An FPGA-based 3D Graphics System

Master's thesis completed in Electronics Systems
by

Niklas Knutsson

LiTH-ISY-EX--05/3624--SE
Linköping 2005

An FPGA-based 3D Graphics System

Master's thesis completed in Electronics Systems
at Linköping Institute of Technology
by

Niklas Knutsson

LiTH-ISY-EX--05/3624--SE

Supervisor: **Jonas Carlsson and Mattias Olsson**

Examiner: **Ass. Prof. Kent Palmkvist**

Linköping 13th March 2005.

LINKÖPINGS UNIVERSITET

Avdelning, Institution
Division, Department

Institutionen för systemteknik
581 83 LINKÖPING

Datum
Date
2005-03-11

Språk
Language
Svenska / Swedish
X Engelska / English

Rapporttyp
Report category
Licentiatavhandling
X Examensarbete
C-uppsats
D-uppsats
Övrig rapport

ISBN

ISRN LITH-ISY-EX--05 / 3624--SE

Serietitel och serienummer **ISSN**
Title of series, numbering

URL för elektronisk version
http: / / www.ep.liu.se / exjobb / isy / 2005 / 3624 /

Titel Ett FPGA-baserat 3D-grafiksystem
Title

An FPGA-based 3D Graphics System

Författare Niklas Knutsson
Author

Sammanfattning
Abstract
This report documents the work done by the author to design and implement a 3D graphics system on an FPGA (Field Programmable Gate Array). After a preamble with a background presentation to the project, a very brief introduction in computer graphics techniques and computer graphics theory is given. Then, the hardware available to the project, along with an analysis of general requirements is examined. The following chapter contains the proposed graphics system design for FPGA implementation. A broad approach to separate the design and the eventual implementation was used. Two 3D pipelines are suggested - one fully capable high-end version and one which use minimal resources. The documentation of the effort to implement the minimal graphics system previously discussed then follows. The documentation outlines the work done without going too deep into detail, and is followed by the largest of the tests conducted. Finally, chapter seven concludes the project with the most important project conclusions and some suggestions for future work.

Nyckelord
Keyword
FPGA, 3D, Computer Graphics, IP-Cores

Abstract

This report documents the work done by the author to design and implement a 3D graphics system on an FPGA (Field Programmable Gate Array). After a preamble with a background presentation to the project, a very brief introduction in computer graphics techniques and computer graphics theory is given. Then, the hardware available to the project, along with an analysis of general requirements is examined. The following chapter contains the proposed graphics system design for FPGA implementation. A broad approach to separate the design and the eventual implementation was used. Two 3D pipelines are suggested - one fully capable high-end version and one which use minimal resources. The documentation of the effort to implement the minimal graphics system previously discussed then follows. The documentation outlines the work done without going too deep into detail, and is followed by the largest of the tests conducted. Finally, chapter seven concludes the project with the most important project conclusions and some suggestions for future work.

Acknowledgments

I would like to thank my supervisors Jonas Carlsson and Mattias Olsson for many interesting discussions, their good support and their friendly, encouraging attitudes. I would also like to extend my thanks to the examiner Kent Palmkvist, the opponent Anton Blad, and the rest of the staff at the Division of Electronics Systems.

Finally, I would like to thank everybody who supported and encouraged me during the project. Especially my parents and Andreas Söderlind. You made the project easier for me!

Acronyms and Definitions

AD	Analog Devices
AMBA	ARM's Microcontroller Bus Architecture
ARM	Advanced RISC Machines Ltd.
ASIC	Application Specific Integrated Circuits
ATI	ATI Technologies Inc. A leading manufacturer of graphics adapters.
CAS	Column Address Strobe
CL	CAS Latency
CLB	Configurable Logic Block
CLUT	Color LookUp Table
CMOS	Complementary Metal Oxide Semiconductor
CORDIC	COrdinate Rotation DIgital Computer.
CPU	Central Processing Unit
Crossbar switch bus	See section 4.
CRT	Cathode Ray Tube.
DAC	Digital to Analog Converter
DCM	Digital Clock Manager
DDR	Double Data Rate
DSP	Digital Signal Processing
FIFO	First in, first out.

Fixed-point	A number representation scheme, where a number R is represented by an integer N such that $R = N * B$, where B is the base of the representation.
Floating-point	A number representation scheme, where the number R is represented by $M * R^E$
FPGA	Field-Programmable Gate Array.
FSL	Fast Simplex Link
GPU	Graphics Processing Unit.
GUI	Graphical User Interface
HDL	Hardware Description Language
IP-Core, Core	Intellectual Property-Core.
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LGPL	Lesser General Public Licence
LMB	Local Memory Bus
LUT	Look-Up Table
LVDS	Low Voltage Differential Signal
Microblaze	A soft-core 32 bit RISC microprocessor designed specifically for Xilinx FPGAs.
NVIDIA	NVIDIA is a market leader in graphics and digital media processors.
OPB	Online Peripheral Bus
PCI	Peripheral Component Interconnect
Picoblaze	A fully embedded 8-bit microcontroller macro for the Virtex series of FPGAs.
Pipeline	A sequence of functional units which performs a task in several steps.

Pixel	Contraction of picture element.
Polygon	A plane figure having many angles, and consequently many sides. A triangle is a polygon.
PROM	Programmable Read Only Memory
Raster graphics	Computer graphics in which an image is composed of an array of pixels arranged in rows and columns.
RAM	Random Access Memory
Resource share bus	See section 4.
RGB	Red-Green-Blue
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RS-232	The most common asynchronous serial line standard.
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random Access Memory
TAP	Test Access Port
Tri-state	Allows a connector to either act as a signal driver, or to be set to a high-impedance condition.
TTL	Transistor-Transistor Logic
Vector graphics	The representation of separate shapes such as lines, polygons and text, and groups of such objects and using them to render an image.
Verilog	A Hardware Description Language for electronics design and gate level simulation.
Vertex	The point of intersection of lines.
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language.
VGA	Video Graphics Array
Wishbone	A System-on-Chip Interconnect Architecture for use with IP-Cores.

Xilinx	Founded in San Jose, California, in 1984, and invented the field-programmable gate array.
Z80	An 8-bit microprocessor by Zilog Ltd.
3D	Three dimensional

Contents

Chapter 1

Introduction

1.1 Background

Real-time computer graphics hardware has undergone a major transition the last two years: From supporting a few fixed algorithms, to being fully programmable. This development is likely to continue in the future, and will allow developers to interact more and more with the rendering mechanism. At the same time, the performance increase of graphical processing units (GPUs) is even greater than that of central processing units (CPUs), because GPUs can efficiently exploit the enormous parallelism available in graphic computations. The parallelism implies suitability for hardware implementation. Of course, the manufacturers use this: The latest chips (the 6800 line) from NVIDIA contain 16 separate pipelines with pixel processing units, each doing eight 64-bit floating-point operations in parallel. Compared to the size of a Pentium 4, the GPUs are significantly larger.

The latest generation of GPUs provides pipeline programmability. GPUs here refer to the ones available to low-cost systems. NVIDIA's GeForce 3 and up or ATI's Radeon 8500 are good examples. The most common method to increase programmability, is to include processors attached to the rendering pipeline. The extreme parallelism of graphic algorithms, combined with the brute force computational power and the ability to control the hardware, i.e., programmability, gives a huge performance-gain.

Field Programmable Gate Arrays (FPGAs) are basically pieces of programmable logic. Designs which previously would have required Application Specific Integrated Circuits (ASICs) can now be implemented with a few standard parts connected to a FPGA. As FPGAs have become more affordable they have found their way into more and more designs. It has now become possible to build very complex systems (for example an entire microprocessor with peripherals) based around a €20 FPGA. The appeal of FPGAs is the ability to handle 'in hardware' a number of complex functions which previously would have been handled in the software domain. FPGA development is performed in a similar way to software development, except with different languages and tools. Debugging is somewhat

more difficult, and building a design generally takes longer for an FPGA than for software. FPGAs are best used to augment the functionality of a CPU, to enable the CPU to perform more tasks more quickly. Since hardware can perform many operations in parallel, this can raise performance significantly. The price of this is extra design complexity.

Currently, IP-cores available for integration must normally be purchased from established vendors, often at very high prices. These costs can be burdensome, especially for small design teams with limited funding. The purchaser might not have access to the source, which makes the task of integration much more difficult, perhaps even preventing it. Some Internet communities work in another direction. One of them is OpenCores[6]. Their main objective is to design and publish core designs under a license for hardware modeled on the Lesser General Public License (LGPL) for software. The ideal of freely available, freely usable and re-usable open source hardware is the commitment. Consequently, there is around a hundred cores available at the website for use under the license.

The Division of Electronic Systems, at the Department of Electrical Engineering of Linköping Institute of Technology offer tuition in several topics, and one of them is design of digital systems. To ensure a continuing interest for the divisions education for students, and other informative reasons, a request for a demonstration design exists. The demonstration system should ideally interest the spectator in modern, high-speed digital systems. Available to the division, are several development boards, ideal for a digital system prototype.

1.2 Purpose

- Design and implement an FPGA-based graphics system on one of the divisions development boards.
- Use and evaluate IP-cores in the design of the graphics system.
- GPUs benefit from pipeline programmability. FPGAs provide a flexible compromise between software and hardware, as the hardware itself can become something that software can run on. Examine and investigate how this can be used when designing graphics systems.

1.3 Outline

Chapter one is an introduction to the project and presents the background and the purpose of the report.

Chapter two is an introduction to computer graphics techniques and computer graphics theory. If the reader is already acquainted with basic computer graphics this chapter can be skipped.

Chapter three serves to introduce the reader to the Memec Development Board and the Virtex-II FPGA that was available for prototyping in the project.

Chapter four contains the design of a rendering pipeline for FPGA implementation. A general approach to separate the design and the eventual implementation was used. Two 3D pipelines are suggested. One fully capable high-end version, and one minimal resource use version.

Chapter five documents an attempt to implement the minimal graphics system discussed in chapter four. The documentation outlines the work done without going too deep into detail.

Chapter six portray the largest of the tests conducted throughout the implementation phase.

Chapter seven concludes the project with the most important conclusions and and some suggestions for future work.

Chapter 2

Computer graphics rendering

The process of image synthesis or rendering, is the process of transforming an abstract description of objects into a digital image. The description is often referred to as the virtual world - or more exact, the description represents the geometry of the virtual world in a mathematical form. A form that can be used to render the image. Although a renderer may be able to synthesize many advanced object properties such as material and optical properties, the fundamental ability of a renderer is to be able to render geometric primitives. Objects refer to a specific part of the virtual world. Each object is composed of primitives. There are several ways of describing these objects, but the most common way is often referred to as vector graphics. For example, a virtual world may consist of an apple, lying on a table. A model of this virtual world might consist of two objects, the table and the apple. Both the table and the apple consist of several geometric primitives. The table might consist of five rectangles. One for each leg, and one for the table board.

Obviously a computer screen or display is a two dimensional surface, and a virtual world is in most cases three dimensional. A traditional computer image is a precise description of pixels, where pixel is a commonly used contraction of picture element. An image, being an array of pixels form a raster. The quality of a raster image is determined by the total number of pixels (called its resolution), and the amount of information in each pixel (often called color depth). Raster graphics cannot be scaled (resized) without loss of apparent quality (or more accurately, once an image is rasterized, its quality is fixed and cannot improve even on better display devices). It takes a large amount of data to store a high-quality raster image, often data compression techniques are used to reduce this size. Some of these techniques actually lose information, and therefore image quality. This is in contrast to vector graphics, which easily scale to the quality of the device on which they are rendered. In situations where the users' screens vary in resolutions by as much as a factor of ten in size, scaling is very important. Describing the screen's contents is very often a lot more efficient in comparison to implicitly specifying each pixel. This gives rise to the demand for powerful renderers.

Rendering is a computationally intensive process. When being done in real time on a personal computer(like in modern computer games for example), it is often supported by 3D hardware accelerators in graphic cards.

There are a number of different phenomena that has to be simulated for realistic results when rendering a scene:

- diffuse reflection
- specular reflection
- refraction
- global illumination
- depth of field
- motion blur
- diffraction

In 1986, Kajiya[7] introduced the rendering equation as a way of modeling global illumination in an environment arising from the interplay of lights and surfaces. The rendering equation and its various forms have since formed the basis for geometric primitive-based rendering and enabled a new level of realism to be achieved in computer graphics. Virtually all 3D rendering software and hardware produces an approximation to a solution of some idealized rendering equation.

