ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to the AXI Interface and HDMI/VGA Graphics

Please read this guide carefully and thoroughly as minor mistakes can impact the functionality of your entire project. The document starts with a description of the circuit, which you should understand before moving onto the tutorial (of just the IP customizer software) which starts on IAXI.10.

System Overview

For week 1 of this lab, you will create a simplified text mode graphics controller which is connected to the AXI4-Lite memory-mapped bus and supports 80 column text mode through the HDMI output. This is functionally very similar to the original IBM monochrome graphics adapter (MGA) which was included with the IBM PC 5150 from 1981:

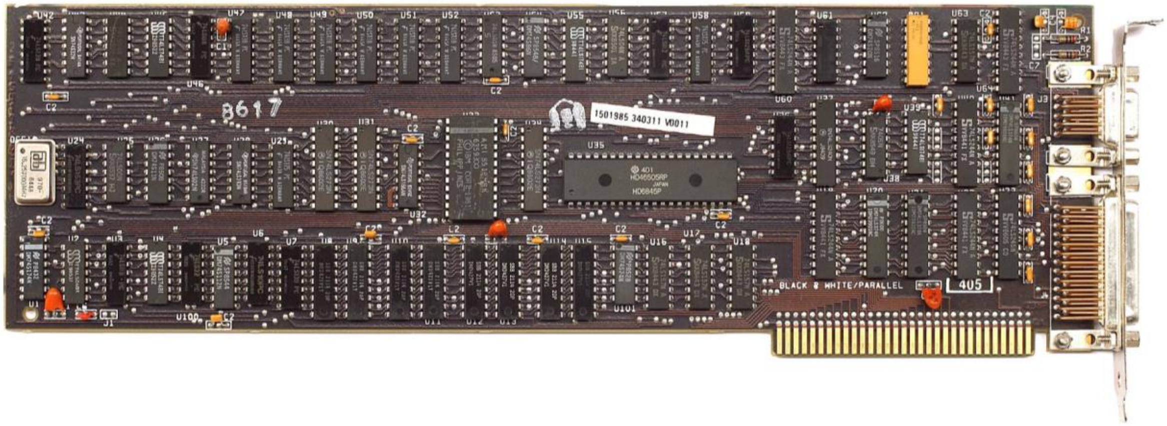
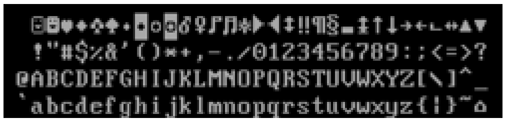


Figure 1. IBM Monochrome Graphics Adapter

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.



```
 ☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→←∟↔▲▼
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~⌂
```

Figure 2. IBM Codepage 437 Subset

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 AXI bus, allowing your MicroBlaze 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 below, note that you will have to design and construct the Graphics Controller IP in this lab.

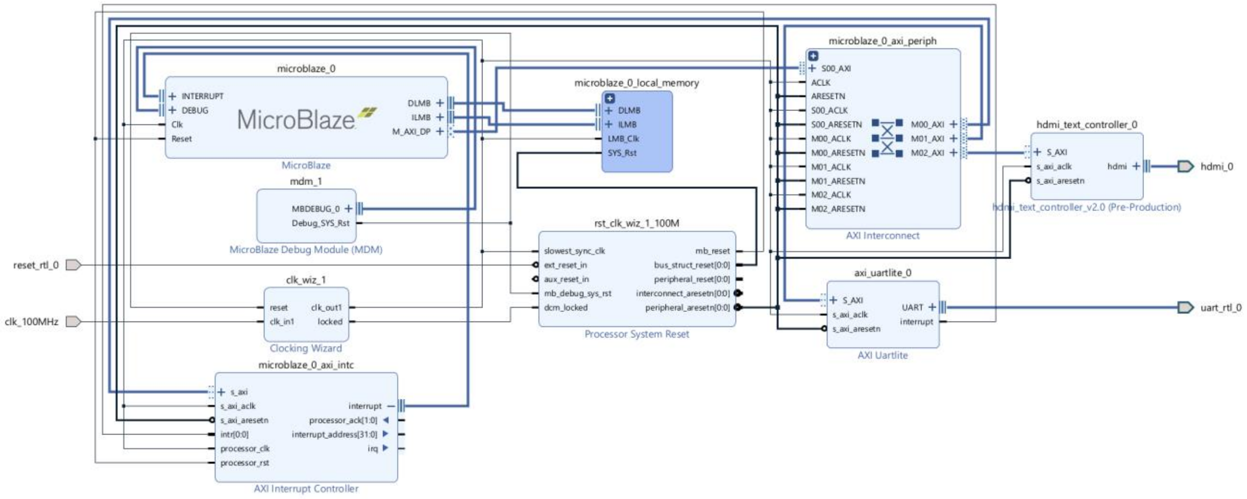


Figure 3. Overall System Design

To begin, start with your Platform Designer setup from the previous lab with the usual components like MicroBlaze, on-chip memory, and UART. In the next section, you will create your own *hdmi_text_controller* component and instantiate it within your new IP. You will then modify the provided AXI4-Lite template (which is provided by Vivado) to support the 2400 bytes of VRAM and a 32-bit control register. Once you have completed the AXI portion of the design, you will implement the graphics drawing logic to draw the characters using the previously provided VGA and HDMI blocks. 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 AXI-4 Lite Interface

The interface module *hdmi_text_controller_v1_0_.sv* will be the top-level file for the HDMI text mode core component on Vivado (note that your overall design should still use the top-level you previously generated). We have provided the input/output signals declaration for you. There is a clock input (axi_aclk), an active-high reset input (axi_aresetn), an exported conduit which consists of the HDMI port signals (hdmi_clk_n/p, hdmi_tx n/p) and finally an AXI4-Lite slave port which contains a bundle of signals whose specifications are given below.

The AXI-4 Lite slave port will complete read and write operations requested by its master, the MicroBlaze processor (read/write operations correspond to load/store instructions in the CPU). While the AXI-4 specifications provide many signals to use for its interface, we only need to use the subset of signals which correspond to the AXI-4 Lite specification. A description of the AXI-4 Lite bus is given below, much of it is referenced from Real Digital's AXI 4 Lite documentation: https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081 .

AXI4-Lite Interface Signals

The AXI4-Lite interface consists of five channels: Read Address, Read Data, Write Address, Write Data, and Write Response. An AXI4 read transaction using the Read Address and Data channels is shown in Figure 5. Similarly, an AXI4 write transaction using the Write Address, Data, and Response channels is shown in Figure 6. Note that these figures depict burst transfers, which AXI4-Lite is incapable of.

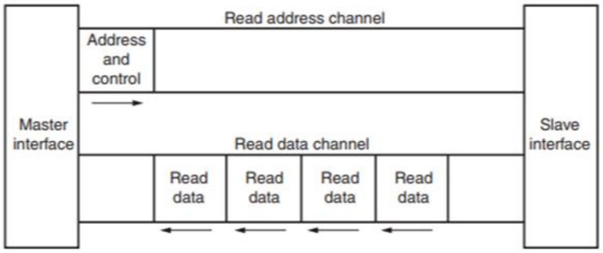


Figure 5 - AXI Read

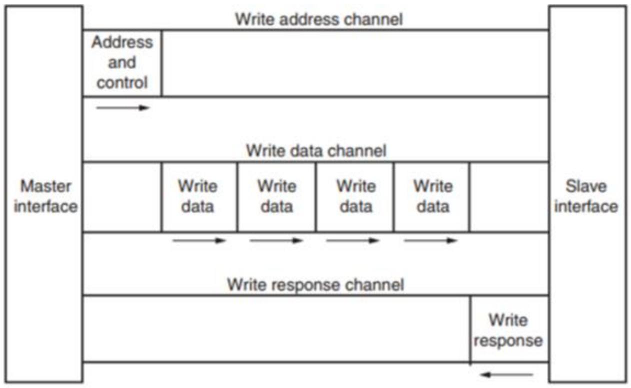


Figure 4 - AXI Write

The following table(s) provides a detailed explanation about each signal on AXI4-Lite Interface. Note there is a standard for the prefixes to each signal name, as they are associated with a channel. AW for the Write Address channel, W for the Write Data channel, B for the Write Response channel, AR for the Read Address channel, and R for the Read Data channel. A is used for global AXI signals (e.g., clock and reset), not associated with a channel.

Table 1 - Global AXI4 Interface Signals

Name	Direction	Width	Description
ACLK	M->S	**1**	Bus clock
ARESETN	M->S	**1**	Bus reset, active low

Table 2 - AXI4 Read Address Channel Signals

Name	Direction	Width	Description
ARADDR	M->S	**WIDTH**	Read address, usually 32-bit wide.
ARPROT	M->S	**3**	Protection type. Xilinx IP usually ignores it as a slave. As a master IP generates transactions with Normal, Secure, and Data attributes (000).
ARVALID	M->S	**1**	Read address valid. Master generates this signal when read address and the control signals are valid.
ARREADY	M<-S	**1**	Read address ready. Slave generates this signal when it can accept the read address and control signals.

Table 3 - Read Data Channel Signals

Name	Direction	Width	Description
RDATA	M<-S	**32**	Read Data
RRESP	M<-S	**2**	Read response. This signal indicates the status of data transfer.
RVALID	M<-S	**1**	Read valid. Slave generates this signal when read data is valid
RREADY	M->S	**1**	Read ready. Master generates this signal when it can accept the Read Data and response.

Table 4 - Write Address Channel Signals

Name	Direction	Width	Description
AWADDR	M->S	**WIDTH**	Write address, usually 32-bit wide.
AWPROT	M->S	**3**	Protection type. Xilinx IP usually ignores it as a slave. As a master IP generates transactions with Normal, Secure, and Data attributes (000).
AWVALID	M->S	**1**	Write address valid. Master generates this signal when write address and the control signals are valid.
AWREADY	M<-S	**1**	Write address ready. Slave generates this signal when it can accept the write address and control signals.

Table 5 - Write Data Channel Signals

Name	Direction	Width	Description
WDATA	M->S	**32**	Write data
WSTRB	M->S	**4**	Write strobes. 4-bit signal indicating which of the 4-bytes of write data. Slaves can choose to assume all bytes are valid, but you should not do this for this lab (otherwise you cannot change single characters in a string)

Table 6 - Write Response Channel Signals

Name	Direction	Width	Description
BRESP	M<-S	**2**	Write response. This signal indicates the status of the write transaction.
BVALID	M<-S	**1**	Write response valid. Slave generates this signal when the write response on the bus is valid.
BREADY	M->S	**1**	Response ready. Master generates this signal when it can accept a write response

AXI4-Lite Handshaking Process

All five transaction channels use the same VALID/READY handshake process to transfer address, data, and control information. This two-way flow control mechanism means both the master and slave can control the rate at which the information moves between master and slave. The information source generates the VALID signal to indicate when the address, data or control information is available. The information destination generates the READY signal to indicate that it can accept the information. The handshake completes if both VALID and READY signals in a channel are asserted during a rising clock edge.

