**Instructions on getting designs with custom hard blocks through the Titan Flow**

**Introduction:**

**The titan flow (shown below in figure 1) allows the use of Quartus to synthesize HDL designs and then pass the synthesized netlist to VPR for place and route. With this flow we can take advantage of the verilog coverage of Quartus and its support for Intel IP cores. This allows designs to then be targeted on like devices.**

**A picture containing chart

Description automatically generated**

**Figure #1: A diagram of the Titan Flow (highlighted in red)**

**One problem with this flow is that custom or new types of hard blocks that are found on a non-Intel FPGA device cannot be included in HDL designs and passed through the Titan Flow. Non-Intel IP cores cannot be successfully synthesized within Quartus. The Quartus compiler will throw an error or synthesize an incorrect netlist. This document outlines a set of instructions that can be used to successfully pass custom hard blocks through the Titan Flow. This can be useful when testing FPGA devices with new/custom hard blocks.**

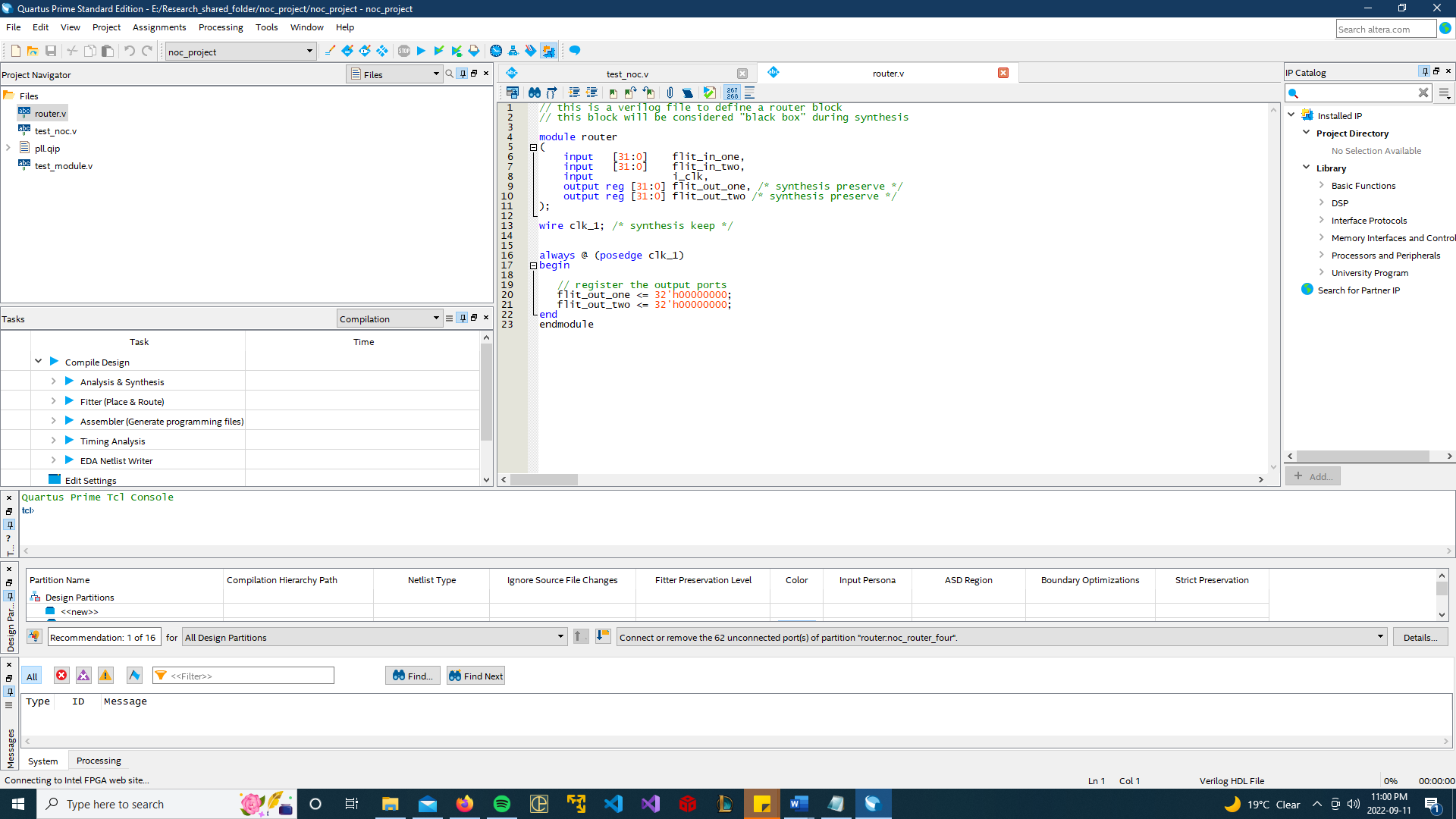
**The remainder of this document will outline the necessary steps to pass HDL designs with custom hard blocks through the Titan Flow. These instructions will go through an example design which includes a custom NoC router and take it through the entire Titan Flow.**

**Important Notes:**

* **This process has been tested on Quartus Prime version 20.1**
* **The latest version of the vqm2blif conversion tool used in this document can be found within the VTR repo (**[**https://github.com/verilog-to-routing/vtr-verilog-to-routing**](https://github.com/verilog-to-routing/vtr-verilog-to-routing)**)**
* **There is a pythin script called ‘titan\_flow.py’ which runs the entire titan flow. This process has not been tested with that script, so it is not recommended to be used.**
* **This document was last updated September 2022**
* **Please contact Srivatsan Srinivasan at** [**srivatsan.srinivasan@mail.utoronto.ca**](mailto:srivatsan.srinivasan@mail.utoronto.ca) **for additional information**

**Step #1: How to include custom hard blocks in the design**

All the custom hard blocks in the design need to be individually defined as modules. An example of this definition is shown in figure 2 below for a NoC router block.



