5.2 Timing Analyzer

The timing analyzer is a very useful tool to verify the timing behavior of the routed design. It offers advantages over the post-layout simulation specifically for synchronous designs. Its ability to trace all nets and find the worst case delays in a short time makes it easy to check if a design meets the required clock speed requirements.

For the asynchronous parts of a design, such as decoding logic, the post-layout simulation is the more practical approach. Both methods complement one another.

2.2.6 C Compiler

The compiler used for this project is an older version of Borland's C++ compiler running under DOS. Although it is not the latest in programming technology, it meets all the needs of this project. While working under DOS the programmed driver routines can be kept small and easy to debug. The various modules, such as download routine, or register configuration, can be integrated in more sophisticated Windows applications later.

2.2.7 Documentation

A new development might be an ingenious thing. But all its advantages are worth nothing if nobody knows how to apply it. This is where documentation comes into play. The easier it is to make documentation, the more likely it is that somebody does it.

One of the great advantages of having the whole tool chain under Windows is the ability to extract information from the various tools and to import it to a word processor. There are many ways to transfer outputs from one program to another. Some common ways are making screen shots, copying to the clipboard, and printing to a postscript file.

3 SPECIFICATIONS

Prototypes are made to show the feasibility of a principle and to evaluate alternative ways of implementing certain functions. The requirements for a prototype are therefore different from the ones of a product. The prototype requirements contain many subjective terms, such as 'as fast as possible,' 'biggest for a reasonable price,' or uncertain specifications like 'if possible.' Product specifications on the other hand must contain measurable quantities, such as '10 MHz clock speed' or 'total cost of less than \$50 in quantities of 1000 pieces'. They are based on the evaluation made with the prototype.

The specifications for this prototype can be summarized as follows:

- Two host interfaces: PC and controller board
- Firmware download from host
- Camera interface for standard black and white camera
- 8 bit resolution
- Trigger output for a flash
- Full-frame memory (256 Kbytes)
- 16-bit-wide memory access
- Low cost
- As much flexibility as possible

The last point is a typical prototype requirement. In this stage of a development, the engineer can use his imagination and implement some helpful add-ons. In our example the flexibility consists in the following:

- Three possible ways of configuring the FPGA (host download, serial PROM, and download cable)

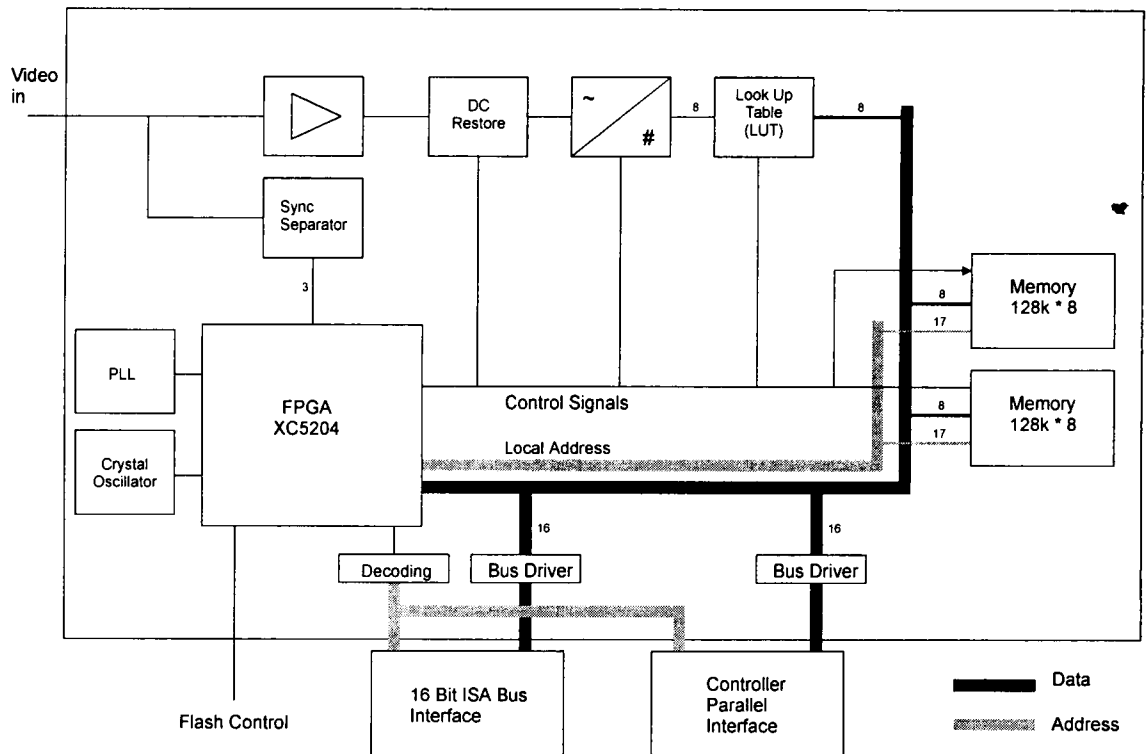
- Three different clock sources for the acquisition (PLL, ISA bus clock, and clock oscillator)
- Picture size configurable
- Horizontal and vertical offset adjustable
- Flash start on odd or even frame
- Flash pulse delay adjustable
- Flash pulse length adjustable
- Programmable PLL frequency
- Look up table (LUT)
- Spare FPGA pins for expansions
- Spare FPGA gates for expansions

The flexibility proved to be very important. The spare pins and FPGA gates were used to implement an additional serial peripheral interface (SPI). The configuration capability via download cable was used to test the circuit when the host download was not working yet. The different clock sources could be used to determine the trade-off between picture quality and cost.

3.1 IMAGE ACQUISITION SYSTEM BLOCK DIAGRAM

Figure 4 shows the block diagram of the image acquisition board. The heart of the system is a Xilinx FPGA. The actual implementation uses a XC5204 type. This circuit is very cost effective, but only average in speed. Because of the (near) pin compatibility of the various families, a larger XC5200 device or a more expensive but faster XC4000 family member can also be used.

Frame Grabber Overview



Stefan Weidmann October 06, 1997

Figure 4: Image Acquisition System Block Diagram

The video signal runs through an amplifier and a DC-restoration stage in order to match the analog to digital converter (ADC) input range. The details of the implementation are described in the implementation section.

3.2 ISA BUS INTERFACE FUNDAMENTALS

The ISA bus is a widely used bus. Nevertheless it involves certain dangers. Unlike other bus systems, ISA was never clearly specified. It is an evolved de facto standard rather than an industry norm (like the VME bus for instance).

For the design of a data acquisition card we can focus on the simplest access type: the I/O-access. This access type is relatively slow but easy to implement. I/O-peripherals can be accessed with either 8 or 16 bit bus width.

3.2.1 8-bit I/O Access

The signals involved in an I/O access are the address lines, the address enable (AEN) signal, I/O read, I/O write, and the data lines.

For an I/O access address lines A0 to A9 have to be considered only. The remaining address signals are used to generate the I/O read and write signals in the ISA bus logic. AEN can be regarded as an active-high address latch enable or an active-low address valid signal. A low state indicates valid address, read, and write signals.

Figure 5 shows the situation of an 8-bit access. All signals are related to the ISA bus clock. The clock frequency is nominally 8 MHz; but it can typically vary between 7 MHz and 12 MHz depending on the chip set and the CMOS setup. All bus signals are related to the bus clock, but the *setup and hold times are not specified*.

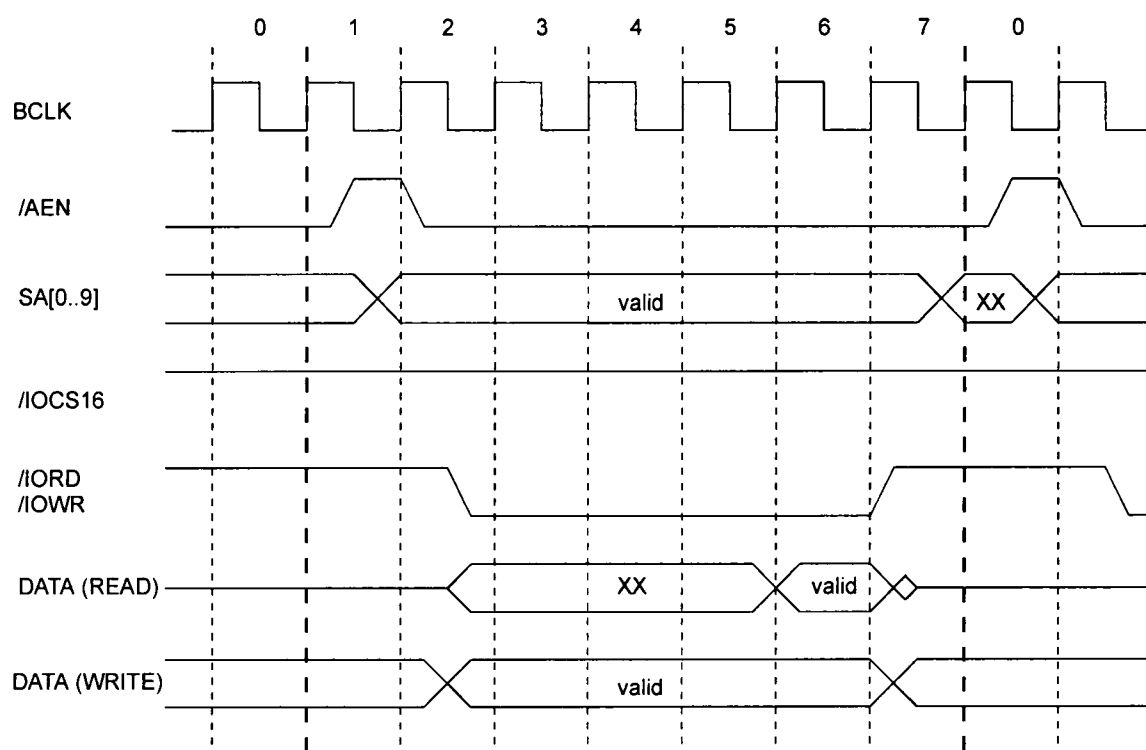


Figure 5: ISA 8 Bit I/O Access

The following list contains some important relations between the individual signals and the clock. The clock cycles are numbered and start with a rising edge:

- A0 to A9 become valid at the latest with the rising clock edge of cycle 2
- IORD and IOWR become valid with the falling clock edge of cycle 2
- In the case of a write access, data become valid before IOWR becomes valid and remain valid until after IOWR becomes invalid
- In the case of a read access, the data lines *can* be driven as soon as IORD is valid and until the falling clock edge of clock cycle 7. The data *must* be valid at least one clock cycle before IORD becomes invalid and data must remain valid until after IORD becomes invalid

- IORD and IOWR become invalid with the rising clock edge of cycle 7
- A0 to A9 remain valid at least until the falling clock edge of cycle 7
- Clock cycle 0 is called I/O recovery cycle. Most CMOS-setups allow the setting of a user-selectable number of recovery cycles.

3.2.2 16-bit I/O Access

The 16-bit I/O access is based on the same principle as the 8-bit access and it can be considered a wider and condensed version with cycles 3 to 6 skipped.

Figure 6 shows the access sequence as it is used in this project. The I/O card indicates its capability of handling 16-bit-wide data with the assertion of the IOCS16 signal. This signal is shared by all ISA bus cards and must therefore be tri-stated when not activated. The signal can be asserted as long as the address lines are valid (dashed line) or as long as the address and read/write signals are active (solid line). If the IOCS16 signal is not asserted, although the host tries to access 16 bits, the ISA bus logic divides the access into two 8-bit accesses as described in the previous chapter. An assertion of the IOCS16 signal during an 8-bit access is ignored.

The 16-bit access implemented for this project requires the software to always access the 16-bit wide registers word-wide and is therefore not fully dynamic. The fully dynamic implementation requires an additional signal and allows access to 16-bit registers either byte or word-wide. A detailed description of this mechanism and other useful information about the ISA bus can be found in Solari's book *ISA & EISA Theory and Operation* (Solari, 1993).

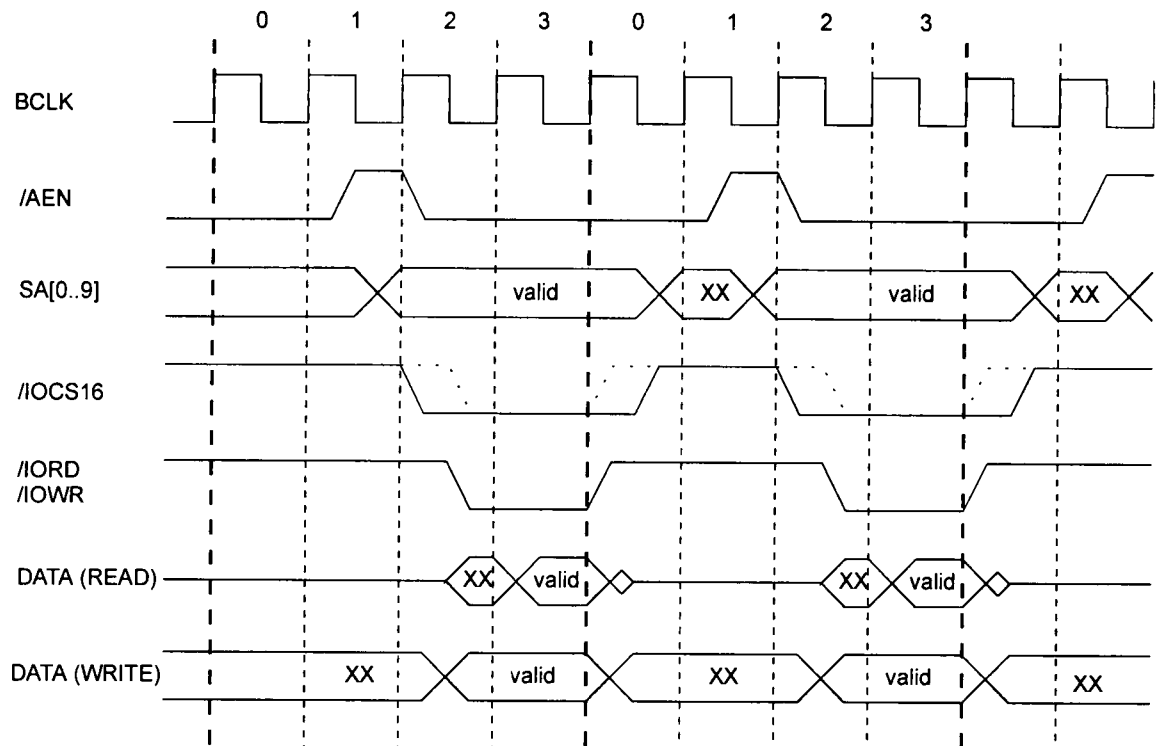


Figure 6: ISA 16 Bit I/O Access

3.3 FPGA DOWNLOAD

The programmable logic device used to implement the largest part of the functionality in this project is a XILINX FPGA. These devices are SRAM based and therefore lose their configuration after power down. The configuration data are usually stored in a non-volatile one-time programmable read only memory (OTP ROM). If the configuration has to be changed, a new ROM has to be programmed and plugged into the circuit board. Since the old ROM can not be reprogrammed, it has to be discarded. In the prototype phase this means a loss of time and money. More important is the case of a product installed at a customer's site. Changing a component needs the intervention of a service

technician in order to change the component. This can be very time consuming and expensive.

If the FPGA code can be downloaded from the host, a change of the configuration can be done by overwriting a file on the host's hard drive. This operation can easily be done by the customer. The needed code can be distributed electronically over a modem, thus avoiding shipping costs and delay.

All XILINX FPGAs offer a configuration mode, called asynchronous peripheral mode, designed for this purpose. Data can be written to the device through an 8-bit wide interface. Since this interface must be operational before the device is programmed the decoding logic must be implemented in a non-volatile logic device. The download mechanism even allows a 'handshake' in order to adjust the download speed.

The electrical and timing specifications are described in the Xilinx data book (Xilinx, 1996) on pages 4.199-4.201. The format of the configuration data stream is described on page 4.202 in the same book (Xilinx, 1996). The most recent data can be found on Xilinx' home page (Xilinx, 1997). The driver software and the decoding logic for the download are described in the implementation chapter.

3.4 SYNCHRONOUS DESIGN

I would like to point out the importance of a synchronous design. Implementing a logic design synchronously to a common clock does not simply have advantages; it is crucial in order to have control over the timing behavior of the system. A very common mistake is the asynchronous generation of clock signals for individual register banks. This leads to individually different clock skew times, which are very hard to control. Such designs very often show nasty behavior in the sense that they fail randomly in remote locations where the developer does not have access.

FPGAs are made for synchronous designs. They offer the entire infrastructure necessary to implement clock-synchronized logic, such as fast global clock networks and clock enable inputs on every flip-flop. Synchronous designs not only turn out to be more stable and predictable but also more compact, because they use the FPGA resources more efficiently.

Figure 7 shows a very-often-used mechanism to generate a synchronized edge detector of an asynchronous signal. The implementation consists of a master and a slave flip-flop. Their respective outputs are named 'Input (synch)' and 'Input (synch) delayed' in Figure 7. The output 'Synchronized edge detect' is derived combinatorially from the two flip-flops and is exactly one clock cycle in length. This technique is used to synchronize external trigger signals and to generate clock enable signals for all data registers in the host interface.

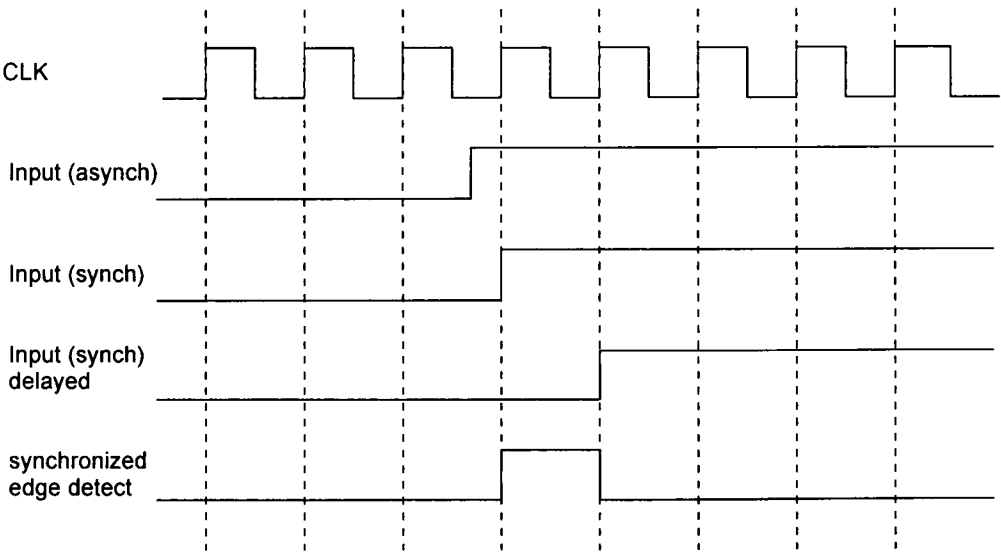


Figure 7: Synchronized Edge Detector

4 IMPLEMENTATION

After the definition of the requirements and the information retrieval the design was implemented. This process consisted of the following steps:

- Drawing a schematic
- Laying out the printed circuit board (PCB)
- Writing and simulating the code for the PLD and the FPGA
- Writing the code for the download and test software
- Assembling the PCBs
- Integrating and testing the PCB, logic code and the software code
- Repeatedly correcting bugs and retesting

4.1 SCHEMATIC

The schematic was drawn using the Protel design environment. The schematic is divided into two sheets. The first contains the interface to the hosts (PC and controller) which mainly consists of the connectors and then decoding PLD, the FPGA, and the acquisition frame memory. The second sheet contains the video-related circuitry, like analog input stage, synch separator, PLL, analog to digital converter (ADC), and the look up table (LUT).

The schematics attached in appendix A represent the state of the board layout. The following changes were made on the board manually:

- U4: Comparator 2 instead of comparator 1 is used. Therefore R16 is connected with pin 13 instead of pin 2.
- U6: In order to allow a LUT read back for test purposes, the output enable pin 1 was connected with a newly introduced FPGA output. The spare connector J6 was used

for this purpose.

- Spare connectors J5, J7, and J1 were used to implement the SPI interface

4.2 PRINTED CIRCUIT BOARD (PCB)

The main function of the PCB is to physically carry the electrically active components and to provide the interconnections between them. It very often adds unwanted and disturbing side effects. Therefore, the board layout is an important part of the circuit and the amount of work associated with this task is often underestimated.

A very common way to speed up the design of a prototype is to have another person carry out the design of the layout. This might save some of the engineer's time in the first place but it very often leads to unsatisfactory results because the drafting person does not have the same background knowledge as the engineer. In fact the time spent to generate the whole layout for this project is about the same as previously used to explain the requirements to the drafting person, review the finished artwork, and make corrections.

The approach selected for this project consists of the use of a user-friendly layout tool in conjunction with a well-performing autorouter. One of the time savers built into the PCB editor is the so-called 'Board wizard.' This tool automatically generates the mechanical outline of a bus board such as the ISA bus board.

4.2.1 PCB Layout

A screen shot of the layout tool containing the actual board layout is shown in Figure 8.

The outline drawing, the mounting holes for the bracket, and the edge connector were generated by the tool automatically. Some of the footprints were not in the library so

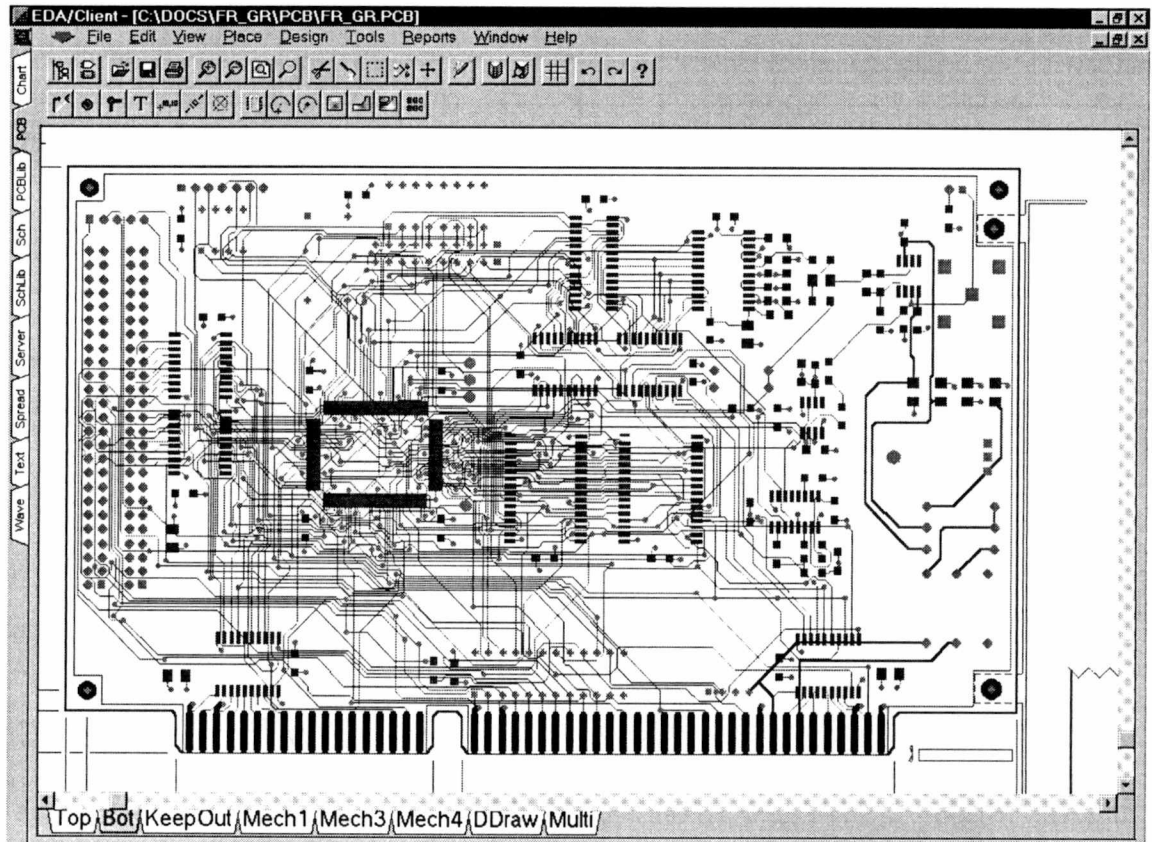


Figure 8: The PCB Layout

they had to be generated for this project. The edge connector plugs into a PC-AT slot, whereas the two headers on the left hand side fit onto the controller evaluation board.

