

Disclaimer

The document is written where each quoted question has answer below it. This is intended to help quicker and easier grading. It has been *best-effort*, but there is no guarantee, that the answer will only appear under that specific question. Please grade each question on all the content appeared in the report.

Introduction

The HDMI Text Mode Controller IP supports a toy-version of the VGA text-mode standards. (It only supports 16 color palette). This allows processor to draw any text on screen with one of 16 colors picked in the customizable palette.

This design is built on top of the Lab 6.2. Specifically, it reuses the module VGA controller and VGA-to-HDMI IP. It also reuses the same Clock Wizard set up for Lab 6.2. It still uses the overall structure of the color mapper. (Checking whether the pixel is on the object, then choose different color for different case). The overall rendering pipeline / process is still quite similar to Lab 6.2 (As the Monochrome Text Display section explains). Yet, instead of using a ball module to generate ball coordinate, it now uses HDMI AXI to communicate text information from processor (store them in the registers), and generate necessary information for color mapper to display these text. In Lab 6.2, these modules are directly in the top level, and they connected to the Microblaze processor (via GPIO module) individually in the top level. All these modules are now packaged into the a IP, and the IP expose few ports that connected to Microblaze processor via AXI4 bus directly. They are now also bundled together, so the block diagram looks less chaotic. (Looking at the similarity in block diagram down below might help).

Compare to Lab 6.2 where we could only pass keycode to the hardware, this design certainly opens more control for the software. In Lab 6.2, we could only control the direction of the ball to move around. Yet, in Lab 7, we could draw any ASCII text on a 80x30 grid, and choose the 1 from 16 colors in the customizable palette. However, the cost of this more flexible control is an increased memory needs for hardware. We need to store all the text character and color information either in register or in BRAM. Increased memory needs means we are more likely to hit timing constraints and running our of LUT or BRAM if we want to add other logics later. Also, since more data needs are likely to be constantly passed through AXI4 bus (all these character information), we are more likely to saturate the overall AXI4 bus transfer speed. Depends on our need, a complete text mode display might not be desired if we are making a game with a simple graphics, which requires to draw certain objects like balls, polygons, instead of text. Yet, this text mode display is extremely useful if we are trying to recreate a old computer system (where a interactive shell is essential).

Text Display System

Monochrome Text Display

Lab 7 System / Text Mode Controller IP

The entire Lab 7 System is revolved around the Text Mode Controller IP. The top level System Verilog file only instantiate the block diagram and connects its pin to the correct physical FPGA pins. The block diagram only consists of Microblaze Processor and its necessary components. The Microblaze Processor connects one instance of Text Mode Controller IP.

(Start Describing IP, note this is the IP for week 1, the difference between week 1 and week 2 IP is detailed in Color Text Display)

The HDMI Text Mode Controller IP communicates with the Microblaze processor via the AXI4 lite bus using the AXI Interconnect module. For week 1, there are 601 32-bit registers (600 text registers and 1 control register), the AXI4 lite bus allow the Microblaze controller manipulate (read/write) these registers. The AXI4 lite bus implementation is detailed in **HDMI AXI register** section.

Internally, it uses few modules to manipulate these registers, and show corresponding text on the screen (HDMI signals). Clock wizard, VGA controller and VGA-to-HDMI are already used in Lab 6 so their importance will not be mentioned again. Most important modules are HDMI AXI and Color Mapper. This part is detailed in **Text Mode controller IP** section.

The VGA text mode has 80x30 text characters (640x480 screen, 8x16 pixel per character). There are 2400 characters and 600 text registers, so each 32-bit register will store 4 characters. There are also one 1 control register that stores the foreground color and background (They are 4-4-4 RGB values). For week 1, the entire screen has only 1 foreground color and background color.

Given that the current drawing pixel X and Y, the IP will first calculate the corresponding text coordinate TextX and TextY, and then find the character data in the text registers. Each character has 7 bit of information storing their index in the IP's internal font rom, and 1 invert bit to indicate whether the foreground and background for that character is swapped. Then the IP will determine the text offset TextOffsetX and TextOffsetY to determine the specific pixel in current character that needs to drawn. Then, using current text coordinate and text offset, one bit information will be retrieved from the Font ROM. Using the current foreground color and background color, 1 means drawing current pixel in foreground color, while 0 means drawing current pixel in background color, (if invert bit is 1, do this reversely). The calculation is detailed in **Drawing Algorithm** section.

This color information is feed into the VGA-to-HDMI module just like what we did in Lab 6. This module then generate HDMI signal and we get the monochrome text displayed on the screen.

(End Describing IP)

The monochrome text display software part is given to us. It is straightforward, write the data to the memory-mapped address, then the IP will have the correct register updated via its internal AXI4 lite bus implementation. The test function draw few different text with different background color. It's detailed in the **Software** section.

These are relevant modules inside the IP (also helpful when looking at the Block Diagram):

Previously used module

This is module that used Lab 6.2, they are not going to be explained in detail.

