ECE 385 - Final Project Report

Software-driven FPGA Video Card

Alexander Greff
Due: 12/17/25

Introduction

For this final project, I continue the pixel drawing process I learned about in Lab 7, but instead of text drawing I am focusing entirely on sprite drawing. I wanted to follow a similar approach by leveraging the ability to create a custom IP and create my own AXI slave interface to interact with the Microblaze SOC in the microcontroller configuration. The goal is to build a full video card that is contained within a custom IP that behaves as an AXI-peripheral. It uses the HDMI Interface to output drawn pixels to the monitor. The Microblaze connects to the IP via AXI4-Lite and sends the sprite that it wants to draw on the monitor, as well as the coordinates, then it draws the sprites in the respective frame. It does the drawing and distinguishes between the foreground pixels and background pixels, merging the drawing logic into a frame that is sent to the HDMI display. The design uses double buffering, with two Block RAM (BRAM) IPs so one ram is displayed/read from while the other is being written to, and they swap once the display BRAM has been fully read out of.

The architecture of the project as a whole starts from lab 6.2 instead of lab 7, as it already provides the necessary AXI peripherals, like the GPIOs for keycodes and the AXI Quad SPI for the MAX3421E USB Host/Peripheral Controller. The project also has the correct amount of memory in the Microblaze, as I use a 64Kb on-chip BRAM as local memory, additionally I use the VGA to HDMI controller from that projector inside of the custom IP. I also knew that my drawing architecture would need to be fundamentally very different from the Lab 7 material, since I am not prioritizing the ability to draw text, as even if I wanted to do so I could easily just make text just be sprites that I read out in the ROMs.

Originally, the goal was to make a video game but I thought that using video game assets to show the power of the double buffered reader and writer as it serves as a good example of lots of sprites being drawn at once, including projectiles and the scrolling background. I also use the keyboard polling entirely on the software side and Hex Drivers, LEDs, Switches in Vitis, as I do not want any of the actual logic of the demo game to be in the hardware, since the software acts as the driver and I want the hardware to purely handle the responsibility of drawing. I created a full end-to-end accelerated graphics drawing pipeline that begins in software, which is entirely self-contained within a single IP. I can easily change the ROMs that want to be drawn and the background and make a separate project, allowing this to be a good base for any video game project on the Urbana board.

Drawing Logic/ Sprites:
In order to do the drawing logic, I needed a way to be able to distinguish between the background and foreground. I take all of the sprite images I found and put them vertically on a sheet of paper using photoshop, filling all background tiles with the color black. Then I used the Image to COE tool in order to generate a common color palette for all sprites, making sure that the color in index zero of the palette was black. This took several attempts of the K-means clustering algorithm in python in order to get black in index zero, but once I did that I split up the common COE file into several independent COE files, so that I could make several independent sprite ROMs that all share a common palette. This allows me to use my merging process for drawing the pixels. The logic is very simple: if the current pixel of the current sprite being drawn would be a black pixel (meaning I am drawing the background) instead of drawing the contents that come out of the sprite ROM, instead draw the current pixel contents coming out of the ROM. It is also worth mentioning that to simplify the ROM interface, I set it up so that every time I give any of the ROMs an address, they will return exactly one pixel at each instance.

Software Architecture

I will begin by giving a high-level overview of the software portion, and then move onto the hardware portion. The software portion is written entirely in C++ using the g++ compiler and linker. Since this project uses lab 6 as its starting point, the “Hello World” template is the template that was used to create the software portion. This program configuration does not support C++ out of the box, but overriding the little-endian requirement and use the following two lines of code within the C/C++ settings in Vitis to get the project up and running with C++:

```
-c -fmessage-length=0 -mlittle-endian -std=c++17 -mxl-barrel-shift -mxl-pattern-compare
-mcpu=v11.0 -mno-xl-soft-mul -Wl,--no-relax -ffunction-sections -fdata-sections -MT"$@"
```

```
${resolvePlatformFile:project=microblaze_ver2_usb,fileType=bspInclude}
```

However, there is still more work to be done on the user side, as the Microblaze and BSP expect a C-style entry point/main function, requiring the main program to still be C but calling supporting C++ functions and classes for the project. Below is a visual diagram of the software hierarchy:

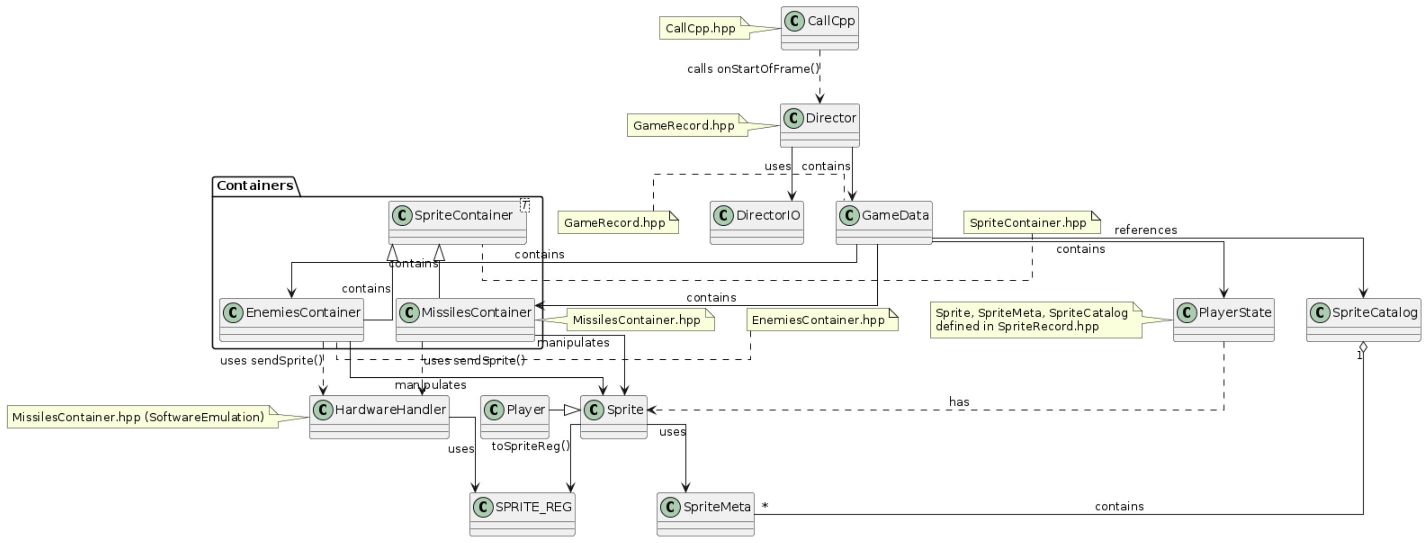


Figure 1: Vitis C++ Software Architecture

At runtime, the software updates game logic based on input and internal state, determines which sprites need to be rendered, and issues concise commands to the video card IP through AXI4-Lite writes. Each command specifies a sprite ID and a framebuffer base address, after which the hardware autonomously fetches sprite data, writes it into the framebuffer, and composites it into the video output. The software can poll the status register to determine when the hardware is ready for the next command, enabling simple synchronization without interrupts.

Overall, the software acts as a control and coordination layer rather than a renderer. This design leverages the strengths of hardware for real-time video output while keeping complex game logic, input handling, and state management in software, resulting in a clean and scalable video card system.

The entry point is callCpp, allowing for the smooth bridge between both languages in Vitis, as I can write highly performant code leveraging features like inlining to do lots of work at compile time. Additionally, I configured the release version of the code in Vitis with optimization flags so that it could condense the memory size of my code greatly. By default, the total size of the unoptimized program is 28Kb, while the optimizer allows me to size it down to 21Kb for the final version. For comparison, the default size of the blank “Hello World” C code to be compiled is 10Kb, so my code remains extremely memory efficient given the 64Kb constraint of the Microblaze. This size difference is due to the fact that function calls are optimized out of the program size using techniques only allowed in C++. The goal of the C++ is to control which sprites are drawn and on what frames, intercept keycodes from the keyboard, perform calculations for collision detection of sprites, interface with all Memory Mapped I/O peripheral base addresses, like the HEX display, on-board switches and LEDs, and poll my custom IP to know whether a sprite can be sent to the hardware at a given moment. The program initializes with just the background scrolling, as only the hardware runs until the microblaze boots up and the platform_init() function finishes running, then the sprites the user wants to draw can appear. The sprites appear rich and colorful but are stored with very small data structures known as bitfields, allowing the information that concerns the drawing of a single sprite to be represented by a mere 32 bits, and this is also a massive contributor to the small program size. C++ also allows for a natural exposition of very convenient internal functions like Xmax() for example, which quickly allows me to obtain the max sprite coordinates in the x-direction for any given sprite. In the Module Descriptions below there are more descriptions of what each specific file in the C++ diagram does. The collision detection is implemented extremely elegantly in C++ as well, as I only need to perform 4 checks at the corner dimensions of the sprites.

Hardware Architecture

The hardware architecture is very clean and follows a simple hierarchical structure. In the top file, I simply instantiate every module: the Director, the Multi-Sprite Reader, Sprite Writer, Frame Buffer (FB) manager, VGA controller, the Background (BG) Manager, the FB-BG Merger, and the provided VGA-to-HDMI controller. The drawing process begins in the Vitis portion of the project. The design choice for the software was to make it as optimized as possible and heavily objected-oriented, as the video card logic would entirely deal with the drawing of sprites. This is because the drawing of the background is handled entirely by the FPGA itself. Additionally, it would be the CPUs responsibility to perform any multiplications as in the hardware I avoid performing any multiplications, divisions, or modulo operations as they are very costly and I want to avoid having any issues meeting timing, notably with my Worst Negative Slack. Part of making this video card project was to try and push the performance of the hardware and software side, and carefully coming up with which side should handle which responsibilities.