In order to achieve a good signal to noise ratio, to avoid cross talk between the digital and analog circuits, and to avoid ground bouncing the following measures were taken:

- Separate power planes for ground and +5V
- Power planes split for analog and digital circuit
- Analog and digital ground connected in one point (ADC)
- Separation of the components of analog and digital part
- Power supply filter capacitors as close as possible to each integrated circuit
- Short tracks for clock and bus signals

4.2.2 Place and Route

Placing the components on the board was performed manually. Some components required a specific placement such as the connectors or the bus drivers; others were arranged in a way that the resulting tracks were as short as possible.

An automatic router performed the routing process. Protel PCB comes with a built in 'rip up and retry' autorouter. But the performance of this tool is not overwhelming. Protel also sells a separate 'shape based' router, which is much better. This tool was used for this project.

Placing and routing was performed in iteration steps. Since the automatic router took only about ten minutes to complete a routed board, I could perform several loops consisting of placing the components, routing the board, and rearranging the components.

The final step consisted of reviewing the automatically generated layout and manually re-routing tracks that were placed clumsily.

4.3 INTERFACE PLD

In order to make the FPGA code downloadable, the logic of the interface between the host and the FPGA must permanently reside on the board. A programmable logic device (PLD) was chosen to implement the decoding logic. These parts are reprogrammable, available from numerous manufacturers, and relatively cheap.

The decoding includes the following functions:

- Decoding of an address range of 64 addresses and generation of a select signal
- Four selectable base addresses
- Generation of a reprogram signal
- Generation of direction and enable signals for the bus drivers
- Busy signal from FPGA to inhibit write accesses from the host when the local data bus is used to store acquired data
- Access selectable from either ISA bus or controller

The PLD code is listed in appendix B.

4.3.1 Simulation

Figure 9 shows the functional simulation of the interface PLD. Four accesses from the ISA bus (ISA_SEL high) and one access from the controller (ISA_SEL low) were simulated. The active-low chip select signal for the FPGA (!CS_FPGA) is derived from the address lines only. The read and write signals are not considered. The bus driver enable signals '!CS_ISA' and '!CS_CONTR,' on the other hand, are derived from the addresses and the read/write signals. This is necessary because the data bus on the board has to be kept free if no access takes place, in order to allow the FPGA to boot from the ROM or the download cable.

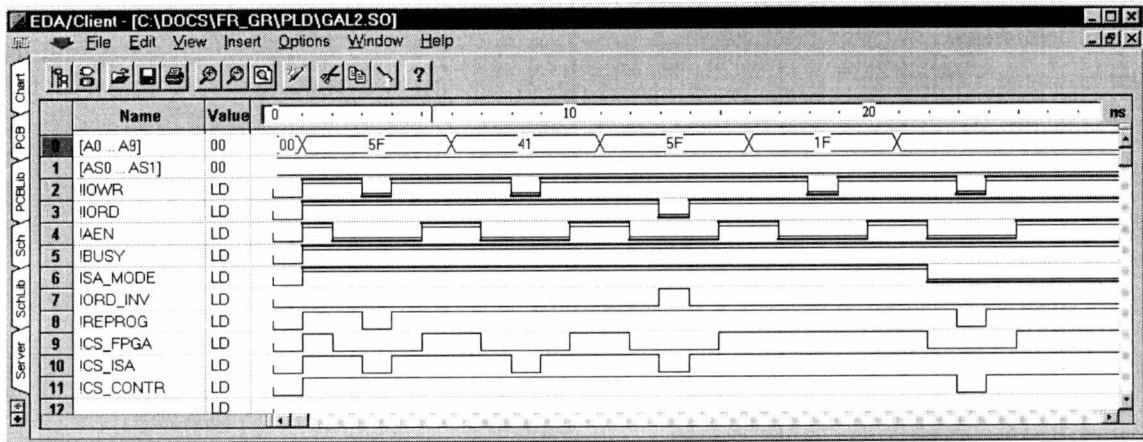


Figure 9: Decoding PLD Simulation

4.4 ANALOG INPUT

The analog input is kept as simple as possible. The video signal is terminated with 75Ω , amplified, and low-pass filtered before it enters the ADC.

Figure 10 shows the filter characteristics. Filter 1 is built around the operational amplifier U8, whereas filter 2 is a passive filter formed by R2 and C40. The solid line in figure 10 shows the overall filter characteristic.

4.5 RESTORATION OF THE DC LEVEL

In order to adjust the AC-coupled video signal to the input range of the ADC, the DC level of the signal has to be restored. The DC restoration is implemented using a tri-state buffer of the FPGA. This output is driven low during the sync pulses and tri-stated

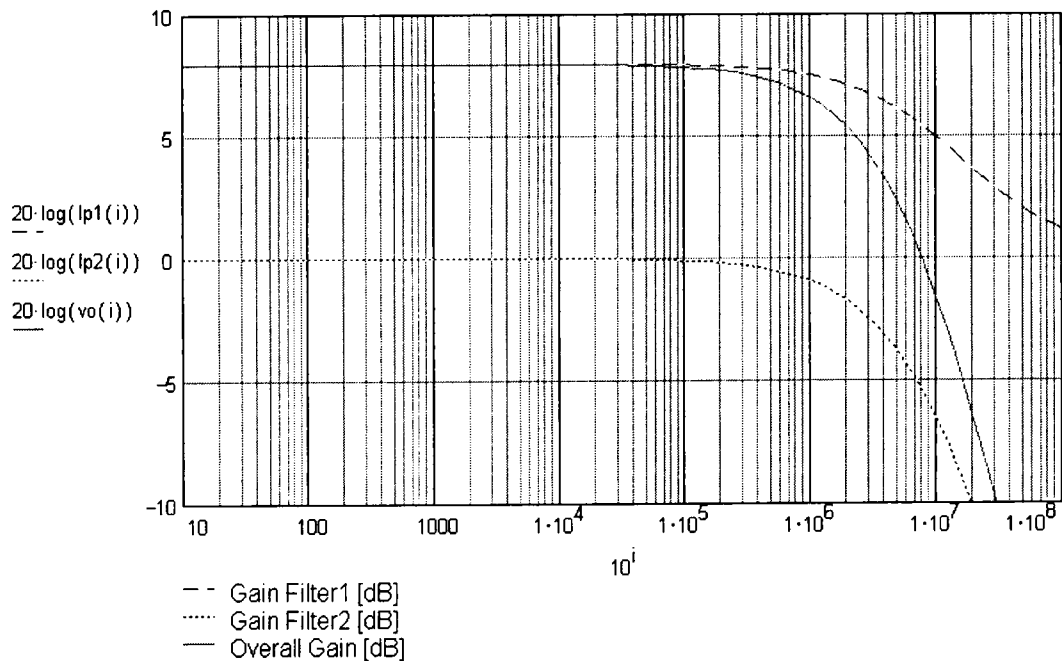
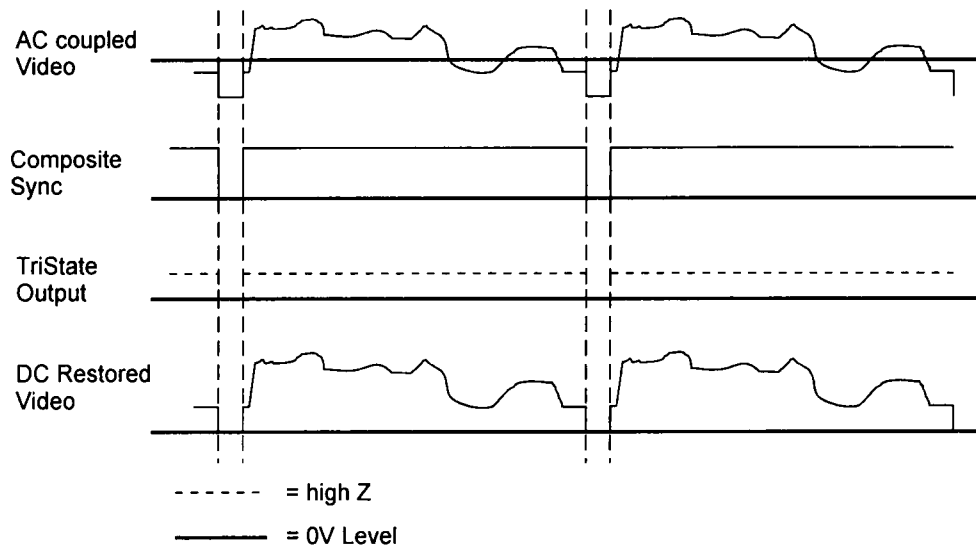


Figure 10: Low Pass Filter Characteristic

otherwise. The sync pulses are generated in the sync separator U1.

Figure 11 shows the signals associated with the DC restoration. The incoming video signal has an amplitude of $1V_{pp}$, where the sync pulse goes from 0V to 0.3V and the actual signal from 0.3V to 1V. The input amplifier has a gain of 2.5. The resulting voltage range of the video portion of the signal after the amplifier and DC restoration is from 0.75V to 2.5V. This matches the input range of the ADC, which is from 0.6V to 2.6V, with sufficient room for tolerances.

**Figure 11: DC Restoration**

4.6 PHASE LOCKED LOOP (PLL)

A phase locked loop (PLL) is used to generate a faster clock signal from a given clock, where the generated clock frequency is exactly N times bigger, where N is an integer. Figure 12 shows the block diagram of a PLL. A comparator generates a voltage proportional to the phase between the input signal and the feedback signal. The voltage is filtered and fed into a voltage-controlled oscillator (VCO). The whole loop adjusts the output frequency so as to keep the phase between the input signal and the feedback signal constant.

The comparator and the VCO usually reside in the same chip, whereas the loop filter is built of discrete components and the divider is implemented using logic devices. A PLL is used in this project to reconstruct the pixel clock from the horizontal sync pulses of the

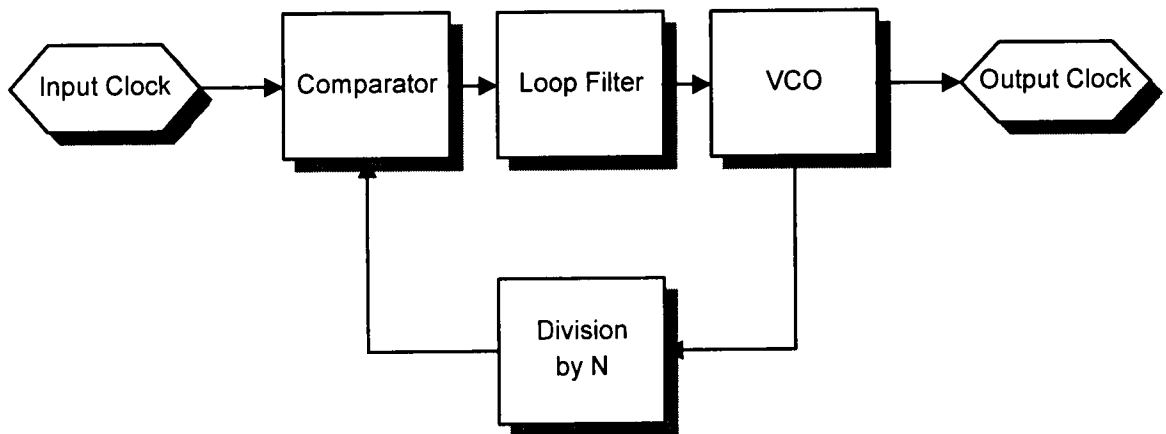


Figure 12: Phase Locked Loop Principle

video signal. The feedback divider is part of the FPGA.

The utilized PLL chip MC74HC4046 (U4) offers three different phase comparators. The behavior of the applied one is shown Figure 13.

The output of the phase comparator is normally tri-stated and the VCO voltage kept stable in the loop filter. If the rising edge of the input clock occurs before the rising edge of the feedback clock, the output is driven high. The VCO voltage and therefore the output frequency are increased. If the rising edge of the feedback clock occurs first, the output is driven low, decreasing the voltage and therefore the frequency.

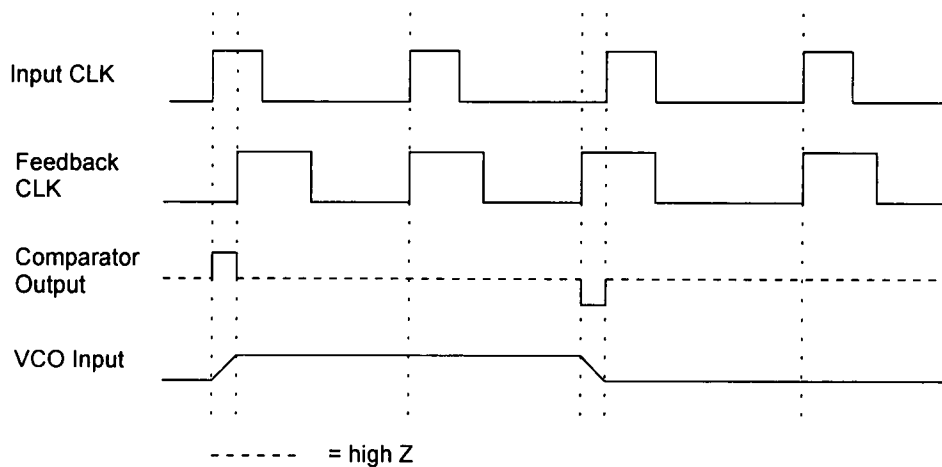


Figure 13: PLL Phase Comparator

4.7 FPGA DESIGN

4.7.1 Overview

The FPGA contains all the logic necessary to build an interface to the host, control a video frame acquisition, form a serial interface, control the flash timing, and generate the feedback signal for the PLL. The logic was implemented using the high-level hardware description language VHDL. All modules are listed in appendix C.

Figure 14 shows the structure of the design. Every block represents an entity of the design.

The library contains information relevant to all entities, such as type and constant definitions. All I/O and register addresses are defined here. Therefore, changing an address has to be performed only once and is recognized by all entities. The library has

the same function as a header file in a C code.

The root entity does not contain any logic. Its sole purpose is to instantiate the individual entities and to provide the connections between them.

Every entity was coded and simulated individually. After connecting all the modules they were re-tested using modified simulation files. Individual testing is much faster because the amount of logic is smaller and no initialization of the registers has to be performed. The individual simulation therefore allows a much deeper test depth than the simulation of the logic as a whole. This final test is mainly performed in order to verify the proper cooperation of the individual modules.

Certain modules were implemented as finite state machines (FSM) using the FSM editor. This is a very efficient way of creating control structures. FSM diagrams are much clearer than their corresponding VHDL text and they are to a large extent self-explanatory.

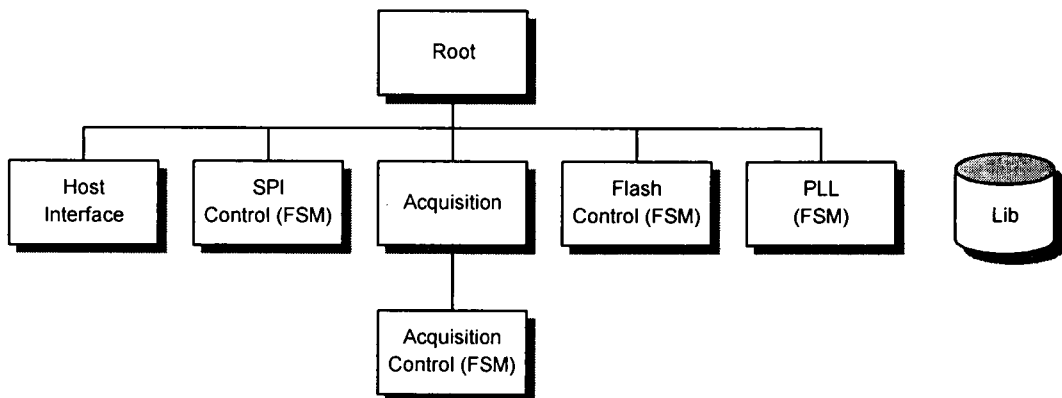


Figure 14: Design Hierarchy

4.7.2 Host Interface

The host interface forms the connection to the controlling PC or micro controller. This entity contains the bank of all control registers as listed in the following tabulated material:

Address	Name	Description
0	wp0	Wiper 0 for electronic potentiometer (SPI data)
1	wp1	Wiper 1 for electronic potentiometer (SPI data)
2	ctrl_reg	Control register
3	ver	Version number (read only)
4	l_fr_addr	Frame memory address low byte (read only)
5	h_fr_addr	Frame memory address high byte (read only)
6	h_sp	Horizontal sync pulse length (not used)
7	v_sp	Vertical sync pulse length (not used)
8	h_px	Horizontal number of pixels per frame (divided by 4)
9	v_px	Vertical number of pixels per frame (divided by 4)
A	h_off	Horizontal offset
B	v_off	vertical offset
C	d_lut	Look up table data
D	fl_pos	Flash pulse position
E	fl_len	Flash pulse length
F	pll	PLL divider (divided by 4)
10	presc	Prescale value for flash pulse position and length

The control register has the following bit assignment:

Bit	Name	Description
0	Busy	Busy (read only)
1	Wiper select	Potentiometer wiper select
2	Memory A16	Frame memory A16 (read only)
3	Odd/Even	Acquisition start with odd or even frame
4		unused
5	Reset address	Reset frame memory address (write only)
6	Start acquisition	Start frame acquisition (write only)
7	Start write	Start potentiometer write over SPI (write only)

Write accesses to the registers are performed synchronously to the ISA clock using the mechanism described in chapter 3.4.

An initial version contained a fully combinatorial output multiplexer in order to perform a readback from all registers. This part of the design turned out to be very large. The actual version contains a reduced multiplexer using internal tri-state buffers instead of logic gates and allowing an output from eight sources.

The local data bus on the acquisition board is shared between host bus drivers, frame memory, LUT, and FPGA. Write accesses from the host are inhibited during an acquisition, thus allowing an undisturbed storage of data. Nevertheless, the host must be able to read back a status flag, the so-called 'busy bit,' in order to know, whether there is an acquisition in progress or not. The data bit 'D0' is treated specially for this reason. A separate Data bit 'D0S' is used for the RAM, allowing a simultaneous data transfer to the RAM and an indication of the acquisition status to the host.

4.7.3 Serial Peripheral Interface (SPI)

The serial peripheral interface (SPI) is implemented using a finite state machine (FSM) as shown in Figure 15. The SPI is a very commonly used interface and sometimes referred to as 'three-wire serial interface' or 'micro-wire interface.' It consists of a parallel to serial shift register for the data, an associated clock, and a select signal. Its principle is best illustrated with the simulation in Figure 16.

The FSM diagram in Figure 16 contains the input and output signals represented by arrows at the top, the local signals represented by small circles, and the actual state-transition diagram below.

The sequence starts with the 'Ready' state. This state is represented by a register value of '0'. Since all flip-flops are set to '0' at power-up and when the external reset signal is applied, the state machine always starts with the 'Ready' state. This state is also the so-called trap-state, indicated by the hatching. Should the state-machine go into an illegal

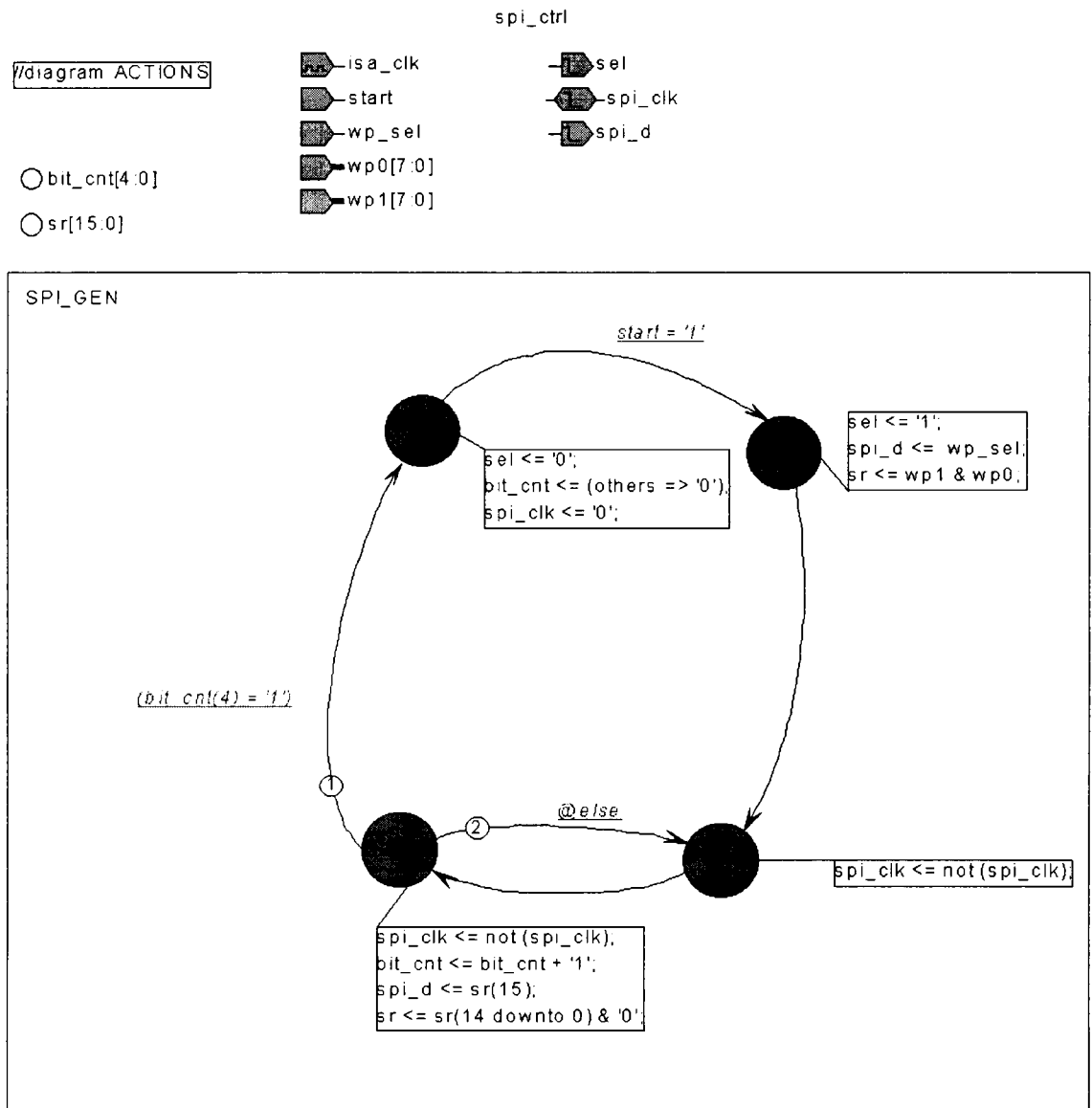


Figure 15: SPI Finite State Machine

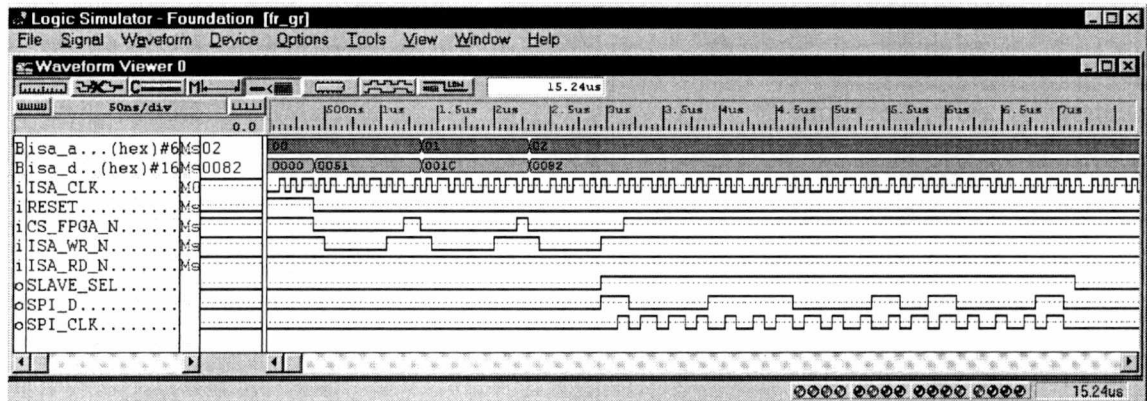


Figure 16: Simulation of the SPI

state, it automatically returns to the trap-state.