- Clock Wizard: generate 25MHz, 125MHz clock used by VGA controller and VGA-to-HDMI signal transformer.
- VGA controller: based on pixel clock (25MHz), generate
 - current pixel X and Y that should be drawn
 - active blank, horizontal sync and vertical sync that specified by VGA standard.
- VGA-to-HDMI:
 - given 25MHz and 125MHz clock from the clock wizard, Color from color mapper, and active blank & vsync from VGA standard.
 - return the actual HDMI signal output directly to the screen

Important Module

- HDMI AXI
 - responsible for AXI4 lite bus communication, store the data in its internal 601 registers.
 - given the selected register address (0~599), return the foreground color, background color, character and invert bit based on its stored registers. The specific algorithm is explained in later sections.
- Color mapper
 - given the foreground color, background color, character and invert bit from the HDMI AXI.
 - given the pixel X and Y from VGA controller.
 - return the color used for current pixel X and pixel Y. (this is found via the character input and the internal font rom (IBM ascii character set)). The specific algorithm is explained in later sections.

Accessing AXI registers

There are 2 operations, of course, for accessing AXI registers: reading and writing. These logic is largely given to us.

adapted from 15.1 Introduction to AXI4-lite interface and HDMI/VGA Graphics

The `ACLK` is the clock used for AXI4 bus. The `ARESETN` is the active-low reset signal. Notice the code shown here is slightly different from the provided file for simplicity.

Reading

- `ARADDR` is the read address (master)
- `ARVALID` indicates the read address is valid (master)
- `ARREADY` indicates slave accept the read address (slave)
- `RVALID` indicates the read data is valid (slave)
- `RREADY` indicates the master is ready to accept read data (master)
- `RDATA` is the read data (slave)

- `RRESP` is the response from slave (slave)

The entire process aims to get the `ARADDR` from the master, and return `RDATA` from the slave. (also `RRESP`, but it's just `OKAY` in this case).

Handshake

The Master raise the reset pin, initiate the communication process. The Master puts an address on the `ARADDR` as well as asserting `ARVALID`, indicating the address is valid. Slave asserts `ARREADY` in next cycle. (Check the condition `~ARREADY && ARVALID`). This assertion `AREADY` will only last for one cycle. The slave will latch `ARADDR`.

```
if (~ARREADY && ARVALID) begin    // check if master read address is valid
    ARREADY <= 1'b1;                // response ready
    raddr   <= ARADDR;              // latch read address
end else begin
    ARREADY <= 1'b0;                // the conditonal check tell us this only last
    for one cycle
end
```

The master should also de-assert `ARVALID` at next cycle after `ARREADY` is asserted.

Read Data

Slave will put the data on `RDATA`, assert `RVALID` indicating data in the channel is valid (Check logic `ARREADY && ARVALID && ~RVALID`). Slave will also put `RRESP` equal to 0 in this case to indicate a `OKAY` response. This happens exactly at the cycle when `ARREADY` and `ARVALID` de-asserted. The assertion `RVALID` will be on until the master assert `RREADY` (in which case `RVALID` will be off next cycle), indicating the master is ready to receive data from the slave.

```
if (ARREADY && ARVALID && ~RVALID) begin // RVALID will be asserted at next
cycle after the HANDSHAKE-complete cycle
    RVALID <= 1'b1;                      // assert RVALID
    RRESP  <= 2'b0;                      // 'OKAY' response
end else if (RVALID && RREADY) begin      // RVALID will be deasserted only after
master assert RREADY
    RVALID <= 1'b0;                      // deassert RVALID
end
```

Master should also de-assert `RREADY` at next cycle after `RVALID` is asserted.

Since at the cycle when `ARREADY` and `ARVALID` de-asserted (that is, the `RVALID` is asserted), the data should be available. That means the data should be prepared when `ARREADY` and `ARVALID` is both asserted (and therefore of course before the `RVALID` is asserted), so

```
if (ARREADY & ARVALID & ~RVALID) begin // RDATA will be loaded at next cycle
after the HANDSHAKE-complete cycle
    RDATA <= register[raddr[11:2]];      // trim the last two bits (byte-address
-> word-address)
end
```

The `RDATA` is always 32 bits. So, we just return the entire register. The `raddr` is in byte-address range (`0x000 - 0x960`), our register is 32-bit (i.e in word-address range), therefore we need to trim last two bits to get the corresponding word-address range. The Microblaze processor will figure out which byte from these 4 bytes by itself.

The master should retrieve the read data by now.

Writing

- `AWADDR` is the write address (master)
- `WSTRB` is the write strobe (master)
- `AWVALID` indicates the write address is valid (master)
- `WDATA` is the write data (master)
- `WVALID` indicates the write data is valid (master)
- `AWREADY` indicates the slave could accept the write address (slave)
- `AWVALID` indicates the slave could accept the write data (slave)
- `BVALID` indicates the the response is available (slave)
- `BREADY` indicates the the master could accept the response (master)
- `BRESP` is the response (slave)

Handshake

The Master raise the reset pin, initiate the communication process. The Master puts an address on the `AWADDR`, and the write data on the `WDATA`. It also assert `AWVALID` and `WVALID` to indicate that the address and write data on the corresponding channel is ready. Slave will assert `AWREADY` and `WREADY` on the next cycle, indicating it's ready to receive both the address and the data. The assertion for `AWREADY` and `WREADY` will only last for one cycle. The slave will latch `AWADDR`.

```
if (
    (~WREADY || ~AWREADY) &&
    WVALID && AWVALID           // check if the master write address and data is
    valid
) begin
    WREADY  <= 1'b1;
    AWREADY <= 1'b1;
    waddr   <= AWADDR;          // latch the write address
end else begin                         // the conditonal check tell us this only last
for one cycle
    WREADY  <= 1'b0;
    AWREADY <= 1'b0;
end
```

The master should also de-assert `WVALID` and `AWVALID` at next cycle after `WREADY` and `AWREADY` is asserted.