When using geometric primitive-based rendering, the primitives are constructed from vertexes. A vertex is a point in three dimensional space with a particular location, usually given in terms of its x, y, and z coordinates. It is one of the fundamental structures in polygonal modeling: two vertexes, taken together, can be used to define the endpoints of a line; three vertexes can be used to define a planar triangle.

2.1 Three dimensional mathematics

Objects are represented as sets of vertexes, and each vertex can be seen as a vector in a coordinate system. Often, it is required to find how vectors change when the basis for the vector is changed. For example, an object is normally defined in its own local coordinate system. This is synonymous with the more mathematical description: the object is normally defined with its vectors and local basis. At some point, an object will appear in front of the camera. It is natural then to convert from the object's basis to the cameras. This conversion can be done elegantly using matrix math, where a set of fundamental matrices are multiplied together to perform huge amounts of work.

Instead of representing each point in three-dimensional space with a three-dimensional vector, homogeneous coordinates allow each point to be represented by an infinite number of four dimensional vectors. Homogeneous coordinates utilize

a mathematical trick to embed three-dimensional coordinates and transformations into a four-dimensional matrix format. The three-dimensional vector corresponding to any four-dimensional vector can be computed by dividing the first three elements by the fourth, and a four-dimensional vector corresponding to any three-dimensional vector can be created by simply adding a fourth element and setting it equal to one.

There are excellent tutorials available on the topic of matrix math and its uses in three dimensional graphics, and to repeat them here would be to go beyond the scope of this report. Suggested reading is Edward Angels "Interactive Computer Graphics" [8]. A summary follows:

- A vertex is normally represented as a 4x1 column vector
- There are fundamental 4x4 matrices that, when multiplied with a 4x1 vector, perform a fundamental transformation
- Three fundamental matrix operations are: Translation, Rotation and Scaling
- The matrices can be multiplied together, forming a new matrix that apply all of the matrices' operations at the same time
- A vector or vertex is represented by 4 floating-point numbers

Example: Consider a virtual world that consists of 200 vertexes. Each vertex has to be scaled, translated and rotated twice to be projected to the screen basis (i.e the two dimensional x and y coordinates). The matrix multiplication can be written as STR_1R_2v where S,T,R_1 and R_2 each represent a fundamental operation (scale, translate, rotate one, rotate two). v represents a vertex in the form of a vector. Each vertex vector is matrix multiplied four times, resulting in 800 4x4 matrix by 4x1 vector multiplications. If however, given that the matrices S,T,R_1 and R_2 are constant throughout the current frame, the matrices can be pre-multiplied. This will result in a new matrix, M. Hence $M = STR_1R_2$. To build is 3 multiplications and to process the vertexes are another 200 (200 vertexes), for a total of 203. The prerequisite for this huge improvement was that the matrices are constant. This is in fact the same as a constant position of the camera for the frame. This is almost always the case.

2.2 A rendering pipeline

Figure 2.1 shows an abstract overview of a rendering pipeline.

2.2.1 Primitive decomposition and tessellation

The first step of rendering is the decomposition of objects used for modeling into points, lines, or polygons suitable for the image synthesis algorithms. In order to allow geometric transformations, the allowed type of objects should be limited. The

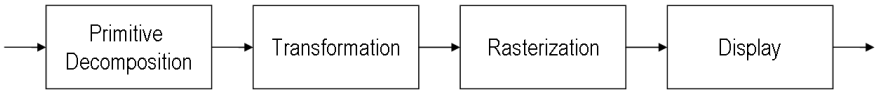

Figure 2.1. Overview of a computer graphics renderer

tessellation finds the sets of geometric objects whose type is invariant under homogeneous linear transformation. These types are the point, the line segment and the polygon. Tessellation approximates all surface types by points, line segments and polygons.

2.2.2 Transformation and clipping

Objects are defined in a variety of local coordinate systems. However, the generation of the image requires the objects to be in the coordinate system of the screen since the color distribution of the screen has to be determined. This requires geometric transformation. On the other hand, objects which lie outside the finite view volume shape defined by the camera, and the sides of the viewing window, are not interesting for further processing. The process of removing those invisible parts (they are are outside what is refered to as the viewing frustum) is called clipping. An object that lies inside the view volume, will not be displayed if it is obscured by other objects. Algorithms for Hidden-surface removal must be carried out to prevent rendering of surfaces that lie within the view volume, but face away from the camera or is completely or partially hidden by other surfaces.

Collectively, these operations constitute what has been called front-end processing. All involve three-dimensional computation and all require floating point operations.

2.2.3 Rasterization

At this stage in the pipeline only visible, two dimensional objects remain. For example, after front-end processing, a line segment that in three dimensions was defined by two three-dimensional vertexes are now represented by two two dimensional vertexes. The final projection to the screen coordinate system, i.e., pixels, is called the rasterization.

2.2.4 Display

The actual pixels has to be written to the frame buffer, i.e., RAM. Note that transformation and clipping handle geometric primitives such as points, lines or polygons, while in visibility computation - if it is done in image space - the primary object is the pixel. Since the number of pixels is far more than the number of primitives, the last step is critical for real-time rendering.

2.3 Vertex and pixel processors

Today, the most common video cards for personal computers now support pixel and vertex shaders in hardware, allowing their use for real-time rendering. The term shader is often used and refers to an assembly language program run on a dedicated processor. This enables the developer to complement the rasterization stage inside the actual pipeline. The technology is by now quite mature, and the latest generation of games, including Doom3 and Half-Life 2 make extensive use of hardware shaders.

The addition of programmable vertex shaders and pixel shaders makes visual quality in real-time graphics take a enormous leap toward cinematic realism. One downside to the shaders is that older graphics card cannot support it because they do not have programmable graphics processors, but all future cards will most probably have one. Example effects that very successfully use, or require, shaders:

- Hair and fur
- Per-pixel lighting
- Underwater effects
- Clothing

Vertex shader

A vertex shader is a graphics processing function which manipulates vertex data values through mathematical operations. These variations range from differences in color, texture coordinates, orientations in space, fog (how dense it may appear at a certain elevation) and point size.

When a vertex shader is enabled, it might replace the fixed-function pipeline for vertexes. The shader does not operate on a primitive like a triangle, but on a single vertex. A vertex shader cannot create or destroy vertexes, it can only manipulate the vertexes. For every vertex to be processed, the shader program executes.

Pixel shader

Pixel shaders operate after the geometry pipeline and before final rasterization. They often operate in parallel with texturing to produce a final pixel color and z-value for the final, rasterization step in the graphics pipeline. Pixel shaders often require data from and are ”driven” by the vertex shader. For example to calculate per-pixel lighting the pixel shader needs the orientation of the triangle, the orientation of the light vector and in some cases the orientation of the view vector.

Chapter 3

FPGA-based design

The chapter serves to introduce the reader to the Memec Development Board and the Virtex-II FPGA that was available for prototyping in the project. General design issues are then listed, regarding both the design process and the general hardware requirements around a graphics system.

3.1 Memec Design development board

The Division of Electronic Systems owns several FPGA- oriented development boards. Designated for use in this project was the Virtex-II MB1000 Development Kit from Memec Design. It provides a complete solution for prototype development with the Xilinx Virtex-II FPGA. Optional P160 expansion modules enable further application specific prototyping and testing. The system board includes a 16M x 16 DDR memory, two clock sources, RS-232 port, and additional support circuits. An LVDS interface is provided with a 16-bit transmit and 16-bit receive port plus clock, status, and control signals. Xilinx ISE software and a JTAG cable complement the kit and easing development. Figure 3.1 gives an overview of the system.

3.1.1 The FPGA

The development board utilizes the 1-million gate Xilinx Virtex-II device (XC2 V1000-4FG456C) (speedgrade -4) in the 456 fine-pitch ball grid array package. The high gate density and large number of user I/Os allows complete system solutions to be implemented in the advanced platform FPGA. The Virtex-II FPGA family has the advanced features needed to fit demanding, high-performance applications such as graphic systems.

3.1.2 DDR-memory

The development board provides 32MB of DDR memory on the system board. This memory is a *Toshiba TC59WM815-80 16Mx16 DDR* device. The -80 version is the

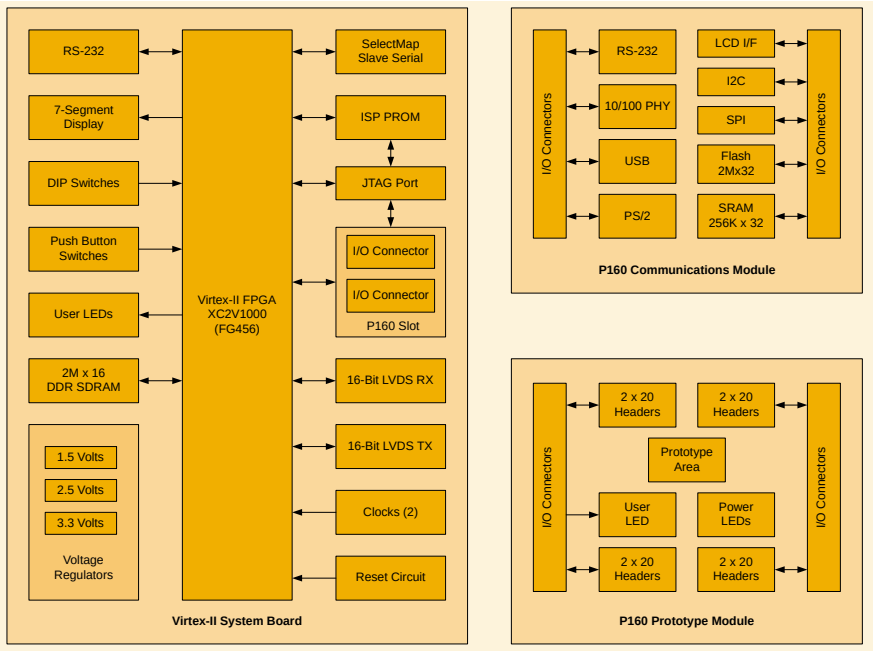

Figure 3.1. Functional overview of the development board and the available expansion modules[9]

slowest one, with the most important parameter tCK (clock cycle time) being 8 ns when using a CL (CAS latency) of 2.5. Its also possible to use CL 2.0 with a resulting 10 ns for tCK. In effect, at CL=2 maximum frequency for the memory is 100 MHz (which results in 200MHz with the DDR capability).

3.1.3 PROM

The development board utilizes the Xilinx XC18V04 ISP PROM, which allows FPGA designers to quickly download revisions of a design and verify the design changes in order to meet the final system-level design requirements. The XC18V04 ISP PROM uses two interfaces to accomplish the configuration of the Virtex-II FPGA. The JTAG port on the XC18V04 device is used to program the PROM with the design bit file. Once the XC18V04 has been programmed, the user can configure the Virtex-II device in Master Serial or Master SelectMap mode. Once selected, the XC18V04 device will use its FPGA Configuration Port to configure the Virtex-II FPGA.

3.1.4 JTAG

The Virtex-II development board provides a JTAG connector that can be used to program the onboard ISP PROM and configure the Virtex-II FPGA.

3.1.5 P160

The development kit includes a P160 Prototype module, which connects to the main system development board via the I/O module connectors. This board can be used to prototype various user I/O interfaces. A high-level block diagram of this module is given in figure 3.2.

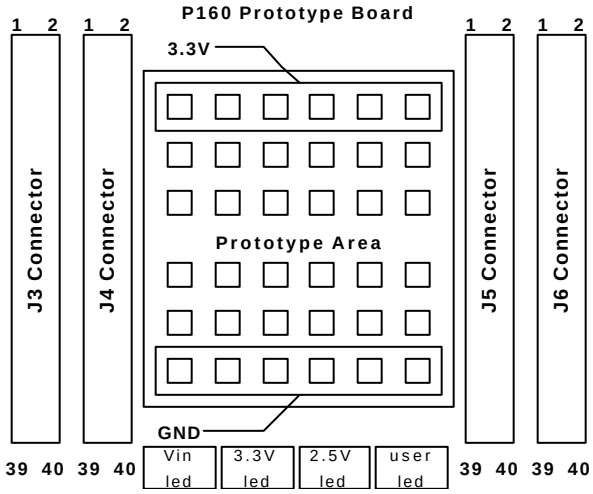

Figure 3.2. P160 Prototype board overview[10]

3.1.6 ADV7120

The ADV7120 is a digital to analog video converter on a single monolithic chip. The part is specifically designed for high resolution color graphics and video systems. It is also ideal for any high speed Communications type applications requiring low cost, high speed DACs. It consists of three, high speed, 8-bit, video D/A converters (RGB); a standard TTL input interface and high impedance, analog output, current sources. The ADV7120 has three separate, 8-bit, pixel input ports, one each for red, green and blue video data. Additional video input controls for the part include composite sync, blank and reference white. A 5 V supply and an external 1.23 V reference is needed for proper operation.

