Technische Projektarbeit 1

Video - Hardware

Beaurel I. Ngaleu

Supervised by: Prof. Dr.-Ing. Gundolf Kiefer,

Michael Schäferling

Matrikelnr.: **2197137**

Field of study: **Computer Engineering**

University: **Technical University Augsburg**

Faculty: **Electrical Engineering and Computer Science**

July 24, 2025

The LaTeX template is based on the content available at.

https://github.com/beaurel2/Projektarbeit1_TI

Abstract

As part of this project, a standalone hardware implementation for graphics output was developed and successfully integrated alongside an existing software simulation module. The goal was to design a module capable of translating its internal framebuffer into a VGA-compatible video signal for direct display on standard monitors. Various common color formats and resolutions were implemented and tested to ensure the module's flexibility for future applications. The video signal is generated entirely in hardware and meets VGA timing specifications, providing a stable and flicker-free display. The functionality was successfully demonstrated through a demo animation displayed on a VGA-compatible screen. Furthermore, the architecture was intentionally designed to be extendable in the future. This enables potential support for additional interfaces such as HDMI and DVI, or the use of the system's main memory as the framebuffer. This project illustrates how software-based simulation approaches can be effectively complemented by targeted hardware development to achieve high-performance and resource-efficient image output in embedded systems.

Table of contents

Glossary

Abbreviations

IDE ..... Integrated Development Environment

VHDL ..... Very Hardware Description Language

FPGA ..... Field Programmable Gate Array

I/O ..... Input/Output

VGA ..... Video Graphics Array

SPI ..... Serial Peripheral Interface

HDI ..... High-Density Interconnect

HDMI ..... High-Definition Multimedia Interface

UART ..... Universal Asynchronous Receiver

ASIC ..... Application-Specific Integrated Circuit

Acknowledgments

I want to thank all of the members of the EES-Group Efficient Embedded Systems from my technical university for their support and helpful advice. Especially my team colleague Martin Erichsen for their good cooperation and also Johannes Hofmann for his help from the beginning of this project. My supervisor and adviser Prof. Dr. Gundolf Kiefer and Michael Schäferling provided me with great insights, a good structure, and a lot of helpful feedback.

1 Introduction

1.1 Motivation

The implementation of video output in hardware imposes stringent requirements on timing, data paths, and protocol compliance. To ensure flawless operation while efficiently utilizing development time and resources, prior simulation is essential. Simulation enables detailed verification of the hardware architecture's functionality, testing of various color formats and resolutions, and thorough analysis of critical signal sequences and synchronization mechanisms before deployment on an actual FPGA or ASIC design. Furthermore, it allows systematic evaluation and optimization of design decisions regarding memory architecture, clock frequencies, and module structures. Such simulation not only forms the foundation for successful hardware implementation but also facilitates future extensions, such as adding HDMI or DVI interfaces or integrating the main system memory as a framebuffer. Finally, video simulation provides students and developers with a practical opportunity to design, test, and understand digital systems in a hands-on, visually verifiable manner. The author also has a lot of interest in hardware, FPGA, simulation and has gained a lot of knowledge about it during his studies

1.2 Purpose of the Work

The objective of this project was to develop a hardware module capable of generating VGA-compatible video signals based on a framebuffer architecture. The design aimed to translate the contents of an internal image memory into a synchronized VGA output signal, supporting various color formats and resolutions. Additionally, the implementation was intended to serve as a foundation for future extensions, such as HDMI or DVI output, and integration with system main memory as a framebuffer.

2 Basics

To achieve this project, a design was first created and then each module was implemented. Finally, all modules were instantiated in a top_level module.

In the figure Figure 1 you can see all the modules that played an important, decisive role in completing this project. The exact description of the available registers, their addressing and their functionality can be found in the following external document:

[https://ti-build.informatik.hs-augsburg.de:8443/piconut_developers/piconut/-/blob/dev_wb_video/hw/peripherals/wb_graphics/Graphics-interface%20description.md]

Repository: **piconut**, Branch: *dev_wb_video*. Please note that internal permissions (*VPN* or *university login*) may be required to access the document.