Write Data

Slave will write the data `WDATA` into `registers`, assert `BVALID` indicating a response in the channel is available. Slave will also put `BRESP` equal to 0 in this case to indicate a `OKAY` response. This happens exactly at the cycle when `AWREADY`, `AWVALID`, `WREADY`, `WVALID` is de-asserted. The assertion `BVALID` will be on until the master assert `BREADY` (in which case `BVALID` will be off next cycle), indicating the master is ready to receive the response from the slave.

```
if (~BVALID && AWREADY && WREADY && WVALID) begin // BVALID will be asserted at next cycle  
    after the HANDSHAKE-complete cycle  
    BVALID <= 1'b1; // There is a response  
    BRESP <= 2'b0; // 'OKAY' response  
end else (BREADY && BVALID) begin // BVALID will be deasserted only after  
    master assert BREADY  
    BVALID <= 1'b0; // deassert BVALID  
end
```

Master should also de-assert `BREADY` at next cycle after `BVALID` is asserted.

The Slave should write the data at the same cycle it tries to send a response, therefore if the parameter is invalid it could send an invalid response (although in this case it always return a `OKAY`). Therefore,

```
if (AWREADY && AWVALID && WREADY && WVALID) begin // Data will be write at  
    next cycle after the HANDSHAKE-complete cycle  
    for (integer index = 0; index < 4; index++)  
        if (WSTRB[index] == 1) begin // write to the byte where  
            write strobe is present  
            register[waddr[11:2]][(index*8) +: 8] <= WDATA[(index*8) +: 8];  
        end  
    end
```

Writing data is not as simple as reading data. The `WSTRB` given by Master indicates which byte in the 4 byte address it's actually writing to. (Most significant bit corresponds to most significant byte, etc.). We check if the corresponding `WSTRB` bit is on, and copy the data from `WDATA` to corresponding register (Again, the `waddr` is in byte-address range (`0x000 - 0x960`), our register is 32-bit (i.e in word-address range), therefore we need to trim last two bits to get the corresponding word-address range). The syntax `[(index*8) +: 8]` used here just mean `[(index * 8) + 7 : index * 8]`, this notation just make our code nicer.

The Slave therefore successfully write data to the correct register.

Drawing Algorithm

From the VGA controller, we have two initial value X and Y (the coordinate of current drawing pixel). The X and Y is stored in a 10-bit logic (X from 0 - 799, Y from 0 to 524, the blanking period is added so it's bigger than standard 640x480).

We first need to calculate the **Text Address** (from 0-2399), we need a 12-bit address value here ($2^{12} = 4096$). Since there 80x30 characters, and each character has a size of 8x16 pixels, we know that **Text Address X = X / 8**, **Text Address Y = Y / 16** (it's integer division here). Then the **Text Address = (Text Address Y x 80) + Text Address X**. Since the division is applied on 2's exponentials, it could be transformed into the System Verilog expression just shown below. Note few padding to make sure the operand has same bit width.

```
// mathematical expression
Text Address = ((Y / 16) * 80 + X / 8)
// verilog expression
assign Text_Address = ({6'b0, Y[9:4]} * 12'd80) + {5'b0, X[9:3]};
// Y[9:4] <-> Y/16; Remove X[9:3] <-> X/8
```

We mentioned one register stores 4 characters, and the register layout looks like

31	30-24	23	22-16	15	14-8	7	6-0
Invert Bit 3	Code 3	Invert Bit 2	Code 2	Invert Bit 1	Code 1	Invert Bit 0	Code 0

where the code is the character index in the Font ROM (From IBM Codepage 437). Notice, the drawing order is Code 0, Code 1, Code 2, Code 3 (from relative left to relative right on the screen).

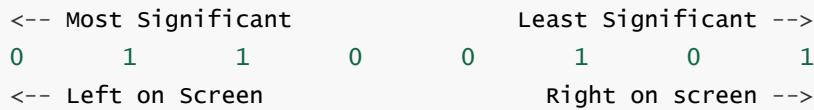
Given **Text Address**, we want to find the corresponding **Character Index** and **Invert Bit**. We need to first find the register where the character is in, and this is easily done by **Text Address / 4** (since they are grouped into 4) then we need to find, which slot it's in. From the drawing order we observe, **Character Index** should be from **(Text Address mod 4) x 8** to **(Text Address mod 4) x 8 + 6** (inclusive), and **Invert Bit** should be **(Text Address mod 4) x 8 + 7**. Luckily, since the division and modulo is applied on 2's exponentials, it could be transformed into some System Verilog expression just shown below.

```
// mathematical expression
in register[Text Address / 4],
    Character_Index = from (Text Address mod 4) * 8 to (Text Address mod 4) * 8 +
6
    Invert_bit = (Text Address mod 4) * 8 + 7

// verilog expression
assign Character_Index = register[Text_Address[11:2]][Text_Address[1:0]] * 8 +:
7;
assign Invert           = register[Text_Address[11:2]][Text_Address[1:0]] * 8 +:
7;
// Text_Address[11:2] <-> Text_Address / 4
// Text_Address[1 :0] <-> Text_Address mod 4
// [a +: 7] <-> [a + 6: a]
```