In the 'Ready' state the clock and select output are set to '0' and the counter for the number of bits shifted out is reset. If the condition of a transition, indicated by an underlined italic statement, is met, the state machine proceeds to the next state. This means in our case that if the 'Start' input is '1', the state machine proceeds to state 'S1.' This state is represented by a register value of '1'. The select output is set to '1', the first data bit (wiper select) is assigned to the output, and the two data registers 'wp0' and 'wp1' are copied to the shift register 'sr.'

The next transition is unconditional and leads to state 'S2.' The clock output is toggled here. Since it was '0' before it is assigned a '1'. States S2 and S3 are run through a total of 17 times (two 8-bit registers and the select bit). With every pass a full clock cycle is generated and data are shifted one bit.

If the counter reaches the number 16, the state-machine returns to the 'Ready' state.

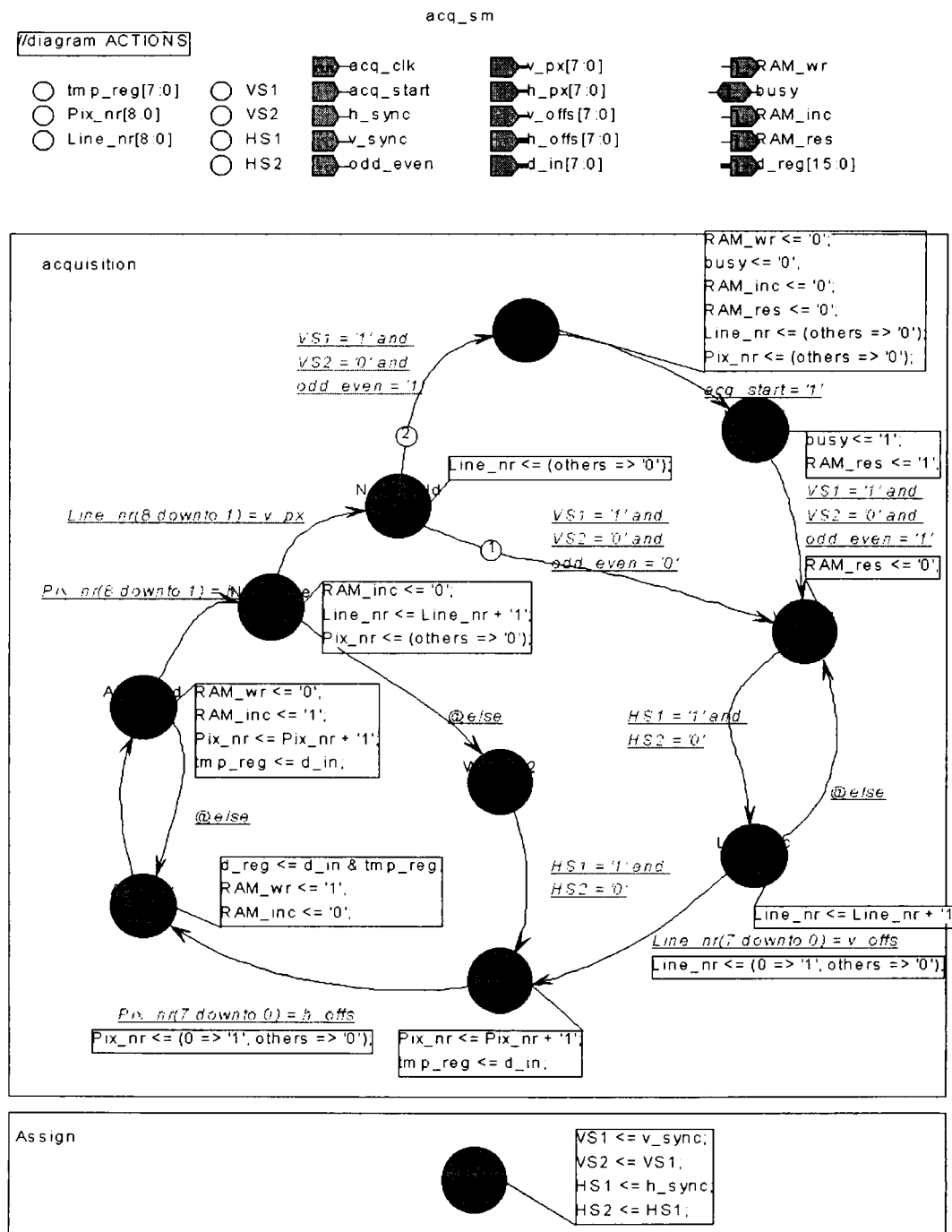
This is expressed with the condition `bit_cnt(4)='1'`. The number in circles in the transition arrows determines the order in which the statements are evaluated. In this case it is redundant, because the second statement is `else`. This means that only one of both conditions can be true and therefore no priority is necessary.

4.7.4 Acquisition Subsystem

The acquisition subsystem is controlled by a FSM as shown in Figure 17. A new element compared to the SPI FSM is the concurrent FSM called 'Assign.' Its only purpose is to synchronize and delay the incoming signals 'h_sync' and 'v_sync.' The resulting signals are used to detect a rising edge using the technique explained in chapter 3.4.

The best way to explain the FSM is to pass through it:

- *Idle* is the trap and the reset state (encoded as '0'). All output signals and counters are reset in this state. If a high on the 'acq_start' signal is encountered, the FSM proceeds to *Wait_V*.
- *Wait_V* sets the busy signal and the RAM address reset signal. The FSM waits for a rising edge of the vertical sync signal in conjunction with a high level on the 'odd_even' signal. After this edge is detected, the FSM proceeds to the *Wait_H* state.
- *Wait_H* resets the RAM address reset signal and the FSM waits for a rising edge of the horizontal sync signal.
- *Line_Inc* counts the number of lines encountered. If the count is smaller than the given vertical offset, the FSM goes back to *Wait_H*. The FSM remains in this loop until the number of lines skipped corresponds to the number given in the 'v_offs' register.



- *PrePix* counts the number of pixels to be skipped in a line and works similar to 'Line_Inc.' These two states use the same counter register. It is reset in the transition between the two states. The incoming ADC data are stored in the temporary register 'tmp_reg.'
- *Acq_Ev* stores the current and the temporarily stored pixel value in the 'd_reg' register. This register forms the data for the frame memory. In this state the write signal is asserted and the RAM address increment signal is deasserted. The pixel counter was set to a value of '1' in the transition between 'PrePix' and 'Acq_Ev.'
- *Acq_Odd* resets the RAM write signal, sets the RAM address increment signal, increments the pixel counter, and stores the incoming ADC data in the temporary register 'tmp_reg.' The FSM loops through 'Acq_Ev' and 'Acq_Odd' until the desired number of pixels is acquired. The number pixels actually acquired is four times the value of the 'h_px' register.
- *Next_line* deasserts the RAM address increment signal, increments the counter for the number of lines processed, and resets the pixel counter. If the number of lines reaches twice the content of the 'v_px' register, the FSM proceeds to 'Next_field' state, otherwise it goes to the 'Wait_H2' state.
- *Wait_H2* waits for a rising edge of the horizontal sync signal in order to process the next line.
- *Next_field* waits for the next field of the video frame. Since we started with 'odd_even' being high, the next frame starts with 'odd_even' low. If 'odd_even' is high again, we already run twice through the state machine and the acquisition is terminated by proceeding to 'Idle.'

Originally, the acquisition FSM and the RAM address counter were running on PLL clock permanently. This caused some problems when the RAM was accessed for read.

Although all signals were properly synchronized, the automatic address increment failed from time to time leading to wrong read values. This was due to the fact that the PLL does not generate a constant frequency. Since the horizontal sync pulses have doubled frequency before, during, and immediately after a vertical sync pulse, the PLL shifts the frequency up and down.

The work-around consisted of the clock multiplexer shown in Figure 18. The multiplexer switches between the two clock sources such that a minimal clock period can be maintained. Switching asynchronously can lead to very short pulses indicated by time $t1$. Using the faster clock as synchronizing clock and switching only if both clocks are high, a minimum clock period $t2$, which is half of $T2$, can be guaranteed.

The clock multiplexer as well as the RAM address counter are located in the 'acquisition' module. The acquisition controller state machine is instantiated in this module.

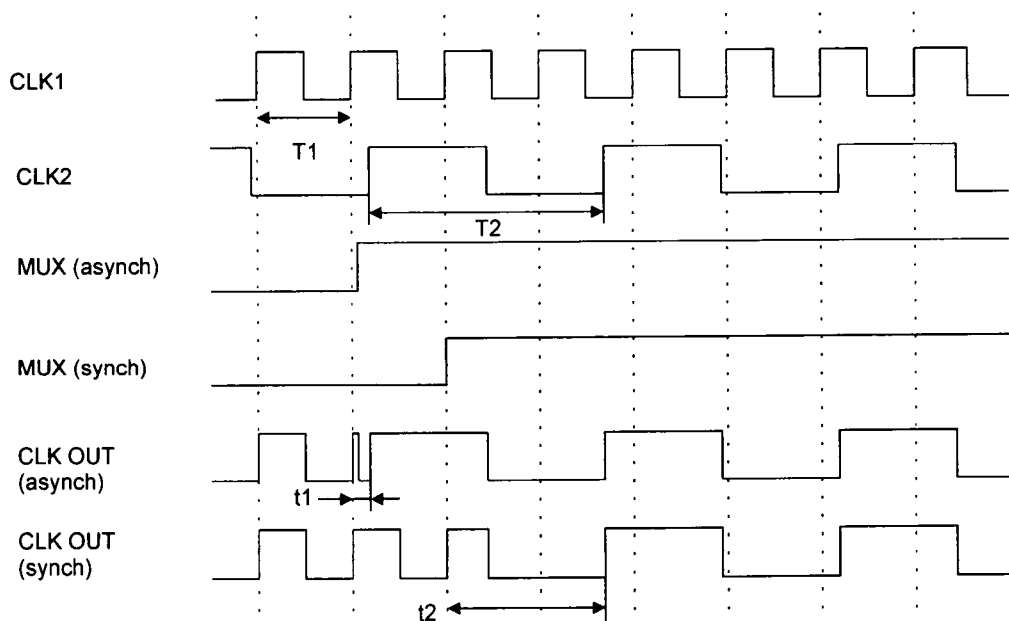


Figure 18: Synchronized Clock Multiplexer

Figure 19 shows a part of the simulation of the acquisition. One can see that, after the horizontal sync pulse goes high, a number of pixels are overread and subsequently data are written to the RAM. Also the assertion of the DC restoration signal can be seen.

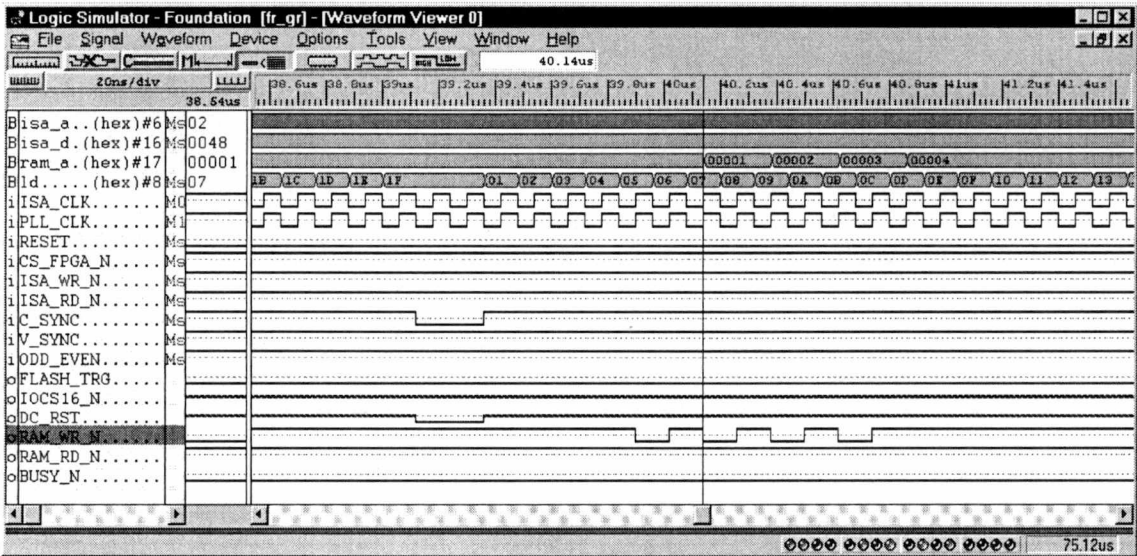


Figure 19: Acquisition Simulation

4.7.5 Flash Trigger

The flash trigger is a timing circuit implemented to generate an ignition signal for a flash. Figure 20 shows the FSM of the implementation. The timer is started alternatively with the beginning of an odd or even frame. Both the delay between the frame start and the pulse start and the pulse length are set by 8-bit counter registers and 4-bit clock prescalers. As in the acquisition FSM, a concurrent state machine is used to assign the synchronized and delayed vertical sync signals.

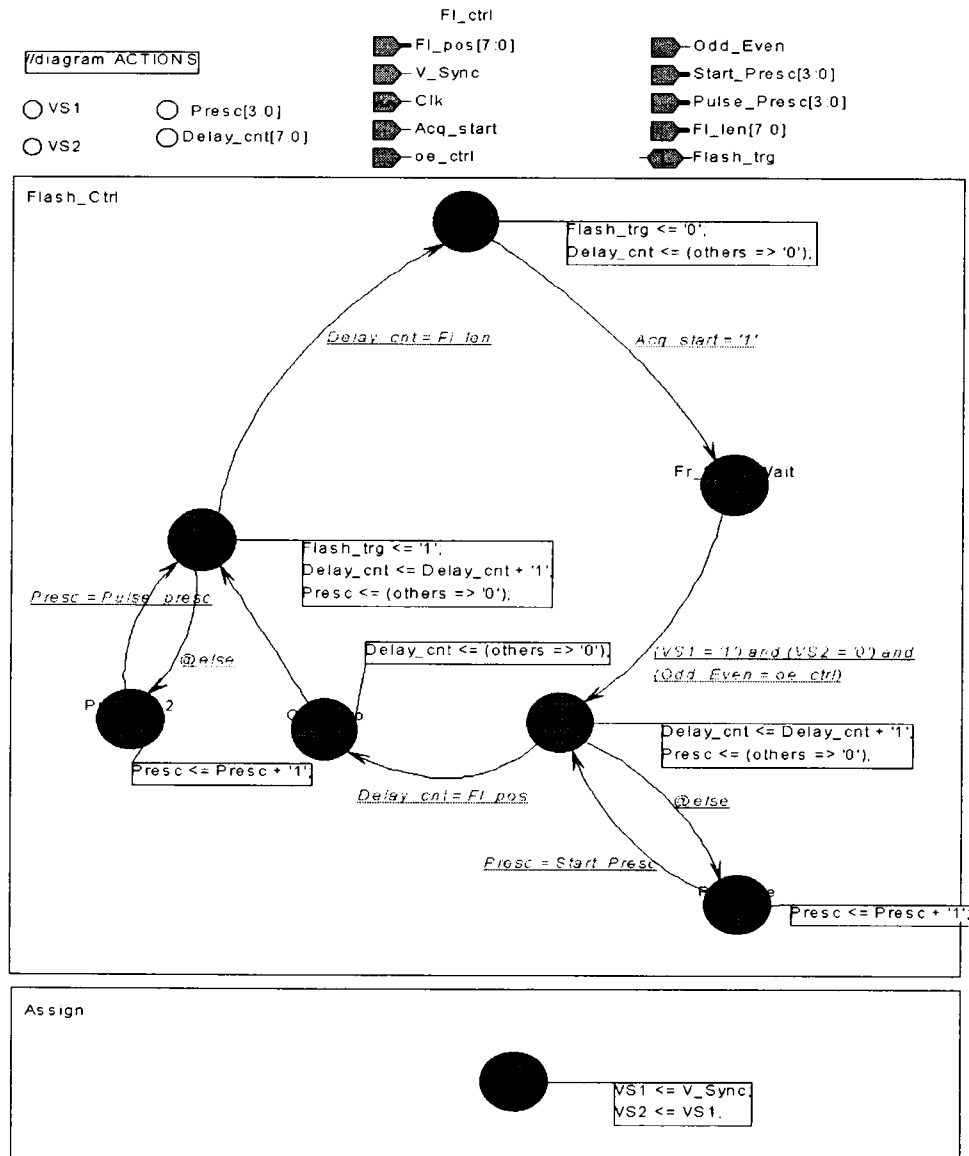


Figure 20: Flash Trigger Finite State Machine

4.7.6 Phase Locked Loop (PLL)

The logic portion of the PLL, consisting of a divider, is implemented in the FPGA. Figure 21 shows the corresponding FSM. Only two states are needed, a toggle state and a down scale state. The generated clock is symmetric and its frequency is $4 \cdot (n+1)$ times lower than the input frequency, where n is the value set in the PLL register.

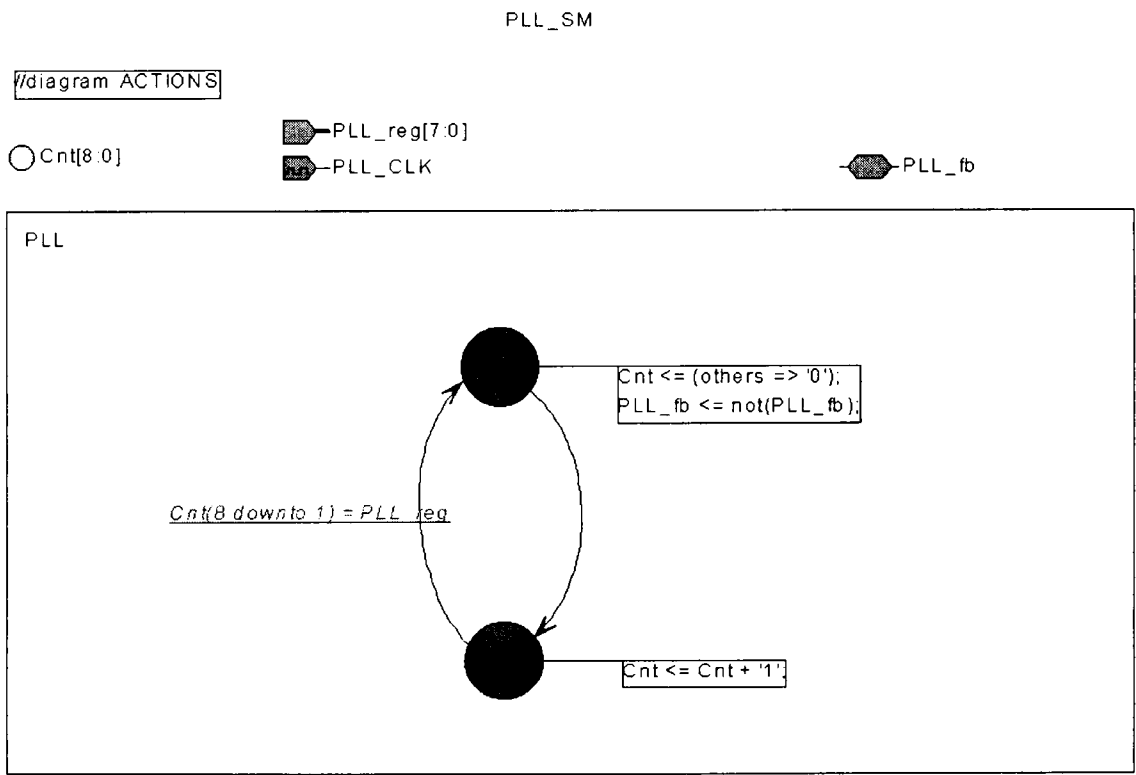


Figure 21: PLL Finite State Machine

Figure 22 shows the simulation of the PLL. The generated frequency after the reset is four times lower than the PLL frequency since the PLL register is initially set to '0'. After writing a value of '4' to the register, the output frequency is 20 times lower than the input frequency.

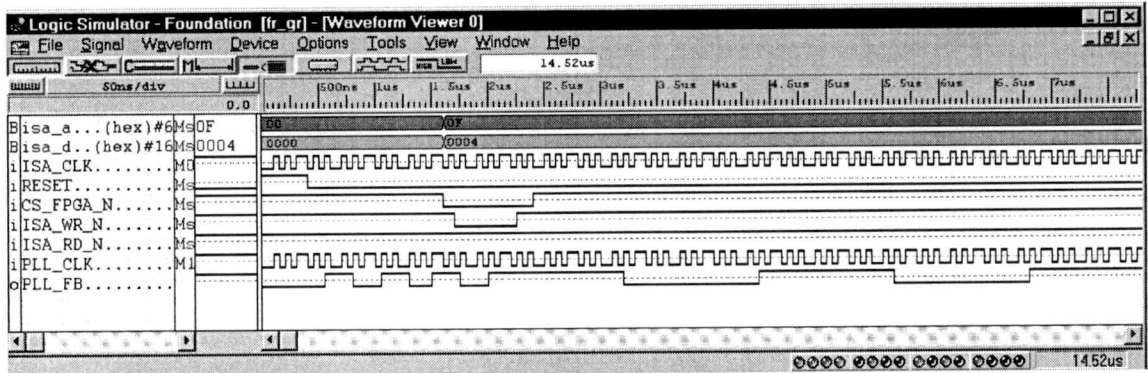


Figure 22: Simulation of the PLL

4.8 TIMING ANALYSIS

The timing analysis is a very important step in the design flow. Poor timing performance can lead to randomly occurring errors, which are very hard to reproduce and correct. These errors mostly do not occur in the prototype phase but later in production, when larger lots are being produced and real operating conditions have to be faced.

There are two, basically different approaches for a timing analysis. The first is the use of a timing analysis tool. This analyzer traces every signal through the whole logic and filters out the worst case delays of all paths. For synchronous logic, these delays can be sorted according to the different clock sources and the maximum clock frequency for each source can be derived. This is a very powerful tool, since a statement about the timing performance can be made within seconds. This analysis can only be applied to

synchronous logic, which is another reason to favor synchronous designs.

The timing report for this project looks like this:

Timing Analyzer Report File:

Design: c:\docs\xilinx\fr_gr\xproject\v1_11\rev1\fr_gr.lca
(5204PQ100-6)

Program: TIMING ANALYZER 5.2.0

Speedsfile: File 5204.spd, Version 5200.5, Revision 5204.11

Xdelay timing analysis options:

From all.

To all.

