ECE385

Fall 2023

Experiment #7

Lab 7

Haoyu Huang Eitan Tse

Lab section: ZY 10/Day & Time: 11.13

Your TA's Name: Yang Zhou

1. Introduction

HDMI interface is used to assign every pixel a specific RGB value. The design aims to create an interface that allows for the conversion of VGA signals to HDMI signals, which is a more modern and widely used standard for high-definition multimedia transmission. We want this design to implement pixel drawing as what we do in lab6.

The module in HDMI IP is very similar to lab6.2. We need clk_wiz_0 to generate different clock signals. We used a vga_controller to produces VGA signals, including horizontal sync (hsync), vertical sync (vsync), and other control signals (drawX, drawY). And hdmi_tx_0 module is used to converts VGA into HDMI signals. The only different is that we have hdmi_text_controller_v1_0_AXI. It manages AXI transactions for reading or writing text-related data and likely controls the positioning and coloring of text on the HDMI display.

In Lab 6.2, we had a different method for hardware->software communication (e.g., the keycode). Describe some advantages/disadvantages of the IP approach in Lab 7 compared to the approach in Lab 6.2

Packed IP blocks are designed for reuse. Once you have a verified and functional IP block, you can reuse it in multiple projects, promoting consistency and reducing redundant work. We can also update the version of IP which can backup the previous one and can help us make some significant change. Packed IP blocks save time in design and verification. They have been tested and we can use it directly instead of creating a large block and having to change significant things every time we make an update to the block.

We used the AXI bus to read and write data in this lab, while in lab6.2 we used the SPI interface to get the keycodes from the MAX3421E. SPI is catered to interfacing with external devices. Note that we used an externally mapped interface in lab 6.2 to talk to the MAX3421E. In contrast, we implement an IP that can directly interface with the CPU in this lab. AXI is intended more for direct interfacing between modules on the chip, which we can see as our custom IP is hooked directly up to the AXI peripheral interface bus. We use the AXI4-Lite bus, which lacks the burst and massive, rapid data transfer capabilities a full AXI interface (or the AXI-Stream) would have. Interfacing through SPI is slower, because we have software set up the address to write to, configure the data to send, so on and so forth. AXI is a direct hardware connection, rendering it much faster- no need for a slower software layer to interface.

2. Written Description of Lab 7 System

Week 1 (Monochrome Text Display)

i. Written Description of the entire Lab 7 system.

In lab7, we try to create our own package IP to draw sprite on the screen. We use the same clock wizard, vga controller and hdmi_tx module as lab6 to assign pixel with specific RGB color. We also have AXI module which can interact with AXI bus, VRAM and font_ROM. We also implement drawing logic inside AXI module. Lab 7 system can read or write data to VRAM with software and AXI module. We have a function in the software which can write specific text or spirit to the screen and set its color. In summary, Lab 7 system has its own palette and we can draw specific pictures and set colors by software.

ii. Describe at a high level your HDMI Text Mode controller IP

The HDMI Text Mode controller uses an AXI4-Lite interface to allow text to be written to a screen, controlled by the CPU. This interface is memory-mapped, so the CPU can write to specific memory locations to control the IP. It outputs signals that drive the HDMI port, using a VGA-to-HDMI converter IP within it to generate these signals. Within it, we use 601 registers to map each character on-screen to its requisite pixels. Using the data within each register, we fetch color and font_data to draw and color the text on screen, with the possibility to "invert" it by swapping the background and foreground colors. Lab 7.1 does not have the ability to set background and foreground colors per character. The background and foreground is set for the entire screen using a singular 32-bit control register.

iii. Describe the logic used to read and write your HDMI AXI registers.

To read and write from the AXI registers, we use the axi_awaddr and axi_araddr signals to get the appropriate register and write to or read from it. To enable reading, we wait for the arready and S_AXI_ARVALID signals to go high and the axi_rvalid signal to be low so we know the address on the bus is valid and we are not currently reading data. Then, we read from the specified register and put that data on the bus and assert RVALID to signal the master that the data is ready to be read. To enable writing, we do wait for the S_AXI_WVALID and S_AXI_WVALID signals to go high so we know that the address and data on bus are valid. Then, if we are ready we assert AWREADY and WREADY to fetch data from the bus and write the data to the specified memory location. This is how the AXI-4 LITE interface works between the master and slave to send data back and forth.

iv. Describe the algorithm used to draw the text characters from the VRAM and font ROM (specifically, describe the equations required to generate the correct addresses to index into the VRAM as well as the font ROM).