Next we need to find where the character is in the Font ROM, the Font ROM data is stored in 8-bit group, each 8-bit represent one row for that character, and therefore each character has consecutive row in it. So there are $16 \times 128 = 2048 = 2^{11}$ rows in the rom, and therefore 11 bit to address it. To know the base row (the row 0) for one character, it's obviously **Character Index x 16**, to know the row offset, it therefore **Y mod 16**. (each character start at **Y mod 16 = 0** row, so its offset could be determined this way). After we get the row, we need to get the column index within

that one row. Here is the tricky part, the ROM stores the left most pixel data on its most significant bit. Therefore, the column index is actually $7 - (\text{X} \bmod 8)$ (We need to reverse the order, so when $\text{X} \bmod 8$ is small (left), most significant bit is used. Vice versa)



Since we are dealing with 2's exponentials, same trick could be used. Note few padding to make sure the operand has same bit width.

```
// mathematical expression
Font Row    = (Character Index * 16) + (Y mod 16)
Font Column = 7 - (X mod 8)

// verilog expression
assign Font_Row    = 11'd16 * {4'b0, Character_Index} + {7'b0, Y[3:0]}
assign Font_Column = 7 - DrawX[2:0]

// pixel bit
font_rom rom (
    .addr(Font_Row),
    .data(row)
);

assign pixel_bit = row[Font_Column];
```

Now, we get (**Invert Bit, Pixel Bit**), with the background and foreground we get in next section, we could calculate the final color in our color mapper. (next section)

Invert color bit & Control Register

Control register is stored just in the 601th register in the previously mentioned color group. This is according to the Memory Map specified.

Word Address Range	Byte Address Range	Description
0x000 - 0x257	0x0000 0000 - 0x0000 095F	VRAM – 1 character per byte, 4 characters per word. 80 column x 30 row. Data format is in raster order (one line at a time).
0x258	0x0000 0960	Control register
0x259 - 0xFFFF	0x0000 0961 - 0x0000 FFFF	Unused but reserved by Vivado

This is how foreground color and background color is got:

```
assign ForegroundRed  = register[600][24:21];
assign ForegroundGreen = register[600][20:17];
assign ForegroundBlue = register[600][16:13];
assign BackgroundRed   = register[600][12:9];
assign BackgroundGreen = register[600][8:5];
assign BackgroundBlue   = register[600][4:1];
```

31- 25	24-21	20-17	16-13	12-9	8-5	4-1	0
/	Foreground Red	Foreground Green	Foreground Blue	Background Red	Background Green	Background Blue	/

The invert bit is calculate in the previous section. We now get **Invert Bit**, **Pixel Bit**, **Foreground Color** and **Background Color**. The final step is to generate the color for this pixel in color mapper. The **Pixel Bit = 1** means we should use foreground color, the **Pixel Bit = 0** means we should use the background color. If the **Invert Bit** is present, we do it reversely. This logic could be implemented as simple as **Pixel Bit ^ Invert Bit**, a simple XOR operation. Then, set output color to foreground color if this is true, and to background color otherwise.

```

always_comb begin: Font_Rom
    if (X >= 640 || Y >= 480) begin
        is_on = 0;
        is_foreground = 0;
    end else begin
        is_on = 1;
        is_foreground = Pixel_Bit ^ Invert_Bit // XOR to check if foreground
should be used
    end
end
always_comb begin:RGB_Display
    if (is_on && is_foreground) begin
        Red   = ForegroundRed;
        Green = ForegroundGreen;
        Blue  = ForegroundBlue;
    end
    else if (is_on && ~is_foreground) begin
        Red   = BackgroundRed;
        Green = BackgroundGreen;
        Blue  = BackgroundBlue;
    end else begin // use default gradient color
(although this will never been seen )
        Red = 4'hf - DrawX[9:6];
        Green = 4'hf - DrawX[9:6];
        Blue = 4'hf - DrawX[9:6];
    end
end

```

This is roughly how the color mapper works, it just adds few boundary checks to above explained logic.

Software

Week 1's software logic is relatively simple. There will be a magic address `XPAR_HDMI_TEXT_CONTROLLER_0_AXI_BASEADDR` prepared by Vitis header files for the starting address of the VRAM. Using the struct like this

```

struct HDMI_TEXT_STRUCT {
    uint8_t VRAM [ROWS*COLUMNS]; // ROWS = 30, COLUMNS = 80
    uint32_t CTRL;
};
```

allow us to access the VRAM directly. We could then write the character code into the VRAM, and change the `CTRL` register according to specification given above. We could just define a 4-4-4 RGB value in a `int`, and then doing

```
hdmi_ctrl->CTRL = foreground << 13 | background << 1;
```

will move the color to the correct place in register.

There is not much interesting thing going on in software part for Week 1.

Color Text Display

BRAM

Week 2 requires 1208 registers. (1200 registers, 1 register for 2 character, 8 control register). In this case it's almost impossible to use register-based VRAM. A Block RAM is used. Although the Lecture recommended to use Simple Dual Port RAM, I found out True Dual port RAM are more easier to work with.

