ECE 4156/6156 Hardware-Oriented Security and Trust

Spring 2025

Assoc. Prof. Vincent John Mooney III

Georgia Institute of Technology

Lab 2, 100 pts.

Due Friday, February 14 prior to 11:55pm

(Please turn in the zipped lab files electronically on Canvas)

Lab 2

In this lab, you will simulate and verify a VHDL implementation of the Advanced Encryption Standard (AES). You will be given VHDL code which implements both encryption and decryption, which you will then simulate. A sample testbench is also provided to help you set up the simulation. After simulating the sample test bench, you will be asked to modify it to check test cases you will generate. To be able to make an accurate comparison, you are also given a correctly functioning AES C code.

After testing the design in ModelSim, you will synthesize and implement VHDL code to implement encryption and decryption for the DE-10 Standard board using Quartus. You will also interface the HPS and FPGA (Field Programmable Gate Array) over the AXI bus. We will provide an SD card with an OS containing a pre-synthesized bitstream implementing AES encryption. You will have to read and write inputs to interface with FPGA logic and read the output via File I/O on a terminal and verify with your C code output.

The motivation for this lab is to execute AES code both in software and in hardware.

Please type your answers to the questions in this lab into a lab report.

This lab will not reiterate the steps explained in Lab 1. If you need a reminder of the functionality of Quartus or ModelSim, please refer to Lab 1.

I. VHDL/ DE-10 Help

There is plenty of documentation available on how to write good VHDL. Some good simple examples can be found here. Some good YouTube videos introducing VHDL basics can be found here.

Documentation and resources for the DE-10 Standard board can be found at the following:

https://rocketboards.org/foswiki/Documentation/DE10Standard

https://www.utilities-online.info/ascii-to-hex, this is a resource to convert ascii to hex.

II. Generation of Baseline Test Cases Using C Code

Inside the "lab2" directory, you will find a folder named "c_code". This folder contains a functioning version of AES in the C programming language. Compile and use the provided code to generate correct output results.

As for testing the code, you must submit 5 plaintext test cases, 5 ciphertext test cases, and five keys (thus, you will test all 10 input cases on each key for 50 tests (half testing encryption and half testing decryption). These test cases must be your own; under the GT (Georgia Tech) Honor Code, you are _not_ allowed to share your test cases with other teams! The reason for this is that the focus should be on good coding and testing practices, not comparison with another student's results.

You must provide a brief rationale for your chosen five plaintext cases. The five cases must be unrelated to each other; e.g., you may not choose numeric sequences (e.g., adding one to each prior case).

The test cases and keys must be submitted in .txt files with following names:

- Key.txt (5 keys)
- Plaintextin.txt (5 plaintext test cases)
- Ciphertextout1.txt (5 ciphertext from using key1 and the plaintextin.txt)
- Ciphertextout2.txt (5 ciphertext from using key2 and the plaintextin.txt)
- Ciphertextout3.txt (5 ciphertext from using key3 and the plaintextin.txt)
- Ciphertextout4.txt (5 ciphertext from using key4 and the plaintextin.txt)
- Ciphertextout5.txt (5 ciphertext from using key5 and the plaintextin.txt)
- Ciphertextin.txt (5 ciphertext test cases, please pick one ciphertext output from each of the ciphertextoutX.txt (ciphertextout1.txt, ciphertextout2.txt, ... , ciphertextout5.txt)
- Plaintextout1.txt (5 plaintext from using key1 and the ciphertextin.txt)
- Plaintextout2.txt (5 plaintext from using key2 and the ciphertextin.txt)
- Plaintextout3.txt (5 plaintext from using key3 and the ciphertextin.txt)
- Plaintextout4.txt (5 plaintext from using key4 and the ciphertextin.txt)
- Plaintextout5.txt (5 plaintext from using key5 and the ciphertextin.txt)

The result of decrypting a ciphertext produced by a particular encryption of an input plaintext should result in the same plaintext output (assuming the same key is used). Similarly, the result of encrypting a plaintext produced by a particular decryption of an input ciphertext should result in the same ciphertext output (assuming the same key is used).

As you are selecting one ciphertext from each of your files, and decrypting all five chosen ciphertexts with one key, you should expect to only see only one correct original plaintext in each of your plaintextoutX files. You are allowed to use the ciphertextoutX files to choose candidates for your ciphertextin file.

Please note that your code must be able to take in the 5 plaintext cases and produce 5 ciphertext cases per key for a total of 25 ciphertext outputs. Similarly, your code must be able to take in the 5 ciphertext cases per key for a total of 25 plaintext outputs. Finally, to grade your assignment we will use additional inputs and keys not specified by you; therefore, try to avoid buggy code, as this will affect your final grade.

This section's task is to modify the "main" function in the code to adapt it to your input/output file's structure. Please also submit a README to show me how to run your C code. Your C code is not required to write ciphertextout.txt and plaintextout.txt directly to a file. If you choose not to write the results to a file, your program must print the results in the terminal. Please make sure your print statements to the terminal can be understood by the TA.

III. ModelSim Simulation of VHDL Code

After generating the baseline results for AES using the C code, you will use ModelSim to simulate the VHDL implementation and compare your new simulation results to those of the C code.

When programming in any language, it is useful to debug, test, or simulate your code to verify its functionality. When programming in VHDL, the convention is to have functional VHDL code and a testbench which tests the code. In this section, you will simulate the provided testbench (tb_AES_decrypt.vhd for decryption and enc_tb.vhd for encryption) with the given VHDL code in ModelSim.

During this portion, note that the byte endianness of the **Encryption** VHDL code is opposite that of the previous C code. An example is as follows, with partial color coding for understanding:

--p = 0123456789abcdeffedcba9876543210 C code Plaintext

plaintext <= x"1032547698badcfeefcdab8967452301"; VHDL code plaintext

--c = ac6c9fd5b14bb5ec1ef70964ac34a9ce Ciphertext 1

ciphertext <= x"cea934ac6409f71eecb54bb1d59f6cac"; VHDL code ciphertext

--k = 1972bce09da0f71290f710bc83109edf C code key

key <= x"df9e1083bc10f79012f7a09de0bc7219"; VHDL code key

To simulate the code, proceed as follows:

- Let us start with Encryption
  - Open ModelSim
  - Create a new project by clicking on the following: File --> New --> Project...
  - Verify that the project location is set to lab2/vhdl/AES_ENC/ and give the project a name, say "ENC", and click OK.
  - After creating the project, we must add all the VHDL files to it. Do so by clicking on the following: Add Existing File --> Browse...
  - Choose all the VHDL files in the directory. Click Open --> OK.
  - Once you are done adding **all** the VHDL files, click Close.
  - Now we must compile the design files. To do so click on the following: Compile --> Compile Order, then click on Auto Generate and press OK. Check the transcript window to make sure all files were successfully compiled. You may need to compile twice in a row.
  - Now we can start the simulation by clicking on: Simulate --> Start Simulation... In the Start Simulation window, make sure you are on the design tab, expand the work library, and choose the testbench for this code named "enc_tb", and click OK.
  - ModelSim will change the view into simulation mode and a couple of other windows will show up.
  - Now let us add our signals of interest to the wave window to monitor their changes as simulation proceeds. To do so, go to the "sim" window and click on the instance named "test_enc" to add the testbench signals. Next, open the "Objects" window and choose all the signals (inputs, outputs, and internal). Right-click on the selection and click on Add Wave. Do the same process to add all the signals of the instance "aes_enc." aes_enc should be your top-level module, while test_enc is your testbench. Check that the signals have been successfully added to the "Wave" window.
  - Our last step is to run the simulation for a specific time. For this testbench, run the simulation until all testcases have been resolved. To do so, type **run 500 ns** in the command line of the "Transcript" window. Read the transcript to check that all your testcases have been resolved (either failed or successful), otherwise, run another 500 ns.

- Navigating back to the "Wave" window will now show you the result of the simulation for all the signals that we added. To better read the values, select all the signals and right-click, then change the Radix to Hexadecimal. Also, towards the bottom left of the signals pane (to the left of where it says "Now"), there is a blue button that has a description of "Toggle leaf names <-> full names" if you hover the mouse over it. Click on that button to show the signal names only without the hierarchy.
- Look through the wave window and try to understand how the signals are changing values with respect to the simulation time. Specifically, look for the output signal that shows the encrypted/decrypted value.
- Now that we have run the simulation, make sure that you set the zoom of the wave window to a full view. To do so, right-click anywhere in the wave window and click on Zoom Full. Export an image of your simulated waveform. To do so, click on File --> Export --> Image... and save it as an image, or take a screenshot with any other method. **Include this image in your submission**.
- Before ending the simulation, open the transcript window and verify that no reports are generated by the testbench indicating a failure of any of the test cases.
- Finally, to end the simulation, click on: Simulation --> End Simulation.

