Creating a custom IP in Vivado
Interfacing a VHDL IP with an AXI interface

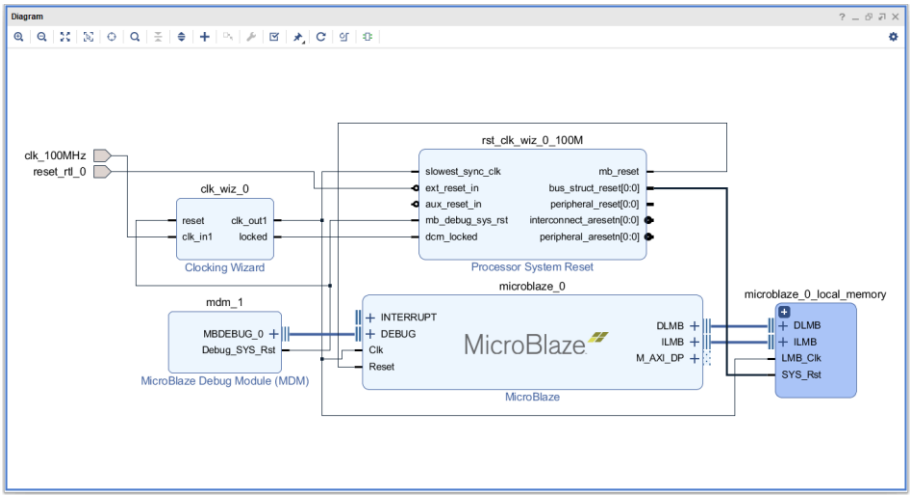
In this tutorial we'll create a custom AXI IP block in Vivado and modify its functionality by integrating custom VHDL code. The screenshots show a project with a Microblaze microprocessor core, but it can be applied to any SoC project, including projects with Zynq7. For simplicity, our custom IP will be a multiplier which our processor will be able to access through register reads and writes over an AXI bus.

The VHDL multiplier takes in two 16-bit unsigned inputs and outputs one 32-bit unsigned output. Both inputs and outputs are linked to the "software world" through the AXI protocol and can therefore be used from SDK.

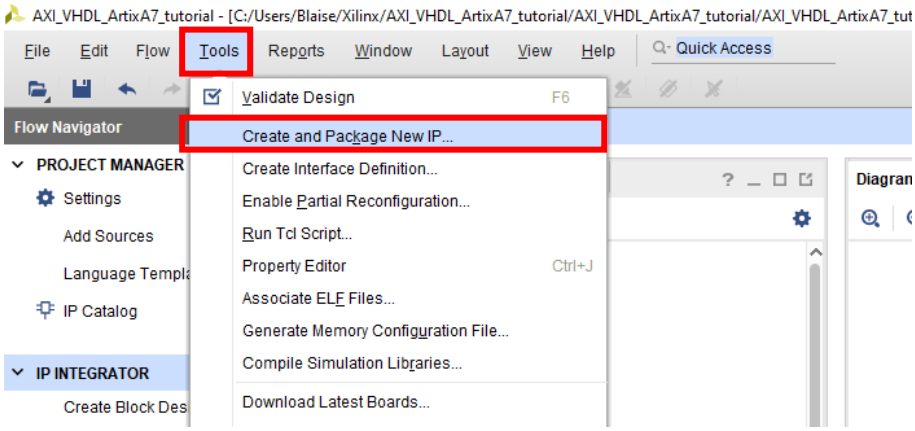
The design doesn't serve much purpose, but it is a good example of integrating your own code into an AXI IP block.

