Hes·so VALAIS WALLIS
School of Engineering

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Degree Programme
Systems Engineering

Major Infotronics

# Bachelor's thesis
# Diploma 2017

*Joel Bodenmann*

## *2D Hardware Acceleration*

*Professor*
François Corthay

*Expert*
Yann Thoma

*Submission date of the report*
18.08.2017

HES-SO Valais-Wallis • rte du Rawyl 47 • C.P. • 1950 Sion 2
+41 27 606 85 23 • hei@hevs.ch • www.hevs.ch

Valais excellence

**HES-SO Valais**

| SYND | ETE | TEVI |
|---|---|---|
| X | X | X |

# Données du travail de diplôme
## *Aufgabenstellung der Bachelorarbeit*

FO 1.2.02.07.CB
haj/11/01/2016

| Filière / Studiengang<br>**SYND** | Année académique / *Studienjahr*<br>**2016/2017** | No TD / *Nr. DA*<br>**it/2017/42** |
|---|---|---|
| Mandant / *Auftraggeber*<br>☐ HES—SO Valais<br>☒ Industrie<br>☐ Etablissement partenaire *Partnerinstitution* | Etudiant / *Student*<br>**Joel Bodenmann**<br><br>Professeur / *Dozent*<br>**François Corthay** | Lieu d'exécution / *Ausführungsort*<br>☐ HES—SO Valais<br>☒ Industrie<br>☐ Etablissement partenaire *Partnerinstitution* |
| Travail confidentiel / *vertrauliche Arbeit*<br>☐ oui / ja [1] ☒ non / nein | Expert / *Experte (données complètes)*<br>**Yann Thoma**<br>http://reds.heig-vd.ch/equipe/details/yann.thoma | |

Titre / *Titel*

**VHDL operators for 2D graphic acceleration**

Description / *Beschreibung*

A 2D graphics computing system has been developed based on a NIOS processor softcore and a framebuffer within Intel FPGAs.

The aim of this work is to develop a library of IP cores in order to provide hardware acceleration and other advanced features. This will allow to offload a number of 2D graphics computer tasks from the processor. The system is aimed to be modular: the end user will have the choice of which functions will be implemented in hardware and which ones in software. The existing software environment will have to be adapted to properly integrate the modularity of the system.

This project will be realized in collaboration with B-electronics in Brigerbad and with Intel California.

Objectifs / *Ziele*

- Setup a development board with the NIOS processor and an operating system
- Install the uGFX libraries and develop a performance benchmark
- Build a list of operators to instanciate in hardware
- Develop the IP cores, interface them to the library and run the benchmark

| Signature ou visa / *Unterschrift oder Visum* | Délais / *Termine* |
|---|---|
| Responsable de l'orientation / filière<br>*Leiter der Vertiefungsrichtung / Studiengang:*<br>.....<br><br>1 Etudiant / Student :<br>..... | Attribution du thème / *Ausgabe des Auftrags:*<br>**15.05.2017**<br><br>Présentation intermédiaire / *Zwischenpräsentation*<br>**08 – 09.06.2017**<br><br>Remise du rapport / *Abgabe des Schlussberichts:*<br>**18.08.2017 / 12h00**<br><br>Expositions / *Ausstellungen der Diplomarbeiten:*<br>**30.08 – 31.08 – 01.09.2017**<br><br>Défense orale / *Mündliche Verfechtung:*<br>**Semaine 36 / *Woche 36*** |

1 *Par sa signature, l'étudiant-e s'engage à respecter strictement la directive DI.1.2.02.07 liée au travail de diplôme.*
*Durch seine Unterschrift verpflichtet sich der/die Student/in, sich an die Richtlinie DI.1.2.02.07 der Diplomarbeit zu halten.*

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats* .....

# 2D rendering with hardware acceleration

SILIZIUM

Graduate Joel Bodenmann

## Objectives

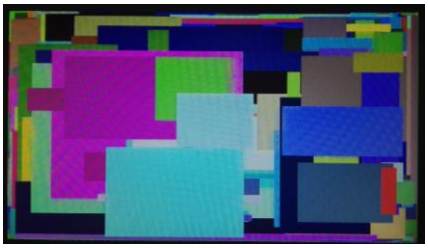
The objective of the project is to develop an IP-core that provides hardware acceleration for common 2D rendering operations in an embedded system.

## Methods | Experiences | Results

The requirements for graphical user interfaces (GUI) on modern display and touchscreen based systems are increasing steadily. Rendering complex and attractive GUIs requires a lot of processing power. At the same time, energy consumption for most of these embedded systems should decrease. Being able to off-load processor intensive tasks such as rendering of 2D shapes to dedicated hardware vastly decreases rendering time and frees a lot of processor resources which leads to a faster GUI and a less power consuming system.

Existing solutions often work only under very strict conditions and with specific platforms and are not flexible enough for the demands of the µGFX library which has been designed to run on virtually any system.

The result of the project is a ready-to-use IP-core that provides hardware acceleration for rendering solid rectangles in a NIOS-II based system. The design can be adapted to be used in any system. New hardware renderers for additional shapes can be added easily without modifying the rest of the design.



The *"drawing infinitely many rectangles"* to confirm the stability of the Silizium IP-core and to demonstrate the speed improvement.

A typical graphical user interface on an embedded system running NIOS-II and the µGFX library.

Bachelor's Thesis
| 2017 |

Degree programme
*Systems engineering*

Field of application
*Infotronics*

Supervising professor
*Dr François Corthay*
*francois.corthay@hevs.ch*

Partner
*uGFX GmbH*
*https://ugfx.io*

# Rendu en 2D avec accélération matérielle

Diplômant Joel Bodenmann

SILIZIUM

Travail de diplôme
| édition 2017 |

Filière
*Systèmes industriels*

Domaine d'application
*Infotronique*

Professeur responsable
*Dr François Corthay*
*francois.corthay@hevs.ch*

Partenaire
*uGFX Sarl*
*https://ugfx.io*

## Objectif du projet

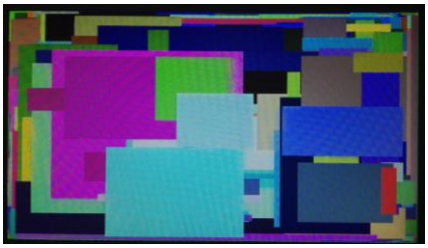
L'objectif du projet est de développer un coeur IP qui fournit une accélération matérielle pour les opérations de rendu graphique en 2D dans un système embarqué.

## Méthodes | Expériences | Résultats

Les exigences pour les interfaces utilisateur graphiques (*graphical user interface* / GUI) sur l'affichage moderne des systèmes à écran tactile augmentent régulièrement. Le rendu visuel des interfaces graphiques complexes et attrayantes nécessite beaucoup de puissance de traitement. En même temps, la consommation d'énergie de la plupart de ces systèmes embarqués est censée diminuer. Le fait de pouvoir décharger les tâches intensives du processeur telles que le rendu des formes 2D sur un circuit dédié diminue considérablement le temps de dessin et libère beaucoup de ressources du processeur. Ceci nous conduit à une interface graphique plus rapide et à un système moins consommateur d'énergie.

Les solutions existantes ne fonctionnent souvent que dans des conditions très strictes et sur des plates-formes spécifiques. Elles ne sont pas suffisamment souples pour répondre aux exigences de la bibliothèque µGFX qui a été conçue pour fonctionner sur pratiquement n'importe quel système.

Le résultat du projet est un coeur IP prêt à l'emploi qui fournit une accélération matérielle pour le dessin de rectangles solides dans un système basé sur NIOS-II. Le circuit peut être adapté pour être utilisé dans n'importe quel système. De nouveaux calculateurs matériels pour des formes supplémentaires peuvent être ajoutés facilement sans modifier le reste de la conception.



Le «dessin d'infiniment de rectangles» pour confirmer la stabilité du coeur IP Silizium et pour démontrer l'amélioration de la vitesse.

Une interface utilisateur graphique typique sur un système embarqué fonctionnant avec NIOS-II et la bibliothèque µGFX.

# Table of Contents

# 1 Preamble

This is the official and final report for the bachelor thesis.

# 2 Motivation

This project has been proposed as a bachelor thesis by myself. As the author and maintainer of the µGFX[1] library I know how many CPU resources rendering even simple shapes such as a solid rectangle can take. On small microcontroller systems or FPGAs with soft-cores this is usually a big problem as these CPUs are not only comparably slow but also need to take care of a magnitude of other things such as processing user events or handling wireless communication which often involve hard-realtime requirements. These hard-realtime requirements often require the 2D rendering tasks to be split up into multiple different operations which makes rendering a user interface even slower and therefore quickly leads to a non-smooth user interface which does not hold up to the ever-growing performance demands of modern graphical user interfaces.

Being able to off-load even simple tasks such as rendering a filled rectangle to a dedicated part of the hardware (GPU) doesn't only vastly increase rendering speed but also frees up a lot of CPU resources.

Dedicated hardware to off-load rendering operations for smaller systems is nothing new. There are several display controllers such as the *RA8875* from *RAiO* which provide hardware support for rendering basic 2D shapes. Furthermore, there are more complex systems such as the *FT800* from *FTDI* which implement hardware rendering for complex shapes and widgets such as pushbuttons, sliders and even entire on-screen keyboards. More advanced microcontrollers that feature built-in display controllers also start to provide very basic hardware acceleration support for rectangle drawing and other basic operations such as the *LTDC* display controller that can be found in some of the higher-end *STM32* microcontrollers. These existing solutions tend to be very inflexible and are always highly proprietary. They can almost always only be used with the matching closed-source software of the vendor or they come with other restrictions that vastly limit the field of use. The goal of this project is to develop a system that can be integrated easily into any existing and new system.

# 3 Project goal

The goal of this thesis is to develop an IP-core that provides 2D hardware acceleration to a system-on-chip[2] (SoC) to speed up graphical user interfaces on embedded systems.

More specifically, the IP-core will be implemented in VHDL[3] and will be optimized for the use with the µGFX library and the NIOS-II[4] processor.

While we will refer to this IP-core as *"a GPU"* (graphical processing unit) it is not to be compared to a traditional GPU of a desktop computer system. The goal of this project is to implement hardware 2D acceleration for small embedded systems that do not require fancy animations or 3D renderings. The

[1] https://ugfx.io
[2] https://en.wikipedia.org/wiki/System_on_a_chip
[3] https://en.wikipedia.org/wiki/VHDL
[4] https://www.altera.com/products/processors/overview.html

GPU will be optimized to be small & simple and also easy to use for the CPU as we want to save as much CPU time as possible on those smaller low-performance systems.

# 4 Document structure

The complete bachelor thesis report consists of three separate documents:

- The actual report itself (this document)
- The datasheet of the developed IP-Core
- A document that describes the problem that was faced when implementing the Avalon-MM master bus interface

These three documents refer to each other were necessary. To a person unfamiliar with the project it is best to start with the datasheet and then moving on to reading the rest of this report.

The reason for having a separate document regarding the problem with the Avalon-MM master bus interface is that the problem has not been solved yet. A workaround that is suitable to finish this thesis has been found and applied but solving the problem has yet to be done.

# 5 Terms & abbreviations

The following table gives an overview of terms and abbreviations that are commonly used throughout this document:

| Abbreviation | Description |
|---|---|
| 2D | Two dimensional |
| Avalon | A bus standard created by Altera for FPGA internal communication |
| CPU | Central processing unit |
| FBI | Framebuffer interface |
| FIFO | First-in First-out (a type of memory/buffer) |
| Framebuffer | A section of memory that holds the pixel data that is shown on the display |
| FSM | Finite state machine |
| GPU | Graphics processing unit |
| HAL | Hardware abstraction layer |
| IP-Core | (Intellectual property) A pre-fabricated block of something ready to be used |
| Qsys | Tool of the Quartus toolchain used to create a SoC |
| Quartus | The FPGA IDE & Toolchain by Intel |
| Silizium | The name of this 2D hardware acceleration IP-Core |
| SoC | System-on-chip |

*Table 1: Commonly used abbreviations & terms in this document*

# 6 Technical advantages

This section of the document explains the technical advantages of having dedicated hardware for 2D rendering operations. This is basically the technical variant of section 2 (motivation). This section assumes that the reader is familiar with the basic concepts of a computer system (especially the CPU).

A CPU is usually designed to be very good at executing a broad variety of different tasks (tasks in terms of logical operations and bit operations which together form calculations). Just as with most other things, when something is designed to be usable for many different things it is usually not very good at one particular thing. While a CPU offers everything required to render two dimensional shapes such as rectangles and polygons or to copy images around, it is not really optimized for that. This section of the document contains two examples that will illustrate why having dedicated hardware for 2D rendering operations can be a huge benefit in terms of the overall application speed.

## 6.1 Example 1: Image blitting

Blitting (historically also known as *bit blit*) stands for *bit block transfer* and describes a technique to assemble a bitmap (ultimately an image) from different parts of multiple different images. In layman's terms, it's taking a region of pixels and copying to a different location. A good example is rendering an image: When the developer of an (embedded) GUI wants to show an image on the display said image needs to be loaded from memory (eg. an SD-Card). Afterwards, the image needs to be decoded (usually by the CPU) and gets cached (*"stored temporarily"*) into memory. The cached version of this image is somewhere in memory but not inside the framebuffer. This means that the image is fully rendered but simply not on the display yet. Therefore, the next operation is to create a copy of the cached (decoded) image in memory. Copying memory is a very inefficient task for a CPU. To copy memory, a CPU must read a chunk of memory and store it in its internal registers and then write it back to a different memory location. As those registers are usually just big enough for a few bytes (eg. 32 to 48 bytes in a typical modern microcontroller CPU) a lot of those read-store-write transactions need to be executed to copy a rendered image which often takes 2 to 4 bytes of memory per pixel. Each of those read-store-write transactions comes with overhead and also keeps the CPU from doing anything else. Assuming a small icon of 64 x 64 pixels size with a color format that takes 4 bytes per pixel it would take a CPU with the ability to store 32 bytes in registers at once 512 of those transactions:

$$number\ of\ transactions = \frac{64 * 64 * 4\ bytes}{32\ bytes} = 512$$

Almost all embedded systems (such as microcontroller) which would be used to implement graphical user interface offer DMA[5] (direct memory access) which is a part of dedicated hardware next to the CPU that allows copying memory without using/occupying the CPU. However, those DMAs cannot be used efficiently in such an application as they only support linear memory spaces. Copying a two-dimensional object in a framebuffer requires to wrap at the end of the object and then move to a different starting address for the next row. Traditionally DMAs cannot do this as they only provide a start and end address configuration parameter and constantly increment between the two. Therefore, a CPU would have to setup the DMA to copy one row of the image, wait for it to finish and then configure it to copy the next row.

[5] https://en.wikipedia.org/wiki/Direct_memory_access

A hardware blitting engine is basically a DMA that knows the concept of having two-dimensional information in a linear memory space (a framebuffer). Instead of a start and stop address it gets provided with the start address and the width and height of the object in pixels. There are two benefits of having dedicated hardware for this:

- This dedicated hardware will be a lot faster at copying that memory (as it has been built especially for this task)
- After configuring and starting the hardware blitting engine the CPU can occupy itself with other things such as reacting on user inputs or decoding the next resource (eg. image) that will be used

These two benefits combined speed up the final application a lot as the operation of copying the image is faster and the latency to react on user input and similar decreases at the same time.

While the latest and most high-end microcontrollers such as some of the STM32 F4 and F7 series microcontrollers are equipped with such a DMA2D it is still rarely the case and in all cases those two-dimensional DMAs are only able to be used together with the built-in display controller of those microcontroller which in turn imply a large set of new limitations.

## 6.2 Example 2: Area filling

In the first example, the main argument for having a dedicated piece of hardware was that a CPU is a lot slower at copying memory around than dedicated hardware that has been optimized for this. As the dedicated hardware is a lot faster performing the same operation the time required to perform the copying operation drops down. However, there are also cases where the CPU itself would be fast enough to perform the job but having dedicated hardware still saves overall time because the CPU can perform other tasks in the meantime. A good & easy to understand example for this is area filling (rendering a filled rectangle in the framebuffer). Ultimately, the value of each pixel in the framebuffer needs to be modified. A (modern) CPU is not a whole not slower at this compared to dedicated hardware but when the CPU can off-load this task to dedicated hardware the CPU has time to perform other tasks.

The two examples above illustrate why and how dedicated hardware for 2D rendering operations can vastly speed up graphical user interfaces.

# 7 Used tools

The overall goal of this project is to develop an IP-core that can be used in any system afterwards. However, for the scope of this bachelor thesis project we will focus on implementing a design that runs on a MAX10 FPGA using the NIOS-II system. The following tools are used to achieve that goal:

Software

- Quartus 17.0 (Quartus Prime Version 17.0.0 Build 595 04/25/2017 SJ Standard Edition)
- Eclipse for NIOS-II Kepler Service Revision 2, Build 20140224-0627
- ModelSim Intel FPGA starter edition 10.5b (Revision 2016.10)
- µGFX library v2.7

Hardware

- Terasic MAX10 Neek[6]

# 8 QSys

A NIOS-II based system is usually developed using a utility named *QSys*[7] that is part of Intels FPGA development suite named *Quartus*[8]. QSys provides a graphical interface for designing a SoC. As the 2D hardware acceleration IP-core would be part of the resulting SoC it is essential that the developed block can be added easily to a QSys project through the graphical user interface.