- Now, let us work on Decryption. **NOTE: Decryption uses the same endianess as the C code.**
  - Create a new project by clicking on the following: File --> New --> Project...
  - Verify that the project location is set to lab2/vhdl/AES_DE/ and give the project a name, say "DEC", and click OK.
  - After creating the project, we must add all the VHDL files to the project. Do so by clicking on the following: Add Existing File --> Browse...
  - Choose all the VHDL files in the directory. Click Open --> OK.
  - Once you are done adding **all** the VHDL files, click Close.
  - Now we must compile the design files. To do so click on the following: Compile --> Compile Order, then click on Auto Generate and press OK. Check the transcript window to make sure all files were successfully compiled.
  - Now we can start the simulation by clicking on the following: Simulate --> Start Simulation... In the Start Simulation window, make sure you are on the design tab, expand the work library, choose the testbench for this code named "tb_AES_decrypt", and click OK.
  - ModelSim will change the view into simulation mode and a couple of other windows will show up.
  - Now let us add our signals of interest to the wave window to monitor their changes as simulation proceeds. To do so, go to the "sim" window and click on the instance named "tb_AES_decrypt" to add the testbench signals. Next, open the "Objects" window and choose all the signals (inputs, outputs, and internal). Right-click on the selection and click on Add Wave. Follow the same process to add all the signals of the instance "aes_decrypter." aes_decrypter should be

your top-level module, while test_enc is your testbench. Check that the signals have been successfully added to the "Wave" window.

- Our last step is to run the simulation for a specific time. For this testbench, run the simulation until all testcases have been resolved. To do so, type **run 500 ns** in the command line of the "Transcript" window. Read the transcript to check that all your testcases have been resolved (either failed or successful); otherwise, run for another 500 ns.
- Navigating back to the "Wave" window will now show you the result of the simulation for all the signals that have been added. To better read the values, select all the signals and right-click, then change the Radix to Hexadecimal. Also, towards the bottom left of the signals pane (to the left of where it says "Now"), there is a blue button that has a description of "Toggle leaf names <-> full names" if you hover the mouse over it. Click on that button to show the signal names only without the hierarchy.
- Look through the wave window and try to understand how the signals are changing values with respect to the simulation time. Specifically, look for the output signal that shows the encrypted/decrypted value.
- Now that we have run the simulation, make sure that you set the zoom of the wave window to a full view. To do so, right-click anywhere in the wave window and click on Zoom Full. Export an image of your simulated waveform. To do so, click on File --> Export --> Image... and save it as an image, or take a screenshot with any other method. **Include this image in your submission**.
- Before ending the simulation, open the transcript window and verify that no reports are generated by the testbench indicating a failure of any of the test cases.
- Finally, to end the simulation, click on: Simulation --> End Simulation.

Now that you have simulated the design, modify the test bench to add all your test cases. Notice how the testbench uses assert statements to check for the expected output and verify it. Feel free to automate the process by implementing file I/O in VHDL. Once you are done modifying the testbench, save it, recompile it and resimulate the design. You may also hardcode all your testcases if you do not want to implement File I/O in VHDL.

An example of File I/O if you wish to use it can be found here:

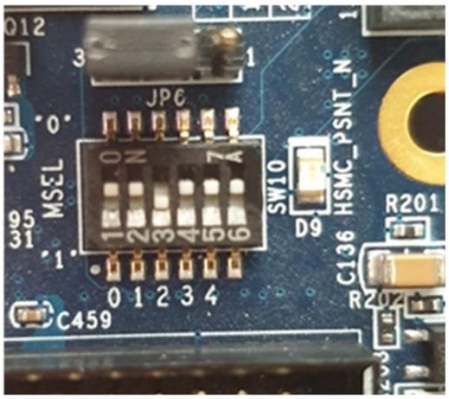
VHDL Example Code of File IO (nandland.com)

For the lab report provide **for AES ENCRYPTION ONLY** an explanation of the function of each provided VHDL file (what they accomplish), what part of the AES algorithm the file accomplishes (if applicable), and a description of the overall hierarchical connections of the files (i.e., how are they connected together).

IV. Running AES on the DE-10 board

You will now have to run AES on the DE-10 board. For this portion of the lab, you will have to use the SD card given. To set up the board to work with the SD card, you need to do the following steps.

a. Insert the SD card into the board
b. Set the MSEL Bit of the board to the following setting:



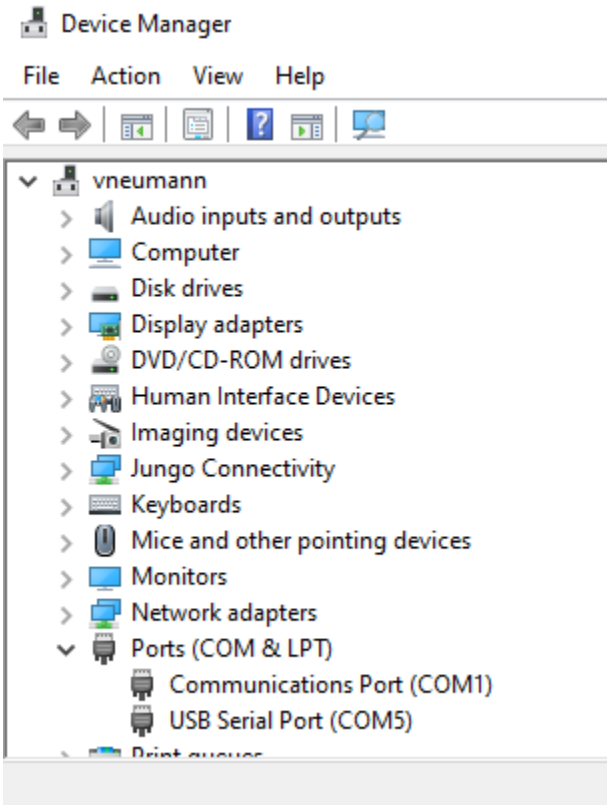
c. Connect the board to the Windows PC using UART to USB Cable
d. Connect the board to the power using the power cable
e. Press the power button to power on the board

The UART to USB cable is the following:

If you are missing this cable, please email the TA or meet the TA during office hours.

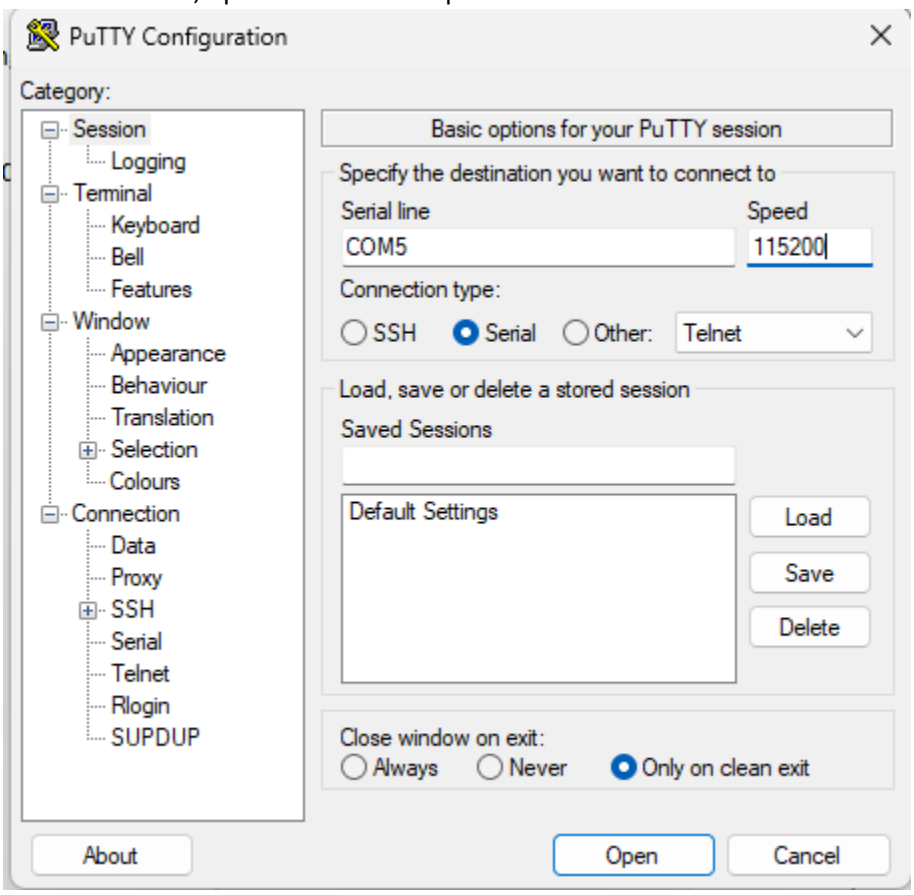
Steps to Connect the board in Windows:

1. Open device manager and look for the proper port:



For example, above the port number is 5.

