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 27, 2025

Abstract

This report presents the design, integration, and verification of a custom graphics peripheral for a System-on-Chip (SoC) based on the Hummingbird E203 RISC-V core. The primary objective was to create a functional hardware module capable of rendering ASCII characters, received via a UART interface, onto a standard 640x480@60Hz display using the HDMI protocol. The entire system was designed for implementation on a Gowin GW2A-18 FPGA.

The architecture is centered around a custom GPU subsystem that interfaces with the E203 core via the Inter-Chip Bus (ICB). Key hardware modules include a standard-compliant ICB slave interface and an intelligent DPRAM adapter, both implemented in Verilog, which manage command decoding and addressing for a dual-port block RAM (DPBRAM) instantiated from the Gowin IP core library. The final video signal is produced by an HDMI controller that handles timing generation, framebuffer reading, and pixel rendering. To manage the limited on-chip memory, a character-based rendering approach with a monochrome framebuffer was adopted. The system is controlled by a dedicated C firmware running on the RISC-V core, which handles UART communication, character-to-bitmap translation, and orchestrates the hardware via a custom 32-bit command protocol.

The project was rigorously validated through a multi-tiered verification strategy, progressing from unit-level tests to a full end-to-end SoC simulation. The final system-level verification successfully demonstrated the complete data flow: from receiving a character serially via the testbench, through processing by the firmware, to the correct rendering of the corresponding pixels on the video output. The successful implementation confirms the viability of integrating custom peripherals into the Hummingbird E203 ecosystem and provides a robust foundation for future enhancements, such as color support and hardware acceleration.

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 [2]. Figure 1.1 provides a visual representation of this fundamental scanning pattern.

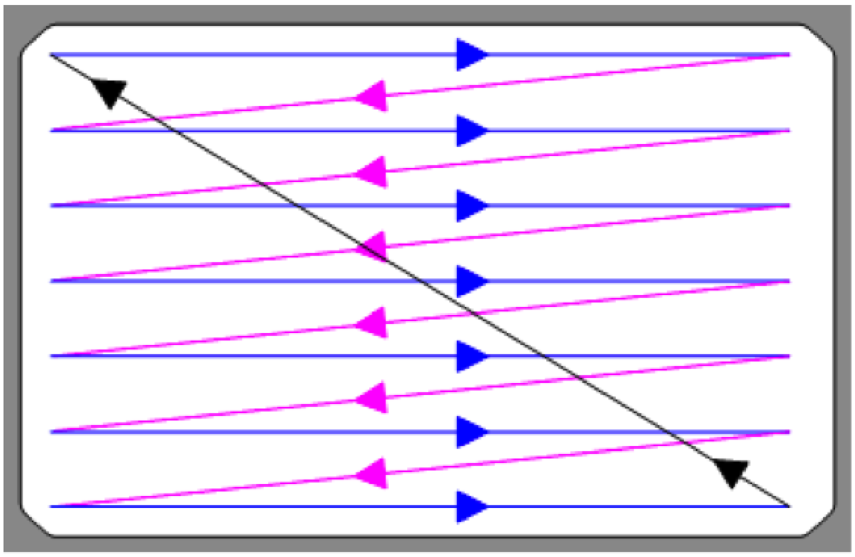


Figure 1.1: Illustration of the raster scanning process, showing the path taken to draw lines of pixels sequentially across and down the display area.

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 [2], 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 [2]. 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 [2].

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.

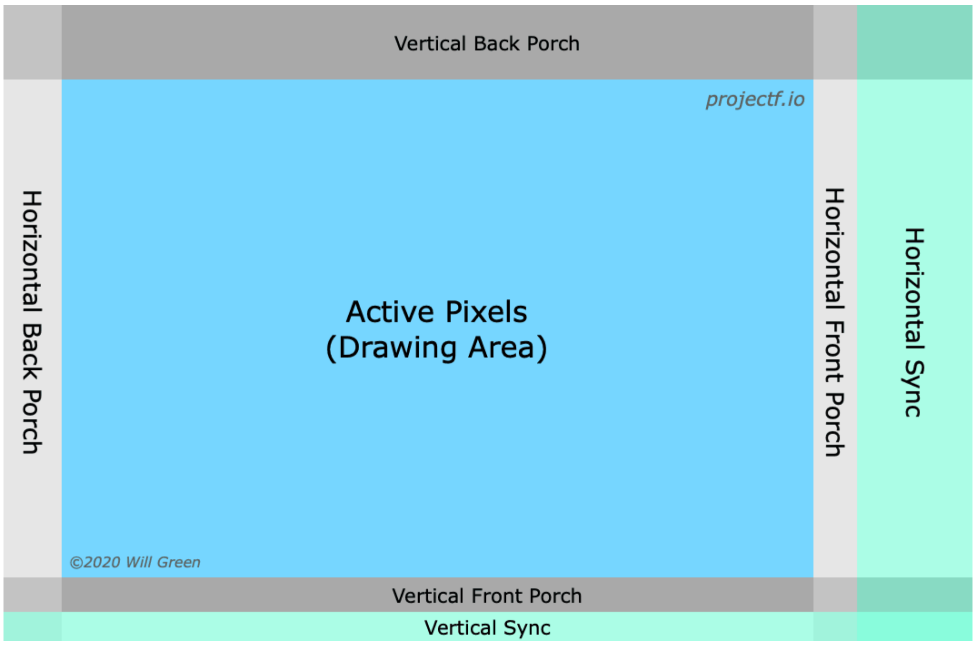


Figure 1.2: Conceptual Diagram of Video Timing Signals including Active Pixels and Blanking Intervals (Front Porch, Sync Pulse, Back Porch) for both Horizontal and Vertical Synchronization. Inspired by [2].

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.

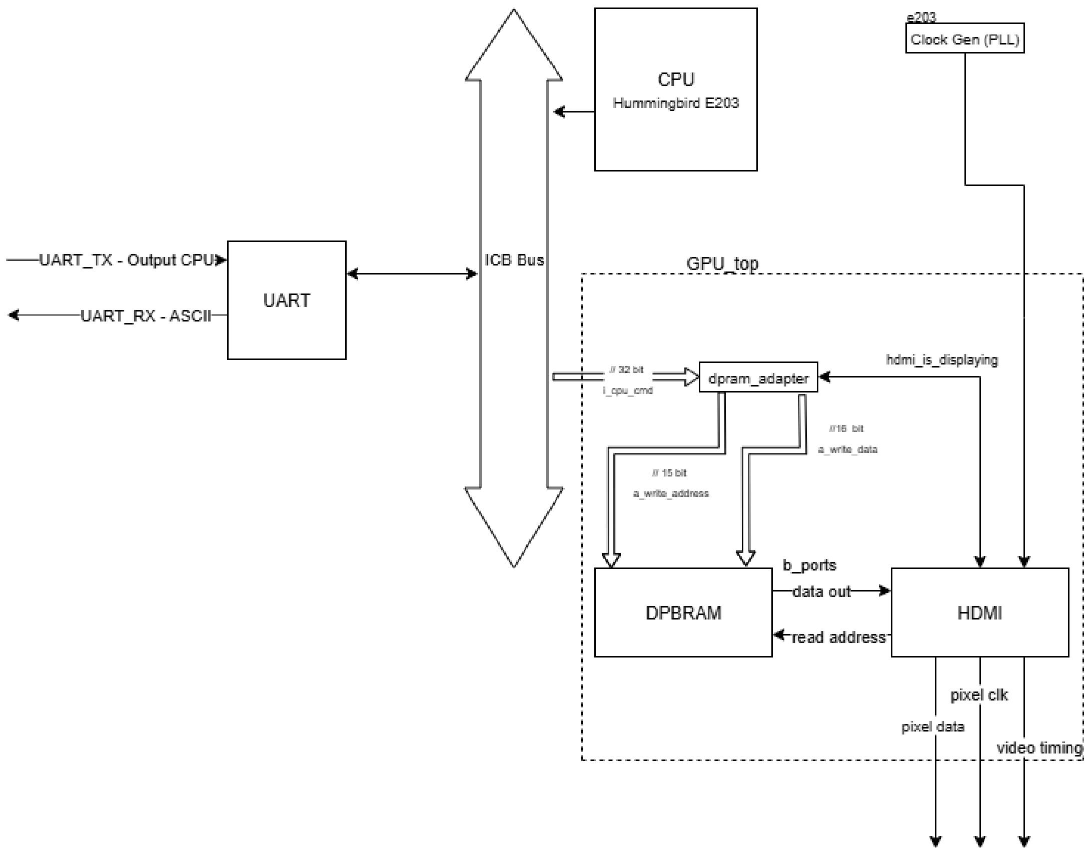


Figure 2.1: High-Level System Block Diagram illustrating the key components and data paths of the display system.

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

Detailed Implementation of Core Modules

This chapter delves into the specific hardware implementation details of the key modules that constitute our HDMI display subsystem. Building upon the architectural overview and interface descriptions provided in Chapter 2, the subsequent sections will explore the internal logic, state machines, primary signals, and operational principles of each core component. Significant Verilog code snippets from our implementation, which closely follows the functional design of the reference project [1], will be presented to illustrate the practical realization of these modules, from the ICB peripheral interface through to the final HDMI signal generation.

3.1 ICB Bus Interface: The `ICB_bus` Module

The `ICB_bus` module serves as the dedicated hardware interface between the Hummingbird E203 core's Inter-Chip Bus and our custom `GPU_top` peripheral. Its sole responsibility is to decode memory-mapped commands from the CPU, handle ICB write and read transactions to a single internal register, and forward the data to the main GPU logic. This module effectively abstracts the bus protocol details away from the core video processing tasks.

3.1.1 Module Ports and Naming Convention

The module implements a standard ICB slave interface, but with a project-specific naming convention. All ICB ports are prefixed with `GPU_` (e.g., `GPU_icb_cmd_valid`, `GPU_icb_cmd_addr`). Its primary data output is `o_cpu_cmd[31:0]`, which carries the payload to the `GPU_top` module. Internally, the module's logic is built around a core 32-bit register, `gpu_command_register`, which holds the data being transferred.

3.1.2 Address Decoding and ICB Handshake

The module is hardcoded to respond to a single, specific address defined by the `PERIPHERAL_ADDRESS` parameter (`32'h10014004`). A combinational logic block continuously checks if an incoming command from the CPU targets this specific address, generating internal `write_enable` and `read_enable` signals.

```verilog
// 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;
```

Listing 3.1: Address Decoding Logic in `ICB_bus`.

The ICB handshake logic is implemented to be robust and compliant with multi-slave bus architectures. The `GPU_icb_cmd_ready` signal is asserted only when a command is valid and it targets the peripheral's specific address, as determined by the `is_our_address` wire. This ensures the module only engages in transactions intended for it. The internal address decoding logic then determines whether the command is actually executed by this peripheral.

```verilog
// The slave signals that it is ready to accept a command only when
// the transaction is directed at its specific address. This is a robust
// approach that works correctly in a multi-slave bus environment.
assign GPU_icb_cmd_ready = is_our_address;
```

Listing 3.2: ICB Address Decoding and Handshake Logic.

3.1.3 Core Register Logic and Dataflow

The main sequential logic, implemented in an `always` block, handles the read and write operations on the internal `gpu_command_register`. This register serves as the pipeline stage that decouples the bus write operation from the GPU's consumption of the data.

```verilog
always @(posedge clk or posedge reset) begin
    if (reset) begin
        gpu_command_register        <= 32'b0;
        response_is_valid_internal <= 1'b0;
        // ... other resets ...
    end else begin
        response_is_valid_internal <= 1'b0; // Default de-assertion

        if (write_enable) begin
            // A write command is active: update our register with the
    master's data.
            gpu_command_register <= GPU_icb_cmd_wdata;
            // Signal that the write operation was successful.
            response_is_valid_internal <= 1'b1;
        end

        if (read_enable) begin
            // A read command is active: place our register's value into
     the read buffer.
            read_data_buffer <= gpu_command_register;
            // Signal that the read data is now valid on the response
    bus.
            response_is_valid_internal <= 1'b1;
        end
    end
end
```

Listing 3.3: Core Sequential Logic for Read/Write Operations.

Finally, the data path to the GPU is completed with a simple continuous assignment, ensuring the `GPU_top` module always sees the most recently written command. This design provides a robust and standard-compliant mechanism for the firmware to send 32-bit command words to the custom peripheral.

3.1.4 Verification and Simulation

A dedicated unit-level testbench, `testbench_ICB_bus`, was created to verify the functionality of the `ICB_bus` module in isolation. The testbench simulates an ideal ICB master and executes a sequence of tests to validate the module's behavior against its specification.

Testbench Design

The testbench instantiates the `ICB_bus` as the Device Under Test (DUT) and drives its inputs to perform a series of structured tests. The main test sequence is implemented within an `initial` block and covers four key scenarios:

1. **System Reset:** The testbench applies an active-low reset to ensure all internal registers of the DUT are initialized to a known state.

2. **Register Write:** A write command is sent to the peripheral's correct address (`PERIPHERAL_ADDRESS`) with a test value (`32'hDEADBEEF`). The testbench verifies that the DUT's output (`o_cpu_cmd`) reflects this new value.

3. **Register Read:** A read command is sent to the same address. The testbench verifies that the data returned on the response bus (`GPU_icb_rsp_rdata`) matches the value previously written.

4. **Incorrect Address Access:** Write and read commands are sent to a wrong address (`WRONG_ADDRESS`). The testbench verifies that while the DUT may assert `GPU_icb_cmd_ready` (due to its direct connection to `GPU_icb_cmd_valid`), its internal state and `o_cpu_cmd` output remain unchanged, and it does not generate a valid response (`GPU_icb_rsp_valid`).

The following snippet shows the structure of the write-test portion of the testbench.

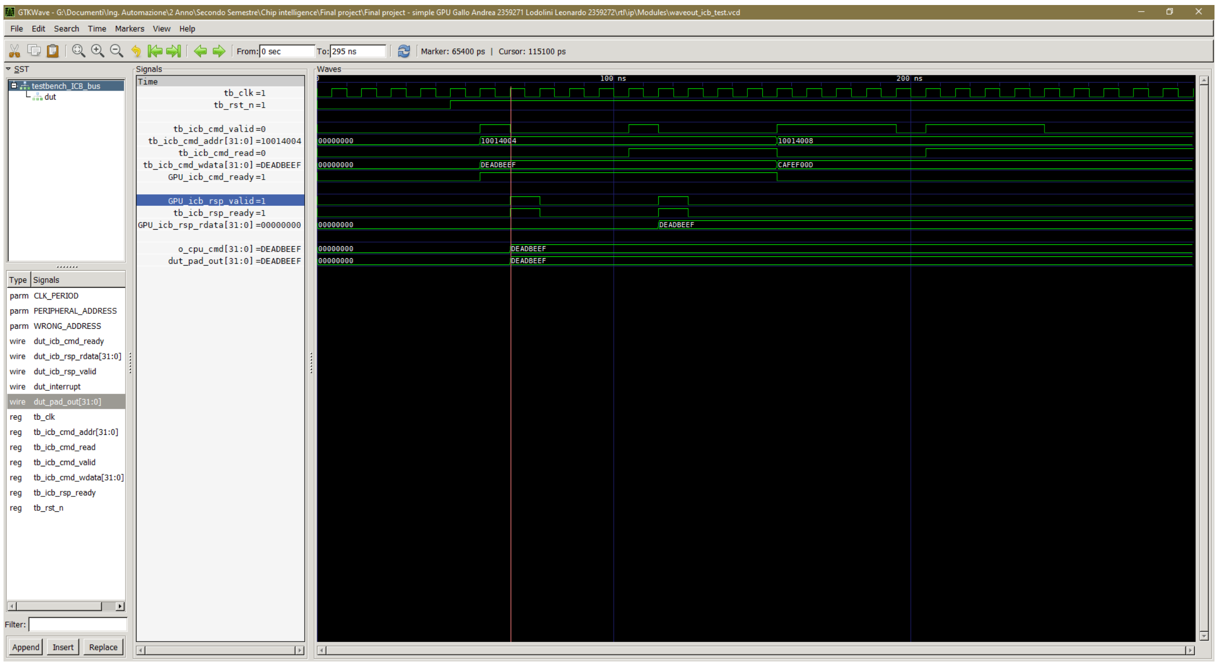
```
// --- TEST 1: Write to the correct address ---
$display("[TB] TEST 1: Writing 0xDEADBEEF...");
// Start command
tb_icb_cmd_valid <= 1'b1;
tb_icb_cmd_addr  <= PERIPHERAL_ADDRESS;
tb_icb_cmd_read  <= 1'b0; // This is a write
tb_icb_cmd_wdata <= 32'hDEADBEEF;

// Wait for the peripheral to be ready to accept the command.
// For a correct address, this should happen in the same clock cycle.
wait (dut_icb_cmd_ready);
@(posedge tb_clk);
tb_icb_cmd_valid <= 1'b0; // De-assert command after it's taken.
```

```

// The peripheral has a single-cycle response. Be ready to accept it.
tb_icb_rsp_ready <= 1'b1;
wait (dut_icb_rsp_valid);
@(posedge tb_clk);
tb_icb_rsp_ready <= 1'b0; // De-assert after capturing response.

// Verification
if (dut_pad_out === 32'hDEADBEEF) begin
    $display("[TB] SUCCESS: o_cpu_cmd is now 0x%h.", dut_pad_out);
end else begin
    $error("[TB] FAILURE: o_cpu_cmd is 0x%h.", dut_pad_out);
    $finish;
end
```

Listing 3.4: Write Test Sequence in `testbench_ICB_bus`.