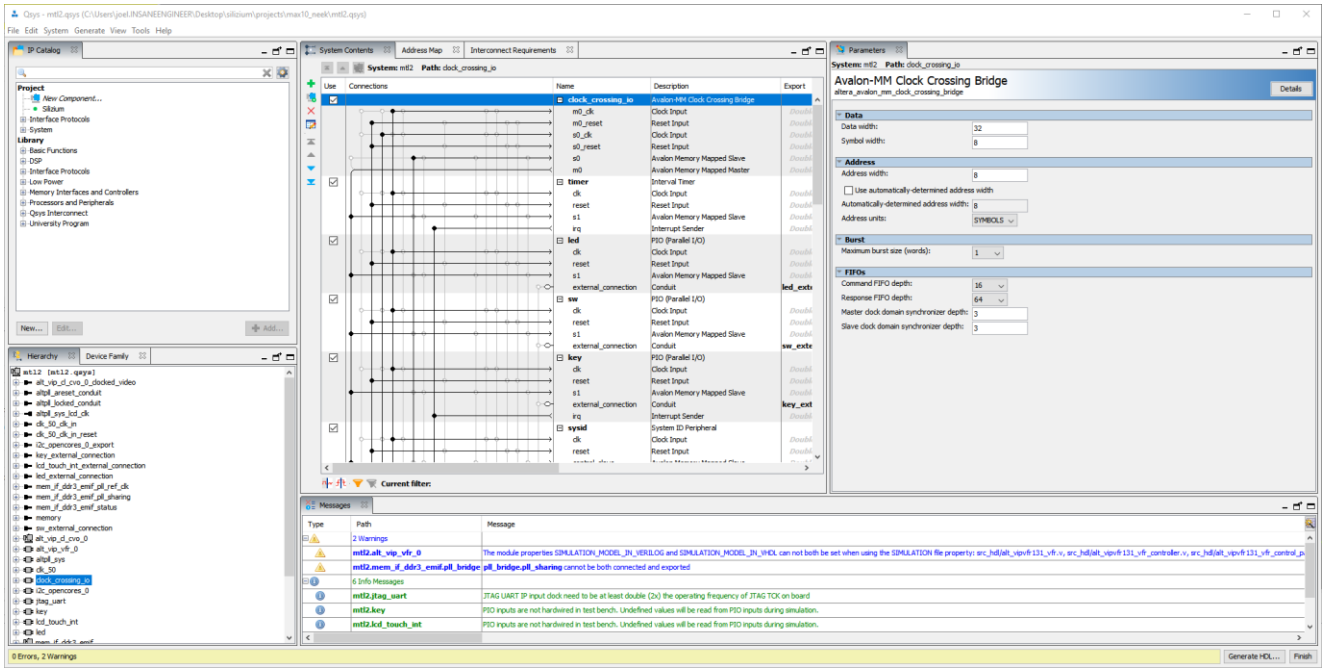



*Figure 1: QSys screenshot*

[6] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=956
[7] https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-qsys.html
[8] https://en.wikipedia.org/wiki/Altera_Quartus

### 8.1 Avalon

The IP-core that provides the hardware accelerations needs an interface through which it can communicate with the CPU. The NIOS-II CPU relies heavily on the Avalon bus[9], specifically the Avalon memory mapped (Avalon-MM) bus. The Avalon-MM bus offers a master-slave topology with multi-master capabilities. Our IP-core will have an Avalon-MM bus slave through which the CPU can send commands to the GPU. Furthermore, the GPU will also feature an Avalon-MM bus master that allows accessing the framebuffer.

## 9 Simplifications

Due to time limitations, the following decisions have been made to keep the design simpler:

- No parallel rendering

After identifying the resources & systems that will be used and after deciding which simplifications are being made for the scope of this bachelor thesis it is time to determine which functions bring the most benefits when implemented in hardware.

## 10 Hardware acceleration features

A crucial step of the project is to identify which rendering functions bring the most benefits when implemented in hardware. For this, Table 2 has been created that lists the basic rendering functions provided by the µGFX library together with a rating of their complexity (in terms of implementation) and how often they are used in a typical application. From those two values, a priority rating has been deduced from which in turn the order of implementation for this project has been deduced.

Column description:

- **Complexity:** The complexity of the existing software algorithm relative to all other rendering algorithms in the table. 10 is the most complex.
- **Occurrences:** The number of occurrences in a practical real-world GUI implementation relative to all other rendering algorithms in that table. 10 is the most used one.
- **Priority:** The priority in terms of order of implementation for this thesis/project.

The complete API reference for all primitive rendering functions that the µGFX library provides can be found here: http://api.ugfx.io/group___g_d_i_s_p.html

[9] https://en.wikipedia.org/wiki/Nios_II#Avalon_switch_fabric_interface

| Function | Description | Complexity | Occurrences | Priority |
|---|---|---|---|---|
| Blend Colors | Blend two colors | 1 | 3 | - |
| Draw Pixel | Draw a single pixel | 1 | 10-5 | - |
| Draw Line | Draw a line (one pixel thick) | 5 | 8 | 3 |
| Fill Area | Fill an area with a solid color | 1 | 10 | 1 |
| Blit Area | Fill an area with a given buffer (2D memory copy) | 2 | 6 | 2 |
| Draw Box | Draw an unfilled rectangle | 5 | 6 | 4 |
| Stream (Start, Color, End) | Stream pixel values | 2 | 2 | - |
| Draw Circle | Draw an unfilled circle | 7 | 4 | 9 |
| Fill Circle | Draw a filled circle | 7 | 4 | 10 |
| Fill Dual Circle | Draw a filled circle with border | 8 | 1 | 11 |
| Draw Ellipse | Draw an unfilled ellipse | 9 | 2 | 12 |
| Fill Ellipse | Draw a filled ellipse | 9 | 2 | 13 |
| Draw Arc Sectors | Draw one or more arc sectors | 9 | 2 | - |
| Draw Thick Arc | Draw a thick arc | 9 | 3 | - |
| Fill Arc | Draw a filled arc | 9 | 3 | - |
| Vertical Scroll | Scroll a certain area of the framebuffer | 2 | 5 | 6 |
| Draw Polygon | Draw an unfilled polygon | 7 | 4 | 5 |
| Fill Convex Polygon | Draw a filled (convex) polygon | 7 | 7 | 7 |
| Draw Thick Line | Draw a thick line | 7 | 5 | 8 |
| Draw Character | Draw a single character | 10 | 10 | - |
| Fill Character | Draw a single character and fill the background | 10 | 10 | - |
| Draw String | Draw a string | 10 | 10 | - |
| Fill String | Draw a string and fill the background | 10 | 10 | - |
| Draw String Box | Draw a string with justification | 10 | 10 | - |
| Fill String Box | Draw a string with justification and fill the backgrounds | 10 | 10 | - |
| Draw Rounded Box | Draw an unfilled rectangle with rounded corners | 5 | 5 | - |
| Fill Rounded Box | Draw a filled rectangle with rounder corners | 5 | 5 | - |

*Table 2: µGFX rendering functions overview*

# 11 Architecture

Initially the idea was to create one IP-core per hardware acceleration function. However, after reading a lot of material about QSys it became clear that the best solution is to create just one IP-core that internally dispatches the jobs to the different sub components. The reason for this is that all of these components would require the same bus interface which would result in a lot of code duplication and would also require the user of the IP-core to hook up each block manually which quickly becomes a tedious task with a growing number of renderers and also decreases readability of the overall SoC design. Furthermore, the QSys tool provides a nice graphical configuration dialog for each IP-core. Once the Silizium IP-core is completed we can create a configuration dialog where each hardware acceleration feature can be either enabled or disabled and further configured. Another advantage of having just one IP-core is that we can handle the bus arbitration internally ourselves. This allows for further optimizations such as running multiple different rendering jobs in parallel, optimizing for burst transactions and similar.

From the end user's perspective, there will be just one IP-core with one Avalon-MM slave and one Avalon-MM master interface:


*Figure 2: IP core interfaces*

The slave interface will be used to configure and control the GPU as well as issuing rendering commands. The master interface is used to access the framebuffer memory.

Figure 3 shows two different use cases for the Silizium IP-core in a typical NIOS-II based SoC. It is to note that the Silizium block is not affected by other aspects of the systems such as the location of the framebuffer (eg. Internal or external memory) or the used display interface.

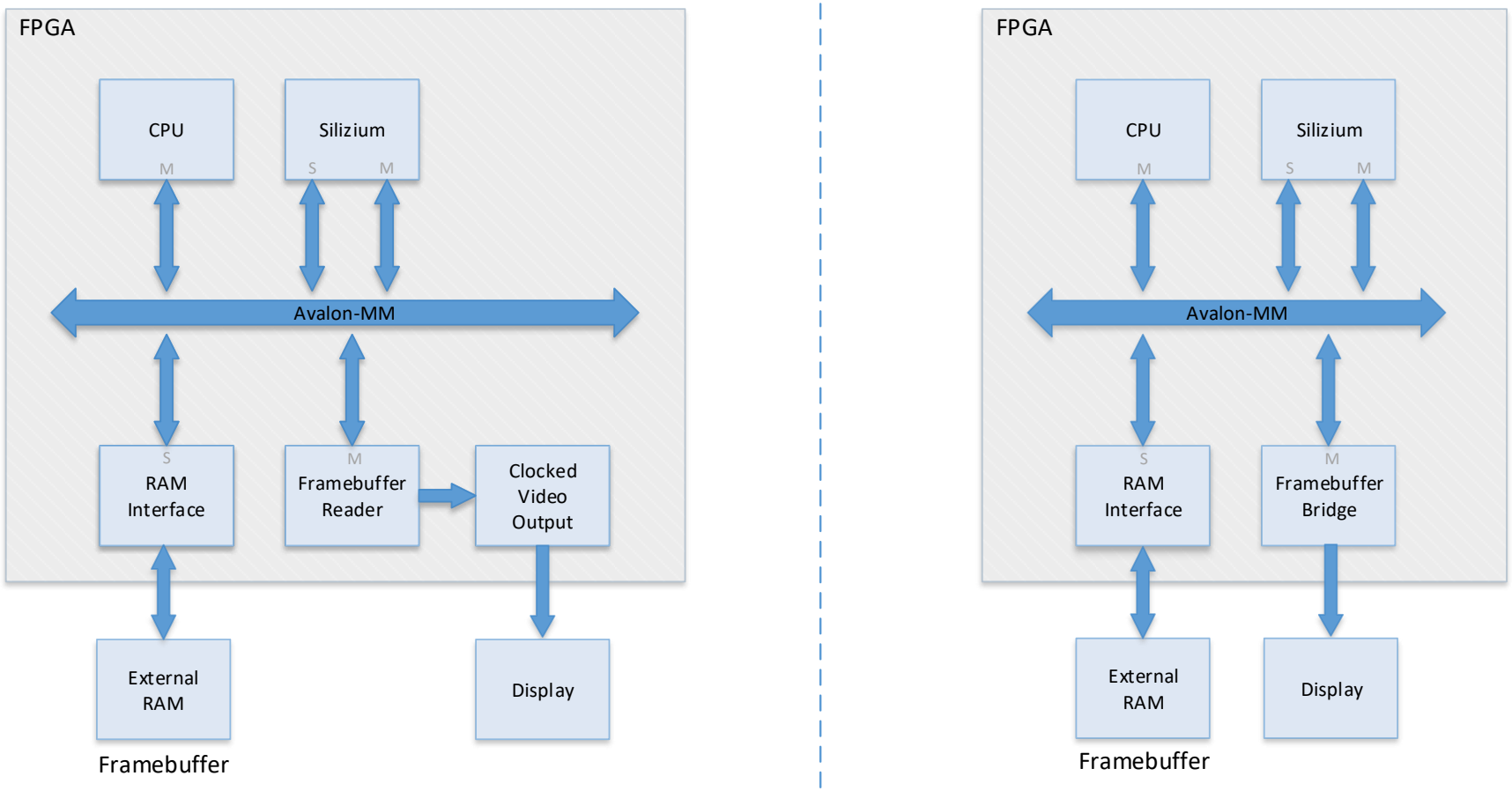



*Figure 3: Typical applications / use cases*

# 12 Internal design

Figure 4 gives an overview of the internal design of the Silizium IP-core. The design can be split into the following components:

- Command & Control bus interface
- Registers
- Command FIFO
- Dispatcher
- Renderers
- Framebuffer interface

Note: The purpose of said figure is to give an overview of how things work internally. The actual implementation is split up into different entities.

## 12.1 Command & Control bus interface

The command & control bus interface (from here on referred to as *"the slave interface"*) is used by the CPU to initialize, configure and control the GPU. The interface is compliant to the Avalon-MM slave standard and has the following requirements:

- Address bus width: At least 4 bits
- Data bus width: At least 32 bits
- Read and write operations

The run-time configuration of the GPU usually only consists of setting the framebuffer base address and the framebuffer span. Other parameters such as the display size, the pixel format and similar are handled by generic values and therefore do not need to be changed during run-time. Once the configuration is completed the slave interface is usually only used to send commands to the GPU, to control the different enable flags (if required) and to read back the current status of the GPU (eg. busy flags) and the command queue status.

## 12.2 Registers

The GPU features different control and configuration registers. The most important ones are:

- Status register
- Control register
- Framebuffer base address & span register

These registers can be accessed directly via the slave interface.

The documentation of the values and effects of these registers are documented in the datasheet.

## 12.3 Command FIFO

The CPU issues commands to the GPU such as *"draw a rectangle at this position with that size and this color"*. These commands are stocked in the internal command FIFO of the GPU. Whenever there is a pending command in the FIFO the dispatcher will grab it as soon as the different renderers of the GPU are no longer busy and handle it accordingly.

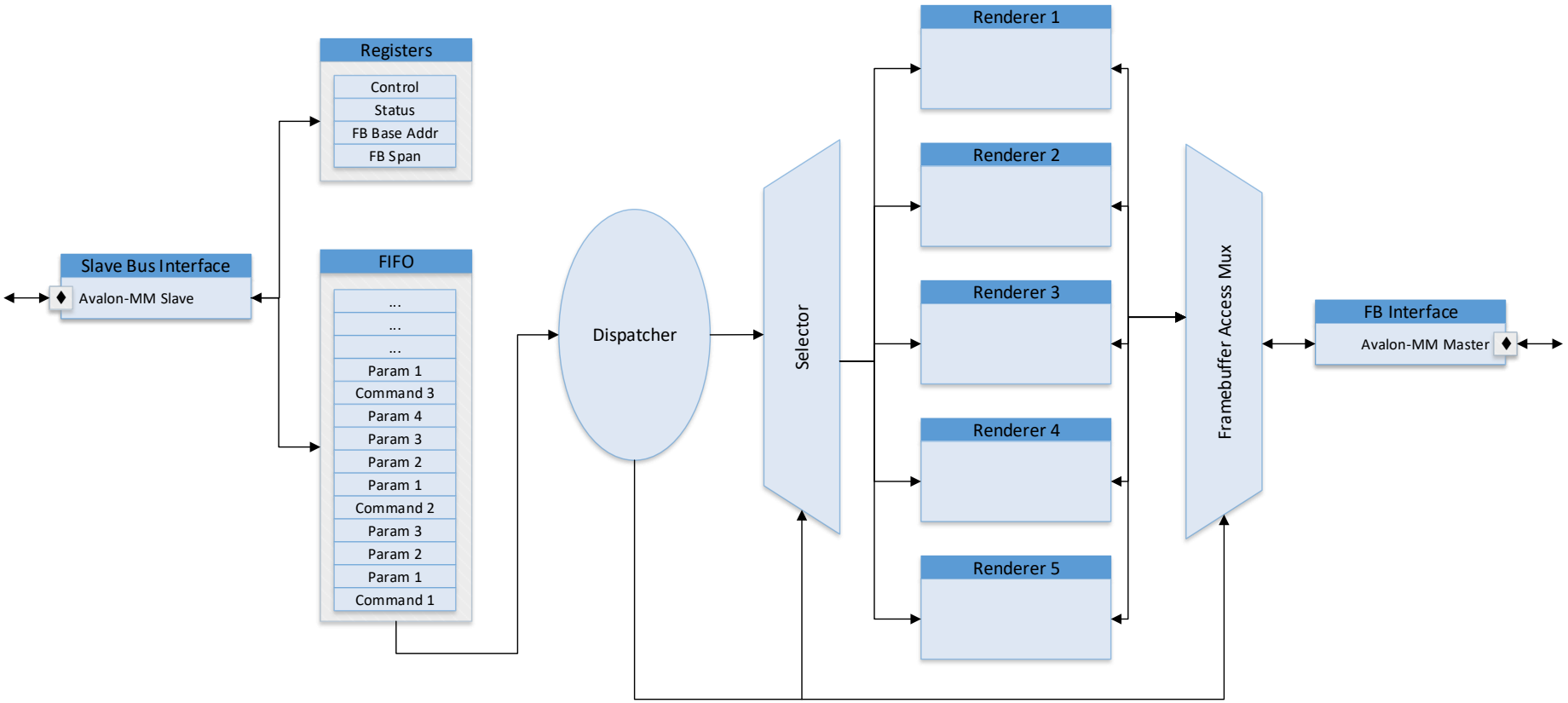



*Figure 4: Internal design overview*

## 12.4 Dispatcher

The dispatcher takes commands out of the command FIFO and dispatches them to the appropriate hardware rendering blocks. To fulfill that task, the dispatcher also handles the internal bus arbitration and parallel rendering in the future (not implemented yet).

