Partial Reconfiguration: A Simple Tutorial

Richard Neil Pittman

Microsoft Research

February 2012

Technical Report

MSR-TR 2012-19

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

Partial Reconfiguration: A Simple Tutorial

A Tutorial for XILINX FPGAs

Neil Pittman – 2/12, version 1.0

Introduction

Partial Reconfiguration is a feature of modern FPGAs that allows a subset of the logic fabric of a FPGA to dynamically reconfigure while the remaining logic continues to operate unperturbed. Xilinx has provided this feature in their high end FPGAs, the Virtex series, in limited access BETA since the late 1990s. More recently it is a production feature supported by their tools and across their devices since the release of ISE 12. The support for this feature continues to improve in the more recent release of ISE 13. Altera has promised this feature for their new high end devices, but this has not yet materialized.

Partial Reconfiguration of FPGAs is a compelling design concept for general purpose reconfigurable systems for its flexibility and extensibility. Despite the significant improvements in software tools and support, the Xilinx partial reconfiguration design option has a reputation for being an expert level flow that is difficult to use. In this tutorial we will show that it can actually be quite simple. As a case study, we apply Partial Reconfiguration to the Simple Interface for Reconfigurable Computing (SIRC) toolset. Combining SIRC and partial reconfiguration makes the idea of general purpose hardware and software user systems deployed on demand on generic platforms viable. The goal is to make developers more confident in the practicality of this concept and in their own ability to use it, so that more will take advantage of what it has to offer.

Prerequisites

Before beginning there are some prerequisites for successfully completing the tutorial.

A working knowledge of the Xilinx IDE is required to understand and successfully complete this tutorial. You must have at least version 12 of the Xilinx Integrated Development Environment (IDE), including ISE and PlanAhead. It is recommended that you have the latest version available. For this tutorial, all screenshots and procedures will be performed using version 13.2 of the IDE. In addition, the partial reconfiguration design flow requires an additional license to activate the feature in the tools. Please contact for Xilinx representative for assistance in acquiring such a license.[6]

This tutorial requires the latest version of the SIRC hardware/software API. For this tutorial, the hardware and software sources are derived from SIRC release 1.1.[1]

The partial reconfiguration feature as presented in this tutorial is supported the Xilinx Virtex series since the Virtex 4 and in the Spartan series since Spartan 6. All the Xilinx 7 series FPGAs support partial reconfiguration. In this tutorial, the system will be designed and implemented on a Virtex 6 LX 240t (xc6vlx240t-1ff1156) on a Xilinx ML605 evaluation board.[3]

Generalities

The regular synthesis flow generates a single bitstream for programming the FPGA. This considers the device as a single atomic entity. In contrast, the PR flow physically divides the device in regions. One region is called the "static region", which is the portion of the device that is programmed at startup and never changes. One region is the "dynamic region" aka "the PR region", which is the portion of the device that will be reconfigured dynamically, potentially multiple times and with different designs. It is possible to have multiple PR regions, but we will consider only the simplest case here.[4]

The PR flow generates at least two bitstreams, one for the static and one for the PR region. Most likely, there will be multiple PR bitstreams, one for each design that can be dynamically loaded. In our case, we place the basic SIRC functionality into the static region. The user circuits instead go into the PR region. Another practical example is an FPGA board where we place the PCIe interface logic in a static region of the FPGA and the user circuits in the PR region. This eliminates the need for a full-device reset on each separate application run, with potentially negative effects on the OS.

The PR region is a physical entity, with a given geometry. PlanAhead is the tool that allows you to define the exact location of the PR region on your target device. [4]

Building SIRC with Partial Reconfiguration

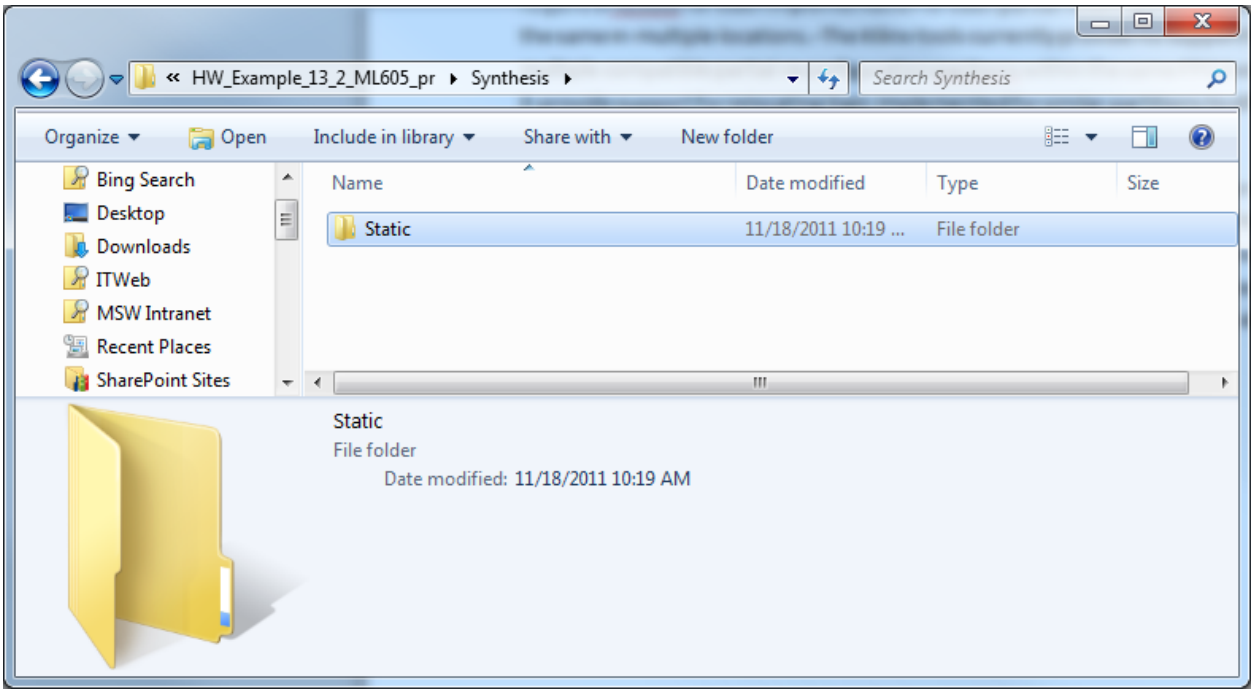
Synthesis

The first step in implementing a system using the Xilinx partial reconfiguration design flow is to synthesize the netlists from the HDL sources that will be used in the implementation process. The process requires separate netlists for the static (top level) design and for the partial reconfigurable partition(s). A netlist must be generated for each implementation of the partial reconfiguration partition used in the design. If the system design has multiple partial reconfiguration partitions, then it will require a netlists for each implementation of each partial reconfiguration partition, even if the logic is the same in multiple locations.

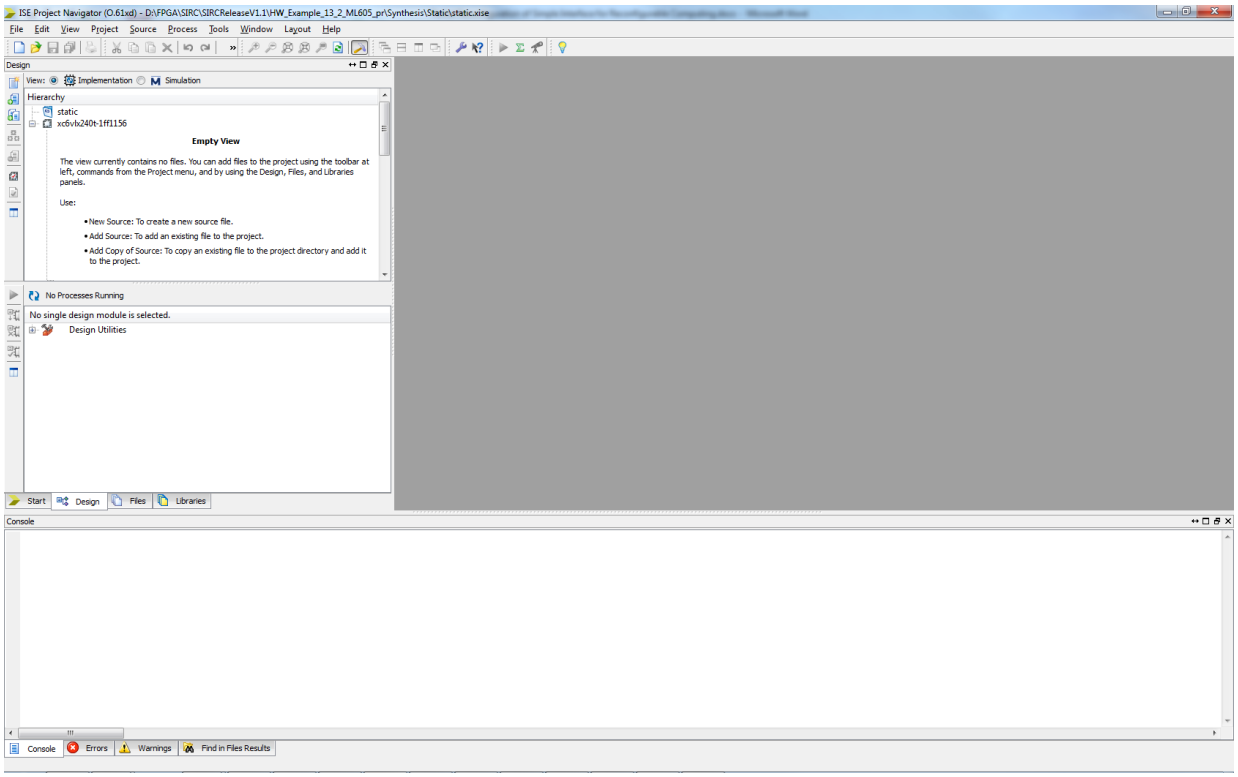
First, you should create a new directory to hold the synthesis files and name it appropriately. We will use 'Synthesis' in this tutorial. This folder will eventually contain several separate synthesis projects for each of the netlists that the design requires for implementation. Then proceed to next section to begin implementing the netlists for the SIRC example. The SIRC release like many other designs uses of some ip cores generated by CoreGen. These ip cores are included in the design as netlists rather than source. These will also be required in the implementation process.

Synthesizing the Static Region

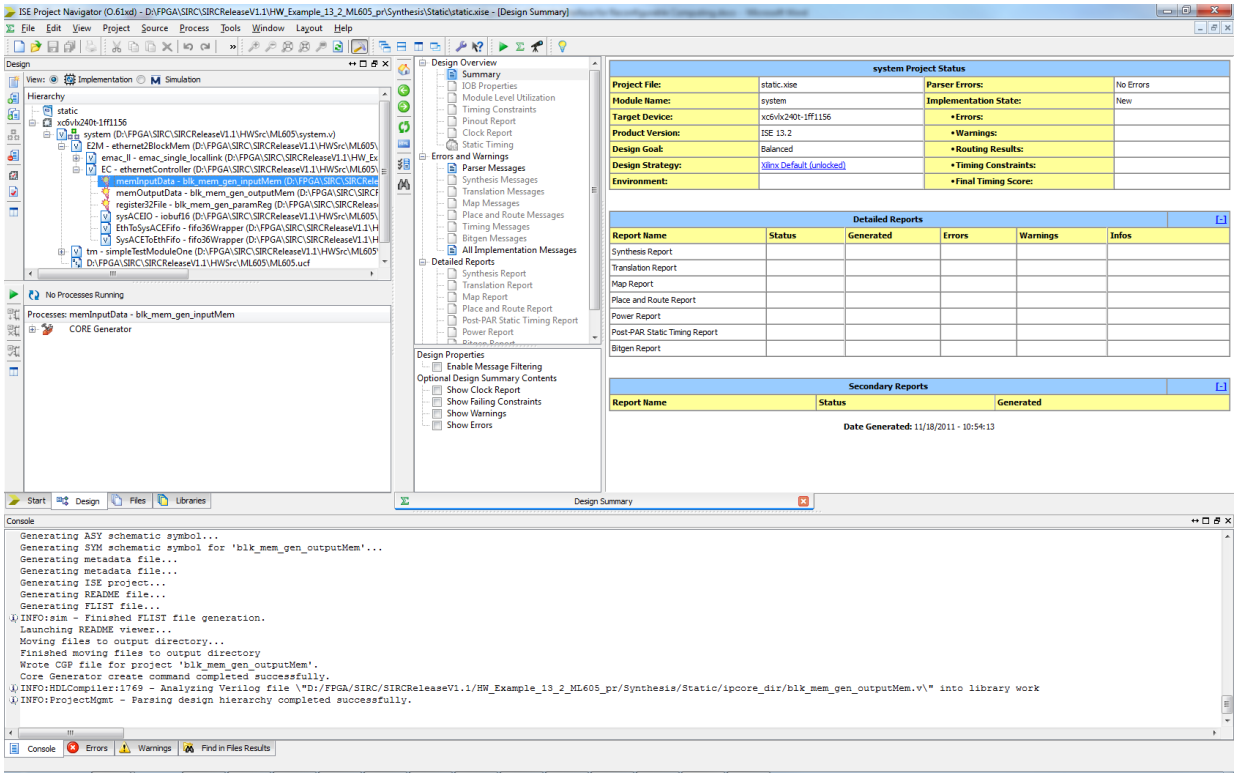
1. Create a new folder inside the Synthesis Folder and label it 'Static'



2. Inside that Folder, create a new ISE project.

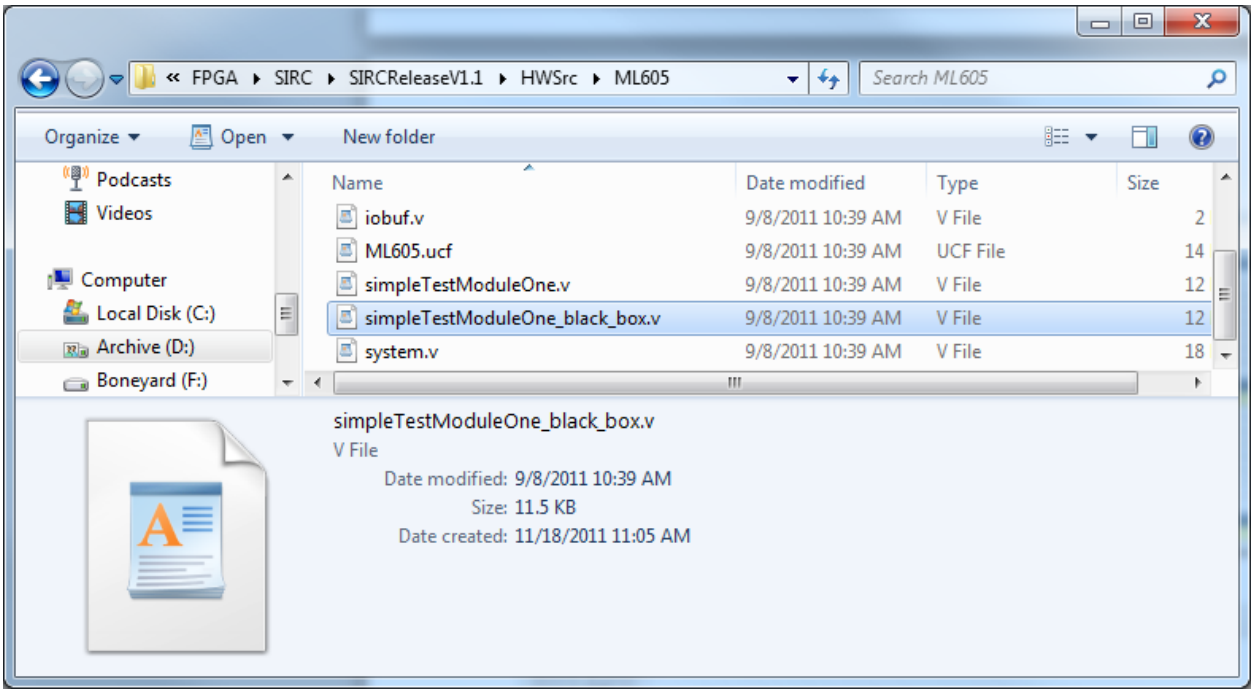


3. Fill the new project with the SIRC design files as described in the SIRC Release README.

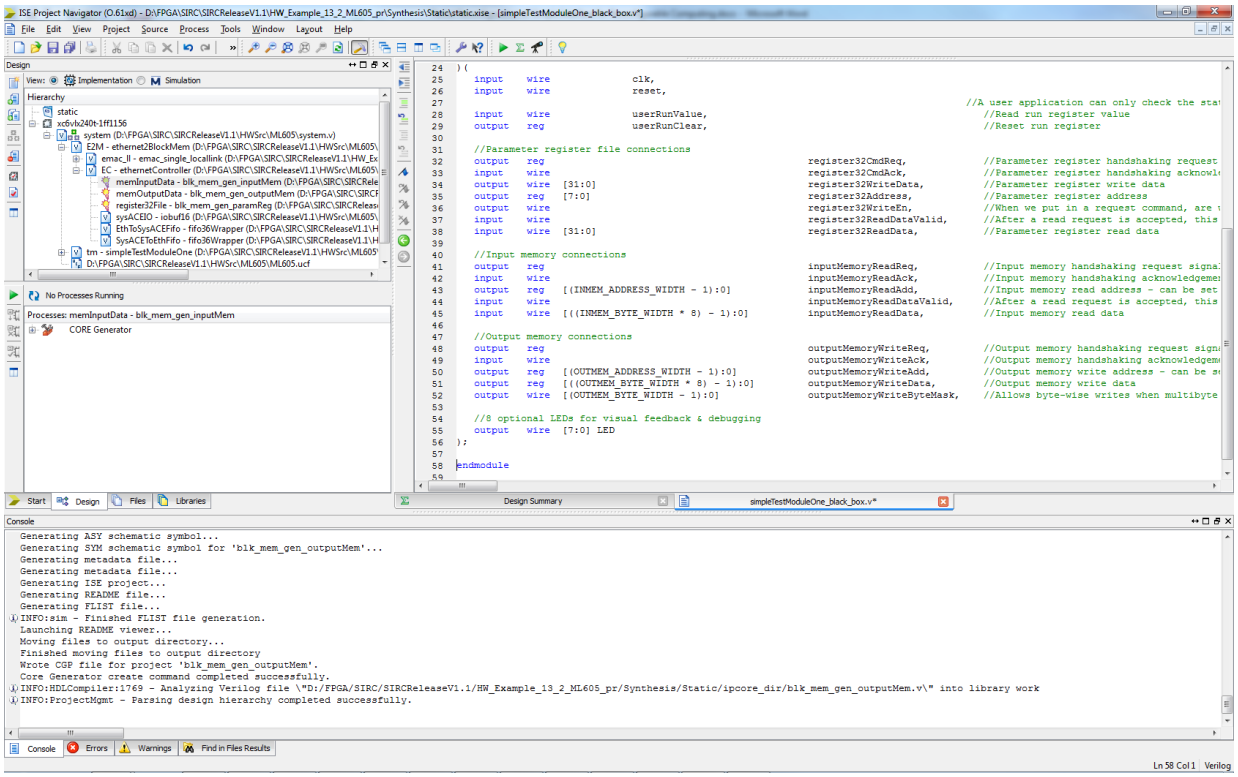


4. Create a Black Box Instance of the user module.

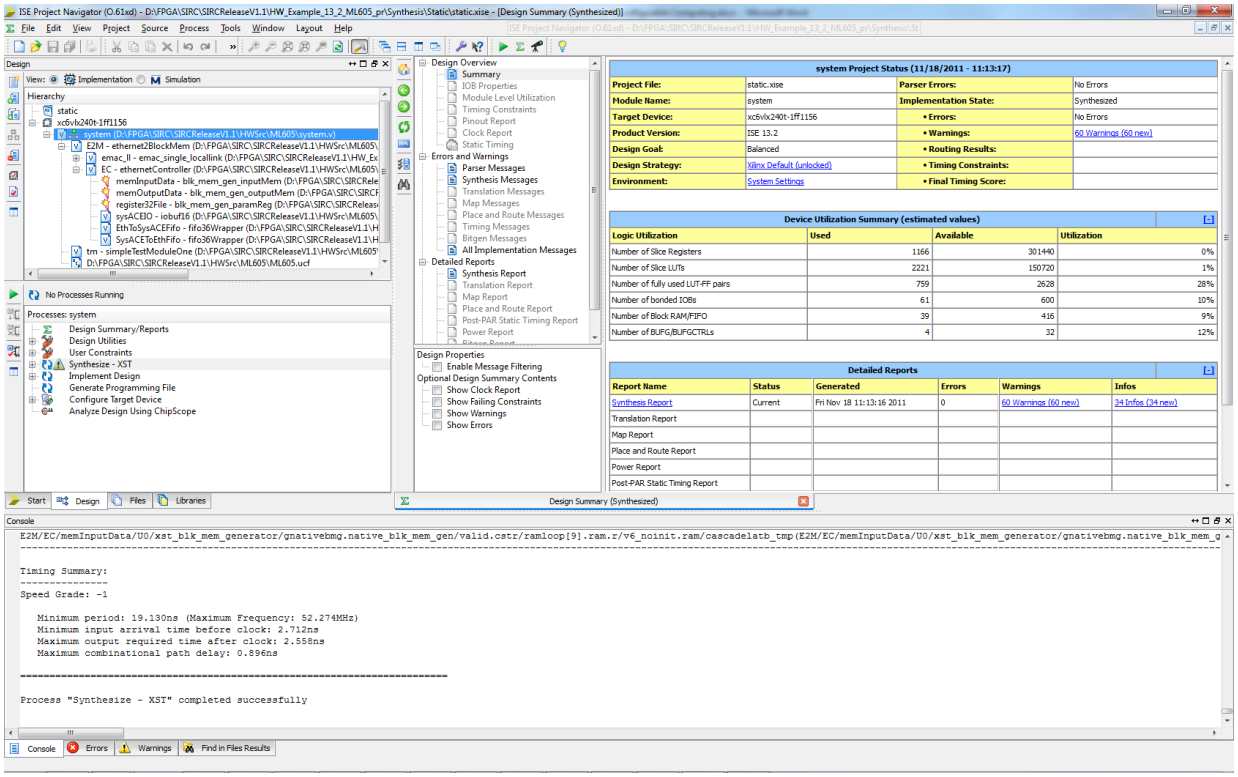
a. Copy the user module file (EX. simpleTestModuleOne.v)
b. Rename file to form <original_file_name>_black_box.v (EX. simpleTestModuleOne_black_box.v)



c. Open the file and delete all the contents from the module declaration to the endmodule tag.



5. Remove the original user module from the project (EX. simpleTestModuleOne.v)
6. Replace the user module file with the black box (EX. simpleTestModuleOne_black_box.v)
7. Synthesize the static design.

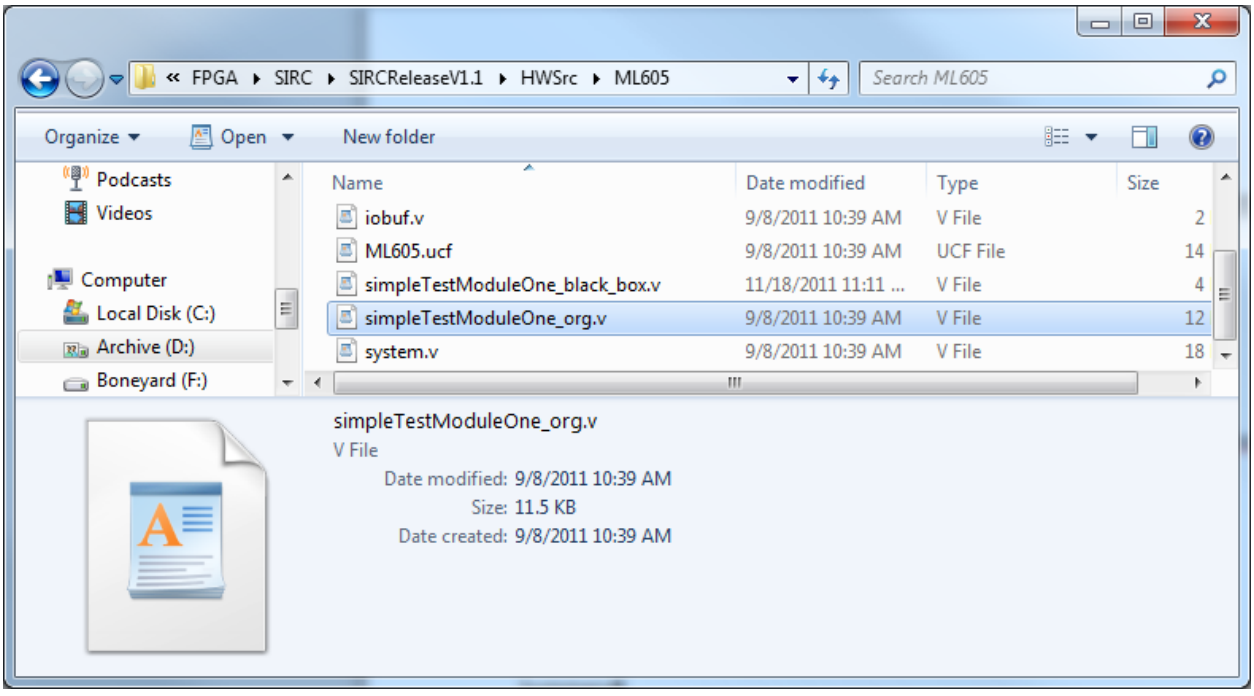


This first step creates the netlists for the static design including the SIRC hardware communication API.

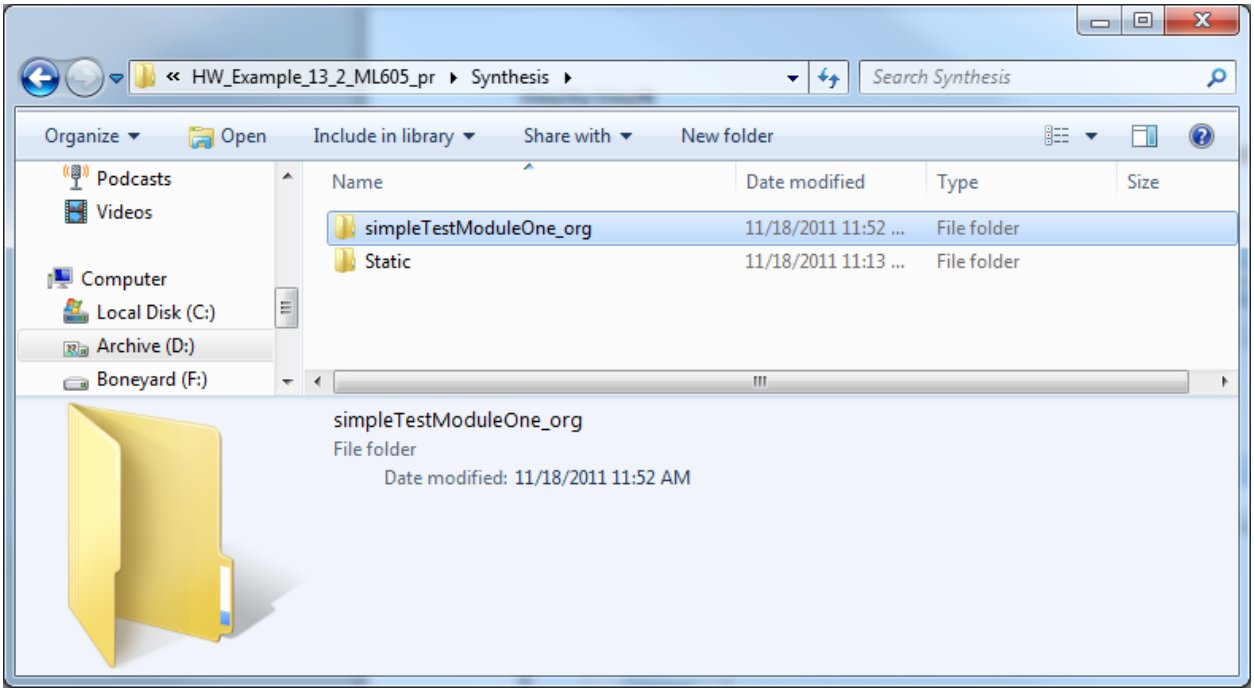
There is a bug in the official SIRC release that will cause issues with the routing of the partial reconfiguration implementation. The BUFG between the input buffer and the PLL is commented out and the input buffer is directly connected to the PLL. This BUFG must be replaced for the design to successfully implement using the Xilinx partial reconfiguration flow. Uncomment the BUFG and change the output of the input buffer to the input of the BUFG. Also uncomment the wire declaration for the wire between the input buffer and the BUFG.

Synthesizing the first Partial Reconfiguration Partition Instance

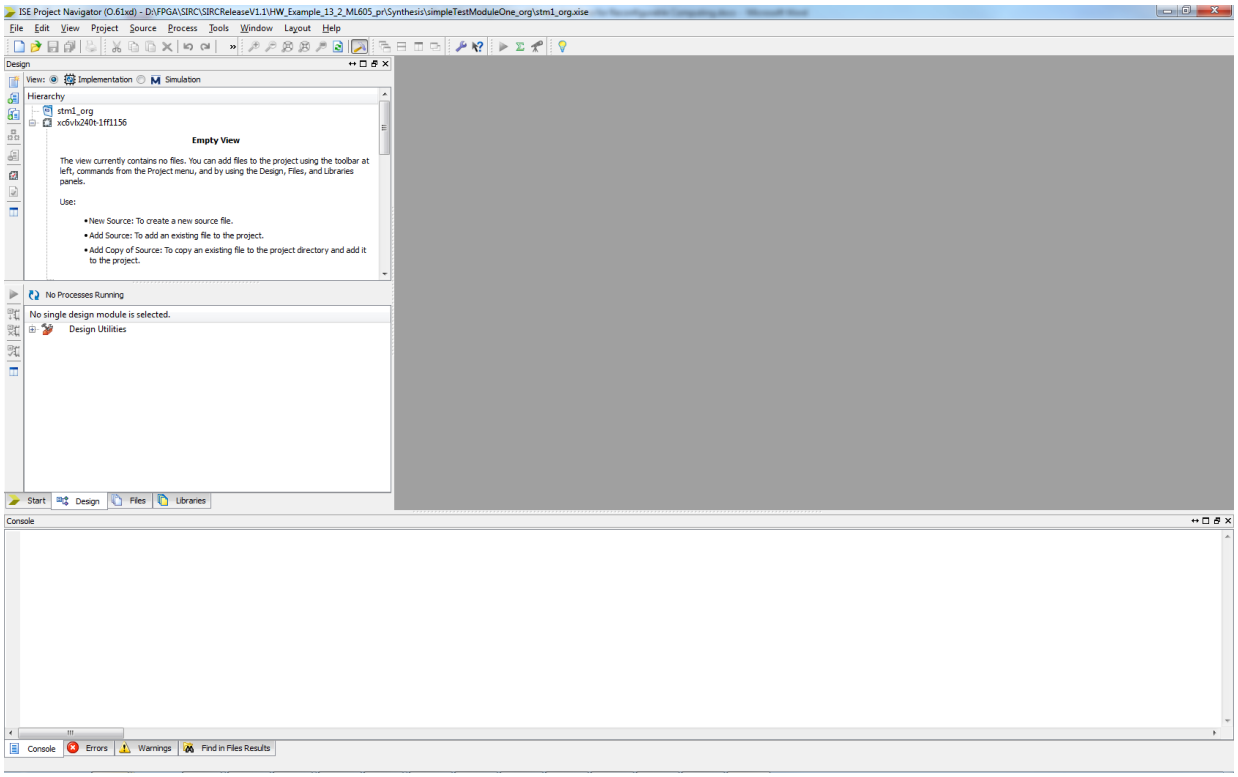
1. Rename the original user module file to '<partition_name>_<instance_name>.v' where '<instance_name>' is 'org' (EX. simpleTestModuleOne_org.v)



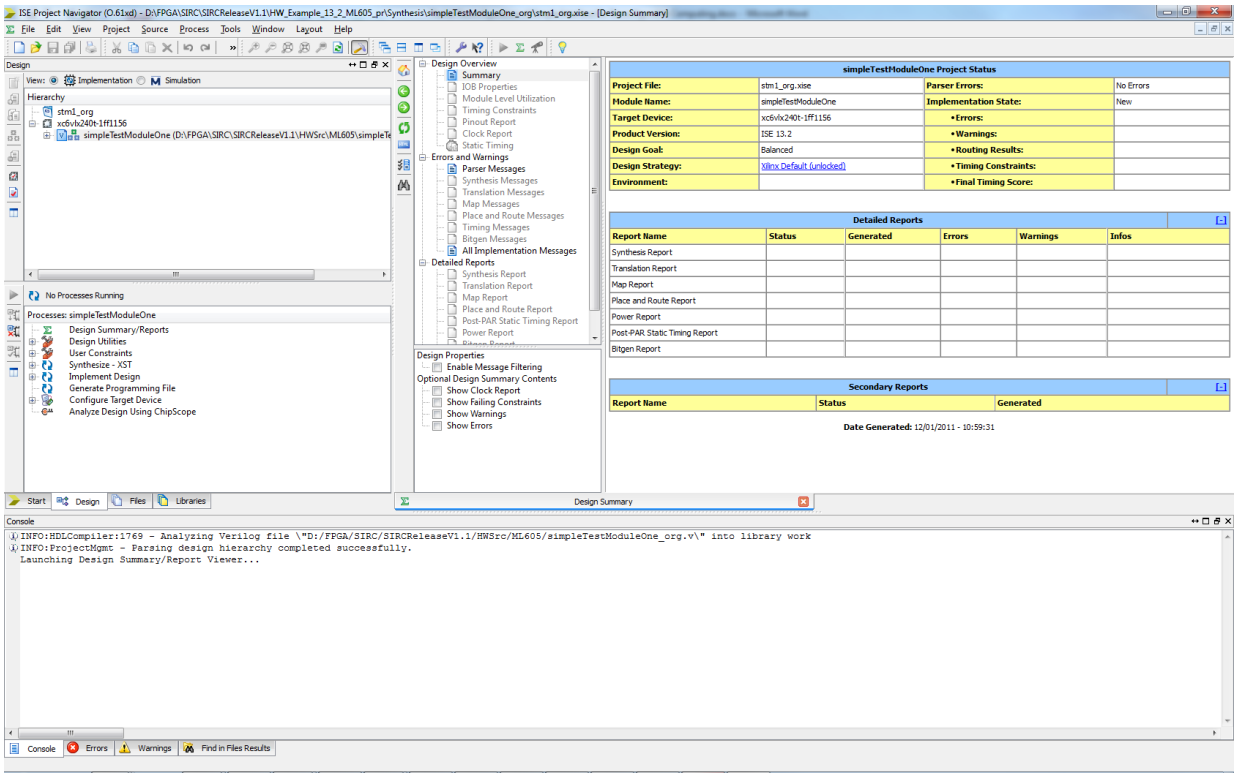
2. Create synthesis folder and label it '<partition_name>_<instance_name>' (EX. simpleTestModuleOne_org)



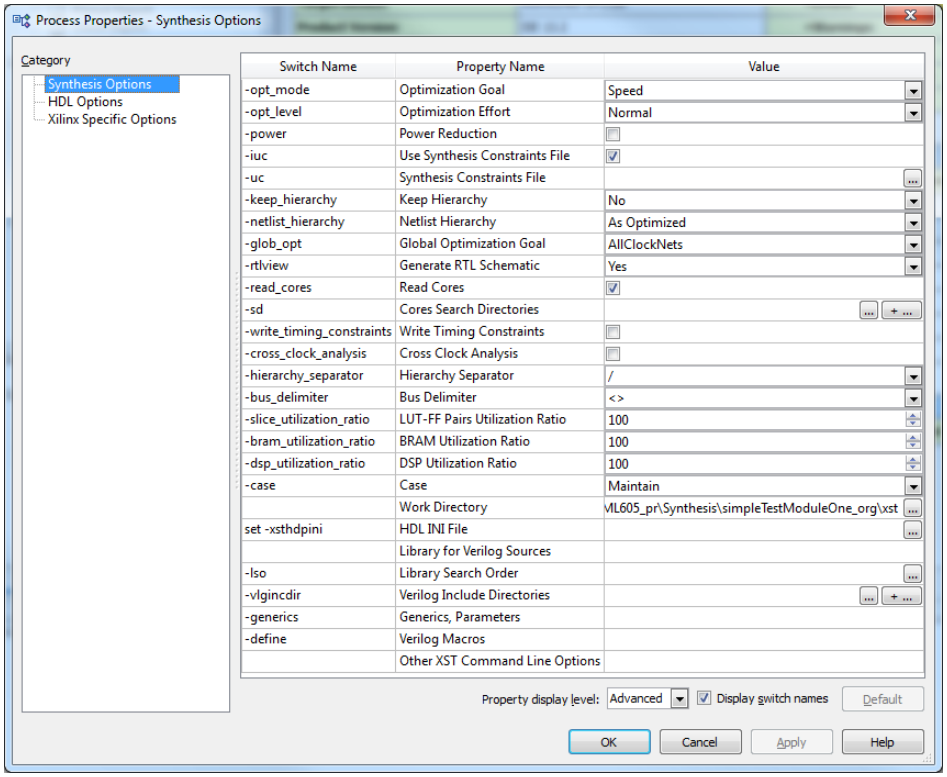
3. Inside that Folder, create a new ISE project.