AXI4-Lite Read Transaction

Below, the sequence for an AXI4-Lite read is shown:

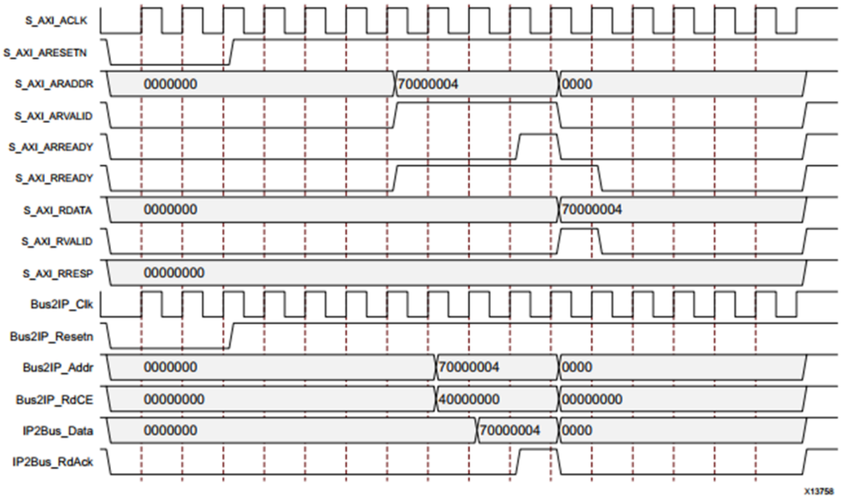


Figure 5 - AXI4-Lite Read Transaction

1. The Master puts an address on the Read Address channel as well as asserting ARVALID, indicating the address is valid, and RREADY, indicating the master is ready to receive data from the slave.
2. The Slave asserts ARREADY, indicating that it is ready to receive the address on the bus.
3. Since both ARVALID and ARREADY are asserted, on the next rising clock edge the handshake occurs, after this the master and slave de-assert ARVALID and the ARREADY, respectively. (At this point, the slave has received the requested address).
4. The slave puts the requested data on the Read Data channel and asserts RVALID, indicating the data in the channel is valid. The slave can also put a response on RRESP, though this does not occur here.
5. Since both RREADY and RVALID are asserted, the next rising clock edge completes the transaction. RREADY and RVALID can now be de-asserted.

Similarly, an AXI4-Lite write transaction operates as follows:

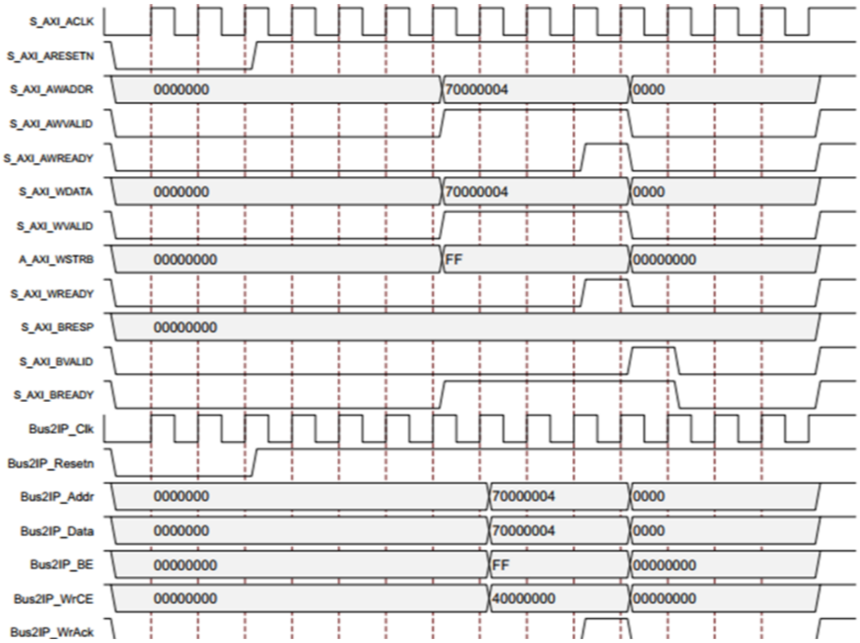


Figure 6 - AXI4-Lite Write Transaction

1. The Master puts an address on the Write Address channel and data on the Write data channel. At the same time, it asserts AWVALID and WVALID indicating the address and data on the respective channels is valid. BREADY is also asserted by the master, indicating it is

ready to receive a response.

2. The slave asserts AWREADY and WREADY on the Write Address and Write Data channels, respectively.
3. Since Valid and Ready signals are present on both the Write Address and Write Data channels, the handshakes on those channels occur and the associated Valid and Ready signals can be de-asserted. (After both handshakes occur, the slave has the write address and data)
4. The Slave asserts BVALID, indicating there is a valid response on the Write response channel. (In this case the response is 2’b00, that being ‘OKAY’).
5. The next rising clock edge completes the transaction, with both the Ready and Valid signals on the write response channel high.

The Handshakes on the Write Address and Write Data channel do not necessarily occur simultaneously (as they do in the shown transaction). However, the AXI4 specification states that both must occur before the slave can send a write response. Both Write Address and Write Data handshakes can occur independently or simultaneously, and no order is enforced, only that both must occur to complete the transaction.

The 2-bit response codes (RRESP and BRESP) are as follows:

Code	Description
2’b00	Normal access success or exclusive access failure. OKAY is the response that is used for most transactions. OKAY indicates that normal access has been successful. This response can also indicate that exclusive access has failed. An exclusive access is when more than one manager can access a subordinate at once, but the se managers cannot access the same memory range.
2’b01	Exclusive access is okay. EXOKAY indicates that either the read or write portion of an exclusive access has been successful.
2’b10	Subordinate error. SLVERR is used when the access has reached the subordinate successfully, but the subordinate wants to return an error condition to the originating manager. This indicates an unsuccessful transaction. For example, when there is an unsupported transfer size attempted, or a write access attempted to read-only location.
2’b11	Decode error. DECERR is often generated by an interconnect component to indicate that there is no subordinate at the transaction address.

Note that the data width of 32-bit and address width of 16-bit are chosen for this lab. We are using 32-bit data widths because they match the data width of the MicroBlaze, a 32-bit processor. As for the 16-bit address, which gives $2^{16} = 65536$ locations, this is the default when creating a Vivado IP, and sufficient for our purposes for both weeks. At the minimum, the VRAM (2400 bytes) as well as the 32-bit control register need to be memory mapped. Each address is a byte address in AXI, so for the Week 1 lab, only 12 address bits are needed – with the remainder being unused. Note that despite the presence of the write-strobe signals, AXI4 addresses are always assumed to be byte addresses.

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 604 registers, each 32-bit, that hold the values being read and written. In addition, write strobe enables determines the bytes being written according to the table below (note that all 16 combinations of write strobe are valid, though only some are shown).

Table 7. Write Strobe Description

WSTRB [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 to serve as your VRAM and control register. Internally, you should allow the AXI4-Lite bus to read and write those 604 registers, according to the memory map below.

Table 8. 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 - 0x25B	0x0000 0960 – 0x0000 970	Control registers (4 x 32-bits)
0x25C - 0xFFF	0x0000 0971 - 0x0000 FFFF	Unused but reserved by Vivado

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

Bit	31	30-24	23	22-16	15	14-8	7	6-0
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 0x0101038E, then the character starting at position column = 4, row = 1 (assuming we start at column/row 0) should display:

Table 10. Example image – VRAM[0x15] = 0x0101038E

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	-	-	♫	♥	☺	☺
2	-	-	-	-	-	-	-	-

Only the first 8 columns and 3 rows are shown.

Note that character 0x0E (double 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 11. Bit Encoding for Control Registers (Word Address 0x258-25B)

Address	**31-28**	**27-24**	**23-20**	**19-16**	**15-12**	**11-8**	**7-4**	**3-0**
0x258	UNUSED	FGD_R	FGD_G	FGD_B	UNUSED	BKG_R	BKG_G	BKG_B
0x259	FRAME_COUNTER							
0x25A	CURRENT_DRAW_X							
0x25B	CURRENT_DRAW_Y							

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.
FRAME_COUNTER = Read only register which reflects the number of v-sync pulses the controller has generated since AXI reset
CURRENT_DRAW_X and CURRENT_DRAW_Y, the current values of the DrawX and DrawY signals from the VGA controller. Read only registers from AXI.

Although the first control register (color 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 the VRAM and control register 0x248 (the color register) need to have write support, while the remaining three control registers are read only. These are useful for the synchronization of software events with the video frame, for example, detecting when a new frame has occurred to compute scrolling or detecting when the current raster is in the vertical blanking interval to change the screen without artifacts. The DrawX and DrawY signals may come directly from the VGA controller, however, the frame counter requires you to implement some additional hardware within your IP. You can either use a counter module which is clocked by the v-sync signal (as in the ball module of the previous lab) or design a falling-edge detection circuit in conjunction with a counter to count the frames synchronously.