## 12.5 Framebuffer interface

The framebuffer interface (FBI) is the sub-component which actually talks to the framebuffer. In this particular project, we're focusing on a NIOS-II system with a framebuffer that is accessible over an Avalon-MM bus. However, adapting to a different system is very easy.

Figure 5 gives a detailed overview of the architecture of the framebuffer interface. The left side is the communication towards the individual renderers and the right side is the communication towards the Avalon-MM bus. The first thing to note is that the framebuffer interface allows for bidirectional communication. This is due to the fact that some renderers require read-back from the framebuffer to fulfill their tasks. A common example is a hardware renderer for scrolling: Such a block takes a portion of the display contents and moves it inside the framebuffer. For this, the block needs to read the contents of the framebuffer first. Furthermore, read-back is also required for anti-aliasing and loading miscellaneous contents from other parts of the memory (off-screen memory) which is often used to store decoded images or complex pre-rendered shapes that just need to be copied to the on-screen portion of the framebuffer.

The communication towards the renderers is being buffered through two individual FIFOs: A *write-FIFO* and a *read-FIFO*. There are two benefits of having buffers at this place: Firstly, a renderer can keep rendering when the bus towards the framebuffer (the Avalon-MM bus) is currently being locked by another bus member. Secondly, having a write queue the framebuffer interface logic (the state machine) can optimize bus transfers by making use of burst writes (currently not implemented).

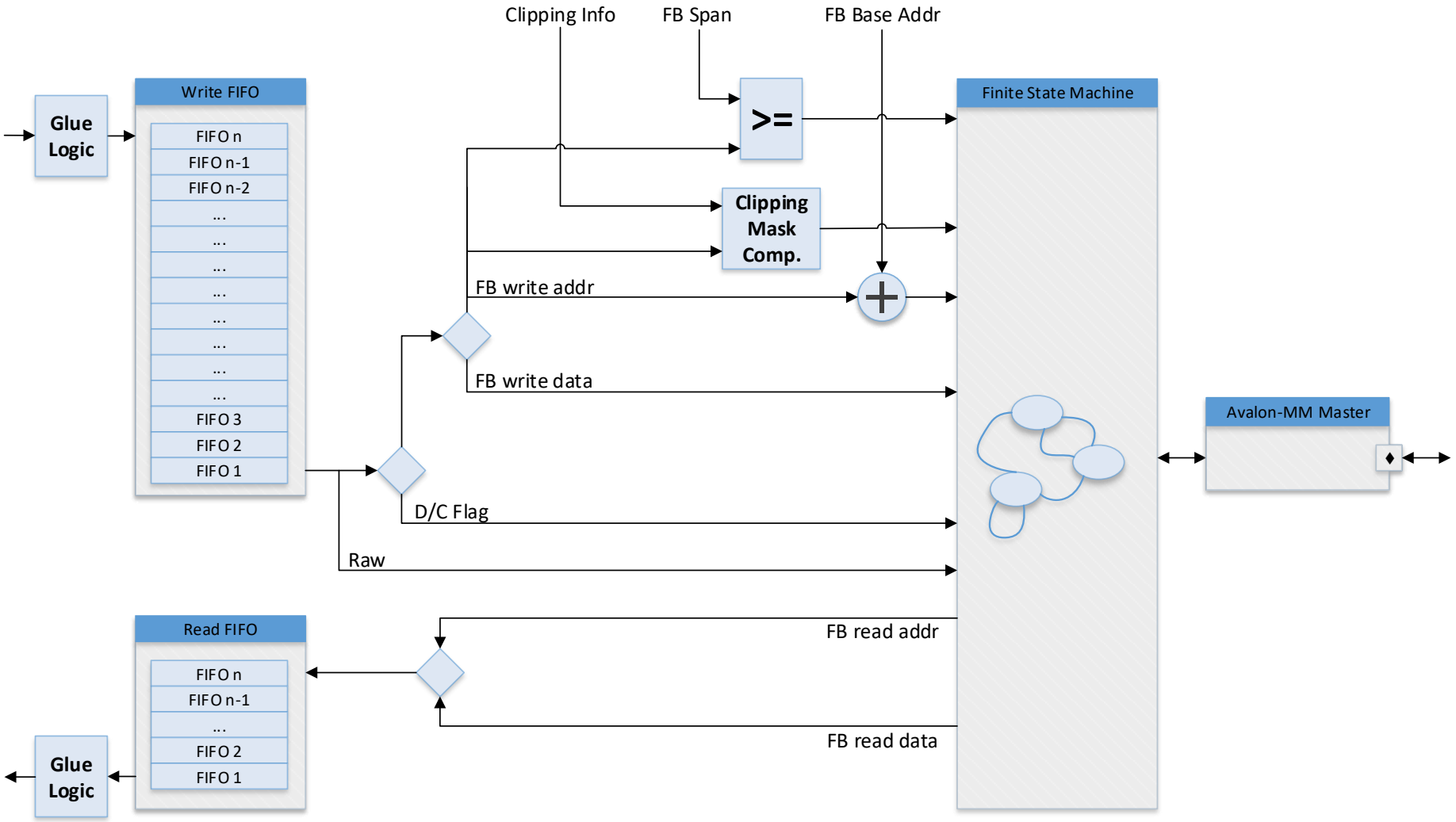



*Figure 5: Framebuffer interface design overview*

### 12.5.1 Write

At the very basic requirements, a renderer would simply provide the framebuffer interface with an address inside of the framebuffer area and a matching value to write there. The framebuffer interface would then take care of issuing the correct write transaction on the Avalon-MM bus. There are two minor draw-backs with this:

1. Each renderer would have to know the exact framebuffer location (base address)
2. A renderer could issue invalid write transactions (outside of the framebuffer location) which results in memory corruption and most likely a crash of the entire system.

Handling the former is done by using relative addresses inside of the renderers. A renderer always addresses the framebuffer starting at 0x00 which represents the first pixel. The framebuffer interface simply adds the framebuffer base address (which is being provided by the CPU via the corresponding configuration register) to the addresses calculated by the renderers.
The second draw-back is handled by simply comparing the framebuffer addresses provided by the renderers to the framebuffer span which is also provided by the CPU via the corresponding configuration register. As the renderers provide all addresses relative to 0x00 this is simply a matter of comparing the raw relative address to the span itself prior to adding the base address. Write requests outside of the framebuffer address space are simply being ignored by the framebuffer interface.

At this point it is worth to note that using the framebuffer span to prevent invalid write transactions isn't a mechanism meant to prevent bugs in the renderers themselves (as they should be verified prior to using them in a critical application) but rather because writing outside of the framebuffer area can actual be requested by the user. There are two possible scenarios which lead to a hardware renderer having to render outside of the framebuffer area:

- The user accidentally provided invalid coordinates to the hardware rendering. This can be as simple as a typo in a constant while developing a new GUI or just a complex run-time calculation that went wrong.
- The user might want to just render a portion of a shape on the screen.

For the latter, a good example is a hardware renderer that allows rendering polygons. The user might want to just render a portion of that polygon somewhere in a corner of the screen as shown in Figure 6. In such a case, the user simply asks the hardware renderer to render the polygon with negative X and Y coordinates. The renderer itself doesn't implement any clipping (this will be discussed shortly) so the renderer simply renders the entire polygon and the framebuffer interface will throw out any write transactions outside of the framebuffer area.


*Figure 6: Partial on-screen rendering of a polygon*

### 12.5.2 Read

*Disclaimer: Due to time restrictions, the read portion of the framebuffer interface could never be fully tested. While the design described in this section is final/complete the actual implementation will still need some work.*

As mentioned in the introduction of this chapter there are certain cases where a renderer needs to be able to read-back the current contents of the framebuffer to perform its job. Therefore, a renderer needs a way to ask the framebuffer interface to issue a read transaction on the bus towards the framebuffer memory. In the simplest form, the renderer would simply assert a *read request* signal and wait for the data to be returned by the framebuffer interface. However, that would lead to synchronization issues. When a hardware renderer requires the contents of a portion of the framebuffer that is usually in order to modify that same portion (eg. By performing a scroll operation or by overlying a filter). This means that the renderer needs the most recent & up-to-date version of the framebuffer contents in respect of the rendering commands already issued by the user. The *write FIFO* of the framebuffer interface might still contain data that is relevant for the read-back operation that hasn't been written to the actual framebuffer just yet. To prevent these sorts of problem, the *read requests* of a renderer are also being passed through the same FIFO as the *write request*. However, this means that the state machine that dispatches from the *write FIFO* needs to be able to distinguish between data-to-write and commands. This is achieved by adding an additional bit to the FIFO element width. This bit is from here on referred to as *the D/C bit* (Data/Command). The bit set to *'1'* means that everything after that bit represents a command that needs to be interpreted by the framebuffer interface while the bit set to *'0'* means that everything after that bit is the relative framebuffer address and the data to be written at that address.

At the moment, the only commands that the framebuffer interface can interpret is a read request and modifying the clipping area (explainer later on). In fact, there are two different read commands that can be issued: One is for linear reads and the other one is for window wrapper reads.

Data that was read back from the framebuffer by the framebuffer interface are stocked into the read FIFO from which the hardware renderers can dispatch. There are two benefits from having this FIFO:

- If a renderer would have to do some processing that prevents it from reading the data that was read back from the FBI immediately this would mean that the renderer would lock up the Avalon bus (the FBI would have to assert the *waitrequest* signal which would force the memory controller to hold on which would in turn potentially halt the entire system (other components such as the CPU couldn't access the memory either during that time). This FIFO allows the FBI to check whether there's enough room to stock the data that it will request from the memory controller prior to actually issuing the read command.
- The FBI can make burst reads as multiple values can be stocked in the FIFO.

Note that the *read FIFO* is usually many times smaller than the *write FIFO*. This is due to the fact that a renderer that requested a pixel-readback will ***usually*** have to wait for that information before it can continue and therefore immediately dispatches the data from the FBI read FIFO.

#### *12.5.2.1 Linear read*

With a linear read, the renderer simply provides the relative address of the first pixel value he wants to read back and a count in number of pixels. The framebuffer interface will not provide any wrapping but simply increment the address until the specified amount of pixels values have been read back.

A linear read request consists of the following values being written to the FIFO in that order:

1. Read command (D/C bit set to *'1'* and proper command constant (0x02))
2. Start address of the first pixel that will be read back
3. Number of reads to be performed (in pixels)

This is explained in more detail in section 12.5.5.2.

#### *12.5.2.2 Rectangular read*

In contrast to the linear read described above, the rectangular read allows a hardware renderer to specify a rectangular window inside the framebuffer. The framebuffer interface will take care of calculating the corresponding addresses and wrap at the edges of the rectangle. Basically, this allows a renderer to get a copy of the contents of a rectangular section of the framebuffer.

A rectangular read request consists of the following values being written to the FIFO in that order:

1. Read command (D/C bit set to *'1'* and proper command constant (0x03))
2. Framebuffer address of the first pixel that will be read back
3. Width of the window (in pixels)
4. Height of the window (in pixels)

This is explained in more detail in section 12.5.5.3.

In both reading modes, the framebuffer interface uses the framebuffer base address to transform the relative addresses to absolute ones and uses the framebuffer span information to prevent reading from memory that isn't part of the framebuffer. Note that the latter is a limitation that prevents off-screen area blitting – a scenario where for example the CPU decodes an image in RAM (but not the framebuffer) and then simply asks the hardware blitting engine (renderer) to copy the decoded image to the appropriate location(s) inside of the framebuffer.

### 12.5.3 Clipping

Clipping[10] is a technique that allows to limit the area (in the framebuffer) that will be affected by a rendering operation. At its most basic form (and with respect to operating only in two dimensions) clipping consists of a rectangular area with a given size and at a given position within the framebuffer. When rendering something only pixels inside that rectangular area get updated. All the pixels outside of the clipping area are unaffected by any rendering operations. More complex clipping engines allow defining clipping areas of arbitrary shapes. For example, this allows rendering only a circular area of a regular rectangular image.

Currently the clipping takes place in the framebuffer interface and only one clipping region is supported. Changing the clipping area happens by changing the clipping parameters in the corresponding registers through the CPU. The values of the clipping registers are being passed through

[10] https://en.wikipedia.org/wiki/Clipping_(computer_graphics)

the *write FIFO* as commands by the internal dispatcher. The reason for this is the same as for why *read requests* get passed through the *write FIFO*: The user might be changing the values of the clipping registers while some of the rendering operations are still in the *write FIFO* which would lead to unexpected results on the display. Passing everything through the same FIFO provides complete determinism as everything is being executed in chronological order.

Changing the clipping area position & size consists of the following values being written to the FIFO in that order:

1. Read command (D/C bit set to *'1'* and proper command constant (0x01))
2. X coordinate of the window (in pixels)
3. Y coordinate of the window (in pixels)
4. Width of the window (in pixels)
5. Height of the window (in pixels)

This is explained in more detail in section 12.5.5.4.

### 12.5.4 Read and write FIFO

The various hardware renderers communicate with the framebuffer interface through the write- and the read-FIFOs. The formats of the data inside the FIFOs are complex as they combine multiple information into one FIFO element. For example, the write-FIFO contains the D/C bit and depending on that D/C bit the rest of the data is either a command value or the framebuffer address and data combined (concatenated). To simplify the life of a hardware renderer developer both FIFOs provide glue logic towards the hardware renderers to split-up and combine these signals to individual, more intuitive signals.

Figure 7 shows the write-FIFO with the corresponding glue logic in front of it:

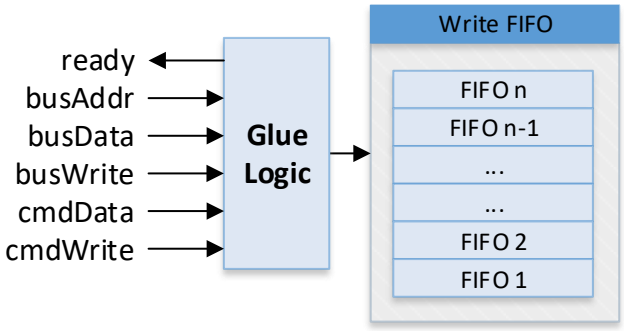



*Figure 7: FBI write-FIFO glue logic*

The *ready* signal indicates where the framebuffer interface is ready to receive new data or commands. This is the inverted *isFull* signal of the write-FIFO. A renderer **must not** issue new write transactions to the write-FIFO by strobing *busWrite* or *cmdWrite* if *ready* is logic '0'. Behavior is undefined in such a case.

A renderer can alter the framebuffer contents by assigning the relative framebuffer address of the pixel in question to *busAddr* and by setting *busData* to the corresponding pixel value (color value). A one clock cycle long strobe on *busWrite* will store that information as a *"data package"* in the write-FIFO in a format that the framebuffer interface behind the FIFO understands.

A renderer can issue commands to the framebuffer interface (such as issuing a read request or changing the clipping mask) by setting *cmdData* to the corresponding command and by applying a write strobe on *cmdWrite*.

Similarly, the read FIFO is also interfaced by glue logic to simplify the life of a renderer developer:

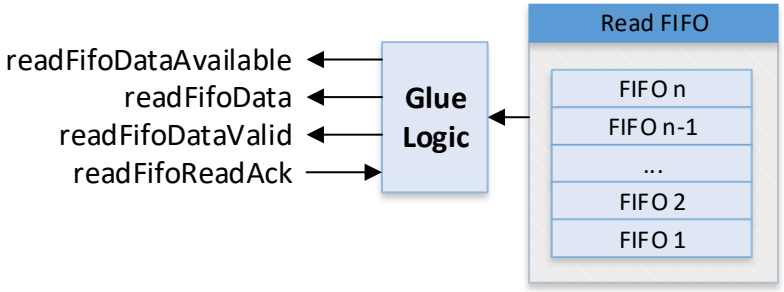



*Figure 8: FBI read-FIFO glue logic*

The *readFifoDataAvailable* signal indicates that there is new data in the read-FIFO ready to be dispatched by the renderer. This is the inverted *isFull* signal of the read-FIFO.

The *readFifoData* is the actual data on the output port of the read-FIFO.

It can happen that the framebuffer interface can't complete a requested read operation for example because the renderer requested reading from an invalid memory section. The *readFifoDataValid* signal indicates, whether the data is actually to be interpreted by the renderer as in such a case the framebuffer interface will just provide dummy data to complete the read request.

### 12.5.5 FIFO data format

The previous sections explained the historical development of the framebuffer interface and its capabilities. The following is a summary of the data that needs to be written into the *write-FIFO* of the framebuffer interface to perform the corresponding tasks.

