ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to the Avalon-MM Interface and VGA Graphics

Please read this guide carefully and thoroughly as minor mistakes can impact the functionality of your entire project.

System Overview

For week 1 of this lab, you will create a simplified text mode graphics controller which is connected to the Avalon memory-mapped bus and supports 80 column text mode through the VGA output. This is functionally very similar to the original IBM monochrome graphics adapter (MGA) which was included with the IBM PC 5150 from 1981:
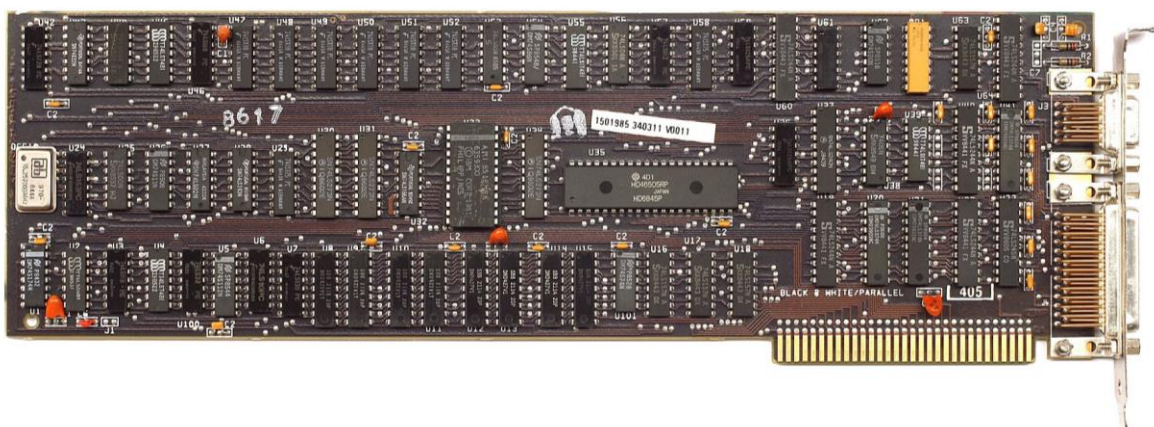

Figure 1. IBM Monochrome Graphics Adapter

Figure 2. IBM Codepage 437 Subset

Your graphics controller will support 80 columns by 30 rows, for a total of 2400 characters. Each character may be set to one of 128 glyphs (a subset of IBM codepage 437, shown in Figure 2) using 7 bits. In addition, the most significant bit may be set to draw the glyph with inverted colors, so a total of 8 bits are needed to specify each character. The bitmap for each glyph is provided in the included font_rom.sv, and each character consists of 8x16 pixels, so the total screen resolution is 80*8 horizontal by 30*16 vertical pixels – the familiar 640x480 VGA screen resolution from the previous labs.
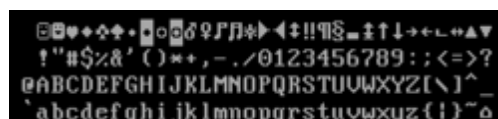
As your graphics controller is only required to support a character (text) drawing mode, each pixel is not individually addressable. Instead, each character will consist of 8 bits of data, and as the total screen consists of 2400 characters, your controller will provide 2.4kBytes of video memory (VRAM) which will define the contents on the screen. This VRAM will be memory mapped directly to the Avalon bus, allowing your Nios II CPU to access and modify the contents of the VRAM, thereby allowing the software to control the text being displayed. The provided

simple video driver code (.c/.h files) provide basic access to the controller you will design, as well as some test routines to validate the functionality of your hardware. In addition, there is a control register which you must implement to provide support for drawing the text in different colors. You are free to extend this as necessary for your final project to provide more sophisticated graphics capabilities. The overall system you will create appears as below, note that you will have to design and construct the Graphics Controller IP in this lab.
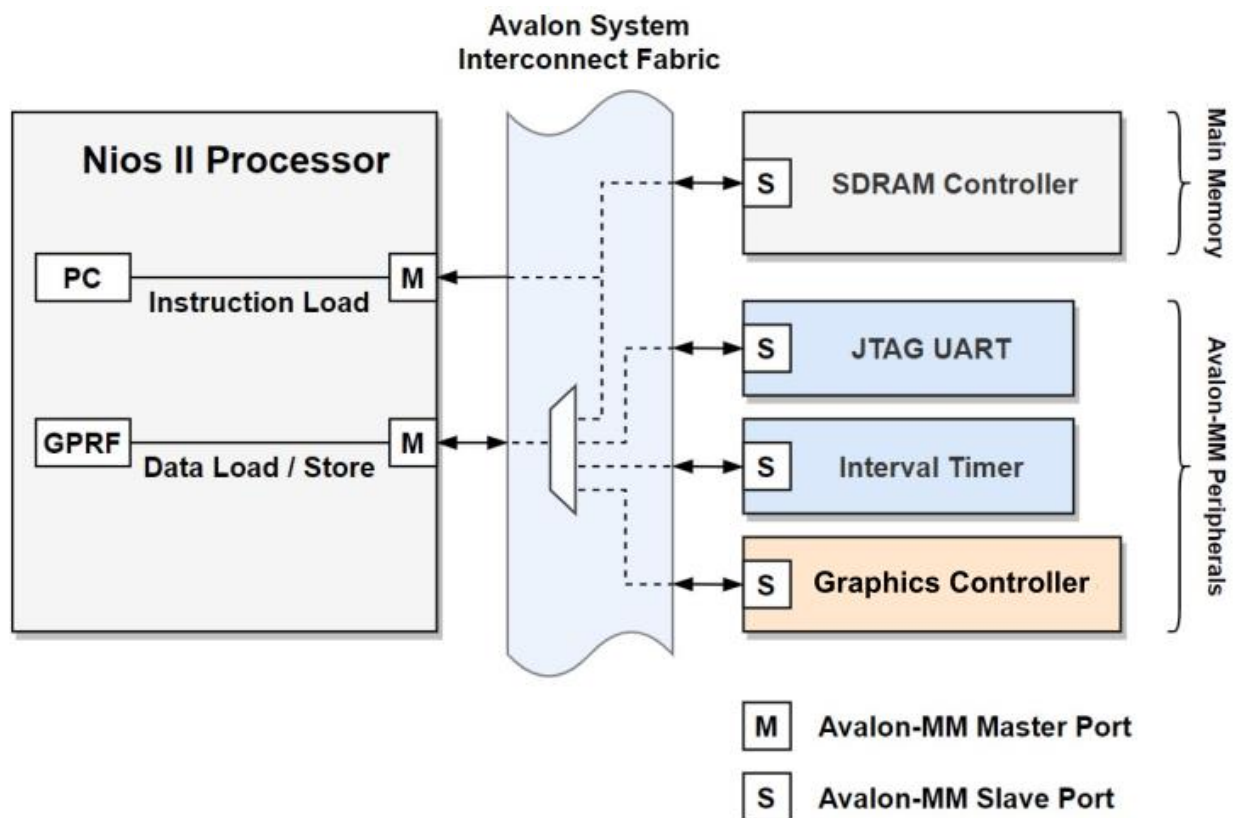

Figure 3. Overall System Design

Note: not all peripherals are shown in this figure. (PLLs, System ID, etc.)

To begin, start with your Platform Designer setup from the previous USB lab with the usual components like Nios II, SDRAM, and JTAG UART. Save and close it for now. In addition, you should start with your top-level design from the previous lab but delete the ball, VGA_controller, and Color_mapper module instantiations. In the next section, you will create your own *VGA_text_mode* component and add it to Platform Designer. You will then populate the provided SystemVerilog template with code which implements the Avalon MM bus. One you have completed the Avalon portion; you will then implement the graphics drawing logic to draw the characters using the VGA output. You may instantiate previously provided modules such as the VGA_controller or the Color_mapper inside the graphics controller module, as well as the newly provided font ROM (note, this is different from before, as those modules will now be located inside the graphics controller module, rather than the top-level). Finally, you will put it all together and run the test routines from the provided device driver code.