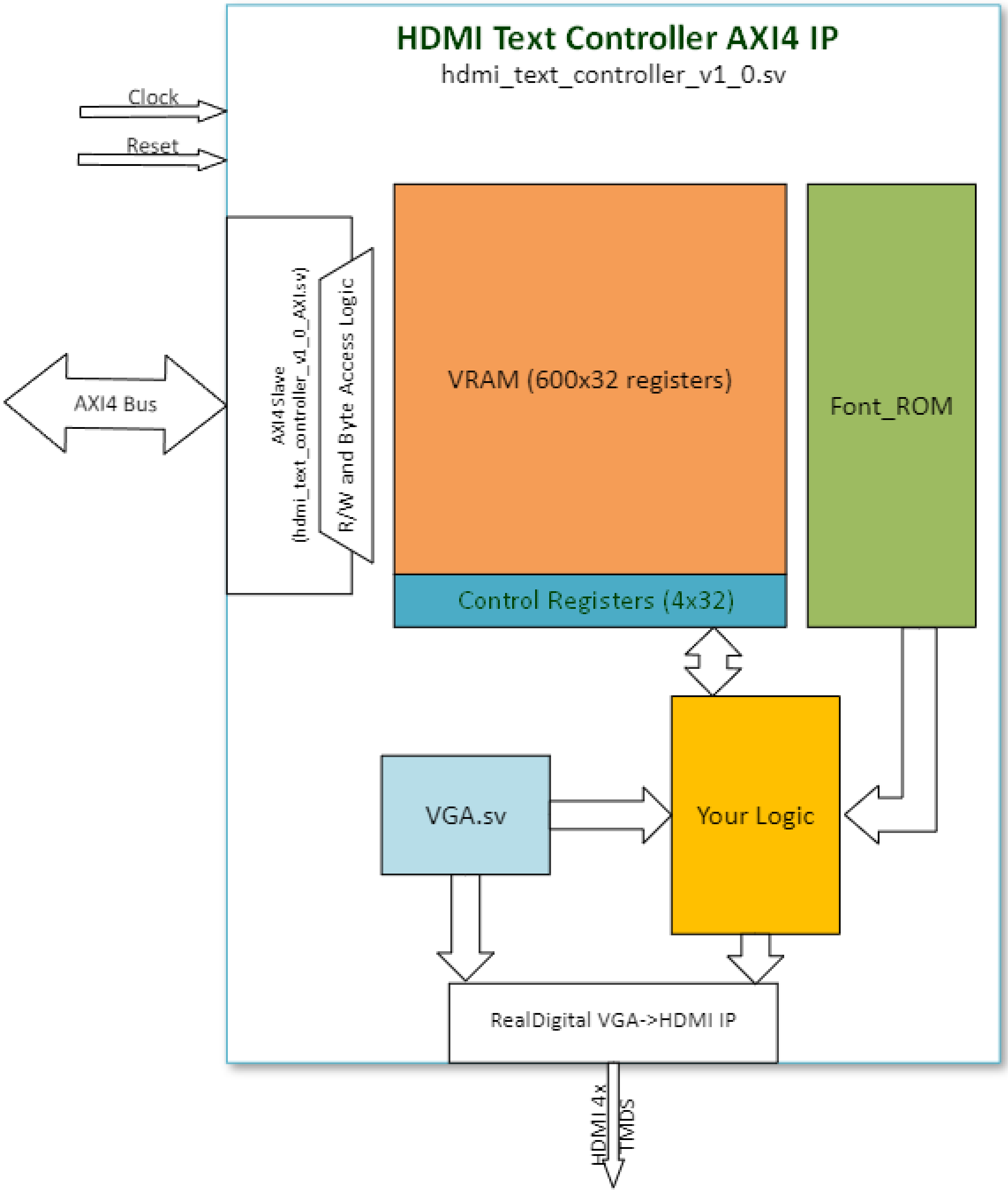
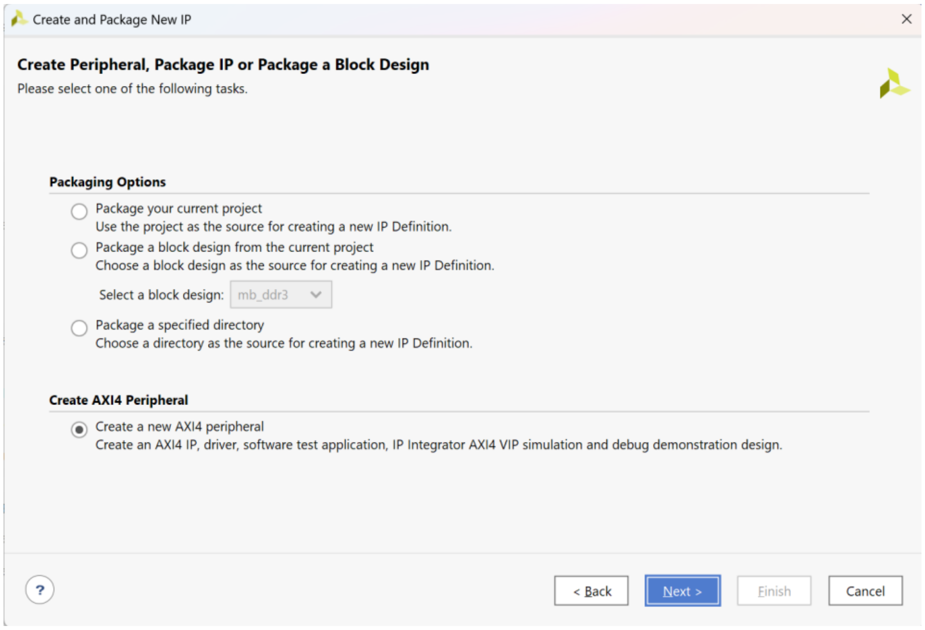


Figure 7. HDMI Text Mode Interface Incomplete Block Diagram

Creating a AXI4-Lite IP

To start implementation of the HDMI Text Controller, follow these steps to create the IP example design and add it to the Vivado IP Catalog:

1. Launch Vivado and go to Tools -> Create and Package New IP
2. *Click Next and select Create a new AXI4 Peripheral…*



Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.

Packaging Options

○ Package your current project
Use the project as the source for creating a new IP Definition.

○ Package a block design from the current project
Choose a block design as the source for creating a new IP Definition.

Select a block design: mb_ddr3

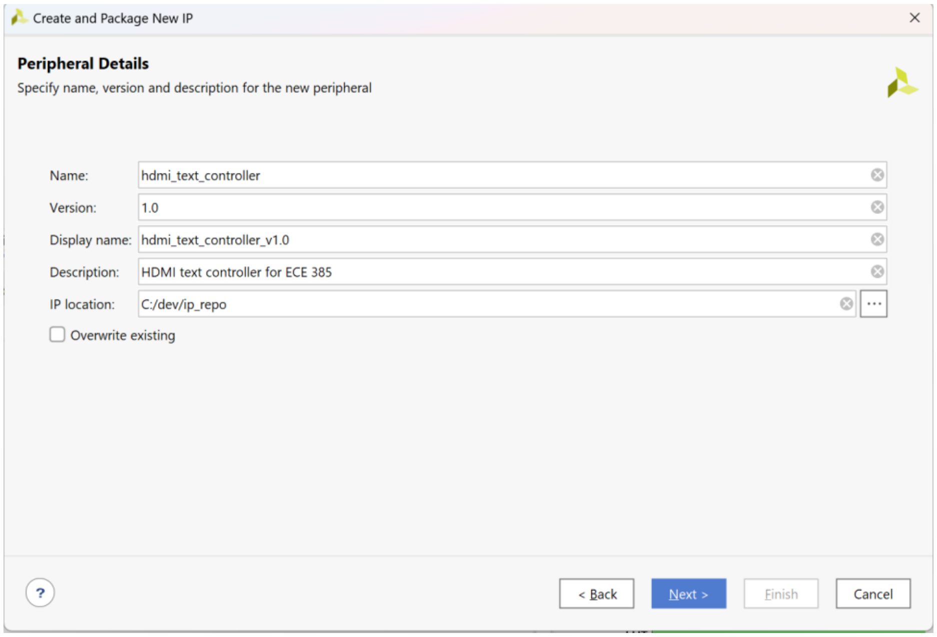
○ Package a specified directory
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

● Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

Figure 8 - Create Peripheral, Package IP…

3. Enter the Name and Display name for your component (make sure this is spelled and capitalized the same as in the screenshot). It is good practice to also keep track of the version of your IP, increment it when you make major changes (e.g., week 2 of this lab). Give your IP a brief description, although the exact text here is not important. Make note of the IP location, this should be a location on a local drive that only you have access to (e.g., C:\Users\ljames23\ece385\ip_repo). Also make sure you have access to the Real Digital HDMI IP from the previous lab.



Peripheral Details

Specify name, version and description for the new peripheral

Name: hdmi_text_controller

Version: 1.0

Display name: hdmi_text_controller_v1.0

Description: HDMI text controller for ECE 385

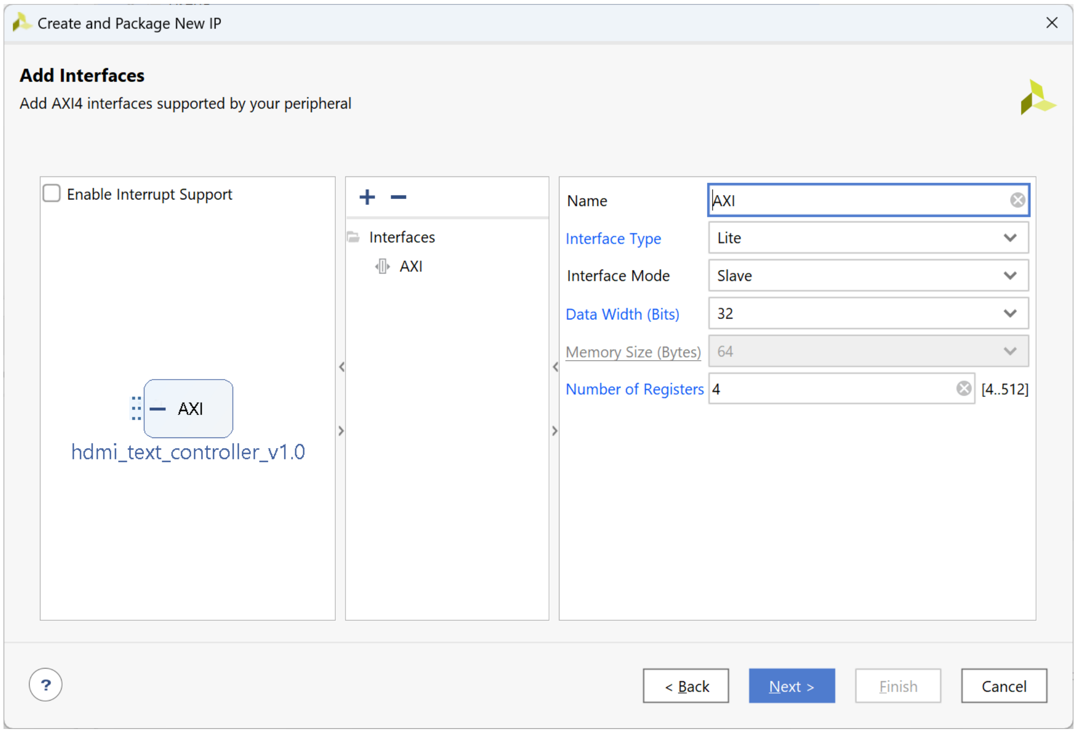
IP location: C:/dev/ip_repo

□ Overwrite existing

Figure 9 - Create New IP

4. Click next and fill in the following, most of the entries can be kept as their defaults, though the AXI slave port should be named as “AXI” to be consistent with the provided files (and other IPs in Vivado). Note that Vivado can generate the AXI interface logic automatically, but only for up to 512 registers (which is not even enough for Week 1). You will replace that portion of the code with your own modifications to allow for decoding 604 32-bit registers (Week 1) and 1200 32-bit words of BRAM (Week 2). For now, we will configure the template code to support 4 registers to use as a base for modification.