The commands that need to be written into the *write-FIFO* of the framebuffer interface by a renderer are documented in tabular form:

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 0 | constant | - |
| **2** | 0 | < variable > | Pixel count |
| **3** | 1 | constant | - |
| **4** | 1 | < variable > | Pixel value |
| **5** | Don't care | constant | - |
| **6** | Don't care | < variable > | Pixel coordinate |

*Table 3: Framebuffer interface write-FIFO data format synopsis*

Parameters are being listed in ascending order from top to bottom. This means that the top most parameter is the first one to be written to the FIFO. The *D/C (MSB)* column specifies the state of the *D/C bit*. The *MSB-1 downto 0* column the rest of the bits of the FIFO width. The values of the columns *D/C (MSB)* and the *MSB-1 downto 0* will be concatenated (in that order, left-to-right) and written to the FIFO.

Note that the number of bits for the data and the address are configured through the corresponding generics named *dataBitNb* and *addrBitNb*. From this, the *write-FIFO* size has been deduced as:

$$writeFifoBitNb = dataBitNb + addrBitNb + 1$$

The *+1* is to take the *D/C* bit discussed earlier into account.

All framebuffer addresses are relative addresses where 0x00 is the first pixel in the framebuffer as the framebuffer interface will add the base address / offset of the framebuffer itself.

#### *12.5.5.1 Write*

Issuing a write transaction to the framebuffer is pretty straightforward and simple. As this will be by far the most used operation in the framebuffer interface it is crucial that it takes as little time as possible.

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 0 | <data> & <address> | - |

*Table 4: Framebuffer interface write-FIFO data format for write transaction*

**Example:** Writing the color value *0xAAAAAA* to the address *0xBBBBBBBB* assuming that both *dataBitNb* and *addrBitNb* are 32 requires writing the following into the *write-FIFO*:

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 0 | 0x00AA AAAA BBBB BBBB | - |

**Combined:** 0x0 00AA AAAA BBBB BBBB

*Table 5: Framebuffer interface write-FIFO data format for write transaction example*

#### *12.5.5.2 Linear read*

Performing a linear read requires two parameters: The start address (address of the first pixel in the framebuffer) and the number of pixels to be read back:

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 1 | 0x02 | - |
| **2** | Don't care | < Start address > | Pixel address |
| **3** | Don't care | < Number of pixels > | Pixel count |

*Table 6: Framebuffer interface write-FIFO data format for linear read*

**Example:** Reading back 13 pixels starting at the relative framebuffer address 0x3E7 assuming that both *dataBitNb* and *addrBitNb* are 32 requires writing the following into the *write-FIFO*:

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 1 | 0x0000 0000 0000 0002 | - |
| **2** | Don't care | 0x0000 0000 0000 03E7 | Pixel address |
| **3** | Don't care | 0x0000 0000 0000 000D | Pixel count |

*Table 7: Framebuffer interface write-FIFO data format for linear read example*

#### *12.5.5.3 Rectangular read*

Performing a rectangular read requires three parameters: The start address (address of the first pixel in the framebuffer) and the width and height of the rectangular window:

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 1 | 0x03 | - |
| **2** | Don't care | < Start address > | Pixel address |
| **3** | Don't care | < Rectangle width > | Pixel count |
| **4** | Don't care | < Rectangle height > | Pixel count |

*Table 8: Framebuffer interface write-FIFO data format for rectangular read*

**Example:** Reading back the pixels of a rectangle measuring 82 pixels in width and 64 pixels in height starting at the first pixel at address 0x3E7 assuming that both *dataBitNb* and *addrBitNb* are 32 requires writing the following into the *write-FIFO*:

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 1 | 0x0000 0000 0000 0003 | - |
| **2** | Don't care | 0x0000 0000 0000 03E7 | Pixel address |
| **3** | Don't care | 0x0000 0000 0000 0052 | Pixel count |
| **4** | Don't care | 0x0000 0000 0000 0040 | Pixel count |

*Table 9: Framebuffer interface write-FIFO data format for rectangular read example*

#### *12.5.5.4 Modifying clipping area*

Changing the clipping area requires writing the corresponding command followed by the four parameters which are X and Y coordinates and width and height dimensions of the new rectangular clipping area. Note that as described above the size of the FIFO is determined by the address and data bus width. However, the clipping parameters will be resized to *coordBitNb*.

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 1 | 0x01 | - |
| **2** | Don't care | < Clipping area X coordinate > | Pixel coordinate |
| **3** | Don't care | < Clipping area y coordinate > | Pixel coordinate |
| **4** | Don't care | < Clipping area width > | Pixel count |
| **5** | Don't care | < Clipping area height > | Pixel count |

*Table 10: Framebuffer interface write-FIFO data format for clipping area modification*

**Example:** Changing the clipping area to *X = 5*, *Y = 15*, *Width = 680*, *Height = 480* assuming that both *dataBitNb* and *addrBitNb* are 32 requires writing the following into the *write-FIFO:*

| Order | D/C (MSB) | MSB-1 downto 0 | Unit |
|---|---|---|---|
| **1** | 1 | 0x0000 0000 0000 0001 | - |
| **2** | Don't care | 0x0000 0000 0000 0005 | Pixel coordinate |
| **3** | Don't care | 0x0000 0000 0000 000F | Pixel coordinate |
| **4** | Don't care | 0x0000 0000 0000 02A8 | Pixel count |
| **5** | Don't care | 0x0000 0000 0000 01E0 | Pixel count |

*Table 11: Framebuffer interface write-FIFO data format for clipping area modification*

## 12.6 Renderers

The GPU contains an isolated hardware rendering for each rendering operation that the GPU offers.

Note: In the future, it will also be possible to have multiple instances of the same hardware rendering inside the GPU which allows to handle multiple rendering operations of the same type simultaneously. For example: Having more than one block to hardware render filled rectangles means that multiple filled rectangles can be rendered in parallel.

This section of the document will explain the basic theory of operation of the existing renderers. To properly understand the different terms used it might be required to read the datasheet first.

### 12.6.1 Pixel

The pixel renderer consists of a simple finite state machine (FSM) that dispatches the required parameters from the command FIFO and then calculates the relative framebuffer offset to issue a write request to the framebuffer interface.

The relative framebuffer address is calculated by using the following equation:

$$FB\ addr = (x + fbWidth * y) * fbBytesPerPixel$$

The FSM is shown in Figure 9:

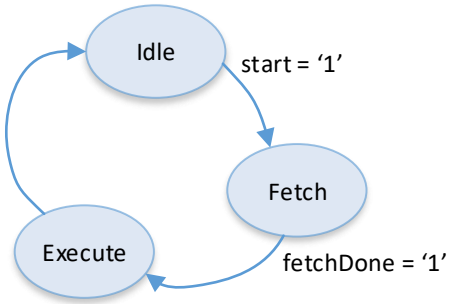



*Figure 9: Pixel renderer FSM*

Fetching is done by a simple counter that is used to grab all three parameters (X, Y, color) from the command FIFO.

### 12.6.2 Filled rectangle

The filled rectangle renderer works the same way as the pixel renderer except that it contains two more counters which are used to count in X and Y direction until the width and height of the rectangle have been reached. The pixel address calculation reflects this by extending the equation of the pixel renderer with the two counter values *countX* and *countY*:

$$FB\ addr = \big(x + countX + fbWidth * (countY + y)\big) * fbBytesPerPixel$$

This calculation is performed for each pixel in the rectangle.

### 12.6.3 Clipping

The clipping renderer functions a bit different to the pixel and the filled rectangle renderers as it doesn't alter the framebuffer contents but issues a command to the framebuffer interface instead. The only job this clipping renderer does it dispatching the clipping mask parameters (X, Y, width and height) from the command FIFO and feeding them to the write-FIFO of the framebuffer interface. The finite state machine is simpler in this case as the fetching and the execution can be merged into the same state. The clipping renderer dispatches the first parameter and immediately pushes it to the framebuffer interface write-FIFO and therefore only adds one clock cycle of latency.

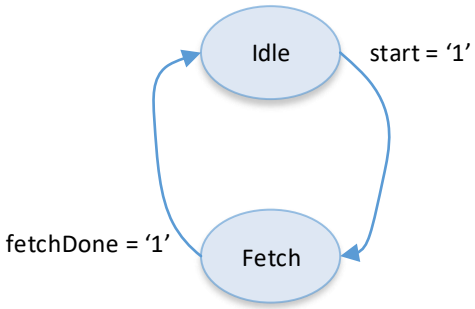



*Figure 10: Clipping renderer FSM*

After discussing the theory of operation of the different hardware renderers it is time to look at the actual implementation.

# 13 Implementation

Section 12 explains the internal structure of the IP-core. This section of the document looks at the actual implementation in code. As mentioned in section 12, the overall design shown there doesn't necessarily correspond to the design implementation. Entities and files have been named differently and sometimes parts that are shown individually in the design overview are simply embedded somewhere else. While this might sound like bad practice it's to keep everything clean and simple. The overview given in section 12 has been structured to show the flow & handling of data in a way that is easy to grasp and understand. However, in code it's sometimes often simpler to organize the structure slightly differently.

This section won't go through every line of code but instead just list the important bits and pieces that are necessary to understand the overall theory of operation. Mainly this consists of explaining the different files & entities. The entity tables will not list the data/signal types and ranges but instead just give a description detailed enough to understand the purpose of the generic/signal. Further details can be gathered easily from reading the code.

Note that the generics and ports of all hardware renderers are exactly the same. They are not listed in this section as section 14 will describe everything required to write a new hardware renderer and therefore will also list the entity in great detail.

## 13.1 Silizium.vhd

The s*ilizium.vhd* file contains the top-level entity of the Silizium IP-core. It doesn't do anything except for forwarding everything to an entity named *silizium_implementation*. This technique is there to prevent having to re-generate the entire code of the SoC in the QSys tool every time something changes in the actual implementation: This way QSys knows the interface of the Silizium IP-Core which is the only important thing for it. All the other code we write will just be copy-pasted by the QSys tool upon code generation and therefore doesn't requiring re-generating the entire SoC design which can take several minutes.

Generics:

| Name | Description |
|---|---|
| dataBitNb | Internal data handling width. Will be used in the future. **DO NOT CHANGE THIS**. |
| coordBitNb | The number of bits required to represent a coordinate in two's complement. |
| colorBitNb | The number of bits required to represent a color value (internally). |
| cmdFifoNumWordsExp | The exponent of the size of the input command FIFO in elements. Actual size will be 2^n of this value. |
| fbAddrBitNb | The width of the address bus of the framebuffer interface. |
| fbDataBitNb | The width of the data bus of the framebuffer interface. |
| fbBurstcountBitNb | The number of bits required for the burstcount signal. |
| fbBytesPerPixel | The number of bytes per pixel. |
| fbWidth | The width of the framebuffer in pixels. |
| fbHeight | The height of the framebuffer in pixels. |
| fbWriteFifoNumWordsExp | The exponent of the size of the framebuffer write FIFO in elements. Actual size will be 2^n of this value. |
| fbReadFifoNumWordsExp | The exponent of the size of the framebuffer read FIFO in elements. Actual size will be 2^n of this value. |

*Table 12: silizium.vhd generics*

Ports:

| Name | Direction | Description |
|---|---|---|
| reset | in | Global clock input |
| clock | in | Global reset input |
| avalon_slave_address | in | Avalon-MM slave interface |
| avalon_slave_read | in | |
| avalon_slave_readdata | out | |
| avalon_slave_write | in | |
| avalon_slave_writedata | in | |
| avalon_slave_waitrequest | out | |
| avalon_master_address | out | Avalon-MM master interface |
| avalon_master_write | out | |
| avalon_master_writedata | out | |
| avalon_master_waitrequest | in | |
| avalon_master_read | out | |

| | | |
|---|---|---|
| avalon_master_readdata | in | |
| avalon_master_readdatavalid | in | |
| avalon_master_burstcount | out | |

*Table 13: silizium.vhd ports*

## 13.2 Silizium_implementation.vhd

This file contains the actual top-level implementation of the Silizium IP-Core. The entity is exactly he same as the one listed for the *silizium.vhd* file as all generics and ports are just being forwarded.

This file contains the implementation of the slave interface which means that it hosts all registers and also creates the instance of the command FIFO. Furthermore, the instances of the dispatcher and the framebuffer interface are also being created here.

Generics:

| Name | Description |
|---|---|
| dataBitNb | Internal data handling width. Will be used in the future. **DO NOT CHANGE THIS**. |
| coordBitNb | The number of bits required to represent a coordinate in two's complement. |
| colorBitNb | The number of bits required to represent a color value (internally). |
| cmdFifoNumWordsExp | The exponent of the size of the input command FIFO in elements. Actual size will be 2^n of this value. |
| fbAddrBitNb | The width of the address bus of the framebuffer interface. |
| fbDataBitNb | The width of the data bus of the framebuffer interface. |
| fbBurstcountBitNb | The number of bits required for the burstcount signal. |
| fbBytesPerPixel | The number of bytes per pixel. |
| fbWidth | The width of the framebuffer in pixels. |
| fbHeight | The height of the framebuffer in pixels. |
| fbWriteFifoNumWordsExp | The exponent of the size of the framebuffer write FIFO in elements. Actual size will be 2^n of this value. |
| fbReadFifoNumWordsExp | The exponent of the size of the framebuffer read FIFO in elements. Actual size will be 2^n of this value. |

*Table 14: silizium.vhd generics*

Ports:

| Name | Direction | Description |
|---|---|---|
| reset | in | Global clock input |
| clock | in | Global reset input |
| avalon_slave_address | in | Avalon-MM slave interface |
| avalon_slave_read | in | |
| avalon_slave_readdata | out | |
| avalon_slave_write | in | |
| avalon_slave_writedata | in | |
| avalon_slave_waitrequest | out | |
| avalon_master_address | out | Avalon-MM master interface |
| avalon_master_write | out | |
| avalon_master_writedata | out | |
| avalon_master_waitrequest | in | |
| avalon_master_read | out | |
| avalon_master_readdata | in | |
| avalon_master_readdatavalid | in | |
| avalon_master_burstcount | out | |

*Table 15: silizium.vhd ports*

## 13.3 Silizium_dispatcher.vhd

The dispatcher receives all signals of the command FIFO required for reading from it and takes care of selecting the proper hardware renderer to execute the job. Each hardware renderer will dispatch his parameters itself from the command FIFO as the dispatcher doesn't know how many parameters each hardware renderer needs.

Generics:

| Name | Description |
|---|---|
| dataBitNb | Internal data handling width. Will be used in the future. **DO NOT CHANGE THIS**. |
| coordBitNb | The number of bits required to represent a coordinate in two's complement. |
| colorBitNb | The number of bits required to represent a color value (internally). |
| fbAddrBitNb | The width of the address bus of the framebuffer interface. |
| fbDataBitNb | The width of the data bus of the framebuffer interface. |
| fbBytesPerPixel | The number of bytes per pixel. |
| fbWidth | The width of the framebuffer in pixels. |
| fbHeight | The height of the framebuffer in pixels. |

*Table 16: silizium_dispatcher.vhd generics*

Ports:

| Name | Direction | Description |
|---|---|---|
| reset | in | Global clock input |
| clock | in | Global reset input |

| | | |
|---|---|---|
| fifoEmpty | in | *isEmpty* signal of the command FIFO |
| fifoRdAck | out | Read acknowledge signal of the command FIFO |
| fifoData | in | Data output port of the command FIFO |
| enable | in | Dispatcher enable input |
| busy | out | Dispatcher *isBusy* output |
| fbiBusAddr | out | FBI *write-pixel* address bus |
| fbiBusData | out | FBI *write-pixel* data bus |
| fbiBusWrite | out | FBI *write-pixel* write strobe |
| fbiCmdData | out | FBI *command* data bus |
| fbiCmdWrite | out | FBI *command* write strobe |
| fbiIsReady | in | FBI *isReady* signal |
| fbiReadFifoDataAvailable | out | FBI read-FIFO *newDataAvailable* signal |
| fbiReadFifoData | out | FBI read-FIFO data output port |
| fbiReadFifoDataValid | out | Whether the *fbiReadFifoData* data is valid |
| fbiReadFifoReadAck | in | Read acknowledge signal of the FBI read FIFO |

*Table 17: silizium_dispatcher.vhd ports*

## 13.4 Silizium_framebufferinterface.vhd

The framebuffer interface implemented in this file dispatches maintains the write-FIFO and read-FIFO and talks to the actual framebuffer memory over the corresponding Avalon-MM master interface.