Waveform Analysis

The simulation results, shown in Figure 3.1 and Figure 3.2, confirm the correct operation of the `ICB_bus` module.



```
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

SST
testbench_ICB_bus
  dut

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

Signals
Time
                  tb_clk =1
                tb_rst_n =1

        tb_icb_cmd_valid =0
  tb_icb_cmd_addr[31:0] =10014004      00000000 → 10014004 → 10014008
         tb_icb_cmd_read =0
 tb_icb_cmd_wdata[31:0] =DEADBEEF      00000000 → DEADBEEF → CAFEF00D
        GPU_icb_cmd_ready =1

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

        o_cpu_cmd[31:0] =DEADBEEF      00000000 → DEADBEEF
      dut_pad_out[31:0] =DEADBEEF      00000000 → DEADBEEF
```

Figure 3.1: Simulation waveform showing a successful write transaction followed by a successful read transaction to the correct peripheral address (`0x10014004`).

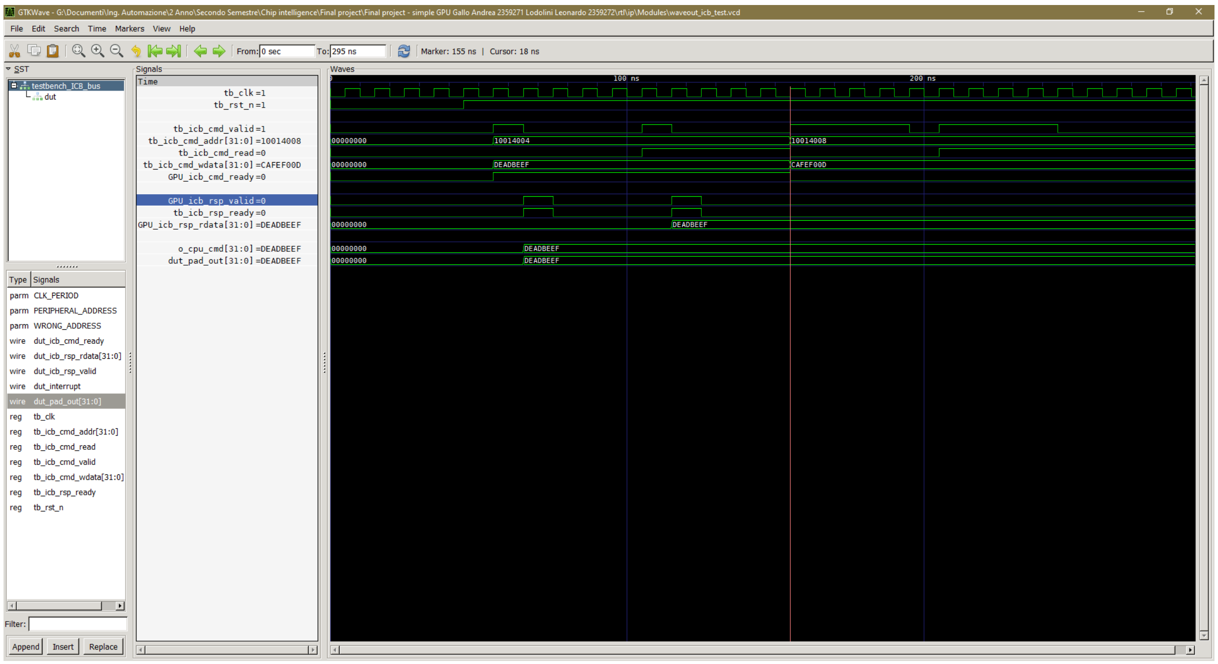
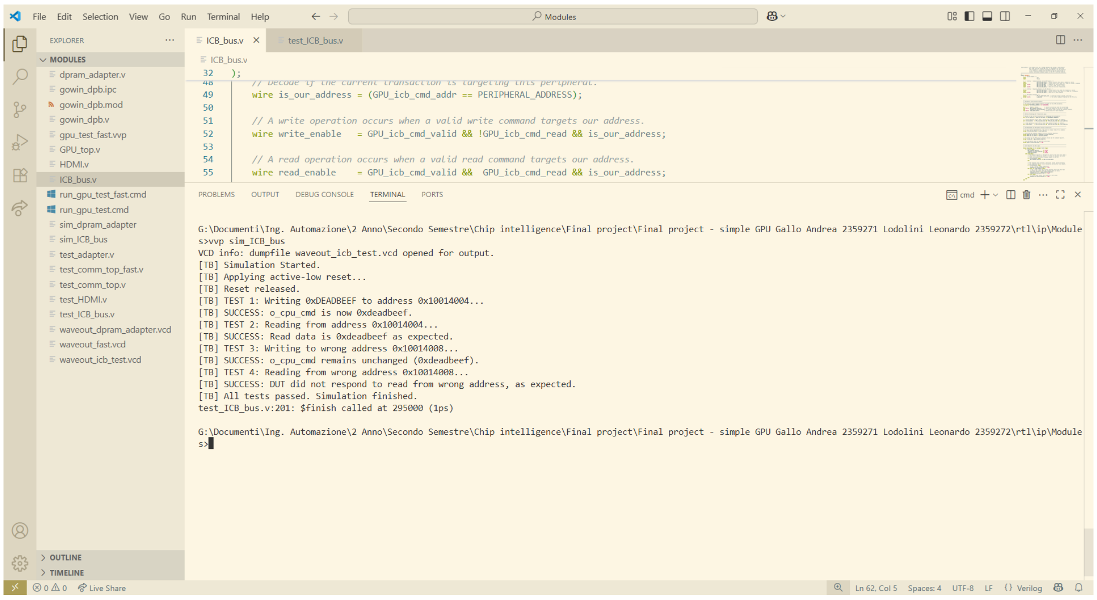


Figure 3.2: Simulation waveform showing the module's correct behavior when accessed with an incorrect address (`0x10014008`). The peripheral correctly ignores the transactions.

The successful execution of all test scenarios is confirmed by the simulation log output, shown in Figure 3.3, which explicitly states that all assertions passed.



```
);
// 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 3.3: Terminal output from the Icarus Verilog simulation, confirming that all test cases (write, read, and incorrect address access) completed successfully.

The waveform analysis reveals the following key behaviors, executed sequentially by the testbench:

- **Write Operation (Figure 3.1):** The testbench initiates a write by asserting `tb_icb_cmd_valid` and driving the correct peripheral address (`0x10014004`) and data (`0xDEADBEEF`). The DUT correctly asserts `GPU_icb_cmd_ready` in the same cycle. On the next rising edge of `tb_clk`, the value is latched internally and appears on the `o_cpu_cmd` output. The DUT then asserts `GPU_icb_rsp_valid` for one cycle to acknowledge the successful write.

- **Read Operation (Figure 3.1):** Following the write, the testbench issues a read command to the same address. The DUT again asserts `GPU_icb_cmd_ready`. On the next clock cycle, it places the content of its internal register (`0xDEADBEEF`) onto the `GPU_icb_rsp_rdata` bus and asserts `GPU_icb_rsp_valid`. The testbench confirms the read data is correct.

- **Incorrect Address Access (Figure 3.2):** The testbench then attempts a write transaction to a wrong address (`0x10014008`) with data `0xCAFEF00D`. The waveform clearly shows that the DUT does not assert `GPU_icb_cmd_ready`. Consequently, the internal state and the `o_cpu_cmd` output remain unchanged, still holding the value `0xDEADBEEF`. The DUT also generates no response (`GPU_icb_rsp_valid` stays low). This confirms the address decoding logic works as specified, preventing the peripheral from erroneously processing transactions intended for other modules. The subsequent read attempt to the same wrong address is similarly ignored.

The simulation results successfully validate the `ICB_bus` module's compliance with the required protocol and its readiness for integration into the larger system.

3.2 DPRAM Write Controller: The `dpram_adapter` Module

The `dpram_adapter` module is the intelligent core of the write-path to the framebuffer. It receives the 32-bit command words from the `ICB_bus` via its `i_cpu_cmd` input and translates them into specific write operations for the Dual-Port RAM. Far from being a simple splitter, this module employs a Finite State Machine (FSM) and sophisticated decoding logic to manage character positioning, cursor control, and special commands like "Enter" and "Delete".

3.2.1 State Machine and Command Processing

The module's operation is governed by a 3-state FSM designed to handle the flow of character data correctly. The states are:

- `S_IDLE`: The initial state. The FSM waits for the first valid character data command before transitioning to the active state.

- `S_RUN`: The normal operational state. In this state, the FSM processes incoming data commands. Its main task is to monitor for the "end-of-character" command. This command, explicitly sent by the firmware as a full 32-bit zero value after the 16 character rows, signals that the character transmission is complete and the cursor position should be advanced.

- `S_POST_INCREMENT`: A transient state entered after processing an "end-of-character" command. It serves to prevent the same command from being processed multiple times if it remains on the input bus for more than one clock cycle, thus ensuring the character position is incremented only once per character. The FSM returns to `S_RUN` as soon as new data appears.

Crucially, the module also processes high-priority commands that override the normal FSM flow. These commands, encoded in the most significant bits of `i_cpu_cmd`, are handled immediately:

- **Reset (`i_cpu_cmd[31]`):** A synchronous reset that brings all internal registers, including the character position, to their initial state.
- **Delete (`i_cpu_cmd[30]`):** Sets an internal `delete_mode_active` flag. When this flag is active, the subsequent "end-of-character" signal will cause the module to clear the `delete_mode_active` flag without incrementing the `character_position` register. This prevents the cursor from advancing, a behavior leveraged by the firmware to implement a backspace-and-overwrite sequence.
- **Enter (`i_cpu_cmd[29]`):** Calculates the address for the start of the next line and updates the `character_position` register accordingly.

The core sequential logic implementing this FSM and command handling is shown below.

```verilog
// This excerpt shows the case statement for the FSM and the
// preceding high-priority command checks.

always @(posedge clk or posedge reset) begin
    if (reset) begin
        // ... Reset logic ...
    end else begin
        // --- High-Priority Command Processing ---
        if (i_cpu_cmd[31]) begin
            // Synchronous Reset logic
        end else if (i_cpu_cmd[30]) begin
            // Delete command logic
        end else if (i_cpu_cmd[29]) begin
            // Enter command logic
        end else begin
            // --- FSM State-Based Flow Control ---
            case (state)
                S_IDLE:           begin /* ... */ end
                S_RUN:            begin /* ... */ end
                S_POST_INCREMENT: begin /* ... */ end
                default:          begin /* ... */ end
            endcase
        end
    end
end
```

Listing 3.5: FSM and High-Priority Command Logic in `dpram_adapter`.

3.2.2 DPRAM Address and Data Generation

The output logic of the `dpram_adapter` is combinational, deriving the final write address and data for the DPRAM from the input command and internal state registers.

- **Write Enable:** The `write_enable` signal is asserted only when a valid data command is present (i.e., the upper 16 bits are non-zero) and none of the high-priority command flags are set.
- **Write Data:** The 16-bit data to be written, `o_dpram_wdata`, is simply the lower 16 bits of the input command, `i_cpu_cmd[15:0]`.
- **Write Address:** The 15-bit write address, `o_dpram_addr`, is calculated dynamically. It combines the base address of the current character cell, stored in the `character_position` register, with the 13-bit row offset provided by the firmware in `i_cpu_cmd[28:16]`.

The Verilog implementation for these output assignments is shown below.

```verilog
// The write_enable is active only if i_cpu_cmd contains valid data
// and is not a special command flag.
assign write_enable = (i_cpu_cmd[31:16] != 16'h0000) &&
                      !i_cpu_cmd[31] && !i_cpu_cmd[30] && !i_cpu_cmd
    [29];

// The final write address is calculated from the character's base
    position
// and the 4-bit row offset provided in the command.
// Base Address = character_position * 16
// Row Offset   = i_cpu_cmd[19:16]
assign o_dpram_addr = (character_position * 16) + i_cpu_cmd[19:16];

// The write data is the lower 16 bits of the CPU command.
assign o_dpram_wdata = i_cpu_cmd[15:0];
```

Listing 3.6: Combinational Output Logic in `dpram_adapter`.

This intelligent adapter module is therefore central to the system's ability to render a full screen of text and handle terminal-like interactions, translating high-level commands from the firmware into precise memory write operations.

3.2.3 Verification and Simulation

The correct behavior of the `dpram_adapter` is critical for the entire display system. A dedicated unit-level testbench, `testbench_adapter`, was developed to validate its functionality under a sequence of realistic command scenarios.

Testbench Design

The testbench simulates the CPU by driving the `i_cpu_cmd` input of the DUT. It uses a helper task, `write_character_unrolled`, to emulate the process of sending the 16 data rows of a character followed by the end-of-character command. The main test sequence, implemented in an `initial` block, covers key operations such as sequential writes and special commands. The following snippet illustrates the test sequence for the "Delete" and "Enter" commands.

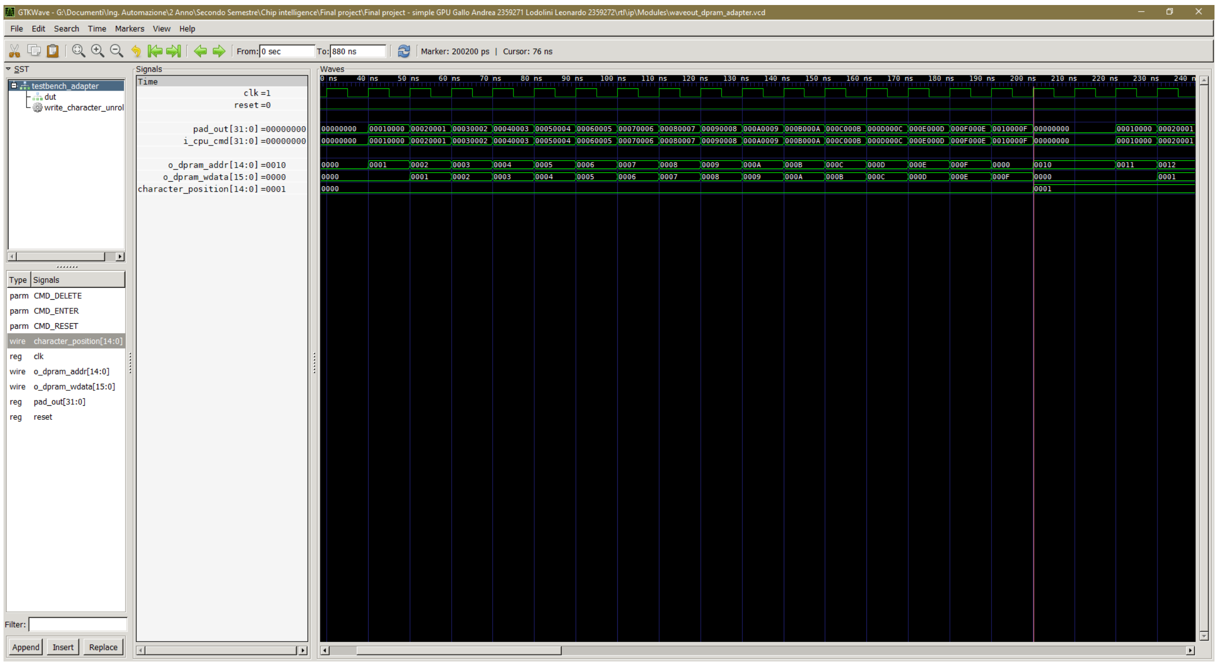
```verilog
// --- Scenario 2: Press 'Delete' then write a new character ---
$display("TEST: Pressing Delete key. Cursor should move from 2 to 1.");
@(posedge clk); pad_out = CMD_DELETE;
// The DUT will now be in 'delete mode'.