Note: Although you might think that the peripheral only allowing 4 32-bit registers implies that Vivado only reserves 4 bits (16 bytes worth) of address space, it seems that the minimum AXI4 peripheral will still reserve 16 address bits (64K bytes of address)



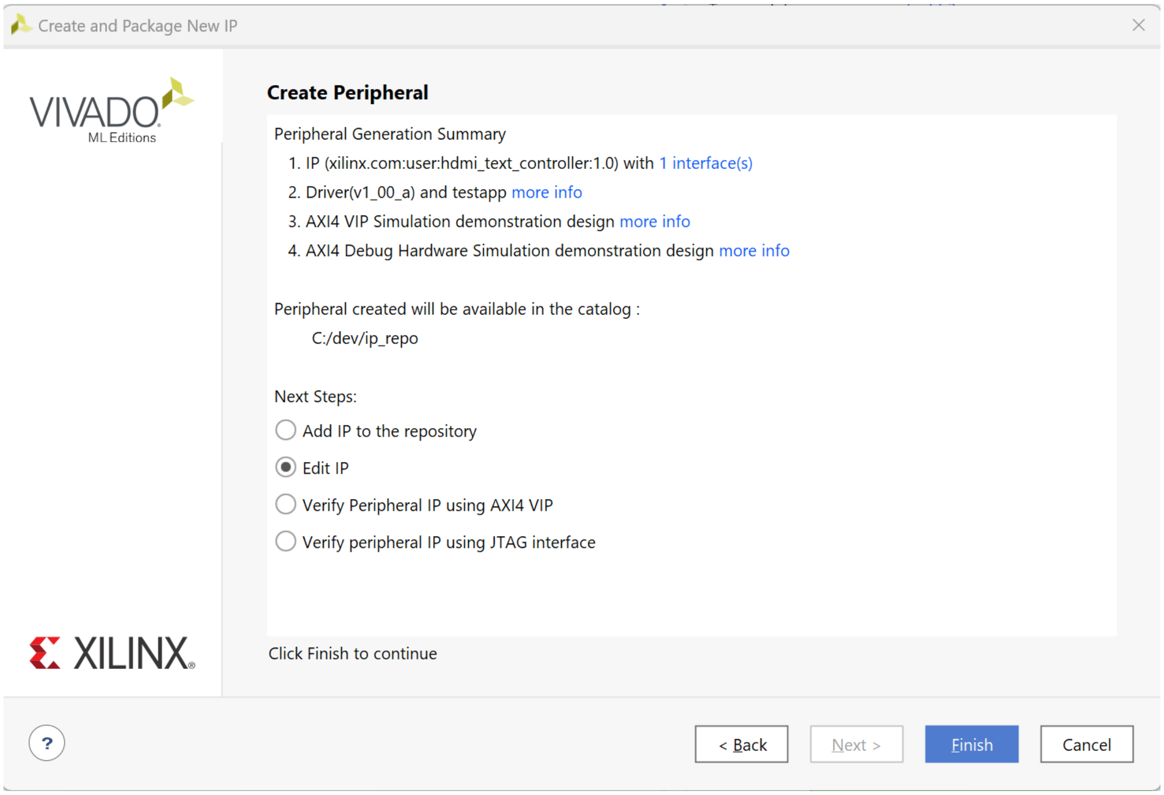
Create and Package New IP
Add Interfaces
Add AXI4 interfaces supported by your peripheral
+ −
Interfaces
AXI
AXI
hdmi_text_controller_v1.0
?
< Back
Next >
Finish
Cancel

□ Enable Interrupt Support

Name	AXI	
Interface Type	Lite	
Interface Mode	Slave	
Data Width (Bits)	32	
Memory Size (Bytes)	64	
Number of Registers	4	[4..512]

Figure 10 - AXI Interface Configuration

Go ahead and click Next, and you will be asked what the next step is:



Create Peripheral

Peripheral Generation Summary

1. IP (xilinx.com:user:hdmi_text_controller:1.0) with 1 interface(s)
2. Driver(v1_00_a) and testapp more info
3. AXI4 VIP Simulation demonstration design more info
4. AXI4 Debug Hardware Simulation demonstration design more info

Peripheral created will be available in the catalog :

C:/dev/ip_repo

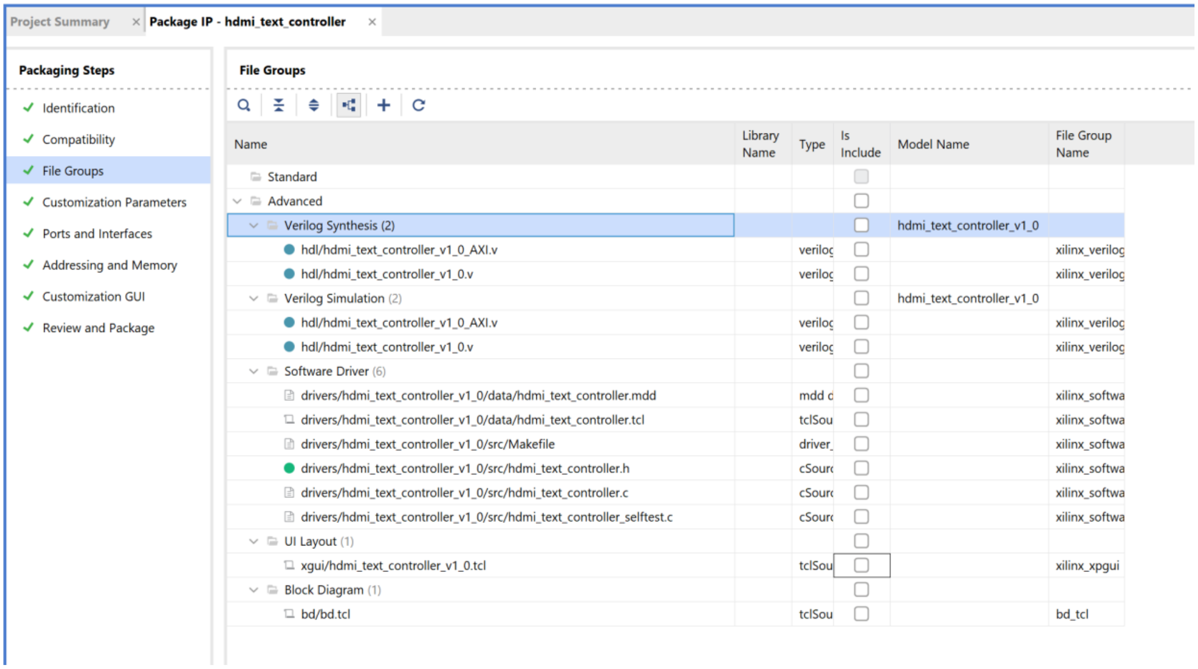
Next Steps:

○ Add IP to the repository
● Edit IP
○ Verify Peripheral IP using AXI4 VIP
○ Verify peripheral IP using JTAG interface

Click Finish to continue

Figure 11 - Create IP Wizard

5. Now select the “Edit IP” option and Finish. This will pop up a new temporary project “edit_hdmi_text_controller”. Note that the project that Vivado creates is temporary (in fact, it will be deleted once you package the IP), what’s important is the IP that gets packaged into the IP repository. Also note that to return to the IP packager screen, you must select the IP from “IP Catalog” (on left side under Flow Navigator) and then right click on your IP and click “Edit in IP Packager” this will re-create the temporary project to allow you to edit the files inside (which you will likely have to do many times to complete the lab.
6. Your ‘edit’ project will now have an IP Packager module. Click on this module if it’s not already selected and go over to “File Groups”. This contains all the files that will be extracted from the project and packaged together to form the actual IP (which will go into the MicroBlaze block design). Notice there are lots of files which are already pre-generated, including 2 Verilog files (one for the IP itself, one for the AXI controller), and some .c and .h files under the “Software Driver” category. In addition, there are several tcl scripts which describe the graphical portions of the IP you are creating. If you wanted to be fancy, you could modify the scripts to give your IP a custom ‘wizard’ style UI when customizing, or a custom block diagram picture (this is how the MicroBlaze CPU has a logo in the block diagram and how it has a graphical configuration panel).



Packaging Steps

- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✓ Customization Parameters
- ✓ Ports and Interfaces
- ✓ Addressing and Memory
- ✓ Customization GUI
- ✓ Review and Package

File Groups

Name	Library Name	Type	Is Include	Model Name	File Group Name
Standard			☐		
Advanced			☐		
Verilog Synthesis (2)			☐	hdmi_text_controller_v1_0	
hdl/hdmi_text_controller_v1_0_AXI.v		veriloc	☐		xilinx_verilog
hdl/hdmi_text_controller_v1_0.v		veriloc	☐		xilinx_verilog
Verilog Simulation (2)			☐	hdmi_text_controller_v1_0	
hdl/hdmi_text_controller_v1_0_AXI.v		veriloc	☐		xilinx_verilog
hdl/hdmi_text_controller_v1_0.v		veriloc	☐		xilinx_verilog
Software Driver (6)			☐		
drivers/hdmi_text_controller_v1_0/data/hdmi_text_controller.mdd		mdd c	☐		xilinx_softwa
drivers/hdmi_text_controller_v1_0/data/hdmi_text_controller.tcl		tclSou	☐		xilinx_softwa
drivers/hdmi_text_controller_v1_0/src/Makefile		driver_	☐		xilinx_softwa
drivers/hdmi_text_controller_v1_0/src/hdmi_text_controller.h		cSourc	☐		xilinx_softwa
drivers/hdmi_text_controller_v1_0/src/hdmi_text_controller.c		cSourc	☐		xilinx_softwa
drivers/hdmi_text_controller_v1_0/src/hdmi_text_controller_selftest.c		cSourc	☐		xilinx_softwa
UI Layout (1)			☐		
xgui/hdmi_text_controller_v1_0.tcl		tclSou	☐		xilinx_xpgui
Block Diagram (1)			☐		
bd/bd.tcl		tclSou	☐		bd_tcl

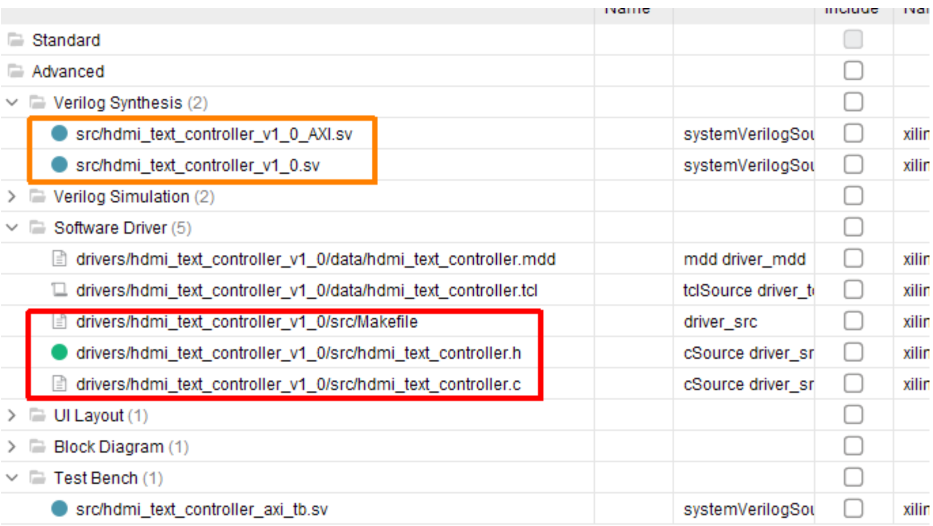
Figure 12 - IP Packager File Groups

7. For Week 1, you are provided with the complete software driver which includes some basic AXI read/write tests, as well as some graphical test routines. Therefore, delete the provided makefile, both c files, and the .h file, and add in their replacements from the provided files. Make sure you check “Copy source into the IP Directory”, otherwise your IP will refer to files all over your path, which causes lots of problems (e.g., if your files are in a temporary directory or in your Downloads directory, etc.). Note, the provided makefile is almost identical to the auto-generated one but fixes the following bugs in Vivado on Windows machines relating to wildcards (*).