The VGA Text Avalon Interface

The interface module *vga_text_avl_interface.sv* will be the top-level file for the VGA text mode core component on Platform Designer (note that your overall design should still use the top level from the USB+VGA lab). We have provided the input/output signals declaration for you. There is a clock input (CLK), an active-high reset input (RESET), an exported conduit which consists of the VGA port signals (Red[3:0], Green[3:0], Blue[3:0], horizontal and vertical sync) and finally an Avalon-MM slave port which contains a bundle of signals whose specifications are given below.

The Avalon-MM slave port will complete read and write operations requested by its master, the Nios II processor (read/write operations correspond to load/store instructions in Nios). While the Avalon specifications provide many signals to use for its interface, we only need to use 7 signals to implement the slave port for this lab.

Table 1. Avalon-MM Slave Port Interface Signals

Name	Direction	Width	Description
read	Input	**1**	High when a read operation is to be performed.
write	Input	**1**	High when a write operation is to be performed.
readdata	Output	**32**	32-bit data to be read.
writedata	Input	**32**	32-bit data to be written.
address	Input	**10**	Address of the read or write operation.
byteenable	Input	**4**	4-bit active high signal to identify which byte(s) are being written.
chipselect	Input	**1**	High during a read or write operation.

Note that the data width of 32-bit and address width of 10-bit are chosen for this lab, they may be up to 1024-bit and 64-bit, respectively. We are using 32-bit **readdata** and **writedata** signals because they match the data width of Nios II, a 32-bit processor. As for the 10-bit address, which gives $2^{10} = 1024$ locations. At the minimum, the VRAM (2400 bytes) as well as the 32-bit control register need to be memory mapped. As each word in the Avalon bus is 32 bits, we will need 600 words (2400 bytes / 4 bytes per word) for the VRAM, and 1 word for the control register – a total of 601 words. The remainder of the 10-bit address space will be unmapped and unused.

Now it is up to you to implement the body of this module that completes the incoming read and write requests. Internally, you should create 601 registers, each 32-bit, that hold the values being read and written. There are some requirements that your design must satisfy:

- Read has a 1 cycle wait latency. This means that when read is high, **readdata** should have the value of the addressed register on the next cycle.
- Write has a 0 cycle wait latency. This means that when write is high, the corresponding write address, data, and byteenables will be present on the input ports simultaneously. The data should be committed to your module on the next cycle.
- Byte enable determines the bytes being written according to the table below.

Table 2. Byte Enable Description

byteenable[3:0]	Write Action
1111	Write full 32-bits.
1100	Write the two upper bytes.
0011	Write the two lower bytes.
1000	Write byte 3 only.
0100	Write byte 2 only.
0010	Write byte 1 only.
0001	Write byte 0 only.

You must create the registers as stated above. Internally, you should allow the Avalon bus to read and write those 601 registers, according to the memory map 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

Your hardware must then draw the characters which have been transmitted to the VRAM in the following way, based on the contents of each register in memory.

Table 4. Bit Encoding for VRAM (Word Addresses 0x000-0x257)

Bit	31	30-24	23	22-16	15	14-8	7	3-0

Function	IV3	CODE3	IV2	CODE2	IV1	CODE1	IV0	CODE0

IVn = Inverse bit N

CODEn = Glyph code from IBM Codepage 437

For example, if the memory at **word address** 0x15 contains 0x0101038D, then the character starting at position column = 1, row = 4 (assuming we start at column/row 0) should display:

Table 5. Example image – VRAM[0x15] = 0x0101038D

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	-	-	♫	🖤	🔲	🔲
2	-	-	-	-	-	-	-	-

Only the first 8 columns and 3 rows are shown.

Note that character 0x0D (music note) is displayed with inverted colors (black foreground on white background) due to bit IV0 (inverse character 0) being set. Also note that the word is specific in little endian (though the display driver provided accesses individual bytes, so that should be invisible to you).

The control register has the following bit mapping:

Table 6. Bit Encoding Control Register (Word Address 0x258)

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

BKG_R/G/B = Background color, flipped with foreground when IVn bit is set
FGD_R/G/B = Foreground color, flipped with background when IVn bit is set

Although the control register allows you to set colors, this is not true color text, as this only allows you to set the colors for the entire screen (e.g., you can have white on black text, or light green on dark green, etc.) You will implement true per-character color support for Week 2.

Note that although in normal operation, the VRAM is only written into (by the Nios II), this lab will ask that you implement bi-directional communication through the Avalon – MM bus. This allows you to extend the VGA controller with additional features, such as the reading back of the current line/pixel to do more sophisticated software control.
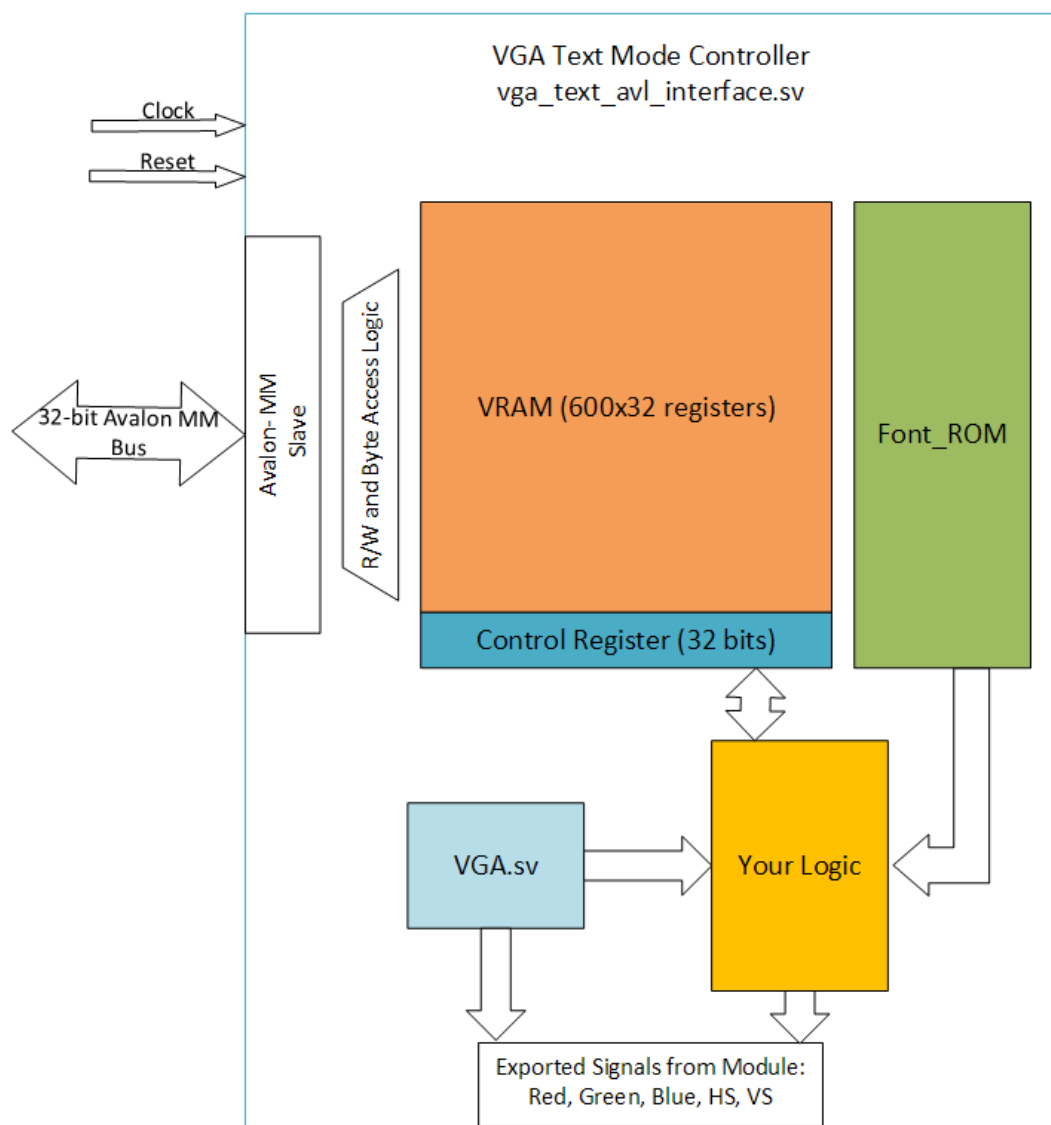
*Figure 4. VGA Text Mode Interface Block Diagram*

Creating a Platform Designer Component

To start implementation of the VGA Text Mode Controller *vga_text_avl_interface.sv*, follow these steps to add it to the Platform Designer IP catalog with component editor.