$display("TEST: Writing character 'C' (overwrite at pos 1).");
write_character_unrolled(0); // Expected final position: 2

// --- Scenario 3: Press 'Enter' then write a new character ---
$display("TEST: Pressing Enter key. Cursor moves to next line (pos 40)."
    );
@(posedge clk); pad_out = CMD_ENTER;

$display("TEST: Writing character 'D' on the new line (at pos 40).");
write_character_unrolled(0); // Expected final position: 41
```

Listing 3.7: Test Sequence for Special Commands in `testbench_adapter`.

The testbench probes the internal `character_position` register of the DUT using a hierarchical reference. This provides direct visibility into the module's state transitions without altering its port list, allowing for precise verification of the cursor management logic.

Waveform Analysis

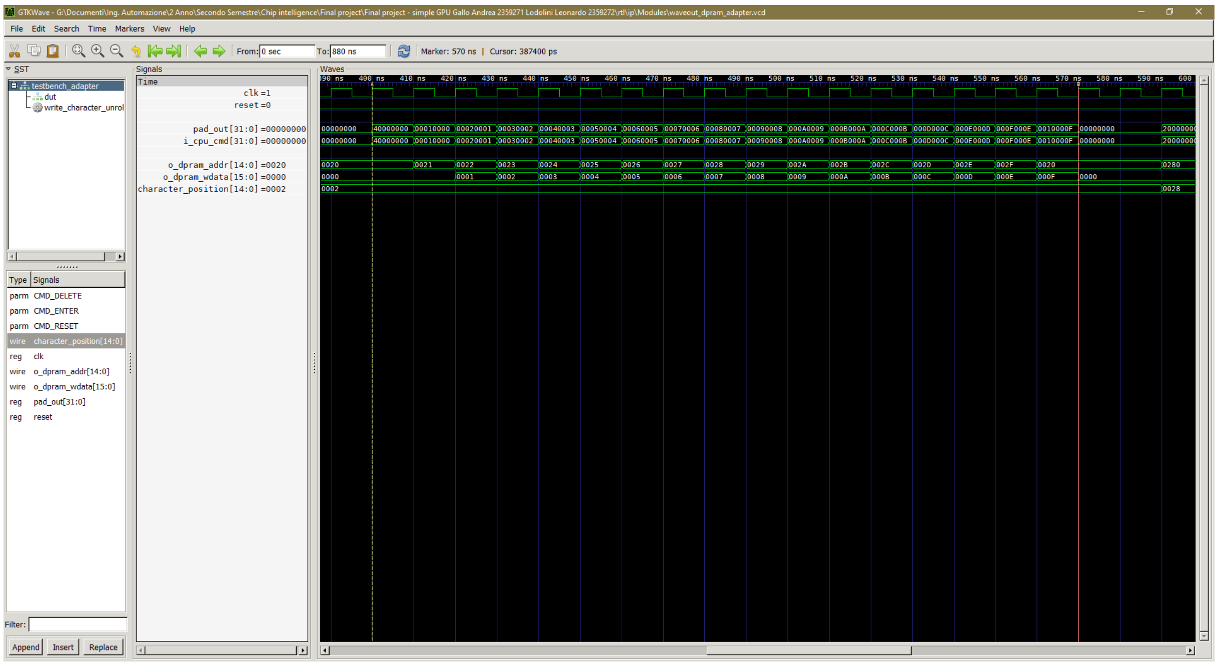
The simulation waveforms, shown in Figures 3.4, 3.5, and 3.6, confirm the successful execution of these tests.


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: 200200 ps | Cursor: 76 ns
SST
testbench_adapter
dut
write_character_unrol
Signals
Waves
Filter:
Append
Insert
Replace

```
Time
                    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
```

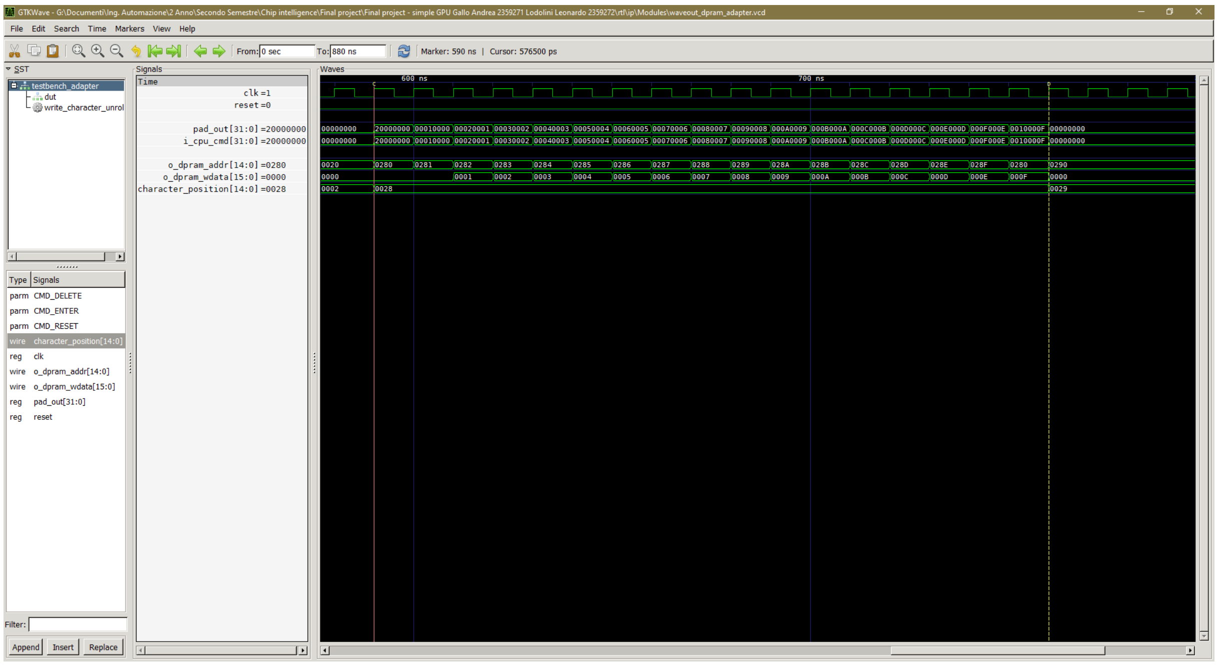
Type	Signals
parm	CMD_DELETE
parm	CMD_ENTER
parm	CMD_RESET
wire	character_position[14:0]
reg	clk
wire	o_dpram_addr[14:0]
wire	o_dpram_wdata[15:0]
reg	pad_out[31:0]
reg	reset

Figure 3.4: Simulation waveform showing the write of the first character ('A'). As the 16 data rows are sent, the `o_dpram_addr` increments. After the final end-of-character command, the internal `character_position` advances from 0 to 1.



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

Figure 3.5: Waveform illustrating the delete-and-overwrite sequence. After writing character 'B' (which moves `character_position` to 2), a `FLAG_DELETE` command is issued. The next character ('C') is then written starting at the same base address as 'B', and `character_position` increments to 2 again, confirming the overwrite logic.



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

Figure 3.6: Waveform for the 'Enter' command. After the previous operation, the `character_position` is 2. The `FLAG_ENTER` command (value `0x20000000`) causes the `character_position` to jump to 40 (`0x28`), correctly positioning the cursor at the start of the next line.

The analysis of the waveforms demonstrates that the module behaves exactly as specified:

- **Standard Write (Figure 3.4):** During the transmission of a character's 16 rows, the `o_dpram_addr` correctly sweeps through the 16 memory locations for the current character cell. The `o_dpram_wdata` reflects the data being sent. Upon receiving the end-of-character command (value 0), the internal `character_position` register is correctly incremented.

- **Delete Operation (Figure 3.5):** The testbench first writes character 'B', which moves the `character_position` to 2. It then sends the `FLAG_DELETE` command. When the next character ('C') is written, the waveform shows that the write addresses start from the base address of position 1, effectively overwriting 'B'. After character 'C' is fully written, the `character_position` correctly advances to 2, demonstrating the non-incrementing behavior of the delete sequence.

- **Enter Operation (Figure 3.6):** The `FLAG_ENTER` command is shown to have an immediate effect. The `character_position` register, which was previously 2, is updated to 40 (`0x28` in hex), correctly reflecting the logic for moving to the start of the next line on a 40-character-wide screen.

These results confirm that the `dpram_adapter` FSM and command decoding logic are implemented correctly and robustly, ready for system integration.

3.3 Video Signal Generator: The `HDMI` Module

The `HDMI` module is the final and most critical stage in the hardware pipeline. It is responsible for generating all the necessary video timing signals for a 640x480@60Hz display, reading pixel data from the DPRAM, and outputting a 24-bit RGB data stream. The module operates in two distinct modes: an initial color bar test pattern mode, followed by a monochrome text rendering mode.

3.3.1 VGA Timing and Position Generation

The core of the module consists of two free-running counters, `h_pos_counter` and `v_pos_counter`. These counters continuously scan the screen dimensions, which are based on the VESA 640x480@60Hz standard but have been adjusted. As noted in the implementation, the total pixel counts per line (`H_TOTAL`) and per frame (`V_TOTAL`), along with the blanking intervals, have been slightly increased to compensate for a system clock frequency that is not an exact multiple of the standard 25.175 MHz pixel clock. This tuning ensures that the refresh rate remains close to 60 Hz, maintaining compatibility with most displays.

```verilog
// This block implements the core horizontal and vertical counters that
// continuously scan the screen.
always @(posedge clk or posedge reset) begin
    if (reset) begin
        h_pos_counter <= 12'd0;
        v_pos_counter <= 12'd0;
    end else begin
        if (h_pos_counter < H_TOTAL - 1) begin
```

```verilog
            h_pos_counter <= h_pos_counter + 1;
        end else begin
            h_pos_counter <= 12'd0;
            if (v_pos_counter < V_TOTAL - 1) begin
                v_pos_counter <= v_pos_counter + 1;
            end else begin
                v_pos_counter <= 12'd0;
            end
        end
    end
end
```

Listing 3.8: Core Timing Counters in the `HDMI` Module.

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

```verilog
// The sync signals are active-high during the pulse duration.
assign o_hsync = (h_pos_counter < H_SYNC_PULSE);
assign o_vsync = (v_pos_counter < V_SYNC_PULSE);

// True if the current pixel is within the visible display area.
wire is_in_active_area =
    (h_pos_counter >= H_SYNC_PULSE + H_BACK_PORCH) &&
    (h_pos_counter < H_SYNC_PULSE + H_BACK_PORCH + H_DISPLAY) &&
    (v_pos_counter >= V_SYNC_PULSE + V_BACK_PORCH) &&
    (v_pos_counter < V_SYNC_PULSE + V_BACK_PORCH + V_DISPLAY);
```

Listing 3.9: Sync and Active Area Signal Generation.

3.3.2 Dual-Mode Operation: Color Bars and Text

A timer, `mode_switch_timer`, controls the transition between the two display modes. Upon reset, the module enters color bar mode for a brief, fixed duration defined by `COLOR_BAR_DURATION_CYCLES` (50,000 clock cycles). This short interval is sufficient to serve as a quick visual diagnostic, confirming that the video output and timings are active before the system switches to its primary text-rendering function. After the timer expires, it permanently switches to text display mode by setting the `text_display_mode` flag.

- **Color Bar Mode:** A dedicated set of counters (`colorbar_pixel_counter` and `colorbar_strip_counter`) divides the screen into eight vertical strips. A combinational `case` statement maps the current strip to a predefined 24-bit RGB color.
- **Text Display Mode:** This mode is active when `text_display_mode` is high. The module calculates the address for the DPRAM, reads the 16-bit pixel data, and renders it.

A final multiplexer selects the output based on the current mode. If in color bar mode, the output is the current strip's color. If in text mode, the output is the monochrome pixel color inside the active area, and black during blanking intervals.

```verilog
// This logic selects the final pixel color based on the current display
    mode.
assign final_pixel_data =
    text_display_mode == 0 ? colorbar_current_color :
    is_in_active_area     ? monochrome_pixel_color :
                             C_BLACK;