Generics:

| Name | Description |
|---|---|
| coordBitNb | The number of bits required to represent a coordinate in two's complement. |
| addrBitNb | The width of the address bus of the framebuffer interface. |
| dataBitNb | The width of the data bus of the framebuffer interface. |
| burstcountBitNb | The number of bits required for the burstcount signal. |
| bytesPerPixel | The number of bytes per pixel. |
| writeFifoNumWordsExp | The exponent of the size of the framebuffer write FIFO in elements. Actual size will be 2^n of this value. |
| readFifoNumWordsExp | The exponent of the size of the framebuffer read FIFO in elements. Actual size will be 2^n of this value. |

*Table 18: silizium_framebufferinterface.vhd generics*

Ports:

| Name | Direction | Description |
|---|---|---|
| reset | in | Global clock input |
| clock | in | Global reset input |
| enable | in | Enable/disable the framebuffer interface activities |
| boundaryChecksEnable | in | Enable/disable the memory boundary checks |
| clippingEnable | in | Enable/disable the rectangular clipping mask |
| avalon_master_address | buffer | Avalon-MM master interface |
| avalon_master_write | out | |
| avalon_master_writedata | out | |
| avalon_master_waitrequest | in | |
| avalon_master_read | out | |
| avalon_master_readdata | in | |
| avalon_master_readdatavalid | in | |
| avalon_master_burstcount | out | |
| fbAddrBase | in | Framebuffer memory base address |
| fbAddrSpan | in | Framebuffer memory section span in bytes |
| busAddr | in | *write-pixel* address bus |
| busData | in | *write-pixel* data bus |
| busWrite | in | *write-pixel* write strobe |
| cmdData | in | *command* data bus |
| cmdWrite | in | *command* write strobe |
| ready | out | *isReady* signal (inverted *writeFifoIsFull* signal) |
| readFifoDataAvailable | out | Whether new data is available to be read |
| readFifoData | out | Read FIFO data output port |
| readFifoDataValid | out | Whether the data on *readFifoData* is valid (valid read) |
| readFifoReadAck | in | Read acknowledge signal of the read FIFO |
| writeFifoUsedWords | out | Number of used words of the write FIFO |
| writeFifoClear | in | Signal to clear the write FIFO |
| readFifoUsedWords | out | Number of used words of the read FIFO |
| readFifoClear | in | Signal to clear the read FIFO |
| clipX | out | The X coordinate of the current clipping mask |
| clipY | out | The Y coordinate of the current clipping mask |
| clipWidth | out | The width of the current clipping mask |
| clipHeight | out | The height of the current clipping mask |

*Table 19: silizium_framebufferinterface.vhd generics*

The internal design of the Silizium-IP core has been designed to allow for easy adding of new hardware renderers. The following section provides all the information necessary to implement new hardware renderers.

# 14 Adding new renderers

This section explains everything required to implement a new hardware renderer into the existing infrastructure of the Silizium IP-Core.

A hardware renderer must have an entity that matches the ports listed in Table 21. Table 20 lists all generics that are available to a hardware renderer.

## 14.1 Generics

Table 20 lists the generics with type information, default values and the corresponding descriptions:

| Name | Type | Default | Description |
|---|---|---|---|
| dataBitNb | positive | 32 | The width of the command FIFO. |
| coordBitNb | positive | 32 | The number of bits required to represent a coordinate in two's complement. |
| colorBitNb | positive | 32 | The number of bits required to represent a color value. |
| cmdFifoNumWordsExp | positive | 7 | The exponent of the size of the input command FIFO in elements. Actual size will be 2^n of this value. |
| fbAddrBitNb | positive | 32 | The width of the address bus of the framebuffer interface. |
| fbDataBitNb | positive | 32 | The width of the data bus of the framebuffer interface. |
| fbBytesPerPixel | positive | 4 | The number of bytes per pixel. |
| fbWidth | positive | 800 | The width of the framebuffer in pixels. |
| fbHeight | positive | 480 | The height of the framebuffer in pixels. |

*Table 20: Silizium renderers interface generics*

## 14.2 Ports

A hardware renderer must implement the port interface shown in Table 21:

| Name | Direction | Type | Range |
|---|---|---|---|
| reset | in | std_logic | - |
| clock | in | std_logic | - |
| start | in | std_logic | - |
| dataInReady | in | std_logic | - |
| dataIn | in | std_logic_vector | dataBitNb-1 downto 0 |
| readAck | out | std_logic | - |
| busy | out | std_logic | - |
| fbiBusAddr | out | unsigned | fbAddrBitNb-1 downto 0 |
| fbiBusData | out | std_logic_vector | fbDataBitNb-1 downto 0 |
| fbiBusWrite | out | std_logic | - |
| fbiCmdData | out | std_logic_vector | fbAddrBitNb+fbDataBitNb-1 downto 0 |
| fbiCmdWrite | out | std_logic | - |
| fbiIsReady | in | std_logic | - |

*Table 21: Silizium renderers interface ports*

Detailed explanation of each signal:

***reset:*** Global reset input.

***clock:*** Global clock input.

***start:*** This signal will be set to '1' for one clock cycle by the dispatcher once the renderer is supposed to start the rendering job. The current data element of the command FIFO is guaranteed to be the first parameter for the renderer.

***dataInReady:*** This signal is set to '1' if there is data to be dispatched in the command FIFO. The renderer must not attempt to read from the command FIFO if this signal is not '1'. This signal is the inverted *isEmpty* signal of the command FIFO.

***readAck:*** The renderer must assert this signal (setting it to '1' for the duration of one clock cycle) once a data element has been read from the FIFO.

***busy:*** This active-high signal allows the dispatcher to check whether the renderer is still busy or not. The renderer must set this signal to '1' one clock cycle after the *start* strobe has been issued by the dispatcher. If this signal is not being driven '1' by the renderer within that time frame the dispatcher assumes that the renderer finished and will start dispatching the next command from the command FIFO which will result in framebuffer corruption as the order of commands in the command FIFO has been mixed up (because the renderer didn't get a change to dispatch the parameters from the FIFO).

***fbiBusAddr:*** The renderer puts the framebuffer address of a pixel it wants to change on this signal.

***fbiDataAddr:*** The renderer puts the pixel value (color value) of a pixel it wants to change on this signal.

***fbiBusWrite:*** Asserting this signal causes the glue logic of the framebuffer interface to take the address and data provided through *fbiBusAddr* and *fbiDataAddr* and putting them into the write-FIFO of the framebuffer interface. Therefore, asserting this signal will result in a change of the framebuffer memory (provided that the address passes the boundary checks (if enabled) and is inside the clipping mask (if enabled).

***fbiCmdData:*** This signal is used to pass a command to the framebuffer interface. The available commands are documented in section 12.5.1.

***fbiCmdWrite:*** Asserting this signal causes the glue logic of the framebuffer interface to take the command provided through *fbiCmdData* and putting it into the write-FIFO of the framebuffer interface.

***fbiIsReady:*** This signal is provided by the framebuffer interface. The renderer must not assert *fbiBusWrite* or *fbiCmdWrite* if this signal is not '1'. Behavior in such a case is undefined. This signal is the inverted *isFull* signal of the framebuffer interface write-FIFO. Therefore, *fbiIsReady = '0'* indicates that the write-FIFO is full.

## 14.3 Infrastructure

Once a new renderer has been written it needs to be added into the existing Silizium infrastructure. This is done by creating an instance of the renderer in the dispatcher found in the file *silizium_dispatcher.vhd*. That file also contains the implementation of the demux and mux to select the appropriate renderer to which the new renderer needs to be hooked up. That process is

straightforward as it's just a matter of doing the same thing that has already been done for the existing renderers.

Figure 11 shows multiple renderers with the dispatcher, selector and framebuffer interface mux.

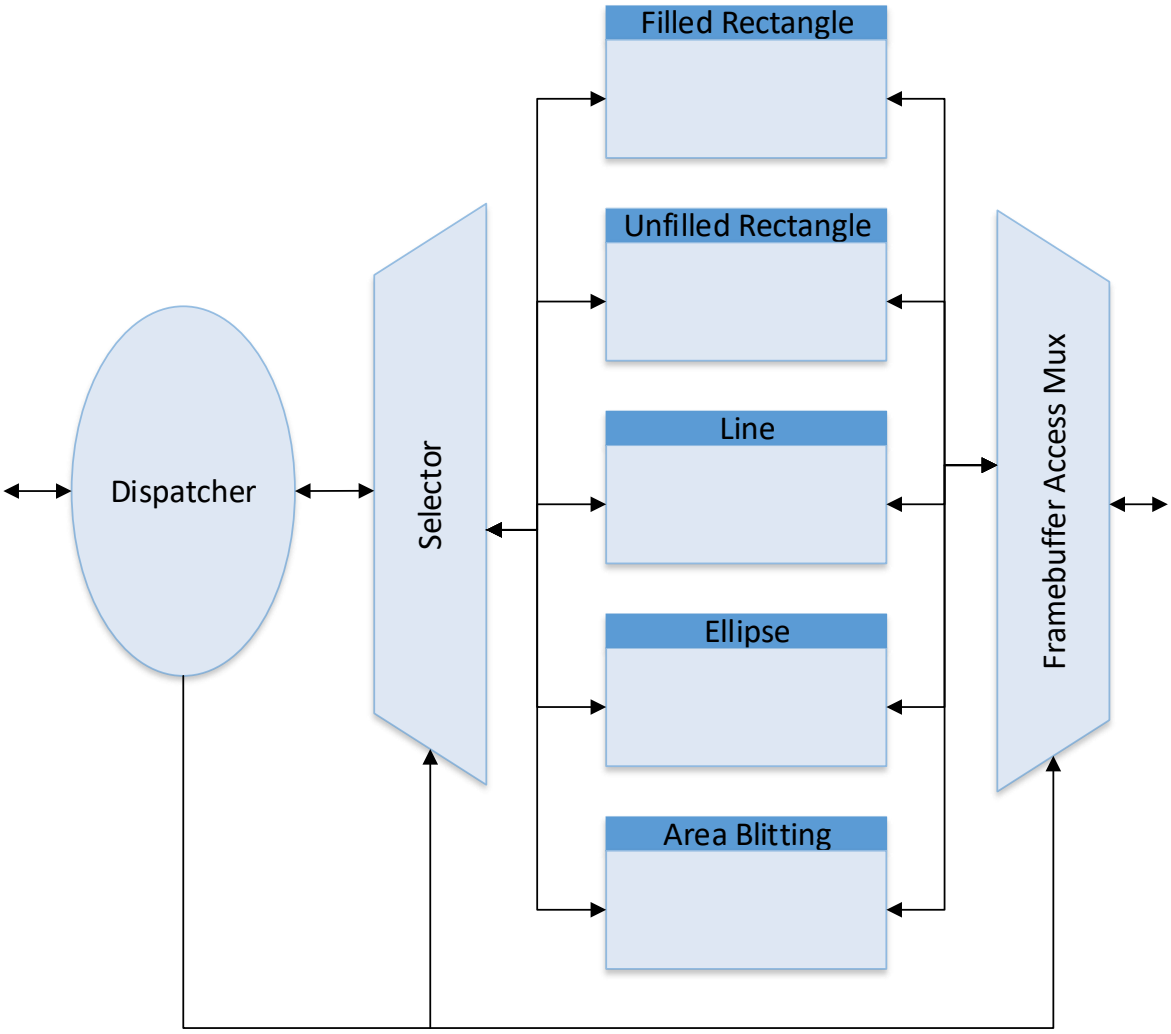



*Figure 11: Renderers with dispatcher, selector and mux*

## 15 Tests & Verification

Proper development of this IP-Core (or for any HDL IP-Core for that matter) would involve creating a test bench for every single component being designed. After losing a huge chunk of time due to a problem with the Avalon-MM master interface (described in section 18.1) I decided not to spend whole lot of time on this. As of today, there is a dedicated test bench for the framebuffer interface and one test bench for the entire system. This way the framebuffer interface can be tested individually as that is one of the most crucial/complex components. The correct functioning of the dispatcher can be easily verified in the overall test bench as a test bench dedicated for the dispatcher alone would be almost identical. The functioning of the renderers can also be tested easily through the overall test bench as the renderers dispatch the required parameters from the command FIFO themselves once they got selected by the dispatcher. The renderers are directly connected to the framebuffer interface

(via a mux) and therefore an additional test bench for the section between the renderers and the framebuffer interface is unnecessary.

At the beginning of the project it was planned to use a higher-level test & verification tool such as cocotb[11] for each individual component as well as for the entire IP-Core. This hasn't been done due to loosing quite some time with the Avalon-MM master interface as mentioned above. Setting up a tool like cocotb would have required a more time than working with bare traditional VHDL test benches, especially as I never worked with such a tool before.

Additionally, a software that constantly issues new rendering requests for filled rectangles while the CPU itself modifies the framebuffer contents every few rectangles has been written that was left running for 72 hours. The system kept running stable and no issues were encountered.

# 16 Future steps

This section of the document lists a couple of things that should be taken care of when continuing the project after finishing this bachelor thesis project.

## 16.1 More hardware renderers

Although the current design implements three hardware renderers only one is very useful in a typical application: The filled rectangle renderer. Other renderers were planned but couldn't be implemented during this project. However, the current design of the Silizium IP-core provides everything necessary to add new hardware renderers. Section 14 provides all the information required to add new renderers.

## 16.2 Test benches

A higher-level testing framework (such as cocotb) should be used to allow for a more efficient workflow. Currently only two test benches exist for the entire system. This has been enough for this thesis project but continuing the project, potentially with more developers, would call for a more intricate testing system.

## 16.3 FIFO abstractions

The current design creates instances of the *scfifo* (single clock FIFO) provided by the Intel IP-core catalogue. These instances are currently hard-configured and assigned for the MAX10 FPGA families. A FIFO abstraction layer should be implemented to allow the Silizium IP-core to be easily ported to other non-Intel FPGA platforms.

[11] https://cocotb.readthedocs.io/en/latest/introduction.html

### 16.4 Bus abstractions

The current design contains hard-coded implementations for the Avalon-MM master and slave bus interfaces. Adding abstraction layers or bridge components to allow the Silizium IP-core to be used in non-Avalon systems should be added.

### 16.5 Clipping

Currently, clipping is implemented by providing a single rectangular clipping mask in the framebuffer interface. The FBI checks whether the address provided by a renderer through the write-FIFO when issuing a write-request is inside that mask and simply ignores it if it is not. In a real-world application, this means that a renderer potentially issues hundreds of write requests to the write-FIFO of the FBI that are outside the clipping mask. Providing each renderer with the clipping mask dimensions would ease the traffic on the FBI write-FIFO.

### 16.6 Framebuffer interface burst transactions

Writing pixel to the framebuffer currently takes up four clock cycles due to the state machine in the framebuffer interface. This could be optimized down to one clock cycle per pixel if the framebuffer interface issues burst writes (and reads). The entire framebuffer interface has been designed to support this eventually (hence the write-FIFO). The reason that burst writes have not been implemented is due to the fact that a lot of time was already lost on the Avalon-MM master interface (as described in section 18) and furthermore due to the fact that an Intel engineer once mentioned in a WebEx conference that bursts can be tricky to get working with the DDR3 memory controller that the reference design used in this project uses.

## 17 Parallel Rendering

At the very start of this project it was decided that the GPU will not support parallel rendering. In this case, *"parallel rendering"* means being able to render multiple shapes at the same time. Theoretically, it would be possible to render a filled rectangle at the same time as a line as these two hardware renderers are completely stand-alone blocks. Alternatively, it would also be possible to have multiple instances of the filled rectangle renderer which would allow to render multiple filled rectangles at once. The decision not to implement parallel rendering was made due to the time restrictions of the project. However, after developing the design of the GPU it was clear that parallel rendering wouldn't be that much of an advantage in this case anyway. There are two reasons for this which are described in detail in this section:

1. Framebuffer interface bottleneck
2. Synchronization

### 17.1 Framebuffer interface bottleneck

Whenever a hardware renderer wants to change the contents of the framebuffer it eventually has to issue a write transaction to the framebuffer memory. In Silizium, this happens by the renderer issuing a write-transaction to the write-FIFO of the framebuffer interface. The Framebuffer interface dispatches from the write-FIFO and eventually writes to the framebuffer memory. The currently

implemented hardware renderers are all able to *"produce"* one pixel per clock cycle. Even if the framebuffer interface would use burst transactions (currently not implemented) it would still require one clock per pixel itself. This means that two hardware renderers working in parallel wouldn't be any faster as the framebuffer interface can't possibly write any faster to the framebuffer memory. Parallel rendering would only speed things up for hardware renderers that need more than one clock cycle per pixel.

### 17.2 Synchronization

Parallel rendering works well as long as the simultaneously rendering renderers affect different, non-interfering sections of the framebuffer. Real-world objects such as pushbuttons, sliders and other GUI elements are usually assembled from multiple different objects such as rectangles, lines, text and so on. Some of these are rendered by the GPU and some of those are rendered by the CPU (as some things are just too expensive (in terms of resources) or too complex to implement in hardware such as font rendering. Often pixels are being over written multiple times when rendering such a complex element. An example: A push-button might consist of a rectangle and a border. Those two elements can be calculated in a way that they don't overlap. But ultimately the CPU will want to render some text over that element. In this case the chronology of all rendering operations (not just those inside of the GPU) matters: If the text gets rendered before the rectangle, no text will be visible as the rectangle overwrites the text pixels. Therefore, parallel rendering would only work if non-overlapping regions of the framebuffer are being modified.

The two sections above show that implementing parallel rendering would not offer any advantage as long as the Silizium IP-core does not provide any hardware renderers that are not capable of producing (calculating) one pixel per clock.

## 18 Problems

Almost any project encounters problems at some point or another. This section doesn't list all of the problems that were encountered (as many of them were only small and had a minimal impact on the project) but instead only lists the major problems which had a noticeable impact on the overall project.