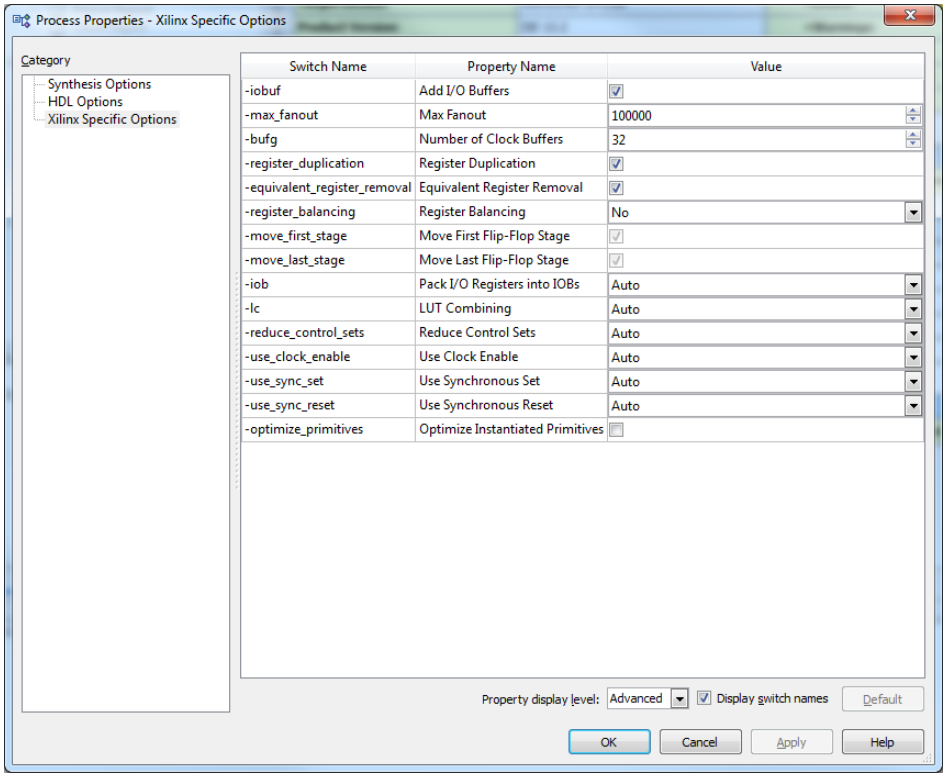
4. Add user module files to project.



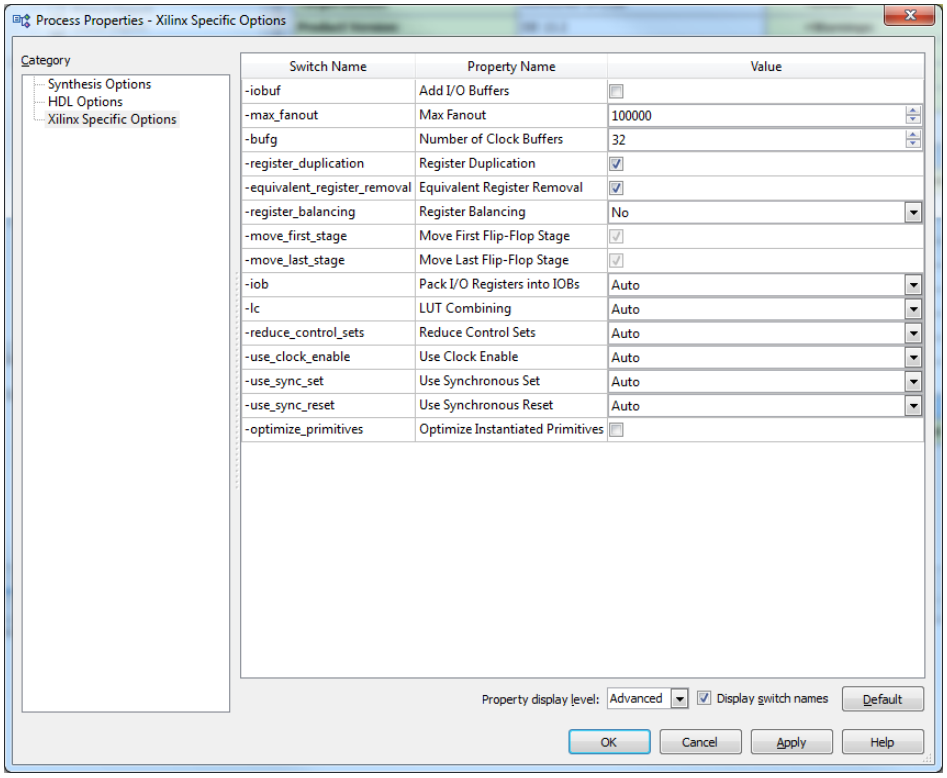
5. Open the ‘Synthesis Options’ dialogue.



6. Navigate to ‘Xilinx Specific Options’ pane.

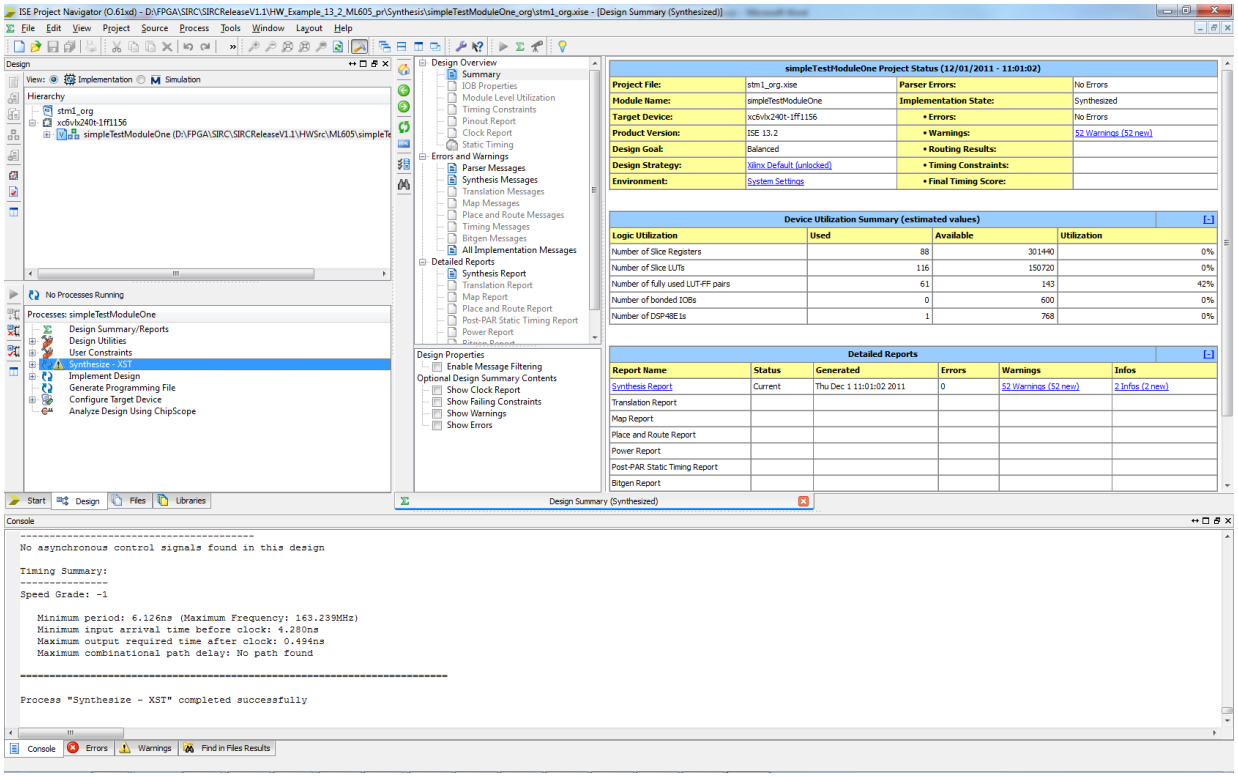


7. Uncheck ‘Add I/O Buffers’ (-iobuf) option.



8. Click ‘OK’ button.

9. Synthesize Module.



10. You may have additional user module instances. An example is provided in the next section. Repeat Steps 2 through 9 for each user module instance.

In this section we have created and synthesized our first partial reconfigurable design.

There is a bug in the SIRC release that if not corrected will prevent the partial bit streams from functioning correctly. The register 'register32CmdReq' is not initialized by the reset. Since register values can be of unknown state following partial reconfiguration and there is not global reset for the FPGA, these must be reset in the logic. Add the following line to the reset state of 'simpleTestModuleOne' stating on line 114 of 'simpleTestModuleOne_org.v'.

```
register32CmdReq <= 0;
```

Synthesizing more Partial Reconfiguration Partition Instances

By now, the original user module instance netlist has been generated. However, the purpose of the partial reconfiguration flow is to have multiple instances of the partial reconfiguration partition. In

order to accomplish this make another copy of the original user module file and rename it to the form '<partition_name>_<instance_name>.v' where '<instance_name>' is 'alt' (EX. simpleTestModuleOne_alt.v). Open the file and make the following change to line 204:

```
outputMemoryWriteData <=
(inputFifoDataOut * multiplier) % 256;
```

→

```
outputMemoryWriteData <=
 (inputFifoDataOut + multiplier) % 256;
```

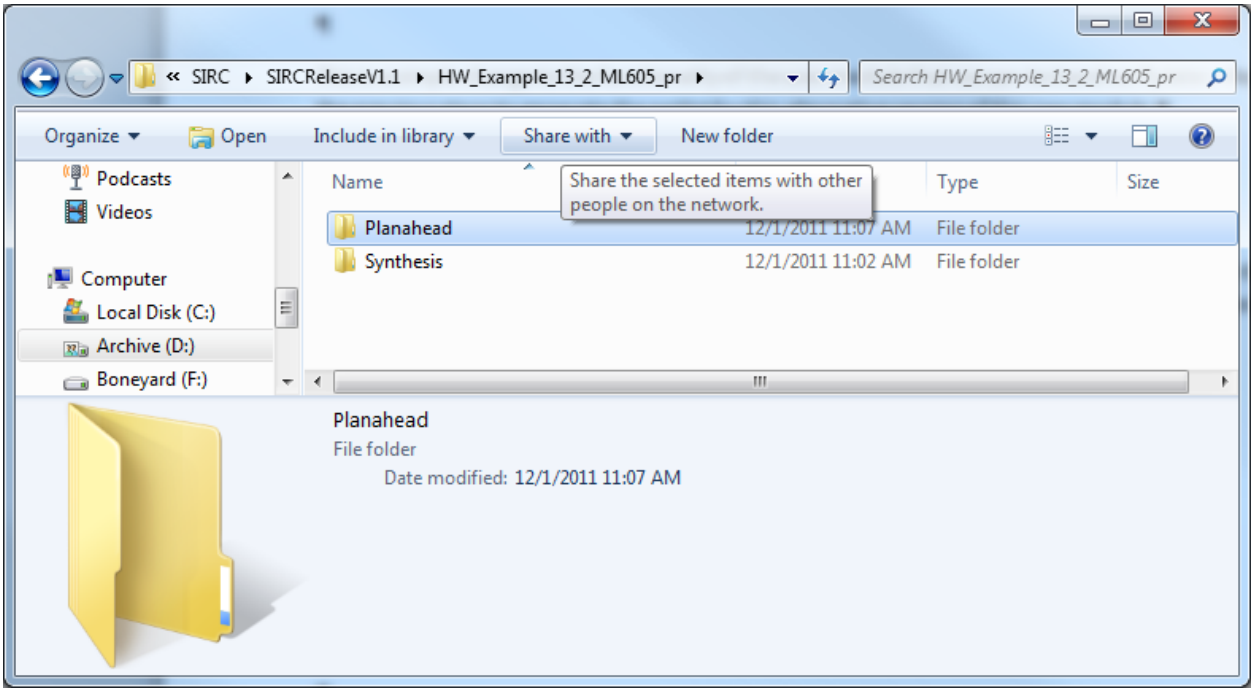
E.g. instead of multiplying we will be adding. This is a simple change to the functionality of the user module that we can test and verify later. Repeat the steps in the previous section to generate the netlist for this alternative version of the user module.

PlanAhead

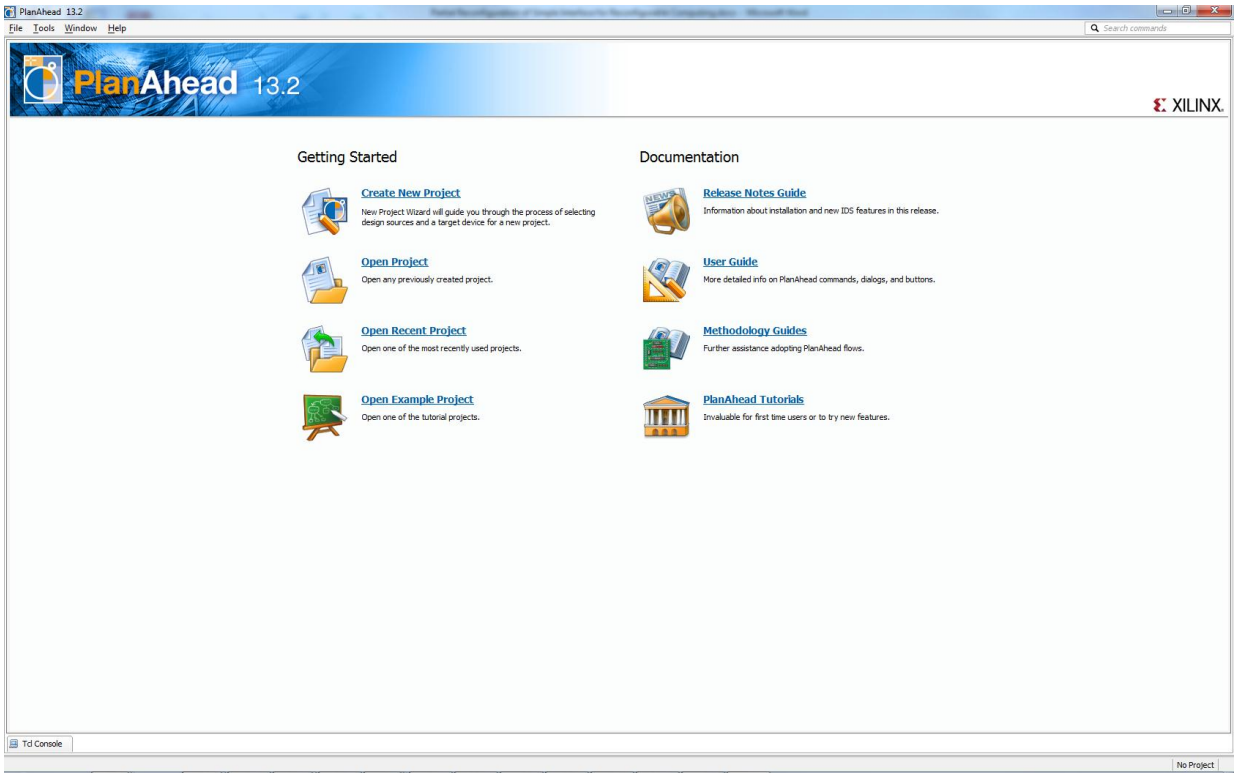
The Xilinx partial reconfiguration design flow is managed by the PlanAhead application included in the Xilinx IDE. This is the tool that allows you to define the physical placement of the static and PR regions on your target FPGA. The netlists generated in the previous sections must be imported into a PlanAhead project and used to implement the design for the targeted FPGA.

Creating a PlanAhead project

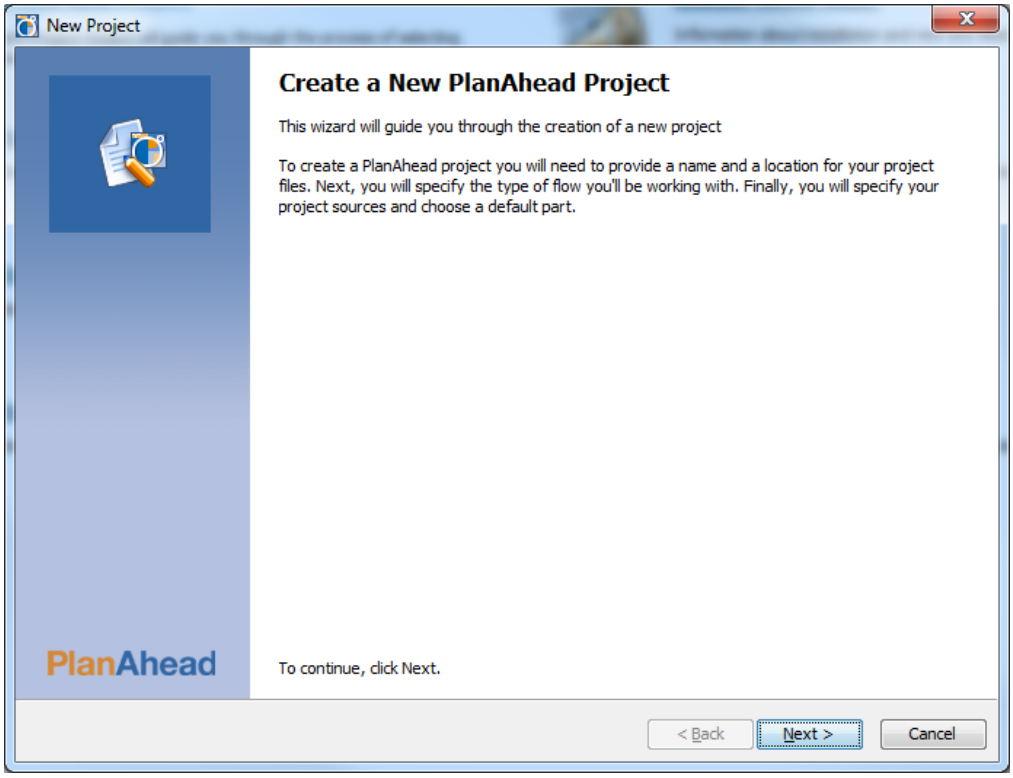
1. Create a new PlanAhead folder.



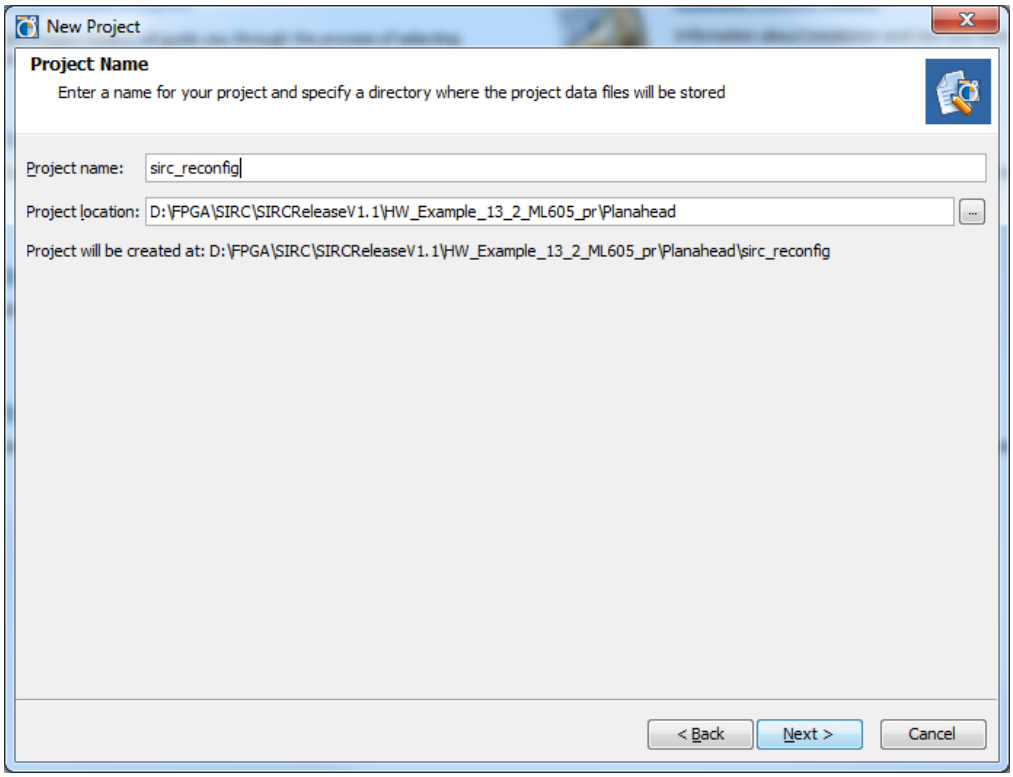
2. Open PlanAhead to create a new project.



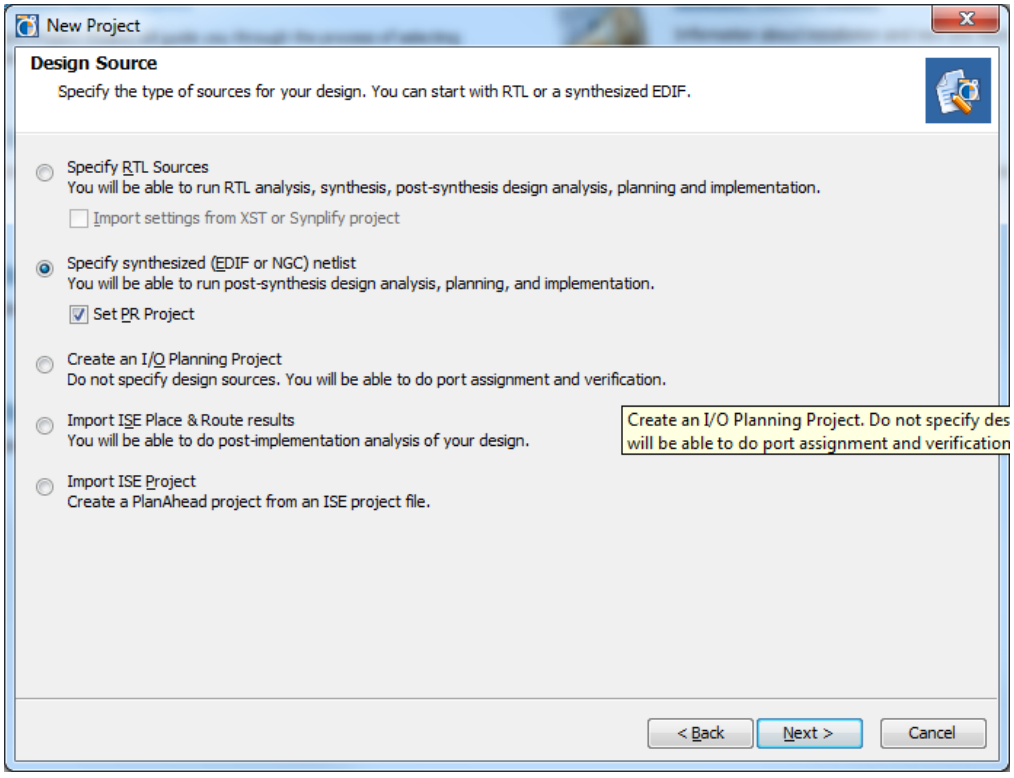
3. In the ‘New Project’ Dialogue click the ‘Next’ button.



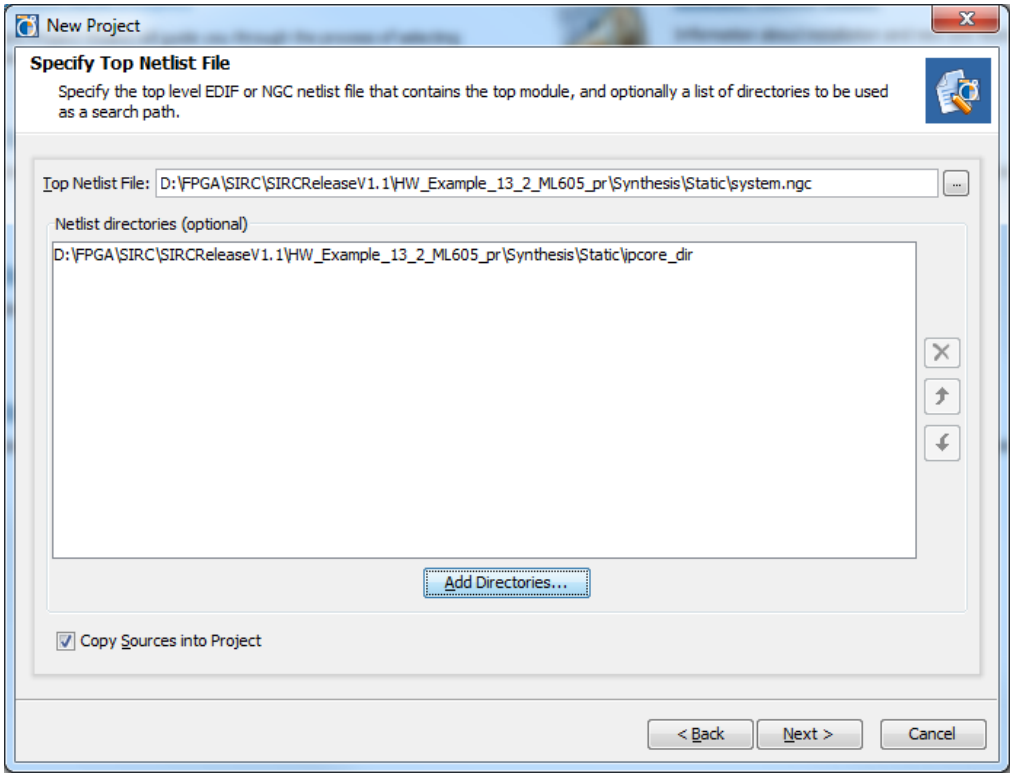
4. Name the project (EX. sirc_reconfig) and navigate to the PlanAhead folder. Click the 'Next' button.



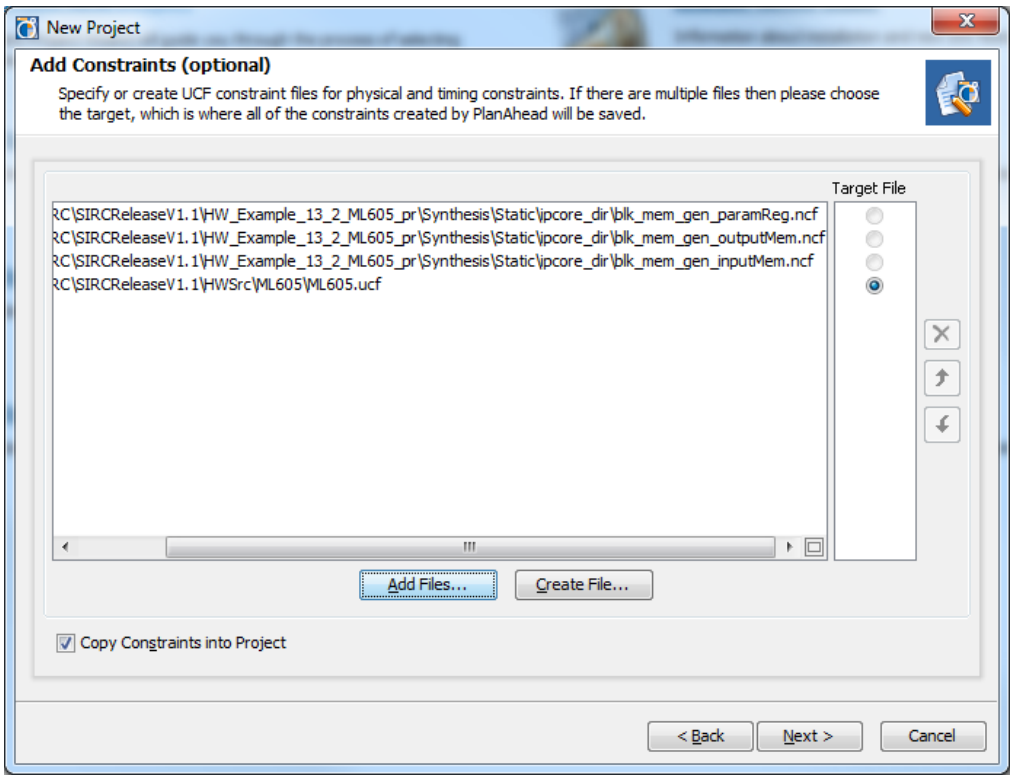
5. Select ‘Specify synthesized (EDIF or NGC) netlists’. Check the ‘Set PR Project’ box. Click the ‘Next’ Button.



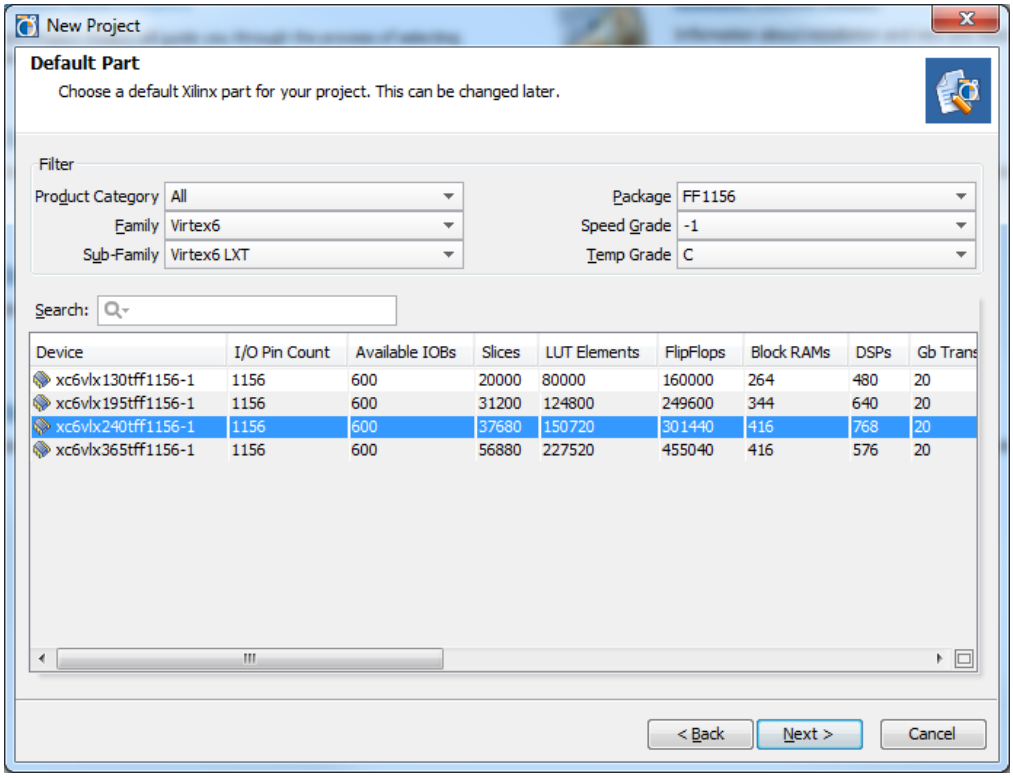
6. Set the location of the static netlists. Add the ip core directory. Click the ‘Next’ button.



