grittyengineer.com

Vivado Non-Project Mode: Releasing Vivado's True Potential - Gritty Engineer

Christopher Hogstrom

9–11 minutes

Most digital circuit design engineers are familiar with Vivado's GUI. It allows engineers to create a project, select the target part, add or create source files for the RTL design, add physical and timing constraints, and go through the synthesis, implementation, and bitstream generation process. What most engineers don't know is that there is another, more powerful mode called Non-Project Mode that is only available through Tcl scripts. If you're not familiar with Tcl check out these free tutorials we put together here.

This mode is called Non-Project Mode because you do not create a Vivado project. Instead, the design is built in memory. There are a number of benefits to using Non-Project Mode including

- High resolution over the optimizations performed in each step of the implementation process
- Ability to re-run steps with different directives to iteratively improve timing
- Create design checkpoints (dcp) which save the in-memory design to a dcp file. These can be used as starting points instead of starting the entire build process over (for example, you can place the design, save it as a dcp, and then route using one of the available options. If it fails to

meet timing just open the dcp you created and re-route using a different directive)

- The build process typically runs faster
- Build automation tools can be integrated
- Source control is far easier as the Tcl script is a simple text file and changes to configurations are easy to see using simple diff tools
- Increased repeatability

If you think of other advantages worthy of inclusion, put them in the comment section and we'll update the post accordingly. With all of these advantages, it's no wonder why serious engineers prefer to use Vivado's Non-Project Mode.

In this post, we'll go over a simple Tcl script that you can use to build your design in Non-Project Mode. This post uses the example provided in UG894.

Create Output Directory

The first step is to define the output directory. This is where your reports are going to go, any dcp files you create, and your bitstream. It's also a good idea to define the target part near the top so you can easily find later. Once you define the output directory, check to make sure it's empty. The code below shows how to delete all of the files and sub-folders if the output directory is not empty.

```
#Define target part and create output directory
set partNum TargetPartNum
set outputDir ./path/to/outPutDir
file mkdir $outputDir
set files [glob -nocomplain "$outputDir/*"]
```

```
if {[llength $files] != 0} {
    # clear folder contents
    puts "deleting contents of $outputDir"
    file delete -force {*}[glob -directory $outputDir *];
} else {
    puts "$outputDir is empty"
}
```

Add HDL Source Files and Constraints

Next, add the HDL source files and constraints shown in the code below. When adding HDL source files there are two commands: **read_vhdl** and **read_verilog**. Notice that these commands are not color coded inside the code block. This is because they are not built-in Tcl commands but are procedures (basically Tcl functions) defined in the Vivado namespace. The **read_vhdl** command is used when reading VHDL files and **read_verilog** is used when reading Verilog files (pretty easy to remember). Notice that the method of adding files is different in Non-Project Mode than Project Mode which uses the commands **add_files** or **import_files** (to see how to create a Project Mode Tcl script click here). In Non-Project mode, source files are referenced without creating a dependency on the files. Files must be monitored for changes and the design updated accordingly.

```
#Reference HDL and constraint source files
read_vhdl -library usrDefLib [ glob path/to/vhdl/sources/
*.vhdl ]
read_verilog [ glob path/to/verilog/sources/*.v ]
read_xdc path/to/constraint/constraint.xdc
```

The **read*** commands can be given the optional parameters listed below.

-library <arg>	The library the files should reference. If no library is specified the default library, xil_defaultlib, is used.
-quiet	All messages generated by the command are surpressed, including error messages.
-verbose	Override messge limits and output all messages generated by the command.
-sv	For use with **read_verilog**. Specifies the file to be of type SystemVerilog.
-vhdl2008	To be used with **read_vhdl**. Specifies the file to be of type VHDL 2008.