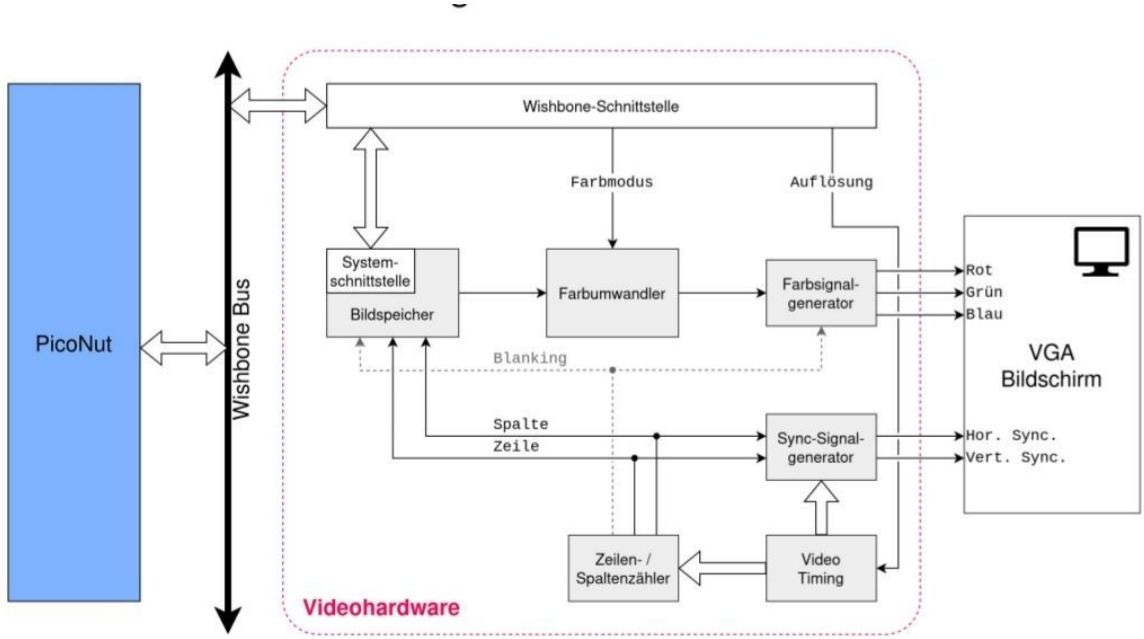

Figure 1: Design von Martin Erichsen 2025

2.1 Explanation of some modules

2.1.1 Wishbone graphic

The wb_graphics_module forms the central control unit of a modular VGA video system. It integrates several specialized submodules, including a Wishbone slave interface for configuration, a framebuffer source for pixel data, a color converter, VGA synchronization logic, and timing modules for generating the

necessary clock and control signals. This provides complete functionality for video output via VGA. The goal of the module is to connect various subsystems to form a functional graphics system that is configurable via the Wishbone bus, supports different resolutions and color modes, and can output a synchronized video signal via VGA. This allows flexible control of a display, for example, for embedded systems with graphics output. Communication between the modules takes place via dedicated internal sc_signal<> channels. These signals are linked using the connect_ports_to_signals() method. The init_submodule() method handles the instantiation and connection of the submodules. A central control function is implemented by the mux_fb_data_read() method. It decides which source provides the read pixel data depending on the context (video access or Wishbone access). This avoids conflicts when reading the framebuffer from different clock domains simultaneously. The wb_graphics_modul instantiates several submodules, each responsible for clearly defined tasks.