1. Launch Platform Designer editor and load your design that already has Nios II, SDRAM, UART, etc.
2. On the upper left **IP Catalog** panel, double click *New Component…*
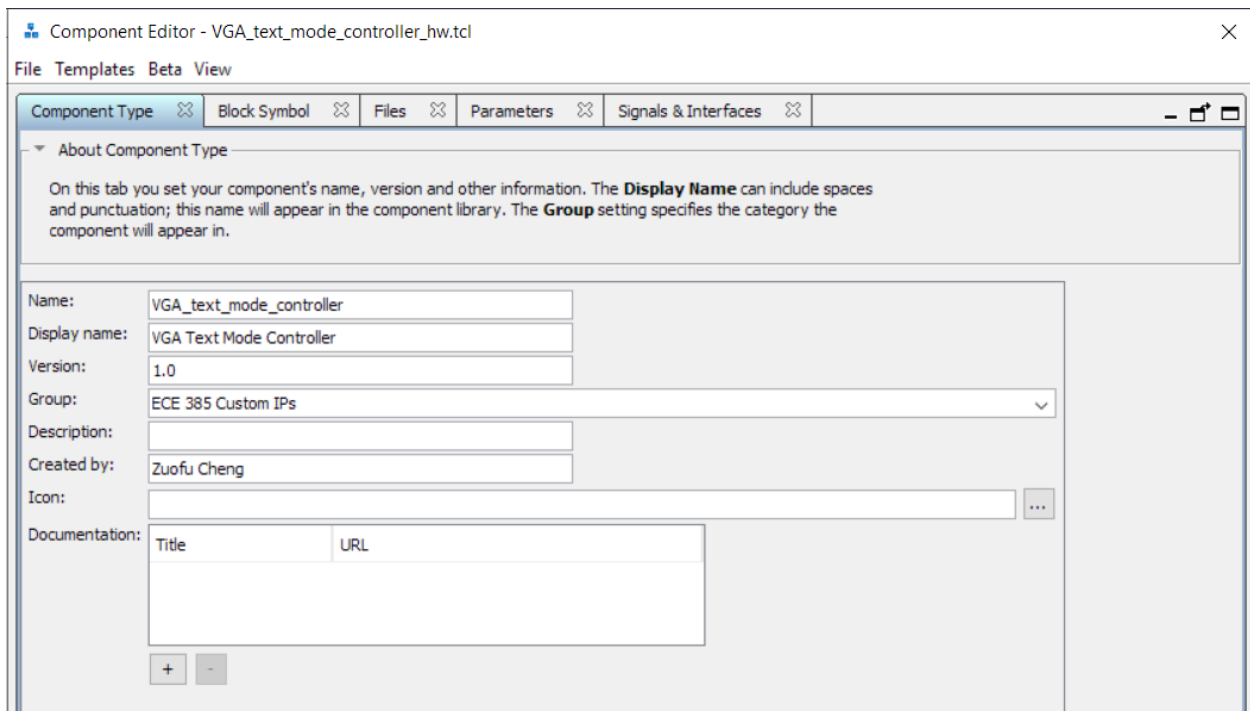
3. Enter the Name and Display name for your component. Display name is the one shown in Platform Designer IP Catalog. It is good practice to also keep track of the version of your IP, increment it when you make changes or fix issues. You can also categorize it to an existing or new group, here we make a group called "ECE 385 Custom IPs". Finally, give it a brief description.

4. Click on the Files tab.
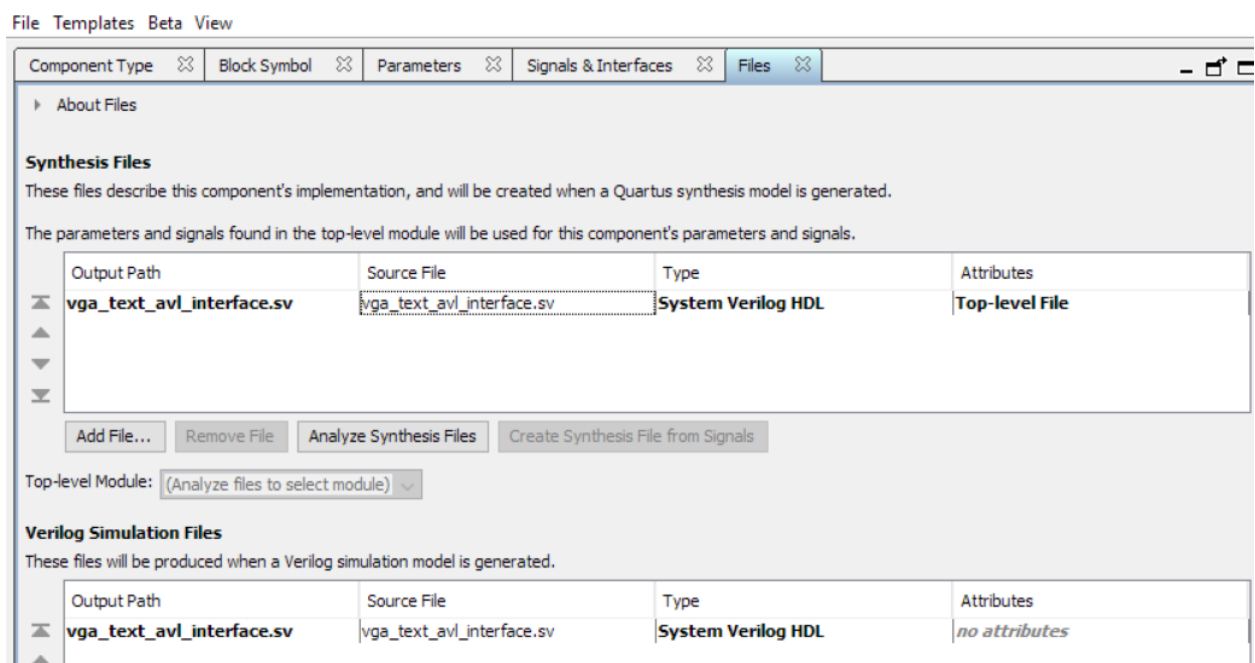
5. Under **Synthesis Files**, click on Add File… and choose *vga_text_avl_interface.sv*, which is the top-level module for this component. Under **Verilog Simulation Files**, click on **Copy from Synthesis Files**, this is useful if you want to simulate your system in ModelSim.

6. Click on the Signals & Interfaces tab. Now we want to create the Avalon ports and match the Avalon defined interface signals with our input/output declarations in *vga_text_avl_interface.sv*,

7. Let us add the clock input first. Click on *<<add interface>>*.
   - In Quartus 15.0, a default name of "avalon_slave" will appear, replace that with "CLK" and press Enter. The default port type is Avalon Memory Mapped Slave, but we want a clock input, click on the drop-down list for **Type**, and choose **Clock Input**.
   - In Quartus 16.0+, you are asked to choose from a list of port types, click on **Clock Input**, then rename it to "CLK" for **Name**.