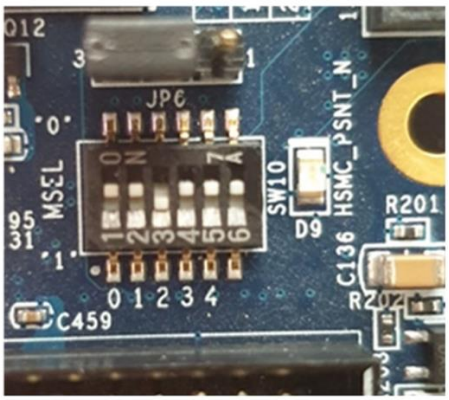
2. Connect the board to the Windows PC using PuTTY
    a. On Windows PC, open PuTTY to set up the board's Serial Connection



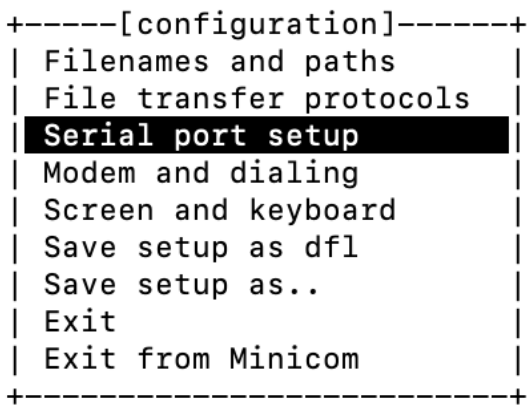
    b. Fill the right COM port and set the speed as 115200
3. After connecting to PuTTY successfully, power off and then power on the board again; rsyocto boots automatically.
4. Log in as a root user
    a. After rsyocto finishes booting, terminal will prompt you to login
        i. Login: root
        ii. Password: eit

Steps to Connect the board in Linux (if you are connecting the DE-10 to a Linux computer):

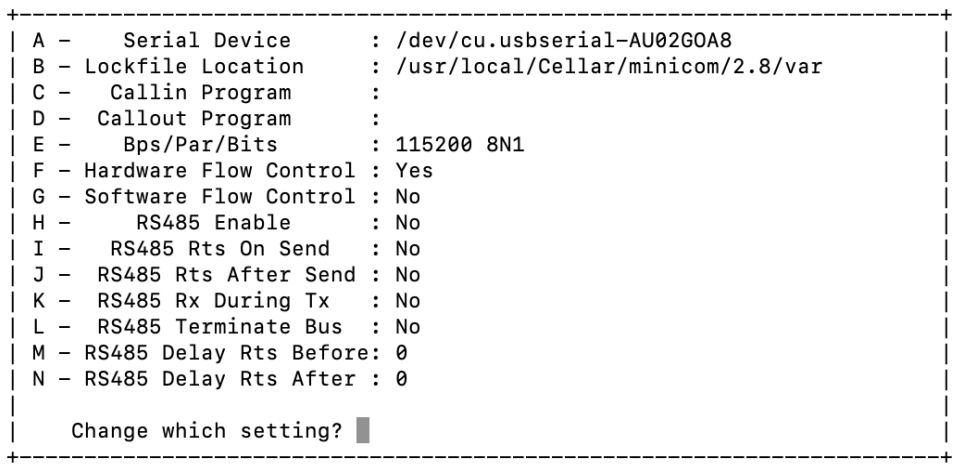
1. Prepare the board
    a. Insert the SD card into the board
    b. Set the MSEL Bits of the board to the following setting (“001000”):



    c. Connect the board to the Linux PC using UART to USB Cable
    d. Connect the board to the power using the power cable
    e. Press the power button to power on the board
2. Connect the board to the Linux PC using Minicom
    a. On Linux PC, open Minicom to set up the board’s Serial Connection
        i. Before opening Minicom, we need to find the board name
            1. To do this, open a terminal on Linux PC
            2. Execute the following command
                a. ls /dev/cu.*
                b. After this, find the board
                c. It might be shown as something like this:
                d. /dev/cu.usbserial-AU02GOA8
                e. Make sure to copy this (what is shown above in part d) down, you will need this in the next step
        ii. After getting the board name, open the Minicom
            1. To do this, in the terminal on Linux PC
                a. Execute the following command
                    i. minicom -s
                    ii. After opening minicom, your terminal would be like this



2. Open Serial port setup in minicom
    i. After opening it, your terminal would be like this



3. Paste what you got in step 22(a)(i)(2)(d) to the “A - Serial Device” shown in the screen above

b. After connecting to Minicom successfully, power off and then power on the board again, rsyocto boots automatically

3. Log in as a root user
    a. After rsyocto finishes booting, terminal will prompt you to login
        i. Login: root
        ii. Password: eit

Running the AES Binary file on the SD card

We have developed a raw binary file (rbf) which already runs AES encryption. To run this, Load the .rbf file and read/write to the LW AXI bridge using the instructions here

a. In the terminal, the following three commands are used to interact with the AES accelerator:
    1. FPGA-writeConfig -f aes_axi.rbf
        1. This command loads the FPGA bitstream onto the FPGA fabric
    2. FPGA-writeBridge -lw 20 -h ab
        1. This command writes data 0xab to the light weight AXI bridge at address 20
    3. FPGA-readBridge -lw 80
        1. This command reads the light weight AXI bridge offsetting at address 80

The binary file given has the following locations with associated functionality.

Name	Address Location	I/O
Key segment 0	0	Input
Key segment 1	10	Input
Key segment 2	20	Input
Key segment 3	30	Input
Plaintext segment 0	40	Input
Plaintext segment 1	50	Input
Plaintext segment 2	60	Input
Plaintext segment 3	70	Input
Ciphertext segment 0	80	Output
Ciphertext segment 1	90	Output
Ciphertext segment 2	100	Output
Ciphertext segment 3	110	Output
Reset	120	Input

Each location holds a 32-bit value. This is a hex value. By default, it holds the value 0 at each of the input locations. A new ciphertext is only calculated after "pressing" the reset button, which we will do by writing a '1' followed by a '0' to the reset address.

An example of how to encrypt x"00007890 00005678 00003456 00001234" with key x"00007890 00005678 00003456 00001234" (if these values are using the C endianess) is as follows:

1. FPGA-writeConfig -f aes_axi.rbf
2. FPGA-writeBridge -lw 0 -h 90780000
3. FPGA-writeBridge -lw 10 -h 78560000
4. FPGA-writeBridge -lw 20 -h 56340000
5. FPGA-writeBridge -lw 30 -h 34120000
6. FPGA-writeBridge -lw 40 -h 90780000

7. FPGA-writeBridge -lw 50 -h 78560000
8. FPGA-writeBridge -lw 60 -h 56340000
9. FPGA-writeBridge -lw 70 -h 34120000
10. FPGA-writeBridge -lw 120 -h 1
11. FPGA-writeBridge -lw 120 -h 0

Note: In the above example, the first 32 bits (0x00007890) comprise segment 0, the second 32 bits (0x00005678) comprise segment 1, and so on.

You would then read each ciphertext segment with the following commands:

1. FPGA-readBridge -lw 80
2. FPGA-readBridge -lw 90
3. FPGA-readBridge -lw 100
4. FPGA-readBridge -lw 110

Note: The ciphertext segments are split similarly to the plaintext and key (I.e., address 80 holds the 32 MSBs)

Your task for this portion of the lab is to connect to the board using UART and run five of the tests you ran from the previous section. Specifically, for each key you chose, load one of your plaintexts and ensure that the resulting ciphertext matches.

Using the rbf file, perform the five encryptions and take of screenshot of the resulting ciphertext values (all four 32-bit segments). Also, include a table in your report of each key and plaintext and corresponding ciphertext.

Note: We have provided the AXI bus addressable AES accelerator for you. An appendix is provided at the end of the lab on how to create your own rbf file and run it on the DE-10 board if interested.

Lab Report

Please type all your answers and include a cover sheet with your first and last name, GT ID number, and GT username.

Code Modification and Simulation Section:

1. Submit all the code (C code and VHDL files) with in-line comments pointing out the modifications you made in the files.
2. Provide a brief explanation of the modifications that you made to the main function and to the VHDL testbench.
3. Simulation results compiled into .txt files as described in Section II must show all keys, plaintext and ciphertext values.
4. Waveforms in the lab report verifying one encryption testcase and one decryption testcase in Section II must be provided. Please make sure the values shown on the waveform are legible.
5. Provide an explanation of the function of each provided VHDL file **for AES ENCRYPTION ONLY** (what they accomplish), what part of the AES algorithm the file accomplishes (if applicable), and a description of the overall hierarchical connections of the files (I.e., how are they connected together).

Synthesis and Implementation:

1. Screenshots of the Ciphertext values
2. Table of test vectors

Canvas Submission of Lab 2 Files

1. Place all files into a single folder and submit your work on Canvas.

Appendix

How to interface the FPGA and HPS over the AXI Bus on the Intel DE-10 Standard Board (Linux Version)

Authors: Kevin Hutto and Yiming Tan

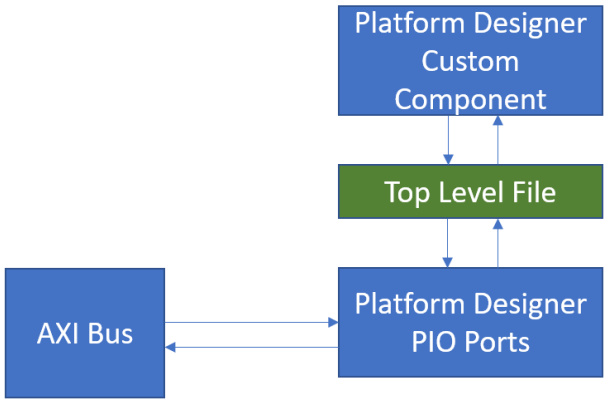
These instructions are certainly not the most effective way to interface the FPGA and the HPS over the AXI Bus in terms of FPGA efficiency or workflow efficiency, but it is the best way that works so far. These instructions are also not meant to be standalone or a complete tutorial on how to use Quartus or write VHDL.