Specifically, I created a 32-bit width, 1208 depth True Dual Port RAM. The reason for using a True Dual Port RAM is that the AXI bus requires us to read and write data to the BRAM, while the HDMI drawing routine requires us to read from RAM at a constant rate. If we use a Simple Dual Port RAM, there will be only one read and one write port available, and if we want to read the pixel data that AXI4 bus requires and a different pixel data that HDMI drawing routine requires at same cycle, it will complicate the design. (We might need to put the AXI4 bus to temporary wait, and that requires a state machine). On the other hand, we could read and write to True Dual Port RAM using its PORT A for AXI4 bus, and read from its PORT B for HDMI drawing routine.

One concern is that what happens when Port A is writing and Port B is reading. This at worst will cause current writing not be updated to screen until next frame according to documentation, and we are not concerned about the update latency (it's only one frame ~1/60 s). So there is no special prevention for this.

Furthermore, to make life easier, we changed Port B read / write width to 16, so won't need to manually choose whether we want the top half or bottom half (16 character represent a character here). So the Port A has an address of 11 bits, and Port B has an address of 12 bits. I also turn off the *primitive output register*, so that BRAM will only have one clock latency. I turn the ECC off, and enable byte write, so we could use the write strobe we receive from the AXI4 bus.

Also, notice that the original documentation said

Word Address Range	Byte Address Range	Description
0x000 - 0x4AF	0x0000 0000 - 0x0000 12BF	VRAM – 2 bytes per character, 2 characters per word. 80 column x 30 row. Data format is in raster order (one line at a time).
0x4B0 - 0x7FF	0x0000 12C0 - 0x0000 1FFF	Unused but reserved by Vivado
0x800 - 0x807	0x0000 2000 - 0x0000 201F	Palette- 8 words of 2 colors each, for 16-color palette
0x808 - 0x3FFF	0x0000 2020 - 0x0000 FFFF	Unused but reserved by Vivado

We will waste a lot of space on our BRAM (0x4B0 - 0x7FF) if we don't do memory map, we could just memory map the register (2048 - 2055) to the memory (1200 - 1207). This makes sure we only use 1208 registers, the minimal number of register that we have to use.

IP Modification

I added the BRAM block to the IP, and then changed the HDMI AXI module. BRAM is detailed in previous section. Modification to HDMI AXI module is detailed in **Additional modification** section.

Modified sprite drawing algorithm

The logic for calculating `Text_Address` doesn't change at all. But now each character is stored in 2 bytes (16 bits), we need a little bit different way of getting it. According to the specification:

31	30-24	23-20	19-16	15	14-8	7-4	3-0
Invert Bit 3	Code 1	Foreground Code 1	Background Code 1	Invert Bit 0	Code 0	Foreground Code 0	Background Code 0

Do we need to do something like `bram_dout[Text_Address[11:1]][(Text_Address[0] * 16) +: 16]`? Good news is that we are using 16 bits read width, so every time the BRAM will automatically give us the correct 16 bits character data. We just need to do

```
b1k_mem_gen_0 mem (
    // PORT A (omitted)

    // PORT B
    .addrb(Text_Address),
    .c1kb(ACLK),
    .doutb({ Invert, Character, ForegroundColor, BackgroundColor }),
    .enb(1'b1),
    .web(1'b0)
);
// If we used 32 bit width, it will look something like
// assign Invert      = mem_doutb[Text_Address[11:1]][(Text_Address[0] * 16
) + 15]
// assign Character   = mem_doutb[Text_Address[11:1]][(Text_Address[0] * 16 +
8) +: 4]
// assign ForegroundColor = mem_doutb[Text_Address[11:1]][(Text_Address[0] * 16 +
4) +: 4]
```

```
// assign BackgroundCode = mem_doutb[Text_Address[11:1]][(Text_Address[0] * 16
) +: 4]
// so, really, rely on the vivado's builtin ability, so you don't need do this by
yourself.
```

That looks nice. We then just split the 16 bits character data into **Invert Bit**, **Character**, and **Foreground Code** and **Background Code**. We then just need to transfer the foreground code and background code into corresponding color. We could, technically, ask for the BRAM for another read to get the palette. The BRAM operates at 100 MHz, while the pixel clock operates at 25 MHz, so we could get the needed foreground color and background color within one pixel clock. However, this also requires a state machine, since we need to keep track of which cycle we read text character, and we which cycle we read palette. **A easy solution is to just store a copy of palette in the register**, it's just has the size of 8x32 bytes. In this case, we could just retrieve the color from the palette register we store in the registers:

```
always_comb begin
    if (ForegroundCode[0] == 1'b0) begin
        ForegroundBlue = palette[ForegroundCode[3:1]][4:1];
        ForegroundGreen = palette[ForegroundCode[3:1]][8:5];
        ForegroundRed = palette[ForegroundCode[3:1]][12:9];
    end else begin
        ForegroundBlue = palette[ForegroundCode[3:1]][16:13];
        ForegroundGreen = palette[ForegroundCode[3:1]][20:17];
        ForegroundRed = palette[ForegroundCode[3:1]][24:21];
    end

    if (BackgroundCode[0] == 1'b0) begin
        BackgroundBlue = palette[BackgroundCode[3:1]][4:1];
        BackgroundGreen = palette[BackgroundCode[3:1]][8:5];
        BackgroundRed = palette[BackgroundCode[3:1]][12:9];
    end else begin
        BackgroundBlue = palette[BackgroundCode[3:1]][16:13];
        BackgroundGreen = palette[BackgroundCode[3:1]][20:17];
        BackgroundRed = palette[BackgroundCode[3:1]][24:21];
    end
end
```

This just follows the specification:

31-25	24-21	20-17	16-13	12-9	8-5	4-1	0
/	C1 Red	C1 Green	C1 Blue	C0 Red	C0 Green	C0 Blue	/

Therefore we get the **Invert Bit**, **Character**, and **Foreground Color** and **Background Color**, and this passed to color mapper. The remaining steps works the same way.