Create a custom IP

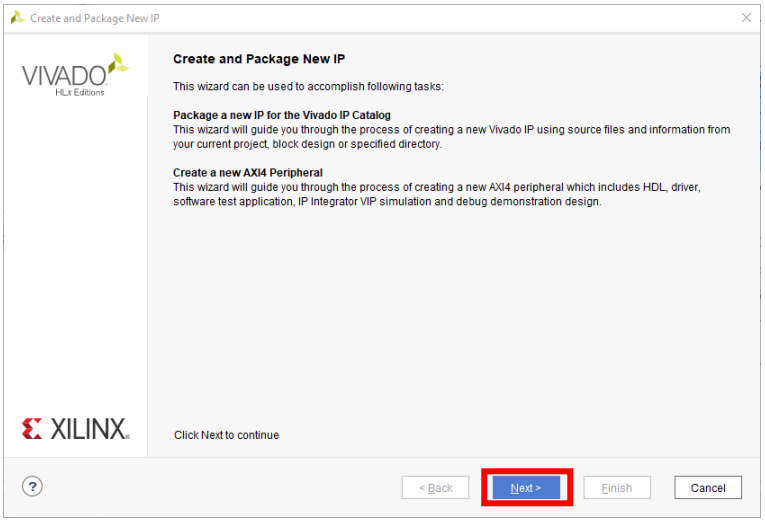
1. First, create a Vivado project. In Project Manager → Settings, set the target language as VHDL. Instantiate a clocking wizard IP and configure it for a 100MHz clock. Instantiate a Microblaze IP. Run block automation and set local memory to 32kb. Run connection automation.

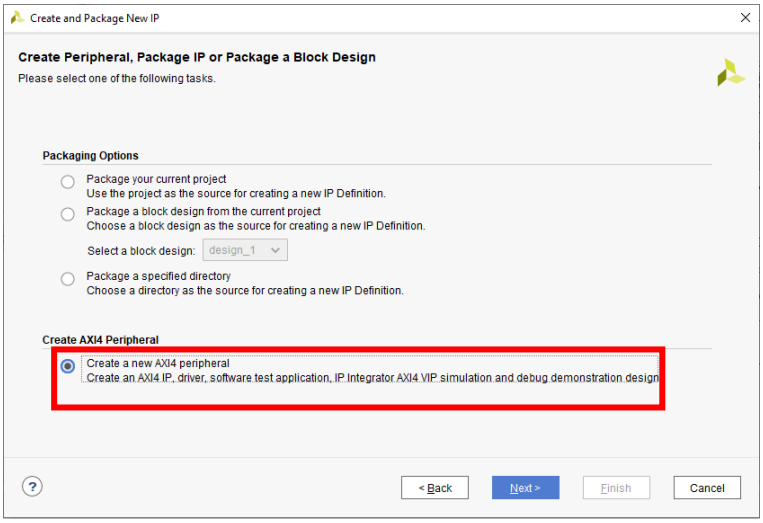


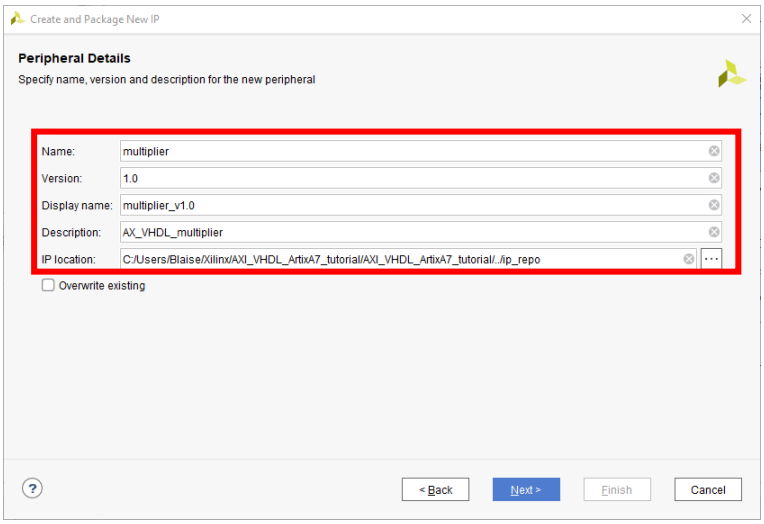
2. Select Tools-> Create and package IP.