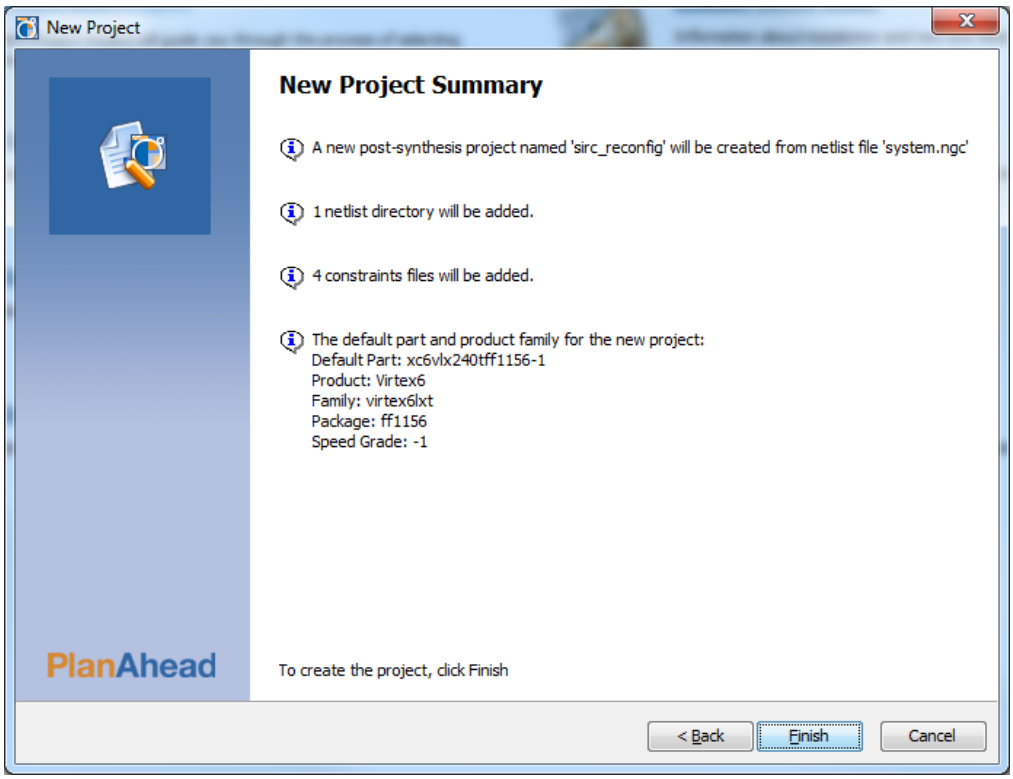
7. Add user constraint file. Click the 'Next' button.



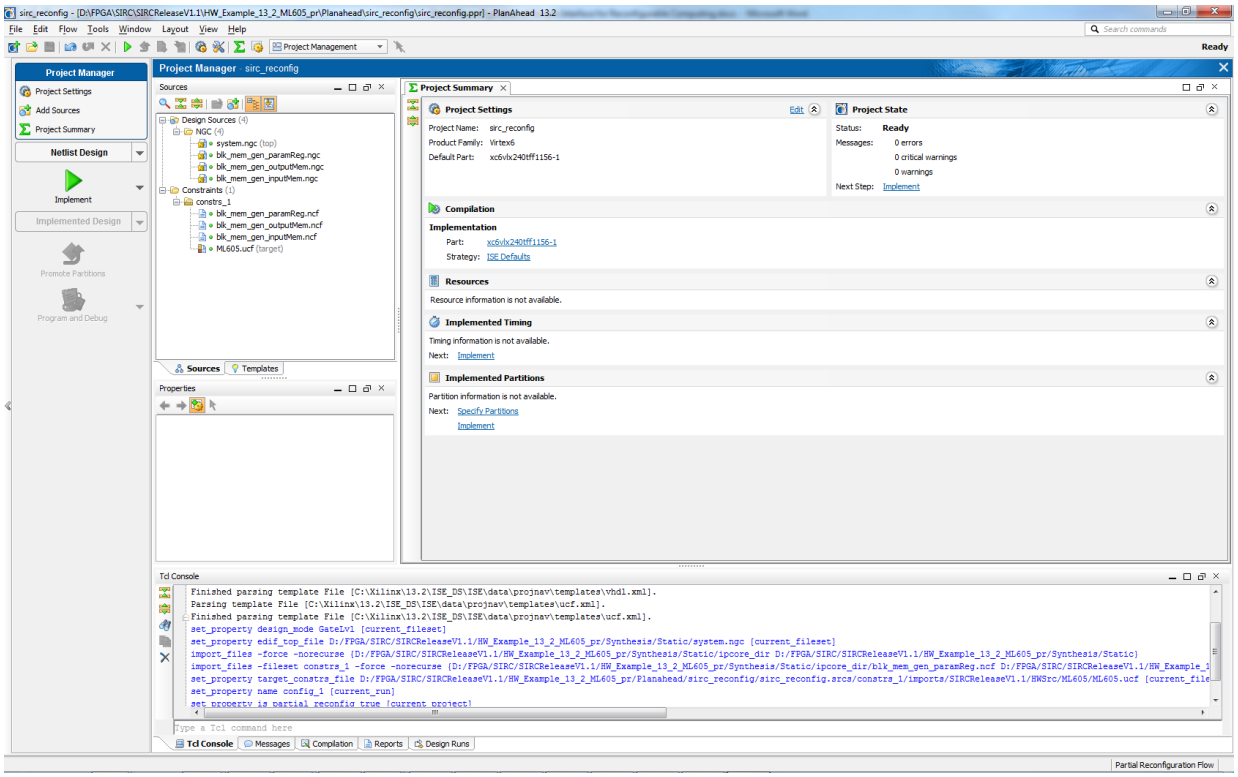
8. Select part. For this we are targeting the Virtex 6 LX240T on the ML605. Click the 'Next' button.



9. Review Project. Click the 'Finish' button.

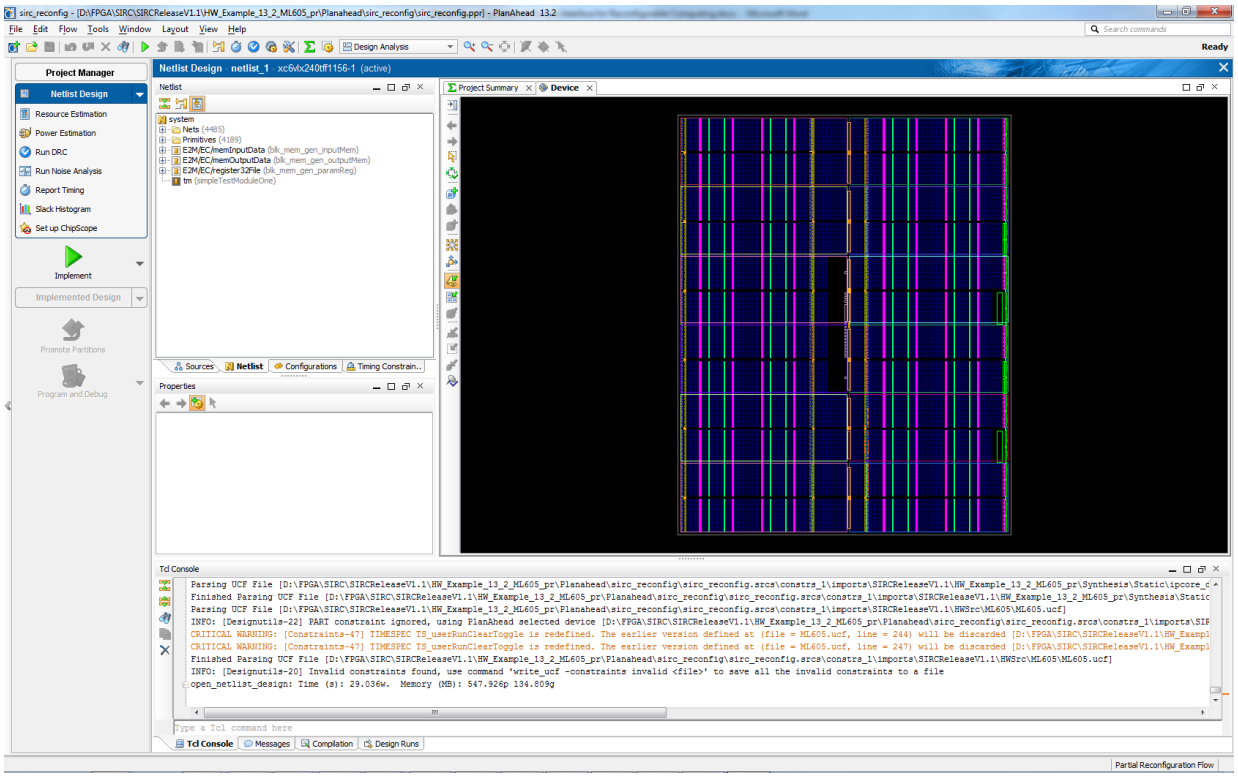


10. Project will open.

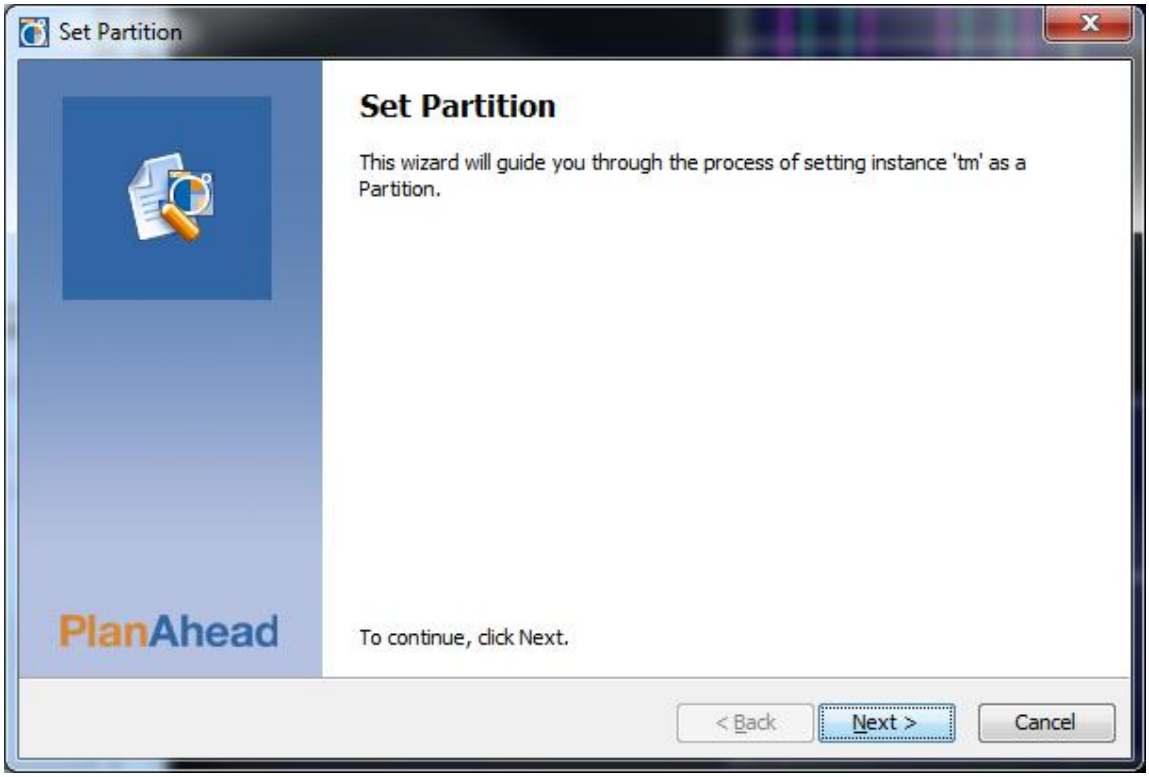


Floor planning Partial Reconfigurable Partition

1. Change Flow to ‘Netlists Design’. (Flow->Netlists Design).

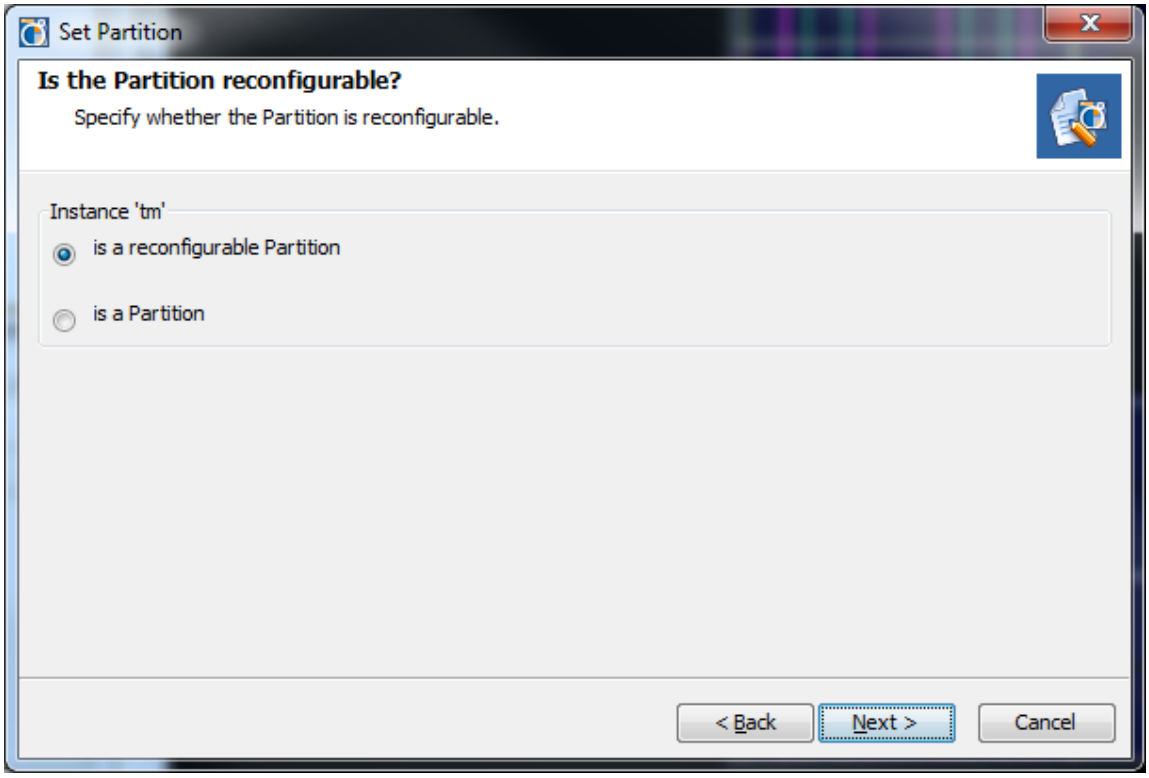


2. Right-click user module in ‘Netlists’ pane and select ‘Set Partition’.

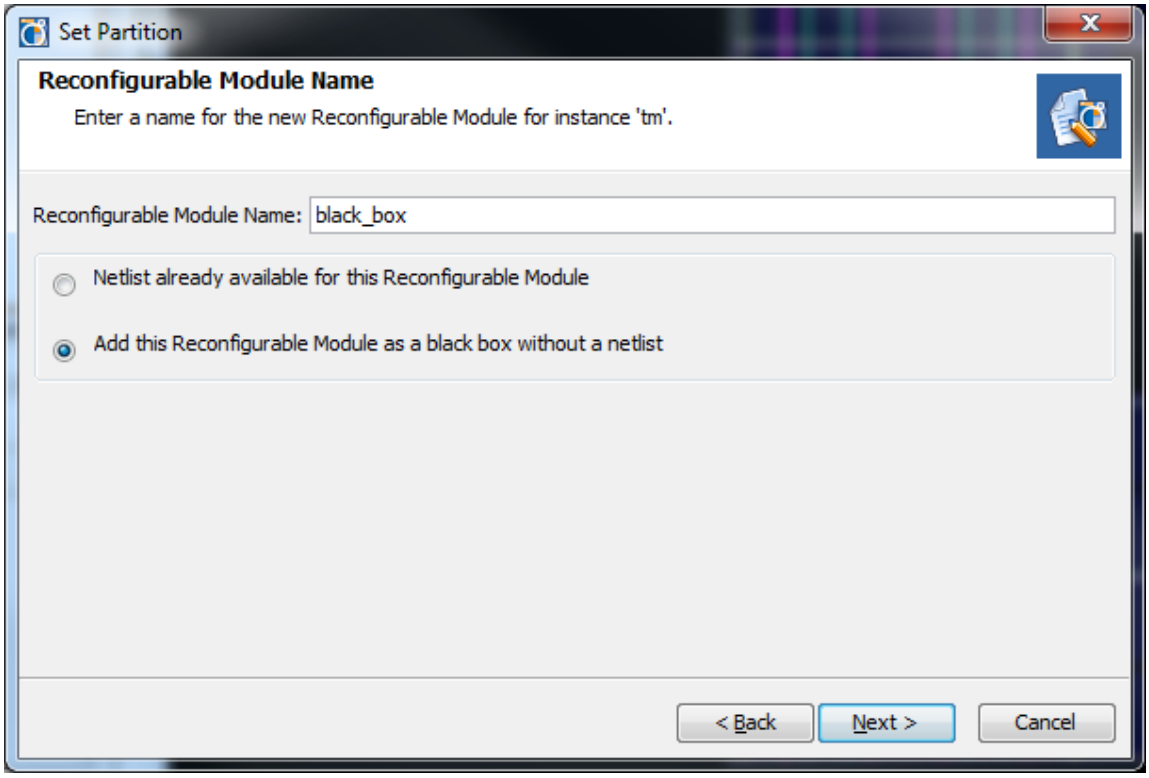


3. Click the ‘Next’ button.

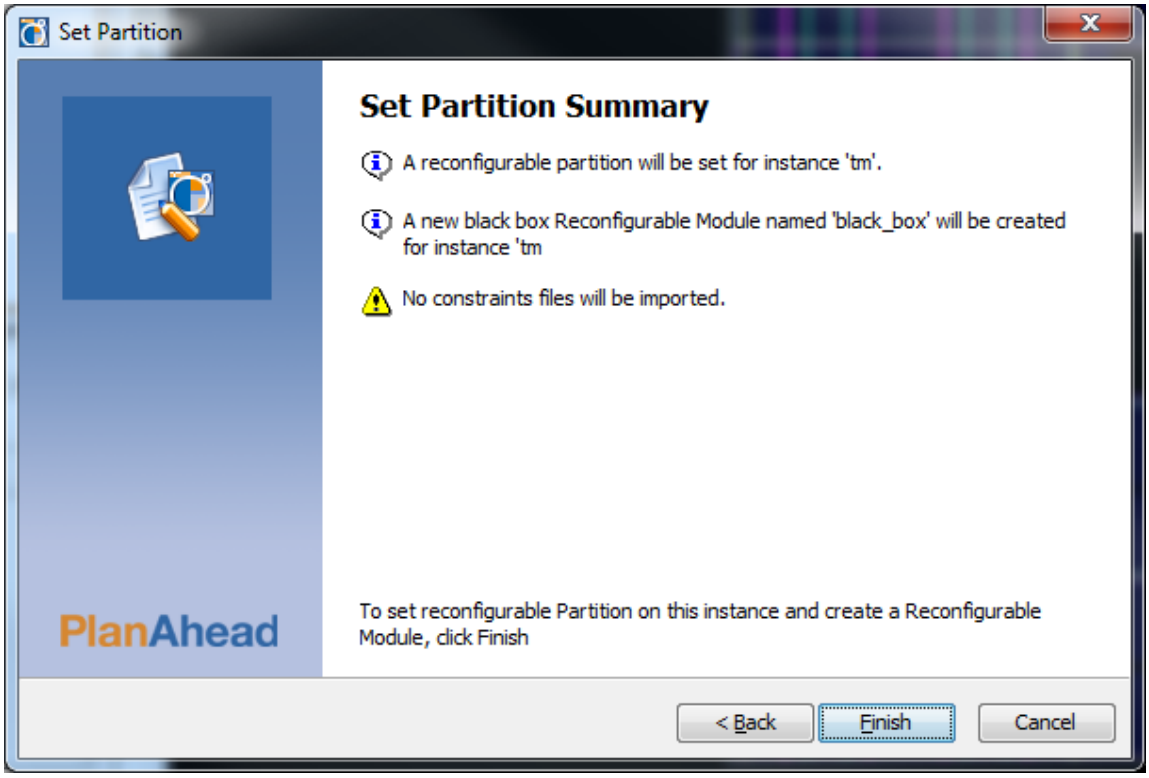
4. Select ‘is a reconfigurable Partition’. Click the ‘Next’ button.



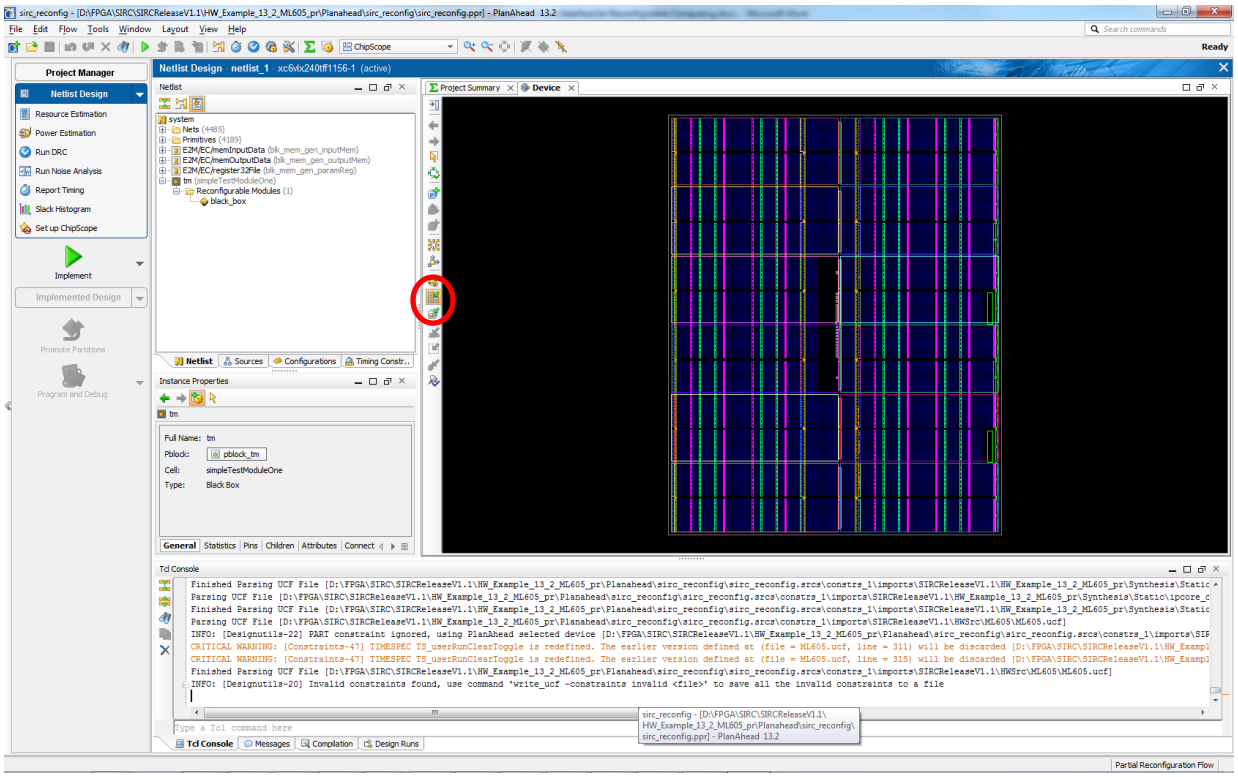
5. Name the module instance ‘black_box’ and select ‘Add this Reconfigurable Module as a black box without a netlist’. Click the ‘Next’ button.



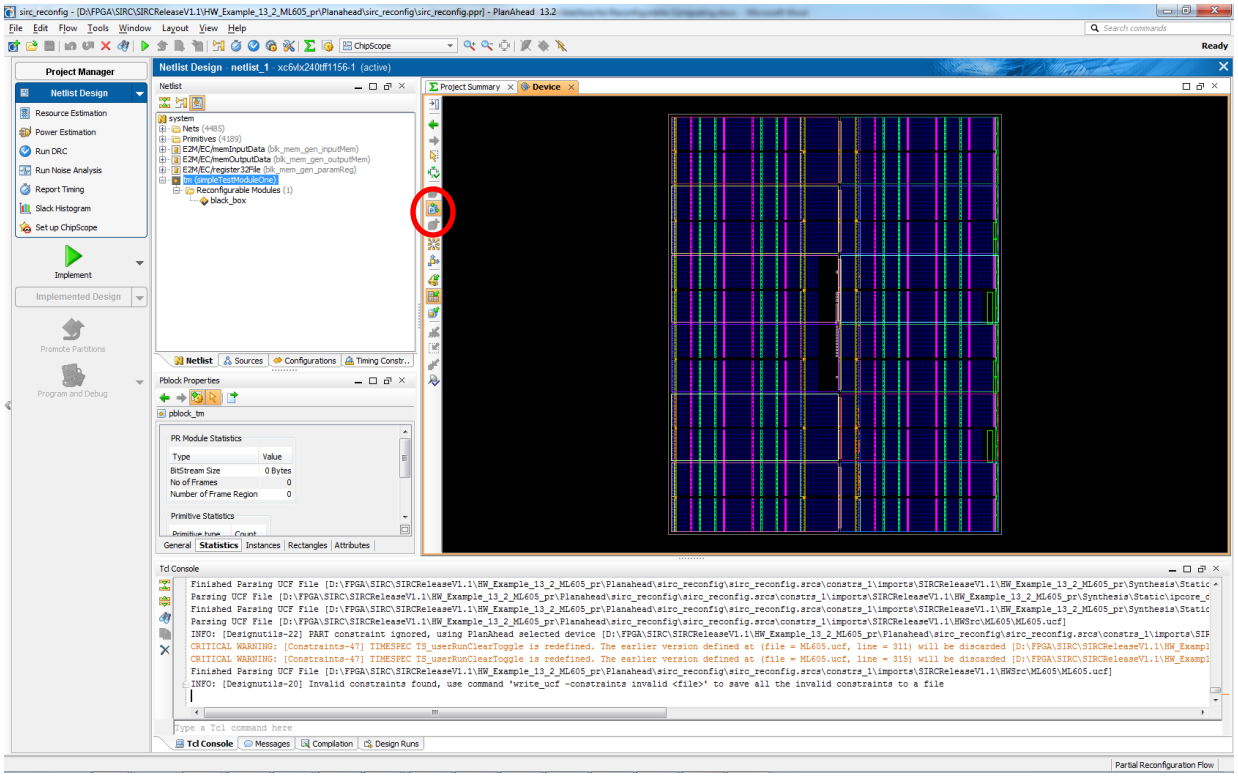
6. Review module details. Click 'Finish' button.



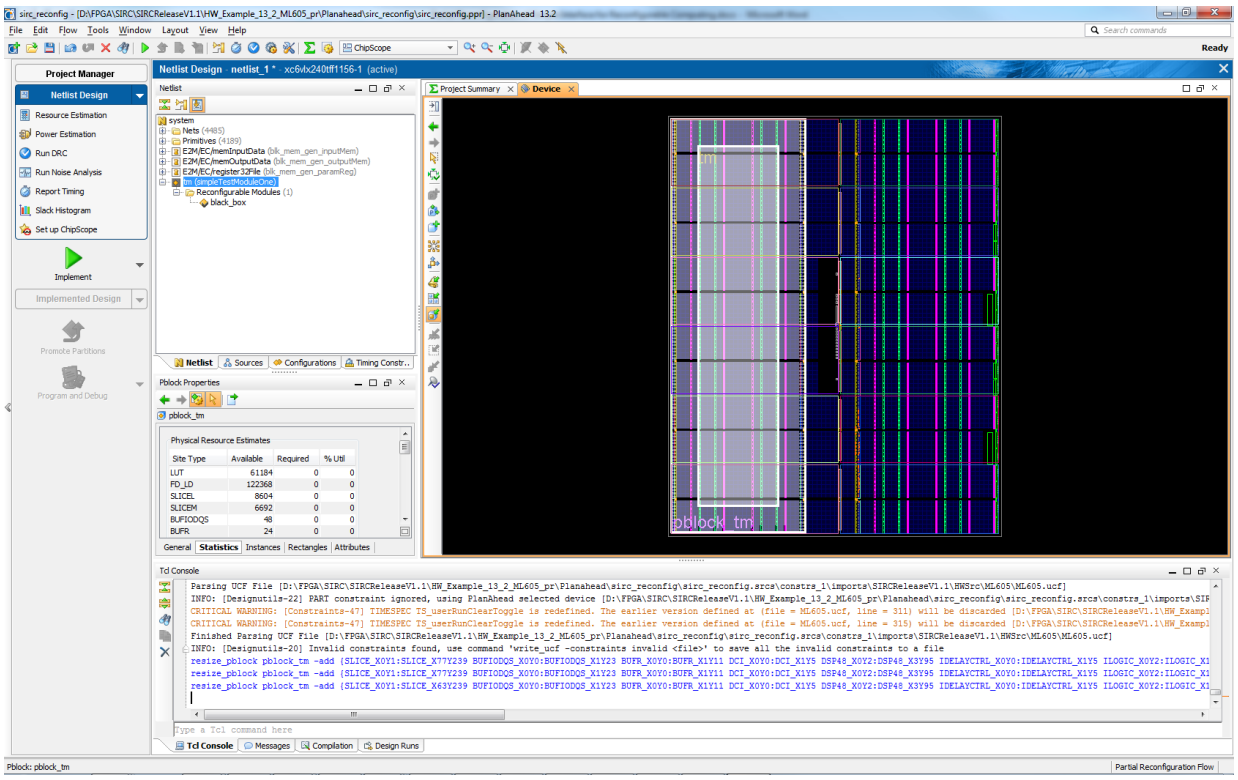
7. In the 'Device' pane, click the 'Assign Pblock Mode' button on the toolbar.



8. Select the user module and press the ‘Set Pblock Size’ on the tool bar.

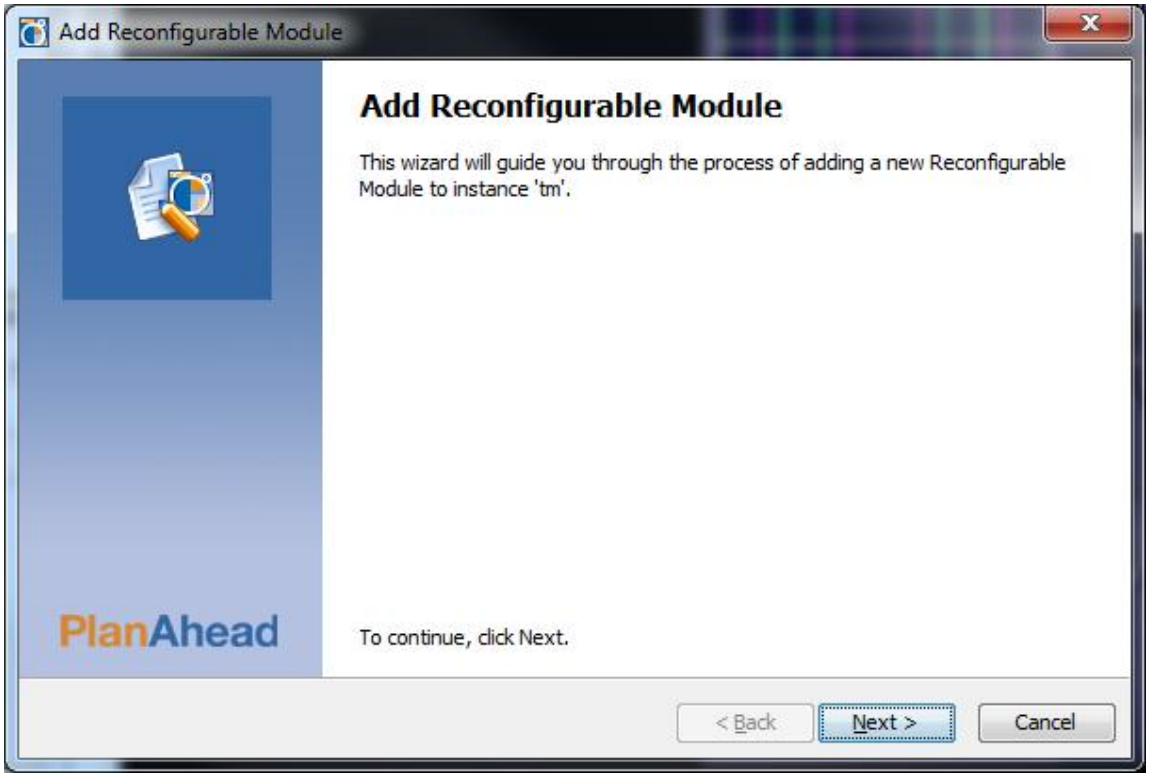


9. Draw a rectangle for your user module. For our simple example we need very few resources, but we will use the entire left half of the FPGA anyways.