The double buffering means I need two BRAMs in the Frame Buffer Manager module, but the problem is that if I made the BRAMs scaled to 16 color, 640*480 pixels output display, I would run out of on-chip memory on the Urbana Board, as the AMD Spartan 7 XC7S50 model only has 2.7Kb total Block RAM. To go around this memory constraint, I make the drawing logic for a 320*240 display instead, and I row and column doubling logic to upscale the output image to 640*480. This is a huge saving of memory because the X and Y dimensions are both halved so I have use only a quarter of the entire memory. Each of the large 24*24 pixel sprites take 18µs to draw, while the individual bullets take 1.5µs to draw. This can be contrasted with the time to draw an entire frame which for a 307200 pixel / 25MHz = 16.7 ms,

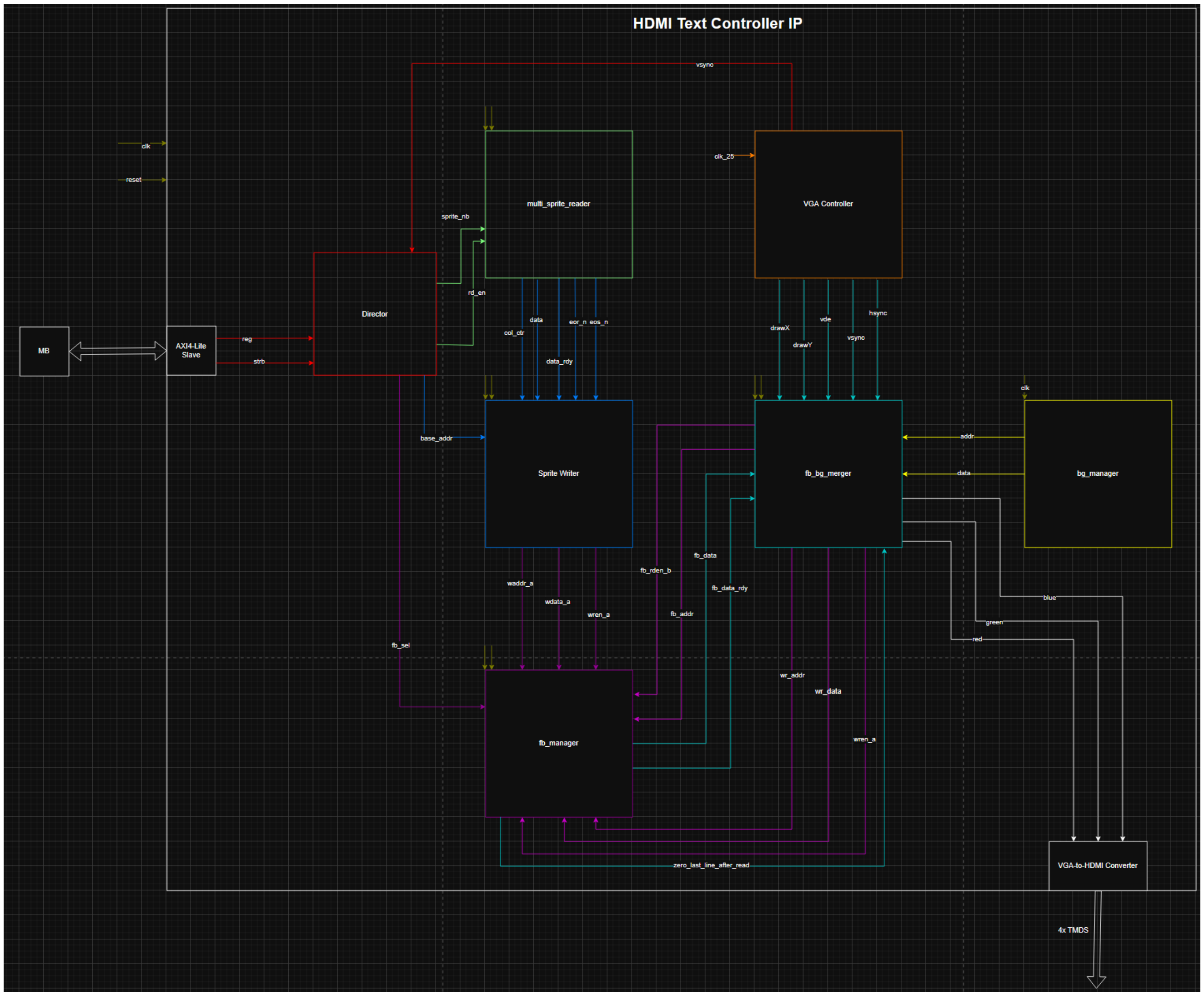


Figure 2: Vivado Hardware Architecture

As previously mentioned, the hardware rendering IP is entirely self contained with the IP called hdmi_text_controller, which should be named hdmi_graphic_controller but due to Vivado crashing, the IP packager xml file got corrupted and I had to follow the IP creation tutorial from Week 7 and didn't want to change any variables like module names in case there were any side effects. The entirety of the hardware project is contained within the IP circled in red, which allows me to be “plug and play” with any project as all logic is encapsulated within the IP.

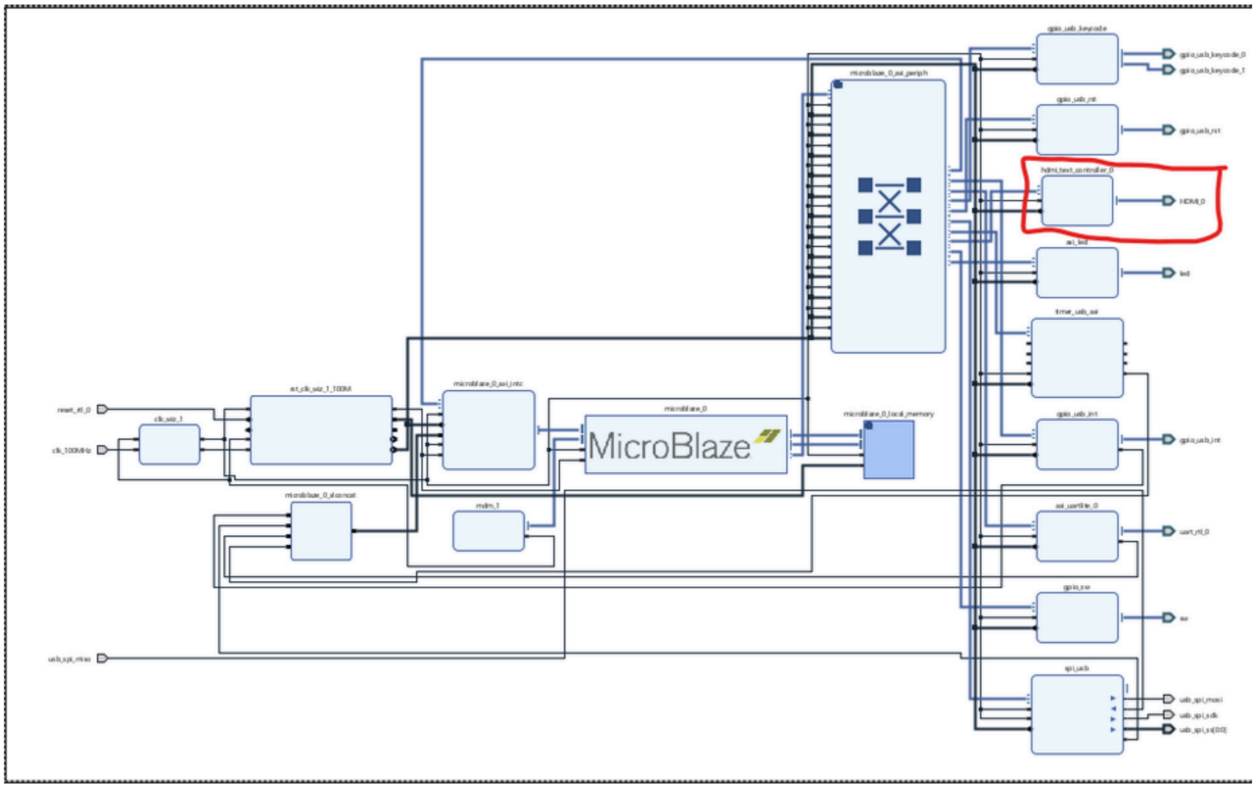
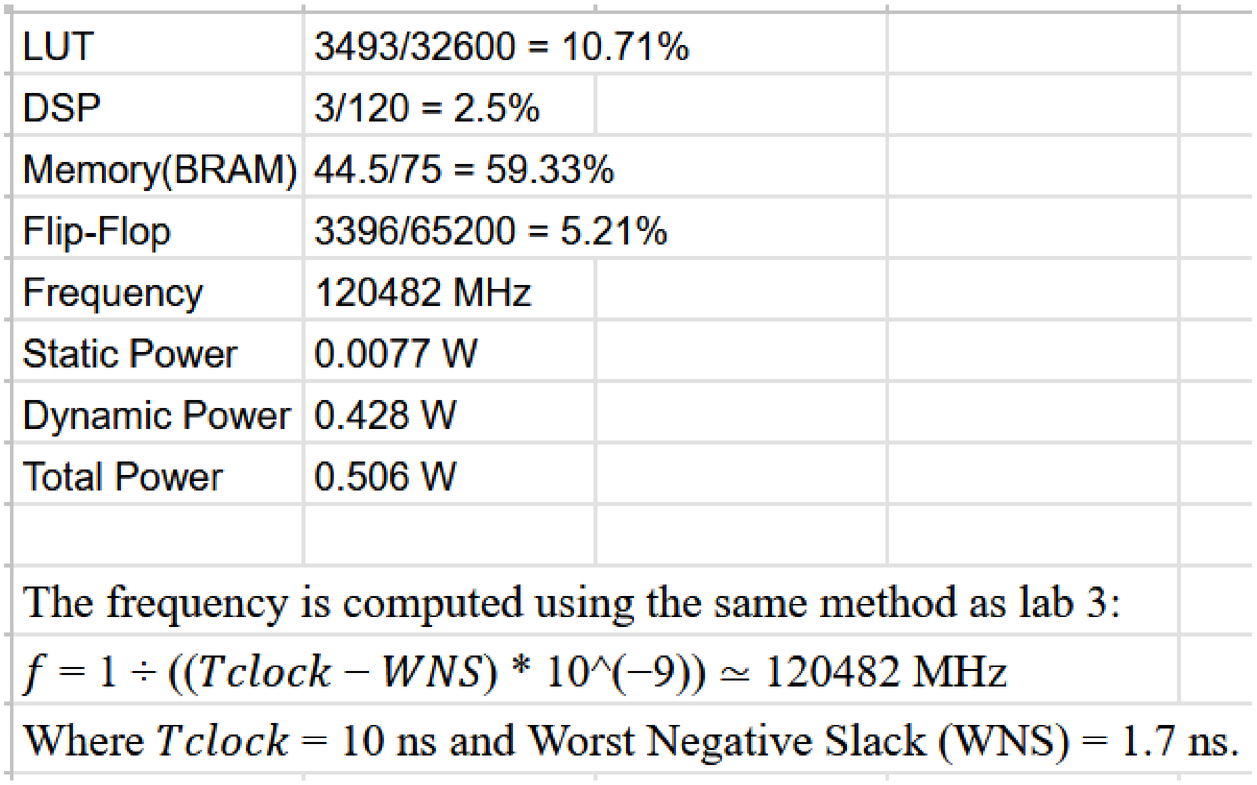