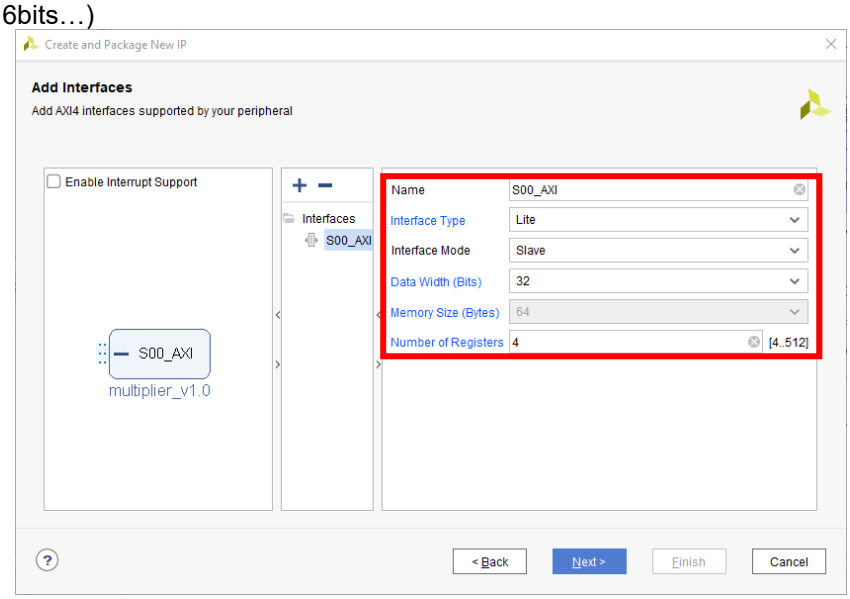
3. Click Next on the first screen, then select “Create a new AXI4 peripheral”. Give a name and a short description to your IP (optional – but it is a good practice)