Additional modification

We use the block ram in this way:

```
blk_mem_gen_0 mem (
    // PORT A
    .addr(mem_addr),           // read / write memory address
    .clka(ACLK),              // clock pin
    .dina(WDATA),             // pass WDATA (write data) to in pin
    .douta(RDATA),             // pass RDATA (read data) to out pin
    .ena(mem_rden | mem_wren), // enable memory when we do either reading
    or writing
    .wea(WSTRB & {4{ mem_wren }}), // disable write if mem_wren is 0 (WSTRB is
    ignored in this case), WSTRB will be passed in
                                // if we enabled the write, since the most
    significant WSTRB bit controls write of most
    // significant byte of the data, we can just pass this as is.
    // PORT B (Omitted)
);
```

This actually looks nicer when writing the data, since the BRAM does this for us, and we don't need to play the index calculation game.

I have mentioned that there is a memory map from address (2048 - 2055) to (1200 - 1207), this is done by following piece of code:

```
always_comb begin
    if (should_write) begin
        if (waddr[13:2] >= 2048 && waddr[13:2] <= 2055) begin      // memory map
palette
        mem_addr = { 8'b10010110, waddr[4:2] };
        mem_wren = 1;
        mem_rden = 0;
    end else if (waddr[13:2] < 2048) begin                      // no memory map
                                                                // smaller than
2048, high bit will always bit 0, trim it
        mem_addr = waddr[12:2];
        mem_wren = 1;
        mem_rden = 0;
    end else begin                                              // out of boundary,
do nothing
        mem_addr = 0;
        mem_wren = 0;
        mem_rden = 0;
    end
end else if (should_read) begin
    if (raddr[13:2] >= 2048 && raddr[13:2] <= 2055) begin      // memory map
palette
        mem_addr = { 8'b10010110, raddr[4:2] };
        mem_wren = 0;
        mem_rden = 1;
    end else if (raddr[13:2] < 2048) begin                      // no memory map
                                                                // smaller than
2048, high bit will always bit 0, trim it
        mem_addr = raddr[12:2];
        mem_wren = 0;
        mem_rden = 1;
```

```

        end else begin // out of boundary,
do nothing
    mem_addr = 0;
    mem_wren = 0;
    mem_rden = 0;
end
end else begin // idle, do nothing
    mem_addr = 0;
    mem_wren = 0;
    mem_rden = 0;
end
end

```

This memory could just be done by `{8'b10010110, addr[4:2]}`. Since the address is byte-address range and we need word-address range, we trim last 2 bits. Then the magic number is actually just `1200 >> 3`. we find that `addr[13:2]`, when it's equal to `2048-2055`, the `addr[4:2]` is just `0-7`, combine this with the magic number, we just memory map from `2048-2055` to `1200-1207` using this simple formula.

Also, we have mentioned that we copied our palette also in the registers, this is easily done by

```

if (should_write & (waddr[13:2] >= 2048 && waddr[13:2] <= 2055) begin
    for (integer i = 0; i < 4; i++) begin
        if (WSTRB[i])
            palette[waddr[4:2]][i*8 +: 8] <= WDATA[i*8 +: 8];
    end
end

```

This is similar to what we done for all address in week 1.

One concern is that the BRAM actually has 1 cycle delay. In this case do we have to use a state machine to hold the AXI4 bus on wait? We could perform a trick. You see that the in week 1 code, although reading from our `register` is a combinational logic, the `RDATA` needs 1 cycle to be updated (sequential logic). Now, we just remove this sequential logic register `RDATA`, and hook this wire directly to the output of BRAM. Although the BRAM has 1 cycle delay, but since we remove the sequential logic `RDATA`, they cancel each other out. The disadvantage is that we could not precisely control what we want to output. Previously there won't be any `RDATA` output if we are doing a Write operation. But now `RDATA` will have some data that we just write into if we are doing Write operation (according to the Xilinx docs). This actually doesn't matter, as the AXI4 doesn't care what is in read bus if it's doing writing.

Modified Software

We have the same magic address `XPAR_HDMI_TEXT_CONTROLLER_0_AXI_BASEADDR` that points to the base of the `VRAM`, in this case we just need to declare a struct like

```

struct TEXT_HDMI_STRUCT {
    uint8_t          VRAM [ROWS*COLUMNS*2];
};

```

and we modified the `x, y` text character by

```

hdmi_ctrl->VRAM[(y*COLUMNS + x + i) * 2] = foreground << 4 | background;
hdmi_ctrl->VRAM[(y*COLUMNS + x + i) * 2 + 1] = c;

```

as detailed in the specification.