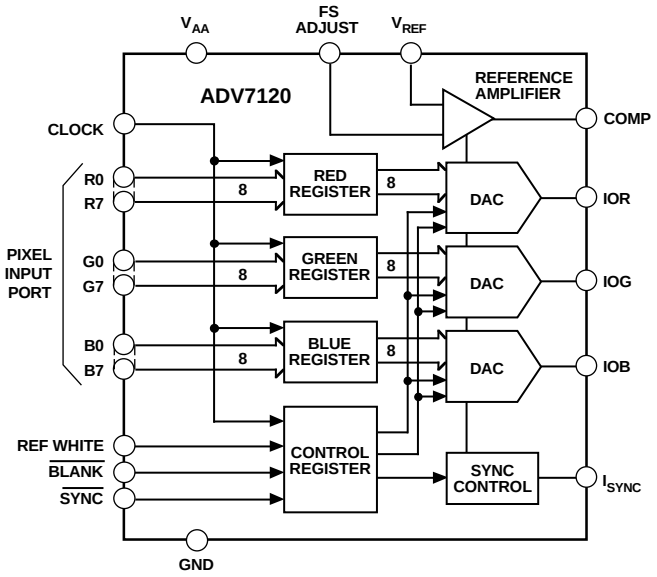

Figure 3.3. ADV7120 functional overview[11]

3.1.7 Other features

- The Virtex-II system board provides two on-board oscillators, one at 100Mhz and the other at 24Mhz.

- Several voltage regulators are used on the Virtex-II development board to provide the required on-board voltage sources. The main 5.0V voltage is provided to all on-board regulators to generate the 1.5V, 2.5V, and 3.3V voltages.

3.2 FPGA - Virtex-II

The Virtex-II family is a platform FPGA developed for high performance from low-density to high-density designs that are based on IP cores and customized modules. The family delivers complete solutions for telecommunication, wireless, networking, video, and DSP applications, including PCI, LVDS, and DDR interfaces. The leading-edge $0.15\mu m/0.12\mu m$ CMOS 8 layer metal process and the Virtex-II architecture are optimized for high speed with low power consumption. Combining a wide variety of flexible features and a large range of densities up to 10 million system gates, the Virtex-II family enhances programmable logic design capabilities.

3.2.1 Features

Configurable Logic Blocks (CLBs)

CLB resources include four slices and two tri-state buffers. The XC2V1000 has 40*32=1280 CLBs and consequentially 5120 slices and 2560 tri-state buffers. Each slice is equivalent and contains:

- Two function generators (F & G)
- Two storage elements
- Arithmetic logic gates
- Large multiplexers
- Wide function capability
- Fast carry look-ahead chain
- Horizontal cascade chain (OR gate)

The function generators F & G are configurable as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM memory. Supported SelectRAM memory for the XC2V1000 is 163840 bits. It is important to realize that using CLB's as distributed SelectRAM make them unavailable for use as logic. Usage of the Block SelectRAM memory is preferred. In addition, the two storage elements are either edge-triggered D-type flip-flops or level-sensitive latches. Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources.

Block SelectRAM Memory

The block SelectRAM memory resources are 18Kb of dual-port RAM, programmable from 16K x 1 bit to 512 x 36 bits, in various depth and width configurations. Each port is totally synchronous and independent, offering three read-during-write modes. Block SelectRAM memory is cascadable to implement large embedded storage blocks. The XC2V1000 contain a total of 40 SelectRAM blocks.

Multipliers

A multiplier block is associated with each SelectRAM memory block. The multiplier block is a dedicated 18 x 18-bit multiplier and is optimized for operations based on the block SelectRAM content on one port. The 18 x 18 multiplier can be used independently of the block SelectRAM resource. Read/multiply/accumulate operations and DSP filter structures are extremely efficient. Both the SelectRAM memory and the multiplier resource are connected to four switch matrices to access the general routing resources.

Global Clocking

The DCM and global clock multiplexer buffers provide a complete solution for designing high-speed clocking schemes. The XCV2-1000 have 8 DCM blocks. To generate de-skewed internal or external clocks, each DCM can be used to eliminate clock distribution delay. The DCM also provides 90-, 180-, and 270-degree phase-shifted versions of its output clocks. Fine-grained phase shifting offers high-resolution phase adjustments in increments of 1/256 of the clock period. Very flexible frequency synthesis provides a clock output frequency equal to any M/D ratio of the input clock frequency, where M and D are two integers.

Boundary Scan

Boundary scan instructions and associated data registers support a standard methodology for accessing and configuring Virtex-II devices that complies with IEEE standards 1149.1-1993 and 1532. A system mode and a test mode are implemented. In system mode, a Virtex-II device performs its intended mission even while executing non-test boundary-scan instructions. In test mode, boundary-scan test instructions control the I/O pins for testing purposes. The Virtex-II Test Access Port (TAP) supports BYPASS, PRELOAD, SAMPLE, IDCODE, and USERCODE non-test instructions. The EXTEST, INTEST, and HIGHZ test instructions are also supported.

3.3 FPGA Design considerations

In this section, various considerations that are important for the design process associated with FPGAs are listed.

3.3.1 Hardware description language, HDL

There are two industry standard hardware description languages, VHDL and Verilog. The complexity of designing with FPGAs with specific tools and methods of mapping the described hardware onto FPGA hardware has increased. As a result, it is important to master both languages, and that the tools used allow both languages to be used together. The choice of HDL is shown to not be based on technical capability, but on three primary issues[12]:

- Personal preference
- Tool availability
- Business and marketing issues.

Given this, it is rather likely that both languages will have to be used. Just as likely is that some problems will arise out of this.

3.3.2 IP-Cores

When designing a digital system of reasonable size using a HDL, the use of pre-designed IP-cores, or cores, should be considered. An IP-core is a component, or collection of components, that when employed supply functions to the design. The advantages are many but the most important ones are perhaps:

- A user of a IP-core must not necessarily know the exact implementation of the IP-core, only how to use it.
- Code and design reusability and flexibility
- Portability through the use of interconnection standards
- Predictability and performance

Example: An IP-core that offer easy and straightforward communication with a DDR-memory to a system. A DDR-memory uses several clocks, a rather sophisticated command protocol and other extras that the some designers probably would prefer not to concentrate on.

3.3.3 OpenCores

Today there are many companies, with a primary business to design IP-cores and sell or license them for use by other companies. The license fee is often high. This is understandable considering the expertise that has been put into the IP-core. Sadly, this fact makes the use of such professional IP-cores not very feasible for university projects. A very interesting development the last years have seen the emergence of communities, primarily Internet based, that supply IP-cores for free. One such community is OpenCores with URL: `www.opencores.org`. Many IP-cores are developed by amateurs with an interest for digital design, but just as many are developed by hardware design professionals that want to contribute. At time of writing there is around a hundred IP-cores available at the website. About half of them are written using Verilog HDL and the rest in VHDL. Most of them are still under constant refinement or development.

Some of the IP-cores available at OpenCores could prove very useful in the project. These include a DDR-controller, IP-core interconnectioncores, VGA/LCD-controller, several microcontrollers and CORDIC-cores.

3.3.4 Xilinx IP-cores

The Division of Electronic Systems is under the possession of several IP-cores under license from Xilinx, one of them is the popular Microblaze IP-core. Microblaze is a popular full featured processor IP-core. Supplied with the Xilinx Foundation Toolkit is also the CoreGen utility. This utility lets the user generate and customize IP-cores for use in the design. The IP-cores are generated for a specific FPGA and makes very good use of the resources available in the FPGA. For instance, inferring Block SelectRAM in the system using CoreGen is a simple process.

3.3.5 IP-Core types

Rajsuman[13] defines soft, firm and hard IP-cores. For FPGA design there are mainly two aspects. All of OpenCores IP-cores are soft cores, meaning they are RTL descriptions that can be mapped to (synthesized) to virtually any FPGA ,i.e., vendor-independent. Xilinx IP-cores on the other hand are firm IP-cores as they are mapped to Xilinx primitives. These primitives can then be mapped to the specific hardware by Xilinx own tools. Hard IP-cores are hardly usable in FPGA design. The effect of this is that, should the project use Xilinx IP-cores, a speed and especially area optimized IP-core can be expected. This at the cost of restriction to Xilinx hardware use.

3.3.6 Bus standards

The hardware, or logic, necessary to interconnect IP-cores is often referred to as glue logic. It is easy to underestimate the time required to design and verify glue logic. The vital testing to ensure proper function can dampen some of the glare associated, at first inspection, with the use of IP-cores. One attempt to solve this problem is specification of bus standards. Apart from being used for pure data transfer, a bus can also be used to specify the communication protocol between the IP-cores. Especially when used together with memory mapped control registers. The VGA/LCD controller from OpenCores contains a bus slave interface that is connected to the bus. Any master interface that is connected to the bus can then write to the slave on different memory locations. Each memory location relates to a function register that control how the IP-core operates. One register for instance control horizontal resolution of the output display.

This is all very good, only that the industry is full of different bus standards. OpenCores for instance use its own WishBone bus standard. Most (not all) IP-cores from OpenCores[6] use this standard. The Xilinx IP-cores on the other hand tend to use the OPB (Online peripheral bus) as its primary, low-speed, bus. Microblaze on its own use three different bus standards (OPB, LMB and FSL). AMBA is another worth mentioning. Solutions exist, there are bus-bridges, that make IP-cores from different bus standards communicate, but sometimes there are none. Bridges can also include unavoidable wait states and thus, slowing down bus cycles which can be critical. If is possible to use groups of IP-cores that are designed for use with the same bus standard, a lot of potential problems and delays can be avoided.

3.3.7 Tools

As the complexity and size of designs becomes larger, the need for advanced development tools increase. Even if HDL design is often text based and the work can be done using a text editor and command-line compilers, there are some very important advantages of using more integrated solutions. Several companies, one of them Xilinx, offer complete development environments that include graphical

design utilities. The use of these tools often aid in designing and viewing the hierarchy of the design. It also greatly simplifies the process of merging Verilog and VHDL modules. This is indeed a very important aspect. Verilog and VHDL source code tend to include redundant code parts. The graphical design tools have the ability to generate HDL from diagrams, saving time for the developer. Consider for example, in VHDL, an entity that use say five counters. None of them use the common counter feature load although the actual component have it. All use the same clock, count enable and reset. In VHDL this will result in a rather large source file with a lot redundancy in the component instantiations. In the graphical diagram much if this is resolved using global connectors. The diagram is quickly built and the VHDL can be automatically generated.

3.4 Hardware prerequisites

Although todays FPGAs have a lot of features it is probably necessary to complement any with external components.

3.4.1 RAM

Graphics systems require RAM and lots of it. Consider a 640*480 pixel resolution screen with a color depth of 16 bits. It would require 614.4 kB of memory to store just one frame. A larger frame and perhaps more modern of 1280*960*32 gives 4.9125 MB. FPGAs often supply RAM but rarely, if ever, this much. To be on the safe side and to allow for texture data and a working space for graphical algorithms, at least 16 MB of RAM should be available to the system. The speed of the RAM is also critical for the performance of the graphics system.

3.4.2 DAC and CRT screen

A cathode ray tube screen take analog signals as input. The most important signals are the color levels of red,green and blue respectively plus two synchronization signals. Conversion from digital data to analog signals is done through DACs. Consider again the 640*480*16 case. Assume the memory where the frame data resides has a word length of 16 bits and hence one read supplies one pixel for the screen. The screen is drawn by a raster beam that move across the screen from side to side, row by row. This must be done at least 60 times per second to fool human eye of a constant screen. Slower updates and flicker would appear. This would yield 640*480*60 pixels per second. That equals to about 18.5 million memory reads, or around 54 ns of memory cycle time. And it gets worse. Because of the time required for the raster beam to retrace when switching line and frames, the frequency must be higher. This can be seen as the resolution is 800*500. Figure 3.4 portrays this representation where total horizontal image size is 800 pixels and the horizontal gate is 640 pixels. Total vertical image size is 500 pixels and the vertical gate is 480 pixels. In effect, this means that a 25 MHz pixel clock frequency

is required. This is equivalent to a 40 ns duty cycle. And this is only to draw the current frame on the screen. What about changes to the actual screen?

The DAC must be able to convert digital data at a rate of at least 25MHz to supply an industry standard VGA screen with data. This is a lot faster than many DACs operate. Fortunately there are special DACs for this purpose, called Video DACs.

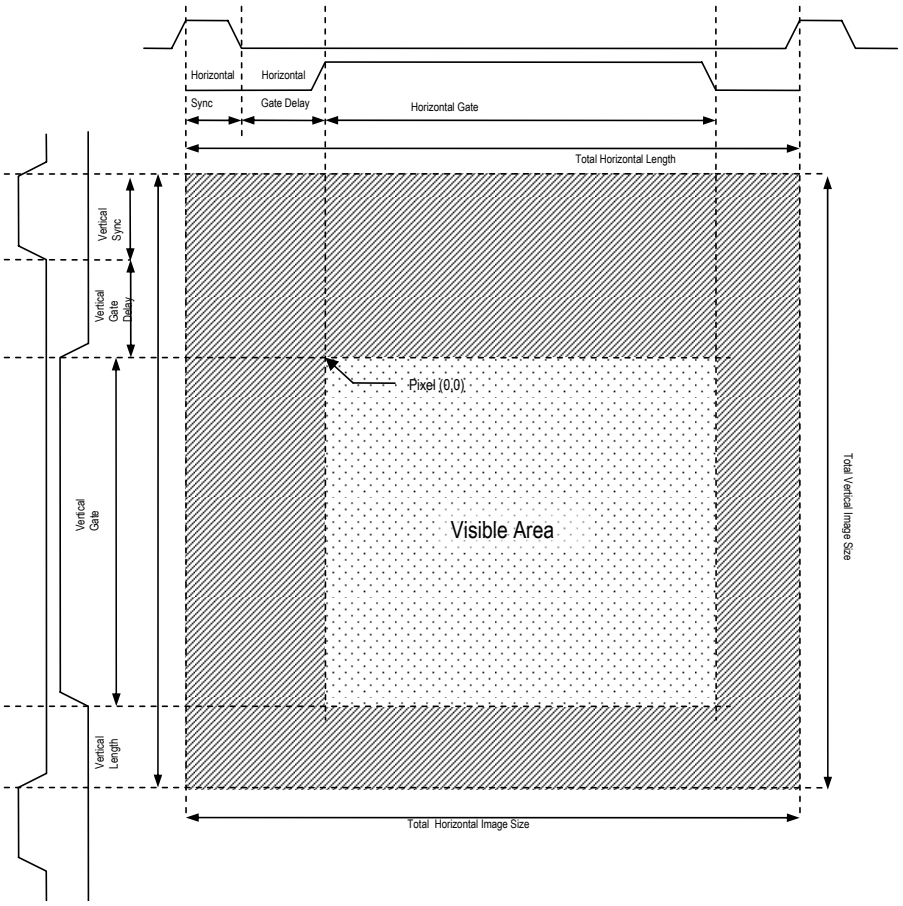

Figure 3.4. Resolution representation with borders and refresh/retrace times.

3.4.3 ROM

Depending on the type of FPGA chosen, a ROM (or EEPROM) might be necessary. The ROM's function is simply to reconfigure the FPGA when the power is switched on. Without a ROM, it would be required to connect the FPGA to the computer

with source BIT-file for reconfiguring. Never, except in an early prototype is this an option.

3.4.4 Other

Since there will be a DAC present in the system that require an analog ground, it is very important to have proper decoupling. Digital circuits typically caused by in the systems, interference signals can cause strange effects on the screen such as flickering and uneven colors. Consequently, some capacitors at different sizes is essential to the system. To improve even more analog ground shielding, a Ferrite bead. Most DACs need reference voltage levels as well.

Chapter 4

Rendering pipeline design

This chapter documents the design of a rendering pipeline for FPGA implementation. The design work was not done with any specific FPGA vendor or type in mind. Instead a more general approach was used to separate design and the eventual implementation. Two 3D pipelines are suggested. One, fully capable, high-end version and one minimal resource focused. The latter is called the minimal pipeline and is used as the basis for an implementation in chapter five.

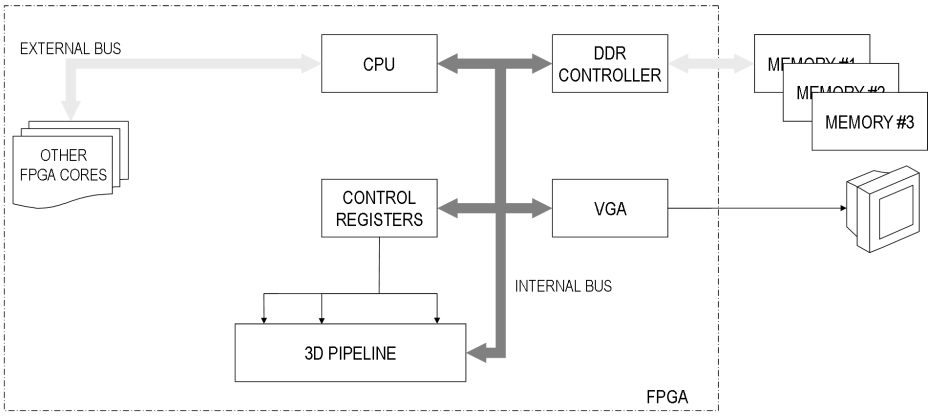

Figure 4.1. System overview

4.1 The system overview

The graphics system could be seen as a gigantic state machine. It does not necessarily require interaction with control units outside the system when started. Such

interaction take place when the state of the graphic system has to change, in a way not previously specified.

Figure 4.1 details a high level overview of the proposed graphics system. There are two buses in the system. The external handles communication between the graphic system and other parts *within* the FPGA. The internal bus is used for the internal data flow of the system. The main reason for this is that the internal bus is bound to be very busy with just providing data for the DACs. This process must not be interrupted as any lack of data for the DACs will at least result in flicker on the screen which can be considered as a failure of the graphics system. Note as well that the buses are assumed to be of *crossbar switch* type. To understand what

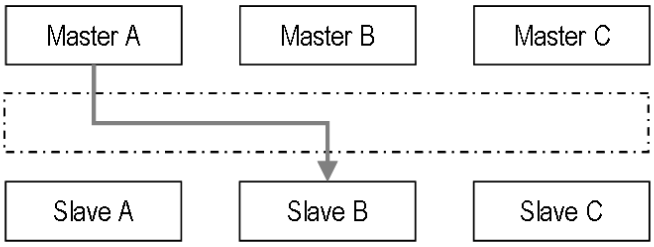

Figure 4.2. A regular type bus

a crossbar switch type bus is, consider an example system with three masters and three slaves. With an ordinary type of bus (sometimes referred to as a *resource share bus*, see figure 4.2), if master A acquires the bus for communication with slave B, then if *any* other of the two masters require the bus for communication with any other slave, the masters have to wait.

With a crossbar switch type bus, see figure 4.3, master B and master C can still acquires the bus for communication with any of slave A or slave C. Resource share bus is similar to a traditional, circuit board, hardware bus between for example a processor chip and a memory chip. A crossbar switch type bus is more of a data router. It requires some extra routing resources.

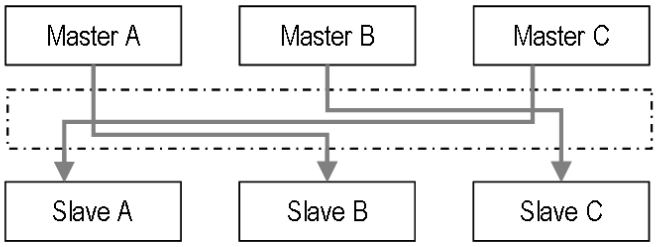

Figure 4.3. A crossbar switch type bus

4.1.1 VGA

This block's task is to supply the DACs with correct color data, blank signals and horizontal and vertical retrace signals. This will require a pixel clock at frequencies in the order of tens of megahertz, as discussed in section 3.4.2. Note that this frequency is the absolute minimum, for the system clock of the whole FPGA. The VGA block accesses the memory and reads the color data from it.

4.1.2 MEMORY

As seen in the figure 4.1 and as stated in section 3.4.2, external memory is required. The memory containing the current frame is bound to be rather busy. A fast SDRAM today can have access cycles of around 20 ns. Each read is bound to be accompanied with some overhead. The memory is not active during the blanking period, but nevertheless the bottle neck potential is obvious already at this stage. The solution is straight forward. A second memory is added. While the first memory is used to supply data for the pixels on screen, the next frame is built up in the second memory. Another advantage of this scheme is that *double buffering* is achieved for free. Double buffer is a common term in computer graphics and means that nothing is drawn on the frame while it is being displayed as this might induce flicker. Instead, drawing takes place on a secondary memory space, and when a new frame is to be displayed the VGA block switches memory to read from. Note that no actual copy of data is done between the buffers, the data is simply read from alternating places. The different frames are often referred to as the primary buffer and the secondary or double buffer.

A third memory is added to the cycle, and thereby improving it to a triple buffer. The reason is that, to clear the memory with a predefined background color or image is a rather time consuming operation. As said earlier over 600kB has to be written for a full frame, and before each buffer is used for drawing, it's important of course that it does not already from the beginning contain the colors of a previous frame. By using three memory capsules we have assured that the drawing mechanisms will have full access to frame memory during a whole frame.

4.1.3 GPU

This block, has several responsibilities. First of all it handles communication, through the external bus, with other parts of the FPGA, or even outside of it. This might for instance be a sophisticated processor IP-core. A user of the graphic system can then use a high level language and compiler to write software to interact with the graphics system. The GPU also acts as a general control unit for the system. It configures the function of the VGA unit. Another task is to start a memory frame data write for clearing with background data. Yet another is to set the control registers, either as a result of commands from outside of the graphics system or as a result of some special event.

4.1.4 3D PIPELINE

While the VGA block handles the supply of data to the screen, the 3D pipeline builds the data in memory in the first place. This perhaps makes it the heart of the graphics system. A special section (4.2) is devoted to its design.

4.1.5 CONTROL REGISTERS

This block is what controls the function of the pipeline. An example register is the projection matrix used for screen projection. Functions also include to turn on and off each different part of the 3D pipeline.

4.2 The 3D pipeline

The 3D pipeline builds the raster data in memory, perhaps making it the heart of the graphics system. As said in section 1.1, performance is achieved mainly by identifying and utilizing parallelism. This in turn is achieved mainly by increasing the size of the design. Many of the blocks of the pipeline are bound by this principle. The more hardware resources allocated, the better the performance. The FPGA available to a designer often has a fixed size. Therefore two designs of the 3D pipeline are presented below. A high-end pipeline, which includes all parts necessary for a high performance graphics system. The minimal pipeline on the other hand contains only the blocks the are mandatory for a rendering process.

Figures 4.4 and 4.5 show the high level organizations of the pipelines. In the figures, each white block signifies a part (or block) of the respective pipeline. Each block retrieves data from the previous block and transfer data to the next. The actual transfer is done using dual port FIFO buffers, each optimized for the specific communication between the blocks. The FIFO buffers are asynchronous, meaning that each block can in effect run at its own clock frequency. While this potentially increases the complexity of the design, it can be necessary and as well performance increasing. By allowing bottle neck parts of the pipeline to run at a higher clock frequency the performance of the whole system is increased.

The names of the blocks in figures 4.4 and 4.5 refer to the different functions as explained in 2.2. Although each block has some specific function associated with it, the slowest of them will inherently limit the performance of the whole pipeline. One of the big advantages of design using FPGAs and HDLs is that during design, simulation will reveal which part is the bottle neck of the pipeline. During implementation, the designer can then apply more design resources to the specific block thereby improving performance. Applying more resources generally means increasing the size of the block, exploiting parallelism.

4.2.1 General blocks

Below follows a short description of each part of the pipeline.

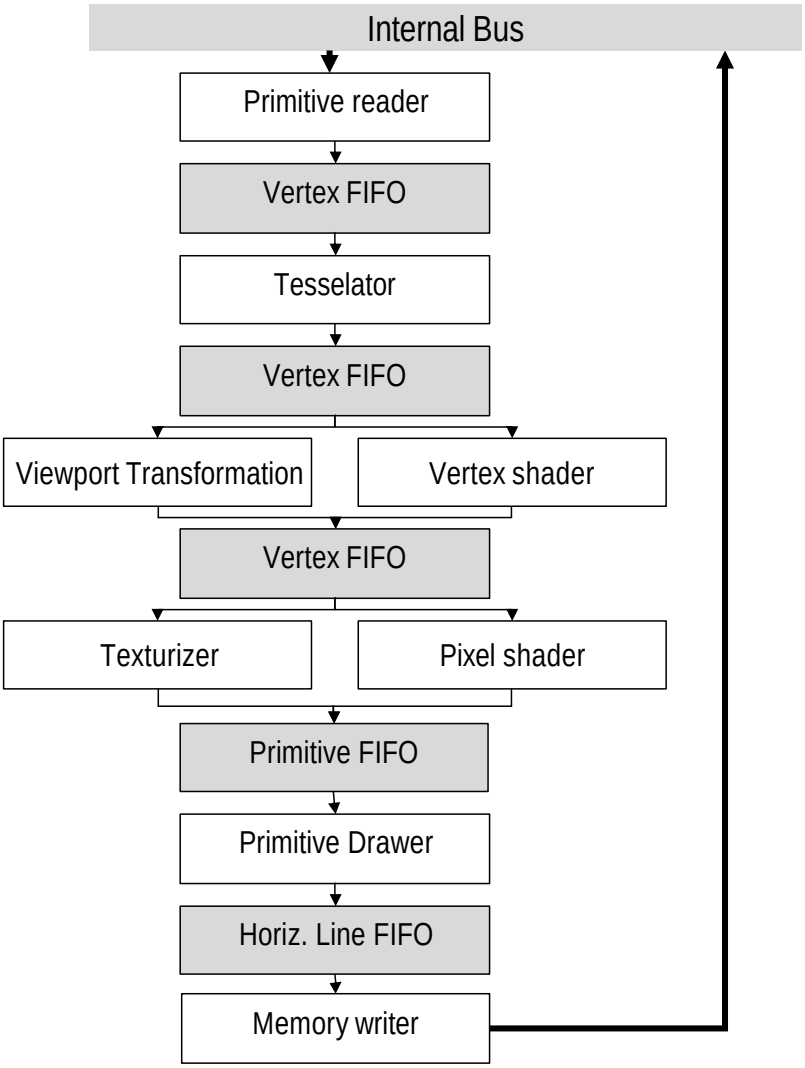

Figure 4.4. High-end 3D Pipeline implementation

Vertex FIFO This FIFO holds information about the world on a vertex level. Each vertex is a structure of four floating point coordinates, color value and how the vertexes are connected with other vertexes to form primitives. Any other information associated with a vertex can easily be appended here to add functionality to the pipeline. A good example is for the high-end pipeline: what programs are run for the vertex by the vertex shader and pixel shader respectively. Example structure bit sizes are given in table 4.1. Bit size 72 for the minimal pipeline is

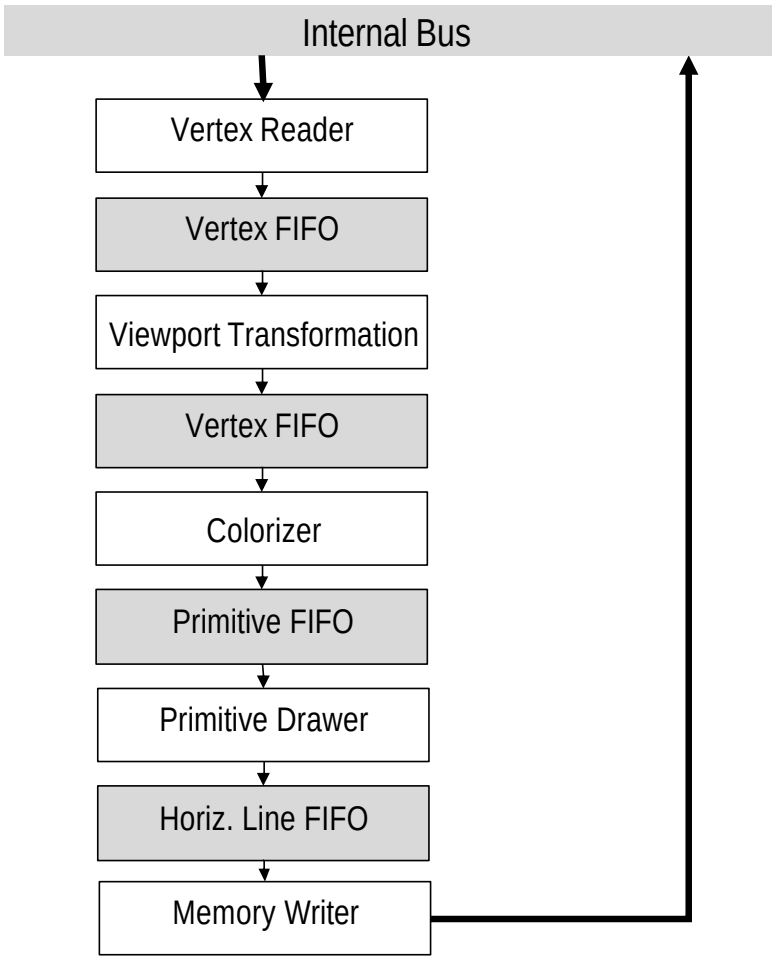

Figure 4.5. Minimal 3D Pipeline implementation

selected to make each entry optimally fit in a Xilinx BlockRam.

Primitive FIFO A storage FIFO buffer for storing 2D-primitives ready to be drawn onto the frame buffer. The primitives supported are normally triangles, lines and points.

Horizontal line FIFO In this FIFO buffer, the primitives have been converted into horizontal line segments and they are stored here until finally being burst-written to the memory.

Field	High-end	Minimal
Coordinates	4*32	4*13
Color	32	16
Identification	16	4
Pixel Shader	8	0
Vertex Shader	8	0
Total bits	192	72

Table 4.1. Vertex FIFO structure example sizes

Viewport transformation Mainly contains the Matrix-vector multiplication blocks. Is responsible for making the objects appear where they should in front of where the camera is in the virtual world.

Memory writer This is the last block of the pipeline and burst-writes horizontal lines into the active frame buffer.

4.2.2 High-end pipeline

Primitive reader This block reads the primitives of the supported primitive types. This can be triangles and lines but also more complex primitives like polygons, squares or circles. The primitives are stored on vertex basis, and they are coupled/identified together using the identification bits.

Tesselator This block works on the primitives of the supported primitive types (which can be triangles and lines or more complex primitives like polygons, squares or circles). These complex primitives are converted into the basic primitives by the tesselator. Basic primitives are normally triangles, lines and dots.

Vertex and Pixel shaders Extra processing power to the pipeline is added with the vertex and pixel processors. See section 2.3 for more information.

Texturizer A block that computes various extra functionality that is supported by the pipeline. See section 2.2.2 for more information. The final pixel color value that is written in the frame buffer is determined here.

4.2.3 Minimal pipeline

Vertex reader This block reads vertexes directly from memory, thereby assuming that the primitives already have been tesselated into basic primitives. The vertexes are written to the area where they are read by the vertex reader by the GPU.

Colorizer A simplified texturizer (see above) that merely computes the color of the pixels associated with the active vertexes.

Chapter 5

Implementation

This chapter documents an attempt to implement the minimal graphics system discussed in chapter four. By implementing the minimal pipeline first, it can then gradually be expanded toward the full pipeline. It is a prototype and the implementation phase was done under a very narrow time budget. As a result a number of different design simplifying decisions had to be taken. The documentation below tries to outline the work done without going too deep into detail.

5.1 General design decisions and simplifications

5.1.1 Hardware

The hardware available for the implementation is the Memec Development board. The Virtex-II FPGA provides plenty of fast, customizable and flexible resources. The FPGA and the ROM are on the development board. A P160 prototype board is also available for use. The P160 can ideally be used to connect the DAC to the development board. The P160 will however not have the space to fit extra ram when used for the DAC. The LVDS connectors on the board will not be used and can be used to connect extra ram. A separate circuit board is required for this, preferably a printed one. As a result of the sparse amount of time available to the implementation phase, the decision was made not to include the extra ram in the first implementation. The FPGA (see section 3.2.1) includes a lot of memory but it is still not enough to be used for raw data storage. Instead the block rams will be used as FIFOs and other temporary storage within the actual design. Only the on-board DDR-memory will be used to store the frame data. While this severely limits the performance of the graphics system, it is also relatively easy to add later, when the basic prototype is functioning.

The DACs available for use is the 30MHz version of ADV7120, meaning the pixel clock has a maximum of 30MHz. Since the resolution 640*480@60Hz already requires a pixel clock of 25Mhz, this resolution will be the highest available. In fact, the decision is taken that in the first implementation the resolution is fixed to

640*480@60Hz with a color depth of 16-bits. Although this limit comes from the speed of the DACs, it does not necessarily impose a further limit on the performance of the system. With the previous decision to use only the on-board DDR-memory, higher resolutions would require too much bandwidth of the memory's capacity for just the VGA-part. There would be almost no bandwidth left for the 3D-pipeline.

The implementation will use both oscillators available on the development board. The 24 MHz oscillator will be used as pixel clock, as the fixed resolution requires exactly this frequency if no borders (or margins) on the CRT display are used. The 100 MHz oscillator will be used as system clock. All parts of the implementation will be designed to be able to run at 100 MHz. As the Virtex-II FPGA available is of speed grade -4 (the slowest one) this demand might prove difficult to meet.

Design decisions summary

- No extra RAM implemented, use only the on-board DDR-memory.
- BlockRam used for internal temporary storage, program memory for processor IP-cores and FIFOs etc.
- Fixed Resolution of 640*480@60Hz, 16-bit color
- Use the ADV7120, 30MHz version, mounted on a P160.
- 100Mhz system clock, 24 MHz pixel clock

5.1.2 FPGA Design decision

There are two main design options: base the design on Xilinx IP-cores, i.e., a mix of free and commercial IP-cores, or base the design on OpenCores IP-cores. By using Xilinx IP-cores, one can expect a more probable success with the implementation of the system. By using the OpenCores IP-cores the risk of using faulty, or just IP-cores with bugs, is considerably higher. On the other hand, should the OpenCores IP-cores work really well the system will remain non-commercial. As one of the goals of the project is to evaluate IP-cores, using the OpenCores family of IP-cores is a sound decision even though the risk is higher. The desicion is hence made to work with OpenCores and the Wishbone interconnection style. This means that the Xilinx IP-cores will only be used to complement the OpenCores IP-cores.

- Use the OpenCores family of IP-cores
- Use Wishbone to interconnect IP-cores
- Complement with Xilinx IP-cores when required

5.2 HDL Implementation

5.2.1 Bus Master

Introduction

The bus master is not depicted in figure 4.1. Its function is to control the traffic on the internal Wishbone bus. OpenCores presently hosts three different IP-cores with this ability. All three are interesting for the implementation.

The first one is the CONBUS IP-core. It is a straightforward Verilog description and it is clear and easy to modify. Sadly the IP-core does not include any support for Wishbone Rev B3 signals. As this revision includes the tag-signals to support transfer modes such as linear incremental burst and such, this IP-core can only be used after improvement or either a severe performance penalty.

The next IP-core is the Interconnect Matrix by Rudolf Usselmann, CONMAX. Main features are 1,2 or 4 priority levels and up to 8/16 masters/slaves. The IP-core is well written and is very well documented although it seems to focus on functionality rather than speed. Sadly there is no Wishbone Rev B3 Support.

The one IP-core that remains and that actually do support the Wishbone Rev B3 signals is the Wishbone builder IP-core. The configuration of the Wishbone arbiter is typed into a define file. The syntax of this file is described in the documentation. The generator is a Perl script that outputs HDL code generated from the contents of the define file. Its even possible to use a GUI for tailoring the arbiter. Independently of the GUI a define file is always generated.

Implementation

There are a lot of parameters to set in the define file, allowing the user to choose how the bus master is implemented. Many signals are optional (allowing reduction of for example multiplexer sizes), the user can choose to use multiplexers, a simple and-or architecture or tri-state gates to implement the switching. There are multiple arbitration options to choose from, round-robin for example. The bus can be implemented as shared bus or crossbar switch.

Problems and notes

When using the IP-core it does not take a long time to run into problems. On the OpenCore website the IP-core is marked as done but the results of the Perl script reveal something else. The script was not tested for all configurations (i.e. define files) but when set to the configuration needed in the graphic system the Perl script generated unusable code. Having identified the obvious errors of having decimal numbers (for example 11.98776) as port sizes in VHDL these were corrected in hope that the error was only a minor bug. However, after having tried to use the IP-core successfully for over a week, it was finally decided to remove it from the system. Since the graphic system needed a fairly simply arbitration method, a completely new IP-core would be designed to do the bus mastering.

BusCop Introduction

The new IP-core was given the name BusCop. It was designed from and implemented from scratch by the author because no functioning IP-core that satisfied the requirements where found. In short, the basic requirements were:

- Support Wishbone Rev. B3 signals
- Allow an external bus arbitration override to ease debugging. This allows the designer to lock bus grant to a master/slave combination, bypassing the arbitration logic.
- Scalable. It must be relatively easy and quick to add or remove masters/slaves.
- Arbitration must be conducted by a separate block so it easily can be improved or customized or even complemented with other arbitration units for different arbitration schemes.

BusCop Implementation

The implementation, in VHDL, is built in HDL designer but is centered around a package header file where the number of masters and slaves to be serviced is entered as constants. Should the number of masters/slaves that are connected to the system change, they can be added to the system graphically. By changing the two constants and updating a component, the necessary ports appear and can be connect. All Wishbone signals are bundled together to reduce code size, readability and ease development. The separate arbitration block simple prioritize between different masters (the VGA block has the highest priority) in the system but this can be expanded at a later point. The external bus arbitration is fully implemented.

Problems and notes

During the integration test (see 6.2), the arbitration was occasionally controlled by the DIP switches on the development board. This was proved really useful for debugging purposes.

5.2.2 DDR memory interface

Introduction

The DDR SDRAM Controller IP-core of OpenCores.org has been designed for use in XILINX Virtex II FPGAs and is easily adapted to different DDR SDRAM devices. It is written by Markus Lemke. On the contrary to most IP-cores originating from OpenCores, the DDR SDRAM Controller IP-core does not have a Wishbone interface. Some key features include:

- up to 100 MHz system clock frequency in -4 speed grade

- CL = 2.0
- Automatic, configurable Auto Refresh
- Automatic precharge/activate when changing ROW/BANK

Implementation

The IP-core uses a rather neat trick to hide the DDR-standard signals with clocking on negative edges from rest of the system. The IP-core uses a burst length of 2, and then presents(or modifies) the data as a 32-bit data word. The IP-cores interface has just two simple commands, read and write. By using two Xilinx Virtex-II DCMs, the IP-core generates the clocks for the DDR SDRAM device. The phase shift of DCM 0 is adjusted, depending an board delays. The data mask signals are not used at all, meaning that there is no way to modify say less than all of the 32 bits. Since a pixel is 16 bits, there is no way to modify less then two pixels at the time. To change the IP-core to use the data mask signals would remedy this.

Problems and notes

The DDR SDRAM Controller IP-core has been designed for use in Xilinx Virtex FPGAs but not specifically for the Toshiba DDR memory. Hence the timing_constants in the ddr_sdr_conf_pkg.vhd file has to be modified to suit the timing parameters of the Toshiba memory. Carefully studying the data sheet of the memory gives the necessary data, the most critical one perhaps being *tRC, Active to Pre-charge delay*. While the simple interface shields the DDR-signals from the rest of the system, it also makes it impossible to use dynamic burst lengths. With image frame data, often huge amounts of data are read in long bursts as they are located linearly in the memory. The lack of burst control will limit the graphic systems performance. The design decision was made to have a Wishbone type system bus. To use the IP-core with the system bus a bridge of some sort is necessary. While it is very likely that such a bridge will become available on the OpenCores website within a year or so, it is not at the time of writing and as a result a bridge was designed. A better name for the specific bridge developed for this implementation is perhaps *interface*, because the interface is not a full bridge. Instead, it was developed to expand on the concept of what is needed for the graphic system and to optimize this. In this way the interface is not a strict Wishbone-validated interface, but close to.

5.2.3 VGA-block

Introduction

The OpenCores VGA/LCD ControllerIP-core provides VGA capabilities for embedded systems and is written by Richard Herveille. It supports CRT with user programmable resolutions and video timings. The video memory is suitably located

outside the primary IP-core. The horizontal, vertical, and composite synchronization polarization levels, as well as the blanking polarization level are programmable by software. There is support for a maximum of two Hardware cursors.

Implementation

A Wishbone Slave interface manages all accesses to user readable/writable registers. The registers define the operation of the IP-core. A Wishbone Master interface manages all accesses to the external memory. It consists of a number of interacting state machines. The color processor and the cursor processor issue requests to the Wishbone Master. The Wishbone Master interface then generates the memory addresses for the image and the cursors. The dual-clocked Line FIFO ensures a continuous data stream toward the CRT display and ensures a correct transformation from the Wishbone clock domain to, i.e., the system clock, and the pixel clock domain. The Color Processor translates the received pixel data to RGB color information.

The cursor buffers are 512x32 bit single clock synchronous static memories. Each buffer contains a copy of the current cursor pattern. Finally, the Color Lookup Table (or CLUT) is a 512x24 bit single clock synchronous static random access memory divided into two separate CLUTs of 256x24 bit each. Each color lookup table contains a 24-bit RGB value for each entry. The output from the color lookup table is the RGB data for the current pixel.

Problems and notes

While the IP-core is in overall very well written and well documented, it contains more functionality than what is necessary for the graphic system. It is also written in a very general way, supporting several color depths and resolution etc. There are also problems to use the IP-core in the Virtex-II FPGA with speed grade -4 at 100 MHz. One of the advantages of using OpenCores is obviously that the source code is available. To remedy the problems, the IP-core has been used as a starting point for rather big changes and optimizations resulting in a much smaller and specialized IP-core.

Important changes

- The support for multiple resolutions has been limited. The IP-core used 32-bit registers for storing the current x and y positions on the frame. The worst case scenario thus gives a ripple carry addition of 64-bits. By limiting each register to 10 bits(i.e. only these bits are used), and adding a wait state the potential critical path was eliminated. 10 bits allow a maximum resolution of 1024 * 1024 pixels.

- The support for multiple color depths has been removed. In both the slave and the master interfaces and as well in the color processor, each color depth case had to handled separately with extra logic and multiplexers. The color processor, when synthesized with Precision RTL, contained a nasty bug where

signals bundles were inverted ,i.e., [15:0] becomes [0:15]. Some time was spent to try to figure out how this could happen, but since the part with the bug could be removed when removing color depth support, the problem remained unsolved.

- The slave interface part of the IP-core contains all the control registers. They have been modified to reset to the 640*480 @ 60 Hz resolution and use a color depth of 16 bits. The slave can still be accessed and thus allowing changes to these values, but the previously mandatory 10 or more Wishbone writes to set up the system through the slave are now optional.

- Hardware cursor functionality was removed. The use of such cursors, might be interesting in an expanded version of the system. Its is likely that support for a lot more cursors are added then in a more complete sprite functionality.

5.2.4 GPU

The block is implemented with a CPU IP-core and complemented with a special memory clear unit. Depending on the chosen CPU IP-core a Wishbone interface might be needed.

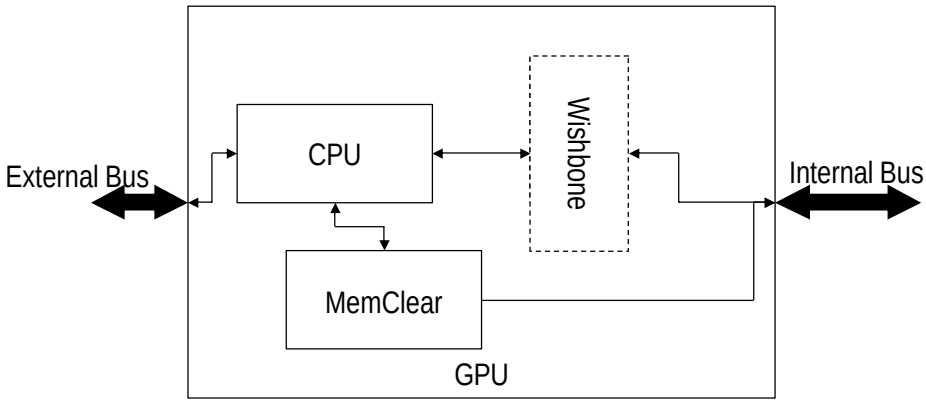

Figure 5.1. GPU implementation overview

Introduction

This block handles communication, through the external bus, with the other parts of the FPGA. The block also acts as a control unit for the system, starting the vga-block's cycles of memory reads for each frame and the memory frame data

write for clearing with background data. The block could be seen more as a state-machine than a devoted CPU running crucial code. Hence there is not a critical speed requirement as such in terms of instructions per second.

Implementation

There were three options available to implement the CPU:

- Use a CPU IP-core from OpenCores. There are numerous CPU IP-cores at OpenCores, but most likely at varying quality.
- Use a commercial IP-core such as Xilinx Microblaze.
  Although this options seems very interesting at first there are complications. The version of the IP-core available to the department at the time of writing it is not the latest version. The MicroBlaze does not use Wishbone but OPB. Using it would require a bus bridge
- Design a specialized CPU from scratch.
  This is the most desirable option and also the most likely one in a late version of the graphic system. It allows complete customization to suit the system. The extra time this would require will most likely exceed the time available to build this early prototype implementation.

For this prototype implementation, option number one is chosen, in hope to save precious project time. OpenCores hosts numerous CPU cores. About half of them feature a Wishbone interface. For this implementation, the three primary requirements are the ability to do 32-bit Wishbone reads/writes and that the IP-core is small and fast. None of the OpenCores CPU IP-cores meet all of these requirements. The only IP-cores even close to run at high clock speeds such as 100 MHz, are the really small 8-bit ones. Hence, the decision was taken to prioritize clock speed and size when choosing a 8 bit CPU IP-core. By adding some extra registers outside of the CPU IP-core a 32-bit Wishbone write can be accomplished even though the data bus for the 8-bit CPU IP-cores are only 8-bit. There are a number of different 8-bit CPU IP-cores available. Perhaps the most interesting one at first sight is the Wishbone Z80 IP-core.

Z80

The Wishbone Z80 from OpenCores is a remake of the classic Zilog Z-80 CPU. It is boosted with a Wishbone interface to enhance interconnectability and portability. The IP-cores was designed to operate efficiently with internal static RAM.

> Thus, a two stage pipeline is implemented to allow instruction execution at the access rate of a 32 kbyte RAM. This could be well over 300 Mhz. depending on implementation technology.

Sadly, when synthesized with Precision RTL for a Virtex-II, the IP-core does not come close to that clock speed. Even 100 MHz wasn't reached, meaning either alteration of the IP-core or using multiple clock domains would be necessary.

MiniRISC

The OpenCores Mini-RISC CPU IP-core is mostly compatible with the PIC 16C57 from Microchip. It was written by Rudolf Usselmann, who also wrote the CONMAX IP-core. The design claims to be fully software compatible with the Microchip Implementation of the PIC 16C57, except for some minor extensions. While the IP-core is very well written, it does not have a Wishbone interface. It will not run at 100 MHz as well but probably not much lower.

- A PIC compatible Microcontroller that runs a lot faster
- Separate (External to the IP-core) Program Memory
- Options to extend the IP-core

PicoBlaze

Using the Xilinx macro-processor PicoBlaze would seem to be a mix of all three options in section 5.2.4. Actually, even though it is designed by Xilinx it is a free to use IP-core and hence not commercial in this way. The advantage of using PicoBlaze is that it is very well documented and as well it has a company to back on it's performance. Furthermore it seems to be able to run at at clock speed of 100MHz yielding 50 MIPS! It is 8-bit and the instruction set is very limited. In fact, one of its more common uses is as a substitute for big state machines. This is possible mainly because the whole IP-core uses only 55 slices of space in a Virtex-II! Program memory is single BlockRAM.

The drawback is the obvious lack of a Wishbone interface. The IP-core has two 8-bit ports (where one is a kind of address port for the other) that must be used to build the full 32-bit Wishbone interface.

Problems and notes

In general, the implementation of the GPU block was successful and everything worked fine at the simulation level. However, when synthesizing the system strange problems surfaced. The whole system started to behave very strangely when the Picoblaze was used. Errors arose in parts of the design that the Picoblaze was not even connected to. The problem seem to lie within the process of synthesizing Picoblaze with Precision. For example, when doing a back-annotate, the primitive HDL descriptions were invalid, implying that Precision was not able to interpret the Picoblaze macro code correctly. At least not the memory contents. Without a functioning back-annotate, and a perfectly working RTL simulation, debugging was really difficult. Especially as the problems seemed very random. Later it was discovered that stability seemed to increase, the bigger the margin toward the estimated maximum clock frequency given by Precision. Nevertheless, the problem was not all together solved. As the debugging was so difficult it is sometime easier to just walk around the problem. During the debugging phase, a block to let

the CPU work at its own clock frequency was added. Hence the MiniRisc IP-core could be implemented as well without adding any new blocks. MPLab is a free development IDE from MicroLabs. Using the IDE, its possible to write and simulate assembly language for different types of PICs, including the 16C47 PIC. There are also free C compilers available, making the MiniRISC very nice to use. It was used successfully and replaced the Picoblaze. The direct advantage of using the MiniRISC over the Picoblaze is that a Xilinx FPGA is not required. The MiniRISC executes instructions with a varying number of clock cycles available. On the whole it is slightly slower then the Picoblaze.

5.2.5 GPU - Wishbone interface

Introduction

The Wishbone interface assumes the 8-bit processor has two 8-bit ports. One is used as an address port and one as data port. The processor also needs a write strobe signal and a read strobe signal. The interface updates the register the address port points to with the value on the data port on a rising edge on write strobe. The actual Wishbone cycle is starting by writing to the Control-Register. In table 5.1 each 8-bit register is associated with an address so that the 8-bit CPU can build the complete 32-bit registers.

With this method it might take 20 CPU instructions to set up the Wishbone register, but as soon as it is started the registers are shifted and saved so the processor can start updating the registers for the next read. Given that the CPU's sole responsibility is to do wishbone cycles, combined with the fact that a Wishbone burst cycle is often over 20 clock cycles, so there should not be a lot of performance decrease by using the small 8-bit CPU.

Address port	Target 8-bit Register	Register use in Wishbone cycle
x00	Data0	WB-DataRegBits 0-7
x01	Data1	WB-DataRegBits 8-15
x02	Data2	WB-DataRegBits 16-23
x03	Data3	WB-DataRegBits 24-31
x04	Addr0	WB-AddrRegBits 0-7
x05	Addr1	WB-AddrRegBits 8-15
x06	Addr2	WB-AddrRegBits 16-23
x07	Addr3	WB-AddrRegBits 24-31
x08	Cntrl	Control-Register
x09-xFF	N/A	Not used

Table 5.1. Wishbone interface registers

Implementation

Figure 5.2 shows a principle overview of the block.

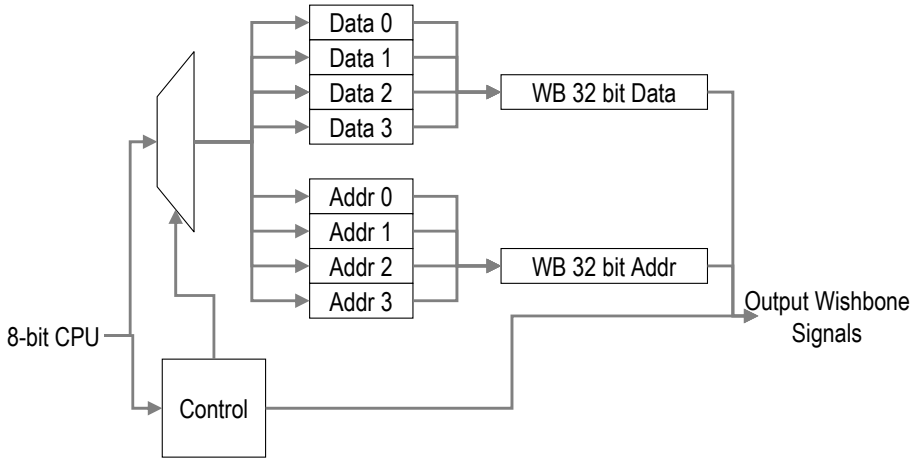

Figure 5.2. Wishbone 8-bit to 32-bit interface overview

Problems and notes

No problems occurred during implementation.

5.2.6 GPU - MemClear

Introduction

The memory clear unit's responsibility is fairly self explanatory. It clears a frame with background data. This is done using many memory write bursts. The memory is assumed to be stored in order and not split up.

Implementation

The target addresses of the memory bursts are increased linearly. Hence, the block can be implemented around two relatively big counters. It keeps track of the starting address and increment it with the burst length. In effect, the counter counts the number of bursts done and when the whole memory clear action is finished.

The second counter works more closely with memory and the Wishbone interface. It increments the address for each write within a burst and receives the starting address for each respective burst from the first counter.

Problems and notes

No problems occurred during implementation.

5.2.7 3D Pipeline

Introduction

This block is the only block in the graphics system that was not adequately implemented. The reason was lack of project time which is unfortunate. On the other hand future work on the system can focus on implementing the minimal or the full 3D-pipeline as specified in chapter 4. The integration test (see section 6.2) shows that the rest of the system can be used as a framework for a 3D-pipeline. It can even be used with most kinds of common image rendering techniques.

The development and the design of the system was not done one block at a time though, meaning that some experimental implementation of the 3D-pipeline was started. For example, the Matrix multiplier which is the main component of the Viewport transformation in the minimal pipeline, is detailed below.

3D pipeline - Matrix multiplier

Introduction

In section 2.1 it was said that a vector based mathematical description of the objects in a scene is very effective. The object is normally defined with its vectors and local basis. When the 3D pipeline translates, scales and rotates the objects to its various different forms the transformation can be done using matrix math. This block performs this multiplication.

Implementation

In the 3D pipeline each vertex, represented as a vector composed of four floating point numbers, is multiplied by a standard viewing matrix. This matrix translates, scales and rotates the camera to achieve the sensation of a moving camera. Moving the whole world instead of moving the camera to change the way the camera views the world might seem a bit odd, but mathematically its just another way of representing the phenomena. By updating the viewing matrix once before each frame is rendered, the effect of a moving camera or viewer can be achieved.

As said, each vector is represented by four floating point numbers. The vector has to be multiplied by the viewing matrix, which in term is represented by 16 (4 by 4) floating point numbers. The result is a new vector which represents the old vector translated, scaled and rotated. The implementation given here is based around some heavy pipelining. A matrix multiplication has to be seen as a rather

complex operation and if it was to be done all at once in one or two clock cycles, the resource requirements would be huge. The numbers has to be represented as floating point since the size numbers can vary enormously. A 16-bit floating point number is perhaps the smallest number of bits that can be used. Smaller sizes reduce the resolution of the floating point number too much for a satisfying result. With 16 matrix values, 16 bits each, that equals 256 bits of data. Size soon becomes a problem. The work around lies with pipelining and resource sharing. The actual calculation can be seen as 16 multiplications and 12 additions (or 16 if seen as an addition with 0).

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} xm_{00} + ym_{01} + zm_{02} + wm_{03} \\ xm_{10} + ym_{11} + zm_{12} + wm_{13} \\ xm_{20} + ym_{21} + zm_{22} + wm_{23} \\ xm_{30} + ym_{31} + zm_{32} + wm_{33} \end{pmatrix}$$

To allow for the resource sharing and pipelining the matrix multiplication is done in four steps. The resulting vector R with the four floating number registers $r_{1...4}$ is assumed to be cleared, and then with each step one row is used to accumulate the result in the registers. Figure 5.3 depicts the implementation.

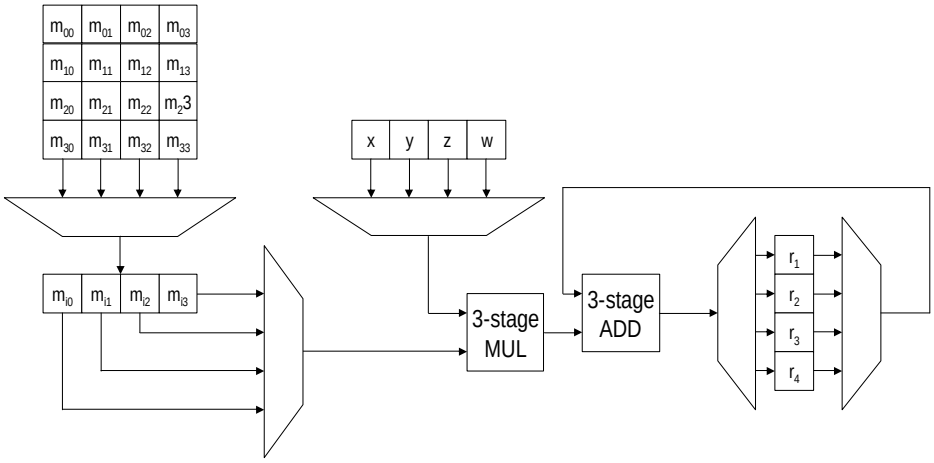

Figure 5.3. Pipeline and resource share implementation of the 4x4 matrix with 4x1 multiplier.

Problems and notes

By pipelining each floating point operation, finally an operational clock speed of over 100 MHz could be reached. The two operations are done in three pipeline

steps each and a new operation can be started each clock cycle. There are 16 multiplications and 12 addition to be done. Since there is only one multiplier at least 16 cycles is required for the floating point operations. Each multiplication result has a delay of three cycles and all of them has to be added to the result register, which in turn has another three cycles delay. As as result, a full matrix multiplication operation is 22 (16+3+3) clock cycles long. This is satisfying given that it was successfully implemented at well over 100 MHz. 100 MHz / 22 is around 4500000 vertex transformations per second. At 60Hz frame rate, 75000 vertexes can be processed for each frame. This will not be the limiting factor of the performance of the pipeline. Very complex worlds can be achieved with as few as 200-300 vertexes.

5.2.8 Circuit Board design

As stated previously, to connect the FPGA with the DACs (they are not on the development board) and the DACs with the CRT screen, a P160 prototype module was used. Figure 5.4 is taken from the ADV7120 data sheet[11]. It shows how the ADV7120 should be connected to minimize noise. This is especially important when having digital circuits on the same board. The circuit was built more or less exactly as suggested by Analog Devices. No soldering was used when the first integration test (see section 6.2) was done. Instead all connections were wired on, leaving a rather messy look (see figure 6.1). The reason was of course that had soldering been used, a new prototype board would be required to change the design, should it become necessary. There is also the matter of heat from the soldering process that might hurt the ADV7120 circuit. Nevertheless, the image on the CRT was close to rock solid. Table 5.2 lists components used.

Component	Description
C1	33mF Tantulum capacitor
C2	10mF Tantulum capacitor
C3, C4, C5, C6	0.1mF Ceramic Capacitor
L1, L2	Ferrite bead
R1, R2, R3	75 Ω 1% Metal Film resistor
R_{set}	560 Ω 1% Metal Film resistor
Z1	1.235V Voltage reference

Table 5.2. Blocks tested during the Integration test

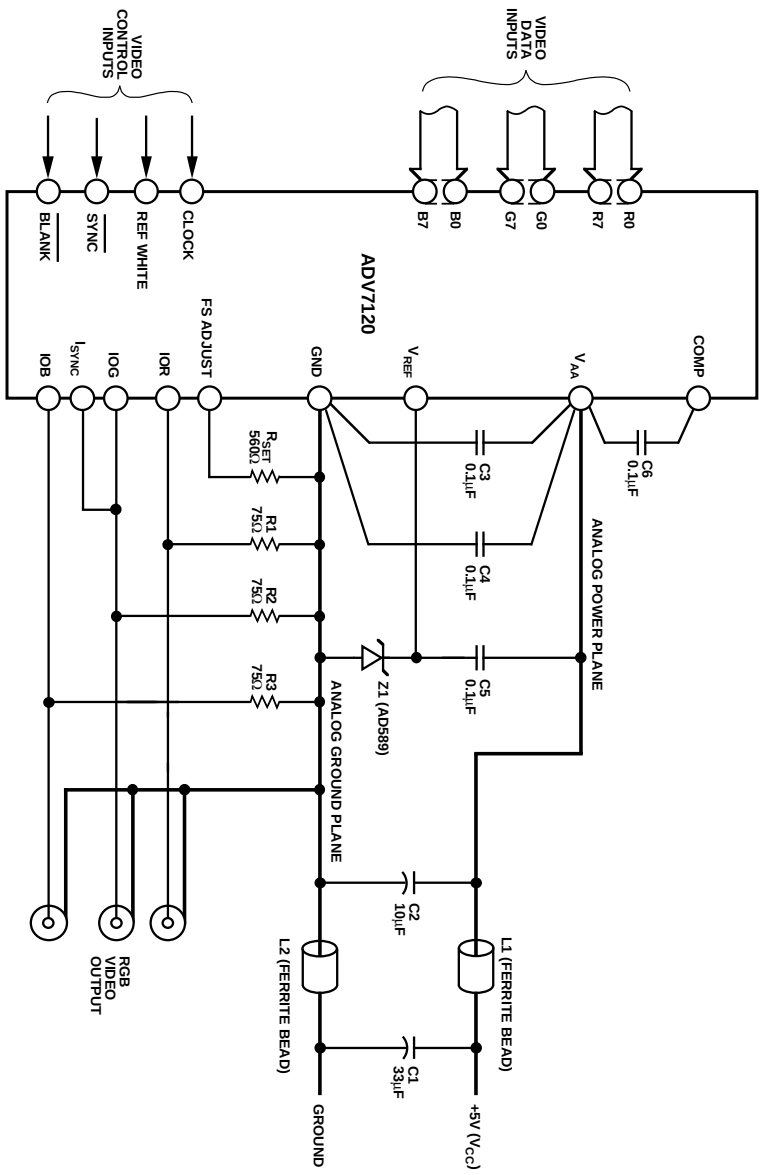

Figure 5.4. Overview of connections and discrete components around the ADV7120.

