CHALMERS

Design and Implementation of a 2D Acceleration engine for a Video Controller

Master of Science Thesis in Electrical Engineering

BJÖRN FAGNER
MARCUS GUSTAFSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, November 2009

Design and Implementation of a 2D Acceleration engine for a Video Controller

BJÖRN FAGNER
MARCUS GUSTAFSSON

Examiner: ARNE LINDE

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden November 2009

Abstract

The market for small, embedded systems is growing exponentially. More functions are required of the systems so the designers need to find new solutions and different approaches for the products to keep up with demands. One way to make systems faster is to introduce specialized cores. This allows the CPU to delegate workload, while it proceeds with other tasks.

Aeroflex Gaisler AB has developed a system-on-chip solution which Linux can run on. However the rendering of graphics is putting a large burden on the processor. This project has designed and implemented an IP core which will relieve the CPU from rendering 2D graphics. The accelerated operations are fill rectangle, copy area and image blit.

The work has resulted in an acceleration of the framebuffer operations by between 10 to 40 times on average. Regardless off this acceleration the operations will be performed in parallel while the CPU executes other instructions, which is an acceleration in itself.

The accelerator is limited to the color depths of 8, 16 or 32 bits per pixel and a maximum resolution of 1024x768 pixels.

Sammanfattning

Marknaden för små, inbyggda system växer exponentiellt. Mer funktioner krävs av systemen vilket gör att utvecklare måste hitta nya lösningar och angreppssätt för att produkterna ska möta efterfrågan. Ett sätt att öka systemens prestanda är att introducera specialiserade kärnor. Det tillåter processorn att delegera arbete medan den arbetar vidare med andra uppgifter.

Aeroflex Gaisler AB har utvecklat ett system-på-kisel vilket kan köra Linux. Rendering av grafik lägger dock stor belastning på processorn. Detta projekt har utvecklat en IP-kärna som avlastar processorn vid rendering av 2D grafik. De accelererade funktionerna är fill rectangle, copy area och image blit.

Arbetet har gett en acceleration av framebufferoperationerna med mellan 10 och 40 gånger i genomsnitt. Oberoende av denna acceleration kommer operationerna att utföras parallellt med att processorn exekverar andra instruktioner, vilket är en acceleration i sig.

Acceleratorn är begränsad till ett färgdjup på 8, 16 eller 32 bitar per pixel och en maximal upplösning på 1024x768 pixlar.

Acknowledgement

This thesis has been both interesting and fun to work on. Our thanks to the whole staff at Aeroflex Gaisler AB for the opportunity and for the support during project. Special thanks to our supervisor Jiri Gaisler, and also to Jan Andersson and Daniel Hellström. They have given us great input and help during the development process. We would also like to thank our examiner at Chalmers, Arne Linde, who has shown an interest in the work and been helpful throughout the whole project.

Göteborg, 2009

Björn Fagner

Marcus Gustafsson

Table Of Contents

Definitions and Abbreviations

2D	Two dimensional.
AHB	**A**dvanced **H**igh performance **B**us. A high performance protocol introduced in AMBA 2.0.
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture. An on-chip communication standard for high performance embedded microcontrollers.
APB	**A**dvanced **P**eripheral **B**us. A protocol for low power peripherals with reduced interface complexity, part of the AMBA specification.
BLIT	**BL**ock **I**mage **T**ransfer. A computer graphic operation which produces images from compressed source data.
BPP	**B**its **P**er **P**ixel.
DMA	**D**irect **M**emory **A**ccess. Allows certain hardware subsystems within the computer to access system memory.
DVI	**D**igital **V**ideo **I**nteractive. A multimedia desktop video standard.
FPGA	**F**ield **P**rogrammable **G**ate **A**rray. A chip containing reconfigurable logic.
ID	**ID**entity.
GPL	Gnu **G**eneral **P**ublic **L**icense. A free software license.
IP core	**I**ntellectual **P**roperty **core**. A reusable unit of logic design or layout.
JTAG	**J**oint **T**est **A**ction **G**roup. A connection used for debugging integrated circuits or as a probing port.
MAC	**M**edia **A**ccess **C**ontrol.
PCI	**P**eripheral **C**omponent **I**nterconnect. A standard expansion bus in computers.
PS2	**P**ersonal **S**ystem/**2**. A standard serial data bus used for keyboards and mice.
ROP	**R**aster **Op**erations. A computer graphic operation that defines how existing destination data combines with new color data.
RS232	**R**ecommended **S**tandard **232**. A standard serial data bus.
SOC	**S**ystem **O**n a **C**hip. Refers to an electronic system that are integrated in to a chip.
USB	**U**niversal **S**erial **B**us.
VGA	**V**ideo **G**raphics **A**rray. A common computer graphic standard.
VHDL	**V**HSIC (Very High Speed Integrated Circuit) **H**ardware **D**escription **L**anguage
XOR	e**X**clusive **OR.** A logic operator.

1 Introduction

The market for small, embedded systems is growing exponentially. More functions are required of the systems so the designers need to find new solutions and different approaches for the products to keep up with demands. One way to make systems faster is to introduce specialized cores. This allows the CPU to delegate workload while it proceeds with other tasks.

The Aeroflex Gaisler LEON3 SPARC V8 processor is distributed as part of the GRLIB IP library and can be used for system-on-chip design. It has support for a special version of the Linux distribution SnapGear which is provided by Aeroflex Gaisler AB. The system has a VGA Controller Core which is used to run X on top of SnapGear on the LEON3. However, currently all rendering has been done by software, putting a relatively large burden on the system processor. A 2D graphics accelerator would relieve the processor of rendering operations and allow it to perform other tasks instead. This work is an implementation of a AMBA interface Plug&Play IP core with the goal to complement the GRLIB IP library with its addition.

1.1 Description of Task

The object of the project was to read and understand the algorithms of the framebuffer operations in the Linux video driver and then recreate the algorithms in a IP core using the hardware description language VHDL. The existing VGA Controller Core that is handling the framebuffer is limited to a pixel depth of 8, 16 or 32 bits, this also seemed adequate for the new core.

A number of issues are addressed during this project. An AMBA compatible interface is needed for both memory access and operation command calls from the CPU. The software driver will need to be altered or rewritten to accommodate the hardware calls. The software algorithms will not be optimal for hardware implementation which means that they will have to be rewritten. Also the different color depths and resolutions might need adaptation in the hardware.

1.2 Outline of Thesis

The first chapters present technical background and introduction to existing technology used while working on the project. Chapters 3 and 4 gives the reader insight to the work process. The end product of the project is described in Chapter 5 where the reader will be presented the full system and the sub-components. This is followed by Chapter 6 which describes the tests performed to verify the functionality and performance of the accelerator core. The results of the synthesis and performance tests are then presented in Chapter 7. Finally the report is summed up in conclusion, discussion and future development in Chapters 8, 9 and 10.

2 Technical Background

This section will introduce the reader to existing technologies used to complete the project and to the environment in which the IP core will come to function.

2.1 Framebuffer

A framebuffer is a video output device that drives a display from a memory buffer which contains a full frame, a representation of what is to be put on the screen. The data in the buffer is typically color values that describes every pixel to be displayed on the screen.

2.1.1 Framebuffer Operations

When the data in the framebuffer needs to be modified a framebuffer operation is performed. There is a multitude of framebuffer operations that make changes on the screen easier to perform. The three most common, which have an impact on CPU performance and are a minimum requirement for acceleration, are:

- Fill rectangle — A rectangle area on the screen is filled with a color. The operation uses two different Raster Operations, ROP. The area can be filled with or without regard to the original color of the destination pixel. If consideration to original color should be taken, the new color pattern and the original data is combined using the logic operator XOR, otherwise the original data is overwritten.
- Copy area — A rectangle area is copied from one part of the screen to another. If the areas overlap the copying might have to be done in reverse depending on whether the source data will be overwritten before or after it is read.
- Image blit — An image is written in the framebuffer area, the image is produced by source data fetched from the system memory. There are two kinds of image blits, monochrome and color. In the monochrome image blit every bit corresponds to a foreground or a background color of a pixel. In color image blit every byte of image data corresponds to a color of a pixel.

Our system uses packed pixel framebuffer organization, this means that the data for each pixel data is grouped together and is lined up consecutively (contiguously) one after another from the memory start address of the framebuffer to the last byte.

2.2 GRLIB

This section is a short introduction to the GRLIB IP Library.

2.2.1 Overview

The GRLIB IP Library is a set of reusable IP cores written in VHDL and designed for system-on-a-chip development. The cores are centralized around a common on-chip bus with the LEON3 as CPU. Examples of additional cores in the library are 32-bit PCI bridge with DMA, USB-2.0 host and device controllers, 10/100/1000 Mbit Ethernet MAC, and VGA Controller Core. It is developed and maintained by Aeroflex Gaisler AB and is available under the GNU GPL license [3]. An illustration of a template design can be found in Figure 1.

A short introduction to the features of GRLIB will follow. For more documentation on the GRLIB IP library and available cores, refer to the GRLIB User's Manual [3] and the GRLIB IP Cores Manual [2].

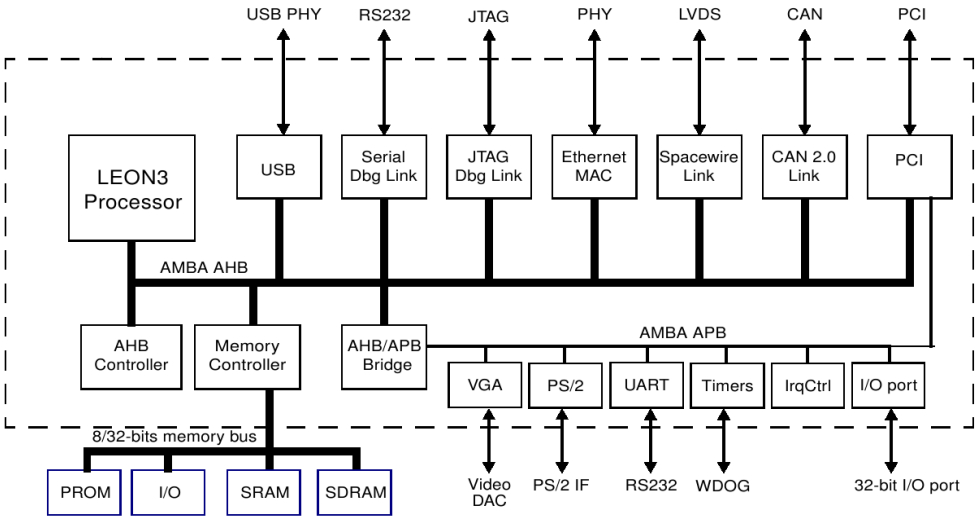


Figure 1: LEON3 Template Design [3].

2.2.2 LEON3

The CPU of the GRLIB system is the LEON3 32 bit synthesizable processor based on the SPARC V8 architecture [7]. It is available in several versions and is highly configurable with an advanced 7-stage pipeline, high

performance IEEE-754 FPU, multiprocessor support and more [6]. Figure 2 shows a block diagram of a LEON3 core configuration.

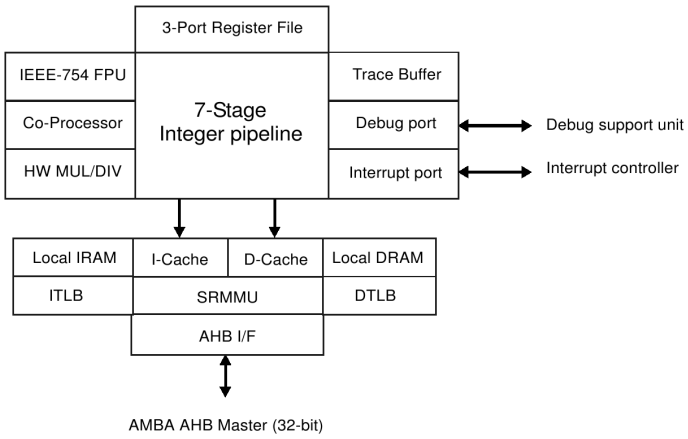


Figure 2: LEON3 Processor Core Block Diagram [5].

2.2.3 Plug&Play

The Plug&Play concept of the GRLIB system is an expansion of the AMBA 2.0 Specification [1]. It should be interpreted in the broad sense that the system hardware can be detected and identified through the software, which thereby can be configured automatically to match the underlying hardware.

In GRLIB the Plug&Play information consists of three items:

- A unique IP core ID.
- AHB/APB memory mapping.
- An interrupt vector.

This information is sent as constant vectors from the components that is connected to the bus to the arbiter/decoder where it is stored in a small read-only area accessible for all AHB masters through standard bus cycles. The configuration words are defined as shown in Figure 3. There are eight 32 bit words where four contain configuration words defining the core type and interrupt routing The other four, defining the memory mapping, are called ‘bank address registers’ (BAR) [3].

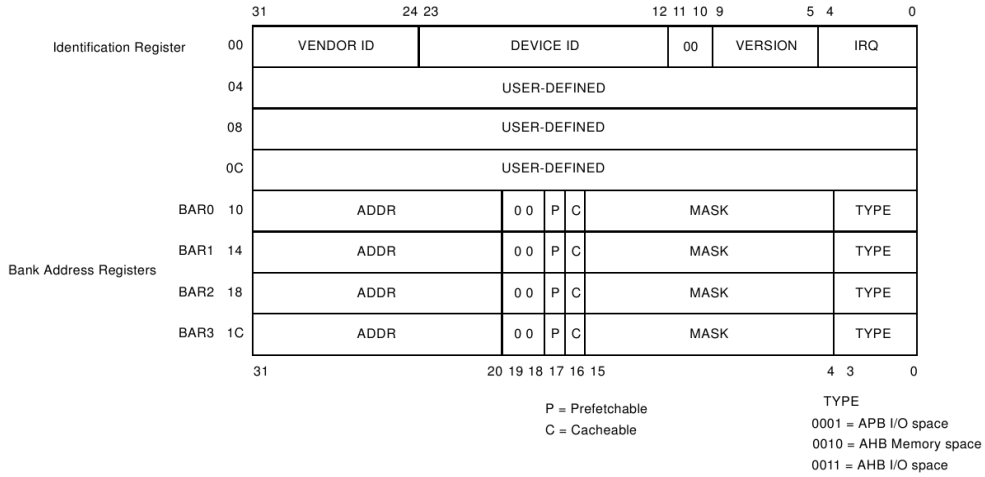


Figure 3: AHB plug&play information record [2].

2.2.4 GRMON

GRMON is a debug monitor for LEON processors and SOC IP cores based on GRLIB IP library. It is communicating with the LEON debug support unit (DSU) and allows non-intrusive debugging of the whole target system. GRMON supports the following functions [6]:

- Read/write access to all system registers and memory
- Built-in disassembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Support for USB, JTAG, RS232, PCI, Ethernet and SpaceWire debug links

2.3 AMBA

The Advanced Microcontroller Bus Architecture (AMBA) protocol is a specification for on-chip buses developed by ARM Limited [1].

The AMBA 2.0 specification includes three different buses:

- Advanced High-performance Bus (AHB).
- Advanced System Bus (ASB).
- Advanced Peripherals Bus (APB).

2.3 AMBA

The GRLIB system uses a combination of two of them. The backbone bus is of AHB type and for low power peripherals the APB is used, accessed through a AHB/APB bridge connection, such a system is illustrated in Figure 4.

Although the implementation is AMBA 2.0 compatible it has been expanded with a unique Plug&Play method for both AHB and APB, which allows users to configure and connect the IP cores without the need to modify any global resources [3].

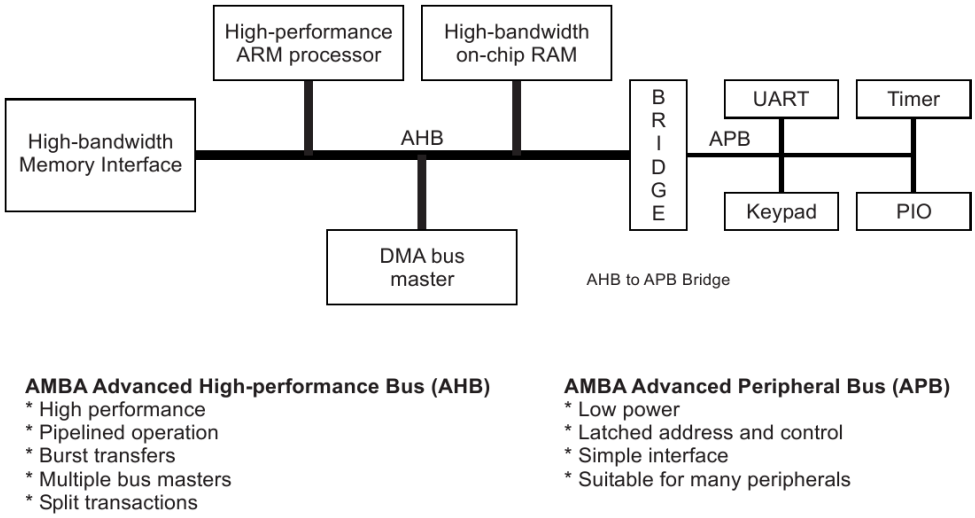


Figure 4: A Typical AMBA AHB Based System [1].

2.3.1 AHB

The AHB was developed to address the requirements of high performance, high clock frequency synthesizable designs and has several features required of such a system [1], including:

- Burst transfers.
- Single cycle bus master handover.
- Single clock edge operation.
- Wider data bus configurations (64/128 bits).

A typical system has the following AHB components:

AHB master A bus master is able to initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time.

AHB slave A bus slave responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.

AHB arbiter The bus arbiter ensures that only one bus master at a time is allowed to initiate data transfers. An AHB would include only one arbiter, although this would be trivial in single bus master systems.

AHB decoder The AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. A single centralized decoder is required in all AHB implementations.

2.3.2 APB

The APB is optimized for minimal power consumption and reduced interface complexity. It appears as a single AHB slave device which acts as a bridge module between the two buses. The bridge is the only master on the APB as the rest of the APB modules are slaves which allows for a simple interface with these specifications [1]:

- Address and control valid throughout the access (unpipelined).
- Zero-power interface during non-peripheral bus activity (peripheral bus is static when not in use).
- Write data valid for the whole access (allowing glitch-free transparent latch implementations).

APB master The APB bridge is the only bus master on the AMBA APB. In addition, the APB bridge is also a slave on the higher-level system bus.

The bridge unit converts system bus transfers into APB transfers and performs the following functions:

- Latches the address and holds it valid throughout the transfer.
- Decodes the address and generates a peripheral select, PSELx, which indicates which slave is being addressed. Only one select signal can be active during a transfer.
- Drives the data onto the APB for a write transfer.
- Drives the APB data onto the system bus for a read transfer.
- Generates a timing strobe, PENABLE, for the transfer.

APB slave APB slaves have a simple, yet very flexible, interface. The exact implementation of the interface will be dependent on the design style employed and many different options are possible.

For a write transfer the data can be latched at one of the following points:

- On the rising edge of PCLK, when PSEL is HIGH.
- On the rising edge of PENABLE, when PSEL is HIGH.

The select signal PSELx, the address PADDR and the write signal PWRITE can be combined to determine which register should be updated by the write operation.

For read transfers the data can be driven on to the data bus when PWRITE is LOW and both PSELx and PENABLE are HIGH. While PADDR is used to determine which register should be read.

2.4 SnapGear Linux

Aeroflex Gaisler AB has a specially developed version of SnapGear Linux that is supported for the SOC design around LEON3 [6]. This is the operating system that will run on the SOC, incorporating the project's IP core, and the environment in which the project will be run and tested. This is also where the software drivers, from which the hardware algorithms are derived, can be found. These software framebuffer operations and the driver for the GRLIB VGA Controller Core are described in this chapter.

2.4.1 Fill Rectangle (cfbfillrect.c)

This is a generic algorithm to perform fill rectangle for framebuffers with packed pixels for any pixel depth [8]. It makes a pattern to match the color depth of the framebuffer and writes the color pattern to the destination address of the rectangle.

There are four subroutines, described below, and one of them is called for each row of the rectangle's height. Which one that is called depends on which ROP to perform and whether or not the number of bits per pixel and start address requires the pattern to be realigned for each word.

The called subroutine proceeds by calculating bitmasks used to handle leading and trailing bits, at the start and end of the row, if the pixel is not described by a full word. If there are any leading- or trailing bits, the existing pixel data at the destination is fetched and merged with the new data by using the bitmask. The created word is then used as the new pattern to write to the destination address. If the operation call is to write a single word, the first word is also the last and the two bitmasks needs to be merged together before they are applied to the data.

Subroutine: Aligned with ROP COPY

The leading bits, if any, are handled first. Then, if there are multiple words to write, most of them are written by a loop and then the trailing bits of the row, if any, are handled last. The program then returns from the subroutine and the destination address for the next row is calculated. After that, a new subroutine call is made and this procedure loops over the height of the rectangle.

Subroutine: Unaligned with ROP COPY

The leading bits, if any, are handled first, and then the pattern is realigned. This realignment is made after each written word in the subroutine. Then, if there are multiple words to write, most of them are written by a loop and then the trailing bits of the row, if any, are handled last. The program then returns from the subroutine and the destination address for the next row is calculated and the pattern is realigned to match the new row. After that a new subroutine call is made and this procedure loops over the height of the rectangle.

Subroutine: Aligned with ROP XOR

With the Raster Operation XOR the existing pixel data at each destination is merged with the new pattern via an XOR operation. This means that a read operation is added to the algorithm for each word to be written and the number of bus requests double.

The leading bits, if any, are handled first. Then, if there are multiple words to write, most of them are written by a loop and then the trailing bits of the row, if any, are handled last. The program then returns from the subroutine and the destination address for the next row is calculated. After that, a new subroutine call is made and this procedure loops over the height of the rectangle.

Subroutine: Unaligned with ROP XOR

With the Raster Operation XOR the existing pixel data at each destination is merged with the new pattern via an XOR operation. This means that a read operation is added to the algorithm for each word to be written and the number of bus requests double.

The leading bits, if any, are handled first, and then the pattern is realigned. This realignment is made after each written word in the subroutine. Then, if there are multiple words to write, most of them are written by a loop and then the trailing bits of the row, if any, are handled last. The program then returns from the subroutine and the destination address for the next row is calculated and the pattern is realigned to match the new row. After that, a new subroutine call is made and this procedure loops over the height of the rectangle.

2.4.2 Copy Area (cfbcopyarea.c)

This is a generic algorithm to perform copy area for framebuffers with packed pixels for any pixel depth [8]. There are two subroutines, but they could be split up in four, like fill rectangle. Which one is called depends on whether the area have to be copied in reverse due to overlap.