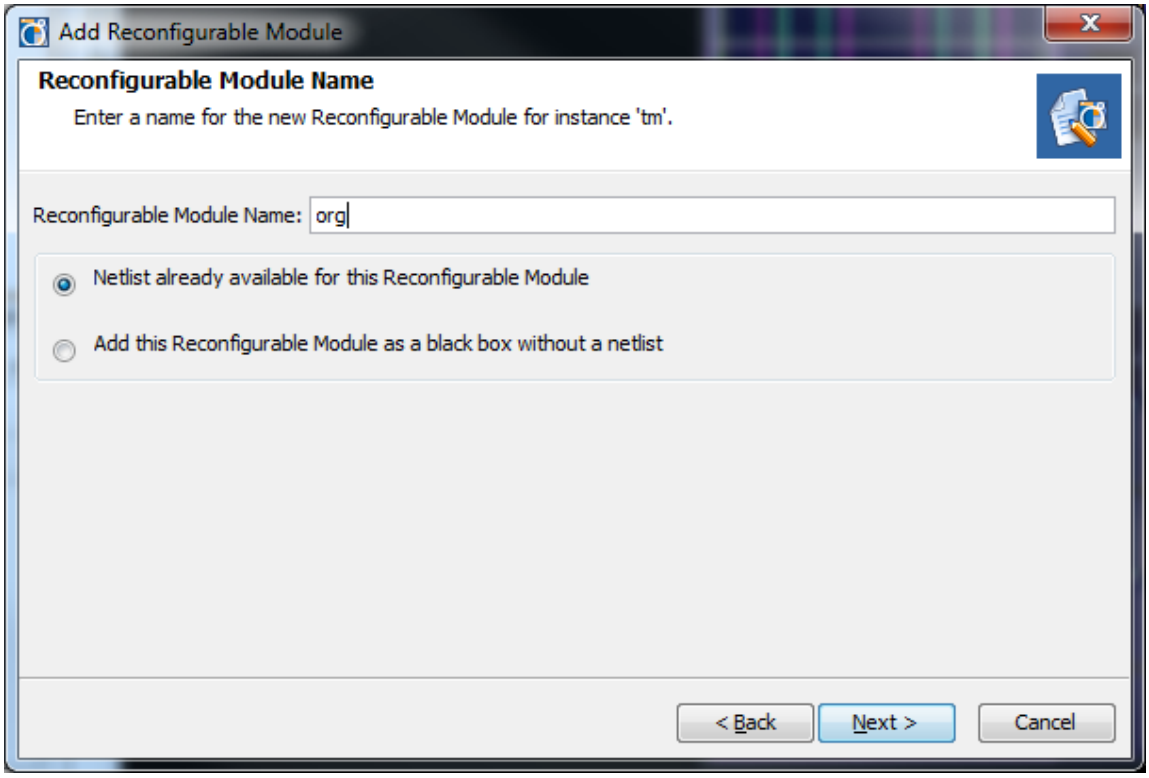


Adding Reconfigurable Instances to the Partial Reconfiguration Partition

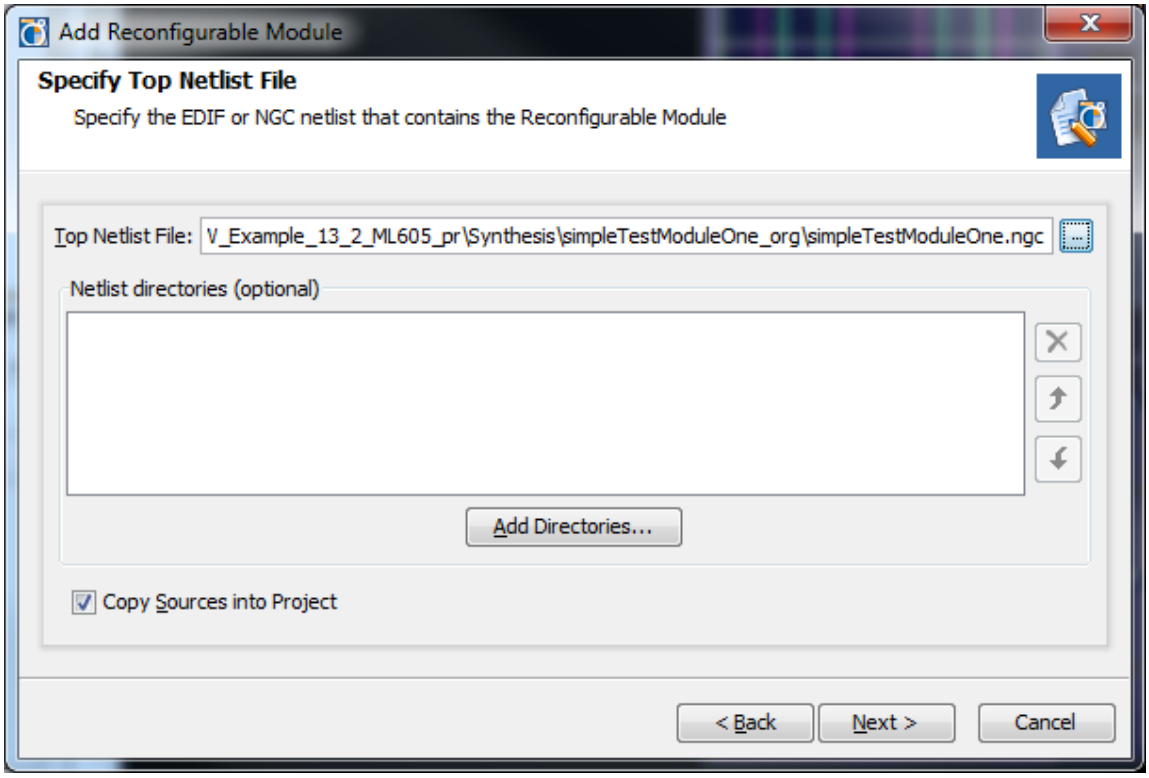
1. Right-click the user module and select ‘Add Reconfigurable Module’.



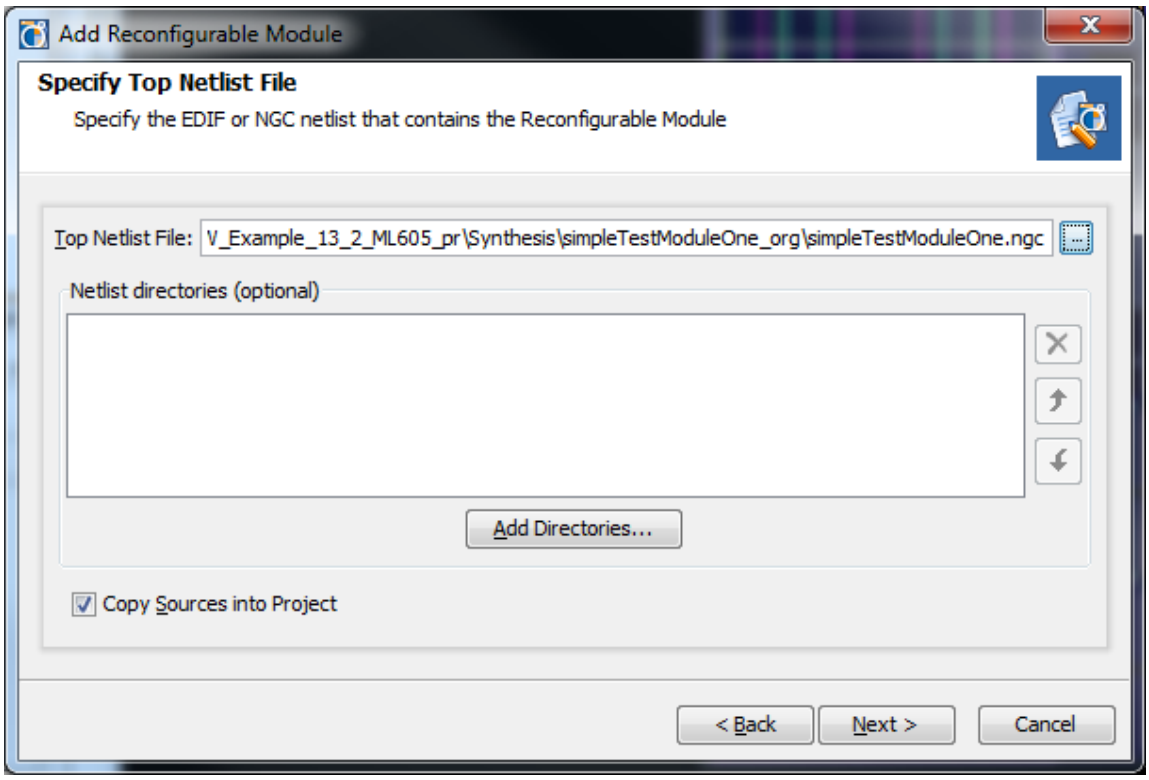
2. Click the ‘Next’ button.
3. Name the module (EX. org) and select ‘Netlist already available for this Reconfigurable Module’. Click the ‘Next’ button.



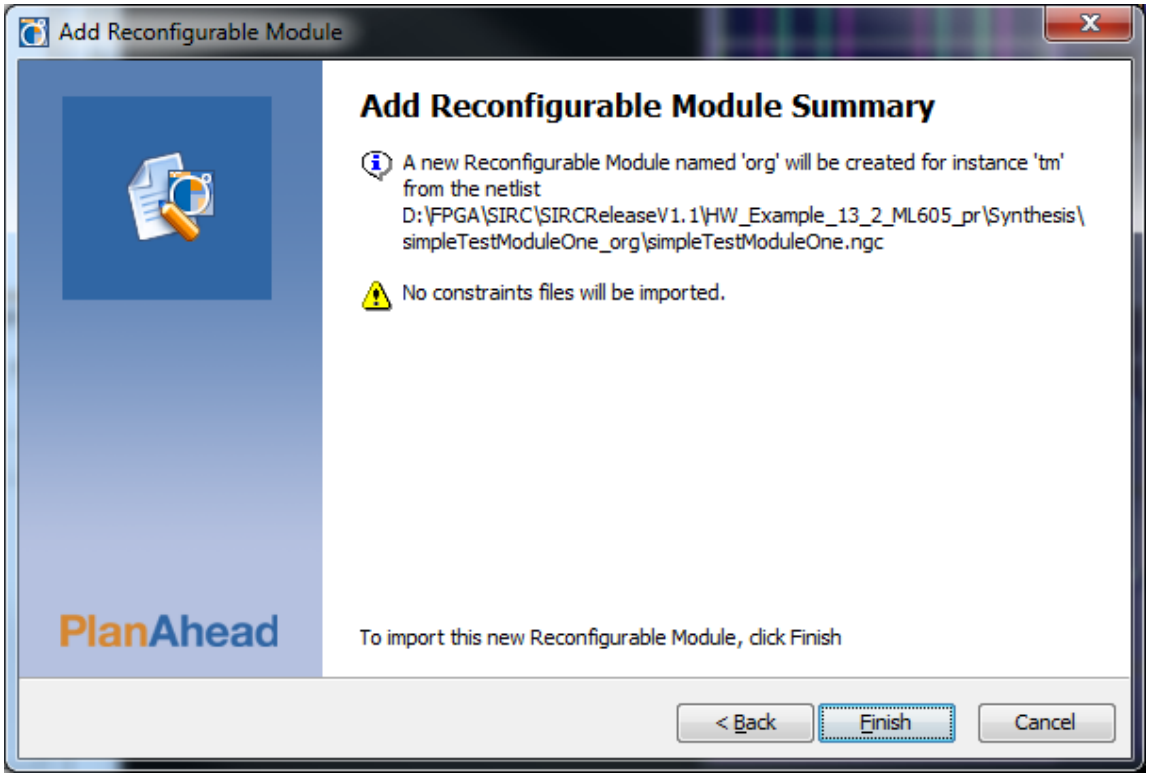
4. Locate the module netlist in the synthesis directory. Click the 'Next' button.



5. Here you may add a module constraint file. There is none for this example. Click the 'Next' button.



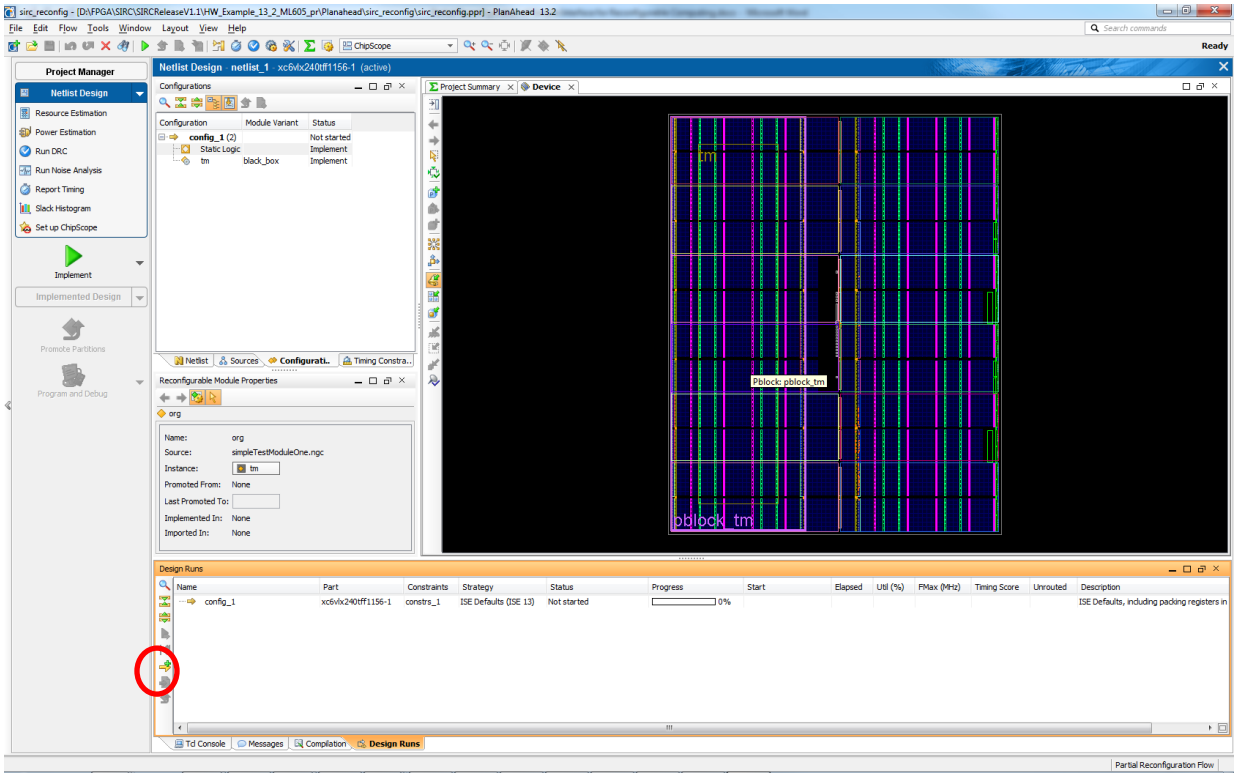
6. Review module parameters. Click the 'Next' button.



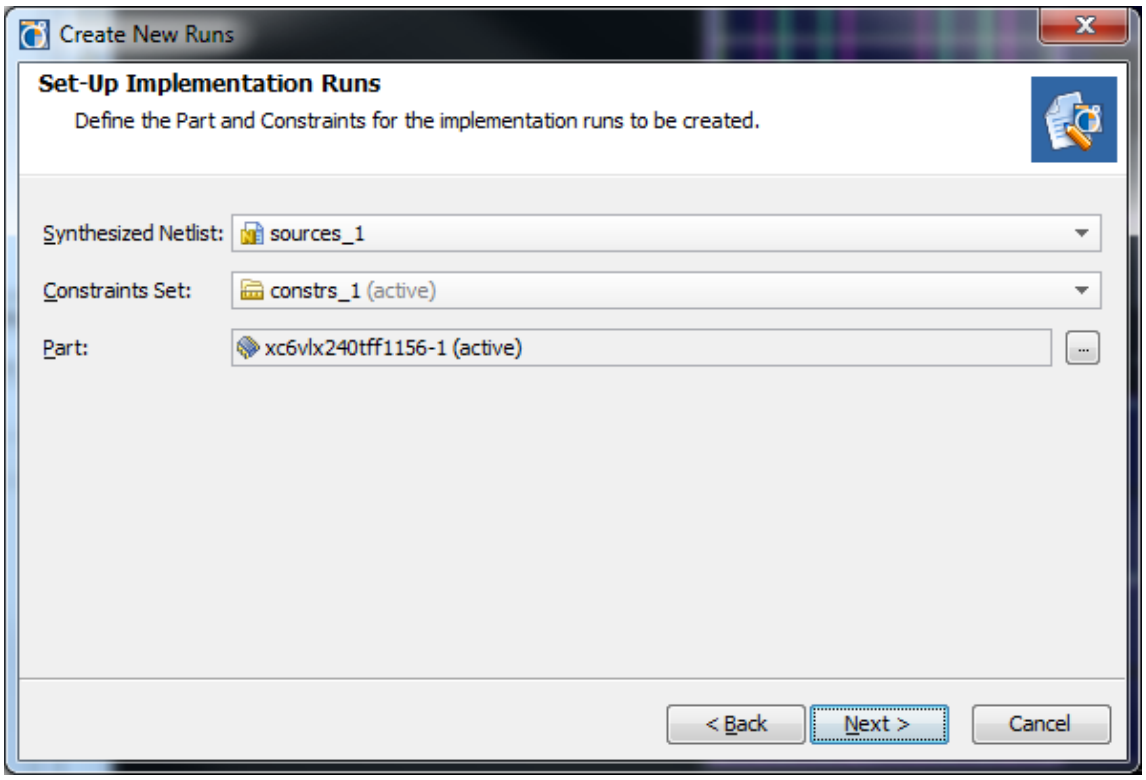
7. Repeat steps 21 through 26 for the alternate version of the user module, the one that does the addition instead of the multiplication.

Create Design Instances for Implementation

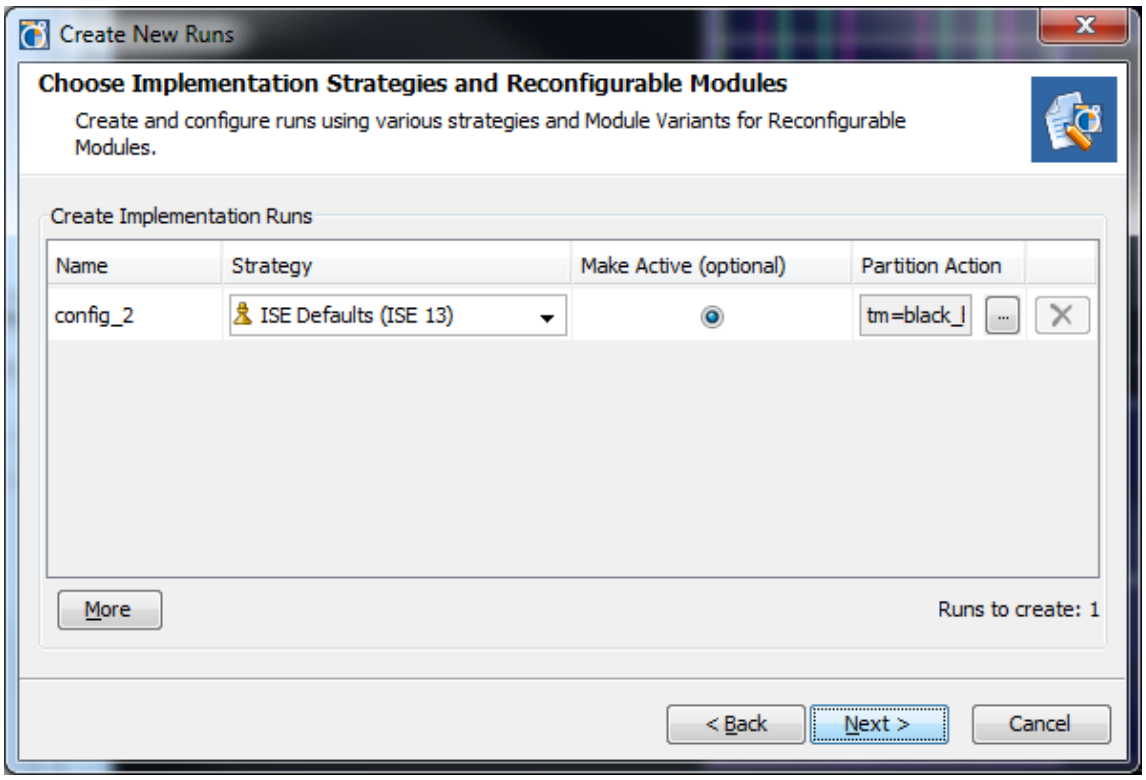
1. In the ‘Design Runs’ pane, press the ‘Create New Runs’ button on the toolbar.



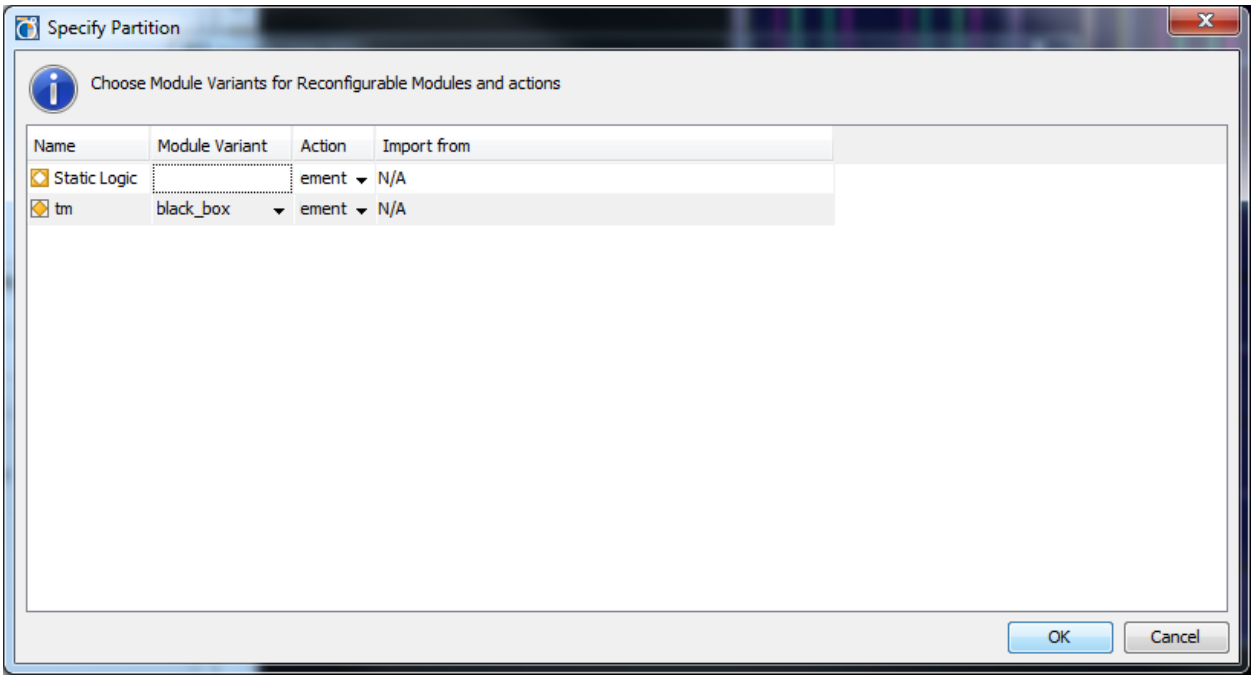
2. Click the ‘Next’ Button.



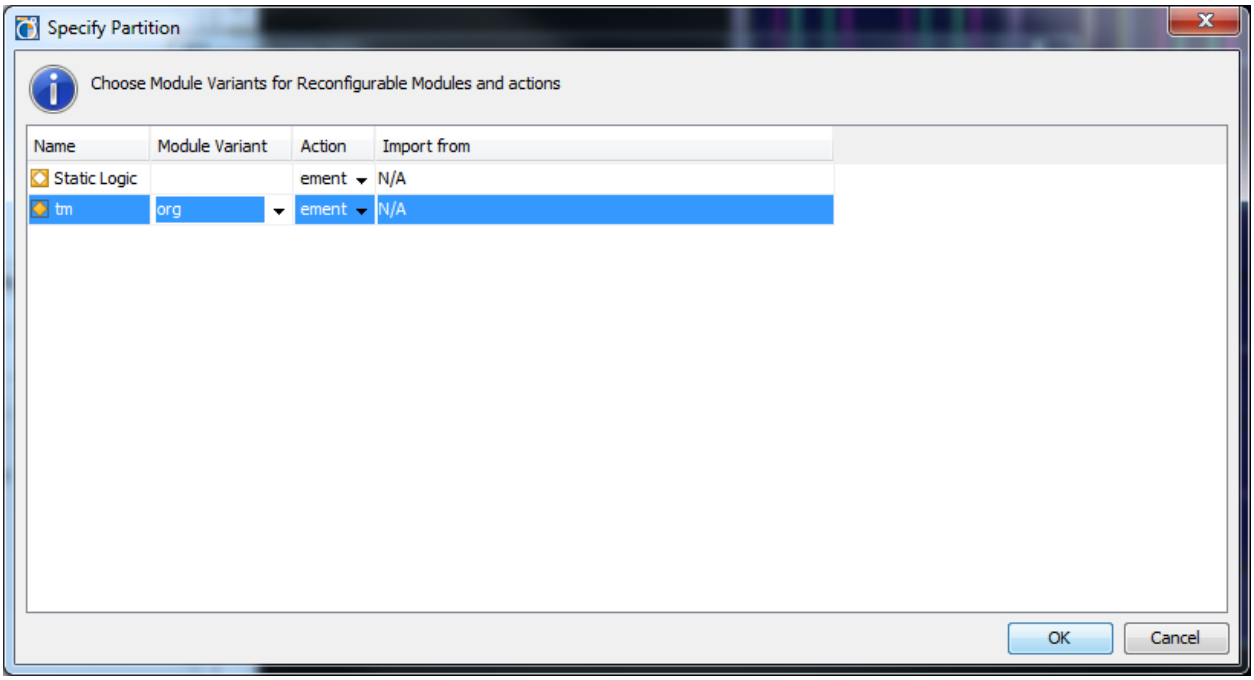
3. Click the ‘Next’ Button.



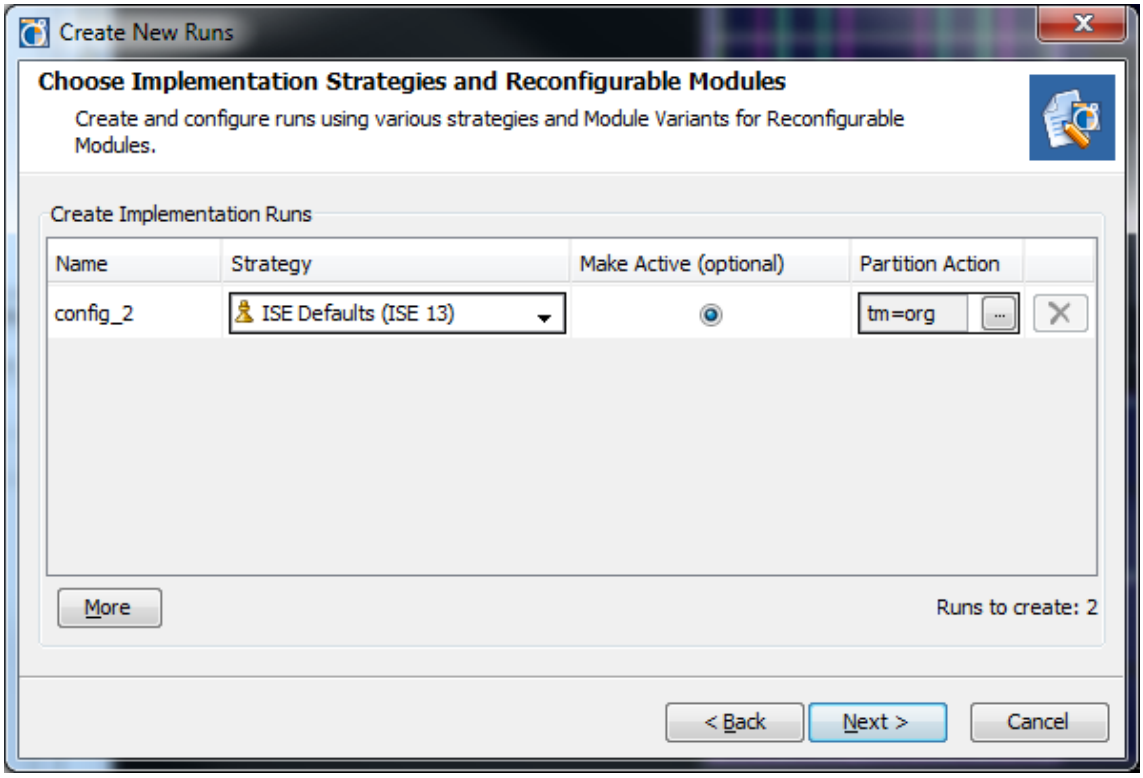
4. Click ‘Partition Action’ Button for ‘config_2’.



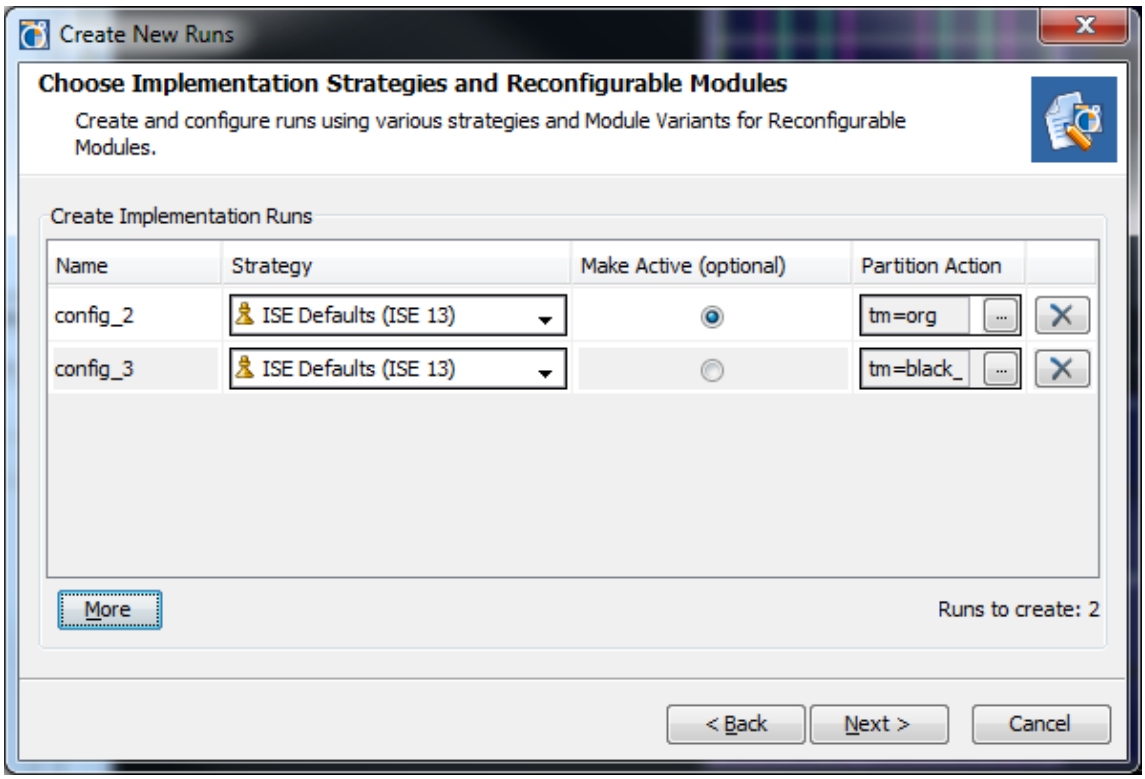
5. Select a non-black box instance of the user module (EX. org).



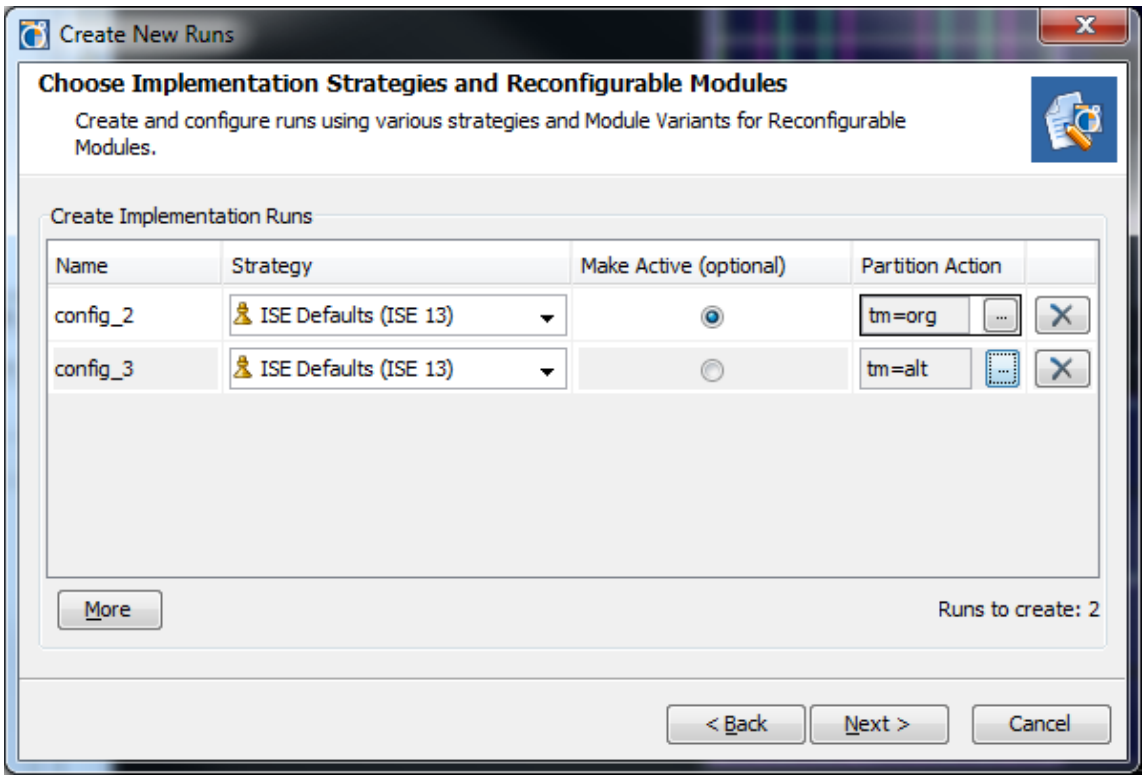
6. Click ‘OK’ button.