// The final 24-bit pixel data is split into the three color channels.
assign o_red   = final_pixel_data[7:0];
assign o_green = final_pixel_data[15:8];
assign o_blue  = final_pixel_data[23:16];
```

Listing 3.10: Final Pixel Data Multiplexer.

3.3.3 DPRAM Read-Path Logic

In text mode, the module must generate the correct read address for the DPRAM (`pixel_addr_out`) and process the returned data (`pixel_data_in`).

- **Address Generation:** The read address is calculated combinationally from a set of dedicated counters that track the character's position on the screen (`screen_char_row/col_counter`) and the pixel's position within the character cell (`char_pixel_row/col_counter`). This ensures that the correct 16-pixel row is requested from the DPRAM for every screen location.

- **Pixel Serialization:** The 16-bit word read from the DPRAM, `pixel_data_in`, represents a full horizontal slice of a character. A single bit from this word must be selected for the current pixel. Due to the DPRAM's one-cycle read latency, a delayed version of the horizontal pixel counter, `char_pixel_col_delayed`, is used to select the correct bit from the just-arrived data. The selection logic, `pixel_data_in[15 - char_pixel_col_delayed]`, also accounts for the font data format, where the leftmost pixel of a character corresponds to the most significant bit (MSB) of the 16-bit word. The resulting single bit is combinationally assigned to the `pixel_is_on` wire.

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

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

3.3.4 Verification and Simulation

Verification of the `HDMI` module was performed through an observational testbench, `testbench_HDMI`. The primary goal was to generate a comprehensive waveform trace to allow for manual inspection of the module's behavior across its different operational modes.

Testbench Design

The testbench is intentionally simple. It instantiates the DUT, provides a clock and reset signal, and ties the `pixel_data_in` input to a constant value (`16'hAAAA`). This alternating bit pattern makes it easy to visualize the rendering of text pixels in the waveform viewer.

The simulation runs for a duration sufficient to capture several full video frames, allowing for the observation of:

1. The initial color bar test pattern.
2. The automatic transition from color bar mode to text rendering mode.
3. The continuous and stable generation of synchronization signals (`Hsync`, `Vsync`).
4. The correct generation of DPRAM read addresses (`pixel_addr_out`) that sequentially scan the entire framebuffer.

Waveform Analysis

Visual inspection of the generated waveform file confirms that all aspects of the module function as expected. The key operational phases are illustrated in Figures 3.7 through 3.10.

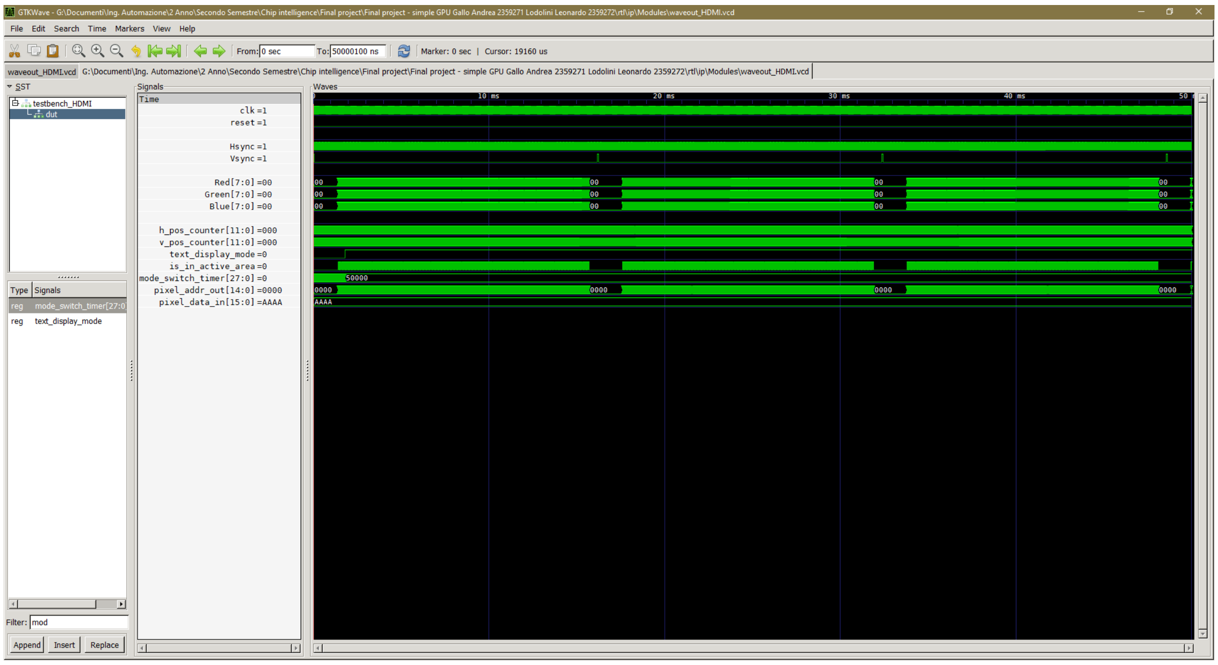
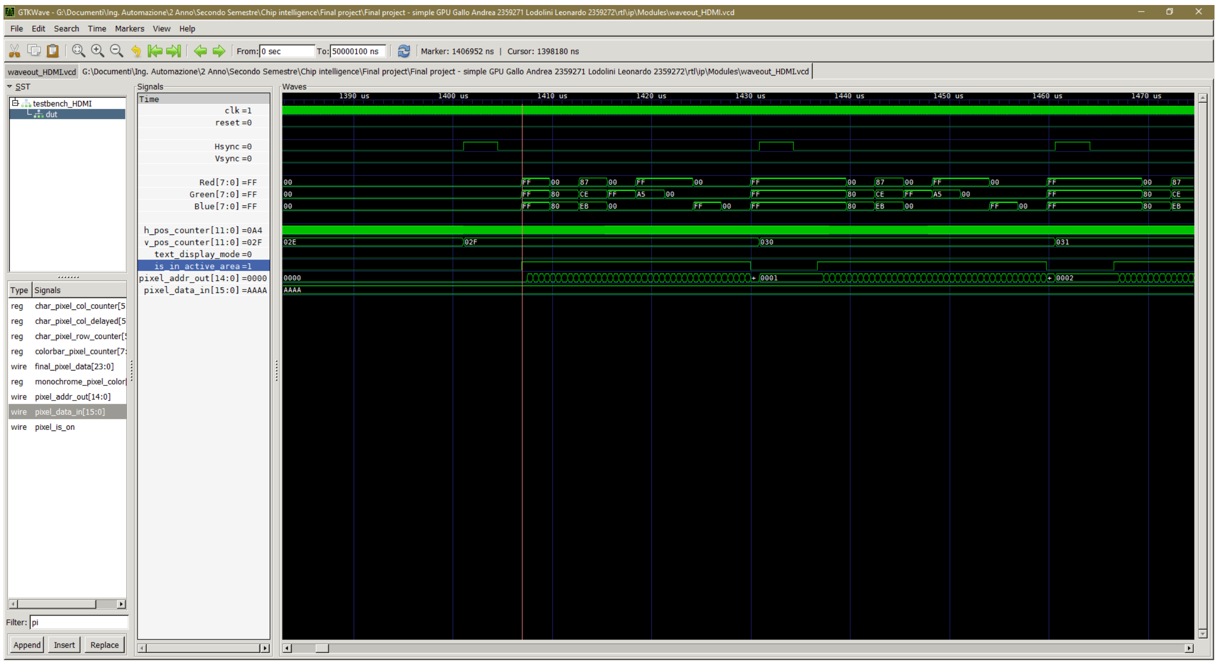
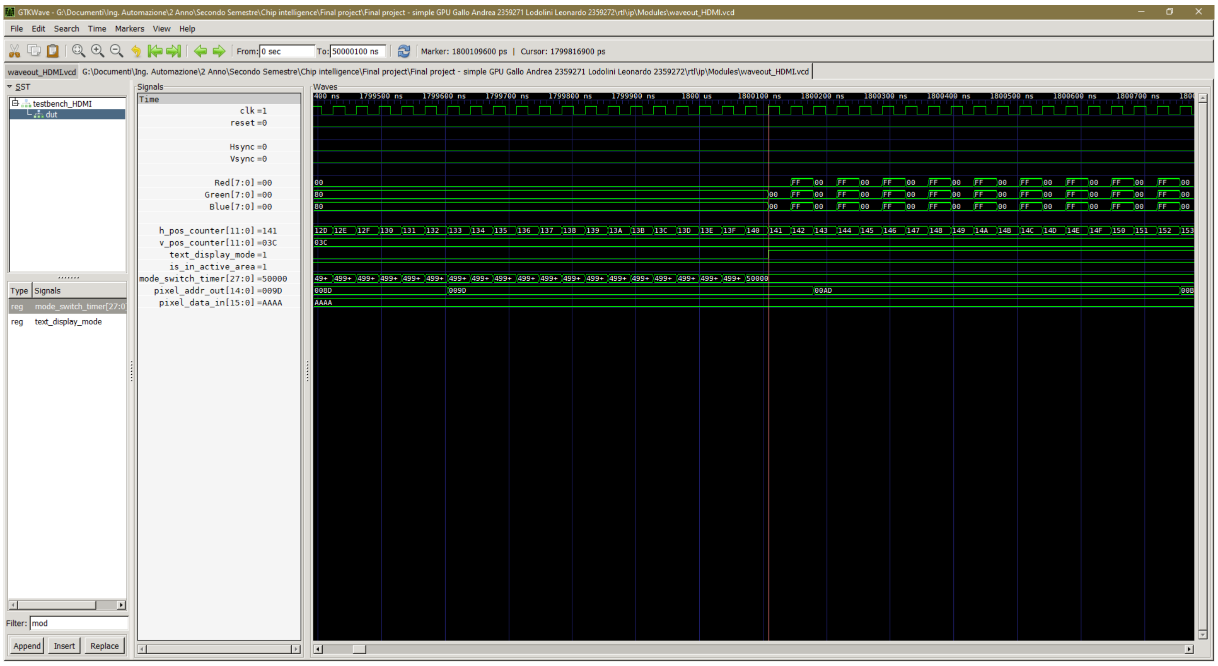


Figure 3.7: A wide view of the simulation showing multiple video frames. The periodic nature of the `Vsync` signal and the stable behavior of the position counters (`h_pos_counter`, `v_pos_counter`) over time are clearly visible.



```
Marker: 1406952 ns  |  Cursor: 1398180 ns

Time
                 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 3.8: A zoomed-in view of the initial color bar mode. The RGB output signals (`Red`, `Green`, `Blue`) change as the horizontal position counter advances, generating the distinct vertical color strips. The `text_display_mode` flag is low.



```
Marker: 1800109600 ps  |  Cursor: 1799816900 ps

Time
                     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 3.9: The transition from color bar mode to text mode. The `mode_switch_timer` reaches its target value (50000), causing the `text_display_mode` flag to go high. The RGB output immediately changes from colored strips to a monochrome pattern.

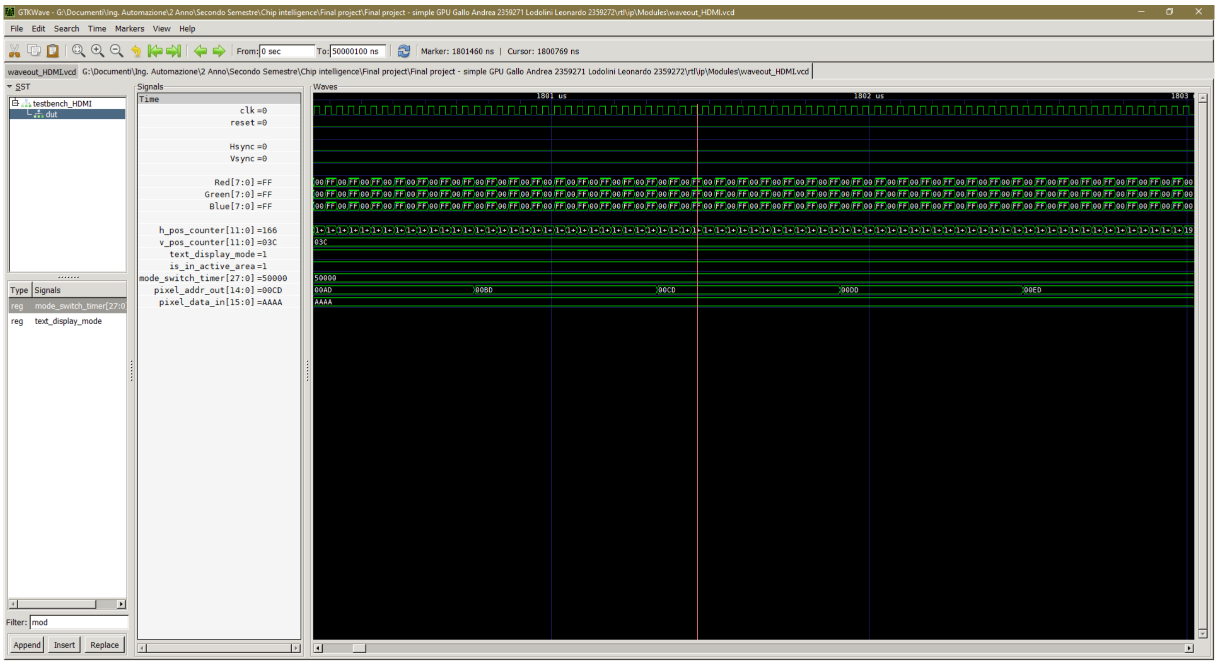

```
From: 0 sec   To: 50000100 ns   Marker: 1801460 ns  |  Cursor: 1800769 ns

Time
                     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
```

Figure 3.10: Detailed view of the text rendering mode. The `pixel_addr_out` is actively generated to read from the DPRAM. The RGB output reflects the black and white pattern derived from the static `pixel_data_in` (`16'hAAAA`), confirming the pixel serialization logic is working correctly.

The analysis of the waveforms demonstrates the following key points:

- **Overall Stability (Figure 3.7):** The macroscopic view confirms that the timing counters and synchronization signals are generated continuously and periodically, which is essential for a stable video output.
- **Color Bar Mode (Figure 3.8):** During the initial phase, the module correctly generates different RGB color values, cycling through the predefined palette to create the vertical test strips. This confirms the color bar generation logic is functional.
- **Mode Transition (Figure 3.9):** The waveform clearly shows the precise moment the `mode_switch_timer` expires. As soon as this happens, the `text_display_mode` flag is asserted, and the video output switches to rendering data based on the `pixel_data_in` input, as expected.
- **Text Rendering Logic (Figure 3.10):** In text mode, the module correctly generates the DPRAM read addresses (`pixel_addr_out`). The RGB output corresponds to the black-and-white pattern of the static input `16'hAAAA`. This verifies that the read-path logic, including the pipelined pixel selection and color mapping, is operating correctly.

The successful observation of all these phases confirms the module's readiness for integration.

Chapter 4

Integration and System-Level Verification

This chapter details the integration of the individual hardware modules into a complete GPU subsystem, encapsulated within the `GPU_top` module. It then describes the methodology used to verify this integrated system, including the design of a system-level testbench and the automation scripts developed to streamline the simulation process.

4.1 Top-Level IP: The `GPU_top` Module

The `GPU_top` module serves as the master container for the entire custom display peripheral. Its primary role is to instantiate and interconnect the core functional blocks: the `dpram_adapter`, the `Gowin_DPB` (Dual-Port RAM), and the `HDMI` signal generator. It exposes a simple interface to the rest of the system, taking the 32-bit CPU command as input and providing the final 24-bit RGB video signals and synchronization pulses as outputs.

4.1.1 Internal Structure and Dataflow

The internal architecture of `GPU_top` is defined by the connections between its three main components, facilitated by a set of internal wires. The dataflow follows two distinct, parallel paths: a write-path and a read-path, both centered around the DPRAM.

- **Write-Path:** The incoming 32-bit command, `i_cpu_cmd`, is fed directly to the `dpram_adapter` instance. This adapter decodes the command and generates the corresponding 15-bit write address (`a_write_address`) and 16-bit write data (`a_write_data`). These signals are then wired to the write port (Port A) of the `Gowin_DPB` instance, populating the video memory.

- **Read-Path:** Concurrently, the `HDMI` instance generates a 15-bit 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 DPRAM responds by placing the 16-bit data from that memory location onto the `b_read_data_out` wire, which is then fed back into the `HDMI` instance for processing and final rendering.

This clear separation of write and read paths, enabled by the dual-port nature of the RAM, is fundamental to the system's performance, as it allows the CPU to update the framebuffer without interfering with the continuous, real-time video signal generation.

4.1.2 Component Instantiation

The core of the GPU_top module consists of the instantiation and wiring of its submodules. The Verilog snippet below shows how these components are connected. A key feature implemented at this level is the anti-tearing logic: the write-enable signal from the dpram_adapter is gated with a flag from the HDMI module that indicates when the display is in a blanking interval. This ensures that memory writes only occur when the screen is not being actively drawn, preventing visual artifacts.

```verilog
// Internal Wires for Inter-Module Communication
wire [14:0] a_write_address;   // Address for writing into the DPRAM.
wire [15:0] a_write_data;      // Data to be written into the DPRAM.
wire        we_a;              // Raw write enable from the adapter.
wire [14:0] b_read_address;    // Address for reading from the DPRAM.
wire [15:0] b_read_data_out;   // Data read from the DPRAM.
wire        hdmi_is_displaying;// Flag is high during active display
    area.

// Component 1: HDMI Signal Generator
HDMI hdmi_instance (
    .clk(clk),
    .reset(reset),
    .pixel_data_in(b_read_data_out),
    .pixel_addr_out(b_read_address),
    .o_hsync(o_gpu_hsync),
    .o_vsync(o_gpu_vsync),
    .o_red(o_gpu_red),
    .o_green(o_gpu_green),
    .o_blue(o_gpu_blue),
    .is_in_active_area(hdmi_is_displaying)
);

// Component 2: DPRAM adapter
dpram_adapter adapter_instance (
    .clk(clk),
    .reset(reset),
    .i_cpu_cmd(i_cpu_cmd),
    .write_enable(we_a), // Output: Raw write enable.
    .o_dpram_addr(a_write_address),
    .o_dpram_wdata(a_write_data)
);

// Anti-Tearing Logic: Allow writes only during blanking intervals.
wire final_wrea = we_a && !hdmi_is_displaying;

// Component 3: Dual-Port Block RAM (Gowin Primitive)
Gowin_DPB dpram_instance (
    // Port A: Write Port (driven by dpram_adapter)
    .clka(clk),
    .reseta(reset),
    .cea(1'b1),
    .wrea(final_wrea), // Use the gated write enable signal.
    .ada(a_write_address),
    .dina(a_write_data),
    .douta(),          // Port A read data output is unused.
    .ocea(1'b1),       // Output Clock Enable for Port A is always on.