Before a subroutine is called the source and destination addresses are calculated, along with indexes to indicate word unalignment of the first pixel. Then the appropriate subroutine is called. This is done for each row of the area.

The called subroutine proceeds by calculating bitmasks used to handle leading and trailing bits, at the start and end of the row, if the pixel is not described by a full word. If there are any leading- or trailing bits the existing pixel data at the destination is fetched and merged with the source data by using the bitmask. The created word is then the new color data to write to the destination address. If the operation call is to write a single word, the first word is also the last and the two bitmasks needs to be merged together before they are applied to the data.

Subroutine: Bitcopy

The source and destination are tested for alignment. If they are aligned the leading bits are handled first. Then, if there are multiple words to write, most of them are written by a loop where a source word is read and written to the destination word for word. The addresses are increased after each write operation. The trailing bits of the row, if any, are handled last.

If the source and destination addresses are unaligned, the source data could be divided over two words. This means that the two source words has to be fetched, shifted and merged to align with the destination.

The leading bits are handled first. Then, if there are multiple words to

write, most of them are written by a loop where the source words are realigned for each destination word. Before each write to the destination, new source data is fetched and after each write operation the addresses are increased. The trailing bits of the row, if any, are handled last. They could require two source words as well.

Subroutine : Reverse Bitcopy

The source and destination addresses and indexes are recalculated to accommodate for the reversed order copy. The source and destination are tested for alignment. If they are aligned the leading bits, which actually are the last bits of the row, are handled first. Then, if there are multiple words to write, most of them are written by a loop where a source word is read for each destination word and written word for word. The addresses are decreased after each write operation. If there are trailing bits, which are the first bits of the row, they are handled last.

If the source and destination addresses are unaligned, the source data could be divided over two words. This means that the two source words has to be fetched, shifted and merged to align with the destination.

The leading bits are handled first. Then, if there are multiple words to write, most of them are written by a loop where the source words are realigned for each destination word. Before each write to the destination new source data is fetched and after each write operation the addresses are decreased. The trailing bits of the row, if any, are handled last. They could require two source words as well.

2.4.3 Image Blit (cfbimgblt.c)

The software image blit in Linux consists of three functions: fast, slow and color image blit. Fast and slow image blit are monochrome and the color image blit supports up to full 32 bits color images. All the blit functions writes rectangular images [8].

The monochrome image blit copies a monochrome picture from the system memory to the framebuffer area. The picture is compressed in the system memory and is a bitmap where every '0' represents background color pixel and a '1' represents foreground color pixel in the actual picture. In the software there are two versions of monochrome blit, slow and fast image blit. The slow subroutine is generalized and can do both blits, but if possible it is preferable to use the fast subroutine. This can be done if the picture data and placement call fulfill the requirements. The requirements for fast image blit is a color depth of 8, 16, or 32 bits per pixel and that the picture width is divisible by the number of pixels per word. Also, the screen line length has to be divisible by 4 and the beginning and end of the image rows needs to be word aligned. The typical usage of monochrome image blit is font handling.

The color image blit copies a color picture from the memory to the

framebuffer area. In this case the source data is compressed and if the color resolution is higher than four bits the function fetches an 8 bit word and uses the four least significant bits to address a fake palette (pseudo palette) that consist of 16 colors. The software continuously updates the fake palette to keep the colors current. The picture's color data must have same format as the data in the framebuffer.

Subroutine: Fast

The fast image blit fetches 8 bits of source data to address a table. There is one table for each color resolution. The 8 bpp table consists of 16 32 bit words and is addressed by a 4 bit word, half of the source data, and the addressed 32 bit word is written to the framebuffer memory. The 16 bpp table consists of four 32 bit words and is addressed by a 2 bit word, two bits from the source byte. The addressed 32 bit word is written to the framebuffer memory. The 32 bpp table consists of two 32 bit words and is addressed by one bit from the source byte, and the addressed 32 bit word is written to the framebuffer memory. After this operation the source word is shifted to get new source data. When the source word has run out of bits a new source data fetch is performed and the process is repeated until the end of the row. After reaching the end of the row a constant, of screen line length, is added to the destination address and a new source address is calculated by adding a constant. The function loops over the height of a rectangle.

Subroutine: Slow

Slow image blit fetches an 8 bit source word and starts by handling leading bits if the destination start address is unaligned. This is done by using a bitmask to mask the original framebuffer data at the 32 bit aligned start address. The write block loop shifts in data, one pixel at a time, every cycle and when a full 32 bit word is accumulated the word is written to the framebuffer memory. New source data is fetched when needed and the loop runs until the last word. Next step is to write possible trailing bits and pad the remaining bits with framebuffer data. When the end of the row is reached a constant, of screen line length, is added to the destination address and a new source address is calculated by adding a constant. The function loops over the height of a rectangle.

Subroutine: Color

Color image blit fetches an 8 bit source word and starts by handling leading bits if the destination start address is unaligned. This is done by using a bitmask to mask the original framebuffer data at the 32 bit aligned start address. The write loop shifts in data, one pixel at a time, every cycle and when a full 32 bit word is accumulated the word is written to the framebuffer memory. New source data is fetched every cycle and color data

from the pseudo palette is gathered by indexing the palette with the source data. Next step is to write possible trailing bits and pad the remaining bits with framebuffer data. When the end of the row is reached a constant, of screen line length, is added to the destination address and a new source address is calculated by adding a constant. The function loops over the height of a rectangle.

2.4.4 Gaisler Framebuffer Driver (grvga.c)

The framebuffer driver initiates and registers the GRSVGA VGA controller when Linux boots up. This is where the software framebuffer operations are linked to the framebuffer device. To make the driver use the accelerator instead of the software these operation calls needs to be redirected to the accelerator. In Chapter 5.3 the modifications made to patch in the accelerator are described.

2.5 Target Technology

Two boards were used to test the system. Information on the platforms are presented in this section.

2.5.1 GR-XC3S-1500

The GR-XC3S-1500 Development Board incorporates a 1.5 million gates XC3S1500 FPGA device from Xilinx Spartan-3 family. This board is a compact, low cost board developed by Pender Electronic Design GmbH in cooperation with Gaisler Research for evaluation of the LEON2 and LEON3/GRLIB processor systems [4]. A picture of the topside of the board can be found in Figure 5.

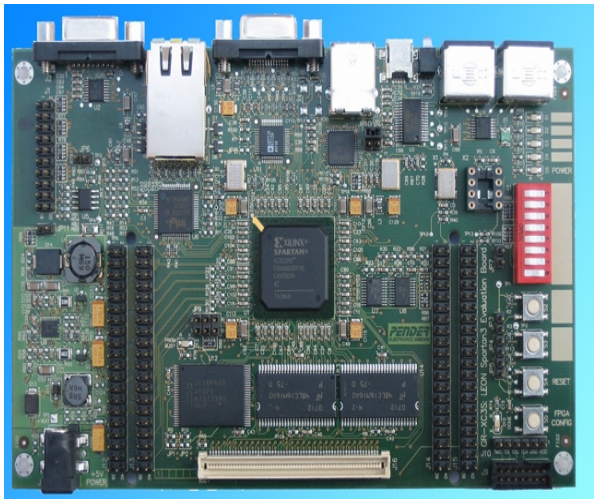


Figure 5: Topside view of the GR-XC3S-1500 Development Board [6]

2.5 TARGET TECHNOLOGY

The system features Ethernet, JTAG, USB, Video and PS2 interfaces for off-board communication and has on-board memory in the form of SDRAM and Flash memory.

2.5.2 ML501 Evaluation Platform

The ML501 Evaluation Platform from Xilinx sports a Virtex-5 XC5VLX50 FPGA with PS2, JTAG, USB, Ethernet interfaces, DVI video connector and much more [9]. The ML501 is a versatile and feature-rich low-cost development platform for multiple applications [10]. This board is well equipped to be the platform for the tests run under Linux. A picture of the topside of the board can be found in Figure 6.

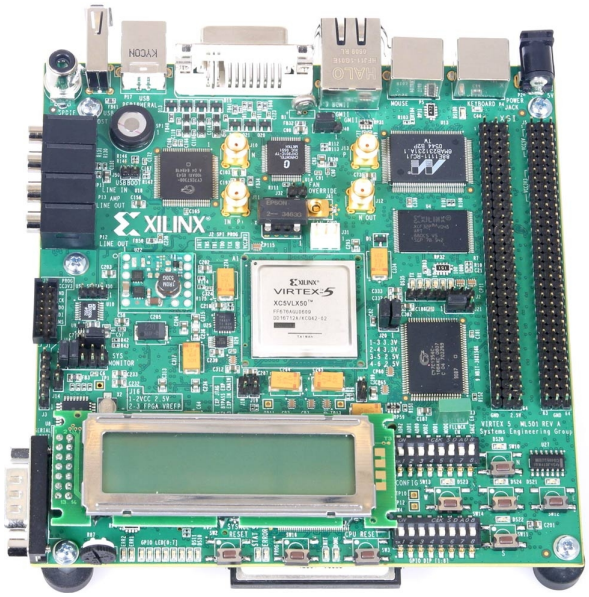


Figure 6: Topside view of the ML501 Evaluation Platform[10]

3 Development Process

The development process roughly followed the outline of the project proposal. The proposal can be found in Appendix C.

1. Development of a specification, defining which operations to be accelerated, supported resolution and color depth, register interface and DMA handling.
2. Implementing the 2D engine in VHDL, and verification in simulation.
3. Implementation on the Spartan3 GR-XC3S-1500 board
4. Testing of the accelerated functions using low-level C programs
5. Final testing of Linux-2.6 kernel with the X window system

The background of the GRLIB and AMBA system was studied as well as the function of the framebuffer operations. It was understood that the IP-core, to be implemented as a result of this project, needed interfaces to both the AMBA AHB and APB. The faster AHB was required for memory access, to read and write to the framebuffer, and the APB was more suited to relay the command and initial data needed from the LEON3 CPU.

The implementation in VHDL followed, this was verified by simulating the hardware in a test bench. Described in more detail in Chapter 6.

The VHDL was then synthesized and programmed on to the development board where it was tested and verified by using GRMON and low-level test programs in C. This is also described in more detail in Chapter 6.

Finally the Linux drivers were modified and the design tested under the X window system.

4 Design Choices

This section describes the limitations and the choices made during the design process.

4.1 Framebuffer Operations

The choice of which framebuffer operations to be implemented in the accelerator was made based on which operations that are most common. However, choosing which part of the framebuffer operations to implement in hardware was reevaluated several times during the development process. When smaller problems were solved, additional functions were added to the hardware implementation in iterations.

4.2 Resolution and Color Depth

The existing VGA Controller Core handling the framebuffer is limited to a color depth of 8, 16 or 32 bit which also seemed adequate for this project's IP core. The accelerator is also limited to a maximum resolution of 1024x768 pixels and resolution widths that are divisible by pixel per word. There is also the prerequisite that the framebuffer always starts aligned to an 32 bit word address. These limitation gives that each line of the screen is aligned by a fixed address increment for each supported resolution. This means that handling of row unalignment will not be necessary.

4.3 Optional Core

The accelerator is designed in two versions. The version that writes one row per call has the advantage that it is smaller than the one that writes the entire rectangle at once, but on the other hand it is slower and puts more load on the processor. In a small system with limited available area the version writing one row at a time might be preferable and in a large system the version drawing the whole rectangle should be faster since the processor does not have to call the module as often.

4.4 VHDL Coding Techniques

The design is written in the VHDL coding technique called the two-process design method [15]. This method ensures readable and efficient code, both for simulation and synthesis.

The design is also compartmentalized into smaller blocks which have a specific purpose. The blocks can be seen in Figure 7 and are presented in Chapter 5. By using a hierarchical design with a separate block, and a separate file to describe it, for each module of the core it was easy to divide the work. It also facilitated development, testing and readability.

5 Gaisler 2D VGA Graphics Accelerator

This chapter is intended as a guide to the source code of the hardware of the project. Also the changes to the framebuffer driver in Linux are addressed.

5.1 Overview

This section briefly describes the contents of each entity block in the design. The reader should get a basic understanding of the function and structure of the design.

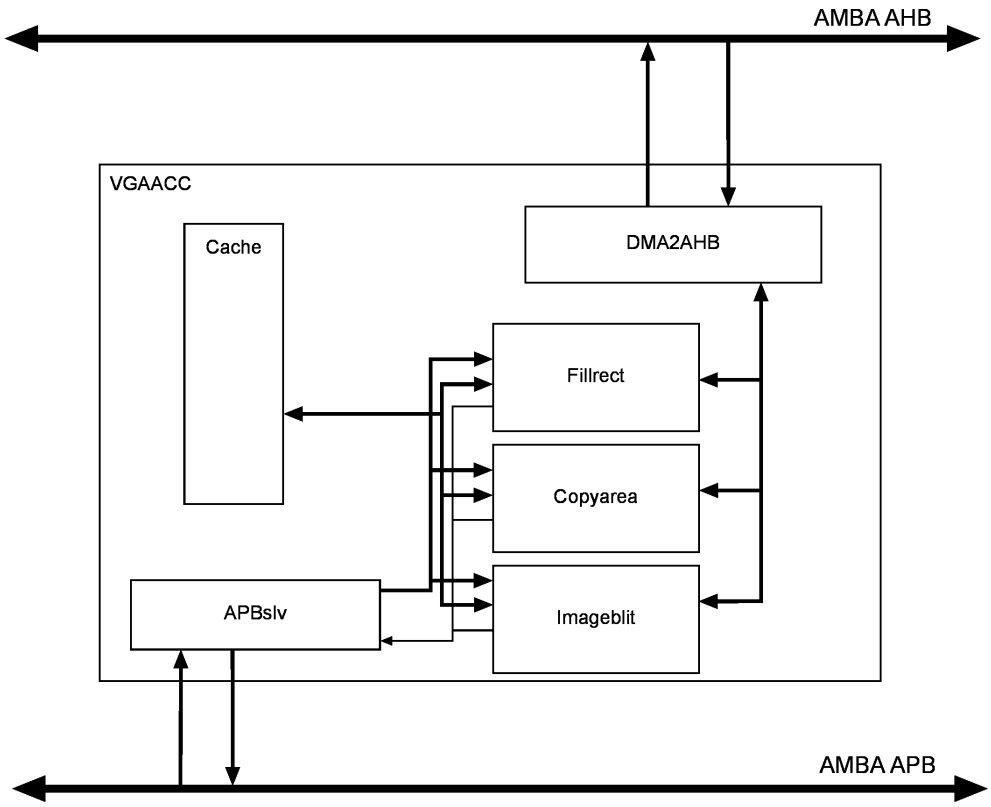


Figure 7: Relations between VHDL entities.

The graphics accelerator IP-core consists of a main block, which is connected to the AHB and APB, and six internal blocks which handle different functions of the cores operations, see Figure 7. All the blocks are described in the following sections.

5.1 OVERVIEW

5.1.1 VGA Accelerator (VGAACC.vhd)

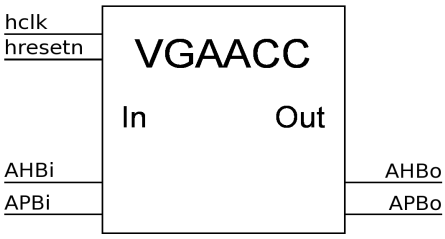


Figure 8: VGAACC core with in and out ports.

The main entity of the accelerator uses generics to set variables and instantiate the separate underlying entities. The internal control signals are routed to the appropriate framebuffer operation module by using the operation selection signals from the APB slave. Since this is the main block, it is the only entity with interfaces to the system. The ports of the block can be found in the illustration in Figure 8.

5.1.2 AMBA AHB Master Interface (DMA2AHB.vhd)

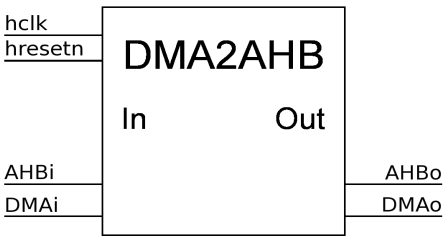


Figure 9: DMA2AHB block with in and out ports.

To access the memory via the AHB an existing IP design from the GRLIB VHDL IP library is used. The DMA2AHB core is a AMBA AHB master interface with DMA input. Though the functionality of the AHB is reduced this interface is well suited to support the requirements of memory access by the graphics accelerator. The ports of the block can be seen in the illustration in Figure 9.

5.1.3 APB slave (APBslv.vhd)

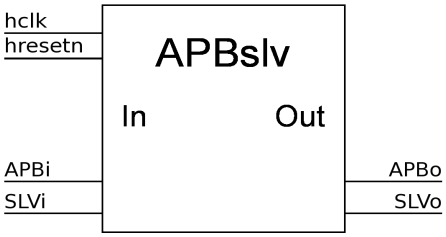


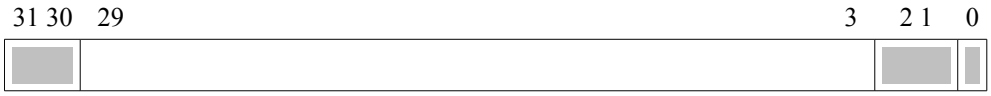
Figure 10: APBslv block with in and out ports.

In Figure 10 an illustration of the APBslv block and its ports can be seen. The operation commands and data from the CPU are sent on the APB. For the graphics accelerator IP-core to be able to access the APB an AMBA APB interface, APBslv, has been constructed. It is a basic interface with seven registers. Six to store the commands and the data required to perform the framebuffer operations and one to present information on the cores current state, see Table 1.

Table 1: Offsets for APBslv registers.

Register number	Offset	Register Information
1	0x00	Command Call Reg(0)
2	0x04	Command Call Reg(1)
3	0x08	Command Call Reg(2)
4	0x0C	Command Call Reg(3)
5	0x10	Command Call Reg(4)
6	0x14	Command Call Reg(5)
7	0x18	Operation information output

When all the necessary data for a specific framebuffer operation has been received a signal to execute the operation is sent to the operation blocks. During the execution of the operation no new command can be initiated by writing to the slaves address space, however all registers can be read. When the core is busy register seven contains information about the ongoing operation. See Figure 11 for details of register seven.



Bit 31:30, Current operation.

Bit 2, Operation running.

Bit 1, Error occurred during execution of operation.

Bit 0, VGAACC block busy.

Figure 11: Contents of APB slaves register 7.

5.1.4 Fill Rectangle (Fillrect.vhd)

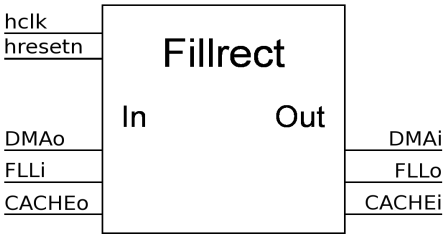


Figure 12: Fillrect block with in and out ports.

The block Fillrect performs the framebuffer operation fill rectangle by writing a pattern to the framebuffer memory using the DMA2AHB interface. The interface ports of the block can be seen in Figure 12. This is done in a state machine with three states as shown in Figure 13. The idle state is used to wait for the execute signal and for row changes.

If the operation includes the ROP XOR, the existing pixel data will be fetched, the raster operation performed and data stored to the local cache in the receive state. In the send state the modified pattern is written to the framebuffer memory. The memory accesses are done in bursts up to 16 words long.

In the case of the ROP COPY, the existing data will be overwritten by the pattern throughout the whole rectangle width in an incremental burst, with the exception of handling leading and trailing bits.

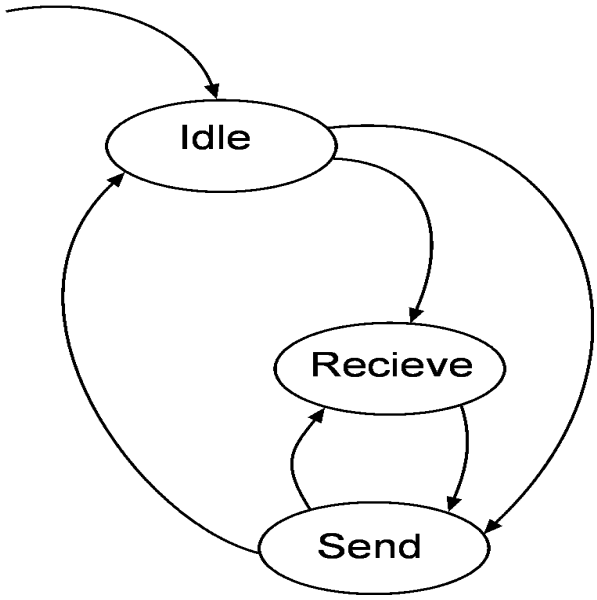


Figure 13: Fill rectangle state machine.

No row unalignment can occur due to the limitation in pixel depth and screen resolution, however leading and trailing bits must be handled in the case of 8 or 16 bits per pixel. This is done in the receive state by applying bitmasks to both the fetched pixel data and to the fill pattern, before merging the data into a single word stored in the local cache. The result is then written to framebuffer memory in the send state.

The memory accesses are done in bursts up to 16 words long. When the last word of the row has been written, the machine returns to idle state. If another row is to be filled, an address calculation is performed and the procedure starts over. This is repeated for each row of the rectangle's height. When the last pixel pattern of the rectangle has been written, a signal indicating that the operation is done is sent to the APBslv block.

5.1.5 Copy Area (Copyarea.vhd)

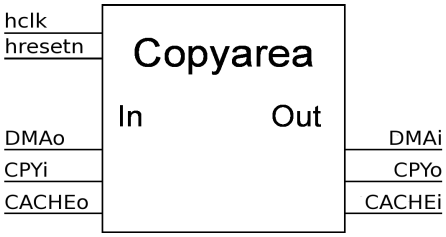


Figure 14: Copy area block with in and out ports.

5.1 OVERVIEW

The block Copyarea which performs the framebuffer operation copy area is similar to the Fillrect block. The difference being that instead of a static pattern to write to the framebuffer memory, the pixel data to write already exists within the framebuffer memory. This resembles the case of the fill rectangle framebuffer operation with the ROP XOR, except the data fetched is from another address than the destination address. The ports of the Copyarea block can be seen in Figure 14.

There are still the leading and trailing bits to consider. They are handled in the same way as in the Fillrect block by applying bitmasks to both existing pixel data, from the destination address, and to the new pixel data from the source address. The data is then merged into a single word stored in the local cache. This is done in the receive state before writing the result to framebuffer memory in the send state.