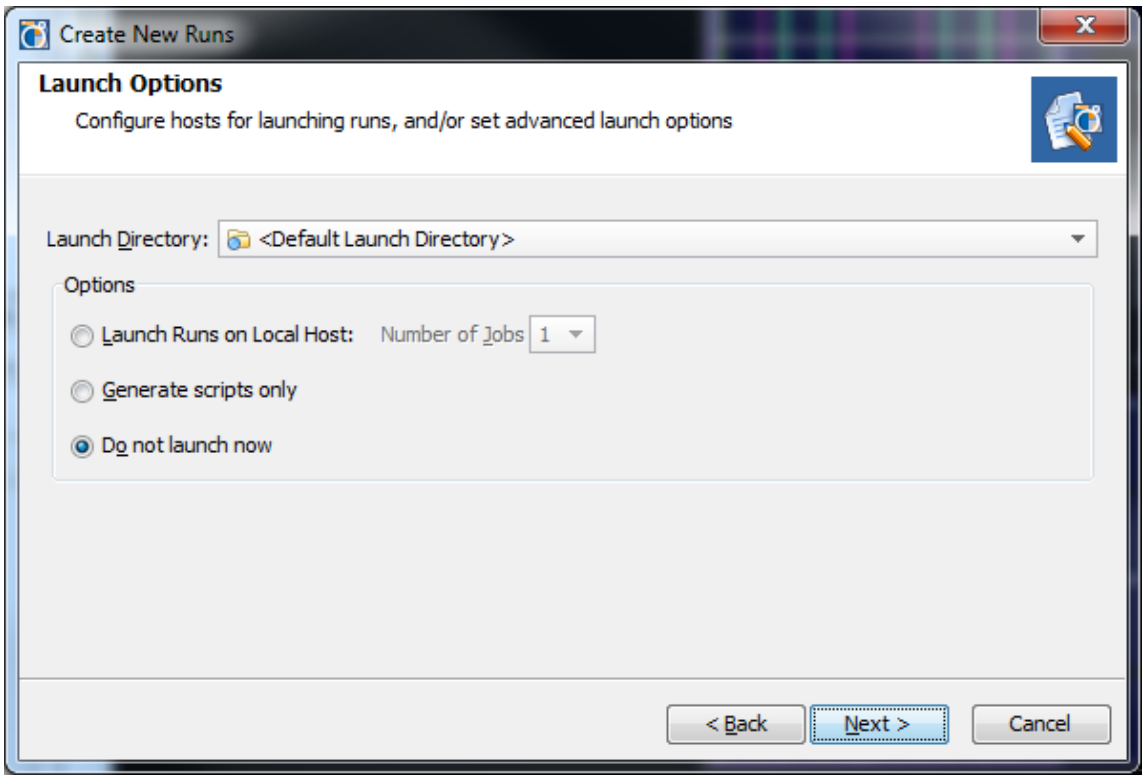
7. Click ‘More’ Button.



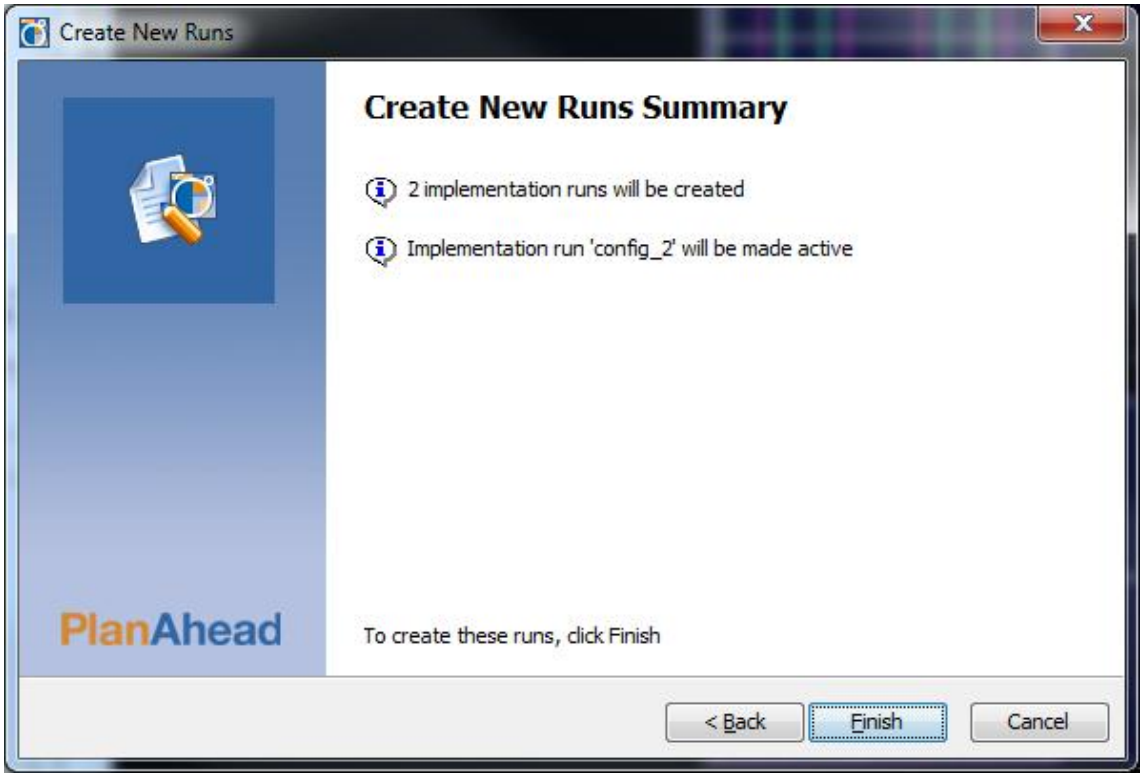
8. Repeat steps 31 through 34 for ‘alt’ instance of user module.



9. Click the ‘Next’ button.



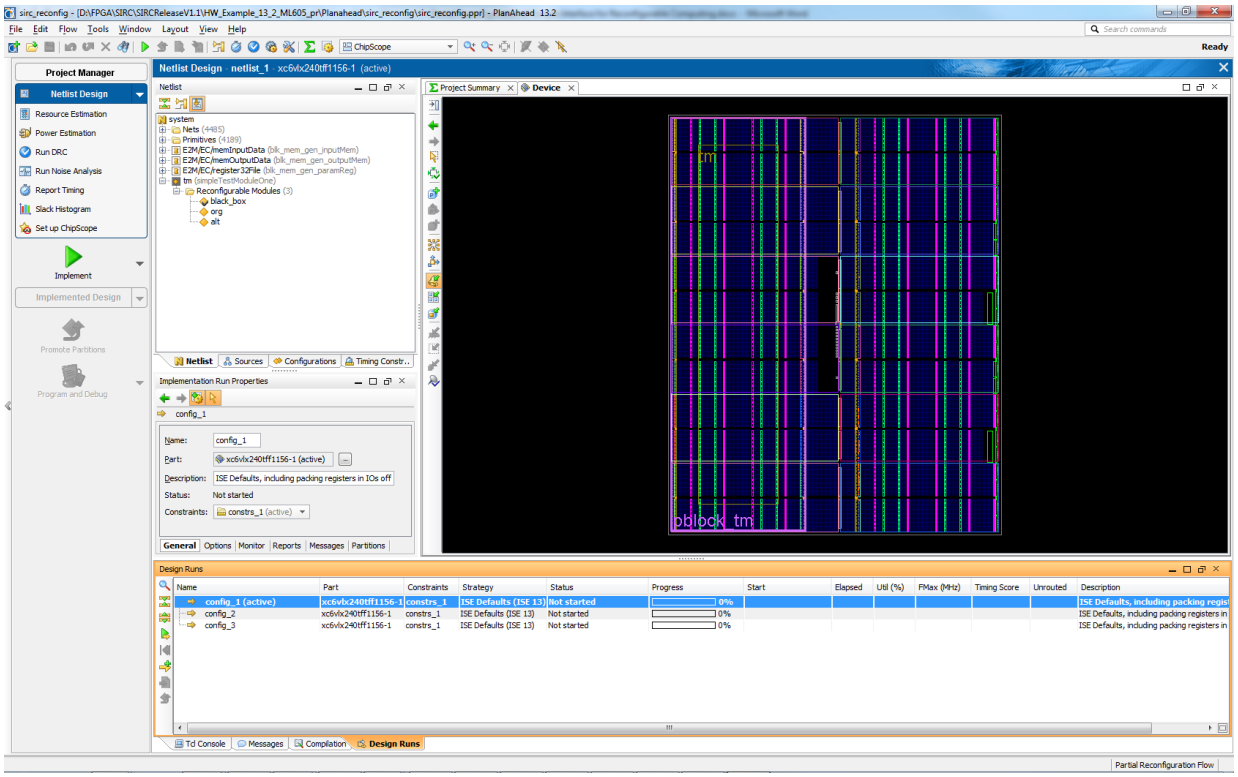
10. Click the ‘Next’ button.



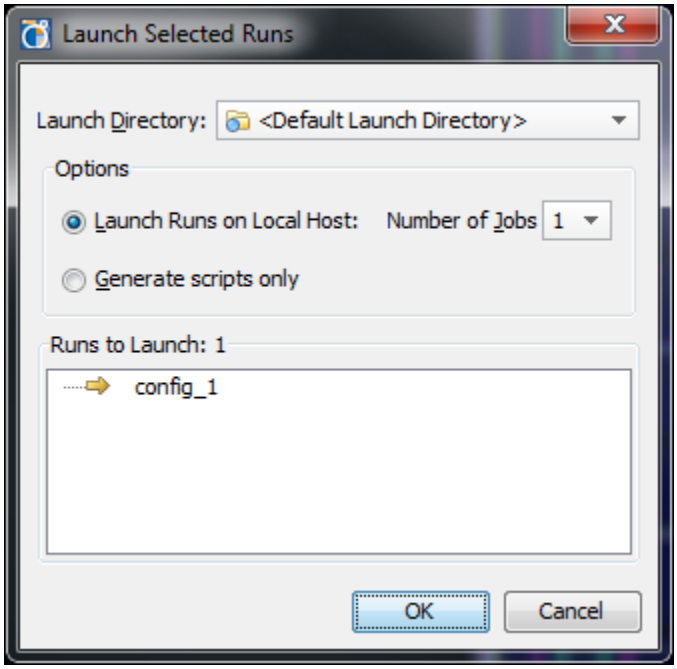
11. Review Configuration details. Click ‘Finish’ button.

Implement Designs

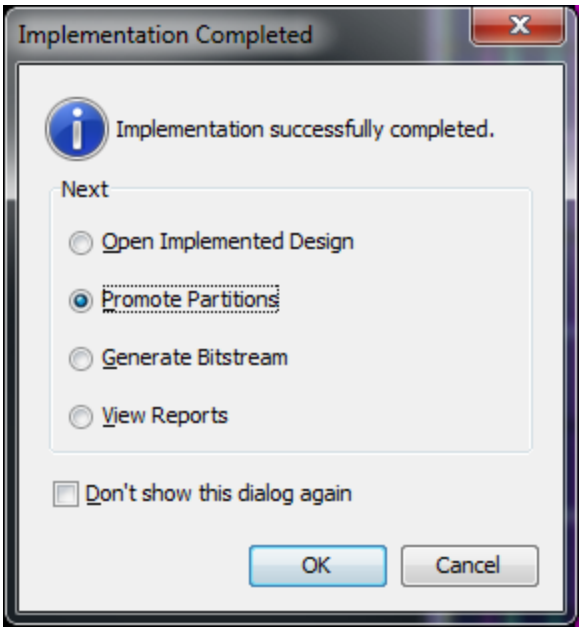
1. Right-click ‘config_1’ in the ‘Design Runs’ pane and select ‘Make Active’.



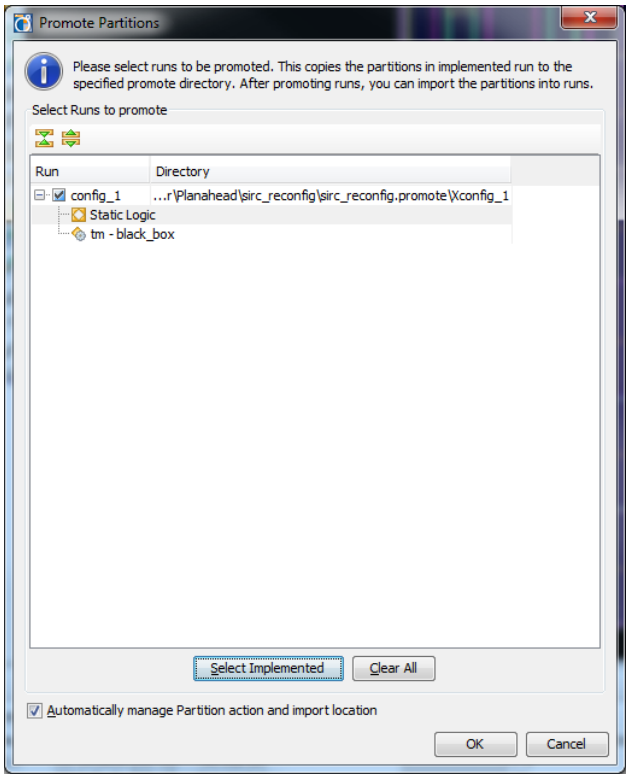
2. Right-click ‘config_1’ and select ‘Launch Runs’.



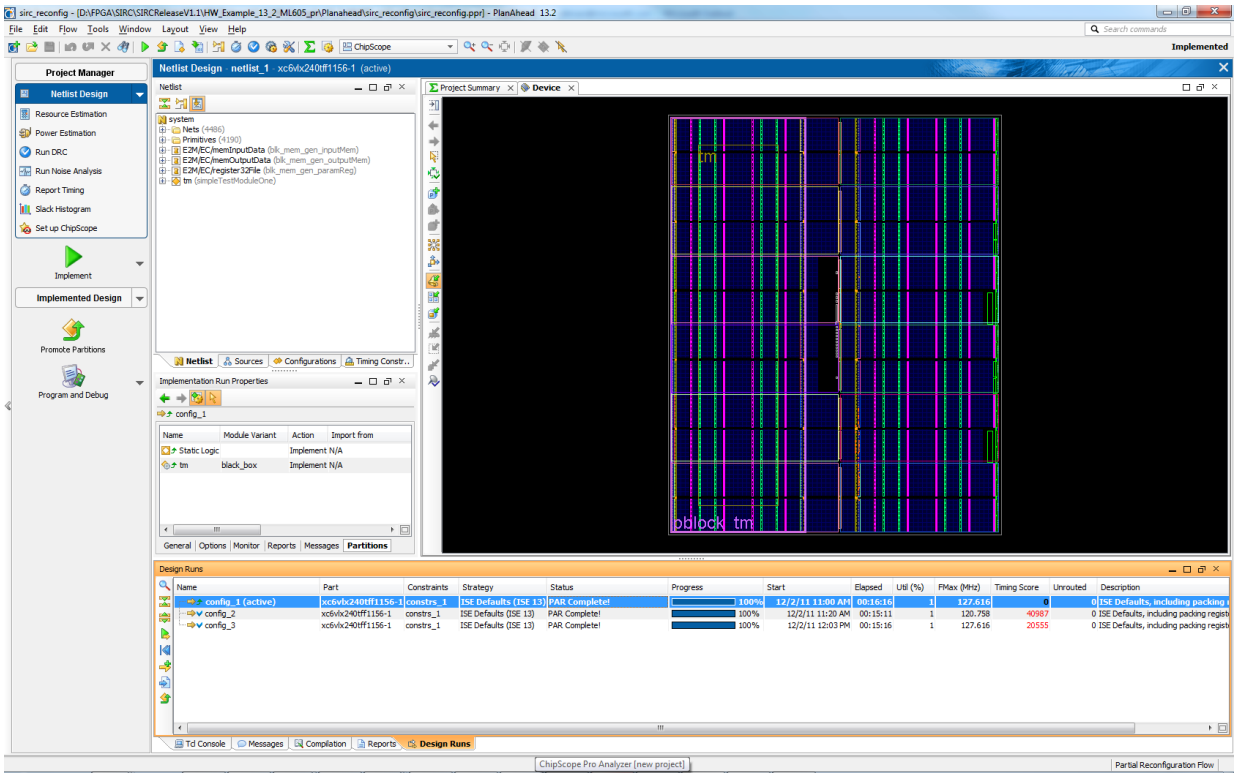
3. Click ‘OK’ button.
4. When the implementation completes, select ‘Promote Partitions’ and click ‘OK’ button.



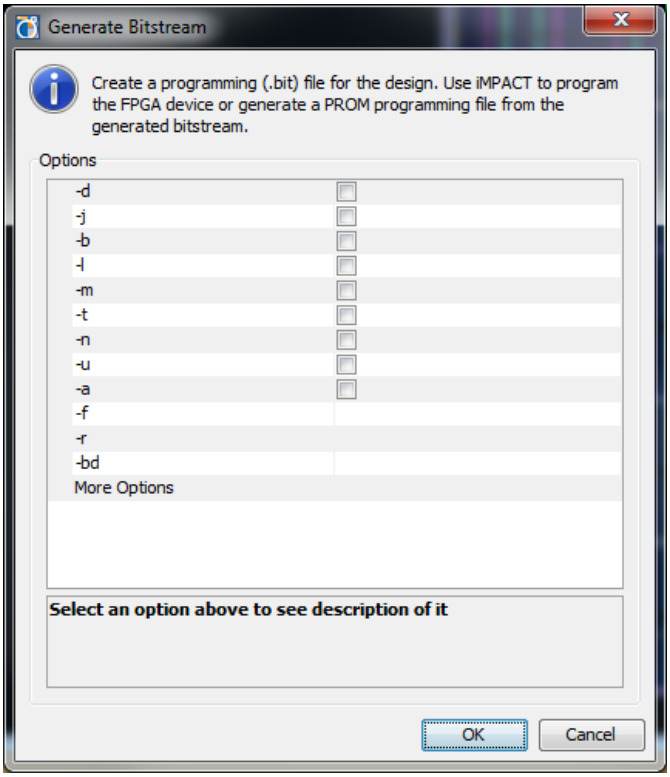
5. Click ‘OK’ button.



6. Repeat steps 1 through 5 for each Design Run Configuration i.e config_2 and config_3.

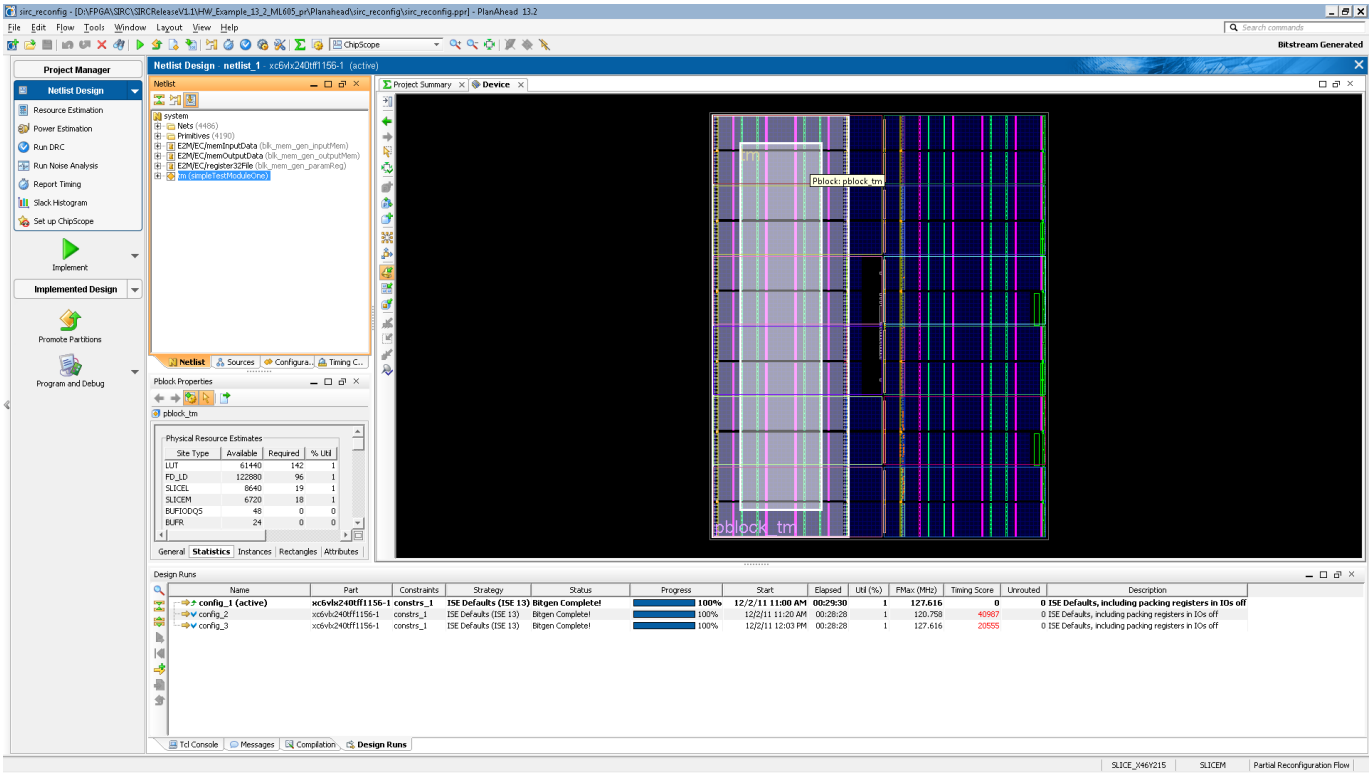


7. Right-click ‘config_1’ and select ‘Generate Bitstreams’.



8. Click the ‘OK’ button.

9. Repeat steps 7 through 8 for each configuration.



Troubleshooting

Provided you have generated all the necessary netlists for the design, PlanAhead is used to implement the design using the Xilinx Partial Reconfiguration flow. There are some potential pitfalls in this process, depending on how the user module region is allocated in the floor plan.

If you get a MAP error regarding the allocations of BUFRs between the partial reconfiguration partition and the static partition, this is due to the design rule that all the BUFRs of a clock region must belong to the same partition. To correct this adjust the bounds of your partial reconfiguration partition until you have achieved this. An alternative is to not assign any BUFRs to the partial reconfiguration partition by editing the UCF file. Simply delete the line in the AREAGROUP block for the partial reconfiguration partition. For the purposes of this example tutorial, the following AREAGROUP block may be inserted in to the UCF file in place of what PlanAhead added from the floor planning phase.

```
INST "tm" AREA_GROUP = "pblock_tm";
AREA_GROUP "pblock_tm" RANGE=SLICE_X0Y0:SLICE_X63Y239;
AREA_GROUP "pblock_tm" RANGE=BUFIODQS_X0Y0:BUFIODQS_X1Y23;
AREA_GROUP "pblock_tm" RANGE=BUFR_X0Y0:BUFR_X1Y11;
AREA_GROUP "pblock_tm" RANGE=DCI_X0Y0:DCI_X1Y5;
AREA_GROUP "pblock_tm" RANGE=DSP48_X0Y0:DSP48_X3Y95;
AREA_GROUP "pblock_tm" RANGE=IDELAYCTRL_X0Y0:IDELAYCTRL_X1Y5;
AREA_GROUP "pblock_tm" RANGE=ILOGIC_X0Y0:ILOGIC_X1Y239;
AREA_GROUP "pblock_tm" RANGE=IOB_X0Y0:IOB_X1Y239;
AREA_GROUP "pblock_tm" RANGE=IODELAY_X0Y0:IODELAY_X1Y239;
AREA_GROUP "pblock_tm" RANGE=OLOGIC_X0Y0:OLOGIC_X1Y239;
AREA_GROUP "pblock_tm" RANGE=RAMB18_X0Y0:RAMB18_X3Y95;
AREA_GROUP "pblock_tm" RANGE=RAMB36_X0Y0:RAMB36_X3Y47;
```

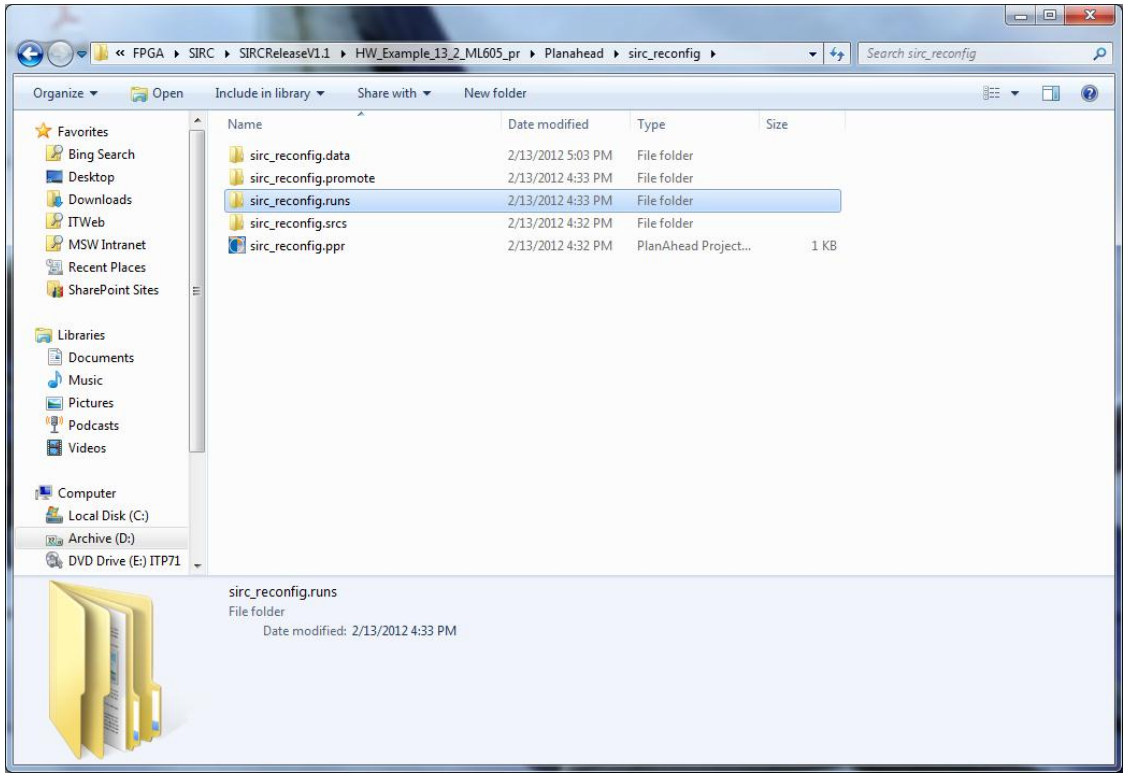
If you have a PAR error due to the CLK200 signal failing to route, this is due to a bug in the SIRC release code (as previously mentioned). The BUFG between the input buffer and the PLL is commented out and the input buffer is directly connected to the PLL. This BUFG must be replaced for the design to successfully implement using the Xilinx partial reconfiguration flow. Uncomment the BUFG and change the output of the input buffer to the input of the BUFG. Also uncomment the wire declaration for the wire between the input buffer and the BUFG. Resynthesize the static netlists and update the netlist in the PlanAhead project by going to the Sources pane and right-clicking on the 'system.ngc' file. In the menu, select 'Update File'. Follow the dialogue to import the updated netlist.

If your design fails to meet timing, you may need to reduce the clock frequency of the user circuit. If the timing violations only involve the SystemACE circuit, the violations can be ignored. Otherwise, reduce the user circuit frequency by increasing the divider of 'CLKOUT1' of 'clkBPLL' on line 124 of 'system.v'. Resynthesize the static netlists and update the netlist in the PlanAhead project by going to the Sources pane and right-clicking on the 'system.ngc' file. In the menu, select 'Update File'. Follow the dialogue to import the updated netlist.

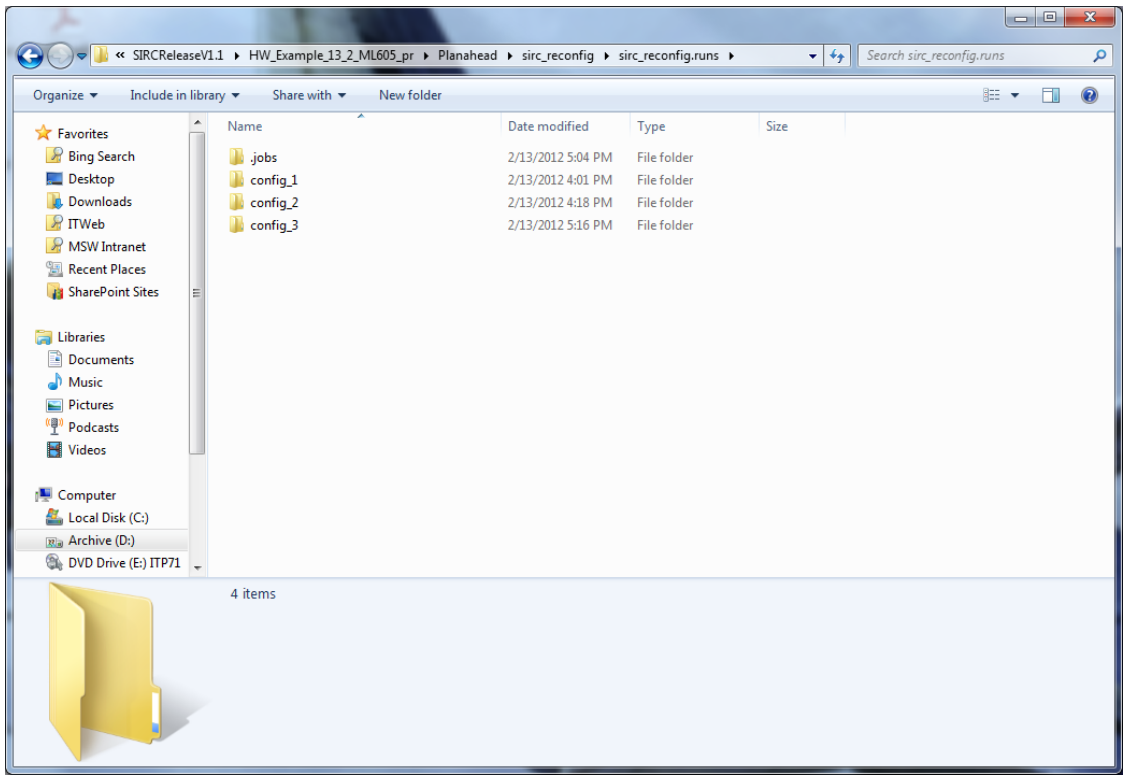
With the netlists updated with the appropriate fixes, repeat the steps in 'Implement Designs'.

Bitstreams

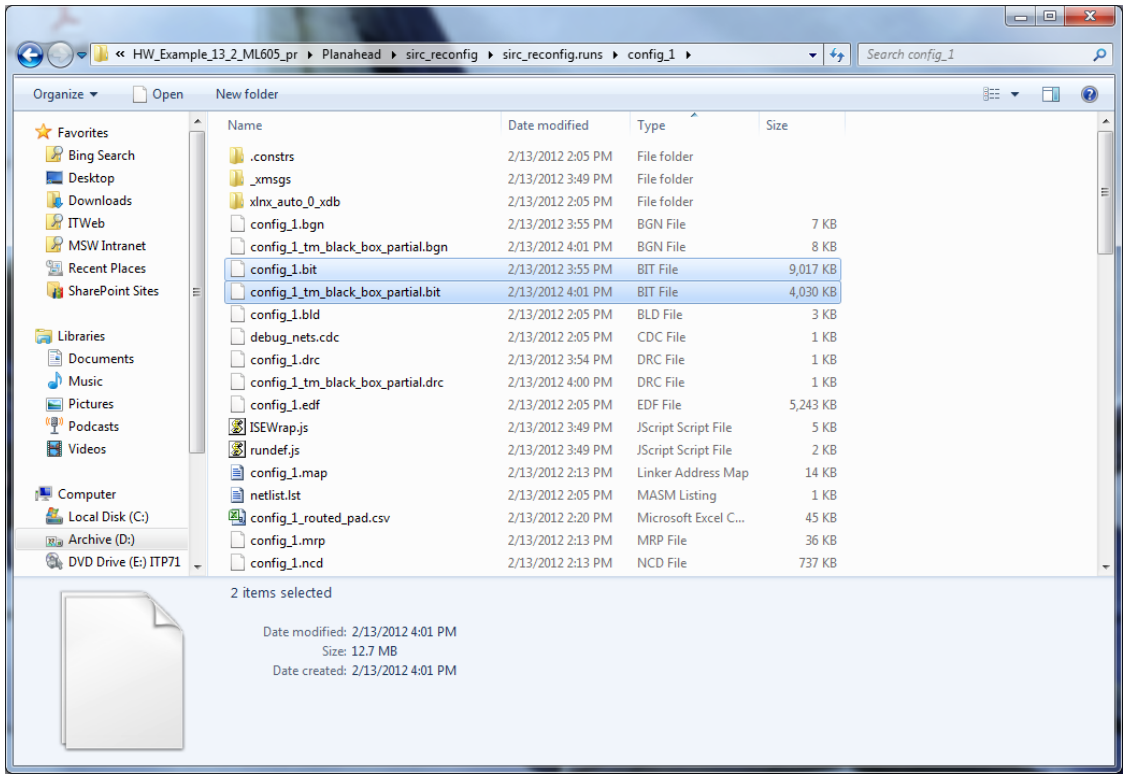
By now, you should have generated six bitstreams using PlanAhead and Xilinx Partial Reconfiguration flow. Of these, three are full bitstreams and three are partial bitstreams. The difference between these is that the full bitstreams include the static region configuration and the partial bitstreams do not. The partial bitstreams may only be used after one of the full bitstreams has been applied.



The bitstream are located in the ‘runs’ directory of your project directory.



Inside the ‘run’ directory is a directory for each of the configurations you implemented in PlanAhead: config_1 with the empty black box partition, config_2 with the original user module and config_3 with the alternate user module.



Each directory contains the build data for that configuration of the system including the two bitstreams generated for that configuration, one full bitstream and one partial bitstream.

Each full bitstream includes the configuration of the static region and version of the partial reconfiguration partition. The partial bitstreams contain only the configuration of a given version of the partial reconfiguration partition. The following table summarizes the contents of the bitstreams.