   Finally, enter the **Clock rate** of 50MHz, your screen should look like this in both versions. If you added other interfaces by mistake, you can always right click those interfaces on the left tab and choose remove.
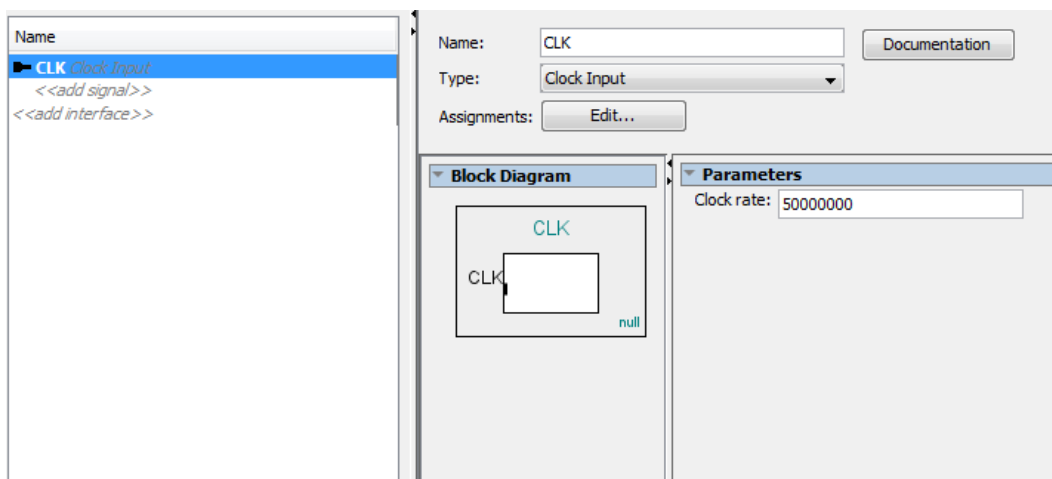

8. We now have a Clock Input interface for our component called CLK, but it's not associated with any signals defined in our top-level module yet. Obviously, the clock input defined in *vga_text_avl_interface.sv* is "input logic CLK". To make that association, click on *<<add signal>>*.
   - In Quartus 15.0, a default name of "new_signal" will appear, replace that with "CLK" to match the input name declared in *vga_text_avl_interface.sv*.
   - In Quartus 16.0+, choose the only option of "clk", then change the Name on the right tab to "CLK" to match the input name declared in *vga_text_avl_interface.sv*.

   The remaining default values should be correct, if not, change them to match the picture below.
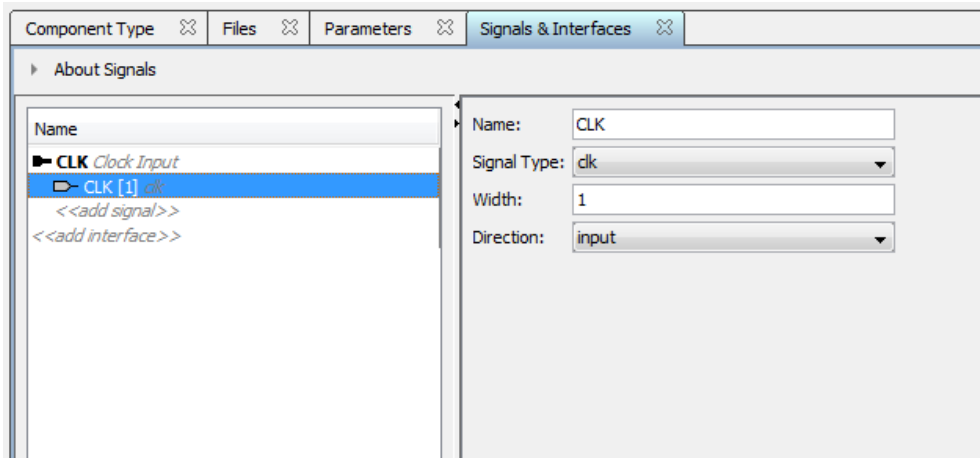
9. The Clock Input interface only needs one signal, so we are done with it. Now let's add the Reset Input interface. Click on *<<add interface>>*.
    - In Quartus 15.0, like the steps above, replace the default name with "RESET", then choose **Reset Input** for **Type**.
    - In Quartus 16.0+, likewise, choose **Reset Input** from the list of port types, then change the name to "RESET".

    The associated clock should be set to the "CLK" interface we just defined by default, if not, set it.
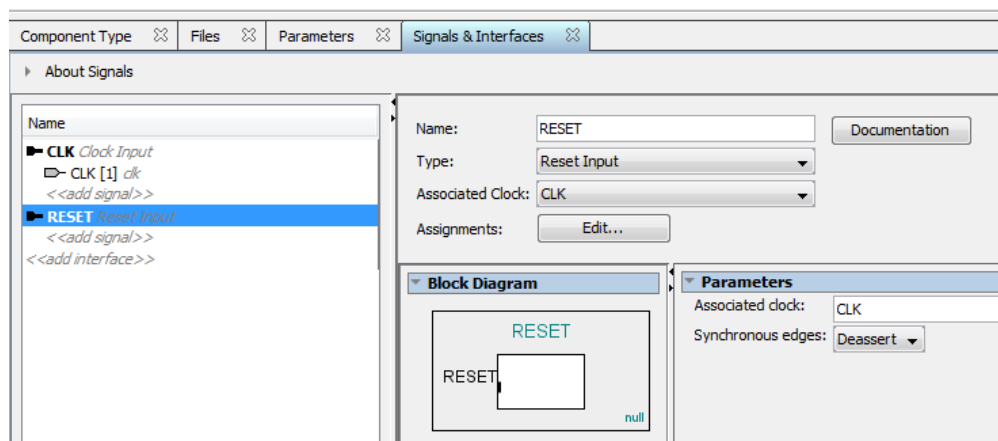
10. Now, add the associated RESET input to this interface. Click on *<<add signal>>*.
    Follow a similar procedure to Step 7, except use "RESET" for **Name**, and **reset** for **Type**.
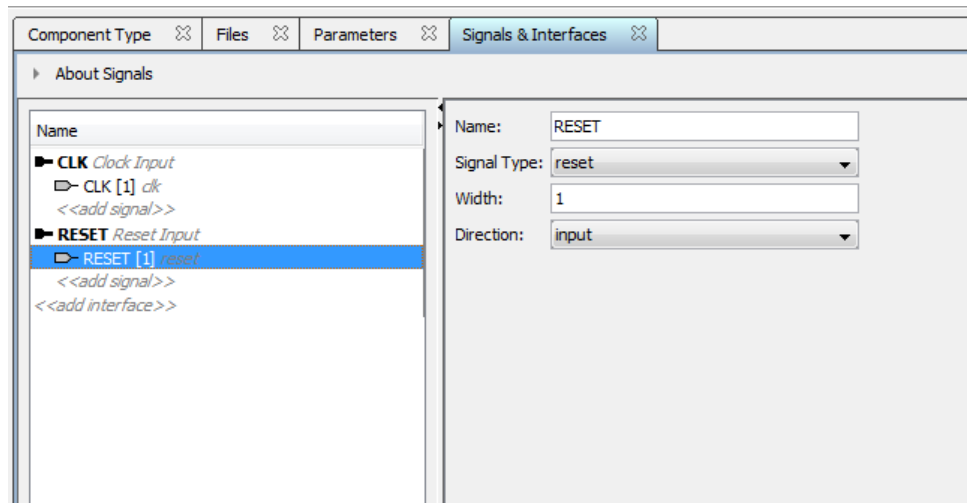
11. Next, add the main Avalon-MM Slave port. Click on *<<add interface>> -> Avalon Memory Mapped Slave*. You should be familiar with how to do this now, set the **Name** to "avl_mm_slave", **Type** to **Avalon Memory Mapped Slave**. Also set the **Associated Clock** and **Reset** to the CLK and RESET ports we defined earlier.
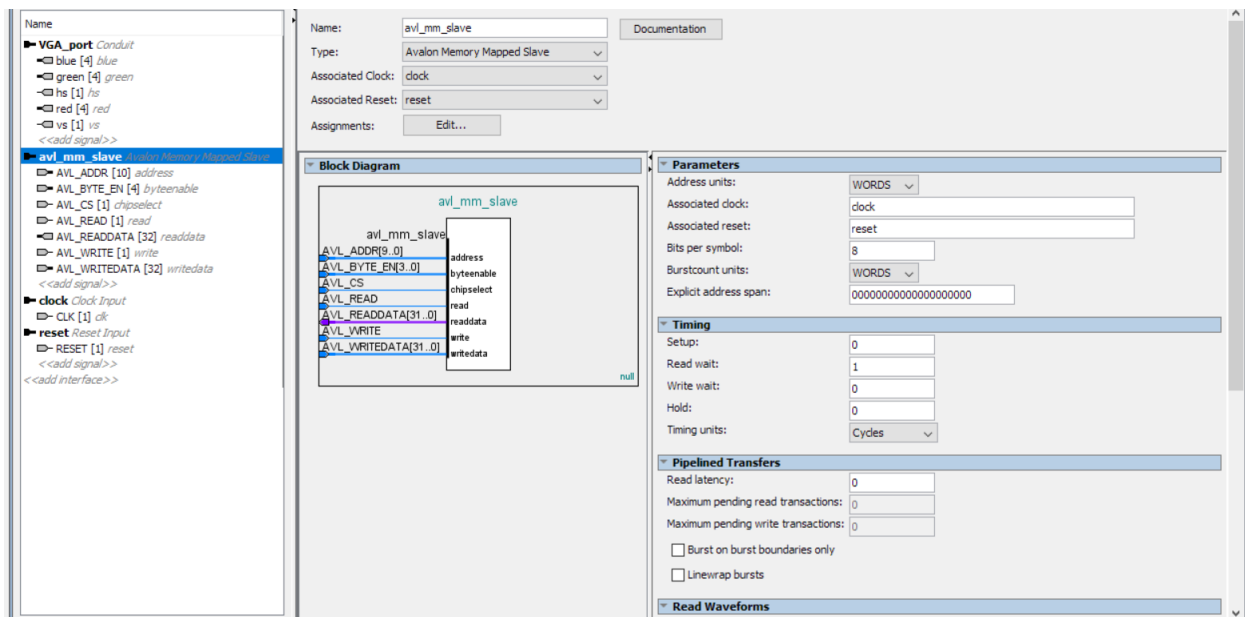

