Design and SoC Integration of an ICB-based Graphics Peripheral for the Hummingbird E203 RISC-V Core

Gallo Andrea
Student ID: 2359271
`andrea.gallo13@studio.unibo.it`

July 28, 2025

Abstract

This report details the design, firmware development, and verification of a custom graphics peripheral for a System-on-Chip (SoC) based on the Hummingbird E203 RISC-V core. The project's primary objective was to create a fully functional hardware module capable of rendering ASCII characters, received via a UART interface, onto a standard 640x480@60Hz HDMI display.

The system's operation is orchestrated by a dedicated C firmware running on the RISC-V core. This software "brain" implements a finite-state machine to manage application flow and utilizes a custom 32-bit command protocol to control the graphics hardware. The hardware, implemented in Verilog, acts as the "muscle" of the system and consists of a custom graphics pipeline. Key hardware modules include an ICB slave interface, a command adapter that manages a framebuffer stored in a Gowin Dual-Port Block RAM (DPBRAM), and a video signal generator. To address the limited on-chip memory of the target Gowin GW2A-18 FPGA, a character-based rendering approach with a monochrome framebuffer was adopted.

A cornerstone of this project is the rigorous, multi-tiered verification strategy, culminating in a full SoC hardware-software co-simulation. This final validation involved compiling the C firmware into an executable image using a Makefile, loading this image into the simulated SoC's boot memory, and verifying the entire end-to-end data path. The test successfully demonstrated the flow from a character transmitted by a testbench emulating a host PC, through processing by the on-chip firmware, to the correct rendering of pixels on the final video output. The successful implementation validates the complete co-design and provides a robust foundation for future enhancements.

Contents

Chapter 1

Introduction

1.1 Project Overview and Objectives

This report details the design, implementation, and verification of a custom High-Definition Multimedia Interface (HDMI) display module tailored for a RISC-V based system. The primary objective of this project is to develop a functional hardware module capable of rendering ASCII characters, received via a Universal Asynchronous Receiver-Transmitter (UART) interface, onto an external display.

The HDMI module is designed to be integrated with the Hummingbird E203 RISC-V core, a popular choice for embedded applications, by interfacing with its Inter-Chip Bus (ICB). The target deployment platform for this project is the Tang Primer 20k development board, equipped with a Gowin GW2A-18 Field-Programmable Gate Array (FPGA). This platform provides the necessary hardware resources for implementing and testing the video display capabilities.

Key functional requirements include the initial display of color strips upon power-up for a brief, fixed duration to serve as a visual diagnostic, followed by the rendering of monochrome (black and white) characters. These characters will represent information typically received through the UART at system startup, such as boot messages or diagnostic data. The system must support a minimum display resolution of 640x480 pixels at a 60Hz refresh rate.

1.2 The System-on-Chip (SoC) Context

The Hummingbird E203, which forms the core of our target system, is not merely a standalone processor but is typically part of a System-on-Chip (SoC) design. An SoC is an integrated circuit (IC) that combines multiple essential components of a complete electronic system onto a single silicon chip. This contrasts with traditional approaches where different functionalities (like CPU, memory controllers, peripherals) reside on separate chips on a printed circuit board (PCB).

A typical SoC, such as one built around the E203 core, would integrate:

- **A Central Processing Unit (CPU):** In our case, a RISC-V core like the Hummingbird E203, responsible for executing software instructions.
- **Memory Blocks:** Such as SRAM for data and instruction storage, and potentially interfaces to external DRAM.

- **Peripheral Interfaces:** Controllers for various input/output devices, including UART (for serial communication), SPI, I2C, GPIOs (General Purpose Input/Outputs), and, crucially for this project, buses for connecting custom peripherals like our HDMI module.

- **Interconnect Fabric:** A system of buses (like the ICB in the E203 environment) that allows these different components to communicate with each other efficiently.

- **Clocking and Power Management Units:** To manage system timing and power consumption.

The primary advantages of the SoC approach include reduced system size, lower power consumption, higher performance (due to shorter signal paths), and often lower manufacturing costs for high-volume production. Our HDMI display module is designed to function as a custom peripheral within such an SoC environment, leveraging the E203's processing capabilities and its ICB bus for integration. Understanding this SoC context is vital for appreciating the interface requirements and the overall system interactions discussed in this report.

1.3 Key Design Challenges and Constraints

The development of the HDMI display module, while aiming for comprehensive functionality, is subject to several inherent design challenges and hardware constraints imposed by the target FPGA platform. A primary constraint is the limited internal Static Random-Access Memory (SRAM) available on the Gowin GW2A-18 FPGA featured on the Tang Primer 20k board. The GW2A-18 offers approximately 100KB of total block SRAM.

Rendering a full-frame buffer for a 640x480 resolution display consumes a significant amount of memory. A color display, even with a modest 8-bit color depth (256 colors), would require $640 \times 480 \times 8$ bits $= 2,457,600$ bits, or 300KB of SRAM. This figure far exceeds the available resources of the FPGA. Therefore, a primary design constraint was to adopt a monochrome (1-bit per pixel) display. The resulting framebuffer, while still substantial, requires $640 \times 480 \times 1$ bit $= 307,200$ bits, which translates to 37.5KB of SRAM. This memory footprint, though manageable, influenced the overall architecture, reinforcing the decision to limit the display to monochrome text to ensure sufficient memory remains available for other system components and logic.

Further challenges include the precise generation of HDMI timing signals compliant with industry standards to ensure compatibility with various display monitors, and the efficient handling of data flow from the UART, through the RISC-V core and ICB bus, to the HDMI output logic. The design must also manage the initial color strip generation sequence before transitioning to character display mode.

1.4 Core Video Concepts: Digital Video Timing

The generation of a stable and coherent image on a digital display relies on a meticulously orchestrated sequence of timing signals. These signals govern the flow of pixel data from a video source (e.g., a processor like RISC-V) to a display device (e.g., an HDMI monitor), ensuring that each pixel is illuminated at the correct coordinate and at the precise moment. Without such synchronization, the visual output would be chaotic and uninterpretable.

This section lays the groundwork by explaining the core video timing signals and their interplay, which are foundational to the design of our HDMI display module.

At the heart of digital video transmission is the concept of *raster scanning*, a process where the image is painted on the screen pixel by pixel, line by line, from top to bottom [1]. Figure 1.1 provides a visual representation of this fundamental scanning pattern.

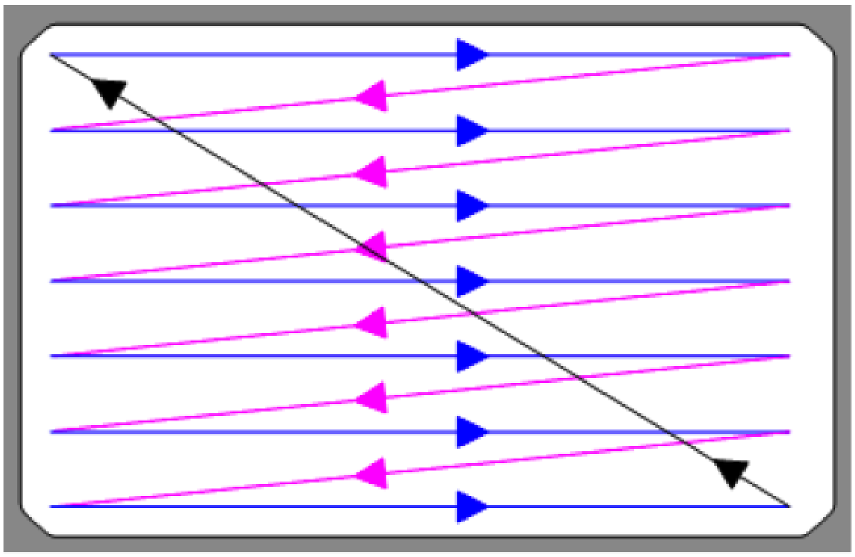


Figure 1.1: Illustration of the raster scanning process, showing the path taken to draw lines of pixels sequentially across and down the display area.

To manage this raster scan and the associated data flow, several key synchronization signals are employed:

- **Pixel Clock (PCLK):** Although not always explicitly transmitted as a separate *synchronization* signal in all interfaces (like HDMI which uses TMDS clocking), the pixel clock is the fundamental heartbeat that dictates the rate at which individual pixels are transmitted. Every active edge of this clock typically corresponds to one pixel's data.
- **Horizontal Synchronization (HSync):** This signal marks the end of an active line of pixels and the beginning of a horizontal blanking interval. The horizontal blanking interval, as detailed by Green [1], consists of three parts:
  - **Front Porch:** A brief period after the last active pixel of a line and before the HSync pulse.
  - **Sync Pulse:** The actual HSync pulse, which signals the display to return its horizontal scanning mechanism to the beginning of the next line.
  - **Back Porch:** A period after the HSync pulse and before the first active pixel of the next line. This allows time for the scanning mechanism to stabilize.
- **Vertical Synchronization (VSync):** Analogous to HSync but for an entire frame (or field in interlaced video), this signal indicates the end of the last active line of a frame and the start of a vertical blanking interval. The vertical blanking interval also comprises a front porch, the VSync pulse itself, and a back porch [1]. The VSync pulse instructs the display to return its scanning mechanism to the top-left corner for the next frame.

- **Data Enable (DE) or Active Video (AV):** This signal is crucial as it explicitly indicates when the data being transmitted corresponds to active, visible pixels on the screen. When DE is asserted (typically high), the display interprets the incoming data as pixel information to be rendered. When DE is de-asserted (low), the data being transmitted occurs during the blanking intervals (horizontal or vertical) and is typically ignored by the display for rendering purposes [1].

The precise durations of these porches, sync pulses, and active video periods are defined by industry standards such as those from VESA (Video Electronics Standards Association) or CEA (Consumer Electronics Association, e.g., CEA-861 for HDMI). These timings are specific to the desired resolution and refresh rate, for instance, the 640x480 @ 60Hz target for this project. Our HDMI module must generate these signals with strict adherence to such timings to ensure compatibility and a correct display output. An illustrative diagram of these video timing components is presented in Figure 1.2.

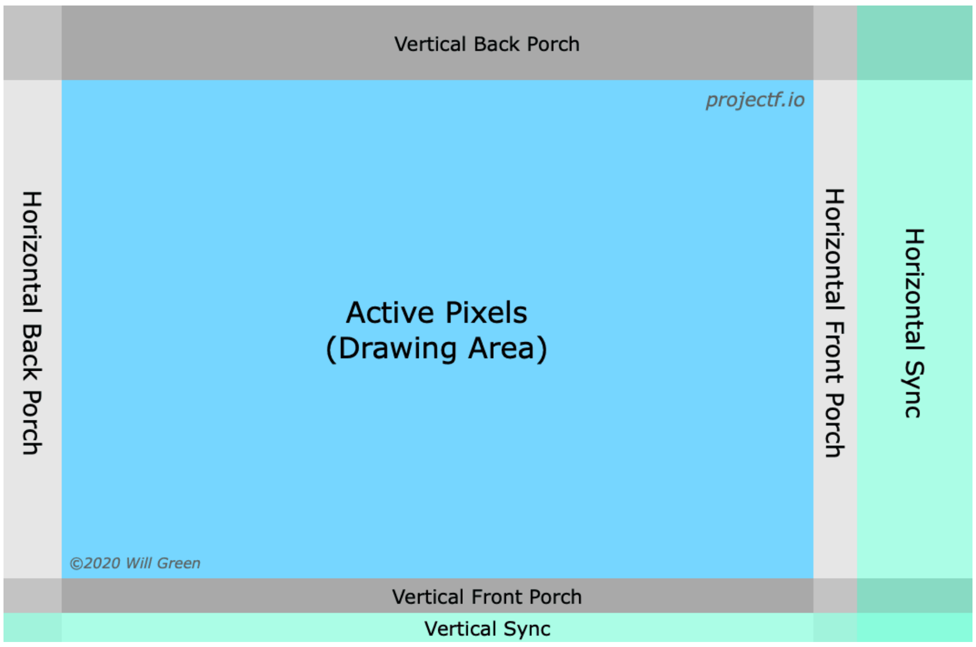


Figure 1.2: Conceptual Diagram of Video Timing Signals including Active Pixels and Blanking Intervals (Front Porch, Sync Pulse, Back Porch) for both Horizontal and Vertical Synchronization. Inspired by [1].

1.5 Report Structure

The remainder of this report is organized to provide a comprehensive overview of the HDMI display module project. Chapter 2 describes the overall system architecture, detailing the main components and their interconnections, including the interfaces between the RISC-V core, the ICB bus, and the custom HDMI logic. Chapter 3 provides a detailed explanation of the implementation of key hardware modules, covering the timing generator, data path logic, and the ICB peripheral interface. Chapter 4 presents the firmware design responsible for UART communication and character-to-bitmap conversion. Chapter 5 discusses the simulation setup, methodology, and presents key waveforms to verify the functionality of individual modules and the integrated system. Chapter 6 outlines the testing methods and processes employed on the FPGA platform. Finally, Chapter 7 offers a discussion of the results, challenges encountered, and potential future work, followed by concluding remarks. References cited throughout the report are listed at the end.

Chapter 2

System Architecture and Interfaces

2.1 Overall System Architecture

The designed HDMI display system is centered around the Hummingbird E203 RISC-V processor core, which acts as the main controller. Our custom display logic is interfaced with the core through a custom `ICB_bus` module. This module handles transactions from the E203 core and outputs a 32-bit command payload on the `o_cpu_cmd` signal. This signal serves as the primary data path to our top-level IP, named `GPU_top`, where it is received as the `i_cpu_cmd` input.

Input ASCII characters are received from an external source, such as a host PC, through a standard UART interface integrated within the E203 SoC environment. The CPU executes firmware to process each character by sending its 16 bitmap rows one by one. For each row, it constructs a 32-bit command word: the lower 16 bits contain the pixel data for that row, while the upper 16 bits are reserved for control information. Specifically, bits `[19:16]` encode the row offset (0-15) within the character cell, while the most significant bits `[31:29]` are used for special commands (e.g., Reset, Delete, Enter). This command word is then sent to the `ICB_bus` module, which forwards it to the `GPU_top` via the command signal chain.

Within the main IP block, `GPU_top`, the `dpram_adapter` module receives the 32-bit `i_cpu_cmd` payload. This module is not a simple splitter, but a sophisticated controller. It decodes the command word, extracts the 16-bit pixel data from the lower half, and uses the control information from the upper half to calculate the precise write address in the DPBRam. This design allows for advanced features like cursor control and special commands (e.g., Delete, Enter).

The final `hdmi` module reads the 16-bit pixel data from the DPBRam as needed. It is responsible for generating all necessary video timing signals (HSync, VSync) and the Pixel Clock. The overall data flow is thus: UART input → CPU processing (creates 32-bit data+control word per row) → `ICB_bus` output (`o_cpu_cmd`) → `GPU_top` input (`i_cpu_cmd`) → `dpram_adapter` (decodes command, extracts data) → DPBRam storage (16-bit pixel rows) → `hdmi` block (retrieves data, generates video) → HDMI Video Output.

A high-level block diagram of this system architecture is depicted in Figure 2.1.

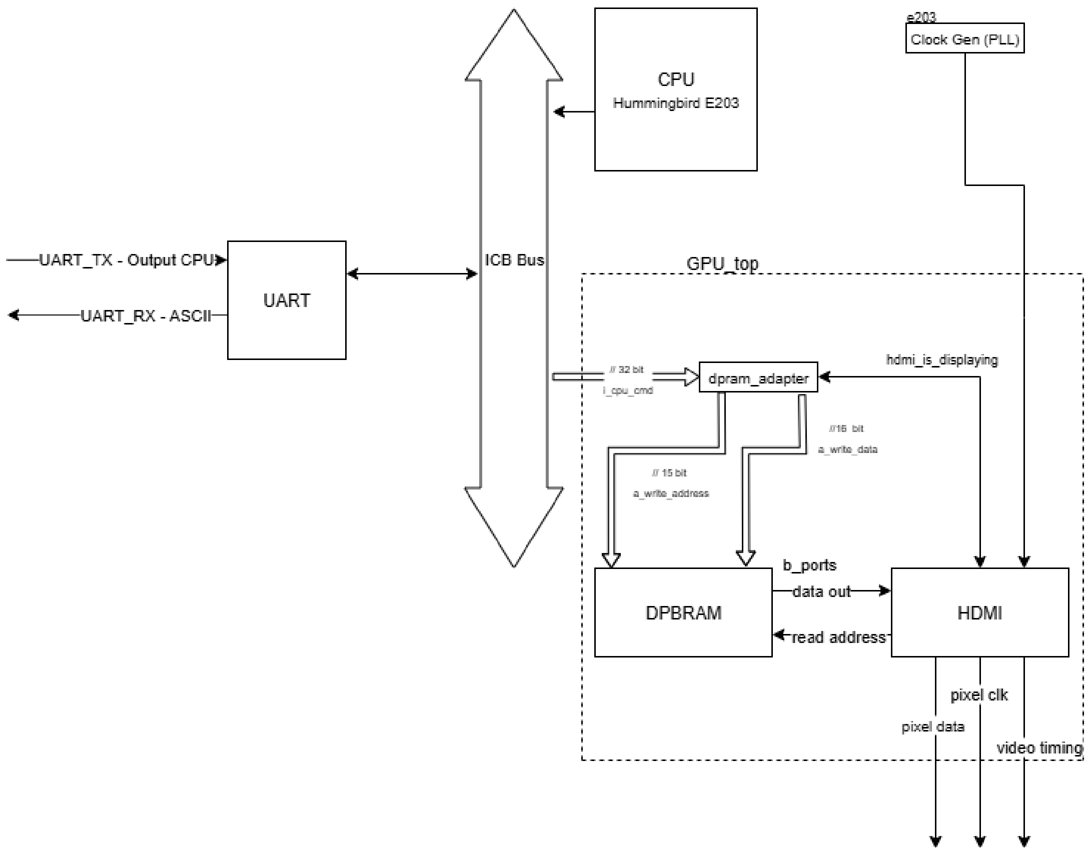


Figure 2.1: High-Level System Block Diagram illustrating the key components and data paths of the display system.

2.2 System Interfaces

The functionality of the system relies on a set of well-defined interfaces that manage the communication between the E203 core, our custom hardware, and external devices. This section provides a high-level overview of these key interfaces. The detailed hardware implementation of each module's logic will be discussed in Chapter 3.

2.2.1 Hummingbird E203 Core and ICB Interface

The heart of the processing capability is the Hummingbird E203 RISC-V core. Communication between the core and our custom peripheral, `GPU_top`, is mediated by the `ICB_bus` module, which implements a slave interface for the E203's native Inter-Chip Bus (ICB). The ICB is a synchronous, address-mapped bus. Our firmware uses ICB write transactions to send 32-bit command words to a specific memory-mapped address corresponding to our peripheral. The `ICB_bus` module is responsible for capturing this data and making it available to the rest of our hardware.

2.2.2 Memory Interface (DPRAM)

A Dual-Port Block RAM (DPBRAM) serves as the video framebuffer, storing the 16-bit pixel data for character rows. This memory has two independent ports, which is critical for system performance:

- **Write Port (Port A):** This port is exclusively controlled by the `dpram_adapter` module, which writes new character data received from the CPU.
- **Read Port (Port B):** This port is exclusively controlled by the `hdmi` module, which continuously reads data from the framebuffer in sync with the video raster scan to generate the display output.

This dual-port architecture effectively decouples the CPU's asynchronous updates from the `hdmi` module's real-time, continuous read requirements. To further enhance visual stability and prevent artifacts such as screen tearing, a synchronization mechanism is implemented. Writes to the framebuffer are gated and only permitted during the video signal's blanking intervals, ensuring that the memory content is not modified while it is being actively read for display.

2.2.3 Video Output Interface

The final interface is the video output itself, generated by the `hdmi` module. This interface consists of the standard synchronization signals (`o_gpu_hsync`, `o_gpu_vsync`) and a 24-bit parallel RGB data stream (`o_gpu_red`, `o_gpu_green`, `o_gpu_blue`). These signals are generated with precise timing for the target resolution and are ready to be fed into a downstream TMDS encoder for transmission over an HDMI cable.

2.2.4 UART Interface

The UART interface provides the primary channel for user interaction, allowing ASCII characters to be sent from a host computer to the E203 core. The firmware polls the UART's registers (via the ICB) to detect and read incoming characters. Once a character is received, it is processed by the firmware and its corresponding bitmap data is sent to the display hardware, thus completing the data path from external input to video output.

Chapter 3

Firmware Design and Control Logic

The hardware graphics pipeline, while powerful, is inert without intelligent direction. This chapter details the design of the firmware, the software "brain" of the system, which runs on the Hummingbird E203 RISC-V core. Written entirely in C, the firmware is responsible for interpreting user input, managing the application's lifecycle, and translating high-level requests into precise, low-level commands that orchestrate the custom GPU hardware. The following sections will describe the communication protocol established between the CPU and the hardware, the state-machine-based application flow, and the core routines that handle character rendering.

