Creating and interfacing custom IP controller to Zynq PS

October 23, 2013

Many a times, it is required to interface custom IP cores to Zynq PS since the standard IP cores available with EDK do not satisfy all the user requirements. For example in the previous class we saw a custom audio controller IP and an OLED controller IP required for interfacing a microphone and an OLED display to Zynq. Xilinx XPS provides a template to generate custom IP cores. The template includes a bus interface logic to connect to external interface and a user logic interface to connect to user developed logic. The custom IP can have different bus standards to interface with external world. It can be

- AXI-Lite
- AXI4
- AXI-stream

AXI-Lite is a simple address-data interface with low throughput. AXI4 is a high-performance burst capable interface and AXI-stream is a data streaming interface (no address). Zynq PS can be directly interfaced to AXI-Lite or AXI4 interface based IP cores. AXI-stream peripherals cannot be directly interfaced to PS, but can be connected through bridges which convert AXI-Lite/AXI4 to AXI-stream. Follow the steps below to generate a custom IP in XPS.

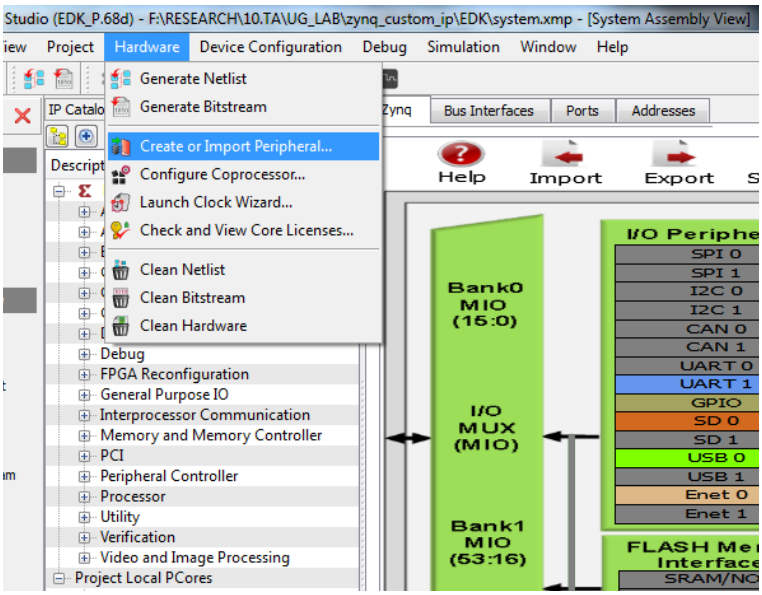


Figure 1: From the Hardware menu, select Create or Import Peripheral and click next