    // Port B: Read Port (driven by hdmi_instance)
    .clkb(clk),
    .resetb(reset),
    .ceb(1'b1),
    .wreb(1'b0),
    .adb(b_read_address),
    .doutb(b_read_data_out),
    .dinb(),            // Port B write data is unused.
    .oceb(1'b1)         // Output Clock Enable for Port B is always on.
);
```

Listing 4.1: Instantiation of core components and anti-tearing logic within `GPU_top`.

The top-level ports of the `GPU_top` module (`o_gpu_hsync`, `o_gpu_vsync`, etc.) are directly connected to the corresponding outputs of the `HDMI` instance, thus exposing the final video signals to the higher levels of the SoC design, ready to be routed to the FPGA's physical output pins.

4.2 Rapid Datapath Verification

The first step in verifying the integrated system is to ensure the integrity of its core datapath. A targeted test, using the `test_fast_datapath` testbench, was designed to quickly validate the write-and-read functionality of the `dpram_adapter` and the `Gowin_DPB` RAM working in tandem.

4.2.1 Testbench Design: `test_fast_datapath`

This testbench instantiates only the adapter and the DPRAM, deliberately excluding the much slower `HDMI` controller to allow for rapid simulation. The test sequence, shown in Listing 4.2, follows a clear write-then-read-and-verify pattern:

1. **Write Phase:** The testbench iterates through a predefined number of memory locations. For each location, it constructs a 32-bit command for the `dpram_adapter`, embedding the address value itself into the data field (e.g., writing data `0x0005` to address `0x0005`).

2. **Read and Verify Phase:** After the write phase is complete, the testbench directly controls the read port of the DPRAM. It reads back the data from each location and asserts an error using `$error` if the data read does not match the data that was written.

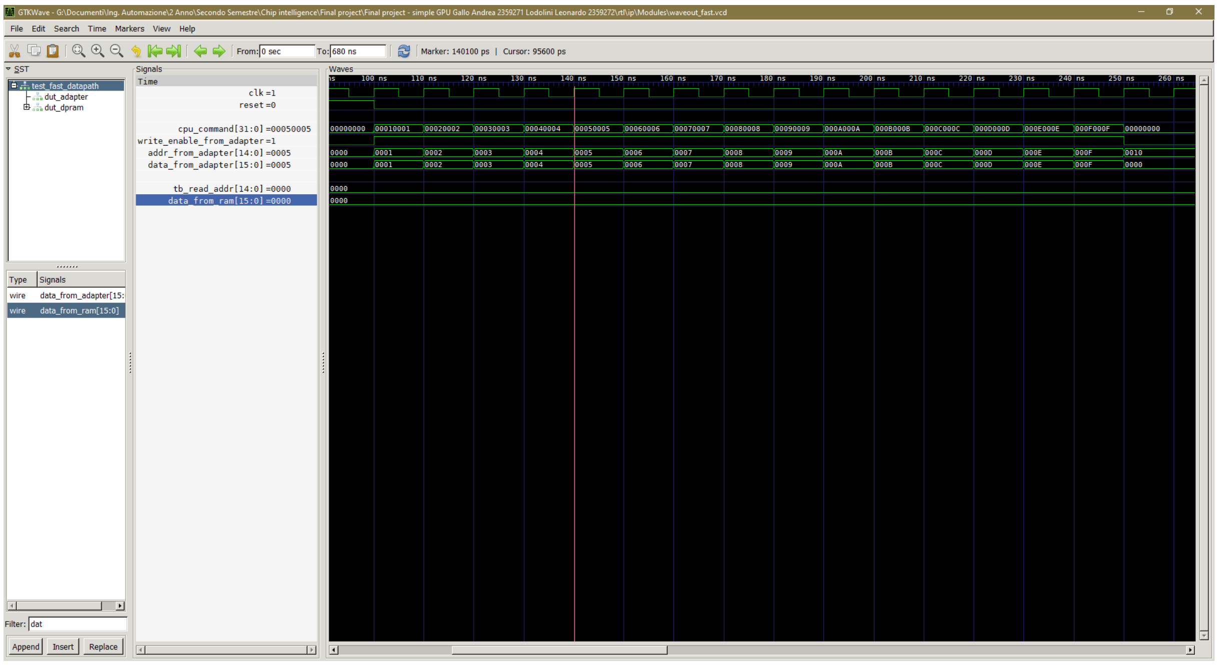
```
// --- PHASE 1: WRITE a predictable pattern ---
for (integer i = 1; i <= NUM_WRITES; i = i + 1) begin
    // Format the command to write the address value into the data field
  .
    cpu_command = {16'b0, i[3:0], i[15:0]};
    @(posedge clk);
end

// ... wait for stability ...

// --- PHASE 2: READ and VERIFY ---
for (integer i = 1; i <= NUM_WRITES; i = i + 1) begin
    // Apply the address to the read port of the DPRAM
```

```verilog
    tb_read_addr = i;
    @(posedge clk); // Cycle 1: Address is registered
    @(posedge clk); // Cycle 2: Data appears on output

    // Verify that the data read matches the data written
    if (data_from_ram === i) begin
        $display("  OK: Addr[0x%h] -> Read: 0x%h.", i, data_from_ram);
    end else begin
        $error("  FAIL: Addr[0x%h] -> Read: 0x%h.", i, data_from_ram);
    end
end
```

Listing 4.2: Main test sequence from `test_fast_datapath.v`.

4.2.2 Simulation Results

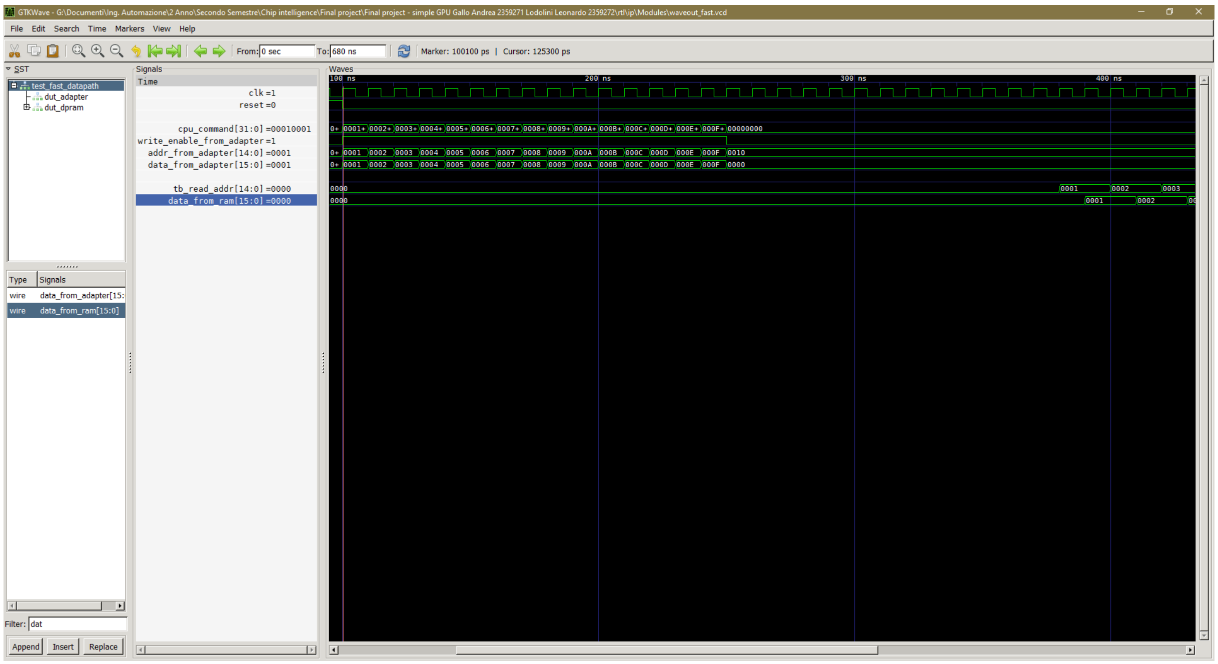
The successful execution of this test is confirmed by both waveform analysis and the terminal log.


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: 140100 ps | Cursor: 95600 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
Waves
100 ns 110 ns 120 ns 130 ns 140 ns 150 ns 160 ns 170 ns 180 ns 190 ns 200 ns 210 ns 220 ns 230 ns 240 ns 250 ns 260 ns

```
Time
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
```

```
cpu_command:       00000000 00010001 00020002 00030003 00040004 00050005 00060006 00070007 00080008 00090009 000A000A 000B000B 000C000C 000D000D 000E000E 000F000F 00000000
addr_from_adapter: 0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F 0010
data_from_adapter: 0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F 0000
tb_read_addr:      0000
data_from_ram:     0000
```

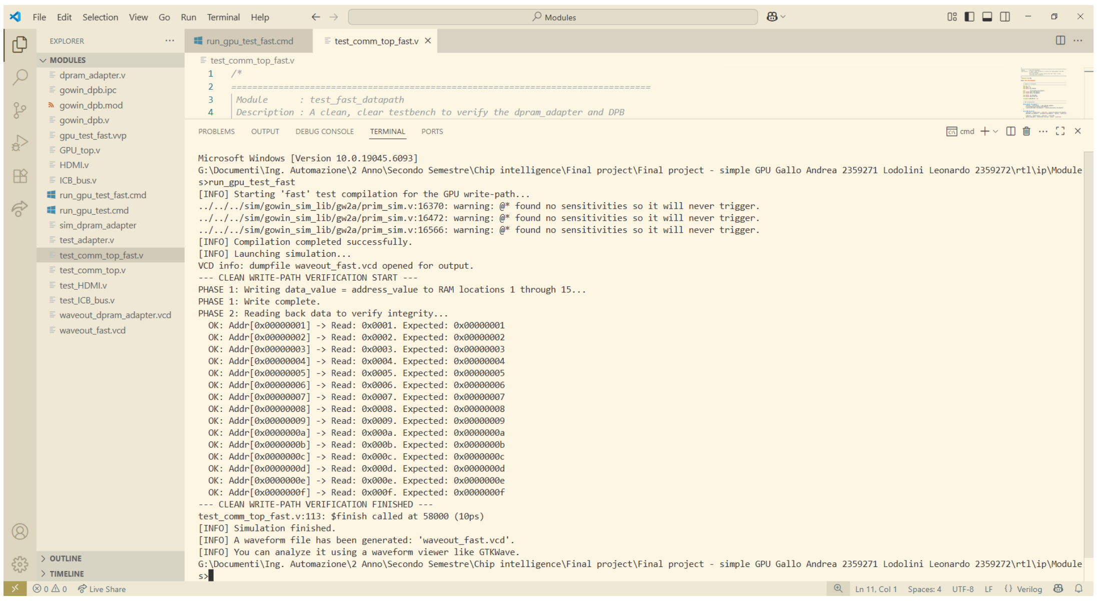
Figure 4.1: Waveform showing the write phase of the `test_fast_datapath` simulation. The `cpu_command` is driven with a new value at each clock cycle, and the `dpram_adapter` correctly generates the corresponding write address and data.



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
Marker: 100100 ps
Cursor: 125300 ps

Figure 4.2: Waveform showing the read and verify phase. The testbench drives the `tb_read_addr` port of the DPRAM. After a one-cycle read latency, the correct data appears on the `data_from_ram` output, matching the address that was read.

The waveforms in Figure 4.1 and 4.2 clearly demonstrate that the write commands are correctly translated by the adapter and that the subsequent read operations retrieve the exact data that was stored. Furthermore, the terminal log in Figure 4.3 provides an explicit confirmation that all data integrity checks passed, proving the datapath is functionally correct.



```
/*
===============================================================================
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 4.3: Terminal output from the `test_fast_datapath` simulation, confirming that all read-back data matched the expected values and the test passed.

4.3 End-to-End Local System Verification

While the datapath verification confirms the integrity of the core write/read mechanism, a full local system test is necessary to validate the interaction of all components, including the `HDMI` controller's timing generation and dual-mode operation. This was achieved using the `test_full_system` testbench.

4.3.1 Testbench Design: `test_full_system`

This testbench instantiates the complete `GPU_top` module. Its primary purpose is to simulate a realistic operational sequence and allow for visual inspection of the final video output signals. The test flow is as follows:

1. **Wait for Mode Transition:** The testbench first waits for a fixed duration (500,000 ns), allowing the DUT's internal timer to expire and switch from the initial color bar mode to text rendering mode.

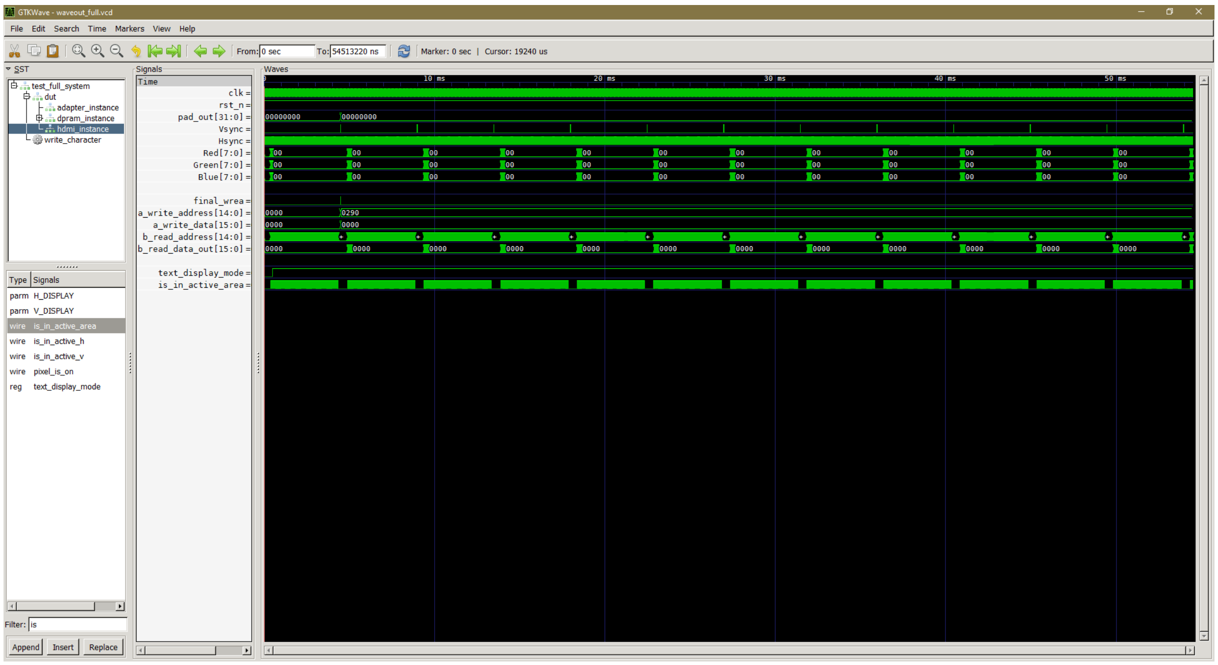
2. **Synchronize with VBI:** To ensure safe writing to the framebuffer and validate the anti-tearing logic, the testbench waits for the next vertical blanking interval (VBI) by synchronizing with the `Vsync` signal.

3. **Write Character Data:** Once in the VBI, the testbench sends a burst of commands to write a short message ("HI!") to the framebuffer.

4. **Observe Output:** The simulation is then left to run for several frame durations, allowing the rendered text to be observed on the RGB output signals in the waveform.

