ECE385

Fall 2022

Experiment 7

VGA Text Mode Controller with Avalon-MM Interface

Lai Shu Xian
Section NL
TA: Nick Lu

Introduction

The VGA interface we created interacts with our FPGA registers/on-chip memory via an Avalon memory-mapped bus. The VGA interface provides as inputs two addresses – one for read and one for write to our VRAM. This interface essentially provides additional two ports (one for Avalon read, another for Avalon write) for VRAM's memory accesses, on top of the VGA controller module, which only reads from VRAM. This interface allows us to program (read/write) to VRAM from the NIOS II software end, which we program in Eclipse in C.

The design IP that we created *(vga_text_avl_interface.sv)* builds on top of lab 6.2 design by instantiating an actual memory structure inside the IP, as well as including the VGA controller inside the IP. The memory structure that we created, VRAM, enables more complex drawings, colour, and fonts onto the screen besides just a ball and some background colour.

Written Description of Lab 7 System

Week 1

Description of Lab 7 System, VGA Text Mode Controller IP

We have a memory structure that utilizes 600 FPGA registers, called the VRAM. VRAM is the memory structure that determines what is being shown on the screen. Each VRAM address contains 32 bits of data, where we use 8 bits to represent a code of what character is being shown on our VGA screen. 7 out of each of the 8 bits is used as an index to a Font_ROM that was readily provided. The 8^{th} bit is used as a Boolean to flip the background/foreground of our glyph.

Font_ROM consists of 128-character glyphs – basically bitmaps of 128 different characters. Each glyph takes the dimension of 16 rows by 8 columns. Each of those bits inside the 16x8 block is 0 or 1, indicating foreground or background.

Our VRAM also consists of 1 extra register (hence VRAM has a total of 601 regs). The control register contains the RGB representation of two colours - one for foreground and one for background. We only have one of these, meaning our display screen on every instance at most display two colour.

The address of which we access our VRAM (which leads us to the character glyphs) is based on a VGA controller module that provides us the coordinate information about our electron gun. Using the coordinate information, we perform some calculation and feed out the RGB values to our DAC port.

All of the components that we have addressed here:

a. The 601 register VRAM structure
b. The font_ROM
c. The VGA_controller

These components are all wrapped inside a module which we would later customize as our own IP to be instantiated in Platform Designer.

On the software end, we were provided some readily available written program that prints writes to the VRAM certain texts while modifying the control register. This also makes use of the AVL bus that we created.

Logic used to read/write VGA registers

If AVL_CS is high and AVL_WRITE is high, we write AVL_WRITEDATA into VRAM[AVL_ADDR].

If AVL_CS is high and AVL_READ is high, we read VRAM[AVL_ADDR] into AVL_READDATA.

AVL_BYTE_EN is a four-bit bitmask telling us which 8-bit chunk we are writing/reading to on the VRAM.

Algorithm/Calculation to perform correct memory accesses

DrawX = dx, DrawY = dy

A character block is 16 pixel tall, 8 pixels wide.

$$characterBlockRow = \frac{dy}{16}$$
$$characterBlockColumn = \frac{dx}{8}$$

After knowing the character coordinates we are looking at, we need to calculate which VRAM register we are interested in. A VGA monitor has 640 pixels per row, 480 pixels per column. By the dimension of the screen and the character, a VGA screen holds 80 characters wide (per row), 30 characters tall (per column).

Since one VRAM register holds information for 4 characters,

$$VRAM_{index} = characterBlockRow * \frac{80}{4} + \left(\frac{characterBlockColumn}{4}\right)$$

Using VRAM_index, we will access a register that holds the 32-bit information for 4 characters. Because of the little-endian storage, for characters in column 0, 4, 8, … we will be accessing the highest 8-bit chunk; for characters in column 1, 5, 9… we will access the second highest 8-bit chunk and so on.

$$if\ (characterBlockColumn\%4 == 0)\ font_{index} = VRAM[VRAM_{index}][30:24]$$
$$else\ if\ (characterBlockColumn\%4 == 1)\ font_{index} = VRAM[VRAM_{index}][22:16]$$
$$else\ if\ (characterBlockColumn\%4 == 2)\ font_{index} = VRAM[VRAM_{index}][14:8]$$
$$else\ if\ (characterBlockColumn\%4 == 3)\ font_{index} = VRAM[VRAM_{index}][6:0]$$

Font_index only tells us which character we are drawing, but each character has 16 rows with 8 pixels on each row. To calculate which row we are accessing from font_rom

$$fontROM_{row} = dy\%16$$
$$fontROM_{data} = FONTROM_{ARRAY}[(font_{index} * 16) + fontROM_{row}]$$

From fontROM_data, we need to access that exact pixel out of 8 pixels

$$fontROM_{pixel} = fontROM_{data}[(7 - dx)\%8]$$

This is a single bit with one indicating foreground, 0 indicating background.