   Vivado Issue: 000034569
   Vivado Issue: 75527

8. You will also note that Vivado provides 2 Verilog (.v) files which create an example AXI peripheral. Together, they create a memory mapped peripheral which can read and write into 4, 32-bit registers via AXI (as you previously filled out in the GUI Wizard). However, since you have been working with SystemVerilog in this course, we have provided you with the equivalent files re-written into SystemVerilog. Remove the provided Verilog (.v) files from the IP Packager and add in the provided SystemVerilog equivalents. Remember to check “Copy source into the IP Directory”, as before.

9. The resulting file groups tab should look as shown in the next Figure (Figure 13). Note that the new files are shown in orange, while the replaced files (that have the same name as the auto-generated files) are shown in red. As you complete the assignment, more files will be added under “Verilog Synthesis” (for example, the font_rom.sv, vga.sv), but for now this will be acceptable for the minimum setup to configure the IP. Notice hdmi_text_controller_selftest.c, which is an automatically generated file should be deleted. Also note the proper relative subdirectory structure of the files, fix the directory structure in Windows Explorer if it is incorrect and re-add the files.

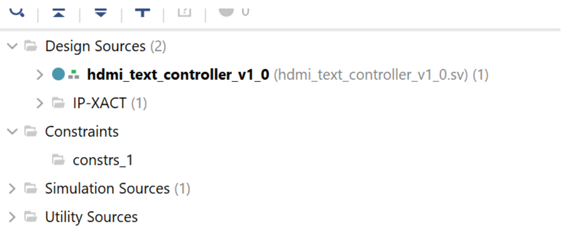


	Name		Include	Nai
Standard			□	
Advanced			□	
Verilog Synthesis (2)			□	
src/hdmi_text_controller_v1_0_AXI.sv		systemVerilogSou	□	xilir
src/hdmi_text_controller_v1_0.sv		systemVerilogSou	□	xilir
Verilog Simulation (2)			□	
Software Driver (5)			□	
drivers/hdmi_text_controller_v1_0/data/hdmi_text_controller.mdd		mdd driver_mdd	□	xilir
drivers/hdmi_text_controller_v1_0/data/hdmi_text_controller.tcl		tclSource driver_t	□	xilir
drivers/hdmi_text_controller_v1_0/src/Makefile		driver_src	□	xilir
drivers/hdmi_text_controller_v1_0/src/hdmi_text_controller.h		cSource driver_sr	□	xilir
drivers/hdmi_text_controller_v1_0/src/hdmi_text_controller.c		cSource driver_sr	□	xilir
UI Layout (1)			□	
Block Diagram (1)			□	
Test Bench (1)			□	
src/hdmi_text_controller_axi_tb.sv		systemVerilogSou	□	xilir

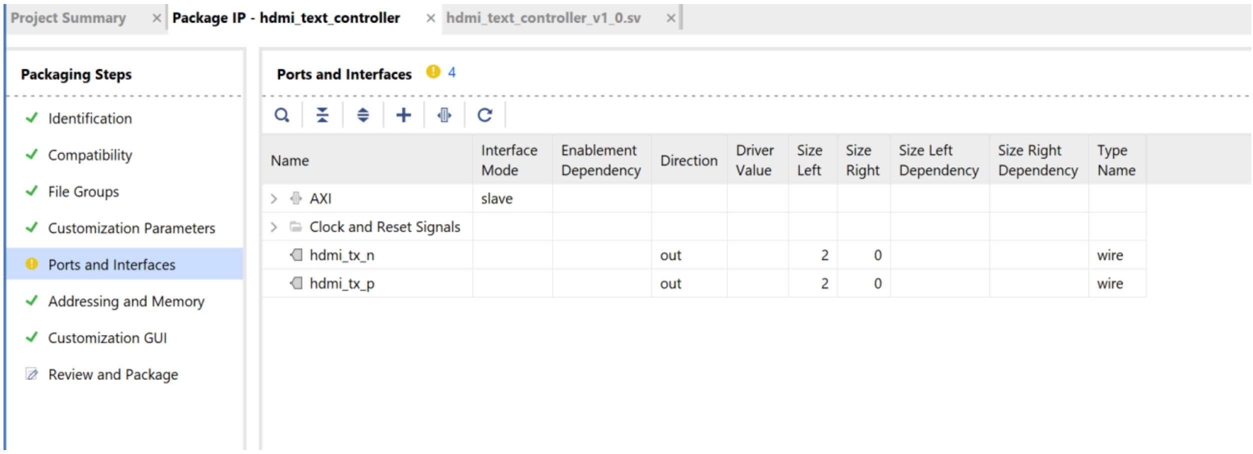
Figure 13 - File Groups after replacing provided files.

10. Our HDMI Text Controller IP also needs to have a HDMI output, so we will add that interface. Switch over to the Ports and Interfaces panel. If there is a warning which asks you to merge changes, click on it and merge the changes. If you do not get the option to merge changes, see the side note on the right side. If you do automatically get the note to merge changes, go ahead and click on that option and the additional signals should appear as in Figure 14. This will add any new inputs and outputs in the IP-top level to the IP graphical interface. Since the provided hdmi_text_controller_v1_0.sv file has ports for the HDMI output, we need to tell IP Packager about the HDMI port so that it appears correctly when we go to add the new IP to your block design. Switch over to Ports and Interfaces and click the "+" symbol.

*Sometimes Vivado doesn't properly update the IP's top-level (which should be *hdmi_text_controller_v1_0.sv*) if you've changed the files via the IP packager. If this happens to you, that means the project will still try to open the Verilog file instead of the SystemVerilog file as it should (see below):



In this case, you should manually delete the Verilog files from the /hdl directory of the IP and add the SystemVerilog files (if you've already imported them via the IP Packager, they will be under /src within the IP). Note that because you've already copied them into your IP directory, you will need to uncheck "Copy source into the IP Directory". You will then get the warning to merge changes, and the IP should now have the correct .SV files.



Project Summary × | Package IP - hdmi_text_controller × | hdmi_text_controller_v1_0.sv ×

Packaging Steps

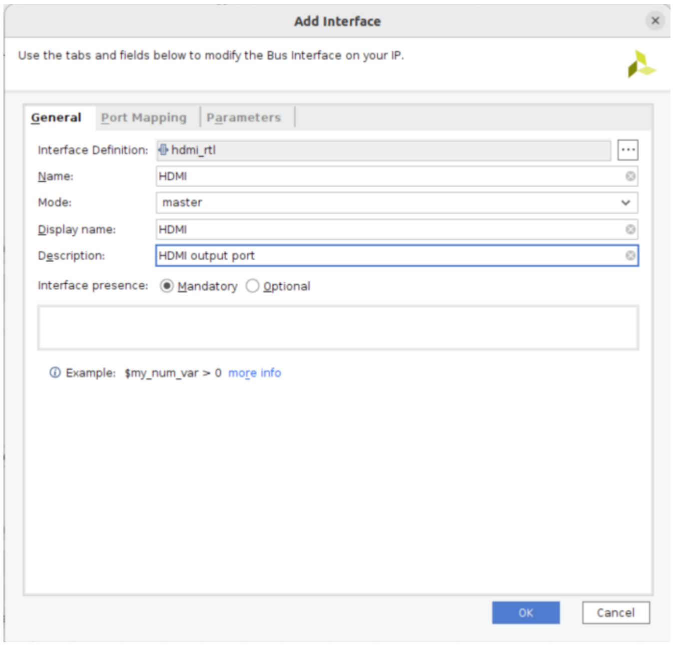
- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✓ Customization Parameters
- ● Ports and Interfaces
- ✓ Addressing and Memory
- ✓ Customization GUI
- Review and Package

Ports and Interfaces ● 4

Name	Interface Mode	Enablement Dependency	Direction	Driver Value	Size Left	Size Right	Size Left Dependency	Size Right Dependency	Type Name
> AXI	slave								
> Clock and Reset Signals									
hdmi_tx_n			out		2	0			wire
hdmi_tx_p			out		2	0			wire

Figure 14 - Ports and Interfaces

Then fill in the following on the first tab. This creates the graphical connection which is an output to your module within the block design:



Add Interface

Use the tabs and fields below to modify the Bus Interface on your IP.

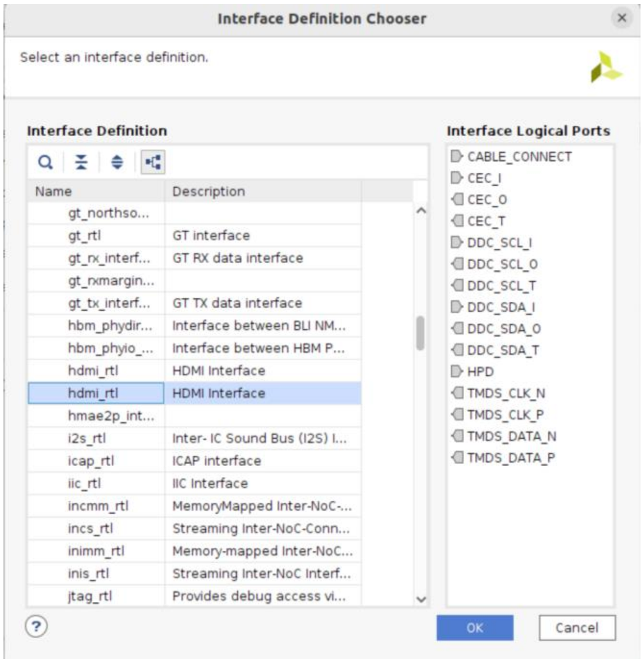
General | Port Mapping | Parameters

Interface Definition:	hdmi_rtl
Name:	HDMI
Mode:	master
Display name:	HDMI
Description:	HDMI output port
Interface presence:	● Mandatory ○ Optional

ⓘ Example: $my_num_var > 0 more info

Figure 15 - Add Interface

Note that the Vivado already defines the HDMI interface and by default it is located under “Advanced” – note there are two versions, we want the version which has the TMDS pairs.