3.1 The CPU-Hardware Command Protocol

A robust and efficient communication protocol is the foundation of the hardware-software interface. All interactions are funneled through a single, 32-bit memory-mapped register within the GPU peripheral, accessible at a fixed address. The firmware constructs specific 32-bit command words for each transaction, where different bits or bit-fields are allocated for control flags or data payloads. This unified approach allows for both high-level control and granular data transfer through one interface.

The protocol defines three primary types of commands:

- **Control Commands:** These commands instruct the hardware to perform high-level actions related to cursor management. They are encoded using the three most significant bits of the 32-bit command word, ensuring they do not conflict with data-carrying commands. The defined flags are:
  - `GPU_CMD_FLAG_RESET_CURSOR (1U « 31)`: Instructs the hardware to move the cursor to its origin position (top-left of the screen).
  - `GPU_CMD_FLAG_BACKSPACE (1U « 30)`: Instructs the hardware to move the internal cursor position back by one character.
  - `GPU_CMD_FLAG_NEWLINE (1U « 29)`: Instructs the hardware to move the cursor to the beginning of the next line.
- **Data Commands:** These commands carry the pixel data for character rendering. The 32-bit word is partitioned into two fields:

  - **Row Index (Bits [31:16]):** A 16-bit field encoding the vertical row index (from 0 to 15) of the pixel data within a character's 16x16 bitmap. This tells the hardware which row of the character is being sent.
  - **Pixel Data (Bits [15:0]):** A 16-bit field containing the monochrome pixel data for that specific row.

- **Terminator Command:** A special-case command with a value of all zeros (`0x00000000`) is used to signal the end of a character transmission. When the hardware receives this command, it knows that all 16 rows of the previous character have been sent and advances its internal character position counter, preparing for the next character.

This protocol design is highly efficient. The low-level task of writing a command to the hardware register is encapsulated in the `GPU_WriteCommand` function, which abstracts the memory-mapped I/O operation using platform-specific macros (e.g., `MY_PERIPH_REG`). This allows the main application logic to focus on ***what*** to send, not ***how*** to send it.

3.2 Application Flow and State-Machine Logic

To manage the system's operational lifecycle in an orderly and robust manner, the firmware implements a Finite-State Machine (FSM). This design choice avoids complex, nested conditional logic and results in a clean, maintainable, and highly predictable application flow. The FSM is elegantly realized using function pointers in C, where each state is represented by a dedicated function.

A global function pointer, `g_CurrentStateHandler`, holds the address of the function corresponding to the current state. The main application loop is exceptionally simple: it continuously calls the function pointed to by this variable. All state transition logic is encapsulated within the state functions themselves; a state transition is achieved by simply assigning the address of the next state's function to `g_CurrentStateHandler`.

```c
// Defines a pointer to a function that represents a state in the FSM.
typedef void (*FSM_StateHandler)(void);

// Global variable holding the current state.
FSM_StateHandler g_CurrentStateHandler = FSM_State_Startup;

// Main application loop continuously executes the current state.
while (1) {
    g_CurrentStateHandler();
}
```

Listing 3.1: The core FSM implementation using a global function pointer.

The system progresses through four distinct states:

1. **FSM_State_Startup:** This is the entry-point state upon system boot. Its sole responsibility is to display a predefined startup message on the screen. Once the message is fully rendered, it transitions the system to the next state, `FSM_State_WaitForHost`. This provides immediate visual feedback that the GPU and firmware are operational.

2. **FSM_State_WaitForHost:** In this state, the system idly polls the UART's status register, waiting for the first character to be sent from the host. This signifies that the user is ready to interact. Upon receiving any character (which is immediately consumed and discarded), it triggers a transition to `FSM_State_ClearScreen`, preparing the display for the interactive session.

3. **FSM_State_ClearScreen:** This transient state prepares the screen for terminal mode. It first issues a `GPU_CMD_FLAG_RESET_CURSOR` command to move the hardware cursor to the origin. It then overwrites the startup message area by rendering a sequence of space characters. Finally, it issues another reset cursor command to ensure the terminal starts with the cursor at the top-left corner, and then transitions to the final, persistent state: `FSM_State_Terminal`.

4. **FSM_State_Terminal:** This is the main operational state where the system functions as an interactive terminal. It continuously polls the UART for new characters. Received characters are processed based on their ASCII value: standard printable characters are rendered on the screen, while control codes like Carriage Return, Line Feed, Backspace, and Delete are translated into the corresponding high-level hardware commands. This state is persistent and handles all further user interaction.

3.3 Core Character Rendering Routines

The firmware's primary responsibility is to translate abstract characters into a stream of hardware-understandable commands. This task is handled by a set of core helper functions that encapsulate the rendering logic and the details of the command protocol.

At the heart of this process is the `GPU_DisplayChar` function. This routine is responsible for the entire lifecycle of rendering a single character, from ASCII code validation to the final hardware handshake. The function's logic is straightforward and robust, as shown in Listing 3.2.

```c
void GPU_DisplayChar(uint8_t ascii_code) {
    // 1. Validate input: fall back to a '?' for non-printable characters.
    if (ascii_code < ASCII_PRINTABLE_START || ascii_code > ASCII_PRINTABLE_END) {
        ascii_code = ASCII_FALLBACK_CHAR;
    }
    // 2. Calculate the index into the font data array.
    uint16_t bitmap_index = ascii_code - ASCII_PRINTABLE_START;

    // 3. Iterate through all 16 rows of the character's bitmap.
    for (uint16_t i = 0; i < FONT_HEIGHT_PIXELS; i++) {
        // 4. Construct the 32-bit data command.
        uint32_t command = ((uint32_t)i << 16) | (uint32_t)font_bitmap[bitmap_index][i];
        GPU_WriteCommand(command);
    }
    // 5. Send the terminator command to signal completion.
    GPU_WriteCommand(GPU_CMD_END_OF_CHAR);
}
```

Listing 3.2: The core character rendering function.

The process involves five distinct steps: first, it sanitizes the input by mapping any non-printable ASCII codes to a fallback character ('?'). Second, it computes the correct index into the `font_bitmap` array. Third, it enters a loop that iterates through each of the 16 pixel rows of the character. Inside the loop, it constructs the 32-bit data command by packing the row index and the corresponding 16-bit pixel data, and sends it to the hardware. Finally, after the loop completes, it sends the crucial `GPU_CMD_END_OF_CHAR` command to notify the hardware that the character transmission is complete and the internal cursor can be advanced.

The bitmap font itself, `font_bitmap`, is a large, constant two-dimensional array embedded directly in the firmware. It stores the 16x16 pixel patterns for all 96 printable ASCII characters, from space (ASCII 32) to DEL (ASCII 127).

Handling of special commands, such as those generated by the Enter or Backspace keys, is managed within the `FSM_State_Terminal`. For instance, the backspace sequence is implemented with a three-step hardware interaction:

1. A `GPU_CMD_FLAG_BACKSPACE` command is sent to move the hardware cursor back one position.

2. The `GPU_DisplayChar` function is called with a space character (' ') to visually erase the previous character from the framebuffer.

3. A final `GPU_CMD_FLAG_BACKSPACE` command is sent to reposition the cursor correctly over the newly cleared space, ready for the next input.

This modular approach, separating low-level command sending from high-level state logic and character rendering, forms the backbone of a robust and easily extensible firmware design.

Chapter 4

Hardware Implementation of the Graphics Pipeline

This chapter details the Verilog implementation of the custom GPU's hardware components. The graphics pipeline is designed as a series of specialized modules that work in concert to receive commands, manage memory, and generate a standard video signal. Each module is described following the logical flow of data through the system: starting from the interface that receives commands from the CPU, moving to the controller that processes these commands and writes to the framebuffer, and concluding with the generator that reads from the framebuffer to produce the final HDMI output. For each module, we will discuss its specific role, its internal logic, and the results of its unit-level verification.

4.1 The ICB Slave Interface: Bridging CPU and GPU

The initial point of hardware interaction between the RISC-V CPU and the custom graphics peripheral is the `ICB_bus` module. This component serves a critical role as a protocol-compliant slave on the Hummingbird E203's native Inter-Chip Bus (ICB). Its primary function is to continuously monitor bus transactions, recognize those targeting its uniquely assigned peripheral address, and manage the complete handshake protocol for both read and write operations. In essence, it acts as a robust and reliable hardware gateway, abstracting the complexities of bus-level communication from the core logic of the graphics pipeline.

Address Decoding and Transaction Identification

The module's core logic is predicated on its ability to respond exclusively to transactions intended for it. This is achieved through a hardcoded local parameter, `PERIPHERAL_ADDRESS`, set to the specific value `32'h10014004`. A purely combinational logic block continuously compares the incoming `GPU_icb_cmd_addr` signal with this parameter.

```verilog
// The unique memory-mapped address for this peripheral.
localparam PERIPHERAL_ADDRESS = 32'h10014004;

// 1. Check if the incoming transaction's address matches our own.
wire is_our_address = (GPU_icb_cmd_addr == PERIPHERAL_ADDRESS);

// 2. A write is enabled only if the transaction is valid, is a write
//    (cmd_read is low), and matches our address.
wire write_enable   = GPU_icb_cmd_valid && !GPU_icb_cmd_read &&
    is_our_address;

// 3. A read is enabled only if the transaction is valid, is a read
//    (cmd_read is high), and matches our address.
wire read_enable    = GPU_icb_cmd_valid &&  GPU_icb_cmd_read &&
    is_our_address;
```

Listing 4.1: Combinational logic for address and command decoding.

This comparison generates the `is_our_address` signal, which becomes the primary condition for any further action. This signal is then combined with the main `GPU_icb_cmd_valid` signal and the read/write indicator (`GPU_icb_cmd_read`) to produce two internal control strobes: `write_enable` and `read_enable`. These strobes are active for a single clock cycle and trigger the sequential logic to perform the requested operation.

Data Latching and Stable Command Output

A crucial design decision in this module is the method used to provide a stable command to the downstream GPU logic. Bus transactions are inherently transient, but the GPU core requires a persistent command signal to operate on. The `ICB_bus` module solves this by using a 32-bit internal register, `gpu_command_register`.

The sequential logic is designed such that this register latches the value from the `GPU_icb_cmd_wdata` bus *only* on the clock cycle where a valid write operation is enabled. At all other times, it holds its previous value. This register's output is then continuously assigned to the module's primary output, `o_cpu_cmd`. The result is a stable, persistent command signal that only changes when the CPU explicitly performs a new write, effectively decoupling the bus timing from the operational timing of the main GPU pipeline.

ICB Handshake Protocol Management

The module fully implements the ICB slave-side handshake protocol for both command and response channels.

- **Command Channel Handshake:** The slave's readiness to accept a command is signaled by the `GPU_icb_cmd_ready` output. In this design, the module is always able to process a transaction in a single cycle. Therefore, this signal is driven directly by the `is_our_address` wire. This is an efficient implementation that ensures the module only signals readiness when it is actually being addressed, preventing it from stalling the bus during transactions intended for other peripherals.
- **Response Channel Handshake:** After a read or write operation is completed, the slave must notify the master. This is handled by the `GPU_icb_rsp_valid` signal. The sequential logic generates a single-cycle pulse on this signal in the clock cycle immediately following a successful transaction (`write_enable` or `read_enable`). For read operations, the data from `gpu_command_register` is first transferred to a dedicated `read_data_buffer`, which in turn drives the `GPU_icb_rsp_rdata` bus, ensuring the correct data is available to the master when the response is flagged as valid.

This complete and correct implementation of the handshake protocol ensures seamless and reliable communication with the CPU master, forming a solid foundation for the entire system.

4.1.1 Unit-Level Verification and Waveform Analysis

To ensure the reliability of the `ICB_bus` module, a comprehensive unit-level testbench, `testbench_ICB_bus`, was developed. Its primary role is to act as an ideal ICB master, rigorously exercising the Device Under Test (DUT) to confirm its adherence to the bus protocol. The testbench is structured to be self-checking, automating the verification process through a sequence of four key test vectors:

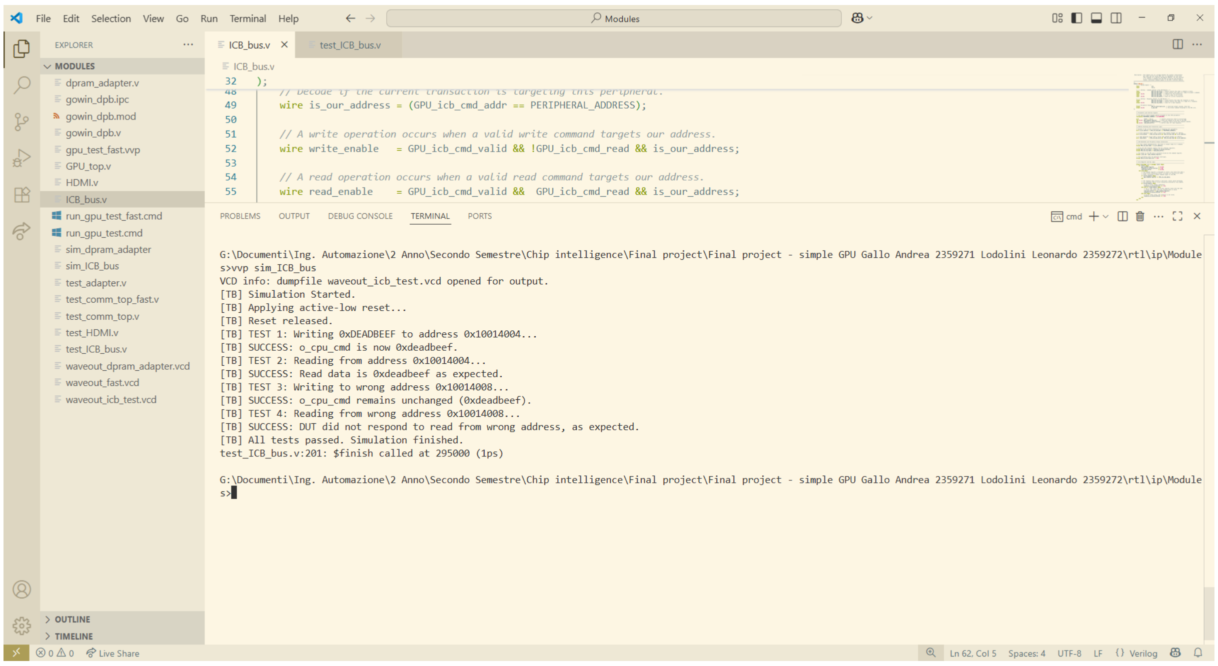
1. **System Reset:** An initial active-low reset is applied to ensure all internal registers are initialized to a known, stable state.

2. **Register Write:** A write transaction is performed to the DUT's specific peripheral address, sending a known data pattern (`0xDEADBEEF`).

3. **Register Read:** A subsequent read transaction targets the same address to verify that the data was stored correctly and can be retrieved.

4. **Incorrect Address Access:** Both write and read transactions are directed to an invalid address to confirm that the DUT's address decoding logic correctly ignores them.

The testbench provides immediate feedback on the simulation's outcome via the console. As shown in Figure 4.1, the terminal output explicitly confirms the successful completion of all four test stages, providing the first layer of validation that the module is functionally correct.

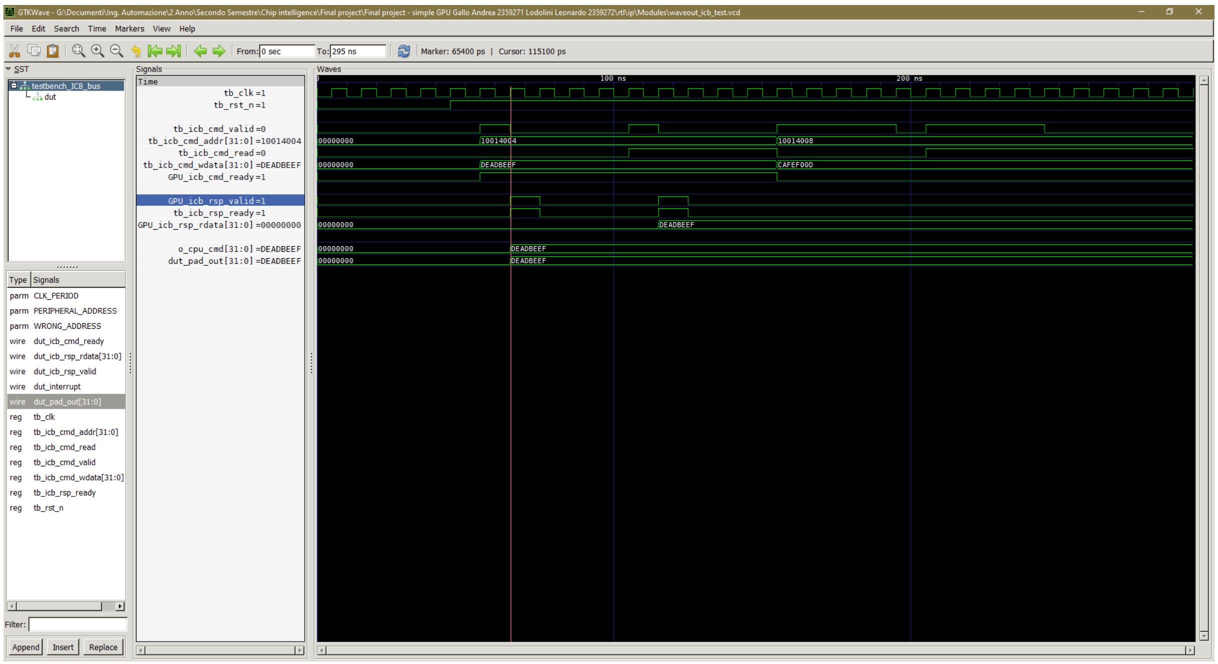


```
    );
    // Decode if the current transaction is targeting this peripheral.
    wire is_our_address = (GPU_icb_cmd_addr == PERIPHERAL_ADDRESS);

    // A write operation occurs when a valid write command targets our address.
    wire write_enable   = GPU_icb_cmd_valid && !GPU_icb_cmd_read && is_our_address;

    // A read operation occurs when a valid read command targets our address.
    wire read_enable    = GPU_icb_cmd_valid &&  GPU_icb_cmd_read && is_our_address;
```

```
G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Module
s>vvp sim_ICB_bus
VCD info: dumpfile waveout_icb_test.vcd opened for output.
[TB] Simulation Started.
[TB] Applying active-low reset...
[TB] Reset released.
[TB] TEST 1: Writing 0xDEADBEEF to address 0x10014004...
[TB] SUCCESS: o_cpu_cmd is now 0xdeadbeef.
[TB] TEST 2: Reading from address 0x10014004...
[TB] SUCCESS: Read data is 0xdeadbeef as expected.
[TB] TEST 3: Writing to wrong address 0x10014008...
[TB] SUCCESS: o_cpu_cmd remains unchanged (0xdeadbeef).
[TB] TEST 4: Reading from wrong address 0x10014008...
[TB] SUCCESS: DUT did not respond to read from wrong address, as expected.
[TB] All tests passed. Simulation finished.
test_ICB_bus.v:201: $finish called at 295000 (1ps)

G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Module
s>
```

Figure 4.1: The terminal log from the Icarus Verilog simulation. The output clearly reports the success of each test case, confirming that the module passed all self-checking assertions.

A deeper, cycle-by-cycle analysis is made possible by examining the simulation waveforms. Figure 4.2 visualizes the behavior of the key bus signals during the successful write and read transactions.


GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_icb_test.vcd

From: 0 sec To: 295 ns Marker: 65400 ps | Cursor: 115100 ps

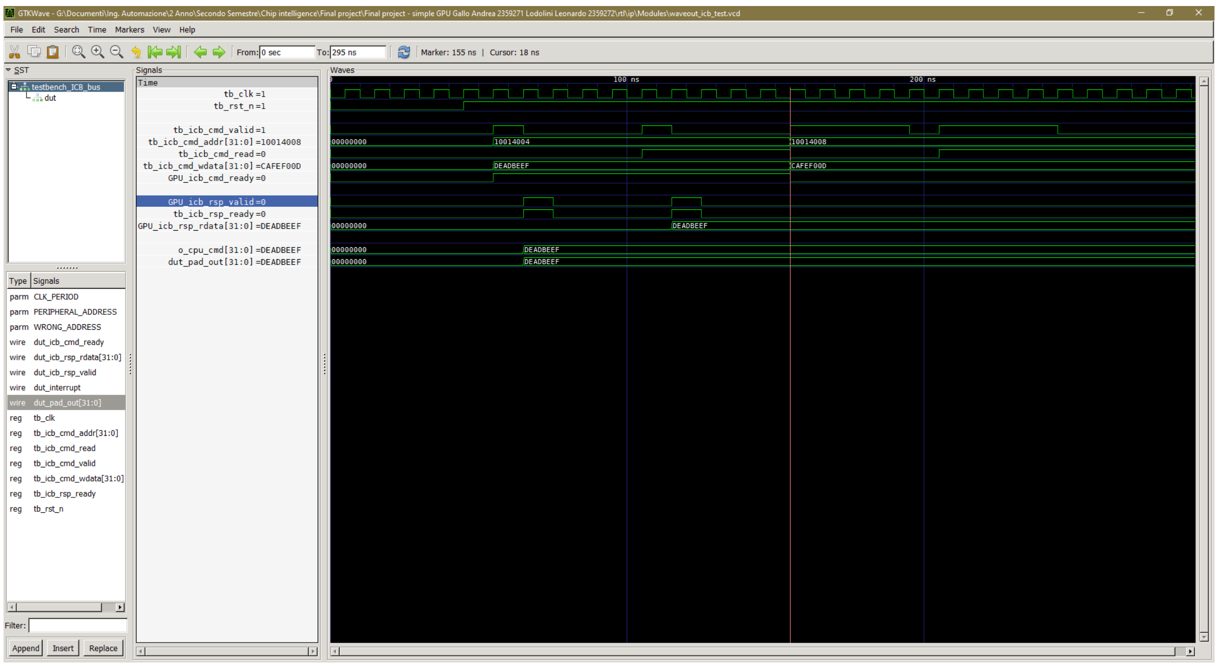
```
tb_clk=1
tb_rst_n=1

tb_icb_cmd_valid=0
tb_icb_cmd_addr[31:0]=10014004
tb_icb_cmd_read=0
tb_icb_cmd_wdata[31:0]=DEADBEEF
GPU_icb_cmd_ready=1

GPU_icb_rsp_valid=1
tb_icb_rsp_ready=1
GPU_icb_rsp_rdata[31:0]=00000000

o_cpu_cmd[31:0]=DEADBEEF
dut_pad_out[31:0]=DEADBEEF
```

Type	Signals
parm	CLK_PERIOD
parm	PERIPHERAL_ADDRESS
parm	WRONG_ADDRESS
wire	dut_icb_cmd_ready
wire	dut_icb_rsp_rdata[31:0]
wire	dut_icb_rsp_valid
wire	dut_interrupt
wire	dut_pad_out[31:0]
reg	tb_clk
reg	tb_icb_cmd_addr[31:0]
reg	tb_icb_cmd_read
reg	tb_icb_cmd_valid
reg	tb_icb_cmd_wdata[31:0]
reg	tb_icb_rsp_ready
reg	tb_rst_n

Figure 4.2: Waveform detailing successful write and read operations to the correct address (`0x10014004`). The handshake signals and data buses behave exactly as specified by the protocol.

The waveform clearly illustrates the ICB handshake protocol. During the write phase, the testbench asserts `tb_icb_cmd_valid`, and the DUT responds with `dut_icb_cmd_ready`. The data `0xDEADBEEF` is captured by the DUT, and its internal register, visible on `dut_pad_out`, is updated accordingly. The subsequent read operation shows the same data being correctly placed on the `dut_icb_rsp_rdata` bus.

Furthermore, Figure 4.3 provides visual proof of the address decoder's robustness. When the testbench drives a transaction to an incorrect address, the DUT correctly keeps `dut_icb_cmd_ready` de-asserted, ignoring the request and protecting its internal state.



```
tb_clk=1
tb_rst_n=1

tb_icb_cmd_valid=1
tb_icb_cmd_addr[31:0]=10014008
tb_icb_cmd_read=0
tb_icb_cmd_wdata[31:0]=CAFEF00D
GPU_icb_cmd_ready=0

GPU_icb_rsp_valid=0
tb_icb_rsp_ready=0
GPU_icb_rsp_rdata[31:0]=DEADBEEF

o_cpu_cmd[31:0]=DEADBEEF
dut_pad_out[31:0]=DEADBEEF
```

Figure 4.3: Waveform demonstrating the DUT's correct response to an invalid address access. The peripheral does not acknowledge the transaction, leaving its output state unchanged.

The combination of a successful pass report from the self-checking testbench and a detailed visual confirmation from the waveform analysis provides high confidence in the `ICB_bus` module's correctness and its readiness for system-level integration.

4.2 The Command Adapter and Framebuffer Logic

The `dpram_adapter` module represents the primary intelligent component within the GPU's write datapath. Its role extends beyond simple signal routing; it functions as a protocol translator, converting the high-level, 32-bit command words from the firmware into the granular address, data, and write-enable signals required by the Dual-Port RAM framebuffer. This module effectively bridges the abstract software domain with the physical memory implementation, handling critical logic for cursor management, command interpretation, and dynamic address calculation.

The internal architecture is built upon a synchronous sequential circuit that combines two distinct control strategies for maximum efficiency and responsiveness: a direct, prioritized handling for special commands and a Finite-State Machine (FSM) for standard data flow.

Prioritized Command Handling

The design recognizes that certain operations, such as cursor reset or newline, must be executed instantaneously and unconditionally. To achieve this, the three most significant bits of the `i_cpu_cmd` input are decoded as high-priority flags. As seen in the main `always` block, these flags are checked on every rising edge of the clock *before* the FSM logic is evaluated.

- **Synchronous Reset (`i_cpu_cmd[31]`):** If this flag is asserted, the module immediately forces its internal state and registers to their default values. The `character_position` register is reset to zero, and the FSM is returned to its `S_IDLE` state. This provides the firmware with a powerful mechanism to reset the cursor to the screen's origin (0,0) with a single command.
- **Delete Flag (`i_cpu_cmd[30]`):** Asserting this flag sets the internal `delete_mode_active` register. This flag modifies the behavior of the subsequent end-of-character processing in the FSM, preventing the cursor from advancing. This is a key part of the backspace implementation.
- **Enter/Newline Flag (`i_cpu_cmd[29]`):** This flag triggers an immediate recalculation of the `character_position` register. The logic `(character_position / SCREEN_WIDTH + 1) * SCREEN_WIDTH` correctly computes the index of the first character on the line immediately following the current one, effectively moving the cursor to the start of the next row.

This prioritized approach ensures that critical control operations are never delayed by the state of the standard character-writing process.

FSM-based Character Data Processing

For normal character rendering, the module relies on a compact three-state FSM. This machine orchestrates the sequential process of writing the 16 rows of a single character's bitmap into the framebuffer.

- **S_IDLE:** The default state. The machine remains here until it detects the first valid row of a new character, which is identified by a non-zero value in the upper 16 bits of `i_cpu_cmd`. This serves as the trigger to begin a character-writing cycle, transitioning the state to `S_RUN`.
- **S_RUN:** The main operational state for data transfer. While in this state, the module processes each incoming data command, and the combinational logic generates the corresponding write signals for the DPRAM. The FSM continuously monitors the command input, waiting for the special "end-of-character" command (a word where the upper 16 bits are zero). Upon detecting this terminator, it performs one final action: if `delete_mode_active` is not set, it increments the `character_position` register. It then transitions to `S_POST_INCREMENT`.
- **S_POST_INCREMENT:** This is a transient but critical "debounce" state. Its purpose is to prevent the `character_position` from being incremented multiple

times if the firmware holds the end-of-character command on the bus for more than one clock cycle. The FSM waits in this state until a new, non-zero data command appears, at which point it safely transitions back to `S_RUN` to process the next character.

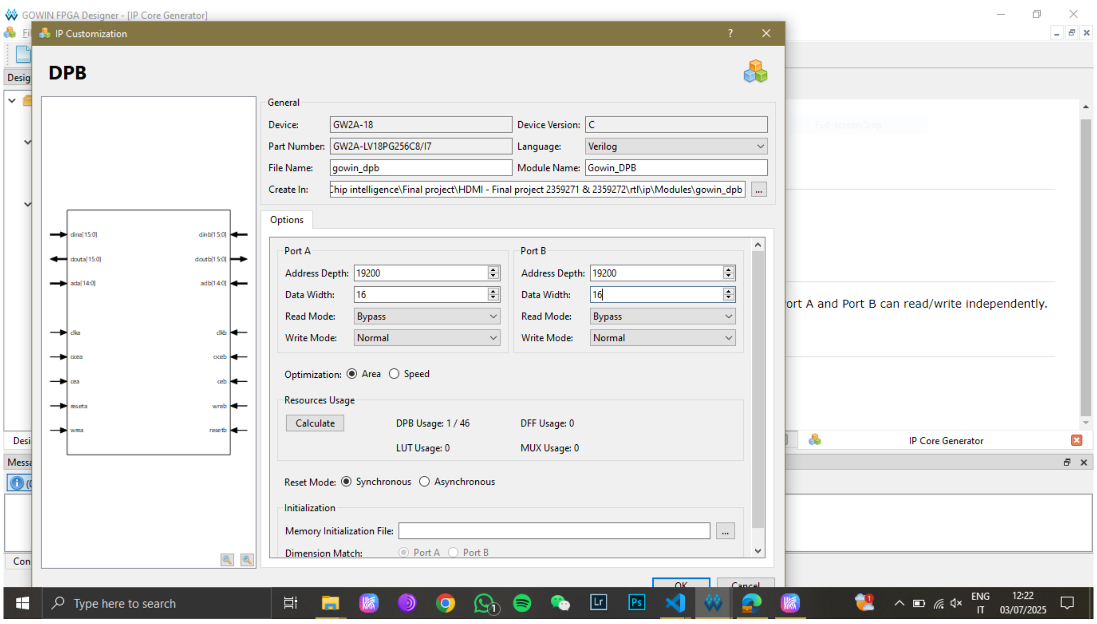
Combinational Output Generation

The final output signals for the DPRAM are generated by a purely combinational block that depends on the current state registers and the input command. The write address (`o_dpram_addr`) is calculated by multiplying the `character_position` by 16 (to get the base address of the 16x16 character cell) and adding the 4-bit row offset from the command (`i_cpu_cmd[19:16]`). The write data (`o_dpram_wdata`) is simply the lower 16 bits of the command. Crucially, the `write_enable` signal is carefully gated: it is only asserted for valid data commands, and is disabled for any high-priority flag commands or the end-of-character terminator, preventing erroneous writes to the framebuffer.

4.2.1 The Role of the Gowin Dual-Port BRAM

The video framebuffer is the most critical memory resource in the system. The choice of which memory primitive to use was a key design decision, driven by the need to resolve a fundamental conflict in the system architecture: the CPU needs to write new character data to the framebuffer sporadically and in bursts, while the HDMI video generator needs to read from it continuously and at a fixed, real-time rate.

A standard single-port RAM would be unable to service these two masters simultaneously, leading to contention, visual artifacts like screen tearing, and a complex arbitration logic requirement. The solution was to instantiate a Dual-Port Block RAM (DPBRAM) directly from the Gowin FPGA vendor's IP Core library. As shown in Figure 4.4, a true Dual-Port RAM was configured.



General

Device: GW2A-18 | Device Version: C
Part Number: GW2A-LV18PG256C8/I7 | Language: Verilog
File Name: gowin_dpb | Module Name: Gowin_DPB
Create In: Chip intelligence\Final project\HDMI - Final project 2359271 & 2359272\rtl\ip\Modules\gowin_dpb

Options

	Port A		Port B
Address Depth:	19200	Address Depth:	19200
Data Width:	16	Data Width:	16
Read Mode:	Bypass	Read Mode:	Bypass
Write Mode:	Normal	Write Mode:	Normal

Optimization: [✓ Area] Speed

Resources Usage
DPB Usage: 1 / 46 DFF Usage: 0
LUT Usage: 0 MUX Usage: 0

Reset Mode: [✓ Synchronous] Asynchronous

Initialization
Memory Initialization File:
Dimension Match: Port A Port B

ort A and Port B can read/write independently.

ENG IT 12:22 03/07/2025

Figure 4.4: Configuration of the Dual-Port Block RAM primitive using the Gowin IP Core Generator. Both Port A and Port B are configured for independent read/write access.

This primitive provides two entirely independent access ports (Port A and Port B), each with its own address, data, clock, and control signals. This architectural choice elegantly solves the contention problem:

- **Port A (The Write Port)** is connected exclusively to the outputs of the `dpram_adapter` module. It receives the asynchronous write commands generated in response to firmware instructions.
- **Port B (The Read Port)** is connected exclusively to the video signal generator module. It handles the continuous, synchronous read requests required to scan the framebuffer and generate the display.

By using a dedicated hardware resource provided by the FPGA vendor, the system achieves a clean separation of concerns, simplifies the design, and guarantees artifact-free visual performance by allowing simultaneous read and write operations without contention. The configured memory depth of 19200 words and data width of 16 bits are sufficient to store the entire 640x480 monochrome framebuffer, organized as character rows.

4.2.2 Unit-Level Verification and Waveform Analysis

The complex state-based logic of the `dpram_adapter` requires thorough verification to ensure its correctness under all operational conditions. A dedicated testbench, `testbench_adapter`, was created to simulate realistic command sequences and validate the module's behavior. The testbench simulates the CPU by driving the `i_cpu_cmd` input of the DUT. To simplify the stimulus, it uses a helper task, `write_character_unrolled`, which emulates the 17-cycle process of sending a full character's data followed by the end-of-character command.

To gain direct visibility into the DUT's internal state without altering its port list, the testbench employs a hierarchical reference to probe the `character_position` register. This technique is invaluable for precisely verifying the cursor management logic. The main test sequence validates three critical scenarios:

1. **Standard Character Write:** The testbench first writes a standard character, simulating the transmission of its 16 data rows. As shown in the initial phase of the simulation in Figure 4.5, the `o_dpram_addr` output correctly sweeps through the 16 memory locations corresponding to the first character cell. After the final end-of-character command, the internal `character_position` register correctly increments from 0 to 1.

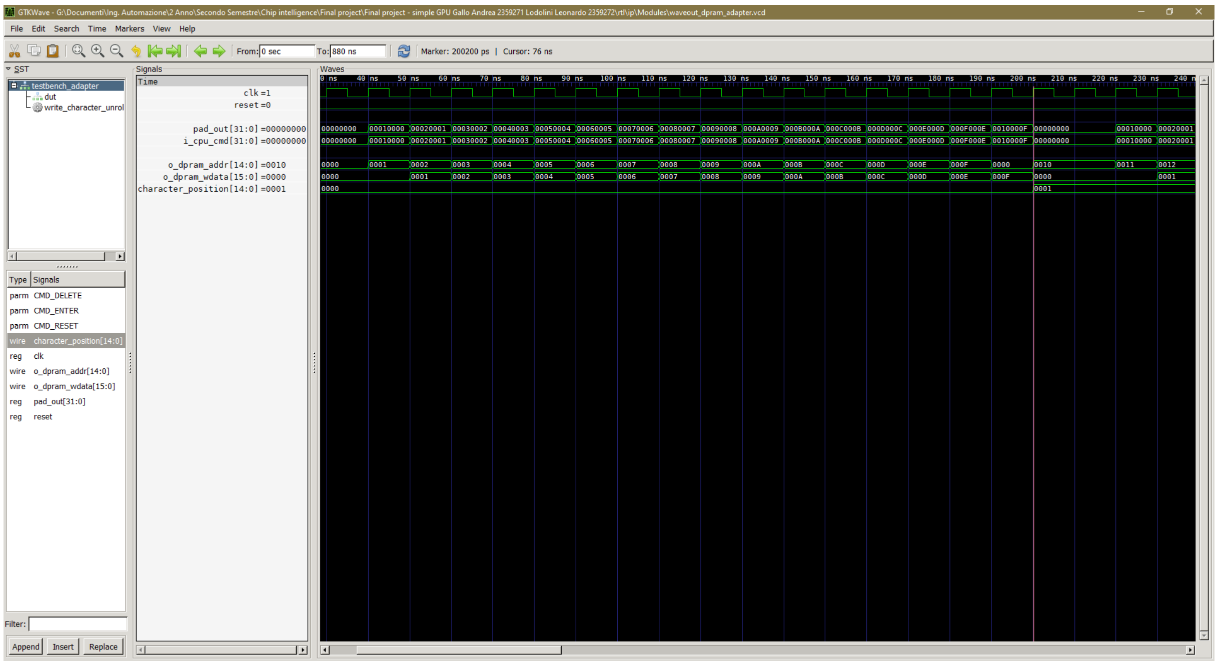

clk =1
reset =0
pad_out[31:0] =00000000
i_cpu_cmd[31:0] =00000000
o_dpram_addr[14:0] =0010
o_dpram_wdata[15:0] =0000
character_position[14:0] =0001
Marker: 200200 ps | Cursor: 76 ns

Figure 4.5: Waveform of a standard character write. As the 16 data rows are sent, the output address increments. The internal cursor position advances from 0 to 1 only after the end-of-character command is processed.

2. **Delete-and-Overwrite Operation:** The testbench then simulates a backspace operation by issuing a `CMD_DELETE` command, followed by writing a new character. The waveform in Figure 4.6 illustrates this sequence. After a character 'B' moves the `character_position` to 2, the `CMD_DELETE` command activates the module's delete mode. When the next character, 'C', is written, the write addresses correctly start from the base address of character position 1, effectively overwriting 'B'. The cursor then increments back to 2, confirming the non-incrementing behavior of the delete logic.

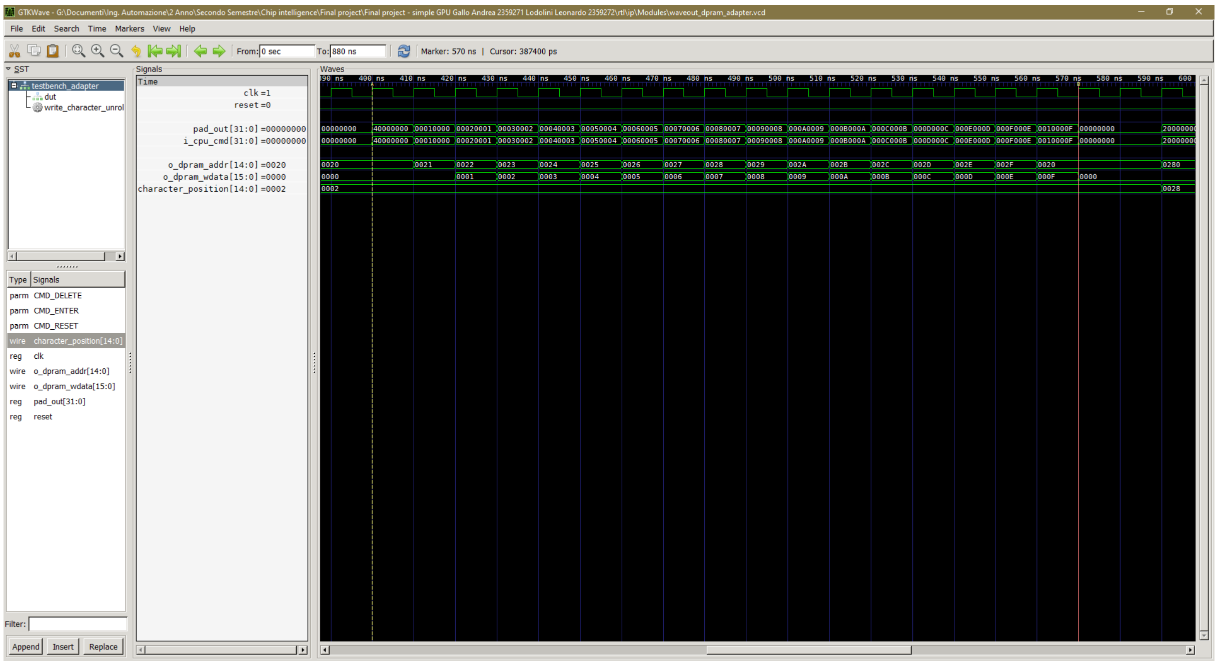

GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_dpram_adapter.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 880 ns
Marker: 570 ns | Cursor: 387400 ps
SST
testbench_adapter
dut
write_character_unrol
Type Signals
parm CMD_DELETE
parm CMD_ENTER
parm CMD_RESET
wire character_position[14:0]
reg clk
wire o_dpram_addr[14:0]
wire o_dpram_wdata[15:0]
reg pad_out[31:0]
reg reset
Filter:
Append Insert Replace
Signals
Time
clk =1
reset =0
pad_out[31:0] =00000000
i_cpu_cmd[31:0] =00000000
o_dpram_addr[14:0] =0020
o_dpram_wdata[15:0] =0000
character_position[14:0] =0002
Waves
390 ns 400 ns 410 ns 420 ns 430 ns 440 ns 450 ns 460 ns 470 ns 480 ns 490 ns 500 ns 510 ns 520 ns 530 ns 540 ns 550 ns 560 ns 570 ns 580 ns 590 ns 600
00000000 40000000 00010000 00020001 00030002 00040003 00050004 00060005 00070006 00080007 00090008 000A0009 000B000A 000C000B 000D000C 000E000D 000F000E 0010000F 00000000 20000000
00000000 40000000 00010000 00020001 00030002 00040003 00050004 00060005 00070006 00080007 00090008 000A0009 000B000A 000C000B 000D000C 000E000D 000F000E 0010000F 00000000 20000000
0020 0021 0022 0023 0024 0025 0026 0027 0028 0029 002A 002B 002C 002D 002E 002F 0020 0280
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F 0000
0002 0028

Figure 4.6: Waveform illustrating the delete-and-overwrite sequence. The `CMD_DELETE` command is issued, causing the next character write to start at the previous character's location, and the final cursor position confirms the correct logic.