Figure 3: Vivado Block Diagram



LUT	3493/32600 = 10.71%
DSP	3/120 = 2.5%
Memory(BRAM)	44.5/75 = 59.33%
Flip-Flop	3396/65200 = 5.21%
Frequency	120482 MHz
Static Power	0.0077 W
Dynamic Power	0.428 W
Total Power	0.506 W

The frequency is computed using the same method as lab 3:

$f = 1 \div ((Tclock - WNS) * 10\text{^}(-9)) \simeq 120482$ MHz

Where $Tclock = 10$ ns and Worst Negative Slack (WNS) = 1.7 ns.

Figure 4: Design Resources and Statistics

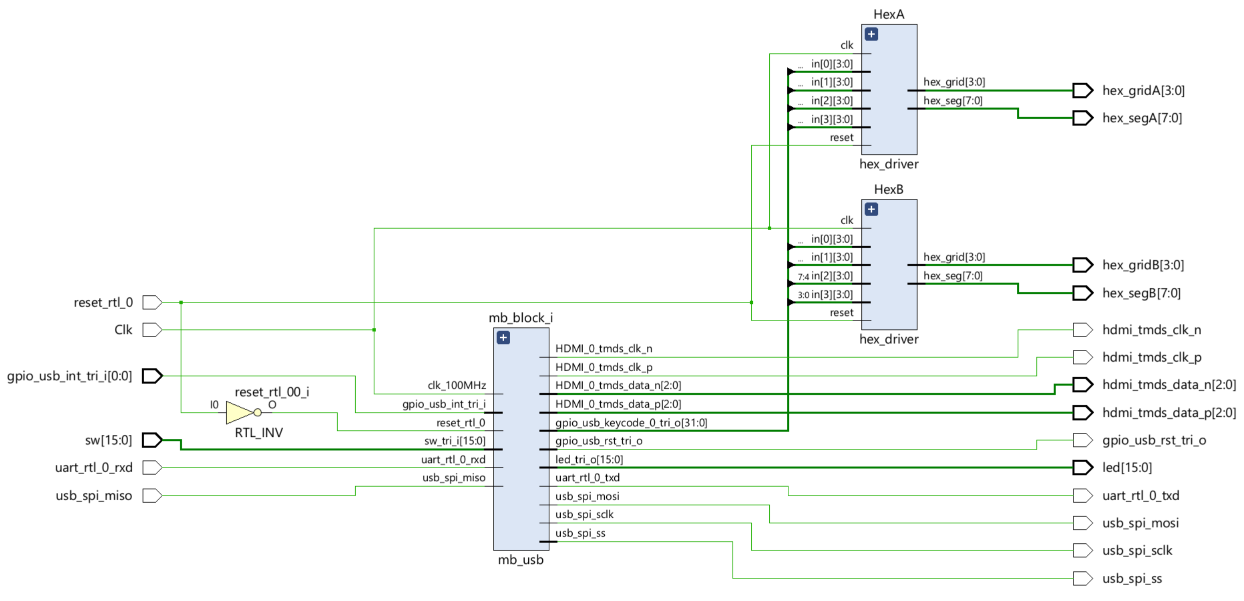
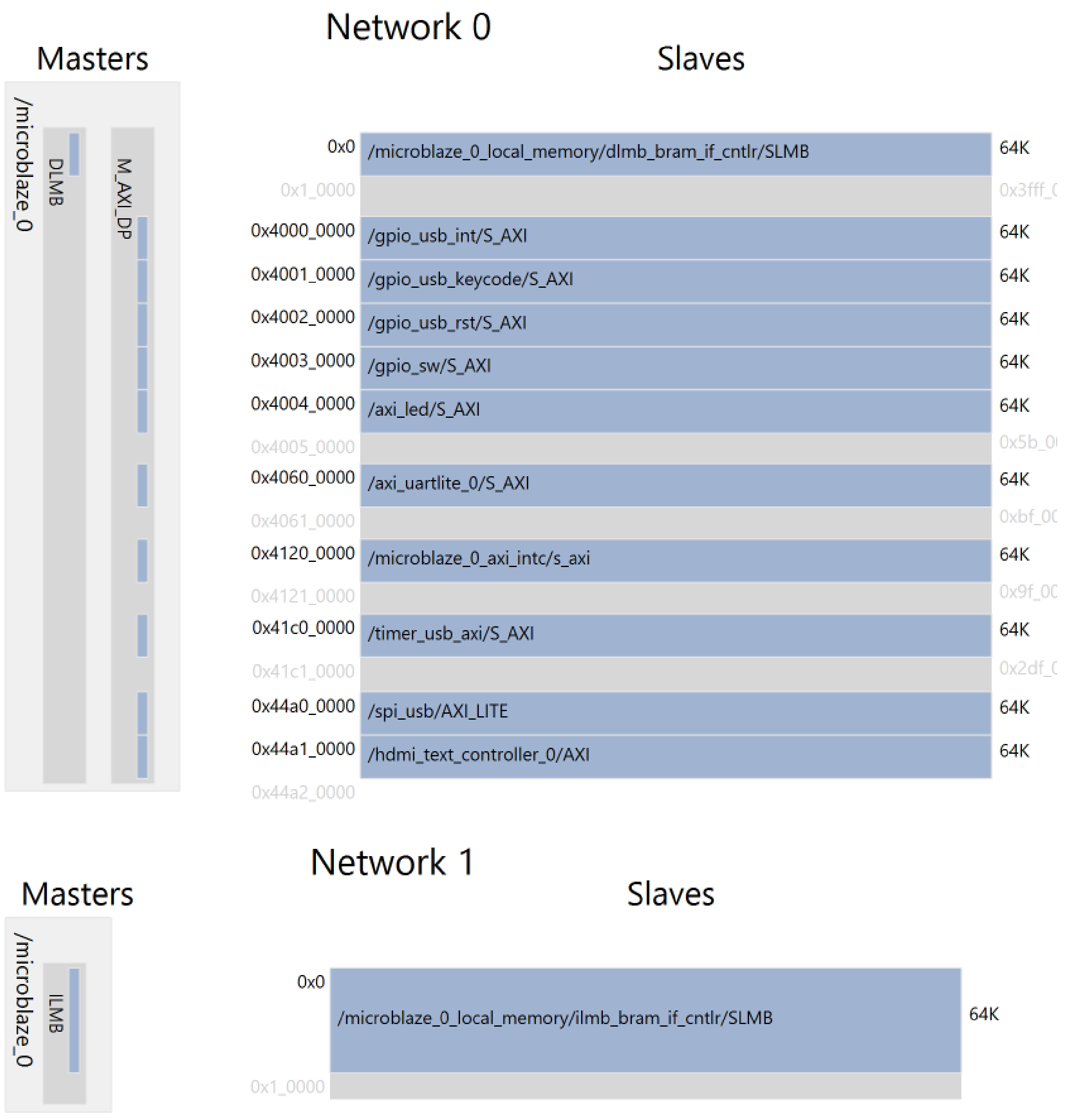