### 18.1 Framebuffer interface

The by far biggest problem occurred with the framebuffer interface. More specifically: With the Avalon-MM master interface. This problem had a big impact on the project (time-wise). A separate document (that can be found at the end of this report) has been created as the problem hasn't been properly solved yet. Instead, a workaround that is suitable to finish this thesis project has been applied. The problem appears to be a bug in the code generated by the QSys tool and therefore a proper fix needs to be provided by Intel.

## 18.2 Missing pixels

The framebuffer interface problem mentioned in section 18.1 resulted in changing the clock domain of this custom IP-Core as explained in the dedicated document. One of the draw-backs of this workaround is that the Silizium IP-core now runs on a much faster clock. The used clock is half the DDR3 clock which is 150 MHz. It sometimes happens that the hardware renderers (namely the rectangle renderers) miss out on a pixel or two. Simulations and careful examination of the entire IP-Core design revealed that there's no problem in the logic itself. It is *very* likely that the clock is simply too fast for the logic to keep up. The synthesizer tool also throws corresponding warnings in the time analysis. Further investigating the problem revealed that most likely the framebuffer address calculation in the rectangle renderer which involves three additions and two multiplications is too slow (longest path). A solution (or workaround) for this problem would be to pipeline the calculation of the framebuffer address. This would increase the latency by one clock cycle but wouldn't affect the throughput (speed). This wasn't implemented during this project because it was decided to spend the little time left after finding and solving the Avalon-MM bus problem in finalizing the rest of the design. After all, this problem shouldn't appear as the QSys tool is responsible for adding the clock domain crossing bridge when using the slower clock.

# 19 Conclusion

The goal of this thesis was to develop a proof-of-concept implementation of an IP-core that provides hardware acceleration for 2D rendering operations in a NIOS-II system. This goal has been reached. The current state of the IP-core provides everything necessary to render single pixels and filled rectangles without using the CPU. Furthermore, other features such as a clipping mask and framebuffer memory boundary checks have been implemented as well. But most importantly, the current design provides all the infrastructure to implement hardware renderers for more shapes such as lines, polygons, circles and so on. A new hardware renderer can be written isolated from the entire system and just plugged in at the end without changing anything else in the design. Everything required to read back pixel data from the framebuffer is also already implemented which is required for more complex hardware renderers that render with anti-aliasing or for hardware renderers that are used to copy images or scrolling the framebuffer contents.

At the beginning, the plan was to implement hardware renderers for lines, circles, polygons and other shapes as well during this thesis. Unfortunately, that couldn't be achieved due to a problem that was encountered while working on the Avalon-MM master bus interface implementation in the framebuffer interface. This problem resulted in about four weeks of delay (see section 18.1). After the problem was resolved (or rather work-arounded in this case) I decided to move on with implementing the rest of the infrastructure after the filled-rectangle renderer was completed. The current state of the IP-core appears to be stable and is ready to be extended with more hardware renderers.

The resulting IP-core has been demonstrated to various people at Intel which resulted in very positive feedback. Intel provides a large set of IP-cores for 2D graphics and especially video handling but none of them provides hardware acceleration for 2D rendering. The speed at which Silizium can render rectangles while leaving the CPU unused was therefore very impressive.

Although this is the end of the bachelor thesis this is not the end of the project. The Silizium IP-core will be extended and further maintained by the µGFX company. Furthermore, Intel is interested in getting a license that allows them to add Silizium to their catalog of existing IP-cores.

# 20 Signatures

Joel Bodenmann, 2017/08/18

# 21 Credits

Special thanks go to the following people which helped to realize this project:

- **François Corthay**, responsible professor, HEVs
- **Yann Thoma**, project expert, HEVs
- **Scott Prigmore**, technical assistance, Intel
- **Doug Rydberg**, pre-project management, Intel

# 22 Appendix

- IP-Core datasheet (part of this report)
- Avalon master interface problem report (part of this report)
- CD-ROM / SD-CARD / USB-Stick containing:
    - PDF versions of these documents (and related documents listed below)
    - VHDL source code of the IP-Core
    - C library HAL for the IP-Core

# 23 Bibliography


| | |
|---|---|
| **Document name** | Avalon interface specifications |
| **Document ID** | MNL-AVABUSREF |
| **Issuer** | Altera / Intel |
| **Version or date** | 2017.05.08 |
| **Link** | https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf |

| | |
|---|---|
| **Document name** | SCFIFO and DCFIFO IP Cores User Guide |
| **Document ID** | UG-MFNALT_FIFO |
| **Issuer** | Altera / Intel |
| **Version or date** | 2017.05.08 |
| **Link** | https://www.altera.com/en_US/pdfs/literature/ug/ug_fifo.pdf |

| | |
|---|---|
| **Document name** | Quartus II Handbook Volume I |
| **Document ID** | QII5V1 |
| **Issuer** | Altera / Intel |
| **Version or date** | 2015.05.04 |
| **Link** | https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts_qii5v1.pdf |

| | |
|---|---|
| **Document name** | Silizium IP core - Datasheet |
| **Document ID** | N/A |
| **Issuer** | uGFX GmbH |
| **Version or date** | v1.0 |
| **Link** | https://silizium.io/datasheet.pdf |

# 24 List of illustrations



# 25 List of tables

SILIZIUM

# Silizium IP core - Datasheet

Document version: v1.0
IP-Core version: v0.1

2017-08-15

# Introduction

Silizium is a GPU (graphics processing unit) IP-core written in VHDL that provides hardware acceleration for various 2D rendering operations. The IP-core is optimized for the use in a NIOS-II system.

# Features

- Avalon-MM bus compatible
- Can use an already existing framebuffer (doesn't maintain a dedicated framebuffer internally)
- Works with any display resolution and pixel format
- Easy to use
- Framebuffer memory boundary checks (can be disabled)
- Rectangular clipping mask support (can be disabled)

# Requirements

- NIOS-II system
- QSys tool
- Avalon-MM bus for command & control communication
- Framebuffer that is accessible through an Avalon-MM bus

# Current limitations

- No parallel rendering operations
- No burst transactions to/from the framebuffer memory

# Currently implemented hardware renderers

- Draw pixel
- Fill area
- Rectangular clipping
- Coming soon:
    - Memory blitting (2D DMA)
    - Vertical & Horizontal scroll
    - Lines
    - Polygons
    - Circles
    - Arcs
    - Color keying
    - …

# Terms & Abbreviations

The following table gives an overview of terms and abbreviations that are commonly used throughout this document:

| Abbreviation | Description |
|---|---|
| 2D | Two dimensional |
| Avalon | A bus standard created by Altera for FPGA internal communication |
| CPU | Central processing unit |
| FBI | Framebuffer interface |
| FIFO | First-in First-out (a type of memory/buffer) |
| Framebuffer | A section of memory that holds the pixel data that is shown on the display |
| FSM | Finite state machine |
| GPU | Graphics processing unit |
| HAL | Hardware abstraction layer |
| IP-Core | (Intellectual property) A pre-fabricated block of something ready to be used |
| Qsys | Tool of the Quartus toolchain used to create a SoC |
| Quartus | The FPGA IDE & Toolchain by Intel |
| Silizium | The name of this 2D hardware acceleration IP-Core |
| SoC | System-on-chip |

TABLE 1: COMMONLY USED ABBREVIATIONS & TERMS IN THIS DOCUMENT

# Typical application

Figure 1 shows typical applications of the Silizium IP-Core inside a NIOS-II system. However, due to the versatility of the Avalon interface, many different and alternative configurations are possible. Everything that Silizium requires is an Avalon-MM interface towards the CPU for configuration and an Avalon-MM interface towards the framebuffer for rendering. How these are actually implemented (eg. whether the Framebuffer is on-chip or external, whether it's SDRAM or DDR3, whether the display controller is internal or external and so on) doesn't have any impact on the Silizium IP-Core.

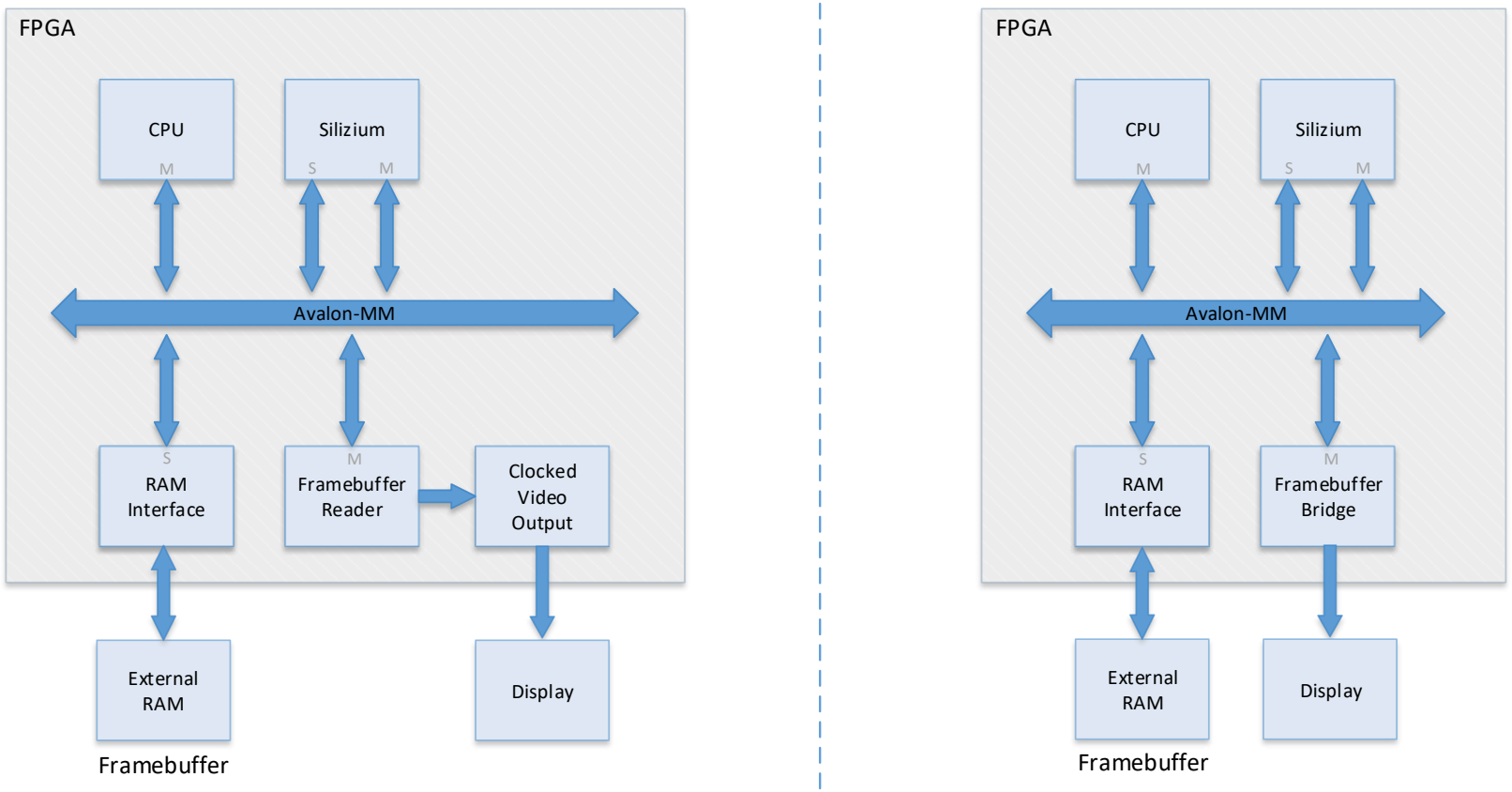



FIGURE 1: TYPICAL APPLICATIONS

# 1 Internal design

Figure 2 illustrates the overall internal design of the system:

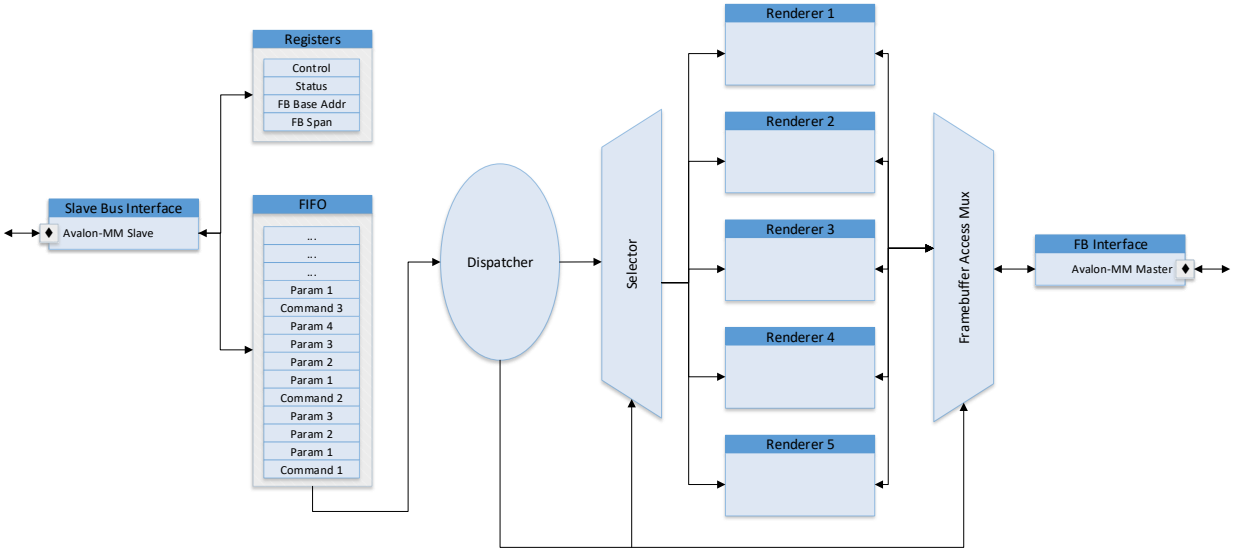



FIGURE 2: INTERNAL DESIGN OVERVIEW

The internal design can be split into the following groups:

- Command & control bus interface
- Registers
- Command FIFO
- Dispatcher
- Renderers
- Framebuffer interface (FBI)

## 1.1 Command & control bus interface

The command & control bus interface (from here on referred to as *"the slave interface"*) is used by the CPU to initialize, configure and control the GPU. The interface is compliant to the Avalon-MM slave standard and has the following requirements:

- Address bus width: At least 4 bits
- Data bus width: At least 32 bits
- Read and write operations

The run-time configuration of the GPU usually only consists of setting the framebuffer base address and the framebuffer span. Other parameters such as the display size, the pixel format are handled by generic values and therefore do not need to be changed during run-time. Once the configuration is completed the slave interface is usually only used to send commands to the GPU, to control the different enable flags (if required) and to read back the current status of the GPU (eg. busy flags, queue status and so on).

## 1.2 Registers

Silizium features different control and configuration registers. The most important ones are:

- Status register
- Control register
- Framebuffer base address & span register

These registers can be accessed directly via the slave interface.

Section 5 gives a detailed overview over all registers.

## 1.3 Command FIFO

The CPU issues commands to the GPU such as *"draw a rectangle at this position with that size and this color"*. These commands are stocked in the internal command FIFO of the GPU. Whenever there is a pending command in the FIFO, the dispatcher will grab it as soon as the different renderers of the GPU are no longer busy and handle it accordingly.

## 1.4 Dispatcher

The dispatcher takes commands out of the command FIFO and dispatches them to the proper hardware rendering blocks. To fulfill that task, the dispatcher also handles the internal bus arbitration and later parallel rendering (not implemented yet).

## 1.5 Renderers

Silizium contains an isolated hardware renderer for each rendering operation that the GPU has to offer.

Note: In the future, it will also be possible to have multiple instances of the same hardware rendering inside the GPU which allows to handle multiple rendering operations of the same type simultaneously. For example: Having more than one block to hardware render filled rectangles means that multiple filled rectangles can be rendered in parallel.

## 1.6 Framebuffer interface

Silizium needs access to the framebuffer to perform the actual rendering operations. The different hardware renderers send data and commands to the framebuffer interface (FBI) to alter the framebuffer contents. The framebuffer interface consists of two different FIFOs that the renderers can access: A write-FIFO and a read-FIFO. Renderers send commands and framebuffer data via the write-FIFO to the FBI. Some renderers require to read back data from the framebuffer. The FBI reads the requested data from the framebuffer and places it into the read FIFO where the renderers can dispatch it from. This way, the renderers are completely isolated from the FBI which means that the FBI never has to wait on a renderer (unless the write FIFO is empty) which in turn means that the FBI can optimize framebuffer transfers by using burst transactions.

Note that the write- and the read-FIFOs are not accessible through the slave interface. They are used exclusively internally.

