Design of HDMI display module for RISC-V

ICB bus of Hummingbird E203

SUMMARY

PROJECT FOLDER :
https://liveunibo-my.sharepoint.com/:f:/g/personal/tommaso_magelli_studio_unibo_it/EtmgQNGPrdxChEI7Vfyt934BIKQawiVNHMzT2kQFUveNNA?email=jiang.lei%40tongji.edu.cn&e=eMbtGZ

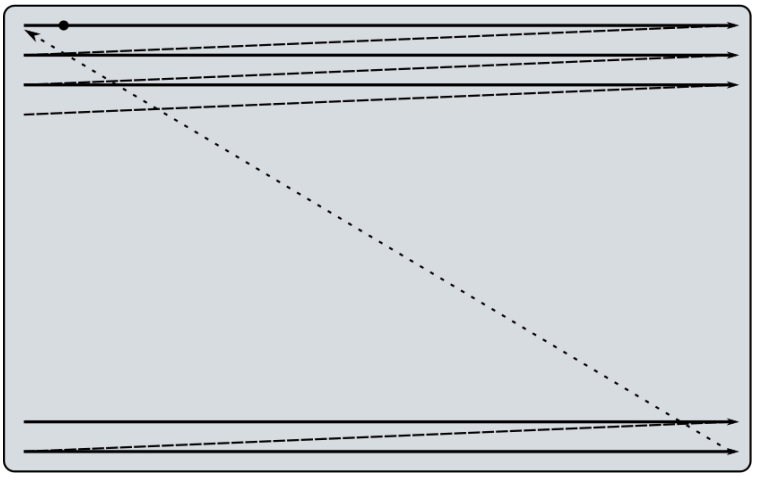
INTRODUCTION

Timing Signals

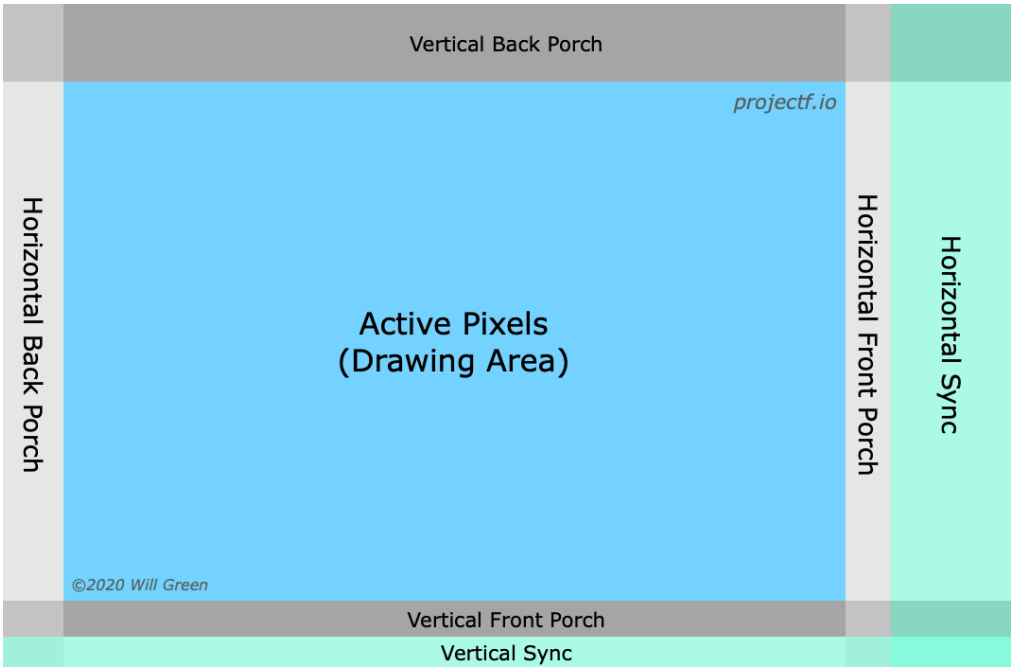
Before diving into the code for our HDMI driver project, let us introduce video timing signals, hsync, vsync, and de, how they work, and why they are present.

Video timing signals are crucial for synchronizing the transmission of video data from the source to the display. These signals ensure that each pixel is drawn at the correct position and time, resulting in a stable and coherent image on the screen. Horizontal synchronization (hsync) and vertical synchronization (vsync) signals are fundamental components of this timing system. Hsync signals mark the end of drawing one line of pixels and the start of the next, while vsync signals indicate the end of drawing one frame and the beginning of the next. These synchronization pulses help align the pixel data with the display's scanning process, maintaining proper timing and preventing image distortion.

In VGA (Video Graphics Array) systems, the display timing involves several periods: the front porch, sync pulse, and back porch, both horizontally and vertically. The front porch is a brief period after the end of the visible line or frame but before the sync pulse begins. The sync pulse itself is a signal indicating the end of the current line or frame. Following the sync pulse is the back porch, which allows time for the electron beam in CRT displays (or the equivalent in modern displays) to settle before the active video period starts, where actual pixel data is transmitted.

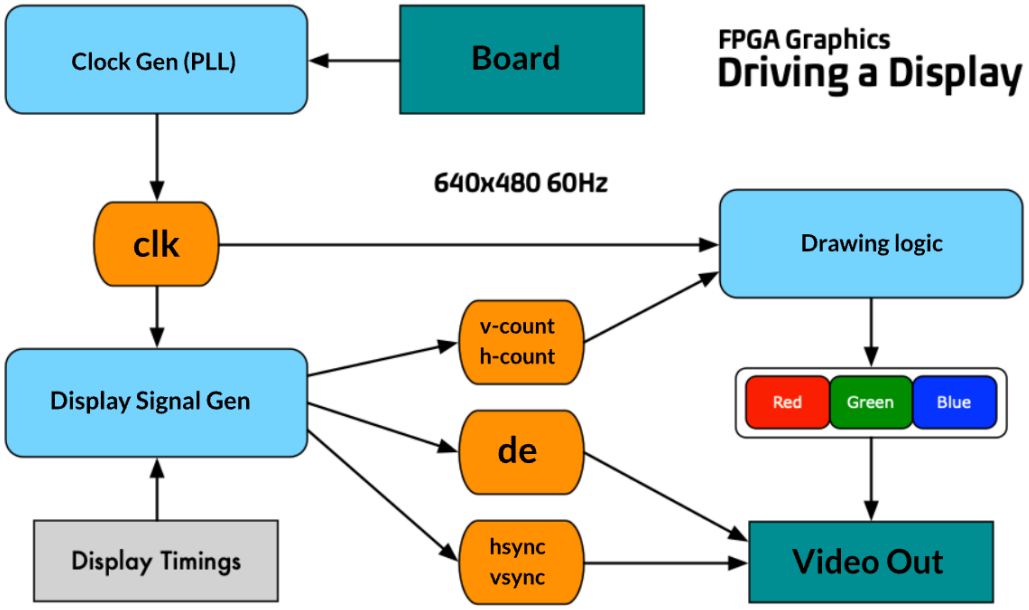


The data enable (de) signal is used to indicate the active video period when pixel data is being transmitted. It helps differentiate between the active display area and the blanking intervals (the front porch, sync pulse, and back porch).

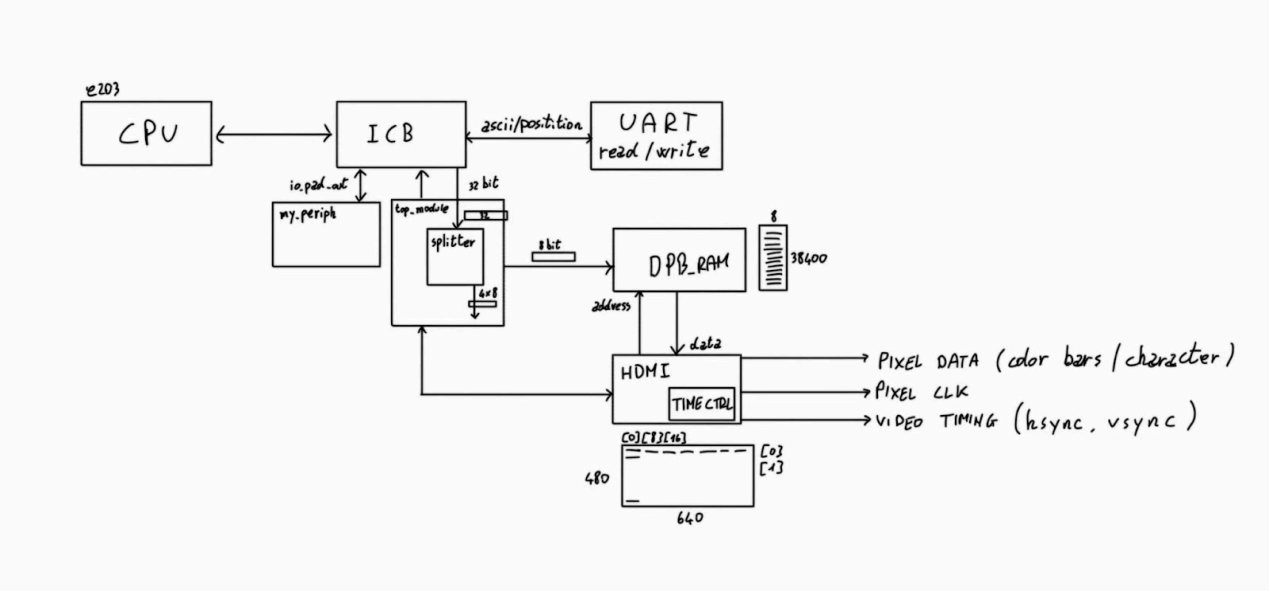


When comparing VGA and HDMI (High-Definition Multimedia Interface), the primary difference lies in their nature: VGA is an analog standard, while HDMI is digital. HDMI also carries audio and additional

control signals. However, since our focus is solely on video output, the basic principles of timing signals, such as hsync, vsync, and de, remain largely the same. In HDMI, these signals ensure that the digital data aligns correctly, similar to the way VGA signals manage the analog data alignment.



The block diagram above depicts the overall system of generating and managing these signals within an FPGA graphics setup. Understanding these timing signals is crucial for designing and implementing effective HDMI drivers, ensuring smooth and accurate video output.
By combining the explanations with the images, we can better visualize how video timing signals work and the slight differences between VGA and HDMI, particularly in their application to video output.



The design of our project shows connections between cpu, uart and modules, to communicate between each other they use ICB connections.

The core of the system is the e203 RISC-V processor. Data from external devices, such as a keyboard, is received by the CPU through the UART in ASCII format. The e203's main program then converts this data into bitmaps, which are transmitted to the relevant module through `io_pad_out`.

The data subsequently reaches the project's top-level module, which acts as a hub connecting all other modules. Here, the incoming 32-bit data is divided into four bytes, each assigned an address, and then written into the DPBram.

The HDMI module plays a crucial role in controlling screen timing (hsync, vsync, and data enable signals) and outputting data from the DPBram. It generates read addresses for the RAM and, upon receiving the 8-bit data, renders it at the appropriate screen location. The HDMI module also interfaces with a color bar generator module, which displays color bars for the initial five seconds before transitioning to the UART-received characters.

CORE

In our experiment, we aimed to create an implementation to be mounted on the ICB bus of the Hummingbird E203 architecture. We were provided with the core modules of the Hummingbird E203 and had to modify several of them to drive outputs and integrate our implementation on the ICB bus. The modules we modified, in descending order of hierarchy, were:

1. **E203_SOC_DEMO**
2. **E203_SOC_TOP**
3. **E203_SUBSYS_TOP**
4. **E203_SUBSYS_MAIN**
5. **E203_SUBSYS_PERIPS**

These modules act as the middleman between software and hardware and are the heart of the Hummingbird E203 architecture.

Here’s a detailed explanation of each module and why modifications were necessary:

E203_SOC_DEMO

***Purpose*:** This module serves as the top-level system-on-chip (SoC) design for demonstration purposes. It integrates all subsystems and provides a complete working example of the Hummingbird E203 in action.

***Modification Reason*:** We needed to ensure that our signals and data from the peripheral could propagate through the entire SoC. By modifying this module, we allowed our HDMI signals (V_sync, H_sync, and hdmi_data_out) to be accessible at the highest level, facilitating system-wide communication and control.

Code Insertion:

•Outputs:

```
module e203_soc_demo(
output              V_sync,
output              H_sync,
output      [23:0]    hdmi_data_out,
```

•Instantiation:

```
e203_soc_top u_e203_soc_top (
  .V_sync(V_sync),
  .H_sync(H_sync),
  .hdmi_data_out(hdmi_data_out),
  ...
);
```

E203_SOC_TOP

Purpose: This module is the top-level integrator for the subsystems, including the main processing unit, memory interfaces, and peripheral subsystems.

Modification Reason: To propagate the signals upward from E203_SUBSYS_TOP to E203_SOC_DEMO, ensuring that our implementation can interact with other high-level components of the SoC.

Code Insertion:

•Outputs:

```
module e203_soc_top(
output              V_sync,
output              H_sync,
output      [23:0]    hdmi_data_out,
```

•Instantiation:

```
e203_subsys_top u_e203_subsys_top (
  .V_sync(V_sync),
  .H_sync(H_sync),
  .hdmi_data_out(hdmi_data_out),
  ...
);
```

E203_SUBSYS_TOP

Purpose: This module acts as the top-level for the subsystem integration, managing communication between the main subsystem and peripheral subsystems.

Modification Reason: To allow the signals and data from our peripheral implementation to flow from E203_SUBSYS_MAIN to E203_SOC_TOP.

Code Insertion:

•Outputs:

```
module e203_subsys_top(
   output              V_sync,
   output              H_sync,
   output      [23:0]    hdmi_data_out,
```

•Instantiation:

```
e203_subsys_main u_e203_subsys_main (
   .V_sync(V_sync),
   .H_sync(H_sync),
   .hdmi_data_out(hdmi_data_out),
   ...
);
```

E203_SUBSYS_MAIN

Purpose: This module coordinates the main subsystem, including the core processor and memory controllers, and interfaces with peripheral subsystems.

Modification Reason: To facilitate communication between the E203_SUBSYS_PERIPS (where our custom peripheral is implemented) and the top-level subsystems.

Code Insertion:

•Outputs:

```
module e203_subsys_main(
   output              V_sync,
   output              H_sync,
   output      [23:0]    hdmi_data_out,
```

•Instantiation:

```
e203_subsys_perips u_e203_subsys_perips (
   .V_sync(V_sync),
   .H_sync(H_sync),
   .hdmi_data_out(hdmi_data_out),
   ...
);
```

E203_SUBSYS_PERIPS

Purpose: This module integrates various peripheral subsystems and interfaces them with the main subsystem. This module is where all the peripherals of the SOC are connected. It’s the bridge between peripherals (UART, GPIO,etc) and the SOC.

Modification Reason: To directly incorporate our custom peripheral module (my_periph_example) and handle the ICB bus communication, as well as driving the HDMI outputs. Additionally, we connected our implementation top module, which manages all the video logics.

Code Insertion:

•Outputs:

```
module e203_subsys_perips(
    output              V_sync,
    output              H_sync,
    output      [23:0]    hdmi_data_out,
```

•Instantiation:

```
// my peripheral example
my_periph_example u_my_periph_top(
    .clk            (clk),
    .rst_n          (rst_n),
    .i_icb_cmd_valid (my_periph_icb_cmd_valid),
    .i_icb_cmd_ready (my_periph_icb_cmd_ready),
    .i_icb_cmd_addr  (my_periph_icb_cmd_addr ),
    .i_icb_cmd_read  (my_periph_icb_cmd_read ),
    .i_icb_cmd_wdata (my_periph_icb_cmd_wdata),
    .i_icb_rsp_valid (my_periph_icb_rsp_valid),
    .i_icb_rsp_ready (my_periph_icb_rsp_ready),
    .i_icb_rsp_rdata (my_periph_icb_rsp_rdata),
    .io_interrupts_0_0 (my_irq),
    .io_pad_out       (my_io_pad_out)
);
wire [31:0] my_io_pad_out;
// top instantiation
top_module my_top(
    .clk  (clk),
    .rst_n  (rst_n),
    .io_pad_out (my_io_pad_out),
    .V_sync (V_sync),
    .H_sync (H_sync),
    .hdmi_data_out  (hdmi_data_out)
);
```

By systematically modifying these modules, we ensured that our custom peripheral could communicate effectively within the E203 architecture, driving the necessary outputs and interfacing correctly with the ICB bus.

Further in the report we will analyze the connection between the CORE, the IP, the FIRMWARE, and lastly the SIM.

IP

MY PERIPH

```
module my_periph_example (
    input                   clk,
    input                   rst_n,

    input                   i_icb_cmd_valid,
    output                  i_icb_cmd_ready,
    input  [32-1:0]         i_icb_cmd_addr,
    input                   i_icb_cmd_read,
    input  [32-1:0]         i_icb_cmd_wdata,

    output                  i_icb_rsp_valid,
    input                   i_icb_rsp_ready,
    output [32-1:0]         i_icb_rsp_rdata,

    output                  io_interrupts_0_0,
    output [32-1:0]         io_pad_out
);

    //define a 32-bit register for operating your module
    reg [31:0] io_value_reg;

    reg [31:0] icb_data_out;
    reg        icb_rsp_valid;

    wire reset;
    wire clock;
    //read enable signal for register reading, this signal assert when proper address issued.
    wire io_value_reg_rd_en;

    //write enable signal for register writing, this signal assert when proper address issued.
    wire io_value_reg_wr_en;
```

This module is designed to interface with the ICB (Inter-Chip Bus) protocol. This module facilitates communication and data exchange.

Inputs

- `clk`: This input provides the clock signal that synchronizes the module's operations.
- `rst_n`: This is an active-low reset signal. When this signal goes low, it resets the module to its initial state, clearing any previous data or configurations.

- `i_icb_cmd_valid`: This input signal indicates that a valid command is being sent to the module over the ICB bus. It acts as a flag to inform the module that it should pay attention to the incoming command data.
- `i_icb_cmd_addr[31:0]`: This input carries the 32-bit address associated with the ICB command. The address specifies the location within the module's memory or registers where the command should be applied.
- `i_icb_cmd_read`: This input signal indicates whether the current ICB command is a read operation (to retrieve data) or a write operation (to store data).
- `i_icb_cmd_wdata[31:0]`: This input carries the 32-bit data associated with the ICB command. If the command is a write operation, this data is the value to be written to the specified address.

Outputs

- `i_icb_cmd_ready`: This output signal indicates that the module is ready to accept a new command from the ICB bus. It acts as a handshake signal to ensure proper synchronization between the module and the bus.
- `i_icb_rsp_valid`: This output signal indicates that the module has a valid response ready to be sent back over the ICB bus. This could be in response to a read command or an acknowledgment of a write command.
- `i_icb_rsp_rdata[31:0]`: This output carries the 32-bit data associated with the module's response. If the response is to a read command, this data contains the value read from the specified address.
- `io_interrupts_0_0`: This output signal is used to generate interrupts. Interrupts are signals that can be sent to a processor to request its attention, typically to handle some event or data transfer.
- `io_pad_out[31:0]`: This output connects the module's internal 32-bit register (`io_value_reg`) to external I/O pads. This allows the module to interact with other devices or components outside of the chip.

Internal Registers and Wires

- `io_value_reg[31:0]`: This is a 32-bit register used for storing data within the module. It can be read from or written to depending on the ICB commands received.
- `icb_data_out[31:0]`: This register is used to temporarily hold the data that will be sent as a response over the ICB bus.
- `icb_rsp_valid`: This register is a flag that indicates whether the data in `icb_data_out` is a valid response.
- `reset`: This wire is derived from the `rst_n` input and represents the reset state of the module.
- `clock`: This wire is derived from the `clk` input and represents the clock signal used for synchronization.
- `io_value_reg_rd_en`: This wire is a read enable signal. It is asserted (set to 1) when a valid read command is received for the `io_value_reg` register.

- `io_value_reg_wr_en`: This wire is a write enable signal. It is asserted (set to 1) when a valid write command is received for the `io_value_reg` register.

```
assign reset = ~rst_n;
   assign clock = clk;

   //judge if register is selected for read, 3'h4 is the offset address of the register
   assign io_value_reg_rd_en = i_icb_cmd_valid && i_icb_cmd_read && (i_icb_cmd_addr[11:0] ==
3'h4);
   //for write
   assign io_value_reg_wr_en = i_icb_cmd_valid && (~i_icb_cmd_read) && (i_icb_cmd_addr[11:0]
== 3'h4);

   //no wait state, so direct connect valid to ready signal
   assign i_icb_cmd_ready = i_icb_cmd_valid;

   assign i_icb_rsp_valid = i_icb_rsp_ready && icb_rsp_valid;

   assign i_icb_rsp_rdata = icb_data_out;

   //connect io pad to register
   assign io_pad_out = io_value_reg; //32

   always @(posedge clock or posedge reset) begin
       if (reset) begin
           io_value_reg <= 32'd0;
           icb_rsp_valid <= 1'b0;
       end
       else begin
           if (io_value_reg_rd_en) begin
               icb_data_out <= io_value_reg;
               icb_rsp_valid <= 1'b1;
           end
           else begin
               icb_rsp_valid <= 1'b0;
           end

           if(io_value_reg_wr_en) begin
               io_value_reg <= i_icb_cmd_wdata;
               icb_rsp_valid <= 1'b1;
           end
       end
   end

endmodule
```

1. **Signal Assignments:**
   - `reset`: This wire is assigned the inverted value of the `rst_n` input.
   - `clock`: This wire is directly assigned the value of the `clk` input.
2. **ICB Response:**
   - `i_icb_cmd_ready`: This signal is directly assigned the value of `i_icb_cmd_valid`. This implies that the module is always ready to accept a new command as soon as a valid command is presented on the bus
   - `i_icb_rsp_valid`: This signal is high when both `i_icb_rsp_ready` and `icb_rsp_valid` are high.
   - `i_icb_rsp_rdata`: This signal is assigned the value of `icb_data_out`, which holds the data to be sent as a response.
3. **I/O Pad Connection:**
   - `io_pad_out`: This signal is directly connected to the `io_value_reg` register, allowing the register's contents to be accessible on the external I/O pins.

Sequential Logic (always block)

The `always` block describes the sequential behavior of the module, triggered by the rising edge of the clock (`posedge clock`) or a reset event (`posedge reset`).

- **Reset Condition:** When `reset` is high, the `io_value_reg` is initialized to 0, and the `icb_rsp_valid` flag is cleared.
- **Read Operation:** If `io_value_reg_rd_en` is high, the contents of `io_value_reg` are transferred to `icb_data_out` for transmission as a response, and `icb_rsp_valid` is set to indicate a valid response. Otherwise, `icb_rsp_valid` is cleared.
- **Write Operation:** If `io_value_reg_wr_en` is high, the data from `i_icb_cmd_wdata` is written into `io_value_reg`, and `icb_rsp_valid` is set to acknowledge the write.

TOP MODULE

```
module top_module (
    input                   clk,
    input                   rst_n,
    input       [31:0]      io_pad_out,

    output                  V_sync,
    output                  H_sync,
    output      [23:0]      hdmi_data_out
);



//define a 32-bit register for operating your module
reg     [31:0]  io_value_reg;

// Test DPB RAM
```

```
wire    [15:0]  r_add_ram;
wire    [7:0]   data_out_dpb;
wire    [15:0]  w_add_ram;
wire    [7:0]   data_in_ram;
wire            w_enable;
```

`top_module` orchestrates the interaction between data input source, a register splitter, a DPBRAM, and an HDMI module to generate video output.

```
// ---------------------------------------------------------------------------
// Instance for register splitter
// ---------------------------------------------------------------------------

register_splitter my_reg_splitter (
    .clk(clk),
    .data_in_32(io_pad_out),
    .data_out_8(data_in_ram),
    .address(w_add_ram),
    .write_enable(w_enable)
);
```

1. register_splitter (my_reg_splitter):

- **Purpose:** This module splits the incoming 32-bit data (`io_pad_out`) into 8-bit bytes, preparing it for storage in the DPBRAM.
- **Connections:**
  - `clk`: System clock for synchronization.
  - `data_in_32`: 32-bit input data from `io_pad_out`.
  - `data_out_8`: 8-bit output data connected to `data_in_ram` (input of DPBRAM).
  - `address`: Write address for the DPBRAM, connected to `w_add_ram`.
  - `write_enable`: Signal controlling when data is written to the DPBRAM, connected to `w_enable`.

```
// ---------------------------------------------------------------------------
// Instance for dbp ram
// ---------------------------------------------------------------------------

Gowin_DPB your_instance_name(
    .douta(),               //output [7:0] douta
    .doutb(data_out_dpb),   //output [7:0] doutb
    .clka(clk),             //input clka
    .ocea(1'b1),                //input ocea
    .cea(1'b1),         //input cea
    .reseta(1'b0),          //input reseta
    .wrea(1'b1),            //input wrea
    .clkb(clk),             //input clkb
```

```
    .oceb(1'b1),                //input oceb
    .ceb(1'b1),              //input ceb
    .resetb(1'b0),          //input resetb
    .wreb(1'b0),             //input wreb
    .ada(w_add_ram),         //input [15:0] ada
    .dina(data_in_ram),      //input [7:0] dina
    .adb(r_add_ram),         //input [15:0] adb
    .dinb()                  //input [7:0] dinb
    );
```

2. Gowin_DPB (your_instance_name):

- **Purpose:** This module represents the Double Data Rate Block RAM (DPBRAM) used for storing video data.
- **Connections:**
  - `douta`, `dinb`: Unused in this code.
  - `doutb`: Output data from the DPBRAM, connected to `data_out_dpb` (input of the HDMI module).
  - `clka`, `clkb`: Both connected to the system clock (`clk`).
  - `ocea`, `cea`, `oceb`, `ceb`: Output and chip enable signals set to 1, indicating the DPBRAM is always enabled.
  - `reseta`, `resetb`: Reset signals set to 0, meaning no reset is applied.
  - `wrea`: Write enable for port A set to 1, allowing writes.
  - `wreb`: Write enable for port B set to 0, disabling writes.
  - `ada`: Write address for port A, connected to `w_add_ram`.
  - `adb`: Read address for port B, connected to `r_add_ram`.
  - `dina`: Input data to be written, connected to `data_in_ram`.

```
// ------------------------------------------------------------------------
// Instance for hdmi module
// ------------------------------------------------------------------------

hdmi my_hdmi(
    .clk(clk),
    .rst_n(rst_n),
    .ram_data(data_out_dpb),
    .address_read_dbp(r_add_ram),
    .hsync(H_sync),
    .vsync(V_sync),
    .de(de_temp),
    .hdmi_data(hdmi_data_out)
);
endmodule
```

3. hdmi (my_hdmi):

- **Purpose:** This module generates the video timing signals (H_sync, V_sync), data enable signal (de), and the final HDMI output data (hdmi_data_out).
- **Connections:**
  - `clk`: System clock.
  - `rst_n`: Reset signal.
  - `ram_data`: Pixel data from the DPBRAM (`data_out_dpb`).
  - `address_read_dbp`: Read address for the DPBRAM.
  - `hsync`, `vsync`: Connected to the top module's outputs.
  - `de`: Connected to an internal signal `de_temp`.
  - `hdmi_data`: 24-bit RGB pixel data, connected to `hdmi_data_out`.

Overall Data Flow:

1. **Input:** 32-bit data enters through `io_pad_out`.
2. **Splitting:** `register_splitter` divides the data into 8-bit bytes.
3. **DPBRAM Storage:** The bytes are written into the DPBRAM at addresses specified by `w_add_ram`.
4. **HDMI Processing:** The `hdmi` module reads pixel data from the DPBRAM using `address_read_dbp`, generates timing signals, and outputs the final 24-bit RGB data (`hdmi_data_out`) for display.

SPLITTER

```
module register_splitter (
    input wire clk,
    input wire [31:0] data_in_32,
    output reg [7:0] data_out_8,
    output reg [15:0] address,
    output reg write_enable
);

    reg [7:0] reg_array [0:3];
    reg [15:0] addr_counter = 0;
    reg data_valid = 1'b0;
    reg [31:0] prev_data_in_32 = 32'd0;     // Register to hold previous data_in_32
    reg [1:0] byte_index = 2'b00;           // To track which byte we are writing
```

This section of the code defines several registers and variables for the module's operation:

- `[7:0] reg_array [0:3]`: This is an array of four 8-bit registers, each designed to hold one byte of the split 32-bit data.

- `addr_counter = 0`: This is a 16-bit register initialized to 0. It serves as an address counter, keeping track of the memory address where the next byte should be written in the RAM.
- `data_valid = 1'b0`: This is a single-bit register (a flag) initialized to 0. It indicates whether there is valid data ready to be output from the `reg_array` to the RAM. When new 32-bit data is received and split, this flag is set to 1. It is reset to 0 after all four bytes have been output.
- `prev_data_in_32 = 32'd0`: This 32-bit register stores the previous value of the `data_in_32` input. It is used to detect when new data has arrived from the CPU. The module begin a new data splitting process only if the current `data_in_32` value is different from the previous value stored in `prev_data_in_32`.
- `byte_index = 2'b00`: This is a 2-bit register initialized to 0. It acts as an index to track which byte within the `reg_array` is currently being output to the RAM. Since there are four bytes, the possible values for `byte_index` are 00, 01, 10, and 11.

```
// Always block for data splitting and flagging
   always @(posedge clk) begin
       if (data_in_32 != prev_data_in_32) begin  // Detect new data
           reg_array[3] <= {data_in_32[0],  data_in_32[1],  data_in_32[2],  data_in_32[3],
  data_in_32[4],  data_in_32[5],  data_in_32[6],  data_in_32[7]};
           reg_array[2] <= {data_in_32[8],  data_in_32[9],  data_in_32[10], data_in_32[11],
  data_in_32[12], data_in_32[13], data_in_32[14], data_in_32[15]};
           reg_array[1] <= {data_in_32[16], data_in_32[17], data_in_32[18], data_in_32[19],
  data_in_32[20], data_in_32[21], data_in_32[22], data_in_32[23]};
           reg_array[0] <= {data_in_32[24], data_in_32[25], data_in_32[26], data_in_32[27],
  data_in_32[28], data_in_32[29], data_in_32[30], data_in_32[31]};
           data_valid <= 1'b1;                    // Mark data as valid
           prev_data_in_32 <= data_in_32;         // Update previous data
           byte_index <= 2'b00;                   // Reset byte index for new data
       end
   end
```

This section of the `register_splitter` module focuses on two critical tasks:

1. **Data Splitting:** When the CPU sends a 32-bit data word, the logic first identifies if the data is new. If it is, the module split this 32-bit word into its four constituent 8-bit bytes.
2. **Data Synchronization and Preparation:** In anticipation of sending these bytes to the RAM, the code reverses their order. A "data valid" flag is then raised, signaling that the split and correctly ordered data is ready for transfer to the RAM.

Detailed Explanation:

- **New Data Detection (`data_in_32 != prev_data_in_32`)**: The module checks if the incoming data (`data_in_32`) is different from the previously stored data

(`prev_data_in_32`). If there's a change, it means the CPU has sent new data that needs to be processed.

- **Data Splitting and Reversal (`reg_array[3] <= ...`)**: The 32-bit data is divided into four 8-bit segments. Each segment is assigned to a specific register in the `reg_array`. The order of assignment is reversed to adjust data before printing them.
- **Data Validity Signaling (`data_valid <= 1'b1`)**: A flag called `data_valid` is set to 1. This acts as a signal, informing other parts of the circuit that the split and ordered data is now ready to be written into the RAM.
- **New cycle (`prev_data_in_32 <= data_in_32; byte_index <= 2'b00`)**: The `prev_data_in_32` register is updated with the current data, preparing it for the next comparison. The `byte_index`, which keeps track of which byte is being sent, is reset to 0 to start from the beginning for the next set of data.

```
// Always block for output and address increment
   always @(posedge clk) begin
       if (data_valid) begin
           data_out_8 <= reg_array[byte_index];
           address <= addr_counter;
           write_enable <= 1'b1;             // Enable write for each byte

           byte_index <= byte_index + 2'b01; // Increment byte index and address counter
           addr_counter <= addr_counter + 1;

           if (byte_index == 2'b11) begin    // Reset data_valid after processing 4 bytes
               data_valid <= 1'b0;
           end
       end else begin
           write_enable <= 1'b0;             // Disable write if no new data
       end
   end
endmodule
```

This part of the module is responsible for two key functions:

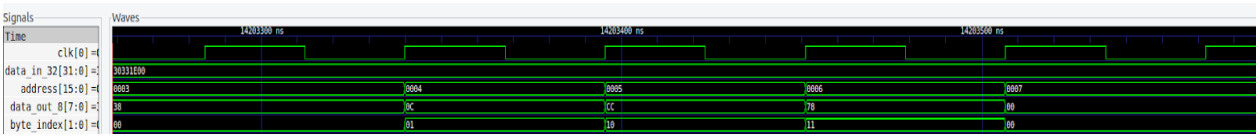
1. **Sequential Byte Output:** It ensures that the individual bytes extracted from the 32-bit word are sent to the RAM in the correct order and at the right times.
2. **Write Enable Control:** It generates a signal to tell the RAM precisely when to write each byte into its memory.

Detailed Explanation:

- **Conditional Output (`if (data_valid)`)**: The code first checks the `data_valid` flag. If it's set to 1, it means there's valid data ready to be sent to the RAM.
- **Outputting Data and Address (`data_out_8 <= reg_array[byte_index]; address <= addr_counter`)**: The current byte, determined by the `byte_index`, is

placed on the `data_out_8` line for the RAM to receive. Simultaneously, the `address` signal provides the RAM with the correct memory location to store this byte.

- **Write Enable (`write_enable <= 1'b1`)**: The `write_enable` signal is set to 1. This is a direct command to the RAM, instructing it to write the data present on the `data_out_8` line into the memory location specified by the `address` signal.
- **Incrementing Counters (`byte_index <= byte_index + 2'b01; addr_counter <= addr_counter + 1`)**: After each byte is sent, the `byte_index` is incremented to point to the next byte in the `reg_array`. The `addr_counter` is also incremented to point to the next memory address in the RAM.
- **Resetting After Four Bytes (`if (byte_index == 2'b11)`)**: When all four bytes have been sent (indicated by `byte_index` reaching '11'), the `data_valid` flag is reset to 0. This stops further data output until the next 32-bit word arrives from the CPU.
- **Disabling Write When Idle (`else write_enable <= 1'b0`)**: If there's no valid data (`data_valid` is 0), the `write_enable` signal is kept low (0). This prevents the RAM from writing any incorrect or outdated data.



Signals in the Waveform:

- **CLK[0]:** This is the main clock signal that drives the entire system. In the code, it's represented by the `clk` input in each module. The rising edges of this clock trigger the sequential logic in the design.
- **data_in[31:0]:** This is the 32-bit input data bus. In the code, it corresponds to the `io_pad_out` signal in the `top_module`. The `register_splitter` module takes this input and splits it into four 8-bit bytes.
- **address[15:0]:** This is the 16-bit address bus used to address the DPBRAM.
- **data_out[7:0]:** This is the 8-bit data output bus. In the code, it corresponds to `data_out_dpb`, which carries the pixel data read from the DPBRAM to the `hdmi` module.
- **byte_index[1:0]:** This 2-bit signal indicates which byte of the 32-bit input data is being processed. In the code, it's represented by the `byte_index` register within the `register_splitter` module.

Code Functionality Reflected in the Waveform:

1. **Data Splitting:** The waveform shows the `data_in[31:0]` signal changing, followed by transitions on the `byte_index[1:0]` signal as each byte is processed.
2. **Address Generation:** The waveform shows the `r_add_ram` signal changing in a pattern that corresponds to the way the `hdmi` module reads pixel data from the DPBRAM.
3. **Data Output and Timing Signals:** We can see the `data_out[7:0]` signal carrying the pixel data. The `hsync`, `vsync`, and `de` signals exhibit the characteristic timing patterns of video signals, with high and low periods corresponding to the sync pulses, blanking intervals, and active video periods.

COLOR BARS MODULE

```
module color_bar_hor (
   input wire        I_pxl_clk,     // Pixel clock
   input wire        I_rst_n,       // Active-low reset
   input wire [31:0] I_h_total,     // Total horizontal pixels per line
   input wire [31:0] I_h_sync,      // Horizontal sync pulse width (pixels)
   input wire [31:0] I_h_bporch,    // Horizontal back porch (pixels)
   input wire [31:0] I_h_fporch,    // Horizontal front porch (pixels)
   input wire [31:0] I_h_res,       // Horizontal active video area (pixels)
   input wire [31:0] I_v_total,     // Total vertical lines per frame
   input wire [31:0] I_v_sync,      // Vertical sync pulse width (lines)
   input wire [31:0] I_v_bporch,    // Vertical back porch (lines)
   input wire [31:0] I_v_fporch,    // Vertical front porch (lines)
   input wire [31:0] I_v_res,       // Vertical active video area (lines)
   input wire [11:0] I_h_count,
   input wire [11:0] I_v_count,
   input wire        I_hs_pol,      // HSYNC polarity (1 for active high)
   input wire        I_vs_pol,      // VSYNC polarity (1 for active high)
   input wire        I_de,
   output     [7:0]  O_data_r,      // Red color component
   output     [7:0]  O_data_g,      // Green color component
   output     [7:0]  O_data_b       // Blue color component
);
// Color Definitions
localparam [23:0] WHITE   = 24'hFFFFFF;
localparam [23:0] YELLOW  = 24'hFFFF00;
localparam [23:0] CYAN    = 24'h00FFFF;
localparam [23:0] GREEN   = 24'h00FF00;
localparam [23:0] MAGENTA = 24'hFF00FF;
localparam [23:0] RED     = 24'hFF0000;
localparam [23:0] BLUE    = 24'h0000FF;
localparam [23:0] BLACK   = 24'h000000;
// Calculate the height of each color bar segment
wire [9:0] segment_height = I_v_res / 8;
```

Inputs:

- `module color_bar_hor (...)`: This line declares the module and its name. The module has several input ports:
  - `I_pxl_clk`: The pixel clock.
  - `I_rst_n`: An active-low reset signal. When this signal is low, the module resets its internal state.
  - `I_h_total`, `I_h_sync`, `I_h_bporch`, `I_h_fporch`, `I_h_res`: These inputs define the horizontal timing parameters of the video signal, including the total number of pixels per line, the width of the horizontal sync pulse, and the back and front porch intervals.
  - `I_v_total`, `I_v_sync`, `I_v_bporch`, `I_v_fporch`, `I_v_res`: These inputs define the vertical timing parameters of the video signal, including the total number of lines per frame, the width of the vertical sync pulse, and the back and front porch intervals.
  - `I_h_count`, `I_v_count`: These inputs provide the current horizontal and vertical pixel counts within the video frame.
  - `I_de`: The data enable signal. When this signal is high, the module outputs color data; otherwise, it outputs black.

Outputs:

- `O_data_r`, `O_data_g`, `O_data_b`: These are the output signals for the red, green, and blue color components of the video signal. Each output is 8 bits wide, allowing for 256 different intensity levels per color.

Color Definitions (localparam):

- This section defines several `localparam` constants, each representing a specific color in 24-bit RGB format.

Segment Height Calculation:

- `wire [9:0] segment_height = I_v_res / 8;`: This line calculates the height of each color bar segment. The vertical active video area (`I_v_res`) is divided by 8, as there are eight different colors in the color bar. The result is stored in the `segment_height` wire.

```
// Color Bar Pattern Generation
always @(posedge I_pxl_clk or negedge I_rst_n) begin
   if (!I_rst_n) begin
       // Initialize outputs on reset
       {O_data_r, O_data_g, O_data_b} <= BLACK;
   end else if (I_de) begin // Only update colors when data enable is active
       // Determine which color to output based on the current vertical count
       case ((I_v_count-33) / segment_height)
           0: {O_data_r, O_data_g, O_data_b} <= WHITE;
```

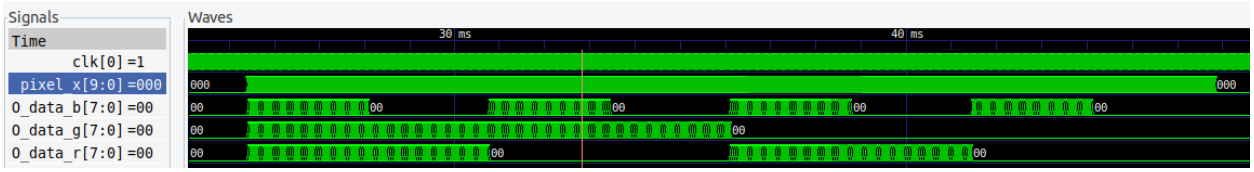
```
            1: {O_data_r, O_data_g, O_data_b} <= YELLOW;
            2: {O_data_r, O_data_g, O_data_b} <= CYAN;
            3: {O_data_r, O_data_g, O_data_b} <= GREEN;
            4: {O_data_r, O_data_g, O_data_b} <= MAGENTA;
            5: {O_data_r, O_data_g, O_data_b} <= RED;
            6: {O_data_r, O_data_g, O_data_b} <= BLUE;
            7: {O_data_r, O_data_g, O_data_b} <= BLACK;
            default: {O_data_r, O_data_g, O_data_b} <= BLACK;
        endcase
    end else begin
        {O_data_r, O_data_g, O_data_b} <= BLACK; // Black during blanking intervals
    end
end

endmodule
```

Color Bar Pattern Generation (always block):

- `always @(posedge I_pxl_clk or negedge I_rst_n) begin`: This line indicates that the code within this block will execute whenever there's a positive edge on the pixel clock (`I_pxl_clk`) or a negative edge (low-to-high transition) on the reset signal (`I_rst_n`). This ensures that the color bar generation is synchronized with the video timing.
- `if (!I_rst_n) begin ... end`: This is the reset condition. If the reset signal is active low (logic 0), the output color components (`O_data_r`, `O_data_g`, `O_data_b`) are all set to `BLACK`. This initializes the color bar to black when the module is reset.
- `else if (I_de) begin ... end`: This is the main color generation logic. It only executes when the data enable signal (`I_de`) is active high (logic 1), meaning it's time to output color data.
    - `case ((I_v_count-33) / segment_height)`: This line calculates which color bar segment the current pixel belongs to. It does this by:
        1. Subtracting 33 from the current vertical pixel count (`I_v_count`). This is likely an adjustment for some offset in the video timing.
        2. Dividing the result by `segment_height`. This determines which of the eight color segments the pixel falls into.
    - `0: {O_data_r, O_data_g, O_data_b} <= WHITE; ...`: This is a case statement that assigns the appropriate RGB color values to the output signals (`O_data_r`, `O_data_g`, `O_data_b`) based on the calculated segment. Each case corresponds to a different color segment in the color bar.
    - `default: {O_data_r, O_data_g, O_data_b} <= BLACK;`: If the calculated segment doesn't match any of the defined cases (which shouldn't happen in normal operation), the output is set to black.
- `else begin ... end`: This part handles the blanking intervals of the video signal. When the data enable signal (`I_de`) is not active, the output is set to black.

Overall Functionality:

This module generates a horizontal color bar test pattern where each bar has a height determined by `segment_height`. The colors of the bars are defined by the `localparam` constants. The pattern is generated synchronously with the pixel clock and is only active when the data enable signal is high. During blanking intervals, the output is black.



The waveform shows the three output blue, green and red changes following 8 different colors.

HDMI

```
module hdmi (
    input wire              clk,                // System clock (25.2 MHz)
    input wire              rst_n,              // Asynchronous reset (active low)

    input wire      [7:0]   ram_data,           // Response data from dpbram
    output reg      [15:0]  address_read_dbp,   // Read address for dpb ram

    output reg              hsync, vsync, de,   // Video timing signals
    output wire     [23:0]  hdmi_data           // Pixel data to display
);

// ------------------------------------------------------------------------
// Parameters for video timing and character size
// ------------------------------------------------------------------------

parameter H_ACTIVE          = 640;          // Horizontal active video area (pixels)
parameter H_FRONT_PORCH     = 16;           // Horizontal front porch (pixels)
parameter H_SYNC_PULSE      = 96;           // Horizontal sync pulse width (pixels)
parameter H_BACK_PORCH      = 48;           // Horizontal back porch (pixels)
parameter H_TOTAL           = 800;          // Total horizontal pixels per line

parameter V_ACTIVE          = 480;          // Vertical active video area (lines)
parameter V_FRONT_PORCH     = 10;           // Vertical front porch (lines)
parameter V_SYNC_PULSE      = 2;            // Vertical sync pulse width (lines)
parameter V_BACK_PORCH      = 33;           // Vertical back porch (lines)
parameter V_TOTAL           = 525;          // Total vertical lines per frame

parameter COLOR_BAR_TIME    = 3;            // Color bar display time in seconds
parameter CLK_FREQ          = 385200;       // System clock frequency (Hz)
```

```
// ----------------------------------------------------------------------------
// Registers and Wires
// ----------------------------------------------------------------------------

reg  [11:0] h_count;                        // Horizontal pixel counter
reg  [11:0] v_count;                        // Vertical line counter
reg  [21:0] color_bar_timer;                // Timer for displaying color bars

wire [7:0]  color_bar_data_r;               //Register for color bar data red
wire [7:0]  color_bar_data_g;               //Register for color bar data green
wire [7:0]  color_bar_data_b;               //Register for color bar data blue

reg  [18:0] rd_addr_reg;                    // Registered read address for DPB RAM
```

This initial part of the `hdmi` module sets up the foundation for the video generation process. It includes:

1. Module Declaration and Header:
    - Declares the module `hdmi` and its input/output ports.
    - Includes external Verilog file `color_bar_hor.v` that contains a module for generating horizontal color bars.
2. Input/Output Ports:
    - `clk`: The system clock, which drives the timing of the video generation.
    - `rst_n`: An active-low reset signal to initialize the module.
    - `ram_data`: Input data from the DPBRAM, containing the image data to be displayed.
    - `address_read_dbp`: Output signal for the read address to the DPBRAM.
    - `hsync`, `vsync`, `de`: Output signals for horizontal sync, vertical sync, and data enable, respectively. These are essential timing signals for video displays.
    - `hdmi_data`: The 24-bit RGB pixel data to be sent to the HDMI output.
3. Parameters:
    - Parameters for the video timing (`H_ACTIVE`, `H_FRONT_PORCH`, etc.) determine the resolution and timing characteristics of the video signal.
    - `COLOR_BAR_TIME`: Specifies how long color bars should be displayed at the beginning.
4. Registers and Wires:
    - `h_count`, `v_count`: Registers to keep track of the current horizontal and vertical pixel positions.
    - `color_bar_timer`: A timer used to control the duration of the color bar display.
    - `hdmi_data_temp`: A temporary wire for the HDMI data.
    - `color_bar_data_r`, `color_bar_data_g`, `color_bar_data_b`: Wires to receive the RGB color data from the color bar generator modules.

- `rd_addr_reg`: A registered version of the read address for the DPBRAM.

```
// ----------------------------------------------------------------------------
// Video Timing Generation
// ----------------------------------------------------------------------------

always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
       // Reset counters and timing signals
       h_count <= 0;
       v_count <= 0;
       hsync <= 1'b1;                          // Start in horizontal sync
       vsync <= 1'b1;                          // Start in vertical sync
       de <= 1'b0;                             // Data enable initially inactive
       color_bar_timer <= 22'b0;
       show_color_bars <= 1'b1;                // Start by showing color bars
   end else begin
       // Increment counters
       if (h_count == H_TOTAL - 1) begin
           h_count <= 0;
           if (v_count == V_TOTAL - 1) begin
               v_count <= 0;                   // Reset vertical counter at end of frame
           end else begin
               v_count <= v_count + 1;         // Increment vertical counter
           end
       end else begin
           h_count <= h_count + 1;             // Increment horizontal counter
       end
                                               // Generate timing signals based on
counters and parameters
       hsync <= (h_count <= (H_BACK_PORCH + H_ACTIVE + H_FRONT_PORCH)) ? 1'b1 : 1'b0;
// Low during sync pulse
       vsync <= (v_count <= (V_BACK_PORCH + V_ACTIVE + V_FRONT_PORCH)) ? 1'b1 : 1'b0;
// Low during sync pulse
       de    <= ((h_count >= H_BACK_PORCH) && (h_count <= (H_BACK_PORCH + H_ACTIVE))) &&
((v_count >= V_BACK_PORCH) && (v_count <= (V_BACK_PORCH + V_ACTIVE))) ? 1'b1 : 1'b0;//
High during active video

       // Update Color Bar Timer and show_color_bars
       if (color_bar_timer < (140000/*425000*/ * COLOR_BAR_TIME)) begin
           color_bar_timer <= color_bar_timer + 1;
       end else begin
           show_color_bars <= 1'b0;            // Disable color bars after 5 seconds
           color_bar_timer <= 0;               // Reset the timer
           show_image <= 1'b1;                 // Set flag for output character
       end
```

```
    end
end
```

This section of the code is responsible for generating the timing signals (hsync, vsync, and de) that control the display of video data on a screen. It also includes logic for displaying color bars for a specified duration at the beginning.

Video Timing Generation:

- **Counters (h_count, v_count):** These registers keep track of the current position within the video frame. `h_count` counts the horizontal pixels in a line, and `v_count` counts the vertical lines in a frame.
- **Reset Condition:** When the `rst_n` signal is low (active), the counters are reset to 0, and the `hsync` and `vsync` signals are set high. The `de` signal is set low, and the `color_bar_timer` is reset. The `show_color_bars` flag is set to 1, indicating that color bars should be displayed initially.
- **Counter Increments:** The `h_count` increments on each clock cycle. When it reaches the end of a line (`H_TOTAL - 1`), it resets to 0, and `v_count` increments. When `v_count` reaches the end of a frame (`V_TOTAL - 1`), it also resets to 0.
- **Timing Signal Generation:**
    - **hsync:** This signal goes low during the horizontal sync pulse, which marks the beginning of each horizontal line. The sync pulse duration is determined by `H_BACK_PORCH + H_ACTIVE + H_FRONT_PORCH`.
    - **vsync:** This signal goes low during the vertical sync pulse, which marks the beginning of each vertical frame. The sync pulse duration is determined by `V_BACK_PORCH + V_ACTIVE + V_FRONT_PORCH`.
    - **de (Data Enable):** This signal is high only during the active video period, which is when the actual pixel data should be displayed on the screen. The active video period is defined by the range between `H_BACK_PORCH` and `H_BACK_PORCH + H_ACTIVE` for the horizontal direction, and between `V_BACK_PORCH` and `V_BACK_PORCH + V_ACTIVE` for the vertical direction.

Color Bar Timer:

- **Purpose:** This timer controls how long the color bars are displayed at the beginning.
- **Operation:** The `color_bar_timer` increments until it reaches a value calculated based on `COLOR_BAR_TIME` (the desired duration in seconds) and `CLK_FREQ` (the clock frequency).
- **Disabling Color Bars:** Once the timer reaches the calculated value, the `show_color_bars` flag is set to 0, disabling the color bar display. The timer is then reset to 0.

```
// ----------------------------------------------------------------------
// Pixel Data Output
// ----------------------------------------------------------------------

```

```
reg show_image;                                                                    //
Flag to control when to display image data
reg show_color_bars;                                                               //
Flag to control when to display color bars

// Create combinational logic for hdmi_data based on the state of show_image and
show_color_bars
//assign hdmi_data = (show_color_bars && !show_image) ? {color_bar_data_r,
color_bar_data_g, color_bar_data_b} : {24{ram_data[pixel_x[2:0]]}};
reg [23:0] hdmi_data_reg;

always @* begin
   if (show_color_bars) begin
       hdmi_data_reg = {color_bar_data_r, color_bar_data_g, color_bar_data_b};
   end else begin
       hdmi_data_reg = {24{ram_data[pixel_x[2:0]]}};
   end
end

assign hdmi_data = hdmi_data_reg;
```

This part of the code is responsible for controlling the output of pixel data to the HDMI interface. It determines whether to display color bars or image data based on the `show_color_bars` and `show_image` flags.

Pixel Data Output (always block):

- **Initialization (Reset Condition):**
  - When the `rst_n` signal is low (active), the `hdmi_data` output is set to 0 (black), the `de` (data enable) signal is set to 0, and the `show_image` flag is set to 0. This means that initially, no image data is displayed, and the system is prepared to show color bars.
- **Color Bar Display:**
  - If the `show_color_bars` flag is high (1) and the `show_image` flag is low (0), the `hdmi_data` output is assigned the RGB color data from the `color_bar_data_r`, `color_bar_data_g`, and `color_bar_data_b` signals.
- **Image Display:**
  - In any other case (for example, when `show_color_bars` is low or `show_image` is high), the `show_image` flag is set to 1. This indicates that the system should switch from displaying color bars to displaying the actual image data.

```
// ------------------------------------------------------------------------
// Color Bar Generator (Integrated from color_bar module)
// ------------------------------------------------------------------------

color_bar_hor my_color_bar (
```

```
    .I_pxl_clk(clk),                        // Clock for color bar
    .I_rst_n(rst_n),
    .I_h_total(H_TOTAL),
    .I_h_sync(H_SYNC_PULSE),
    .I_h_bporch(H_BACK_PORCH),
    .I_h_fporch(H_FRONT_PORCH),
    .I_h_res(H_ACTIVE),
    .I_v_total(V_TOTAL),
    .I_v_sync(V_SYNC_PULSE),
    .I_v_bporch(V_BACK_PORCH),
    .I_v_fporch(V_FRONT_PORCH),
    .I_v_res(V_ACTIVE),
    .I_h_count(h_count),
    .I_v_count(v_count),
    .I_de(de),
    .O_data_r(color_bar_data_r),
    .O_data_g(color_bar_data_g),
    .O_data_b(color_bar_data_b)
);
```

Module Instantiation:

- These lines create an instance of the `color_bar_hor` module and give it the name `my_color_bar`.

Signal Connections:

- **Clock and Reset:** The pixel clock (`clk`) and reset signal (`rst_n`) from the `hdmi` module are connected to the corresponding inputs of the `color_bar_hor` module. This ensures that the color bar generation is synchronized with the rest of the video system.
- **Video Timing Parameters:** The horizontal and vertical timing parameters (`H_TOTAL`, `H_SYNC_PULSE`, `H_BACK_PORCH`, etc.) are passed from the `hdmi` module to the `color_bar_hor` module. This allows the color bar generator to match the timing of the video signal being generated.
- **Pixel Counters:** The current horizontal and vertical pixel counts (`h_count`, `v_count`) are also passed to the `color_bar_hor` module. This information is used to determine which color should be displayed at each pixel position in the color bar.
- **Data Enable:** The `de` (data enable) signal is connected to the `color_bar_hor` module.
- **Color Bar Outputs:** The red, green, and blue color components generated by the `color_bar_hor` module are connected to the `color_bar_data_r`, `color_bar_data_g`, and `color_bar_data_b` wires in the `hdmi` module. These wires are then used to drive the `hdmi_data` output when color bars are being displayed.

Functionality:

The `color_bar_hor` module takes the timing parameters and pixel counts as inputs and generates the appropriate RGB color values for each pixel in the horizontal color bar pattern. The `hdmi` module then uses these color values to drive the `hdmi_data` output when the `show_color_bars` flag is active.

```
// ------------------------------------------------------------------------
// Read address generation logic
// ------------------------------------------------------------------------

reg [15:0]  rd_addr;                                        // Read address for DPB
RAM (16 bits for 64K addresses)
reg [6:0]   line_counter;
reg [6:0]   line_cnt_2;

always @(posedge clk) begin
   if (~(hsync | vsync)) begin
       rd_addr <= 16'd0;
       line_cnt_2 <= 7'b0;
       line_counter <= 7'b0;  // Reset address at the beginning of each frame (top-left
corner)
   end else if(((pixel_x % 8) == 0) && (pixel_x != 640)) begin // Increment read address
every 8 pixels
       rd_addr <= ((pixel_x + pixel_y) + ((line_counter * 640 - line_counter*8)));
       if ((v_count > 34) && (((v_count - 34) % 8) == 0) && (h_count == 30)) begin
           line_counter <= line_counter + 1;
       end
   end
end

assign address_read_dbp = rd_addr;          // Output the read address to the DPB RAM
```

Read Address Generation Logic (always block):

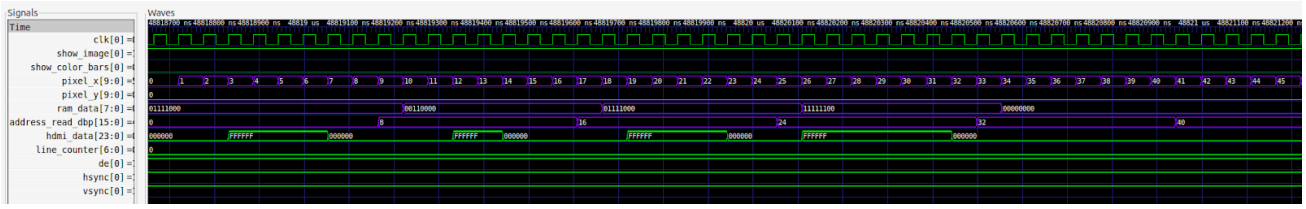
This code generates the read address (`address_read_dbp`) for the DPBRAM, which is used to fetch the pixel data to be displayed. The address is calculated based on the current pixel coordinates (`pixel_x` and `pixel_y`) and a `line_counter` that helps track the current line being read.

Functionality:

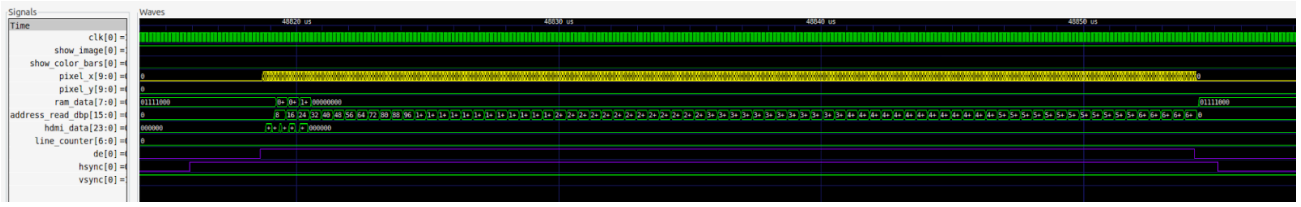
1. **Initialization (Reset Condition):**
   - When the reset signal (`rst_n`) is low, the read address (`address_read_dbp`) is reset to 0, and the `line_counter` is also reset to 0. This ensures that reading starts from the beginning of the frame (top-left corner) at the start of every frame.
2. **Address Calculation:**

- The `else if` condition checks if the horizontal pixel counter (`pixel_x`) is a multiple of 8 (meaning we're at the start of a new 8-pixel chunk) and not at the end of a line (`pixel_x != 640`).
- If these conditions are met, a new read address is calculated. The calculation involves the following:
  - `(pixel_x + pixel_y)`: This represents the base address for the current pixel.
  - `(line_counter * 640 - line_counter * 8)`: This term adjusts the base address based on the current line (`line_counter`). It seems like the image is 640 pixels wide, and this calculation is meant to compensate for the already processed pixels in the current line.

3. **Line Counter Update:**
   - The `if` condition within the `else if` block updates the `line_counter`. It checks if:
     - The vertical pixel count (`v_count`) is not equal to 34.
     - The vertical pixel count is not 0.
     - The vertical pixel count minus 34 is a multiple of 8 (meaning we're at the start of a new 8-line block).
   - If all these conditions are met, the `line_counter` is incremented by 1.



This waveform illustrates the behavior of the read address and the corresponding RAM data., the counter pixel_x is used to generate a read address for the dpb ram which, when an input address is received, send the corresponding byte as output, this value in the waveform shown is called ram data. We can see how, following the logic in the code, the address read for the ram increment every 8 pixel_x changes according to the length of each input byte to write in the output.

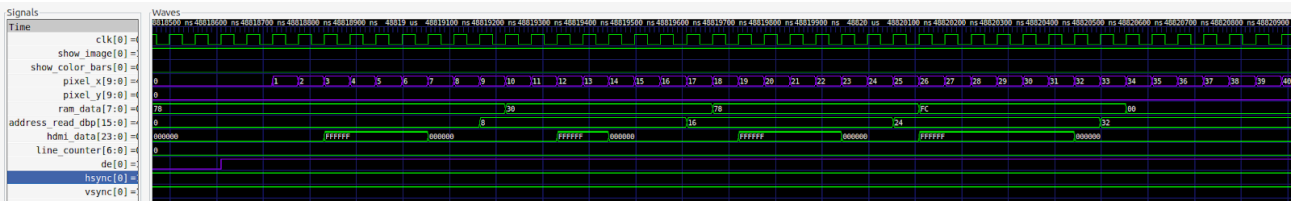


If we zoom out we can see an entire line, we can see how the h_sync flag (active low) work at the end of each line, this flag is used to reset the pixel_x counter and to manage the active zone flag, this is call de and represent the area of screen where its possible to write pixel output data.
In yellow it possible to observe the counter pixel_x's behavior following every line, from zero to 639 pixel, the counter pixel_y has a similar waveform but, instead of change its value every pixel,

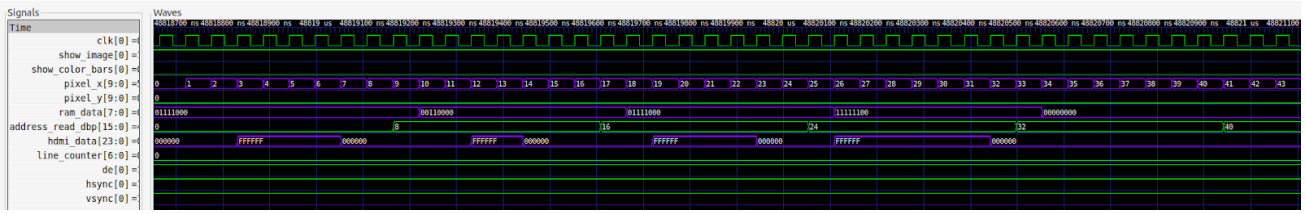
it changes its value every line, from 0 to 479 and then reset its value when both hsync and vsync are active (low), so then where an entire frame has been printed.

```
// ----------------------------------------------------------------------------
// Data output to HDMI
// ----------------------------------------------------------------------------

reg [9:0] pixel_x, pixel_y;                          // Pixel coordinates (10 bits
for 640x480 resolution)

always @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
       pixel_x <= 10'd0;                             // Reset x coordinate
       pixel_y <= 10'd0;                             // Reset y coordinate
   end else begin
       if (de && show_image) begin                   // Only update during active
display and when showing image
           pixel_x <= h_count - H_BACK_PORCH - 1;    // Subtract non-active pixels
from h_count
           pixel_y <= v_count - V_BACK_PORCH - 1;    // Subtract non-active lines from
v_count
       end else begin
           pixel_x <= 10'd0;                         // Reset x coordinate to 0 when
not in active area
           pixel_y <= 10'd0;                         // Reset y coordinate to 0 when
not in active area
       end
   end
end

endmodule
```

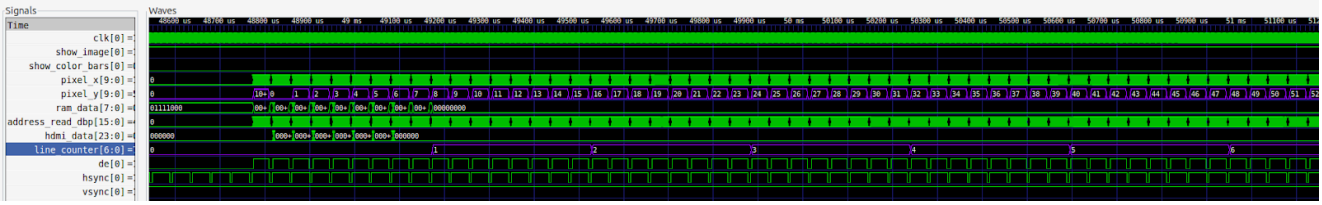
This final section of the code is responsible for outputting the pixel data to the HDMI interface. It calculates the pixel coordinates and determines whether to display the color bar or the image data from the RAM.



In this waveform is shown the behavior of pixel_x and pixel_y counters, the value of these two registers increment only in the active zone of the screen, pixel_x then reset its value at the end of every line while pixel_y reset every frame. These counters are used for different logic in the hdmi module.



From these waveform its possible to observe how the data from ram are write in the output data hdmi, following the pixel_x values, every output hdmi_data is a 24 bit value and, because the display, for character, is monochromatic, the value of the hdmi_data its 1 or 0 following the data input from ram.



In this waveform we can observe how the line counter affects the address read, the value of the line counter increments every 8 lines, this value then is used for the address read generator.

This counter is necessary because of the bitmap dimension, every character uses 8*8 pixel so, every 8 lines, it's possible to print 80 characters, with address from 0 to 639; after the first character have been printed the line counter increment it's value, showing that a new line of bitmap has begin and so the address should be between 640 and 1279 and so go on for all bitmaps lines until the last 8 lines where the read address for the ram changes from 37760 to 38399 that it's the dimension of the dpb ram.

Data Output to HDMI (always block):

- **Initialization (Reset Condition):**
  - When the `rst_n` signal is low (active), the `hdmi_data` output is cleared (set to black), and the pixel coordinates (`pixel_x`, `pixel_y`) are reset to 0.
- **Conditional Data Output:**
  - The `if (de && show_image)` condition checks if the data enable (`de`) signal is high (meaning it's time to output pixel data) and the `show_image` flag is high (meaning the color bar display should be off).
  - If both conditions are true, the code proceeds to output the image data.
- **Image Data Output:**
  - `if (ram_data[pixel_x[2:0]] == 1)`: This line checks the least significant bit of the `pixel_x` coordinate. This bit is used to select one of the 8 bits in the `ram_data` byte. If the selected bit is 1, the `hdmi_data` output is set to white (`6'h111111`); otherwise, it's set to black (`6'h000000`).
  - The commented-out line `//hdmi_data <= {24{ram_data[pixel_x[2:0]]}};` suggests an alternative way to output the pixel data, where the selected bit is replicated 24 times to create a 24-bit RGB value. However, this line is not currently active.

- **Pixel Coordinate Calculation:**
  - `pixel_x <= h_count - H_BACK_PORCH - 1;`: This line calculates the x-coordinate of the current pixel within the active video area. It subtracts the horizontal back porch and an additional 1 from the `h_count` value.
  - `pixel_y <= v_count - V_BACK_PORCH - 1;`: This line calculates the y-coordinate of the current pixel within the active video area. It subtracts the vertical back porch and an additional 1 from the `v_count` value.
- **Black Output (Outside Active Area or Color Bars):**
  - If the `de` signal is low or the `show_image` flag is low, the `hdmi_data` output is set to black, and the pixel coordinates are reset to 0. This ensures that black is displayed outside the active video area or when color bars are being shown.

FIRMWARE

Introduction

This project involves creating a system that reads ASCII characters from the UART interface and transfers their corresponding 8x8 bitmap representations to a peripheral register. For simplicity and ease of use, we have consolidated all constants and functions within a single main file. This decision was influenced by the need to manually execute the make hex function using a Python script on a Mac.

Code Explanation

Constants and Definitions

```
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <platform.h>
#include "init.h"

#define STRBUF_SIZE          256  // String buffer size
#define UART_DATA_ADDR 0x04
#define UART_BASE_ADDR 0x10013000

const char font8x8_basic[128][8] = {
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0000 (nul)
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0001
   { 0x18, 0x3C, 0x3C, 0x18, 0x18, 0x00, 0x18, 0x00},   // U+0021 (!)
   { 0x36, 0x36, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0022 (")
   … //* all the bitmaps can be found at the end of the report*\\
   { 0x63, 0x63, 0x36, 0x1C, 0x1C, 0x36, 0x63, 0x00},   // U+0058 (X)
   { 0x33, 0x33, 0x33, 0x1E, 0x0C, 0x0C, 0x1E, 0x00},   // U+0059 (Y)
```

```
    ...
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}    // U+007F
};
```

Function Definitions

print_bitmap

This function prints the 8x8 bitmap of a character in binary form.

```
void print_bitmap(uint8_t ascii_code) {
    printf("\nBitmap for ASCII code %d ('%c'):\n", ascii_code, ascii_code);
    for (int i = 0; i < 8; i++) {
        uint8_t line = font8x8_basic[ascii_code][i];
        for (int j = 7; j >= 0; j--) {
            printf("%c", (line & (1 << j)) ? '1' : ' ');
        }
        printf("\n");
    }
}
```

print_32bit_word_as_8bit_lines

This function prints a 32-bit word as four lines of 8-bit binary values.

```
void print_32bit_word_as_8bit_lines(uint32_t word) {
    for (int line = 0; line < 4; line++) {
        uint8_t byte = (word >> (8 * (3 - line))) & 0xFF;  // Extract the byte
        for (int bit = 0; bit <= 7; bit++) {
            printf("%c", (byte & (1 << bit)) ? '1' : ' ');
        }
        printf("\n");
    }
}
```

print_32bit_word

This function prints a 32-bit word in binary form, with spaces every 8 bits for readability.

```
void print_32bit_word(uint32_t word) {
    for (int bit = 31; bit >= 0; bit--) {
        printf("%c", (word & (1 << bit)) ? '1' : '0');
        if (bit % 8 == 0) {
            printf(" "); // Add a space every 8 bits for readability
        }
    }
    printf("\n");
}
```

Main

```
int main(void) {

    const char *sample_string = "A2A"; // Sample string to display
    uint32_t transfer_data[2]; // To store two 32-bit words for each character's bitmap
    size_t string_length = strlen(sample_string);
    uint8_t ascii_code;
    uint32_t reg_value;
    uint32_t flag;
    _init();

    while (1) {
        printf("Test\n");
        reg_value = UART0_REG(UART_DATA_ADDR);
        flag = reg_value >> 31;
        ascii_code = reg_value & 0xFF;

        if (flag == 0) {
            printf("ASCII Code: %d ('%c')\n", ascii_code, ascii_code);
            for (int i = 0; i < 2; i++) {
                transfer_data[i] = 0; // Initialize the 32-bit word to 0
                for (int j = 0; j < 4; j++) {
                    transfer_data[i] |=
                    ((uint32_t)font8x8_basic[ascii_code][i * 4 + j] << (8 * (3 - j)));
                }
                MY_PERIPH_REG(MY_PERIPH_REG_IO) = (uint32_t)transfer_data[i];
                printf("Writing to peripheral register: 0x%08X\n", transfer_data[i]);
            }
        }
    }
    return 0;
}
```

Detailed Explanation

Main Loop

The main loop continuously performs the following steps:

1. **Read Data from UART**:
   - Read the value from the UART data register using UART0_REG(UART_DATA_ADDR).
   - Extract the flag bit (most significant bit) from reg_value.
   - Extract the ASCII code (least significant byte) from reg_value.
2. **Check Flag**:
   - If the flag is 0, it indicates valid data and the following steps are performed.
3. **Print ASCII Code**:
   - Print the ASCII code and corresponding character for debugging purposes.
4. **Convert ASCII to Bitmap**:
   - **Outer Loop**:

- Iterate twice to process the 8x8 bitmap in two 32-bit words.
- Initialize the current 32-bit word in transfer_data to 0.
- **Inner Loop**:
  - Iterate four times to combine four 8-bit rows into the current 32-bit word.
  - Access the appropriate byte from the 8x8 bitmap based on the ASCII code.
  - Shift the byte to its correct position within the 32-bit word.
  - Use a bitwise OR operation to combine the shifted byte into the 32-bit word.

5. **Write to Peripheral Register**:
   - Write the constructed 32-bit word to the peripheral register MY_PERIPH_REG(MY_PERIPH_REG_IO).
   - Print the 32-bit word written to the peripheral register for debugging purposes.

Schematic Flow

1. **Initialization**:
   - System Setup: _init()
   - Declare Variables: sample_string, transfer_data, string_length, ascii_code, reg_value, flag
2. **Main Loop**:
   - **Read Data from UART**:
     - reg_value = UART0_REG(UART_DATA_ADDR);
     - flag = reg_value >> 31;
     - ascii_code = reg_value & 0xFF;
   - **Check Flag**:
     - if (flag == 0) {
   - **Print ASCII Code**:
     - printf("ASCII Code: %d ('%c')\n", ascii_code, ascii_code);
   - **Convert ASCII to Bitmap**:
     - **Outer Loop** (for (int i = 0; i < 2; i++)):
       - transfer_data[i] = 0;
       - **Inner Loop** (for (int j = 0; j < 4; j++)):
         - transfer_data[i] |= ((uint32_t)font8x8_basic[ascii_code][i * 4 + j] << (8 * (3 - j)));
   - **Write to Peripheral Register**:
     - MY_PERIPH_REG(MY_PERIPH_REG_IO) = (uint32_t)transfer_data[i];
     - printf("Writing to peripheral register: 0x%08X\n", transfer_data[i]);

Explanation of Data Flow

1. **Reading Data**:
   - Data is read from the UART interface and stored in reg_value.
   - The flag and ASCII code are extracted from reg_value.
2. **Processing ASCII Code**:
   - The ASCII code is printed for debugging.
   - The 8x8 bitmap of the character is processed in two 32-bit words using nested loops.
3. **Bitmap Conversion**:
   - For each 32-bit word, four bytes (rows) from the bitmap are combined.
   - Bytes are shifted to their correct positions and combined using bitwise OR operations.
4. **Writing to Peripheral**:
   - The constructed 32-bit words are written to the peripheral register.
   - Each write operation is printed for debugging.

Additional Functionality (Commented Out)

The commented-out code is for handling strings and printing bitmaps. It demonstrates how to convert each character in a string to its bitmap representation and print the resulting 32-bit words. These functions can be enabled for further debugging or extended functionality.

```
/*
for (size_t char_idx = 0; char_idx < string_length; char_idx++) {
   ascii_code = sample_string[char_idx]; // Get ASCII code of the character
   printf("ASCII Code: %d ('%c')\n", ascii_code, ascii_code);

   // Convert the ASCII code to its corresponding bitmap
   for (int i = 0; i < 2; i++) {
       transfer_data[i] = 0; // Initialize the 32-bit word to 0
       for (int j = 0; j < 4; j++) {
           transfer_data[i] |= ((uint32_t)font8x8_basic[ascii_code][i * 4 + j] << (8 *
(3 - j)));
       }
       print_32bit_word(transfer_data[i]); // Print the 32-bit word for debugging
       MY_PERIPH_REG(MY_PERIPH_REG_IO) = (uint32_t)transfer_data[i];
       printf("Writing to peripheral register: 0x%08X\n", transfer_data[i]);
   }
}
```

Conclusion

This project demonstrates a practical approach to reading ASCII characters from a UART interface and transferring their 8x8 bitmap representations to a peripheral register. The decision to use a single main file simplifies the build process, especially when using a Mac, where the make hex function requires manual execution via a Python script. This document provides a comprehensive overview of the implemented functions, their purposes, and their operation within the system.

SIM

The simulation environment connects the core IP and drives the simulation to validate the correct operation of the HDMI driver. Understanding how the UART signals work is essential for this simulation. UART (Universal Asynchronous Receiver-Transmitter) employs TX (transmit) and RX (receive) signals for serial communication, allowing asynchronous data transfer between devices without the need for a shared clock signal. By simulating these UART operations, we ensure our HDMI driver functions as expected in a real-world scenario.

The provided Verilog code simulates a system that reads ASCII characters from a UART interface and sends them to the e203 core. The UART fetches ASCII characters from an input.txt file, and the simulation drives the signals to validate the system's functionality.

Clock and Reset Setup

```
`timescale 1ns/10ps
`define USING_IVERILOG

module sys_tb_top();
```

```
// Clock and reset signals
reg  clk;
reg  lfextclk;
reg  rst_n;

// Signal assignments
wire hfclk = clk;
wire uart_rx;
wire [31:0] gpio;
assign uart_rx = gpio[17];
```

Clock and Reset Signals:

- clk: The main clock signal.
- lfextclk: A low-frequency external clock signal.
- rst_n: Active-low reset signal.

Signal Assignments:

- hfclk: Alias for the main clock (clk).
- uart_rx: Alias for the UART RX signal, mapped to gpio[17].

Waveform Generation

```
// Waveform generation for Icarus Verilog
`ifdef USING_IVERILOG
initial begin
  $dumpfile("waveout.vcd");
  $dumpvars(0, sys_tb_top);
end
`endif

// Waveform generation for VCS
`ifdef USING_VCS
initial begin
  $fsdbDumpfile("test.fsdb");
  $fsdbDumpvars;
end
`endif
```

Waveform Generation:

- For Icarus Verilog (USING_IVERILOG): Dumps waveform data to waveout.vcd.
- For VCS (USING_VCS): Dumps waveform data to test.fsdb.

Simulation End Condition and Initial Conditions

```
// Simulation end condition
initial begin
  #150ms;
  $finish;
end
```

```
// Initial conditions
initial begin
  clk        <= 0;
  lfextclk   <= 0;
  rst_n      <= 0;
  #320us rst_n <= 1;
end
```

Simulation End Condition:

- The simulation runs for 150 milliseconds and then finishes.

Initial Conditions:

- clk, lfextclk, and rst_n are initialized to 0.
- rst_n is asserted (set to 1) after 320 microseconds to simulate a reset pulse.

Clock Generation

```
// Clock generation: 27 MHz
always begin
   #18.52 clk <= ~clk;
end

// Low frequency clock generation
always begin
   #33 lfextclk <= ~lfextclk;
end
```

Clock Generation:

- The main clock (clk) toggles every 18.52 nanoseconds, generating a 27 MHz clock.
- The low-frequency clock (lfextclk) toggles every 33 nanoseconds.

UART Data Transfer Setup

```
// UART data transfer setup
int uart_tx_period = 1e9 / 115200; // UART TX period calculation based on baud rate
int uart_rx_period = 8750; // UART RX period calculation

// UART signal declarations
reg [31:0] gpio_in;  // default value bit 16 (uart_tx) should be 1
reg [31:0] gpio_out; // bit 17 is uart rx

reg [7:0] uart_rx_byte;
```

UART Period Calculation:

- uart_tx_period is calculated based on a baud rate of 115200 bits per second (bps).
- uart_rx_period is set to 8.75 microseconds.

UART Signal Declarations:

- gpio_in: Used for transmitting data (TX), with bit 16 representing the UART TX signal.
- gpio_out: Used for receiving data (RX), with bit 17 representing the UART RX signal.

UART Transmission Task

```
// UART transmission task
task uart_tx_data(input bit [7:0] tx_data);
  // 1 bit start bit
  gpio_in[16] = 1'b0;
  #uart_tx_period;

  // 8 bit data: LSB first
  for (int i = 0; i < 8; i++) begin
    gpio_in[16] = tx_data[i];
    #uart_tx_period;
  end

  // 1 bit stop bit
  gpio_in[16] = 1'b1;
  #uart_tx_period;
endtask
```

uart_tx_data:

- This task handles UART data transmission.
- It sends a start bit (logic low), followed by the 8-bit data (LSB first), and ends with a stop bit (logic high).

Reading Data from File and Sending to e203 Core

```
// Reading file and sending data to e203 core
initial begin
  reg [7:0] sim_data[3:0];
  gpio_in[16] = 1'b1; // UART TX idle state is high
  $readmemh("./input.txt", sim_data); // Load data from input.txt into sim_data array

  #7ms; // Wait for system to stabilize

  foreach (sim_data[x]) begin
    $display("tx_data[%x] = %x", x, sim_data[x]); // Display data being sent
    uart_tx_data(sim_data[x]); // Transmit each byte via UART
  end
end
```

Initial Block:

- The sim_data array is used to store the ASCII characters read from input.txt.
- The UART TX signal (gpio_in[16]) is set to 1 when idle.
- $readmemh("./input.txt", sim_data): Reads hexadecimal data from input.txt into the sim_data array.
- After a 7 millisecond delay to allow the system to stabilize, the simulation begins transmitting data.
- foreach (sim_data[x]): Loops through each byte in the sim_data array.
  - $display("tx_data[%x] = %x", x, sim_data[x]): Prints the index and value of the current data byte being transmitted.

- uart_tx_data(sim_data[x]): Calls the uart_tx_data task to transmit the current data byte via UART.

Instantiating the e203_soc_demo Module

```
  // Instantiating the e203_soc_demo module
e203_soc_demo uut (
  .V_sync             (),
  .H_sync             (),
  .hdmi_data_out      (),
  .clk_in             (clk),
  .tck                (),
  .tms                (),
  .tdi                (),
  .tdo                (),
  .gpio_in            (gpio_in),
  .gpio_out           (gpio_out),
  .qspi_in            (),
  .qspi_out           (),
  .qspi_sck           (),
  .qspi_cs            (),
  .erstn              (rst_n),
  .dbgmode0_n         (1'b1),
  .dbgmode1_n         (1'b1),
  .dbgmode3_n         (1'b1),
  .bootrom_n          (1'b0),
  .aon_pmu_dwakeup_n  (),
  .aon_pmu_padrst     (),
  .aon_pmu_vddpaden   ()
);
endmodule
```

Module Instantiation:

- The e203_soc_demo module is instantiated and connected to the simulation environment.
- Signals such as clk_in, gpio_in, gpio_out, and rst_n are connected to the corresponding ports of the module.
- Other signals (V_sync, H_sync, hdmi_data_out, etc.) are left unconnected (()).

Data Flow and Connections

1. **Data Reading**:
   - ASCII characters are read from the input.txt file into the sim_data array.
2. **UART Transmission**:
   - The uart_tx_data task sends each character from sim_data to the e203_soc_demo module via the gpio_in[16] signal.
3. **Peripheral Module**:
   - The e203_soc_demo module receives the transmitted characters and processes them as required by the system.
4. **Simulation Control**:

- The simulation environment includes waveform generation for debugging and visualizing signal changes.
- The simulation ends after 150 milliseconds.

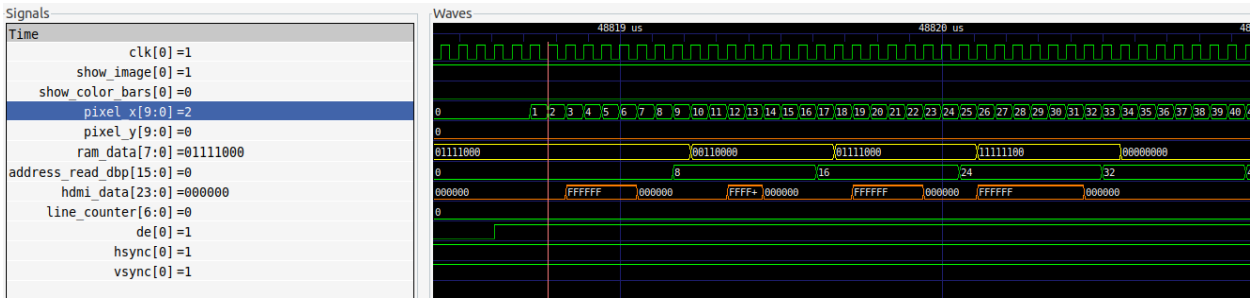
This simulation setup ensures that the HDMI driver's functionality is thoroughly tested by transmitting ASCII characters via UART and observing the behavior of the e203_soc_demo module. The UART interface fetches characters from an input file, mimicking a real-world scenario where data is received serially.

OUTPUT DATA ANALYSIS

For the sake of this simulation and analysis we input from the uart the characters "3A27"
For ease and the sake of not being repetitive we showcase a thorough analysis of the first two pixel lines of the screen and line seven (line 8 is always zeros).

Each byte of ram_data represents 8 pixels, with each bit corresponding to a pixel on the display. A '1' bit in ram_data represents an active (high) pixel, and a '0' bit represents an inactive (low) pixel. Let's analyze the output for the first two lines.

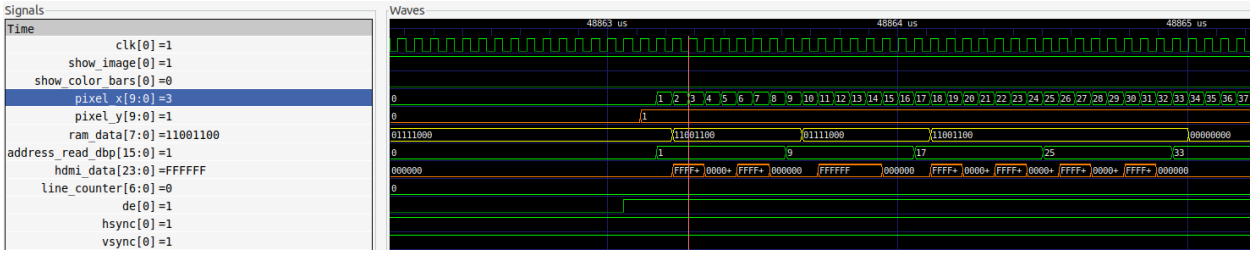
First Two Lines
The ram_data for the first two lines is as follows:
Line 1: 01111000 00110000 01111000 11111100
Line 2: 11001100 01111000 11001100 11001100



Pixel_y = 0 (First Line of the Characters)

- **ram_data[0] = 01111000:**
  - Pixel_x = 0: ram_data[0][7] = 0 -> hdmi_data = 000000
  - Pixel_x = 1: ram_data[0][6] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 2: ram_data[0][5] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 3: ram_data[0][4] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 4: ram_data[0][3] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 5: ram_data[0][2] = 0 -> hdmi_data = 000000
  - Pixel_x = 6: ram_data[0][1] = 0 -> hdmi_data = 000000
  - Pixel_x = 7: ram_data[0][0] = 0 -> hdmi_data = 000000

- **ram_data[8] = 00110000:**
  - Pixel_x = 8: ram_data[8][7] = 0 -> hdmi_data = 000000
  - Pixel_x = 9: ram_data[8][6] = 0 -> hdmi_data = 000000
  - Pixel_x = 10: ram_data[8][5] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 11: ram_data[8][4] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 12: ram_data[8][3] = 0 -> hdmi_data = 000000
  - Pixel_x = 13: ram_data[8][2] = 0 -> hdmi_data = 000000
  - Pixel_x = 14: ram_data[8][1] = 0 -> hdmi_data = 000000
  - Pixel_x = 15: ram_data[8][0] = 0 -> hdmi_data = 000000
- **ram_data[16] = 01111000:**
  - Pixel_x = 16: ram_data[16][7] = 0 -> hdmi_data = 000000
  - Pixel_x = 17: ram_data[16][6] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 18: ram_data[16][5] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 19: ram_data[16][4] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 20: ram_data[16][3] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 21: ram_data[16][2] = 0 -> hdmi_data = 000000
  - Pixel_x = 22: ram_data[16][1] = 0 -> hdmi_data = 000000
  - Pixel_x = 23: ram_data[16][0] = 0 -> hdmi_data = 000000
- **ram_data[24] = 11111100:**
  - Pixel_x = 24: ram_data[32][7] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 25: ram_data[32][6] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 26: ram_data[32][5] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 27: ram_data[32][4] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 28: ram_data[32][3] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 29: ram_data[32][2] = 1 -> hdmi_data = FFFFFF
  - Pixel_x = 30: ram_data[32][1] = 0 -> hdmi_data = 000000
  - Pixel_x = 31: ram_data[32][0] = 0 -> hdmi_data = 000000
-

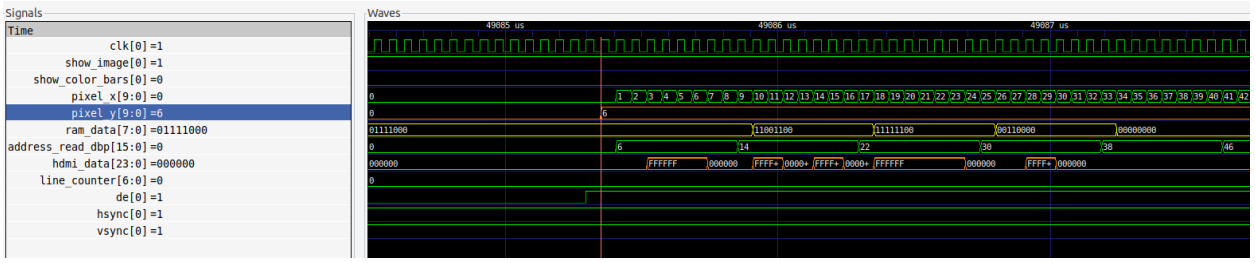


Pixel_y = 1 (Second Line of the Characters)

- **ram_data**[1] = 11001100:
  - **Pixel_x** = 0: **ram_data**[1][7] = 1 -> **hdmi_data** = FFFFFF
  - **Pixel_x** = 1: **ram_data**[1][6] = 1 -> **hdmi_data** = FFFFFF
  - **Pixel_x** = 2: **ram_data**[1][5] = 0 -> **hdmi_data** = 000000
  - **Pixel_x** = 3: **ram_data**[1][4] = 0 -> **hdmi_data** = 000000
  - **Pixel_x** = 4: **ram_data**[1][3] = 1 -> **hdmi_data** = FFFFFF
  - **Pixel_x** = 5: **ram_data**[1][2] = 1 -> **hdmi_data** = FFFFFF
  - **Pixel_x** = 6: **ram_data**[1][1] = 0 -> **hdmi_data** = 000000
  - **Pixel_x** = 7: **ram_data**[1][0] = 0 -> **hdmi_data** = 000000
- **ram_data**[9] = 01111000:

    - **Pixel_x** = 8: **ram_data**[9][7] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 9: **ram_data**[9][6] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 10: **ram_data**[9][5] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 11: **ram_data**[9][4] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 12: **ram_data**[9][3] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 13: **ram_data**[9][2] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 14: **ram_data**[9][1] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 15: **ram_data**[9][0] = 0 -> **hdmi_data** = 000000
- **ram_data**[17] = 11001100:
    - **Pixel_x** = 16: **ram_data**[17][7] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 17: **ram_data**[17][6] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 18: **ram_data**[17][5] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 19: **ram_data**[17][4] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 20: **ram_data**[17][3] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 21: **ram_data**[17][2] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 22: **ram_data**[17][1] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 23: **ram_data**[17][0] = 0 -> **hdmi_data** = 000000
- **ram_data**[25] = 11001100:
    - **Pixel_x** = 24: **ram_data**[33][7] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 25: **ram_data**[33][6] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 26: **ram_data**[33][5] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 27: **ram_data**[33][4] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 28: **ram_data**[33][3] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 29: **ram_data**[33][2] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 30: **ram_data**[33][1] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 31: **ram_data**[33][0] = 0 -> **hdmi_data** = 000000

Line 7: 01111000 11001100 11111100 00110000



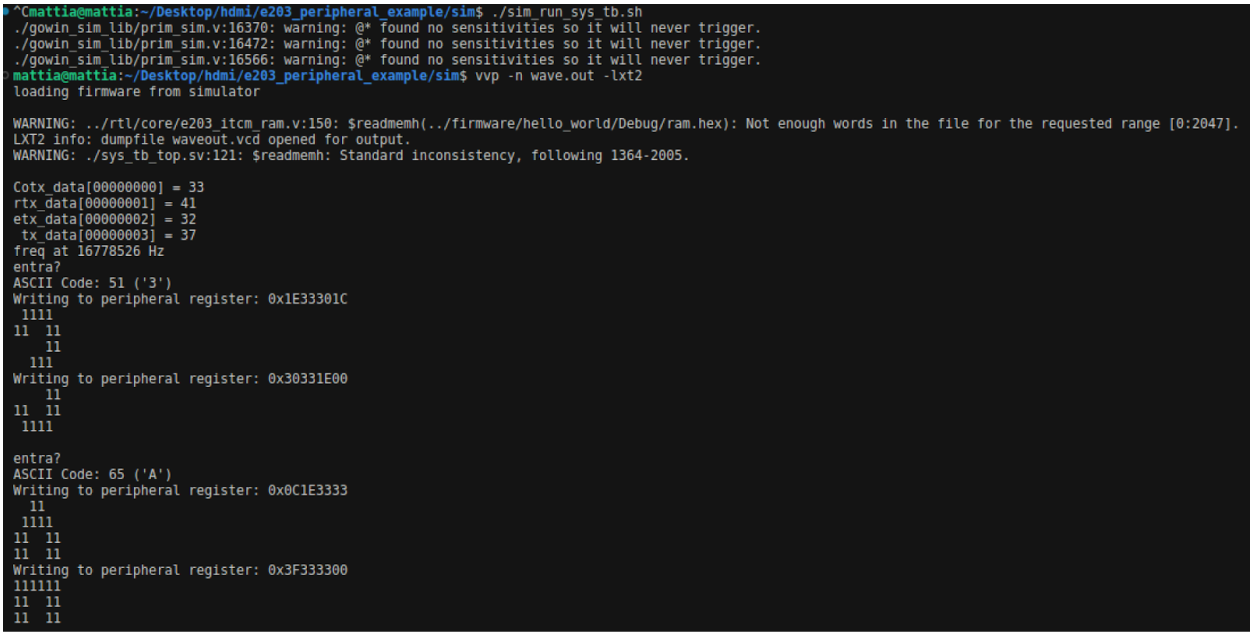
.**Pixel_y** = 6 (Seventh Line of the Characters)

- **ram_data**[6] = 01111000:
    - **Pixel_x** = 0: **ram_data**[6][7] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 1: **ram_data**[6][6] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 2: **ram_data**[6][5] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 3: **ram_data**[6][4] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 4: **ram_data**[6][3] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 5: **ram_data**[6][2] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 6: **ram_data**[6][1] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 7: **ram_data**[6][0] = 0 -> **hdmi_data** = 000000
- **ram_data**[14] = 11001100:
    - **Pixel_x** = 8: **ram_data**[14][7] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 9: **ram_data**[14][6] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 10: **ram_data**[14][5] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 11: **ram_data**[14][4] = 0 -> **hdmi_data** = 000000

    - **Pixel_x** = 12: **ram_data**[14][3] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 13: **ram_data**[14][2] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 14: **ram_data**[14][1] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 15: **ram_data**[14][0] = 0 -> **hdmi_data** = 000000
- **rram_data**[30] = 00110000:
    - **Pixel_x** = 24: **ram_data**[30][7] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 25: **ram_data**[30][6] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 26: **ram_data**[30][5] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 27: **ram_data**[30][4] = 1 -> **hdmi_data** = FFFFFF
    - **Pixel_x** = 28: **ram_data**[30][3] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 29: **ram_data**[30][2] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 30: **ram_data**[30][1] = 0 -> **hdmi_data** = 000000
    - **Pixel_x** = 31: **ram_data**[30][0] = 0 -> **hdmi_data** = 000000

This analysis illustrates how each bit in the ram_data is mapped to a corresponding pixel on the display, with hdmi_data being updated to reflect the state of each pixel (active or inactive). By systematically scanning through pixel_x and pixel_y, the module constructs the image line by line, pixel by pixel.

To further prove the correctness and functionality of the data flow we've inserted some printf statements in the firmware. The following are snapshots of the simulation terminal with easy to read data and bitmaps for all 4 characters showcased:



The information printed on screen is the following:
• flag to check whether the loop starts
• First hex 32 bit word
• First half of bitmap
• Second hex 32 bit word
• Second half bitmap

For conversion purposes from the bitmap to the ram, the 32 bit words figure as “inverted”.

DISCUSSION & CONCLUSION

This project has been a significant learning experience, providing a comprehensive understanding of HDMI module development, SoC architecture, UART data handling, and FPGA basics. We successfully built an HDMI module capable of displaying bitmap characters by driving pixel data based on RAM data, despite encountering several challenges.

Hardware and Testing Limitations:

Utilizing Apple Macs and virtual machines presented notable challenges. Compiling simulations was time-consuming, slowing down the development process. Additionally, synthesizing and testing on Gowin chips with Apple machines proved infeasible. We relied on online servers for these tasks, further complicating and delaying the development and testing phases.

Positive Outcomes:

1. **Understanding SoC Architecture:** We gained a solid understanding of System-on-Chip (SoC) architecture. This included learning how different components interact within an SoC, which is crucial knowledge for future projects.
2. **Driving UART Data:** We developed the capability to handle UART data, a critical skill for managing serial communication in embedded systems applications.
3. **Firmware Development on Hardware Chips:** We acquired hands-on experience in writing and debugging firmware for hardware chips, essential for configuring and controlling hardware components in real-world scenarios.
4. **FPGA Development Basics:** We learned the fundamentals of FPGA development, expanding our technical expertise in a valuable area. Understanding how to program and utilize FPGAs is a vital skill in the tech industry.
5. **Learning VHDL Language:** Additionally, we became proficient in VHDL, a key language for designing and implementing digital circuits. This knowledge will be beneficial for any future FPGA-related projects.

In summary, while the project presented several challenges, it provided a comprehensive learning journey. We now have a better grasp of SoC architecture, UART communication, firmware development, FPGA basics, and VHDL. These skills will undoubtedly enhance the efficiency and effectiveness of future projects.

REFERENCES:

Introduction on fpga graphics - https://projectf.io/posts/fpga-graphics/

Here are all the encoded bitmaps*

```
char font8x8_basic[128][8] = {
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0000 (null)
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0001
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0002
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0003
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0004
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0005
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0006
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0007
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0008
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0009
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000A
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000B
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000C
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000D
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000E
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+000F
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0010
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0011
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0012
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0013
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0014
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0015
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0016
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0017
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0018
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0019
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001A
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001B
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001C
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001D
```

```
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001E
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+001F
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0020 (space)
    { 0x18, 0x3C, 0x3C, 0x18, 0x18, 0x00, 0x18, 0x00},   // U+0021 (!)
    { 0x36, 0x36, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0022 (")
    { 0x36, 0x36, 0x7F, 0x36, 0x7F, 0x36, 0x36, 0x00},   // U+0023 (#)
    { 0x0C, 0x3E, 0x03, 0x1E, 0x30, 0x1F, 0x0C, 0x00},   // U+0024 ($)
    { 0x00, 0x63, 0x33, 0x18, 0x0C, 0x66, 0x63, 0x00},   // U+0025 (%)
    { 0x1C, 0x36, 0x1C, 0x6E, 0x3B, 0x33, 0x6E, 0x00},   // U+0026 (&)
    { 0x06, 0x06, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0027 (')
    { 0x18, 0x0C, 0x06, 0x06, 0x06, 0x0C, 0x18, 0x00},   // U+0028 (()
    { 0x06, 0x0C, 0x18, 0x18, 0x18, 0x0C, 0x06, 0x00},   // U+0029 ())
    { 0x00, 0x66, 0x3C, 0xFF, 0x3C, 0x66, 0x00, 0x00},   // U+002A (*)
    { 0x00, 0x0C, 0x0C, 0x3F, 0x0C, 0x0C, 0x00, 0x00},   // U+002B (+)
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x0C, 0x0C, 0x06},   // U+002C (,)
    { 0x00, 0x00, 0x00, 0x3F, 0x00, 0x00, 0x00, 0x00},   // U+002D (-)
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x0C, 0x0C, 0x00},   // U+002E (.)
    { 0x60, 0x30, 0x18, 0x0C, 0x06, 0x03, 0x01, 0x00},   // U+002F (/)
    { 0x3E, 0x63, 0x73, 0x7B, 0x6F, 0x67, 0x3E, 0x00},   // U+0030 (0)
    { 0x0C, 0x0E, 0x0C, 0x0C, 0x0C, 0x0C, 0x3F, 0x00},   // U+0031 (1)
    { 0x1E, 0x33, 0x30, 0x1C, 0x06, 0x33, 0x3F, 0x00},   // U+0032 (2)
    { 0x1E, 0x33, 0x30, 0x1C, 0x30, 0x33, 0x1E, 0x00},   // U+0033 (3)
    { 0x38, 0x3C, 0x36, 0x33, 0x7F, 0x30, 0x78, 0x00},   // U+0034 (4)
    { 0x3F, 0x03, 0x1F, 0x30, 0x30, 0x33, 0x1E, 0x00},   // U+0035 (5)
    { 0x1C, 0x06, 0x03, 0x1F, 0x33, 0x33, 0x1E, 0x00},   // U+0036 (6)
    { 0x3F, 0x33, 0x30, 0x18, 0x0C, 0x0C, 0x0C, 0x00},   // U+0037 (7)
    { 0x1E, 0x33, 0x33, 0x1E, 0x33, 0x33, 0x1E, 0x00},   // U+0038 (8)
    { 0x1E, 0x33, 0x33, 0x3E, 0x30, 0x18, 0x0E, 0x00},   // U+0039 (9)
    { 0x00, 0x0C, 0x0C, 0x00, 0x00, 0x0C, 0x0C, 0x00},   // U+003A (:)
    { 0x00, 0x0C, 0x0C, 0x00, 0x00, 0x0C, 0x0C, 0x06},   // U+003B (;)
    { 0x18, 0x0C, 0x06, 0x03, 0x06, 0x0C, 0x18, 0x00},   // U+003C (<)
    { 0x00, 0x00, 0x3F, 0x00, 0x00, 0x3F, 0x00, 0x00},   // U+003D (=)
    { 0x06, 0x0C, 0x18, 0x30, 0x18, 0x0C, 0x06, 0x00},   // U+003E (>)
    { 0x1E, 0x33, 0x30, 0x18, 0x0C, 0x00, 0x0C, 0x00},   // U+003F (?)
    { 0x3E, 0x63, 0x7B, 0x7B, 0x7B, 0x03, 0x1E, 0x00},   // U+0040 (@)
    { 0x0C, 0x1E, 0x33, 0x33, 0x3F, 0x33, 0x33, 0x00},   // U+0041 (A)
    { 0x3F, 0x66, 0x66, 0x3E, 0x66, 0x66, 0x3F, 0x00},   // U+0042 (B)
    { 0x3C, 0x66, 0x03, 0x03, 0x03, 0x66, 0x3C, 0x00},   // U+0043 (C)
    { 0x1F, 0x36, 0x66, 0x66, 0x66, 0x36, 0x1F, 0x00},   // U+0044 (D)
    { 0x7F, 0x46, 0x16, 0x1E, 0x16, 0x46, 0x7F, 0x00},   // U+0045 (E)
    { 0x7F, 0x46, 0x16, 0x1E, 0x16, 0x06, 0x0F, 0x00},   // U+0046 (F)
    { 0x3C, 0x66, 0x03, 0x03, 0x73, 0x66, 0x7C, 0x00},   // U+0047 (G)
    { 0x33, 0x33, 0x33, 0x3F, 0x33, 0x33, 0x33, 0x00},   // U+0048 (H)
    { 0x1E, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x1E, 0x00},   // U+0049 (I)
    { 0x78, 0x30, 0x30, 0x30, 0x33, 0x33, 0x1E, 0x00},   // U+004A (J)
```

```
  { 0x67, 0x66, 0x36, 0x1E, 0x36, 0x66, 0x67, 0x00},   // U+004B (K)
  { 0x0F, 0x06, 0x06, 0x06, 0x46, 0x66, 0x7F, 0x00},   // U+004C (L)
  { 0x63, 0x77, 0x7F, 0x7F, 0x6B, 0x63, 0x63, 0x00},   // U+004D (M)
  { 0x63, 0x67, 0x6F, 0x7B, 0x73, 0x63, 0x63, 0x00},   // U+004E (N)
  { 0x1C, 0x36, 0x63, 0x63, 0x63, 0x36, 0x1C, 0x00},   // U+004F (O)
  { 0x3F, 0x66, 0x66, 0x3E, 0x06, 0x06, 0x0F, 0x00},   // U+0050 (P)
  { 0x1E, 0x33, 0x33, 0x33, 0x3B, 0x1E, 0x38, 0x00},   // U+0051 (Q)
  { 0x3F, 0x66, 0x66, 0x3E, 0x36, 0x66, 0x67, 0x00},   // U+0052 (R)
  { 0x1E, 0x33, 0x07, 0x0E, 0x38, 0x33, 0x1E, 0x00},   // U+0053 (S)
  { 0x3F, 0x2D, 0x0C, 0x0C, 0x0C, 0x0C, 0x1E, 0x00},   // U+0054 (T)
  { 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x3F, 0x00},   // U+0055 (U)
  { 0x33, 0x33, 0x33, 0x33, 0x33, 0x1E, 0x0C, 0x00},   // U+0056 (V)
  { 0x63, 0x63, 0x63, 0x6B, 0x7F, 0x77, 0x63, 0x00},   // U+0057 (W)
  { 0x63, 0x63, 0x36, 0x1C, 0x1C, 0x36, 0x63, 0x00},   // U+0058 (X)
  { 0x33, 0x33, 0x33, 0x1E, 0x0C, 0x0C, 0x1E, 0x00},   // U+0059 (Y)
  { 0x7F, 0x63, 0x31, 0x18, 0x4C, 0x66, 0x7F, 0x00},   // U+005A (Z)
  { 0x1E, 0x06, 0x06, 0x06, 0x06, 0x06, 0x1E, 0x00},   // U+005B ([)
  { 0x03, 0x06, 0x0C, 0x18, 0x30, 0x60, 0x40, 0x00},   // U+005C (\)
  { 0x1E, 0x18, 0x18, 0x18, 0x18, 0x18, 0x1E, 0x00},   // U+005D (])
  { 0x08, 0x1C, 0x36, 0x63, 0x00, 0x00, 0x00, 0x00},   // U+005E (^)
  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF},   // U+005F (_)
  { 0x0C, 0x0C, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00},   // U+0060 (`)
  { 0x00, 0x00, 0x1E, 0x30, 0x3E, 0x33, 0x6E, 0x00},   // U+0061 (a)
  { 0x07, 0x06, 0x06, 0x3E, 0x66, 0x66, 0x3B, 0x00},   // U+0062 (b)
  { 0x00, 0x00, 0x1E, 0x33, 0x03, 0x33, 0x1E, 0x00},   // U+0063 (c)
  { 0x38, 0x30, 0x30, 0x3e, 0x33, 0x33, 0x6E, 0x00},   // U+0064 (d)
  { 0x00, 0x00, 0x1E, 0x33, 0x3f, 0x03, 0x1E, 0x00},   // U+0065 (e)
  { 0x1C, 0x36, 0x06, 0x0f, 0x06, 0x06, 0x0F, 0x00},   // U+0066 (f)
  { 0x00, 0x00, 0x6E, 0x33, 0x33, 0x3E, 0x30, 0x1F},   // U+0067 (g)
  { 0x07, 0x06, 0x36, 0x6E, 0x66, 0x66, 0x67, 0x00},   // U+0068 (h)
  { 0x0C, 0x00, 0x0E, 0x0C, 0x0C, 0x0C, 0x1E, 0x00},   // U+0069 (i)
  { 0x30, 0x00, 0x30, 0x30, 0x30, 0x33, 0x33, 0x1E},   // U+006A (j)
  { 0x07, 0x06, 0x66, 0x36, 0x1E, 0x36, 0x67, 0x00},   // U+006B (k)
  { 0x0E, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x1E, 0x00},   // U+006C (l)
  { 0x00, 0x00, 0x33, 0x7F, 0x7F, 0x6B, 0x63, 0x00},   // U+006D (m)
  { 0x00, 0x00, 0x1F, 0x33, 0x33, 0x33, 0x33, 0x00},   // U+006E (n)
  { 0x00, 0x00, 0x1E, 0x33, 0x33, 0x33, 0x1E, 0x00},   // U+006F (o)
  { 0x00, 0x00, 0x3B, 0x66, 0x66, 0x3E, 0x06, 0x0F},   // U+0070 (p)
  { 0x00, 0x00, 0x6E, 0x33, 0x33, 0x3E, 0x30, 0x78},   // U+0071 (q)
  { 0x00, 0x00, 0x3B, 0x6E, 0x66, 0x06, 0x0F, 0x00},   // U+0072 (r)
  { 0x00, 0x00, 0x3E, 0x03, 0x1E, 0x30, 0x1F, 0x00},   // U+0073 (s)
  { 0x08, 0x0C, 0x3E, 0x0C, 0x0C, 0x2C, 0x18, 0x00},   // U+0074 (t)
  { 0x00, 0x00, 0x33, 0x33, 0x33, 0x33, 0x6E, 0x00},   // U+0075 (u)
  { 0x00, 0x00, 0x33, 0x33, 0x33, 0x1E, 0x0C, 0x00},   // U+0076 (v)
  { 0x00, 0x00, 0x63, 0x6B, 0x7F, 0x7F, 0x36, 0x00},   // U+0077 (w)
```

```
    { 0x00, 0x00, 0x63, 0x36, 0x1C, 0x36, 0x63, 0x00},    // U+0078 (x)
    { 0x00, 0x00, 0x33, 0x33, 0x33, 0x3E, 0x30, 0x1F},    // U+0079 (y)
    { 0x00, 0x00, 0x3F, 0x19, 0x0C, 0x26, 0x3F, 0x00},    // U+007A (z)
    { 0x38, 0x0C, 0x0C, 0x07, 0x0C, 0x0C, 0x38, 0x00},    // U+007B ({)
    { 0x18, 0x18, 0x18, 0x00, 0x18, 0x18, 0x18, 0x00},    // U+007C (|)
    { 0x07, 0x0C, 0x0C, 0x38, 0x0C, 0x0C, 0x07, 0x00},    // U+007D (})
    { 0x6E, 0x3B, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},    // U+007E (~)
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}     // U+007F
};
```