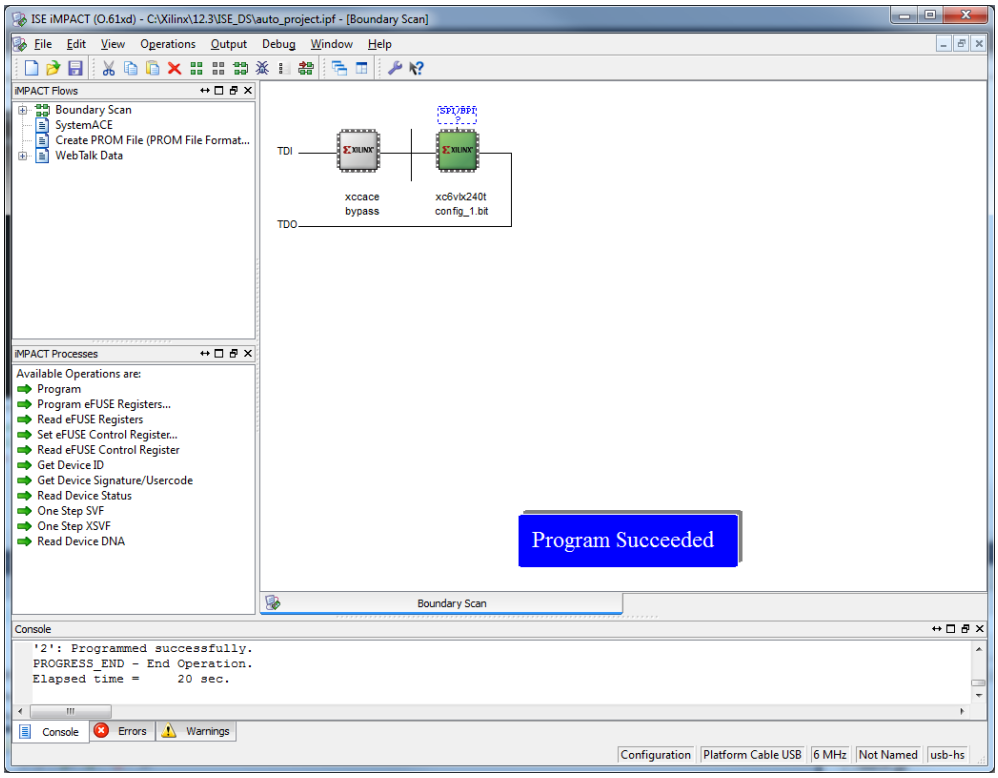
Bitstream Name		Contents
config_1.bit	Full Bitstream	Static region and black box user module
config_1_tm_black_box_partial.bit	Partial Bitstream	Black box user module (blanking bitstream)
config_2.bit	Full Bitstream	Static Region and original user module
config_2_tm_org_partial.bit	Partial Bitstream	Original user module
config_3.bit	Full Bitstream	Static Region and alternate user module
config_3_tm_alt_partial.bit	Partial Bitstream	Alternate user module

After the FPGA is powered up it must be configured with one of the full bitstreams that include the static region. After the full bitstream is applied the static region will be operational with whatever user module was included. After that any partial bitstream may be used to change the functionality of the partial reconfiguration partition while the static region remains operational. The 'config_1_tm_black_box_partial.bit' is a special case bitstream since it configures the partial reconfiguration partition to a black box state. This called a blanking bitstream. This is used to remove any configuration from the partial reconfiguration partition and leave it in an unconfigured state when it is not in use. This can have considerable power saving implications if the partial reconfiguration partition is to be inactive for a significant amount of time. It is not necessary to blank the partial reconfiguration partition before applying a new user configuration.

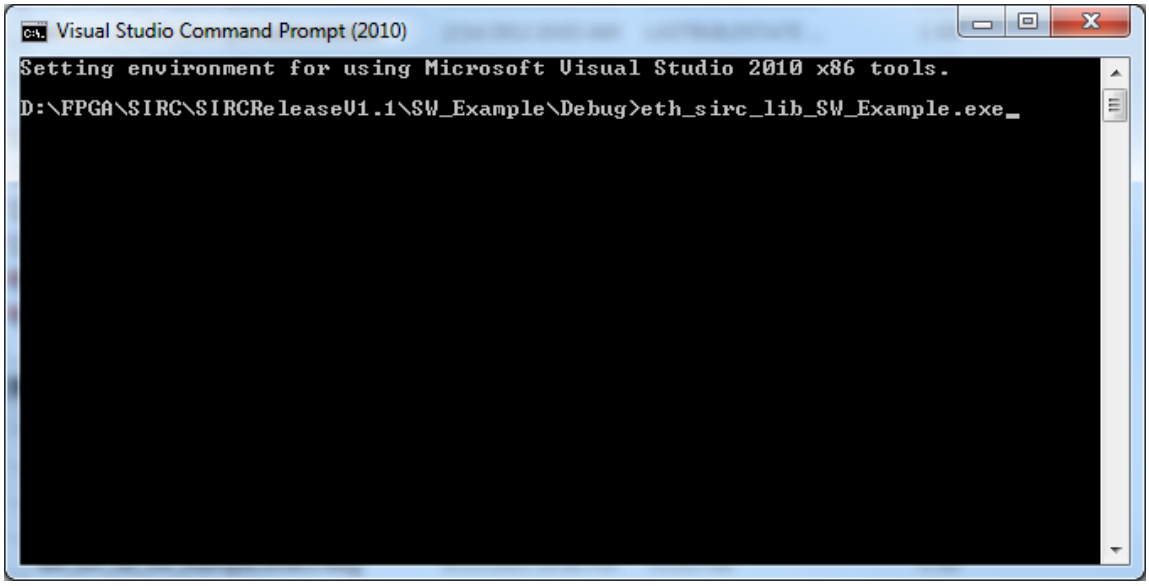
Testing

The following is a procedure for verify the functionality of these bitstream configuration files.

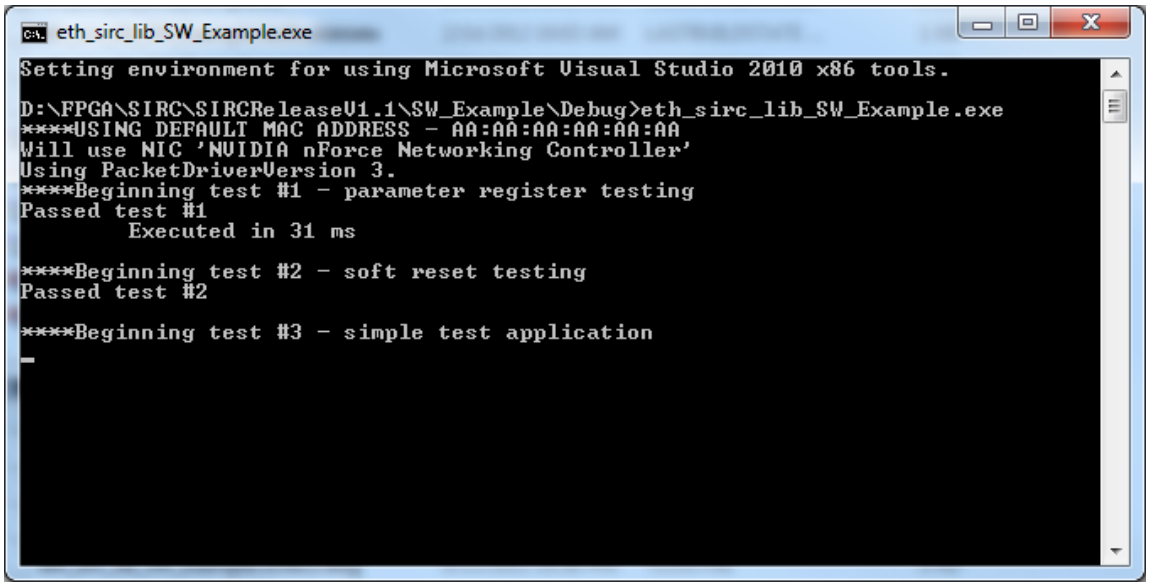
1. Build the software example project provided in the SIRC release.
2. Connect the FPGA to the host machine per the instructions in the SIRC release.
3. Using Impact, configure the FPGA with the bitstream config_1.bit, or the full bitstream containing the back box instance of the user module.



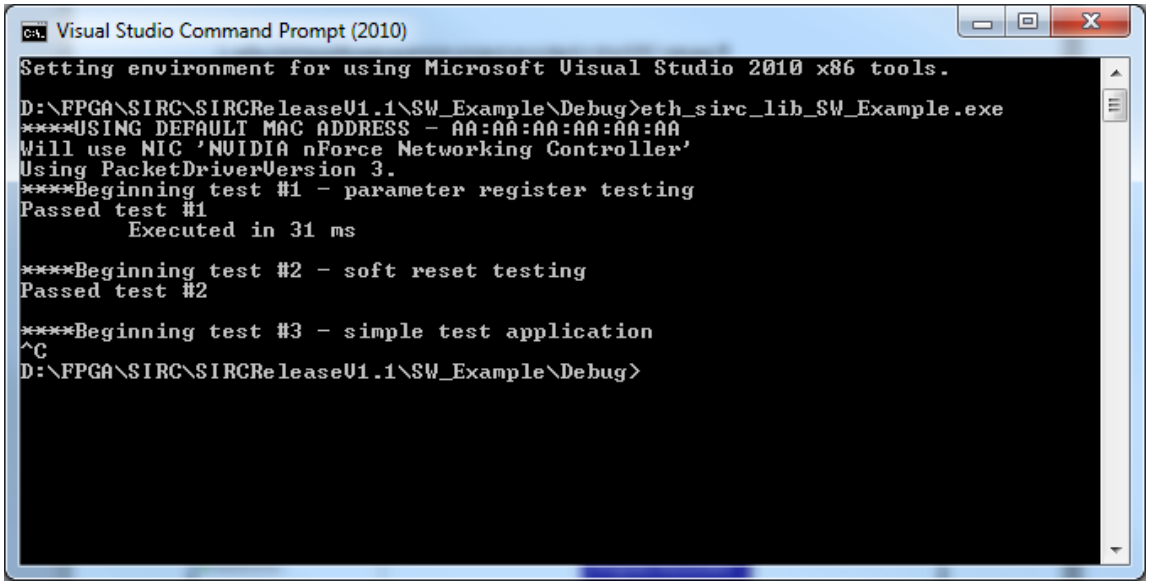
4. Open a command prompt in the build directory.
5. Run the software example by typing ‘eth_sirc_lib_SW_Example.exe’ into the command prompt.



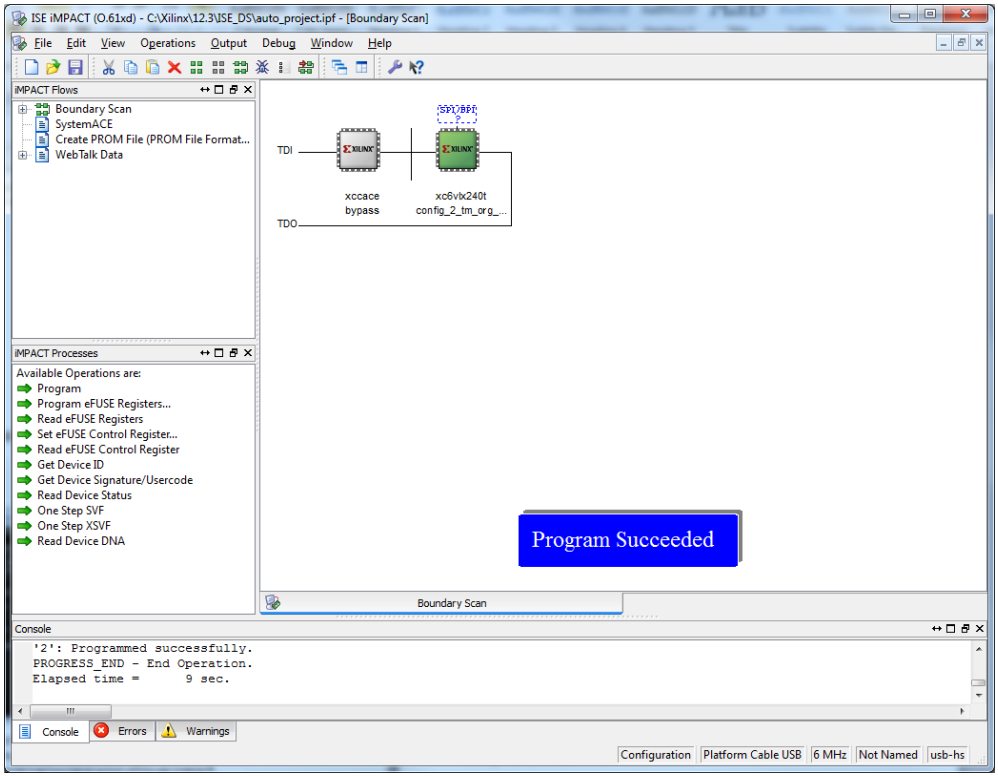
6. The application will hang because the partial reconfiguration region is empty.



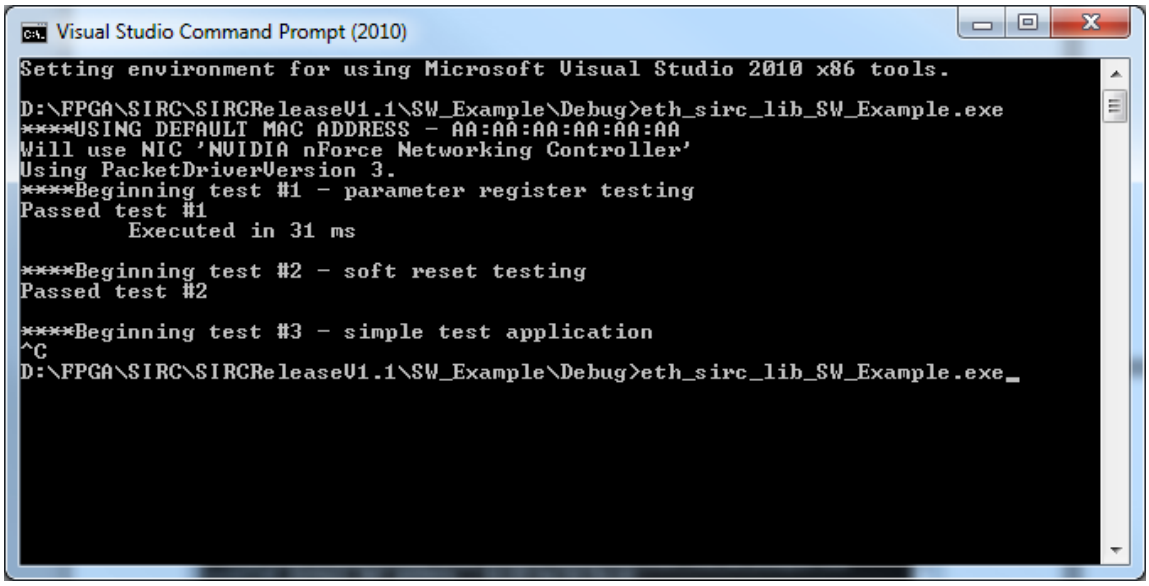
7. Type Ctrl-C to exit the application.



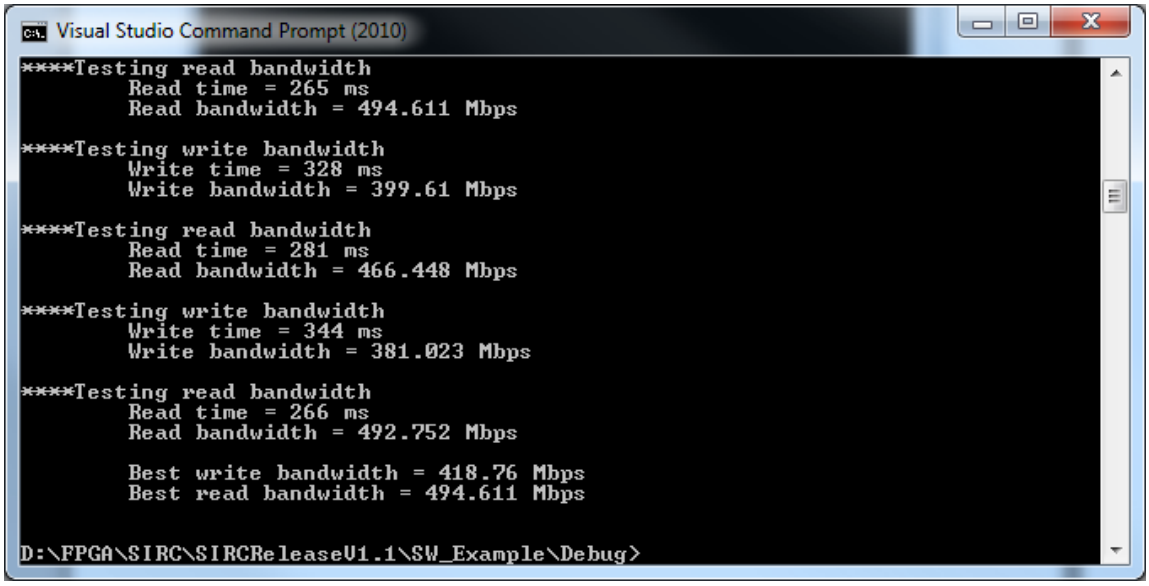
8. Press the CPU RESET button on the ML605 Evaluation Board
9. Using Impact, configure the FPGA with the partial bitstream config_2_tm_org_partial.bit, meaning the partial bitstream containing the original version of the user module.



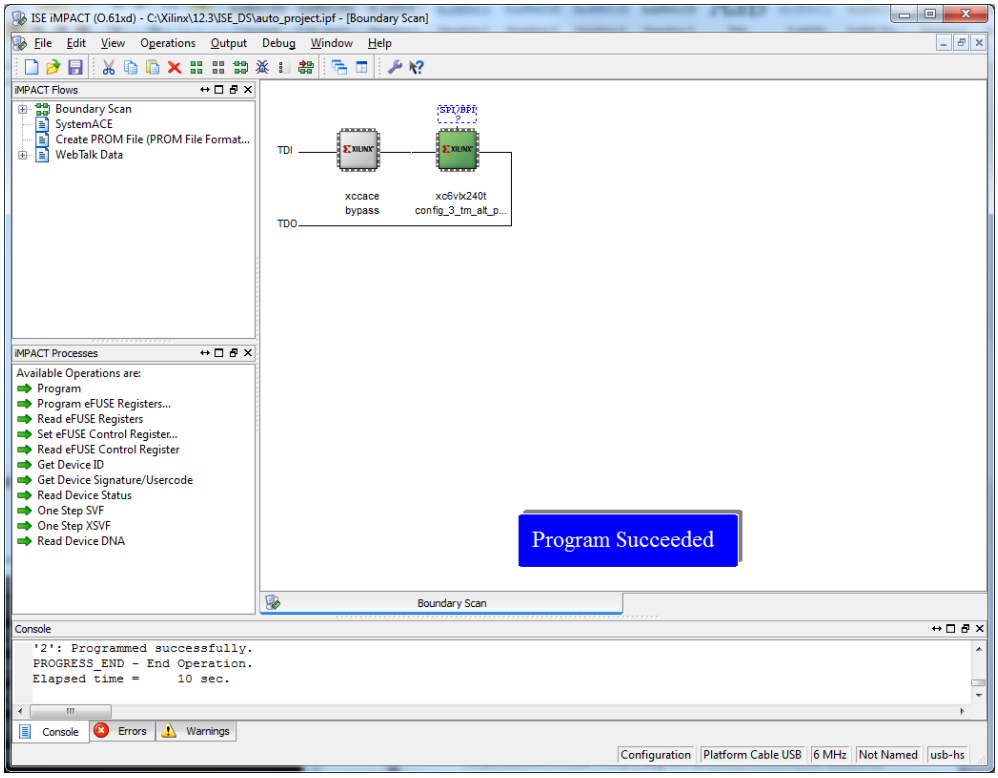
10. Run the software example by typing 'eth_sirc_lib_SW_Example.exe' into the command prompt.



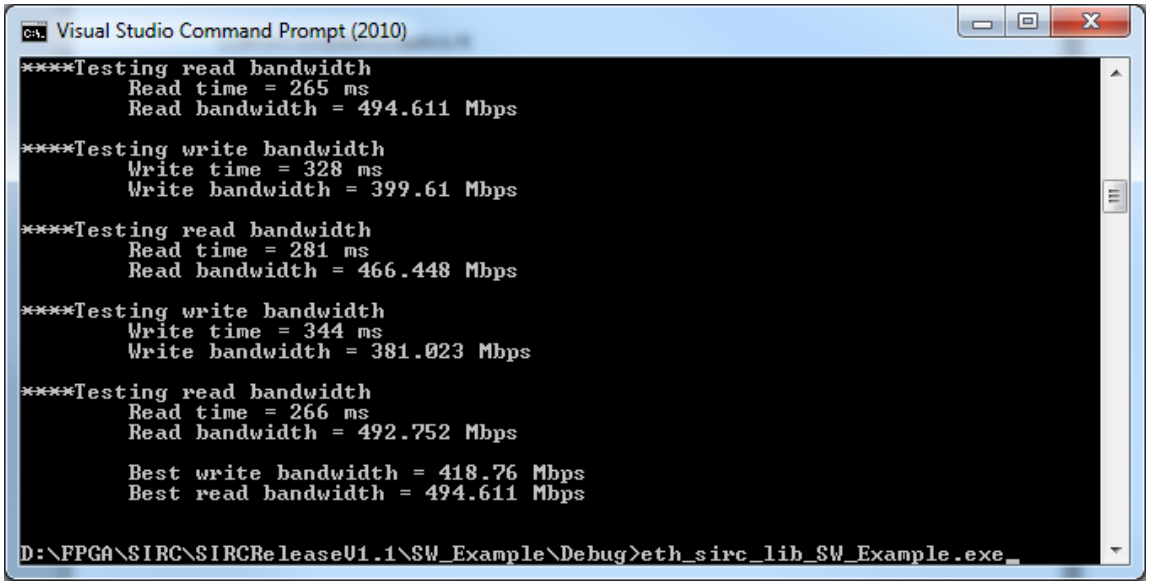
11. It will complete successfully.



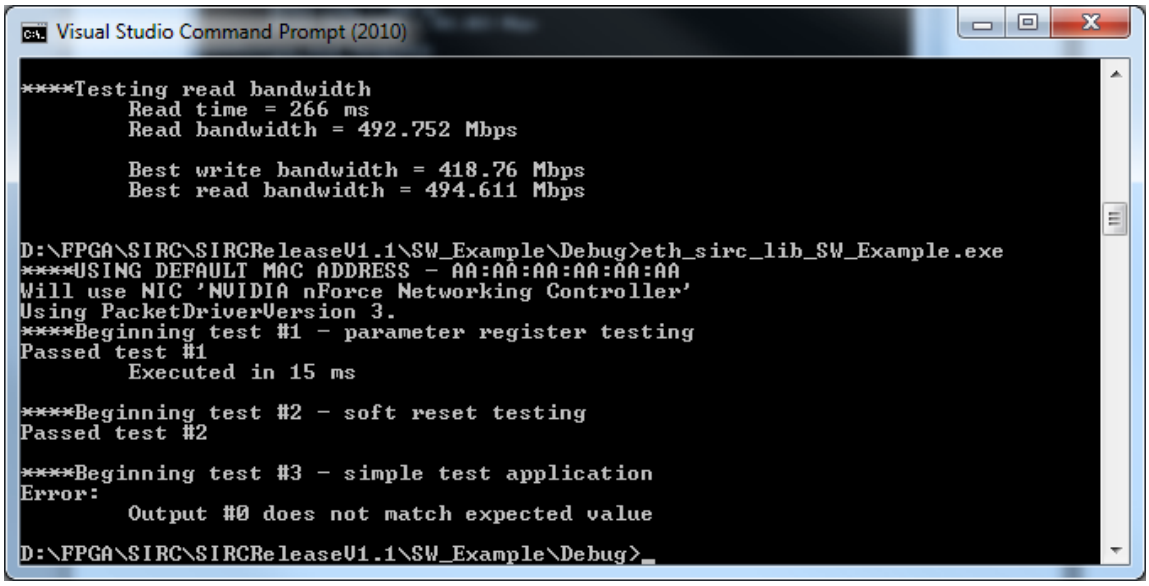
12. Using Impact, configure the FPGA with the partial bitstream config_3_tm_alt_partial.bit, meaning the partial bitstream containing the alternate version of the user module.



13. Run the software example by typing ‘eth_sirc_lib_SW_Example.exe’ into the command prompt.



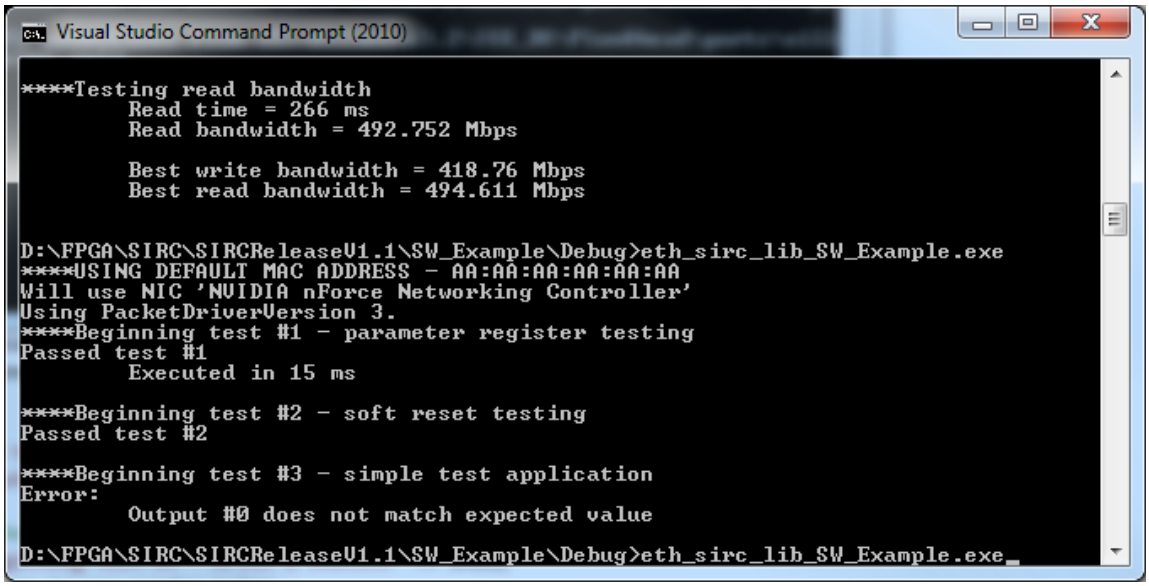
14. It will fail on the first output.



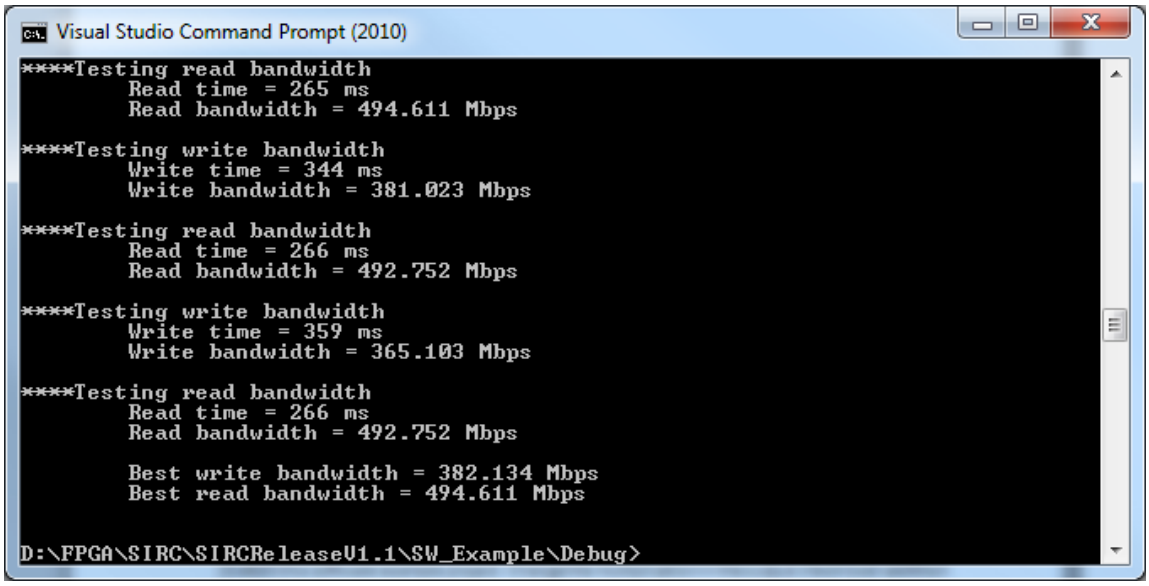
15. Edit the software example project. Change the multiplication in the output check to an addition to reflect the change in the alternate version of the user module on lines 363, 421 and 482.

```
if((inputValues[i] * 3) % 256 != outputValues[i]){
```
→
```
if((inputValues[i] + 3) % 256 != outputValues[i]){
```

16. Rebuild the software example project with the changes.
17. Run the software example by typing 'eth_sirc_lib_SW_Example.exe' into the command prompt.



18. It will complete successfully.



This procedure confirms that the full and partial reconfiguration bitstreams work as expected. Using the partial bitstreams, we have shown that it is possible to change the functionality of the user module while the SIRC Hardware/Software API remains active. It is now possible for you to write and generate bitstreams for your own user modules that can be configured on top of the existing static implementation of SIRC in this tutorial.

Advanced Materials

This concludes the tutorial. You can now program the FPGA with the static bitstream and use the partial bitstreams to dynamically change the functionality of the user circuit and communicate between software and this circuit using SIRC. In this section we will present some more advanced material, such as practical limitations, and debugging of the PR circuits which is currently not very well documented by Xilinx.

Preparing the Design for Partial Reconfiguration

Before using the partial reconfiguration design flow to implement a complex system, we recommend that you have already debugged and tested each instance or reconfigurable state of your system as much as possible, using the standard flow. The partial reconfiguration design flow itself can introduce new problems that can be difficult to diagnose and correct if bugs remain in the base system. These problems include but are not limited to errors in the design hierarchy, timing errors resulting from floor planning constraints and lost optimization, and under specified reset and metastate conditions introduced by runtime reconfiguration.

The first step in creating a system using partial reconfiguration from an existing static system, is selecting what parts of the system you think should be able to reconfigure dynamically. When making this decision, it is required that the functionality that you want to reconfigured be encapsulated within a module. To simplify design and maintenance, functions that are intended to be changed together should whenever possible be wrapped within the same higher level model to minimize the number of independently reconfigurable partitions in the design. The complexity of the flow significantly increases with additional partitions.

The Xilinx partial reconfiguration design flow imposes some restrictions on the contents of modules and partitions meant to be reconfigured dynamically [4]. Most but not all components available within the FPGA fabric can be included in a partial reconfiguration partition. Most of these exceptions are related to clocking resources. All global clocking resources and those architectural features that are used to modify clock signals cannot be included in a partial reconfiguration partition. These include BUFG, MMCM, PLL and DCMs. Additionally architectural features related to configuration and FPGA state such as BSCAN, STARTUP, ICAP, CAPTURE, DCIRESET, FRAME_ECC, KEY_CLEAR, USR_ACCESS, etc. cannot be included in the partial reconfiguration partition. Additionally, any IP core that makes use of these components may not be included in a partial reconfiguration partition. This includes some EDK blocks and MIG. Under normal circumstances this would also preclude the use of Chipscope for debugging within the partial reconfiguration partition because of the use of BUFGs and BSCAN components. However we have a work around that will be presented later in this tutorial.

The Xilinx partial reconfiguration design flow also imposes restrictions on the interface to the partial reconfiguration partition. Bidirectional ports on the interface between the partial reconfiguration partition and the remaining design are not allowed. If the design makes the use of such ports they will need to be separated into dedicated input and output ports and recombined into bidirectional ports higher in the hierarchy. [4]

One major difference between a full configuration and a partial reconfiguration of the FPGA is that there is no global reset of the logic after the configuration is complete. Thus the state of the logic in the partial reconfiguration partition is unknown immediately following the completion of the reconfiguration process. For this reason, all logic within the partial reconfiguration partition must be initialized by a reset state and the partial reconfiguration partition should be reset after the reconfiguration process and prior to resuming operation. Similarly, the state of partial reconfiguration partition outputs will be of unknown state during the reconfiguration process. This process can take a long period of clock cycles depending on the size partial reconfiguration partition and bandwidth of the reconfiguration channel. For this reason, it is important to place the system into a safe state where the outputs of the partial reconfiguration partition are ignored until the reconfiguration is complete and the partition is reset. However, this does not preclude the remainder of the FPGA from operating on other tasks that do not interact with the partial reconfiguration partition. [4]

The partial reconfiguration partition will be synthesized and implemented independently of the rest of the design. This means that some optimizations that the tools took advantage of when the design was flat will no longer apply now that these pieces are separated. In addition when you floor plan the design

later in the tutorial, you are imposing additional restrictions on the placement and routing of the design. This in some cases is enough to push the timing of certain signals out of compliance and result in timing errors. For this reason, additional pipeline stages as well as registering the inputs and output of partial reconfiguration partitions is recommended to improve slack in the timing whenever possible. Xilinx recommends expecting a 10% performance loss in clock frequency and expecting to not exceed 80% of the FPGA resources. [4]

A big question when implementing a partial reconfiguration system relates to the floor planning. In some cases, partial reconfiguration partitions will interact with fixed elements in the FPGA such as IO pins and other features. It is possible to use these as anchors to guide the placement of the partial reconfiguration partition in the floor plan. But what should be done when such anchors do not exist? A good starting point is the original design implemented in the standard flow. If the standard flow placed the logic in a certain area it is possible to constrain the partial reconfiguration to do the same, which will provide similar results.

To view the location of a module on the FPGA in the design, locate the routed netlists or *.ncd file for the design. Open that file in Xilinx FPGA Editor. Then search for the nets and components associated with the module of the selected partial reconfiguration partition and highlight them. You will see a mass of logic somewhere in the FPGA. If you are lucky, the mass will be fairly localized and you can aim to place the region constraint for your partial reconfiguration centered on that mass. However, if the mass is more widespread this may indicate that when partial reconfiguration flow is used timing and other implementation issues may arise. In this case, more effort may be needed to better decouple the partial reconfiguration partition from the design. Timing errors that surface later can also be dealt with using timing constraints on the using partition pins of the partial reconfiguration partition.

Debugging Partial Reconfiguration Designs with Chipscope

Chipscope is the on-chip debugging solution for Xilinx FPGAs. Using the Chipscope tool you can probe signals of the design under development within the FPGA. The signal values are sampled each cycle of a reference clock and stored in a provided blockram. Sampling begins when user defined trigger conditions are satisfied until the allocated blockrams are filled. When the blockrams are filled the captured data is sent to the host PC over the JTAG. Chipscope is well supported and partially automated for non-partial reconfigurable designs, whereas using it with partial reconfiguration is not too well supported. However, with some effort and knowledge it is possible to use Chipscope to debug designs using partial reconfiguration. It is recommended though to do as much debugging and verification on a non-partial reconfiguration version before moving to the partial reconfiguration flow.