12. Under the **Parameters** section, check that you have identical settings as the picture above. Note that we make this Memory Mapped Slave byte addressable (8-bit per symbol) and the address units are in words (32-bit per word), so we expect the address to span a range of $4 \times 2^{10} = 4096$ (0x00 to 0xFFF), making it no different than accessing no regular memory. In C, we can access this range directly as an array of 4096 elements (though not all are used).

13. Under the **Timing** section, set Read wait to '1' and Write wait to '0', as previously described, make note of the Read and Write waveforms, you will need to design

hardware that implements these waveforms within your *vga_text_avl_interface.sv* module.

14. Now, we begin to add the relevant signals for this Memory Mapped slave port. Unlike previous ports, the MM Slave has multiple signals (read, write, chipselect, etc.) Let's start with the **address** signal, as before, click on *<<add signal>>* and set the **Name** to "AVL_ADDR", the **Signal Type** to **address**, **Width** to 10, and **Direction** to **input** in order to match what we declared in the top-level file *vga_text_avl_interface.sv* ("input logic [9:0] AVL_ADDR").
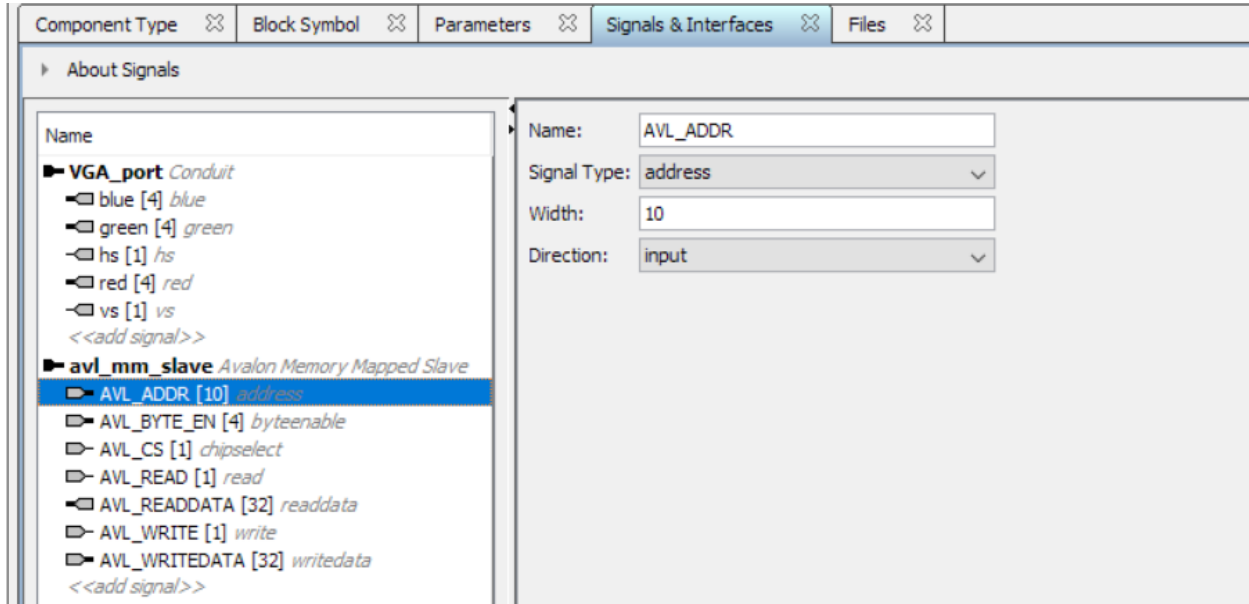