**Figure #2: A NoC router block module definition. This example shows how custom hard blocks should be defined within a design**

The following steps should be followed when defining a custom hard block module:

* When defining the custom hard block module, start by including all its input and output ports
* Add the **preserve synthesis attribute** to all the output ports in the hard block module
* Within the module definition, create a new wire that will act as a clock signal and add the **keep preserve synthesis** attribute to this signal
* Within the module definition, create an always block that has the previously created signal inside its sensitivity list
* Inside the always block, register all the output ports by setting them equal to an arbitrary value (in the example above the output ports are set to 0)

Once the custom hard block module has been defined as shown above, it can be instantiated in the design as needed. Refer to figure 3 below which shows how the router mode in figure 2 above is instantiated within a design.

Graphical user interface, text, application

Description automatically generated

**Figure #3: A number of NoC router blocks being instantiated within a design.**

**Step #2: How to synthesize the design**

If the design was synthesized normally then the Quartus compiler would either fail or create an incorrect netlist. Before synthesis, all the custom hard blocks in the design need to be set to empty partitions. Doing this ensures that the Quartus compiler doesn’t fail, and this helps preserve all hard blocks within the netlist.

The titan flow has a script called ‘q2\_flow.tcl’ that automatically goes through the design and sets all the custom hard blocks to empty partitions. Follow the steps below on how to run this script:

1. Make sure Quartus is installed, and its install directory is added as an environment variable
2. Using the terminal, go to the directory where Quartus is installed
3. Run the following command:

|  |
| --- |
| quartus\_sh -t <path\_to\_script\_file>/q2\_flow.tcl -project <quartus\_project\_file\_name>.qpf -family <FPGA\_device\_name> -synth -vqm\_out\_file <vqm\_output\_file\_name>.vqm -partition\_hard\_blocks hard\_block\_one\_module\_name;hard\_block\_one\_module\_name |

The command above should partition all the custom hard blocks, then synthesize the project and finally generate a vqm netlist file. The option ‘parition\_hard\_blocks’ creates all the empty partitions, and it needs the module names of every unique custom hard block in the design. Other commands are available and can be found in the ‘q2\_flow.tcl’ file. Please note that by using the ‘parition\_hard\_blocks’ option, the auto partitioning options cannot be used.

Below is an example of the command used to generate a vqm file for the design example shown in figure 3. In this example the ‘q2\_flow.tcl’ script file was in the same directory as the project, and this was targeting the Stratix 4 device.

|  |
| --- |
| quartus\_sh -t q2\_flow.tcl -project noc\_project.qpf -family stratixiv -synth -vqm\_out\_file test\_noc.vqm -partition\_hard\_blocks router |

**Step #3: Netlist Conversion**

After the design has been synthesized by Quartus, the netlist needs to be converted to a blif format so it can then be passed to VPR. The is conversion is done by the vqm2blif program which is found within the VTR project. In addition to converting the netlist, this program also fixes the netlist errors which were created by partitioning all the custom hard blocks in the design. Setting the custom hard blocks to empty partitions in step 2 causes the Quartus compiler to create several LUTs and Flip Flops to represent the ports of the hard block within the netlist. So instead of a single hard block being present in the netlist, now all the ports of the hard block are found individually. To fix the netlist, the LUTs and Flip Flops that represent all the hard block ports need to removed and replaced by the hard block itself and this needs to be done for each and every custom hard block within the netlist.

The vqm2blif program can be used to fix the netlist when custom hard blocks are present. The command below can be used to generate the blif netlist for a design which has custom hard blocks:

|  |
| --- |
| vqm2blif -vqm <netlist\_to\_convert>.vqm -arch <FPGA\_architecture\_decsription\_the\_design\_is\_being\_targeted\_on>.xml -out <converted\_netlist>.blif -insert\_custom\_hard\_blocks custom\_hard\_block\_module\_name\_one custom\_block\_module\_name\_two |

The above command should convert a vqm netlist and fix up the netlist errors created by any custom hard blocks within the design. The option ‘insert\_custom\_hard\_blocks’ takes as an input the module names of all the custom hard blocks in the design and fixes the netlist. For this wo work the hard block needs to be described within the FPGA architecture description file.

Below is an example of the vqm2blif program being used to convert the vqm netlist file for the design shown in figure 3.

|  |
| --- |
| vqm2blif -vqm test\_noc.vqm -arch stratixiv\_arch.timing.xml -out test\_noc.blif -insert\_custom\_hard\_blocks router |

**Step #4: Place and Route**

Once the blif netlist file has been created, then VPR can be used normally to place and route the design. It is important to note that the FPGA architecture file given as input to VPR must also include the custom hard blocks in the design.