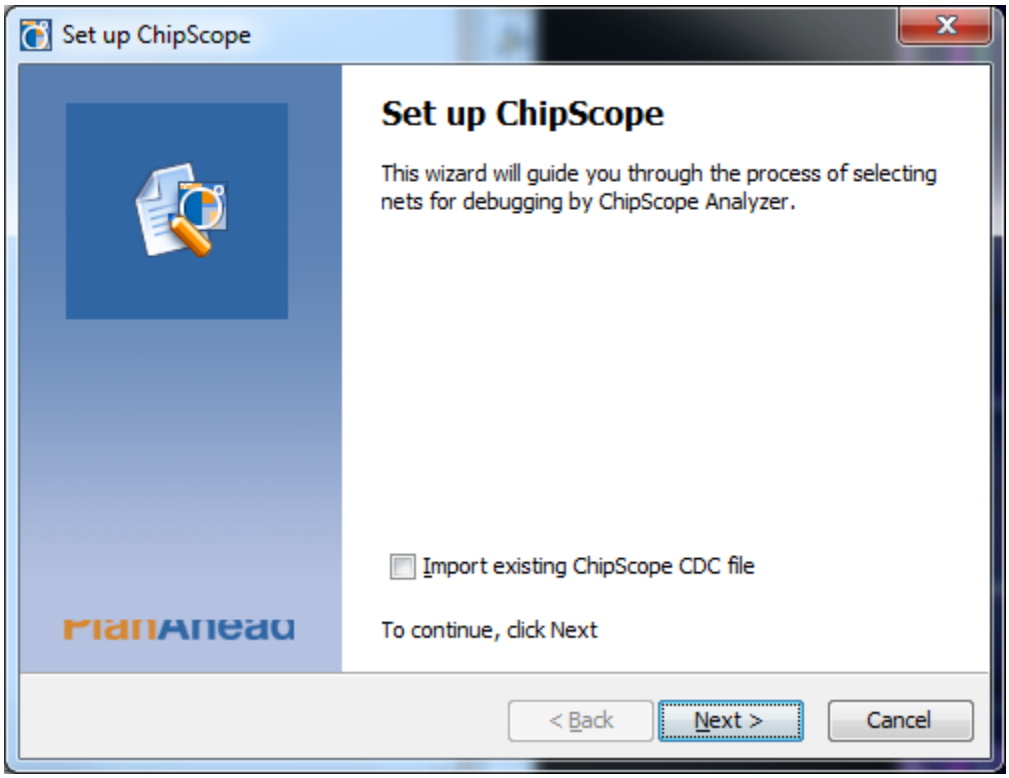
Integrated Chipscope Support

Xilinx PlanAhead provides integrated support for setting up Chipscope probes for debugging. This works similarly to the GUI based interface used in the ISE Project Navigator and carries the same advantages and disadvantages. The user selects the signals to probe and the tools add the appropriate logic to the

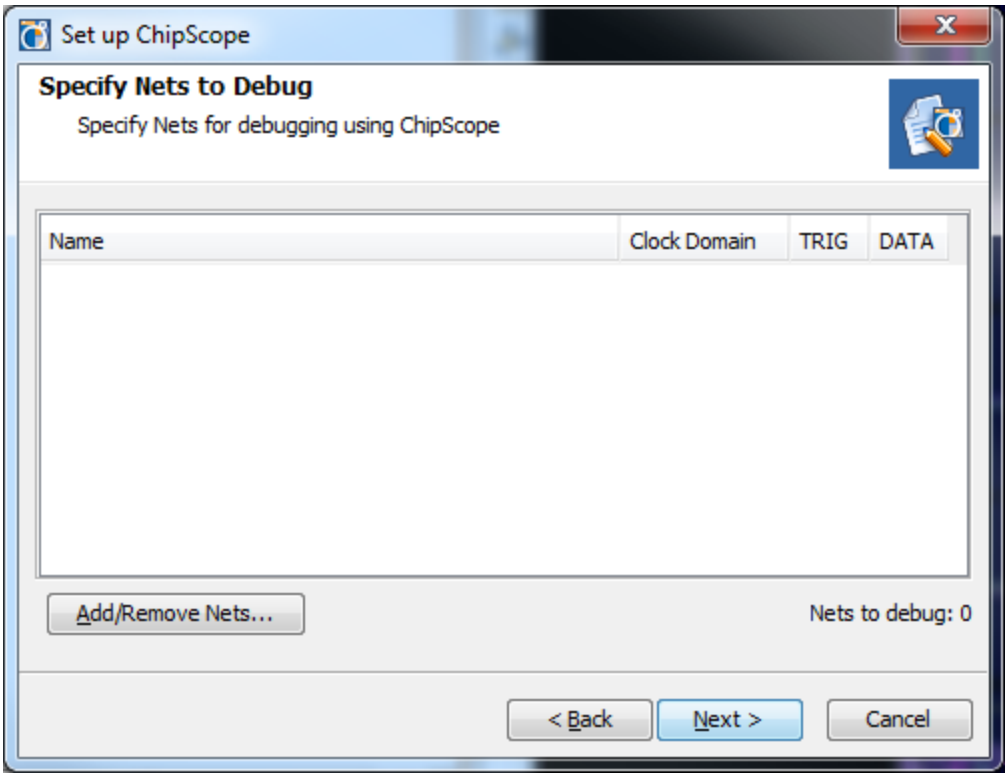
design after synthesis during design translation. This makes adding debugging logic easier assuming the signals required to diagnose the bug are understood. This effectively changes the design and how it implements. After the probe logic is selected and added the design must be mapped, placed and routed again. A debug cycle can take 15 minutes to several hours depending on the design complexity.

There is an additional issue when applying it to partial reconfiguration designs. The integrated Chipscope support can only probe signals accessible from the static region of the design. Signals that are contained within the partial reconfiguration partition cannot be probed by Chipscope in this way unless you change the design sources to output those signals from the partial reconfiguration partition.

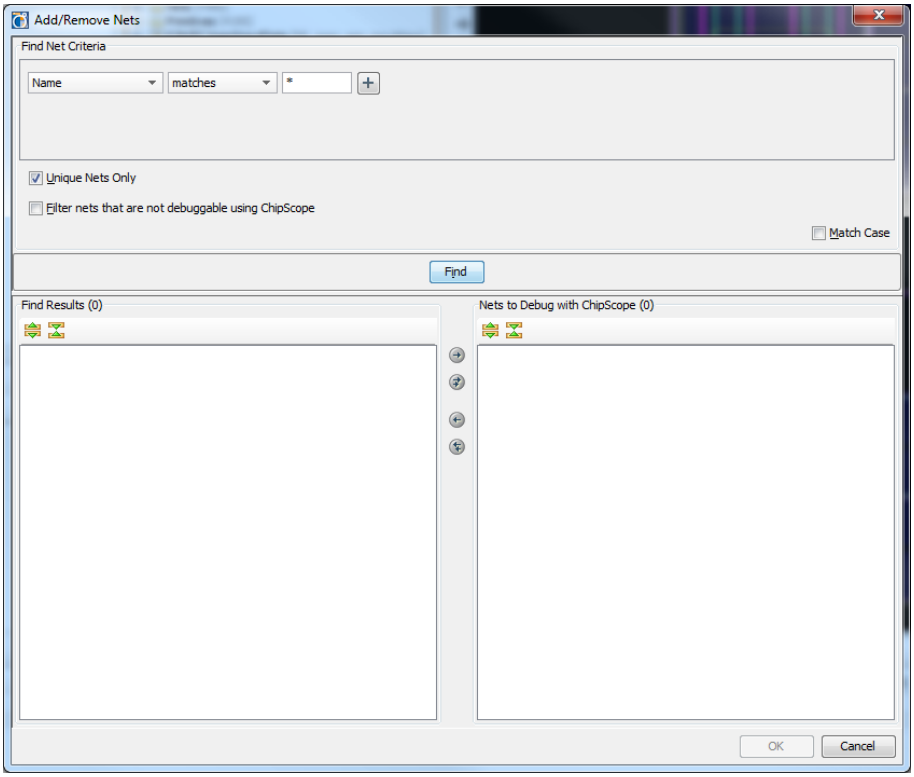
1. In the 'Project Manager Pane' click 'Set up Chipscope'.



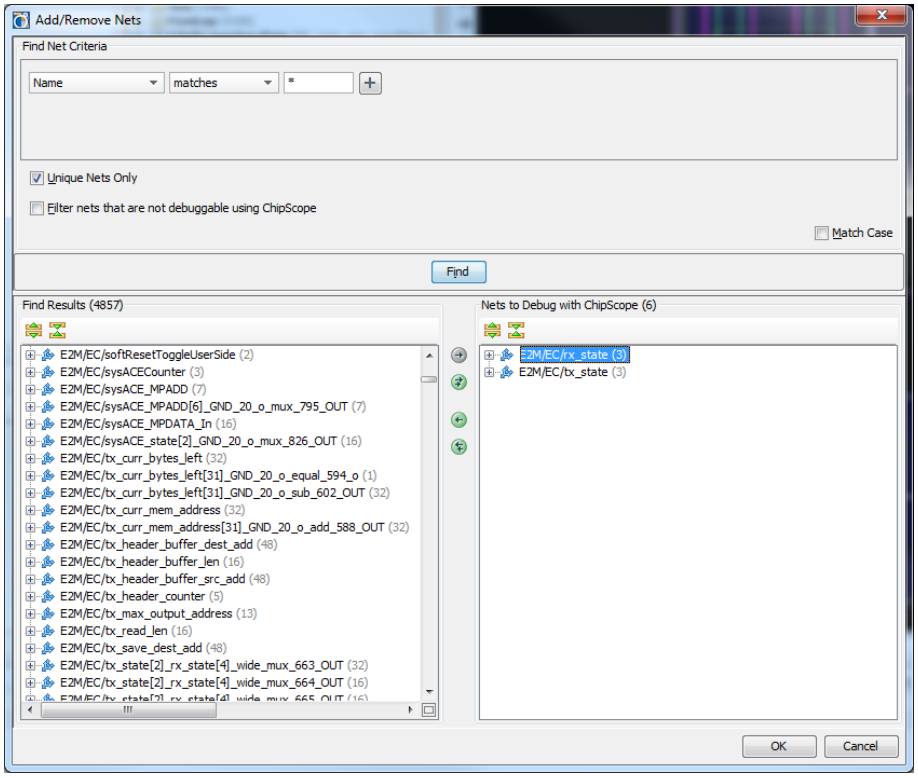
2. Here you can import a Chipscope CDC file from your debugging in the standard flow so long as it does not contain signals of the user module in the partial reconfiguration partition. In this example, we will proceed manually. Click the 'Next' button.



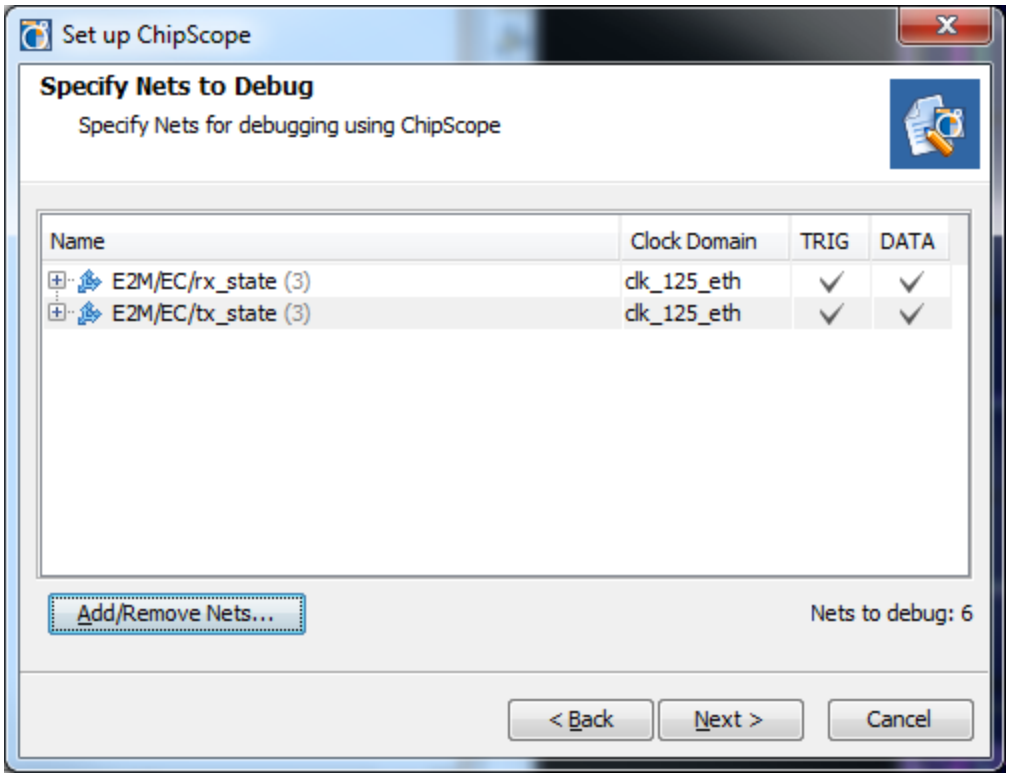
3. Click the ‘Add/Remove Nets’ button.



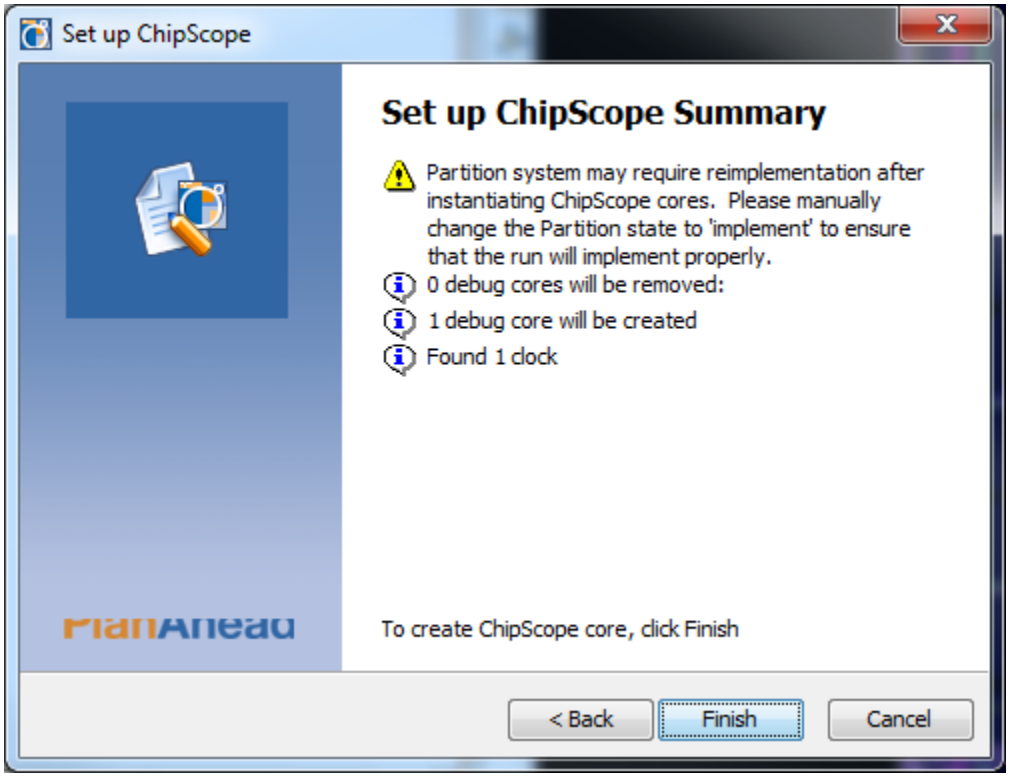
4. Here you may search and select signals for debugging from the static region. We might select, for instance the TX and RX states of the SIRC interface.



5. When you are done selecting signals click the ‘OK’ button.



6. Click the ‘Next’ button.

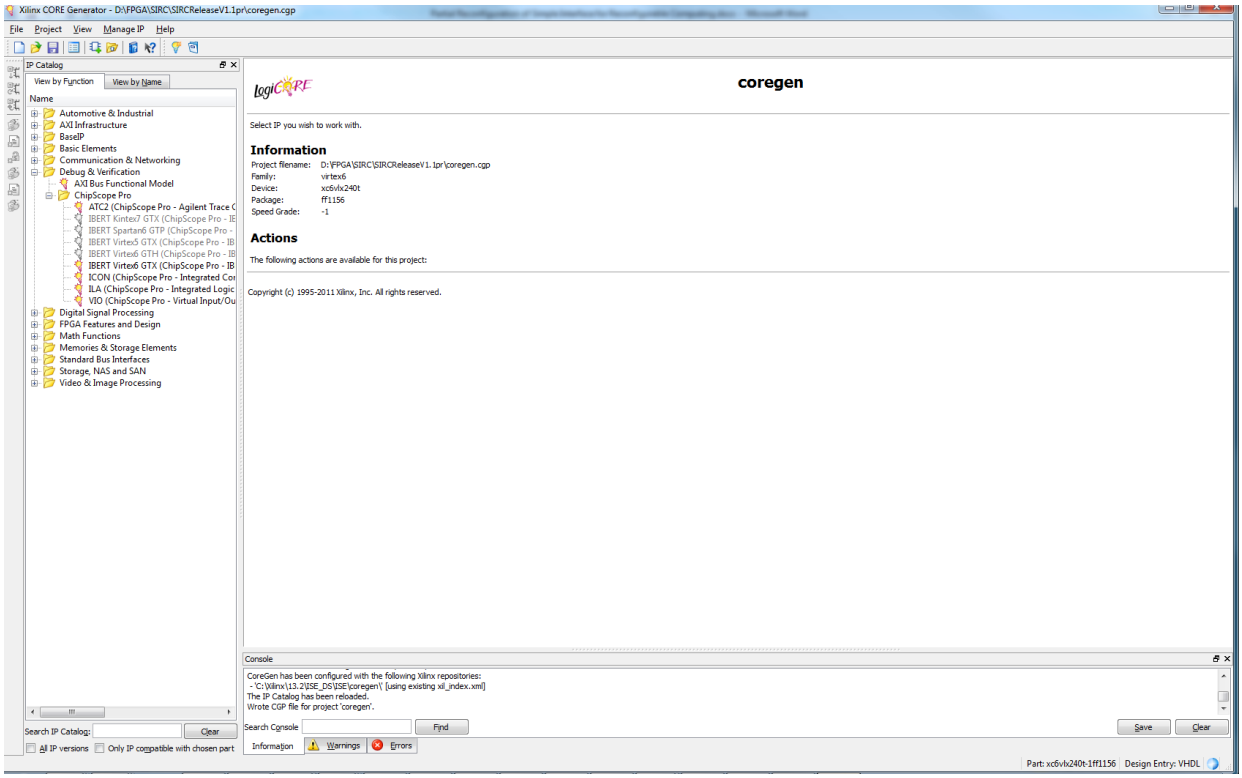


7. Review the summary and click the ‘Finish’ button.

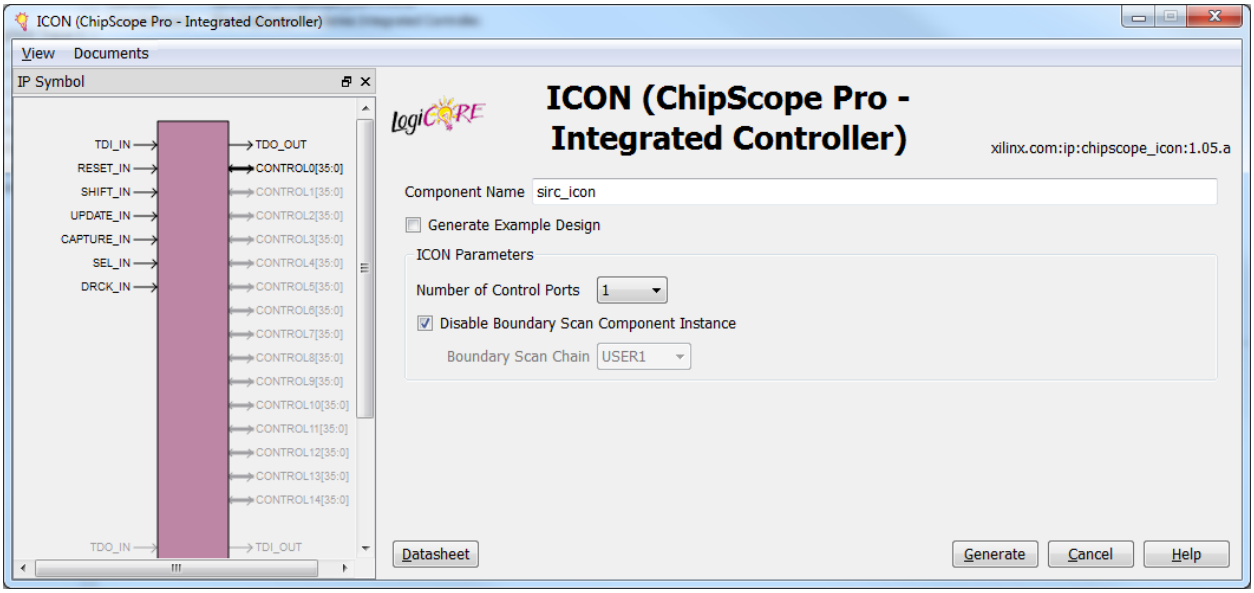
Manually Generating Chipscope IP Cores using CoreGen.[2]

The integrated Chipscope support has its advantages since it automates much of the process for inserting the debugging infrastructure. However, it also has its limitations. It is possible to use the debugging capabilities of Chipscope to greater efficiency by manually constructing the debugging infrastructure in your design. The first step in this is to generate the Chipscope cores required in Xilinx CoreGen for later instantiation in your design.

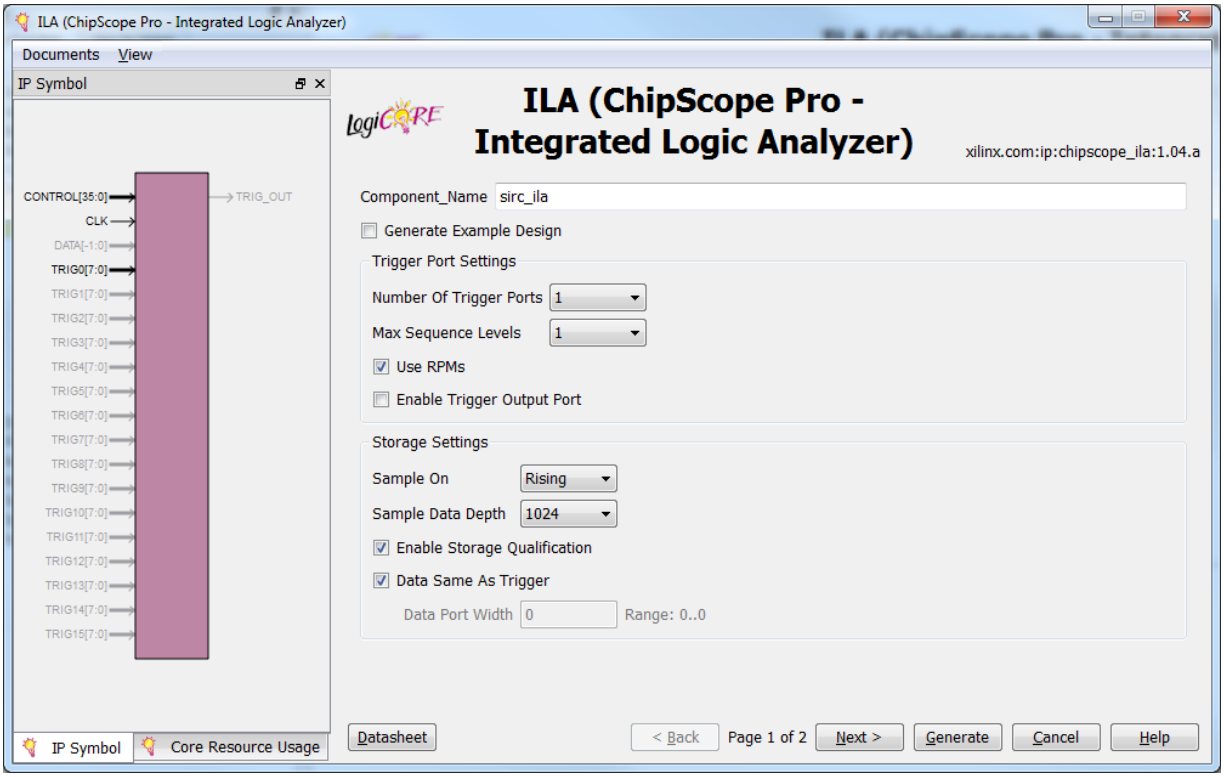
1. Using Xilinx CoreGen, create the Chipscope IP Cores for the capturing traces and sending the traces to the host PC over the JTAG. In the 'IP Catalog', navigate to 'Debugging & Verification->Chipscope'.



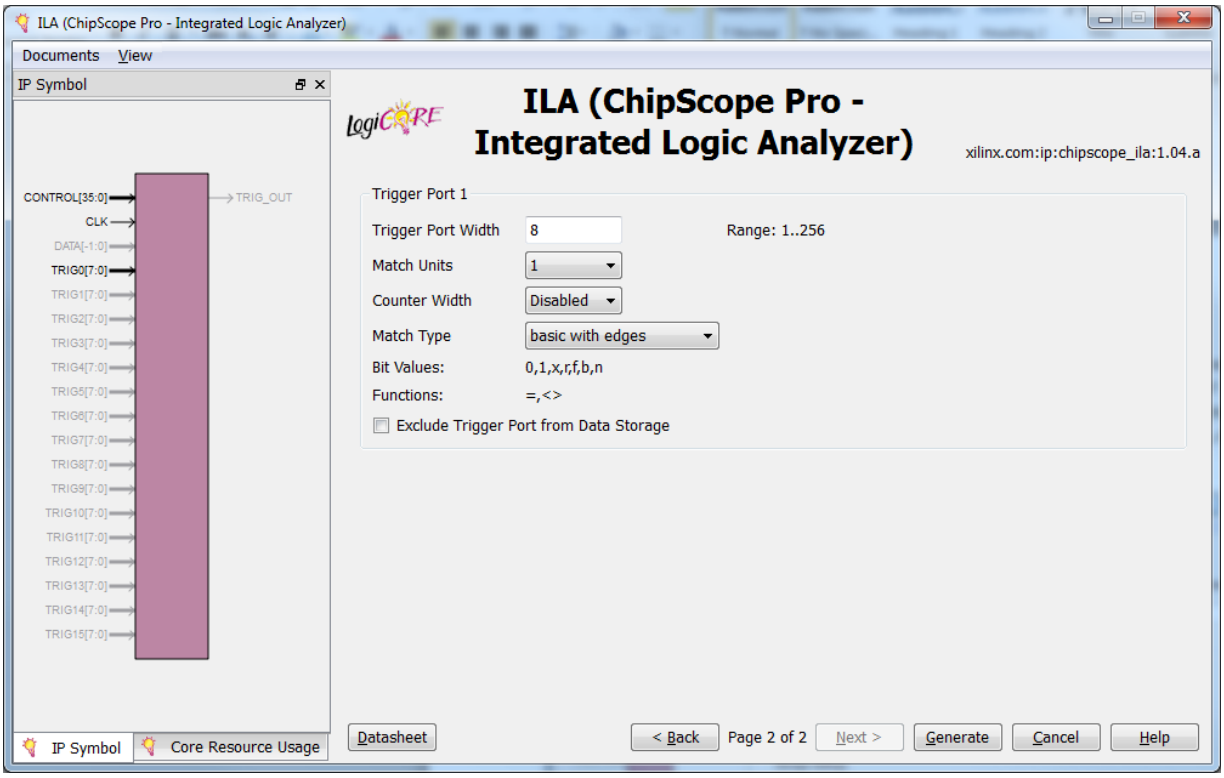
2. Select the ICON core and double click. Give the ICON core a name (EX. sirc_icon). Select the number of desired control ports. One control port is required for each ILA that will be connected to the ICON. For this example, one control port is selected. Check the 'Disable Boundary Scan Component Instance'. It is permissible to have this unchecked in some cases. However, each case is doable with it checked, so this example will be implemented with it checked. Click 'Generate' button when ready.



3. Select the ILA core and double click. Give the ILA core a name (EX. sirc_ila). This example is using the default settings of the ILA. Options to note are the 'Sample On' and the 'Sample Data Depth' parameters in 'Storage Settings'. The 'Sample On' should match the desired capture edge of the reference clock and the 'Sample Data Depth' to the appropriate size to for the debug traces. The more samples the more blockram and debug logic will be required.



4. Set the 'Trigger Port Width' to the number of signals desired to probe using the ILA. This can be 1 to 256 signals. Click 'Generate' button when ready.



5. In addition, using Chipscope debugging capability requires a connection to the FPGA JTAG resources. This accomplished using the BSCAN primitive from the Xilinx HDL library. It is recommended using a wrapper file like the one below when implementing the BSCAN in your design.

```
`timescale 1ns / 1ps

`define V6 1

module bscan_wrapper #(
        parameter       JTAG_CHAIN      =       1
        )       (
                output wire CAPTURE,
                output wire DRCK,
                output wire RESET,
                output wire RUNTEST,
                output wire SEL,
                output wire SHIFT,
                output wire TCK,
                output wire TDI,
                output wire TMS,
                output wire UPDATE,
                input wire TDO
                );

`ifdef V6
        BSCAN_VIRTEX6 #(
                .DISABLE_JTAG("FALSE"),
                .JTAG_CHAIN(JTAG_CHAIN) // Chain number.
                ) bsi (
                        .CAPTURE(CAPTURE),      // 1-bit Scan Data Register Capture instruction.
                        .DRCK(DRCK),            // 1-bit Scan Clock instruction. DRCK is a gated version of
                                                // TCTCK, it toggles during the CAPTUREDR and SHIFTDR
                                                // states.
                        .RESET(RESET),          // 1-bit Scan register reset instruction.
                        .RUNTEST(RUNTEST),      // 1-bit Asserted when TAP controller is in Run Test Idle state.
                                                // Make sure is the same name as BSCAN primitive used in
                                                // Spartan products.
                        .SEL(SEL),              // 1-bit Scan mode Select instruction.
                        .SHIFT(SHIFT),          // 1-bit Scan Chain Shift instruction.
                        .TCK(TCK),              // 1-bit Scan Clock. Fabric connection to TAP Clock pin.
                        .TDI(TDI),              // 1-bit Scan Chain Output. Mirror of TDI input pin to FPGA.
                        .TMS(TMS),              // 1-bit Test Mode Select. Fabric connection to TAP.
                        .UPDATE(UPDATE),        // 1-bit Scan Register Update instruction.
                        .TDO(TDO)               // 1-bit Scan Chain Input.
                        );
`endif

endmodule
```

Chipscope ICON and ILA in the Static

You can think of this procedure as the manual version of what the integrated Chipscope is doing, except this procedure provides some additional control. For this additional control you pay with the time and effort to add the debugging logic into the source code manually. Add the Chipscope cores and the BSCAN wrapper to the top level design and add the signals to the trigger of the ILA. Refer to the following example.

```
wire        CAPTURE_BS;
wire        DRCK_BS;
wire        RESET_BS;
wire        RUNTEST_BS;
wire        SEL_BS;
wire        SHIFT_BS;
wire        TCK_BS;
wire        TDI_BS;
wire        TMS_BS;
wire        UPDATE_BS;
wire        TDO_BS;

wire [35:0]             CONTROL;
wire [255:0]            TRIG;

assign TRIG = <insert signals here>;

bscan_wrapper #(
            .JTAG_CHAIN(1)
            ) bsw(
                        .CAPTURE(CAPTURE_BS),
                        .DRCK(DRCK_BS),
                        .RESET(RESET_BS),
                        .RUNTEST(RUNTEST_BS),
                        .SEL(SEL_BS),
                        .SHIFT(SHIFT_BS),
                        .TCK(TCK__BS),
                        .TDI(TDI__BS),
                        .TMS(TMS__BS),
                        .UPDATE(UPDATE_BS),
                        .TDO(TDO_BS)
                        );

            sirc_icon scsi(
                        .CONTROL0(CONTROL),
                        .TDO_OUT(TDO_BS),
                        .TDI_IN(TDI_BS),
                        .RESET_IN(RESET_BS),
                        .SHIFT_IN(SHIFT_BS),
                        .UPDATE_IN(UPDATE_BS),
                        .CAPTURE_IN(CAPTURE_BS),
                        .SEL_IN(SEL_BS),
                        .DRCK_IN(DRCK_BS)
                        );

            sirc_ila scii(
                        .CONTROL(CONTROL),
                        .CLK(clk),
                        .TRIG0(TRIG)
                        );
```

Assign the signals to be probed to the trigger bus, TRIG. If the signals are in modules somewhere else in the hierarchy, even partial reconfiguration partitions, they must be passed to the top static module and assigned to the trigger bus. If the signals are part of the partial reconfiguration partition, they will cross the partition boundary and create a new partition pin for that signal. When viewing the signals in the Chipscope Analyzer, signals from the partial reconfiguration partition will read as high or logic 1 when the partition is not configured or blank. This way of setting up the Chipscope debugging cores is pre synthesis and thus does not depend on how the synthesis tools optimize signals when it comes to the signals available to probe.

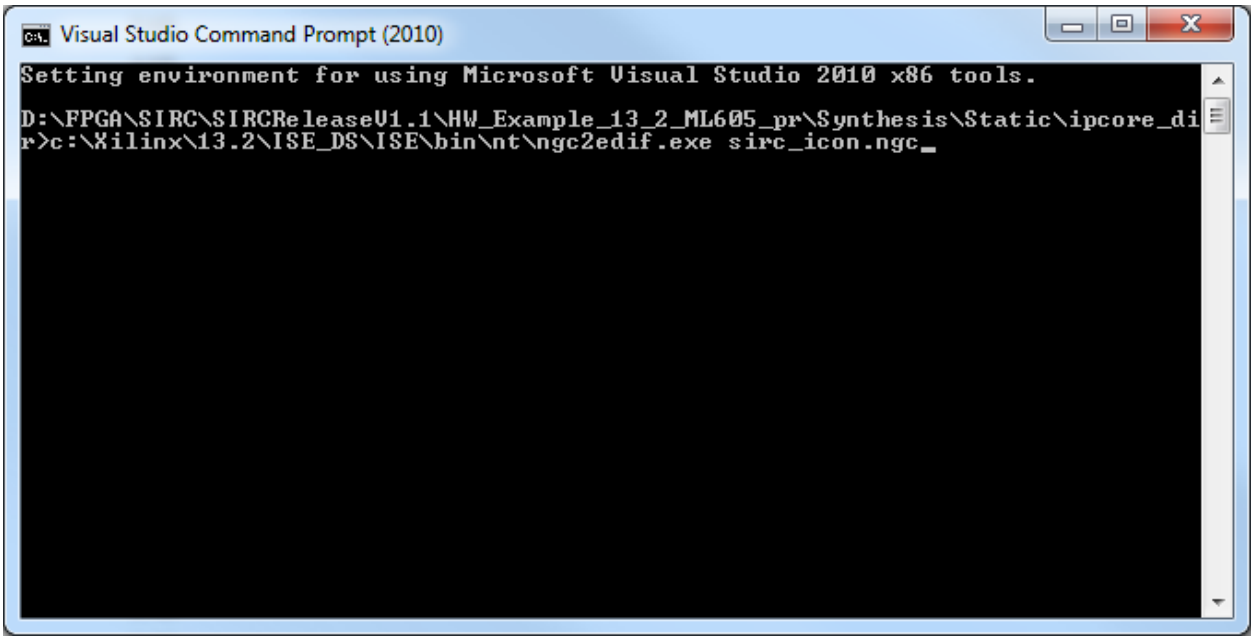
Resynthesize the static netlist and update the file the PlanAhead project. Go to the Sources pane and right-clicking on the 'system.ngc' file. In the menu, select 'Update File'. Follow the dialogue to import the updated netlist. The Chipscope ip cores should appear as black boxes in the hierarchy of the Sources pane. In the File Menu, select 'Add Sources' and follow the dialog for 'Add Design Sources' to add the netlists files for the ICON and ILA i.e sirc_icon.ngc and sirc_ila.ngc. With the netlists updated with the appropriate fixes, repeat the steps in 'Implement Designs'.