15. Add the **byteenable** signals next, click on *<<add signal>>* and set the **Name** to "AVL_BYTE_EN", the **Signal Type** to **byteenable**, **Width** to 4, and **Direction** to **input** once again to match what we declared in the top-level file *vga_text_avl_interface.sv* ("input logic [3:0] AVL_BYTE_EN").
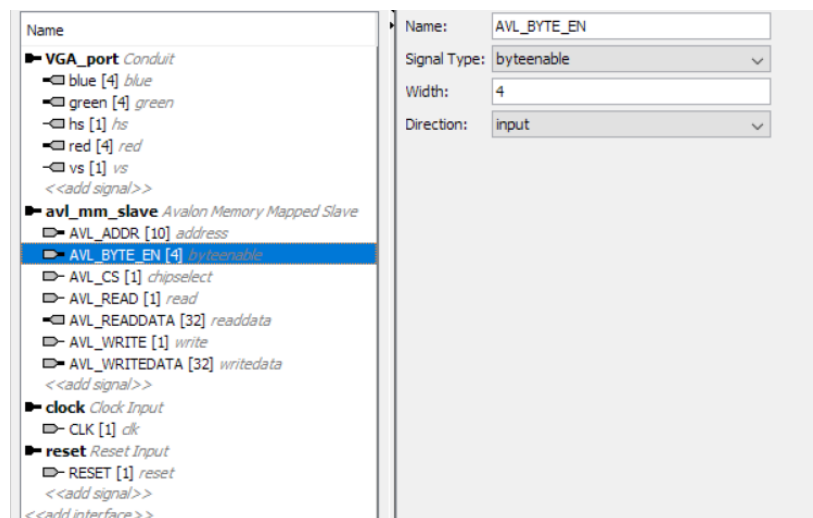
16. Complete the previous step the remaining signals for avl_mm_slave. Match the names, signal types, width, and direction for each signal. Be careful that **readdata** (AVL_READDATA) is an output unlike the others. When done, it should look like this:
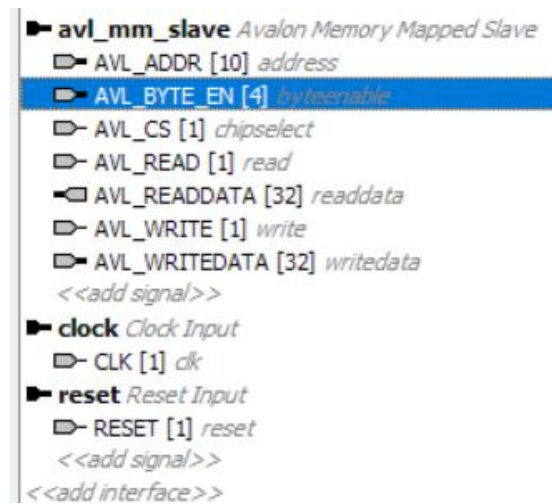
17. We are done with the avl_mm_slave. The last port to add is the exported conduit (VGA_port) to output the signals which go to the VGA port (red, green, blue, hs, vs). Click on *<<add interface>> -> Conduit* and choose the settings as follows:
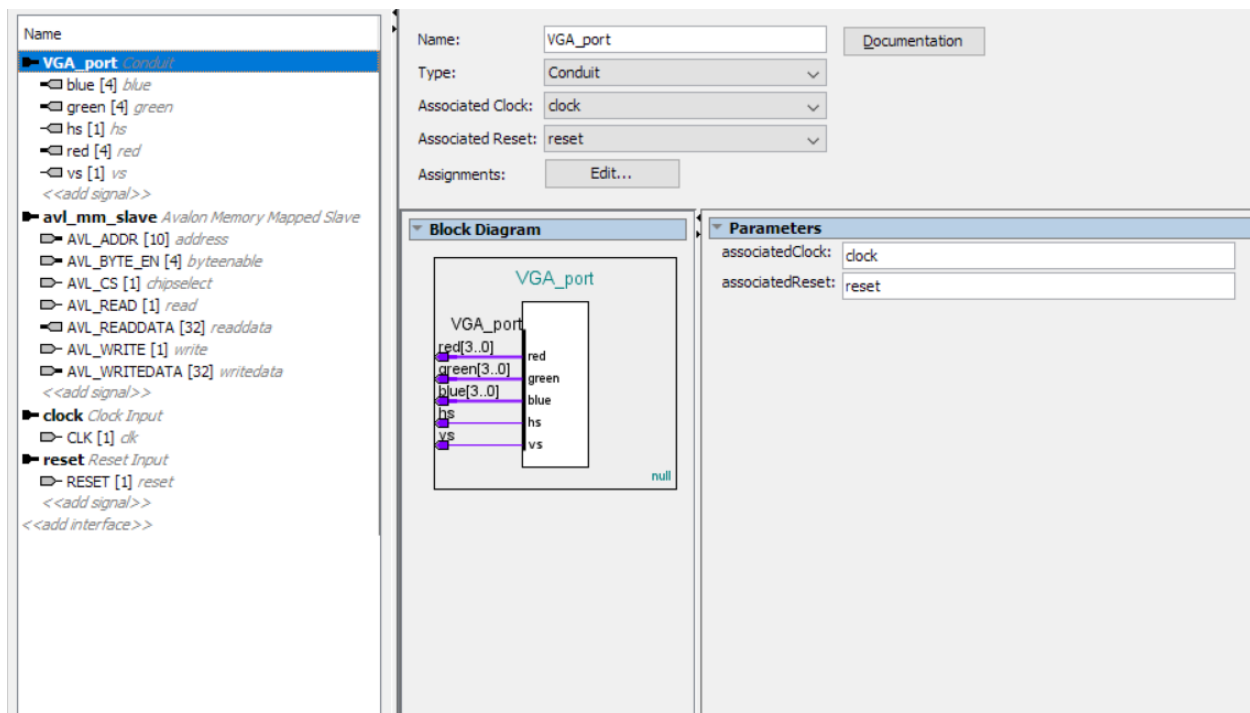
18. Add the signals for this conduit, namely, the 4-bit red, green, and blue outputs, and the 1-bit hs and vs outputs. Set **Name** to VGA_port, **Signal Type** to * (it will be changed automatically or may appear as "new_signal" depending on your Quartus version),