Figure 5: IP Elaborated Design schematic view



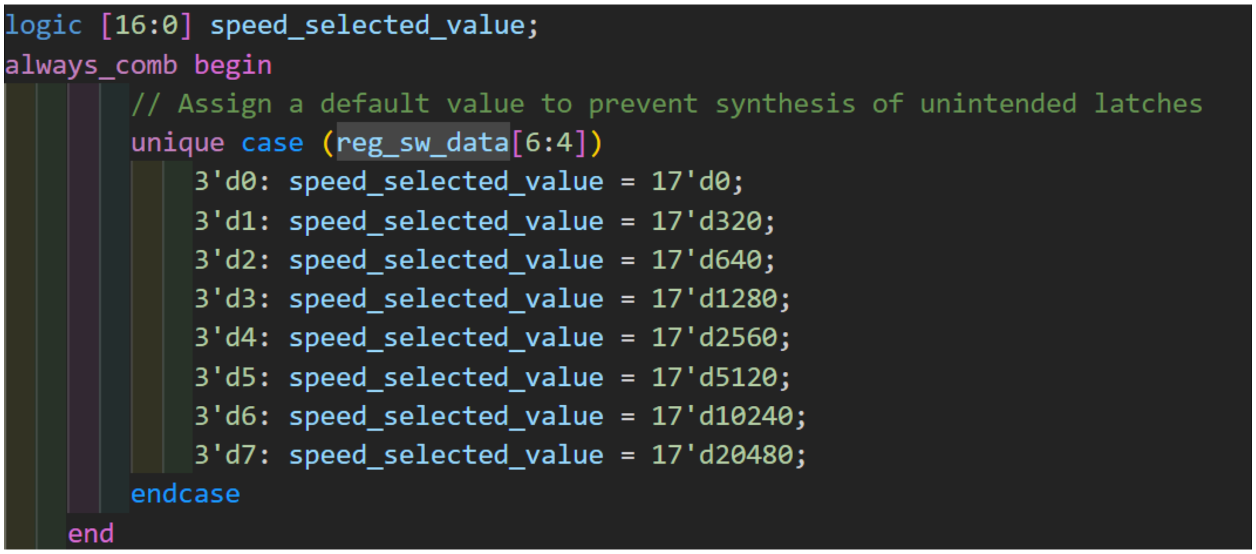
0x0	/microblaze_0_local_memory/dlmb_bram_if_cntlr/SLMB	64K
0x1_0000		0x3fff_(
0x4000_0000	/gpio_usb_int/S_AXI	64K
0x4001_0000	/gpio_usb_keycode/S_AXI	64K
0x4002_0000	/gpio_usb_rst/S_AXI	64K
0x4003_0000	/gpio_sw/S_AXI	64K
0x4004_0000	/axi_led/S_AXI	64K
0x4005_0000		0x5b_0
0x4060_0000	/axi_uartlite_0/S_AXI	64K
0x4061_0000		0xbf_0
0x4120_0000	/microblaze_0_axi_intc/s_axi	64K
0x4121_0000		0x9f_0
0x41c0_0000	/timer_usb_axi/S_AXI	64K
0x41c1_0000		0x2df_(
0x44a0_0000	/spi_usb/AXI_LITE	64K
0x44a1_0000	/hdmi_text_controller_0/AXI	64K
0x44a2_0000		

0x0	/microblaze_0_local_memory/ilmb_bram_if_cntlr/SLMB	64K
0x1_0000		

Figure 6: Address Map

Replicating the demo:
In the demo, I have the ship at the bottom shooting many projectiles at the other ships at the top of the screen. I have a demo override mode using the rightmost switch, switch 0. This allows me to be able to change some of the hardware features easily. So long as switch zero is set high, the following switches control the following behaviors:
Switch 1: allows the user to stop the zeroing of the frame buffer, leading to massive screen tearing.

Switch 2: changes the direction of the background scrolling hardware logic to make it scroll backwards.
Switch 3: controls whether or not the background scrolls or not, disables the hardware scrolling logic.
Switches 4-6: allows the user to select how fast the background scrolls. When all switches are set to high, the scroll speed is maximized.



```
logic [16:0] speed_selected_value;
always_comb begin
    // Assign a default value to prevent synthesis of unintended latches
    unique case (reg_sw_data[6:4])
        3'd0: speed_selected_value = 17'd0;
        3'd1: speed_selected_value = 17'd320;
        3'd2: speed_selected_value = 17'd640;
        3'd3: speed_selected_value = 17'd1280;
        3'd4: speed_selected_value = 17'd2560;
        3'd5: speed_selected_value = 17'd5120;
        3'd6: speed_selected_value = 17'd10240;
        3'd7: speed_selected_value = 17'd20480;
    endcase
end
```

Figure 6: Demo speed settings for onboard switches 4-6

Module Descriptions - SystemVerilog:

Module: bg_manager.sv
Inputs: cpu_clk, addr[16:0]
Outputs: data[1:0]

Description: This module is a simple wrapper around the background ROM used by the display system. It takes in the CPU clock and a 17-bit address, and it instantiates the bg1_rom memory block. On each clocked ROM read, the addr input selects a specific background entry, and the ROM outputs a 2-bit value on data. That 2-bit output represents the encoded background information for the selected address location (for example, a palette index, tile attribute, or background pixel category depending on how the rest of the video pipeline interprets it). The module itself contains no extra logic beyond wiring the address, clock, and output to the ROM instance.

Purpose: To provide a clean interface for accessing background data from a dedicated ROM, allowing the rest of the graphics/text pipeline to fetch compact 2-bit background values by address without directly managing the ROM instantiation details.

Module: bg_palette.sv
Inputs: index[1:0]
Outputs: red[3:0], green[3:0], blue[3:0]

Description: This module implements a small, hard-coded color palette for background rendering. It takes a 2-bit index input and uses it to select one of four predefined RGB color entries stored in a local parameter array. Each palette entry consists of three 4-bit values corresponding to the red, green, and blue color channels. The selected 12-bit RGB value is then unpacked and assigned to the red, green, and blue outputs using a single combinational assignment. The logic is purely combinational, so any change in the input index immediately updates the output color.

Purpose: To translate compact 2-bit background color indices into full 12-bit RGB values that can be directly driven to the display pipeline. This allows background graphics to use small memory-efficient indices while still producing the correct visible colors on the screen.

Module: clk_wiz_0.sv
Inputs: clk_in1, reset
Outputs: clk_out1, clk_out2, clk_out3, locked

Description: This module implements a simplified clock generation and distribution block intended to mimic the behavior of a clocking wizard in simulation. It takes a single 100 MHz input clock (clk_in1) and produces multiple derived clock outputs. The clk_out3 output is a direct pass-through of the input clock, guaranteeing it remains exactly in phase with clk_in1. The clk_out2 output is also assigned directly from the input clock and serves as an additional pass-through clock for logic expecting a higher-frequency or shared clock domain. The clk_out1 output is generated by dividing the 100 MHz input clock by four using a 2-bit counter, producing an effective 25 MHz clock. The divider logic is synchronous to clk_in1 and is resettable using an active-high reset, which initializes the counter and forces clk_out1 low to ensure a known phase relationship after reset. The locked signal is hardwired high to indicate a stable clock condition for simulation purposes, bypassing the complexity of real PLL lock behavior.

Purpose: To provide multiple clock signals with known frequencies and phase relationships for the system, including a 25 MHz pixel clock and pass-through 100 MHz clocks, while simplifying simulation by avoiding a full PLL/MMCM implementation. This module allows the rest of the design to be clocked and reset consistently during development and testing.

Module: director.sv
Inputs: reset, cpu_clk, reg_command[31:0], reg_strobe, vga_vsync, sp_writer_eos_n
Outputs: sprite_nb[7:0], base_addr[16:0], rd_en, reg_status[31:0], frame_counter[31:0], frame_ticks[31:0], clock_ticks[31:0], merger_fb_sel, writer_fb_sel

Description: This module acts as the control "director" that sequences sprite fetch operations and manages frame-buffer selection/timing. It monitors a CPU-written command interface (reg_command + reg_strobe) and, when a new command strobe is detected on a rising edge, it latches the requested sprite number (reg_command[31:24]) and a base memory address (reg_command[16:0]). It then asserts rd_en to initiate a read/transfer of that sprite's data into the framebuffer path, and sets an internal is_busy flag to indicate an active operation. The module deasserts rd_en when it detects the falling edge of reg_strobe, so the read-enable behaves like a pulse aligned to the strobe window. Completion of the sprite-writing process is detected using sp_writer_eos_n, an active-low "end of sprite" signal; on its falling edge, the director clears is_busy, allowing new commands to be accepted. The module also tracks display timing by edge-detecting vga_vsync (on its falling edge) to increment a frame_counter once per frame and reset frame_ticks. Between vsync events, frame_ticks increments continuously as a per-frame cycle counter, and clock_ticks increments as a free-running uptime counter. Finally, the module selects which framebuffer is read vs written using the LSB of frame_counter: merger_fb_sel is the inverse and writer_fb_sel is the direct value, implementing a ping-pong (double-buffered) scheme.

Purpose: To coordinate CPU-initiated sprite rendering by issuing sprite/address read commands, reporting busy/ready status, and synchronizing double-buffered framebuffer selection with the display's frame timing. This ensures sprites are transferred and written at the right times while the VGA/HDMI output can continuously read from the opposite framebuffer without tearing.

Module: fb_bg_merger.sv
Inputs: reset, cpu_clk, drawX[9:0], drawY[9:0], vde, vsync, hsync, fb_data[3:0], fb_data_rdy, bg_data[1:0], reg_strobe_sw, reg_switch[31:0], zero_last_line_after_read_hd
Outputs: fb_rden_b, fb_addr[16:0], bg_addr[16:0], red[3:0], green[3:0], blue[3:0], fb_merger_wr_addr[16:0], fb_merger_wr_data[3:0], fb_merger_wren_a

Description: This module merges foreground pixel data from the frame buffer with background pixel data from a background source and outputs final RGB values for the display. It converts the incoming 640×480 VGA scan coordinates (drawX, drawY) into a 320×240 "logical" coordinate space by counting drawX in steps of 2 pixels (using drawX[0] edge detection) and duplicating lines based on drawY[0]. Using these scaled coordinates, it generates a frame-buffer read address (fb_addr = drawX_320 + cumm_drawY_320) and a background address (bg_addr) that either matches the normal background mapping or includes a scrolling offset when scrolling is enabled. The module asserts fb_rden_b not only during the visible region (vsync & vde) but also during a prefetch window near the end of each line (drawX >= 798), ensuring frame-buffer data is ready at the start of the next line. Foreground pixels are treated as transparent when fb_data equals 0; in that case, the background color is output instead. Otherwise, the foreground color is output. Both foreground and background indices are converted into 12-bit RGB using palette lookup modules (fg_palette and bg_palette).