Please note for the Windows version, you need a router.

First, a list of the software and hardware needed to interface the FPGA and HPS over the AXI Bus:

- Software:
  - Quartus
  - Etcher
  - Minicom
  - rsyocto
- Hardware:
  - Linux PC
  - Intel DE-10 Standard Board (with its power cable and UART to USB cable)
  - Empty SD Card
  - SD Card Reader
- Note: all software listed above should be installed on the Linux PC

Second, a brief overview of what we will try to accomplish:



We have a custom component that we want to interface over the AXI bus. We will instantiate this component in Platform Designer. However, we will not be connecting this component to the AXI bus yet. Instead, we will instantiate components known as Parallel Input/Output (PIO) ports. We will connect these to the AXI bus, and they will essentially act as registers. We will then generate the HDL (Hardware Description Language) files automatically from the Platform Designer tool, and from the auto generated top level file we will create a new top-level file which will connect the PIO port registers to the signals we want to connect to on our custom component. This is a roundabout way of creating a method to directly read and write from all interfaces on our desired custom component from the AXI bus on the HPS.

Much of this was learned from these other sources of information:

https://github.com/robseb/rsyocto

https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/Making_Qsys_Components_15_0.pdf#:~:text=VHDL%20code%20for%20the%20memory-mapped%20new-register%20interface.%204Avalon,to%20request%20and%20send%20data%20to%20slave%20components.

Third, more in depth steps to be followed strictly to interface the FPGA and HPS over the AXI Bus:

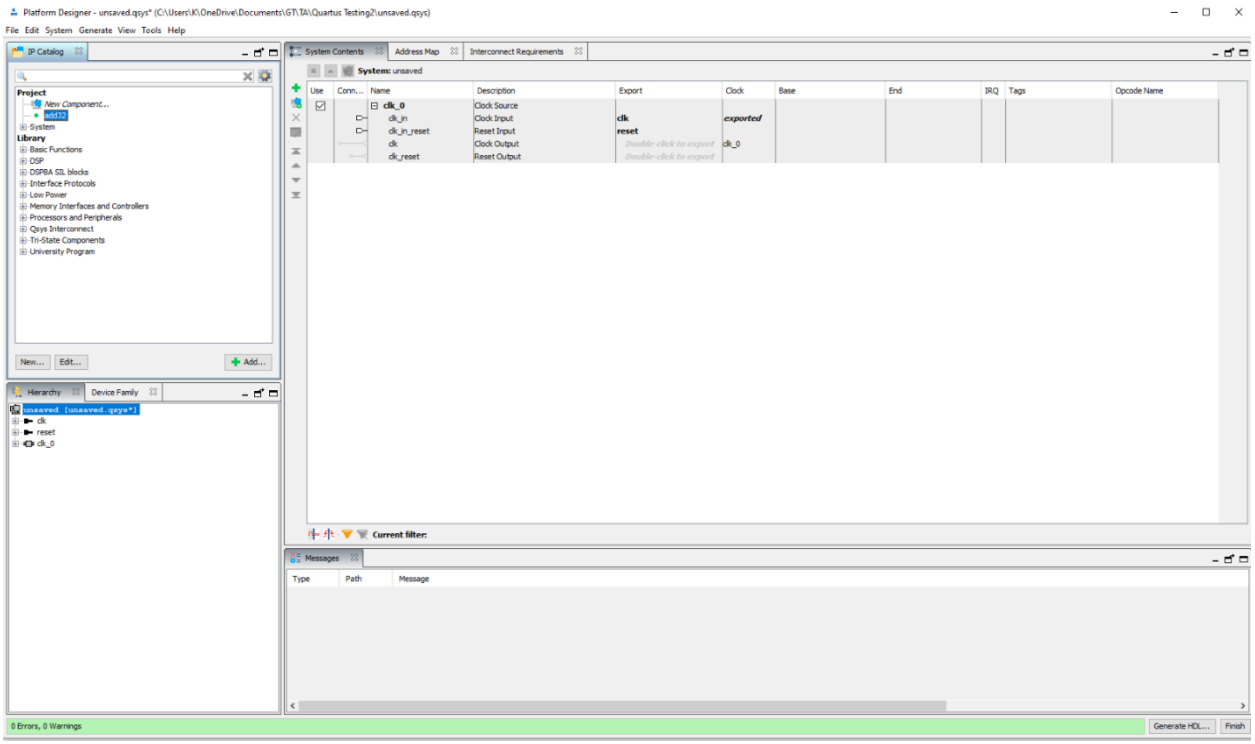
1. On the Linux PC, open Quartus
    a. Note: Step 1 to Step 18 should all be done in Quartus
2. In Quartus, create your desired HDL custom component. It must have the following:
    a. A clk signal
    b. A reset signal

This instruction guide will use the following code as the example:

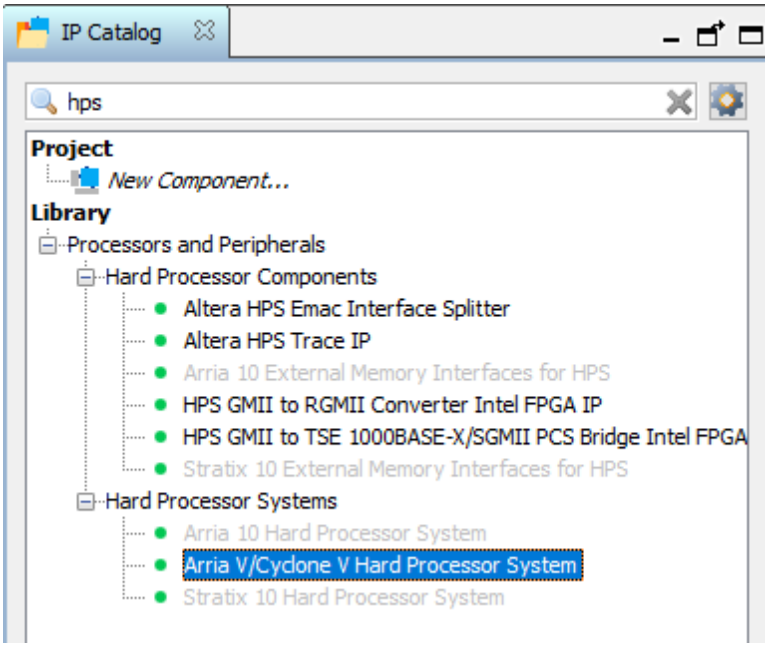
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY add32 IS
PORT (clk, resetn : in std_logic;
input1, input2 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
output1 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END add32;



ARCHITECTURE Behavior OF add32 IS

BEGIN


process(clk, resetn)
begin
if (resetn <='0') then
output1 <= (others => '0');
elsif (rising_edge(clk)) then
output1 <= input1 xor input2;
end if;
end process;