# 2 Generics

Table 2 lists the generics available to adjust the hardware Silizium IP-Core hardware prior to synthesizing a design using it:

| Name | Type | Default | Description |
|---|---|---|---|
| dataBitNb | positive | 32 | Internal data handling width. Will be used in the future. **DO NOT CHANGE THIS**. |
| coordBitNb | positive | 32 | The number of bits required to represent a coordinate in two's complement. |
| colorBitNb | positive | 32 | The number of bits required to represent a color value (internally). |
| cmdFifoNumWordsExp | positive | 6 | The exponent of the size of the input command FIFO in elements. Actual size will be 2^n of this value. |
| fbAddrBitNb | positive | 32 | The width of the address bus of the framebuffer interface. |
| fbDataBitNb | positive | 32 | The width of the data bus of the framebuffer interface. |
| fbBurstcountBitNb | positive | 2 | The number of bits required for the burstcount signal. |
| fbBytesPerPixel | positive | 4 | The number of bytes per pixel. |
| fbWidth | positive | 800 | The width of the framebuffer in pixels. |
| fbHeight | positive | 480 | The height of the framebuffer in pixels. |
| fbWriteFifoNumWordsExp | positive | 8 | The exponent of the size of the framebuffer write FIFO in elements. Actual size will be 2^n of this value. |
| fbReadFifoNumWordsExp | positive | 4 | The exponent of the size of the framebuffer read FIFO in elements. Actual size will be 2^n of this value. |

TABLE 2: GENERICS

When using QSys, these generics are also available as configuration parameters in the graphical component configuration dialog:

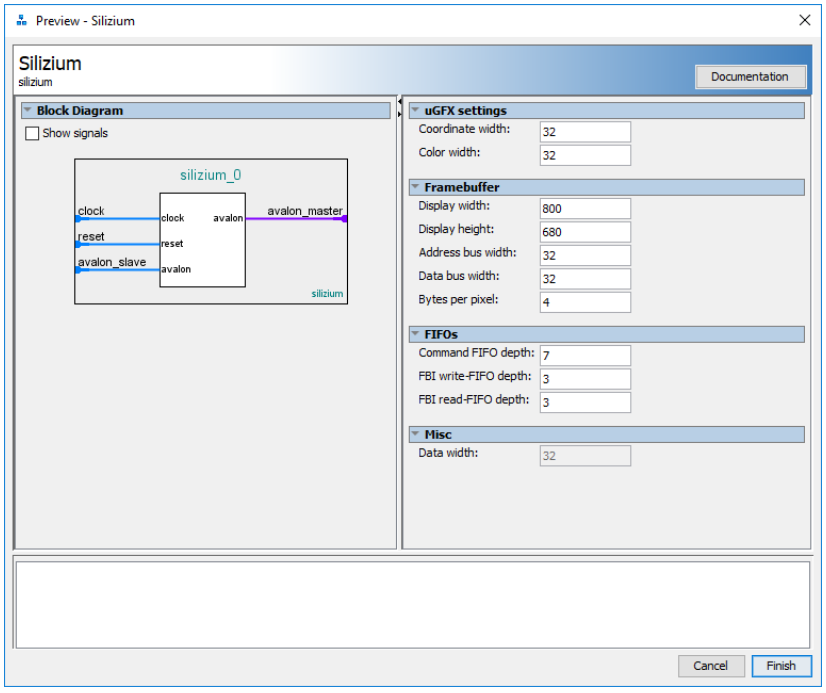



FIGURE 3: QSYS IP-CORE CONFIGURATION DIALOG

## 2.1 Recommendations

Except for the FIFO sizes all values are determined by the rest of the system (mainly the used display and display controller).

### 2.1.1 Command FIFO depth

The command FIFO depth can be set through the *cmdFifoNumWordsExp* generic value. The actual number of elements of the command FIFO is two to the power of this value:

$$command\ FIFO\ depth = 2^{cmdFifoNumWordsExp}$$

The value of *cmdFifoNumWordsExp* must be at least *2*.

This FIFO should be large enough to hold all subsequent rendering commands that the CPU will issue at once (directly one after another) to prevent blocking the CPU as it will have to wait if the command FIFO is full. However, in a typical application a CPU will issue just a few rendering commands and then wait for them to complete prior to continuing render other elements manually on top of that. This means that the CPU will never issue hundreds or even just dozens of rendering commands and then do other non-rendering jobs for a long time. A larger FIFO would therefore be mostly empty.

The default (and recommended) depth of the command FIFO is 64 elements.

### 2.1.2 FBI write-FIFO depth

The FBI write FIFO depth can be set through the *fbWriteFifoNumWordsExp* generic value. The actual number of elements of the FBI write-FIFO is two to the power of this value:

$$FBI\ write\ FIFO\ depth = 2^{fbWriteFifoNumWordsExp}$$

The value of *fbWriteFifoNumWordsExp* must be at least *2*.

There are two reasons for having this FIFO:

1. The FBI can issue burst writes
2. The renderer does not have to wait if another component locks up the bus to the framebuffer memory

The default (and recommended) depth of the FBI write FIFO is 256 elements.

### 2.1.3 FBI read-FIFO depth

The FBI read-FIFO depth can be set through the *fbReadFifoNumWordsExp* generic value. The actual number of elements of the FBI read FIFO is two to the power of this value:

$$FBI\ read\ FIFO\ depth = 2^{fbReadFifoNumWordsExp}$$

The value of *fbReadFifoNumWordsExp* must be at least *2*.

The purpose of this FIFO is to allow the FBI to perform burst reads and to prevent bus locks if a renderer is busy and can't dispatch the read-back value(s) immediately. The maximum size of read bursts is usually limited by the used memory controller. The default (and recommended) depth of the FBI read FIFO is 8 elements.

# 3 Ports

Besides the clock and reset input, Silizium only requires an Avalon-MM slave interface and an Avalon-MM master interface to operate:

| Name | Direction | Type | Range |
|---|---|---|---|
| reset | in | std_logic | - |
| clock | in | std_logic | - |
| avalon_slave_address | in | std_logic_vector | 7 downto 0 |
| avalon_slave_read | in | std_logic | - |
| avalon_slave_readdata | out | std_logic_vector | 31 downto 0 |
| avalon_slave_write | in | std_logic | - |
| avalon_slave_writedata | in | std_logic_vector | 31 downto 0 |
| avalon_slave_waitrequest | out | std_logic | - |
| avalon_master_address | out | std_logic_vector | fbAddrBitNb-1 downto 0 |
| avalon_master_write | out | std_logic | - |
| avalon_master_writedata | out | std_logic_vector | fbDataBitNb-1 downto 0 |
| avalon_master_waitrequest | in | std_logic | - |
| avalon_master_read | out | std_logic | - |
| avalon_master_readdata | in | std_logic_vector | fbDataBitNb-1 downto 0 |
| avalon_master_readdatavalid | in | std_logic | - |
| avalon_master_burstcount | out | std_logic_vector | fbBurstcountBitNb-1 downto 0 |

TABLE 3: PORTS

Note that the Avalon interfaces can be connected to buses that have larger widths as the QSys tool will automatically generate the required bus rippers and mergers. For example, Silizium can be connected to an Avalon-MM bus with an address width of 32 bits without any problems.

Figure 4 and Figure 5 illustrate the waveforms required to communicate with the slave interface:

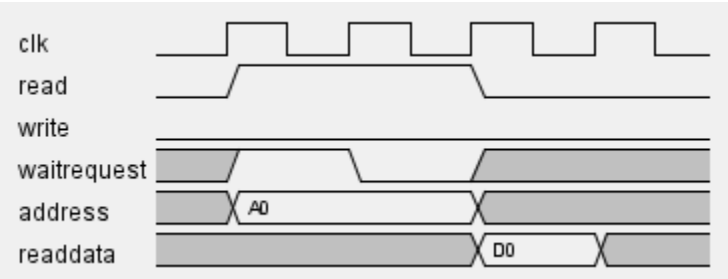



FIGURE 4: SLAVE INTERFACE READ WAVEFORM

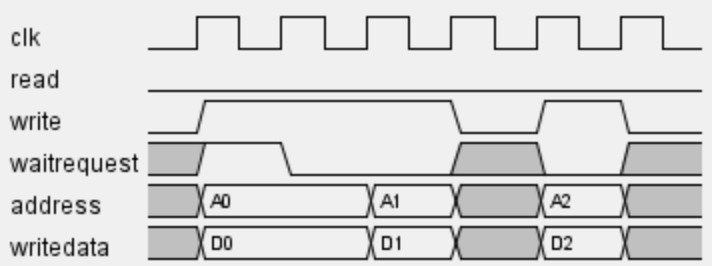



FIGURE 5: SLAVE INTERFACE WRITE WAVEFORM

In the current version, the master interface is not yet configurable except for the different signal width. Figure 6 and Figure 7 show the waveforms the master interfaces uses to read from and write to the framebuffer memory:

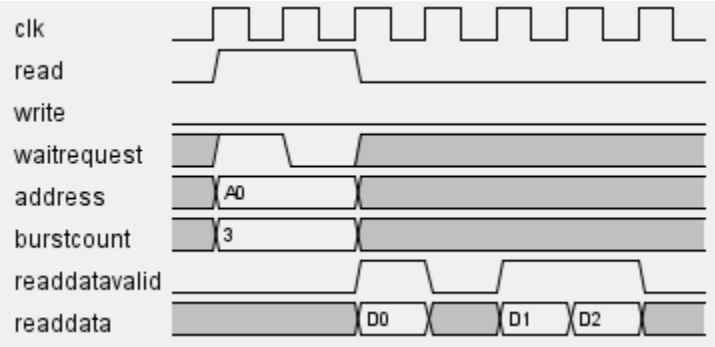



FIGURE 6: MASTER INTERFACE READ WAVEFORM

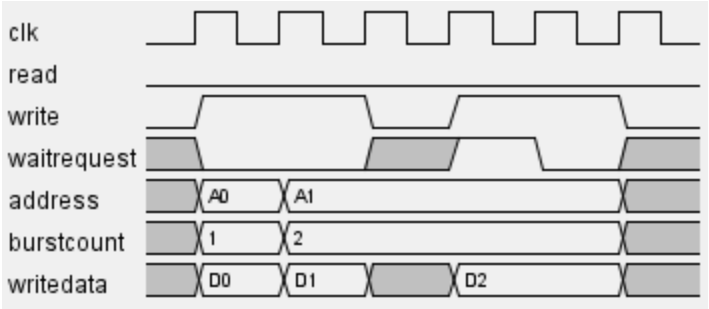



FIGURE 7: MASTER INTERFACE WRITE WAVEFORM

# 4 Additional features

Some of the features offered by Silizium are not relevant to the rendering operations themselves but are still relevant to the overall system.

## 4.1 Boundary checks

The framebuffer interface (FBI) can prevent write and read transactions to and from the framebuffer by performing boundary checks on the framebuffer addresses provided by the renderers prior to actually issuing the requested transaction on the bus. Any transactions that would occur outside of the framebuffer will be ignored / discarded by the FBI if the corresponding *BoundaryChecksEnable*-Bit (BCEN) is set in the control register.

# 5 Registers

This section of the datasheet documents all the publicly available registers. Every single bit is documented with the position inside the register and the corresponding read, write or read-write information.

Note: Do not try to read or write bits marked as *"Reserved"*. Behavior is undefined.

The following table gives an overview of all the existing registers:

| Address offset | Name | Width | Access |
|---|---|---|---|
| 0x00 | VERSION | 32-bits | read-only |
| 0x01 | CMD_FIFO | 32-bits | write-only |
| 0x02 | STATUS | 32-bits | read-only |
| 0x03 | CONTROL | 32-bits | read/write |
| 0x04 | FB_BASE | 32-bits | read/write |
| 0x05 | FB_SPAN | 32-bits | read/write |
| 0x06 | CLIP_X | 32-bits | read-only |
| 0x07 | CLIP_Y | 32-bits | read-only |
| 0x08 | CLIP_W | 32-bits | read-only |
| 0x09 | CLIP_H | 32-bits | read-only |
| 0x0D | DUMMY_1 | 32-bits | read-only |
| 0x0E | DUMMY_2 | 32-bits | read-only |
| 0x0F | DUMMY_3 | 32-bits | read/write |

TABLE 4: REGISTER MAP

## 5.1 Version

Register to read the Silizium version number.

Address offset: 0x0000 0000
Reset value: 0x<version>

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VERS[31:16] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VERS[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

Bits 31:0 **VERS[31:0]:** Version number
The version number that is burned into the hardware.

## 5.2 Command FIFO

This is a virtual register used to write to the command FIFO. Silizium will take values written to this register and move them to the command FIFO the clock cycle following the write operation.

Address offset: 0x0000 0001
Reset value: Undefined (write only)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CMD_FIFO[31:16] | | | | | | | | | | | | | | | |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CMD_FIFO[15:0] | | | | | | | | | | | | | | | |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |

Bits 31:0 **CMD_FIFO[31:0]:** Command FIFO
The command FIFO to which commands are written to.

## 5.3 Status

Generic status register to check the current state of Silizium.

Address offset: 0x0000 0002
Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CMDFUW[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | | | | | | MPTY | FULL | BUSY |
| | | | | | | | | | | | | | r | r | r |

Bit 0 **BUSY** Busy flag
Flag that indicates whether a rendering operation is being executed.
0: No rendering operation is being executed. Silizium is idling.
1: A rendering operation is being executed

Bit 1 **FULL** Command FIFO full flag
0: Command FIFO is not full.
1: Command fifo is full.

Bit 2 **MPTY** Empty
The input command FIFO is empty
0: Command FIFO is not empty
1: Command FIFO is empty

Bits 15:8 **CMDFUW[15:0]:** Command FIFO used words

The number of used words of the command FIFO.

Note: The actual width depends on the command FIFO depth set via the generics value of the IP-core block. This field will always be right-aligned to the 16th bit. All "unused" bits on the left in case of a FIFO depth that is less than 2^16 will be read back as '0'.

## 5.4 Control

Generic control register to control the behavior of Silizium.

Address offset: 0x0000 0003
Reset value: 0x0001 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | | | | | | | CMEN | BCEN |
| | | | | | | | | | | | | | | rw | rw |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | CLR3 | CLR2 | CLR1 | Reserved | | | | | | EN2 | EN1 |
| | | | | | rw | rw | rw | | | | | | | rw | rw |

Bit 0 **EN1:** Dispatcher enable

The enable control bit for the dispatcher.

0: Dispatcher disabled

1: Dispatcher enabled

Bit 1 **EN2:** Framebuffer interface enable

The enable control bit for the framebuffer interface.

0: Framebuffer interface disabled

1: Framebuffer interface enabled

Bit 8 **CLR1:** Clear the command input FIFO

Control bit to issue a hardware clear of the input command FIFO. If set to '1', a hardware clear will be issued. The bit will be reset to '0' automatically once the clear operation completed.

Clearing the FIFO takes 1 clock cycle.

Bit 9 **CLR2:** Clear the framebuffer write FIFO

Control bit to issue a hardware clear of the framebuffer interface write FIFO. If set to '1', a hardware clear will be issued. The bit will be reset to '0' automatically once the clear operation completed.

Clearing the FIFO takes 1 clock cycle.

Bit 10 **CLR3:** Clear the framebuffer read FIFO

Control bit to issue a hardware clear of the framebuffer interface read FIFO. If set to '1', a hardware clear will be issued. The bit will be reset to '0' automatically once the clear operation completed.
Clearing the FIFO takes 1 clock cycle.

Bit 16 **BCEN:** Boundary check enable
The enable control bit for framebuffer memory boundary check.
0: Framebuffer memory boundary checks disabled
1: Framebuffer memory boundary checks enabled

Bit 17 **CMEN:** Clipping mask enable
The enable control bit for the clipping mask in the framebuffer interface.
0: Clipping mask disabled
1: Clipping mask enabled

## 5.5 Framebuffer base address

This register is used to provide the framebuffer base address to Silizium. This register must be configured to hold the address of the first pixel of the framebuffer.

Note that although Silizium provides a framebuffer memory boundary check feature it is crucial that this value is set properly. The memory boundary checks are based on this value. Misconfiguring this register can lead to memory corruption.

Check section 9 for examples regarding the proper usage of this register.

Address offset: 0x0000 0004
Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FB_BASE[31:16] | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FB_BASE[15:0] | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 31:0 **FB_BASE[31:0]:** Framebuffer base address
The base address of the framebuffer.

## 5.6 Framebuffer address span

This register is used to inform Silizium about the size of the framebuffer memory section. The framebuffer interface uses this information to perform boundary checks to prevent illegal memory accesses if the *BCEN* bit is set to *'1'* in the control register.

Note that although Silizium provides a framebuffer memory boundary check feature it is crucial that this value is set properly. The memory boundary checks are based on this value. Misconfiguring this register can lead to memory corruption.

Check section 9 for examples regarding the proper usage of this register.

Address offset: 0x0000 0005
Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FB_SPAN[31:16] | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FB_SPAN[15:0] | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 31:0 **FB_SPAN[31:0]:** Framebuffer address span
The size of the framebuffer memory sections in bytes.

## 5.7 Clip X

This register can be used to read back the current X coordinate of the rectangular clipping mask provided by the FBI. Note that this is a read-only register: The value of this register can only be changed by issuing a clipping mask modification command as described in section 6.3 to prevent synchronization issues.

Address offset: 0x0000 0006
Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPX[31:16] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPX[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

Bits 31:0 **CLIPX[31:0]:** Clip X
The X coordinate of the rectangular clipping mask of the FBI.