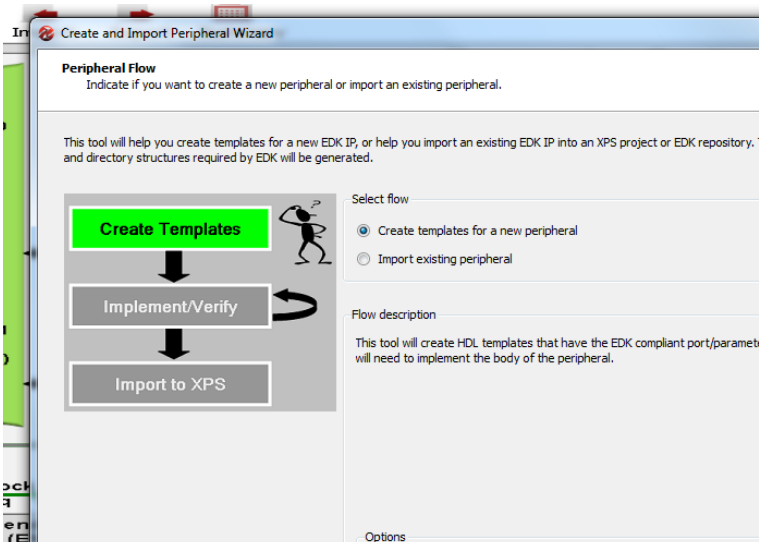


Figure 2: Select create template for new peripheral and click next

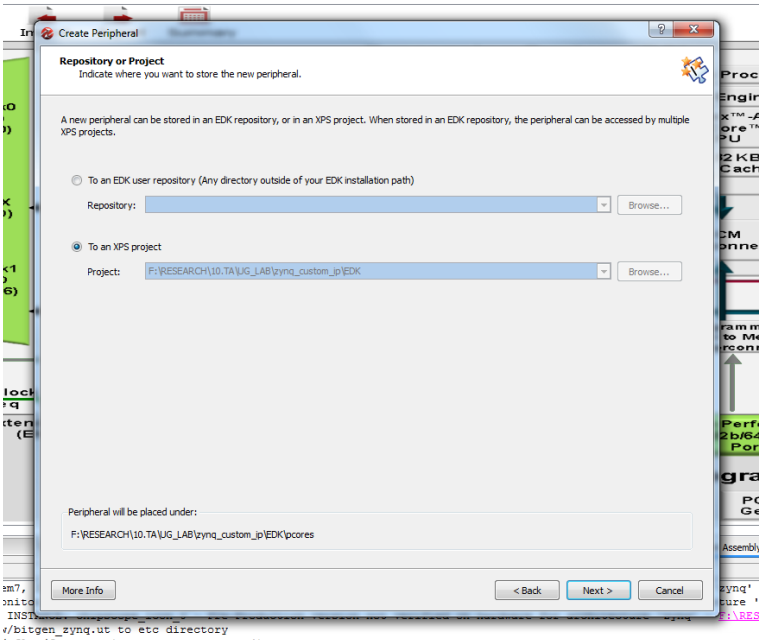


Figure 3: Select to an XPS project and click next

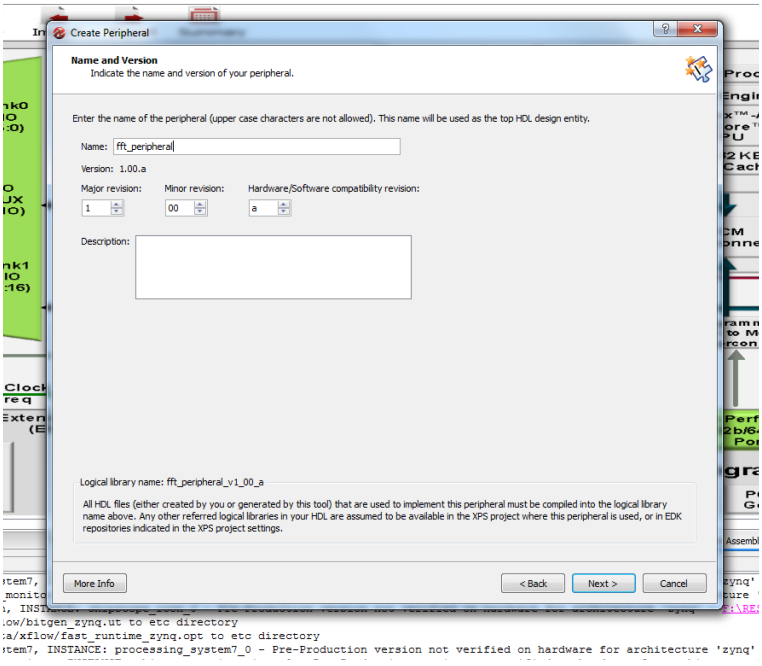


Figure 4: Give a name to your peripheral and click next

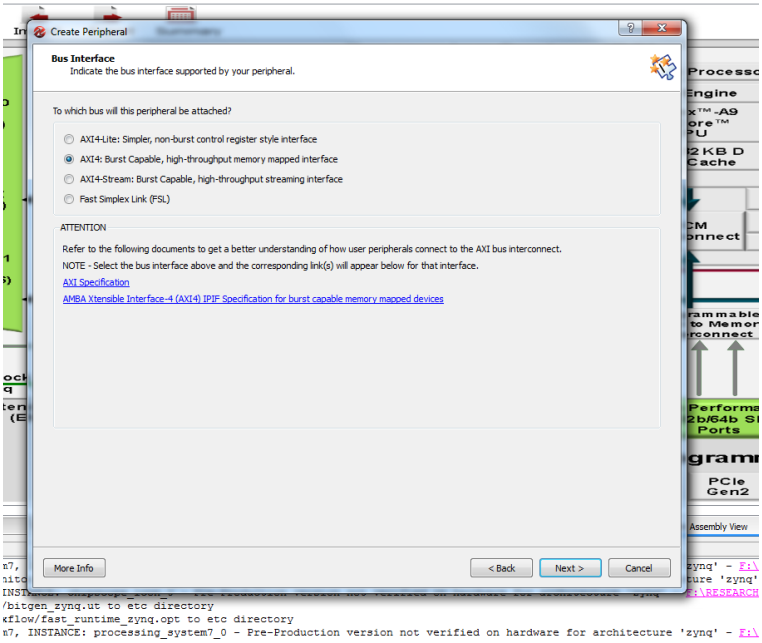


Figure 5: Select the bus standard for the peripheral. For the current peripheral we are going to use a high throughput AXI4 interface

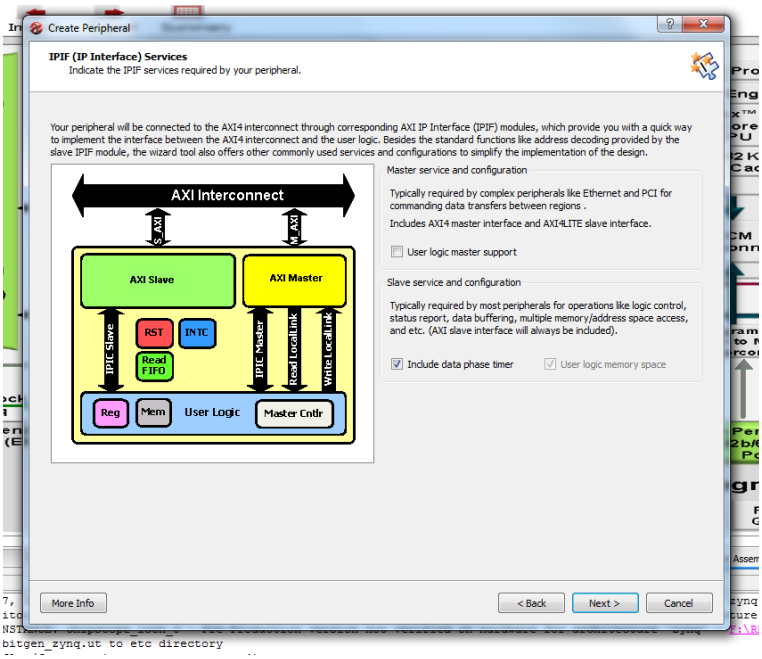


Figure 6: This window shows the architecture of the peripheral. Since our IP always acts as a slave to the processor, keep the *User logic master support* unchecked and click next