END Behavior;
```

All this circuit does is XOR two inputs. It outputs the result after a clk edge.

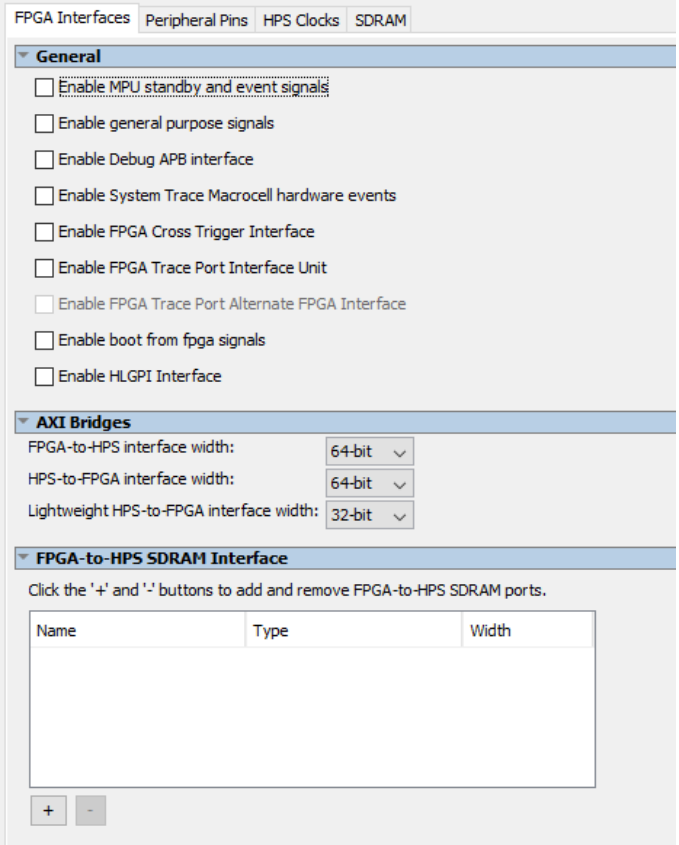
3. Create a new Quartus project with the desired settings. Leave the project blank for now.
4. Open Platform Designer
    a. Found inside Quartus. Tools -> Platform Designer



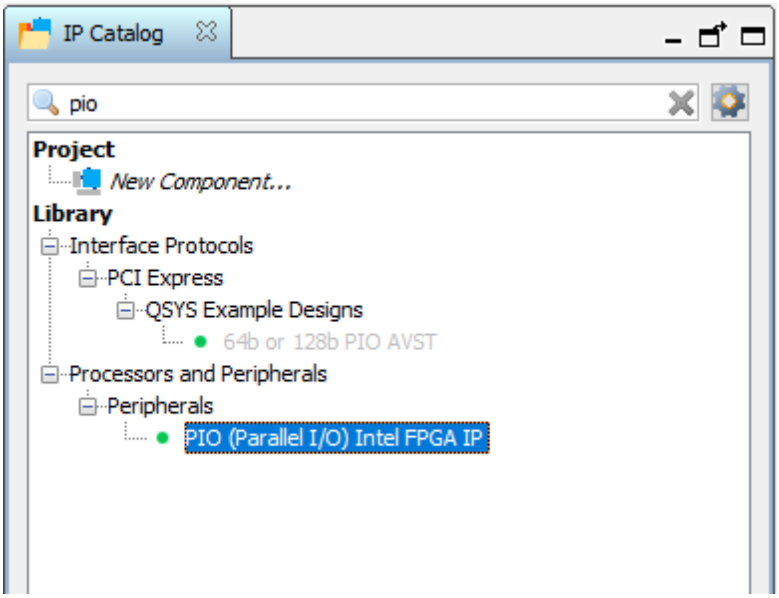
5. To your platform designer project add the following by searching in the IP Catalog box on the left:
    a. Arria V/Cyclone V HPS



    i. Remove the SDRAM, unclick “Enable MPU standby and event signals”
    ii. Leave all other parameters as default, click finish in bottom right corner

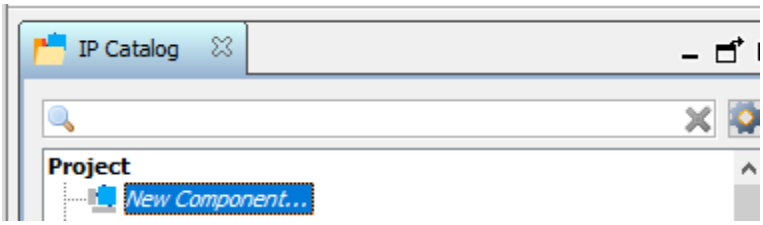


b. Add PIO components equal to the number of signals (as a vector, not individual lines) you wish to interface with. For the example XOR/Adder code we need three PIO ports (Two for input, one output):

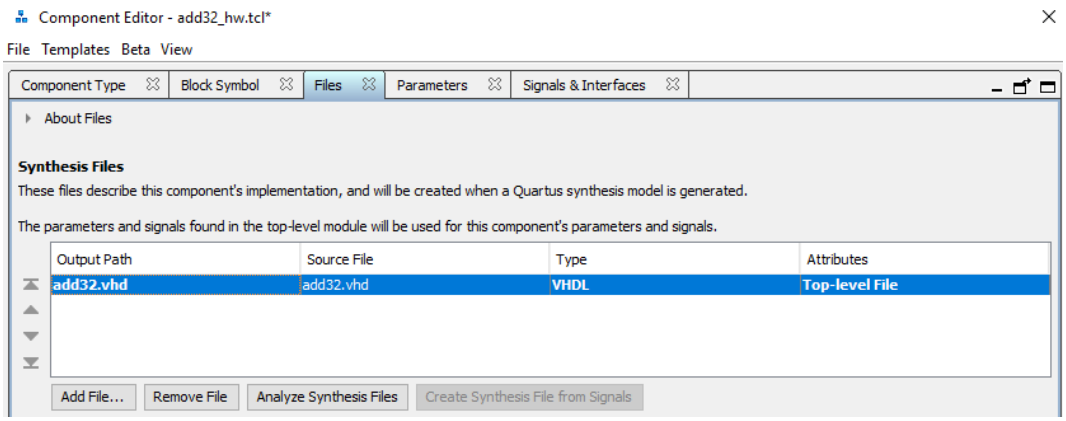


i. Set the size of the PIO width to the same size as you need but set the direction OPPOSITE of what your custom component is. In other words, if your custom component has an input of 16 bits, create a PIO component with a 16-bit output
    1. The example code would need two output PIO set to 32 bits, and one input set to 32 bits
ii. If a signal vector is > 32 bits, you need to use two PIO components to handle the signal. In other words, if you have a 48-bit input, you could have two 24-bit outputs, or a 32-bit output and a 16-bit output.

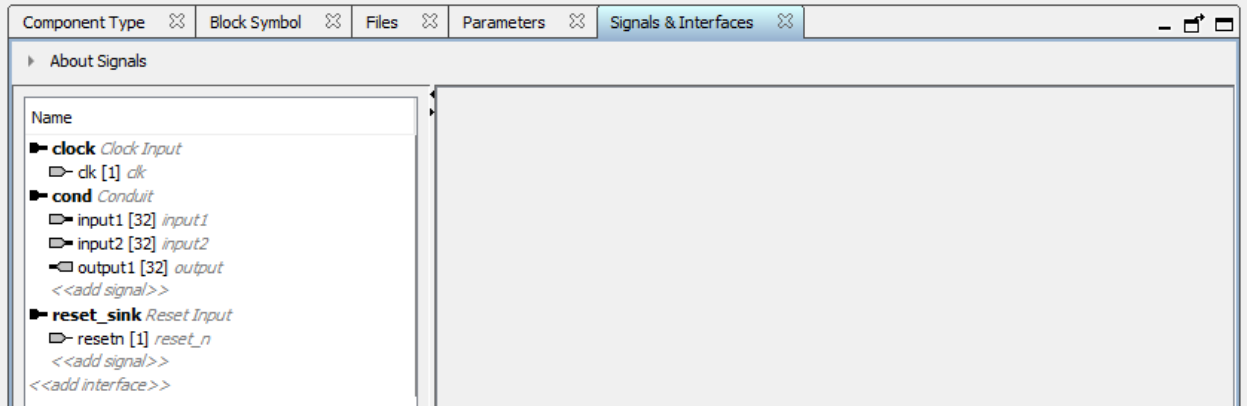
c. Create a custom component by clicking on the new component tab



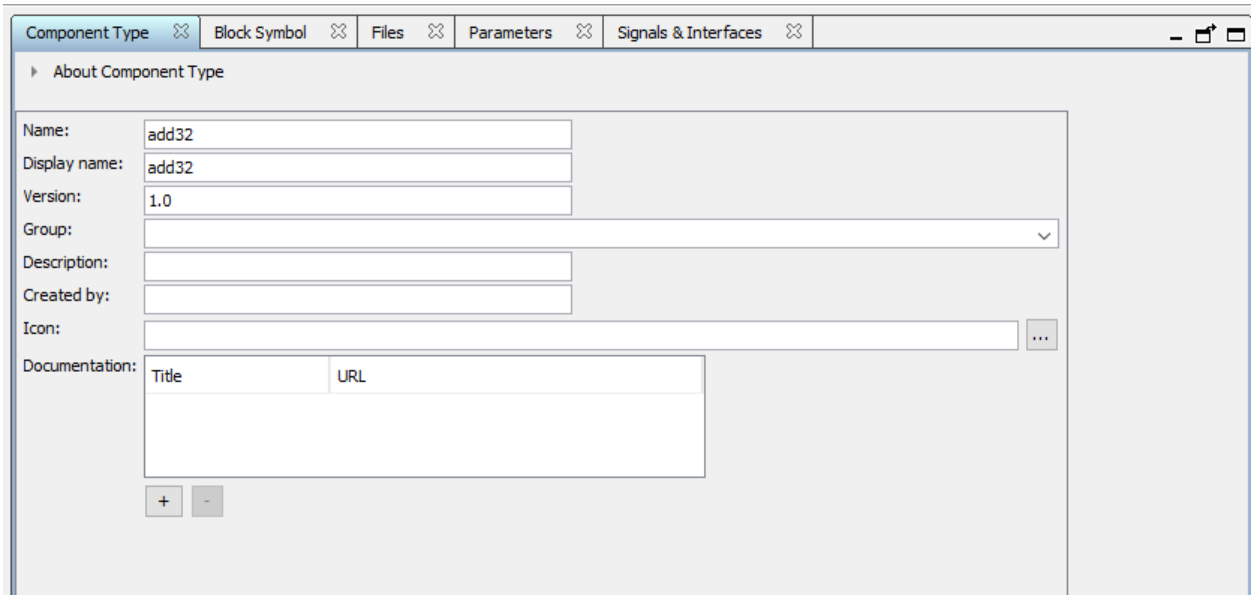
i. Add all your required custom VHDL files to the "Files" tab. Ensure the top-level file is set to have the attribute as top-level file. Click "Analyze Synthesis Files."



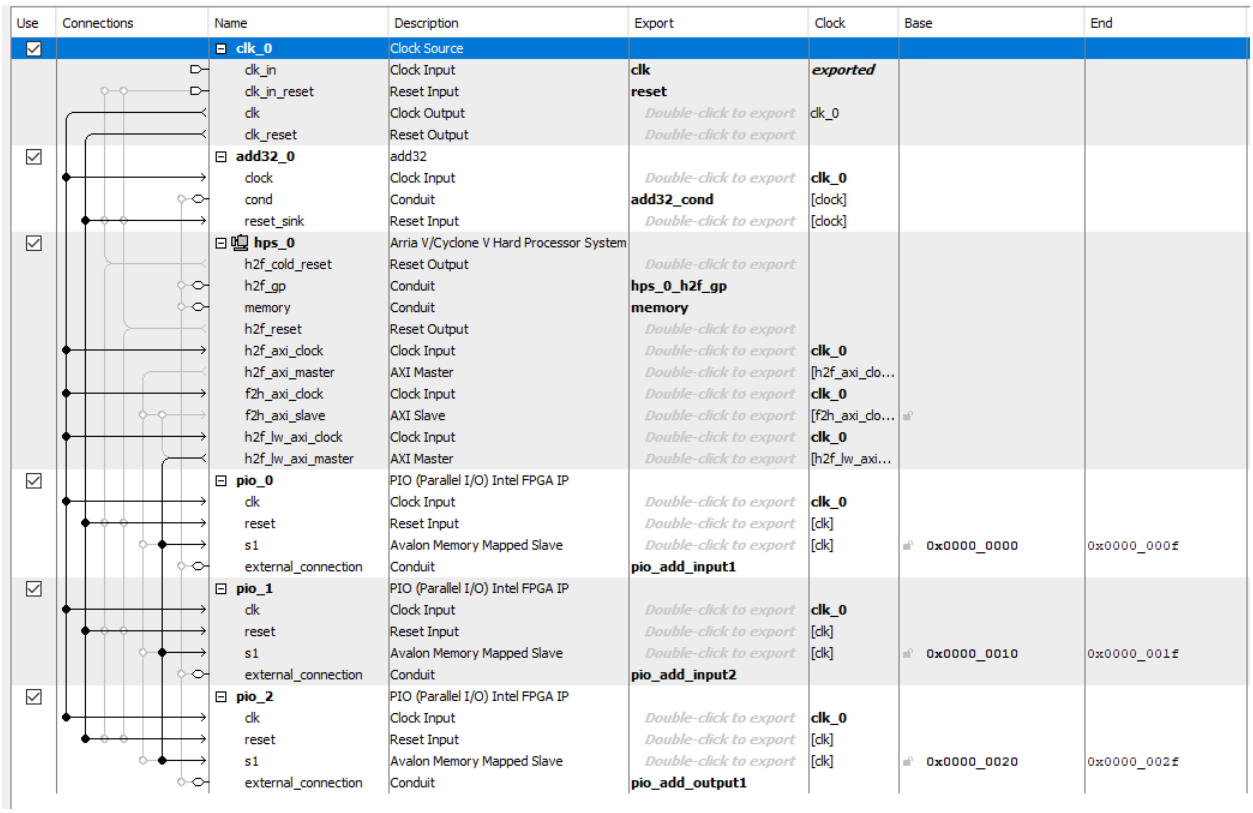
ii. Go to the Signals and Interfaces tab and set all signals as conduits, other than clk or reset. Clock must be set under "Clock Input" and the reset must be set under "Reset Input."
iii. The input and output signals that you reassigned to be conduits will have odd names now in light grey such as "readdata," "burstenable," and others. This is not wanted. Rename the signals to something that makes sense. This is not necessary, but these names are how the signals will appear in the auto generated VHDL, so it is recommended to rename them to make it easier to understand what you are doing later.



iv. Rename the component to something different than the top level VHDL file, the Platform Designer project, or the Quartus project. In general, just use new names for everything

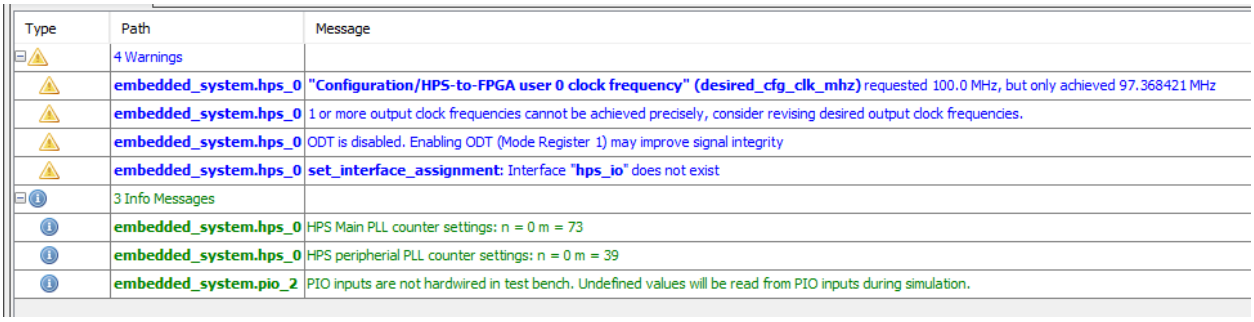


6. Connect everything together
    a. Make all the connections by clicking on the empty circles.
        i. All clk inputs must be connected,
        ii. All resets should be connected to the clk_reset.
        iii. Connect the line “h2f_lw_axi_master” on the hps component to the “s1” line on each of the PIO components.
    b. Double click to export the “Conduits” for the PIO components and your custom component. The grey text should become bolded



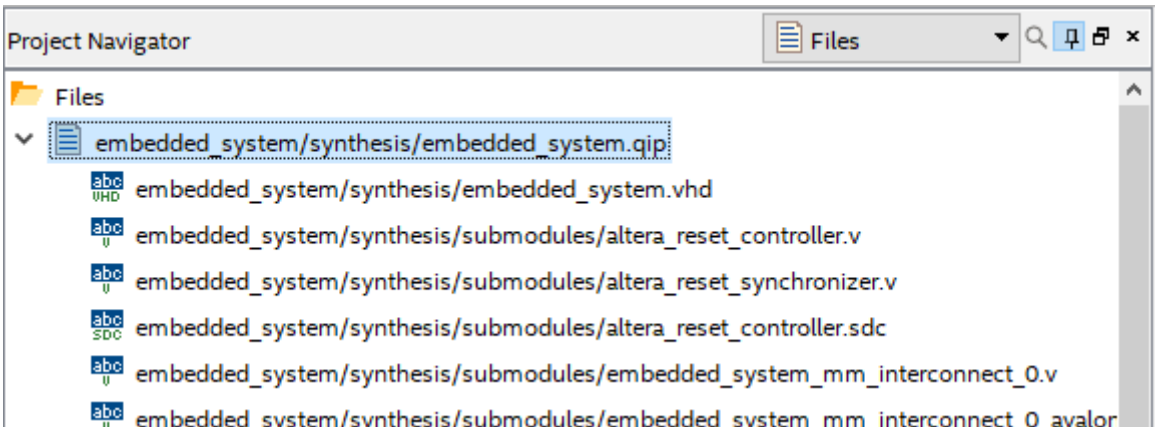
Use	Connections	Name	Description	Export	Clock	Base	End
☑		**clk_0**	Clock Source				
		clk_in	Clock Input	**clk**	***exported***		
		clk_in_reset	Reset Input	**reset**			
		clk	Clock Output	*Double-click to export*	clk_0		