Interface Definition Chooser

Select an interface definition.

Interface Definition

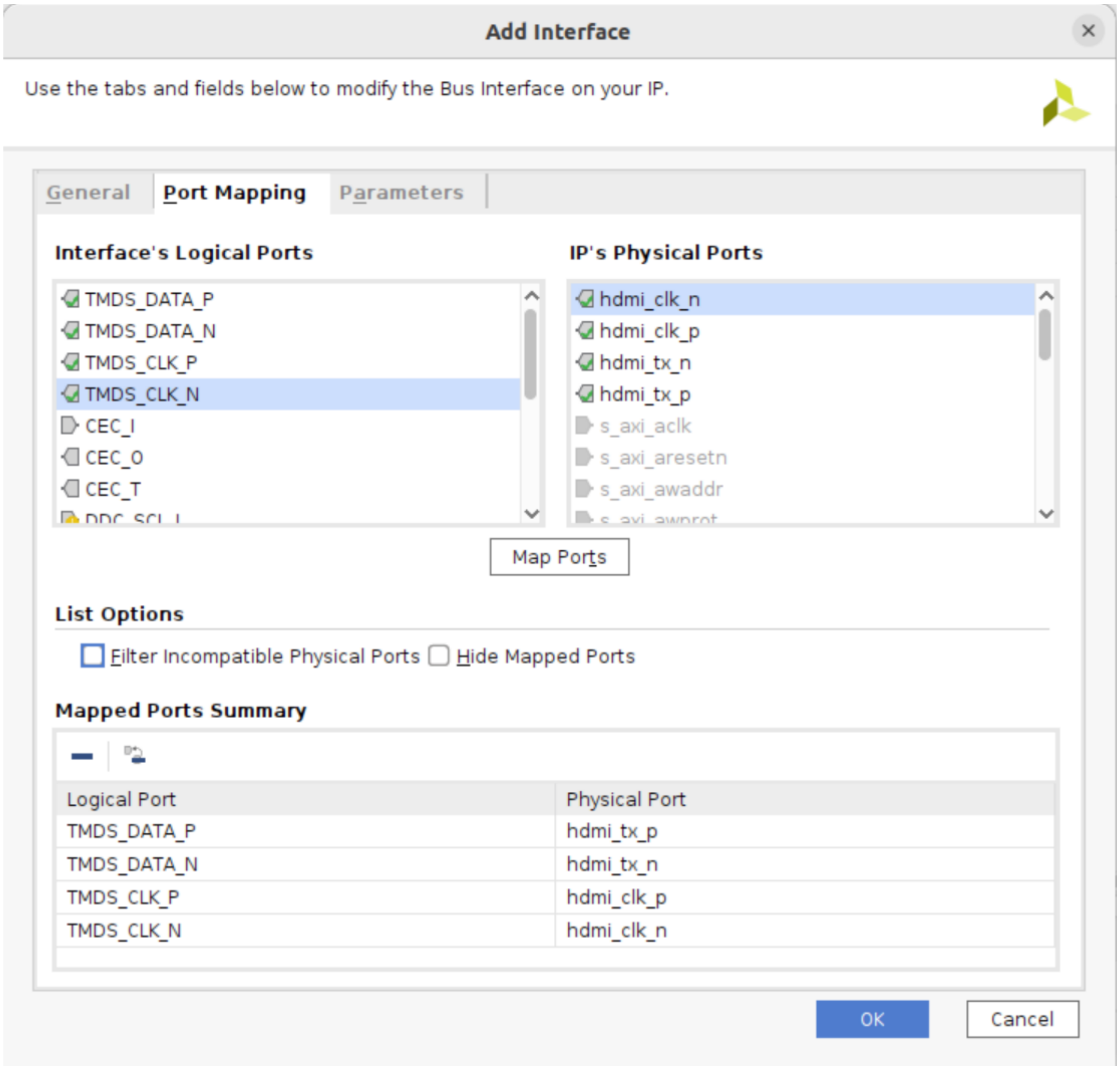
Name	Description
gt_northso...	
gt_rtl	GT interface
gt_rx_interf...	GT RX data interface
gt_rxmargin...	
gt_tx_interf...	GT TX data interface
hbm_phydir...	Interface between BLI NM...
hbm_phyio_...	Interface between HBM P...
hdmi_rtl	HDMI Interface
hdmi_rtl	HDMI Interface
hmae2p_int...	
i2s_rtl	Inter- IC Sound Bus (I2S) I...
icap_rtl	ICAP interface
iic_rtl	IIC Interface
incmm_rtl	MemoryMapped Inter-NoC-...
incs_rtl	Streaming Inter-NoC-Conn...
inimm_rtl	Memory-mapped Inter-NoC...
inis_rtl	Streaming Inter-NoC Interf...
jtag_rtl	Provides debug access vi...

Interface Logical Ports

- CABLE_CONNECT
- CEC_I
- CEC_O
- CEC_T
- DDC_SCL_I
- DDC_SCL_O
- DDC_SCL_T
- DDC_SDA_I
- DDC_SDA_O
- DDC_SDA_T
- HPD
- TMDS_CLK_N
- TMDS_CLK_P
- TMDS_DATA_N
- TMDS_DATA_P

Figure 16 - Choose this HDMI interface.

We also must associate the inputs/outputs in our new IP to Vivado's interface definition for HDMI. Go into the Port Mapping tab and associate the following signals.



Add Interface

Use the tabs and fields below to modify the Bus Interface on your IP.

General | Port Mapping | Parameters

Interface's Logical Ports

- TMDS_DATA_P
- TMDS_DATA_N
- TMDS_CLK_P
- TMDS_CLK_N
- CEC_I
- CEC_O
- CEC_T
- DDC_SCL_I

IP's Physical Ports

- hdmi_clk_n
- hdmi_clk_p
- hdmi_tx_n
- hdmi_tx_p
- s_axi_aclk
- s_axi_aresetn
- s_axi_awaddr
- s_axi_awprot

Map Ports

List Options

□ Filter Incompatible Physical Ports □ Hide Mapped Ports

Mapped Ports Summary

Logical Port	Physical Port
TMDS_DATA_P	hdmi_tx_p
TMDS_DATA_N	hdmi_tx_n
TMDS_CLK_P	hdmi_clk_p
TMDS_CLK_N	hdmi_clk_n

Figure 17 - Bus Interface Port Mapping

Notice that TMDS_DATA_x is associated with hdmi_tx_x while TMDS_CLK_x is associated with hdmi_clk_x. Make sure to match up positive (p) with the correct signal and negative as well. This way, when we make a connection in the block design editor, we can connect the HDMI output as a single interface, instead of a bunch of individual signals. Note there are also some additional signals on the HDMI interface which we don't have, so we will leave unconnected. These are signals associated with CEC (Consumer Electronics Control) and DDC (Display Data Channel) – as we won't support these functions (feel free to research what these features provide). You can click on OK once you have correctly associated the ports. Run synthesis and ensure there are no errors in synthesis (there shouldn't be because you have only used provided code so far, but this is good practice for when you will modify the code to implement this assignment. Return to the IP Packager tab and select the "Review and Package" option. You may have some warnings associated with the ports and interfaces panel (due to the unconnected HDMI signals), but otherwise you should have only green checkmarks, and click on re-package IP. The temporary project you made will be closed and your IP will be packaged and ready for use in a MicroBlaze block design.

Note: We used a pre-defined interface in IP Packager (in this case HDMI) and associated that interface with our physical ports (hdmi_tx_p[0], etc.) Wouldn't it be convenient if we could do this in SystemVerilog as well? That means that instead of connecting individual signals in port mappings, we could connect entire interfaces at the same time. It turns out that this functionality **is** part of SystemVerilog – feel free to do some research on "SystemVerilog interface" to learn more.

Modifying the MicroBlaze Block Design to Add the New IP

Open the block design from the previous MicroBlaze tutorial or make a new one using the "microcontroller" preset. Since we are using the full versions of `printf` for this lab as a debugging aid, you should configure the on-chip memory for 128KB. Note that this will reduce the amount of on-chip memory available for your hardware designs. Therefore, if you do decide to use a similar MicroBlaze setup for your final project, you should reduce the on-chip memory and either do not use `printf` or use the minimal memory version `xil_printf` (which does not support certain formatting strings and only supports a single field). Configure the block design as follows by adding your new IP and connecting it to the AXI bus. Notice that the other modules are configured as with the previous MicroBlaze tutorial, although the GPIO for the blinking LED has been removed as it is no longer needed. Note also that any specific components for the USB controller (SPI and related GPIOs) may be removed as well – though you should maintain the UARTlite for debugging purposes.

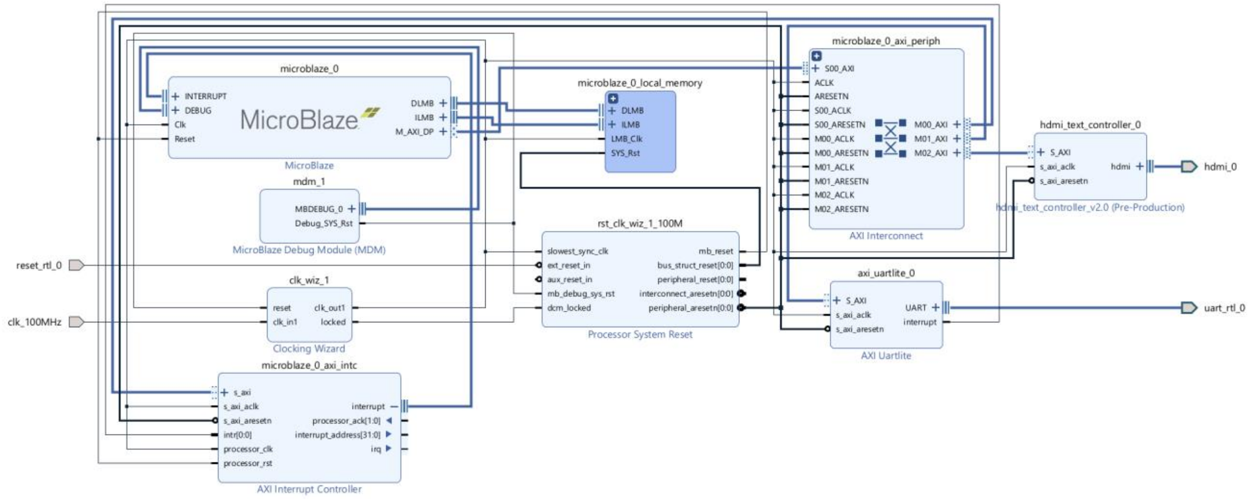


Figure 18 - Overall Connected System

You may use the automatic connection feature in the block design editor to connect the new peripheral up to the AXI bus. One connection you will have to manually make is to create an HDMI output (make external) the HDMI connection.