If the source and destination areas overlap the addresses might have to be reversed so that no source data is overwritten before it is read. This is further complicated by the way the data transmitted in bursts over the bus. The address to the data in memory is incremented throughout the length of the burst operation and the burst cannot be reversed and the address decremented instead. When performing a reverse copy the last word of the first received burst is actually the first word of the area to be copied.

Also, alignment can be an issue when source data and destination data does not start at the same address offset. To handle unalignment, source data has to be realigned before it is stored in the cache and written to the destination address of the framebuffer memory. And if the operation is both reversed and unaligned, the first word of the first received burst has to be realigned with data from the last word of the next received burst before written to the destination address.

The operation is performed in a state machine loop as seen in Figure 15.

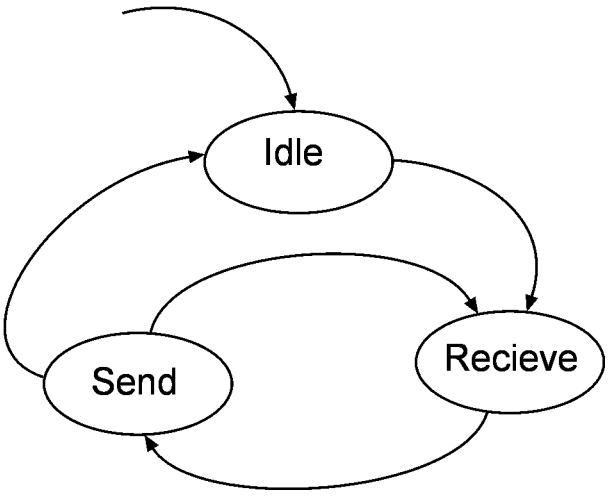


Figure 15: Copy area state machine.

The idle state is used to wait for the execute signal and for row changes. The functionality of the rest of the loop is this.

Pixel data is fetched from the source address, realigned and masked if necessary, and stored in the local cache in the receive state. Then written to the framebuffer memory in the send state. The memory accesses are done in bursts up to 16 words long. When the last word of the row has been written the machine returns to idle state. If there is another row to copy, an address calculation is performed and the copy procedure restarts. This is repeated for all rows of the area's height. When the last pixel has been copied, a signal indicating that the operation is done is sent to the APBslv block.

5.1.6 Image Blit (Imageblit.vhd)

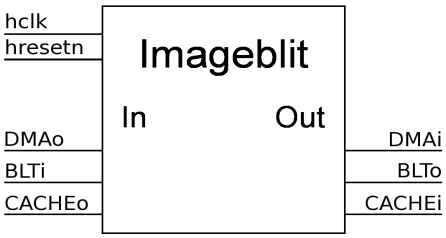


Figure 16: Imageblit block with in and out ports.

Monochrome image blit

The image blit block read source data from system memory and produces pixel data from the source data. The illustration in Figure 16 shows the interface ports of the block. The module can fetch up to 16 32 bit words which is the size of the local cache memory. This means that 16 times 32 pixels can be written before new source data has to be fetched. The choice to fetch 32 bit source words, regardless of the number of bits needed, was made due to efficiency. It takes the same amount of time to fetch a 32 bit word as an 8 bit word. If the start source word is not 32 bit aligned the module can address an 8 bit word inside the 32 bit source word.

When source data is received the module determines if the destination address is unaligned and calculates the first table address if the color depth is 8 or 16 bits per pixel.

If any of the transmissions has an unaligned destination address, this can occur in the beginning or end of the row, they need to be handled. At 16 bit color depth, a single 16 bit word will be written and only one source bit will be consumed. At 8 bit color depth there are three cases. One, two or three 8 bit words has to be written and the same number of source bits are used.

After writing the possible initial unaligned words, the module will set up an incremental burst that will transmit until the end of the row or possible unaligned addresses. Or, until the cache runs empty and new source data

must be fetched. All regular transmissions are 32 bit words which, for example, causes a transfer at 8 bit color depth to write four pixels per clock cycle.

The data is extracted from a source data vector which is continuously updated with new source data from the cache when data has been used and expended. If the color depth is 16 or 8 bits per pixel the module will use two or four bits respectively, of the most significant bits of the source data vector. These are used to address a corresponding mask table which creates the color pattern. At 32 bits per pixel the module will index the source data vector directly since only one bit is used for every 32 bit word to send. If two data bits is used, the source vector will be shifted two steps to the left and two new data bits from the cache will be written on the two least significant bits.

This was necessary due to possible earlier unaligned transfers using up an odd number of source bits and the source data vector which would cause it to be uneven in the end. This way there will always be data available for aligned data writing until the end of the row or the possible trailing bits that will end the row.

The data is extracted from a source data vector which is continuously updated with new source data from the cache when data has been used and expended. For example if two data bits is used the source vector will be shifted two steps to left and two new data bits will be written on the two least significant bits. This way there will always be data available for aligned data writing until the end of row or the eventual unaligned addresses that will end the row.

If the color depth is 16 or 8 bits per pixel the module will use two or four bits respectively, of the most significant bits of the source data vector. Else, if the pixel depth is 32 bit, only the most significant bit in the source data vector will be used to index the source data.

When the last pixel in a row is written, the module will calculate the address to the next row and the address to next source data. New source data is fetched and the module will start writing the new row. This will continue until the module reaches the last pixel of the last row. When the last pixel has been written, a signal indicating that the operation is done is sent to the APBslv block.

This is a brief description of the state machine, illustrated in Figure 17, more information can be found in Chapter 5.2.6. The first state entered is the RX state, when all source data has been received a jump to Setup_TX will take place. There it will be determined if the first address is aligned or not. If the first address is unaligned next state is Unaligned_TX, or TX if the first address is aligned.

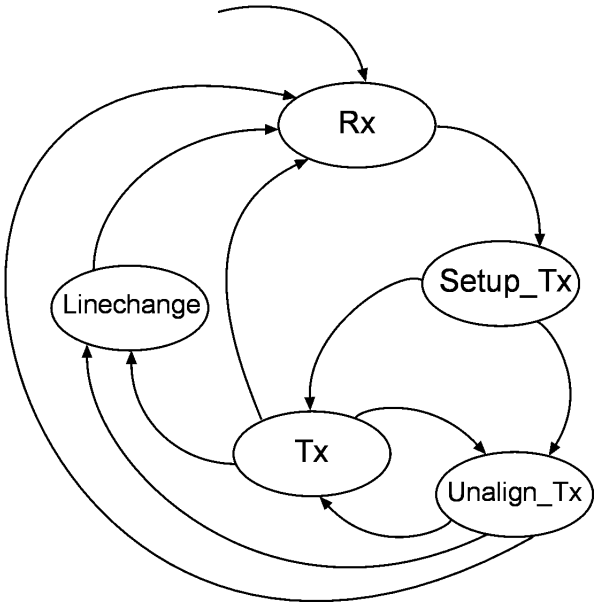


Figure 17: Image blit state machine.

From Unaligned_TX a jump to TX is made if the next address is aligned. Otherwise the process will remain in Unaligned_TX if the address is unaligned, jump to Linechange if the row is completed or to RX if the cache runs out of data. When the TX state is entered it will write until it runs out of source data or the row is completed, with the exception for trailing bits. From here the next state can be Linechange if the row is completed, Unaligned_TX if the last address is unaligned or, if the cache runs out of data, RX. When the Linechange state is done next state is RX if there are more rows to write, otherwise the task is completed and all variables resets.

Color image blit

The algorithm of color image blit is quite similar to slow monochrome image blit with the difference that color image blit fetches its pixel data from a pseudo palette. It is a structure that consists of 16 32 bit words that contains color data. This might not seem to be much data but the software is continuously updating the palette. Thus it is very hard to use for hardware acceleration since the hardware must know when the software updates the palette and must wait for the update to complete. In order to see if there are any good solutions to this problem other hardware drivers were examined. However, since other hardware accelerators do not accelerate color image blit the choice to exclude the operation was made.

5.1.7 SyncRAM Cache

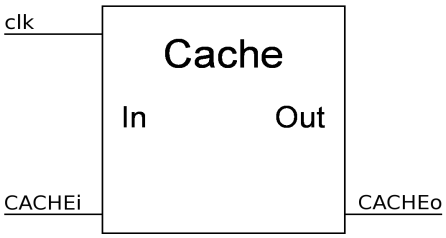


Figure 18: Cache block with in and out ports.

A cache memory has been placed locally to make bursting data to and from the memory possible. It is a synchronous single port RAM and all of the framebuffer operation blocks use the same cache with 16 32 bit words. The cache is clocked by an inverted AMBA system clock. The illustration in Figure 18 shows the cache block and its ports. More details on the cache interface can be found in Table 2.

Table 2: Signal descriptions of Cache interface.

Signal name	**Field**	**Type**	**Function**	**Active**
clk	N/A	Input	Clock	-
CACHEo	DATA[31:0]	Input	Data	-
CACHEi	Addr[0 to 15] en DATA[31:0]	Output	Address [Integer] Write enable Data	- High -

5.1.8 VGA Accelerator Package(VGAACC_pkg.vhd)

The package contains the accelerator specific record declarations, used for interfacing the internal blocks, and declarations of functions used in the design.

5.2 Details

This section describes the design with more details. The reader should get a good understanding of the function and structure of the source code and be able to read and follow the code with little or no problem.

5.2.1 VGA Accelerator (VGAACC.vhd)

This is the main entity, it contains all the designs components which makes the interface to the surrounding system as small and neat as possible. The function of the main block is to instantiate the sub-components and handle internal signal routing. This is necessary to make sure that the shared resources: cache, APB slave and DMA access, is at disposal to the active framebuffer operation block, and to avoid multiple drivers to shared signals.

The framebuffer operations requires different amounts of data to perform their tasks. The blocks therefore each have their own input records which are routed from the output of the APBslv block. Also, by using the operation selection signal in the first register of the slave output, the input to the shared resources are connected to the output of the currently active framebuffer operation block.

In Table 3 the interface signals to the accelerator are described. As seen, the only contact with the surrounding system are through AMBA interface signals.

Table 3: Signal descriptions of VGAACC interface.

Signal name	Field	Type	Function	Active
hclk	N/A	Input	Clock	-
hresetn	N/A	Input	Reset	Low
APBi	*	Input	APB slave input signals	-
APBo	*	Output	APB slave output signals	-
AHBi	*	Input	AHB master input signals	-
AHBo	*	Output	AHB master output signals	-

** see GRLIB IP Library User's Manual [3]*

5.2.2 AMBA AHB Master Interface (DMA2AHB.vhd)

The AMBA AHB interface has, through this block, been reduced in function to support only what is required for DMA access. Although not included in the GRLIB IP Core User's Manual, the block is a part of the core library and can be found in the AMBA sub-directory of the GRLIB folder. For more information the reader is referred to the VHDL code.

In Table 4 the interface signals for the DMA2AHB block are described. The AMBA AHB interface is directly connected to the AHB through the VGAACC block. The DMA interface is routed to the active framebuffer operation block through the main VGAACC block.

Table 4: Signal descriptions of DMA2AHB interface.

Signal name	Field	Type	Function	Active
hclk	N/A	Input	Clock	-
hresetn	N/A	Input	Reset	Low
DMAi	Reset Address[31:0] Data[31:0] Request Burst Beat[1:0] Size[1:0] Store Lock	Input	Reset Address Data Access requested Burst requested Incrementing beat Size Data write requested Locked transfer	Low - - High High - - High High
DMAo	Grant OKAY Ready Retry Fault Data[31:0]	Output	Access accepted Write access ready Read data ready Retry Error occurred Data	High High High High High -
AHBi	*	Input	AHB master input signals	-
AHBo	*	Output	AHB master output signals	-

** see GRLIB IP Library User's Manual [3]*

5.2.3 APB Slave (APBslv.vhd)

The accelerator is called by writing the necessary data to the APB slave's address space on the APB. The calls must be written to the address space in incrementing offset order starting with the offset 0x00, followed by 0x04 and 0x08 etc. The slave uses the first word to determine the desired framebuffer operation and how many data words that are needed for that operation.

All incoming data from the APB is stored in a local register. When the required number of data words for the desired operation has been received, the APB slave sends an execute signal and forwards the data through the VGAACC block.

In Table 5 the interface signals for the APBslv block are described. The AMBA APB interface is directly connected to the APB through the VGAACC block. The slaves internal interface of the core is routed to the active framebuffer operation block through the main VGAACC block.

Table 5: Signal descriptions of APBslv interface.

Signal name	Field	Type	Function	Active
hclk	N/A	Input	Clock	-
hresetn	N/A	Input	Reset	Low
SLVi	done opInfo[1:0]	Input	Operation complete Operation information	High -
SLVo	execute reg[0:5] [31:0]	Output	Execute operation Data registers	High -
APBi	*	Input	APB slave input signals	-
APBo	*	Output	APB slave output signals	-

** see GRLIB IP Library User's Manual [3]*

5.2.4 Fill Rectangle (Fillrect.vhd)

This section will describe the Fillrect block with more detail. The flowcharts can be a visual aid when reading the code and the tables describing the interfaces and signals should help to interpret the connections between the blocks. The reader can also find the required data to perform the fill rectangle operation in the details of the operation call record.

Flowcharts of Fillrect

The fill rectangle operation is performed in three states, as mentioned earlier. The full algorithm is here described with more detail through four flowcharts, one for each state and one describing the combinatorial process of the Fillrect architecture.

In Figure 19 the flowchart of the combinatorial process is depicted. By using the flowcharts as a reference and looking at the source code the function of the algorithm should become clear.

5.2 DETAILS

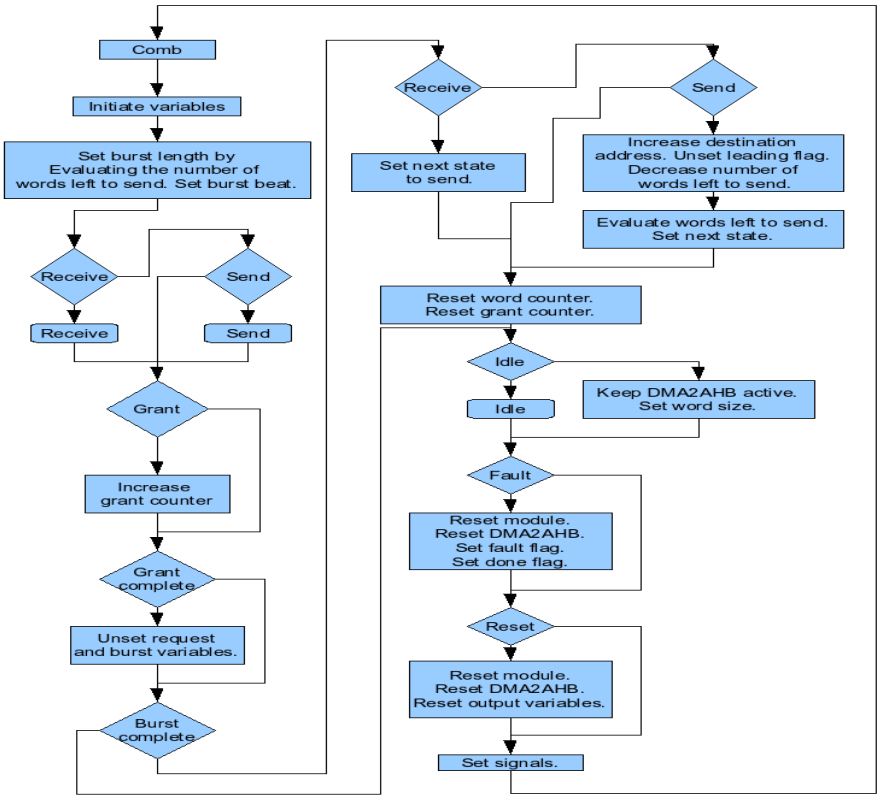


Figure 19: Flowchart of combinatorial process of Fillrect.

The process starts by setting variables and interpreting incoming data from the APB slave. Then the burst length and burst beat [1] is set by evaluating the number of words left to write. If the current state is either receive or send the respective state is entered and processed before returning to the combinatorial flowchart.

During the burst counters keep track of how many granted bus accesses acquired and how many words received or sent. While continuing through the flowchart, the bus request is withdrawn if all the grants needed for the burst are acquired. After that, if the burst has been completed the state is changed by evaluating the current state and the number of words left to complete the row. The counters are reset and new addresses are calculated if needed.

In the idle state each row is initiated and the operation always starts and ends there, this is described in Figure 20. During receive or send phases the DMA2AHB is kept active by variables set if the current state is not the idle state.

At the end of the process DMA errors and system reset is handled before signals are set by the variables used in the process.

To go into more details the flowcharts of the different states of the block are depicted separately. Figure 20 shows the idle state of the Fillrect block.

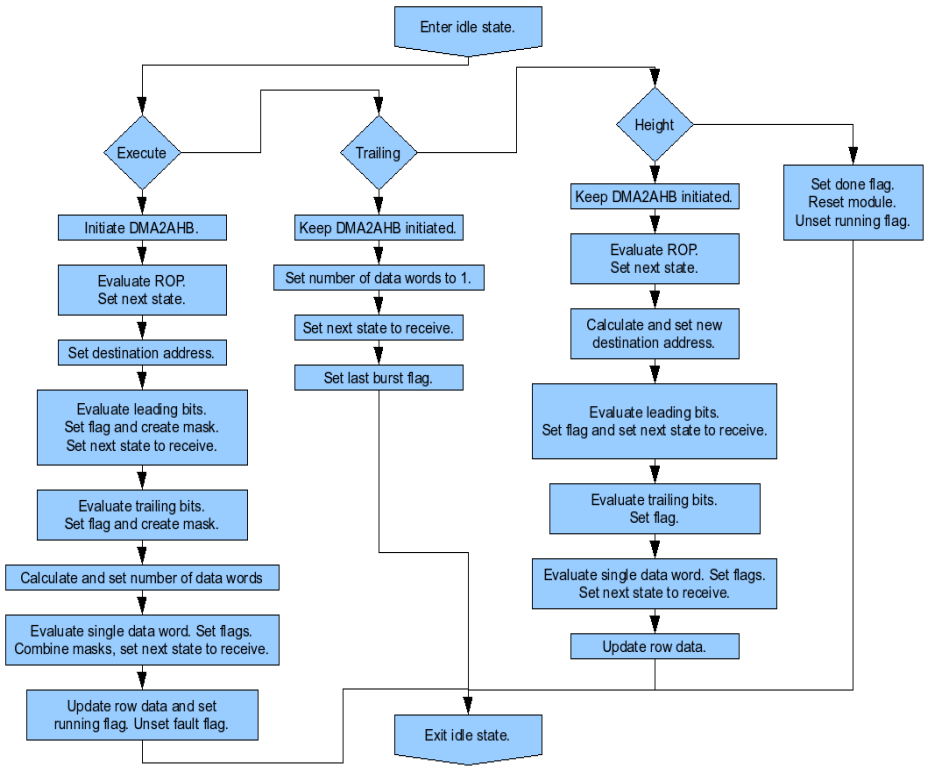


Figure 20: Flowchart of Idle state of Fillrect.

The idle state is the beginning and end of each row of the rectangle, as well as of the whole operation. As seen in the flowchart the state is divided into three possible paths and one default setting. When waiting on an operation call the module is a reset state, but when the execute signal is sent the module is initiated by the left path of the flowchart and the first row of the rectangle is filled.

Trailing bits, if any, are handled separately by an additional, single word, send-receive cycle at the end of each row. This is the middle path of the flowchart.

The path to the right initiates each subsequent row of the rectangle, if there are more than one.

In Figure 21 the receive state is illustrated.

5.2 DETAILS

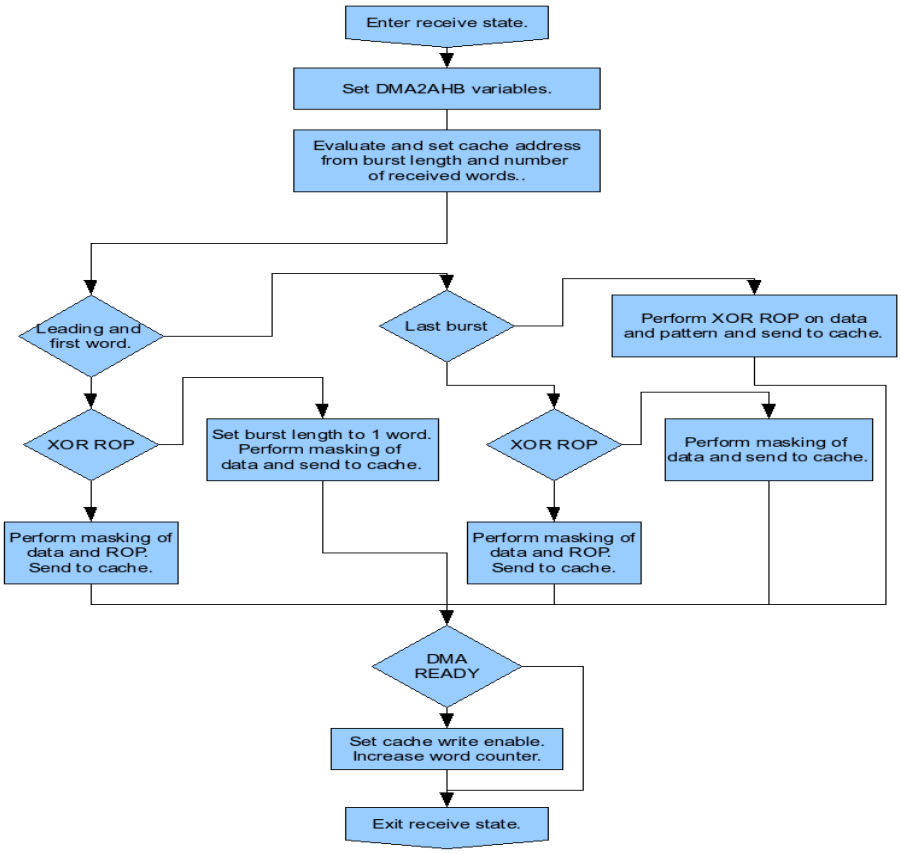


Figure 21: Flowchart of Receive state of Fillrect.

The receive state of Fillrect is mainly used with the ROP XOR, however it is also used to handle leading and trailing bits as seen in Figure 21. When handling leading bits for ROP COPY the burst length has to be adjusted to one word. After the data has been sent to the cache the word counter is increased and the flow of the process exits the receive state.

Finally the flowchart of the send state is depicted in Figure 22.