		clk_reset	Reset Output	*Double-click to export*			
☑		**add32_0**	add32				
		clock	Clock Input	*Double-click to export*	**clk_0**		
		cond	Conduit	**add32_cond**	[clock]		
		reset_sink	Reset Input	*Double-click to export*	[clock]		
☑		**hps_0**	Arria V/Cyclone V Hard Processor System				
		h2f_cold_reset	Reset Output	*Double-click to export*			
		h2f_gp	Conduit	**hps_0_h2f_gp**			
		memory	Conduit	**memory**			
		h2f_reset	Reset Output	*Double-click to export*			
		h2f_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_clo...		
		f2h_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_clo...		
		h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi...		
☑		**pio_0**	PIO (Parallel I/O) Intel FPGA IP				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clk]		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	**0x0000_0000**	0x0000_000f
		external_connection	Conduit	**pio_add_input1**			
☑		**pio_1**	PIO (Parallel I/O) Intel FPGA IP				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clk]		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	**0x0000_0010**	0x0000_001f
		external_connection	Conduit	**pio_add_input2**			
☑		**pio_2**	PIO (Parallel I/O) Intel FPGA IP				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clk]		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	**0x0000_0020**	0x0000_002f
		external_connection	Conduit	**pio_add_output1**			

7. Set unique addresses for each PIO component (e.g., 0000-000f, 0010-001f, 0020-002f). Make sure you know these addresses, as these are how you will read/write to your component
8. Verify that you have no red errors in the messages at the bottom. You should expect to see four warnings

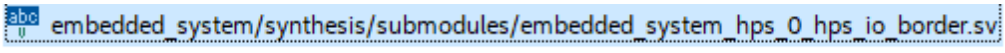


Type	Path	Message
⚠	4 Warnings	
⚠	**embedded_system.hps_0**	**"Configuration/HPS-to-FPGA user 0 clock frequency" (desired_cfg_clk_mhz)** requested 100.0 MHz, but only achieved 97.368421 MHz
⚠	**embedded_system.hps_0**	1 or more output clock frequencies cannot be achieved precisely, consider revising desired output clock frequencies.
⚠	**embedded_system.hps_0**	ODT is disabled. Enabling ODT (Mode Register 1) may improve signal integrity
⚠	**embedded_system.hps_0**	**set_interface_assignment:** Interface "**hps_io**" does not exist
ℹ	3 Info Messages	
ℹ	**embedded_system.hps_0**	HPS Main PLL counter settings: n = 0 m = 73
ℹ	**embedded_system.hps_0**	HPS peripherial PLL counter settings: n = 0 m = 39
ℹ	**embedded_system.pio_2**	PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.

9. Generate HDL from platform designer, bottom right of screen. We are now done with Platform Designer
10. Back in main Quartus, add the generated .qip file to the project. The .qip file is placed in /" Quartus Project Name"/" Platform Designer Project Name"/synthesis
11. Open the .qip file by double clicking it and delete all lines that reference SDRAM
    a. Use Ctrl-f, type SDRAM, delete all lines with reference
    b. Delete all SDRAM module in the border file
12. Delete SDRAM component from border file



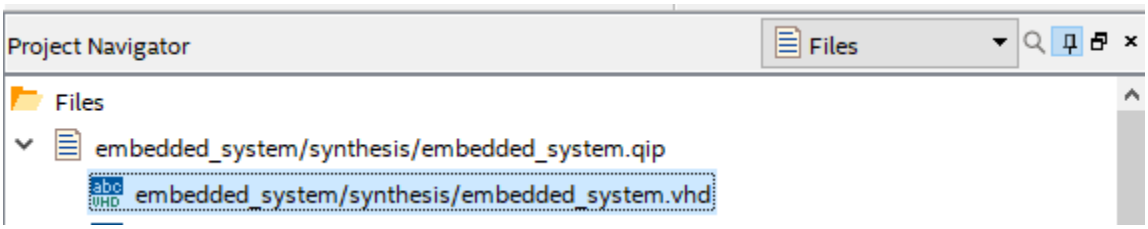
a. Click the arrow on the .qip file to see the subfiles
b. Scroll to the bottom and open the following file

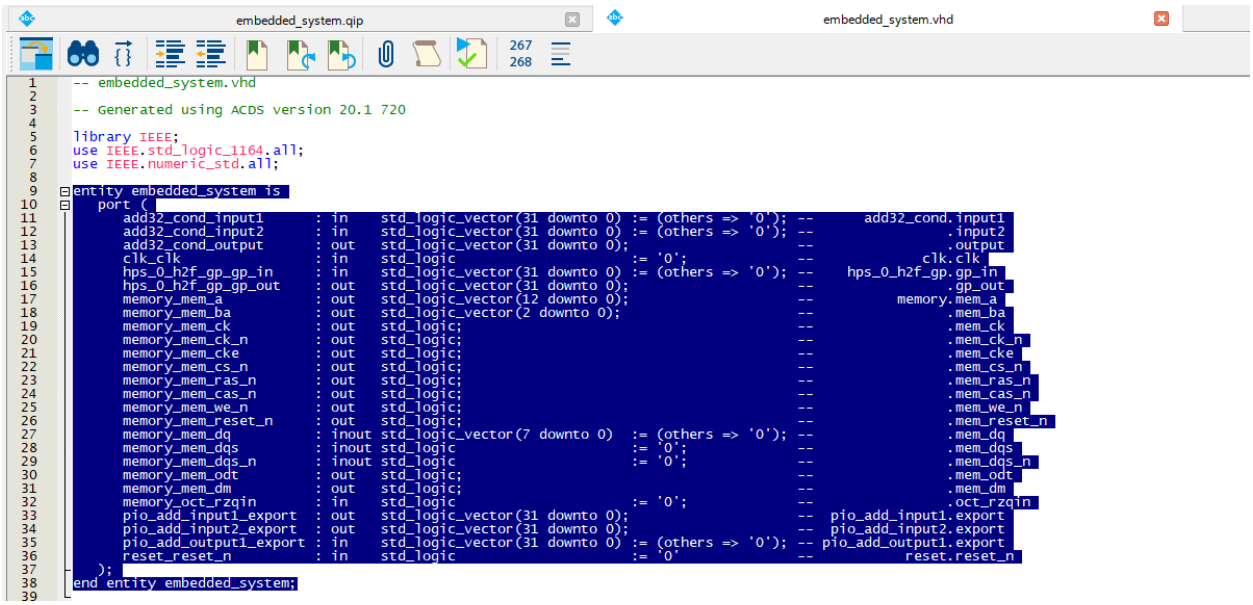


c. Delete from the file the second component instantiation, the SDRAM one. Only remove the highlighted text shown below.