Our screen is 640 * 480. Suppose we want to draw pixel in (x,y). Since every address will store 32bit (4byte). We first use equation **column = x / 32** to derive column with respect to 4 characters. (Here we consider 4 characters as a column). Since every character have 16-bit height. We use **row = y / 16** to derive row. Then we consider our address 4 (byte) * address = row * 20 * 4 (byte) + column * 4(byte).

column = x / 32 offset_x= Drawx % 32
row = y / 16 offset_y= Drawy % 16
address = row * 20 + column

Since we have 4 characters in one address. We also have equation **offset_x= Drawx % 32** to get the correct location. Besides **offset_y= Drawy % 16.** It is used to locate which line in fon_address. Once we derive the address, we can access this address in our VRAM. Then we can get our code for character. We use equation **fon_address = code[n] * 16 + offsety** to get the correct address for character data. Besides, fon_address is mirrored which means we need to adjust our offset_x with equation (offset = 7 – offset_x)

v. Describe your implementation of the inverse color bit, as well as the implementation of the control register.

In lab7 week1, we set 601 registers as our VRAM and set the last register as control register. We store in inverse bit in previous 600 registers with character code. From the diagram, bit 31, 23, 15, 7 are used to store inverse bit. And we store the color of background and foreground in register 600. If the inverse bit is high, we will select background color as our foreground. In other words, if we want to draw pixel in the current location, we will choose between FGD and BKG according to inverse bit.

Control register consisting of 32-bit which store RGB value of BKG and FGD. We will assign our RGB value to it.

Bit	**31**	**30-24**	**23**	**22-16**	**15**	**14-8**	**7**	**6-0**
Function	IV3	CODE3	IV2	CODE2	IV1	CODE1	IV0	CODE0

Bit	**31-25**	**24-21**	**20-17**	**16-13**	**12-9**	**8-5**	**4-1**	**0**
Function	UNUSED	FGD_R	FGD_G	FGD_B	BKG_R	BKG_G	BKG_B	UNUSED

Week 2 (Color Text Display)

1.Describe the hardware changes you had to make to support the use of multi-color text. At the minimum you must describe: 7.4 1. Modification of register-based VRAM to on-chip memory-based VRAM. How did your design share the limited on-chip memory ports?

To support the use of multi-color text, we moved the design from registers to BRAM so we had the resources to support all 1200 32-bit registers needed, doubled from the 600 needed in lab 7.1. We used true dual-port RAM, though the second port was used only for reading and processing data to generate the video output. The first port was used to read and write via the AXI interface. The control register from 7;.1 has been switched to a bank of 8 registers encoding 16 different foreground and background colors for lab 7.2.

2. Corresponding modifications to the IP Editor.

Modifications were made to instantiate the block memory using Vivado's provided block memory generation tool. An additional eight registers were instantiated to serve as the palette. The AXI interface was also tweaked, with a counter used to add two cycles of delay before asserting RVALID to allow for the latency in reading from our block memory and prevent incorrect data from being sent out. In addition, significant logic changes were made to allow selection between writing to palette and the on-chip memory, reading the appropriate background and foreground RGB values, and fetching the appropriate register to get data to draw the character correctly.

3. Modified sprite drawing algorithm with the updated indexing equations from on-screen pixels to VRAM.

In lab7.2, we only have two byte every address. The logic of computing address of VRAM is quite similar to week 1.

Logic expression:

column = DrawX / 16 offset_x = DrawX % 16

row = DrawY / 16 offset_y = DrawY % 16

address = row * 40 + column -> (address * 2 = row * 80 + 2 *column)

Since we have two characters in every address, we divide x by 16 to get the column index.

Once we derive the address of VRAM, we can get the fon data which use same logic as week 1 including inverse offset to adjust the mirrored fon data.

Another difference is that we no longer have control register but 8 addition register which store the colors. We have to get index of background and foreground color from fon data. Then we can get the value of color by checking inverse bit and accessing color registers.

Logic for color register:

index_register = (fgd) bkg_index / 2

color_data = (fgd)bkg_index % 2

Bit	**31**	**30-24**	**23-20**	**19-16**	**15**	**14-8**	**7-4**	**3-0**
Function	IV1	CODE1	FGD_IDX1	BKG_IDX1	IV0	CODE0	FGD_IDX0	BKG_IDX0

Address	**31-25**	**24-21**	**20-17**	**16-13**	**12-9**	**8-5**	**4-1**	**0**
0x800	UNUSED	C1_R	C1_G	C1_B	C0_R	C0_G	C0_B	UNUSED
0x801	UNUSED	C3_R	C3_G_	C3_B	C2_R	C2_G	C2_B	UNUSED
…	…	…	…	…	…	…	…	…
0x807	UNUSED	C15_R	C15_G	C15_B	C14_R	C14_G	C14_B	UNUSED

4. Any additional modifications which were necessary to support multicolored text.

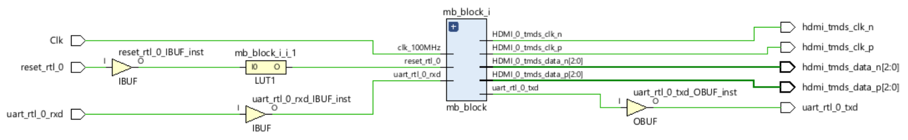
As mentioned in Q1, we have modified the VRAM by replacing it with OCM. We design logic to read and write to OCM and 8 additional registers to hold the palette colors. The AXI interface was also tweaked, with a counter used to add two cycles of delay before asserting RVALID to allow for the latency in reading from our block memory and prevent incorrect data from being sent out.