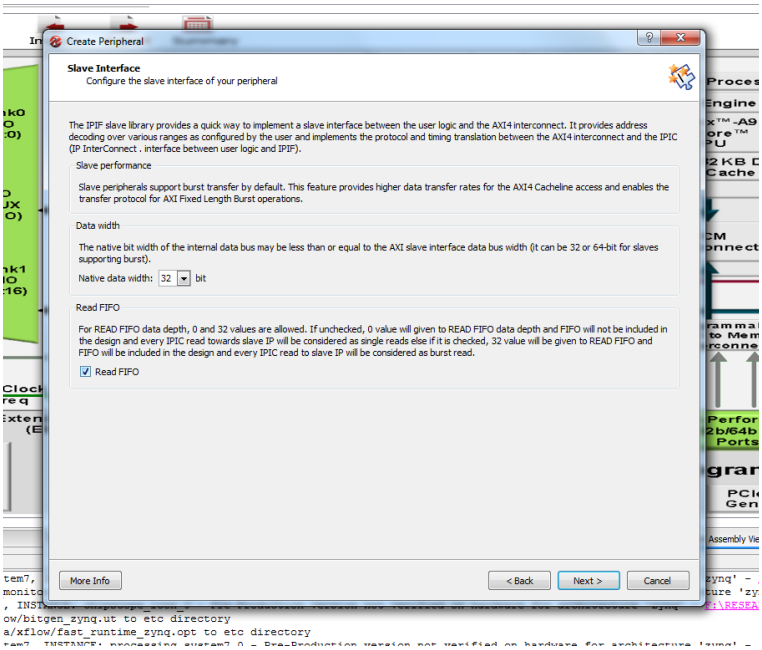


Figure 7: Check the *Read FIFO* option so that a FIFO is instantiated with in the peripheral for high performance. Keep the data width as 32 since that is the AXI data width. and click next

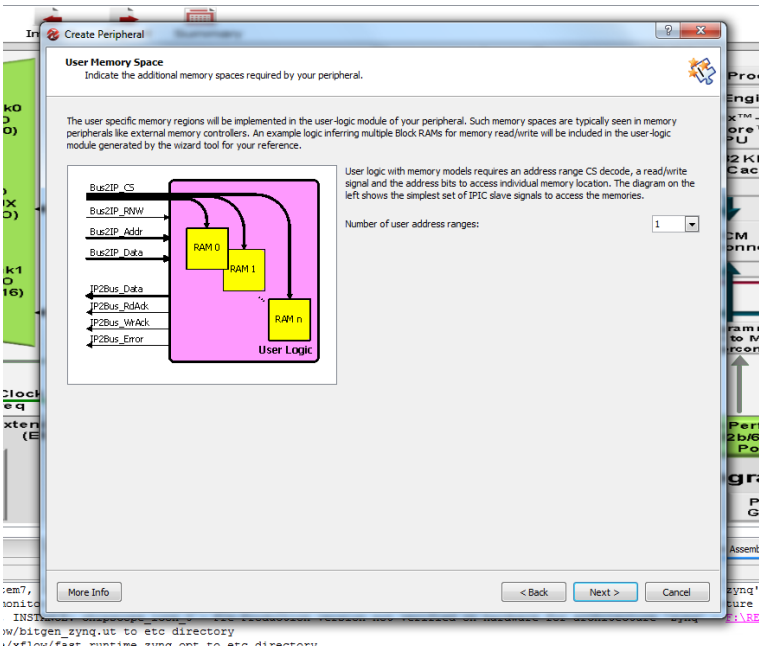


Figure 8: The peripheral is seen by the processor as a memory mapped region. We can optionally partition the peripheral memory map to several regions based on address range. For the present case, we use a single address range. Simply click next.