```
31      ,input wire [1 - 1 : 0 ] oct_rzqin
32    );
33
34
35    hps_sdram hps_sdram_inst(
36     .mem_dq({
37        mem_dq[7:0] // 7:0
38      })
39    ,.mem_odt({
40        mem_odt[0:0] // 0:0
41      })
42    ,.mem_ras_n({
43        mem_ras_n[0:0] // 0:0
44      })
45    ,.mem_dqs_n({
46        mem_dqs_n[0:0] // 0:0
47      })
48    ,.mem_dqs({
49        mem_dqs[0:0] // 0:0
50      })
51    ,.mem_dm({
52        mem_dm[0:0] // 0:0
53      })
54    ,.mem_we_n({
55        mem_we_n[0:0] // 0:0
56      })
57    ,.mem_cas_n({
58        mem_cas_n[0:0] // 0:0
59      })
60    ,.mem_ba({
61        mem_ba[2:0] // 2:0
62      })
63    ,.mem_a({
64        mem_a[12:0] // 12:0
65      })
66    ,.mem_cs_n({
67        mem_cs_n[0:0] // 0:0
68      })
69    ,.mem_ck({
70        mem_ck[0:0] // 0:0
71      })
72    ,.mem_cke({
73        mem_cke[0:0] // 0:0
74      })
75    ,.oct_rzqin({
76        oct_rzqin[0:0] // 0:0
77      })
78    ,.mem_reset_n({
79        mem_reset_n[0:0] // 0:0
80      })
81    ,.mem_ck_n({
82        mem_ck_n[0:0] // 0:0
83      })
84    );
85
86    endmodule
87
88
```

13. Create a new top-level file that instantiates the .qip component
    a. Open the .vhd file generated under the .qip file