Chipscope ICON and ILA in the PR Partition.

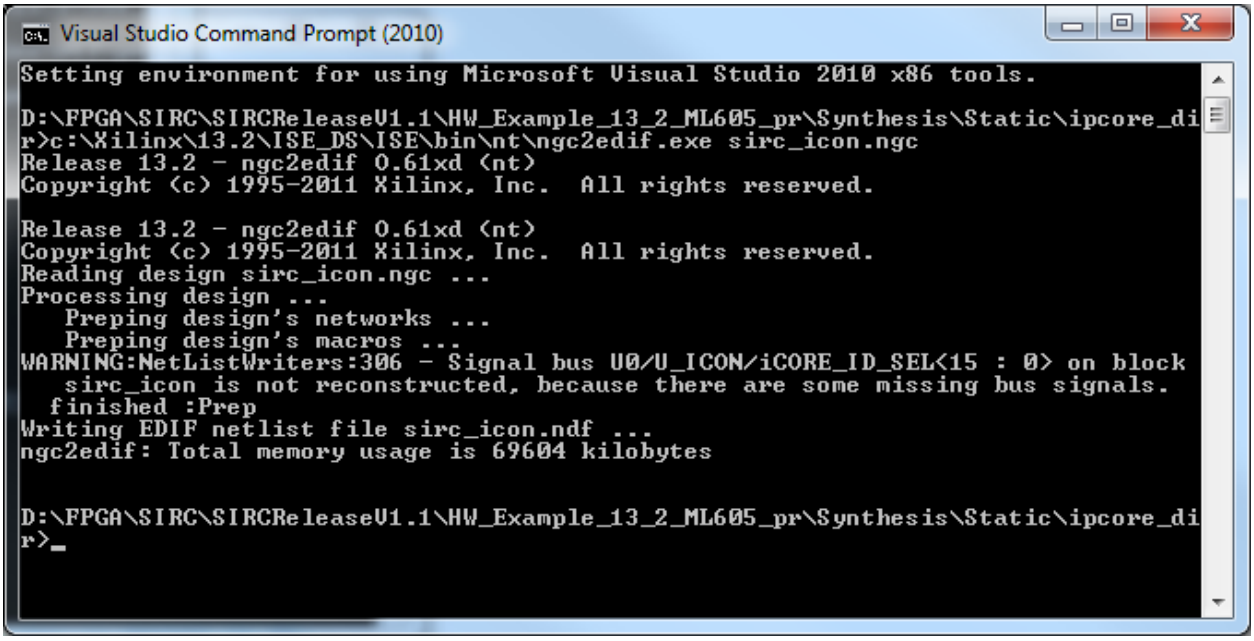
Normally the Chipscope ICON can only be inserted into the static region, because inside the ICON netlist there is a BUFG primitive. The Xilinx Partial Reconfiguration design rules do not allow global clock resources such as BUFGs to reside in a partial reconfiguration partition. However, the BUFR is allowed. It is possible to edit the ICON netlist and replace the BUFG with a BUFR so it can be used in the partial reconfiguration partition.

Edit ICON Netlist

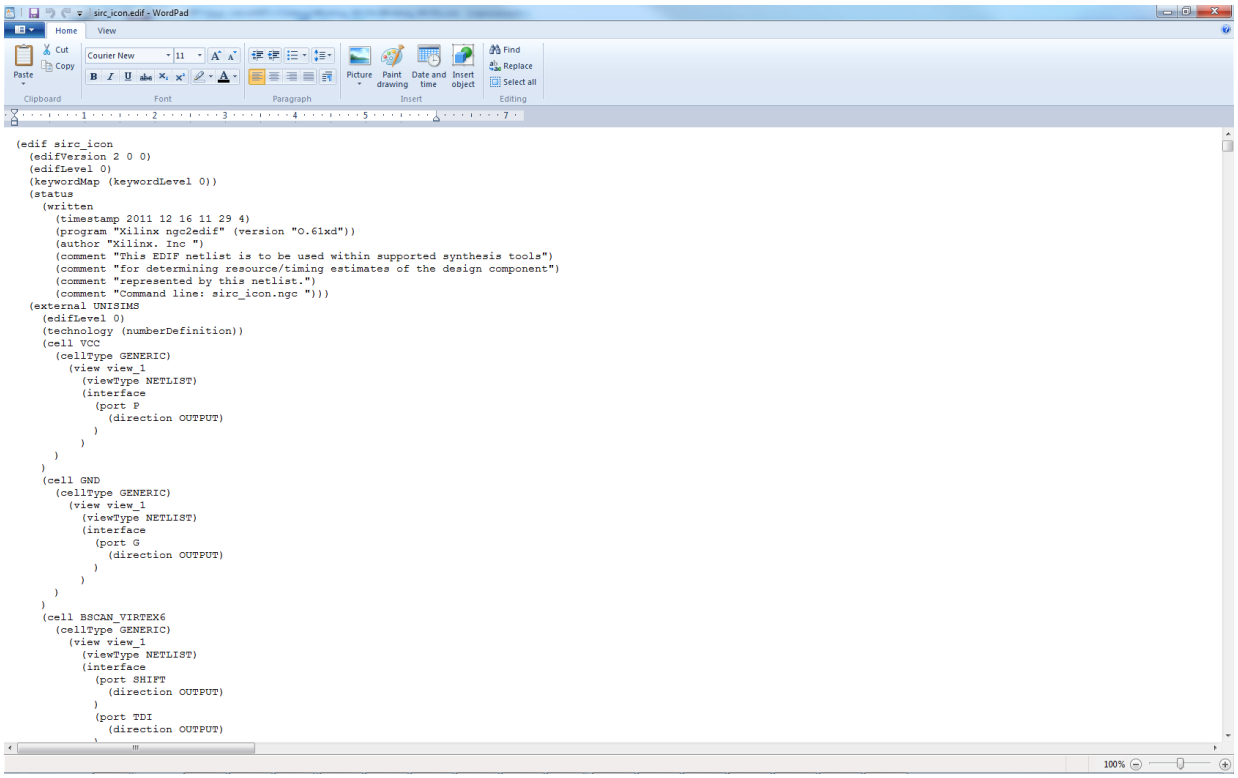
1. Go the location where the Chipscope file are located and open and command prompt.
2. Type "<Xilinx Install>\ISE_DS\ISE\bin\nt\ngc2edif.exe <ICON file>.ngc



3. Press Enter.



4. Change sirc_icon.ndf to sirc_icon.edif
5. Open sirc_icon.edif in a text editor.

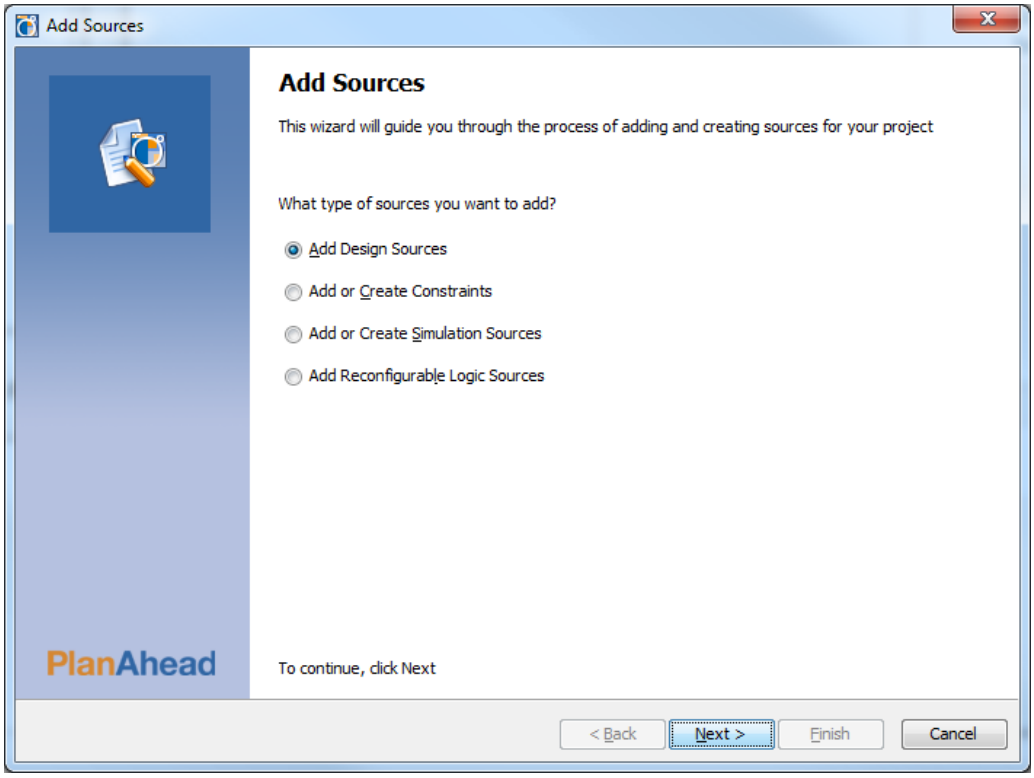


6. Replace all instances of the name 'BUFG' with 'BUFR' using 'Replace'.
7. Save the file.

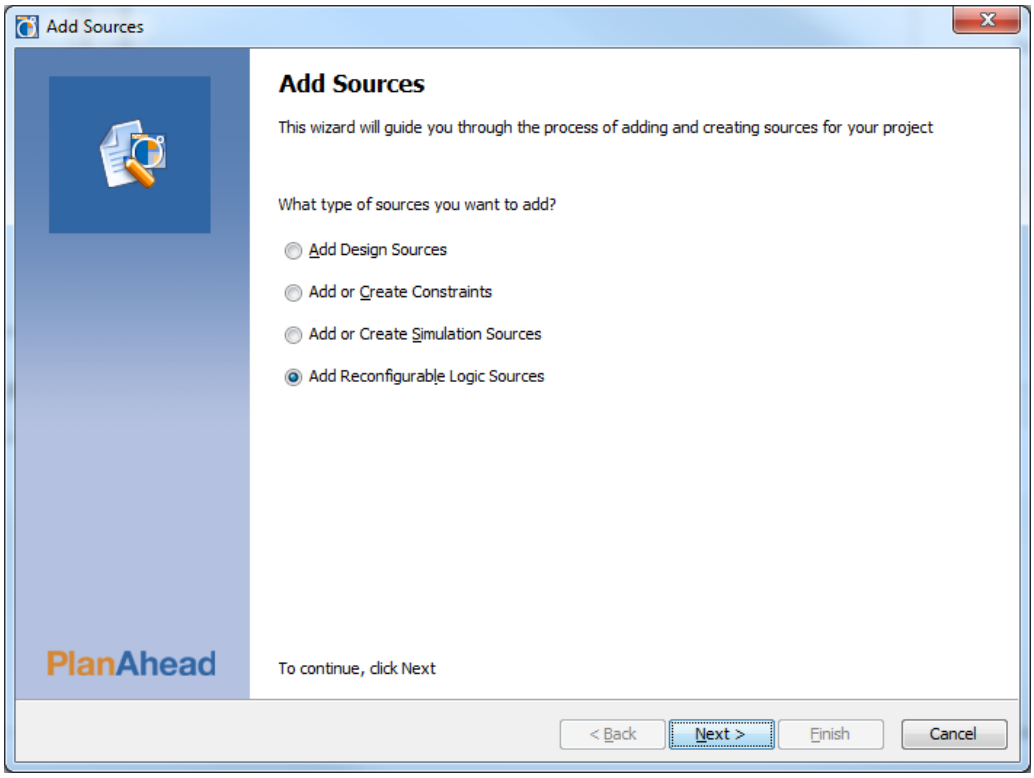
Now edit the top level module of the partial reconfiguration partition by adding the Chipscope modules similarly to the previous section. Resynthesize the user module netlist and update the file in the PlanAhead project. Go to the Sources pane and right-clicking on the 'simpleTestModuleOne.ngc' file. In the menu, select 'Update File'. Follow the dialogue to import the updated netlist. In the 'Netlists' pane, right-click on the module instance that was just updated and select from the menu 'Set as Active Reconfigurable Module'. The Chipscope modules should appear as black boxes in their hierarchy of the Sources pane under the partial reconfiguration partition. In order to proceed, these black boxes must be resolved. With the netlists updated with the appropriate fixes, repeat the steps in 'Implement Designs'.

Resolving Black Boxes of Chipscope IP Core Modules

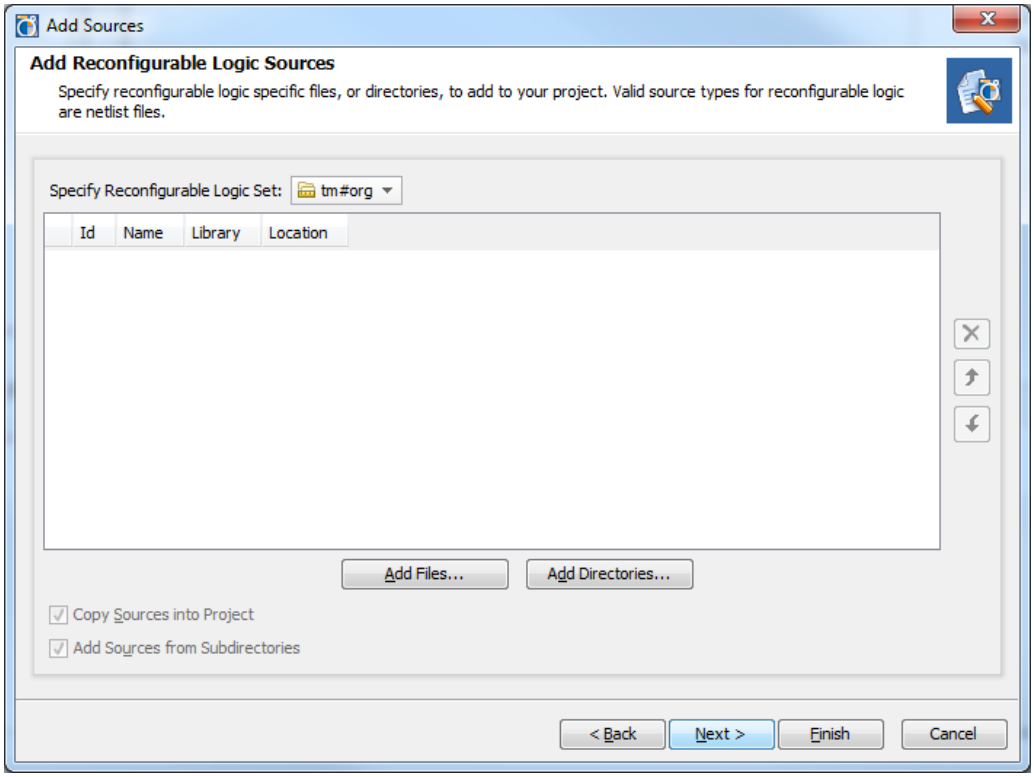
1. In the menu, select 'File->Add Sources'.



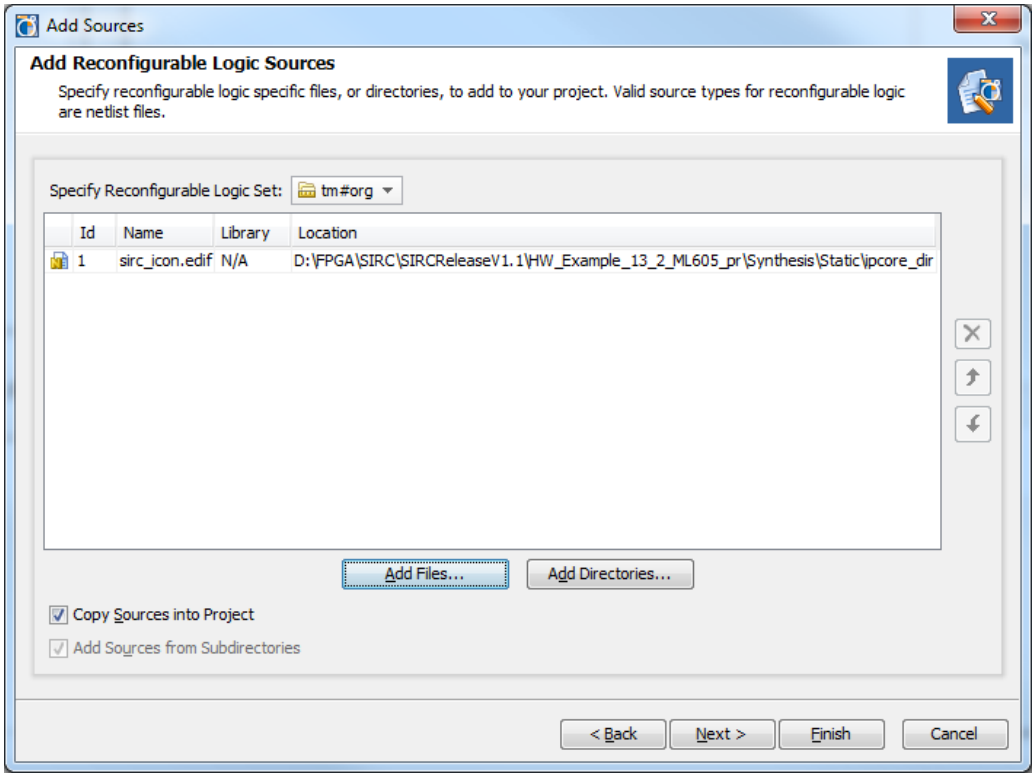
2. Select 'Add Reconfigurable Logic Sources'.



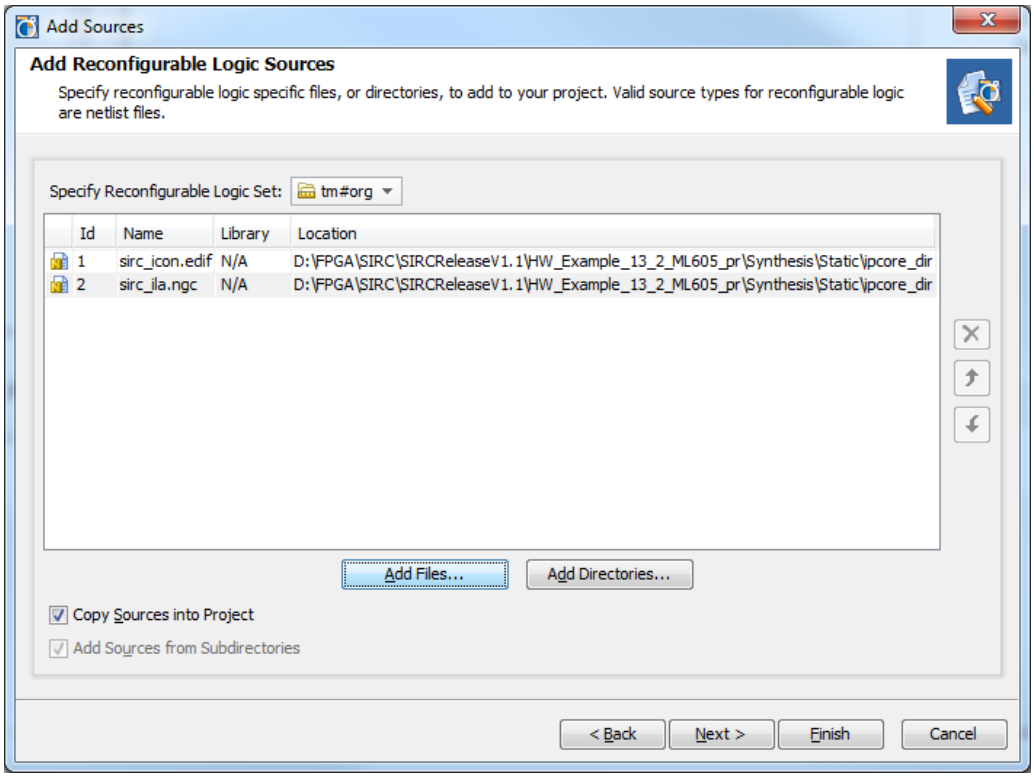
3. Click the ‘Next’ button.



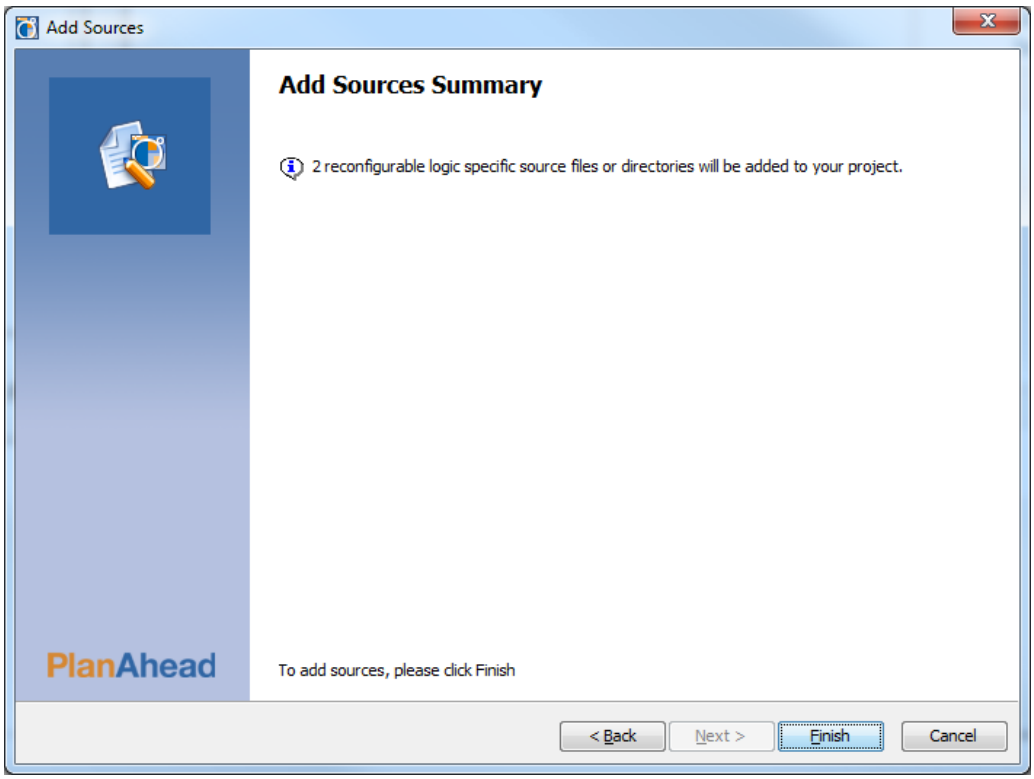
4. Select the partial reconfiguration partition module instance in the ‘Specify Reconfigurable Logic Set:’ drop box.
5. Click the ‘Add Files’ button.
6. Navigate to the modified ‘sirc_icon.edif’ and select it.
7. Click the ‘OK’ button



8. Repeat steps 5 through 7 for the ‘sirc_ila.ngc’.



9. Click the ‘Next’ button.



10. Review the summary and click the ‘Finish’ button.

The advantage of this approach as opposed to placing the Chipscope modules in the static regions is that to implement the partial reconfiguration region, it is first required to implement the static region. Then as long as no changes are required, that implementation is reused for every instance of the partial reconfiguration region. This means that so long as the resources allocated to the partial reconfiguration partition are adequate for the user circuit and the Chipscope debug cores, the impact of the debug logic can be limited to the user circuit and reduce the cases of the debug logic greatly modifying the design being debugged. This also reduces the debug cycle by only requiring the reimplementation of the partial reconfiguration partition.

Chipscope can only support connecting to one ICON within the FPGA. The FPGA has four JTAG chain positions. The Chipscope software on the host PC will always connect to the first ICON it sees on the JTAG chain (e.g. 1 before 2 before 3 before 4). It is possible to have more than one ICON in the design on different positions. If an ICON is in the static, it should be on the last position such that if a partial reconfiguration partition instance is loaded with an ICON, the host will connect to that one before the one in the static when debugging the user circuit.

Chipscope ICON in Static and ILA in PR Partition [5]

The debugging use cases presented thus far make it possible to debug partial reconfiguration designs using the existing tools. However, these have the limitation of only allowing the developer to debug the static and partial reconfiguration partition in isolation from each other. A hybrid solution may be appropriate when designing a general purpose platform using partial reconfiguration, e.g. when the debugging infrastructure is included as a feature of the system.

The Chipscope ICON can be connected to 15 ILAs cores. By implementing a single ICON in the static region and connecting the control ports to an ILA in the static and the partial reconfiguration partition, it is possible to debug the static and user circuits. In this way, it would also be possible to have multiple partial reconfiguration partition with their own ILA for debugging when necessary and removed for production and deployment. The ICON and ILA in the static would remain as a debugging feature of the system platform.

This use cases has several advantages but it requires a little more information and effort to implement than the previous cases. The first issue is that the control ports of the ICON and ILA are declared as INOUT ports. The design rules do not allow for bidirectional signals on the partial reconfiguration partition. Fortunately, these control signals are not truly bidirectional. Xilinx has chosen to declare these as INOUT so they can collect the different signals, both input and output, into a single bus. If you know the directions of each wire in the bus, it is possible to separate it out using a wrapper. The following are example wrappers that can be used.

```
module sirc_icon_wrapper(
        input           CONTROL0_TO_ICON,
        output [34:0]   CONTROL0_TO_ILA,
        input           CONTROL1_TO_ICON,
        output [34:0]   CONTROL1_TO_ILA,
        …
        output          TDO_OUT;
        input           TDI_IN;
        input           RESET_IN;
        input           SHIFT_IN;
        input           UPDATE_IN;
        input           CAPTURE_IN;
        input           SEL_IN;
        input           DRCK_IN;
        );

        wire [35:0]     CONTROL0;
        wire [35:0]     CONTROL1;
        …

        assign CONTROL0_TO_ILA[2:0]     =       CONTROL0[2:0];
        assign CONTROL0_TO_ILA[34:3]    =       CONTROL0[35:4];
        assign CONTROL0[3]              =       CONTROL0_TO_ICON;

        assign CONTROL1_TO_ILA[2:0]     =       CONTROL1[2:0];
        assign CONTROL1_TO_ILA[34:3]    =       CONTROL1[35:4];
        assign CONTROL1[3]              =       CONTROL1_TO_ICON;

        …

        sirc_ic on scsi(
                .CONTROL0(CONTROL0),
                .CONTROL1(CONTROL1),
                …
                .TDO_OUT(TDO_OUT),
                .TDI_IN(TDI_IN),
                .RESET_IN(RESET_IN),
                .SHIFT_IN(SHIFT_IN),
                .UPDATE_IN(UPDATE_IN),
                .CAPTURE_IN(CAPTURE_IN),
                .SEL_IN(SEL_IN),
                .DRCK_IN(DRCK_IN)
                );

endmodule
```

```
module sirc_ila_wrapper(
        output          CONTROL_TO_ICON,
        input [34:0]    CONTROL_TO_ILA,
        input           CLK,
        input [255:0]   TRIG0
        );

        wire [35:0]     CONTROL;

        assign CONTROL[2:0]         =       CONTROL_TO_ILA[2:0];
        assign CONTROL[35:4]        =       CONTROL_TO_ILA[34:3];
        assign CONTROL_TO_ICON      =       CONTROL[3];

        sirc_ ila scii(
                .CONTROL(CONTROL),
                .CLK(CLK),
                .TRIG0(TRIG0)
                );

endmodule
```

After these or similar wrappers including the previously generated Chipscope IP Cores have been added the appropriate synthesis projects, the next step is to insert them into the design. To add the ILA into the partial reconfiguration partition, make the following additions to the user circuit module. This will add the ILA to the user circuit and output the control signals to the static region where they will be connected to the ICON.

```
module simpleTestModuleOne #(
        …
        ) (
                …
                output          CONTROL_TO_ICON,
                input [34:0]    CONTROL_TO_ILA,
                );


        …

        assign TRIG = <insert signals here>;

        sirc_ila_wrapper scii(
                .CONTROL_TO_ICON(CONTROL_TO_ICON),
                .CONTROL_TO_ILA(CONTROL_TO_ILA),
                .CLK(clk),
                .TRIG0(TRIG)
                );

endmodule
```

Next, add the BSCAN and ICON to the static region by making the following changes to the top level module.

Here an ICON is added with two control ports. To generate an ICON with two control ports return to ‘Manually Generating Chipscope IP Cores using CoreGen’ and follow the procedure except change the number of control ports from one to two.

The first control port is connected to an ILA in the static region that can be used for debugging scenarios that involve crossing the partition boundary. The second control port is connected user circuit module using the ports added in the previous steps for adding the ILA in the partial reconfiguration partition.

The BSCAN is still required for this ICON because we have chosen not to include it inside of the ICON for these examples. Here it is recommend using the original ICON with the BUFG rather than the modified one with the BUFR.

```
wire        CAPTURE_BS;
wire        DRCK_BS;
wire        RESET_BS;
wire        RUNTEST_BS;
wire        SEL_BS;
wire        SHIFT_BS;
wire        TCK_BS;
wire        TDI_BS;
wire        TMS_BS;
wire        UPDATE_BS;
wire        TDO_BS;

wire                CONTROL0_TO_ICON,
wire [34:0]         CONTROL0_TO_ILA,
wire                CONTROL1_TO_ICON,
wire [34:0]         CONTROL1_TO_ILA,

wire [255:0]        TRIG;

assign TRIG = <insert signals here>;

bscan_wrapper #(
        .JTAG_CHAIN(1)
        ) bsw(
                .CAPTURE(CAPTURE_BS),
                .DRCK(DRCK_BS),
                .RESET(RESET_BS),
                .RUNTEST(RUNTEST_BS),
                .SEL(SEL_BS),
                .SHIFT(SHIFT_BS),
                .TCK(TCK__BS),
                .TDI(TDI__BS),
                .TMS(TMS__BS),
                .UPDATE(UPDATE_BS),
                .TDO(TDO_BS)
                );

        sirc_icon_wrapper scsi(
                .CONTROL0_TO_ICON(CONTROL0_TO_ICON),
                .CONTROL0_TO_ILA(CONTROL0_TO_ILA),
                .CONTROL1_TO_ICON(CONTROL1_TO_ICON),
                .CONTROL1_TO_ILA(CONTROL1_TO_ILA),
                .TDO_OUT(TDO_BS),
                .TDI_IN(TDI_BS),
                .RESET_IN(RESET_BS),
                .SHIFT_IN(SHIFT_BS),
                .UPDATE_IN(UPDATE_BS),
                .CAPTURE_IN(CAPTURE_BS),
                .SEL_IN(SEL_BS),
                .DRCK_IN(DRCK_BS)
                );

        sirc_ila_wrapper scii(
                .CONTROL_TO_ICON(CONTROL0_TO_ICON),
                .CONTROL_TO_ILA(CONTROL0_TO_ILA),
                .CLK(clk),
                .TRIG0(TRIG)
                );

        simpleTestModuleOne #(
                ...
                ) tm(
                        ...
                        .CONTROL_TO_ICON(CONTROL1_TO_ICON),
                        .CONTROL_TO_ILA(CONTROL1_TO_ILA)
                        );
```

Resynthesize the static and user modules netlist and update the file in the PlanAhead project. Go to the Sources pane and right-clicking on the 'simpleTestModuleOne.ngc' file. In the menu, select 'Update File'. Follow the dialogue to import the updated netlist. Repeat for system.ngc. In the 'Netlists' pane, right-click on the module instance that was just updated and select from the menu 'Set as Active Reconfigurable Module'. The Chipscope modules should appear as black boxes. In order to proceed, these black boxes must be resolved. . In the File Menu, select 'Add Sources' and follow the dialog for 'Add Design Sources' to add the netlist files for the ICON, sirc_icon.ngc. Add the netlist for the ILA using steps similar to 'Resolving Black Boxes of Chipscope IP Core Modules' for the sirc_ila.ngc. With the netlists updated with the appropriate fixes, repeat the steps in 'Implement Designs'

In this scenario the debugging infrastructure is included as part of the static platform. The impact of the debugging logic is for the most part known and accounted for. It will not change so long as you use this version of the static platform. In this way, you have greatly limited how much the debugging logic will change your design, one of the greatest aggravations of FPGA on-chip debuggers. Additionally, this scenario provides the greatest visibility of both the static platform and the user module being debugged.

References

1. Ken Eguro. "SIRC: An Extensible Reconfigurable Computing API." Proc. IEEE Symposium on Field-Programmable Custom Computing Machines 2010 (FCCM 2010).
2. Xilinx, Inc. "Chipscope Pro Software and Cores User Guide." Xilinx UG029, July 6, 2011. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/chipscope_pro_sw_cores_ug029.pdf
3. Xilinx, Inc. "ML605 Hardware User Guide." Xilinx UG534. July 18, 2011. http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf
4. Xilinx, Inc. "Partial Reconfiguration User Guide." Xilinx UG702. July 6, 2011. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/ug702.pdf
5. Xilinx, Inc. "Partial Reconfiguration – Can I insert Chipscope Cores within Reconfigurable Modules?" Answer Record 42899. December 8, 2011. http://www.xilinx.com/support/answers/42899.
6. Xilinx, Inc. Xilinx Homepage. www.xilinx.com