Worst case Pad to Pad path delay: 78.8ns (7 block levels)
Pad "ISA_A<4>" (P24) to Pad "ISA_D<0>" (P75.0)

Clock net "BUFG1/N23P" path delays:

Pad to Setup: 53.1ns (4 block levels)
(Includes an external input margin of 0.0ns.)
Pad "CS_FPGA_N" (P61) to LC2_FF Setup (D) at "N19G.LC2.F3"
Target LC2_FFX drives output net "U6_RAM_A<7>"

Clock to Pad: 69.9ns (5 block levels)
(Includes an external output margin of 0.0ns.)
Clock to Q, net "U6_U1_BUSY_3" to Pad "ISA_D<2>" (P71.0)

Clock to Setup (same edge): 68.8ns (8 block levels)
Clock to Q, net "U6_RAM_A<3>" to LC3_FF Setup (D) at
"N20H.LC3.F3"
Target LC3_FFX drives output net "U6_RAM_A<16>_1"

Clock to Setup (to other setup type): 42.5ns (3 block levels)
Clock to Q, net "U6_U1_BUSY_3" to LC1_FF Setup (D) at
"N17M.LC1.F2"
Target LC1_FFX drives output net "U2_D_LUT_WR_EN"

Minimum Clock Period : 69.9ns

Estimated Maximum Clock Speed : 14.3Mhz

Clock net "BUFG2/N48S" path delays:

Pad to Setup: 17.5ns (0 block levels)

(Includes an external input margin of 0.0ns.)
 Pad "V_SYNC" (P38) to LC0_FF Setup (D) at "CLB7.LC0.F4"
 Target LC0_FFX drives output net "U5_V_SYNC_1"

Clock to Pad: 20.1ns (0 block levels)
 (Includes an external output margin of 0.0ns.)
 Clock to Q, net "U5_PLL_FB" to Pad "PLL_FB" (P2.O)

Clock to Setup (same edge): 32.1ns (3 block levels)
 Clock to Q, net "U5_CNT<0>" to LC1_FF Setup (D) at
 "N24R.LC1.F3"
 Target LC1_FFX drives output net "U5_CNT<7>"

Minimum Clock Period : 32.1ns

Estimated Maximum Clock Speed : 31.1Mhz

Clock net "N47H" path delays:
 Pad to Setup: 46.0ns (2 block levels)
 (Includes an external input margin of 0.0ns.)
 Pad "ISA_WR_N" (P81) to LC1_FF Setup (D) at "N16T.LC1.F3"
 Target LC1_FFX drives output net "U2_RIS_LUT_A_Q"

Clock to Pad: 30.5ns (1 block level)
 (Includes an external output margin of 0.0ns.)
 Clock to Q, net "U2_WP0_REG<2>" to Pad "ISA_D<2>" (P71.O)

Clock to Setup (same edge): 67.3ns (7 block levels)
 Clock to Q, net "U4_FLASH_CTRL<0>_1" to LC3_FF Setup (D) at
 "N44Q.LC3.F3"
 Target LC3_FFX drives output net "U4_DELAY_CNT<4>"

Clock to Setup (rising to falling edge) : 8.3ns (0 block levels)
 Clock to Q, net "U2_RIS_WR_Q" to LC2_FF Setup (D) at
 "U6/RAM_WR_N.LC2.DI"
 Target LC2_FFX drives output net "U2_FAL_WR_Q"

Clock to Setup (to other setup type): 49.6ns (5 block levels)
 Clock to Q, net "U2_H_PX_REG<0>" to LC1_FF Setup (D) at
 "N24Y.LC1.F2"
 Target LC1_FFX drives output net "U6_U1_ACQUISITION<1>"

Minimum Clock Period : 67.3ns

Minimum Clock High pulse width : 8.3ns

Estimated Maximum Clock Speed : 14.8Mhz

The clock names are unfortunately not the same as in the source code, because a new name is assigned to them as they run through a buffer. But the original names can easily be determined. The timing file can be summarized as follows:

- The multiplexed clock for the acquisition state machine and the RAM address generation has a maximum clock frequency of 14.3MHz
- The PLL clock used for the feedback divider has a maximum clock frequency of 31.1MHz
- The ISA bus clock has a maximum frequency of 14.8MHz

With an ISA clock running at 8MHz and a PLL frequency of 10MHz the design has a sufficient timing performance.

The second approach for a timing analysis is the back-annotation and post-layout simulation of the design. This process is much more time consuming than the automatic timing analysis, but it is necessary for asynchronously generated signals. The XACT design environment allows the export of a back-annotated netlist, but it does not have generation capability (or I did not find it). Therefore, a batch file with the following commands was used to generate the back-annotated netlist:

```
xdelay -d -w fr_gr  
lca2xnf -g fr_gr.lca fr_grp.xnf  
xnfba fr_gr.xff fr_grp.xnf
```

This kind of timing analysis was performed for the LUT and RAM write signals. Figure 23 shows the functional simulation of a 16-bit access to the LUT address register and a 8-bit access to the LUT RAM. Since the 'IO-write' signal of the 16-bit access is only one-and-a-half clock cycles in length, this is the most critical one. It is not specified whether the 'IO-write' is valid before or after the falling ISA clock edge. Therefore, the generated write signal is asserted asynchronously and deasserted with the falling clock edge following the next rising clock edge (see Figure 23). It is important that the write signal is

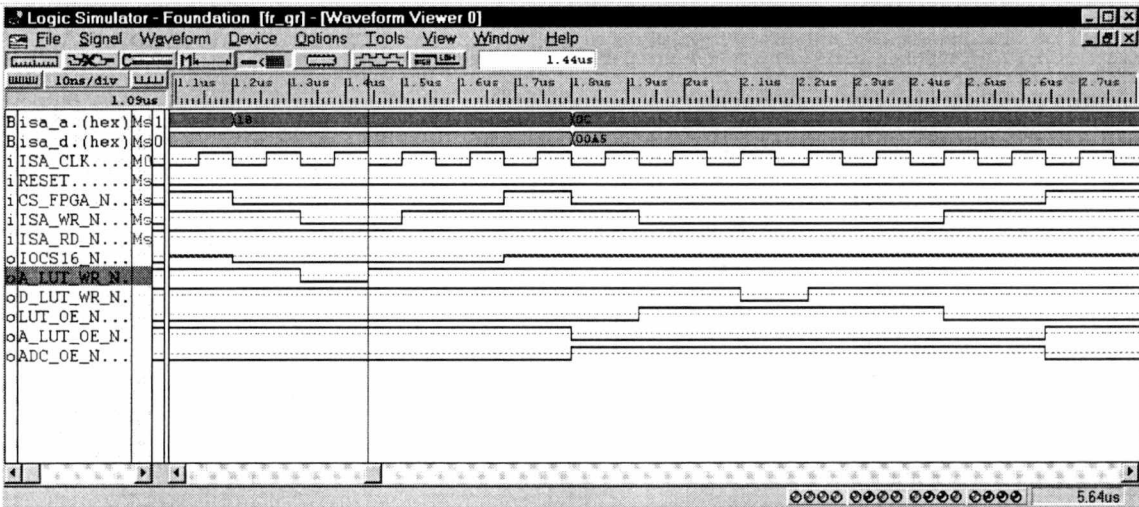


Figure 23: LUT Write Access Pre-Layout Simulation

deasserted before the deassertion of the 'IO-write' signal, because data are valid only for this time.

Figure 24 shows the same signals with the back-annotated timing information. One can see that the leading edge of the generated write signal is significantly delayed and that the trailing edge is moderately shifted, leaving enough hold time for the data.

Figure 25 shows the detailed timing analysis with the corresponding timing information. It becomes obvious that the asynchronous generation of the leading edge involves a much larger delay than the synchronously generated trailing edge. This is due to the fact that the combinatorial inputs, used for the asynchronous part, have a much higher input path delay than the clock signals.

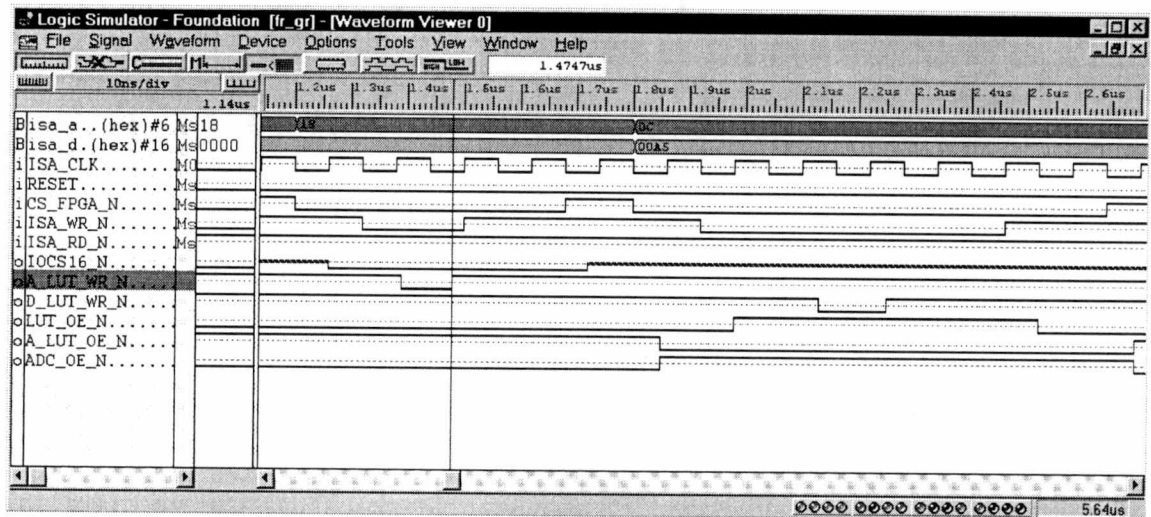


Figure 24: LUT Write Access Post-Layout Simulation

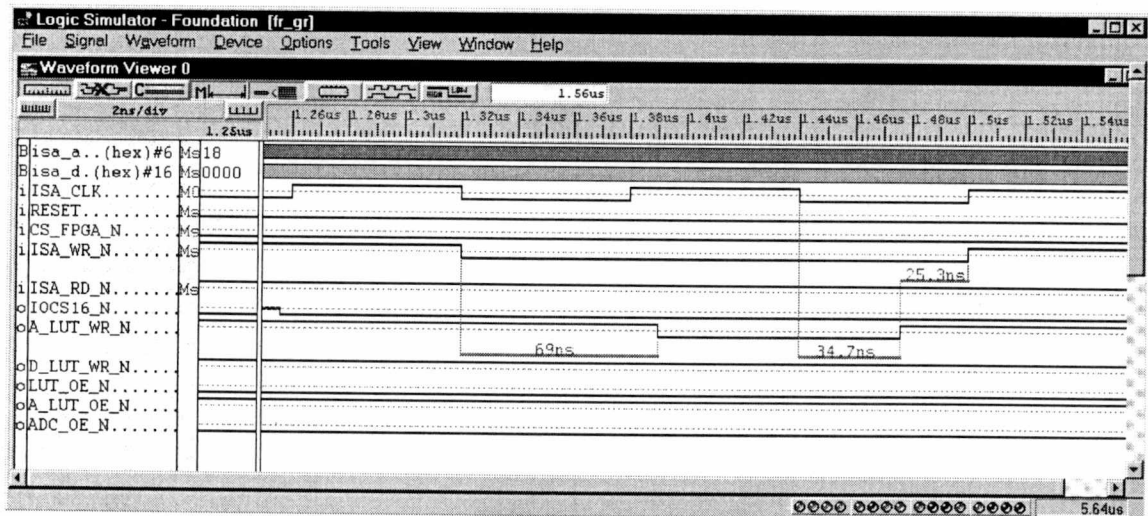


Figure 25: LUT Write Access Timing Analysis

4.9 SOFTWARE

The software written for this project was not intended to be used in a product. Its sole purpose was to operate and test the image acquisition system. Nevertheless, almost all routines were re-used and incorporated into the Windows based production software. The listing is attached in appendix C.

4.9.1 Drivers

The following driver routines were implemented for this project:

Function declaration	Description
int cfg_download(void);	Extracts and downloads the FPGA configuration code
char reverse(char ib);	Reverses bit order of a byte. Used by cfg_download
int RAM_test(void);	Performs a frame RAM test with a cyclic pattern
void conf_reg(void);	Sets all configuration registers in the FPGA
void start_pot_wr(void);	Initiates a SPI transfer to the electronic potentiometer
void start_acq(void);	Initiates a frame acquisition
void RAM_addr_rst(void);	Resets the frame memory address
void LUT_load(void);	Initializes the Look Up Table
int LUT_test(void);	Performs a LUT RAM test similar to the frame RAM test
int wait_acq_start(void);	Waits until the busy bit is set or the user aborts
int wait_acq_end(void);	Waits until the busy bit is reset or the user aborts
void make_tiff(void);	Reads back the frame memory content and generates a TIFF file

4.9.2 Download Tips & Tricks

The download driver software contains two mechanisms worth mentioning: the bit file parser and the bit reverser.

The actual code of the Xilinx bit file is preceded by a header. This header contains information such as time, date, and file name, and is therefore variable in length. The

download driver uses a parser, which is looking for a certain pattern that indicates the start of configuration data. This pattern is referred to as 'Dummy bits' and 'Preamble' in the Xilinx data book (Xilinx, 1996). The following tabulated material contains the respective data for the different families and download types:

FPGA Family	<i>Normal mode</i>		<i>Express mode</i>	
	Dummy bits	Preamble	Dummy bits	Preamble
XC3000, XC3000A XC3100A	11111111	0010	N/A	N/A
XC4000	11111111	0010	11111111	11110010
XC5200	11111111	11110010	11111111	11110010

The software and hardware implementation is set up for the XC5200 family in normal mode. If the driver is utilized for a different family or configuration mode, the code has to be changed appropriately.

Xilinx does not exactly specify the bit file format. It took some time to figure out that data stored in the bit file is actually bit reversed. This means that data bit D0 read from the file has to be written to the FPGA as bit D7, bit D1 as bit D6, etc. This order reversing is performed in the function 'reverse' of the driver software. Another possibility is to reverse the order of the data lines in the layout. Extra caution has to be taken, because the pin description of the FPGA is misleading in this case.

4.9.3 TIFF Format

Instead of writing my own viewer software, I decided to store the acquired data in a common format. A widely used format is the Tagged Image File Format (TIFF). The file is built of a header, the Image File Directory (IFD), the directory entries, and the raw gray scale values as read from the frame grabber:

Address	Value (Hex)	Description	Name
0	49 49	Byte order (LSB first)	Header
2	2A 00	TIFF Version Nr.	
4	08 00 00 00	Offset to the IFD (8)	
8	08 00	Number of tags (8)	IFD
A	FF 00	Tag 255: Type	Directory entry #0
C	03 00	type =3 (short)	
E	01 00 00 00	length = 1	
12	01 00 00 00	value = 1	
16	00 01	Tag 256: Image width	Directory entry #1
18	03 00	type =3 (short)	
1A	01 00 00 00	length = 1	
1E	00 02 00 00	value = 512	
22	01 01	Tag 257: Image length	Directory entry #2
24	03 00	type =3 (short)	
26	01 00 00 00	length = 1	
2A	E0 01 00 00	value = 480	
2E	11 01	Tag 273: Strip offset	Directory entry #3
30	03 00	type =3 (short)	
32	01 00 00 00	length = 1	
36	56 00 00 00	value = 56	
3A	15 01	Tag 277: Samples per pixel	Directory entry #4
3C	03 00	type =3 (short)	
3E	01 00 00 00	length = 1	
42	01 00 00 00	value = 1	
46	02 01	Tag 258: Bits per sample	Directory entry #5
48	03 00	type =3 (short)	
4A	01 00 00 00	length = 1	
4E	08 00 00 00	value = 8	
52	00 00 00 00	Offset to next IFD (0 → no more IFDs)	
56	xx xx xx ...	Start of image gray scale data	

A description of the basics was found in an image processing book (Phillips, 1994). The header, consisting of byte order, version number, and offset to the IFD, always has to be at the beginning of the file. The IFD and directory entries can be located either at the beginning of the file, like in the example above, or at the end of the file. The actual image data starts at address 56 (hex) as indicated by tag 273 and is continued to the end of the file.

The implementation is a minimal subset of the TIFF standard and works with *many* software packages. Some programs, like MS Word, seem not to like the implemented format. They either accept the file, ignore the data, or even shut down depending on the content. Some others, like Corel Draw, can handle the images.

5 RESULTS

Figure 26 shows the first actual picture taken with the image acquisition board. The picture shows several shortcomings, which had to be fixed in the further development. It is unfocused and has distortions at the top due to an unlocked and slowly settling PLL; the first line contains random data and every odd line is a repetition of the previous even line due to a software error. It is the drowsy view of the awakening frame grabber. It is not a beauty, but a milestone in the development of the board.

Figure 27 and Figure 28 show the same picture taken with different settings of the Look Up Table (LUT). For the first picture the LUT translates the incoming data one-to-one. For the second picture the LUT was simply inverted, i.e. a value of '0' was translated as '255', a value of '1' as '244', and so on. No CPU power was needed to do this conversion. The LUT can be used for various conversions like thresholding or contrast enhancements.

The two pictures also show a crisp image, an indicator for low noise, little distortion, and sufficient bandwidth of the analog signal.

Figure 29 shows the effect of an unsynchronized clock. Instead of the PLL clock, which is a reconstructed pixel clock, the ISA bus clock was used. This clock has no defined relation to the original pixel clock, and therefore, interference is produced. It is visible as 'zigzagged' edges.

The last picture, shown in Figure 30, shows the frame grabber board itself.

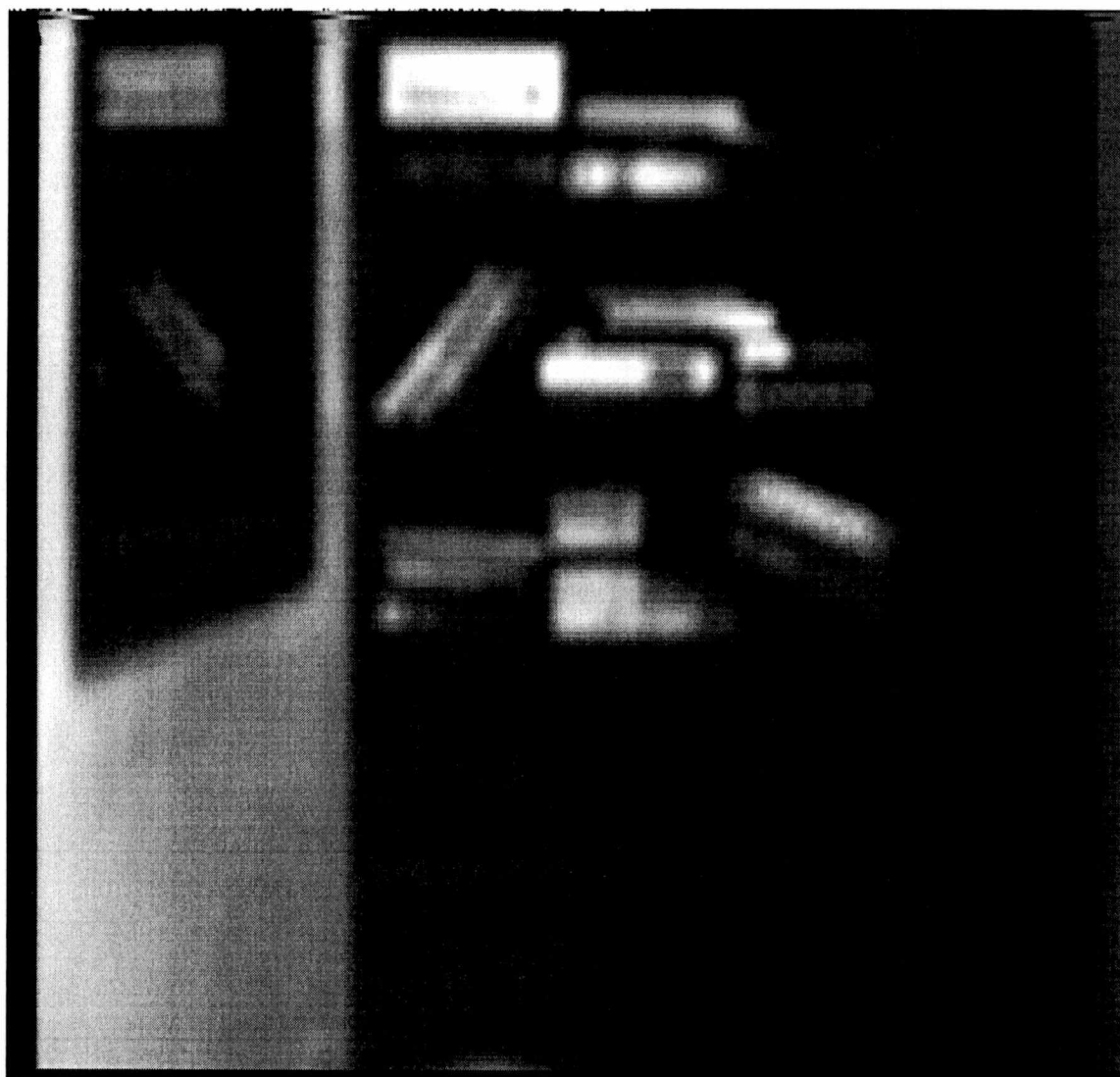


Figure 26: First Picture

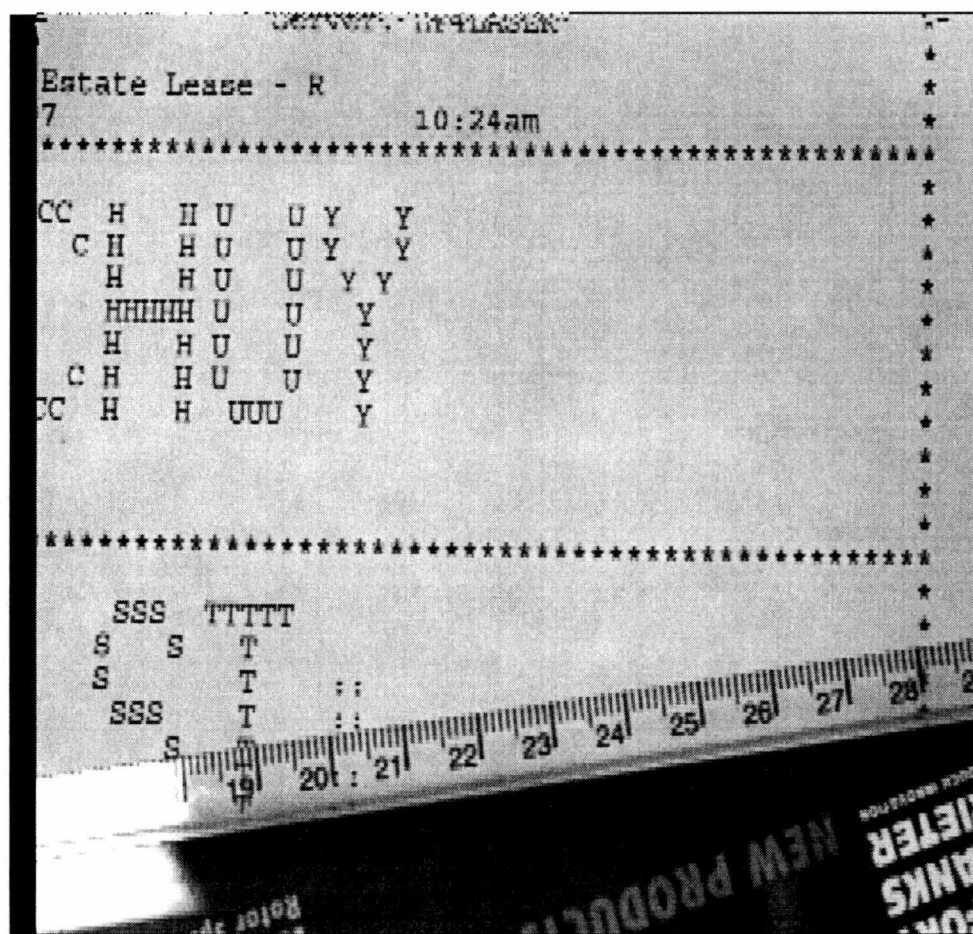


Figure 27: Positive Image

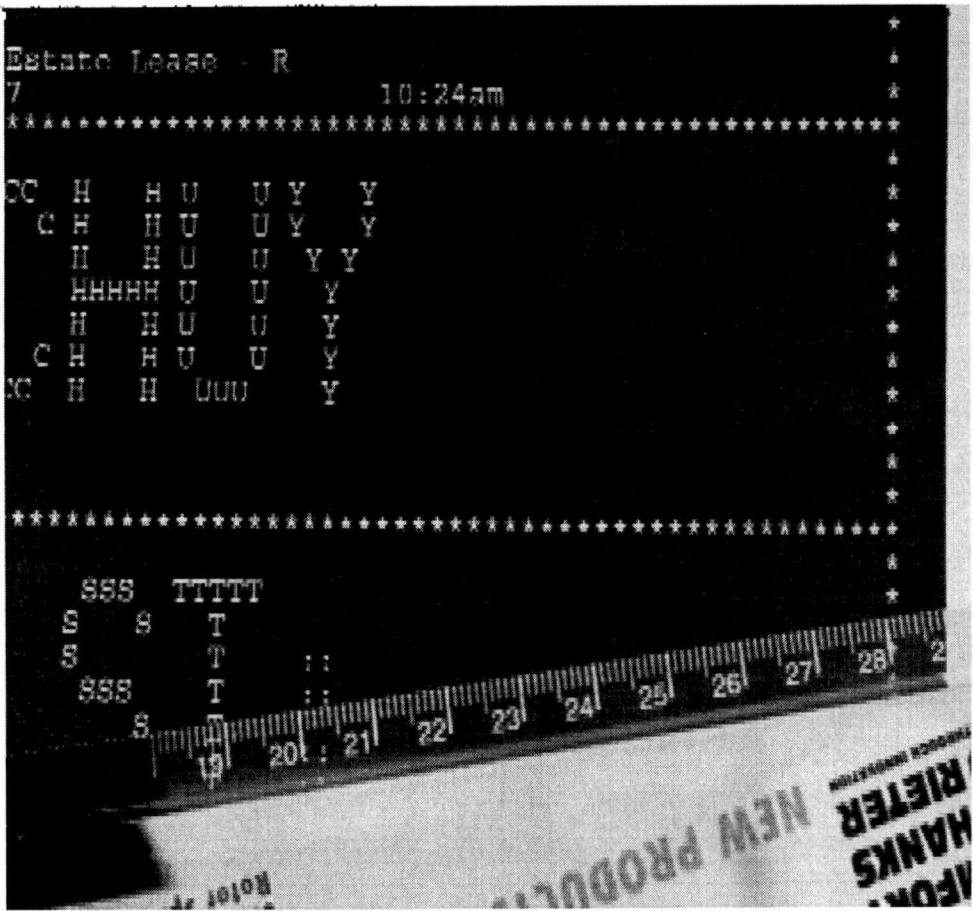


Figure 28: Negative Image (Inverted LUT)