Inverse colour bit

After extracting the pixel from previous steps, we recall that we still have information about the most significant bit from our 8-bit chunk we access in VRAM.

$$InverseBit = VRAM[VRAM_{index}][31\ or\ 23\ or\ 15\ or\ 7]$$
$$if\ (InverseBit\ XOR\ fontROM_{pixel}) RGB = controlRegister[foregroundRGB]$$
$$else\ RGB = controlRegister[backgroundRGB]$$

Week 2

On-chip memory

Created a RAM-2 Megafunction from the IP catalogue GUI. This gives me four ports:

- AVL_READADDR uses one for NIOS II to read from on-chip memory.
- AVL_WRITEADDR uses one for NIOS II to write to on-chip memory.
- VGA_READADDR uses one to allow us to use dx, dy to read from the correct index of VRAM.
- VGA_WRITEADDR is left blank, we never write to VRAM from VGA's side.

Now each VRAM address still holds data with data-width = 32 bits. However, we use 16 bits to describe a single character now (previously we used 8). This doubles the VRAM on chip memory size that we need.

We will need to expand our memory range from week 1 (first table) to table below:

Table 3. Peripheral Memory Map

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 - 0x3FF	0x0000 0961 - 0x0000 0FFF	Unused but reserved by Platform Designer

Table 8. Peripheral Memory Map (Week 2, Color Mode)

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 Platform Designer
0x800 - 0x807	0x0000 2000 - 0x0000 201F	Palette- 8 words of 2 colors each, for 16-color palette
0x808 – 0xFFF	0x0000 2020 – 0x0000 3FFF	Unused but reserved by Platform Designer

Because we expanded our memory range, we need to modify our platform designer IP to take in a wider address width from originally 10-bits to 12 bits.

Modified sprite drawing algorithm

There are two modification that we need to do to our previous algorithm. The first modification is due to additional information being stored inside each VRAM register – the foreground colour index and the background colour index. We need to use these indexes (still dependant on invert bit) to access colour from VRAM again.

The second modification is how we calculated which part of VRAM we are accessing, since now each VRAM memory access is no longer about 4 characters, it's two now.

Previously,

$$VRAM_{index} = characterBlockRow * \frac{80}{4} + \left(\frac{characterBlockColumn}{4}\right)$$

Now,

$$VRAM_{index} = characterBlockRow * \frac{80}{4} + \left(\frac{characterBlockColumn}{2}\right)$$

Previously,

$$if\ (characterBlockColumn\%4 == 0)\ font_{index} = VRAM[VRAM_{index}][30:24]$$
$$else\ if\ (characterBlockColumn\%4 == 1)\ font_{index} = VRAM[VRAM_{index}][22:16]$$
$$else\ if\ (characterBlockColumn\%4 == 2)\ font_{index} = VRAM[VRAM_{index}][14:8]$$
$$else\ if\ (characterBlockColumn\%4 == 3)\ font_{index} = VRAM[VRAM_{index}][6:0]$$

Now,

$$if\ (characterBlockColumn\%2 == 0)\ font_{index} = VRAM[VRAM_{index}][30:24]$$
$$else\ if\ (characterBlockColumn\%2 == 1)\ font_{index} = VRAM[VRAM_{index}][14:8]$$

Additional code needed because due to dual port read-write. If NIOS-II attempts to write to registers that stores the RGB colour, we want to see that change immediately on our screen too. There should be no delay for colour switching, so we need the extra "palette_content" variable to help react to that.