2.1.1.1 Wishbone Slave

A wishbone slave is a component in a digital system that uses the Wishbone bus standard – an open bus interface standard that is primarily used in FPGA and ASIC designs to connect modules such as processors, memory, and peripherals. Slave Responds to the master's commands, such as delivering data on a read or storing data on a write access. Actually, a wishbone slave is a component (e.g. a peripheral module such as UART, SPI, timer, memory block, etc.) that

processes requests from a wishbone master. The slave itself does not initiate communication. He only responds to inquiries that come from the master. The author can say that the entire project is based on this module, since it contains all the important **addresses** for **reading**, for example, as well as for writing, as well as the addresses for **status**, **color** and **image resolution**. this implementation has the purpose of **decoding the address** on the one hand, so that it recognizes whether an incoming address belongs to it, on the other hand, **reading and writing the data** so that in case of a read request, the slave delivers data; in the event of a write request, it stores the data received. The goal was also to provide the **status information** so that, for example, signals such as acknowledge (ACK) can be sent. [1]

2.1.1.2 Line/Colum Counter

The **Line/Colum Counter** module is part of a video signal processing system and implements a synchronous line and column counter to generate a raster scan pattern. It is used to derive the current position within a video frame based on a clock signal (e.g., pixel clock) and to indicate whether the current position is within the visible area of the frame. The **Line/Colum Counter** module is part of a video signal processing system and implements a synchronous line and column counter to generate a raster scan pattern. It is used to derive the current position within a video frame based on a clock signal (e.g., pixel clock) and to indicate whether the current position is within the visible area of the frame.

2.1.1.2.1 Behavior

The main functionality [2] is implemented in the counter_process() method:

- **Reset active**: When reset is asserted, both column and line counters are reset to zero. Output signals **vid_column**, **vid_line**, and **vid_enable** are also reset to 0 or false.
- **Counting enabled**: When enable is high: The column counter increments on each clock cycle. If the column reaches or exceeds **vid_column_end**, it is reset to 0, and the line counter is incremented.If the line counter also reaches **vid_line_end**, it resets to 0 as well.
- **Output update**: The current column and line values are written to **vid_column** and **vid_line**, respectively. The **vid_enable** signal is calculated as follows:

```
bool active = (column <= vid_column_active.read()) &&
              (Line < vid_line_active.read());
```

2.1.1.3 Sync Signal Generator

Provides all the necessary timing information for generating a VGA-compatible video signal. This includes both the visible image areas and the synchronization intervals. The current version supports VGA mode with a resolution of 640x480 pixels. The Overview function is primarily responsible for calculating

the video timing. Based on an input value (resolution_mode), the horizontal and vertical image dimensions, as well as the synchronization times, are calculated. A method, proc_comb_timings(), has been defined as an internal function. This method calculates all required timing values depending on the selected resolution mode. It is typically used as combinatorial logic in simulation or synthesis. The following values apply to the currently implemented mode, RESOLUTION_MODE_640x480 Table Table 1:

Table 1: I/O von vga_timing

Parameter	Meaning	Value
vid_column_active	Number of visible pixels per line	640
vid_column_end	Total number of pixels per line (incl. sync)	800
vid_line_active	Number of visible lines	480
vid_line_end	Total number of lines per frame	525
vid_hsync_begin	Start of horizontal sync pulse	663

Parameter	Meaning	Value
vid_hsync_end	End of horizontal sync pulse	759
vid_vsync_begin	Start of vertical sync pulse	490
vid_vsync_end	End of vertical sync pulse	492

2.2 Technical basics

2.2.1 Sofware, languages and Library used

2.2.1.1 Sofware and languages

For each module implementation, the module was implemented in SystemC [3], a C++-based language for modeling and simulating hardware at the transaction and register transfer levels. This system enables flexible, object-oriented modeling combined with the efficiency and portability of C++. This enables both functional simulations and precise timing analyses without relying on traditional HDLs such as VHDL or Verilog. GTKWave was used to visualize and analyze signal traces. This powerful open-source tool for displaying VCD/FSDB files allows for efficient tracing of signal behavior during simulation. The combination

of these tools enables a modern, software-centric approach to hardware development, suitable for both early verification and later synthesis preparation.

2.2.1.2 The SystemC Library

- **Purpose**: **#include <systemc.h>** includes the official SystemC library, which provides all the core constructs needed to write and simulate hardware modules in C++. [4]
- **Key features provided by systemc.h:** [4]
  - `SC_MODULE(...)` – Macro to define SystemC modules.
  - `sc_signal`, `sc_in`, `sc_out` – Signal types for communication.
  - `SC_METHOD`, `SC_THREAD`, `SC_CTOR` – Process and constructor macros.
  - `sc_start()` – Starts the simulation.

> **!** Hinweis
>
> Without systemc.h you cannot write a real SystemC-Modul

3 Description of the solution

3.1 Simulation

[5] For each module, three header files were created, namely a **.h**, **.cpp**, and **tb.cpp** file. The **.h** file (header file) is used to declare your module. This means

you describe here which ports, signals, processes, and constructors your module has. It tells the compiler what exists but does not contain a complete implementation yet. In this header file, for example, you write the **SC_MODULE** definition with its ports and methods. The **.cpp** file is then responsible for the implementation. Here, you define everything that you previously only declared in the header file. For example, you write the constructor of your module in the .cpp file, add **SC_METHOD** or **SC_THREAD** calls, and implement the functional contents of the processes. It answers the question: "*How does it work?*". The third important file is **tb.cpp**, the testbench file. In tb.cpp, you create an instance of your module (DUT, Design Under Test), connect it to signals, generate stimuli (i.e., inputs), check the outputs, and run the simulation using sc_start(). It is therefore used to test your module in a simulation and to verify whether it works as intended. Then a Makefile was implemented next to each tb.cpp file, meaning that for each tb.cpp file of a module in the project there was a corresponding Makefile. The Makefile is used in projects with tb.cpp files to automate the compilation and linking of your SystemC code. Normally, a long compilation command with g++ and all the SystemC library paths would need to be entered manually for each testbench (tb.cpp) and all associated .cpp and .h files. That would be error-prone and cumbersome.With a Makefile, these commands are written down once in a structured form. Then, the project or testbench can be easily compiled in the terminal with the command make. The Makefile ensures that all dependencies are automatically taken into account, so

for example, the .cpp file is also compiled when changes are made to it, and that the SystemC library is correctly linked. In projects, a Makefile is created for each tb.cpp file, so that the respective testbench can be built quickly and cleanly without having to type all the compiler options manually each time. It is an important tool for efficiency and error-free compilation, especially with many different tests and modules in SystemC.

3.2 Test and result

First, **make clean** is executed in the *Shell* to delete all previously compiled files and old outputs. This ensures that everything is rebuilt from scratch during the next compilation, without any old object files (e.g., .o) or binaries that could cause issues. After that, you run **make run-tb**. This command calls a target in the Makefile that compiles the testbench and then executes it directly. You will see the simulation outputs in the terminal.

As shown in the illustration, for example Figure 2.

If you also want to create a trace file for waveform analysis, you need to execute **make run-tb-trace** in the *Shell*. This target compiles and runs the testbench with VCD trace generation enabled. This results in a file, usually *trace.vcd*, where all signal transitions and events of your simulation are stored. Finally, you open this trace file with **gtkwave trace.vcd**. This launches GTKWave, a graphical tool for viewing and analyzing the signal waveforms of the simulation in a clear

```
(INFO): 0 s, line_column_counter_tb.cpp:85:   Reset and initialize signals
(INFO):          20 ns, line_column_counter_tb.cpp:98:   Start counting...
@30 ns col=1 line=0 enable=1
@40 ns col=2 line=0 enable=1
@50 ns col=3 line=0 enable=0
@60 ns col=4 line=0 enable=0
@70 ns col=0 line=1 enable=1
@80 ns col=1 line=1 enable=1
@90 ns col=2 line=1 enable=1
@100 ns col=3 line=1 enable=0
@110 ns col=4 line=1 enable=0
@120 ns col=0 line=2 enable=0
@130 ns col=1 line=2 enable=0
@140 ns col=2 line=2 enable=0
@150 ns col=3 line=2 enable=0
@160 ns col=4 line=2 enable=0
@170 ns col=0 line=3 enable=0
@180 ns col=1 line=3 enable=0
@190 ns col=2 line=3 enable=0
@200 ns col=3 line=3 enable=0
@210 ns col=4 line=3 enable=0
@220 ns col=0 line=0 enable=1
(INFO):          220 ns, line_column_counter_tb.cpp:120:   Simulation complete.
```

Figure 2: Output, for example, of the module line_column_counter on terminal after the command make run-tb

time diagram. This allows you to check whether the design actually does what is expected.

An example from Gtkwave shows this illustration Figure 3.

3.3 Problem and solution

3.3.1 Problem

When using make clean, make run-tb, and make run-tb-trace, various problems can occur. If make clean is executed, it is possible that the clean command in the Makefile has not been implemented correctly, so not all old object files or binaries are deleted. As a result, old compiled files might be included in the next compilation, leading to unexpected errors or incorrect behavior. Addition-

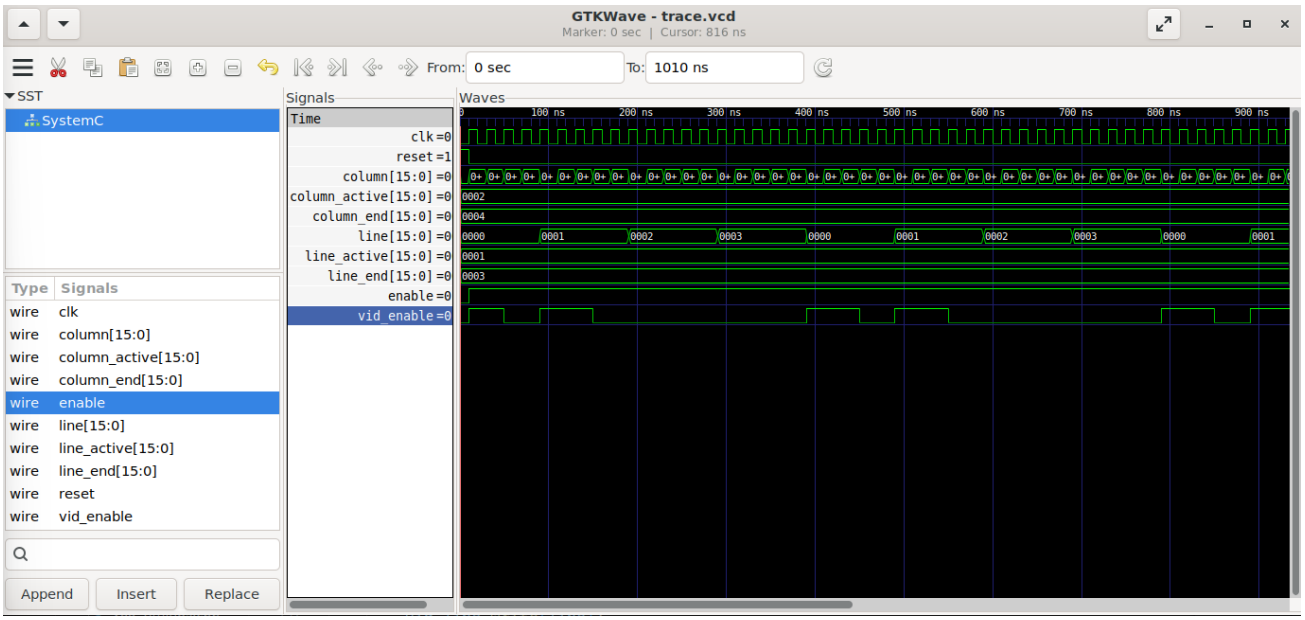

Figure 3: GTKwave output of the module line_column_counter as an example

ally, there may be insufficient write permissions in the project directory, causing the rm command to fail. During make run-tb, compiler errors often occur, for example if includes have been written incorrectly, header files are missing, or the namespace using namespace sc_core; has not been declared. Furthermore, if the SystemC library paths are not correctly set in the Makefile, the compiler cannot find the SystemC headers and stops with error messages such as “fatal error: systemc.h: No such file or directory”. Even if compilation succeeds, linker errors may still occur if -lsystemc is omitted or if the SystemC library is not linked correctly. In such cases, a message like “undefined reference to sc_main” appears. When make run-tb-trace is executed, problems may arise if the trace file has not been opened or closed correctly in the code. In this case, the file trace.vcd is not created, and when gtkwave trace.vcd is run afterwards, GTK-

Wave reports that the file does not exist or is empty. Moreover, the simulation can crash during tracing if, for example, invalid signal assignments are made or uninitialized pointers are used. This leads to a segmentation fault, causing the simulation to terminate immediately.

3.3.2 Solution

To resolve problems with make clean, it should be ensured that in the Makefile, within the clean target, all relevant files are deleted, especially .o files, executable files such as tb.x, as well as old trace.vcd files if applicable. Additionally, it should be checked whether write permissions are available in the project directory. If no trace file is generated when executing make run-tb-trace, the SystemC code should be checked to ensure that an sc_trace_file has been correctly opened, for example with sc_create_vcd_trace_file(“trace”). Furthermore, all required signals must be registered with sc_trace, and the trace file must be properly closed at the end using sc_close_vcd_trace_file, so that the file is created correctly.

4 Conclusion

4.1 Summary

As part of this project, a dedicated hardware solution for graphics output was developed and successfully integrated with an existing software simulation mod-

ule. The key achievement was the design of a module capable of converting its internal framebuffer into a stable, flicker-free VGA-compatible video signal for direct display on standard monitors. Various color formats and resolutions were implemented and tested, demonstrating the module's flexibility for future applications.

- **Main results include**:
  - Fully hardware-generated VGA output meeting timing specifications
  - Successful demonstration through a live animation on a VGA monitor
  - An extendable architecture designed to support future interfaces such as HDMI and DVI or to use main memory as framebuffer
- **The main insight gained** is that software-based simulation approaches can be effectively complemented by targeted hardware development

4.2 Outlook

Within the scope of this project, stable VGA output was successfully implemented in hardware; however, digital interfaces such as HDMI or DVI have not yet been implemented. In the future, the architecture could be extended to support these modern video outputs, enabling compatibility with a wider range of displays. Another meaningful next step would be the integration of the framebuffer into the system's main memory, allowing larger image data to be managed

efficiently and enabling more flexible dynamic content. Additionally, implementing features such as hardware scrolling, overlays, or direct GPU interfacing could further enhance graphics performance and expand the potential application areas of the system.

5 References

[1] Prof. Dr.-Ing. G. Kiefer, “”Bus-systeme”,” in *Entwurf of digital design 2, s. 265-273*, WS 2024/2025.

[2] Prof. Dr.-Ing. G. Kiefer, “”Bus-systeme”,” in *Entwurf of digital design 2, s. 279*, WS 2024/2025.

[3] Prof. Dr.-Ing. G. Kiefer, “”Bus-systeme”,” in *Entwurf of digital design 2, s. 274-314*, WS 2024/2025.

[4] A. S. Initiative, “SystemC grundlagen und konstrukte,” in *IEEE standard 1666-2011, language reference manual*, 2011.

[5] Prof. Dr.-Ing. G. Kiefer, “”Bus-systeme”,” in *Entwurf of digital design 2, s. 282-288*, WS 2024/2025.