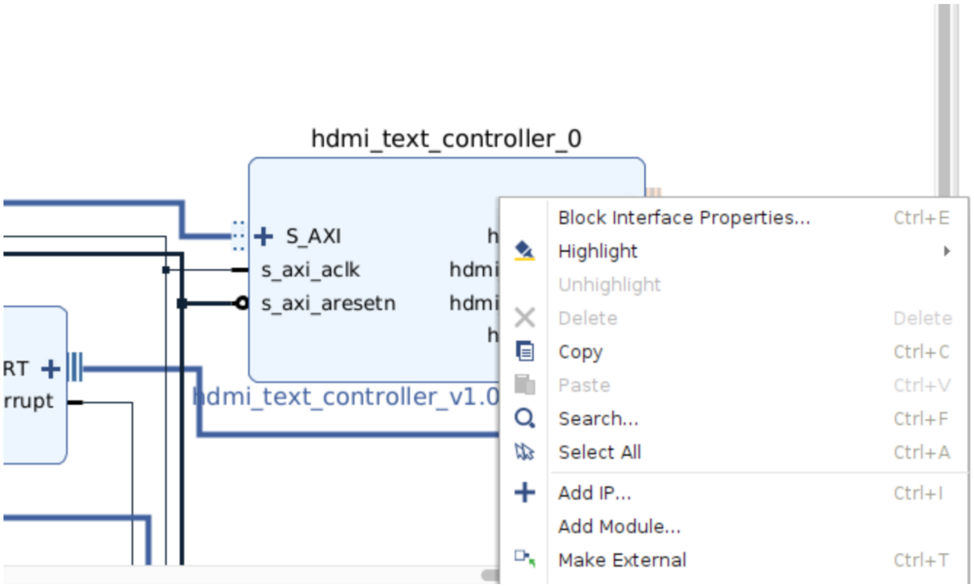


Figure 19 - Make External

Right click on the HDMI interface in your new IP and select Make External. This will create a new connection called "HDMI_0", which should go directly to the FPGA pins for HDMI.

Recall that one of the main advantages of using hardware IPs is reusability. Using a standardized interface like AXI4-Lite allows your IP to be reused across different projects, even if other components change. You should then validate and close the block design and import the provided top-level. Add the instantiation of your MicroBlaze block design into the provided top level by writing the module instantiation. Note that as a minimum, your top level should include and map the following inputs and outputs:

```
input logic Clk,
input logic reset_rtl_0,

//UART
input logic uart_rtl_0_rxd,
output logic uart_rtl_0_txd,

//HDMI
output logic hdmi_tmds_clk_n,
output logic hdmi_tmds_clk_p,
output logic [2:0]hdmi_tmds_data_n,
output logic [2:0]hdmi_tmds_data_p);
```

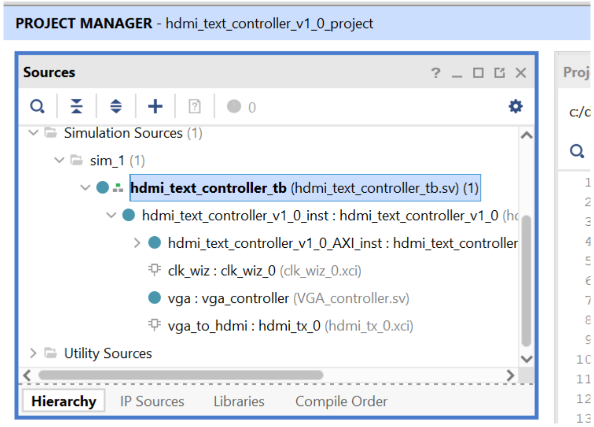
Make sure that you generate an inverted reset signal as well, as the reset expected in a Block Design is active low. As you may have already noticed, pushbuttons are active high on the Urbana board, but MicroBlaze (and any AXI4 IPs) expect resets to be active low. You may include additional debug signals (e.g. HEX drivers and LEDs) as necessary, though none are required for the demo.

You should now set up the software (Vitus) portion of the project by following the instructions from the MicroBlaze tutorial.

IMPORTANT: Note that to make changes to the hardware portion of the IP block, you must right click on the IP block and click Edit in IP Packager. You will then make the changes using this temporary project (the edit project). Once you are done, make sure you merge all the changes into the IP packager you created, and then repackage the IP, which will close and delete the temporary project. You must then upgrade the Block Design to the now updated version of your IP block. You will likely have to repeat this process many times to complete this lab (some of my solution codes easily have 30 or 40 ‘revisions’, before everything is working correctly).

Simulating the AXI4-Lite Bus and HDMI Output

You will find that due to the large number of steps and long compile times involved in modifying, rebuilding, re-synthesizing, and testing your IP block in hardware, it is advisable that you test the IP in simulation as much as possible to work efficiently. It turns out that with some slight modifications, it is straightforward to simulate the IP block in isolation within the **edit project**. Open the edit project by right clicking on the IP block and clicking “Edit in IP Packager”.



PROJECT MANAGER - hdmi_text_controller_v1_0_project

Sources

Simulation Sources (1)
sim_1 (1)
hdmi_text_controller_tb (hdmi_text_controller_tb.sv) (1)
hdmi_text_controller_v1_0_inst : hdmi_text_controller_v1_0 (hc
hdmi_text_controller_v1_0_AXI_inst : hdmi_text_controller
clk_wiz : clk_wiz_0 (clk_wiz_0.xci)
vga : vga_controller (VGA_controller.sv)
vga_to_hdmi : hdmi_tx_0 (hdmi_tx_0.xci)
Utility Sources

Figure 20 - hdmi_text_controller_tb set as testbench.

Add the following testbench to the edit project: *hdmi_text_controller_tb.sv* and make sure to "Set as top" so that it will function as the current testbench (see Figure 20). Note that if you haven't attempted much of the lab yet, the various sub-components such as the vga_controller or hdmi_tx_0 won't show up in the hierarchy yet – as you fill out more of the hdmi_text_controller_v1_0 file with the correct modules, they will appear in the hierarchy. The provided testbench serves two functions. The first is that as provided, it will test AXI write transactions. You will need to fill in the functionality for testing AXI read transactions. Once both these transactions have been thoroughly tested in simulation, the AXI portion of your IP (including reading back the checksum) should function correctly on hardware. The provided testbench also allows the simulation of the video screen, once you fill in the hierarchical assignments starting around line 107 and uncomment the `SIM_VIDEO` definition.

For now, let's work on simulating the basic AXI write transaction. If you try to simulate right now, you will get an error, something about invalid entry in project file. This is because unfortunately, due to strange Vivado design decision, IP edit projects are not set up for simulation by default. Every time you run XSim, Vivado must associate the files involved in simulation to a library, which is a database of objects associated with a simulation run. Vivado automatically does this for normal projects, but not for IP edit projects. An easy way to fix this is to enter the following command into the Tcl Console at the bottom of Vivado, which adds all the files to the xil_defaultlib library.

```
set_property library xil_defaultlib [get_files]
```

In addition, as was recommended in the Introduction to Vivado document, you should set the simulation time to 'all' instead of 1000ns. This allows you to monitor the entire simulation to the $finish () task without micromanaging the simulation time. Note that you must do this every time you open the edit project.

You should now be able to simulate an AXI write transaction. The provided code attempts to write 0000 to address 0000, 0001 to address 0004, etc. (keep in mind AXI addresses are byte addresses). It will also attempt to read back those addresses and alert you when the readback does not match. Probably this will fail as you have neither filled in the readback task, nor filled in the remainer of the register space (the provided code only populates 4 of the required 604 slv_regs).

In the process of testing, you should fill in the `axi_read()` task. In SystemVerilog, a task is a typically non-synthesizable unit which behaves similarly to a function in C/C++. Follow the example `axi_write()` task as well as the AXI4-Lite read waveform in Figure 6 (as well as the associated text steps. One this is completed, and the number of registers has been extended to 604, you should be able to simulate writing and reading back on your new IP.

Once you have completed some of the Next Steps below, you should revisit the testbench and attempt to simulate the entire 640x480 image, as it will be drawn via HDMI. To do this, you need to uncomment the `SIM_VIDEO` definition and fill in the hierarchical references starting in line 107 of the provided testbench. These are necessary because every student will have different internal signals connecting the various sub-modules within the *hdmi_text_controller_v1_0_.sv*. These signals tap into the video signals (h/v syncs, pixel clock, RGB values, etc.) and generate an image (as a BMP). Note that his may take several minutes, as it is simulating ~16 milliseconds worth of logic (millions of values) to generate the image.

When you have completed the HDMI_text_controller Week 1 design, the simulation should be able to draw the test image as seen in Figure 21. This is one of the required materials necessary for the lab report (the other one requires you to modify the testbench so that it simulates the Week 2 design with your NetIDs and specific colors – see the Lab manual document).

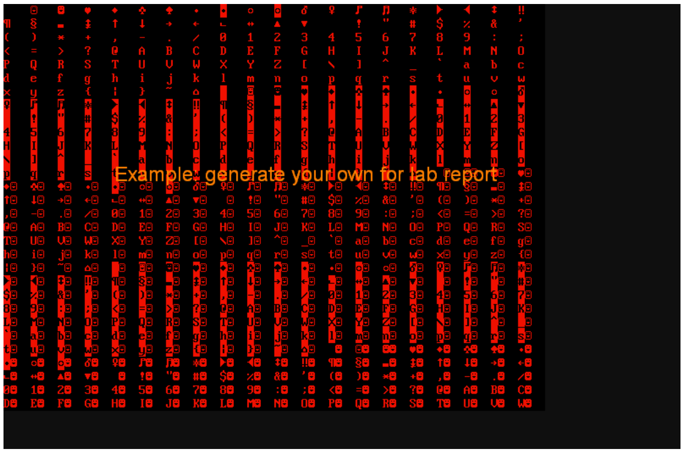


Figure 21 - Simulated Output of Week 1 Design

Next Steps

It will be up to you to populate the rest of the *hdmi_text_controller_v1_0_.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), and furthermore properly convert the VGA signal to HDMI for output from your IP block. For this, you will 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. Configure the VGA/HDMI modules to draw a static image (e.g., a single-color screen or a gradient). This ensures that the basic portions of the IP are set up correctly, and the HDMI signal is properly being transmitted from inside the IP to your monitor). Note that because the HDMI outputs a differential signal, until you create enough of the design within your *hdmi_text_controller_v1_0_.sv* module so that the VGA->HDMI block doesn't get optimized out you will have issues generating the bitstream. This is because the if the VGA-HDMI IP gets optimized out (because for example, it has no connected inputs or clocks) then the pins which it is supposed to drive with differential logic are instead no longer driven – which triggers a Vivado Design Rule Check (DRC) preventing bitstream generation. You will need to look at

the previous lab and instantiate at least VGA.sv, the clocking wizard, and the Real Digital VGA-HDMI IP, as well as connect them up to draw a simple image within *hdmi_text_controller_v1_0_.sv.* Refer to the previous week's lab assignment for an example on how to hook up these core modules.