Width to either 1 or 4, and **Direction** to output as shown below.
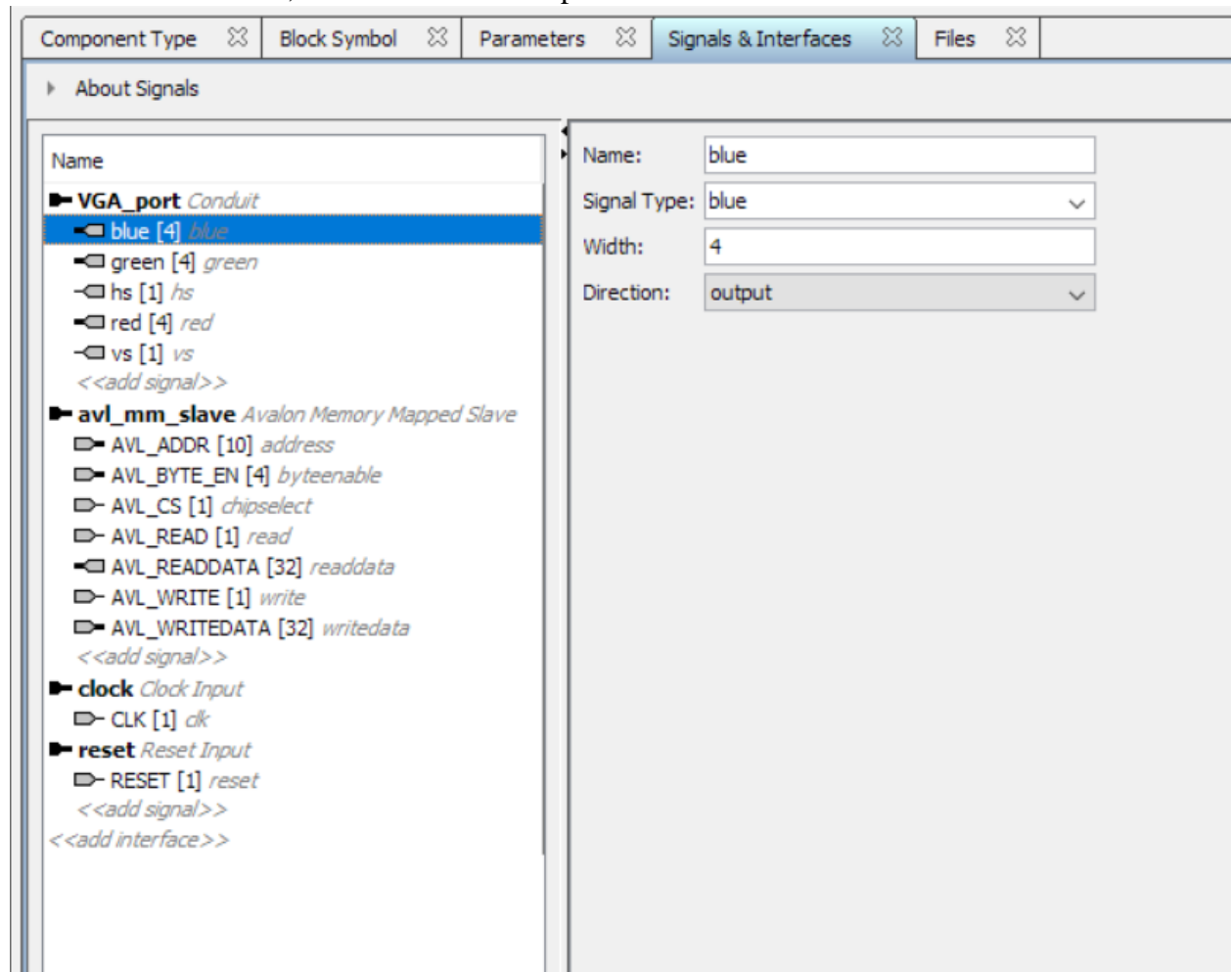
19. Go back to the Files tab, and click [Analyze Synthesis Files], you may get errors if you have bugs in your source files or if you instantiated other modules in vga_text_avl_interface.sv, in the latter case, click [Add File…] to add them (e.g. VGA.sv). Check that there are no more errors in the Message tab at the bottom and go back to the Signals & Interfaces tab to check that everything is still correct. If there are errors like "[*port name*] Interface must have an associate clock/reset", click on that port on the left tab, and set the Associated Clock or Reset to CLK and RESET respectively if they have been reset to none (this is a minor bug that happens in some versions of Quartus). If there are no errors, click on *Finish…* at the bottom right and click "Yes, Save" to save the generated TCL script.
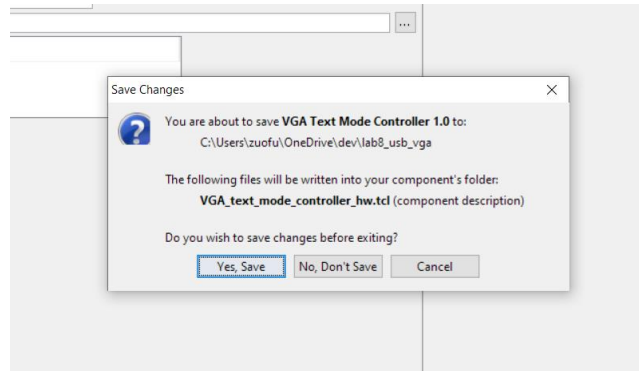
20. Now, in Platform Designer, your newly created VGA Text Mode Controller component should appear in IP Catalog, grouped under "ECE 385 Custom IPs". It can now be added to Platform Designer like other Intel provided IPs. If you made a mistake, you can right-click it and click Edit to make any changes needed such as increasing the version number.
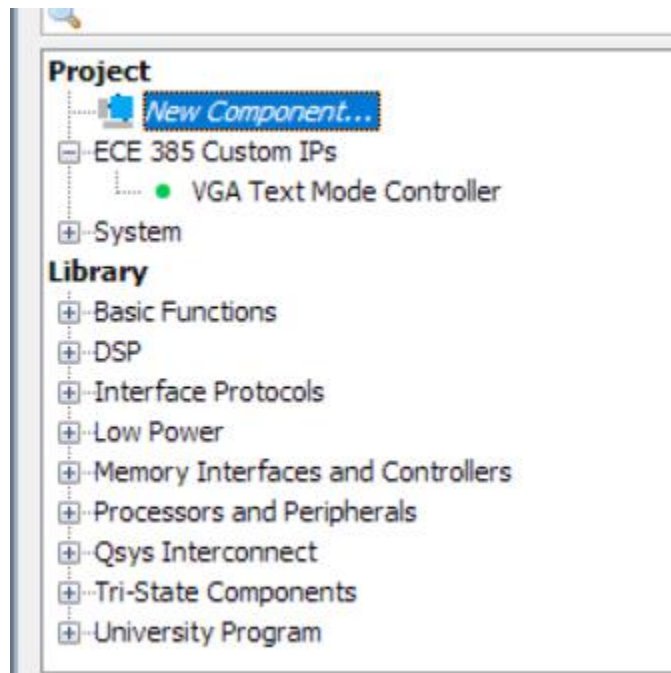
Finalizing your Platform Designer Design

Add your newly created component VGA Text Mode Controller to Platform Designer by double clicking it. We did not define any parameters, so just click Finish to add it to Platform Designer, rename it if you want to. Make the appropriate **CLK** and **RESET** connections, and most importantly, connect the **avl_mm_slave** to Nios II's **data_master** to allow Nios to access its registers through reads and writes. We have set the base address automatically; in this case it chose the range 0x0800 0000 to 0x0800 0FFF – make sure this range is consistent with 4096 byte addresses (1024 word addresses). You may start from a previous Platform Designer setup, for example, the USB + VGA one. Make sure to "export" the VGA_port conduit, in this case it is named **vga_port.**
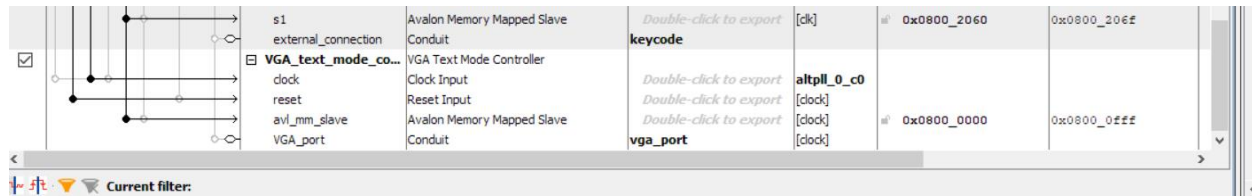
Finally, generate the HDL for your Platform Designer design and include the QIP in your project.

Recall that one of the main advantages of using hardware IPs is reusability. Using a standardized interface like Avalon allows your IP to be reused across different projects, even if other components change.

IMPORTANT: Whenever you change the SystemVerilog code in *vga_text_avl_interface.sv* to fix bugs or add things after this, you need to use Platform Designer to regenerate the HDL so that those changes take effect. (The actual file being compiled by Quartus is the one generated by Platform Designer located in labx\labx_soc\synthesis\submodules\vga_text_avl_interface.sv)

IMPORTANT: There is a bug in some versions of Quartus including 18.1 that causes it to misname your new Platform Designer component. If Platform Designer' generated top level Verilog file instantiates ***new_component*** for your ***vga_text_avl_interface*** module, close Platform Designer, go to your project folder and edit the file vga_text_mode_controller_hw.tcl with a text editor: under the file sets section, manually change the TOP_LEVEL property to "*vga_text_avl_interface*" as shown below, also check that the other lines are the same (unless your SystemVerilog file path is different). Then, open Platform Designer and update the version of your component (right click it under IP Catalog > Edit... and change version from 1.0 to 1.1 or any bigger number, then click Finish... to save), save your Platform Designer system and regenerate your HDL.