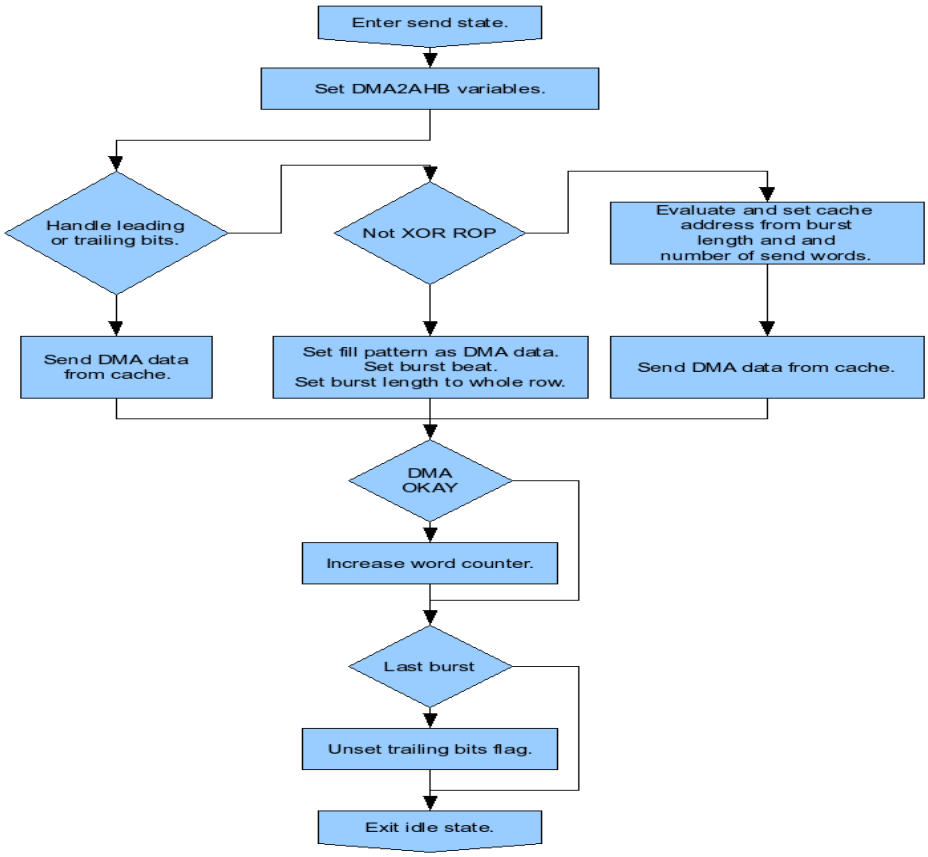


Figure 22: Flowchart of Send state of Fillrect.

The flow of the send state shown in Figure 22 depends on which raster operation to perform and if there are any leading- or trailing bits, the difference being which data to send and how long the burst should be.

In the case of ROP XOR, or in the case of leading or trailing bits, the data sent if fetched from the cache. However if the operation is the ROP COPY the data sent is simply the pattern included in the operation call and the whole row can be written in a single incremental burst, as opposed to bursts up to 16 words long which are limited by the size of the cache.

The number of words sent are counted and before the process exits the send state the flag for trailing bits are unset.

Signals and Interfaces of Fillrect

In Table 6 the interface signals for the Fillrect block are described. They connect the block to the APB slave, cache and DMA access.

5.2 DETAILS

Table 6: Signal descriptions of Fillrect interface.

Signal name	**Field**	**Type**	**Function**	**Active**
hclk	N/A	Input	Clock	-
hresetn	N/A	Input	Reset	Low
FLLi	execute reg[0:3][31:0]	Input	Execute operation Data registers	High -
FLLo	done opInfo[1:0]	Output	Operation complete Operation information	High -
DMAi	Reset Address[31:0] Data[31:0] Request Burst Beat[1:0] Size[1:0] Store Lock	Input	Reset Address Data Access requested Burst requested Incrementing beat Size Data write requested Locked transfer	Low - - High High - - High High
DMAo	Grant OKAY Ready Retry Fault Data[31:0]	Output	Access accepted Write access ready Read data ready Retry Error occurred Data	High High High High High -
CACHEo	DATA[31:0]	Input	Data from cache	-
CACHEi	Addr[0 to 15] en DATA[31:0]	Output	Address [Integer] Write enable Data to cache	- High -

The command call interface from the APB slave is described in detail in Figure 23. This is the information needed from the Linux driver to perform the fill rectangle operation.

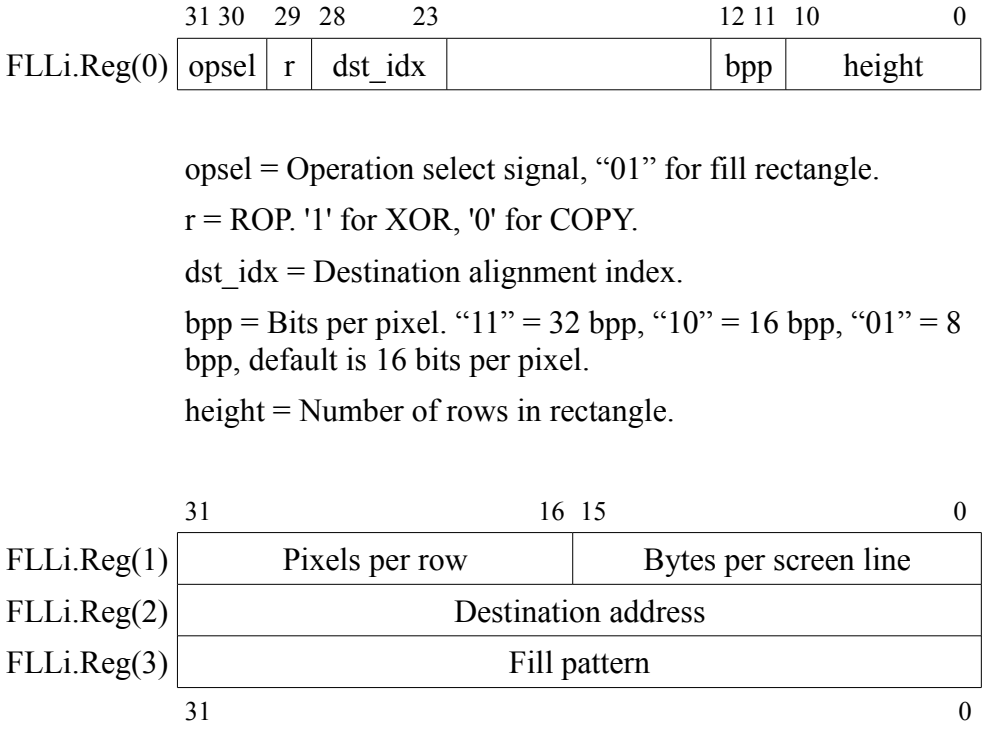


Figure 23: Details of Fillrect command call record.

5.2.5 Copy Area (Copyarea.vhd)

This section will describe the Copyarea block with more detail. The flowcharts can be a visual aid when reading the code and the tables describing the interfaces and signals should help to interpret the connections between the blocks. The reader can also find the required data to perform the copy area operation in the details of the operation call record.

Flowcharts of Copyarea

The copy area operation is performed in three states, as mentioned earlier. The full algorithm is here described with more detail through five flowcharts describing each state and one depicting flow of the combinatorial process of the Copyarea architecture.

In Figure 24 the flowchart of the combinatorial process is depicted. By using the flowcharts as a reference and looking at the source code the function of the algorithm should become clear.

5.2 DETAILS

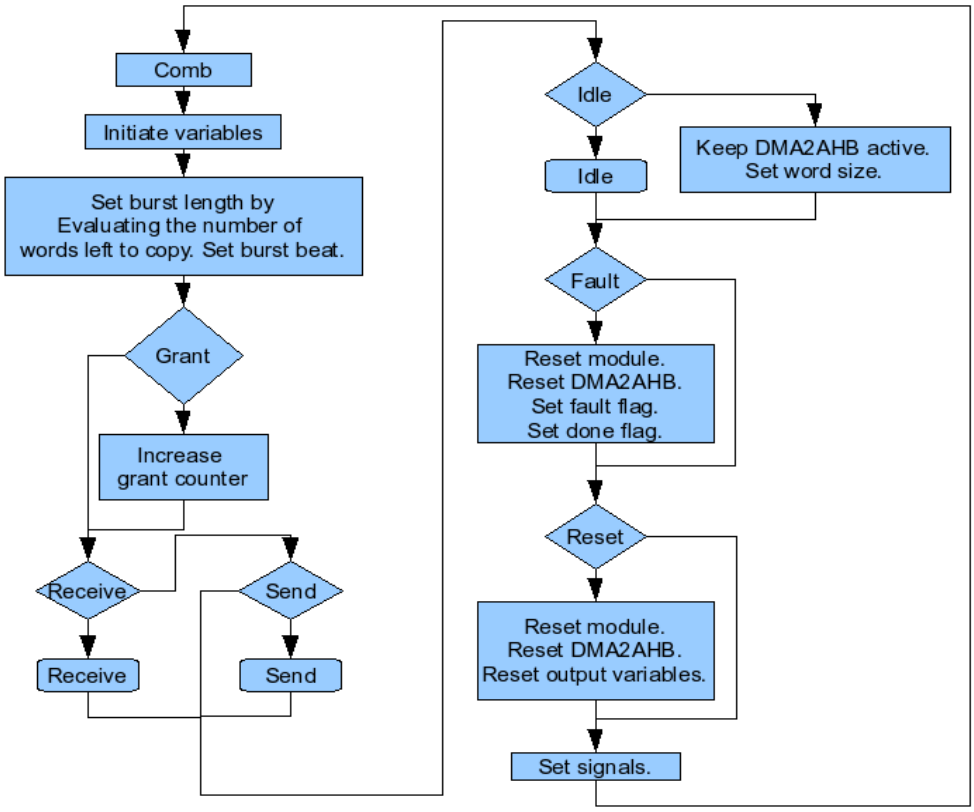


Figure 24: Flowchart of combinatorial process of Copyarea.

The process starts by setting variables and interpreting incoming data from the APB slave. Then the burst length and burst beat [1] is set by evaluating the number of words left to write. If bus access is granted a counter is increased to keep track of the number of acquired accesses. If the current state is either receive or send the respective state is entered and processed before returning to the combinatorial flowchart. The states are described later.

In the idle state each row is initiated and the operation always starts and ends there, this is described below. During receive or send phases the DMA2AHB is kept active by variables set if the current state is not the idle state.

At the end of the process DMA errors and system reset is handled before signals are set by the variables used in the process.

To go into more details the flowcharts of the different states of the block are depicted separately. Two states, idle and receive, are divided further into two flowcharts each. Figure 25 and Figure 26 shows the idle state of the Copyarea block.

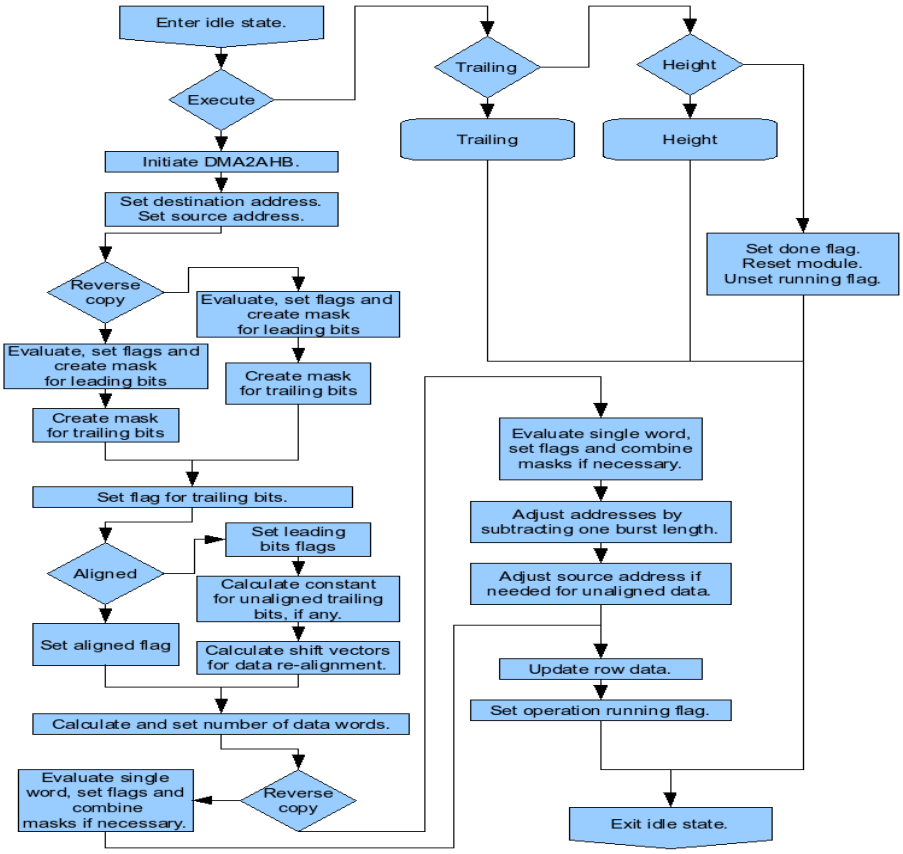


Figure 25: Flowchart of Idle state of Copyarea, 1 of 2.

The idle state is the beginning and end of each row of the area, as well as of the whole operation. As seen in the flowchart the state is divided into three possible paths and one default setting. When waiting on an operation call the module is in a reset state, but when the execute signal is sent the module is initiated by the left path of the flowchart and the first row of the area is copied.

The initiation of the module is different depending on whether or not the area has to be reversed copied and if the data is aligned.

Trailing bits, if any, are handled separately by an additional, single word, send-receive cycle at the end of each row. This initiation path is described in Figure 26.

The path to the right initiates each subsequent row of the area, if there are more than one row. This is also described in Figure 26.

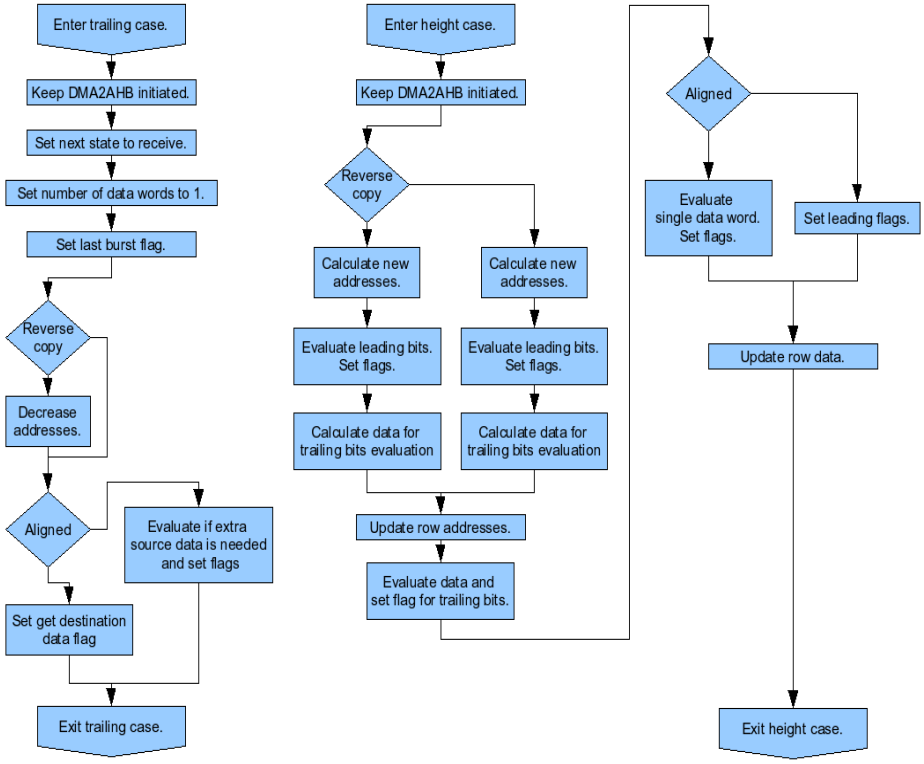


Figure 26: Flowchart of Idle state of Copyarea, 2 of 2.

To the left in Figure 26 the path handling trailing bits in the idle state of the Copyarea block is illustrated. A receive-send cycle of one data word is initiated. This setup is different depending on whether or not the operation is reverse copy and if the data is aligned.

To the right in Figure 26 the path handling the height of the area is shown. This also depends on whether or not the operation is reverse copy and if the data is aligned.

In Figure 27 and Figure 28 the receive state is illustrated.

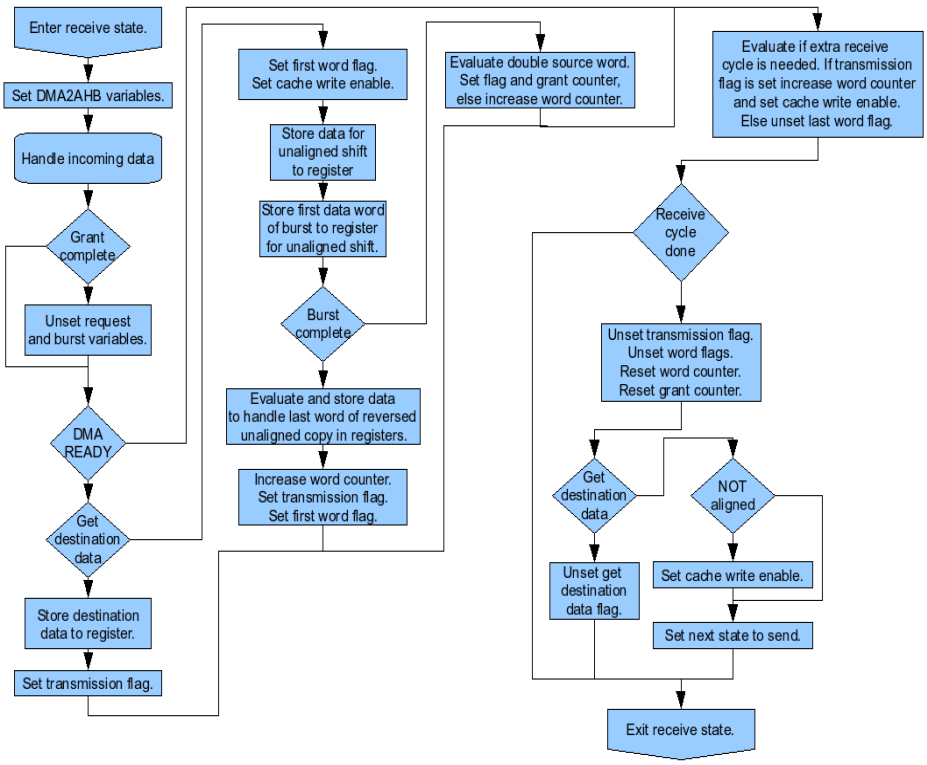


Figure 27: Flowchart of Receive state of Copyarea, 1 of 2.

The receive state of Copyarea is complex due to the handling of unaligned data. Therefore the handling of incoming data is described in a separate flowchart, Figure 28. The variables for DMA access is set in the beginning of the state and then the handling of incoming data is performed. If the number of bus accesses acquired matches the burst length the bus request variable is unset.

If data from the DMA2AHB block is valid the receive state restarted if the data fetched was destination data. Otherwise some data handling, necessary for unaligned data, is done. If the burst is complete, flags are set and some more data handling is done. If not the burst length might be extended to get extra source data or the word counter is increased.

If the receive cycle is finished counters and flags are reset and the next state is set.

5.2 DETAILS

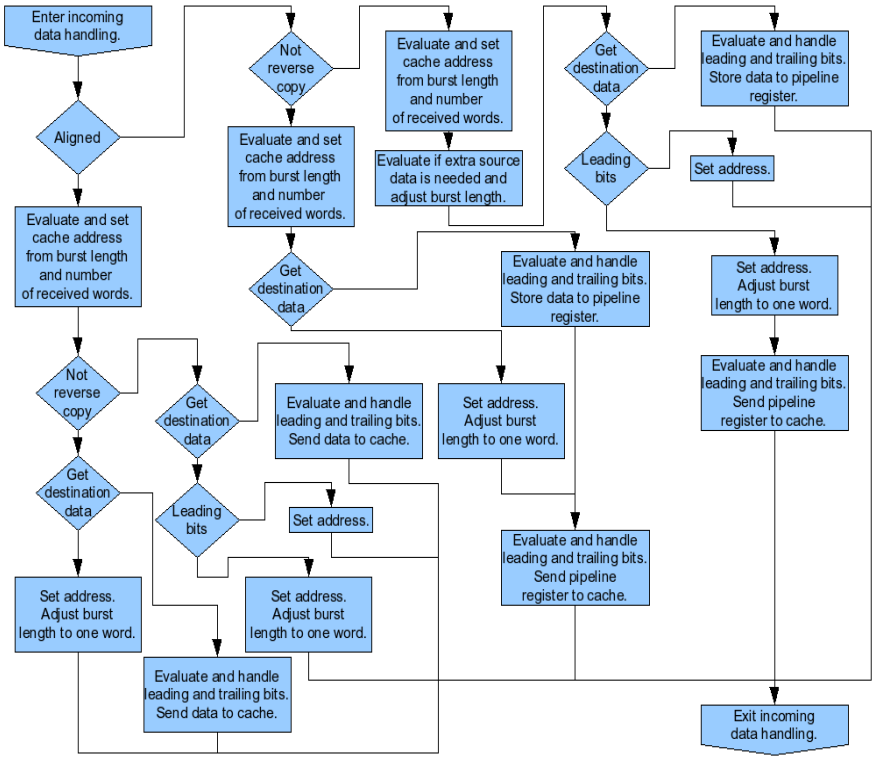


Figure 28: Flowchart of Receive state of Copyarea, 2 of 2.

To understand the handling of the incoming data the flowchart in Figure 28 should be of help. The incoming data is handled differently depending on several variables.

- Destination data is received.
- There are leading- or trailing bits.
- Copy operation is aligned.
- Copy operation is unaligned
- Copy operation is reversed and aligned.
- Copy operation is reversed and unaligned.

To avoid long data paths due to shifting and merging of unaligned data, a pipeline register has been introduced. This splits the modification of the data into two cycles, which reduces the data path, and delays the store to cache by one cycle. Also there are special cases to handle, for example when leading or trailing bits overlap two source words. Commentary in the code should also help the reader to a better understanding of what is done to the data.