5. Additional hardware/code to draw palette colors.

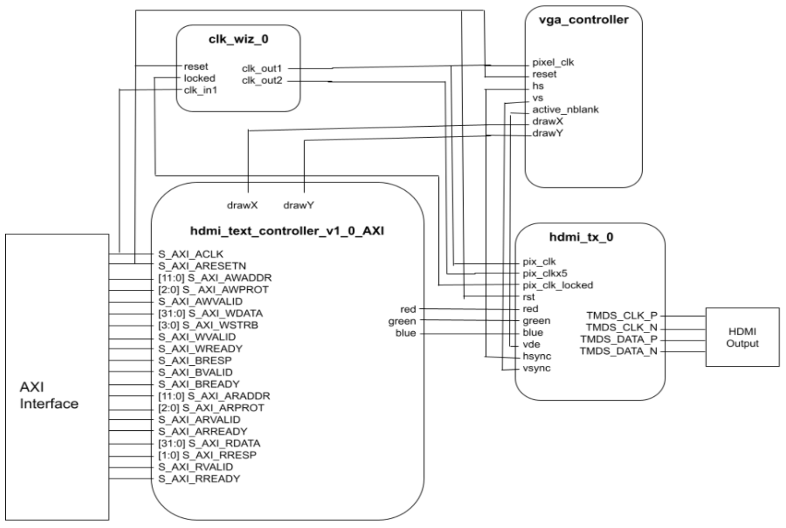
Eight additional registers were instantiated to serve as the palette. Changes were also made to the C code to write the palette colors correctly, namely bitshifting in the 4-bit RGB values to the appropriate memory address. We also accounted for the palette register's memory offset by inserting 3392 bytes between the end of the 1200 VRAM registers and the start of the palette addresses.

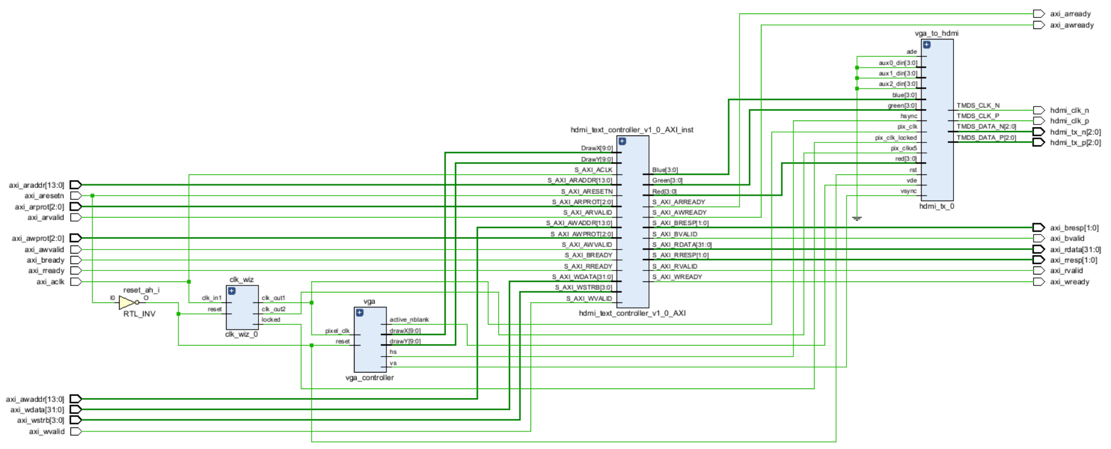
3 Block diagram

Toplevel:



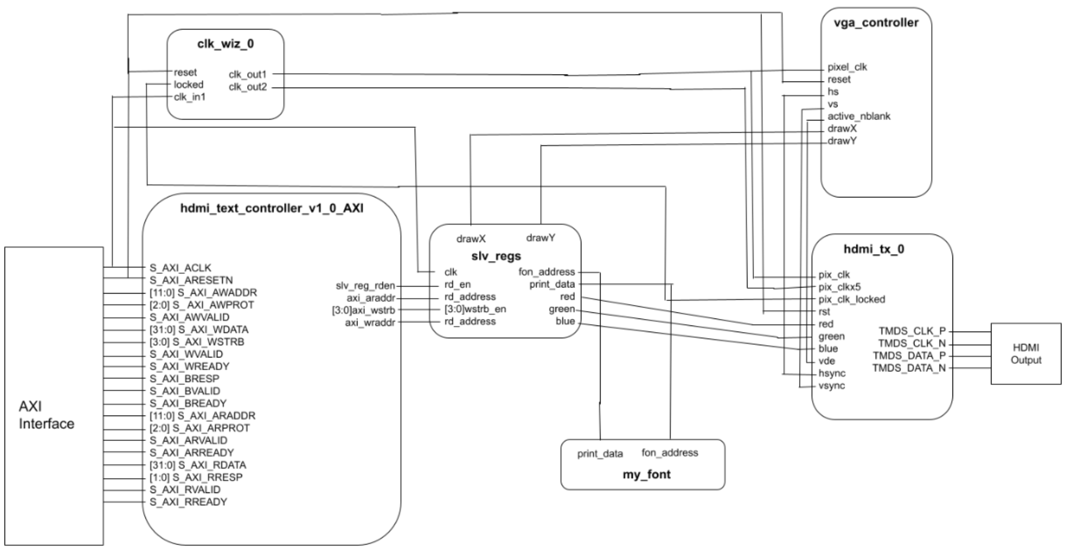
IP SoC setup





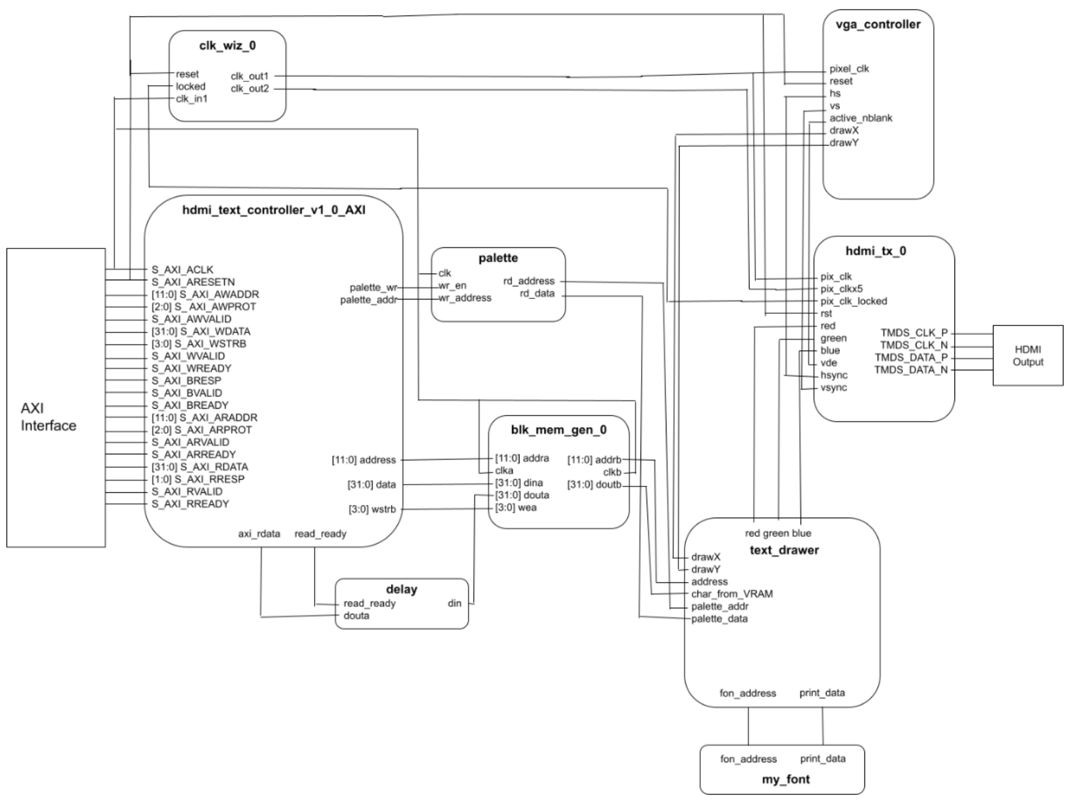
axi_araddr[13:0]
axi_aresetn
axi_arprot[2:0]
axi_arvalid
axi_awprot[2:0]
axi_awvalid
axi_bready
axi_rready
axi_aclk
reset_ah_i
RTL_INV
clk_wiz
clk_in1
reset
clk_out1
clk_out2
locked
clk_wiz_0
vga
pixel_clk
reset
active_nblank
drawX[9:0]
drawY[9:0]
hs
vs
vga_controller
hdmi_text_controller_v1_0_AXI_inst
DrawX[9:0]
DrawY[9:0]
S_AXI_ACLK
S_AXI_ARADDR[13:0]
S_AXI_ARESETN
S_AXI_ARPROT[2:0]
S_AXI_ARVALID
S_AXI_AWADDR[13:0]
S_AXI_AWPROT[2:0]
S_AXI_AWVALID
S_AXI_BREADY
S_AXI_RREADY
S_AXI_WDATA[31:0]
S_AXI_WSTRB[3:0]
S_AXI_WVALID
Blue[3:0]
Green[3:0]
Red[3:0]
S_AXI_ARREADY
S_AXI_AWREADY
S_AXI_BRESP[1:0]
S_AXI_BVALID
S_AXI_RDATA[31:0]
S_AXI_RRESP[1:0]
S_AXI_RVALID
S_AXI_WREADY
hdmi_text_controller_v1_0_AXI
vga_to_hdmi
ade
aux0_din[3:0]
aux1_din[3:0]
aux2_din[3:0]
blue[3:0]
green[3:0]
hsync
pix_clk
pix_clk_locked
pix_clkx5
red[3:0]
rst
vde
vsync
TMDS_CLK_N
TMDS_CLK_P
TMDS_DATA_N[2:0]
TMDS_DATA_P[2:0]
hdmi_tx_0
axi_arready
axi_awready
hdmi_clk_n
hdmi_clk_p
hdmi_tx_n[2:0]
hdmi_tx_p[2:0]
axi_bresp[1:0]
axi_bvalid
axi_rdata[31:0]
axi_rresp[1:0]
axi_rvalid
axi_wready
axi_awaddr[13:0]
axi_wdata[31:0]
axi_wstrb[3:0]
axi_wvalid