Chapter 6

Testing and Analysis

Throughout the implementation phase a respectable amount of testing and calculation was done to ensure proper function and performance of the system. This chapter documents the largest of the tests.

6.1 High-level MATLAB simulation

The purpose of this test was to ensure, at an early stage, whether or not some of the IP-cores from OpenCores actually did function.

6.1.1 The test

The IP-cores are the expanded DDR-controller and the VGA-unit. Added to the DDR-controller was now the customized Rev. B3 Wishbone-interface. The VGA-unit could hence in theory read its bursts of video data from the DDR chip. The principle of the test, in three steps, can be seen in table 6.1. The test unit is implemented in VHDL, but is not to be synthesized in hardware. For example it contains the functions that access a Jpeg from disk. After the simulation has been

Step	Test unit(Simulation VHDL)	VGA unit	DDR controller
1.	Converting Jpeg into VHDL data	Disabled	Writing full frame of VHDL data(the Jpg)
2.	Idle	Disabled	Idle
3.	Converting output R,G and B data into MATLAB data	Reading in memory	Reading full frame of R,G and B data

Table 6.1. High level memory function test

run, a separate MATLAB script can read the R,G and B data and assemble a new Jpeg image. Obviously, if the test was a success, the new Jpeg looks the same as the Jpeg that was written to the memory in the first step. The test uses a VHDL simulation model for the memory chip.

During phase 1, burst writes are tested to ensure the proper function of the write part of the Wishbone-interface and the actual DDR-controller. During phase 3, burst reads are tested in a similar way, again to ensure the proper function of the Wishbone-interface and the actual DDR-controller. Now, the data is also used by the VGA-unit. The data, is stored in a FIFO temporarily inside the VGA-unit to be used at the correct time when the actual pixel that the data represents is to be displayed. The test hence rigidly tests the DDR-controller, the Wishbone-interface and the VGA-unit.

6.1.2 Results

After some rather time consuming scripting, the test was possible to be executed. On the SUN workstation that the test was run on, the simulation time in ModelSim for a full frame was just over 4 hrs. The test Jpeg (the classic Lenna bitmap) appeared, making the test successful. At this relatively early point in the implementation it was reassuring to have a system level test verifying the design and the function of some parts of the IP-core.

6.2 Integration test

6.2.1 Introduction

The largest test that was carried out in the project was called the integration test. The reason for this is simply as its purpose was to integrate and employ as many parts of the complete graphic system, thereby achieving a limited system. The time budget, optimistic at first, was almost used up. Completely assembling the system and actually connecting all the blocks in the FPGA was no small feat. The compensation was to verify that what was done so far actually worked. To enable further work on the project by other developers, the integration test was mandatory.

6.2.2 The test

To do the test the hardware was assembled according to section 5.4, the CPU block, the memory controller and the BusCop block were all more or less finished and could be used in the test. The 3D pipeline was only designed in theory, so two counters were developed into a very simple horizontal line drawer. In this way, a very inactive 3D pipeline could be simulated. In various configurations, and by using the push buttons and switches available on the FPGA various simple horizontal line based digital images could be rendered on a VGA screen. What is important to realize is that with each second, 60 frames were drawn on the screen.

For each pixel on every frame, the correct color value in 16-bit format was read from the DDR-memory. The system was thus very active indeed even though the still image on the screen might fool an observer into something else. Table 6.2 lists the most important blocks and how rigorously they were tested during the integration test. Column two very briefly summarize what was expected of the block and column three gives an estimate of to what extent did the integration test evaluate the features of the block. This is in relation to how much of them are implemented. Thorough means the all functions were tested. Some means most functions were tested but more testing is needed to employ all the capabilities of the block. Blocks tested less than this are not included in table 6.2.

Part of system	Functions tested	Extent
BusCop	Arbitration and crossbar switch, both forced and priority mode.	Thorough
DDR-core	Normal and burstdata transfers. Both reads and writes.	Thorough
Wishbone DDR	Normal and burst data transfers. Both reads and writes	Some
Mod. VGA-core	Full operation at design resolution/color depth. Act as master on bus to read frame pixel data.Act as slave on bus to receive commands.	Thorough
8-bit CPU	Execute instructions in dedicated program memory (block ram) High speed operation of the wishbone interface.	Some
CPU Wishbone interface	Act as master on bus to write to several blocks.	Some
MemClear	Act as master on bus to write frame pixel data.	Thorough

Table 6.2. Blocks tested during the Integration test

Similarly, the hardware listed in table 6.3 were used.

Hardware	Part	Extent
The FPGA development board	DDR-memory	Thorough
	JTAG	Thorough
	P160 prototype and socket	Thorough
	Oscillators	Thorough
	Voltage regulatiors	Thorough
VGA-screen connection		Thorough
ADV7120 DACs		Thorough
Discrete components		Thorough
Virtex-II	BlockRams	Thorough
	DCMs	Thorough

Table 6.3. Hardware tested during the Integration test

6.2.3 Results

A lot of time was spent assembling and conducting the test, but the results were rewarding. Once the inevitable bugs were removed the systems worked as expected. In a coarse way, most of the system had been tested to verify that high-speed graphics is possible with FPGAs. As the system was set up, the memory was idle for long periods each frame, ideal for some drawing to update the frame, achieving high-color 60 frames per second graphics. The main test result in a designer's point of view was perhaps, it was time to start to implement a proper 3D-pipeline. In figure 6.1 the system assembled during the implementation test can be seen.

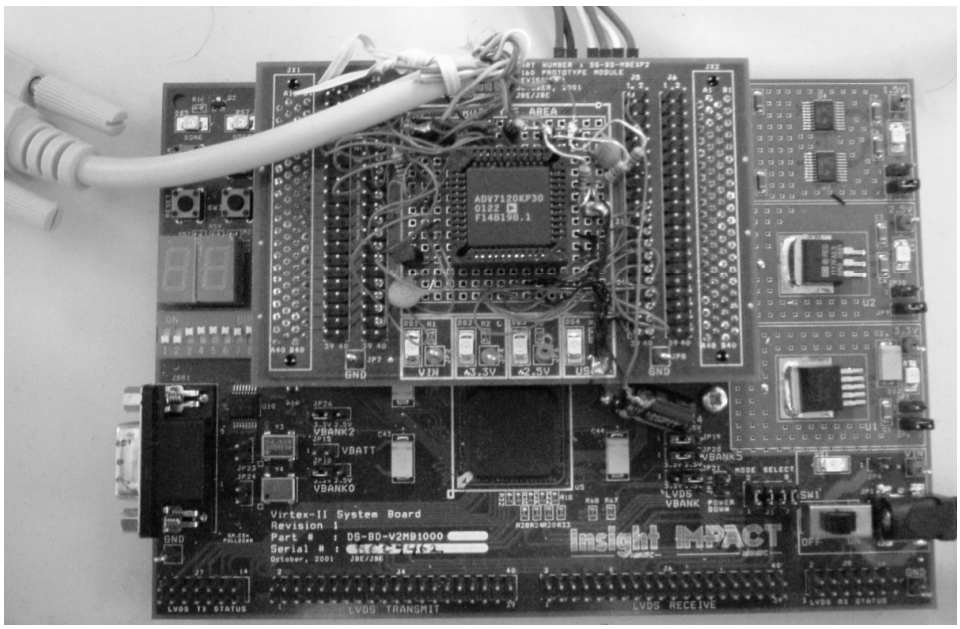

Figure 6.1. The ADV7210 on the P160 prototype board, with discrete components and standard 15-pin connection to VGA-screen.

Chapter 7

Conclusion

GPUs benefit from pipeline programmability. FPGAs provide a unique middle way between software and hardware, as the hardware itself can be configured as a processor where the software can run. One of the aims of this project was to investigate how this can be used for graphic system design. After theoretical design and an investigating implementation, the experience is that FPGAs can be used rather effectively for graphic system design. The key to this lies with being able to use the advantage of implementing in hardware were preferable over software, but also since the graphic system's performance is limited to the slowest link of the chain, the capability to assign the FPGAs resources to where it is needed the most is important. During the implementation phase of this project, this advantage was used constantly to reach various performance goals.

Entwined with this aim was the purpose of implementing a simple graphics system on one of the division's development boards. An incomplete system was implemented and the Memec development board with a Xilinx Virtex-II FPGA was used. The actual implementation development was satisfying in terms of performance and function, but unsatisfactory when it came to progress pace. The inexperience of the author when it came to estimation of the implementation time is the prime reason that the project does not include a complete, implemented and functioning graphic system. Nevertheless conclusions regarding the development board and the FPGA can be drawn. The Virtex-II version used meets the high speed and resource demands to host a quite complex and able graphic system. As said, the capability to assign the Virtex-II's resources to where it was needed the most was critical.

During the project, it was also found that synthesis tools from different vendors other than that of the FPGA can impose problems. Generally, a frustrating part of the problems that did occur could be blamed on the tools, meaning that the developer was not really to blame. When these problems arise the developer is hopelessly in the hands of the tools, which can be a rather discouraging experience.

Another objective of the project was to use and evaluate IP-cores in the design the graphic system. At a relatively early phase of the project, this purpose

was complemented with the more strict rationale of evaluating IP-cores from the OpenCores Internet community. It was found that, even though the principle of free IP-cores is very compelling, the fact that nobody guarantees the quality of the IP-cores severely limits the IP-cores' uses. This is perhaps the most certain and important conclusion of the project. When using an IP-core from the Internet as such the designer is taking a big risk since the function is by no means assured by anybody. Thus the function can not be taken for granted. Some of the IP-cores tried in this project were full of bugs or simply faulty, even though they were marked as done. The Internet community of OpenCores deserves all the support they can get though.

7.1 Future work

By continuing the work begun in this project done here a more complete graphic system can most likely be achieved. The framework around the 3d-pipeline is truly essential for the systems performance, and a lot of attention was put on this in the project. The BusCop block is a very good starting point for an advanced bus arbiter. The Wishbone burst accesses use the available memory effectively to push performance possibilities high. By using the framework developed, a developer can focus on the 3D pipeline block. Maybe even a master thesis project like this can be based around the 3D pipeline.

By expanding the development board with a memory board (3 memory chips or more) would improve the performance immensely, opening for much higher resolutions and deeper color-depth.

Some sort of interaction with the graphics system would increase its potential use.

More work could be done to evolve some parts developed from scratch into full, general IP-cores. These in term can be uploaded to the OpenCores community. The BusCop arbiter and the wishbone interfaces for the DDR-controller and 8-bit CPUs are primarily referred to.

Continued evaluation of open source IP-cores from OpenCores or any other source is always feasible and can contribute a lot to the community. One must never forget that a community can not evolve unless its members contribute.

The principle of open source hardware is still very new and not even close to where the software equivalent is today in terms of use, availability, number of users or just simply age. By enhancing the forum capabilities of the community, users can document how they used the IP-core and how it function and, most import, make other potential users aware of the IP-core's quality. As the communities grow, more quality IP-cores will appear. A pleased IP-core user is more likely to contribute. With interest will hardware designers all over the world observe how the community evolves, and the author is one of them.

Bibliography

[1] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[2] James C. Leiterman. *Learn Vertex and Pixel Shader programming with DirectX 9*. Wordware Publishing Inc., 2004.

[3] J.F. Wakerly. *Microcomputer architecture and programming*. John Wiley & Sons, 1981.

[4] Stefan Sjöholm and Lennart Lindh. *VHDL för konstruktion*. Studentlitteratur,Lund, 1999.

[5] James R. Armstrong and Gail F. Gray. *VHDL Design Representation and Synthesis*. Prentice-Hall Inc., 2000.

[6] The opencores internet community.
`http://www.opencores.org`, [DEC 2004].

[7] James T. Kajiya. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

[8] Edward Angel. *Interactive Computer Graphics*. Pearson Education, 2003.

[9] Virtex-ii microblaze development kit product brief.

[10] P160 prototype module user's guide.
Version 1.0 February 2002.

[11] Datasheet, analog devices triple 8-bit dac adv7120.
`http://www.analog.com/UploadedFiles/Data_Sheets/173587347adv7120.pdf`, [DEC 2004].

[12] Vhdl & verilog compared & contrasted plus modeled example written in vhdl, verilog and c.
`http://www.bawankule.com/verilogcenter/verilogvhdl.html`, [DEC 2004].

[13] Rochit Rajsuman. *System-on-a-Chip: Design and Test*. Artech House signal processing library, 2000.