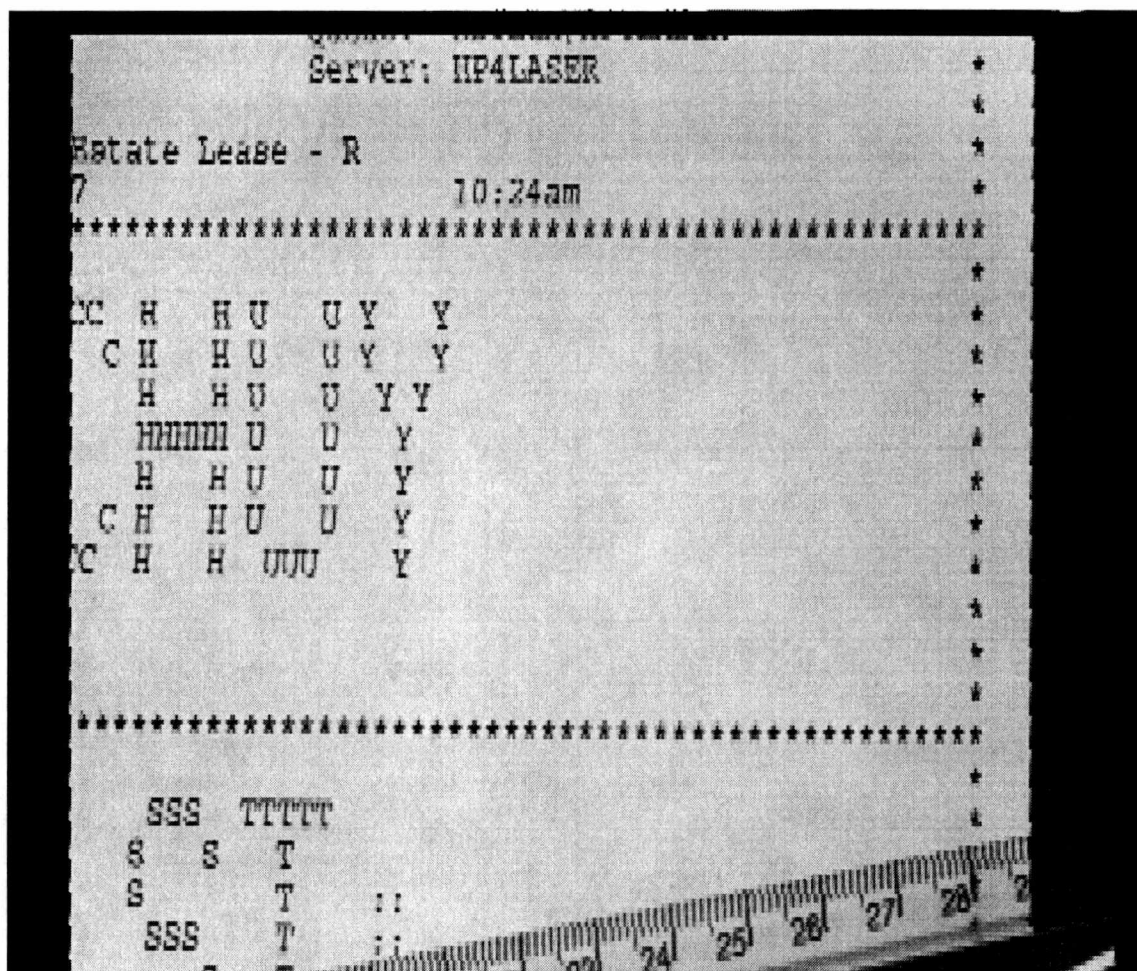


Figure 29: 'Zigzag' Effect

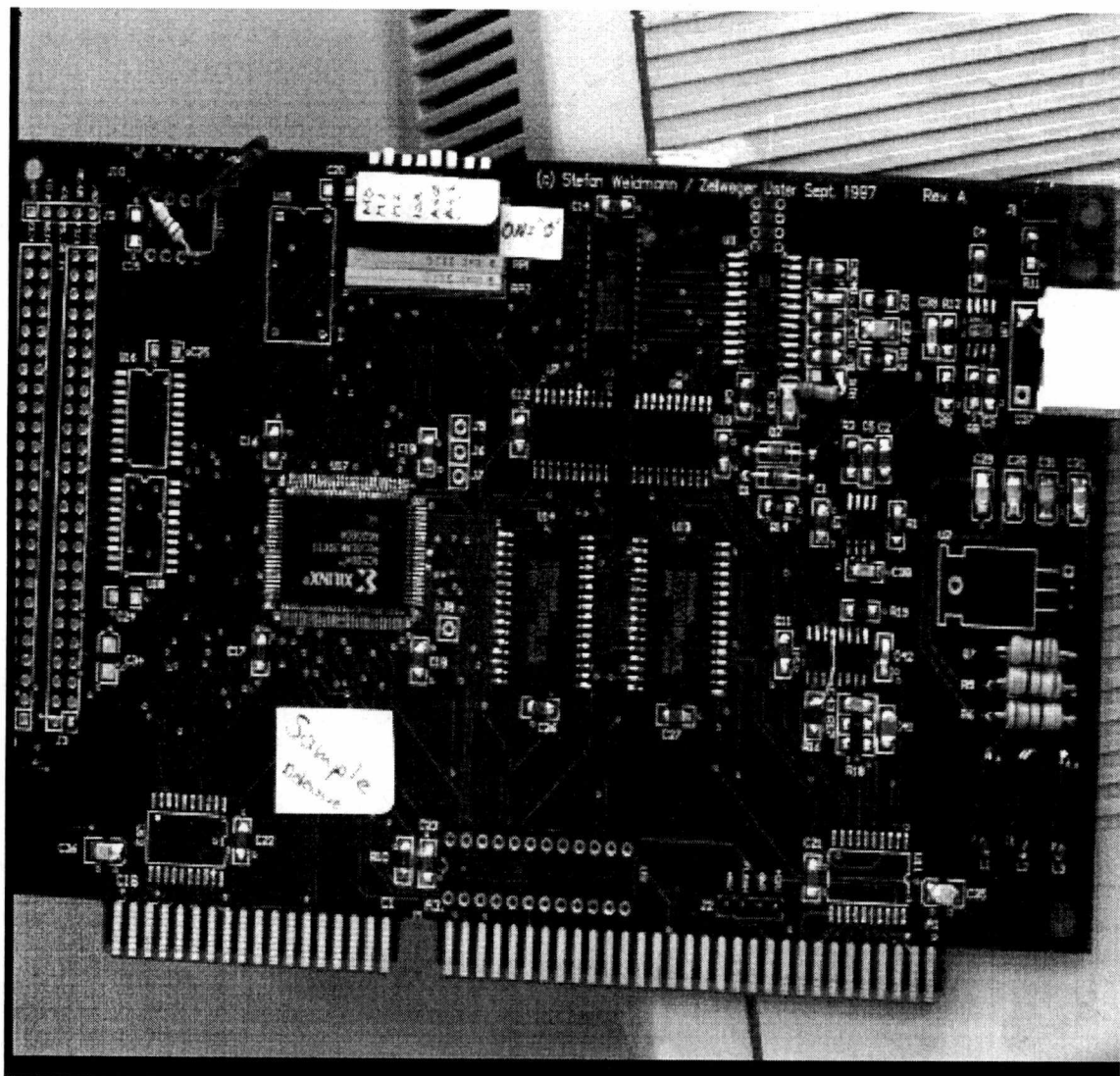


Figure 30: Picture of the Image Acquisition Board

6 SUMMARY AND CONCLUSION

An image acquisition system was entirely designed, simulated, implemented, and tested using PC based tools for the project presented here. For maximum flexibility a FPGA was used to implement the logic. The FPGA code is downloaded from the PC in order to be able to easily change and update the logic in the final product.

The PC turned out to be an attractive platform for the realization of a prototype. The powerful development tools for software, printed circuit boards (PCB), and field programmable gate arrays (FPGA) made it possible to concentrate on the problem and therefore to perform a very fast implementation. Moreover, the fact that all tools run under the same environment makes data interchange and particularly documentation very efficient.

The use of reprogrammable FPGAs, in conjunction with a high level description language and graphical tools, gives the design an additional flexibility. The generated logic code can easily be re-targeted to other technologies, such as ASICs, for high volume production.

7 OUTLOOK

Affordable and easy to use development tools have just started to become available on PCs. Especially graphically oriented tools with an intuitive MMI allow non-specialists to use them. This tendency will certainly be continued in the future, allowing a stronger concentration on the problem rather than on the tools themselves. Furthermore, the line between software and hardware becomes thinner, allowing software designers to implement logic and vice versa.

7.1 REUSABLE MODULES

One of the objectives of this project was to implement the design using a modular structure while keeping reusability in mind. There are two different kinds of reusable parts: strategies and methodologies on one side, code and schematics on the other side.

The following methods will certainly be re-applied for further projects:

- PC compatible as development platform
- PC compatible as (prototype) host
- Use of ISA bus cards for prototyping
- Use of automatically routed PCBs instead of bread-boards or wire-wrap boards
- Use of FPGAs for flexible logic implementation
- Configuration download instead of PROMs for easy upgrade
- Synchronous logic design
- Logic design with graphical tools (FSM editor)
- Extensive simulation on module and system level
- Modular system design

The reusable code and schematic parts are:

- Interface logic (PLD)
- Register bank in the FPGA
- Serial Peripheral Interface (SPI)
- Download software
- PC-side software drivers

Some of the modules can be embedded in new projects as they are, while some might need changes or modifications. But I think all of them will speed up the next design. Hopefully not only my future projects will profit of these results, but I hope that others can also take advantage of the described principles and the implementation.

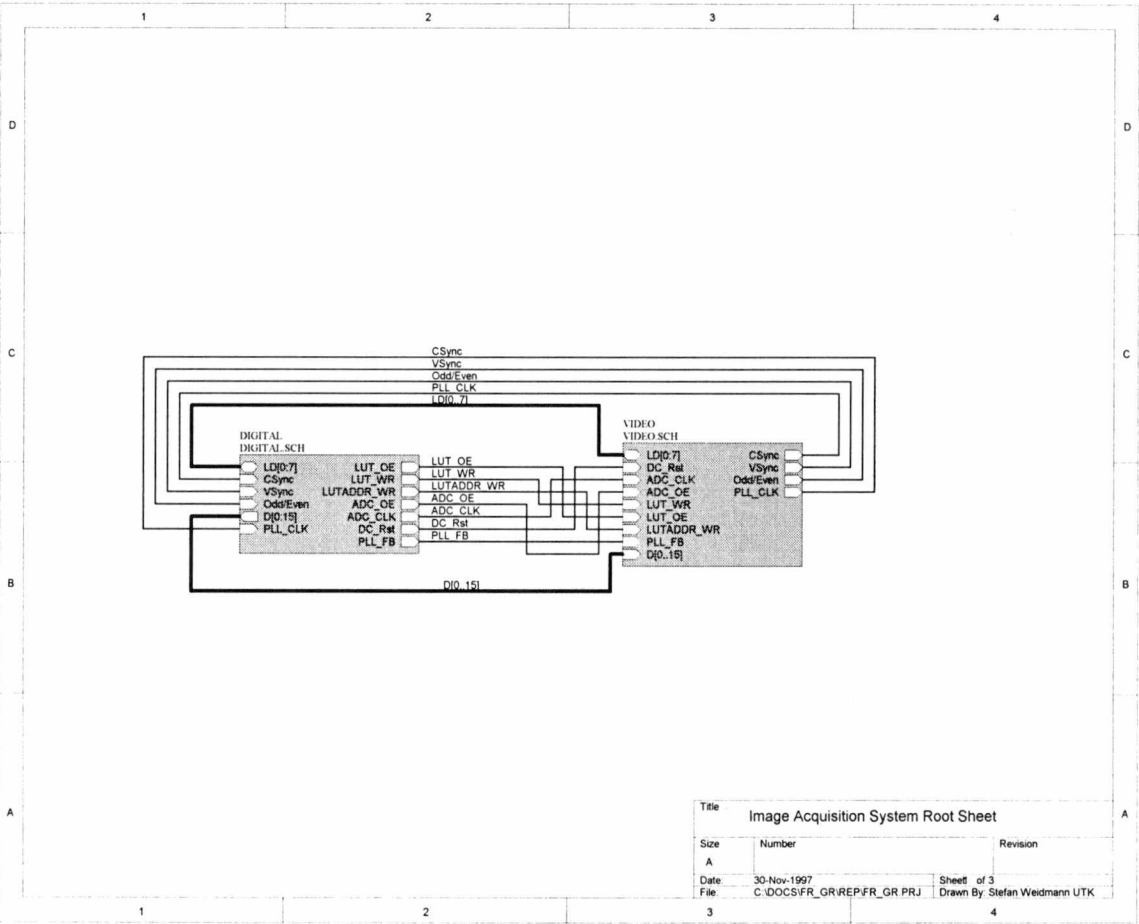
REFERENCES

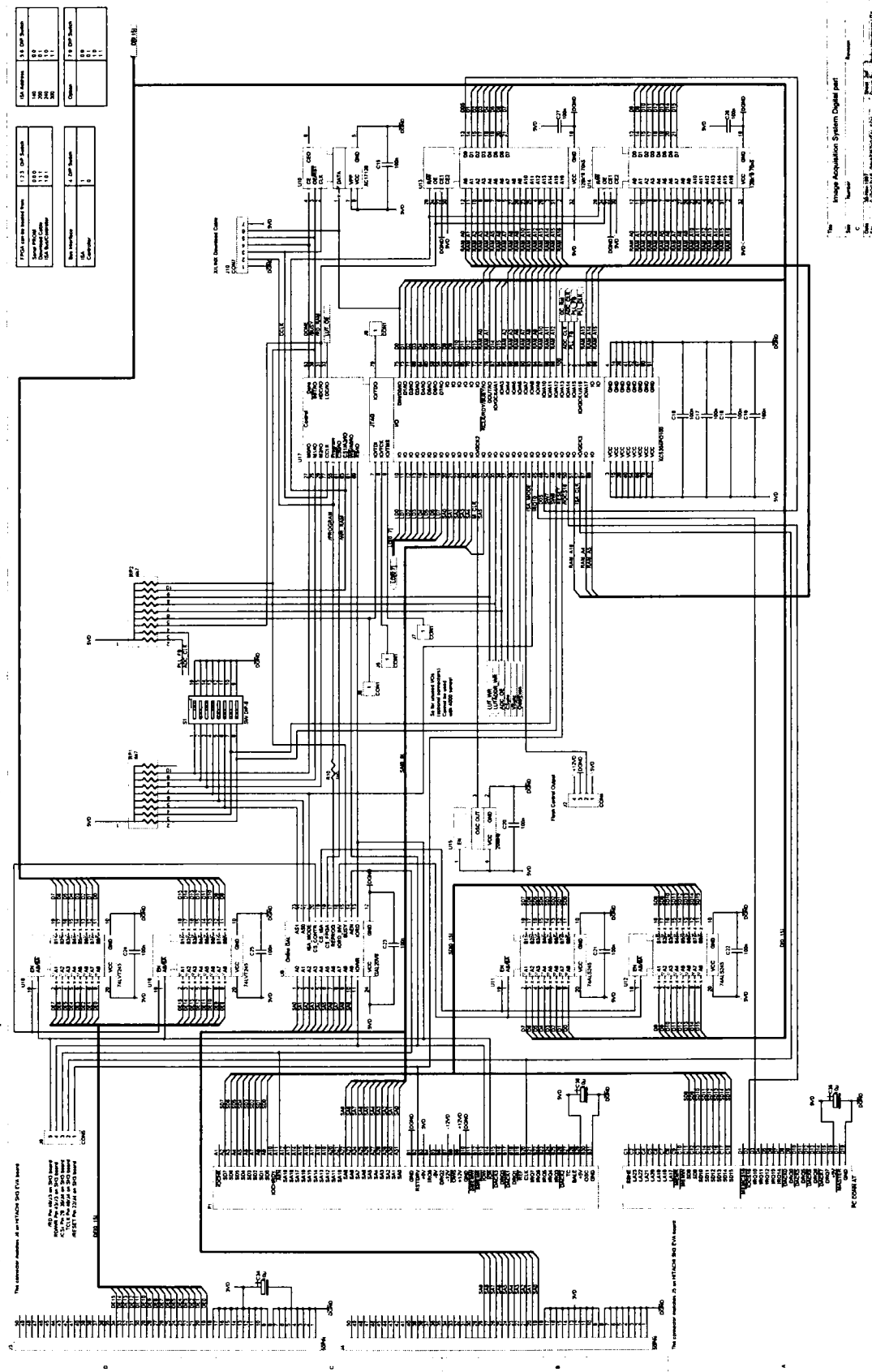
REFERENCES

- (Solari, 1993): Edward Solari, *ISA & EISA Theory and Operation*, Annabooks, San Diego, 1993
- (Xilinx, 1996): Xilinx Corp., *Programmable Logic Data Book*, (1996)
- (Xilinx, 1997): <http://www.xilinx.com>
- (Phillips, 1994): Dwayne Phillips, *Image Processing in C*, R&D Publications, Lawrence, 1994

APPENDICES

APPENDIX A: SCHEMATICS





02



APPENDIX B: PLD LISTING

```

Name      Frame grabber ISA decoding;
Partno    N/A;
Date      09/15/97;
Revision  0.3;
Designer  Stefan Weidmann;
Company   The University of Tennessee;
Assembly  ;
Location  Knoxville, TN;
Device    g20v8;

```

```

/*****
/*      Revision Date: 10/03/97      */
/*      10/10/97                    */
/*                                  */
/*****
/* Allowable Target Device Types: GAL20V8, PALCE20V8,
/* xx22V10
/*****

```

```

/** Inputs **/

```

```

Pin[1..10] = [A0..9]; /*
Pin 11     = !IOWR;   /*
Pin 13     = !IORD;   /*
Pin 14     = !AEN;    /*
Pin 15     = !BUSY;   /*
Pin 21     = ISA_MODE; /*
Pin 22     = AS0;     /*
Pin 23     = AS1;     /*

```

```

/** Outputs **/

```

```

Pin 16     = IORD_INV; /*
Pin 17     = !REPROG;  /*
Pin 18     = !CS_FPGA; /*
Pin 19     = !CS_ISA;  /*
Pin 20     = !CS_CONTR; /*

```

```

/** Declarations and Intermediate Variable Definitions **/

```

```

FIELD IOADR = [A9..0]; /* PC IO Space */
FIELD C_ADR = [A4..0]; /* Controller IO Space */
FIELD ADR_SEL = [AS1..AS0];

```

```

REPG0 = ADR_SEL : [0] & AEN & IOADR : [15F] ; /* Addr 140 */
ADDR0 = ADR_SEL : [0] & AEN & IOADR : [140..15F]; /* Addr 140..15F */
REPG1 = ADR_SEL : [1] & AEN & IOADR : [21F] ; /* Addr 200 */
ADDR1 = ADR_SEL : [1] & AEN & IOADR : [200..21F]; /* Addr 200..21F */
REPG2 = ADR_SEL : [2] & AEN & IOADR : [25F] ; /* Addr 240 */
ADDR2 = ADR_SEL : [2] & AEN & IOADR : [240..25F]; /* Addr 240..25F */
REPG3 = ADR_SEL : [3] & AEN & IOADR : [31F] ; /* Addr 300 */
ADDR3 = ADR_SEL : [3] & AEN & IOADR : [300..31F]; /* Addr 300..31F */
REPGC = AEN & IOADR : [1F] ; /* Topmost Addr */

```

```
/** Logic Equations **/  
REPROG  = ISA_MODE & IOWR & (REPG0 # REPG1 # REPG2 # REPG3) #  
         !ISA_MODE & IOWR & REPGC;  
  
CS_ISA   = ISA_MODE & (!BUSY & IOWR # IORD) & (ADDR0 # ADDR1 # ADDR2 # ADDR3);  
CS_CONTR = !ISA_MODE & (!BUSY & IOWR # IORD) & AEN;  
  
CS_FPGA  = ISA_MODE & (ADDR0 # ADDR1 # ADDR2 # ADDR3) #  
         !ISA_MODE & AEN;  
  
IORD_INV = IORD;
```

APPENDIX C: VHDL LISTINGS

Root

```

-----
-- File:      C:\DOCS\XILINX\FR_GR\FRGR_TOP.VHD
-- created:   10/04/97
-- by:        Stefan Weidmann
-- Company:   The University of Tennessee
-----

-- Revision history
-- Rev.      Date      Author      Description
-----
-- 0.0       10/04/97   Stefan Weidmann   Creation
-- 1.0       10/14/97   S.W.             Frame grabber project split from 'flash'
-----

library IEEE;
use IEEE.std_logic_1164.all;

-- SYNOPSYS library declaration
library SYNOPSYS;
use SYNOPSYS.std_logic_arith.all;
use SYNOPSYS.std_logic_unsigned.all;

library METAMOR;
use METAMOR.ATTRIBUTES.all;

-- user definitions
library FRGR_LIB;
use FRGR_LIB.DEFINITIONS.all;

entity FR_GR_TOP is
  port (m_clk:          in STD_LOGIC;
        pll_clk:       in STD_LOGIC;
        isa_clk:       in STD_LOGIC;
        reset:         in STD_LOGIC;
        cs_fpga_n:     in STD_LOGIC;
        isa_wr_n:      in STD_LOGIC;
        isa_rd_n:      in STD_LOGIC;
        isa_mode:      in STD_LOGIC;
        isa_a:         in STD_LOGIC_VECTOR (5 downto 0);
        isa_d:         inout STD_LOGIC_VECTOR (15 downto 0);
        d0s:          inout STD_LOGIC;
        sw:            in STD_LOGIC_VECTOR (1 downto 0);
        ld:           inout STD_LOGIC_VECTOR (7 downto 0);
        c_sync:        in STD_LOGIC;
        v_sync:        in STD_LOGIC;
        odd_even:      in STD_LOGIC;
        flash_trg:     buffer STD_LOGIC;
        irq10:         out STD_LOGIC;
        iocsl6_n:      out STD_LOGIC;
        a_lut_wr_n:    out STD_LOGIC;
        d_lut_wr_n:    out STD_LOGIC;
        lut_oe_n:      out STD_LOGIC;
        a_lut_oe_n:    out STD_LOGIC;
  );