2. Design the logic which writes into the registers from the AXI bus, the provided example already writes into 4, 32-bit registers – so extend this to the full 604 required.
3. Design the logic which reads back from the registers using the AXI bus, you will need the same signals as before. Note that the provided design should already do this, but verify via modifications to the testbench.
4. The provided `hdmiTestWeek1()` routine will test register writing and readback on the AXI4 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. Note that no main file/function is included for this lab, although the driver which contains `hdmiTestWeek1()` will automatically be included as part of the Vitis platform when you've included the IP in your Block Design. Therefore, you can simply use the "Hello World" example from Vitis and add a call to `hdmiTestWeek1()`.
5. Once you have some confidence that the register data is correctly accessed through AXI, start working on the video drawing portion. This consists of modifying the color mapper so that the DrawX and DrawY coordinates index into the appropriate registers of VRAM, and looking up the pixel values in the font_rom.
6. Run the full `hdmiTestWeek1()` 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 AXI through the MicroBlaze.

Week 2 Modifications – Color Graphics Adaptor

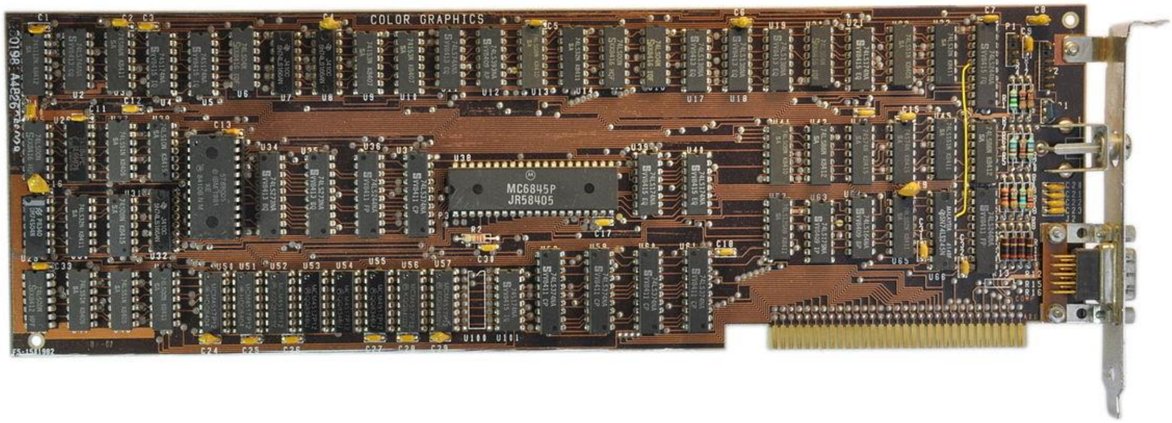


Figure 22 - IBM Color Graphics Adaptor

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 AXI-4 Lite 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 (BRAM).

Note: a key difference between registers and 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 (BRAM) blocks on the FPGA have two ports, each of which may be read or write. Therefore, only two memory locations may be accessed simultaneously.

In addition, recall that BRAM by default is pipelined, with a read pipeline depth of 2. We accommodated this latency using wait-states in the previous SLC-3 lab, where we also used BRAM (recall the reasoning behind S33_0, S33_1, etc.) Because AXI4 is a handshaking bus, it does not require fixed wait states (this is especially useful for peripherals such as dynamic memory controllers, which may have variable latency depending on the address being accessed). However, you may need to modify the AXI4 handshaking which was provided as part of the IP template *hdmi_text_controller_v1_0_.sv and hdmi_text_controller_v1_0_S_AXI.sv* files to account for this latency to support proper read back of both VRAM and palette registers (which is required for full credit).

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 *hdmi_text_controller_v1_0_.sv* 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 / synchronization 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 Vivado
0x800 - 0x807	**0x0000 2000 - 0x0000 201F**	Palette- 8 words of 2 colors each, for 16-color palette
0x808 - 0x80A	**0x0000 2020 - 0x0000 202B**	Synchronization read-only registers
0x80B - 0xFFF	**0x0000 202C - 0x0000 3FFF**	Unused but reserved by Vivado

The memory layout of the color palette is as follows:

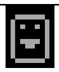
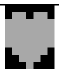
Table 9. Color Palette Organization / Control Registers (0x800-0x80A)

Address	**31-28**	**27-24**	**23-20**	**19-16**	**15-12**	**11-8**	**7-4**	**3-0**
0x800	UNUSED	C1_R	C1_G	C1_B	UNUSED	C0_R	C0_G	C0_B
0x801	UNUSED	C3_R	C3_G	C3_B	UNUSED	C2_R	C2_G	C2_B
…								
0x807	UNUSED	C15_R	C15_G	C15_B	UNUSED	C14_R	C14_G	C14_B
0x808	FRAME_COUNTER							
0x809	CURRENT_DRAW_X							
0x80A	CURRENT_DRAW_Y							

Note that the synchronization registers (0x808-80A) function the same as they do in the previous week’s design apart from their base addresses.

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

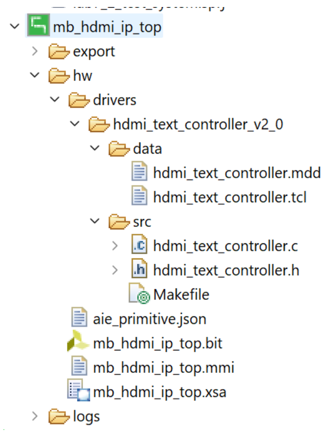
	0	1	2	3	4	5	6	7
0	 Green on Red	 Red on Green	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-

Next Steps (Week 2)

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

1. Back-up your Week 1 code, making sure to backup both the Vivado/Vitis projects as well as the IP.
2. **Starting with your working Week 1 design**, modify the IP and increment the version to version 2.0. Then modify the *hdmi_text_controller_v1_0_.sv* to support on-chip memory VRAM, and maybe rename it to *hdmi_text_controller_v2_0_.sv* to properly reflect the second portion. 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.**
3. 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.
4. Download the Week 2 .zip archive, delete the hdmi_text_controller.c/h files, and replace with the new provided hdmi_text_controller.c/h files.

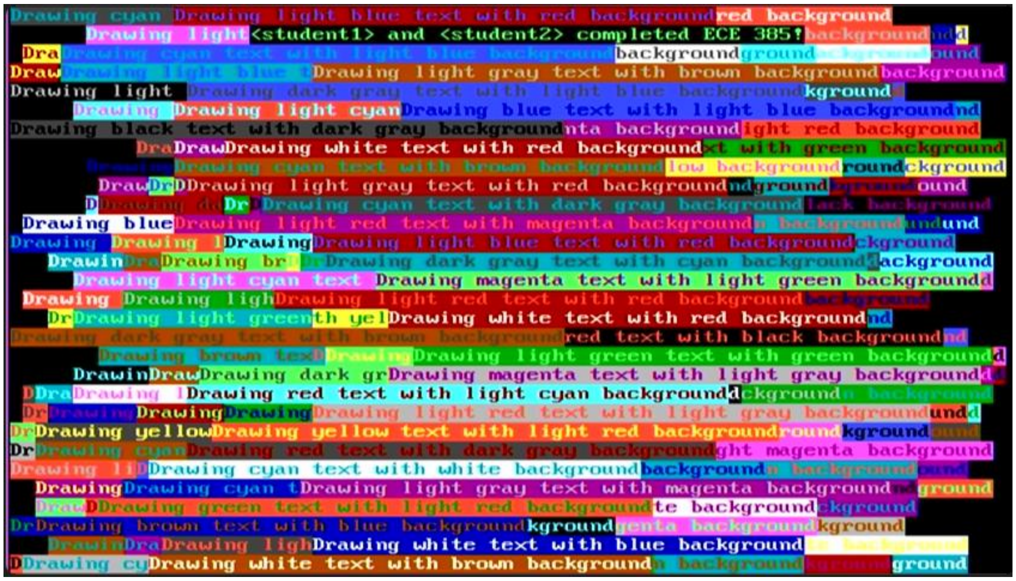
5. Add the appropriate functionality in the `void setColorPalette(…)` function; this is used to initialize the color palette (the demo code uses the CGA standard colors). However, do not hardcode the palette, as the demo code also relies on the ability to dynamically rewrite the palette. Note that it is quite awkward to repackage the IP every time you want to change the software driver. Instead, modify the 'implemented' version of the driver, which is in Vitis under the platform (see figure to the right) and make sure to do a clean build to force an update. However, once you're satisfied with the implementation of `setColorPalette(…)`, you should copy the function back into the IP file to package your IP nicely.



```
mb_hdmi_ip_top
  export
  hw
    drivers
      hdmi_text_controller_v2_0
        data
          hdmi_text_controller.mdd
          hdmi_text_controller.tcl
        src
          hdmi_text_controller.c
          hdmi_text_controller.h
          Makefile
    aie_primitive.json
    mb_hdmi_ip_top.bit
    mb_hdmi_ip_top.mmi
    mb_hdmi_ip_top.xsa
  logs
```

Figure 23 - Location of drivers

6. Call the `hdmiTestWeek2()` function from main, which will run through a couple of checksum and palette test routines and then draw a nice screensaver. Modify the `hdmi_text_controller.h` file so that it properly displays you and your partner's NetID. Note that proper synchronization and drawing of the DVD screensaver's text relies on properly working synchronization registers. Note that the "<student1> and <student2> completed ECE 385!" text should be animated at 6 times a second (every 10 frames) while the background updates at 2 times a second (every 30 frames). The moving text should **not** leave a trail of blank/garbage characters.



<student1> and <student2> completed ECE 385!
Drawing light gray text with brown background
Drawing white text with red background
Drawing white text with blue background
Drawing white text with brown background

Figure 24 - Output Screensaver from Week 2 Design