4.3.2 Simulation Results

The waveform analysis confirms that the entire GPU subsystem operates correctly as a cohesive unit. The key operational phases are illustrated in Figures 4.4 through 4.7.



From: 0 sec To: 54513220 ns Marker: 0 sec | Cursor: 19240 us

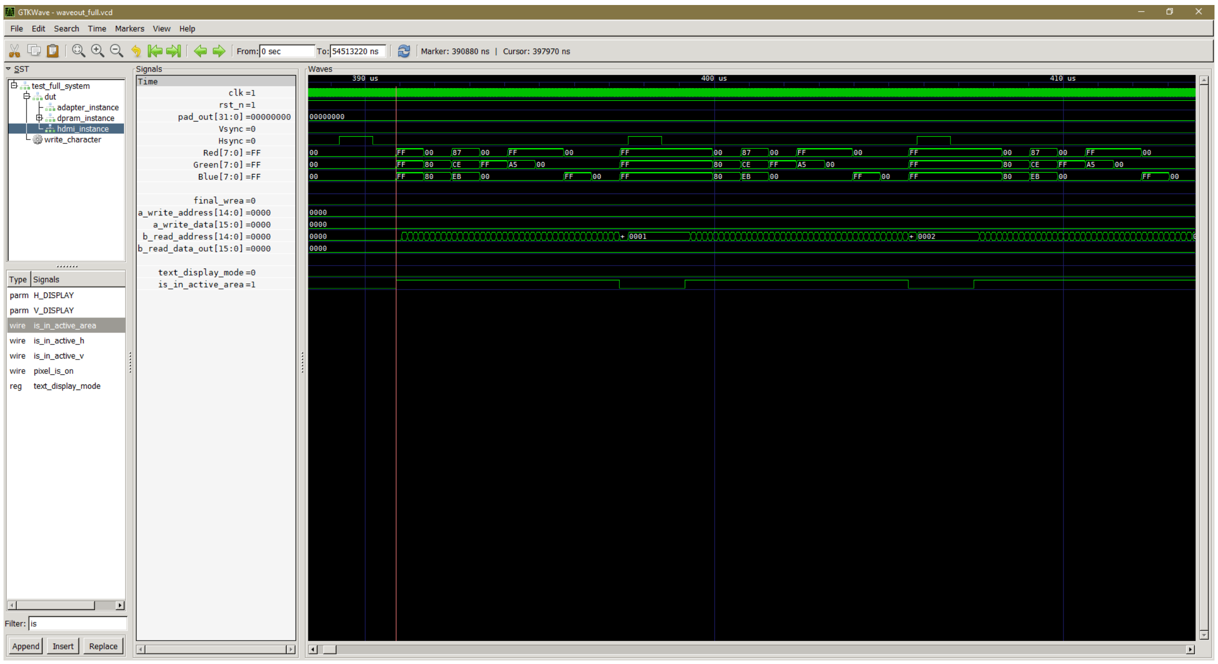
```
test_full_system
  dut
    adapter_instance
    dpram_instance
    hdmi_instance
  write_character
```

Filter: is

Type	Signals
parm	H_DISPLAY
parm	V_DISPLAY
wire	is_in_active_area
wire	is_in_active_h
wire	is_in_active_v
wire	pixel_is_on
reg	text_display_mode

```
clk =
rst_n =
pad_out[31:0] =
Vsync =
Hsync =
Red[7:0] =
Green[7:0] =
Blue[7:0] =

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

text_display_mode =
is_in_active_area =
```

Figure 4.4: A wide view of the `test_full_system` simulation. 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.



From: 0 sec To: 54513220 ns Marker: 390880 ns | Cursor: 397970 ns

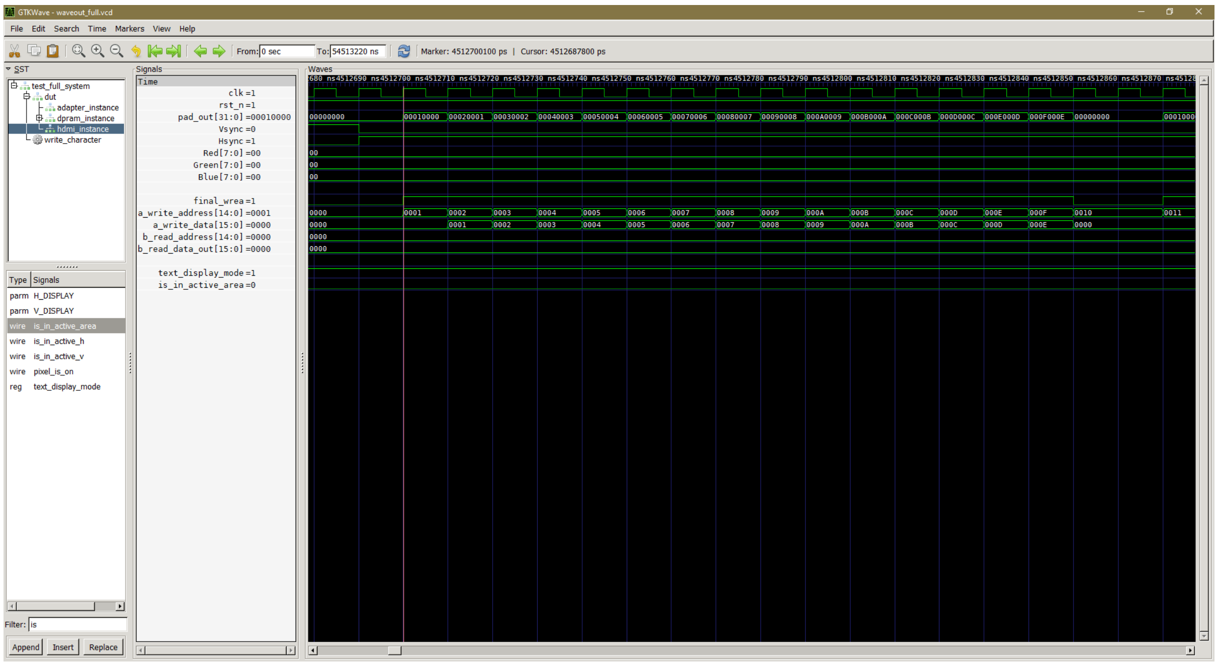
```
test_full_system
  dut
    adapter_instance
    dpram_instance
    hdmi_instance
  write_character
```

Filter: is

Type	Signals
parm	H_DISPLAY
parm	V_DISPLAY
wire	is_in_active_area
wire	is_in_active_h
wire	is_in_active_v
wire	pixel_is_on
reg	text_display_mode

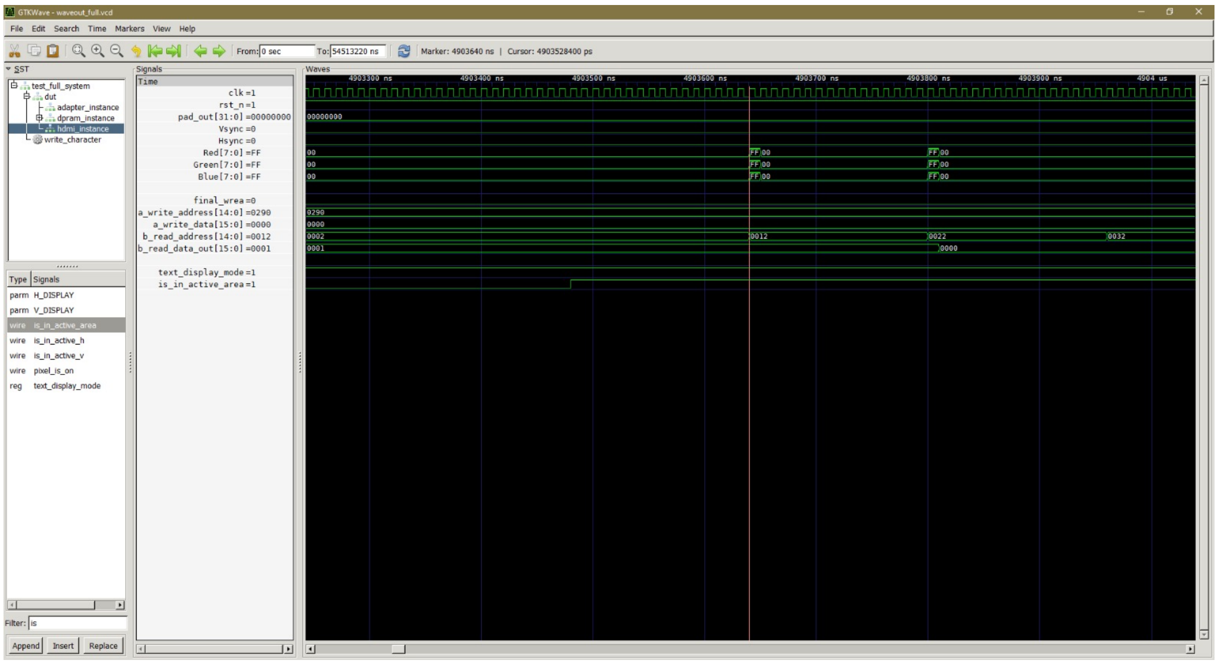
```
clk =1
rst_n =1
pad_out[31:0] =00000000
Vsync =0
Hsync =0
Red[7:0] =FF
Green[7:0] =FF
Blue[7:0] =FF

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 4.5: Detailed view of the initial color bar mode. The RGB output signals change as the horizontal position counter advances, generating the distinct vertical color strips. The `text_display_mode` flag is low during this phase.


GTKWave - waveout_full.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 54513220 ns
Marker: 4512700100 ps | Cursor: 4512687800 ps
SST
Signals
Waves
Filter: is
Append
Insert
Replace

```
test_full_system
  dut
    adapter_instance
    dpram_instance
    hdmi_instance
  write_character

Type  Signals
parm  H_DISPLAY
parm  V_DISPLAY
wire  is_in_active_area
wire  is_in_active_h
wire  is_in_active_v
wire  pixel_is_on
reg   text_display_mode

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 4.6: The character write phase, synchronized with the VBI. The testbench drives the `pad_out` bus to write the characters "H" and "I", followed by an "Enter" command. This occurs while the `HDMI` module is in a non-displaying state, validating the anti-tearing mechanism.


GTKWave - waveout_full.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 54513220 ns
Marker: 4903640 ns | Cursor: 4903528400 ps
SST
Signals
Waves
Filter: is
Append
Insert
Replace

```
test_full_system
  dut
    adapter_instance
    dpram_instance
    hdmi_instance
  write_character

Type  Signals
parm  H_DISPLAY
parm  V_DISPLAY
wire  is_in_active_area
wire  is_in_active_h
wire  is_in_active_v
wire  pixel_is_on
reg   text_display_mode

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] =0290
   a_write_data[15:0] =0000
 b_read_address[14:0] =0012
b_read_data_out[15:0] =0001

     text_display_mode =1
     is_in_active_area =1
```

Figure 4.7: The video rendering phase in text mode. This view captures the moment the HDMI raster scan reaches a memory location recently updated by the testbench. The `HDMI` module's read address (`b_read_address`) retrieves the character data (`0xFFC0` for a row of 'H'), which is then correctly rendered on the RGB output.

The key observations from the waveforms are:

- **Overall Stability (Figure 4.4):** The macroscopic view confirms that the system is stable over multiple frame generation cycles and correctly transitions from an initial colored pattern to the final text-based display.

- **Color Bar Mode Validation (Figure 4.5):** The detailed view of the startup phase clearly shows the module generating different RGB values to create the vertical test strips. This confirms the color bar generation logic and the initial state of the HDMI controller are functional.

- **Safe Framebuffer Writing (Figure 4.6):** The testbench successfully writes character data during the vertical blanking interval. The waveform shows the write commands being sent to the `dpram_adapter` while the video output is blank, confirming that the anti-tearing logic is effective.

- **Correct End-to-End Rendering (Figure 4.7):** This waveform demonstrates the successful completion of the end-to-end 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 row of the character 'H' (`0xFFC0`) 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.

These results validate that the `GPU_top` module functions as a complete and correct video display subsystem.

4.4 Simulation Automation with Scripts

To streamline the verification process and ensure repeatability, the simulation workflow was automated using batch scripts. This approach allows for the execution of different test suites—from rapid unit tests to comprehensive system-level simulations—with a single command. Two primary scripts were created to manage this process.

4.4.1 Fast Simulation for Datapath and Unit Testing

For rapid regression testing, a "fast" simulation script, `run_gpu_test_fast.cmd`, was developed. Its purpose is to compile and run only the necessary files for a targeted test, such as the datapath verification described in Section 4.2. By excluding the full `GPU_top` and `HDMI` modules, the simulation completes in a fraction of the time, providing immediate feedback during development. The core commands of this script are shown in Listing 4.3.

```
REM Compile only the modules required for the fast datapath test.
iverilog -o gpu_test_fast.vvp ^
    dpram_adapter.v ^
    gowin_dpb.v ^
    test_comm_top_fast.v ^
    ../../../sim/gowin_sim_lib/gw2a/prim_sim.v

REM Check for compilation errors.
IF %ERRORLEVEL% NEQ 0 (
    ECHO [ERROR] Compilation failed!
    GOTO :EOF
)

ECHO [INFO] Launching simulation...

REM Execute the compiled simulation.
vvp gpu_test_fast.vvp
```

Listing 4.3: Core commands from the `run_gpu_test_fast.cmd` script.

This script is essential for maintaining an efficient development cycle, allowing developers to quickly check the impact of their changes on core components without waiting for a full system simulation.

4.4.2 Complete Simulation for Local System Verification

For a full, end-to-end validation of the integrated `GPU_top` subsystem, the `run_gpu_test.cmd` script is used. This script compiles all the Verilog modules that constitute the GPU, including the `HDMI` controller and the comprehensive `test_full_system` testbench. The commands for this script are shown in Listing 4.4.

```
REM Compile ALL Verilog source files for the complete GPU subsystem.
iverilog -o gpu_test.vvp ^
    GPU_top.v ^
    dpram_adapter.v ^
    HDMI.v ^
    gowin_dpb.v ^
    test_comm_top.v ^
    ../../../sim/gowin_sim_lib/gw2a/prim_sim.v

REM Check for compilation errors.
IF %ERRORLEVEL% NEQ 0 (
    ECHO [ERROR] Compilation failed!
    GOTO :EOF
)

ECHO [INFO] Launching full system simulation...

REM Execute the compiled simulation.
vvp gpu_test.vvp
```

Listing 4.4: Core commands from the `run_gpu_test.cmd` script for full system simulation.

While this simulation is significantly slower due to the complexity of generating full video frames, it is indispensable for final validation, ensuring that all components interact correctly and that the system meets its overall functional requirements.

Chapter 5

Firmware Design and Operation

The firmware, written in C and compiled for the Hummingbird E203 RISC-V core, serves as the brain of the display system. It is responsible for initializing the hardware, handling user input from the UART, processing characters into a displayable format, and communicating with the custom HDMI peripheral via the Inter-Chip Bus (ICB). The entire logic is contained within a single `main.c` file, which includes the character font data, helper functions, and the main application logic structured as a Finite State Machine (FSM).

5.1 CPU-to-Hardware Communication Protocol

A key aspect of the firmware design is the communication protocol established between the CPU and the custom HDMI hardware. All interactions are managed through a single, 32-bit memory-mapped command register within the peripheral, accessible at the offset defined by `HDMI_COMMAND_OFFSET`. The firmware constructs a 32-bit command word for each transaction, with specific bits allocated for control flags and data. The definitions for these flags are centralized for clarity and maintainability.

```c
// --- CPU-to-Hardware Protocol Flags ---
// These flags are embedded in the 32-bit command sent to the HDMI
    peripheral.
#define FLAG_RESET_ADDR  (1U << 31) // Instructs the hardware to reset
    its character position counter.
#define FLAG_DELETE      (1U << 30) // Instructs the hardware to perform
     a backspace operation.
#define FLAG_ENTER       (1U << 29) // Instructs the hardware to move
    the cursor to the next line.
```

Listing 5.1: Definition of Hardware Command Flags in `main.c`.