3. **Enter (Newline) Command:** Finally, the test validates the immediate effect of the newline command. As shown in Figure 4.7, with the cursor at position 2, a single `CMD_ENTER` command is issued. The `character_position` register immediately jumps to 40 (0x28), which is the calculated starting position of the next line on a 40-character-wide screen. This confirms that the high-priority command logic works correctly and instantly.

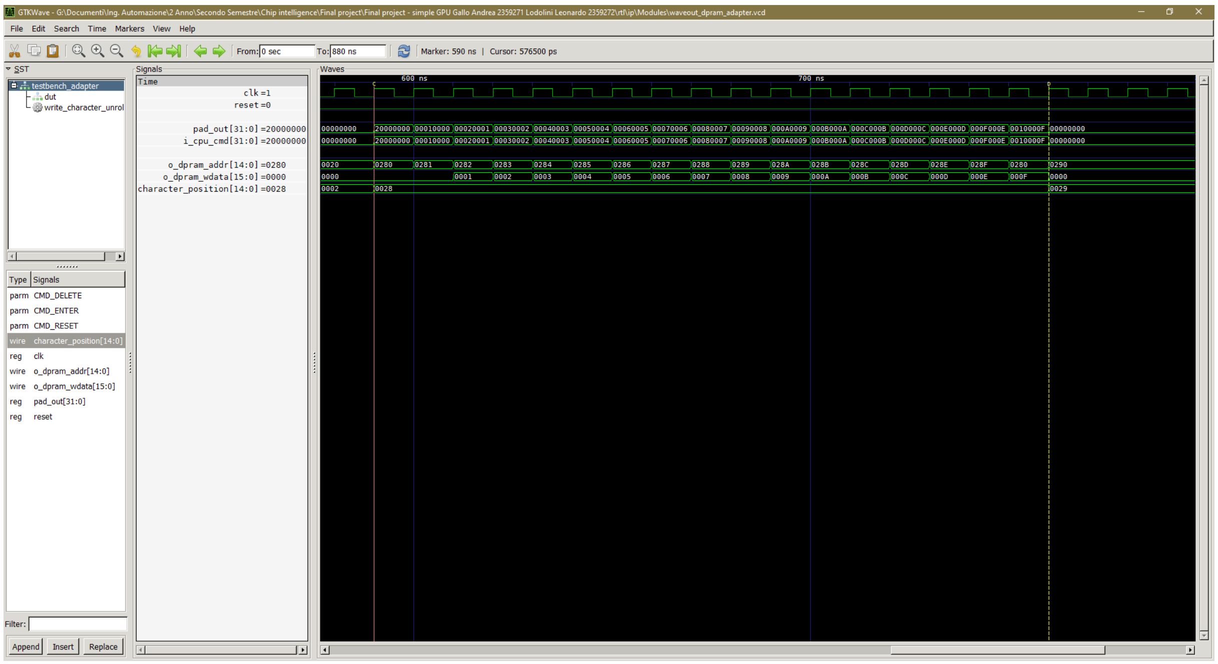

GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_dpram_adapter.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 880 ns
Marker: 590 ns | Cursor: 576500 ps
SST
testbench_adapter
dut
write_character_unrol
Type Signals
parm CMD_DELETE
parm CMD_ENTER
parm CMD_RESET
wire character_position[14:0]
reg clk
wire o_dpram_addr[14:0]
wire o_dpram_wdata[15:0]
reg pad_out[31:0]
reg reset
Filter:
Append Insert Replace
Signals
Time
clk =1
reset =0
pad_out[31:0] =20000000
i_cpu_cmd[31:0] =20000000
o_dpram_addr[14:0] =0280
o_dpram_wdata[15:0] =0000
character_position[14:0] =0028
Waves
600 ns
700 ns
00000000 20000000 00010000 00020001 00030002 00040003 00050004 00060005 00070006 00080007 00090008 000A0009 000B000A 000C000B 000D000C 000E000D 000F000E 0010000F 00000000
00000000 20000000 00010000 00020001 00030002 00040003 00050004 00060005 00070006 00080007 00090008 000A0009 000B000A 000C000B 000D000C 000E000D 000F000E 0010000F 00000000
0020 0280 0281 0282 0283 0284 0285 0286 0287 0288 0289 028A 028B 028C 028D 028E 028F 0280 0290
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F 0000
0002 0028 0029

Figure 4.7: Waveform showing the effect of an Enter command. The `CMD_ENTER` command causes an immediate jump in the internal `character_position` register to 40 (0x28), correctly moving the logical cursor to the start of the next line.

The successful execution of these structured test scenarios provides high confidence that the `dpram_adapter`'s FSM, priority command handling, and address generation logic are all implemented correctly.

4.3 The HDMI Video Signal Generator

The `HDMI` module is the final and most critical stage in the hardware graphics pipeline. It is responsible for the demanding real-time task of generating a stable, 640x480@60Hz VGA-compatible video signal. This module acts as the "read master" for the framebuffer, continuously scanning the DPRAM and converting the stored character data into a serial stream of colored pixels, precisely synchronized with the horizontal and vertical timing signals required by a display monitor.

A key feature of this module is its dual-mode operation, designed to provide both diagnostic feedback and functional output:

1. **Color Bar Mode:** Upon system reset, the module enters a diagnostic mode, displaying a standard color bar test pattern for a fixed duration. This serves as an immediate visual confirmation that the clocking, timing generation, and output stages are functional, independent of the rest of the system.

2. **Text Mode:** After the initial diagnostic phase, the module permanently transitions to its primary operational mode. In this mode, it actively reads the monochrome character data from the DPRAM framebuffer and renders it on screen.

The implementation of these functions is partitioned into several distinct logical blocks, which are described in the following subsections.

VGA Timing and Position Generation

The fundamental heartbeat of the module is a pair of free-running counters, `h_pos_counter` and `v_pos_counter`. These registers implement the core raster scan, with `h_pos_counter` incrementing on every clock cycle to sweep across a line and `v_pos_counter` incrementing once per line to sweep down the screen.

The timing parameters that define the frame—such as the active display dimensions (`H_DISPLAY`, `V_DISPLAY`), sync pulse widths, and front/back porch durations—are based on the VESA 640x480@60Hz standard. However, these values have been carefully adjusted, as noted in the code, to compensate for a system clock frequency that is not an exact multiple of the standard 25.175 MHz pixel clock. This pragmatic tuning is essential for maintaining a refresh rate close to 60 Hz and ensuring compatibility with a wide range of monitors.

Based on the instantaneous values of these counters, the module generates the active-low horizontal (`o_hsync`) and vertical (`o_vsync`) synchronization signals, as well as an internal `is_in_active_area` flag that functions as a Data Enable (DE) signal.

Dual-Mode Operation and Output Control

The transition between the initial color bar mode and the final text mode is controlled by a simple but effective timer, `mode_switch_timer`. Upon reset, the module starts in color bar mode. The timer increments until it reaches a predefined duration (`COLOR_BAR_DURATION_CYCLES`). Once this threshold is passed, a flag, `text_display_mode`, is permanently set to '1', switching the module's output logic into text rendering mode.

- **In Color Bar Mode,** a dedicated set of counters (`colorbar_pixel_counter`, `colorbar_strip_counter`) divides the active display area into eight vertical strips of equal width. A combinational `case` statement then maps the current strip index to one of eight predefined 24-bit color values.
- **In Text Mode,** the module's output is driven by the framebuffer reading logic, described below.

A final multiplexer selects the data source for the `final_pixel_data` output bus based on the state of the `text_display_mode` flag.

DPRAM Read-Path Logic and Text Rendering

This is the most complex part of the module's logic, responsible for translating the current pixel coordinates on screen into a specific bit read from the DPRAM. This is achieved through a multi-stage process:

1. **Character Coordinate Calculation:** A set of four dedicated counters (`screen_char_row/col_c` and `char_pixel_row/col_counter`) works in unison to determine, for any given pixel coordinate from the main timing generators, which character on the screen is being drawn and which pixel within that character's 16x16 bitmap is currently under the raster beam.
2. **DPRAM Address Generation:** The final read address for the DPRAM (`pixel_addr_out`) is calculated combinationally from the outputs of these character coordinate counters. The address is computed as:

`(char_row * SCREEN_WIDTH + char_col) * 16 + pixel_row_in_char`. This formula correctly maps the 2D character coordinates and the intra-character pixel row into a linear framebuffer address.

3. **Pipelined Pixel Serialization:** The DPRAM, being a synchronous block RAM, has a one-cycle read latency. This means the data read from the memory at cycle *N* corresponds to the address that was provided at cycle *N-1*. To handle this, the module uses a pipelined version of the horizontal pixel counter, `char_pixel_col_delayed`. This delayed signal is used to select the correct bit from the 16-bit `pixel_data_in` word that has just arrived from the DPRAM. The selection logic, `pixel_data_in[15 - char_pixel_col_delayed]`, correctly extracts the single bit corresponding to the current pixel, accounting for the font data being stored with the leftmost pixel in the Most Significant Bit (MSB).

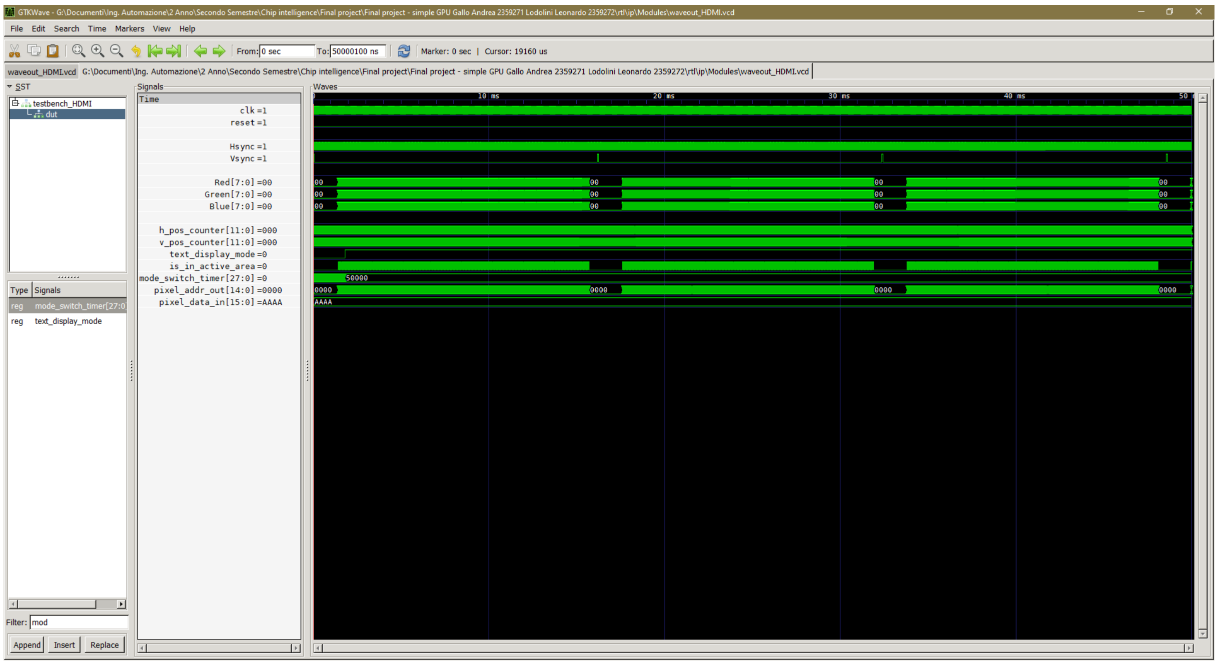
4. **Color Mapping:** The resulting 1-bit signal, `pixel_is_on`, is then mapped to a 24-bit monochrome color (white for '1', black for '0') and fed to the final output multiplexer.

This sophisticated, pipelined read-path logic ensures that the character data stored in memory is accurately translated and rendered at the correct position on the screen, pixel by pixel.

4.3.1 Unit-Level Verification and Waveform Analysis

Given the data-intensive and real-time nature of the `HDMI` module, its verification was approached through direct observation of its output signals. A minimalist testbench, `testbench_HDMI`, was created with the sole purpose of instantiating the module and logging its behavior to a VCD waveform file. This method allows for a detailed, visual inspection of the generated video timings and pixel data across multiple video frames. The testbench ties the `pixel_data_in` input to a static pattern (`16'hAAAA`) to provide a consistent, predictable stimulus for the text rendering logic.

The analysis of the generated waveforms confirms that all aspects of the module's complex logic function correctly.


GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_HDMI.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 50000100 ns
Marker: 0 sec | Cursor: 19160 us
waveout_HDMI.vcd
SST
testbench_HDMI
dut
Signals
Waves
10 ms
20 ms
30 ms
40 ms
50
Filter: mod
Append
Insert
Replace

```
Time
                 clk =1
               reset =1

               Hsync =1
               Vsync =1

          Red[7:0] =00
        Green[7:0] =00
         Blue[7:0] =00

  h_pos_counter[11:0] =000
  v_pos_counter[11:0] =000
    text_display_mode =0
     is_in_active_area =0
mode_switch_timer[27:0] =0
  pixel_addr_out[14:0] =0000
   pixel_data_in[15:0] =AAAA
```

Type	Signals
reg	mode_switch_timer[27:0
reg	text_display_mode

Figure 4.8: A wide view of the simulation showing multiple video frames. The periodic nature of the Vsync signal and the stable, continuous progression of the `h_pos_counter` and `v_pos_counter` demonstrate the overall stability of the timing generator.

As seen in the macroscopic view in Figure 4.8, the timing counters and synchronization signals are generated continuously and periodically, which is the essential foundation for a stable video output. The analysis can be broken down into the three key operational phases:

1. **Color Bar Mode:** During the initial phase immediately following reset, the module correctly operates in color bar mode. Figure 4.9 provides a zoomed-in view where the RGB output signals cycle through different predefined color values as the horizontal position counter advances. This confirms that the diagnostic test pattern generation and the initial state of the mode control logic are functional.

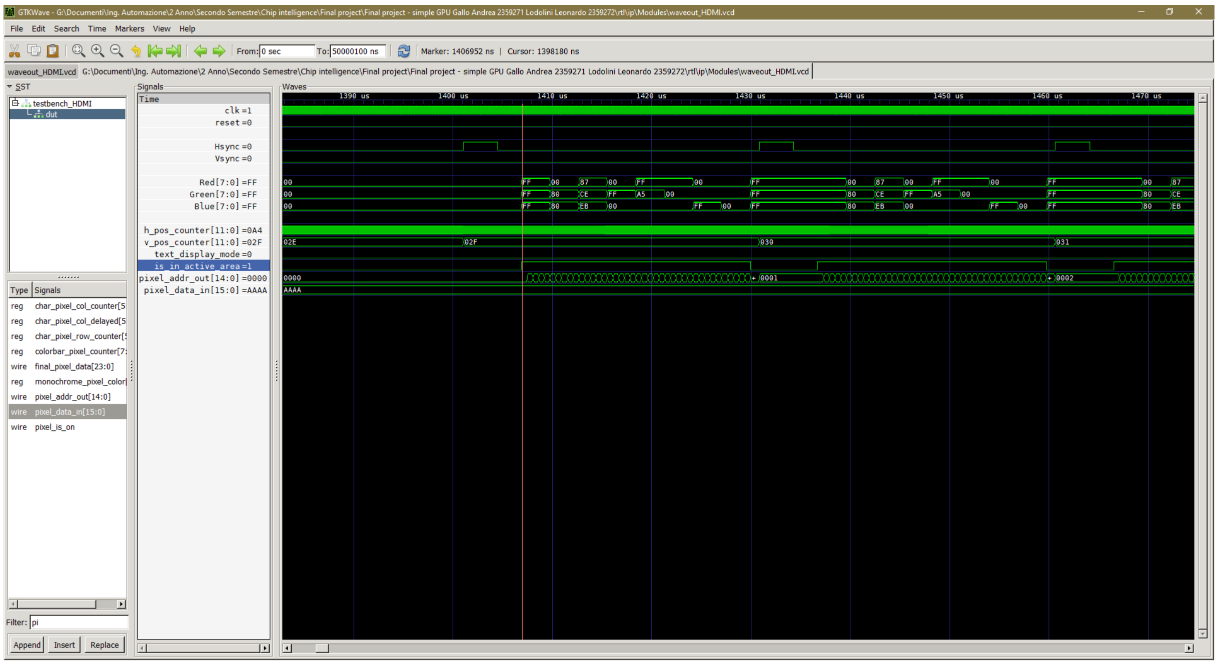

```
GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_HDMI.vcd
From: 0 sec   To: 50000100 ns   Marker: 1406952 ns | Cursor: 1398180 ns

SST: testbench_HDMI / dut
Type  Signals
reg   char_pixel_col_counter[5
reg   char_pixel_col_delayed[5
reg   char_pixel_row_counter[5
reg   colorbar_pixel_counter[7:
wire  final_pixel_data[23:0]
reg   monochrome_pixel_color[
wire  pixel_addr_out[14:0]
wire  pixel_data_in[15:0]
wire  pixel_is_on
Filter: pi

Signals
                clk =1
              reset =0
              Hsync =0
              Vsync =0
          Red[7:0] =FF
        Green[7:0] =FF
         Blue[7:0] =FF
 h_pos_counter[11:0] =0A4
 v_pos_counter[11:0] =02F
   text_display_mode =0
   is_in_active_area =1
pixel_addr_out[14:0] =0000
 pixel_data_in[15:0] =AAAA
```

Figure 4.9: A detailed view of the initial color bar mode. The RGB outputs change based on the horizontal screen position, generating the distinct vertical color strips. The `text_display_mode` flag is low during this phase.

2. **Mode Transition:** The waveform clearly shows the precise moment the internal `mode_switch_timer` expires. As illustrated in Figure 4.10, as soon as the timer reaches its target value, the `text_display_mode` flag is asserted high. The RGB output immediately switches from rendering the colored strips to rendering data based on the `pixel_data_in` input, as expected.

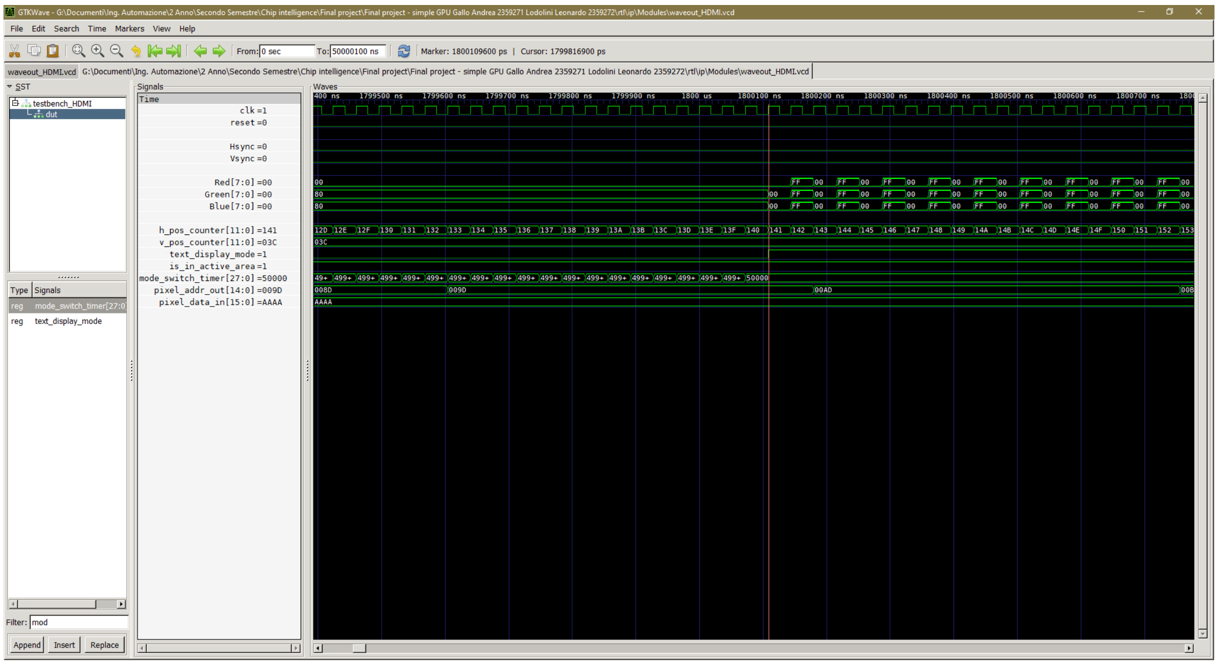
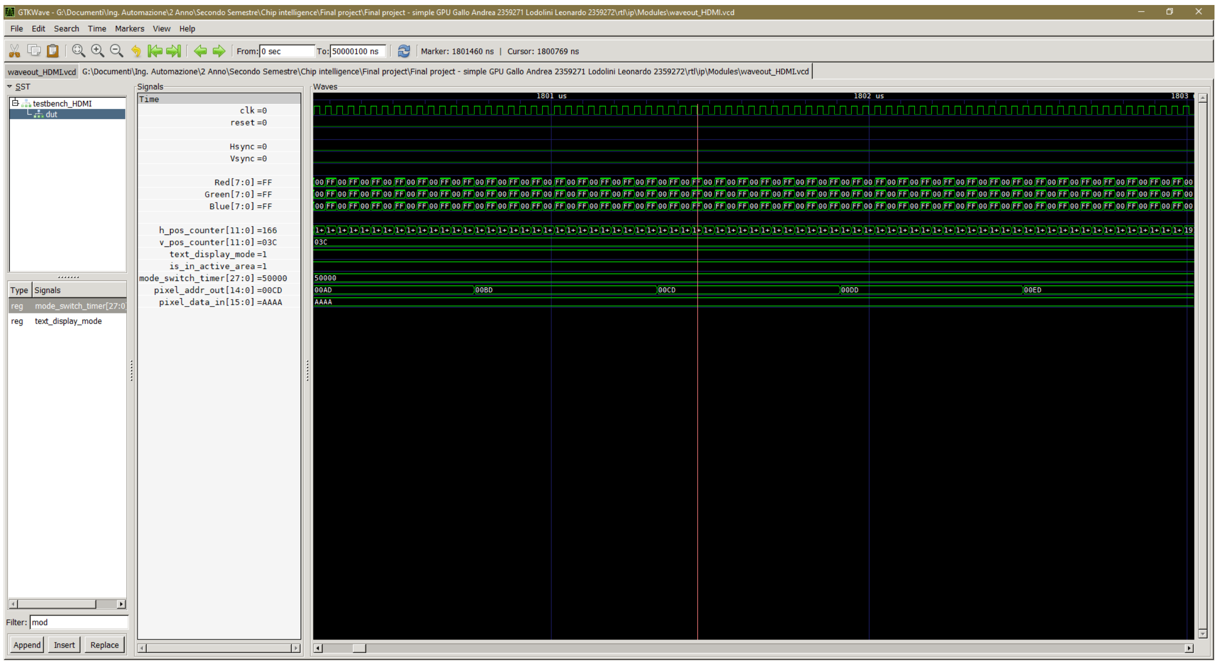

```
GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_HDMI.vcd
From: 0 sec   To: 50000100 ns   Marker: 1800109600 ps | Cursor: 1799816900 ps

SST: testbench_HDMI / dut
Type  Signals
reg   mode_switch_timer[27:0]
reg   text_display_mode
Filter: mod

Signals
                    clk =1
                  reset =0
                  Hsync =0
                  Vsync =0
              Red[7:0] =00
            Green[7:0] =00
             Blue[7:0] =00
     h_pos_counter[11:0] =141
     v_pos_counter[11:0] =03C
       text_display_mode =1
       is_in_active_area =1
mode_switch_timer[27:0] =50000
    pixel_addr_out[14:0] =009D
     pixel_data_in[15:0] =AAAA
```

Figure 4.10: The transition from color bar to text mode. When the `mode_switch_timer` reaches its limit, `text_display_mode` goes high, and the RGB output changes from colored bars to a monochrome pattern.

3. **Text Rendering Logic:** Once in text mode, the module correctly generates DPRAM read addresses (`pixel_addr_out`) and processes the incoming data. Figure 4.11 shows that the RGB output corresponds to the black-and-white pattern of the static input (`16'hAAAA`). This verifies that the entire read-path logic—including character coordinate calculation, address generation, and pipelined pixel serialization—is operating correctly.



```
clk =0
reset =0

Hsync =0
Vsync =0

Red[7:0] =FF
Green[7:0] =FF
Blue[7:0] =FF

h_pos_counter[11:0] =166
v_pos_counter[11:0] =03C
text_display_mode =1
is_in_active_area =1
mode_switch_timer[27:0] =50000
pixel_addr_out[14:0] =00CD
pixel_data_in[15:0] =AAAA
```

GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_HDMI.vcd
File Edit Search Time Markers View Help
From: 0 sec To: 50000100 ns
Marker: 1801460 ns | Cursor: 1800769 ns
testbench_HDMI
dut
Type Signals
reg mode_switch_timer[27:0]
reg text_display_mode
Filter: mod
Append Insert Replace
1801 us
1802 us
1803
00AD
00BD
00CD
00DD
00ED

Figure 4.11: Detailed view of the text rendering mode. The `pixel_addr_out` is actively generated, and the RGB output reflects the black-and-white pattern derived from the static `pixel_data_in`, confirming the read-path logic is working.

The successful visual confirmation of these three phases provides high confidence in the module's complex functionality and its readiness for final system integration.

Chapter 5

Verification Strategy and System Integration

While the previous chapters detailed the design of the individual firmware and hardware components, this chapter focuses on how these parts are brought together, tested, and validated as a cohesive system. The verification strategy is hierarchical, beginning with targeted tests on integrated sub-systems and culminating in a full-chip simulation that validates the end-to-end interaction between software and hardware. This chapter will describe the integration of the GPU modules, the verification methodology including the automation scripts, the final integration of the GPU into the top-level SoC, and an analysis of the final, complete system validation.

5.1 Hierarchical Integration: The Top-Level GPU Module

The first step in system integration is to encapsulate the individual hardware modules into a single, unified peripheral. This is the role of the `GPU_top` module. It serves as the master container for the entire custom display controller, instantiating and interconnecting the three core functional blocks: the `dpram_adapter`, the `Gowin_DPB` (Dual-Port RAM), and the `HDMI` signal generator.

This top-level module exposes a simple and clean interface to the rest of the SoC: it accepts the 32-bit CPU command as its primary input and provides the final 24-bit RGB video signals and synchronization pulses as its outputs. Internally, the dataflow follows two distinct, parallel paths, both centered around the dual-port nature of the framebuffer RAM:

- **The Write Path:** The incoming `i_cpu_cmd` is fed directly to the `dpram_adapter` instance. This "writer" module decodes the command and generates the corresponding write address (`a_write_address`) and data (`a_write_data`) for the framebuffer. These signals are wired to the write port (Port A) of the `Gowin_DPB` instance, allowing the CPU to populate the video memory.
- **The Read Path:** Concurrently and independently, the `HDMI` instance acts as the "reader". It generates a read address (`b_read_address`) based on its internal video timing counters. This address is sent to the read port (Port B) of the DPRAM. The

RAM responds by placing the stored data onto the `b_read_data_out` wire, which is then fed back into the `HDMI` module for processing and final rendering.

A critical piece of logic implemented at this level is the **anti-tearing mechanism**. Writes to the framebuffer should ideally only occur during the video signal's blanking intervals to prevent visual artifacts. This is achieved with a single line of logic:

```
// This logic prevents writes to the DPRAM while the HDMI controller is
// actively drawing the visible area of the screen.
wire final_wrea = we_a && !hdmi_is_displaying;
```

Listing 5.1: The anti-tearing logic implemented in GPU_top.

The raw write enable signal from the adapter (`we_a`) is gated with the inverted `hdmi_is_displaying` flag from the HDMI module. The resulting `final_wrea` signal is then used to drive the DPRAM's write enable port, ensuring that memory updates are synchronized with the vertical and horizontal blanking periods. This simple but crucial feature guarantees a stable, artifact-free visual output.

5.2 Verification Methodology and Automation

A hierarchical verification strategy was employed to validate the GPU peripheral, progressing from targeted sub-system tests to a comprehensive validation of the complete module. This approach relies on two primary test suites, each supported by automation scripts to ensure an efficient and repeatable workflow.

5.2.1 Rapid Write-Path Verification

The first verification stage focuses on the most critical data path: the GPU's write logic. The goal of this "fast" test is to confirm the data integrity between the `dpram_adapter` and the `Gowin_DPB` framebuffer, deliberately excluding the slower video generation logic to allow for rapid simulation and immediate feedback during development.

Testbench Design: `test_fast_datapath`

The verification of the write-path is performed by the `test_fast_datapath` testbench, a purpose-built module designed to confirm data integrity through the coupled `dpram_adapter` and `Gowin_DPB` components. The testbench instantiates both modules as a single sub-system and drives them through a rigorous, self-checking sequence.

The core of the testbench logic resides in a two-phase stimulus block. This procedural block first populates the DPRAM with a known, predictable data pattern and then reads it back to ensure no data corruption has occurred.

1. **Write Phase:** The test begins by iterating through a loop from 1 to `NUM_WRITES`. In each iteration, the testbench emulates the firmware by constructing a 32-bit command word. The command is specifically crafted for easy verification: the data payload (bits [15:0]) is set to be identical to the target memory address, which is encoded in the row offset field (bits [19:16]). For example, to write to address 5, the command word is formatted to carry the data value 5. This command is then presented to the `dpram_adapter`'s input. The adapter processes this command, generating the final write address and data signals which are then stored in the `Gowin_DPB` instance.

```verilog
// In the testbench, we set the data payload to equal the target
   address.
// This creates a predictable pattern in memory for easy
   verification.
for (integer i = 1; i <= NUM_WRITES; i = i + 1) begin
    // Format the command for the dpram_adapter.
    cpu_command = {16'b0, i[3:0], i[15:0]};
    @(posedge clk);
end
```

Listing 5.2: Command construction logic within the testbench's write phase.

2. **Read and Verify Phase:** Once the memory is populated, the testbench transitions to the verification phase. It takes direct control of the `Gowin_DPB`'s read port (Port B). Another loop iterates through the same range of addresses. For each address `i`, the testbench applies `i` to the read address bus and waits for two clock cycles to account for the synchronous RAM's read latency. After the data appears on the `data_from_ram` output, a critical comparison is made: `if (data_from_ram === i)`. The testbench uses Verilog's `$display` and `$error` system tasks to log the outcome of each comparison to the console, providing an explicit, human-readable report on the data integrity for every location tested and terminating the simulation on failure.

This structured, write-then-verify methodology provides a clean and unambiguous validation of the entire write datapath, from the command input of the `dpram_adapter` to the data output of the `Gowin_DPB`.

Automation Script: `run_gpu_test_fast.cmd`

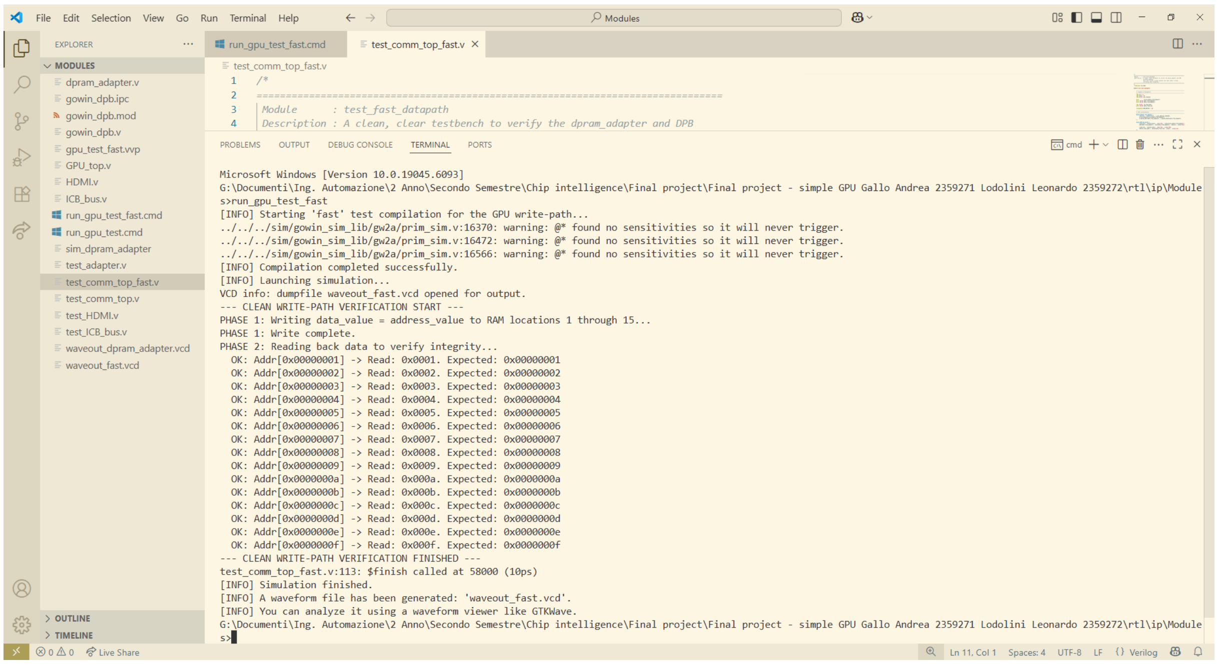
Execution is automated by the `run_gpu_test_fast.cmd` script. This script streamlines the process by selectively compiling only the necessary source files with Icarus Verilog and immediately running the resulting simulation executable. This automated workflow is essential for efficient regression testing.

```bat
REM Compile only the necessary Verilog files
iverilog -o gpu_test_fast.vvp dpram_adapter.v gowin_dpb.v ...

REM Check for compilation errors before proceeding
IF %ERRORLEVEL% NEQ 0 GOTO :EOF

REM Execute the simulation
vvp gpu_test_fast.vvp
```

Listing 5.3: Core commands from the fast simulation script.

Simulation Results and Waveform Analysis

The execution of the automated test script provides immediate, human-readable feedback on the console. This terminal log serves as the first layer of debugging and verification. As shown in Figure 5.1, the output explicitly reports the successful completion of both the write and read phases of the test, with each read-back value matching the expected data. This confirms that the test passed all of its self-checking assertions.



```
/*
================================================================================
Module      : test_fast_datapath
Description : A clean, clear testbench to verify the dpram_adapter and DPB
```

```
Microsoft Windows [Version 10.0.19045.6093]
G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Module
s>run_gpu_test_fast
[INFO] Starting 'fast' test compilation for the GPU write-path...
../../../sim/gowin_sim_lib/gw2a/prim_sim.v:16370: warning: @* found no sensitivities so it will never trigger.
../../../sim/gowin_sim_lib/gw2a/prim_sim.v:16472: warning: @* found no sensitivities so it will never trigger.
../../../sim/gowin_sim_lib/gw2a/prim_sim.v:16566: warning: @* found no sensitivities so it will never trigger.
[INFO] Compilation completed successfully.
[INFO] Launching simulation...
VCD info: dumpfile waveout_fast.vcd opened for output.
--- CLEAN WRITE-PATH VERIFICATION START ---
PHASE 1: Writing data_value = address_value to RAM locations 1 through 15...
PHASE 1: Write complete.
PHASE 2: Reading back data to verify integrity...
  OK: Addr[0x00000001] -> Read: 0x0001. Expected: 0x00000001
  OK: Addr[0x00000002] -> Read: 0x0002. Expected: 0x00000002
  OK: Addr[0x00000003] -> Read: 0x0003. Expected: 0x00000003
  OK: Addr[0x00000004] -> Read: 0x0004. Expected: 0x00000004
  OK: Addr[0x00000005] -> Read: 0x0005. Expected: 0x00000005
  OK: Addr[0x00000006] -> Read: 0x0006. Expected: 0x00000006
  OK: Addr[0x00000007] -> Read: 0x0007. Expected: 0x00000007
  OK: Addr[0x00000008] -> Read: 0x0008. Expected: 0x00000008
  OK: Addr[0x00000009] -> Read: 0x0009. Expected: 0x00000009
  OK: Addr[0x0000000a] -> Read: 0x000a. Expected: 0x0000000a
  OK: Addr[0x0000000b] -> Read: 0x000b. Expected: 0x0000000b
  OK: Addr[0x0000000c] -> Read: 0x000c. Expected: 0x0000000c
  OK: Addr[0x0000000d] -> Read: 0x000d. Expected: 0x0000000d
  OK: Addr[0x0000000e] -> Read: 0x000e. Expected: 0x0000000e
  OK: Addr[0x0000000f] -> Read: 0x000f. Expected: 0x0000000f
--- CLEAN WRITE-PATH VERIFICATION FINISHED ---
test_comm_top_fast.v:113: $finish called at 58000 (10ps)
[INFO] Simulation finished.
[INFO] A waveform file has been generated: 'waveout_fast.vcd'.
[INFO] You can analyze it using a waveform viewer like GTKWave.
G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Module
s>
```

Figure 5.1: Terminal output from the **`run_gpu_test_fast.cmd`** script. The log confirms that all read-back data matched the expected values, successfully validating the data path's integrity.

For a deeper, cycle-accurate analysis of the hardware's behavior, the simulation generates a VCD waveform file. The visual inspection of these waveforms confirms the correctness of the internal logic.

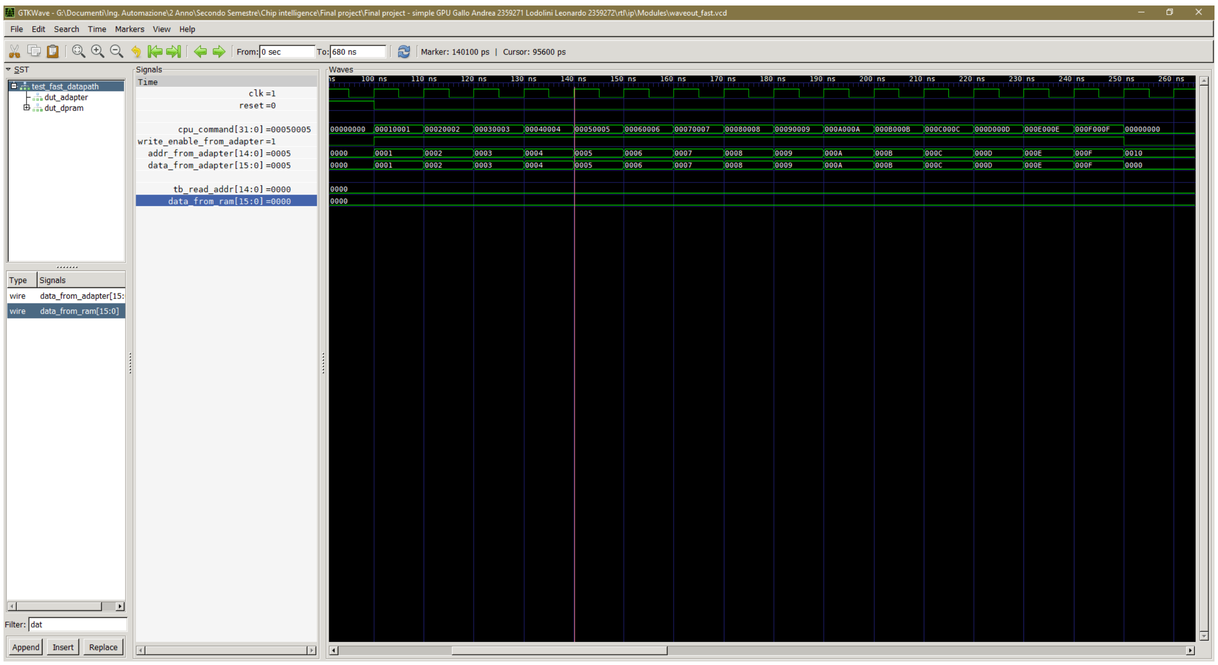

GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_fast.vcd
From: 0 sec
To: 680 ns
Marker: 140100 ps | Cursor: 95600 ps
test_fast_datapath
dut_adapter
dut_dpram
clk =1
reset =0
cpu_command[31:0] =00050005
write_enable_from_adapter =1
addr_from_adapter[14:0] =0005
data_from_adapter[15:0] =0005
tb_read_addr[14:0] =0000
data_from_ram[15:0] =0000
wire data_from_adapter[15:
wire data_from_ram[15:0]
Filter: dat

Figure 5.2: Waveform detailing the write phase of the fast datapath test. The **`cpu_command`** input is updated on each clock cycle, and the **`dpram_adapter`** correctly generates the corresponding write address and data signals for the DPRAM.

Figure 5.2 shows the write phase in detail. At each rising edge of the clock, the testbench drives a new command onto the `cpu_command` bus. In response, the `addr_from_adapter` and `data_from_adapter` signals immediately reflect the correct, decoded values, demonstrating the proper functioning of the adapter's combinational logic.

Figure 5.3 illustrates the subsequent read and verify phase. The testbench drives the `tb_read_addr` input of the DPRAM's read port. After a one-cycle read latency, the corresponding data correctly appears on the `data_from_ram` output bus, matching the address that was read. This confirms that the DPRAM stored the data correctly and that its read port is functioning as expected.

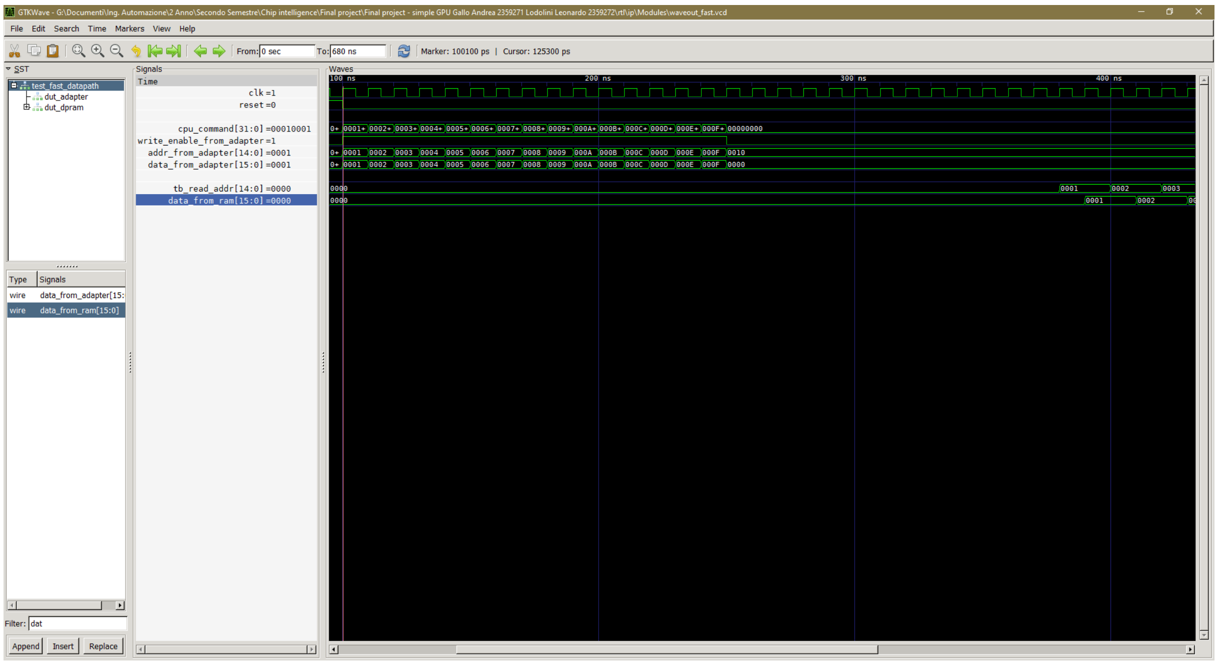

GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\rtl\ip\Modules\waveout_fast.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 680 ns
Marker: 100100 ps | Cursor: 125300 ps
SST
test_fast_datapath
dut_adapter
dut_dpram
Type Signals
wire data_from_adapter[15:
wire data_from_ram[15:0]
Filter: dat
Append Insert Replace
Signals
Time
clk =1
reset =0
cpu_command[31:0] =00010001
write_enable_from_adapter =1
addr_from_adapter[14:0] =0001
data_from_adapter[15:0] =0001
tb_read_addr[14:0] =0000
data_from_ram[15:0] =0000
Waves
100 ns
200 ns
300 ns
400 ns

Figure 5.3: Waveform illustrating the read phase. The testbench drives the read address (`tb_read_addr`), and after the expected read latency, the correct data appears on the `data_from_ram` output, confirming the value previously written.

The combination of the successful terminal log and the detailed waveform analysis provides definitive proof that the write datapath sub-system is fully functional and free of data corruption issues.

5.2.2 Complete GPU System Verification

After validating the core write-path in isolation, the second stage of verification focuses on the entire `GPU_top` module. This "complete" system test is designed to validate the end-to-end functionality of the peripheral, ensuring that all three internal components —the `dpram_adapter`, the `Gowin_DPB`, and the `HDMI` controller— work together harmoniously. The key goal is to verify that commands sent to the GPU result in correctly rendered pixels on the final video output.

Testbench Design: `test_full_system`

The verification is performed by the `test_full_system` testbench, which instantiates the complete `GPU_top` module as its Device Under Test (DUT). Unlike the fast datapath test, this test is primarily observational, designed to generate a comprehensive waveform for manual visual analysis. The test sequence is structured to mimic a realistic operational scenario:

1. **Initialization and Mode Transition:** The test begins by applying a reset. It then waits for a fixed duration (`500,000 ns`), long enough for the `HDMI` module's internal timer to expire. This allows the DUT to autonomously transition from its initial color bar diagnostic mode to its primary text rendering mode.

2. **Synchronized Framebuffer Write:** To ensure a safe, artifact-free update of the framebuffer, the testbench waits for the start of the next Vertical Blanking Interval (VBI). It achieves this by synchronizing with the `Vsync` signal from the DUT. This demonstrates a robust testing methodology that respects the anti-tearing mechanism designed into the `GPU_top`.

```
// Wait for the start of the next Vertical Sync pulse.
$display("[%0t ns] Waiting for next Vsync...", $time);
@(posedge Vsync);
@(negedge Vsync); // Now safely in the vertical blanking interval

// At this point, we are certain that the ‘is_in_active_area‘
   signal
// inside the HDMI controller is low, so writes will be enabled.
$display("[%0t ns] VBI detected. Writing all characters in a burst.
   ", $time);
write_character(0); // Write character ’H’
...
```

Listing 5.4: Synchronizing framebuffer writes with the VBI.

3. **Stimulus and Observation:** Once in the VBI, the testbench writes a short message ("HI!") to the framebuffer in a quick burst. It then allows the simulation to run for a significant duration (50ms), sufficient to capture several full video frames being drawn. This allows for a thorough visual inspection of the final RGB and sync signals in a waveform viewer.

Automation Script: `run_gpu_test.cmd`

As with the fast test, the complete system simulation is automated by a dedicated batch script, `run_gpu_test.cmd`. The primary difference is that this script compiles ALL of the Verilog source files that constitute the GPU peripheral, including the top-level `GPU_top.v`, the `HDMI.v` module, and the `test_full_system.v` testbench. While this full compilation and the subsequent simulation are significantly more time-consuming, they are indispensable for a final, comprehensive validation of the integrated peripheral before its integration into the larger SoC. The script follows the same compile-check-run sequence as its "fast" counterpart, ensuring a reliable and repeatable verification process.

Simulation Results and Waveform Analysis

The primary output of the full system test is the VCD waveform file, which allows for a thorough visual inspection of the `GPU_top` module's end-to-end behavior. The analysis of these waveforms confirms that all internal components interact correctly.

A macroscopic view of the simulation, shown in Figure 5.4, demonstrates the system's stability over multiple video frames. The initial color bar pattern is visible in the RGB signals, followed by a switch to a stable monochrome text output after the mode transition.

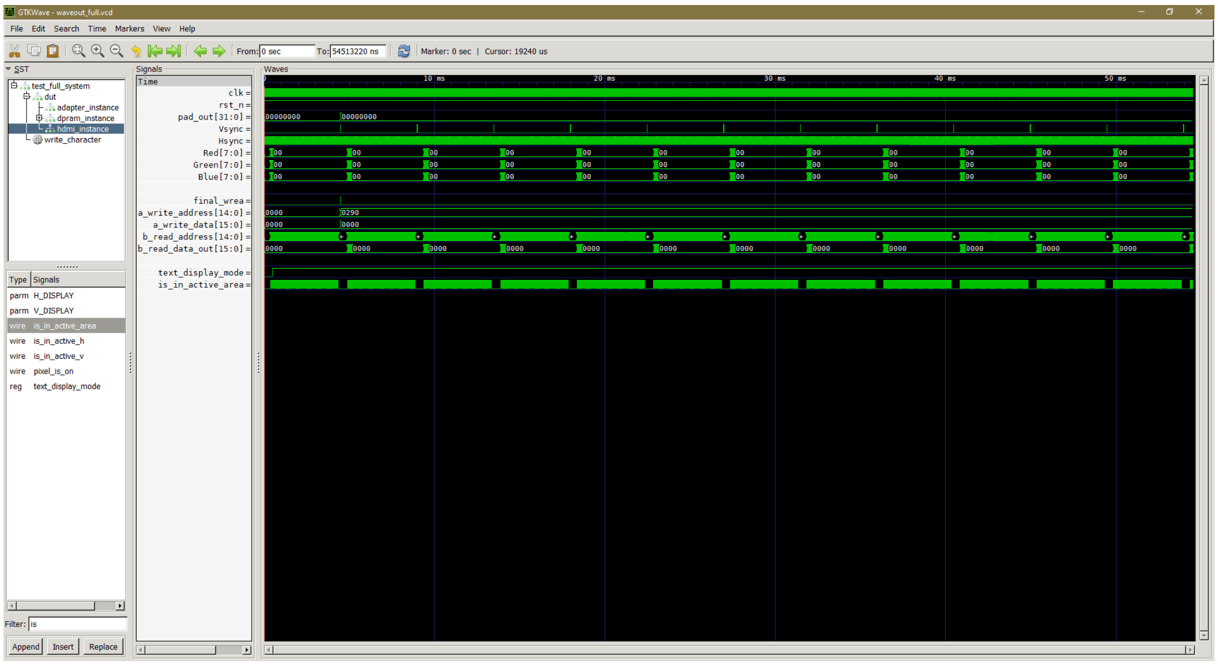
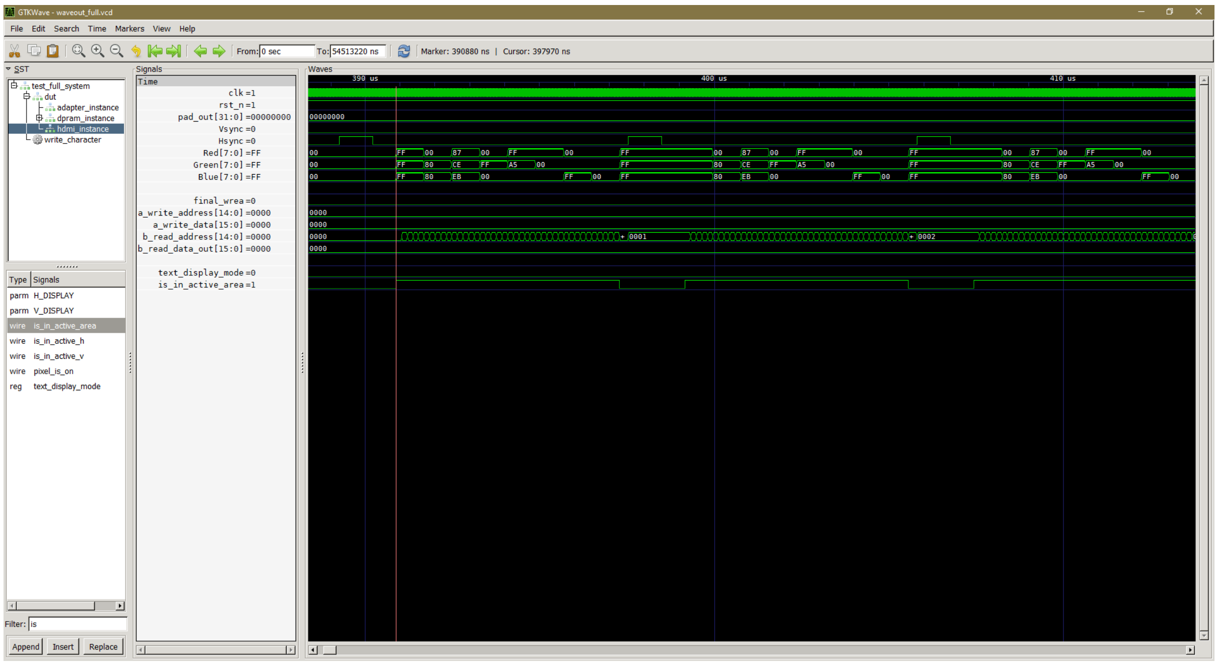


Figure 5.4: A wide view of the full system simulation. The initial diagnostic color bar pattern is visible, followed by a stable monochrome output, confirming the correct high-level behavior.

The key operational phases are confirmed by zooming in on specific moments of the simulation:

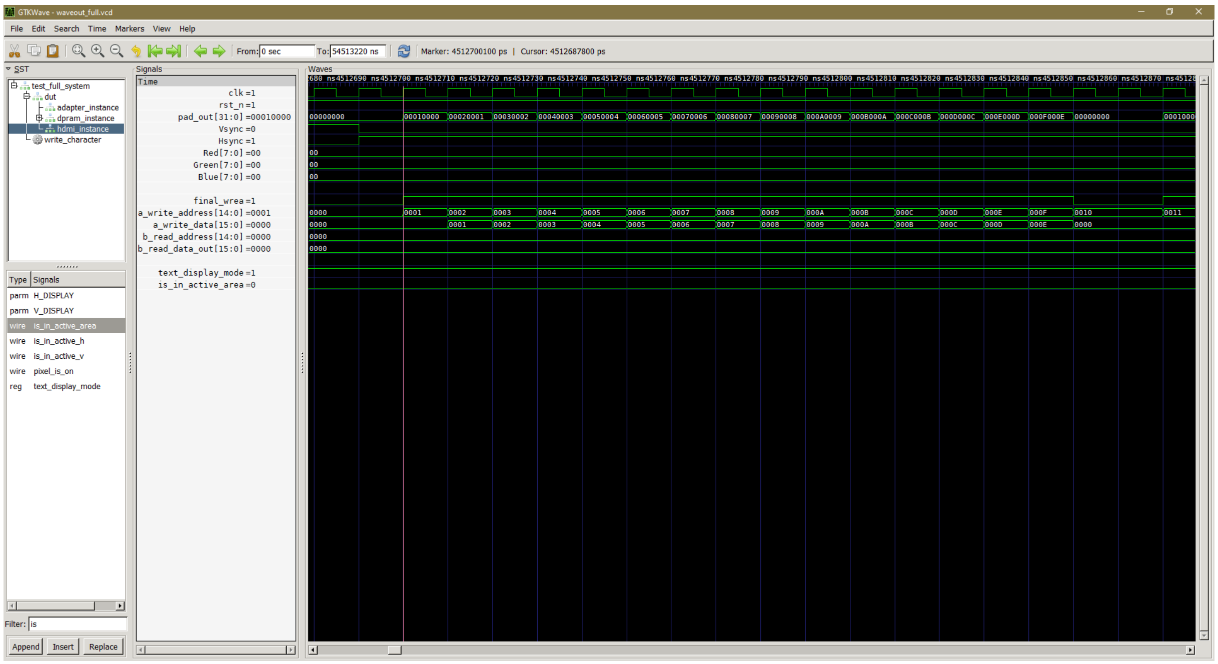
- **Color Bar Mode Validation:** Figure 5.5 shows a detailed view of the startup phase. The module correctly generates different RGB values to create the vertical test strips, confirming that the initial state of the `HDMI` controller is functional. During this time, the command bus (`pad_out`) is idle, and the write-path logic is inactive.



```
GTKWave - waveout_full.vcd
From: 0 sec   To: 54513220 ns   Marker: 390880 ns  |  Cursor: 397970 ns

Time
                 clk =1
               rst_n =1
     pad_out[31:0] =00000000
               Vsync =0
               Hsync =0
           Red[7:0] =FF
         Green[7:0] =FF
          Blue[7:0] =FF

          final_wrea =0
a_write_address[14:0] =0000
  a_write_data[15:0] =0000
 b_read_address[14:0] =0000
b_read_data_out[15:0] =0000

   text_display_mode =0
    is_in_active_area =1
```

Figure 5.5: Detailed view of the initial color bar mode. The RGB signals cycle through various colors while the CPU command bus remains idle.

- **Safe Framebuffer Writing:** Figure 5.6 captures the moment the testbench writes character data. The waveform clearly shows the burst of commands on the `pad_out` bus. Crucially, this activity occurs while the video output is blank (RGB signals are all zero) because the testbench has synchronized the write to the vertical blanking interval. The `final_wrea` signal is asserted, validating that the anti-tearing logic is effective and correctly enables writes to the DPRAM.



```
GTKWave - waveout_full.vcd
From: 0 sec   To: 54513220 ns   Marker: 4512700100 ps  |  Cursor: 4512687800 ps

Time
                 clk =1
               rst_n =1
     pad_out[31:0] =00010000
               Vsync =0
               Hsync =1
           Red[7:0] =00
         Green[7:0] =00
          Blue[7:0] =00

          final_wrea =1
a_write_address[14:0] =0001
  a_write_data[15:0] =0000
 b_read_address[14:0] =0000
b_read_data_out[15:0] =0000

   text_display_mode =1
    is_in_active_area =0
```

Figure 5.6: The character write phase, synchronized with the Vertical Blanking Interval. Write commands are sent to the DUT while the RGB output is blank, and the `final_wrea` signal is correctly asserted, confirming the anti-tearing logic.

- **Correct End-to-End Rendering:** Finally, Figure 5.7 demonstrates the successful completion of the entire data flow. Shortly after the testbench writes the character data, the `HDMI` module's raster scan reaches that memory region. It correctly reads the bitmap data for a character row from the DPRAM via its read port (`b_read_address` and `b_read_data_out`). This data is then immediately translated into the correct sequence of white and black pixels on the final RGB output, validating that the entire integrated system works as intended.

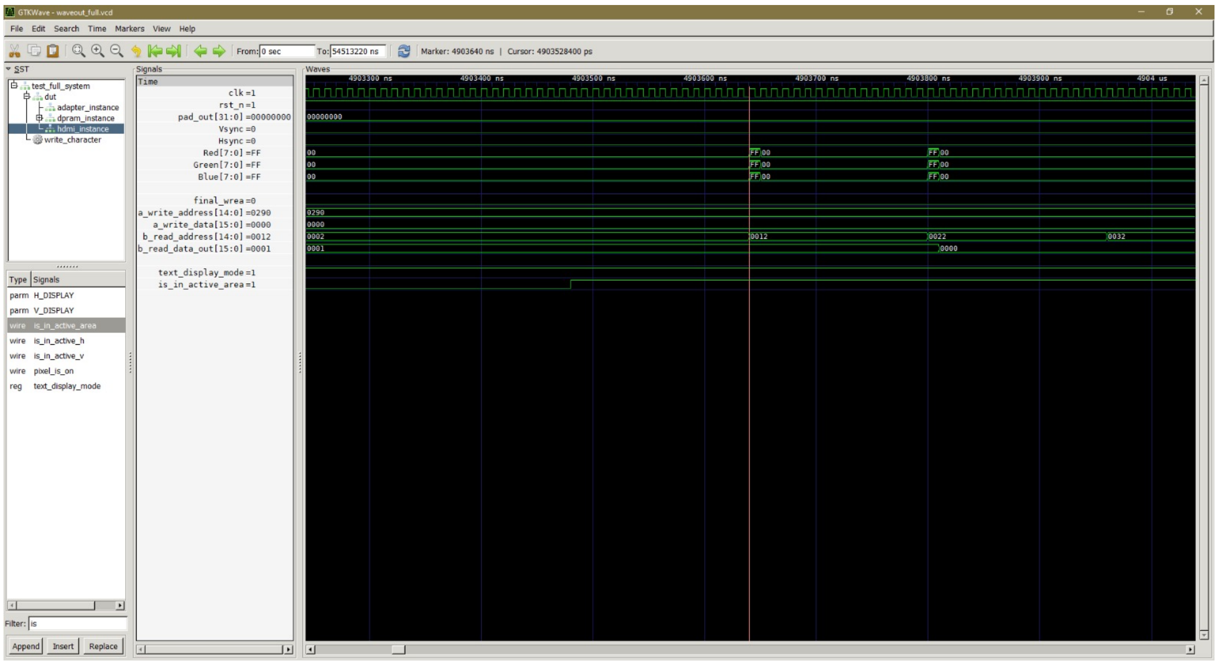


Figure 5.7: The video rendering phase in text mode. The waveform captures the moment the `HDMI` module's raster scan reads a memory location recently updated by the testbench (`b_read_address` and `b_read_data_out`) and correctly renders the corresponding pixels on the RGB output.

The successful validation of these key phases confirms that the `GPU_top` module functions as a complete and correct video display peripheral, ready for final integration into the full SoC.

5.3 Integrating the Custom GPU into the SoC Hierarchy

Integrating the custom GPU peripheral required more than just designing the IP; it necessitated a series of targeted modifications to the hierarchical modules of the Hummingbird E203 SoC. This process involved two primary engineering tasks: first, connecting the GPU as a slave peripheral to the Inter-Chip Bus (ICB) for communication with the CPU master; and second, carefully routing the generated video signals upwards through the SoC's layered structure to the top-level output pins. The following subsections detail the specific changes made to each core module, from the peripheral hub up to the top-level wrapper, explaining the purpose and rationale behind each modification.

5.3.1 Integration into the Peripheral Hub: `e203_subsys_perips`

Module Purpose The `e203_subsys_perips` module serves as the central hub for all memory-mapped peripherals within the SoC. Its core component is the `sirv_icb1to16_bus`, a bus fabric that acts as a 1-to-16 address decoder and multiplexer. It receives bus transactions from the CPU's master port and routes them to the correct slave peripheral based on the transaction's address.

Modifications Implemented This module was the primary integration point for the GPU and thus underwent the most significant modifications. First, the module's port list was expanded to include five new outputs for the video signals: `H_sync`, `V_sync`, `Red`, `Green`, and `Blue`. Inside the module, we instantiated our two custom IPs: `ICB_bus`, which handles the ICB protocol, and `GPU_top`, which contains the entire graphics pipeline. As shown in Listing 5.5, the `ICB_bus` instance is connected as a slave to the `o5` port of the bus fabric, receiving commands from the CPU. The 32-bit command payload is then passed from the `ICB_bus` to the `GPU_top` via the internal `my_io_pad_out` wire. The video outputs of the `GPU_top` are then connected directly to the newly declared output ports of the `e203_subsys_perips` module.

```
// Add video signals to the module's output port list
module e203_subsys_perips(
    // ... other ports ...
    output                           H_sync,
    output                           V_sync,
    output  [7:0]                    Red,
    output  [7:0]                    Green,
    output  [7:0]                    Blue,
    // ... other ports ...
);

// ... (Bus fabric instantiation connects CPU commands to my_periph_icb_* signals on port o5) ...

// Internal wire to connect the two custom modules
wire [31:0] my_io_pad_out;

// 1. Instantiate the ICB Bus Interface
ICB_bus u_ICB_bus(
    .clk               (clk),
    .rst_n             (rst_n),
    .GPU_icb_cmd_valid (my_periph_icb_cmd_valid), // from bus fabric port o5
    .GPU_icb_cmd_ready (my_periph_icb_cmd_ready), // to bus fabric port o5
    .GPU_icb_cmd_addr  (my_periph_icb_cmd_addr ),  // from bus fabric port o5
    // ... other ICB signals connected to port o5 ...
    .o_cpu_cmd         (my_io_pad_out)
);

// 2. Instantiate the main Graphics Peripheral
GPU_top u_GPU_top(
    .clk                  (clk),
    .rst_n                (rst_n),
    .i_cpu_cmd            (my_io_pad_out), // from ICB_bus
    .o_gpu_hsync          (H_sync),        // to module output
    .o_gpu_vsync          (V_sync),        // to module output
    .o_gpu_red            (Red),           // to module output
    .o_gpu_green          (Green),         // to module output
    .o_gpu_blue           (Blue)           // to module output
);
```

Listing 5.5: GPU Instantiation within `e203_subsys_perips`.

Rationale These changes were critical for two fundamental reasons. The first was to establish a physical, memory-mapped communication link between the CPU and the GPU. The second was to begin the propagation path for the generated video signals, making them available to the next level in the SoC hierarchy.

5.3.2 Propagating Signals Through the Main Subsystem: `e203_subsys_main`

Module Purpose The `e203_subsys_main` module acts as a primary structural container within the SoC. Its fundamental role is to instantiate and interconnect the three core operational blocks: the CPU complex (`e203_cpu_top`), which serves as the bus master, the main memory subsystem (`e203_subsys_mems`), and the peripheral hub (`e203_subsys_perips`), both of which act as bus slaves.

Modifications Implemented To continue the upward path of the video signals generated within `e203_subsys_perips`, it was necessary to modify its parent module, `e203_subsys_main`. The modification followed a standard hierarchical design pattern: the `e203_subsys_main` module's own port list was expanded to include the five video outputs (`H_sync`, `V_sync`, `Red`, `Green`, `Blue`). Subsequently, as detailed in Listing 5.6, the instantiation of `u_e203_subsys_perips` within `e203_subsys_main` was updated. The newly created output ports of the `perips` module were wired directly to the corresponding, newly created output ports of the `main` module.

```
// 1. Add video ports to the module's output list
module e203_subsys_main(
    // ... other ports ...
    output                      H_sync,
    output                      V_sync,
    output  [7:0]               Red,
    output  [7:0]               Green,
    output  [7:0]               Blue,
    // ... other ports ...
);

// ... (instantiation of CPU and memory subsystems) ...

// 2. Connect the new ports in the instantiation of the peripheral hub
e203_subsys_perips u_e203_subsys_perips (
    // ... other port connections ...
    .H_sync                   (H_sync),
    .V_sync                   (V_sync),
    .Red                      (Red),
    .Green                    (Green),
    .Blue                     (Blue),
    // ... other port connections ...
);
```

Listing 5.6: Daisy-chaining Video Signals in `e203_subsys_main`.

Rationale This modification was purely structural but absolutely essential. The video signals, now exposed at the boundary of the `perips` module, needed a defined path to traverse the next layer of the design hierarchy. Without these connections, the signals would have been contained within `e203_subsys_main` and would not have been accessible to the higher-level wrappers, effectively terminating their path to the physical I/O pins. This step ensures that the video datapath remains unbroken as it moves up toward the SoC's top level.

5.3.3 Completing the Path to the Top-Level Pins

Module Purposes The modules at the upper levels of the hierarchy serve distinct and critical roles in assembling the final System-on-Chip:

- `e203_subsys_top`: This module acts as the main computational system boundary. It integrates the CPU, memory, and peripheral subsystems (`e203_subsys_main`) into a single cohesive unit, representing the complete, programmable core of the SoC.
- `e203_soc_top`: This is the primary system integrator. Its purpose is to combine the main computational subsystem (`e203_subsys_top`) with essential system-wide services like the RISC-V Debug Module and the Always-On (AON) power management block.
- `e203_soc_demo`: This module represents the final, physical boundary of the chip. It instantiates the entire integrated system (`e203_soc_top`), generates the necessary clocks from an external source, and connects all internal signals to the top-level I/O ports, which correspond to the physical pins of the FPGA.

Modifications Implemented To route the video signals out of the SoC, a recursive "daisy-chaining" pattern was applied to each of these hierarchical wrappers. The process, identical at each stage and illustrated in Listing 5.7, involved augmenting each module's port list with the five video outputs and then connecting them to the corresponding ports of the child module instantiated within. This was performed sequentially: from `e203_subsys_main` up to `e203_subsys_top`, then to `e203_soc_top`, and finally terminating at the `e203_soc_demo` boundary.

```
// In e203_subsys_top.v
module e203_subsys_top(
    // ... other ports ...
    output          H_sync,
    output          V_sync,
    output  [7:0]   Red,
    output  [7:0]   Green,
    output  [7:0]   Blue
);
  // ...
  e203_subsys_main  u_e203_subsys_main(
      // ... other connections ...
      .H_sync          (H_sync),
      .V_sync          (V_sync),
      .Red             (Red),
      .Green           (Green),
      .Blue            (Blue)
  );
// ...

// In e203_soc_top.v
module e203_soc_top(
    // ... other ports ...
    output           H_sync,
    output           V_sync,
    output [7:0]     Red,
    output [7:0]     Green,
    output [7:0]     Blue
);
  // ...
 e203_subsys_top u_e203_subsys_top(
      // ... other connections ...
      .H_sync          (H_sync),
      .V_sync          (V_sync),
      .Red             (Red),
      .Green           (Green),
      .Blue            (Blue)
  );
// ...

// In e203_soc_demo.v
module e203_soc_demo (
    // ... other ports ...
    output           H_sync,
    output           V_sync,
    output [7:0]     Red,
    output [7:0]     Green,
    output [7:0]     Blue
);
  // ...
  e203_soc_top e203_soc_ins (
      // ... other connections ...
      .H_sync          (H_sync),
      .V_sync          (V_sync),
      .Red             (Red),
      .Green           (Green),
      .Blue            (Blue)
  );
```

Listing 5.7: Recursive daisy-chaining of video signals up to the SoC boundary.

Rationale While the primary function of these modules is system integration, our modifications were purely for signal propagation. This systematic wiring was the essential final step to create an unbroken physical datapath from the signal source (our `GPU_top`) to the physical world. By exposing these signals at the `e203_soc_demo` level, we enabled their connection to the system-level testbench for final verification and ensured they could be assigned to the physical HDMI output pins of the FPGA for on-board implementation.

5.4 End-to-End System Validation: A Full SoC Co-Simulation

With the GPU peripheral fully integrated into the `e203_soc_demo` hierarchy, the final validation stage begins. This phase is designed to perform a comprehensive, end-to-end test of the entire system, treating the SoC as a black box. This is the ultimate test of the project, as it validates the seamless interaction between the custom hardware and the C firmware running on the embedded RISC-V core. This process is a true hardware-software co-simulation, proving that all components work together as intended in a realistic operational environment.

5.4.1 The Final Testbench Architecture

The final validation is orchestrated by a comprehensive system-level testbench, `sys_tb_top.sv`, written in SystemVerilog. This testbench is fundamentally different from the previous unit-level tests. Its primary role is not to drive internal module signals, but to simulate the real-world interfaces with which the SoC interacts, such as clock inputs and UART communication lines. The testbench instantiates the entire `e203_soc_demo` module and performs three critical functions:

1. **System Initialization:** The testbench is responsible for generating the system clocks (a high-frequency clock for the core and a low-frequency one for auxiliary functions) and managing the SoC's active-low reset signal, `rst_n`. It holds the SoC in reset for a predefined period to allow for stable startup.

2. **Host Terminal Emulation:** A key function is to act as a host computer sending data to the SoC's UART. This is achieved through a dedicated task, `uart_tx_data`, which accurately simulates the UART serial protocol (start bit, 8 data bits, stop bit) by manipulating the specific GPIO pin mapped to the SoC's UART receiver (`gpio_in[16]`). The main stimulus block of the testbench is designed for realistic data injection:

   - It first uses the SystemVerilog system task `$readmemh` to load a sequence of characters from an external text file (`./input.txt`) into a local register array.
   - It then waits for a significant period (#7ms) to ensure the SoC's firmware has completed its own initialization sequence after reset.
   - Finally, it iterates through the character array using a `foreach` loop, calling the `uart_tx_data` task for each character. A small delay is inserted between byte transmissions to mimic realistic host behavior.

3. **Video Output Observation:** The testbench connects wires to the SoC's primary video output ports (`H_sync`, `V_sync`, `Red`, etc.). It does not perform any automatic data checking on these complex signals. Instead, its purpose is to enable their recording into a Value Change Dump (`.vcd`) file via the `$dumpvars` task. This allows for detailed, offline visual analysis of the final video stream in a waveform viewer.

This architecture creates a comprehensive test environment that closely mimics the SoC's real-world application, providing the highest possible confidence in the project's overall correctness.

5.4.2 Firmware Compilation and Loading Process

A critical aspect of a true hardware-software co-simulation is the ability to run the actual compiled firmware on the simulated hardware model. The verification flow for this project fully implements this methodology. The process involves two key stages: compiling the C firmware into a machine-readable format from its specific development directory and pre-loading this image into the SoC's simulated memory before the simulation begins.

Firmware Compilation via Makefile

The C firmware described in Chapter 3 is not simulated directly. Instead, it is compiled using a standard RISC-V GCC toolchain. The entire build process is managed by a `Makefile`, located within the `firmware/debug` directory of the project structure. Executing the `make` command in this directory orchestrates the full compilation pipeline:

1. It compiles the C source files (e.g., `main.c`) into RISC-V object files.
2. It links these object files with the necessary startup code and libraries provided by the Hummingbird E203 SDK to produce a standard ELF (Executable and Linkable Format) file.
3. Finally, it uses a utility (such as `objcopy`) to extract the raw machine code from the ELF file and convert it into a simple hexadecimal text file, typically named `firmware.hex`. This output file is placed in a location accessible to the simulation tools.

This `.hex` file contains the exact sequence of 32-bit instruction words that the RISC-V core will execute.

Integration into the Simulation Environment

The final step is to make this compiled firmware available to the simulated SoC. This is handled by the top-level simulation script (e.g., `sim_run_sys_tb.cmd`). This script is responsible not only for compiling all the necessary Verilog source files for the SoC core and standard peripherals, but it has also been modified to include the custom GPU modules developed in this project.

```
# ... existing list of SoC source files ...
../rtl/ip/Modules/dpram_adapter.v ^
../rtl/ip/Modules/GPU_top.v ^
../rtl/ip/Modules/gowin_dpb.v ^
../rtl/ip/Modules/HDMI.v ^
../rtl/ip/Modules/ICB_bus.v ^
# ... testbench files ...
```

Listing 5.8: Additions to the SoC simulation script to include the custom GPU modules.

The simulation environment itself is configured so that when the Verilog model of the SoC's Boot ROM is elaborated, it executes the system task `$readmemh("firmware.hex", ...)`. This powerful Verilog task reads the hexadecimal file from the disk and uses its contents to initialize the simulated ROM.

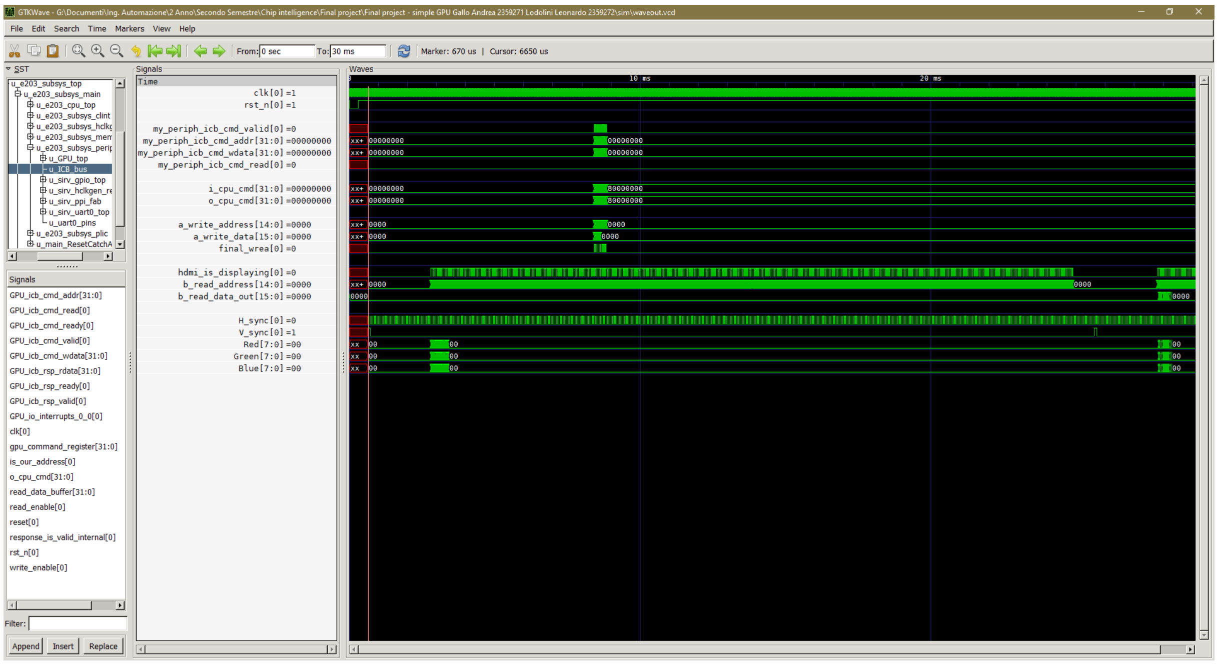
This entire flow—from C code, through a `Makefile` in the `firmware/debug` directory to a `.hex` file, and finally into the simulated hardware via `$readmemh`—creates a perfect replica of the real-world operational environment. When the testbench releases the reset signal, the simulated RISC-V core begins fetching and executing the user-compiled firmware from the pre-loaded ROM, just as it would on a physical FPGA.

5.4.3 End-to-End Waveform Analysis: The Life Cycle of a Character

The final and most definitive proof of the project's success is provided by the detailed analysis of the full SoC simulation waveforms. This analysis allows us to trace the entire life cycle of a single character, from its inception as a command issued by the firmware, through the ICB bus, into the GPU's internal framebuffer, and finally to its manifestation as pixels on the video output. This end-to-end trace provides an unequivocal validation of the seamless integration and correct operation of all hardware and software components.

GPU Initialization and Firmware Interaction

A wide-view of the full SoC simulation, shown in Figure 5.8, reveals the system's high-level behavior. The periodic bursts of activity on the ICB bus correspond to the firmware executing its main polling loop, while the stability of the video synchronization signals (`H_sync`, `V_sync`) is clearly visible.


GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\sim\waveout.vcd

From: 0 sec To: 30 ms | Marker: 670 us | Cursor: 6650 us

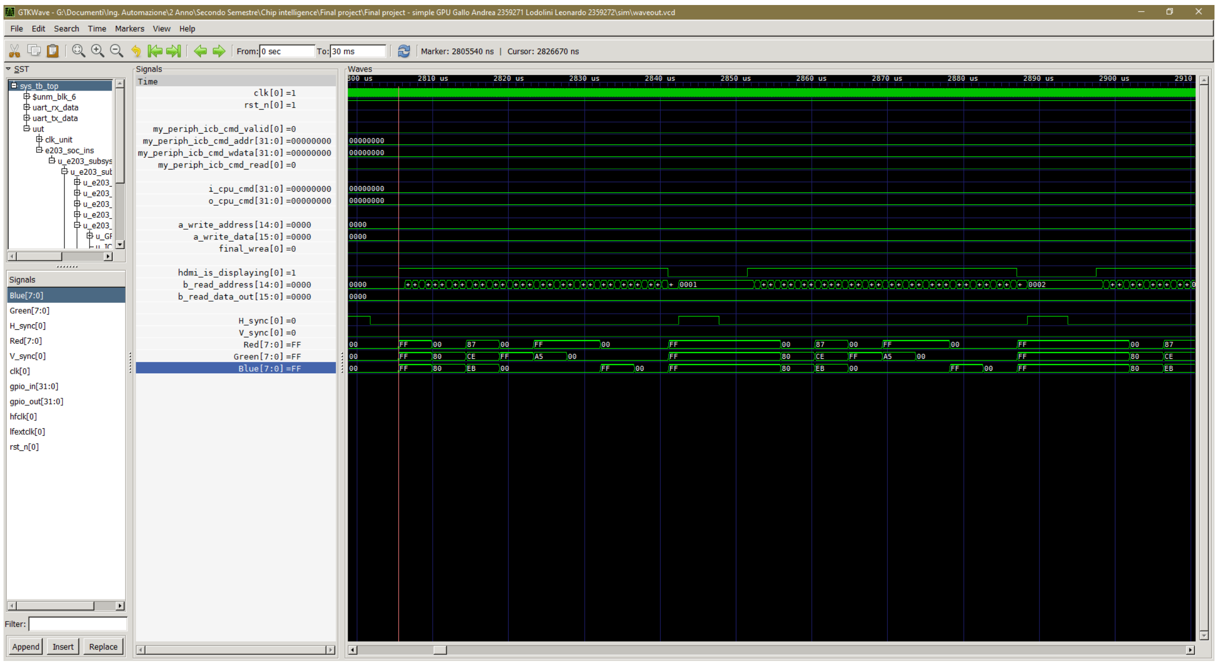
```
u_e203_subsys_top
  u_e203_subsys_main
    u_e203_cpu_top
    u_e203_subsys_clint
    u_e203_subsys_hclkg
    u_e203_subsys_mem
    u_e203_subsys_perip
      u_GPU_top
      u_ICB_bus
      u_sirv_gpio_top
      u_sirv_hclkgen_re
      u_sirv_ppi_fab
      u_sirv_uart0_top
      u_uart0_pins
    u_e203_subsys_plic
    u_main_ResetCatchA
```

```
GPU_icb_cmd_addr[31:0]
GPU_icb_cmd_read[0]
GPU_icb_cmd_ready[0]
GPU_icb_cmd_valid[0]
GPU_icb_cmd_wdata[31:0]
GPU_icb_rsp_rdata[31:0]
GPU_icb_rsp_ready[0]
GPU_icb_rsp_valid[0]
GPU_io_interrupts_0_0[0]
clk[0]
gpu_command_register[31:0]
is_our_address[0]
o_cpu_cmd[31:0]
read_data_buffer[31:0]
read_enable[0]
reset[0]
response_is_valid_internal[0]
rst_n[0]
write_enable[0]
```

```
Time
                        clk[0] =1
                      rst_n[0] =1

      my_periph_icb_cmd_valid[0] =0
  my_periph_icb_cmd_addr[31:0] =00000000
 my_periph_icb_cmd_wdata[31:0] =00000000
       my_periph_icb_cmd_read[0] =0

             i_cpu_cmd[31:0] =00000000
             o_cpu_cmd[31:0] =00000000

       a_write_address[14:0] =0000
          a_write_data[15:0] =0000
                final_wrea[0] =0

         hdmi_is_displaying[0] =0
        b_read_address[14:0] =0000
       b_read_data_out[15:0] =0000

                   H_sync[0] =0
                   V_sync[0] =1
                    Red[7:0] =00
                  Green[7:0] =00
                   Blue[7:0] =00
```

Figure 5.8: A wide-view of the full SoC simulation. The periodic bursts of activity on the ICB bus (center) correspond to the firmware executing its main loop. The stability of the video synchronization signals is also visible.