Finally the flowchart of the send state is depicted in Figure 29

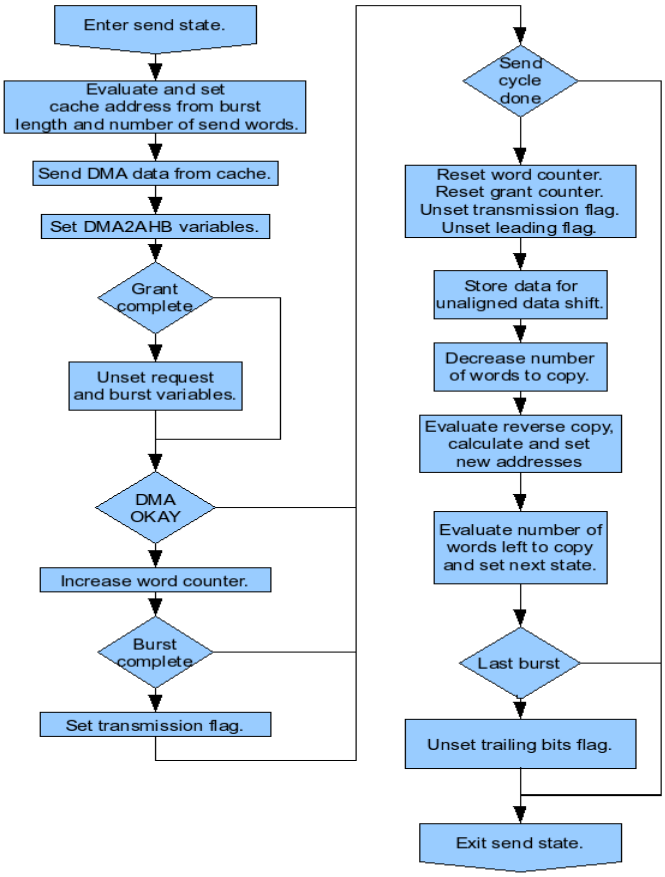


Figure 29: Flowchart of Send state of Copyarea.

The send state is very straight forward. Data is sent from the cache to the framebuffer memory. When the bus accesses needed for the burst is acquired the bus request and burst variables are unset. For each word sent the word counter is increased and when the last word is sent a flag for transmission done is set.

When the send cycle is done the counters and flags are reset, the number of words left to copy decreased and new addresses calculated. If the row is completed the state is set to idle, otherwise to receive.

Signals and Interfaces of Copyarea

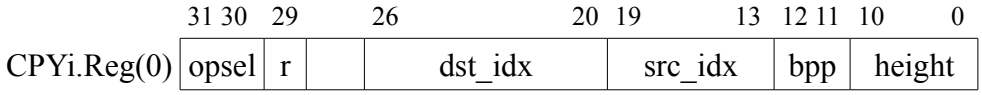
In Table 7 the interface signals for the Copyarea block are described. They connect the block to the APB slave, cache and DMA access.

5.2 DETAILS

Table 7: Signal descriptions of Copyarea interface.

Signal name	Field	Type	Function	Active
hclk	N/A	Input	Clock	-
hresetn	N/A	Input	Reset	Low
CPYi	execute reg[0:3][31:0]	Input	Execute operation Data registers	High -
CPYo	done opInfo[1:0]	Output	Operation complete Operation information	High -
DMAi	Reset Address[31 :0] Data[31:0] Request Burst Beat[1:0] Size[1:0] Store Lock	Input	Reset Address Data Access requested Burst requested Incrementing beat Size Data write requested Locked transfer	Low - - High High - - High High
DMAo	Grant OKAY Ready Retry Fault Data[31:0]	Output	Access accepted Write access ready Read data ready Retry Error occurred Data	High High High High High -
CACHEo	DATA[31:0]	Input	Data from cache	-
CACHEi	Addr[0 to 15] en DATA[31:0]	Output	Address [Integer] Write enable Data to cache	- High -

The command call interface from the APB slave is described in detail in Figure 30. This is the information needed from the Linux driver to perform the copy area operation.



opsel = Operation select signal, “10” for copy area.

r = Reverse copy, active high.

dst_idx = Destination alignment index.

src_idx = Source alignment index.

bpp = Bits per pixel. “11” = 32 bpp, “10” = 16 bpp, “01” = 8 bpp, default is 16 bits per pixel.

height = Number of rows in rectangle.

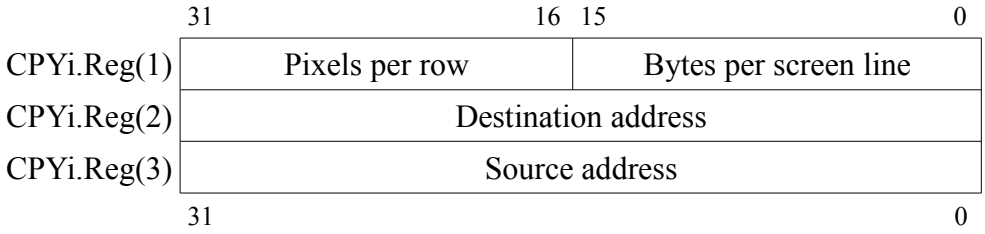


Figure 30: Details of Copyarea command call record.

5.2.6 Image Blit (Imageblit.vhd)

This section will describe the Imageblit block with more detail. The flowcharts can be a visual aid when reading the code and the tables describing the interfaces and signals should help to interpret the connections between the blocks. The reader can also find the required data to perform the image blit operation in the details of the operation call record.

As mentioned earlier, only the monochrome blit operation has been implemented in the core.

Flowcharts of Imageblit

The image blit module consists of five states RX, TX, Linechange, Setup_TX and Unaligned_TX. The algorithm of the module is described in flow charts, one for every state and one for the combinatorial process. In order to keep the flow charts readable they describe a simplified model of the actual states. Describing texts will give more details in connection to the flow charts.

5.2 DETAILS

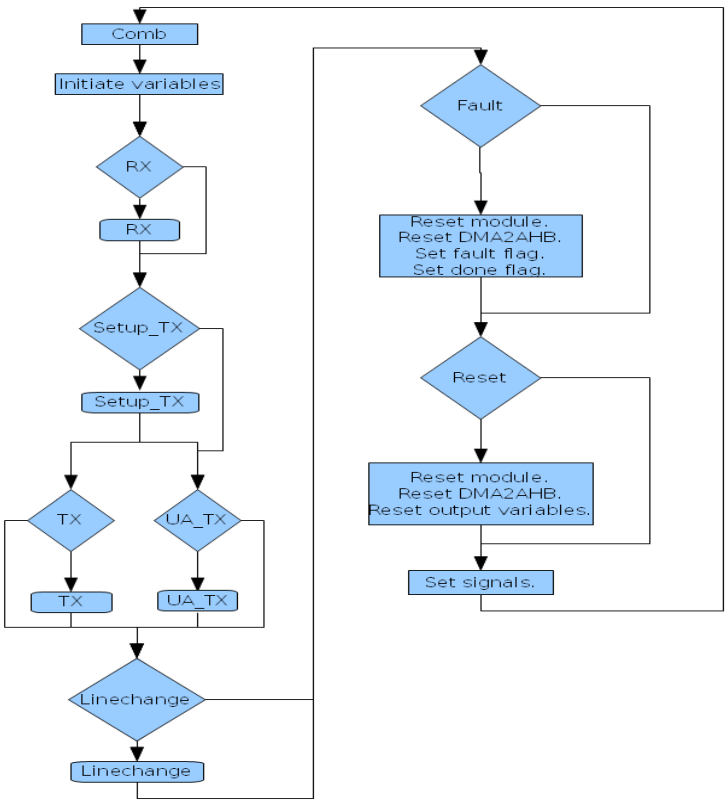


Figure 31: Flowchart of combinatorial process of Imageblit.

Figure 31 shows the main flow in the image blit module. The process starts with initiating variables and parse the data received from the APB slave. The first state entered is the receive state RX in which all of the source data is fetched. Then Setup_TX is entered to determine if the first pixel to write has an unaligned address. Next state is one of the two transmit states, TX or Unaligned_TX. The TX state handle all 32 bit aligned transfers and Unaligned_TX handles unaligned addresses and trailing bits. The process ends with error handling, system reset and setting the signals by the using process variables.

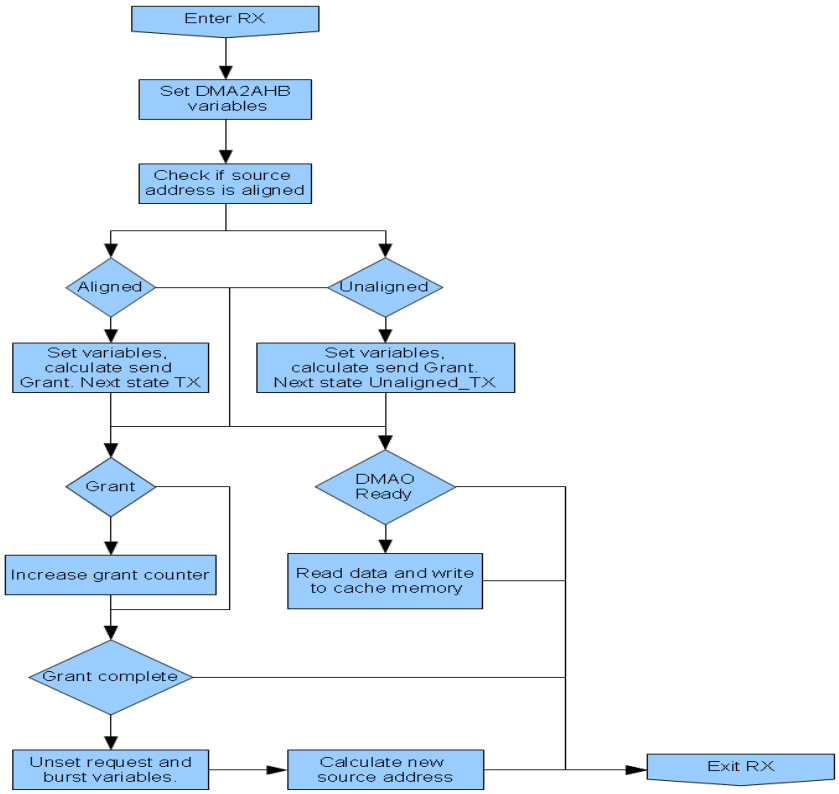


Figure 32: Flowchart of RX state of Imageblit.

In the RX state, shown in Figure 32, the purpose is to fetch source data. This is done as follows: a request is sent to the AHB and then wait for the Grant signal. When Grant is set, the number of grants are counted until it matches the proposed quantity of 32 bit words to fetched. Then the request signal is unset. When the DMAO Ready signal is set, the source data is available to be written in the cache memory. The Grant counter is used to control the number of data words to receive. Before leaving the state a new source address is calculated and the next state is Setup_TX.

A new source data fetch is done for every new row. Since the maximal amount of data is limited by the local cache size, a maximum of 16 32 bit words can be fetched in one burst. If that is not enough data for the whole row of the image it is possible to fetch new data words until the row is finished. The module can address 8 bit words in the 32 bit source word if the source data is unaligned.

5.2 DETAILS

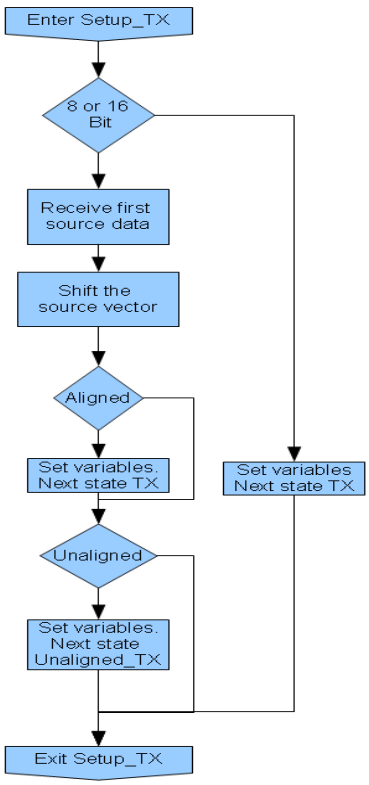


Figure 33: Flowchart of Setup_TX state of Imageblit.

Setup_TX state, depicted in Figure 33, determine if the destination address is unaligned and calculates the first table addresses if 8 or 16 bit color depth. If the destination address is unaligned next state will be unaligned_TX and if the address is 32 bit aligned next state is TX.

The TX state sets a request on the AHB and wait for the Grant signal. When Grant is set, the number of grants are counted until it matches the proposed quantity of 32 bit words to transmitted. Then the request signal is unset. When the DMAO Okay signal is set the bus is available for transmitting data. Next state can be Unaligned_TX or RX or Linechange. The flow of the TX state is illustrated in Figure 34.

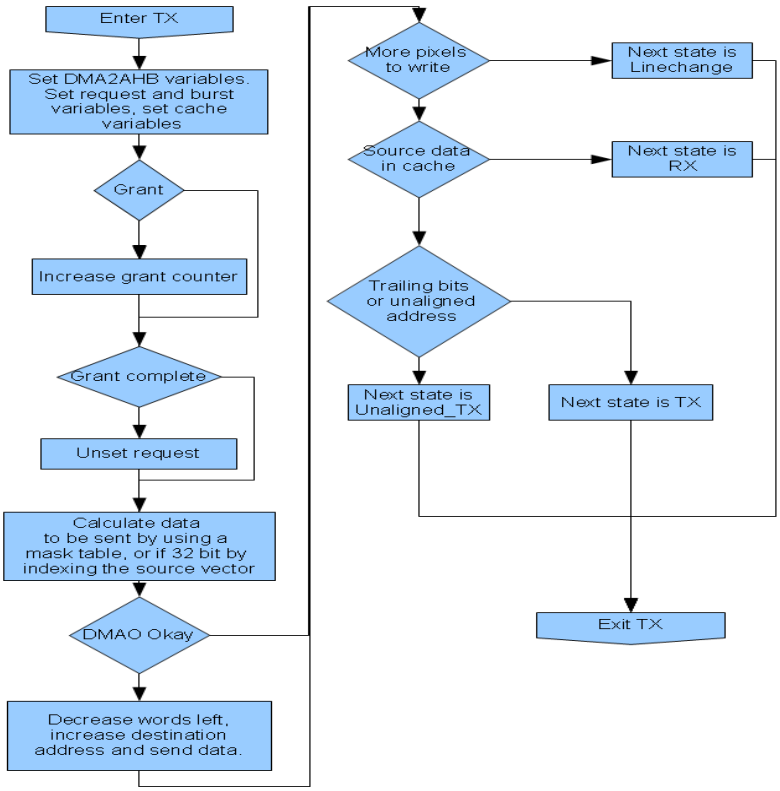


Figure 34: Flowchart of TX state of Imageblit.

The TX state sets up an incremental burst that will transmit until the end of the row to the last trailing bits, if there are any, or if cache runs empty and new data must be fetched. In TX state all transmissions are 32 bit words which means a transfer at 8 bit color depth writes 4 pixels per clock cycle.

The data is extracted from a source data vector which is continuously updated with new source data from the cache when data has been used and expended. If the color depth is 16 or 8 bits per pixel the module will use two or four bits respectively, of the most significant bits of the source data vector. These are used to address a corresponding mask table which creates the color pattern. At 32 bits per pixel the module will index the source data vector directly since only one bit is used for every 32 bit word to send. If two data bits is used, the source vector will be shifted two steps to the left and two new data bits from the cache will be written on the two least significant bits.

This was necessary due to possible earlier unaligned transfers using up an odd number of source bits and the source data vector which would cause it to be uneven in the end. This way there will always be data available for aligned data writing until the end of the row or the possible trailing bits that will end the row.

5.2 DETAILS

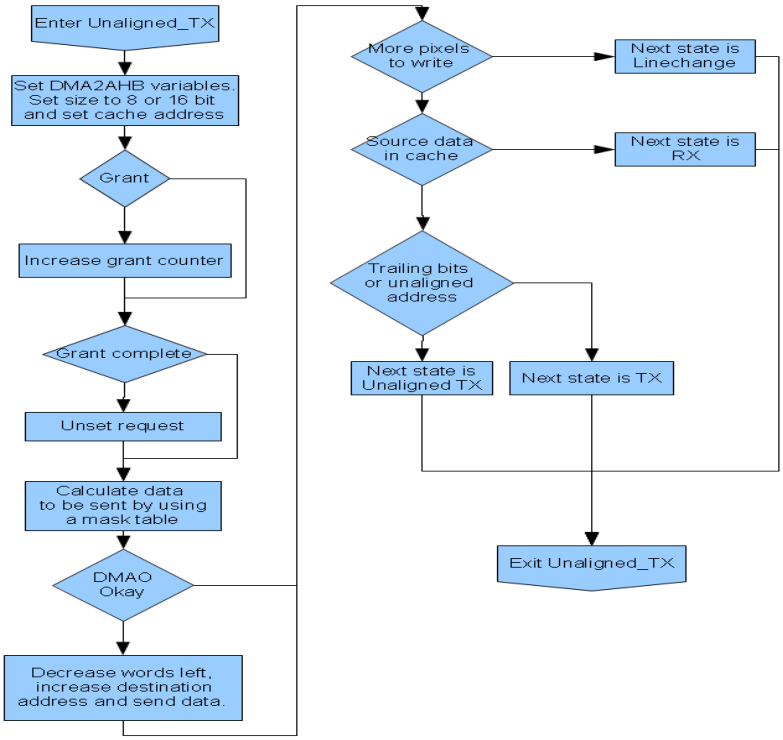


Figure 35: Flowchart of Unaligned_TX state of Imageblit.

The flow through the Unaligned_TX state, shown in Figure 35, is quite similar to the TX with the main difference that the transmission size is set to 8 bit or 16 bit words and single burst is used. This state is used if any of the transmissions has an unaligned destination address or if there are any trailing bits. If the state is needed and the color depth is 16 bits per pixel there will be a single 16 bit word written. If the color depth is 8 bits per pixel there are three cases. One, two or three 8 bit words will be written depending on the destination offset or how many trailing bits there are. The pixel data is calculated by tables, the same way as in TX. Next state can be TX, Linechange or RX.

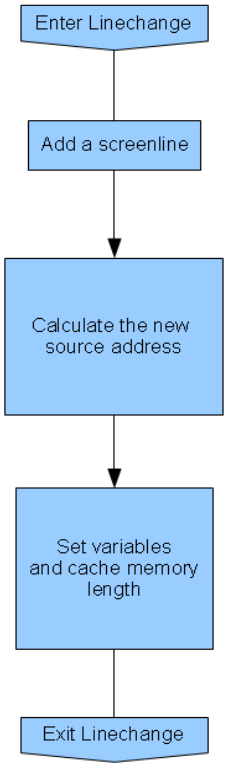


Figure 36: Flowchart of Linechange of Imageblit.

The Linechange state is straightforward and the flow trough the state, depicted in Figure 36, is simple. It will add one screen line to the destination address, calculate the new source address and set the length of the cache memory. The next state is RX if there are more rows in of image to blit. If the image is finished the module will reset and wait for instructions for the next image.

Signals and Interfaces of Imageblit

In Table 8 the interface signals for the Imageblit block are described. They connect the block to the APB slave, cache and DMA access.

5.2 DETAILS

Table 8: Signal descriptions of Imageblit interface.

Signal name	Field	Type	Function	Active
hclk	N/A	Input	Clock	-
hresetn	N/A	Input	Reset	Low
BLTi	execute reg[0:5][31:0]	Input	Execute operation Data registers	High -
BLTo	done opInfo[1:0]	Output	Operation complete Operation information	High -
DMAi	Reset Address[31:0] Data[31:0] Request Burst Beat[1:0] Size[1:0] Store Lock	Input	Reset Address Data Access requested Burst requested Incrementing beat Size Data write requested Locked transfer	Low - - High High - - High High
DMAo	Grant OKAY Ready Retry Fault Data[31:0]	Output	Access accepted Write access ready Read data ready Retry Error occurred Data	High High High High High -
CACHEo	DATA[31:0]	Input	Data from cache	-
CACHEi	Addr[0 to 15] en DATA[31:0]	Output	Address [Integer] Write enable Data to cache	- High -

The command call interface from the APB slave is described in detail in Figure 37. This is the information needed from the Linux driver to perform the image blit operation.

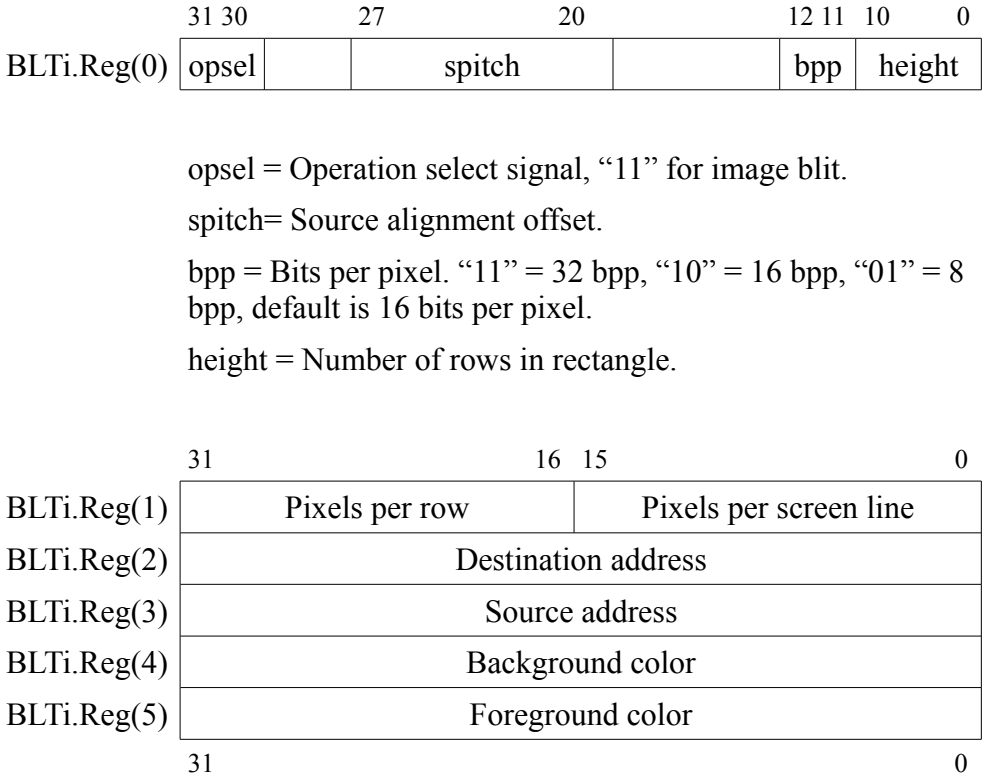


Figure 37: Details of Imageblit command call record.

5.3 Software Driver (grvga.c)

To make Linux utilize the hardware calls instead of the generic software algorithms the existing VGA Controller driver was modified. The added functions are described in this section.