```

```

        adc_oe_n:      out STD_LOGIC;
        adc_clk:       out STD_LOGIC;
        dc_rst:        out STD_LOGIC;
        ram_wr_n:      out STD_LOGIC;
        ram_rd_n:      buffer STD_LOGIC;
        ram_a:         buffer STD_LOGIC_VECTOR (16 downto 0);
        pll_fb:        buffer STD_LOGIC;
        busy_n:         buffer STD_LOGIC;
        slave_sel:     out STD_LOGIC;
        spi_d:          out STD_LOGIC;
        spi_clk:        buffer STD_LOGIC
    );

end;

ARCHITECTURE STRUCTURE OF FR_GR_TOP IS

-- COMPONENTS
component STARTUP
    port (GR: in std_logic);
end component;

component Host_intf
    port (cs_fpga_n:      in STD_LOGIC;
          isa_wr_n:       in STD_LOGIC;
          isa_rd_n:       in STD_LOGIC;
          isa_clk:        in STD_LOGIC;
          -- isa_mode:     in STD_LOGIC;
          busy:           in STD_LOGIC;
          isa_a:          in STD_LOGIC_VECTOR (5 downto 0);
          isa_d:          inout STD_LOGIC_VECTOR (15 downto 0);
          d0s:            inout STD_LOGIC;
          ld:             inout STD_LOGIC_VECTOR (7 downto 0);
          adc_d_reg:      in STD_LOGIC_VECTOR (15 downto 0);
          l_fr_reg:       in STD_LOGIC_VECTOR (7 downto 0);
          h_fr_reg:       in STD_LOGIC_VECTOR (7 downto 0);
          RAM_a16:        in STD_LOGIC;
          iocsl6_n:       out STD_LOGIC;
          irq:            out STD_LOGIC;
          d_lut_wr_en:    out STD_LOGIC;
          d_lut_wr_acc:   buffer STD_LOGIC;
          a_lut_wr_n:     out STD_LOGIC;
          d_lut_dc:       buffer STD_LOGIC;
          ram_wr_isa:     out STD_LOGIC;
          ram_rd_n:       buffer STD_LOGIC;
          res_mem_a:      out STD_LOGIC;
          RAM_rw_inc:     buffer STD_LOGIC;
          spi_start:      out STD_LOGIC;
          acq_start:      out STD_LOGIC;
          wp0_reg:        buffer STD_LOGIC_VECTOR (7 downto 0);
          wpl_reg:        buffer STD_LOGIC_VECTOR (7 downto 0);
          ctrl_reg:       buffer STD_LOGIC_VECTOR (3 downto 1);
          -- h_sp_reg:     buffer STD_LOGIC_VECTOR (7 downto 0);
          -- v_sp_reg:     buffer STD_LOGIC_VECTOR (7 downto 0);
          h_px_reg:       buffer STD_LOGIC_VECTOR (7 downto 0);
          v_px_reg:       buffer STD_LOGIC_VECTOR (7 downto 0);
          h_off_reg:      buffer STD_LOGIC_VECTOR (7 downto 0);
          v_off_reg:      buffer STD_LOGIC_VECTOR (7 downto 0);
          flpos_reg:      buffer STD_LOGIC_VECTOR (7 downto 0);
          fillen_reg:     buffer STD_LOGIC_VECTOR (7 downto 0);
          pll_reg:        buffer STD_LOGIC_VECTOR (7 downto 0);
          Pulse_Presc:    buffer STD_LOGIC_VECTOR (3 downto 0);
          Start_Presc:    buffer STD_LOGIC_VECTOR (3 downto 0)
    );
end component;

component spi_ctrl
    port (isa_clk: in STD_LOGIC;
          start: in STD_LOGIC;
          wp0: in STD_LOGIC_VECTOR (7 downto 0);

```

```

        wpl: in STD_LOGIC_VECTOR (7 downto 0);
        wp_sel: in STD_LOGIC;
        sel: out STD_LOGIC;
        spi_d: out STD_LOGIC;
        spi_clk: buffer STD_LOGIC);
end component;

component fl_ctrl
  port (Acq_start: in STD_LOGIC;
        Clk: in STD_LOGIC;
        Fl_len: in STD_LOGIC_VECTOR (7 downto 0);
        Fl_pos: in STD_LOGIC_VECTOR (7 downto 0);
        Odd_Even: in STD_LOGIC;
        oe_ctrl: in STD_LOGIC;
        Pulse_Presc: in STD_LOGIC_VECTOR (3 downto 0);
        Start_Presc: in STD_LOGIC_VECTOR (3 downto 0);
        V_Sync: in STD_LOGIC;
        Flash_trg: buffer STD_LOGIC);
end component;

component pll_sm
  port (PLL_CLK: in STD_LOGIC;
        PLL_reg: in STD_LOGIC_VECTOR (7 downto 0);
        PLL_fb: buffer STD_LOGIC);
end component;

component Acq
  port (acq_clk:      in STD_LOGIC;
        isa_clk:      in STD_LOGIC;
        acq_start:    in STD_LOGIC;
        adc_d_in:      in STD_LOGIC_VECTOR (7 downto 0);
        h_offs:        in STD_LOGIC_VECTOR (7 downto 0);
        h_px:          in STD_LOGIC_VECTOR (7 downto 0);
        h_sync:        in STD_LOGIC;
        v_sync:        in STD_LOGIC;
        odd_even:      in STD_LOGIC;
        v_offs:        in STD_LOGIC_VECTOR (7 downto 0);
        v_px:          in STD_LOGIC_VECTOR (7 downto 0);
        RAM_usr_wr:    in STD_LOGIC;
        RAM_rw_inc:    in STD_LOGIC;
        RAM_usr_rst:   in STD_LOGIC;
        RAM_wr_n:      out STD_LOGIC;
        adc_clk:       out STD_LOGIC;
        busy:          buffer STD_LOGIC;
        ram_a:         buffer STD_LOGIC_VECTOR (16 downto 0);
        adc_d_reg:     out STD_LOGIC_VECTOR (15 downto 0);
        dc_rst:        out STD_LOGIC);
end component;

-- SIGNALS

--signal ram_clk :      STD_LOGIC;
signal ctrl_reg :      STD_LOGIC_VECTOR (3 downto 1);
--signal h_sp_reg :    STD_LOGIC_VECTOR (7 downto 0);
--signal v_sp_reg :    STD_LOGIC_VECTOR (7 downto 0);
signal h_px_reg :      STD_LOGIC_VECTOR (7 downto 0);
signal v_px_reg :      STD_LOGIC_VECTOR (7 downto 0);
signal h_off_reg:      STD_LOGIC_VECTOR (7 downto 0);
signal v_off_reg:      STD_LOGIC_VECTOR (7 downto 0);
signal flpos_reg:      STD_LOGIC_VECTOR (7 downto 0);
signal fllen_reg:      STD_LOGIC_VECTOR (7 downto 0);
signal pll_reg :      STD_LOGIC_VECTOR (7 downto 0);
signal adc_d_reg:      STD_LOGIC_VECTOR (15 downto 0);
signal spi_start:      STD_LOGIC;
```

```

signal acq_start:      STD_LOGIC;
signal ram_wr_isa:     STD_LOGIC;
signal res_mem_a:      STD_LOGIC;
signal d_lut_wr_en:    STD_LOGIC;
signal d_lut_wr_acc:   STD_LOGIC;
signal d_lut_dc:       STD_LOGIC;
signal RAM_rw_inc:     STD_LOGIC;
signal wp0:            STD_LOGIC_VECTOR (7 downto 0);
signal wpl:            STD_LOGIC_VECTOR (7 downto 0);
signal busy:           STD_LOGIC;
signal Start_presc:    STD_LOGIC_VECTOR (3 downto 0);
signal Pulse_presc:    STD_LOGIC_VECTOR (3 downto 0);

BEGIN

    busy_n      <= not busy;
    d_lut_wr_n  <= not d_lut_wr_en;
    lut_oe_n    <= d_lut_wr_acc;
    adc_oe_n    <= d_lut_dc;
    a_lut_oe_n  <= not d_lut_dc;
-- ram_clk     <= pll_clk when busy = '1' else isa_clk;

    U1: STARTUP port map (GR=>reset);

    U2: Host_intf port map
        (cs_fpga_n => cs_fpga_n,
         isa_wr_n  => isa_wr_n,
         isa_rd_n  => isa_rd_n,
         isa_clk   => isa_clk,
--         isa_mode => isa_mode,
         busy      => busy,
         isa_a     => isa_a,
         isa_d     => isa_d,
         d0s       => d0s,
         ld        => ld,
         adc_d_reg => adc_d_reg,
         l_fr_reg  => ram_a(7 downto 0),
         h_fr_reg  => ram_a(15 downto 8),
         RAM_a16   => ram_a(16),
         iocsl6_n  => iocsl6_n,
         irq       => irq10,
         d_lut_wr_en => d_lut_wr_en,
         d_lut_wr_acc => d_lut_wr_acc,
         d_lut_dc  => d_lut_dc,
         a_lut_wr_n => a_lut_wr_n,
         ram_wr_isa => ram_wr_isa,
         ram_rd_n  => ram_rd_n,
         res_mem_a => res_mem_a,
         RAM_rw_inc => RAM_rw_inc,
         spi_start => spi_start,
         acq_start => acq_start,
         wp0_reg   => wp0,
         wpl_reg   => wpl,
         ctrl_reg  => ctrl_reg,
--         h_sp_reg  => h_sp_reg,
--         v_sp_reg  => v_sp_reg,
         h_px_reg  => h_px_reg,
         v_px_reg  => v_px_reg,
         h_off_reg => h_off_reg,
         v_off_reg => v_off_reg,
         flpos_reg => flpos_reg,
         flen_reg  => flen_reg,
         pll_reg   => pll_reg,
         Pulse_presc => Pulse_presc,
         Start_presc => Start_presc);

    U3: spi_ctrl port map

```

```

(isa_clk      => isa_clk,
 start       => spi_start,
 wp0         => wp0,
 wp1         => wp1,
 wp_sel      => ctrl_reg(1),
 sel         => slave_sel,
 spi_d       => spi_d,
 spi_clk     => spi_clk);

U4: fl_ctrl port map
(Acq_start   => acq_start,
 Clk         => isa_clk,
 Fl_len      => flen_reg,
 Fl_pos      => flpos_reg,
 Odd_Even    => odd_even,
 oe_ctrl     => ctrl_reg(3),
 Pulse_Presc => Pulse_presc,
 Start_Presc => Start_presc,
 V_Sync      => v_sync,
 Flash_trg   => flash_trg);

U5: pll_sm port map
(PLL_CLK     => PLL_CLK,
 PLL_reg     => PLL_reg,
 PLL_fb      => PLL_fb);

U6: Acq port map
(acq_clk     => pll_clk,
 isa_clk     => isa_clk,
 acq_start   => flash_trg,
 adc_d_in    => ld,
 h_offs      => h_off_reg,
 h_px        => h_px_reg,
 h_sync      => c_sync,
 v_sync      => v_sync,
 odd_even    => odd_even,
 v_offs      => v_off_reg,
 v_px        => v_px_reg,
 RAM_usr_wr  => ram_wr_isa,
 RAM_rw_inc  => RAM_rw_inc,
 RAM_usr_rst => res_mem_a,
 RAM_wr_n    => RAM_wr_n,
 adc_clk     => adc_clk,
 busy        => busy,
 ram_a       => ram_a,
 adc_d_reg   => adc_d_reg,
 dc_rst      => dc_rst);

END STRUCTURE;
```

HOST INTERFACE

```

-----
-- File:      C:\DOCS\XILINX\FR_GR\HOST.VHD
-- created:   09/30/97
-- by:        Stefan Weidmann
-- Company:   The University of Tennessee
-----

-- Revision history
-- Rev.      Date      Author      Description
-----
-- 0.0       09/30/97   Stefan Weidmann   Creation
-- 1.0       10/14/97   S.W.      Frame grabber project split from 'flash'
-- 1.1       10/19/97   S.W.      Replaced comb. output multiplexer w/ 3-state
-----

library IEEE;
use IEEE.std_logic_1164.all;

-- SYNOPSIS library declaration
library SYNOPSIS;
use SYNOPSIS.std_logic_arith.all;
use SYNOPSIS.std_logic_unsigned.all;

library METAMOR;
use METAMOR.ATTRIBUTES.all;

-- user definitions
library FRGR_LIB;
use FRGR_LIB.DEFINITIONS.all;

entity Host_intf is
  port (cs_fpga_n:      in STD_LOGIC;
        isa_wr_n:      in STD_LOGIC;
        isa_rd_n:      in STD_LOGIC;
        isa_clk:       in STD_LOGIC;
--        isa_mode:      in STD_LOGIC;
        busy:          in STD_LOGIC;
        isa_a:         in STD_LOGIC_VECTOR (5 downto 0);
        isa_d:         inout STD_LOGIC_VECTOR (15 downto 0);
        d0s:           inout STD_LOGIC;
        ld:            inout STD_LOGIC_VECTOR (7 downto 0);
        adc_d_reg:     in STD_LOGIC_VECTOR (15 downto 0);
        l_fr_reg:      in STD_LOGIC_VECTOR (7 downto 0);
        h_fr_reg:      in STD_LOGIC_VECTOR (7 downto 0);
        RAM_al6:       in STD_LOGIC;
        iocs16_n:      out STD_LOGIC;
        irq:           out STD_LOGIC;
        a_lut_wr_n:    out STD_LOGIC;
        d_lut_wr_en:   out STD_LOGIC;
        d_lut_wr_acc:  buffer STD_LOGIC;
        d_lut_dc:      buffer STD_LOGIC;
        ram_wr_isa:    out STD_LOGIC;
        ram_rd_n:      buffer STD_LOGIC;
        res_mem_a:     out STD_LOGIC;
        RAM_rw_inc:    buffer STD_LOGIC;
        spi_start:     out STD_LOGIC;
        acq_start:     out STD_LOGIC;
        wp0_reg:       buffer STD_LOGIC_VECTOR (7 downto 0);
        wp1_reg:       buffer STD_LOGIC_VECTOR (7 downto 0);
        ctrl_reg:      buffer STD_LOGIC_VECTOR (3 downto 1);
--        h_sp_reg:      buffer STD_LOGIC_VECTOR (7 downto 0);
--        v_sp_reg:      buffer STD_LOGIC_VECTOR (7 downto 0);
        h_px_reg:      buffer STD_LOGIC_VECTOR (7 downto 0);
  );
end entity Host_intf;

```

```

        v_px_reg:      buffer STD_LOGIC_VECTOR (7 downto 0);
        h_off_reg:     buffer STD_LOGIC_VECTOR (7 downto 0);
        v_off_reg:     buffer STD_LOGIC_VECTOR (7 downto 0);
        flpos_reg:     buffer STD_LOGIC_VECTOR (7 downto 0);
        fillen_reg:    buffer STD_LOGIC_VECTOR (7 downto 0);
        pll_reg:       buffer STD_LOGIC_VECTOR (7 downto 0);
        Pulse_Presc:   buffer STD_LOGIC_VECTOR (3 downto 0);
        Start_Presc:   buffer STD_LOGIC_VECTOR (3 downto 0);
    );

end;

architecture rtl of Host_intf is

    signal wr_acc:      STD_LOGIC;
    signal wr_acc_q:    STD_LOGIC;
    signal wr_acc_dt:   STD_LOGIC;
    signal chip_oe:     STD_LOGIC;
    signal i_isa_do:    STD_LOGIC_VECTOR (7 downto 0);
    signal i_d0s:       STD_LOGIC;
    signal wp0_reg_dc:  STD_LOGIC;
    signal wpl_reg_dc:  STD_LOGIC;
    signal ctrl_reg_dc: STD_LOGIC;
    signal ver_dc:      STD_LOGIC;
    signal l_fr_reg_dc: STD_LOGIC;
    signal h_fr_reg_dc: STD_LOGIC;
    --signal h_sp_reg_dc: STD_LOGIC;
    --signal v_sp_reg_dc: STD_LOGIC;
    signal h_px_reg_dc: STD_LOGIC;
    signal v_px_reg_dc: STD_LOGIC;
    signal h_off_reg_dc: STD_LOGIC;
    signal v_off_reg_dc: STD_LOGIC;
    signal flpos_reg_dc: STD_LOGIC;
    signal fillen_reg_dc: STD_LOGIC;
    signal pll_reg_dc:  STD_LOGIC;
    signal wp0_reg_en:  STD_LOGIC;
    signal wpl_reg_en:  STD_LOGIC;
    signal ctrl_reg_en: STD_LOGIC;
    --signal h_sp_reg_en: STD_LOGIC;
    --signal v_sp_reg_en: STD_LOGIC;
    signal h_px_reg_en: STD_LOGIC;
    signal v_px_reg_en: STD_LOGIC;
    signal h_off_reg_en: STD_LOGIC;
    signal v_off_reg_en: STD_LOGIC;
    signal flpos_reg_en: STD_LOGIC;
    signal fillen_reg_en: STD_LOGIC;
    signal pll_reg_en:  STD_LOGIC;
    signal acc_16bit:   STD_LOGIC;
    signal ram_rd_en:   STD_LOGIC;
    signal RAM_wr_dc:   STD_LOGIC;
    signal ris_wr_q:    STD_LOGIC;
    signal fal_wr_q:    STD_LOGIC;
    signal Presc_dc:    STD_LOGIC;
    signal Presc_wr_en: STD_LOGIC;
    signal lut_a_wr_dc: STD_LOGIC;
    signal ris_lut_a_q: STD_LOGIC;
    signal fal_lut_a_q: STD_LOGIC;
    signal d0_oe:       STD_LOGIC;
    signal d1_15_oe:    STD_LOGIC;
    signal d0s_oe:      STD_LOGIC;
    signal irq_asst:    STD_LOGIC;

begin

    -----
    -- generating write enable signals for registers
    -- Note: all internal registers are accessed 8 bits wide (read and write)
    -----
    wr_acc_en: process (cs_fpga_n, isa_wr_n, busy, wr_acc, wr_acc_q, wr_acc_dt,

```

```

        isa_clk, isa_a, isa_d, d_lut_dc, ctrl_reg_dc)

begin
    -- register write control PC side,

    if ((cs_fpga_n = '0') and (isa_wr_n = '0') and (busy = '0')) then
        wr_acc    <= '1';
    else
        wr_acc    <= '0';
    end if;

    wp0_reg_dc    <= '0';
    wpl_reg_dc    <= '0';
    ctrl_reg_dc    <= '0';
    ver_dc        <= '0';
    l_fr_reg_dc    <= '0';
    h_fr_reg_dc    <= '0';
    -- h_sp_reg_dc    <= '0';
    -- v_sp_reg_dc    <= '0';
    h_px_reg_dc    <= '0';
    v_px_reg_dc    <= '0';
    h_off_reg_dc    <= '0';
    v_off_reg_dc    <= '0';
    d_lut_dc       <= '0';
    flpos_reg_dc    <= '0';
    fllen_reg_dc    <= '0';
    pll_reg_dc     <= '0';
    Presc_dc       <= '0';
    if (busy = '0') and (cs_fpga_n = '0') then -- to avoid internal bus contention
        case isa_a is
            when wp0_addr    => wp0_reg_dc    <= '1';
            when wpl_addr    => wpl_reg_dc    <= '1';
            when ctrl_addr   => ctrl_reg_dc    <= '1';
            when ver_addr    => ver_dc        <= '1';
            when l_fr_addr   => l_fr_reg_dc    <= '1';
            when h_fr_addr   => h_fr_reg_dc    <= '1';

            --          when h_sp_addr    => h_sp_reg_dc    <= '1';
            --          when v_sp_addr    => v_sp_reg_dc    <= '1';
            when h_px_addr   => h_px_reg_dc    <= '1';
            when v_px_addr   => v_px_reg_dc    <= '1';
            when h_off_addr  => h_off_reg_dc    <= '1';
            when v_off_addr  => v_off_reg_dc    <= '1';
            when d_lut_addr  => d_lut_dc       <= '1';
            when flpos_addr  => flpos_reg_dc    <= '1';
            when fllen_addr  => fllen_reg_dc    <= '1';
            when pll_addr    => pll_reg_dc     <= '1';
            when Presc_addr  => Presc_dc       <= '1';
            when others => null;
        end case;
    end if;

    if isa_clk'event and isa_clk = '1' then
        wr_acc_q    <= wr_acc;
        wr_acc_dt    <= wr_acc_q;
        if (wr_acc_q = '1') and (wr_acc_dt = '0') then
            wp0_reg_en    <= wp0_reg_dc ;
            wpl_reg_en    <= wpl_reg_dc ;
            ctrl_reg_en    <= ctrl_reg_dc ;
            h_sp_reg_en    <= h_sp_reg_dc ;
            --          v_sp_reg_en    <= v_sp_reg_dc ;
            h_px_reg_en    <= h_px_reg_dc ;
            v_px_reg_en    <= v_px_reg_dc ;
            h_off_reg_en    <= h_off_reg_dc;
            v_off_reg_en    <= v_off_reg_dc;
            d_lut_wr_en    <= d_lut_dc ;
            flpos_reg_en    <= flpos_reg_dc;
            fllen_reg_en    <= fllen_reg_dc;
            pll_reg_en     <= pll_reg_dc ;

```

```

        Presc_wr_en    <= Presc_dc    ;
    else
        wp0_reg_en    <= '0';
        wpl_reg_en    <= '0';
        ctrl_reg_en    <= '0';
--        h_sp_reg_en    <= '0';
--        v_sp_reg_en    <= '0';
        h_px_reg_en    <= '0';
        v_px_reg_en    <= '0';
        h_off_reg_en    <= '0';
        v_off_reg_en    <= '0';
        d_lut_wr_en    <= '0';
        flpos_reg_en    <= '0';
        fillen_reg_en    <= '0';
        pll_reg_en    <= '0';
        Presc_wr_en    <= '0';

        end if;
    end if;

-----
-- Some signals for the frame memory address generation (different clock)
-----

    if (wr_acc = '1') and (ctrl_reg_dc = '1') and (isa_d(5) = '1') then
        res_mem_a    <= '1';
    else
        res_mem_a    <= '0';
    end if;

-----
-- Signal for the LUT memory write access
-----

    if (wr_acc = '1') and (d_lut_dc = '1') then
        d_lut_wr_acc    <= '1';
    else
        d_lut_wr_acc    <= '0';
    end if;

end process wr_acc_en;

-----
-- write data to PC side registers
-----

reg_wr: process (isa_clk, isa_d, wp0_reg_en, wpl_reg_en, ctrl_reg_en)
begin
    if isa_clk'event and isa_clk = '1' then
        if wp0_reg_en    = '1' then wp0_reg    <= isa_d(7 downto 0); end if;
        if wpl_reg_en    = '1' then wpl_reg    <= isa_d(7 downto 0); end if;
        if ctrl_reg_en    = '1' then
            ctrl_reg    <= isa_d(3 downto 1);
            -- d4 is RAM A16 and located in the Acquisition SM
            -- d5 is RAM user reset and located in the Acquisition SM
            acq_start    <= isa_d(6);
            spi_start    <= isa_d(7);
        else
            acq_start    <= '0'; -- these signals are asserted
            spi_start    <= '0'; -- for only 1 clock cycle
        end if;
--        if h_sp_reg_en    = '1' then h_sp_reg    <= isa_d; end if;
--        if v_sp_reg_en    = '1' then v_sp_reg    <= isa_d; end if;
        if h_px_reg_en    = '1' then h_px_reg    <= isa_d(7 downto 0); end if;
    end if;
end process reg_wr;

