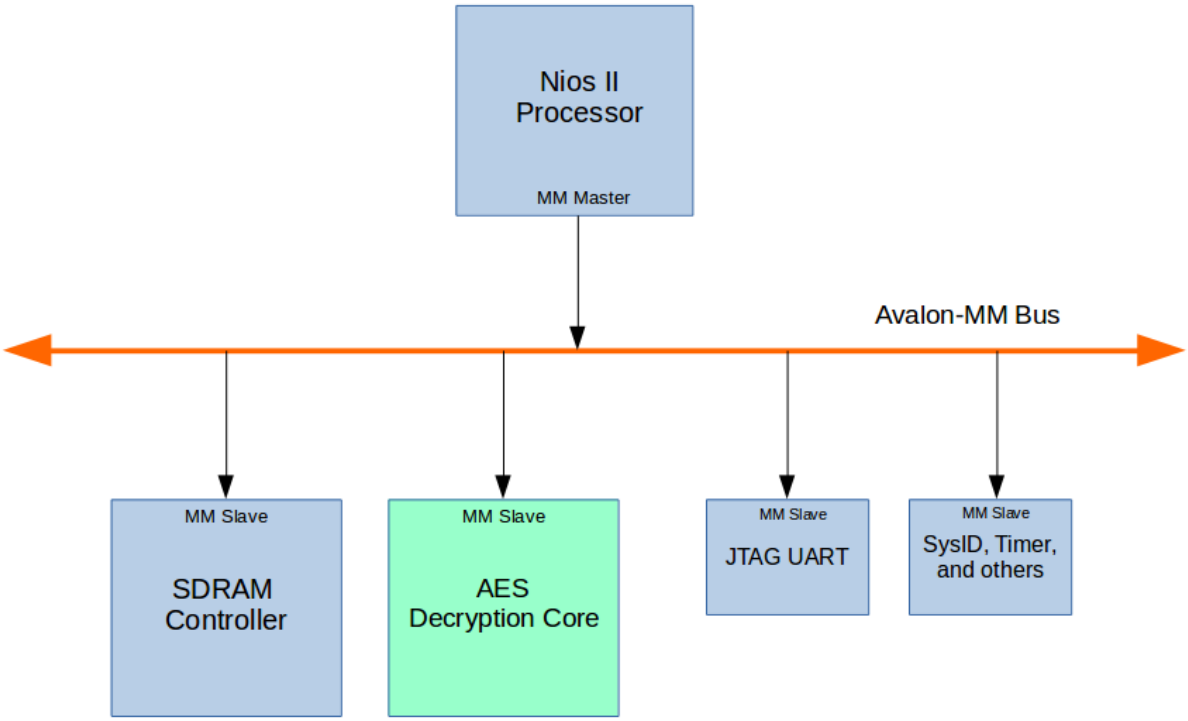
Introduction to the Avalon-MM Interface

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

System Overview

In this lab, we want to create and add an AES decryption IP core on the Avalon-MM (Memory Mapped) interconnect. Nios II will perform encryption in software and communicate with that IP to perform decryption in hardware.



Your custom IP core will have an Avalon-MM Slave port that allows Avalon-MM Masters like Nios II to directly access with read/write operations. To perform decryption, Nios II will write the 128-bit Encrypted Message and Key to the AES Decryption Core, then write a start signal to one of its registers, and wait until decryption is complete in hardware to read back the 128-bit Decrypted Message.

To begin, start with your Qsys setup from Lab 8 with the usual components like Nios II, SDRAM, and JTAG UART, add an interval timer (see page 15), and remove the unnecessary USB related components. Save and close your Qsys window, there's no need to generate HDL right now. In the next section, you will design your own AES Decryption Core component and add it to Qsys.

The AES Decryption Core Interface

The interface module *avalon_aes_interface.sv* will be the top-level file for the AES Decryption Core component on Qsys (note that *lab9_top.sv* is still the top-level file for the entire project). We have provided the input/output signals declaration for you. There is a clock input (CLK), an active-high reset input (RESET), an exported conduit signal which is just a 32-bit output (EXPORT_DATA), and finally an Avalon-MM Slave port which contains a variety of signals whose specifications will be described below.

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

Avalon-MM Slave Port Interface Signals:

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

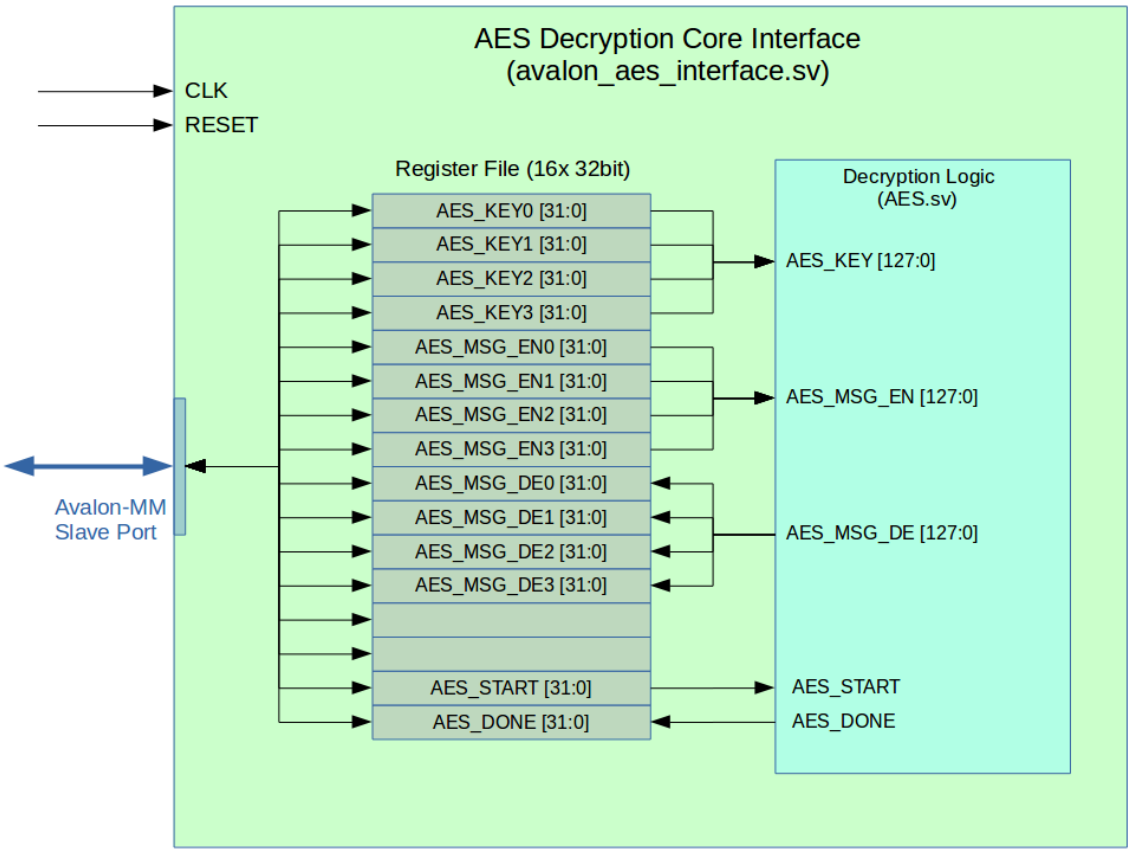
Note that the data width of 32-bit and address width of 4-bit are chosen for this lab, they may be up to 1024-bit and 64-bit respectively. We are using 32-bit readdata/writedata signals because they match the data width of Nios II, which is a 32-bit processor. As for the 4-bit address, which gives $2^4 = 16$ locations, is just enough to hold all the AES related data.

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 16 registers, each 32-bit, that hold the values being read and written. There are some requirements that your design must satisfy:

- Read has a 0 cycle **wait latency**. In other words, when read is high, readdata should have the value of the addressed register on the same cycle.
- Write has a 0 cycle **wait latency**. In other words, when write is high, the addressed register should be updated with writedata on the next cycle.
- Byte enable determines the bytes being written according to the following table, observe that each bit in *byteenable* determines the action of a corresponding byte in *writedata*.

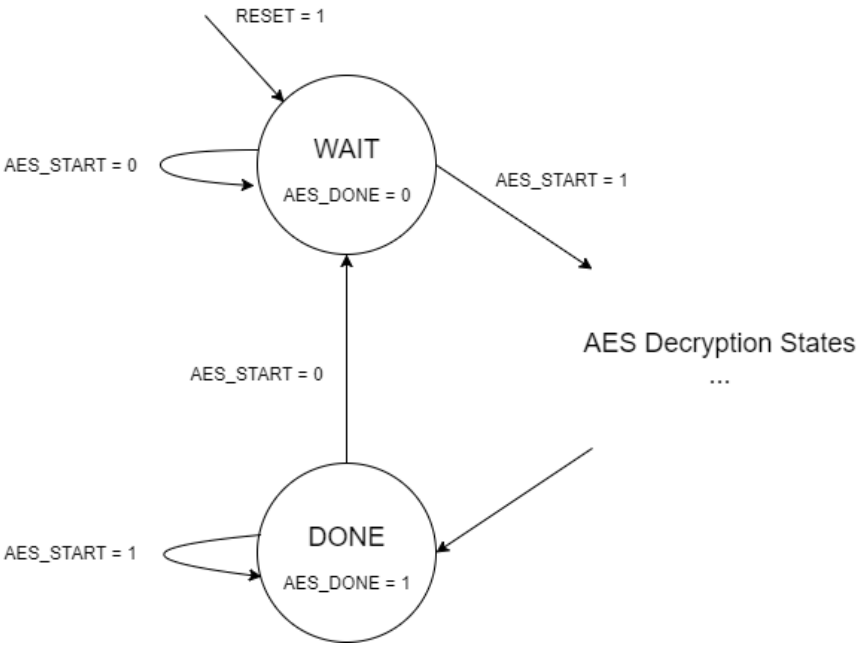
byteenable[3:0]	Write Action
1111	Write full 32-bits
1100	Write the two upper bytes
0011	Write the two lower bytes
1000	Write byte 3 only
0100	Write byte 2 only
0010	Write byte 1 only
0001	Write byte 0 only

Nios will send the 128-bit AES Key and Encrypted Message as 4 writes ($4 \times 32\ bit = 128\ bit$) each, thus we need 4 registers to hold each of the AES Key, Encrypted Message, and Decrypted Message, yielding a total of 12 registers. We need 2 more registers to hold the START and DONE signals that Nios will use to control the hardware state, this brings the total to 14. Since address ranges are in power of 2, we will use 14 of the 16 addressable registers, the remaining 2 simply won't be used. The recommended implementation of *avalon_aes_interface.sv* is shown below.



For week 1, implementing the register array and making sure that you can write your AES Key and Encrypted Message to them will suffice. Keep in mind that EXPORT_DATA should be assigned to the first 2 and last 2 bytes of AES Key, exported from the Qsys design (covered in next section), and displayed on the LEDs with hexdrivers on the lab 9 top level. In software, you need to implement the full AES encryption algorithm and write the key and encrypted message to the specified registers in the figure on page 3, see the section on **Nios Software** (page 14).

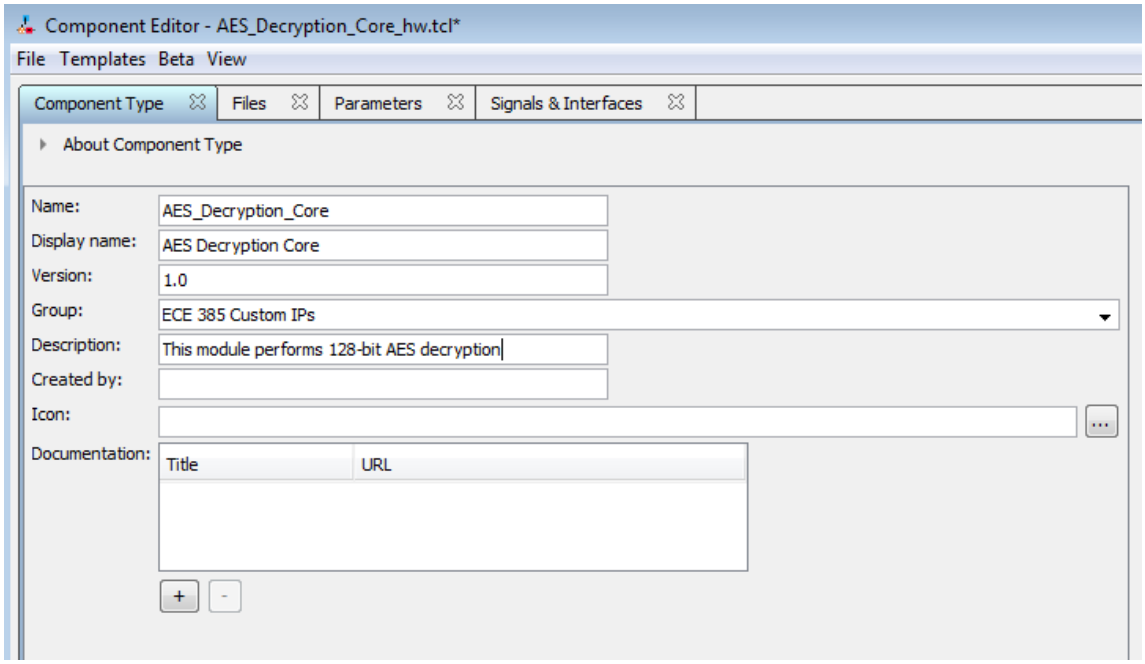
For week 2, instantiate *AES.sv*, the decryption logic with your state machine, and make the appropriate connections by packing or unpacking the 4x 32-bit registers to 128-bit input and output ports for AES Key, Encrypted Message, and Decrypted Message. The START and DONE signals are 1 bit so you can simply use the last bit of Registers 15 and 16 as shown in the figure. Your state machine in *AES.sv* should be able to perform decryption continuously. Create two control states: WAIT and DONE to handle the START input and DONE output signals as described below.



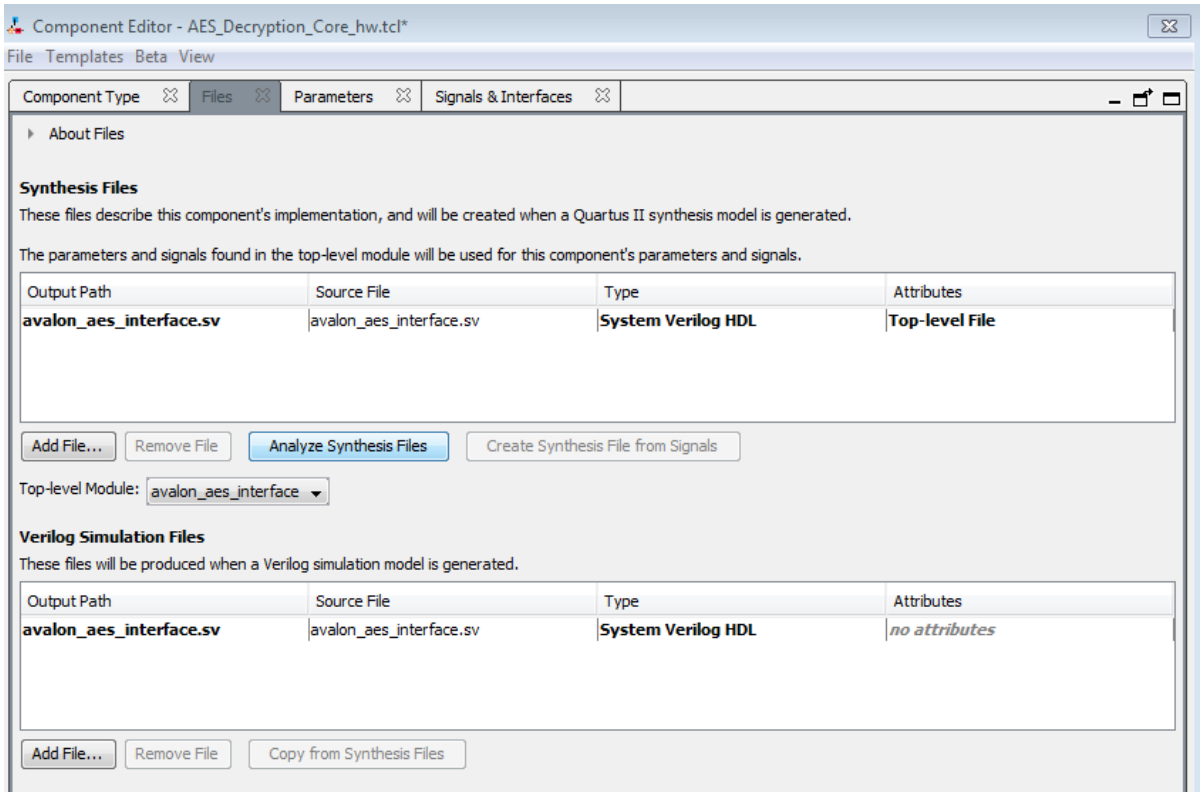
Creating a Qsys Component

When you are done implementing the partial or complete interface module *avalon_aes_interface.sv*, add it to the Qsys IP catalog with component editor.

1. Launch Qsys and load your design that already has Nios II, SDRAM, UART, etc.
2. On the upper left **IP Catalog** panel, double click ***New Component…***

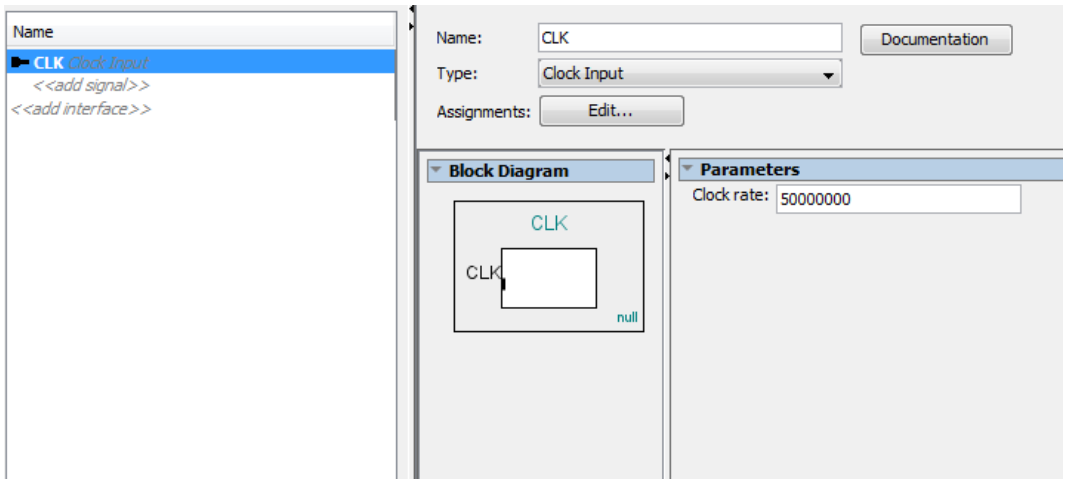


3. Enter the Name and Display name for your component. Display name is the one shown in Qsys IP Catalog. It is good practice to also keep track of the version of your IP, increment it when you make changes or fix issues. You can also categorize it to an existing or new group, here we make a group called “ECE 385 Custom IPs”. Finally, give it a brief description.
4. Click on the Files tab.



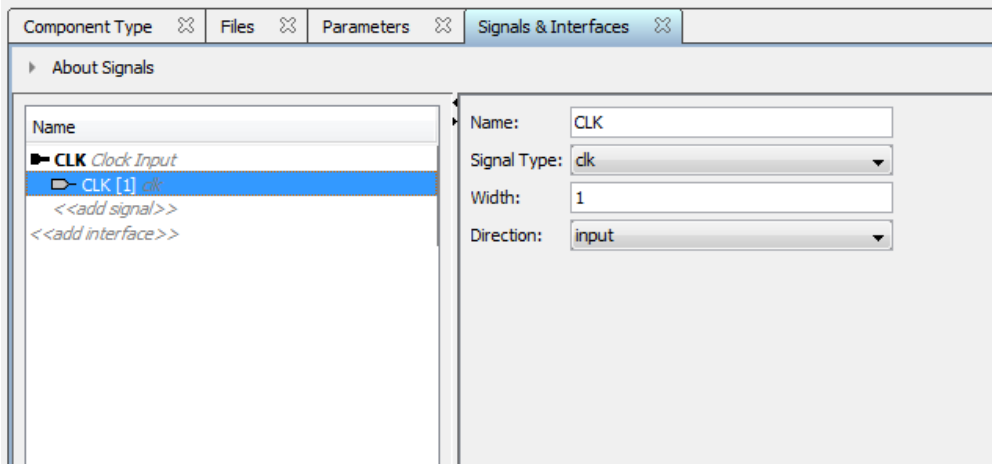
5. Under **Synthesis Files**, click on Add File… and choose *avalon_aes_interface.sv* which is the top-level module for this component. Under **Verilog Simulation Files**, click on **Copy from Synthesis Files**, this is useful if you want to simulate your system in ModelSim.
6. Click on the Signals & Interfaces tab. Now we want to create the Avalon ports and match the Avalon defined interface signals with our input/output declarations in *avalon_aes_interface.sv*.
7. Let's add the clock input first. Click on <<*add interface*>>.
   - In Quartus 15.0, a default name of "avalon_slave" will appear, replace that with "CLK" and press Enter. The default port type is Avalon Memory Mapped Slave, but we want a clock input, click on the drop-down list for **Type**, and choose **Clock Input**.
   - In Quartus 16.0, you are asked to choose from a list of port types, click on Clock Input, then rename it to "CLK" for **Name**.

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



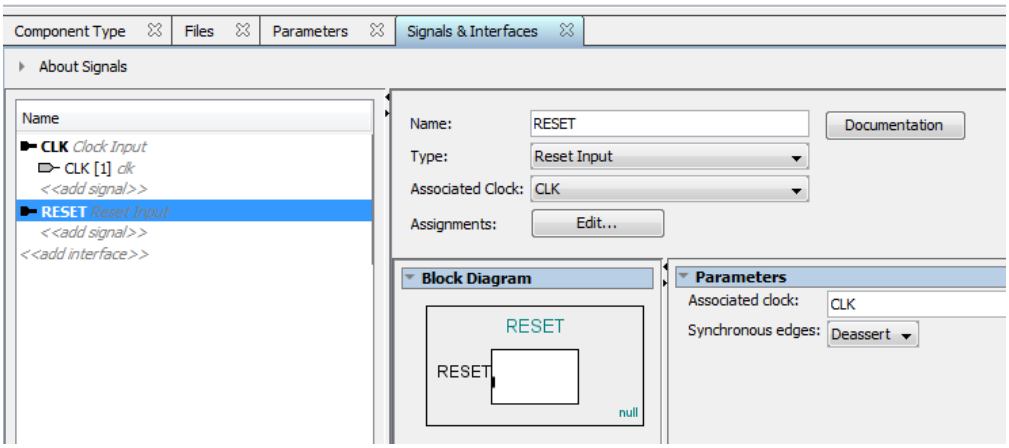
8. We now have a Clock Input interface for our component called CLK, but it's not associated with any signals defined in our top level module yet. Obviously, the clock input defined in *avalon_aes_interface.sv* is "input logic CLK". To make that association, click on <<*add signal*>>.
   - In Quartus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

   The remaining default values should be correct, if not, change them to match the picture below.

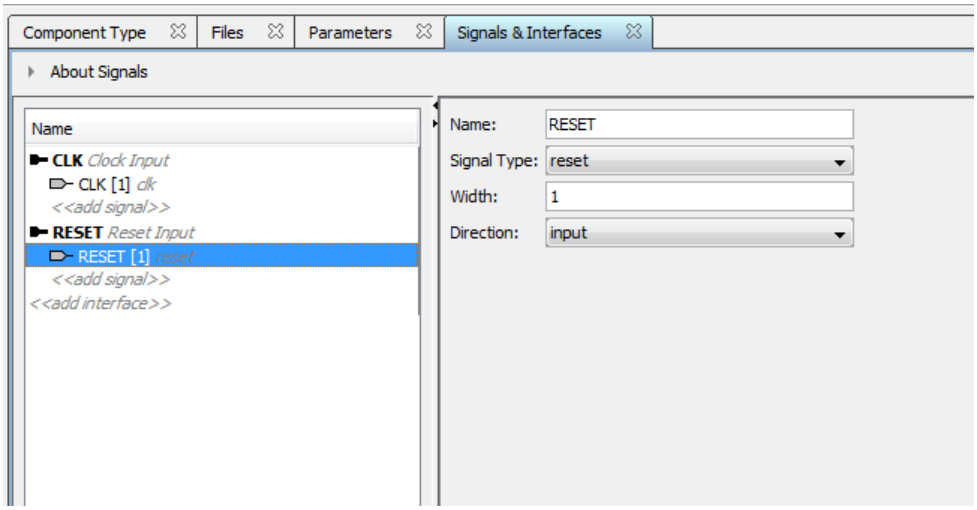


9. The Clock Input interface only needs one signal, so we are done with it. Now let's add the Reset Input interface. Click on *<<add interface>>*.
   - In Quartus 15.0, like the steps above, replace the default name with "RESET", then choose **Reset Input** for **Type**.
   - In Quartus 16.0, likewise, choose **Reset Input** from the list of port types, then change the name to "RESET".

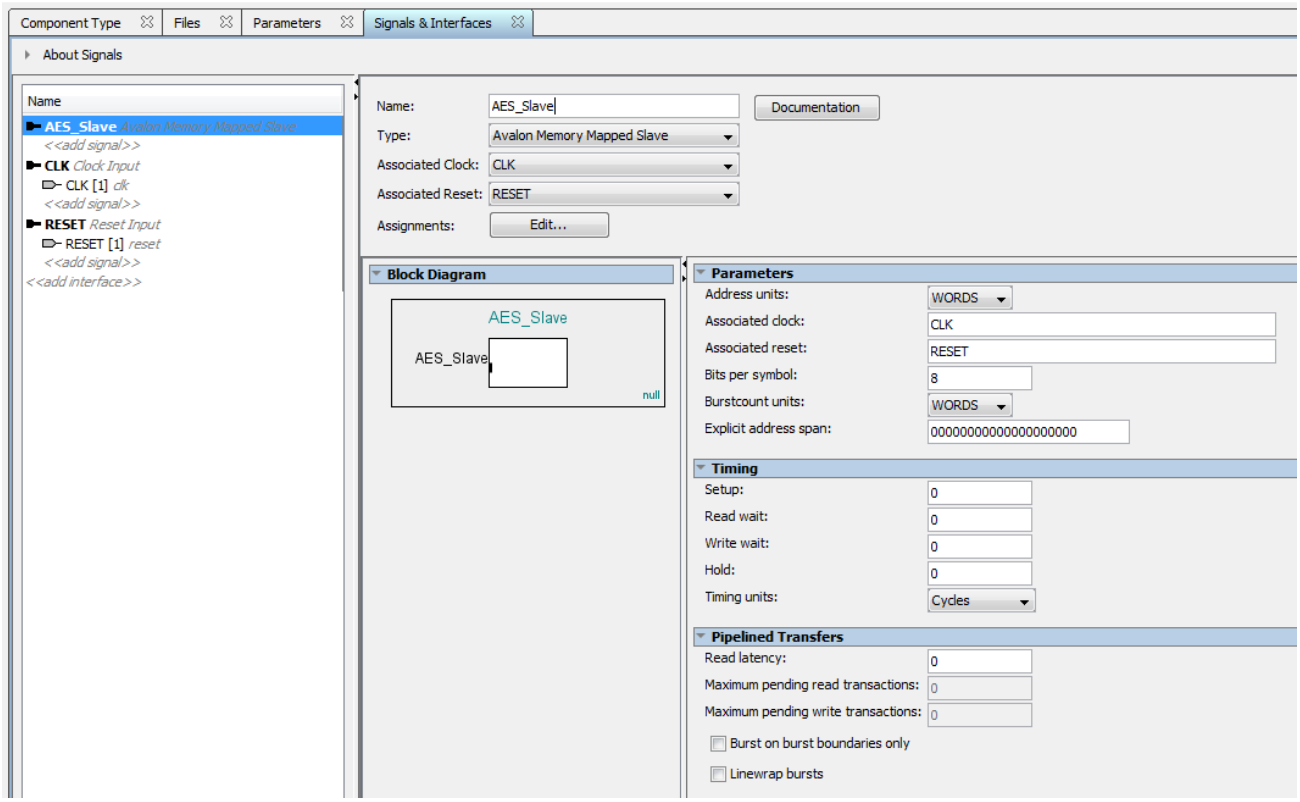
   The associated clock should be set to the "CLK" interface we just defined by default, if not, set it.



10. Now, add the associated RESET input to this interface. Click on *<<add signal>>*. Follow a similar procedure to Step 7, except use "RESET" for **Name**, and **reset** for **Type**.

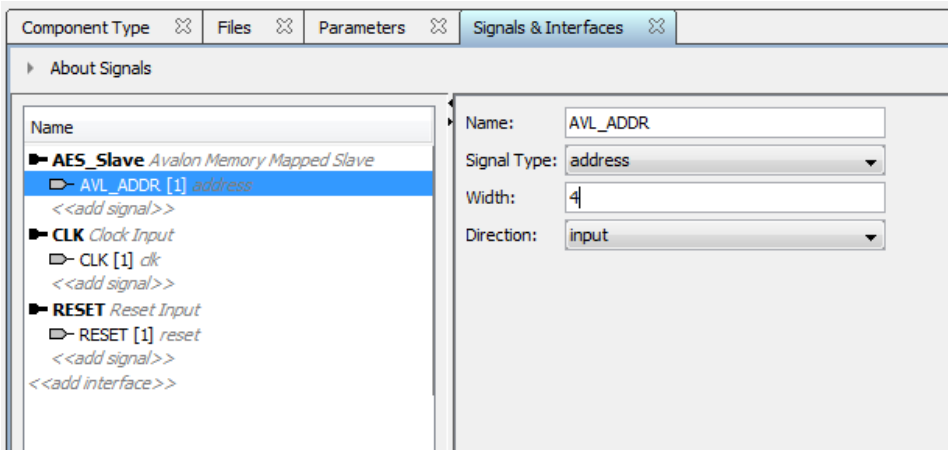


11. Next, add the main Avalon-MM Slave port. Click on <<*add interface*>>. You should be familiar with how to do this now, set the **Name** to "AES_Slave", **Type** to **Avalon Memory Mapped Slave**. Also set the **Associated Clock** and **Reset** to the CLK and RESET ports we defined earlier.

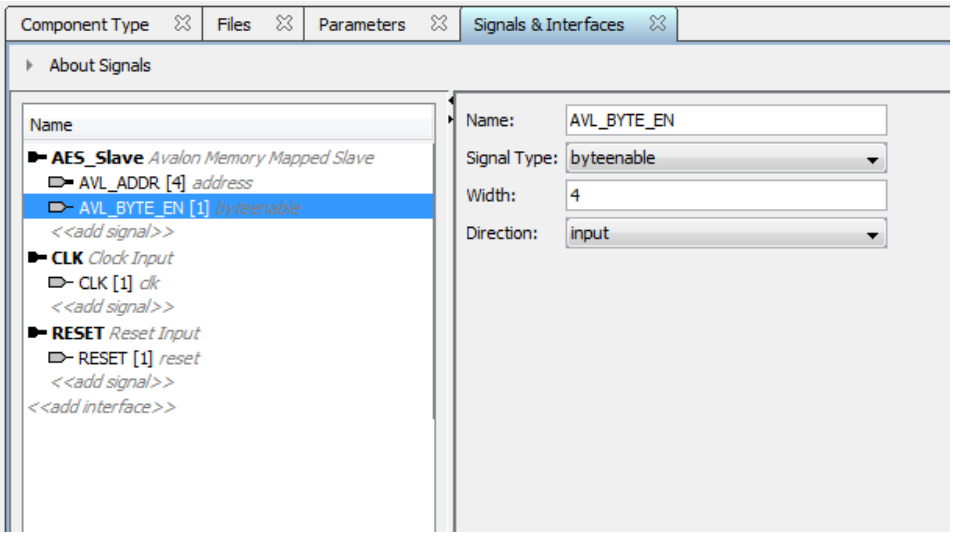


12. Under the **Parameters** section, check that you have identical settings as the picture above. Note that we make this Memory Mapped Slave byte addressable (8-bit per symbol) and the address units are in words (32-bit per word), so we expect the address to span a range of $4 \times 2^4 = 64$ (0x00 to 0x3F), making it no different than accessing no regular memory. In C, we can access this range directly as an array of 16 elements.

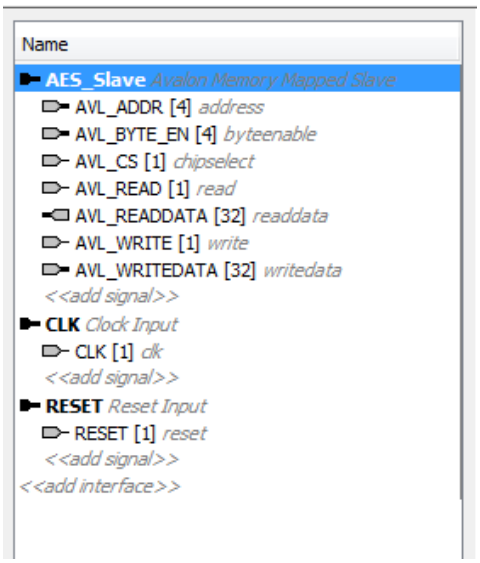
13. Under the **Timing** section, set both Read wait and Write wait to 0, as previously described, we want reads and writes to complete in the same cycle.
14. Now, we begin to add the relevant signals for this Memory Mapped slave port. Unlike previous ports, the MM Slave has multiple signals (read, write, chipselect, etc.) Let's start with the **address** signal, as before, click on *<<add signal>>* and set the **Name** to "AVL_ADDR", the **Signal Type** to **address**, **Width** to 4, and **Direction** to **input** in order to match what we declared in the top-level file *avalon_aes_interface.sv* ("input logic [3:0] AVL_ADDR").



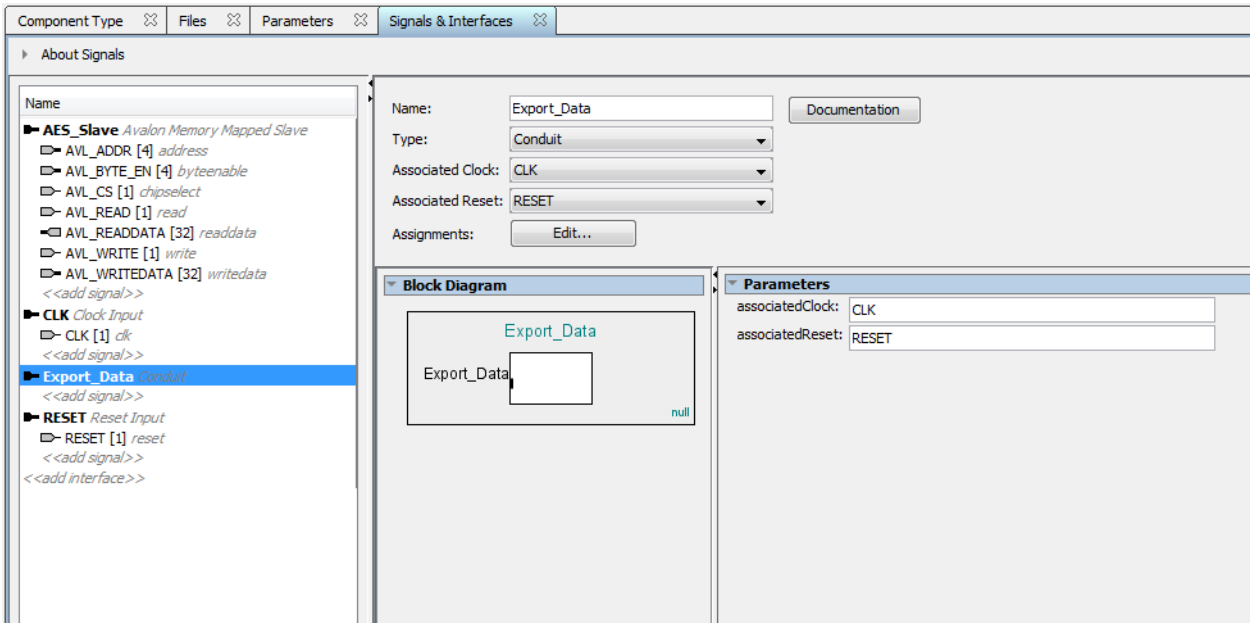
15. Add the **byteenable** signals next, click on *<<add signal>>* and set the **Name** to "AVL_BYTE_EN", the **Signal Type** to **byteenable**, **Width** to 4, and **Direction** to **input** once again to match what we declared in the top-level file *avalon_aes_interface.sv* ("input logic [3:0] AVL_BYTE_EN").



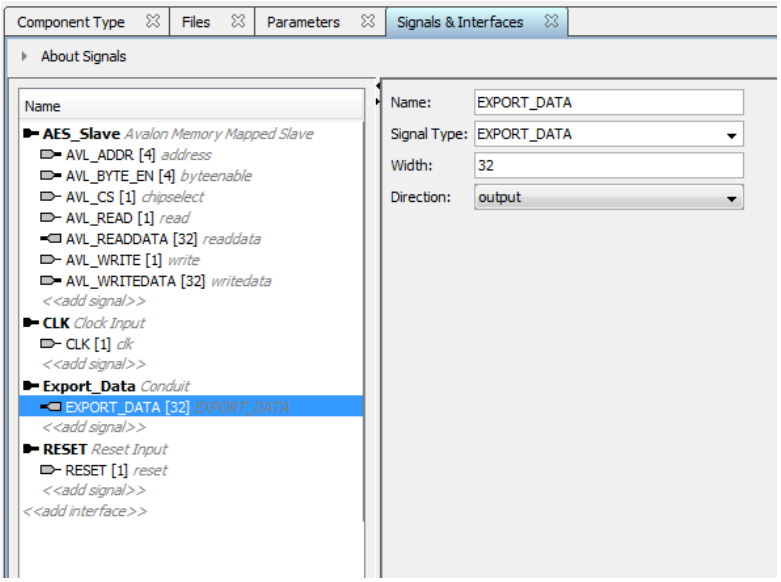
16. Complete the previous step the remaining signals for AES_Slave. Match the names, signal types, width, and direction for each signal. Be careful that **readdata** (AVL_READDATA) is an output unlike the others. When done, it should look like this:



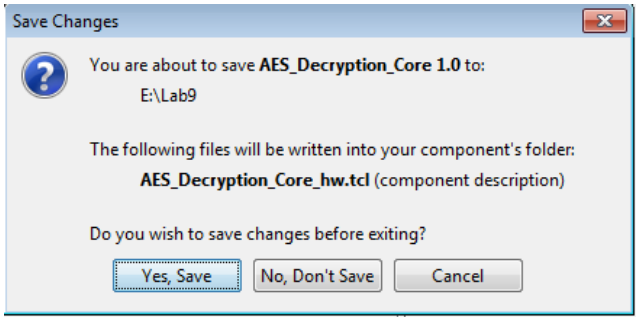
17. We are done with the AES_Slave. The last port to add is the exported conduit (EXPORT_DATA) to output part of the registers to the LEDs on the top level. Click on *<<add interface>>* and choose the settings as follows:



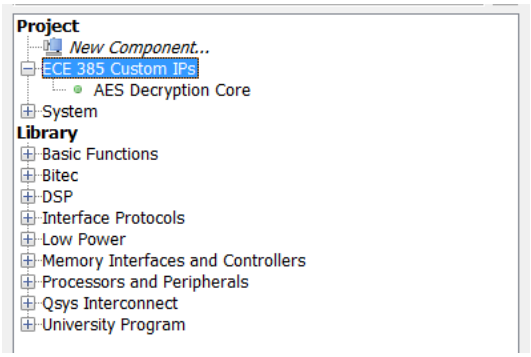
18. Add the only signal for this conduit, namely, the 32-bit EXPORT_DATA output signal. Set **Name** to EXPORT_DATA, **Signal Type** to * (it will be changed automatically or may appear as “new_signal” depending on your Quartus version), **Width** to 32, and **Direction** to output as shown below.



19. Check that there are no errors in the Message tab at the bottom. If there are errors like "[*port name*] Interface must have an associate clock/reset", click on that port on the left tab, and set the Associated Clock or Reset to CLK and RESET respectively if they have been reset to none (this is a minor bug that happens in some versions of Quartus). If there are no errors, click on *Finish…* at the bottom right and click "Yes, Save" to save the generated TCL script.

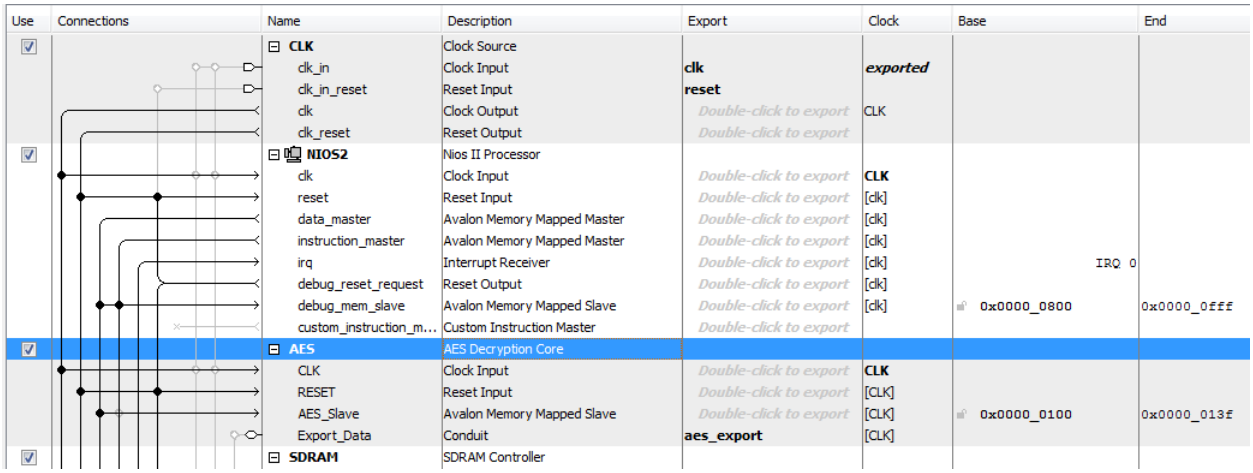


20. Now, in Qsys, your newly created AES Decryption Core component should appear in IP Catalog, grouped under "ECE 385 Custom IPs". It can now be added to Qsys like other Altera provided IPs. If you made a mistake, you can right-click it and click Edit to make any changes needed such as increasing the version number.



Finalizing your Qsys Design

Add your newly created component AES Decryption Core to Qsys by double clicking it. We did not define any parameters, so just click Finish to add it to Qsys, rename it if you want to. Make the appropriate **CLK** and **RESET** connections, and most importantly, connect the **AES_Slave** to Nios II's **data_master** to allow Nios to access its registers through reads and writes. We set its base address to 0x100 by default, you can choose a different address if it conflicts with your other components, and be sure to change it in software as well. Export the Export_Data conduit as "aes_export" and assign that signal to your HexDrivers on your top level file for this lab (*lab9_top.sv*).



Finally, generate the HDL for your Qsys design and include the QIP in your project.

IMPORTANT: Whenever you change the SystemVerilog code in *avalon_aes_interface.sv* to fix bugs or add things after this, you need to use Qsys to regenerate the HDL so that those changes take effect. (The actual file being compiled by Quartus is the one generated by Qsys located in lab9\lab9_soc\synthesis\submodules\avalon_aes_interface.sv)

IMPORTANT: There is a bug in older versions of Quartus including 15.0 that causes it to misname your new Qsys component. If Qsys' generated top level Verilog file instantiates "new_component" for your avalon_aes_interface module, close Qsys, go to your project folder and edit the file "AES_Decryption_Core_hw.tcl": under "# file sets" section, replace them with the lines below. Then, open Qsys and update the version of your component (right click it under IP Catalog > Edit... and change version from 1.0 to 1.1 or any bigger number, then click Finish... to save), save your Qsys system and regenerate your HDL. Alternatively, you can correct the name in the verilog file from "new_component" to "avalon_aes_interface" each time after you generate HDL in Qsys.

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


add_fileset SIM_VERILOG SIM_VERILOG "" ""
set_fileset_property SIM_VERILOG TOP_LEVEL avalon_aes_interface
set_fileset_property SIM_VERILOG ENABLE_RELATIVE_INCLUDE_PATHS false
set_fileset_property SIM_VERILOG ENABLE_FILE_OVERWRITE_MODE false
add_fileset_file avalon_aes_interface.sv SYSTEM_VERILOG PATH
avalon_aes_interface.sv
```

Normally, we would use [Analyze Synthesis Files] in Component Editor to automatically match Avalon signal names and generate the correct script however that tool also unreliable and sometimes outputs “no modules found when analyzing null” on valid SystemVerilog files. This issue hasn’t been resolved yet as of Quartus 17 but you can still try using it.

Nios II Software/Hardware Communication

Refer to how we defined the parameters for the Avalon-MM slave "AES_Slave" in step 12. To access its registers in a C program, we declare a pointer to the slave's base address.

```
// Pointer to base address of AES module, make sure it matches Qsys
volatile unsigned int * AES_PTR = (unsigned int *) 0x00000100;
```

Since the Nios II/e processor used in this course has no Cache or MMU, all addresses are physical addresses and accessing the registers in your module is as simple as dereferencing AES_PTR pointer, for example:

```
// Send the 128-bit Key (Split into 4x 32-bit)
AES_PTR[0] = key[0]; // Write key[0] to Address 0x00000100
AES_PTR[1] = key[1]; // Write key[1] to Address 0x00000104
AES_PTR[2] = key[2]; // Write key[2] to Address 0x00000108
AES_PTR[3] = key[3]; // Wirte key[3] to Address 0x0000010C
```

Note that each array index corresponds to an address offset of 4 times the index because we specifically declared AES_PTR as an *unsigned int* which is 4 bytes wide to match the register width in the AES module.

To test that communication with your registers is working properly, you can write a value to one of the registers and read it back to check whether the value matches.

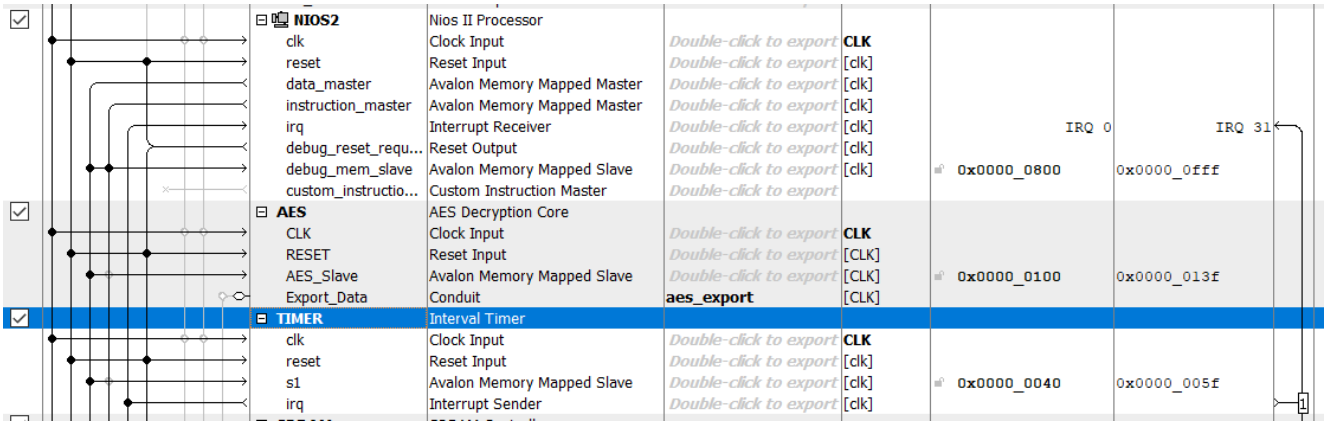
```
AES_PTR[10] = 0xDEADBEEF;
if (AES_PTR[10] != 0xDEADBEEF)
      printf("Error !");
```

For week 2, look at the state machine skeleton on page 4 to determine how you should control your hardware decryption module. Note that after sending the START signal (writing 1 to AES_PTR[14]), you should continuously read the DONE signal (AES_PTR[15]) until it becomes 1 which indicates that the decrypted message is ready, and finally, set the START signal to 0 to allow the state machine to return to WAIT state.

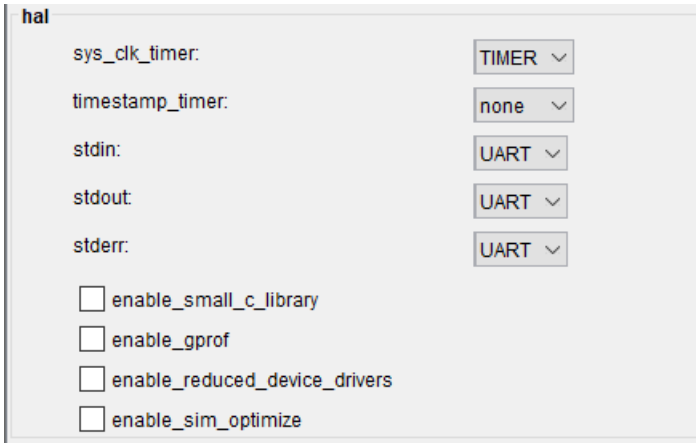
Nios II Benchmarking

When you are done testing the functionality of your code and hardware you can run your program in benchmark mode to get encryption/decryption speed measurements. However, Nios II itself does not have reliable means to measure time so we need to add an Interval Timer in Qsys that sends periodic interrupts.

1. Open your Qsys Project and add an Interval Timer. It can be found under "Processors and Peripherals" > "Peripherals" > Interval Timer. The default setting of 1ms intervals will suffice for our purposes.



2. Make the appropriate connections for each port, they should be familiar to you by now. Be sure to have the Interrupt Sender connected to Nios on the right side (in the example above, it is IRQ 1). As usual, you need to regenerate HDL and recompile your Quartus project.
3. In your Eclipse project, open BSP editor and choose your newly added interval timer for sys_clk_timer. This will make *time.h* function properly.



4. Generate BSP and recompile your Eclipse project before running.