In addition, this module contains scrolling logic that updates a starting background offset once per frame and advances a "current line" offset as lines progress, supporting forward/backward scrolling with wraparound behavior across the full 320×240 image space (76800 pixels). The scroll behavior can be controlled either by hardware defaults or by CPU-configurable switch registers latched on reg_strobe_sw, allowing runtime enabling/disabling of scrolling, direction selection, and speed selection. Finally, it supports an optional "zero last line after read" feature: when enabled, it initiates a writeback sequence (fb_merger_wren_a) that writes zeros across an entire 320-pixel row in the frame buffer using fb_merger_wr_addr and fb_merger_wr_data, clearing the last-read line to prevent artifacts or to support streaming-style rendering.

Purpose: To produce the final on-screen RGB output by compositing a transparent-capable 4-bit foreground framebuffer over a 2-bit indexed background, while handling address generation, prefetch timing, optional vertical background scrolling, and optional clearing of lines after they are consumed. This module is the key "pixel combiner" that makes the double-buffered framebuffer and background system appear as one coherent video image.

Module: fg_palette.sv
Inputs: index[3:0]
Outputs: red[3:0], green[3:0], blue[3:0]

Description: This module implements a fixed foreground color palette used for rendering sprite and text pixels. It accepts a 4-bit color index and uses it to select one of sixteen predefined RGB color entries stored in a local parameter array. Each palette entry consists of three 4-bit values representing the red, green, and blue intensity levels. The selected 12-bit RGB value is then unpacked and driven onto the red, green, and blue outputs through a purely combinational assignment, so the output color updates immediately when the index changes.

Purpose: To convert 4-bit foreground color indices from the frame buffer into full 12-bit RGB values for display. This allows sprites and text to reference colors compactly in memory while still supporting a richer, predefined color set on the screen.

Module: hdmi_text_controller_v1_0_AXI.sv
Inputs: drawX[9:0], drawY[9:0], frame_count[31:0], reg_status[31:0], frame_ticks[31:0], clock_ticks[31:0], S_AXI_ACLK, S_AXI_ARESETN, S_AXI_AWADDR[C_S_AXI_ADDR_WIDTH-1:0], S_AXI_AWPROT[2:0], S_AXI_AWVALID, S_AXI_WDATA[31:0], S_AXI_WSTRB[3:0], S_AXI_WVALID, S_AXI_BREADY, S_AXI_ARADDR[C_S_AXI_ADDR_WIDTH-1:0], S_AXI_ARPROT[2:0], S_AXI_ARVALID, S_AXI_RREADY
Outputs: reg_strobe, reg_strobe_sw, reg_command[31:0], reg_switch[31:0], S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP[1:0], S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA[31:0], S_AXI_RRESP[1:0], S_AXI_RVALID

Description: This module implements an AXI4-Lite slave peripheral that exposes a small memory-mapped register interface to the CPU and also provides a few “special” read-only status/debug registers related to the video system. On the AXI write path, it accepts write transactions when both the write address and write data channels are valid, latches the write address, and writes the incoming 32-bit data into an internal register array (slv_regs[]) using WSTRB byte strobes so partial-word writes are supported. It then returns an OKAY write response on the AXI B channel. On the AXI read path, it latches the read address when ARVALID is asserted and returns either the contents of slv_regs[] or one of several special values based on the decoded address: frame count, packed draw position information, frame tick count, clock tick count, and an externally provided reg_status value.

For user-facing control, the module treats specific registers as command/control registers. Writes to register index 4 produce reg_command and generate a one-cycle reg_strobe pulse (using a delayed write-enable) to signal downstream logic that a new command has been issued. Similarly, writes to register index 6 produce reg_switch and generate a one-cycle reg_strobe_sw pulse to latch switch/configuration values in other modules. This design cleanly separates the AXI protocol handling from the rest of the video/sprite system by converting bus writes into simple internal registers plus short “new data” strobes.

Purpose: To serve as the CPU’s AXI4-Lite interface into the HDMI/video subsystem by (1) allowing software to write command and configuration registers that control hardware behavior via reg_command/reg_switch and their strobes, and (2) allowing software to read back useful live status/debug information such as frame counters, draw coordinates, and timing tick counters. This module is the bridge between software running on the processor and the real-time display/rendering logic in hardware.

Module: hdmi_text_controller_v1_0.sv
Inputs: axi_aclk, axi_aresetn, axi_awaddr[C_AXI_ADDR_WIDTH-1:0], axi_awprot[2:0], axi_awvalid, axi_wdata[31:0], axi_wstrb[3:0], axi_wvalid, axi_bready, axi_araddr[C_AXI_ADDR_WIDTH-1:0], axi_arprot[2:0], axi_arvalid, axi_rready
Outputs: hdmi_clk_n, hdmi_clk_p, hdmi_tx_n[2:0], hdmi_tx_p[2:0], axi_awready, axi_wready, axi_bresp[1:0], axi_bvalid, axi_arready, axi_rdata[31:0], axi_rresp[1:0], axi_rvalid

Description: This module is the top-level integration wrapper for the HDMI/text/sprite display subsystem with an AXI4-Lite control interface. It accepts AXI clock/reset and the full AXI4-Lite read/write channels from the processor and connects them to the internal AXI slave module (hdmi_text_controller_v1_0_AXI), which exposes software-visible control and status registers. On the video side, it generates the required internal clocks using clk_wiz_0, producing a 25 MHz pixel clock for VGA timing and a higher-frequency clock intended for HDMI serialization. A vga_controller module uses the 25 MHz pixel clock to generate the horizontal/vertical sync (hsync, vsync), video data enable (vde), and current pixel coordinates (drawX, drawY).

The module then instantiates the main sprite rendering pipeline. A director monitors CPU-written commands (via reg_command and reg_strobe) and issues sprite fetch requests by outputting the sprite number, base address, and read enable while also maintaining timing counters (frame_count, frame_ticks, clock_ticks) and reporting busy status back to software (reg_status). A multi_sprite_reader reads sprite pixel data and generates end-of-row/end-of-sprite markers along with valid pixel nibbles. The sprite_writer converts that stream into write transactions (address/data/write-enable) targeting the frame buffer. The fb_manager provides the actual dual-ported frame buffer behavior: one port is written by the sprite writer, while the other port is read during display to produce foreground pixel indices (fb_data) along with a ready/valid indication.

For final pixel generation, fb_bg_merger combines the foreground framebuffer output with background ROM data. It produces the final 12-bit RGB values (red, green, blue) based on transparency rules and optional background scrolling, while also generating the display-time frame-buffer read address and read-enable. A bg_manager wraps the background ROM and supplies the 2-bit background index values used by the merger. In this specific version, the actual VGA-to-HDMI TMDS transmitter IP is commented out and the HDMI differential outputs are forced to zero, meaning the module currently acts as a functional simulation/integration top-level for the rendering pipeline even though it is not actively driving a real HDMI signal.

Purpose: To serve as the complete top-level hardware block that connects the CPU (via AXI4-Lite) to the real-time video pipeline, generating timing signals, managing sprite rendering into a framebuffer, compositing foreground sprites with a background image, and producing final

RGB pixel outputs for HDMI/VGA display. This module is the system “glue” that ties together clocking, bus control, rendering, memory, and pixel output into one cohesive design.

Module: hdmi_text_controller_v1_0.sv
Inputs: axi_aclk, axi_aresetn, axi_awaddr[C_AXI_ADDR_WIDTH-1:0], axi_awprot[2:0], axi_awvalid, axi_wdata[C_AXI_DATA_WIDTH-1:0], axi_wstrb[(C_AXI_DATA_WIDTH/8)-1:0], axi_wvalid, axi_bready, axi_araddr[C_AXI_ADDR_WIDTH-1:0], axi_arprot[2:0], axi_arvalid, axi_rready
Outputs: hdmi_clk_n, hdmi_clk_p, hdmi_tx_n[2:0], hdmi_tx_p[2:0], axi_awready, axi_wready, axi_bresp[1:0], axi_bvalid, axi_arready, axi_rdata[C_AXI_DATA_WIDTH-1:0], axi_rresp[1:0], axi_rvalid

Description: This module is the top-level integration wrapper for the HDMI display subsystem with an AXI4-Lite control interface. It generates internal clocks using clk_wiz_0, producing a 25 MHz pixel clock (clk_25MHz) and a 125 MHz clock (clk_125MHz) for HDMI serialization, along with a 100 MHz clock used for framebuffer/background logic. A vga_controller runs on the 25 MHz pixel clock to generate the VGA timing signals (hsync, vsync, vde) and the current pixel coordinates (drawX, drawY). These timing and RGB signals are then passed into the RealDigital VGA-to-HDMI transmitter IP (hdmi_tx_0), which outputs differential TMDS signals (hdmi_clk_*, hdmi_tx_*) suitable for an HDMI display.