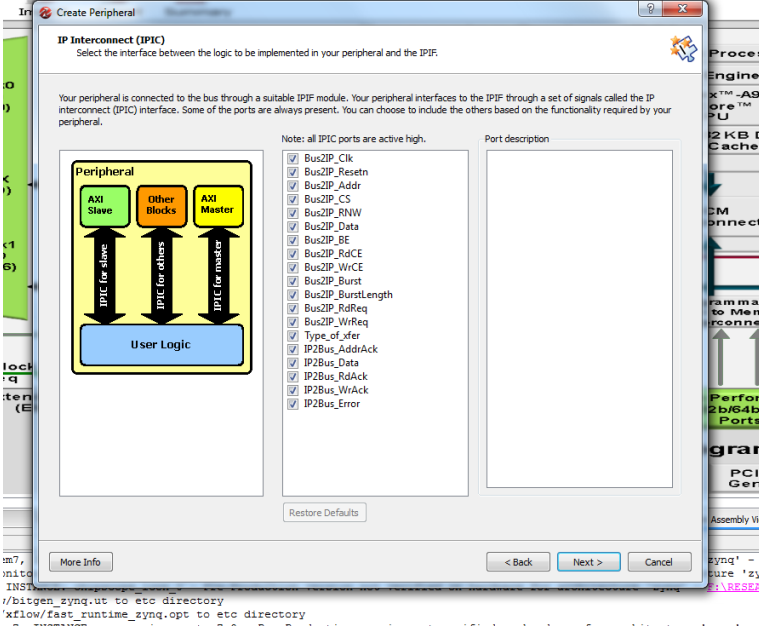


Figure 9: This window lists all the AXI4 bus signals, click next

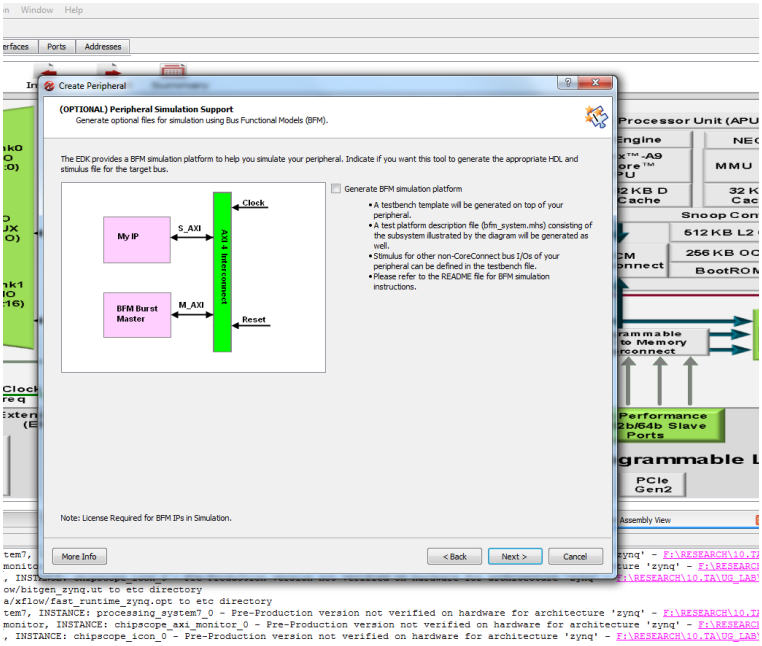


Figure 10: We are not going to do simulation of the IP, click next

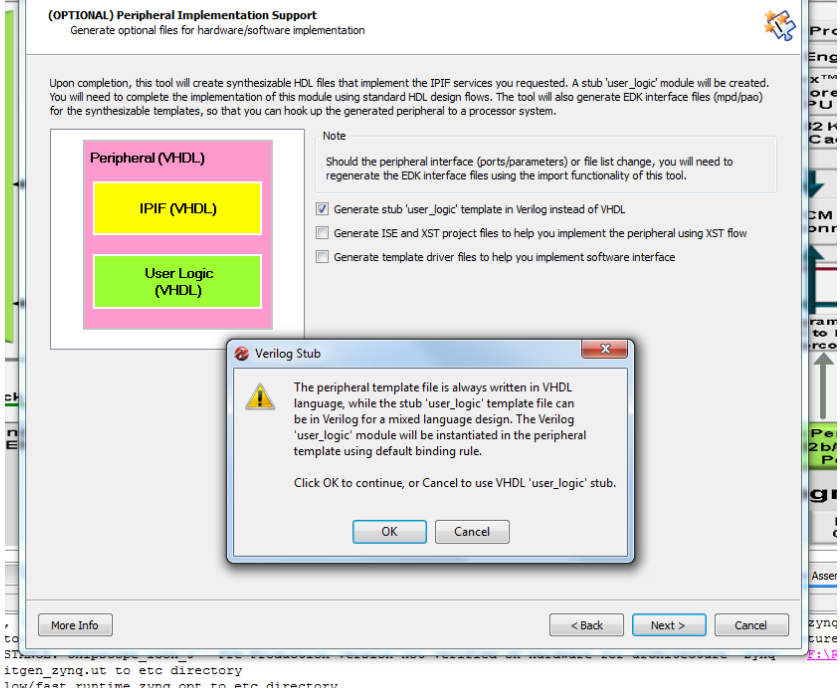


Figure 11: In this window select the *Generate stub user logic in verilog*option, so that we can work on verilog. The top file of the IP is always generated in VHDL. But we don't have to touch it. Click next