5.3.1 grvgaacc_probe

This function probes the system and look for the accelerators IP core. The cores APB slave registers are then mapped to allocated i/o-memory which makes it accessible from the kernel.

5.3.2 grvgaacc_fillrect

To make the hardware call for the fill rectangle framebuffer operation the cfb_fillrect function is replaced by this function. If the required prerequisites set by the limitations in the accelerator engine is not met the software algorithms are called as a default. If they are met, the function prepares the hardware call by calculating the address, destination index and fill pattern necessary for the hardware accelerator. The function then checks if the core

is busy, waits until it is available and writes the data to the memory addresses mapped to the APB slaves registers.

5.3.3 grvgaacc_copyarea

To make the hardware call for the copy area framebuffer operation the cfb_copyarea function is replaced by this function. If the required prerequisites set by the limitations in the accelerator engine is not met the software algorithms are called as a default. If they are met, the function prepares the hardware call by calculating the addresses, destination index and if the area needs to be reversed copied. The function then checks if the core is busy, waits until it is available and writes the data to the memory addresses mapped to the APB slaves registers.

5.3.4 grvgaacc_imageblit

To make the hardware call for the image blit framebuffer operation the cfb_imageblit function is replaced by this function. If the required prerequisites set by the limitations in the accelerator engine is not met the software algorithms are called as a default. If they are met, the function prepares the hardware call by calculating the addresses, destination offset and other parameters needed. If it is a monochrome blit the function then checks if the core is busy, waits until it is available and writes the data to the memory addresses mapped to the APB slaves registers. If the image has a higher color depth, the software color image blit algorithm is called.

6 Testing

The accelerator was to be tested on the GR-XC3S-1500 Development Board [4]. However, to be able to run Linux additional cores, such as an Ethernet interface and an FPU, is required. Unfortunately this made the system design too big for the Spartan-3 FPGA. The Spartan-3 was still used to run the tests in GRMON but to test the system while running Linux we used the ML501 Evaluation Platform [9] which features a larger Virtex-5 FPGA.

Testing has been performed all through the developing process. For the first applications, such as interfacing with the DMA, VHDL testbenches was used. The interface was verified in ModelSim [11], both by optically checking the simulated signals and by automatically comparing written data to the original data. Also, the handling of unaligned addresses was verified in this manner.

When a more complete accelerator core was achieved, the test environment changed to the development board. The design was synthesized using Synplify [12], place and route was done using Xilinx ISE [16] and programmed to the FPGA using iMPACT [13]. The use of the hardware monitor GRMON, available in the GRLIB package, enabled many new ways of verification.

While working in the GRMON environment test programs was written in C-code and run on the LEON3 CPU. This meant that the addressing and APB access to the core could be tested through software. Verification was done by optically confirming that the correct color and shape was displayed on the screen at the right place. The access to frambuffer memory and APB registers, to actually see the data, was also used while debugging the cores functionality.

When the functionality was fully verified, the drivers for SnapGear Linux was modified to make use of the accelerator. The use of the accelerator in console mode was confirmed by setting watch points in the code. The X window system was installed and the performance of was tested and compared with the benchmark program X11perf [14]. Unfortunately the X window system did not make use of the driver as intended and the test data from X11perf could not be used.

Instead test programs were written to simulate the driver's software algorithms and hardware calls. To compare hardware and software the two function calls are a similar sequence and the C-function Clock() was used to time the tests. Each framebuffer operation was performed five times to eliminate unwanted variations. These programs were tested in GRMON and the results were optimistic. However, due to the fact that the GRMON environment is locked at the color depth of 16 bits per pixel, the tests are also limited to this color depth.

7 Results

This chapter will present the results from the synthesis and the performance test programs. Data from both the fully functional version of the design and the reduced functionality version are presented and compared.

7.1 Synthesis

The size of the individual blocks are presented in Table 9. Both versions of the core are represented. Two blocks, Copyarea and Imageblit, are quite big and will need further work to reduce the size. The three smallest blocks should be the same size for both of the versions but differ some. This is discussed in Chapter 8.

Table 9: Area for each block of VGAACC, full and reduced functionality.

VGAACC: Blocks	Full		Reduced	
Name	LUT's [pcs]	Utilization [%]	LUT's [pcs]	Utilization [%]
APB Slave	293	4.09	307	5.21
Copyarea	3332	46.6	1697	28.8
Fillrect	884	12.4	652	11.1
Imageblit	2022	28.3	2545	43.2
DMA2AHB	199	2.78	211	3.58
Cache	0	0	43	0.73
Total LUT's VGAACC	7157	100	5891	100

The size of the whole accelerator core is presented in Table 10 and Table 11, the elements represent combinational and sequential logic respectively. The reduced function core should be smaller in size compared to the full function core, however as seen in Table 10 the total utilization of the FPGA's resources are bigger in the reduced function core. The size of the reduced function core versus the full function core is discussed further in Chapter 8.

Table 10: Combinational logic utilization for top level VGAACC.

VGAACC: Top Level Combinational Logic	Full	Reduced
Name	Elements [pcs]	Elements [pcs]
LUTS	7157	5891
MUXCY	1108	1034
XORCY	1015	905
MULT18x18/MULTI18x185	2	1
SRL16	0	0
Total Comb. Logic	9282	7831
Total Utilization	81.61%	82.31%

In Table 11 the number of registers used by the two versions of the core are presented. The total utilization of resources for the reduced function core is bigger than for the full function core here as well as in Table 10. Additional information of the size of the design can be found in Appendix A which contains the full area report from the synthesis.

Table 11: Sequential elements utilization for top level VGAACC.

VGAACC: Top Level Sequential Elements	Full	Reduced
Name	Elements [pcs]	Elements [pcs]
Registers	1345	1237
Total Utilization	11.83%	13.00%

It should also be mentioned that some timing errors exists in the design, although no problems has been noted during execution.

7.2 Performance

In order to evaluate the efficiency of the construction two test bench programs were made. The test benches were constructed to simulate the real case of software driver calls as well as possible. All test was preformed with

16 bit color depth at 1024x768 pixels screen resolution since GRMON only supports 16 bit color. Each of the two versions of the accelerator core was tested against the generic software algorithm from Linux, modified from the subroutines found of cfb_copyarea, cfb_filrect and cfb_imageblit. To compare hardware to software the two calls are a similar sequence and the C-function *Clock()* was used to time the tests.

The timing of the tests was performed in two different ways. The first to measure how much faster the accelerator core is to complete the framebuffer operation in comparison to the software algorithm, presented in section 7.2.1. And second, to measure the number of freed CPU cycles. That is, the time it takes to make the hardware call versus the time it takes to perform the framebuffer operation in software, presented in section 7.2.2.

7.2.1 Operation Acceleration

The results of the acceleration are presented in tables in this section and a more complete range of tests can be found in Appendix B.

The full function version of the core is as expected faster than the reduced function core, approximately 2 to 3 times faster depending on the length of the rows. Longer rows means less difference which seems reasonable since the main difference between the functions is that the software must make a new hardware call every row when using the reduced function core.

First of we have the results for the full function Fillrect operation.

Table 12: Full function Fill rectangle comparison.

Fill Rectangle				
Rectangle Size [Pixels]	Aligned [Software/Hardware]		Unaligned [Software/Hardware]	
	COPY	XOR	COPY	XOR
1024x768	35	9	-	-
1021x768	29	9	28	9
1021x1	11	9	13	7
4x768	31	17	8	10
512x384	33	10	18	9
32x32	18	12	7	12

In Table 12 the actual acceleration of the performed operations are presented as number of hardware executions per software execution, for

different rectangle sizes. As can be seen the acceleration ranges from 8 times to 35 times faster than the software algorithm, depending on rectangle size and raster operation.

Table 13: Reduced function Fill rectangle comparison.

Fill Rectangle				
Rectangle Size [Pixels]	Aligned [Software/Hardware]		Unaligned [Software/Hardware]	
	COPY	XOR	COPY	XOR
1024x768	9	8	-	-
1021x768	9	8	8	8
1021x1	5	5	5	5
4x768	1	0	6	5
512x384	5	6	5	5
32x32	5	5	5	5

The results for the reduced function Fillrect core are presented in Table 13 in a similar manner as the previous results. The acceleration varies, but as expected the larger, wider, rectangles are accelerated more since the hardware calls are repeated for each row. The reduced function core accelerates the fill rectangle operation by up to 9 times depending on rectangle size and raster operation. Even if there is no acceleration the CPU will still be freed up to execute other instructions by performing the hardware call, as presented in Chapter 7.2.2.

Next up are the results for the full function Copyarea core in Table 14. The performance improvement over the software algorithm ranges between 9 to 18 times, depending on area size and whether or not the copying is done in reverse. There are several different cases of unalignment for copy area, here the case of destination address unalignment is presented. Data on other cases can be found in Appendix B. The area sizes around 32 pixels in width are interesting because of the limit in 16 words per burst. Which means that an extra DMA access is needed when the limit is exceeded.

7.2 PERFORMANCE

Table 14: Full function Copy area comparison.

Copy Area				
Area Size [Pixels]	Aligned [Software/Hardware]		Dst-Unaligned [Software/Hardware]	
	Forward	Reverse	Forward	Reverse
1022x768	11	11	14	15
1021x768	11	11	15	15
512x768	11	12	-	-
33x33	10	13	13	13
32x32	18	15	11	13
31x31	9	13	13	16

Table 15 present the results for reduced function Copyarea core. These results are not as good as for the full function for smaller areas, which matches the results for the two Fillrect cores. The acceleration of the operation ranges in this case between 1 and 10 times, depending on area size and whether or not the copying is done in reverse.

Table 15: Reduced function Copy area comparison.

Copy Area				
Area Size [Pixels]	Aligned [Software/Hardware]		Dst-Unaligned [Software/Hardware]	
	Forward	Reverse	Forward	Reverse
1022x768	10	10	10	10
1021x768	10	9	10	9
512x768	9	9	-	-
33x33	1	1	1	1
32x32	1	1	2	1
31x31	1	1	1	1

Finally the results for the Imageblit core. In Table 16 the results for the full function version are presented. The acceleration of Imageblit is really

good for large images as presented by the results in Table 16. But in reality the blits performed are usually not that large. It is more realistic that the blits are the size of the mid range results. The full function hardware operation image blit can be performed between 2 and 275 times per software execution of the same operation, depending on image size and address alignment.

Table 16: Full function Image blit comparison.

Image Blit		
Image Size [Pixel]	Aligned [Software/Hardware]	Unaligned [Software/Hardware]
1024x768	94	-
1023x768	-	258
512x384	91	275
32x32	16	50
12x12	5	17
8x8	3	8
4x4	2	3

The results for reduced function Imageblit are presented in Table 17.

Table 17: Reduced function Image blit comparison.

Image Blit		
Image Size [Pixels]	Aligned [Software/Hardware]	Unaligned [Software/Hardware]
1024x768	49	-
1023x768	-	141
512x384	32	101
32x32	3	11
12x12	2	4
8x8	1	4

7.2 PERFORMANCE

The acceleration of the reduced function Imageblit compares well against the full function version and software algorithms, as can be seen in Table 17. The time to perform the image blit operation in the reduced function core is between 1 and 141 times faster than to execute the same operation in software, depending on image size and address alignment.

7.2.2 Operation Call Time

Data presented here are spot checks as an example of how the accelerator relieves the CPU. These tests have only been done for the fully functional core because the reduced function core is called in a loop and does not give as big impact for an area that is not that wide.

In Table 18 the usefulness of the accelerator core is clear as the difference between the software execution of the framebuffer operation and the hardware call well over 1000 clock cycles. For larger areas than 32x32 bits, the time to perform the framebuffer operation in software will increase but the number of cycles for the hardware call will remain approximately the same. This results in extra time for the CPU to do other operations.

Table 18: Operation Call Time Gain.

Operation Time Call (32x32 pixels)					
Software Operation vs Hardware Call	Fill rectangle [Clocks]		Copy area [Clocks]		Image blit [Clocks]
	COPY	XOR	Fwd	Rev	
Software Operation	1056	2064	2340	3265	2125
Hardware Call	27	24	58	110	57
Difference	1029	2040	2282	3155	2068

8 Conclusion

We conclude that the core is functional, usable and that it does accelerate the framebuffer operations as intended and presented in Chapter 7.2. In the cases where the acceleration of the actual operation is not that good, the CPU is still relived of the rendering workload and free to execute other tasks during the time the accelerator core performs the framebuffer operation.

Though the code is functional, rewriting it could still be beneficial in respect to eliminating the residual timing errors and to reduce the area. Since some of the blocks are quite large, a decrease in size is recommended before using the core in real world applications.

The two different versions might not be as useful as was intended since the difference in size is less than expected. As mentioned earlier, one of the reasons for making two versions of the accelerator core was to give the user of a platform with limited resources the option of a smaller core. The smaller size of the core should have compensated for the reduced functionality where the area of the core was an issue. However, as can be seen in Table 10 the total size differs only by a few percent and is almost negligible. A closer look at the numbers in Table 9 reveals that the reduced function core is even slightly larger than the full function core for some blocks.

This can be explained by the number of hours put into the different versions. The core with reduced functionality was a stepping stone to the core with full functionality and when the smaller core worked, the focus of the development process shifted to the fully functional core. This means that the later core went through more iterations of design improvements than the earlier core which was only revisited to compare the results of the synthesis and performance tests.

Though the performance in the X window system could not be tested during this project, someone with more experience of Linux and X should be able to adjust the necessary settings so that the modified driver can be used also with X. This would allow the more accepted X11pref performance test to be used as a benchmark while comparing the hardware accelerated graphics to the software's algorithms.

9 Discussion

The results for the fill rectangle operation are, as expected, best for an aligned rectangle with ROP COPY. In this case only one bus access per row is used and the data for the whole row is written in a single burst. There are also no read accesses during this operation. The unaligned ROP COPY is slightly less accelerated because of the leading and trailing bits that require existing data to be read from the framebuffer memory which means extra bus accesses. With ROP XOR there are read- write cycles throughout the operation and the size of the cache then limits the throughput. This causes saturation of the acceleration.

For the copy area operation the results are more even, if compared to the other operations. This is because there always are read- write cycles during the operation, which means that the size of the cache limits the throughput as in the fill rectangle ROP XOR case. Unaligned is accelerated more than aligned due to faster realignment of data in hardware than in software.

The results for the image blit operation are as expected. The hardware is faster at decoding the image data and larger images are accelerated to a greater extent because of fewer source data fetches, compared to the software algorithm. Unaligned blits gets a better improvement because of the slow software algorithm that only handle one pixel at a time, while the hardware implementation handles multiple pixels per transfer. A small acceleration decrement can be seen when source data in the cache is depleted and the operation needs to do another source data fetch.

The project took more time to follow though than what was expected and deadlines were postponed throughout the work. A reason for this was that the preliminary studies took a relatively large amount of the time. Since we have a background in hardware construction, our software knowledge was not that strong and too much time was spent to understand the software algorithms. Aeroflex Gaisler AB is a state of the art company and great help was at hand from the staff at all times, both with software and hardware questions, and we should have asked for help and used their knowledge more. A larger diversity of hardware solutions and drivers could also have been studied while researching the technical background. This would have made it easier to understand the algorithms used and some of the difficulties that emerged during the development could have been avoided.

We were free to specify the modules to be included in the core as we thought appropriate. This was very instructive because we had to consider and evaluate every construction choice that was made. However, this resulted in difficulties to define the project specification. The first choice of which parts, of the framebuffer operations, to implement in hardware was rushed and was too much too soon in terms of what was comprehended of the algorithms at that stage. By taking a step back and isolating problem areas, the work went on smoother. When the smaller problems were solved,

additional functions were added to the hardware implementation in iterations. This was a good development process and lead to the two versions of the core.

The last time consuming problem we encountered was that the modified driver was not used in the X windows system which made it impossible to get benchmarks of the hardware accelerator in Linux. However, even if the driver does not work in X, the core and driver has been verified to work in the Linux console at system start-up.

During the testing and verification a bug was discovered in the software algorithm of the copy area framebuffer operation. A bug report has been sent to the maintainer of the module.

To sum up, the results from the tests show that the core works and we feel that the project has been a success. Although the size of some of the blocks means that a rewrite of the code is necessary we hope that Aeroflex Gaisler AB can benefit from the end product. This was a very interesting and instructive project, the workload of the accelerator engine's implementation and design was appropriate for a master thesis for two students and we had fun with it.

10 Future Development

The first thing that comes to mind for future development is to look at the X window system and adjust its settings to make use of the modified Linux driver. However, the modified driver should be seen as a temporary patch since neither of us are proficient enough regarding C programming to write a driver that is integrated with the kernel. A separate driver for the accelerator should be developed to truly integrate the hardware calls into the Linux system. Also, the modified driver uses a busy-wait form for the hardware calls. This could be replaced with an interrupt steered call for a more efficient use of the CPU time.

The second is to eliminate the residual timing errors in the design and reduce the area by rewriting the VHDL code. One way to reduce the area of the IP core could be to merge the Fillrect and Copyarea function blocks, since they resemble each other.

To look further the GRLIB system will get a wider 64 bit data bus. This would benefit the accelerator and result in fewer bus accesses, faster data transfers and less strain on the bus.

Some work has been done to smooth the transition to incorporate the wider bus. Most of the data vectors widths has been set with a constant, declared in the VGAACC package. This constant could be exchanged for an generic value, which represent the system's bus width, and thereby adapt most of the core. Both the Copyarea and Fillrect blocks are prepared for a 64 bit bus and should not need much altering, although this has not been tested. For Imageblit the transition would be more complex. Larger mask tables will be needed in addition to a redesign of the current algorithm. Another thing to consider is that the local cache would have to double in size unless the burst length's of 16 word are reduced.

While running Linux on the system we noticed that the AHB was overloaded at higher resolutions and pixel depths. Too many cores was using the bus, the VGA controller, the Ethernet controller and the VGAACC core. One idea could be to put the graphics on a separate bus and have a separate graphics memory for the framebuffer. This would work nice for Copyarea and Fillrect but Imageblit will need access to system memory to fetch source data, however the amount of source data needed should be small.

Future development could also mean to add more frambuffer operations to the core, although this would mean alterations to the blocks and operation select signal it should relieve the CPU further.

References

[1] ARM, *AMBA Specification, Revision 2.0*, ARM, 5/13/1999

[2] J.Gaisler, E. Catovic, M. Isomäki, K. Glembo, S. Habinc, *GRLIB IP Core User's Manual, Version 1.0.20*, Gaisler Research AB, Gothenburg, 2/12/2009

[3] J. Gaisler, S. Habinc, E. Catovic, *GRLIB IP Library User's Manual, Version 1.0.20,* Gaisler Research AB, Gothenburg, 2/12/2009

[4] Gaisler Research / Pender Electronic Design GmbH, *GR-XC3S-1500 Development Board - User Manual, Revision 1.1*, Gaisler Research / Pender Electronic Design GmbH, 5/29/2006

[5] J. Gaisler, M. Isomäki, *LEON3 GR-XC3S-1500 Template Design*, Gaisler Research AB, Gothenburg, 2006

[6] Aeroflex Gaisler AB, *Http://www.gaisler.com*, Aeroflex Gaisler AB, 2008

[7] SPARC International, Inc., *Http://www.sparc.org*, SPARC International, Inc., 2008

[8] Unknown authors, *the Linux Cross Reference*, Http://lxr.linux.no, 2009

[9] Xilinx, *ML501 Evaluation Platform – User Guide, Version 1.4*, Xilinx, 8/29/2009

[10] Xilinx, *ML501 Evaluation Platform*, *Http://www.xilinx.com*, Xilinx, 2009

[11] Mentor Graphics, *ModelSim SE PLUS 6.1e, Revision 2006.03*, Mentor Graphics, 3/8/2006

[12] Synopsys, *Synplify PRO, Version C-2009.03*, Synopsys, 2/13/2009

[13] Xilinx, *iMPACT, Release 10.1.03*, Xilinx, 2008

[14] J. McCormack P. Karlton S. Angebranndt, C. Kent, K. Packard, G. Gill, *X11 server performance test program*, Http://www.xfree86.org, 2009

[15] J.Gaisler, A structured VHDL Design Method, Gaisler Research AB, Http://*www.gaisler.com/doc/**vhdl**2proc.pdf*, 2006

[16] Xilinx, *ISE, Release 10.1.03*, Xilinx, 2008

List of Figures

List of Tables

Appendix A : Synthesis

This section presents the detailed printout from the area report of the synthesis with Synplify[12]. The target technology is the GR-XC3S-1500 Development Board [4] with a Spartan3XC3S-1500-4 FPGA. The results are presented block by block in top-down order for both versions of the VGAACC design.