```

```

        if v_px_reg_en = '1' then v_px_reg <= isa_d(7 downto 0); end if;
        if h_off_reg_en = '1' then h_off_reg <= isa_d(7 downto 0); end if;
        if v_off_reg_en = '1' then v_off_reg <= isa_d(7 downto 0); end if;
        if flpos_reg_en = '1' then flpos_reg <= isa_d(7 downto 0); end if;
        if fllen_reg_en = '1' then fllen_reg <= isa_d(7 downto 0); end if;
        if pll_reg_en = '1' then pll_reg <= isa_d(7 downto 0); end if;
        if Presc_wr_en = '1' then Pulse_presc <= isa_d(7 downto 4);
                                Start_Presc <= isa_d(3 downto 0); end if;
    end if;
end process reg_wr;

-----
-- Generate WR pulse for Frame RAM
-- and WR pulse for LUT RAM
-----
ram_wr_gen: process (cs_fpga_n, isa_wr_n, isa_rd_n, busy, isa_clk, isa_a,
                    RAM_wr_dc, ris_wr_q, fal_wr_q, lut_a_wr_dc,
                    fal_lut_a_q, wr_acc)

begin
    -- generate RAM read or write access signal for RAM address increment

    RAM_rw_inc <= '0';
    RAM_wr_dc <= '0';
    RAM_rd_n <= '1'; -- neg. active !!

    if ((cs_fpga_n = '0') and (busy = '0') and
        (isa_a(5 downto 1) = ram_addr(5 downto 1))) then
        if ((isa_wr_n = '0') or (isa_rd_n = '0')) then RAM_rw_inc <= '1'; end if;
        if (isa_rd_n = '0') then RAM_rd_n <= '0'; end if; -- neg.
active !!
        if (isa_wr_n = '0') then RAM_wr_dc <= '1'; end if;
    end if;

    -- same for LUT address
    if ((wr_acc = '1') and (isa_a(5 downto 1) = a_lut_addr(5 downto 1))) then
        lut_a_wr_dc <= '1';
    else
        lut_a_wr_dc <= '0';
    end if;

    if isa_clk'event and isa_clk = '1' then
        ris_wr_q <= RAM_wr_dc;
        ris_lut_a_q <= lut_a_wr_dc;
    end if;

    if isa_clk'event and isa_clk = '0' then
        -- falling edge
        fal_wr_q <= ris_wr_q;
        fal_lut_a_q <= ris_lut_a_q;
    end if;

    -- start wr pulse as soon as possible and let it end with a falling
    -- edge of the clock
    if ((RAM_wr_dc = '1') and (fal_wr_q = '0')) then
        ram_wr_isa <= '1';
    else
        ram_wr_isa <= '0';
    end if;

    -- same for LUT address register
    if ((lut_a_wr_dc = '1') and (fal_lut_a_q = '0')) then
        a_lut_wr_n <= '0'; -- neg. active!
    else
        a_lut_wr_n <= '1';
    end if;

```

```

end process ram_wr_gen;

-----
-- Data output
-----
rd_out: process (ctrl_reg, isa_rd_n, busy, cs_fpga_n, isa_a, RAM_rw_inc,
                RAM_wr_dc, RAM_rd_n)

begin
    if isa_rd_n = '0' then
        d0_oe <= '1';
    else
        d0_oe <= '0';
    end if;

    if ((busy = '1') or ((isa_rd_n = '0') and (RAM_rd_n = '1'))) then
        d1_15_oe <= '1';
    else
        d1_15_oe <= '0';
    end if;

    if ((busy = '1') or (RAM_wr_dc = '1')) then
        d0s_oe <= '1';
    else
        d0s_oe <= '0';
    end if;

end process rd_out;

-- output multiplexer
i_isa_do <= wp0_reg when (wp0_reg_dc = '1') else (others => 'Z');
i_isa_do <= wp1_reg when (wp1_reg_dc = '1') else (others => 'Z');
i_isa_do <= "000" & RAM_al6 & ctrl_reg & busy when (ctrl_reg_dc = '1') else (others => 'Z');
i_isa_do <= version_no when (ver_dc = '1') else (others => 'Z');
i_isa_do <= l_fr_reg when (l_fr_reg_dc = '1') else (others => 'Z');
i_isa_do <= h_fr_reg when (h_fr_reg_dc = '1') else (others => 'Z');
i_isa_do <= ld when (d_lut_dc = '1') else (others => 'Z');
i_isa_do <= adc_d_reg (7 downto 1) & busy when (busy = '1') else (others => 'Z');
i_isa_do(0) <= d0s when (RAM_rw_inc = '1') else 'Z';

-- special case D0S
i_d0s <= isa_d(0) when (busy = '0') else 'Z';
i_d0s <= adc_d_reg(0) when (busy = '1') else 'Z';

-- i_isa_do <= h_sp_reg when (isa_a = h_sp_addr) else (others => 'Z');
-- i_isa_do <= v_sp_reg when (isa_a = v_sp_addr) else (others => 'Z');
-- i_isa_do <= h_px_reg when (isa_a = h_px_addr) else (others => 'Z');
-- i_isa_do <= v_px_reg when (isa_a = v_px_addr) else (others => 'Z');
-- i_isa_do <= h_off_reg when (isa_a = h_off_addr) else (others => 'Z');
-- i_isa_do <= v_off_reg when (isa_a = v_off_addr) else (others => 'Z');
-- i_isa_do <= flpos_reg when (isa_a = flpos_addr) else (others => 'Z');
-- i_isa_do <= flllen_reg when (isa_a = flllen_addr) else (others => 'Z');
-- i_isa_do <= pll_reg when (isa_a = pll_addr) else (others => 'Z');
-- i_isa_do <= Pulse_presc & Start_Presc when (isa_a = Presc_addr) else (others => 'Z');

-----
-- Generate 16 bit access signal if either the frame ram or the LUT
-- address latches are accessed. Don't generate the signal for
-- accesses on odd addresses (A0 = 1).
-----

acc_16bit_gen: process (isa_clk, cs_fpga_n, isa_wr_n, isa_rd_n, busy, isa_a)

begin
    if ((cs_fpga_n = '0') and (isa_a(0) = '0') and
        ((isa_wr_n = '0') or (isa_rd_n = '0')) and -- generate without RD or WR
        (isa_a(5 downto 1) = ram_addr(5 downto 1)) or

```

```

        (isa_a(5 downto 1) = a_lut_addr(5 downto 1))))
    then acc_16bit <= '1';
    else acc_16bit <= '0';
    end if;

    end process acc_16bit_gen;

-----
-- external tristate lines
-----
-- isa bus
    isa_d(0)          <= i_isa_do(0) when (d0_oe = '1')    else 'Z';
    isa_d(15 downto 1) <= (adc_d_reg (15 downto 8) & i_isa_do(7 downto 1))
                        when (d1_15_oe = '1') else (others => 'Z');
    d0s               <= i_d0s      when (d0s_oe = '1')    else 'Z';
    iocsl6_n          <= '0'        when (acc_16bit = '1') else 'Z';
    irq_asst          <= '0';
    irq               <= '0'        when (irq_asst = '1') else 'Z';

-- LUT data
    ld               <= isa_d(7 downto 0) when (d_lut_wr_acc = '1') else (others => 'Z');

end rtl;

```

FLASH TRIGGER

```

--
-- File: C:\DOCS\XILINX\Fr_Gr\F1_ctrl.vhd
-- created: 10/20/97 14:04:37
-- from: 'C:\DOCS\XILINX\Fr_Gr\F1_ctrl.asf'
-- by fsm2hdl - version: 2.0.1.11
--
library IEEE;
use IEEE.std_logic_1164.all;

-- SYNOPSYS library declaration
library SYNOPSYS;
use SYNOPSYS.std_logic_arith.all;
use SYNOPSYS.std_logic_unsigned.all;

library METAMOR;
use METAMOR.ATTRIBUTES.all;

entity fl_ctrl is
    port (Acq_start: in STD_LOGIC;
          Clk: in STD_LOGIC;
          Fl_len: in STD_LOGIC_VECTOR (7 downto 0);
          Fl_pos: in STD_LOGIC_VECTOR (7 downto 0);
          Odd_Even: in STD_LOGIC;
          oe_ctrl: in STD_LOGIC;
          Pulse_Presc: in STD_LOGIC_VECTOR (3 downto 0);
          Start_Presc: in STD_LOGIC_VECTOR (3 downto 0);
          V_Sync: in STD_LOGIC;
          Flash_trg: buffer STD_LOGIC);
end;

architecture fl_ctrl_arch of fl_ctrl is

--diagram signal declarations
    signal Delay_cnt: STD_LOGIC_VECTOR (7 downto 0);

```

```

signal Presc: STD_LOGIC_VECTOR (3 downto 0);
signal VS1: STD_LOGIC;
signal VS2: STD_LOGIC;

-- SYMBOLIC ENCODED state machine: Assign
type Assign_type is (S1);
signal Assign: Assign_type;

-- USER DEFINED ENCODED state machine: Flash_Ctrl
type Flash_Ctrl_type is (Cnt_zero, Delay, Fire, Fr_Start_Wait, Idle, Prescale, Prescale2);
attribute enum_encoding of Flash_Ctrl_type: type is
    "100 " & -- Cnt_zero
    "010 " & -- Delay
    "101 " & -- Fire
    "001 " & -- Fr_Start_Wait
    "000 " & -- Idle
    "011 " & -- Prescale
    "110"; -- Prescale2

signal Flash_Ctrl: Flash_Ctrl_type;

begin
--concurrent signal assignments
--diagram ACTIONS;

process (Clk)
begin
if Clk'event and Clk = '1' then
    case Assign is
        when S1 =>
            VS1 <= V_Sync;
            VS2 <= VS1;
        when others =>
            null;
    end case;
end if;
end process;

process (Clk)
begin
if Clk'event and Clk = '1' then
    case Flash_Ctrl is
        when Cnt_zero =>
            Delay_cnt <= (others => '0');
            Flash_Ctrl <= Fire;
        when Delay =>
            Delay_cnt <= Delay_cnt + '1';
            Presc <= (others => '0');
            if Delay_cnt = Fl_pos then
                Flash_Ctrl <= Cnt_zero;
            else
                Flash_Ctrl <= Prescale;
            end if;
        when Fire =>
            Flash_trg <= '1';
            Delay_cnt <= Delay_cnt + '1';
            Presc <= (others => '0');
            if Delay_cnt = Fl_len then
                Flash_Ctrl <= Idle;
            else
                Flash_Ctrl <= Prescale2;
            end if;
        when Fr_Start_Wait =>

```

```

        if (VS1 = '1') and (VS2 = '0') and
           (Odd_Even = oe_ctrl) then
            Flash_Ctrl <= Delay;
        end if;
    when Idle =>
        Flash_trg <= '0';
        Delay_cnt <= (others => '0');
        if Acq_start = '1' then
            Flash_Ctrl <= Fr_Start_Wait;
        end if;
    when Prescale =>
        Presc <= Presc + '1';
        if Presc = Start_Presc then
            Flash_Ctrl <= Delay;
        end if;
    when Prescale2 =>
        Presc <= Presc + '1';
        if Presc = Pulse_presc then
            Flash_Ctrl <= Fire;
        end if;
    when others => -- trap state
        Flash_Ctrl <= Idle;
    end case;
end if;
end process;

end fl_ctrl_arch;

```

ACQUISITION SUBSYSTEM

Acquisition main module

```

-----
-- File:      C:\DOCS\XILINK\FR_GR\ACQ.VHD
-- created:   10/17/97
-- by:        Stefan Weidmann
-- Company:   The University of Tennessee
-----

```

```

-- Revision history
-- Rev.    Date      Author          Description
-----
-- 0.0     10/17/97   Stefan Weidmann  Creation
-----

```

```

library IEEE;
use IEEE.std_logic_1164.all;

-- SYNOPSIS library declaration
library SYNOPSIS;
use SYNOPSIS.std_logic_arith.all;
use SYNOPSIS.std_logic_unsigned.all;

library METAMOR;
use METAMOR.ATTRIBUTES.all;

```

```

entity Acq is
  port (acq_clk:      in STD_LOGIC;
        isa_clk:      in STD_LOGIC;
        acq_start:    in STD_LOGIC;
        adc_d_in:      in STD_LOGIC_VECTOR (7 downto 0);
        h_offs:        in STD_LOGIC_VECTOR (7 downto 0);
        h_px:          in STD_LOGIC_VECTOR (7 downto 0);
        h_sync:        in STD_LOGIC;
        v_sync:        in STD_LOGIC;
        odd_even:      in STD_LOGIC;
        v_offs:        in STD_LOGIC_VECTOR (7 downto 0);
        v_px:          in STD_LOGIC_VECTOR (7 downto 0);
        RAM_usr_wr:    in STD_LOGIC;
        RAM_rw_inc:    in STD_LOGIC;
        RAM_usr_rst:   in STD_LOGIC;
        RAM_wr_n:      out STD_LOGIC;
        adc_clk:       out STD_LOGIC;
        busy:          buffer STD_LOGIC;
        ram_a:         buffer STD_LOGIC_VECTOR (16 downto 0);
        adc_d_reg:     out STD_LOGIC_VECTOR (15 downto 0);
        dc_rst:        out STD_LOGIC);

```

```
end;
```

```
architecture rtl of Acq is
```

```

  component acq_sm
    port (acq_clk:      in STD_LOGIC;
          acq_start:    in STD_LOGIC;
          d_in:         in STD_LOGIC_VECTOR (7 downto 0);
          h_offs:        in STD_LOGIC_VECTOR (7 downto 0);
          h_px:         in STD_LOGIC_VECTOR (7 downto 0);
          h_sync:        in STD_LOGIC;
          v_sync:        in STD_LOGIC;
          odd_even:      in STD_LOGIC;
          v_offs:        in STD_LOGIC_VECTOR (7 downto 0);
          v_px:         in STD_LOGIC_VECTOR (7 downto 0);
          busy:          buffer STD_LOGIC;
          d_reg:         out STD_LOGIC_VECTOR (15 downto 0);
          RAM_inc:       out STD_LOGIC;
          RAM_res:       out STD_LOGIC;
          RAM_wr:        out STD_LOGIC);
  end component;

```

```

signal RAM_inc:    STD_LOGIC;
signal RAM_res:    STD_LOGIC;
signal RAM_wr:     STD_LOGIC;
signal RAM_rw_inc_dt: STD_LOGIC;
signal RAM_rw_inc_q: STD_LOGIC;
--signal PLL_qp:    STD_LOGIC;
--signal PLL_qn:    STD_LOGIC;
signal clk_src:    STD_LOGIC;
signal ram_clk:    STD_LOGIC;

```

```
begin
```

```

  U1: acq_sm port map
    (acq_clk => acq_clk,
     acq_start => acq_start,
     d_in => adc_d_in,
     h_offs => h_offs,
     h_px => h_px,
     h_sync => h_sync,
     v_sync => v_sync,
     odd_even => odd_even,
     v_offs => v_offs,

```

```

v_px      => v_px,
busy      => busy,
d_reg     => adc_d_reg,
RAM_inc   => RAM_inc,
RAM_res   => RAM_res,
RAM_wr    => RAM_wr);

-----
-- RAM write access
-----
RAM_wr_out: process (RAM_wr, RAM_usr_wr, busy)
begin
    if (RAM_wr = '1') or (RAM_usr_wr = '1') then
        RAM_wr_n <= '0';
    else
        RAM_wr_n <= '1';
    end if;
end process RAM_wr_out;

-----
-- RAM address generation
-----
RAM_addr_gen: process (ram_clk, RAM_usr_rst, RAM_res, RAM_inc, RAM_rw_inc)
begin
    if ram_clk'event and ram_clk = '1' then
        RAM_rw_inc_q <= RAM_rw_inc;    -- synchronize to clock and wait for
        RAM_rw_inc_dt <= RAM_rw_inc_q; -- falling edge
        if (RAM_usr_rst = '1') or (RAM_res = '1') then
            ram_a <= (others => '0');
        elsif (RAM_inc = '1') or      -- this signal is already synchronized
              ((RAM_rw_inc_q = '0') and (RAM_rw_inc_dt = '1')) --falling edge
        then ram_a <= ram_a + '1';
        end if;
    end if;
end process RAM_addr_gen;

-----
-- Clk multiplexer
-----
ram_clk_sel: process (acq_clk, busy, isa_clk, acq_clk, clk_src)
begin
    if acq_clk'event and acq_clk = '1' then
        if ((busy = '1') and (isa_clk = '1')) then
            clk_src <= '1';
        end if;
        if ((busy = '0') and (isa_clk = '1')) then
            clk_src <= '0';
        end if;
    end if;

    if (clk_src = '1') then
        ram_clk <= acq_clk;
    else
        ram_clk <= isa_clk;
    end if;
    adc_clk <= acq_clk;
end process ram_clk_sel;

-----

```

```
-- DC restore generation: tri-state driver
```

```
-----
dc_rst <= '0' when h_sync = '0' else 'Z';

end rtl;
```

Acquisition Finite State Machine

```
--
-- File: c:\docs\xilinx\fr_gr\acq_sm.vhd
-- created: 11/03/97 21:56:59
-- from: 'c:\docs\xilinx\fr_gr\acq_sm.asf'
-- by fsm2hdl - version: 2.0.1.11
--
library IEEE;
use IEEE.std_logic_1164.all;

-- SYNOPSIS library declaration
library SYNOPSIS;
use SYNOPSIS.std_logic_arith.all;
use SYNOPSIS.std_logic_unsigned.all;

library METAMOR;
use METAMOR.ATTRIBUTES.all;

entity acq_sm is
  port (acq_clk: in STD_LOGIC;
        acq_start: in STD_LOGIC;
        d_in: in STD_LOGIC_VECTOR (7 downto 0);
        h_offs: in STD_LOGIC_VECTOR (7 downto 0);
        h_px: in STD_LOGIC_VECTOR (7 downto 0);
        h_sync: in STD_LOGIC;
        odd_even: in STD_LOGIC;
        v_offs: in STD_LOGIC_VECTOR (7 downto 0);
        v_px: in STD_LOGIC_VECTOR (7 downto 0);
        v_sync: in STD_LOGIC;
        d_reg: out STD_LOGIC_VECTOR (15 downto 0);
        RAM_inc: out STD_LOGIC;
        RAM_res: out STD_LOGIC;
        RAM_wr: out STD_LOGIC;
        busy: buffer STD_LOGIC);
end;

architecture acq_sm_arch of acq_sm is

  --diagram signal declarations
  signal HS1: STD_LOGIC;
  signal HS2: STD_LOGIC;
  signal Line_nr: STD_LOGIC_VECTOR (8 downto 0);
  signal Pix_nr: STD_LOGIC_VECTOR (8 downto 0);
  signal tmp_reg: STD_LOGIC_VECTOR (7 downto 0);
  signal VS1: STD_LOGIC;
  signal VS2: STD_LOGIC;

  -- USER DEFINED ENCODED state machine: acquisition
  type acquisition_type is (Acq_Ev, Acq_Odd, Idle, Line_inc, Next_field, Next_line, PrePix,
    Wait_H, Wait_H2, Wait_V);
  attribute enum_encoding of acquisition_type: type is
    "0100 " &      -- Acq_Ev
    "0101 " &      -- Acq_Odd
    "0000 " &      -- Idle
    "0010 " &      -- Line_inc
    "1001 " &      -- Next_field
```

```

"1010 " &      -- Next_line
"0011 " &      -- PrePix
"0001 " &      -- Wait_H
"0111 " &      -- Wait_H2
"0110";      -- Wait_V

signal acquisition: acquisition_type;

-- SYMBOLIC ENCODED state machine: Assign
type Assign_type is (S1);
signal Assign: Assign_type;

begin
--concurrent signal assignments
--diagram ACTIONS;

process (acq_clk)

begin

if acq_clk'event and acq_clk = '1' then
  case acquisition is
    when Acq_Ev =>
      d_reg <= d_in & tmp_reg;
      RAM_wr <= '1';
      RAM_inc <= '0';
      acquisition <= Acq_Odd;
    when Acq_Odd =>
      RAM_wr <= '0';
      RAM_inc <= '1';
      Pix_nr <= Pix_nr + '1';
      tmp_reg <= d_in;
      if Pix_nr(8 downto 1) = h_px then
        acquisition <= Next_line;
      else
        acquisition <= Acq_Ev;
      end if;
    when Idle =>
      RAM_wr <= '0';
      busy <= '0';
      RAM_inc <= '0';
      RAM_res <= '0';
      Line_nr <= (others => '0');
      Pix_nr <= (others => '0');
      if acq_start = '1' then
        acquisition <= Wait_V;
      end if;
    when Line_inc =>
      Line_nr <= Line_nr + '1';
      if Line_nr(7 downto 0) = v_offs then
        acquisition <= PrePix;
        Line_nr <= (0 => '1', others => '0');
      else
        acquisition <= Wait_H;
      end if;
    when Next_field =>
      Line_nr <= (others => '0');
      if VS1 = '1' and
        VS2 = '0' and
        odd_even = '0' then
        acquisition <= Wait_H;
      elsif VS1 = '1' and
        VS2 = '0' and
        odd_even = '1' then
        acquisition <= Idle;
      end if;
    when Next_line =>
      RAM_inc <= '0';
      Line_nr <= Line_nr + '1';
  end case;
end if;

```

```

        Pix_nr <= (others => '0');
        if Line_nr(8 downto 1) = v_px then
            acquisition <= Next_field;
        else
            acquisition <= Wait_H2;
        end if;
    when PrePix =>
        Pix_nr <= Pix_nr + '1';
        tmp_reg <= d_in;
        if Pix_nr(7 downto 0) = h_offs then
            acquisition <= Acq_Ev;
            Pix_nr <= (0 => '1', others => '0');
        end if;
    when Wait_H =>
        RAM_res <= '0';
        if HS1 = '1' and
            HS2 = '0' then
            acquisition <= Line_inc;
        end if;
    when Wait_H2 =>
        if HS1 = '1' and
            HS2 = '0' then
            acquisition <= PrePix;
        end if;
    when Wait_V =>
        busy <= '1';
        RAM_res <= '1';
        if VS1 = '1' and
            VS2 = '0' and
            odd_even = '1' then
            acquisition <= Wait_H;
        end if;
    when others => -- trap state
        acquisition <= Idle;
    end case;
end if;
end process;

```

```

process (acq_clk)
begin
    if acq_clk'event and acq_clk = '1' then
        case Assign is
            when S1 =>
                VS1 <= v_sync;
                VS2 <= VS1;
                HS1 <= h_sync;
                HS2 <= HS1;
            when others =>
                null;
            end case;
        end if;
    end process;

end acq_sm_arch;

```

SERIAL PERIPHERAL INTERFACE (SPI)

```

--
-- File: C:\DOCS\XILINX\FLASH\spi_ctrl.vhd
-- created: 10/27/97 10:19:26
-- from: 'C:\DOCS\XILINX\FLASH\spi_ctrl.asf'
-- by fsm2hdl - version: 2.0.1.11
--
library IEEE;
use IEEE.std_logic_1164.all;

-- SYNOPSYS library declaration
library SYNOPSYS;
use SYNOPSYS.std_logic_arith.all;
use SYNOPSYS.std_logic_unsigned.all;

library METAMOR;
use METAMOR.ATTRIBUTES.all;

entity spi_ctrl is
  port (isa_clk: in STD_LOGIC;
        start: in STD_LOGIC;
        wp0: in STD_LOGIC_VECTOR (7 downto 0);
        wp1: in STD_LOGIC_VECTOR (7 downto 0);
        wp_sel: in STD_LOGIC;
        sel: out STD_LOGIC;
        spi_d: out STD_LOGIC;
        spi_clk: buffer STD_LOGIC);
end;

architecture spi_ctrl_arch of spi_ctrl is

  --diagram signal declarations
  signal bit_cnt: STD_LOGIC_VECTOR (4 downto 0);
  signal sr: STD_LOGIC_VECTOR (15 downto 0);

  -- BINARY ENCODED state machine: SPI_GEN
  type SPI_GEN_type is (Ready, S1, S2, S3);
  attribute enum_encoding of SPI_GEN_type: type is
    "00 " & -- Ready
    "01 " & -- S1
    "10 " & -- S2
    "11"; -- S3

  signal SPI_GEN: SPI_GEN_type;

begin
  --concurrent signal assignments
  --diagram ACTIONS;

  process (isa_clk)
  begin
    if isa_clk'event and isa_clk = '1' then
      case SPI_GEN is
        when Ready =>
          sel <= '0';
          bit_cnt <= (others => '0');
          spi_clk <= '0';
          if start = '1' then
            SPI_GEN <= S1;
          end if;
        when S1 =>
          sel <= '1';
      end case;
    end if;
  end process;