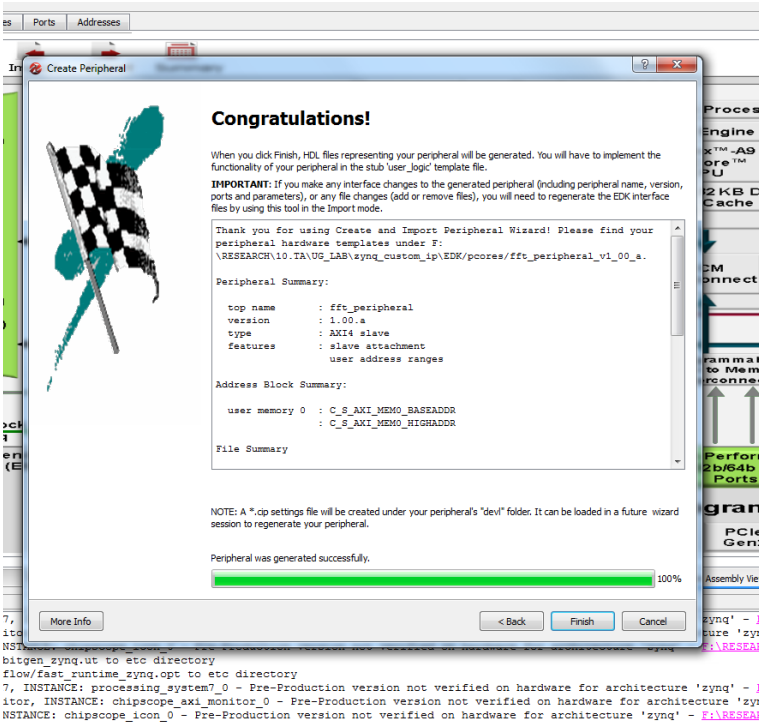


Figure 12: Click finish and the IP is generated.

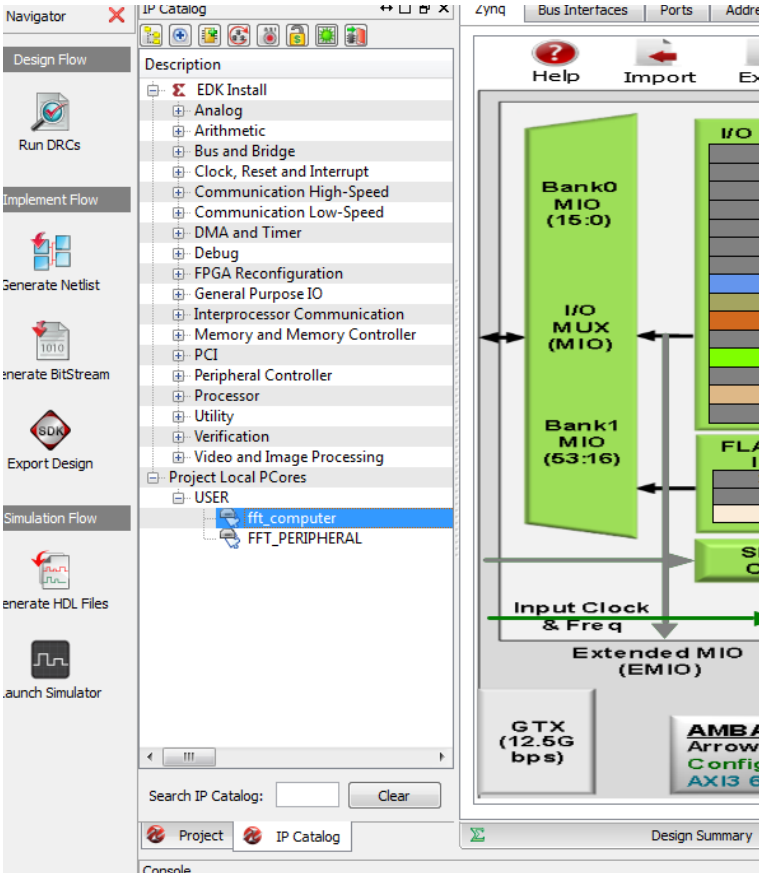


Figure 13: You can see the IP generated listed under Project local Pcores → User in the IP catalogue pane. If not select Project menu and then Rescan User repositories.

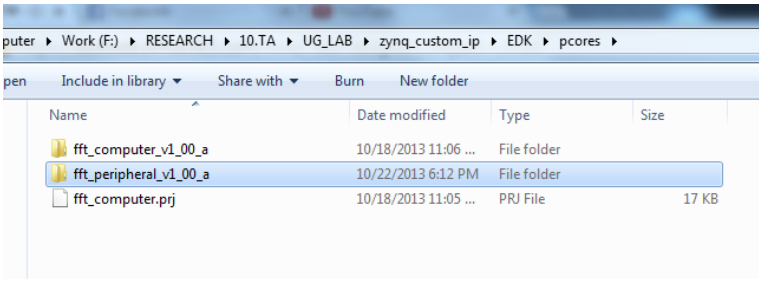


Figure 14: All the files corresponding to the IP are generated inside the pcores directory of the current EDK project directory

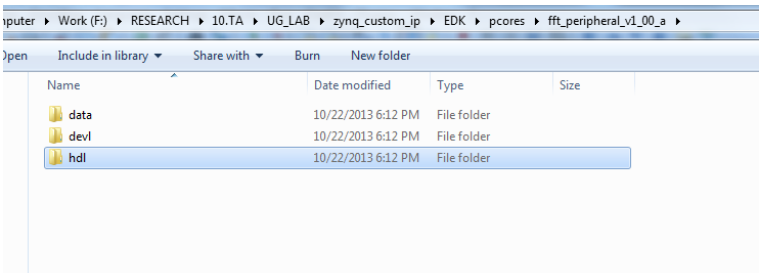


Figure 15: The IP core directory contains three folders namely data,devl,and hdl. The hdl folder contains all the hardware code. Go and open it.