```
-- embedded_system.vhd

-- Generated using ACDS version 20.1 720

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity embedded_system is
    port (
        add32_cond_input1       : in    std_logic_vector(31 downto 0) := (others => '0'); --       add32_cond.input1
        add32_cond_input2       : in    std_logic_vector(31 downto 0) := (others => '0'); --                 .input2
        add32_cond_output       : out   std_logic_vector(31 downto 0);                    --                 .output
        clk_clk                 : in    std_logic                     := '0';             --              clk.clk
        hps_0_h2f_gp_gp_in      : in    std_logic_vector(31 downto 0) := (others => '0'); --    hps_0_h2f_gp.gp_in
        hps_0_h2f_gp_gp_out     : out   std_logic_vector(31 downto 0);                    --                 .gp_out
        memory_mem_a            : out   std_logic_vector(12 downto 0);                    --           memory.mem_a
        memory_mem_ba           : out   std_logic_vector(2 downto 0);                     --                 .mem_ba
        memory_mem_ck           : out   std_logic;                                        --                 .mem_ck
        memory_mem_ck_n         : out   std_logic;                                        --                 .mem_ck_n
        memory_mem_cke          : out   std_logic;                                        --                 .mem_cke
        memory_mem_cs_n         : out   std_logic;                                        --                 .mem_cs_n
        memory_mem_ras_n        : out   std_logic;                                        --                 .mem_ras_n
        memory_mem_cas_n        : out   std_logic;                                        --                 .mem_cas_n
        memory_mem_we_n         : out   std_logic;                                        --                 .mem_we_n
        memory_mem_reset_n      : out   std_logic;                                        --                 .mem_reset_n
        memory_mem_dq           : inout std_logic_vector(7 downto 0)  := (others => '0'); --                 .mem_dq
        memory_mem_dqs          : inout std_logic                     := '0';             --                 .mem_dqs
        memory_mem_dqs_n        : inout std_logic                     := '0';             --                 .mem_dqs_n
        memory_mem_odt          : out   std_logic;                                        --                 .mem_odt
        memory_mem_dm           : out   std_logic;                                        --                 .mem_dm
        memory_oct_rzqin        : in    std_logic                     := '0';             --                 .oct_rzqin
        pio_add_input1_export   : out   std_logic_vector(31 downto 0);                    --  pio_add_input1.export
        pio_add_input2_export   : out   std_logic_vector(31 downto 0);                    --  pio_add_input2.export
        pio_add_output1_export  : in    std_logic_vector(31 downto 0) := (others => '0'); -- pio_add_output1.export
        reset_reset_n           : in    std_logic                     := '0'              --            reset.reset_n
    );
end entity embedded_system;
```

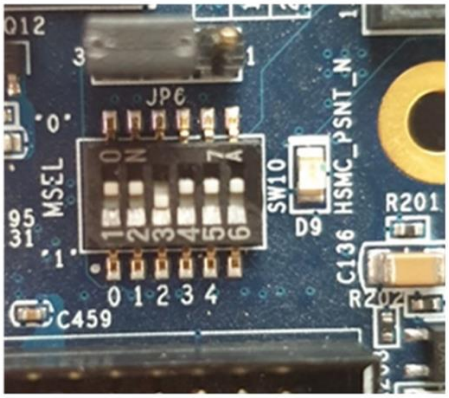
b. Instantiate the top entity in this vhd file in a new custom VHDL file

c. This is why the PIO connections were created opposite of the custom component. The qip component should have matching interfaces for both the PIO components and the custom component. Simply match the correct signals together. As shown below, the memory connections are ignored as they are not being used. Instantiate the clk_clk signal and connect the custom component IOs to the PIO IOs.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity add_toplevel is
port(
clk, resetn : in std_logic
);
end add_toplevel;

architecture behavior of add_toplevel is

signal input1, input2, output1 : std_logic_vector(31 downto 0);

component embedded_system is
    port (
        add32_cond_input1      : in    std_logic_vector(31 downto 0) := (others => '0'); --      add32_cond.input1
        add32_cond_input2      : in    std_logic_vector(31 downto 0) := (others => '0'); --                .input2
        add32_cond_output      : out   std_logic_vector(31 downto 0);                    --                .output
        clk_clk                : in    std_logic                     := '0';             --            clk.clk
        hps_0_h2f_gp_gp_in     : in    std_logic_vector(31 downto 0) := (others => '0'); --    hps_0_h2f_gp.gp_in
        hps_0_h2f_gp_gp_out    : out   std_logic_vector(31 downto 0);                    --                .gp_out
        memory_mem_a           : out   std_logic_vector(12 downto 0);                    --          memory.mem_a
        memory_mem_ba          : out   std_logic_vector(2 downto 0);                     --                .mem_ba
        memory_mem_ck          : out   std_logic;                                        --                .mem_ck
        memory_mem_ck_n        : out   std_logic;                                        --                .mem_ck_n
        memory_mem_cke         : out   std_logic;                                        --                .mem_cke
        memory_mem_cs_n        : out   std_logic;                                        --                .mem_cs_n
        memory_mem_ras_n       : out   std_logic;                                        --                .mem_ras_n
        memory_mem_cas_n       : out   std_logic;                                        --                .mem_cas_n
        memory_mem_we_n        : out   std_logic;                                        --                .mem_we_n
        memory_mem_reset_n     : out   std_logic;                                        --                .mem_reset_n
        memory_mem_dq          : inout std_logic_vector(7 downto 0)  := (others => '0'); --                .mem_dq
        memory_mem_dqs         : inout std_logic                     := '0';             --                .mem_dqs
        memory_mem_dqs_n       : inout std_logic                     := '0';             --                .mem_dqs_n
        memory_mem_odt         : out   std_logic;                                        --                .mem_odt
        memory_mem_dm          : out   std_logic;                                        --                .mem_dm
        memory_oct_rzqin       : in    std_logic                     := '0';             --                .oct_rzqin
        pio_add_input1_export  : out   std_logic_vector(31 downto 0);                    --  pio_add_input1.export
        pio_add_input2_export  : out   std_logic_vector(31 downto 0);                    --  pio_add_input2.export
        pio_add_output1_export : in    std_logic_vector(31 downto 0) := (others => '0'); -- pio_add_output1.export
        reset_reset_n          : in    std_logic                     := '0'              --           reset.reset_n
        );
end component;

begin

embed: embedded_system port map(
add32_cond_input1 => input1,
add32_cond_input2 => input2,
add32_cond_output => output1,
clk_clk => clk,
reset_reset_n => resetn,
pio_add_input1_export => input1,
pio_add_input2_export => input2,
pio_add_output1_export => output1
);

end behavior;
```

14. Add this new top-level file to the Quartus project. The only two files added to the project should be the .qip file, and the top-level file. Ensure the new VHDL file is the top-level file, not the .qip file.
15. Set the clk pin (and anything else needed) in pin planner
16. Compile the design
17. Convert the created output file to .rbf format with mode 16x parallel. You can do this from Quartus by clicking File -> Convert Programming Files
18. Make sure you have the output file (in the .rbf format) saved successfully on your computer. You will need it for future steps.
    a. Note: Until now, you have done with Quartus. You can close it if you want.

19. Install the rsyocto Linux image on the SD card instead of the default yoctolinux provided by Terasic. We are using this due to the easier method of addressing for the AXI bus.
    a. Use an SD Card reader to insert the empty SD Card to the Linux PC
    b. Download rsYocto_1_042_DE10STD.zip from https://github.com/robseb/rsyocto/releases
    c. Use Etcher to boot rsYocto_1_042_DE10STD.zip on the SD Card
20. Move the .rbf file onto the Linux image
    a. Treat SD Card as an USB drive on Linux PC
    b. Move the .rbf file into the /home/root directory
    c. Eject the SD card from the Linux PC
21. Prepare the board
    a. Insert the SD card into the board
    b. Set the MSEL Bit of the board to the following setting:



    c. Connect the board to the Linux PC using UART to USB Cable
    d. Connect the board to the power using the power cable
    e. Press the power button to power on the board
22. Connect the board to the Linux PC using Minicom
    a. On Linux PC, open Minicom to set up the board's Serial Connection
        i. Before opening Minicom, we need to find the board name
            1. To do this, open a terminal on Linux PC
            2. Execute the following command
                a. ls /dev/cu.*
                b. After this, find the board
                c. It might be shown as something like this:
                d. /dev/cu.usbserial-AU02GOA8
                e. Make sure to copy this (what is shown above in part d) down, you will need this in the next step

ii. After getting the board name, open the Minicom
    1. To do this, in the terminal on Linux PC
        a. Execute the following command:
            i. minicom -s
            ii. After opening minicom, your terminal would be like this

```
+-----[configuration]------+
| Filenames and paths      |
| File transfer protocols  |
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
+--------------------------+
```

    2. Open Serial port setup in minicom
            i. After opening it, your terminal would be like this

```
+-----------------------------------------------------------------------+
| A -    Serial Device      : /dev/cu.usbserial-AU02GOA8                |
| B - Lockfile Location     : /usr/local/Cellar/minicom/2.8/var         |
| C -   Callin Program      :                                           |
| D -  Callout Program      :                                           |
| E -    Bps/Par/Bits       : 115200 8N1                                |
| F - Hardware Flow Control : Yes                                       |
| G - Software Flow Control : No                                        |
| H -     RS485 Enable      : No                                        |
| I -   RS485 Rts On Send   : No                                        |
| J -  RS485 Rts After Send : No                                        |
| K -  RS485 Rx During Tx   : No                                        |
| L -  RS485 Terminate Bus  : No                                        |
| M - RS485 Delay Rts Before: 0                                         |
| N - RS485 Delay Rts After : 0                                         |
|                                                                       |
|    Change which setting?                                              |
+-----------------------------------------------------------------------+
```

    3. Paste what you got in step 22(a)(i)(2)(d) to the “A - Serial Device” shown in the screen above
b. After connecting to Minicom successfully, power off and then power on the board again, rsyocto boots automatically

23. Log in as a root user
    a. After rsyocto finishes booting, terminal will prompt you to login
        i. Login: root
        ii. Password: eit

24. Load the .rbf file and read/write to the LW AXI bridge using the following instructions:
    a. In the terminal (the same terminal in step 23(a)), execute the following commands
        i. FPGA-writeConfig -f file_name.rbf
            1. This command loads the FPGA configuration you want onto the FPGA
        ii. FPGA-writeBridge -lw 0 -h ab
            1. This command writes data 0xab to the light weight AXI bridge offsetting at address 0
        iii. FPGA-readBridge -lw 0
            1. This command reads the light weight AXI bridge offsetting at address 0

25. Congratulations! We are done!