A.1 Full Functionality

```
---------------------------------------------------------------------------------------------------
Utilization report for Top level view: VGAACC
=============================================
SEQUENTIAL ELEMENTS
**********************
Name                                    Total elements          Utilization
---------------------------------------------------------------------------------------------------
REGISTERS                                    1345                    100 %
LATCHES                                      0                       0 %
=============================================
Total SEQUENTIAL ELEMENTS in the block VGAACC: 1345 (11.83 %
Utilization)

COMBINATIONAL LOGIC
************************
Name                                    Total elements          Utilization
---------------------------------------------------------------------------------------------------
LUTS                                         7157                    100 %
MUXCY                                        1108                    100 %
XORCY                                        1015                    100 %
MULT18x18/MULT18x18S                         2                       100 %
SRL16                                        0                       0 %
=============================================
Total COMBINATIONAL LOGIC in the block VGAACC: 9282 (81.61 %
Utilization)

Distributed RAM
****************
Name            Total elements      Number of LUTs          Utilization
---------------------------------------------------------------------------------------------------
DISTRIBUTED RAM     32                  32                      100 %
=============================================
Total Distributed RAM in the block VGAACC: 32 (0.28 % Utilization)

---------------------------------------------------------------------------------------------------
Utilization report for cell: APBslave
Instance path: VGAACC.APBslave
=============================================
SEQUENTIAL ELEMENTS
**********************
Name                                    Total elements          Utilization
---------------------------------------------------------------------------------------------------
REGISTERS                                    245                     18.2 %
LATCHES                                      0                       0 %
=============================================
```

Total SEQUENTIAL ELEMENTS in the block APBslave: 245 (2.15 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	293	4.09 %
MUXCY	0	0 %
XORCY	0	0 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

Total COMBINATIONAL LOGIC in the block APBslave: 293 (2.58 % Utilization)

Utilization report for cell: Copyarea
Instance path: VGAACC.Copyarea

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	457	34 %
LATCHES	0	0 %

Total SEQUENTIAL ELEMENTS in the block Copyarea: 457 (4.02 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	3332	46.6 %
MUXCY	588	53.1 %
XORCY	524	51.6 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

Total COMBINATIONAL LOGIC in the block Copyarea: 4444 (39.08 % Utilization)

Utilization report for cell: DMA2AHB
Instance path: VGAACC.DMA2AHB

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	115	8.55 %
LATCHES	0	0 %

Total SEQUENTIAL ELEMENTS in the block DMA2AHB: 115 (1.01 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	199	2.78 %
MUXCY	31	2.8 %
XORCY	31	3.05 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block DMA2AHB: 261 (2.29 % Utilization)

Utilization report for cell: Fillrect
Instance path: VGAACC.Fillrect

==========

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	190	14.1 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block Fillrect: 190 (1.67 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	884	12.4 %
MUXCY	144	13 %
XORCY	121	11.9 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block Fillrect: 1149 (10.10 % Utilization)

Utilization report for cell: Imageblit
Instance path: VGAACC.Imageblit

==========

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	334	24.8 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block Imageblit: 334 (2.94 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	2022	28.3 %
MUXCY	345	31.1 %
XORCY	339	33.4 %
MULT18x18/MULT18x18S	2	100 %
SRL16	0	0 %

Total COMBINATIONAL LOGIC in the block Imageblit: 2708 (23.81 % Utilization)

Utilization report for cell: Syncram
Instance path: VGAACC.Syncram

SEQUENTIAL ELEMENTS

Name	Total elements	Utilization
REGISTERS	4	0.2970 %
LATCHES	0	0 %

Total SEQUENTIAL ELEMENTS in the block Syncram: 4 (0.04 % Utilization)

Distributed RAM

Name	Total elements	Number of LUTs	Utilization
DISTRIBUTED RAM	32	32	100 %

Total Distributed RAM in the block Syncram: 32 (0.28 % Utilization)

Utilization report for cell: generic_syncram
Instance path: Syncram.generic_syncram

SEQUENTIAL ELEMENTS

Name	Total elements	Utilization
REGISTERS	4	0.2970 %
LATCHES	0	0 %

Total SEQUENTIAL ELEMENTS in the block Syncram.generic_syncram: 4 (0.04 % Utilization)

Distributed RAM

Name	Total elements	Number of LUTs	Utilization
DISTRIBUTED RAM	32	128	100 %

Total Distributed RAM in the block Syncram.generic_syncram: 32 (0.28 % Utilization)

A.2 Reduced Functionality

Utilization report for Top level view: VGAACC

==========

SEQUENTIAL ELEMENTS

****

Name	Total elements	Utilization
REGISTERS	1237	100 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block VGAACC: 1237 (13.00 % Utilization)

COMBINATIONAL LOGIC

****

Name	Total elements	Utilization
LUTS	5891	100 %
MUXCY	1034	100 %
XORCY	905	100 %
MULT18x18/MULT18x18S	1	100 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block VGAACC: 7831 (82.31 % Utilization)

Distributed RAM

****

Name	Total elements	Number of LUTs	Utilization
DISTRIBUTED RAM	32	32	100 %

==========

Total Distributed RAM in the block VGAACC: 32 (0.34 % Utilization)

Utilization report for cell: APBslave
Instance path: VGAACC.APBslave

==========

SEQUENTIAL ELEMENTS

****

Name	Total elements	Utilization
REGISTERS	251	20.3 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block VGAACC.APBslave :251 (2.64 % Utilization)

COMBINATIONAL LOGIC

****

Name	Total elements	Utilization
LUTS	307	5.21 %
MUXCY	0	0 %
XORCY	0	0 %

MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block VGAACC.APBslave: 307 (3.23 % Utilization)

Utilization report for cell: Copyarea
Instance path: VGAACC.Copyarea

==========

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	239	19.3 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block VGAACC.Copyarea: 239 (2.51 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	1697	28.8 %
MUXCY	250	24.2 %
XORCY	251	27.7 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block VGAACC.Copyarea: 2198 (23.10 % Utilization)

Utilization report for cell: DMA2AHB
Instance path: VGAACC.DMA2AHB

==========

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	115	9.3 %
LATCHES 0	0 %	

==========

Total SEQUENTIAL ELEMENTS in the block VGAACC.DMA2AHB: 115 (1.21 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	211	3.58 %
MUXCY	31	3 %
XORCY	31	3.43 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block VGAACC.DMA2AHB: 273 (2.87 % Utilization)

Utilization report for cell: FillRect
Instance path: VGAACC.FillRect

==========

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	110	8.89 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block VGAACC.FillRect: 110 (1.16 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	652	11.1 %
MUXCY	98	9.48 %
XORCY	79	8.73 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block VGAACC.FillRect: 829 (8.71 % Utilization)

Utilization report for cell: Imageblit
Instance path: VGAACC.Imageblit

==========

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization
REGISTERS	518	41.9 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block VGAACC.Imageblit: 518 (5.44 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization
LUTS	2545	43.2 %
MUXCY	655	63.3 %
XORCY	544	60.1 %
MULT18x18/MULT18x18S	100 %	
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block VGAACC.Imageblit: 3745 (39.36 % Utilization)

Utilization report for cell: Syncram
Instance path: VGAACC.Syncram

==========

SEQUENTIAL ELEMENTS

****

Name	Total elements	Utilization
REGISTERS	4	0.3230 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block VGAACC.Syncram:4 (0.04 % Utilization)

COMBINATIONAL LOGIC

****

Name	Total elements	Utilization
LUTS	43	0.730 %
MUXCY	0	0 %
XORCY	0	0 %
MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block VGAACC.Syncram: 43 (0.45 % Utilization)

Distributed RAM

****

Name	Total elements	Number of LUTs	Utilization
DISTRIBUTED RAM	32	32	100 %

==========

Total Distributed RAM in the block VGAACC.Syncram: 32 (0.34 % Utilization)

Utilization report for cell: generic_syncram
Instance path: Syncram.generic_syncram

==========

SEQUENTIAL ELEMENTS

****

Name	Total elements	Utilization
REGISTERS	4	0.3230 %
LATCHES	0	0 %

==========

Total SEQUENTIAL ELEMENTS in the block Syncram.generic_syncram:4 (0.04 % Utilization)

COMBINATIONAL LOGIC

****

Name	Total elements	Utilization
LUTS	43	0.730 %
MUXCY	0	0 %
XORCY	0	0 %

MULT18x18/MULT18x18S	0	0 %
SRL16	0	0 %

==========

Total COMBINATIONAL LOGIC in the block Syncram.generic_syncram: 43 (0.45 % Utilization)

Distributed RAM
****

Name	Total elements	Number of LUTs	Utilization
DISTRIBUTED RAM	32	128	100 %

==========

Total Distributed RAM in the block Syncram.generic_syncram: 32 (0.34 % Utilization)

Appendix B : Performance

This section presents the test bench results of the VGAACC. The tests has been performed on the GR-XC3S-1500 Development Board [4] in the GMON environment at a resolution of 1024x768pixels and a color depth of 16 bits per pixel. The results are presented for both versions of the VGAACC design and the time is measured in clock cycles.

B.1 Full Functionality

```
Fillrect: Fill full screen (1024x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 418193
Time, ACC: 11666
SOFT/ACC: 35
-=====================================-

Fillrect: Fill full screen (1024x768pixels), ROP_XOR
-=======================================-
Time, SOFT: 896888
Time, ACC: 89834
SOFT/ACC: 9
-=====================================-

Fillrect: Fill screen (Leading) (1021x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 424769
Time, ACC: 14900
SOFT/ACC: 28
-=====================================-

Fillrect: Fill screen (Trailing) (1021x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 424064
Time, ACC: 14604
SOFT/ACC: 29
-=====================================-

Fillrect: Fill screen (Leading,Trailing) (1022x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 429068
Time, ACC: 17724
SOFT/ACC: 24
-=====================================-

Fillrect: Fill screen (Leading) (1021x768pixels), ROP_XOR
-=======================================-
Time, SOFT: 893236
Time, ACC: 90040
SOFT/ACC: 9
-=====================================-

Fillrect: Fill screen (Trailing) (1021x768pixels), ROP_XOR;
-=======================================-
Time, SOFT: 894347
```

Time, ACC: 91718
SOFT/ACC: 9
-==========-

Fillrect: Fill screen (Leading,Trailing) (1022x768pixels), ROP_XOR
-==========-
Time, SOFT: 897451
Time, ACC: 91398
SOFT/ACC: 9
-==========-

Fillrect: Unaliged (Trailing) (1x768), ROP_COPY
-==========-
Time, SOFT: 14970
Time, ACC: 1042
SOFT/ACC: 14
-==========-

Fillrect: Unaliged (Leading) (1x768), ROP_COPY
-==========-
Time, SOFT: 15263
Time, ACC: 1215
SOFT/ACC: 12
-==========-

Fillrect: Unaligned (Trailing) (1x768pixels), ROP_XOR
-==========-
Time, SOFT: 15128
Time, ACC: 1197
SOFT/ACC: 12
-==========-

Fillrect: Aligned (1x768), ROP_COPY
-==========-
Time, SOFT: 15269
Time, ACC: 1213
SOFT/ACC: 12
-==========-

Fillrect: Aligned (1x768pixels), ROP_XOR
-==========-
Time, SOFT: 15259
Time, ACC: 1152
SOFT/ACC: 13
-==========-

Fillrect: Unaligned (Leading,Trailing) (2x768pixels), ROP_COPY
-==========-
Time, SOFT: 19848
Time, ACC: 1728
SOFT/ACC: 11
-==========-

Fillrect: Unaligned (Leading,Trailing) (2x768pixels), ROP_XOR
-==========-
Time, SOFT: 22496
Time, ACC: 2160

SOFT/ACC: 10
-====================================-

Fillrect: Aligned (2x768pixels), ROP_COPY
-==-
Time, SOFT: 15187
Time, ACC: 1134
SOFT/ACC: 13
-====================================-

Fillrect: Aligned (Trailing) (3x768pixels), ROP_COPY
-==-
Time, SOFT: 19691
Time, ACC: 1564
SOFT/ACC: 12
-====================================-

Fillrect: Unaligned (Leading) (3x768pixels), ROP_XOR
-==-
Time, SOFT: 22203
Time, ACC: 1128
SOFT/ACC: 19
-====================================-

Fillrect: Unaligned (Leading,Trailing) (4x768pixels), ROP_COPY
-==-
Time, SOFT: 23463
Time, ACC: 2714
SOFT/ACC: 8
-====================================-

Fillrect: Unaligned (Leading,Trailing) (4x768pixels), ROP_XOR
-==-
Time, SOFT: 26253
Time, ACC: 2486
SOFT/ACC: 10
-====================================-

Fillrect: Aligned (1024x1pixels), ROP_COPY
-==-
Time, SOFT: 534
Time, ACC: 38
SOFT/ACC: 13
-====================================-

Fillrect: Aligned (1024x1pixels), ROP_XOR
-==-
Time, SOFT: 1132
Time, ACC: 154
SOFT/ACC: 7
-====================================-

Fillrect: Aligned (1024x2pixels), ROP_COPY
-==-
Time, SOFT: 1241
Time, ACC: 56
SOFT/ACC: 22

-==================================-

Fillrect: Aligned (1024x2pixels), ROP_XOR
-==-
Time, SOFT: 2271
Time, ACC: 246
SOFT/ACC: 9
-==================================-

Fillrect: Aligned (Trailing) (1021x1pixels), ROP_COPY
-==-
Time, SOFT: 647
Time, ACC: 42
SOFT/ACC: 15
-==================================-

Fillrect: Aligned (Trailing) (1021x1pixels), ROP_XOR
-==-
Time, SOFT: 986
Time, ACC: 163
SOFT/ACC: 6
-==================================-

Fillrect: Unaligned (Leading) (1021x1pixels), ROP_COPY
-==-
Time, SOFT: 556
Time, ACC: 51
SOFT/ACC: 10
-==================================-

Fillrect: Unaligned (Leading) (1021x1pixels), ROP_XOR
-==-
Time, SOFT: 1133
Time, ACC: 166
SOFT/ACC: 6
-==================================-

Fillrect: Unaligned (Leading,Trailing) (1022x1pixels), ROP_COPY
-==-
Time, SOFT: 568
Time, ACC: 51
SOFT/ACC: 11
-==================================-

Fillrect: Unaligned (Leading,Trailing) (1022x1pixels), ROP_XOR
-==-
Time, SOFT: 1315
Time, ACC: 138
SOFT/ACC: 9
-==================================-

Fillrect: Unaligned (Leading,Trailing) (512x384pixels), ROP_COPY
-==-
Time, SOFT: 114932
Time, ACC: 6361
SOFT/ACC: 18
-==================================-

```
Fillrect: Unaligned (Leading,Trailing) (512x384pixels), ROP_XOR
-=======================================-
Time, SOFT: 230197
Time, ACC: 23160
SOFT/ACC: 9
-=====================================-

Fillrect: Unaligned (Leading,Trailing) (32x32pixels), ROP_COPY
-=======================================-
Time, SOFT: 1580
Time, ACC: 224
SOFT/ACC: 7
-=====================================-

Fillrect: Unaligned (Leading,Trailing) (32x32pixels), ROP_XOR
-=======================================-
Time, SOFT: 2251
Time, ACC: 203
SOFT/ACC: 11
-=====================================-

Fillrect: Aligned (32x32pixels), ROP_COPY
-=======================================-
Time, SOFT: 1347
Time, ACC: 63
SOFT/ACC: 21
-=====================================-

Fillrect: Aligned (32x32pixels), ROP_XOR
-=======================================-
Time, SOFT: 2101
Time, ACC: 157
SOFT/ACC: 13
-=====================================-

Fillrect: Aligned (512x384pixels), ROP_COPY
-=======================================-
Time, SOFT: 109500
Time, ACC: 3314
SOFT/ACC: 33
-=====================================-

Fillrect: Aligned (512x384pixels), ROP_XOR
-=======================================-
Time, SOFT: 228255
Time, ACC: 22684
SOFT/ACC: 10
-=====================================-

Fillrect: Aligned (4x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 16143
Time, ACC: 506
SOFT/ACC: 31
-=====================================-
```

Fillrect: Aligned (4x768pixels), ROP_XOR
-===-
Time, SOFT: 22184
Time, ACC: 1114
SOFT/ACC: 19
-===-
-===-

Copyarea: Copy quadrant (Reversed) (510x382pixels)
-===-
Time, SOFT: 281911
Time, ACC: 23566
SOFT/ACC: 11
-=====================================-

Copyarea: Copy quadrant (510x382pixels)
-===-
Time, SOFT: 271702
Time, ACC: 22739
SOFT/ACC: 11
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (1021x768pixels)
-===-
Time, SOFT: 1415482
Time, ACC: 92978
SOFT/ACC: 15
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (1022x768pixels)
-===-
Time, SOFT: 1421261
Time, ACC: 92797
SOFT/ACC: 15
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading) (1021x768pixels)
-===-
Time, SOFT: 1400718
Time, ACC: 92291
SOFT/ACC: 15
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (1022x768pixels)
-===-
Time, SOFT: 1405596
Time, ACC: 92448
SOFT/ACC: 15
-=====================================-

Copyarea: Aligned Copy (1022x768pixels)
-===-
Time, SOFT: 1054675
Time, ACC: 91875
SOFT/ACC: 11
-=====================================-

Copyarea: Aligned Copy (Trailing) (1021x768pixels)
-===-
Time, SOFT: 1056909
Time, ACC: 93406
SOFT/ACC: 11
-=================================-

Copyarea: Aligned Copy (Reversed) (1022x768pixels)
-===-
Time, SOFT: 1074849
Time, ACC: 90568
SOFT/ACC: 11
-=================================-

Copyarea: Aligned Copy (Reversed,Trailing) (1021x768pixels)
-===-
Time, SOFT: 1083917
Time, ACC: 91532
SOFT/ACC: 11
-=================================-

Copyarea: Aligned Copy (1024x384pixels)
-===-
Time, SOFT: 526355
Time, ACC: 44990
SOFT/ACC: 11
-=================================-

Copyarea: Aligned Copy (Trailing) (1023x384pixels)
-===-
Time, SOFT: 528708
Time, ACC: 45704
SOFT/ACC: 11
-=================================-

Copyarea: Dst-Unaligned Copy (Leading) (1021x384pixels)
-===-
Time, SOFT: 700396
Time, ACC: 45471
SOFT/ACC: 15
-=================================-

Copyarea: Src-Unaligned Copy (Trailing) (1021x384pixels)
-===-
Time, SOFT: 701675
Time, ACC: 46069
SOFT/ACC: 15
-=================================-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (1022x384pixels)
-===-
Time, SOFT: 703222
Time, ACC: 47131
SOFT/ACC: 14
-=================================-

Copyarea: Src-Unaligned Copy (1022x384pixels)

```
-=======================================-
Time, SOFT: 700575
Time, ACC: 46003
SOFT/ACC: 15
-=================================-

Copyarea: Aligned Copy (Reversed) (1024x384pixels)
-=======================================-
Time, SOFT: 538030
Time, ACC: 44990
SOFT/ACC: 11
-=================================-

Copyarea: Aligned Copy (Reversed,Trailing) (1023x384pixels)
-=======================================-
Time, SOFT: 544403
Time, ACC: 45429
SOFT/ACC: 11
-=================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (1021x384pixels)
-=======================================-
Time, SOFT: 708258
Time, ACC: 46838
SOFT/ACC: 15
-=================================-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (1021x384pixels)
-=======================================-
Time, SOFT: 706879
Time, ACC: 45824
SOFT/ACC: 15
-=================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed,Trailing)
(1022x384pixels)
-=======================================-
Time, SOFT: 710401
Time, ACC: 46438
SOFT/ACC: 15
-=================================-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (1022x384pixels)
-=======================================-
Time, SOFT: 705765
Time, ACC: 45464
SOFT/ACC: 15
-=================================-

Copyarea: Aligned Copy (512x768pixels)
-=======================================-
Time, SOFT: 539867
Time, ACC: 44906
SOFT/ACC: 12
-=================================-

Copyarea: Aligned Copy (Reversed)(512x768pixels)
```

```
-=======================================-
Time, SOFT: 552939
Time, ACC: 46095
SOFT/ACC: 11
-=======================================-

Copyarea: Aligned Copy (Trailing) (511x768pixels)
-=======================================-
Time, SOFT: 541899
Time, ACC: 46489
SOFT/ACC: 11
-=======================================-

Copyarea: Aligned Copy (Reversed,Trailing) (511x768pixels)
-=======================================-
Time, SOFT: 733540
Time, ACC: 47363
SOFT/ACC: 15
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading) (509x768pixels)
-=======================================-
Time, SOFT: 726530
Time, ACC: 45676
SOFT/ACC: 15
-=======================================-

Copyarea: Src-Unaligned Copy (Trailing) (509x768pixels)
-=======================================-
Time, SOFT: 728324
Time, ACC: 46821
SOFT/ACC: 15
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (510x768pixels)
-=======================================-
Time, SOFT: 728695
Time, ACC: 47012
SOFT/ACC: 15
-=======================================-

Copyarea: Src-Unaligned Copy (510x768pixels)
-=======================================-
Time, SOFT: 726730
Time, ACC: 46279
SOFT/ACC: 15
-=======================================-

Copyarea: Aligned Copy (Trailing) (31x31pixels)
-=======================================-
Time, SOFT: 2274
Time, ACC: 250
SOFT/ACC: 9
-=======================================-

Copyarea: Aligned Copy (32x32pixels)
-=======================================-
```

Time, SOFT: 2364
Time, ACC: 197
SOFT/ACC: 11
-==========-

Copyarea: Aligned Copy (Trailing) (33x33pixels)
-==========-
Time, SOFT: 2556
Time, ACC: 273
SOFT/ACC: 9
-==========-

Copyarea: Aligned Copy (Reversed,Trailing) (31x31pixels)
-==========-
Time, SOFT: 3233
Time, ACC: 274
SOFT/ACC: 11
-==========-

Copyarea: Aligned Copy (Reversed) (32x32pixels)
-==========-
Time, SOFT: 3215
Time, ACC: 248
SOFT/ACC: 12
-==========-

Copyarea: Aligned Copy (Reversed,Trailing) (33x33pixels)
-==========-
Time, SOFT: 3715
Time, ACC: 332
SOFT/ACC: 11
-==========-

Copyarea: Dst-Unaligned Copy (Leading) (31x31pixels)
-==========-
Time, SOFT: 3605
Time, ACC: 204
SOFT/ACC: 17
-==========-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (32x32pixels)
-==========-
Time, SOFT: 3656
Time, ACC: 244
SOFT/ACC: 14
-==========-

Copyarea: Dst-Unaligned Copy (Leading) (33x33pixels)
-==========-
Time, SOFT: 3816
Time, ACC: 232
SOFT/ACC: 16
-==========-

Copyarea: Dst-Unaligned Copy (Leading,Reversed,Trailing)
(31x31pixels)
-==========-

Time, SOFT: 3234
Time, ACC: 243
SOFT/ACC: 13
-==========-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (32x32pixels)
-==========-
Time, SOFT: 3285
Time, ACC: 241
SOFT/ACC: 13
-==========-

Copyarea: Dst-Unaligned Copy (Leading,Reversed,Trailing)
(33x33pixels)
-==========-
Time, SOFT: 3696
Time, ACC: 294
SOFT/ACC: 12
-==========-

Copyarea: Src-Unaligned Copy (Trailing) (31x31pixels)
-==========-
Time, SOFT: 3526
Time, ACC: 258
SOFT/ACC: 13
-==========-

Copyarea: Src-Unaligned Copy (32x32pixels)
-==========-
Time, SOFT: 3538
Time, ACC: 221
SOFT/ACC: 16
-==========-

Copyarea: Src-Unaligned Copy (Trailing) (33x33pixels)
-==========-
Time, SOFT: 3794
Time, ACC: 271
SOFT/ACC: 13
-==========-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (31x31pixels)
-==========-
Time, SOFT: 3726
Time, ACC: 230
SOFT/ACC: 16
-==========-