week 1:



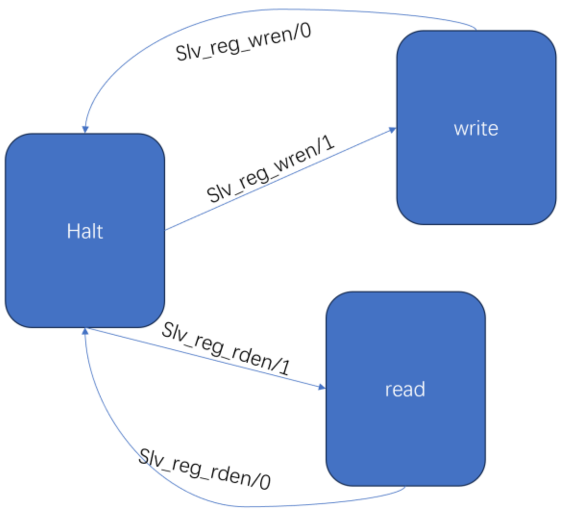
clk_wiz_0
reset
locked
clk_in1
clk_out1
clk_out2
vga_controller
pixel_clk
reset
hs
vs
active_nblank
drawX
drawY
hdmi_text_controller_v1_0_AXI
S_AXI_ACLK
S_AXI_ARESETN
[11:0] S_AXI_AWADDR
[2:0] S_AXI_AWPROT
S_AXI_AWVALID
[31:0] S_AXI_WDATA
[3:0] S_AXI_WSTRB
S_AXI_WVALID
S_AXI_WREADY
S_AXI_BRESP
S_AXI_BVALID
S_AXI_BREADY
[11:0] S_AXI_ARADDR
[2:0] S_AXI_ARPROT
S_AXI_ARVALID
S_AXI_ARREADY
[31:0] S_AXI_RDATA
[1:0] S_AXI_RRESP
S_AXI_RVALID
S_AXI_RREADY
slv_reg_rden
axi_araddr
[3:0]axi_wstrb
axi_wraddr
AXI Interface
drawX
drawY
slv_regs
clk
rd_en
rd_address
[3:0]wstrb_en
rd_address
fon_address
print_data
red
green
blue
print_data
fon_address
my_font
hdmi_tx_0
pix_clk
pix_clkx5
pix_clk_locked
rst
red
green
blue
vde
hsync
vsync
TMDS_CLK_P
TMDS_CLK_N
TMDS_DATA_P
TMDS_DATA_N
HDMI Output

Week 2:



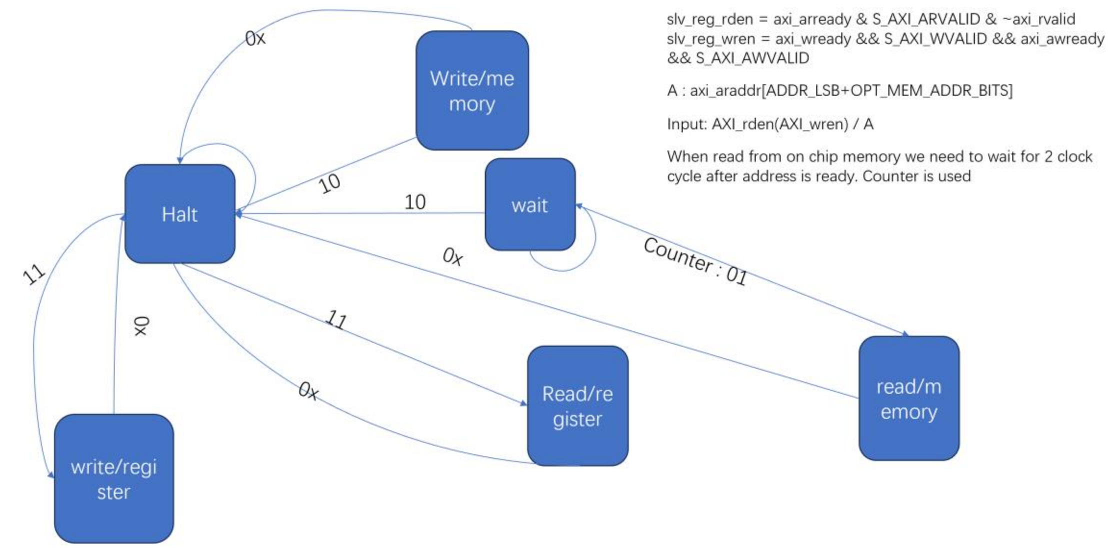
FSM : AXI Read and Write

Week1:



slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid
slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID

Week2



slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid
slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID

A : axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS]

Input: AXI_rden(AXI_wren) / A

When read from on chip memory we need to wait for 2 clock cycle after address is ready. Counter is used

3.Module Description

a. Microblaze system

MicroBlaze:
MicroBlaze is a soft-IP based 32-bit CPU which can be programmed using a high-level language which has 32-bit modified Harvard RISC architecture. In this lab, Microblaze is the system controller and handles tasks which do not need to be high performance. It can be configured with different instruction and data cache sizes, as well as various peripherals to suit the specific application requirements.

Clocking Wizard:
The Clocking Wizard is a clock generating tool that helps in generating clocking structures for the FPGA design. It is used to create clock signals with specific frequencies and phase relationships, which are essential for synchronizing various components in the design.

Processor System Reset:
The Processor System Reset IP block is used to control the reset signals for the MicroBlaze processor and other components in the design. It provides a way to control and synchronize the reset process to ensure the system starts up in a defined state. It is an active low reset.

MicroBlaze Local Memory:
Local memory refers to the on-chip memory resources available to the MicroBlaze processor. The MicroBlaze local memory is an integral part of the processor and is used for storing program code, data, and stack memory.

AXI Interconnect:
The AXI (Advanced extensible Interface) Interconnect is used to connect various IP blocks within an FPGA design. It provides a high-speed and flexible communication infrastructure. The AXI Interconnect can be configured to support different data widths, address ranges, and arbitration schemes to facilitate communication between different components like the MicroBlaze, memory, and peripherals.
The AXI Interconnect follows a bus-based architecture. Various masters (IP cores that initiate transactions) and slaves (IP cores that respond to transactions) are connected through the AXI Interconnect. This architecture allows for multiple masters and slaves to communicate simultaneously.

AXI Interrupt Controller:
The AXI Interrupt Controller controls interrupt signals within the design. It's responsible for handling and distributing interrupt requests from various peripherals to the MicroBlaze processor. It can allow the processor to respond to external inputs.

AXI Uartlite:
The UART (Universal Asynchronous Receiver/Transmitter) communication module is designed to work with AXI (Advanced eXtensible Interface) bus architecture. UART communication is a standard method for serial data transfer, often used to establish communication between an FPGA or SoC system and external devices, such as microcontrollers, sensors, or other computers. In this lab we use UART to print some messages in serial terminal.

Hdmi_text_controller
The hdmi_text_controller is packed IP that created by us. The IP is used to display text and character on screen. It can interact with software to write text the screen.

b. SV module description

Module: toplevel.sv
Input: Clk, reset_rtl_0, uart_rtl_0_rxd,
Output: uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, hdmi_tmds_data_n, hdmi_tmds_data_p
Description: The module functions as the top module of the lab. The input of the module has basic Clk and reset signal. Other signals can be divided into USB signal, HDMI signal and UART signal.
Purpose: The module is the top level of this lab which will instantiate our block design including hdim controller.

c. Internal for HDMI controller IP

Module: vga_controller.sv

Input: pixel_clk, reset,

Output: hs, vs, active_nblank, sync, drawX, drawY

Description: The module takes only a reset and pixel_clk as input. The horizontal sync signal is set to high until 640 pixels have been rendered, then it is set low. Vertical sync is set high after the entire frame has been rendered, and low otherwise. This is a holdover from the old CRTs, where some method would be needed to direct the electron beam to the appropriate positions after finishing rendering a line (reset to beginning of next line) or frame (reset to the top left of the screen). The active_nblank signal is there to dictate whether a pixel should be rendered at that coordinate or not (since the actual size of the counters extend beyond the 640x480 pixel screen size, for housekeeping purposes). DrawX and DrawY provide the current coordinates of the pixel being drawn.