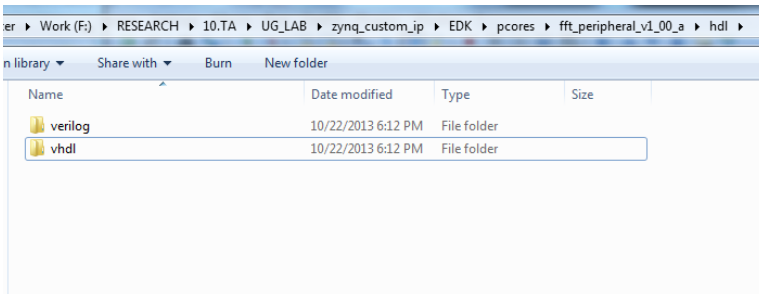
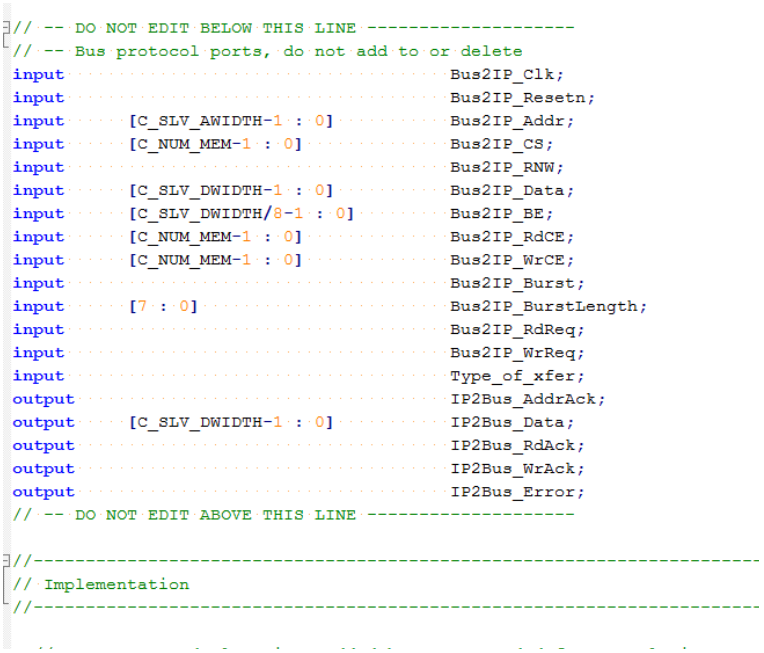


Figure 16: The vhdl folder contains the top file (bus interface and stuff) for the IP and the verilog folder contains the user logic. The user logic is automatically instantiated within the top file. Open the verilog folder and the user_logic.v file



```
// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol ports, do not add to or delete
input                                     Bus2IP_Clk;
input                                     Bus2IP_Resetn;
input      [C_SLV_AWIDTH-1 : 0]           Bus2IP_Addr;
input      [C_NUM_MEM-1 : 0]              Bus2IP_CS;
input                                     Bus2IP_RNW;
input      [C_SLV_DWIDTH-1 : 0]           Bus2IP_Data;
input      [C_SLV_DWIDTH/8-1 : 0]         Bus2IP_BE;
input      [C_NUM_MEM-1 : 0]              Bus2IP_RdCE;
input      [C_NUM_MEM-1 : 0]              Bus2IP_WrCE;
input                                     Bus2IP_Burst;
input      [7 : 0]                        Bus2IP_BurstLength;
input                                     Bus2IP_RdReq;
input                                     Bus2IP_WrReq;
input                                     Type_of_xfer;
output                                    IP2Bus_AddrAck;
output     [C_SLV_DWIDTH-1 : 0]           IP2Bus_Data;
output                                    IP2Bus_RdAck;
output                                    IP2Bus_WrAck;
output                                    IP2Bus_Error;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

//------------------------------------------------------------------------
// Implementation
//------------------------------------------------------------------------
```

Figure 17: There is an example implementation for reading and writing to a memory. We are interested only on the interface signals. All of them are not relevant to us. We just want to know what are the signalling for reading and writing.