The 32-bit command word is structured as follows:

- **Bit 31 (`FLAG_RESET_ADDR`):** When set, this flag instructs the hardware's `dpram_adapter` to reset its internal character position counter to zero. This is used to move the cursor to the top-left corner of the screen.
- **Bit 30 (`FLAG_DELETE`):** When set, this flag activates the hardware's delete-mode logic, which is a key part of the backspace operation.
- **Bit 29 (`FLAG_ENTER`):** When set, this flag instructs the hardware to move the character position to the beginning of the next line.

- **Bits 28-20 (Unused):** These bits are currently unused and reserved for future expansion. They are always set to zero by the firmware for data commands.
- **Bits 19-16 (Row Offset):** These 4 bits encode the vertical row index (from 0 to 15) within a 16x16 character cell. This tells the hardware which of the 16 bitmap rows is being sent.
- **Bits 15-0 (Pixel Data):** These bits contain the 16-bit monochrome pixel data for the specified character row.

This protocol allows the firmware to send both pixel data and high-level commands through a single, unified interface, enabling efficient control over the display hardware.

5.2 Application Flow: The Main FSM

The program's logic is structured around a simple but effective Finite State Machine (FSM) with four states, ensuring an orderly sequence of operations from startup to interactive use. The core of the application resides in the `main` function's `while` loop, which implements the FSM using a `switch` statement.

```c
// Snippet from the main function
while (1) {
    // The switch statement implements the Finite State Machine.
    switch (current_state) {

        case STATE_STARTUP_TEXT:
            // ... logic to display startup message ...
            current_state = STATE_WAIT_FOR_INPUT;
            break;

        case STATE_WAIT_FOR_INPUT:
            // ... logic to poll UART ...
            break;

        case STATE_CLEAR_STARTUP:
            // ... logic to clear startup message ...
            current_state = STATE_TERMINAL_MODE;
            break;

        case STATE_TERMINAL_MODE:
            // ... logic to handle terminal input ...
            break;
    }
}
```

Listing 5.2: The Main FSM structure within the `while(1)` loop.

The state transitions are managed as follows:

1. `STATE_STARTUP_TEXT`: Upon initialization, the FSM enters this state. The firmware iterates through a predefined startup message and calls the `hdmi_display_char` function for each character to render it on the screen. It then transitions to the next state.
2. `STATE_WAIT_FOR_INPUT`: The system waits passively in this state, continuously polling the UART. As soon as any character is received, it signifies user interaction, and the FSM transitions to the clearing state.

3. STATE_CLEAR_STARTUP: To prepare the screen for terminal mode, this state overwrites the startup message with space characters. It cleverly uses the FLAG_RESET_ADDR on the very first space character to command the hardware to move its cursor back to the start of the message. After clearing the message, it sends one final reset command to move the cursor to the top-left of the screen and then transitions to the main operational mode.

4. STATE_TERMINAL_MODE: This is the final, persistent state where the system functions as an interactive terminal. It polls for UART input and processes each received character. Special keys are handled with specific protocols: the Enter key sends the FLAG_ENTER command, while the Backspace key initiates a three-step sequence: it first sends a FLAG_DELETE command, then writes a space character to visually clear the previous position, and finally sends another FLAG_DELETE command to correctly reposition the cursor.

This FSM-based approach provides a robust and clear structure for managing the application's different operational phases.

5.3 Core Functions and Data Structures

5.3.1 Bitmap Font Data

The firmware includes a large, constant 2D array, `const uint16_t bitmap[96][16]`, which stores the graphical representation for all 96 printable ASCII characters. Each character is represented by a 16x16 pixel matrix, where each of the 16 rows is stored as a `uint16_t` value. This embedded font table is the source for all character rendering.

5.3.2 Helper Functions

To keep the main loop clean and modular, several helper functions were implemented. The most important of these is `hdmi_display_char`, which encapsulates the entire process of sending a character's bitmap data to the hardware.

```c
// Renders a single character on the screen.
// It translates an ASCII code into bitmap data and sends it to the GPU
// row by row.
void hdmi_display_char(uint8_t ascii_code) {
    // If the character is not printable, display a question mark instead.
    if (ascii_code < 32 || ascii_code > 127) {
        ascii_code = ASCII_QUESTION_MARK;
    }
    // The bitmap array starts at ASCII 32, so we adjust the index.
    uint16_t bitmap_index = ascii_code - 32;

    // Send the 16 rows of the character's bitmap to the GPU.
    for (uint16_t i = 0; i < 16; i++) {
        // The command is a 32-bit word:
        // - The upper bits [19:16] contain the row index (0-15).
        // - The lower 16 bits contain the bitmap data for that row.
        uint32_t command = ((uint32_t)i << 16) | (uint32_t)bitmap[bitmap_index][i];
        hdmi_write_command(command);
    }
    // A zero command signifies the end of character data, instructing
   the
    // hardware to advance its internal cursor.
    hdmi_write_command(0);
}
```

Listing 5.3: The `hdmi_display_char` helper function.

This function is central to the firmware's operation. It constructs the 32-bit command for each of the 16 rows of a given character and, crucially, sends a final zero-value command to signal the end of the character transmission. Other helper functions, such as `is_uart_data_available()` and `read_uart_char()`, abstract the low-level register access to the UART, simplifying the main application logic.

Chapter 6

Full SoC Implementation and Final System Validation

This final chapter presents the integration of the custom GPU subsystem into a complete System-on-Chip (SoC) based on the Hummingbird E203 RISC-V core. The objective of this ultimate validation step is to demonstrate the end-to-end functionality of the entire project. The test methodology involves executing the project's C firmware on the RISC-V core to process character data received via a UART interface and drive the GPU. The successful rendering of characters, verified through simulation, provides the definitive confirmation that all hardware and software components operate together as intended.

6.1 System-on-Chip Architecture Overview

The test environment for the final validation is a comprehensive SoC, encapsulated within the `e203_soc_demo` module. This system brings together the RISC-V processor, memory, standard peripherals, and our custom GPU, all interconnected via the Inter-Chip Bus (ICB). The key components and their roles in this architecture are described below.

6.1.1 Core Components of the SoC

- **`e203_subsys_main`:** This is the heart of the SoC. It instantiates the Hummingbird E203 processor core, a Boot ROM for firmware storage, and the controllers for standard peripherals. Crucially, it also contains the master interface to the ICB bus, through which the CPU communicates with all other modules. Our custom GPU is integrated within this subsystem to gain access to the ICB bus.

- **Hummingbird E203 Core:** A 32-bit RISC-V processor (RV32IMAC) responsible for executing the C firmware. It fetches instructions from the Boot ROM and performs memory-mapped I/O operations to control the peripherals.

- **UART Peripheral:** A standard Universal Asynchronous Receiver-Transmitter module is included in the SoC. It is mapped to a specific address on the ICB bus and provides the communication link to the outside world, simulated by the testbench.

- **Custom GPU Subsystem:** The `GPU_top` module, as verified in the previous chapter, is instantiated within the SoC and connected as a slave peripheral on the ICB bus. It is assigned a unique base address in the SoC's memory map.

6.1.2 End-to-End Data Flow

The final validation test follows a precise data flow that exercises every critical component of the integrated system:

1. **External Input:** The simulation begins with the testbench, acting as a host computer, sending an ASCII character byte serially to the SoC's designated UART receive pin.
2. **UART Reception:** The UART peripheral inside the SoC receives the serial data, converts it into a parallel byte, and stores it in its receive buffer.
3. **Firmware Execution:** The E203 core, continuously polling the UART's status register via the ICB bus, detects that a new character is available. It performs a memory-mapped read to retrieve the character from the UART's data register.
4. **GPU Command Generation:** The C firmware processes the character. It looks up the corresponding 16x16 bitmap font data and constructs a sequence of 17 commands (16 for the pixel rows and one end-of-character signal).
5. **ICB Transaction:** For each command, the E203 core executes a 32-bit 'store word' instruction to the memory-mapped address of the GPU peripheral. This instruction is translated by the core's bus interface unit into a write transaction on the ICB bus.
6. **GPU Processing and Video Output:** The GPU's ICB interface ('ICB_bus') captures the command from the bus and forwards it to the internal logic. The `dpram_adapter` writes the pixel data to the framebuffer, and the `HDMI` controller reads this data to generate the final RGB video signals.

This complete cycle, from a UART pin to the RGB output pins, represents the ultimate test of the project's functionality.

6.2 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.

6.2.1 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 6.1, 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.

```verilog
// 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 6.1: 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.

6.2.2 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 6.2, 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.

```verilog
// 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 6.2: 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.

6.2.3 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 6.3, 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.

```verilog
// 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 6.3: 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.

6.3 Final Validation Testbench

The final validation of the project was conducted using a comprehensive testbench, `sys_tb_top.sv`, designed to emulate the complete operational environment of the SoC.

This testbench is fundamentally different from the unit and subsystem-level tests described previously. Its primary role is not to drive internal signals, but to simulate the real-world interfaces with which the SoC would interact, thereby providing a true black-box validation of the entire system's functionality.

6.3.1 Testbench Architecture and Key Functions

The `sys_tb_top` testbench instantiates the entire `e203_soc_demo` module as its Device Under Test (DUT). It is architected to perform three critical functions that drive and monitor the SoC's behavior.

1. **System Initialization and Firmware Loading:** The testbench is responsible for generating the system clocks (a high-frequency clock for the core and a low-frequency clock for auxiliary functions) and managing the system's active-low reset signal. *A crucial part of the initialization*, though not directly visible in the testbench code, is the *pre-loading of the compiled C firmware*. The simulation environment is configured to use the Verilog system task `$readmemh` to load the hexadecimal machine code produced by the RISC-V compiler into the Boot ROM memory model within the SoC. This ensures that when the reset is released, the E203 core begins executing our project's specific firmware.

2. **Host Terminal Emulation:** A key function of the testbench is to act as a host computer sending data to the SoC. This is achieved by manipulating the GPIO pin designated as the UART's receive line (`gpio_in[16]`). As shown in Listing 6.4, the main stimulus block first reads character data from an external text file, `input.txt`, into a Verilog array. It then iterates through this array, calling a dedicated task, `uart_tx_data`, to transmit each character serially to the DUT. This process directly tests the UART reception and the firmware's ability to handle incoming data.

3. **Output Observation:** The testbench connects wires to the SoC's primary video output pins: `H_sync`, `V_sync`, `Red`, `Green`, and `Blue`. It does not perform any automatic checks on these signals. Instead, its purpose is to enable their recording into a Value Change Dump (`.vcd`) file, allowing for detailed offline analysis in a waveform viewer (GTKWave). This observational approach is standard for verifying complex, data-intensive outputs like video streams.

```
initial begin
    // This array will hold the test characters read from the input file
    .
    reg [7:0] sim_data[4:0];

    // Initialize the testbench's UART transmit line to the IDLE state (
    high).
    gpio_in[16] = 1'b1;

    // Load test data from an external file into the simulation.
    $readmemh("./input.txt", sim_data);

    // Wait for the DUT to initialize completely after reset.
    #7ms;

    // --- Data Transmission Loop ---
    // This loop reads each character from the loaded data and sends it.
    foreach(sim_data[x]) begin
      $display("TB INFO: Sending tx_data[%x] = %x", x, sim_data[x]);

      // Call the UART transmit task to send the current byte.
      uart_tx_data(sim_data[x]);

      // A small delay between bytes makes the simulation more realistic
   .
      #10_000; // Wait 10 us before sending the next byte.
    end
end
```

Listing 6.4: Main stimulus block from `sys_tb_top.v`, showing file I/O and UART transmission loop.

By orchestrating this realistic sequence —loading firmware, sending serial data, and observing video output— the testbench provides the highest level of confidence in the project's success, confirming that all hardware and software components work together seamlessly.

6.4 End-to-End System Validation: Waveform Analysis

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 running on the RISC-V core, 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.

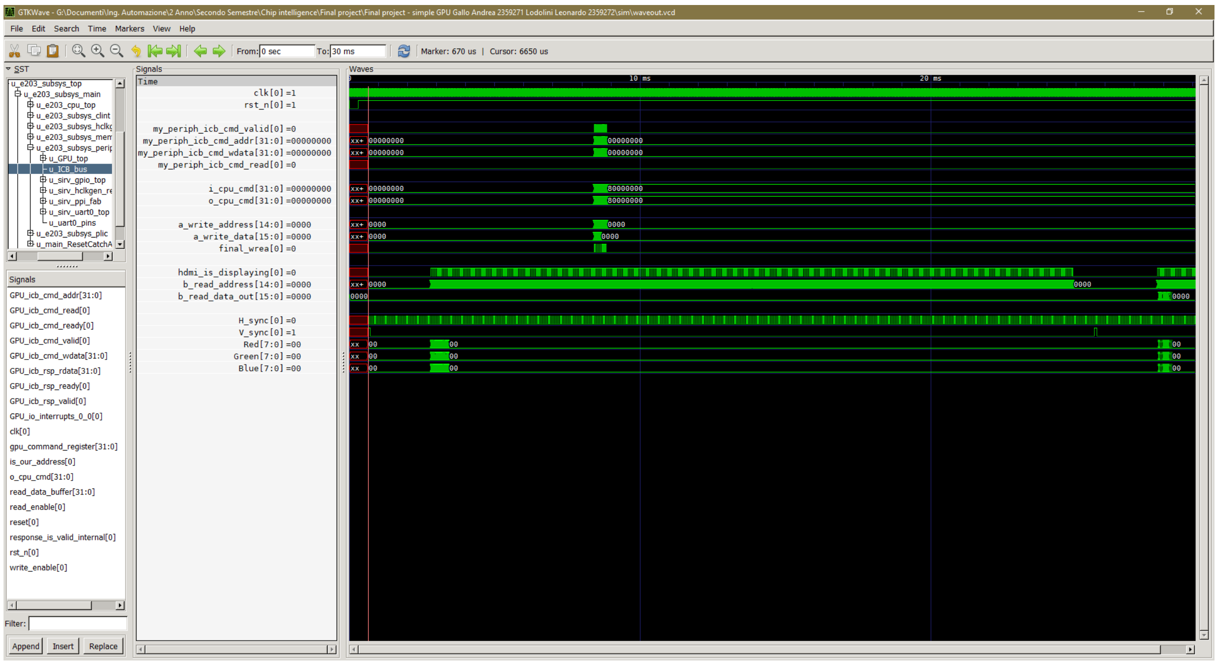
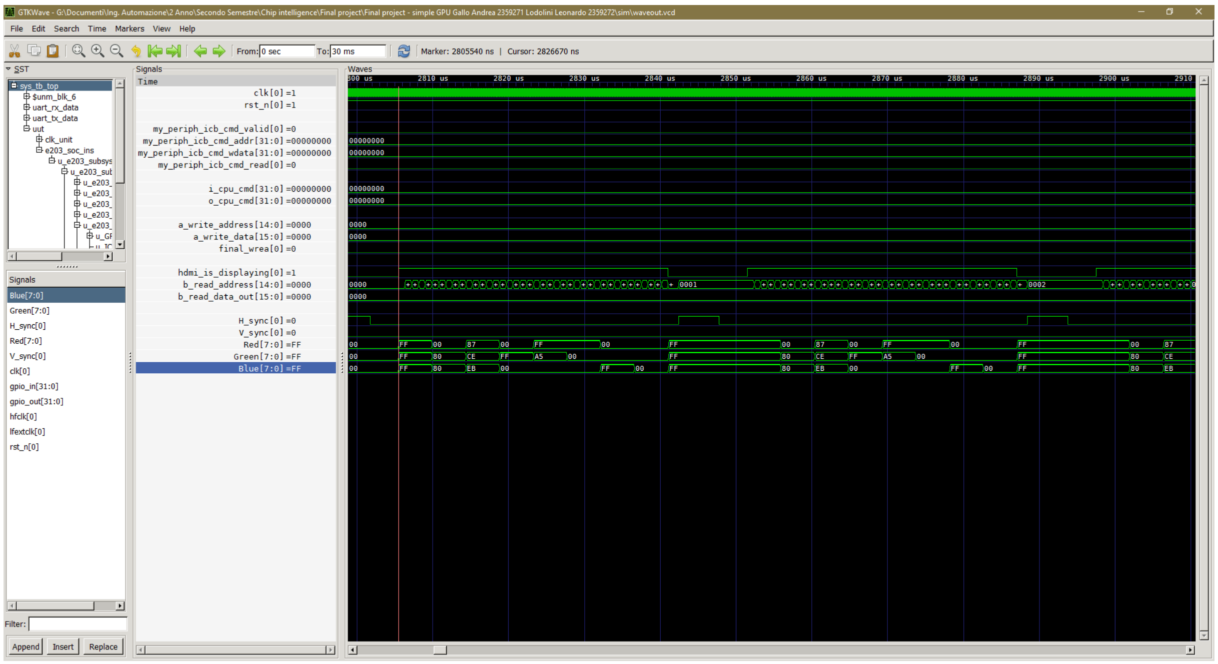


Figure 6.1: A wide-view of the full SoC simulation. The periodic bursts of activity on the ICB bus (center) correspond to the firmware executing its main loop. The stability of the video synchronization signals (`H_sync`, `V_sync`) is also visible.

6.4.1 GPU Initialization and Firmware Interaction

The simulation begins with the GPU in its autonomous startup state. As shown in Figure 6.2, immediately after reset, and before the firmware begins active control, 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 initialization of the video clocking and timing generation logic.



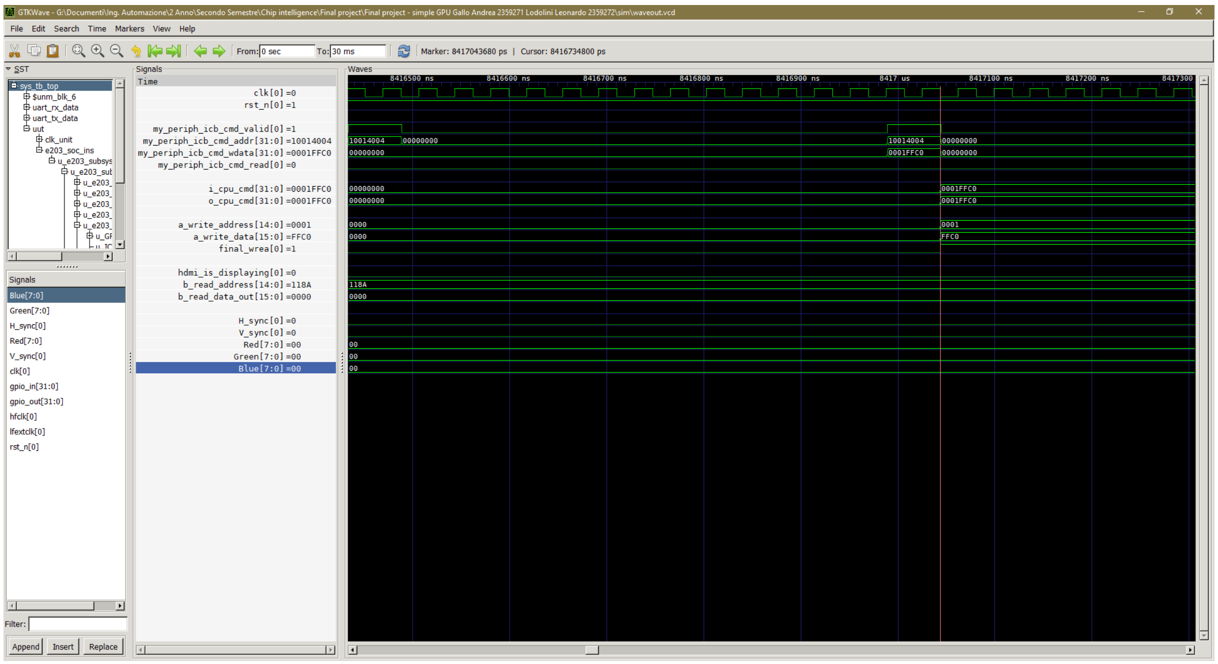
```
Time
                      clk[0] =1
                    rst_n[0] =1