Purpose: This module generates vertical and horizontal sync signals, a screen blanking signal, and the current X and Y coordinates of the pixel being drawn. This is to generate control signals and logic that other modules can use to draw the pixels as necessary on screen.

Module: font_rom.sv

Input: addr

Output: data

Description: The module has input address of sprite. Output 8-bit data which represent a line of character. The data is mirror shaped. The user needs to adjust the order of data otherwise will print wrong data. “1” in the fon data indicate we have to draw pixel in this location, otherwise just draw background color.

Purpose: The module acts as an image catalog. We can search for specific character. We use this module to test whether the pixel need to be drawn.

Module: hdmi_text_controller_v1_0.sv

Input: axi_aclk, axi_aresetn, axi_awaddr, axi_awprot, axi_awvalid, axi_wdata,axi_wstrb, axi_wvalid, axi_bready, axi_araddr, axi_arprot, axi_arvalid, axi_rready

Output: axi_awready, axi_wready, axi_bresp, axi_bvalid, axi_arready, axi_rdata, axi_rresp, axi_rvalid,

Description: The module instantiates AXI Bus Interface AXI, VGA controller, VGA_to_HDMI converter and clock wizard.

Purpose: The module is the toplevel of HDMI IP. It has instantiated many modules. It is used to connect AXI bus with modules including VGA controller and HDMI converter.

Module: hdmi_text_controller_v1_0.sv (week1)

parameter integer C_S_AXI_DATA_WIDTH = 32,

parameter integer C_S_AXI_ADDR_WIDTH = 12

local parameter integer OPT_MEM_ADDR_BITS = 9

Input:

S_AXI_ACLK: Global clock signal for the AXI bus.
S_AXI_ARESETN: Global reset signal for the AXI bus (active low).
S_AXI_AWADDR: Write address issued by the master.
S_AXI_AWPROT: Write channel protection type.
S_AXI_AWVALID: Signal indicating a valid write address and control information.
S_AXI_WDATA: Write data issued by the master.
S_AXI_WSTRB: Write strobes indicating which byte lanes hold valid data.
S_AXI_WVALID: Signal indicating valid write data and strobes.
S_AXI_BREADY: Signal indicating that the master can accept a write response.
S_AXI_ARADDR: Read address issued by the master.
S_AXI_ARPROT: Protection type for read transactions.
S_AXI_ARVALID: Signal indicating valid read address and control information.
S_AXI_RREADY: Signal indicating that the master can accept the read data and response information.
Drawx: X-coordinate for drawing text.
Drawy: Y-coordinate for drawing text.

Outputs:

S_AXI_AWREADY: Signal indicating that the slave is ready to accept a write address.
S_AXI_WREADY: Signal indicating that the slave can accept the write data.
S_AXI_BRESP: Write response status.
S_AXI_BVALID: Signal indicating a valid write response.
S_AXI_ARREADY: Signal indicating that the slave is ready to accept a read address.
S_AXI_RDATA: Read data issued by the slave.
S_AXI_RRESP: Read response status.
S_AXI_RVALID: Signal indicating a valid read response.
RED, GREEN, BLUE: Output signals for the color components in the display.

Description: The module can be divided into two parts. Firstly, the module handle AXI write and read transactions. The module has a memory-mapped register space represented by the slv_regs[601]. The last register is used to store the color. This space can be read from and written to by the master through AXI transactions. It is designed to store user-specific data, and the registers can be accessed and modified by the master through the AXI interface. We have logic expressions:

slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid
slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready
&& S_AXI_AWVALID

According to slv_reg_rden and slv_reg_wren, we will write or read data.
The module contains user logic responsible for drawing text on a display. We calculates the address in the memory-mapped register space based on input coordinates (Drawx and Drawy). We instantiate fon module to get the location of the pixel. Then we access the last register to get

the color of the pixel. Besides, we have IVN bit. If IVN is high, we will inverse background color and foreground color.
Purpose: The module provides an AXI interface that handles AXI write and read transactions. The module serves as a bridge between the AXI bus and the HDMI text controller, providing an interface for external components to interact with and control the text display.

Module: hdmi_text_controller_v1_0.sv (week2)
parameter integer C_S_AXI_DATA_WIDTH = 32,
parameter integer C_S_AXI_ADDR_WIDTH = 14 (need to change in toplevel of IP)
local parameter integer OPT_MEM_ADDR_BITS = 11

Inputs and outputs are the same.