We could change the palette in this case by

```

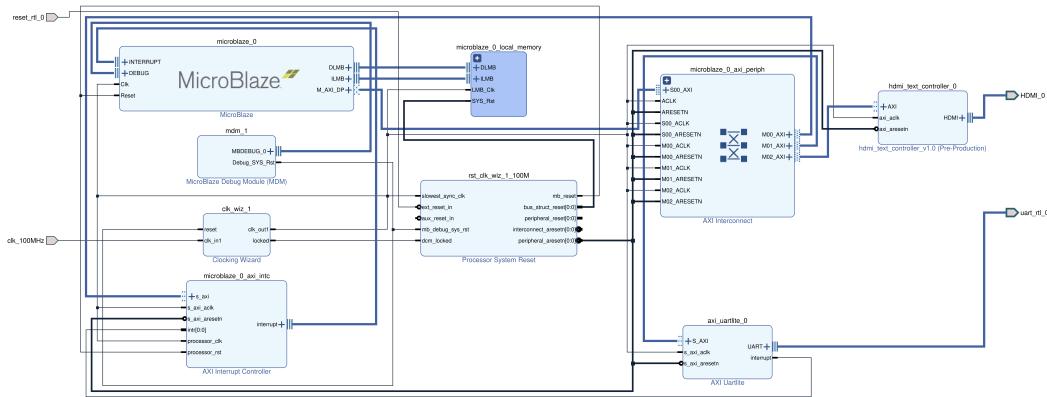
if (color % 2 == 0) {
    // remove [12:1] bits first
    hdmi_palette->PALETTE[color/2] &= (~0x1FFE);
    // set [12:1] bits
    hdmi_palette->PALETTE[color/2] |= (red & 0x0F) << 9 | (green & 0x0F) << 5 |
    (blue & 0x0F) << 1;
} else {
    // remove [24:13] bits first
    hdmi_palette->PALETTE[color/2] &= (~0x1FFE000);
    // set [24:13] bits
    hdmi_palette->PALETTE[color/2] |= (red & 0x0F) << 21 | (green & 0x0F) << 17 |
    (blue & 0x0F) << 13;
}

```

We first clear the corresponding bit in the palette field by using a NOT-AND bitmask. Then left shift the R/G/B we want to the desired place as the specification said (mentioned in previous section). It's pretty straight forward.