    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] =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 6.2: 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, executing on the E203 core, has processed an incoming character from the UART, it takes control of the GPU. This is achieved by initiating a series of write transactions on the ICB bus. Figure 6.3 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 Bus Interface Unit (BIU) asserts `my_periph_icb_cmd_valid`, signaling the start of a transaction.
- **Address Decoding:** The address bus, `my_periph_icb_cmd_addr`, is driven with the value `0x10014004`. This is the precise memory-mapped address assigned to our GPU peripheral, confirming that the ICB bus fabric correctly decodes and routes the request.
- **Data Payload:** The write data bus, `my_periph_icb_cmd_wdata`, carries the 32-bit payload: `0x0001FFC0`. This value is intelligently constructed by the firmware: the upper bits (`0x0001`) represent the row offset (the first row of the character), and the lower bits (`0xFFC0`) contain the 16-bit monochrome pixel data for that specific row of the character 'Z'.

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

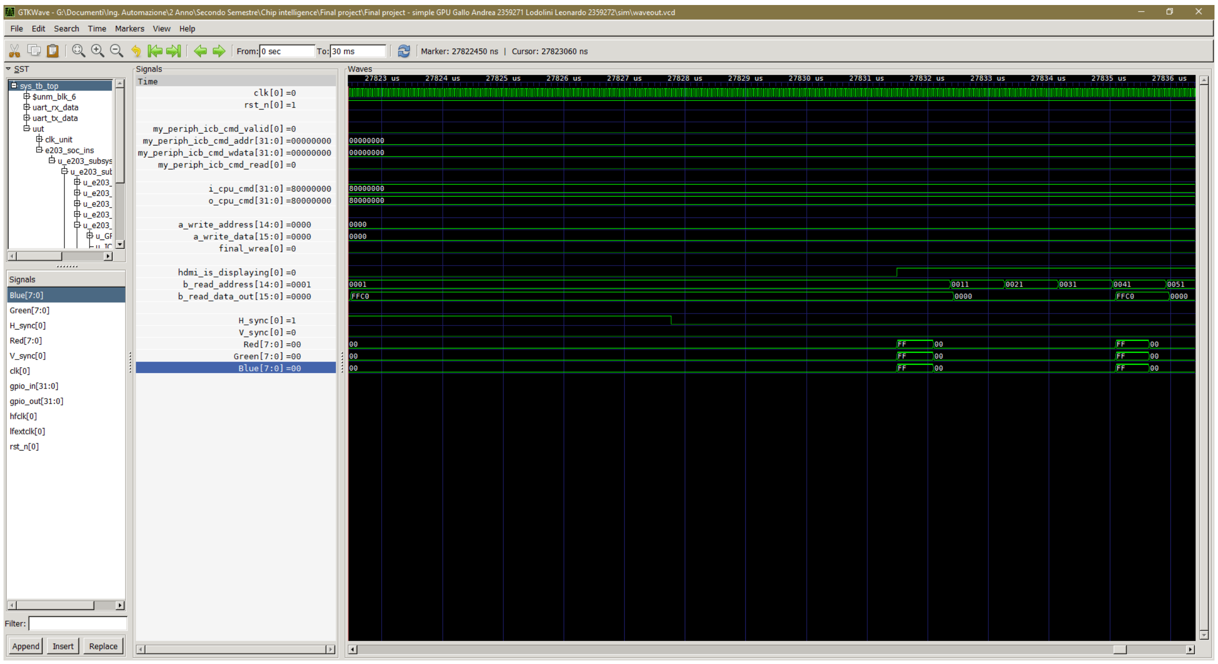


```
Time
                          clk[0] =0
                        rst_n[0] =1

       my_periph_icb_cmd_valid[0] =1
    my_periph_icb_cmd_addr[31:0] =10014004
   my_periph_icb_cmd_wdata[31:0] =0001FFC0
        my_periph_icb_cmd_read[0] =0

               i_cpu_cmd[31:0] =0001FFC0
               o_cpu_cmd[31:0] =0001FFC0

         a_write_address[14:0] =0001
            a_write_data[15:0] =FFC0
                   final_wrea[0] =1

         hdmi_is_displaying[0] =0
          b_read_address[14:0] =118A
         b_read_data_out[15:0] =0000

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

Figure 6.3: 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.

6.4.2 Framebuffer Update and Final Video Rendering

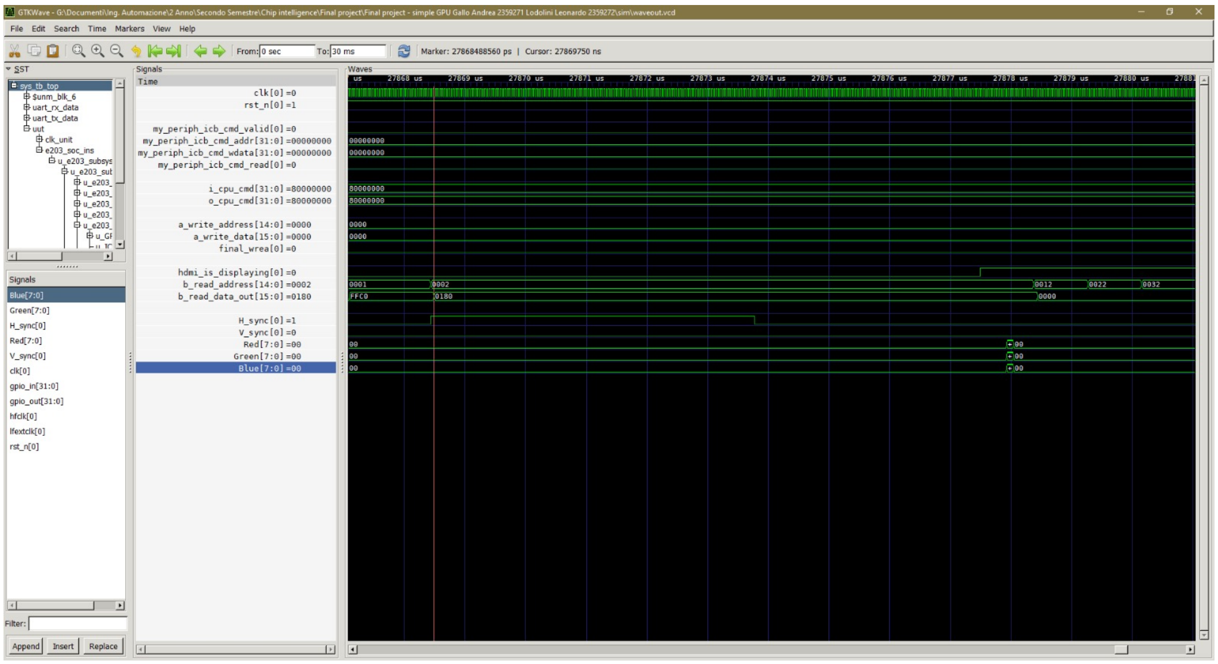
The command sent by the CPU is captured by the GPU's ICB interface and processed internally, resulting in an update to the video framebuffer. The final step is to verify that this new data is correctly rendered on the video output. The tangible result of this process is shown in Figure 6.4. The moment the 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, and the RGB outputs immediately change from black to white, beginning the process of drawing the character on the screen.



```
GTKWave - G:\Document\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 6.4: The effect of the new data on the final video output. As the data for the character 'Z' (value `0xFFC0`) is read from the framebuffer, the RGB signals change from black (`0x00`) to white (`0xFF`).

Finally, Figure 6.5 provides the conclusive, micro-level evidence that closes the validation loop. It confirms the direct, clock-cycle-accurate correlation between the data in memory (`0xFFC0` on `b_read_data_out`) and the pixels being drawn on screen (RGB at `0xFF`). This direct correlation is the ultimate proof of the system's correct operation.



```
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 6.5: 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 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 7

Conclusions

This report has detailed the design, implementation, and rigorous verification of a custom GPU subsystem tailored for a RISC-V based System-on-Chip. The project commenced with a clear objective: to develop a hardware module capable of rendering ASCII characters, received via a UART interface, onto an external display using the HDMI standard. This final chapter summarizes the project's achievements, reflects on the key design choices and challenges encountered, and explores potential avenues for future work.

7.1 Summary of Project Achievements

The project has successfully met all of its primary objectives, resulting in a fully functional and thoroughly verified video display solution. The key accomplishments can be summarized as follows:

- **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 simulation of the complete SoC. The final test successfully demonstrated the entire data flow: from a character being transmitted by a simulated host, through the UART, processed by the RISC-V core running the firmware, and finally rendered correctly on the video output.

- **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.

7.2 Key Design Choices and Challenges Encountered

Throughout the development process, several critical design decisions were made, often in response to specific hardware constraints and technical challenges.

A primary driver of the architecture was the limited on-chip SRAM available on the target FPGA. This constraint directly led to the adoption of a **character-based rendering** approach with a monochrome (1-bit per pixel) framebuffer. This choice proved highly effective, as it made the memory footprint manageable while still meeting the project's core requirement of displaying text.

The decision to use a **Dual-Port Block RAM** as the framebuffer was fundamental to the system's performance. It successfully decoupled the CPU's asynchronous, bursty writes from the HDMI controller's continuous, real-time read requirements. This was further enhanced by the implementation of an **anti-tearing mechanism** at the top level, which gates writes to the framebuffer, permitting them only during the video signal's blanking intervals. This small but critical piece of logic ensures a stable, artifact-free visual output.

The main challenge encountered was the need for precise **video timing synchronization**. The standard VESA timings had to be carefully adjusted to compensate for a system clock that was not an exact multiple of the required pixel clock, a common issue in real-world FPGA designs. The final, successful simulation on the full SoC demonstrates that these adjustments were effective.

7.3 Future Work and Potential Enhancements

While the current implementation is complete and functional, it also serves as a solid foundation for numerous potential enhancements. Future work could expand the GPU's capabilities in several key areas:

- **Color Support:** The most immediate enhancement would be the addition of color. This could be implemented with a simple 8-bit color palette, where each character could have a foreground and background color. This would require expanding the framebuffer, modifying the CPU-to-GPU communication protocol to include color information, and updating the `HDMI` module's rendering logic.
- **Hardware Acceleration:** To reduce the load on the RISC-V core and minimize bus traffic, several functions could be offloaded to hardware. For instance, a **Font ROM** could be integrated directly into the GPU. The CPU would then only need to send ASCII codes, and the GPU would handle the bitmap lookup internally. Further acceleration could include hardware-managed cursors, screen scrolling, and simple 2D primitives (e.g., drawing lines or rectangles).
- **DMA Integration:** For significantly higher performance, especially when updating large portions of the screen, a Direct Memory Access (DMA) controller could be integrated into the SoC. This would allow the CPU to command the DMA to transfer an entire block of data from main memory to the GPU's framebuffer directly, freeing the CPU to perform other tasks.
- **Support for Higher Resolutions:** The current architecture could be adapted to support higher resolutions like 720p. This would primarily require a larger framebuffer, a faster pixel clock, and updated timing parameters in the `HDMI` module.

7.3.1 Concluding Remarks

In conclusion, this project successfully demonstrates the complete design and verification flow of a custom peripheral for a modern RISC-V SoC. Through careful architectural design, robust firmware development, and a meticulous, multi-tiered verification strategy, a functional and reliable GPU subsystem was created. The project not only meets its initial requirements but also provides a versatile and extensible foundation for future explorations into embedded graphics on the Hummingbird E203 platform.

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] G. Giaretta and T. Magelli, “Design of hdmi display module for risc-v icb bus of hummingbird e203,” Unpublished student project report, Tongji University and University of Bologna (Double Degree Program), Jun. 2024.

[2] 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).