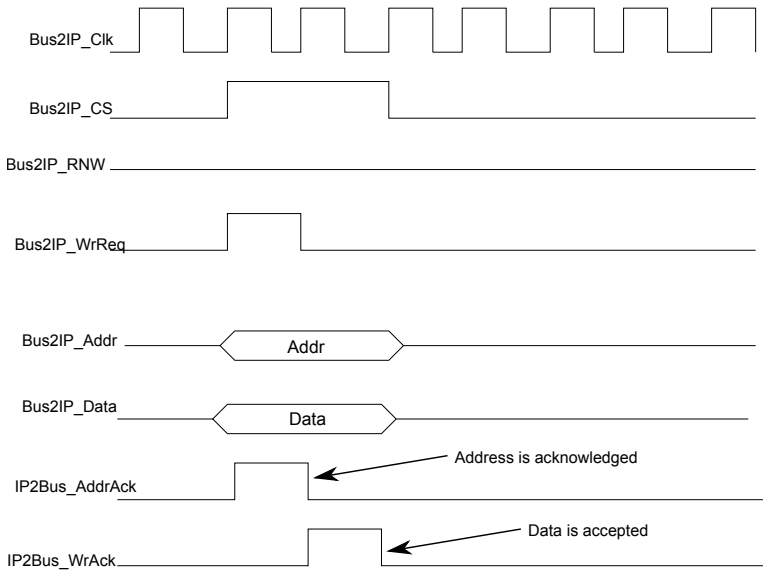


Figure 18: This is an example write case, where data is being written into the user logic (by the processor). The *CS* signal remains high until the user logic acknowledges the data with *IP2Bus_WrAck*. Similarly *Bus2IP_RNW* (read not write) remains low throughput the operation. *Bus2IP_WrReq* reamins high only for one clock cycle. The user logic should acknowledge the input address as well as the data. These two can happen on the same clock or on different clocks. If acknowledges on different clock cycles, address acknowledge should precede write acknowledge. From the diagram any easy way to generate address acknowledge if we don't care what is the target address?

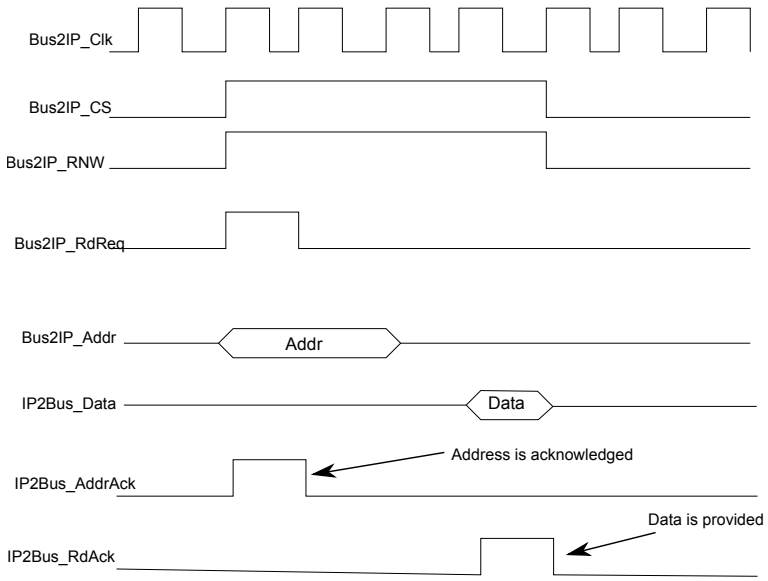


Figure 19: This is the read case. Similar to write case but *Bus2IP_RNW* remains high and the user logic should provide the output data along with read acknowledge

1 Instantiating other modules within the user logic

Suppose we need to instantiate an already built module into the user logic such as a FIFO or a filter. If the interface of this module is different from what is provided in the user logic, how this could be done? This is exactly what we are going to face. The FFT core that we will instantiate with in the user logic is having AXI-stream interface. Now some how we need to map this stream interface to the user logic interface described before. The general AXI-stream interface signals are given below.

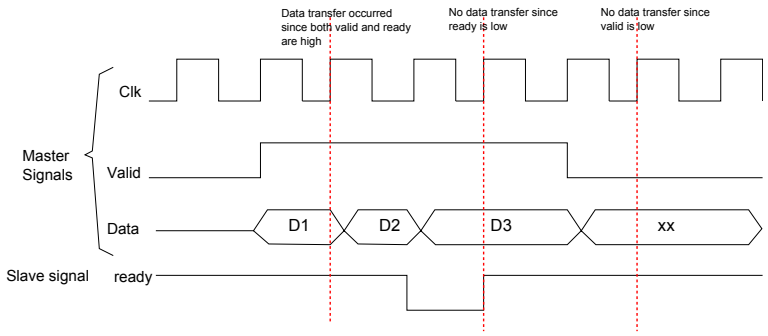


Figure 20: AXI stream signalling

In AXI-stream, whichever device is sending data is called a master device and whichever device receives data is called a slave device. So usually each device will have a master interface as well as a slave interface. From the diagram, it could be seen that there is no address for this interface and data is continuously streamed. Whenever a device wants to send data, it asserts valid high and places the data on the data bus. Whenever a device is ready to accept data, it asserts ready high. A data transaction happens when both valid and ready are asserted. A master device should hold the data on the bus until the slave device asserts ready signal. Now just think how this interface can be mapped to the user logic interface for the custom IP??