Run Synthesis, Write DCP, and Create Reports

Now its time to run synthesis. The first line of the code below tells Vivado to begin running synthesis. The first argument tells Vivado the name of the top module, not the file name. The second argument specifies the target part number.

```
#Run Synthesis
synth_design -top NameOfTopModule -part $partNum
write_checkpoint -force $outputDir/post_synth.dcp
report_timing_summary -file $outputDir/
post_synth_timing_summary.rpt
report_utilization -file $outputDir/post_synth_util.rpt
```

Other optional and useful arguments are:

-generic	Specify generic parameters. Syntax: -generic <name>=<value> – generic <name>=<value>

-gated_clock_conversion	Convert clock gating logic to flop enable. Values: off, on, auto Default: off
-resource_sharing	Sharing arithmetic operators. Value: auto, on, off Default: auto
-verbose	Override messge limits and output all messages generated by the command.

A comprehensive list of optional arguments can be found in UG835 under synth_design.

After Vivado is done synthesizing the design, it's good practice to write the in-memory synthesized design to a checkpoint. This must be manually commanded otherwise no checkpoint is written. The **-force** argument overwrites any .dcp file with the same name. The last two lines generate our timing and utilization reports for later use.

Optimize Design, Write DCPs, and Create Reports

Next, optimize the design for the target part and place the ports and logic cells onto the device resources. After placing the design, we create a clock utilization report.

```
#run optimization
opt_design
place_design
report_clock_utilization -file $outputDir/clock_util.rpt
```

The next piece of code is really cool and highlights the power and flexibility of working in Non-Project Mode. It is possible to identify timing violations and then perform optional optimizations to potentially resolve the issues. This ability is only available in Non-Project Mode as shown below.

```
#get timing violations and run optimizations if needed
if {[get_property SLACK [get_timing_paths -max_paths 1 -
nworst 1 -setup]] < 0} {
 puts "Found setup timing violations => running physical
optimization"
 phys_opt_design
}
write_checkpoint -force $outputDir/post_place.dcp
report_utilization -file $outputDir/post_place_util.rpt
report_timing_summary -file $outputDir/
post_place_timing_summary.rpt
```

To those unfamiliar with Tcl's syntax the code for the if statement might seem a little confusing. In Tcl, commands wrapped in brackets are executed first and the value of the command is returned. In the case of the if statement, there are two commands nested in brackets, **get_property**, and **get_timing_paths**. Because **get_timing_paths** is wrapped within the most nested brackets, it is executed first. The command returns a timing paths Tcl object that can be queried with the **get_property** command (the documentation for the properties that can be quired for First Class Vivado objects can be found here). The **-max_paths** argument specifies the max number of paths to return while the **–nworst** argument specifies the number of paths to show to each endpoint. The **-setup** argument checks for setup violations.

The **get_property** command queries the Tcl object for the **SLACK** property and checks to see if this value is less than zero. If a violation is detected, execute the physical optimization.

Once complete, write out the in-memory design to a DCP and generate utilization and timing reports.

Route Design and Generate bitstream

Now we're ready to route the design using **route_design**. In the code below, we use the **Explore** directive which commands the Vivado router to explore critical paths based on timing after an initial route. There are many different directives to choose from that may give better results depending on your design. There's a cool post on how to use DCPs to run routing with different directives to find the best directive for your design by the Hardware Jedi. You can use the link here to view their post.

```
#Route design and generate bitstream
route_design -directive Explore
write_checkpoint -force $outputDir/post_route.dcp
report_route_status -file $outputDir/
post_route_status.rpt
report_timing_summary -file $outputDir/
post_route_timing_summary.rpt
report_power -file $outputDir/post_route_power.rpt
report_drc -file $outputDir/post_imp_drc.rpt
write_verilog -force $outputDir/cpu_impl_netlist.v -mode
timesim -sdf_anno true
write_bitstream -force $outputDir/nameOfBitstream.bit
```

After routing, we write the final DCP and generate our reports. The **write_verilog** command writes the netlist of the design in Verilog format. **write_vhdl** could have been used to write the netlist of the design in VHDL format. Finally, we write our bitstream.

Hopefully, after seeing how powerful Non-Project Mode can be, you'll begin using it for your Vivado FPGA builds. Please let us know how it goes for you in the comments or if you have any questions.

Bonus: Get a downloadable Non-Project Mode Tcl template