On the control side, the module instantiates hdmi_text_controller_v1_0_AXI, an AXI4-Lite slave that provides memory-mapped registers for software. This AXI block outputs reg_command with a pulse reg_strobe when the CPU writes a new command, and outputs reg_switch with a pulse reg_strobe_sw when the CPU updates switch/configuration settings. It also exposes readback/status information (frame counters, tick counters, and busy/status through reg_status).

The rendering pipeline is built from several submodules. The director consumes CPU commands and issues sprite fetch requests (sprite number, base address, and read enable), tracks frame timing counters, reports busy status, and generates ping-pong framebuffer select signals (merger_fb_sel and writer_fb_sel) for double buffering. The multi_sprite_reader streams sprite pixel data and control markers (end-of-row/end-of-sprite), and the sprite_writer converts that stream into framebuffer write transactions. The fb_manager implements the dual-port framebuffer behavior: one side is written by the sprite writer, while the other side is read during display for foreground pixel indices. The fb_bg_merger then composites the foreground framebuffer data over a background image read from bg_manager, applying transparency rules and optional scrolling/behavior controlled by reg_switch. It outputs the final 12-bit RGB values (red, green, blue) synchronized to the VGA timing, which are ultimately encoded and driven out over HDMI by hdmi_tx_0.

Purpose: To provide a complete hardware display system that bridges software control (AXI4-Lite) with real-time HDMI video output. This top-level ties together clock generation, VGA timing, AXI register access, sprite rendering into a double-buffered framebuffer, background fetching and compositing (including CPU-controlled effects like scrolling), and finally HDMI TMDS signaling so the rendered scene appears correctly on an external display.

Module: hdmi_tx_0.sv
Inputs: pix_clk, pix_clkx5, pix_clk_locked, rst, red[3:0], green[3:0], blue[3:0], hsync, vsync, vde, aux0_din[3:0], aux1_din[3:0], aux2_din[3:0], ade
Outputs: TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P[2:0], TMDS_DATA_N[2:0]

Description: This module is a stub/black-box wrapper for the RealDigital VGA-to-HDMI (TMDS) transmitter IP. It defines the standard HDMI transmitter interface, taking in the pixel clock (pix_clk), a 5× pixel clock (pix_clkx5) for serialization, a lock indicator (pix_clk_locked), and a reset. It also accepts the 4-bit-per-channel RGB pixel data along with VGA timing signals (hsync, vsync, and vde) and auxiliary data inputs (aux*_din and ade) which are unused in this design. The synthesis directive marks it as a black box to match the expected IP interface during implementation, while in simulation this stub simply drives all TMDS differential outputs to zero using constant assignments.

Purpose: To provide the expected module/interface for the HDMI TMDS transmitter IP so the top-level design can compile and integrate cleanly. In hardware, this block would encode RGB + sync/control into TMDS signals for HDMI output, while in simulation (or when the actual IP is not included) this stub prevents build failures by safely tying the HDMI outputs low.

Module: multi_sprite_reader.sv
Inputs: reset, sprite_nb[7:0], rd_en, cpu_clk
Outputs: eos_n, eor_n, data[3:0], col_ctr[4:0], data_rdy

Description: This module is a top-level selector that routes sprite read requests to one of two specialized sprite reader engines: one for “ships” sprites and one for “missiles” sprites. The most-significant bit of sprite_nb (sprite_nb[7]) determines which sprite type to use. Based on that selection, the module forwards the common inputs (reset, clock, and sprite number) to both sub-readers, but gates the rd_en signal so that only the selected reader is actively enabled. It then multiplexes the outputs (eos_n, eor_n, data, col_ctr, and data_rdy) from the active reader back to the module outputs. All routing is combinational, so switching the sprite type changes which reader drives the outputs immediately, while the actual sprite streaming behavior is handled inside the chosen submodule.

Purpose: To provide a unified sprite-reading interface for the rest of the rendering pipeline while supporting multiple sprite memory formats/sources. This allows the system to request sprite data using a single sprite_nb field, while internally selecting the correct sprite ROM/reader logic (ships vs missiles) and producing a consistent output stream of pixel nibbles with end-of-row/end-of-sprite markers.

Module: sprite_reader_missiles.sv
Inputs: reset, sprite_nb[7:0], rd_en, cpu_clk
Outputs: eos_n, eor_n, data[3:0], col_ctr[4:0], data_rdy

Description: This module streams pixel data for "missile" sprites from ROM and presents it in a simple sequential interface for the sprite writing pipeline. When rd_en has a rising edge, the module latches the requested sprite number (sprite_nb_q), resets its internal ROM address counter to 0, and enters an active read state (is_rd = 1). Missile sprites are treated as fixed-size images of 5 pixels wide by 10 pixels tall, stored as 50 bytes total (addresses 0 through 49), with each ROM location producing a 4-bit pixel value (data).

While active, the module increments a small rom_rd_latency counter and only asserts data_rdy when the latency reaches a fixed value (here rom_rd_latency == 2), which effectively models the synchronous ROM read delay so downstream logic samples valid pixel data at the correct time. On each valid output cycle, it selects which missile ROM to use based on the latched sprite_nb_q (e.g., sprite 10 uses test_missiles_rom, sprite 11 uses missiles1_rom; others default to zero). It also tracks the current column within a row using col_ctr (0 to 4). The module asserts eor_n (active-low end-of-row) and eos_n (active-low end-of-sprite) to mark boundaries: eor_n is driven low when a row finishes (after 5 columns), and eos_n is driven low when the final byte (address 49) has been output and the sprite is complete, at which point is_rd is cleared and data_rdy returns low.

ROM access is done through synchronous block-ROM style modules clocked by cpu_clk, using the lower bits of rom_address as the ROM index.

Purpose: To provide a deterministic, clocked pixel stream for missile sprites so they can be written into the framebuffer by the sprite writer. This module abstracts away the sprite ROM storage and exposes a consistent interface (data, data_rdy, eor_n, eos_n, col_ctr) that downstream rendering logic can use to place missile graphics correctly in memory.

Module: sprite_reader_ships.sv
Inputs: reset, sprite_nb[7:0], rd_en, cpu_clk
Outputs: eos_n, eor_n, data[3:0], col_ctr[4:0], data_rdy

Description: This module streams pixel data for “ship” sprites from ROM and outputs it as a sequential 4-bit-per-pixel data stream suitable for the sprite writing pipeline. On a rising edge of rd_en, the module latches the requested sprite number into sprite_nb_q, resets the internal ROM address to 0, clears the column counter (col_ctr), and enters an active read state (is_rd = 1). Ship sprites are treated as fixed-size images of 24 pixels wide by 24 pixels tall, stored as 576 bytes total (ROM addresses 0 through 575). Each ROM address returns one 4-bit pixel index (data).

Because the sprite ROMs are synchronous, the module uses a small rom_rd_latency counter to wait a fixed number of cycles before asserting data_rdy. When rom_rd_latency == 2, the output pixel is considered valid: data_rdy is asserted, the selected ROM output is driven onto data, and the module advances its read position. The sprite image source is chosen using a case on sprite_nb_q—sprite numbers 128–134 map to ships1_rom through ships6_rom and test_rom; any unsupported sprite number outputs zero and prints a debug message.

Row/column progression is tracked using col_ctr. For each valid pixel, the module increments rom_address and keeps eor_n and eos_n high (inactive) while the sprite is still being streamed. When col_ctr reaches 23 (end of a 24-pixel row), it resets col_ctr to 0 and drives low (active-low end-of-row). When the final ROM address (575) has been output and the final row completes, the module exits the read state (is_rd = 0) and drives eos_n low (active-low end-of-sprite), signaling downstream logic that the sprite stream is finished.

Purpose: To provide a consistent, clocked pixel stream for 24×24 ship sprites, including timing (data_rdy) and boundary markers (eor_n, eos_n), so that downstream modules (like the sprite writer) can correctly place ship graphics into the framebuffer without needing to understand the underlying ROM organization or synchronous read latency.

Module: sprite_writer.sv
Inputs: reset, cpu_clk, eos_n, eor_n, data_rdy, base_addr[16:0], col_ctr[4:0], data_in[3:0]
Outputs: waddr[16:0], wdata[3:0], wren

Description: This module takes the streamed sprite pixel data coming from the sprite reader and writes it into the framebuffer memory using an address/data/write-enable interface. The sprite reader provides a continuous sequence of 4-bit pixel indices (data_in) along with two active-low boundary signals: eor_n (end-of-row) and eos_n (end-of-sprite). The writer uses these markers plus the current column index (col_ctr) and a starting framebuffer address (base_addr) to compute the correct framebuffer write address for each pixel.

Internally, the module tracks the current row's base address in prev_row_base_addr, and counts rows using row_count. At the start of each new row (detected by a rising edge on eor_n, i.e., eor_n_prev==0 and eor_n==1), it sets up the base address for that row. For the first row, it loads prev_row_base_addr <= base_addr and asserts wren to begin writing. For subsequent rows, it advances the row base by 320 (+17'd320), which matches the framebuffer width in pixels for the 320×240 backing resolution.

For each valid pixel, when data_rdy is asserted, the module updates waddr to prev_row_base_addr + col_ctr, placing the pixel at the correct column within the current row. The module writes the pixel value by driving wdata <= data_in while eor_n is high (meaning the row is actively being streamed). There is also a guard col_ctr != 0 noted as a "VG BUG" workaround to avoid a problematic write at column 0 during streaming.

Row advancement is controlled using the falling edge of eor_n (detecting eor_n_prev==1 and eor_n==0), which indicates the row has completed. If the sprite is still active (eos_n is high), the module increments row_count so the next row start will shift down by another 320 pixels. When the sprite ends (eos_n == 0), the module resets internal state (row_count, prev_row_base_addr, waddr) and deasserts wren, cleanly stopping framebuffer writes until the next sprite stream begins.