Changes:

1. We change the VRAM from register to on chip memory. We also need to change some parameter to fit the address bit. We no longer use 601 registers but 1200 memory space with 8 additional color register.
2. We change the logic of read and write. Since we still need to read and write register. We can keep some part of week1. We use axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS] (significant bit of address) to decide whether to use OCM or register as OCM address is less than 0x800.
3. We keep the logic for slv_rden and slv_wren. However, we need to add wait state for OCM reading. As mentioned in lab5, we need wait state to wait for reading data. We use a counter to compute a read_ready signal. When address is ready, counter equal to 1. When counter is 2, we insert read ready. And when counter become 3, we valid read data is available at the read data bus.
4. Logic for computing address of character from coordinate have been changed since we only contain two characters in every address. The specific logic has been written in previous part.
5. We no longer have the control register, instead we have 8 palette register. We need to use index which store in VRAM to get access to the palette register.

d. IP Block

Clocking Wizard:
The Clocking Wizard is a clock generating tool that helps in generating clocking structures for the FPGA design. It is used to create clock signals with specific frequencies and phase relationships, which are essential for synchronizing various components in the design.

hdmi_tx_0
The VGA-HDMI IP shares similarities by rendering everything with RGB signals, horizontal and vertical sync, much like VGA does. The VGA-HDMI IP uses complex logic to generate its outputs, creating control signals for the red and green channel and using the blue channel to encode the hsync and vsync signals, generating specific hardware to encode that. There are also multiple serializers to convert the data to appropriate TMDS_DATA and TMDS_CLK outputs, which have

their own output buffers.

Blk_mem (week2)
The blk_mem is IP for constructing on chip memory. We choose true dual port which we can access two different addresses simultaneously. Write and read width is 32 which fit with address bit. Write and read depth is 1200 bit which is the number of VRAM. We used blk_mem to store character code. Each address contains two characters. Since both AXI and character address need to be read or write, we have to use two ports. The blk_mem will make our design more efficient and saves a lot of time on synthesis.

e. C Code

hdmi_text_controller.h
The struct HDMI_TEXT_STRUCT was changed. Originally, it had an array of 4800 bytes with the 8 32-bit palette registers directly afterwards. However, due to our memory-mapping, we had to change this and move the memory location of the palette back so we could write to it properly. This was accomplished by making an array of 3392 bytes.

hdmi_text_controller.c
We changed the function setColorPalette. Since each palette is represented as a 32-bit value, but each RGB value is represented as 12 bits, 4 bits for each RGB channel. There are two such 12-bit blocks of RGB values in each 32-bit registers. The RGB values are written by bitshifting in the values to the appropriate region. The color value of input selects the location to write to, and mathematical operations must be conducted to fetch the appropriate register and shift left the correct distance. The register is retrieved by indexing into the register specified by color integer-divided by 2. Then, the distance to shift left is determined by the equation (color % 2)*12. An additional offset of value 9, 5, or 1 is added to the distance to get the correct offset for the red, green, and blue values being shifted in.

4.Resource Usage

Lab 7.1

LUT	15112
DSP	3
Memory (BRAM)	32
Flip-Flop	21119
Latches	0
Frequency	76.669MHz

Static Power	0.074W
Dynamic Power	0.413W
Total Power	0.487W

Lab 7.2

LUT	2493
DSP	3
Memory (BRAM)	34
Flip-Flop	1838
Latches	0
Frequency	100.667MHz
Static Power	0.074W
Dynamic Power	0.378W
Total Power	0.452W

Analysis of Designs

Using on-chip memory is much, much more efficient. As evidenced by the massive difference in flip-flop and LUT usage, there is a drastic difference in the amount of logic used and resources used. Because our SystemVerilog code defines the 600 registers behaviorally, all the appropriate select and write strobe logic is inferred. Setting aside the footprint of 600 registers, that is a lot of control logic! The complexity of the logic also shows in the timing- our 7.1 design failed timing by a pretty large margin, and I strongly believe this is due to the complexity of the logic to control the registers. In contrast, 7.2 meets timing (albeit narrowly) and uses much less resources, shifting a massive portion of the LUT and FF usage to BRAM.

5 Conclusion

This lab is a good practice of creating IP and help us understand how AXI interact with different signal. In the first week we try to figure out how to use register as VRAM. We have to calculate address for character and understand how palette and VRAM work. In the second week, we updated VRAM to on chip memory which is much more efficient than register. We have to use wait state to read value correctly.

Haoyu met a bug when reading the color of the text. At first, I think we just need to write read logic for OCM and ignore that when calling set_palette in software, we will use the value of color register. I add the logic for reading register. I also have problem with displaying nothing at first as I didn't notice that we need to create memory space in VRAM. We won't use memory address between VRAM and color register.

We may use this lab as an instruction to build palette and sprite. We decided to create our own fon module. We will design similar logic that can access our VRAM. Besides, this lab is also a good practice to implement on-chip memory. We can also keep some parts of this lab to draw some text or character on the screen.

I think the lab manual is clear. The only thing is I think AXI signal part is not very clear in the lab manual and lecture. It is hard to analyze how many cycles we need to wait. But I think the structure of 2-week lab is good since we test the week2 project on week1.