```

```

        spi_d <= wp_sel;
        sr <= wp1 & wp0;
        SPI_GEN <= S2;
    when S2 =>
        spi_clk <= not (spi_clk);
        SPI_GEN <= S3;
    when S3 =>
        spi_clk <= not (spi_clk);
        bit_cnt <= bit_cnt + '1';
        spi_d <= sr(15);
        sr <= sr(14 downto 0) & '0';
        if (bit_cnt(4) = '1') then
            SPI_GEN <= Ready;
        else
            SPI_GEN <= S2;
        end if;
    when others => -- trap state
        SPI_GEN <= Ready;
end case;
end if;
end process;

end spi_ctrl_arch;

```

PHASE LOCKED LOOP (PLL)

```

--
-- File: C:\DOCS\Xilinx\Fr_Gr\PLL_SM.vhd
-- created: 10/14/97 12:39:01
-- from: 'C:\DOCS\Xilinx\Fr_Gr\PLL_SM.asf'
-- by fsm2hdl - version: 2.0.1.11
--
library IEEE;
use IEEE.std_logic_1164.all;

-- SYNOPSIS library declaration
library SYNOPSIS;
use SYNOPSIS.std_logic_arith.all;
use SYNOPSIS.std_logic_unsigned.all;

library METAMOR;
use METAMOR.ATTRIBUTES.all;

entity pll_sm is
    port (PLL_CLK: in STD_LOGIC;
          PLL_reg: in STD_LOGIC_VECTOR (7 downto 0);
          PLL_fb: buffer STD_LOGIC);
end;

architecture pll_sm_arch of pll_sm is

    --diagram signal declarations
    signal Cnt: STD_LOGIC_VECTOR (8 downto 0);

    -- SYMBOLIC ENCODED state machine: PLL
    type PLL_type is (Scale, Toggle);
    signal PLL: PLL_type;

begin
    --concurrent signal assignments
    --diagram ACTIONS;

```

```
process (PLL_CLK)
begin
if PLL_CLK'event and PLL_CLK = '1' then
  case PLL is
    when Scale =>
      Cnt <= Cnt + '1';
      if Cnt(8 downto 1) = PLL_reg then
        PLL <= Toggle;
      end if;
    when Toggle =>
      Cnt <= (others => '0');
      PLL_fb <= not(PLL_fb);
      PLL <= Scale;
    when others => -- trap state
      PLL <= Toggle;
  end case;
end if;
end process;

end pll_sm_arch;
```

APPENDIX D: C LISTING

```

/*****
* File:      C:\DOCS\TC\FRGR.C
* created:   10/08/97
* by:        Stefan Weidmann
* Company:   The University of Tennessee
* *****/
* Driver software for Frame Grabber Board
* *****/
* Revision history
* Rev.    Date      Author      Description
* *****/
* 1.0     10/08/97   Stefan Weidmann   Creation
* 1.1     10/09/97   S.W.      Filter for file header added
* 1.2     10/20/97   S.W.      Ser. interface for flash added
* 1.3     10/23/97   S.W.      Driver routines acquisition added
* 1.4     10/24/97   S.W.      RAM test added
* 1.5     10/29/97   S.W.      Frame read corrected
* 1.6     10/31/97   S.W.      LUT test added
* *****/
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>

char reverse(char ib);
int RAM_test(void);
int cfg_download(void);
void conf_reg(void);
void start_pot_wr(void);
void start_acq(void);
void RAM_addr_rst(void);
void LUT_load(void);
int LUT_test(void);
int wait_acq_start(void);
int wait_acq_end(void);
void make_tiff(void);

FILE *cfg_file;
char cfg_fname[20]      = "fr_gr.bit";
FILE *tiff_file;
char tiff_fname[20]     = "fr_gr.tif";
FILE *tmp_file;
char tmp_fname[20]      = "fr_gr.tmp";
unsigned char pll        = 161; /* (161+1)* 4 = 648 */
unsigned char wp0        = 128; /* wiper0 value */
unsigned char wp1        = 128; /* wiper1 value */
unsigned char wp_sel     = 0;   /* wiper select, 0 or 1 */
unsigned char od_ev_st   = 1;   /* 1 or 0 start on odd or even frame */
unsigned char presc      = 0xFF; /* Pulse and wait prescaler both 15 (2us) */
unsigned char fl_pos     = 125; /* Flash pulse position after 125*2us */
unsigned char fl_len     = 25;  /* Flash pulse length 25*2us */
unsigned char h_px       = 128; /* Horiz. # of pixels (4*128=512) */
unsigned char v_px       = 120; /* Vert. # of pixels (4*120=480) */
unsigned char h_off      = 4;   /* Horiz. pixel offset */
unsigned char v_off      = 4;   /* Vert. pixel offset */

```

```

int base_address          = 0x140;

/*NOTE: The following registers are 8 bits wide.
    Access with in/outportb */

int wp0_addr              = 0x0;
int wp1_addr              = 0x1;
int ctrl_addr             = 0x2;
/* Control reg. Bit assigment:
    0: Busy bit (rd only)
    1: Stack (wiper) select
    2: Frame memory A16
    3: Flash trigger odd or even
    4: unused
    5: Reset frame memory addr.
    6: Start frame acquisition
    7: Start potentiometer write */

int ver_addr              = 0x3;
int l_fr_addr             = 0x4;
int h_fr_addr             = 0x5;
int h_sp_addr             = 0x6;
int v_sp_addr             = 0x7;
int h_px_addr             = 0x8;
int v_px_addr             = 0x9;
int h_off_addr            = 0xA;
int v_off_addr            = 0xB;
int d_lut_addr            = 0xC;
int fl_pos_addr           = 0xD;
int fl_len_addr           = 0xE;
int pll_addr              = 0xF;
int presc_addr            = 0x10;
int prog_off              = 0x1F;

/*NOTE: The following registers are 16 bits wide.
    Access with in/outport */

int a_lut_addr            = 0x18;
int ram_addr              = 0x1A;

unsigned char io_byte      = 0;
unsigned char last_byte    = 0;
unsigned char testpattern1 = 0xA5;
unsigned char testpattern2 = 0x5A;
unsigned char preamb1      = 0xFF;
unsigned char preamb2      = 0xF2;

int cnt                   = 0;
int i, error, usr_abort;

/*****
void main(void)
/*****
{
    error = cfg_download();
    if (!error)
    {
        printf("Configuring registers...\n\n");
        conf_reg();

        delay(250); /* PLL settle */
        printf("RAM test...\n");
        if (RAM_test())
            printf("RAM test failed.\n");
        else
            printf("RAM test successful.\n");
    }
}

```

```

printf("LUT test...\n");
if (LUT_test())
    printf("LUT test failed.\n");
else
    printf("LUT test successful.\n");

printf("Configuring LUT (Look Up Table)...\n\n");
LUT_load();

printf("Press a key when ready to configure potentiometer -->");
getch();
start_pot_wr();
printf("\nPotentiometer configured.\n\n");

printf("Press a key when ready to flash -->");
getch();
start_acq();
printf("\nAcquisition triggered.\n\n");

usr_abort = wait_acq_start();
if (usr_abort == 0)
    usr_abort = wait_acq_end();
if (usr_abort == 0)
    make_tiff();
}
}

/*****/
char reverse(char ib)
/*****/
{
return (((ib&0x80) >> 7) | ((ib&0x40) >> 5) | ((ib&0x20) >> 3) | ((ib&0x10) >> 1) |
        ((ib&0x08) << 1) | ((ib&0x04) << 3) | ((ib&0x02) << 5) | ((ib&0x01) << 7));
}

/*****/
int cfg_download(void)
/*****/
{
    int error;

    cfg_file = fopen (cfg_fname, "rb");
    if (cfg_file == NULL)
    {
        printf("Download aborted. Configuration file '%s' not found.\n",
               cfg_fname);
        error = 1;
    }
    else
    {
        printf("\nXILINX configuration download\n");
        printf("=====\n\n");
        /* Write dummy data to start program mode */
        outportb(base_address + prog_off, io_byte);

        /* Filter the header out and look for pre-amble */
        while ((!feof(cfg_file)) &&
               ((io_byte != preamb2) || (last_byte != preamb1)))
        {
            last_byte = io_byte;
            io_byte = fgetc(cfg_file);
            cnt++;
        }
    }
}

```

```

printf("%i header bytes filtered.\n", cnt-2);
cnt = 0;

/* check if ready */
io_byte = inportb(base_address);
/* wait until ready */
while((io_byte & 0x80) == 0)
    io_byte = inportb(base_address);
outportb(base_address, reverse(preamb1));
io_byte = inportb(base_address);
/* wait until ready */
while((io_byte & 0x80) == 0)
    io_byte = inportb(base_address);
outportb(base_address, reverse(preamb2));

while (!feof(cfg_file))
{
    /* check if ready */
    io_byte = inportb(base_address);
    /* wait until ready */
    while((io_byte & 0x80) == 0)
        io_byte = inportb(base_address);

    io_byte = fgetc(cfg_file);
    outportb(base_address, reverse(io_byte));
    cnt++;
}

/* Write dummies to bring out of config state */
for (i = 0; i < 10; i++)
    outportb(base_address, 0xFF);

printf("%i bytes downloaded.\n", cnt+1); /* +2 preamble, -1 EOF */

io_byte = inportb(base_address + ver_addr);
printf("Version number: %d\n", io_byte);

printf("Checking register access...\n");
io_byte = testpattern1;

outportb(base_address, io_byte);
io_byte = testpattern2;
outportb(base_address + 1, io_byte);
io_byte = inportb(base_address);
if (io_byte != testpattern1)
{
    printf("Download FAILED. Test pattern 1 was %X instead of %X\n",
        io_byte, testpattern1);
    error = 1;
}
else
{
    io_byte = inportb(base_address + 1);
    if (io_byte != testpattern2)
    {
        printf("Download FAILED. Test pattern 2 was %X instead of %X\n",
            io_byte, testpattern2);
        error = 1;
    }
    else
    {
        printf("SUCCESS: Check and Download OK.\n");
        error = 0;
    }
}
}
fclose(cfg_file);
return error;
}

```

```

/*****
int RAM_test(void)
/*****
{
    int ram_cnt, i, pattern, error, rd_back;

    RAM_addr_rst();
    for (ram_cnt = 0; ram_cnt < 8192; ram_cnt++)
    {
        pattern = 0x0001;
        for (i = 0; i < 16; i++)
        {
            outport(base_address + ram_addr, pattern);
            pattern <<= 1;
        }
        error = 0;
        ram_cnt = 0;
        RAM_addr_rst();
        while((ram_cnt < 8192) && (error == 0))
        {
            pattern = 0x0001;
            i = 0;
            while((i < 16) && (error == 0))
            {
                rd_back = inport(base_address + ram_addr);
                if (rd_back != pattern)
                {
                    error = 1;
                    printf("RAM test error at address 0x%X.\nRead 0x%X instaed of 0x%X.\n",
                        16*ram_cnt+i, rd_back, pattern);
                }
                pattern <<= 1;
                i++;
            }
            ram_cnt++;
        }
        return error;
    }
}

/*****
void conf_reg(void)
/*****
{
    outportb(base_address + wp0_addr, wp0);
    outportb(base_address + wp1_addr, wp1);
    outportb(base_address + presc_addr, presc);
    outportb(base_address + fl_pos_addr, fl_pos);
    outportb(base_address + fl_len_addr, fl_len);
    outportb(base_address + h_px_addr, h_px);
    outportb(base_address + v_px_addr, v_px);
    outportb(base_address + h_off_addr, h_off);
    outportb(base_address + v_off_addr, v_off);
    outportb(base_address + pll_addr, pll);

    io_byte = inportb(base_address + ctrl_addr);

    if (wp_sel)    io_byte |= 0x02; /* set wiper sel bit */
    else          io_byte &= 0xFD; /* mask wiper sel bit */

    if (od_ev_st) io_byte |= 0x08; /* set odd/even bit */
    else          io_byte &= 0xF7; /* mask odd/even bit */
    outportb(base_address + ctrl_addr, io_byte);
}

```

```

/*****
void start_pot_wr(void)
/*****/
{
    io_byte = inportb(base_address + ctrl_addr);
    io_byte |= 0x80; /* start potentiometer write */
    outportb(base_address + ctrl_addr, io_byte);
}

/*****
void start_acq(void)
/*****/
{
    io_byte = inportb(base_address + ctrl_addr);
    io_byte |= 0x40; /* start acquisition */
    outportb(base_address + ctrl_addr, io_byte);
}

/*****
void RAM_addr_rst(void)
/*****/
{
    io_byte = inportb(base_address + ctrl_addr);
    io_byte |= 0x20; /* Reset RAM address */
    outportb(base_address + ctrl_addr, io_byte);
}

/*****
void LUT_load(void)
/*****/
{
    unsigned LUT_addr;
    char LUT_data;

    /* Threshold Look Up Table */
    LUT_data = 0;
    for (LUT_addr = 256; LUT_addr < 384; LUT_addr++)
    {
        outport(base_address + a_lut_addr, LUT_addr);
        outportb(base_address + d_lut_addr, LUT_data);
    }
    LUT_data = 0xFF;
    for (LUT_addr = 384; LUT_addr < 512; LUT_addr++)
    {
        outport(base_address + a_lut_addr, LUT_addr);
        outportb(base_address + d_lut_addr, LUT_data);
    }

    /* linear (1:1) Look Up Table */
    LUT_data = 0;
    for (LUT_addr = 0; LUT_addr < 256; LUT_addr++)
    {
        outport(base_address + a_lut_addr, LUT_addr);
        outportb(base_address + d_lut_addr, LUT_data);
        LUT_data++;
    }
    outport(base_address + a_lut_addr, 0);
}

```

```

/* NOTE: The last configured LUT remains active.
128 different LUTs can be stored
simultaneously. Select one by writing
the address to the LUT address register
(only the 8 MSBs are significant). */

}

/*****
int LUT_test(void)
*****/
{
    unsigned LUT_addr;
    char LUT_data, rb_data;
    int error;

    /* linear (1:1) Look Up Table */
    LUT_data = 0;
    for (LUT_addr = 0; LUT_addr < 256; LUT_addr++)
    {
        outport(base_address + a_lut_addr, LUT_addr);
        outportb(base_address + d_lut_addr, LUT_data);
        LUT_data++;
    }

    outport(base_address + a_lut_addr, 0);
    error = 0;
    LUT_addr = 0;
    LUT_data = 0;
    while((LUT_addr < 256) && (error == 0))
    {
        outport(base_address + a_lut_addr, LUT_addr);
        rb_data = inportb(base_address + d_lut_addr);
        if (rb_data != LUT_data)
        {
            error = 1;
            printf("LUT test error at address 0x%X.\nRead 0x%X instaed of 0x%X.\n",
                LUT_addr, rb_data, LUT_data);
        }
        LUT_data++;
        LUT_addr++;
    }
    return error;
}

/*****
int wait_acq_start(void)
*****/
{
    int usr_term;

    printf("Waiting for acquisition to start.\nPress a key to abort ...\n");
    io_byte = inportb(base_address+ctrl_addr);
    /* wait until ready */
    while(((io_byte & 0x01) == 0) && (!kbhit()))
        io_byte = inportb(base_address);

    if ((io_byte & 0x01) == 0)
    {
        usr_term = 1;
        printf("Acquisition not started. User termination.\n");
    }
    else
    {
        usr_term = 0;
        printf("Acquisition started.\n");
    }
}

```

```

    }
    return usr_term;
}

/*****
int wait_acq_end(void)
*****/
{
    int usr_term;

    printf("Waiting for acquisition to complete.\nPress a key to abort ...\n");
    io_byte = inportb(base_address+ctrl_addr);
    /* wait until ready */
    while(((io_byte & 0x01) != 0) && (!kbhit()))
        io_byte = inportb(base_address);

    if ((io_byte & 0x01) != 0)
    {
        usr_term = 1;
        printf("Acquisition not completed. User termination.\n");
    }
    else
    {
        usr_term = 0;
        printf("Acquisition completed.\n");
    }
    return usr_term;
}

/*****
void make_tiff(void)
*****/
{
    div_t size;
    int i, j;
    unsigned tmp_line[512];

    tiff_file = fopen (tiff_fname, "wb");
    tmp_file = fopen (tmp_fname, "wb");

    fprintf(tiff_file, "%c%c%c%c%c%c%c", 0x49, 0x49, 0x2A, 0x00,
        0x08, 0x00, 0x00, 0x00);
    fprintf(tiff_file, "%c%c", 0x08, 0x00);

    /* Tag 255: Type */
    fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0xFF, 0x00, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        0x01, 0x00, 0x00, 0x00);

    /* Tag 256: Width 512 */
    size = div(4*h_px, 256);
    fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0x00, 0x01, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        4*h_px-size.quot*256, size.quot, 0x00, 0x00);

    /* Tag 257: Length 480 */
    size = div(4*v_px, 256);
    fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0x01, 0x01, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        4*v_px-size.quot*256, size.quot, 0x00, 0x00);

    /* Tag 278: Rows per strip */
    /* fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0x16, 0x01, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        0x01, 0x00, 0x00, 0x00);
    */
}

```

```

/* Tag 273: Strip offset */
fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0x11, 0x01, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        0x56, 0x00, 0x00, 0x00);

/* Tag 279: Strip byte count */
/* fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0x17, 0x01, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        0x00, 0xC0, 0x03, 0x00);
*/
/* Tag 277: samples per pixel (1) */
fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0x15, 0x01, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        0x01, 0x00, 0x00, 0x00);

/* Tag 258: Bits per sample (8) */
fprintf(tiff_file, "%c%c%c%c%c%c%c%c%c",
        0x02, 0x01, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
        0x08, 0x00, 0x00, 0x00);

fprintf(tiff_file, "%c%c%c",
        0x00, 0x00, 0x00, 0x00);

RAM_addr rst();
/* Read 2 half images because of interlaced image */
/* Temporarily store first image */
for (j = 0; j < 2*v_px; j++)
{
    for (i = 0; i < 2*h_px; i++)
        tmp_line[i] = inport(base_address + ram_addr);
    fwrite(tmp_line, 4*h_px, 1, tmp_file);
}
fclose(tmp_file);

tmp_file = fopen(tmp_fname, "rb");
for (j = 0; j < 2*v_px; j++)
{
    /* copy line from temp file */
    fread(tmp_line, 4*h_px, 1, tmp_file);
    fwrite(tmp_line, 4*h_px, 1, tiff_file);

    /* copy line from frame grabber memory */

    for (i = 0; i < 2*h_px; i++)
        tmp_line[i] = inport(base_address + ram_addr);
    fwrite(tmp_line, 4*h_px, 1, tiff_file);
}

fclose(tmp_file);
i = remove(tmp_fname); /*
fclose(tiff_file);
}

```

```
|Address table:
|=====
|wp0_addr:    000000 -- Wiper pot. 0
|wpl_addr:    000001 -- Wiper pot. 1
|ctrl_addr:   000010 -- Control reg. Bit assignment:
|                -- 0: Busy bit (rd only)
|                -- 1: Stack (wiper) select
|                -- 2: Frame memory Al6
|                -- 3: Flash trigger odd or even
|                -- 4: unused
|                -- 5: Reset frame memory addr.
|                -- 6: Start frame acquisition
|                -- 7: Start potentiometer write
|ver_addr:    000011 -- Version number address (rd only)
|l_fr_addr:   000100 -- Frame memory address low byte
|h_fr_addr:   000101 -- Frame memory address high byte
|h_sp_addr:   000110 -- Horiz. sync pulse length
|v_sp_addr:   000111 -- Vert. sync pulse length
|h_px_addr:   001000 -- Horiz. # of pixels (*4)
|v_px_addr:   001001 -- Vert. # of lines (*4)
|h_off_addr:  001010 -- Horiz. # of pixels offset
|v_off_addr:  001011 -- Vert. # of lines offset
|d_lut_addr:  001100 -- LUT data
|flpos_addr:  001101 -- Flash position
|flen_addr:   001110 -- Flash pulse length
|pll_addr:    001111 -- PLL divider (*4)
|presc_addr:  010000 -- Prescaler for flash pos. and pulse length
|
|a_lut_addr:  01100X -- LUT addr (wr only) (16 bit wide)
|ram_addr:    01101X -- Frame memory (16 bit wide)
|
|
|
|restart
vector isa_a   isa_a[5:0]
vector isa_d   isa_d[15:0]
vector ram_a   ram_a[16:0]
vector ld      ld[7:0]

watch isa_clk pll_clk reset cs_fpga_n isa_wr_n isa_rd_n isa_a isa_d +
ld c_sync v_sync odd_even flash_trg iocsl6_n +
dc rst ram wr n ram rd n ram a busy n
```

```

step 60ns
clock isa_clk 0 1      |cycle = 120ns

step 60ns
clock pll_clk 0 1      |cycle = 120ns

| Let's go..
assign isa_d      00000000      |start conditions
assign isa_a      000000
assign ld         00000000

h reset
h cs_fpga_n
h isa_wr_n
h isa_rd_n
l odd_even
h c_sync
h v_sync
c 3
l reset

assign isa_d      00110011      |prescaler
assign isa_a      010000
ex wr_mac.cmd

assign isa_d      00000100      |pulse length
assign isa_a      001110
ex wr_mac.cmd

assign isa_d      00000100      |flash position
assign isa_a      001101
ex wr_mac.cmd

assign isa_d      00000010      |horiz # of pixels
assign isa_a      001000
ex wr_mac.cmd

assign isa_d      00000010      |vert # of pixels
assign isa_a      001001
ex wr_mac.cmd

assign isa_d      00000000      |Horiz. # of pixels offset
assign isa_a      001010
ex wr_mac.cmd

assign isa_d      00000100      |Vert. # of pixels offset
assign isa_a      001011
ex wr_mac.cmd

assign isa_d      01001000      |Start with even frame and start
assign isa_a      000010
ex wr_mac.cmd
c 4
l v_sync
h odd_even
c 4
h v_sync
c 50
l odd_even
c 4
l v_sync
h odd_even
c 4
h v_sync
c 30
ex mac2.cmd
ex mac2.cmd

```

```
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd
ex mac2.cmd

c 100
```

The 'write_mac.cmd' file:

```
l cs_fpga_n
c 1
l isa_wr_n
c 4
sim 600
h isa_wr_n
sim 600
c 1
h cs_fpga_n
c 1
```

The 'mac2.cmd' file:

```
h c_sync
assign ld 00000001
c 1
assign ld 00000010
c 1
assign ld 00000011
c 1
assign ld 00000100
c 1
assign ld 00000101
c 1
assign ld 00000110
c 1
assign ld 00000111
c 1
assign ld 00001000
c 1
assign ld 00001001
c 1
assign ld 00001010
c 1
assign ld 00001011
c 1
assign ld 00001100
c 1
assign ld 00001101
c 1
assign ld 00001110
c 1
assign ld 00001111
```

```
c 1
assign ld      00010000
c 1
assign ld      00010001
c 1
assign ld      00010010
c 1
assign ld      00010011
c 1
assign ld      00010100
c 1
assign ld      00010101
c 1
assign ld      00010110
c 1
assign ld      00010111
c 1
assign ld      00011000
c 1
assign ld      00011001
c 1
assign ld      00011010
c 1
assign ld      00011011
c 1
assign ld      00011100
c 1
assign ld      00011101
c 1
assign ld      00011110
c 1
assign ld      00011111
c 1
l c_sync
c 2
```

VITA

Stefan Weidmann was born in Bülach, Switzerland on March 4, 1964. He attended schools in the public system of the canton of Zurich, Switzerland. He graduated from high school in 1983. In 1984 he entered the Swiss Federal Institute of Technology in Zurich, Switzerland and received the Diploma in Electrical Engineering in 1990. After working for six years as a project engineer in the area of industrial electronics, he entered the Master's program in Electrical Engineering at The University of Tennessee, Knoxville in August of 1996. He officially received the Master's degree in May, 1998.