Purpose: To convert the sprite reader's sequential pixel stream into correctly-addressed framebuffer writes, placing a sprite into the 320×240 framebuffer at the specified base_addr, advancing one framebuffer row per sprite row (stride 320), and cleanly gating writes using row/sprite boundary markers and the end-of-sprite condition.

Module Descriptions - Testbenches:

Module: hdmi_text_controller_tb.sv
Inputs: None (self-contained testbench)
Outputs: None (drives DUT signals internally; produces optional BMP output file in simulation)

Description: This module is a SystemVerilog testbench used to simulate the HDMI text controller AXI4-Lite IP and optionally generate a full-frame "video capture" image from the design. It creates an AXI clock (aclk) and active-low reset (arstn), instantiates the DUT (hdmi_text_controller_v1_0), and drives the full AXI4-Lite read/write channels using two tasks: axi_write() (provided) and axi_read() (implemented in the testbench). These tasks perform the standard AXI4-Lite handshakes by asserting valid signals, waiting for ready signals, and completing the transaction once responses/data are received.

When SIM_VIDEO is enabled, the testbench also taps internal DUT signals (hierarchical references) to observe the pixel clock, sync signals (hsync, vsync, vde), draw coordinates (drawX, drawY), and RGB values (red, green, blue). It logs pixel color values into a 2D bitmap buffer sized to the full 800×525 VGA timing space (including blanking). A BMP-writing task (save_bmp) then writes the bitmap to disk once a new frame begins (detected by vsync going low), producing a screenshot-style output for debugging the rendered image.

The test vectors initialize the system, write palette registers, clear/fill VRAM with spaces, and then write a sequence of character/color words into specific VRAM locations to display a known message on screen. This provides both an AXI correctness test (writes/reads) and a visual correctness test (does the expected text appear in the rendered frame). The testbench includes additional commented-out sections for alternate AXI readback verification and broader VRAM fill testing.

Purpose: To verify correct AXI4-Lite behavior (write/read handshakes and register correctness) and to provide a practical visual validation method by capturing the simulated VGA/HDMI pixel output into a BMP image. This allows rapid debugging of VRAM writes, palette handling, text rendering, and timing/coordinate logic without needing to run on hardware.

Module: new_top_ip_internal_tb.sv
Inputs: None (self-contained internal integration testbench)
Outputs: None (drives internal signals directly to exercise the pipeline)

Description: This module is an internal integration testbench that simulates the full sprite-to-framebuffer-to-display pipeline without running the AXI4-Lite bus transactions. Instead of instantiating the full hdmi_text_controller_v1_0 IP wrapper, it directly instantiates the key internal blocks (VGA timing, director, sprite reader/writer, framebuffer manager, background merger, and background ROM) and provides its own generated clocks. It creates both a 100 MHz clock (clk_100MHz) for the "CPU/render" side and a 25 MHz clock (clk_25MHz) for the VGA timing generator. A vga_controller produces hsync, vsync, vde, and pixel coordinates (drawX, drawY), which drive the pixel-compositing stage.

On the rendering side, the testbench manually drives reg_command and reg_strobe to simulate a CPU issuing a sprite draw command, causing the director to kick off a sprite read (rd_en) and generate the sprite number/base address. The multi_sprite_reader streams sprite pixel nibbles and row/sprite termination flags, and sprite_writer converts that stream into framebuffer write transactions. The fb_manager stores these writes and provides foreground pixel reads during display time. The fb_bg_merger then composites the foreground framebuffer pixels over the background fetched from bg_manager, outputting final red/green/blue values aligned with VGA timing.

The initial stimulus sequence in the testbench applies resets, sets the framebuffer select used for reading, issues a sprite command (reg_command = {sprite_nb, …, base_addr}), waits for the writer activity to finish, flips the framebuffer select, releases the VGA reset, and then waits for specific drawX/drawY coordinate conditions to confirm the pipeline is actively rendering through the merger. This makes it a fast way to debug the internal datapath, timing, and state sequencing without needing full AXI or HDMI/TMDS simulation.

Purpose: To provide a lightweight "internal top-level" simulation environment for quickly verifying that the sprite rendering pipeline works end-to-end: command → sprite read → sprite write → framebuffer read → background merge → RGB output. This testbench is useful for debugging reset behavior, framebuffer selection, sprite draw completion, and VGA-coordinate-driven address generation without the complexity of AXI bus drivers or HDMI physical-layer modeling.

Module: sprite_catalog_tb.sv
Inputs: None (self-contained testbench)
Outputs: None (drives DUT signals internally; uses assertions/finishes simulation)

Description: This module is a comprehensive testbench used to validate and debug the sprite rendering pipeline at a lower level than the full AXI/HDMI top-level. It instantiates the key datapath blocks used in the design—multi_sprite_reader, sprite_writer, fb_manager, vga_controller, fb_bg_merger, and bg_manager—and drives them with a locally generated 100 MHz clock (clk) plus a 25 MHz pixel clock (clk_25MHz) for VGA timing. The testbench exercises two main behaviors: (1) verification of the background scrolling math and (2) end-to-end sprite drawing into the framebuffer followed by readout/compositing during active video.

For the scrolling logic, the testbench directly calls the compute_scroller() function inside fb_bg_merger using hierarchical access (DUT4.compute_scroller) and checks the returned next address/direction with assert statements. This validates wraparound behavior near the end of the background memory range (e.g., near address 76480 for a 320×240 image) and verifies correct direction flipping when the scroll would exceed bounds.

For sprite drawing, the testbench drives sprite_nb, base_addr, and rd_en to trigger sprite reads through multi_sprite_reader (which selects between ship/missile sprite readers internally) and streams pixel data into sprite_writer. The sprite_writer converts that stream into framebuffer writes (waddr, wdata, wren) targeting the dual-framebuffer memory in fb_manager. The testbench waits for writing activity to finish (wait(~wren)), then toggles the framebuffer select (sel) so the merger reads from the updated buffer. Finally, it releases the VGA reset (vga_reset = 0) and waits for vde to become active, confirming that the pipeline is producing display-time reads and that compositing/zeroing logic can be observed at specific scan coordinates (e.g., waiting for drawY == 2 && drawX == 645 as a timing "hook").

Additionally, the testbench exposes several internal debug signals via hierarchical references (e.g., selected ROM address, read latency, "is read" state, and low-level framebuffer write ports). This makes it easier to trace and verify internal state sequencing across the reader/writer/framebuffer/merger stages.

Purpose: To provide a focused verification environment for the sprite system and framebuffer compositing logic, including deterministic tests for scroller wraparound and practical end-to-end tests that confirm sprites are correctly read from ROM, written into the framebuffer, and then read/composited during VGA active video. This testbench is primarily a debug/validation tool for internal pipeline correctness before integrating with full AXI control and real HDMI output.

Module: hdmi_text_controller_tb.sv
Inputs: None (self-contained testbench)
Outputs: None (drives DUT signals internally; optional BMP output is disabled/commented)

Description: This module is a simplified SystemVerilog testbench for validating the AXI4-Lite control interface of the HDMI text controller IP. It generates the AXI clock (aclk) and active-low reset (arstn), instantiates the DUT (hdmi_text_controller_v1_0), and drives AXI write/read transactions using the axi_write() and axi_read() tasks. These tasks perform the standard AXI4-Lite handshakes by asserting valid signals, waiting for ready from the slave, and then completing the response/data phase once BVALID (writes) or RVALID (reads) is asserted.

Unlike the earlier video-capture version, the BMP writer and pixel logging logic are commented out, and SIM_VIDEO is also commented out, so the testbench focuses primarily on bus-level verification. The test sequence applies reset, waits a short time, writes a 32-bit command word to register index 4 (the control/command register), then reads back register 4 and register 5 (the status register) to confirm correct AXI access and observe the hardware status response. The testbench also includes hierarchical taps for internal VGA/pixel signals (RGB, pixel clock, syncs, drawX/drawY) for debugging convenience, even though they are not used to generate an image in this version.

Purpose: To validate that the HDMI text controller's AXI4-Lite interface is functioning correctly by exercising basic register writes and reads (especially the command/control and status registers). This provides a quick simulation path to confirm register decoding, strobes, and readback behavior without the overhead of full-frame video logging and BMP generation.

Module Descriptions - C++:

Platform Initialization (platform.c, platform.h, platform_config.h)

The platform files provide the standard MicroBlaze runtime initialization used throughout the project. The init_platform() function enables instruction and data caches (when configured in hardware) and initializes the UART so that standard output can be used for debugging and logging. cleanup_platform() disables caches at program termination. These files are largely boilerplate Xilinx BSP code, but they are essential for ensuring the processor runs reliably and that printf-style debugging works correctly during development. They do not contain any application logic specific to the video card itself.

Hardware Register Definitions (RegistersDef.h)

RegistersDef.h defines the memory-mapped register layout used to communicate with the HDMI video card IP over AXI4-Lite. This file assigns symbolic names to specific register offsets, such as the command register, status register, and any switch/configuration registers exposed by the hardware. By centralizing these definitions, the software can issue sprite draw commands, poll busy/status flags, and configure optional features (such as scrolling behavior) without hardcoding addresses throughout the codebase. This file forms the contract between software and hardware.