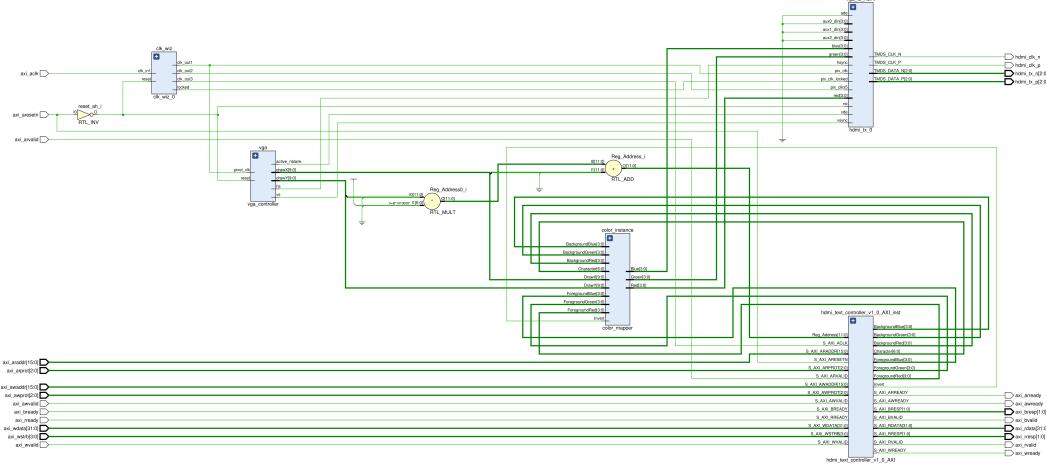
Block Diagram

Top Level Block Diagram

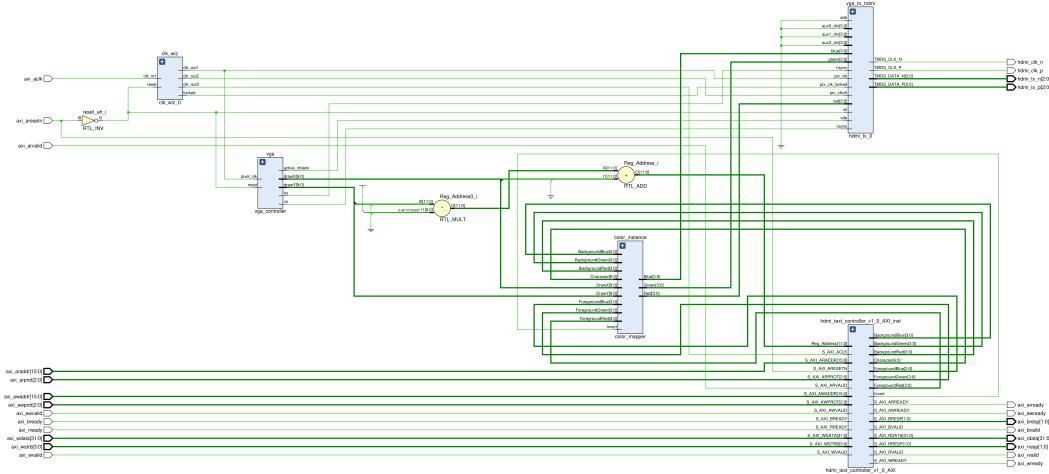


IP Diagram

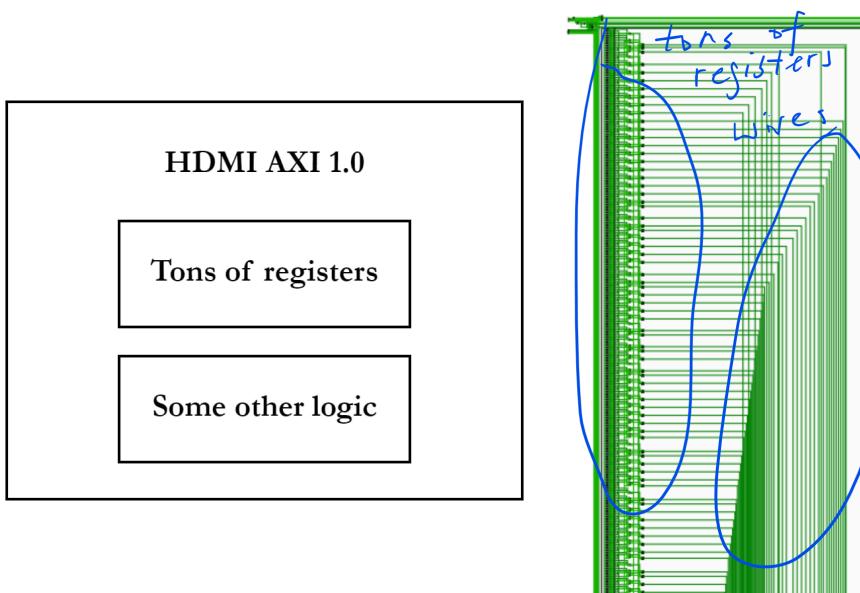
Week 1

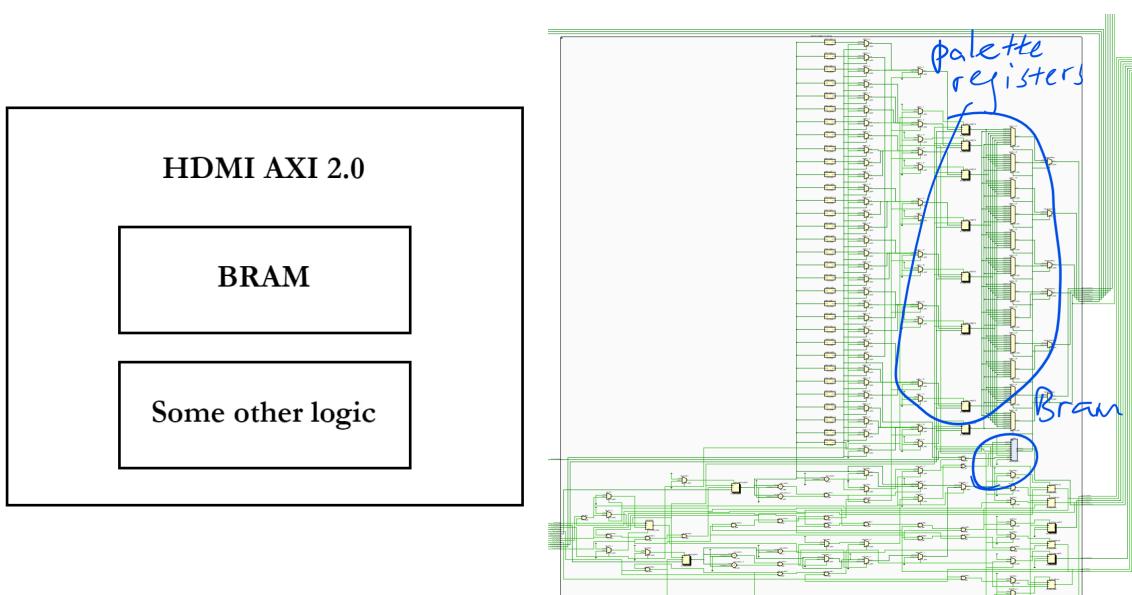


Week 2



You may find that they looks similar. Yes, just from the top-level view, the IP diagram looks similar. That because the register / BRAM is inside the `hdmi_text_controller_v1_0_AXI_inst`, and their internal looks very differently. It actually looks like this:





If you really want to peak at what is actually inside of this, there are expanded schematic pdf attached.

Module Description

See separate Lab 7 Module pdf.

Design Resources and Statistics

Week 1

LUT	15657
DSP	3
Memory (BRAM)	32
Flip-Flop	21127
Latches*	0
Frequency (MHz)	73.98
Static Power (W)	0.074
Dynamic Power (W)	0.368
Total Power (W)	0.442

Week 2

LUT	2187
DSP	3
Memory (BRAM)	34
Flip-Flop	1704
Latches*	0
Frequency (MHz)	101.48
Static Power (W)	0.074
Dynamic Power (W)	0.375
Total Power (W)	0.449

Overall, the week 2's design is much more efficient, we use much less LUT and FF, and in turn achieves higher clock speed. We also avoid potential undefined behavior by having a negative WNS in week 1. However, we do indeed use 2 more BRAM in replace for the Week 1 register design. The energy consumption, though, at least according to the Vivado tools, stays similar.

Conclusion

One potential extension is that I could make this an actual VGA module that supports different text mode and graphic mode. I learnt how VGA works, and this helps a lot with implementing PPU in the NES system (which also works in a similar way).