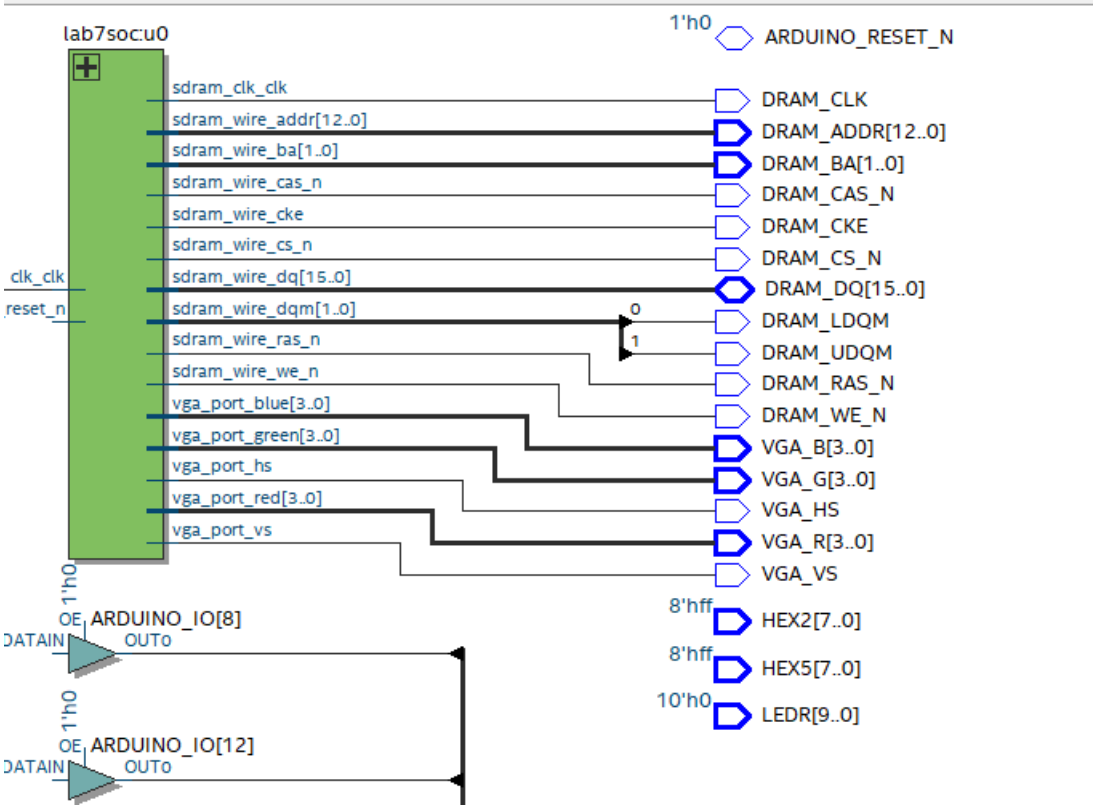
```
always_ff @ (posedge CLK) begin
   //character index is from 0 to 1199
   //reserved index is from 1200 to 2047
   //palette index is from 2048 to 2055
   if (AVL_WRITE && (AVL_ADDR[11] == 1'b1) )
      palette_content[AVL_ADDR - 2048] <= AVL_WRITEDATA;
end
```

Additional code modifications needed for multi-colour, palette-based drawing.

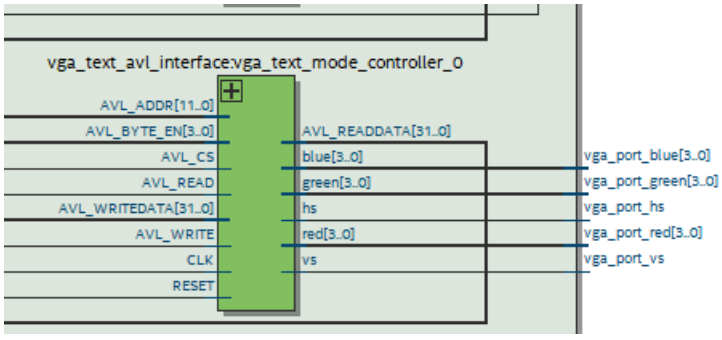
```
always_ff @(posedge pixel_clk) begin
   if (~blank) begin
      red <= 0;
      green <= 0;
      blue <= 0;
   end

   //we take the index divide by two to get us to the correct palette index 0 to 7
   //we take the index % 2 to know which color we are using within that selected palette
   else if (color_index[0]) begin
      //color_index is odd number, take the higher palette
      red <= palette_content[color_index[3:1]][24:21];
      green <= palette_content[color_index[3:1]][20:17];
      blue <= palette_content[color_index[3:1]][16:13];
   end

   else begin
      red <= palette_content[color_index[3:1]][12:9];
      green <= palette_content[color_index[3:1]][8:5];
      blue <= palette_content[color_index[3:1]][4:1];
   end
end
```

Block Diagram

For top-level entity



For the vga_text_avl_interface we written:



Module Descriptions

Module: lab7soc.v Inputs: clk_clk, // clk.clk reset_reset_n, // reset.reset_n Outputs: sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, sdram_wire_dq, [1:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, [3:0] vga_port_blue, [3:0] vga_port_red, [3:0] vga_port_green, vga_port_hs, vga_port_vs Inout: inout wire [15:0]	Description: All the exported wires we export from Platform Designer go here. Purpose: Platform designer generated module based on the connection we set up with the Platform designer GUI. Represents our System-on-Chip
Module: vga_text_avl_interface.sv Inputs: input logic CLK, input logic RESET, input logic AVL_READ, input logic AVL_WRITE, input logic AVL_CS, input logic [3:0] AVL_BYTE_EN, input logic [11:0] AVL_ADDR, input logic [31:0] AVL_WRITEDATA, Outputs: [31:0] AVL_READDATA, [3:0] red, green, blue, hs, vs	Description: Refer to earlier part of this lab document (basically the page after the cover page). Purpose: Avalon-interface for NIOS-II chip to be able to read/write from VRAM.
Module: VGA_controller.sv Inputs: Clk, Reset Outputs: hs, vs, pixel_clk, blank, sync, [9:0] dx, dy	Description: Based on clock and reset signal, output the position of our electron gun and blanking interval

	Purpose: Coordinates position of electron gun is crucial to help us determine which section of the VRAM we should access.
Module: lab7.sv Inputs: input MAX10_CLK1_50, Outputs: output DRAM_CLK, output DRAM_CKE, output [12: 0] DRAM_ADDR, output [1: 0] DRAM_BA, inout [15: 0] DRAM_DQ, output DRAM_LDQM, output DRAM_UDQM, output DRAM_CS_N, output DRAM_WE_N, output DRAM_CAS_N, output DRAM_RAS_N, output VGA_HS, output VGA_VS, output [3: 0] VGA_R, output [3: 0] VGA_G, output [3: 0] VGA_B,	Description: Purpose: Serves as top-level entity
Module: font_rom.sv Inputs: [10:0] addr Outputs: [7:0] data	Description: We use dx, dy and perform some calculation to access the index in VRAM and use the index in VRAM to access the block that we want to draw. Addr is split into two parts, the first 7 bits help us access the start of the character glyph, the last 4 bits help us determine which row of the glyph we want. Data is a row of 8-pixels bitmap. Purpose: Read-only memory to store sprite bitmaps for 128 different characters.
Module: ocm.v Inputs: address_a, address_b, byteena_a, byteena_b, clock,	Description: Dual port on-chip memory that takes in read_enable for the respective ports, two addresses, the write_enable for respective ports and the outputs data for the reads.

data_a, data_b, rden_a, rden_b, wren_a, wren_b Outputs: q_a, q_b	Purpose: Simple dual port synchronous on-chip memory we used to represent VRAM.

Design Resources & Statistics

	Registers (week 1)	On chip memory (week 2)
LUT	31453	3487
DSP	0	0
Memory (BRAM)	46080	193536
Flip-flop	21209	2105
Frequency	67.53MHz	94.97MHz
Dynamic Power	(did not save different versions of weeks 1 and 2, other data was available because I took note in week 1, did not take note on powers)	66.30mW
Static Power	(as above)	96.53mW
Total Power	(as above)	181.31mW

Using on-chip memory are more efficient compared to using registers, especially when a relatively large memory size is needed. This is because using registers will waste a lot of resources as registers are individually synthesized as D flip-flops, this significantly wastes the chip available resources.

Other than chip resources, embedded memories perform much more better than memories synthesized with lookup-tables (registers) as on-chip memories can operate at higher clock rates leading to higher throughput and low latency and lower power dissemination. One exception is that for very small memories, it is probably more simple and efficient to just LUT.

Conclusion

My lab did not pass one of the testcases provided but I don't really know why. The testcase that I failed was supposed to check whether I hardwired the colours but I did not, so I am not sure how to fix it.

Learning how to create and instantiate our own IP is a useful skill. Learning how to perform sprite-based drawing and having the Avalon interface so that we can write software that changes the display is also another useful skill. Learning how to instantiate/infer on-chip memory is another useful skill.