C/C++ Interface Bridge (CallCpp_c.h, CallCpp.hpp)

These files provide a clean interface between C-based application code and higher-level C++ logic. Because the MicroBlaze toolchain and BSP expect a C-style entry point, CallCpp_c.h exposes extern "C" function prototypes that allow C code to safely call into C++ implementations. CallCpp.hpp contains the corresponding C++ declarations. This bridge allows the project to use object-oriented abstractions for game entities (sprites, enemies, missiles) while still integrating cleanly with the MicroBlaze startup and BSP infrastructure.

Sprite Metadata and Records (SpriteInfo.hpp/.cpp, SpriteInfo_c.h, SpriteRecord.hpp)

These files define the software representation of sprites used by the video system. SpriteInfo encapsulates information such as sprite IDs, dimensions, and any associated metadata needed to issue correct draw commands to the hardware. SpriteRecord and related headers act as lightweight data containers for tracking active sprites, their framebuffer base addresses, and their current state in the game. The C header (SpriteInfo_c.h) provides a C-compatible view of the same structures so they can be safely shared across the C/C++ boundary. Together, these files ensure that sprite-related information is consistently packaged before being sent to hardware.

Sprite and Entity Containers (SpriteContainer.hpp, EnemiesContainer.hpp, MissilesContainer.hpp)

These container classes manage collections of game entities at a higher level. Rather than directly issuing hardware commands for individual pixels, the software organizes sprites into logical groups such as enemies and missiles. Each container is responsible for updating positions, handling creation and removal of entities, and determining when a sprite should be drawn or updated on screen. When rendering is required, these containers generate the appropriate sprite number and framebuffer base address and pass them down to the hardware through the AXI command interface. This design cleanly separates game logic from rendering implementation.

Game State Tracking (GameRecord.hpp)

GameRecord encapsulates global game state, such as score, progression, or other persistent variables that affect gameplay. While this module does not interact directly with the video hardware, it influences which sprites are drawn and how often rendering commands are issued. By keeping game state management separate from rendering logic, the software remains modular and easier to extend.

USB / Input Handling (lw_usb_main.c)

This file handles USB input using the MAX3421E USB host controller provided in the earlier lab infrastructure. It processes keyboard or controller input and translates it into game actions, such as player movement or firing missiles. These inputs ultimately influence sprite positions and trigger new draw commands to the HDMI video card IP. Importantly, input handling remains entirely in software; the hardware is only responsible for displaying the resulting sprites.

Testbench Waveforms

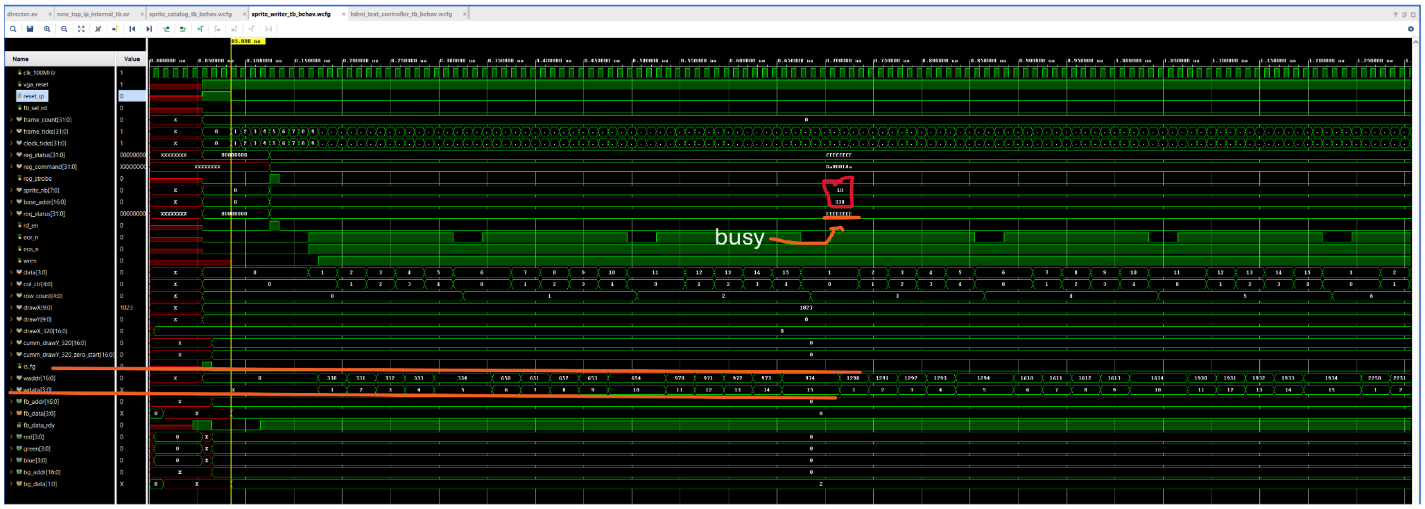


Figure 7:Sprite Writing Testbench

Here we can see that the writing logic is marked as busy with the “FFFFFFFF” register and we see the correct write address and data being written. This testbench was used by me to debug the writing logic as I successfully read the contents out of the ROM and write it into the frame buffer.

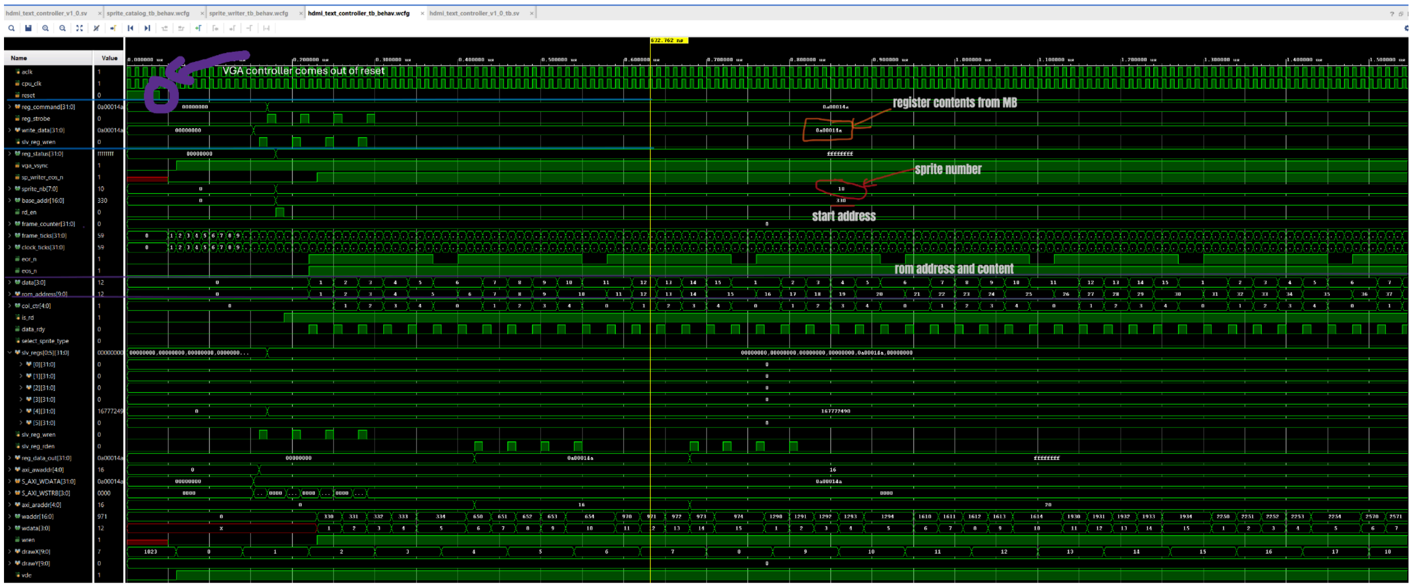


Figure 8: Full Testbench

This testbench is the full testbench I used to assemble the entirety of the hardware design. In the top left, you can see the director receives the instruction from the dummy instruction that says to draw sprite #10 and start address 330, meaning the 10th column of the 2nd row. I can see the write data at the bottom, in this case I am drawing a bullet which is why it’s small and I can easily see it draw from row to row. In blue I see the AXI strobes signaling to the hardware that the CPU has edited the contents of the relevant register as I capture the rising edge of those signals. Right above the data, we can see the “eor_n” which is the active low to signal that a row has ended being drawn.

Conclusion

In conclusion, my design functions exactly as described, simulating multiple sprites being drawn and projectiles with a scrolling background and dynamically clears the double buffers as the contents are read out. I make full use of the Microblaze SOC to handle all computational overhead and efficiently use the software aspect of the project. I learned a lot and ran into many issues, like the aforementioned error where Vivado crashed and deleted my XML file, running out of memory from the 640*480 display errors where variable sizes didn’t match so signals became high impedance. In terms of features I didn’t focus on, I had the Hex Displays fully working although I didn’t make them display anything except the date, to highlight the fact I could use them. Additionally, despite the fact that I do poll the keyboard I don’t actively use WASD keys for anything although making WASD for movement and spacebar for shooting would require about 5 lines of C++, I wanted to prioritize the switches to show the hardware configurability via the switches. Finally, you will find the collision detection logic implemented but not utilized for anything, as I planned to make explosions of sprites, but making the sprites explode on every single projectile would get very messy, since I draw so many projectiles at once in the demo. I also have ideas for how to take this project further, as I could easily allow sprite layering with only a few lines, meaning several sprites can be drawn one on top of another, since right now I always draw the background if the sprite pixel is black, but instead I could just draw the sprite underneath instead of the background. Additionally, I can expose the B bus of the BRAM directly to the Microblaze allowing a Direct Memory Access (DMA). I would just need to get to the right padding in C++ so that I can get to the Frame Buffers, similar to the padding needed in Lab 7, and it could be exposed as a uint8_t because I would use 4 bits per pixel and expose the 76800 pixels (320*240).