Immediately after reset, before the firmware begins active control, the GPU enters its autonomous startup state. As detailed in Figure 5.9, the `HDMI` module enters its color bar diagnostic mode. The RGB outputs display a changing pattern of colors even though the ICB bus is idle. This confirms the correct, autonomous initialization of the video clocking and timing generation logic.



```
clk[0] =1
rst_n[0] =1

my_periph_icb_cmd_valid[0] =0
my_periph_icb_cmd_addr[31:0] =00000000
my_periph_icb_cmd_wdata[31:0] =00000000
my_periph_icb_cmd_read[0] =0

i_cpu_cmd[31:0] =00000000
o_cpu_cmd[31:0] =00000000

a_write_address[14:0] =0000
a_write_data[15:0] =0000
final_wrea[0] =0

hdmi_is_displaying[0] =1
b_read_address[14:0] =0000
b_read_data_out[15:0] =0000

H_sync[0] =0
V_sync[0] =0
Red[7:0] =FF
Green[7:0] =FF
Blue[7:0] =FF
```

Figure 5.9: The GPU's initial color bar mode. The RGB outputs display a test pattern while the ICB bus from the CPU is still inactive, confirming the correct autonomous startup of the video hardware.

Once the firmware has processed an incoming character, it takes control of the GPU by initiating write transactions on the ICB bus. Figure 5.10 captures one such event in detail. This waveform is the critical link between the software and hardware domains:

- **Bus Master Action:** The E203 core's BIU asserts `my_periph_icb_cmd_valid`.
- **Address Decoding:** The address bus, `my_periph_icb_cmd_addr`, is driven with the correct value `0x10014004`.
- **Data Payload:** The write data bus, `my_periph_icb_cmd_wdata`, carries the firmware-constructed payload `0x0001FFC0`.

This single waveform provides definitive proof that the firmware is executing correctly and that the RISC-V core is successfully commanding the custom peripheral.

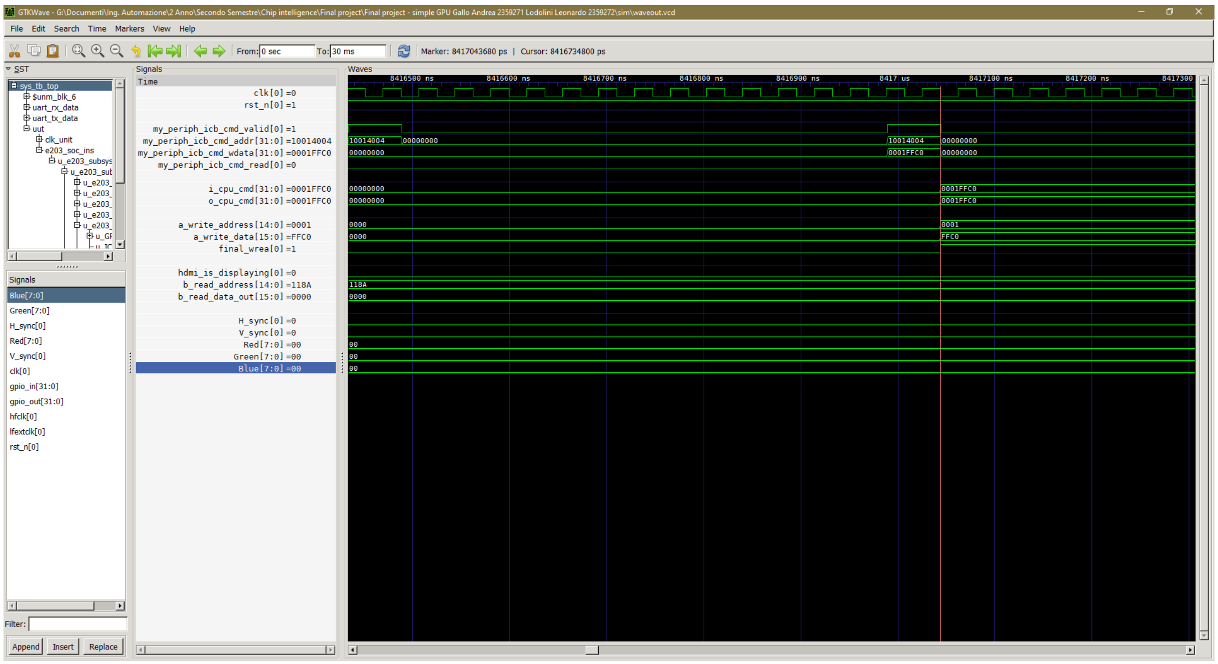
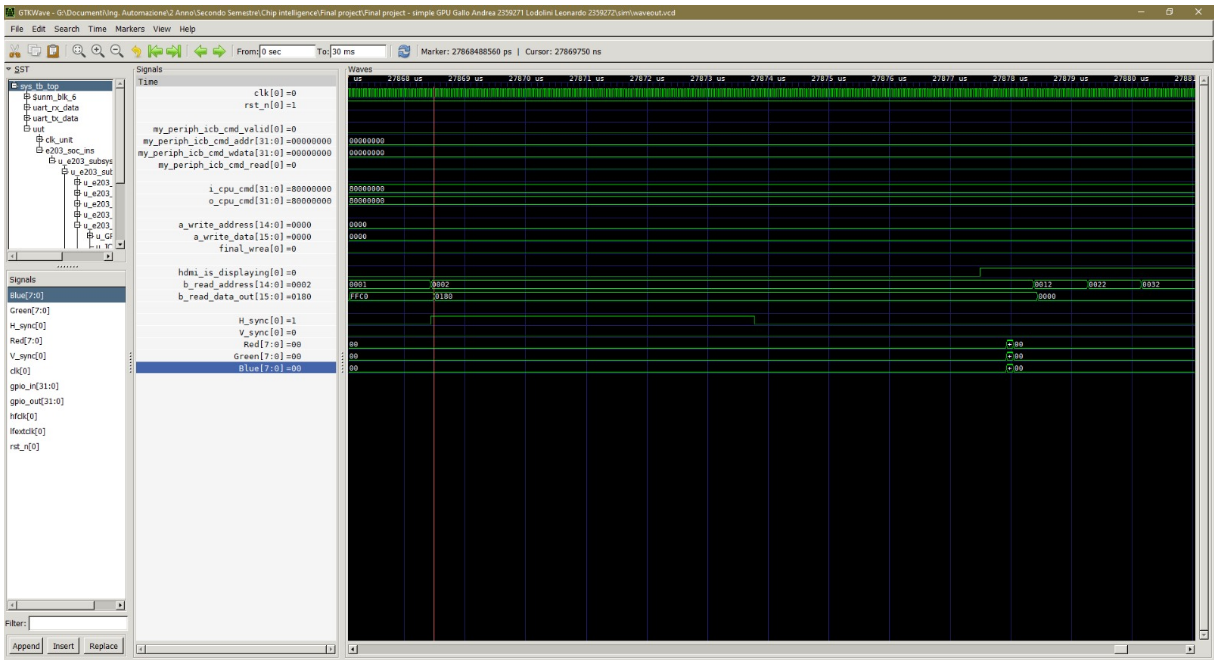


Figure 5.10: A detailed view of a single ICB write transaction initiated by the RISC-V core, demonstrating the successful communication from the CPU to the GPU peripheral.

Framebuffer Update and Final Video Rendering

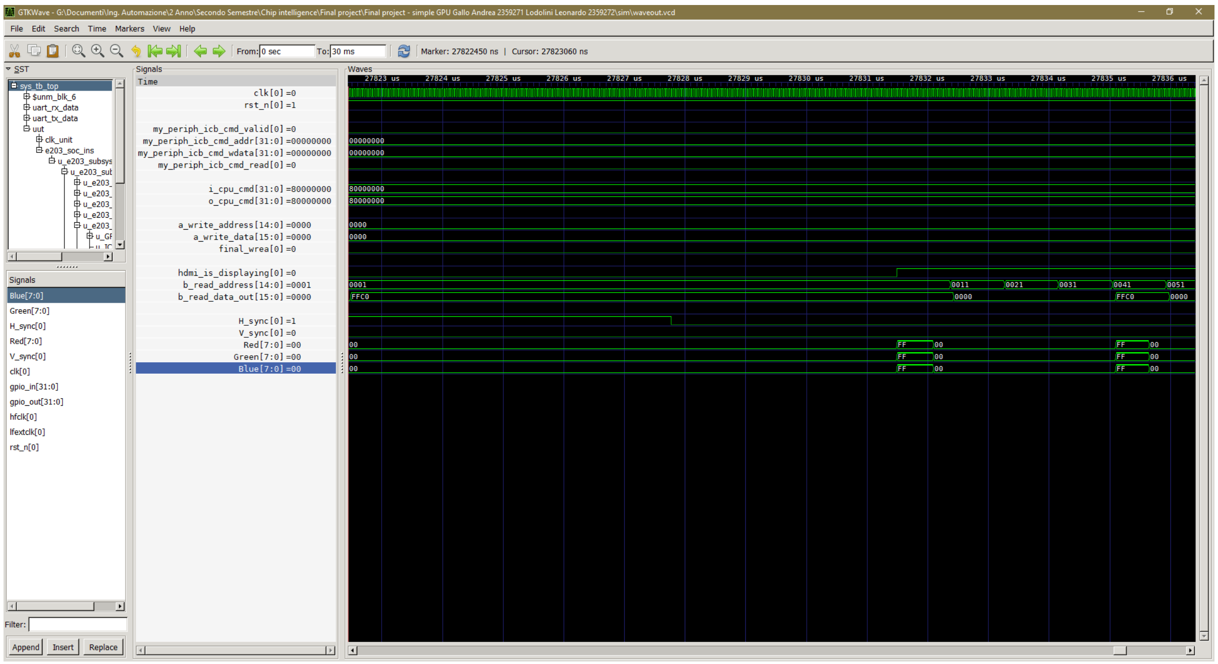
The command sent by the CPU is captured by the GPU's ICB interface and processed internally, resulting in an update to the video framebuffer. The final step is to verify that this new data is correctly rendered on the video output. The tangible result of this process is shown in Figure 5.11. The moment the `HDMI` module's raster scan reaches the screen coordinates corresponding to the new character, the data read from the framebuffer is no longer zero. The `b_read_data_out` bus now carries the character's pixel data (`0xFFC0`), and the RGB outputs immediately change from black to white, beginning the process of drawing the character on the screen.



```
GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\sim\waveout.vcd
From: 0 sec   To: 30 ms   Marker: 27868488560 ps  |  Cursor: 27869750 ns

                               clk[0] =0
                             rst_n[0] =1

           my_periph_icb_cmd_valid[0] =0
     my_periph_icb_cmd_addr[31:0] =00000000
    my_periph_icb_cmd_wdata[31:0] =00000000
            my_periph_icb_cmd_read[0] =0

                 i_cpu_cmd[31:0] =80000000
                 o_cpu_cmd[31:0] =80000000

             a_write_address[14:0] =0000
                a_write_data[15:0] =0000
                        final_wrea[0] =0

                hdmi_is_displaying[0] =0
              b_read_address[14:0] =0002
             b_read_data_out[15:0] =0180

                            H_sync[0] =1
                            V_sync[0] =0
                              Red[7:0] =00
                            Green[7:0] =00
                             Blue[7:0] =00
```

Figure 5.11: The effect of the new data on the final video output. As the data for the character row (0xFFC0) is read from the framebuffer, the RGB signals change from black to white.

Finally, Figure 5.12 provides the conclusive, micro-level evidence that closes the validation loop. It confirms the direct, clock-cycle-accurate correlation between the data in memory and the pixels being drawn on screen.



```
GTKWave - G:\Documenti\Ing. Automazione\2 Anno\Secondo Semestre\Chip intelligence\Final project\Final project - simple GPU Gallo Andrea 2359271 Lodolini Leonardo 2359272\sim\waveout.vcd
From: 0 sec   To: 30 ms   Marker: 27822450 ns  |  Cursor: 27823060 ns

                               clk[0] =0
                             rst_n[0] =1

           my_periph_icb_cmd_valid[0] =0
     my_periph_icb_cmd_addr[31:0] =00000000
    my_periph_icb_cmd_wdata[31:0] =00000000
            my_periph_icb_cmd_read[0] =0

                 i_cpu_cmd[31:0] =80000000
                 o_cpu_cmd[31:0] =80000000

             a_write_address[14:0] =0000
                a_write_data[15:0] =0000
                        final_wrea[0] =0

                hdmi_is_displaying[0] =0
              b_read_address[14:0] =0001
             b_read_data_out[15:0] =0000

                            H_sync[0] =1
                            V_sync[0] =0
                              Red[7:0] =00
                            Green[7:0] =00
                             Blue[7:0] =00
```

Figure 5.12: A micro-level view correlating read data to pixel output. The data 0xFFC0 is read on `b_read_data_out`, and the RGB signals instantly reflect this pattern by generating white pixels. This direct correlation is the ultimate proof of the system's correct operation.

This complete, traceable sequence —from a firmware-driven ICB command to the generation of corresponding colored pixels— serves as the final and conclusive validation of the entire project, demonstrating that all hardware and software components are correctly integrated and fully functional.

Chapter 6

Conclusions and Reflections

This report has documented the complete design, implementation, and rigorous verification of a custom GPU subsystem, a project that proved to be a significant technical challenge from its inception to its final validation. The objective was not merely to create a hardware module for rendering characters, but to successfully integrate it into a complex System-on-Chip ecosystem, navigating numerous obstacles in both hardware design and system-level simulation. This final chapter summarizes the project's achievements, provides a candid reflection on the most formidable challenges encountered, and explores the potential future of this work.

6.1 Summary of Project Achievements

Despite the significant challenges, all primary project objectives were successfully met, culminating in a fully functional and thoroughly verified video display solution. The successful outcome of this demanding endeavor can be distilled into several key accomplishments:

- **Functional Hardware Implementation:** A complete, modular GPU subsystem was successfully designed in Verilog. This includes the `ICB_bus` slave for communication with the CPU, the intelligent `dpram_adapter` for processing commands and managing the framebuffer, and the `HDMI` controller for generating precise 640x480@60Hz video timings and rendering the final pixel data.

- **Robust Firmware Development:** A corresponding C firmware was developed for the Hummingbird E203 core. This firmware implements a well-defined communication protocol, manages the UART for character input, handles the conversion of ASCII codes to bitmap data, and correctly orchestrates the hardware through a structured Finite State Machine.

- **Comprehensive Multi-Tiered Verification:** The project's correctness was ensured through a multi-layered verification strategy. This included unit-level testbenches for each individual module, subsystem-level tests to validate the integration and data paths, and a final, end-to-end co-simulation of the complete SoC. The final test successfully demonstrated the entire data flow, confirming that all hardware and software components were correctly integrated.

- **Fulfillment of Functional Requirements:** The implemented system successfully delivers all the specified functionalities, including the initial color bar diagnostic pattern on startup and the subsequent rendering of monochrome text received from the serial interface.

6.2 Key Design Decisions and Encountered Challenges

The development process was an exercise in overcoming persistent technical hurdles. While initial design decisions, such as the adoption of a character-based monochrome framebuffer to manage the limited on-chip SRAM, proved effective, the true challenge lay in the system-level integration and debugging. The journey from functional individual modules to a working integrated system was fraught with obstacles, the most significant of which are detailed below.

The System-Level Integration Deadlock

The most formidable challenge of the entire project emerged during the final full-SoC validation. While every individual hardware module (`ICB_bus`, `dpram_adapter`, `HDMI`) and every integrated sub-system had passed its verification suite flawlessly, the complete system failed to produce the expected output. Characters sent to the UART were correctly processed by the firmware and written into the framebuffer—this was confirmed by analyzing the ICB bus transactions and the DPRAM's write-port signals—yet the screen remained stubbornly blank.

This issue triggered an intensive, deep-dive debugging effort into the full SoC simulation. The root cause was eventually traced to a subtle but critical deadlock in the read-path: the data flow would progress correctly through the entire system but would halt precisely at the `b_read_data_out` port of the framebuffer. No data was being returned to the `HDMI` module for rendering. Resolving this problem was a non-trivial task, requiring a meticulous, cycle-by-cycle analysis of the interaction between the custom GPU and the pre-existing SoC bus fabric and memory controllers. Its successful resolution was a pivotal moment in the project and a profound learning experience in the complexities of full-chip integration.

The Practical Constraints of Simulation Time

A secondary, but pervasive, challenge was the sheer computational cost of the full system simulation. Executing the end-to-end testbench on the available hardware was an exercise in patience, with each simulation run taking approximately 45 minutes to complete (on old platforms). This incredibly long feedback loop dramatically increased the difficulty of debugging. A simple, one-line code change to test a hypothesis required a three-quarter-hour wait to see the result. This constraint necessitated a highly disciplined and methodical approach to debugging, forcing a greater reliance on static code analysis and careful, targeted changes to maximize the value of each time-consuming simulation run. Overcoming this practical limitation was as much a part of the challenge as solving the technical issues themselves.

The successful navigation of these colossal challenges was not merely a matter of writing code, but of persistent, systematic problem-solving, elevating this project from a design exercise to a true feat of engineering.

6.3 Future Work and Potential Enhancements

While the successful completion of the current implementation marks a significant achievement, the robust and modular architecture of the GPU also serves as an excellent foundation for numerous potential enhancements. The successful navigation of the core integration challenges provides a clear path for future explorations, which could expand the peripheral's capabilities in several key areas:

- **Color Support:** The most immediate evolution would be the introduction of color. This could be achieved by implementing a simple 8-bit color palette, allowing each character to have a distinct foreground and background. This would involve expanding the framebuffer's data width, enhancing the CPU-to-GPU command protocol to carry color information, and updating the `HDMI` module's final rendering stage.
- **Hardware Acceleration:** To further offload the RISC-V core and increase performance, several rendering tasks could be accelerated directly in hardware. A prime candidate would be the integration of a **Font ROM** within the GPU itself. This would allow the CPU to send only ASCII codes, delegating the entire bitmap lookup process to the hardware. Further acceleration could include hardware-managed cursors, efficient screen scrolling logic, and support for drawing simple 2D primitives.
- **Direct Memory Access (DMA) Integration:** For a dramatic increase in rendering performance, particularly for full-screen updates, a DMA controller could be integrated into the SoC. This would empower the CPU to command the transfer of large blocks of data—such as an entire pre-rendered frame—directly from main memory to the GPU's framebuffer, freeing the core to handle other critical tasks.

Concluding Remarks

In conclusion, this project was a journey through the entire design and verification flow of a custom peripheral for a modern RISC-V SoC. It demanded far more than just Verilog and C programming; it required a deep dive into the complexities of system-level integration, a methodical approach to debugging intractable problems, and the perseverance to overcome significant practical limitations. The result is not just a functional and reliable GPU subsystem, but a testament to the challenges and rewards of embedded system engineering. The project not only meets its initial requirements but also provides a versatile and extensible foundation, and more importantly, stands as a significant personal achievement in overcoming a colossal engineering challenge.

Acknowledgments

The authors wishes to acknowledge the assistance provided by AI tools during the preparation of this project. These AI tools were utilized for guidance on document structuring in LaTeX, help in solve some problems encountered in the development of the project design and suggestions on report presentation

Bibliography

[1] W. Green. “Beginning fpga graphics - project f.” Page content last updated 2025-01-22. (2020), [Online]. Available: https://projectf.io/posts/fpga-graphics/ (visited on 07/03/2024).