Copyarea: Src-Unaligned Copy (Reversed) (32x32pixels)
-==========-
Time, SOFT: 3865
Time, ACC: 276
SOFT/ACC: 13
-==========-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (33x33pixels)
-==========-

```
Time, SOFT: 3987
Time, ACC: 279
SOFT/ACC: 14
-=======================================-
-=======================================-

Imageblit: Aligned (32x32pixels)
-=======================================-
Time, SOFT: 2137
Time, ACC: 137
SOFT/ACC: 15
-=====================================-

Imageblit: Unaligned (32x32pixels)
-=======================================-
Time, SOFT: 6408
Time, ACC: 166
SOFT/ACC: 38
-=====================================-

Imageblit: Aligned (1024x768pixels)
-=======================================-
Time, SOFT: 1567258
Time, ACC: 16648
SOFT/ACC: 94
-=====================================-

Imageblit: Unaligned (1023x768pixels)
-=======================================-
Time, SOFT: 4675264
Time, ACC: 18044
SOFT/ACC: 259
-=====================================-

Imageblit: Unaligned (512x384pixels)
-=======================================-
Time, SOFT: 1172820
Time, ACC: 4238
SOFT/ACC: 276
-=====================================-

Imageblit: Aligned (512x384pixels)
-=======================================-
Time, SOFT: 392426
Time, ACC: 4086
SOFT/ACC: 96
-=====================================-

Imageblit: Aligned (8x8pixels)
-=======================================-
Time, SOFT: 215
Time, ACC: 61
SOFT/ACC: 3
-=====================================-

Imageblit: Unaligned (8x8pixels)
-=======================================-
```

Time, SOFT: 477
Time, ACC: 66
SOFT/ACC: 7
-==========-

Imageblit: Aligned (12x12pixels)
-==========-
Time, SOFT: 379
Time, ACC: 71
SOFT/ACC: 5
-==========-

Imageblit: Unaligned (12x12pixels)
-==========-
Time, SOFT: 1103
Time, ACC: 68
SOFT/ACC: 16
-==========-

Imageblit: Aligned (4x4pixels)
-==========-
Time, SOFT: 126
Time, ACC: 52
SOFT/ACC: 2
-==========-

Imageblit: Unaligned (4x4pixels)
-==========-
Time, SOFT: 155
Time, ACC: 54
SOFT/ACC: 2
-==========-

Imageblit: Aligned (64x64pixels)
-==========-
Time, SOFT: 8202
Time, ACC: 300
SOFT/ACC: 27
-==========-

Imageblit: Unaligned (64x64pixels)
-==========-
Time, SOFT: 24840
Time, ACC: 251
SOFT/ACC: 98
-==========-

Imageblit: Unaligned (33x32pixels)
-==========-
Time, SOFT: 6485
Time, ACC: 178
SOFT/ACC: 36
-==========-

Imageblit: Unaligned (31x32pixels)
-==========-
Time, SOFT: 6105

Time, ACC: 149
SOFT/ACC: 40
-=======================================-

B.2 Reduced Functionality

Fillrect: Fill full screen (1024x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 418001
Time, ACC: 31202
SOFT/ACC: 13
-=======================================-

Fillrect: Fill full screen (1024x768pixels), ROP_XOR
-=======================================-
Time, SOFT: 896907
Time, ACC: 102134
SOFT/ACC: 8
-=======================================-

Fillrect: Fill screen (Leading) (1021x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 424432
Time, ACC: 38217
SOFT/ACC: 11
-=======================================-

Fillrect: Fill screen (Trailing) (1021x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 424057
Time, ACC: 41472
SOFT/ACC: 10
-=======================================-

Fillrect: Fill screen (Leading,Trailing) (1022x768pixels), ROP_COPY
-=======================================-
Time, SOFT: 429406
Time, ACC: 45606
SOFT/ACC: 9
-=======================================-

Fillrect: Fill screen (Leading) (1021x768pixels), ROP_XOR
-=======================================-
Time, SOFT: 894041
Time, ACC: 101519
SOFT/ACC: 8
-=======================================-

Fillrect: Fill screen (Trailing) (1021x768pixels), ROP_XOR;
-=======================================-
Time, SOFT: 894388
Time, ACC: 103413
SOFT/ACC: 8
-=======================================-

Fillrect: Fill screen (Leading,Trailing) (1022x768pixels), ROP_XOR
-=======================================-

Time, SOFT: 897361
Time, ACC: 103366
SOFT/ACC: 8
-==========-

Fillrect: Unaliged (Trailing) (1x768), ROP_COPY
-==========-
Time, SOFT: 15159
Time, ACC: 15120
SOFT/ACC: 1
-==========-

Fillrect: Unaliged (Leading) (1x768), ROP_COPY
-==========-
Time, SOFT: 15173
Time, ACC: 15231
SOFT/ACC: 0
-==========-

Fillrect: Unaligned (Trailing) (1x768pixels), ROP_XOR
-==========-
Time, SOFT: 15087
Time, ACC: 15153
SOFT/ACC: 0
-==========-

Fillrect: Aligned (1x768), ROP_COPY
-==========-
Time, SOFT: 15287
Time, ACC: 15257
SOFT/ACC: 1
-==========-

Fillrect: Aligned (1x768pixels), ROP_XOR
-==========-
Time, SOFT: 15296
Time, ACC: 15192
SOFT/ACC: 1
-==========-

Fillrect: Unaligned (Leading,Trailing) (2x768pixels), ROP_COPY
-==========-
Time, SOFT: 19875
Time, ACC: 22520
SOFT/ACC: 0
-==========-

Fillrect: Unaligned (Leading,Trailing) (2x768pixels), ROP_XOR
-==========-
Time, SOFT: 22356
Time, ACC: 23512
SOFT/ACC: 0
-==========-

Fillrect: Aligned (2x768pixels), ROP_COPY
-==========-
Time, SOFT: 15155

Time, ACC: 15201
SOFT/ACC: 0
-==========-

Fillrect: Aligned (Trailing) (3x768pixels), ROP_COPY
-==========-
Time, SOFT: 19459
Time, ACC: 21607
SOFT/ACC: 0
-==========-

Fillrect: Unaligned (Leading) (3x768pixels), ROP_XOR
-==========-
Time, SOFT: 41827
Time, ACC: 44498
SOFT/ACC: 0
-==========-

Fillrect: Unaligned (Leading,Trailing) (4x768pixels), ROP_COPY
-==========-
Time, SOFT: 65334
Time, ACC: 69545
SOFT/ACC: 0
-==========-

Fillrect: Unaligned (Leading,Trailing) (4x768pixels), ROP_XOR
-==========-
Time, SOFT: 91502
Time, ACC: 98746
SOFT/ACC: 0
-==========-

Fillrect: Aligned (1024x1pixels), ROP_COPY
-==========-
Time, SOFT: 92141
Time, ACC: 98822
SOFT/ACC: 0
-==========-

Fillrect: Aligned (1024x1pixels), ROP_XOR
-==========-
Time, SOFT: 93154
Time, ACC: 99000
SOFT/ACC: 0
-==========-

Fillrect: Aligned (1024x2pixels), ROP_COPY
-==========-
Time, SOFT: 94387
Time, ACC: 99120
SOFT/ACC: 0
-==========-

Fillrect: Aligned (1024x2pixels), ROP_XOR
-==========-
Time, SOFT: 96693
Time, ACC: 99481

```
SOFT/ACC: 0
-====================================-

Fillrect: Aligned (Trailing) (1021x1pixels), ROP_COPY
-============================================-
Time, SOFT: 97265
Time, ACC: 99560
SOFT/ACC: 0
-====================================-

Fillrect: Aligned (Trailing) (1021x1pixels), ROP_XOR
-============================================-
Time, SOFT: 98582
Time, ACC: 99777
SOFT/ACC: 0
-====================================-

Fillrect: Unaligned (Leading) (1021x1pixels), ROP_COPY
-============================================-
Time, SOFT: 99138
Time, ACC: 99866
SOFT/ACC: 0
-====================================-

Fillrect: Unaligned (Leading) (1021x1pixels), ROP_XOR
-============================================-
Time, SOFT: 100406
Time, ACC: 100070
SOFT/ACC: 1
-====================================-

Fillrect: Unaligned (Leading,Trailing) (1022x1pixels), ROP_COPY
-============================================-
Time, SOFT: 101056
Time, ACC: 100165
SOFT/ACC: 1
-====================================-

Fillrect: Unaligned (Leading,Trailing) (1022x1pixels), ROP_XOR
-============================================-
Time, SOFT: 102390
Time, ACC: 100375
SOFT/ACC: 1
-====================================-

Fillrect: Unaligned (Leading,Trailing) (512x384pixels), ROP_COPY
-============================================-
Time, SOFT: 217551
Time, ACC: 119338
SOFT/ACC: 1
-====================================-

Fillrect: Unaligned (Leading,Trailing) (512x384pixels), ROP_XOR
-============================================-
Time, SOFT: 447810
Time, ACC: 147465
SOFT/ACC: 3
```

```
-==================================-

Fillrect: Unaligned (Leading,Trailing) (32x32pixels), ROP_COPY
-==========================================-
Time, SOFT: 449499
Time, ACC: 148561
SOFT/ACC: 3
-==================================-

Fillrect: Unaligned (Leading,Trailing) (32x32pixels), ROP_XOR
-==========================================-
Time, SOFT: 451663
Time, ACC: 149743
SOFT/ACC: 3
-==================================-

Fillrect: Aligned (32x32pixels), ROP_COPY
-==========================================-
Time, SOFT: 452902
Time, ACC: 150496
SOFT/ACC: 3
-==================================-

Fillrect: Aligned (32x32pixels), ROP_XOR
-==========================================-
Time, SOFT: 454883
Time, ACC: 151526
SOFT/ACC: 3
-==================================-

Fillrect: Aligned (512x384pixels), ROP_COPY
-==========================================-
Time, SOFT: 563265
Time, ACC: 166751
SOFT/ACC: 3
-==================================-

Fillrect: Aligned (512x384pixels), ROP_XOR
-==========================================-
Time, SOFT: 792751
Time, ACC: 195320
SOFT/ACC: 4
-==================================-

Fillrect: Aligned (4x768pixels), ROP_COPY
-==========================================-
Time, SOFT: 16283
Time, ACC: 15412
SOFT/ACC: 1
-==================================-

Fillrect: Aligned (4x768pixels), ROP_XOR
-==========================================-
Time, SOFT: 22372
Time, ACC: 22881
SOFT/ACC: 0
-==========================================-
```

```
-=======================================-

Copyarea: Copy quadrant (Reversed) (510x382pixels)
-=======================================-
Time, SOFT: 274959
Time, ACC: 29281
SOFT/ACC: 9
-=======================================-

Copyarea: Copy quadrant (510x382pixels)
-=======================================-
Time, SOFT: 268765
Time, ACC: 26272
SOFT/ACC: 10
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (1021x768pixels)
-=======================================-
Time, SOFT: 1082667
Time, ACC: 107012
SOFT/ACC: 10
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (1022x768pixels)
-=======================================-
Time, SOFT: 1075304
Time, ACC: 103946
SOFT/ACC: 10
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading) (1021x768pixels)
-=======================================-
Time, SOFT: 1056262
Time, ACC: 101757
SOFT/ACC: 10
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (1022x768pixels)
-=======================================-
Time, SOFT: 1055120
Time, ACC: 99788
SOFT/ACC: 10
-=======================================-

Copyarea: Aligned Copy (1022x768pixels)
-=======================================-
Time, SOFT: 1055108
Time, ACC: 99762
SOFT/ACC: 10
-=======================================-

Copyarea: Aligned Copy (Trailing) (1021x768pixels)
-=======================================-
Time, SOFT: 1056459
Time, ACC: 101873
SOFT/ACC: 10
-=======================================-
```

Copyarea: Aligned Copy (Reversed) (1022x768pixels)
-=======================================-
Time, SOFT: 1075657
Time, ACC: 103551
SOFT/ACC: 10
-=======================================-

Copyarea: Aligned Copy (Reversed,Trailing) (1021x768pixels)
-=======================================-
Time, SOFT: 1083334
Time, ACC: 107313
SOFT/ACC: 10
-=======================================-

Copyarea: Aligned Copy (1024x384pixels)
-=======================================-
Time, SOFT: 527002
Time, ACC: 49212
SOFT/ACC: 10
-=======================================-

Copyarea: Aligned Copy (Trailing) (1023x384pixels)
-=======================================-
Time, SOFT: 528639
Time, ACC: 50382
SOFT/ACC: 10
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading) (1021x384pixels)
-=======================================-
Time, SOFT: 528370
Time, ACC: 50323
SOFT/ACC: 10
-=======================================-

Copyarea: Src-Unaligned Copy (Trailing) (1021x384pixels)
-=======================================-
Time, SOFT: 528028
Time, ACC: 50762
SOFT/ACC: 10
-=======================================-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (1022x384pixels)
-=======================================-
Time, SOFT: 527976
Time, ACC: 49149
SOFT/ACC: 10
-=======================================-

Copyarea: Src-Unaligned Copy (1022x384pixels)
-=======================================-
Time, SOFT: 528026
Time, ACC: 49533
SOFT/ACC: 10
-=======================================-

Copyarea: Aligned Copy (Reversed) (1024x384pixels)
-=======================================-
Time, SOFT: 537484
Time, ACC: 51679
SOFT/ACC: 10
-=====================================-

Copyarea: Aligned Copy (Reversed,Trailing) (1023x384pixels)
-=======================================-
Time, SOFT: 544321
Time, ACC: 53452
SOFT/ACC: 10
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (1021x384pixels)
-=======================================-
Time, SOFT: 541718
Time, ACC: 53422
SOFT/ACC: 10
-=====================================-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (1021x384pixels)
-=======================================-
Time, SOFT: 541752
Time, ACC: 53592
SOFT/ACC: 10
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed,Trailing)
(1022x384pixels)
-=======================================-
Time, SOFT: 537606
Time, ACC: 52026
SOFT/ACC: 10
-=====================================-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (1022x384pixels)
-=======================================-
Time, SOFT: 541870
Time, ACC: 53673
SOFT/ACC: 10
-=====================================-

Copyarea: Aligned Copy (512x768pixels)
-=======================================-
Time, SOFT: 540875
Time, ACC: 53311
SOFT/ACC: 10
-=====================================-

Copyarea: Aligned Copy (Reversed)(512x768pixels)
-=======================================-
Time, SOFT: 552702
Time, ACC: 59566
SOFT/ACC: 9
-=====================================-

Copyarea: Aligned Copy (Trailing) (511x768pixels)
-===-
Time, SOFT: 542466
Time, ACC: 56462
SOFT/ACC: 9
-=====================================-

Copyarea: Aligned Copy (Reversed,Trailing) (511x768pixels)
-===-
Time, SOFT: 552622
Time, ACC: 59735
SOFT/ACC: 9
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading) (509x768pixels)
-===-
Time, SOFT: 540753
Time, ACC: 56251
SOFT/ACC: 9
-=====================================-

Copyarea: Src-Unaligned Copy (Trailing) (509x768pixels)
-===-
Time, SOFT: 540613
Time, ACC: 56559
SOFT/ACC: 9
-=====================================-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (510x768pixels)
-===-
Time, SOFT: 540234
Time, ACC: 53212
SOFT/ACC: 10
-=====================================-

Copyarea: Src-Unaligned Copy (510x768pixels)
-===-
Time, SOFT: 539922
Time, ACC: 53708
SOFT/ACC: 10
-=====================================-

Copyarea: Aligned Copy (Trailing) (31x31pixels)
-===-
Time, SOFT: 2305
Time, ACC: 1226
SOFT/ACC: 1
-=====================================-

Copyarea: Aligned Copy (32x32pixels)
-===-
Time, SOFT: 2668
Time, ACC: 1365
SOFT/ACC: 1
-=====================================-

Copyarea: Aligned Copy (Trailing) (33x33pixels)

```
-=======================================-
Time, SOFT: 2504
Time, ACC: 1338
SOFT/ACC: 1
-=================================-

Copyarea: Aligned Copy (Reversed,Trailing) (31x31pixels)
-=======================================-
Time, SOFT: 3145
Time, ACC: 1945
SOFT/ACC: 1
-=================================-

Copyarea: Aligned Copy (Reversed) (32x32pixels)
-=======================================-
Time, SOFT: 3256
Time, ACC: 1955
SOFT/ACC: 1
-=================================-

Copyarea: Aligned Copy (Reversed,Trailing) (33x33pixels)
-=======================================-
Time, SOFT: 3626
Time, ACC: 2473
SOFT/ACC: 1
-=================================-

Copyarea: Dst-Unaligned Copy (Leading) (31x31pixels)
-=======================================-
Time, SOFT: 2324
Time, ACC: 1197
SOFT/ACC: 1
-=================================-

Copyarea: Dst-Unaligned Copy (Leading,Trailing) (32x32pixels)
-=======================================-
Time, SOFT: 2365
Time, ACC: 1173
SOFT/ACC: 2
-=================================-

Copyarea: Dst-Unaligned Copy (Leading) (33x33pixels)
-=======================================-
Time, SOFT: 2518
Time, ACC: 1286
SOFT/ACC: 1
-=================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed,Trailing)
(31x31pixels)
-=======================================-
Time, SOFT: 3122
Time, ACC: 2003
SOFT/ACC: 1
-=================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed) (32x32pixels)
```

```
-=======================================-
Time, SOFT: 3304
Time, ACC: 2033
SOFT/ACC: 1
-=================================-

Copyarea: Dst-Unaligned Copy (Leading,Reversed,Trailing)
(33x33pixels)
-=======================================-
Time, SOFT: 3678
Time, ACC: 2436
SOFT/ACC: 1
-=================================-

Copyarea: Src-Unaligned Copy (Trailing) (31x31pixels)
-=======================================-
Time, SOFT: 2536
Time, ACC: 1324
SOFT/ACC: 1
-=================================-

Copyarea: Src-Unaligned Copy (32x32pixels)
-=======================================-
Time, SOFT: 2682
Time, ACC: 1224
SOFT/ACC: 2
-=================================-

Copyarea: Src-Unaligned Copy (Trailing) (33x33pixels)
-=======================================-
Time, SOFT: 2519
Time, ACC: 1347
SOFT/ACC: 1
-=================================-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (31x31pixels)
-=======================================-
Time, SOFT: 3118
Time, ACC: 1979
SOFT/ACC: 1
-=================================-

Copyarea: Src-Unaligned Copy (Reversed) (32x32pixels)
-=======================================-
Time, SOFT: 3341
Time, ACC: 2042
SOFT/ACC: 1
-=======================================-

Copyarea: Src-Unaligned Copy (Reversed,Trailing) (33x33pixels)
-=======================================-
Time, SOFT: 3531
Time, ACC: 2494
SOFT/ACC: 1
-=======================================-
-=======================================-
```

```
Imageblit: Aligned (32x32pixels)
-========================================-
Time, SOFT: 2277
Time, ACC: 344
SOFT/ACC: 6
-====================================-

Imageblit: Unaligned (32x32pixels)
-========================================-
Time, SOFT: 6584
Time, ACC: 347
SOFT/ACC: 18
-====================================-

Imageblit: Aligned (1024x768pixels)
-========================================-
Time, SOFT: 1567232
Time, ACC: 26501
SOFT/ACC: 59
-====================================-

Imageblit: Unaligned (1023x768pixels)
-========================================-
Time, SOFT: 4675505
Time, ACC: 29908
SOFT/ACC: 156
-====================================-

Imageblit: Unaligned (512x384pixels)
-========================================-
Time, SOFT: 1171805
Time, ACC: 11244
SOFT/ACC: 104
-====================================-

Imageblit: Aligned (512x384pixels)
-========================================-
Time, SOFT: 392706
Time, ACC: 9012
SOFT/ACC: 43
-====================================-

Imageblit: Aligned (8x8pixels)
-========================================-
Time, SOFT: 218
Time, ACC: 166
SOFT/ACC: 1
-====================================-

Imageblit: Unaligned (8x8pixels)
-========================================-
Time, SOFT: 486
Time, ACC: 117
SOFT/ACC: 4
-====================================-

Imageblit: Aligned (12x12pixels)
```

```
-========================================-
Time, SOFT: 384
Time, ACC: 166
SOFT/ACC: 2
-====================================-

Imageblit: Unaligned (12x12pixels)
-========================================-
Time, SOFT: 1059
Time, ACC: 183
SOFT/ACC: 5
-====================================-

Imageblit: Aligned (4x4pixels)
-========================================-
Time, SOFT: 132
Time, ACC: 110
SOFT/ACC: 1
-====================================-

Imageblit: Unaligned (4x4pixels)
-========================================-
Time, SOFT: 206
Time, ACC: 189
SOFT/ACC: 1
-====================================-

Imageblit: Aligned (64x64pixels)
-========================================-
Time, SOFT: 8350
Time, ACC: 755
SOFT/ACC: 11
-====================================-

Imageblit: Unaligned (64x64pixels)
-========================================-
Time, SOFT: 24662
Time, ACC: 767
SOFT/ACC: 32
-====================================-

Imageblit: Unaligned (33x32pixels)
-========================================-
Time, SOFT: 6409
Time, ACC: 316
SOFT/ACC: 20
-====================================-

Imageblit: Unaligned (31x32pixels)
-========================================-
Time, SOFT: 6529
Time, ACC: 419
SOFT/ACC: 15
-========================================-
-========================================-
```

Appendix C : Thesis Proposal

2D Acceleration engine for a Video Controller

Background

Gaisler Research develops and supports the GRLIB integrated VHDL IP library. The library is freely available in opensource, and includes blocks such as the LEON3 SPARC V8 processor, PCI, USB host/device controllers, CAN, DDR and ethernet interfaces. The AMBA onchip bus is used as the standard communication interface between the GRLIB cores.

Project description

The work will consist of developing twodimensional (2D) acceleration engine for an existing SVGA frame buffer. The GRLIB library contains an SVGA frame buffer which can display an image on a monitor using several different resolutions and color depths. All rendering is currently done by software, putting a relatively large burden on the system processor. The task will be to define and implement 2D rendering operations in a separate hardware engine to offload the processor.

The video controller is frequently used together with the LEON3 processor to run Xwindows on top of linux. For this, the frame buffer driver (fbdev) in the linux kernel is used. The driver has hooks for accelerated video functions, such as block moves and rectangle fills. These accelerated functions are the primary candidates to be implemented in hardware, as they can be used directly without having to develop a new video driver in the kernel.

The 2D acceleration engine will be implemented together with a leon3 system on the GRXC3S1500 FPGA development board. This board can support a full Leon3 system and also contains a 24bit video DAC and VGA connector.

The work will be split in the following tasks:

1. Development of a specification, defining which operations to be accelerated,supported resolution and color depth, register interface and DMA handling.
2. Implementing the 2D engine in VHDL, and verification in simulation.
3. Implementation on the Spartan3 GRXC3S1500 board
4. Testing of the accelerated functions using lowlevel C programs

5 Final testing of linux2.6 kernel with Xwindows

Qualifications

The applicant(s) should have strong interest in digital design, and be familiar the VHDL language and associated CAD tools. The work is suitable for one or two student(s). Support and mentoring will be provided by the supervisor and other Gaisler Research staff.