## 5.8 Clip Y

This register can be used to read back the current Y coordinate of the rectangular clipping mask provided by the FBI. Note that this is a read-only register: The value of this register can only be changed by issuing a clipping mask modification command as described in section 6.3 to prevent synchronization issues.

Address offset: 0x0000 0007
Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPY[31:16] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPY[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

Bits 31:0 **CLIPY[31:0]:** Clip Y
The Y coordinate of the rectangular clipping mask of the FBI.

## 5.9 Clip width

This register can be used to read back the current width of the rectangular clipping mask provided by the FBI. Note that this is a read-only register: The value of this register can only be changed by issuing a clipping mask modification command as described in section 6.3 to prevent synchronization issues.

Address offset: 0x0000 0008
Reset value: 0x1111 1111

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPW[31:16] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPW[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

Bits 31:0 **CLIPW[31:0]:** Clip width
The width of the rectangular clipping mask of the FBI.

## 5.10 Clip height

This register can be used to read back the current height of the rectangular clipping mask provided by the FBI. Note that this is a read-only register: The value of this register can only be changed by issuing a clipping mask modification command as described in section 6.3 to prevent synchronization issues.

Address offset: 0x0000 0009
Reset value: 0x1111 1111

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPH[31:16] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLIPH[15:0] | | | | | | | | | | | | | | | |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

Bits 31:0 **CLIPH[31:0]:** Clip height
The height of the rectangular clipping mask of the FBI.

## 5.11 Dummy 1

This dummy register can be used to test the slave interface configuration (eg. to recognize wrongly configured read latencies and similar) in the QSys tool.

Address offset: 0x0000 000D
Reset value: 0xD0D0 0D0D

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

## 5.12 Dummy 2

This dummy register can be used to test the slave interface configuration (eg. to recognize wrongly configured read latencies and similar) in the QSys tool.

Address offset: 0x0000 000E
Reset value: 0xE0E0 0E0E

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

## 5.13 Dummy 3

This dummy register can be used to test the slave interface configuration (eg. to recognize wrongly configured read latencies and similar) in the QSys tool.

Address offset: 0x0000 000F
Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DUMMY_3[31:16] | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DUMMY_3[15:0] | | | | | | | | | | | | | | | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 31:0 **DUMMY_3[31:0]:** Dummy 3

A dummy register for debugging purposes. Retains the content indefinitely. The content is not being used by any part of the IP-Core and never gets modified.

# 6 Rendering

This section of the document explains the different available hardware renderers and how they are to be used. The commands that need to be send to the input command FIFO are documented in tabular form:

| Order | Name | Unit | Value |
|---|---|---|---|
| **1** | Parameter 1 | - | constant |
| **2** | Parameter 2 | Pixel Count | < variable > |
| **3** | Parameter 3 | Pixel Value | < variable > |
| **4** | Parameter 4 | Pixel Count | constant |

TABLE 5: FIFO PARAMETERS SYNOPSIS

Parameters are being listed in ascending order from top to bottom. This means that the top most parameter is the first one to be written to the command FIFO.

The *Name* column specifies the name of the parameter which also serves as the description. The *Unit* column specifies the unit of the parameter value. Lastly, the *Value* column either shows the constant that needs to be written or the name of the variable in angle brackets. **Note that the *Value* is the only thing that gets written to the FIFO. All values being written to the command FIFO are 32-bits wide.**

## 6.1 Draw pixel

Silizium offers the possibility to just render single pixels. This hardware rendering feature is not there for acceleration but for two other reasons instead:

- **Debugging:** This feature allows a developer to test whether his display driver is working properly and to figure out whether the framebuffer base address and span have been configured correctly without having to issue a more complex command. Setting just one pixel doesn't involve "off-by-one" errors and similar.
- **Chronology:** Sometimes the user of the GPU might want to render a shape that is built from different sub-shapes. As a rough example: Maybe a filled rectangle with just a differently colored pixel in each corner is required. This would mean that the user asks the GPU to render the filled rectangle by putting the corresponding command into the FIFO. Once that is done the CPU will have to render the pixel on top of that. This means that the CPU has to query the GPU to ask whether the rectangle has been finished because otherwise the CPU might prematurely render the four corner-pixels and the GPU will render the filled rectangle on top of that. This is very difficult to implement. With this *"draw pixel"* command the CPU can simply put a *"fill rectangle"* followed by four *"draw pixel"* commands into the command input FIFO of the GPU.

To draw a pixel, the following parameters must be written to the command input FIFO:

| Order | Name | Unit | Value |
|---|---|---|---|
| **1** | Command | - | 0x0000 0001 |
| **2** | X | Pixel Count | < x > |
| **3** | Y | Pixel Count | < y > |
| **4** | Color | Pixel Value | < color > |

TABLE 6: DRAW PIXEL - FIFO PARAMETERS

**Example:** Drawing a red pixel at the position X = 165 and Y = 504 assuming that the used pixel format is RGB888 requires writing the following into the command FIFO:

| Order | Value |
|---|---|
| **1** | 0x0000 0001 |
| **2** | 0x0000 00A5 |
| **3** | 0x0000 01F8 |
| **4** | 0x00FF 0000 |

TABLE 7: DRAW PIXEL COMMAND EXAMPLE

## 6.2 Filled rectangle

To draw a filled rectangle (solid color), the following parameters must be written to the command FIFO:

| Order | Name | Unit | Value |
|---|---|---|---|
| **1** | Command | - | 0x0000 0002 |
| **2** | X | Pixel Count | < x > |
| **3** | Y | Pixel Count | < y > |
| **4** | Width | Pixel Count | < width > |
| **5** | Height | Pixel Count | < height > |
| **6** | Color | Pixel Value | < color > |

TABLE 8: FILL RECTANGLE - FIFO PARAMETERS

**Example:** Drawing a green rectangle at the position X = 165 and Y = 504 with width = 65 and height = 204 assuming that the used pixel format is RGB888 requires writing the following into the command FIFO:

| Order | Value |
|---|---|
| **1** | 0x0000 0002 |
| **2** | 0x0000 00A5 |
| **3** | 0x0000 01F8 |
| **4** | 0x0000 0041 |
| **5** | 0x0000 00CC |
| **6** | 0x0000 FF00 |

TABLE 9: FILL RECTANGLE COMMAND EXAMPLE

## 6.3 Clipping

Clipping is not a rendering operation as it doesn't directly change the contents of the framebuffer but it's still relevant to the overall rendering process as it affects the rendering output of all the other rendering operations. Silizium provides one rectangular clipping mask that can be enabled and disabled through the corresponding enable bit (CMEN) in the control register. If the clipping mask enable bit is set in the configuration register, everything outside of the rectangular clipping mask that can be defined through this command will not be rendered to the framebuffer.

Clipping values are not implemented as registers but instead get fed through the command FIFO to avoid synchronization problems. For proper operation, it's crucial that the chronological order of rendering operations and everything that affects them doesn't get broken. However, the currently active clipping mask values can still be read back through the corresponding clipping mask values registers.

| Order | Name | Unit | Value |
|---|---|---|---|
| **1** | Command | - | 0x0000 0003 |
| **2** | X | Pixel Count | < x > |
| **3** | Y | Pixel Count | < y > |
| **4** | Width | Pixel Count | < width > |
| **5** | Height | Pixel Count | < height > |

TABLE 10: CLIPPING MASK - FIFO PARAMETERS

**Example:** Setting the clipping mask to X = 150, Y = 170, width = 300 and height = 150 requires writing the following into the command FIFO:

| Order | Value |
|---|---|
| **1** | 0x0000 0003 |
| **2** | 0x0000 0096 |
| **3** | 0x0000 00AA |
| **4** | 0x0000 012C |
| **5** | 0x0000 00096 |

TABLE 11: CLIPPING MASK MODIFICATION COMMAND EXAMPLE

# 7 HAL

A HAL (Hardware Abstraction Layer) implemented fully in C is provided with the IP-Core for easy integration into existing software projects. The HAL is split into a low-level and a high-level part. The low-level HAL provides API functions to access the various registers and FIFOs while the high-level HAL takes this information and provides easier to use interfaces for checking & settings flags and rendering shapes.

Note that for ease-of-use both HALs are implemented in the same files. The entire HAL is encapsulated in one header and one source file named *silizium.h* and *silizium.c*.

The HAL is completely C89 compatible. The following 3rd-party resources are used:

- *stdint.h*
- *stddef.h*
- *stdbool.h*

## 7.1 Low-Level HAL

The low-level HAL provides the following functions:

```
uint32_t siliziumVersion();
uint32_t siliziumStatus();
uint32_t siliziumControl();
void siliziumSetControl(uint32_t value);
```

Detailed descriptions of these functions and their parameters and return values can be found in the corresponding API documentation.

## 7.2 High-Level HAL

The high-level HAL is built on top of the low-level HAL and provides the interfaces that a regular user will use such as easy to use functions to check busy flags, to change enable flags and to issue rendering commands. The following functions are available:

```
void siliziumDispatcherEnable();
void siliziumDispatcherDisable();
void siliziumFramebufferinterfaceEnable();
void siliziumFramebufferinterfaceDisable();
void siliziumEnableAll();
void siliziumDisableAll();
void siliziumClippingEnable();
void siliziumClippingDisable();
void siliziumBoundaryChecksEnable();
void siliziumBoundaryChecksDisable();
bool siliziumTestSlaveInterface();
void siliziumSetFramebufferBaseAddress(uint32_t address);
void siliziumSetFramebufferSpan(uint32_t span);
uint32_t siliziumFramebufferBaseAddress();
uint32_t siliziumFramebufferSpan();
bool siliziumIsBusy();
void siliziumCmdFifoAppend(uint32_t value);
void siliziumCmdFifoClear();
bool siliziumCmdFifoIsEmpty();
bool siliziumCmdFifoIsFull();
size_t siliziumCmdFifoUsedWords();
size_t siliziumCmdFifoFreeWords();
void siliziumRenderPixel(uint32_t x, uint32_t y, uint32_t color);
void siliziumRenderRectangle(uint32_t x, uint32_t y, uint32_t width, uint32_t
height, uint32_t color);
void siliziumRenderSetClippingArea(uint32_t x, uint32_t y, uint32_t width,
uint32_t height);
```

Additionally, the high-level HAL needs to know the size of the command FIFO to calculate the number of free elements. A simple macro is used for this purpose. In this example, the command FIFO depth has been set to 128 elements prior to synthesis by setting the *cmdFifoNumWordsExp* generic to 7:

```
#define SILIZIUM_CMD_FIFO_DEPTH 128
```

Detailed descriptions of these functions and their parameters and return values can be found in the corresponding API documentation.

# 8 µGFX integration

µGFX provides a read-to-use built-in driver starting with µGFX version 2.8 that allows using Silizium in an µGFX application without modifying the application itself.

# 9 Examples

## 9.1 Initialization

A typical initialization sequence of Silizium after power-on looks like this (using the HAL):

```
// Allocate framebuffer
void* fbPointer = malloc(SCREEN_WIDTH * SCREEN_HEIGHT * 4);

// Halt silizium (optional)
siliziumCmdFifoClear();
siliziumDisableAll();

// Setup framebuffer interface
siliziumSetFramebufferBaseAddress(fbPointer);
siliziumSetFramebufferSpan(SCREEN_WIDTH * SCREEN_HEIGHT * 4);

// Enable silizium
siliziumEnableAll();
```

## 9.2 Checking for busy

Sometimes it's necessary for the CPU to know when Silizium finished rendering all the jobs queued up in the command FIFO. This can be done easily using the *siliziumIsBusy()* function provided by the HAL:

```
// Wait for silizium to finish rendering
while (siliziumIsBusy());
```

## 9.3 Checking the command FIFO state

In almost all cases it's important to know the current state of the command FIFO prior to writing to it. The status register provides flags for the *isEmpty* and *isFull* states. Additionally, the number of used words can be retrieved through the status register which allow to calculate the number of free words. The high-level HAL provides high-level functions for all of these:

```
bool siliziumCmdFifoIsEmpty();
bool siliziumCmdFifoIsFull();
size_t siliziumCmdFifoUsedWords();
size_t siliziumCmdFifoFreeWords();
```

Furthermore, the following functions are provided by the high-level HAL to modify the command FIFO:

```
void siliziumCmdFifoClear();
void siliziumCmdFifoAppend(uint32_t value);
```

## 9.4 Rendering

After initialization, rendering commands are being issued by writing to the input command FIFO as explained in section 6. The following code illustrates how to queue the rendering of a filled rectangle using the HAL:

```
// Render a filled rectangle
void fillRectangle(int x, int y, int width, int height, int color)
{
    siliziumCmdFifoAppend(0x00000002);
    siliziumCmdFifoAppend(x);
    siliziumCmdFifoAppend(y);
    siliziumCmdFifoAppend(width);
    siliziumCmdFifoAppend(height);
    siliziumCmdFifoAppend(color);
}
```

However, note that *siliziumCmdFifoAppend()* will not block if there's no room left in the command FIFO. Silizium will ignore any command FIFO appends if the command FIFO is full. The status register allows to check the *isFull* and *isEmpty* states of the command FIFO as well as querying the number of free words in the FIFO. For simplicity, the high-level HAL provides functions that block until there's enough room in the FIFO. Note that those functions are not multi-thread safe:

```
void siliziumRenderRectangle(uint32_t x, uint32_t y, uint32_t width, uint32_t
height, uint32_t color)
{
    // Wait until there's enough room in the FIFO
    // Note that this only works in a single-threaded software model
    while (siliziumCmdFifoFreeWords() < 6);

    siliziumCmdFifoAppend(0x00000002);
    siliziumCmdFifoAppend(x);
    siliziumCmdFifoAppend(y);
    siliziumCmdFifoAppend(width);
    siliziumCmdFifoAppend(height);
    siliziumCmdFifoAppend(color);
}
```

# 10 Metrics

Metrics have been registered by using the following environment, components and configurations:

- Quartus Prime Version 17.0.0 Build 595 04/25/2017 SJ Standard Edition
- MAX 10 10M50DAF484C6G FPGA
- Silizium v0.1

Silizium configuration (generics):

| Name | Value |
|---|---|
| dataBitNb | 32 |
| coordBitNb | 32 |
| colorBitNb | 32 |
| cmdFifoNumWordsExp | 7 |
| fbAddrBitNb | 32 |
| fbDataBitNb | 32 |
| fbBurstcountBitNb | 2 |
| fbBytesPerPixel | 4 |
| fbWidth | 800 |
| fbHeight | 480 |
| fbWriteFifoNumWordsExp | 8 |
| fbReadFifoNumWordsExp | 4 |

TABLE 12: GENERICS VALUES USED FOR RESOURCE USAGE BENCHMARK

Measured resources usage of the Silizium IP-core:

| Resource type | Value |
|---|---|
| LUT-Only LCs | 371 |
| Register-Only LCs | 284 |
| LUT/Register LCs | 668 |
| Logic cells | 1323 |
| Dedicated logic registers | 951 |
| DSP 18x18 | 4 |
| Memory bits | 20736 |
| M9Ks | 3 |

TABLE 13: RESOURCE USAGE ON TEST SYSTEM

Note: "M9ks" signifies multiple "M9k" elements. M9k elements are the RAM cells in this particular FPGA. An M9k element features 9k memory bits.

The memory bits usage (and therefore the M9Ks usage) can be vastly decreased by decreasing the size of the different FIFOs. Especially the command FIFO has been designed quite generously with a depth of 128 elements which most applications won't require.

Table 14 shows the performance/speed of Silizium. Note that these values are not affected by any changes of the generic values or synthesizer optimizations as they are determined by the design itself:

| Renderer | Per pixel | Initial latency | Recovery latency | Latency between pixels |
|---|---|---|---|---|
| Pixel | 1 | 11 | 4 | 4 |
| Filled rectangle | 1 | 13 | 4 | 4 |
| Clipping mask | 0 | 4 | 10 | 0 |

TABLE 14: PERFORMANCE METRICS

All values are number of clock cycles. These metrics are valid under the following conditions:

- All FIFOs are non-full
- There are no hold-offs (waits) on the slave interface bus transactions
- There are no hold-offs (waits) on the bus towards the framebuffer memory

***Per pixel:*** Is the number of clock cycles the renderer needs to render/calculate one pixel

***Initial latency:*** Is the number of clock cycles between the completion of writing the command to the command FIFO and the completion of rendering/calculating the first pixel.

***Recovery latency:*** Is the number of clock cycles between the completion of a rendering operation and the start of the next rendering operation assuming that the next rendering operation was already in the command FIFO when the first rendering operation completed.

***Latency between pixels:*** Is the number of clock cycles between pixel write transactions of the FBI. Note that this is not the amount of clock cycles a hardware renderer needs to render a pixel but the number of clock cycles the FBI needs to read the pixel value calculated by the renderer and actually writing it to the bus towards the framebuffer.

# 11 Document revision history

| Date | Version | Changes |
|---|---|---|
| 2017-08-15 | v1.0 | Adding register map, approving for publishing |
| 2017-08-14 | v0.3 | Improving layout |
| 2017-08-12 | v0.2 | Adding metrics |
| 2017-08-10 | v0.1 | Initial release |