2 Importing modules generated in ISE to EDK

ISE design environment can be used to generate the logic, which will be eventually instantiated in the user_logic.v file. An example design is provided as shown in the figure

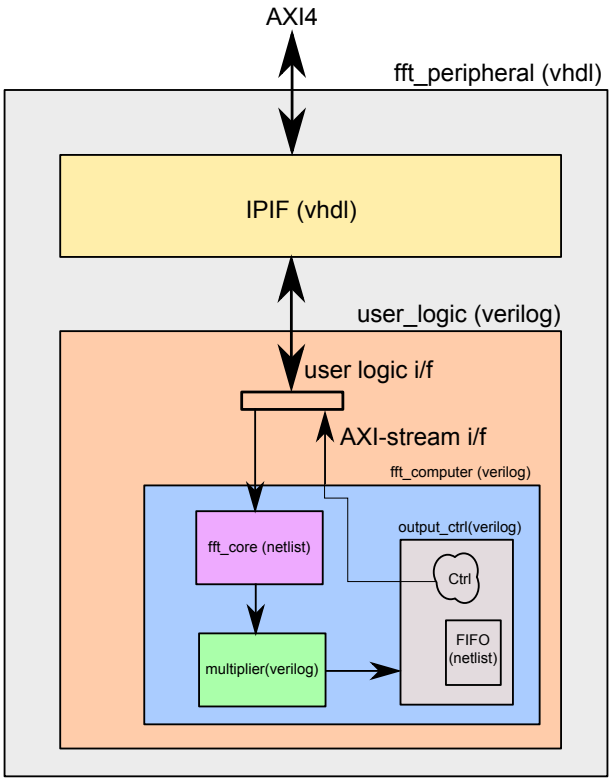


Figure 21: Example custom peripheral

Here all the logic inside the *fft_computer* are generated using ISE. The fft_core and the output_fifo are generated using Xilinx coregen. ISE stores the coregen output files (.v files and .ngc files) in the ipcore_dir folder. Other verilog are hand-written. Once required logic is written, simulate it to validate the functionality. Go through the example codes provided to better understand interfacing AXI-stream based modules.

Now the ISE generated files have to be imported to XPS. For this, follow the steps below.

1. copy all the verilog source files including the ones generated by coregen to the verilog folder within the XPS custom IP core directory (inside pcore/custom_ip/hdl).

2. add the names of the verilog files to the *.pao* file inside the pcores/custom_ip/data folder

3. copy all the netlist files (.ngc files) generated by coregen to the *implementation* folder in the XPS project

4. instantiate the top module generated in ISE inside the *user_logic.v* files and manage the bus signalling

readme.txt | fft_peripheral_v2_1_0.pao

```
1  ##############################################################################
2  ## Filename:               C:\Users\vipin2\Desktop\zynq_custom_ip_lab\microblaze_platfo
3  ## Description:            Peripheral Analysis Order
4  ## Date:                   Tue Oct 22 13:22:11 2013 (by Create and Import Peripheral Wi
5  ##############################################################################
6
7  lib proc_common_v3_00_a all vhdl
8  lib axi_slave_burst_v1_00_a all vhdl
9  lib fft_peripheral_v1_00_a user_logic verilog
10 lib fft_peripheral_v1_00_a fft_computer verilog
11 lib fft_peripheral_v1_00_a fft_core verilog
12 lib fft_peripheral_v1_00_a multiplier verilog
13 lib fft_peripheral_v1_00_a output_ctrl verilog
14 lib fft_peripheral_v1_00_a output_fifo verilog
15 lib fft_peripheral_v1_00_a fft_peripheral vhdl
16
```

Figure 22: pao file after editing for the provided example design

```
//------------------------------------------------------------------------

wire valid_write_data;
wire valid_read_req;
wire [31:0] m_axis_data_tdata;
wire m_axis_data_tvalid;

assign valid_write_data = Bus2IP_CS & ~Bus2IP_RNW;
assign valid_read_req   = Bus2IP_CS & Bus2IP_RdCE;
assign IP2Bus_AddrAck   = Bus2IP_RdReq|Bus2IP_WrReq; //When ever a read or write request comes, ack the address
assign IP2Bus_Error     = 1'b0;                      //Always deassert the error signal

//The Read ack signal generation. This signal needs to be asserted only after acking the address. Hence one level pipelining is required
always @(posedge Bus2IP_Clk)
begin
    if(valid_read_req & m_axis_data_tvalid & ~IP2Bus_RdAck)
    begin
        IP2Bus_Data   <= m_axis_data_tdata;
        IP2Bus_RdAck  <= 1'b1;
    end
    else
        IP2Bus_RdAck  <= 1'b0;
end


// Instantiate the top module generated in ISE here and connect the signals to the user logic interface
fft_computer fft_comp (
    .i_clk(Bus2IP_Clk),
    .i_rst_n(Bus2IP_Resetn),
    .i_data_valid(valid_write_data),
    .i_data(Bus2IP_Data),
    .o_data_ready(IP2Bus_WrAck),
    .o_data_valid(m_axis_data_tvalid),
    .o_data(m_axis_data_tdata),
    .i_data_ready(valid_read_req & ~IP2Bus_RdAck)
    );
```

Figure 23: Mappling of AXI-stream interface signals to the user logic interface

Once all these steps are done, add the custom ip to the system from the IP catalogue pane of the XPS GUI by double clicking the IP name. It can be then exported to SDK and bitstream can be generated.