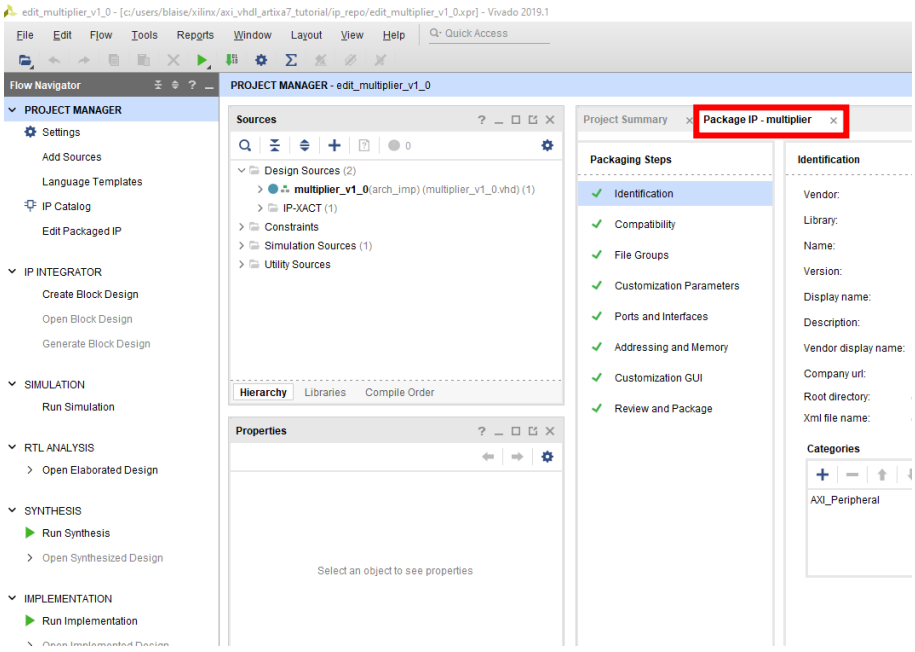




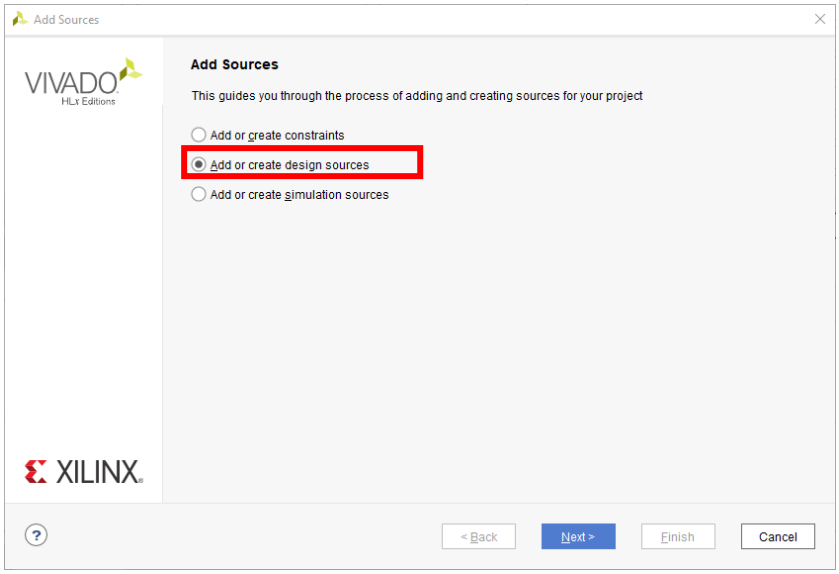
4. Leave the Interface Type unchanged (Lite by default). Choose the number of registers of your IP: these are the data that will be exchanged between the AXI interface and your VHDL code. For the multiplier, we want the AXI interface to send one of the operand values to the multiplier and the multiplier to send the multiplication result to the AXI interface. The number of registers needed is then 2 (but Vivado cannot instantiate less than 4). Note that you can use only one 32 bits AXI register to control several VHDL vectors (2 x 16 bits, 4 x 8 bits, 2 x 8 bits + 1 x 16bits…)

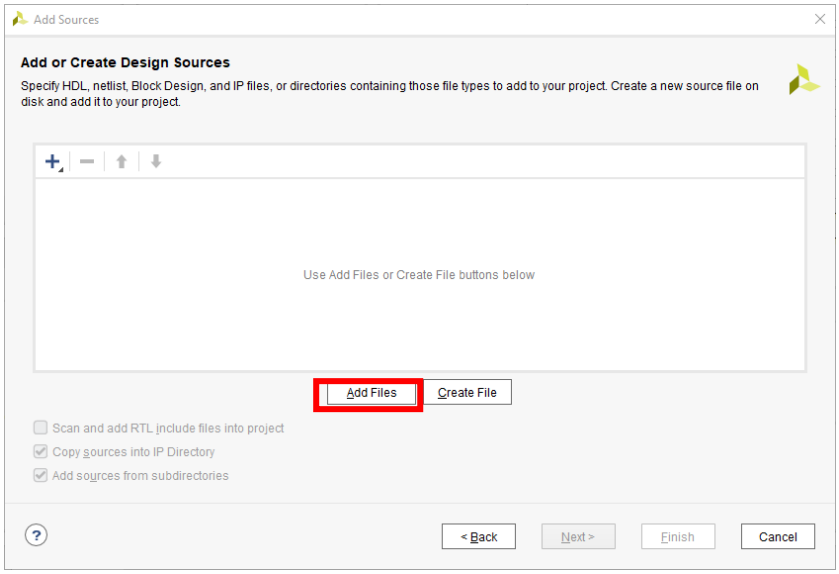


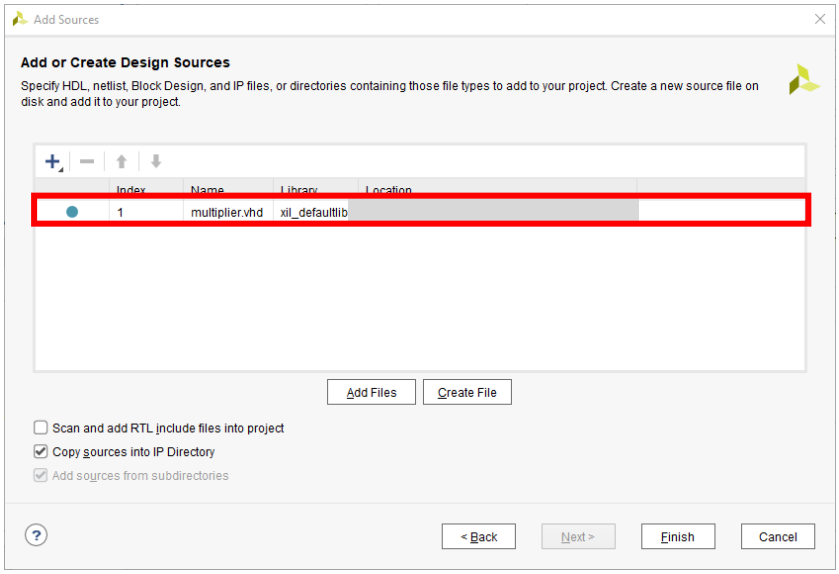
5. A new Vivado session automatically opens. Note on the screenshot below the Package IP – multiplier tab.



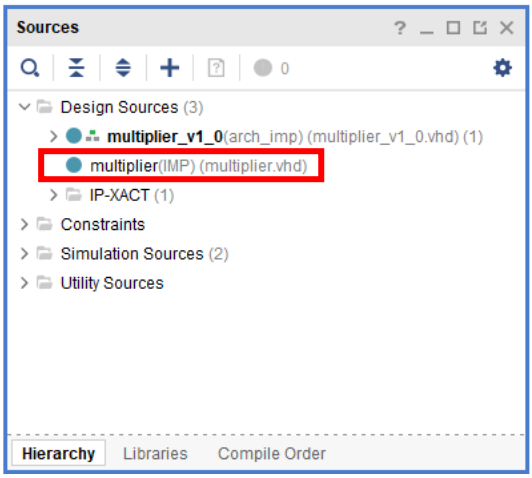
6. Add a VHDL design source. Here, you can use that multiplier.vhd file that you can find on Moodle.



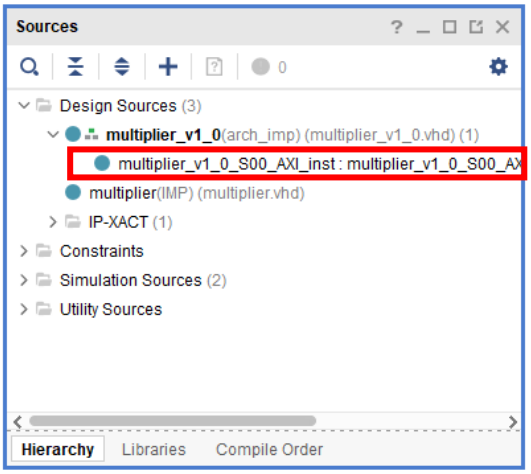




7. The VHDL file is now added to the sources of the IP. We now need to link it to the AXI interface.



Open the multiplier_v1_0_S00_AXI_inst source



8. First, add the external port of your IP in the port declaration section, below the line that says "-- Users to add ports here":

```
Mult_in : in std_logic_vector(15 downto 0);
```

9. Find the line with the "begin" keyword, and add the following code **above** it, to declare the multiplier component.

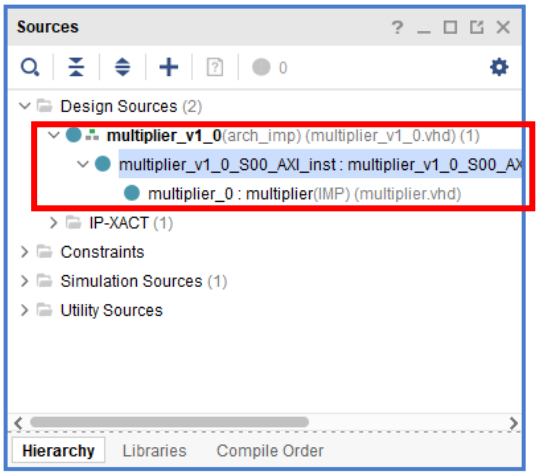
```
signal multiplier_out : std_logic_vector(31 downto 0);

component multiplier
port (
  clk: in std_logic;
  a: in std_logic_VECTOR(15 downto 0);
  b: in std_logic_VECTOR(15 downto 0);
  p: out std_logic_VECTOR(31 downto 0));
end component;
```

Signals are used when a register can be driven by the AXi interface (ALL registers can be driven by it) and by the VHDL function. Here, the output of the multiplier is driven by the multiplier itself and send to the AXI interface. Even it will not be driven by the AXI interface in the application, it is hardwired with such a capability. Therefore, a signal is used to avoid conflicts.

10. Now find the line that says “—Add user logic here” and add the following code below it to instantiate the multiplier:

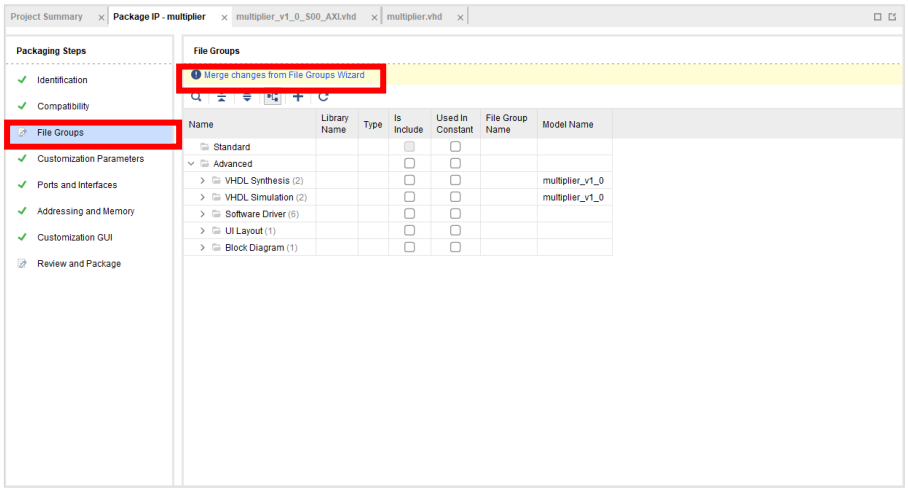
```
multiplier_0 : multiplier
port map (
  clk => S_AXI_ACLK,
  a => slv_reg0(31 downto 16),
  b => Mult_in,
  p => multiplier_out);
```

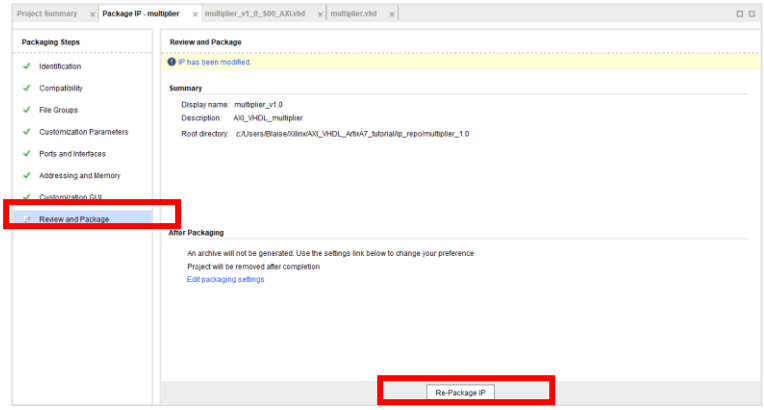
11. Find the line of code “reg_data_out <= slv_reg1;” and replace it with “reg_data_out<=multiplier_out;”. In the process statement just a few line above, replace “slv_reg1” with multiplier_out”. Save. You will notice that the multiplier.vhd file has been integratedinto the hierarchy.



12. Add the Mult_in” input port in all the component and port declarations throughout the whole IP hierarchy.

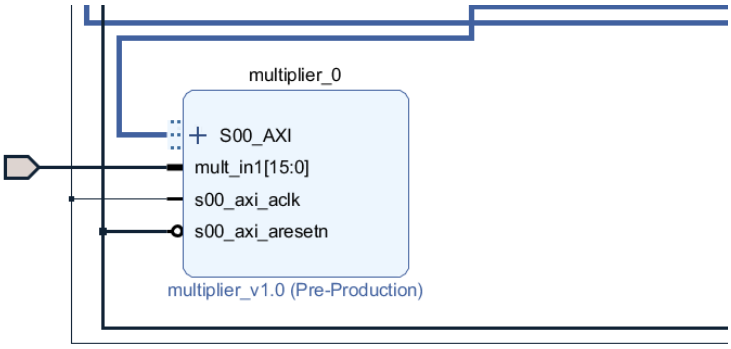
13. In the Package IP tab, in File group, click on Merge changes. In Review and Package, click on Re-package. Depending on your IP, other Packaging steps may have to be updated. Make sure all the steps are green before proceeding further.



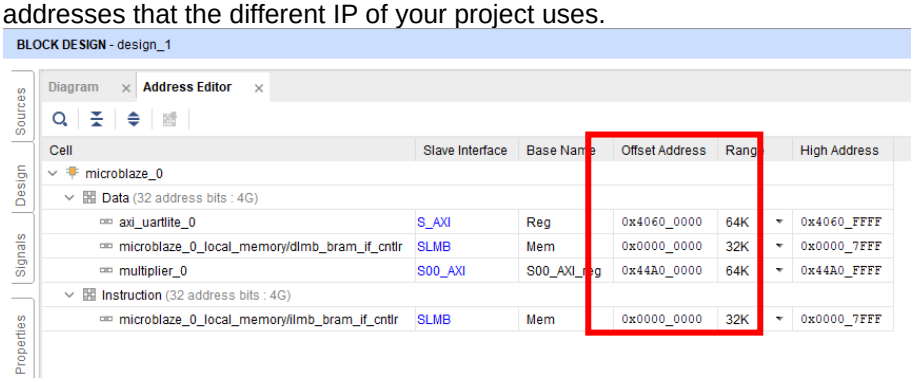


Add the IP to the design

14. Go back to your first session of Vivado. You may have to update your IP database.. Then you will be able to instantiate your IP, by browsing the regular IP library.



15. Generate the bitstream. Once it is done, check the address editor tab. It contains the registers addresses that the different IP of your project uses.

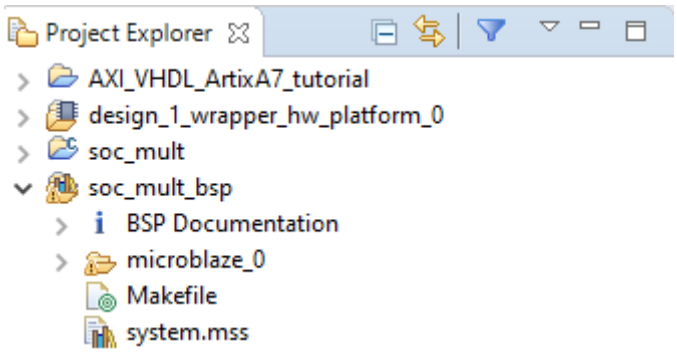


16.

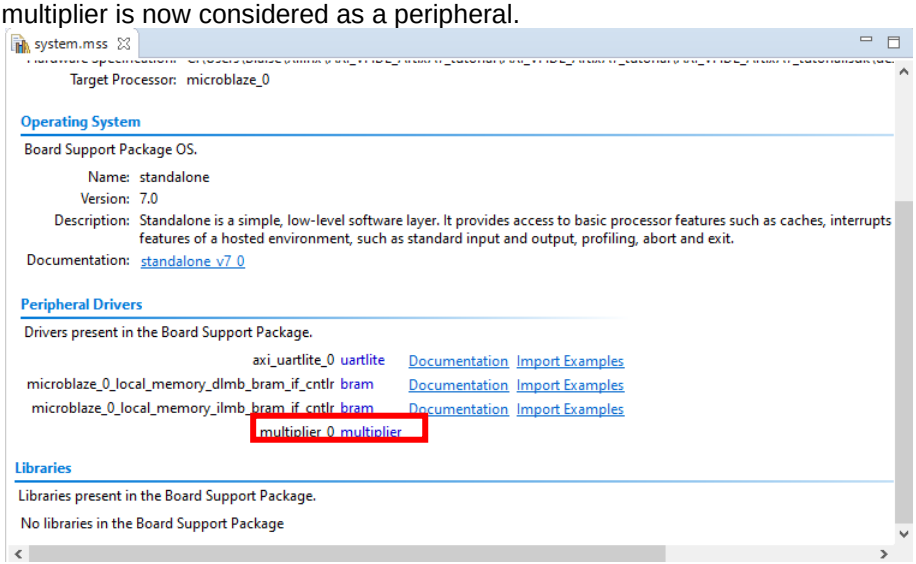
Using the registers in SDK

17. Once the Synthesis/Implementation/Bitstream generation steps are done, you can export your hardware, launch SDK, and create an C application.

18. Open the system.mss file inside the “project_name_bsp” hierarchy.



The multiplier is now considered as a peripheral.



Using the addresses given in Vivado, you are now able to read/write in the registers using standard C functions.