Alternatively, you can correct the instantiated module name in the Verilog file labx_soc.v from "new_component" to " *vga_text_avl_interface* ". This needs to be done each time after you generate HDL in Platform Designer.

```
#
# file sets
#
add_fileset QUARTUS_SYNTH QUARTUS_SYNTH "" ""
set_fileset_property QUARTUS_SYNTH TOP_LEVEL vga_text_avl_interface
set_fileset_property QUARTUS_SYNTH ENABLE_RELATIVE_INCLUDE_PATHS false
set_fileset_property QUARTUS_SYNTH ENABLE_FILE_OVERWRITE_MODE false
add_fileset_file vga_text_avl_interface.sv SYSTEM_VERILOG PATH
vga_text_avl_interface.sv TOP_LEVEL_FILE
```

Next Steps

It will be up to you to populate the rest of the vga_text_avl_interface.sv module with the hardware necessary to read and write the register space, as well as draw the text glyphs from font_rom.sv onto the VGA screen according to the VRAM (the layout of the VRAM is defined in the previous section). For this, you will most likely have to instantiate both font_rom.sv as well as VGA.sv in your module and draw the glyph sprites based on the DrawX and DrawY outputs from the VGA module. A recommended approach is as follows:

1. Design the logic which writes into the registers from the Avalon bus, respecting the ByteEnable signal. You will need to use signals CS, ADDR, BYTE_EN, WRITE, WRITEDATA as well as CLK and RESET.
2. Design the logic which reads back from the registers using the Avalon bus, you will need the same signals as above, except for using READ and READDATA (you may always provide all 32-bits on a readback irrespective of BYTE_EN).
3. The provided `textVGATest()` routine will test register writing and readback on the Avalon bus of your peripheral, printing the results on the UART console. Obviously nothing will be displayed until you've filled in the display code, but getting to this point is a good start.
4. Once you have some confidence the register data is correctly access through Avalon, start working on the video drawing portion. A simple sprite tutorial is provided which may assist you in understanding the problem (written from the context of the Color Mapper/Ball modules you've used before).
5. Run the full `textVGATest()` routine, which should then display moving text of different colors. Once you get to this point, you should have a working terminal output which is controlled by Avalon.

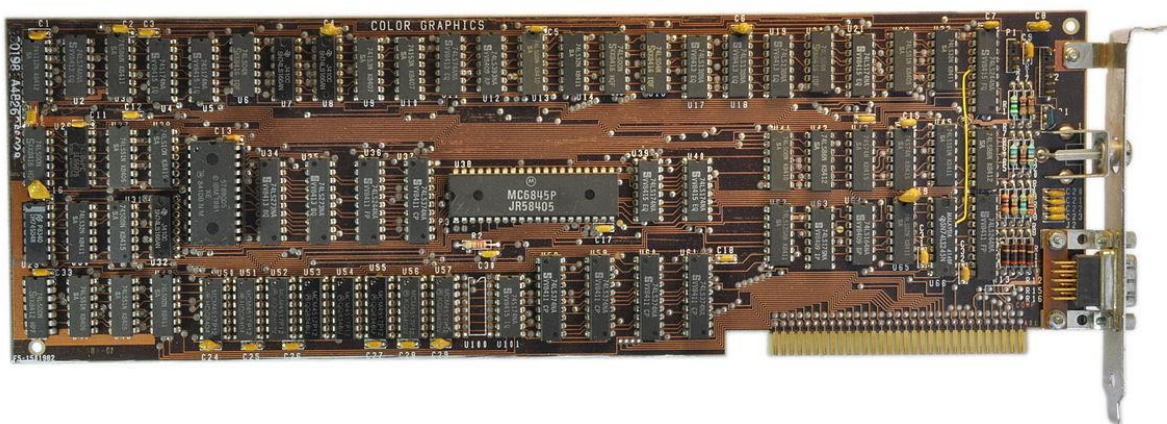**Week 2 Modifications – Color Graphics Adaptor**

Figure 5. IBM Color Graphics Adaptor

Starting with your working Week 1 code, you will extend the VRAM to support (per character) color text. This is equivalent to the text mode capability of the IBM Color Graphics Adaptor (CGA). Note that CGA also supports a pixel mapped graphical mode, which you will not be required to implement for this lab. A key difference between the IBM MGA and the IBM CGA (in addition to the obvious differences in circuitry to support color output) is that the CGA card (pictured) has additional VRAM to support color text and graphics. You will have to make a similar modification to support full color text for your Week 2 design.

You will modify the Platform Designer IP as follows, increasing the number of accessible registers 32-bit registers to 1200, to support the addition of per-character color attributes. This additional color information takes an additional 1 byte per character, so now 2 bytes are needed to represent a single character. The VRAM layout is now as follows, with one 32-bit word now holding only 2 characters worth of data:

Table 7. Bit Encoding for VRAM (Color Mode, Word Addresses 0x000-0x4AF)

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

Note the additional attributes, where:

FGD_IDXn is the **Foreground Color Index for character n**
BKG_IDXn is the **Background Color Index for character n**

Students who successfully completed Week 1 will realize that given the Week 1 design barely fit into FPGA registers; the Week 2 design, which has double the VRAM, will no longer fit into FPGA registers. A design challenge that students will need to solve for Week 2 is how to appropriately relocate the VRAM into on-chip memory. From the perspective of the Platform Designer Part Editor (e.g. Table 1), **only the address will change – it will increase from 10 bits to 12 bits to accommodate the additional VRAM and palette.**

Note: a key difference between registers an on-chip memory is that registers have as many input and output ports as there are storage bits, whereas a memory has a fixed number of input and output ports. Specifically, the on-chip memory blocks (M9K) blocks on the FPGA have two ports, each of which may be read or write. Therefore, only two memory locations may be accessed simultaneously.

The 'control' register from Week 1 will be removed, however you will reserve an additional 8 32-bit registers to support a 16-color palette (note that there will be 16 colors, each color's Red/Green/Blue representation will take up 12 bits, so two colors will be packed into each palette register). Because this data will need to be accessed to draw each pixel, it is recommended you keep the palette data in FPGA registers. To simplify decoding within your *vga_text_avl_interface* module, the palette will be located starting with word address **0x800**, so

you may use the most-significant bit of the word address (e.g., bit 11) to select between on-chip memory VRAM and your color palette in FPGA registers.

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

The memory layout of the color palette is as follows:

Table 9. Color Palette Organization (0x800-0x807)

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

For example, suppose color 0 is set to C0_R = 0xF, C0_G = 0x00, C0_B = 0x00 (e.g., bright red) and color 1 is set to C0_R = 0x0, C0_G = 0x0F, C0_B = 0x00 (e.g., bright green). If VRAM[0] contains: 0x03010110, will draw the following screen:

Table 10. Example image – VRAM[0x0] = 0x03010110

	0	1	2	3	4	5	6	7
0	Green on Red	Red on Green	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-

Next Steps (Week 2)

The second week tests your design prowess and so is more freeform. However, some suggested steps:

6. Back-up your Week 1 code.
7. Modify the Platform Designer part to extend the AVL_ADDR to 12 bits (the easiest way is to find your IP under 'ECE 385 Custom IPs' and right-click and select edit:
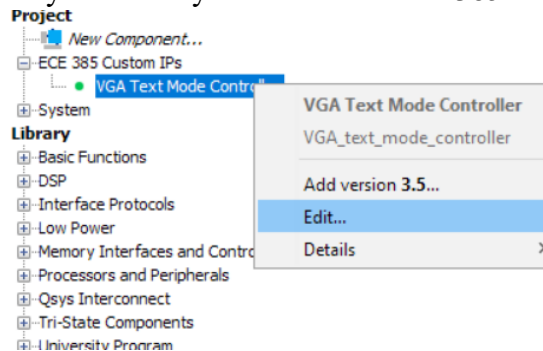
8. **Starting with your working Week 1 design**, modify the *vga_text_avl_interface* to support on-chip memory VRAM. It is recommended that initially you change nothing in the design besides moving the VRAM from registers to on-chip memory and getting your Week 1 design to work correctly again. It is not recommended you work on the per-character color support **until your Week 1 design works completely with VRAM relocated to on-chip memory.**

9. Once your Week 1 design works completely from on-chip memory (you may verify this using the compilation report, as the memory bits usage should go up while the registers usage should go significantly down), start working on the modifications necessary to support per-character color.

10. Download the Week 2 .zip archive, delete the text_mode_vga.c/h files, and replace with the text_mode_vga_color.c/h files.

11. Add the appropriate functionality in the `void setColorPalette(…)` function; this is used to initialize the color palette (the demo code uses the CGA standard colors).

12. Call the `textVGAColorScreenSaver()` function from main, which will draw a nice screensaver. Pat yourself on the back for completing the Week 2 design: