Non-Project Mode as a solution for some code management and testing problems

Ilia Kalistru

This article shows some drawbacks of projects-based FPGA development design flow and to show how Non-Project Mode of Vivado software suite resolves this problems.

1 Introduction

A common way to do an FPGA design is to create a project using your design tool. A project consists of a hierarchy of directories and files in it. Creating a project, you have a nice GUI, wizards, and you even can create your design without touching your keyboard, with mouse clicks only. It works well for small and simple projects or projects where you only combine existing IPs in a purpose-specific mixture.
Unfortunately, if you make something big and special, and if you have to write tons of code, this project-centered approach creates significant problems. We will show that projects create issues with attempts to use source code control systems (SCCS) and with attempts to use test driven development techniques or any other systematic testing methodology.
Using design tools in Non-Project Mode eliminates these problems.

2 What is the problem with projects?

2.1 Incompatibility with source control systems

A source code control system is a tool to manage your code and it prevents many issues with your code. In a big project you are going to have experimental changes in your code, and you want to be able to revert the changes if they will not work out. You also probably want to continue development of the system while you are looking for a reason of a bug you have found in a older version of a code and cannot reproduce in a newer one. This requires switching between the new version and the version where the bug can be reproduced. At last, you simply can have several designers in your team and you would like to merge contributions of different designers from different branches.
Projects of design tools have many files in their directories. It is not only source code files, created by an engineer, but also many files created by a tool. It create problems with SCCS.
If you do not add project's files to your SCCS, you are going to have problems each time you switch branches in the SCCS if this action removes or adds new source code files. As your project files do not change when you switch branches or when you revert changes, your project "doesn't know" that some files have appeared or have disappeared. After each such change you have to manually find out what files have changed and add/remove them from the project. It's a boring and error prone process. If you forget to re-add a file, in the best case you will wait until the end of synthesis, and somewhere during implementation phase you will get an error message about attempt to instantiate a black-box module. Much worse if you spend a day (or even more) debugging your project to find out that it does not work because there is a black-box module which does not drive signals it is supposed to drive.
On the other hand, if you add your project files to your SCCS, it creates another problem. Design tools tend to have huge number of files in their projects. Moreover, they change them each time you open the project (Even if you do not change anything!). This forces you to commit all this changes to your SCCS. Your repository is going to become huge over time, but more importantly, you are going to experience infoxication. Each time you make diff to see the difference between two commits you will see an overwhelming amount of changes in files you do not care about. It is easy to overlook an important difference between two project states among these piles of changes in project files.

Not surprisingly, many FPGA designers do not use source control systems in such circumstances. They create a backup copy of their project every now and then and try to keep track of this copies in a text file with descriptions of these backups. It eventually creates a mess in the sources and the designer mistakenly deletes a wrong version of a project or forgets to transfer some changes from one copy of the project to an another copy.
As it will be shown later, Non-Project Mode simply does not have listed drawbacks and it is fully compatible with source control systems.

2.2 Incompatibility with extensive testing

There are many things that can go wrong in a big project, so rigorous testing is essential. Testing of a system on different levels and processes such as test-driven development facilitate in better quality of code. These techniques imply that you have numerous tests. You have a test for each module, or even several tests for different features of it. Moreover, you can have different tests for one module created with different parameters/generics. To be sure that modules match with each other, you are going to have integration tests for each bunch of modules. Eventually, you have a big system test, which tests all your modules together.
As you have dozens or even hundreds of tests, it would be nice to be able to run all of them at once instead of manual launching of each test. It seems to be a problem in a Project Mode. To run another test you should assign a corresponding testbench to be “the main” testbench and manually launch simulation. That is not a problem for Non-Project Mode. You can run all your tests simultaneously with one click.
Randomized tests further reduce number of bugs in your design because with this approach you create thousands or millions different random situations for your device under the test. Such tests are able to find a fault in your design which becomes evident only under a very specific conditions, which nobody could think of. Test coverage increases even more if each time the tests are run, they run with different seed for their random number generators. As it will be shown later, Non-Project Mode allows such randomization. It also allows to run a test with a specific seed if you want to reproduce conditions of a failure of a previous test.

3 Solution

Non-Project Mode is a way to work with Vivado suite that offers a simple and convenient solution of this problems. In Non-Project Mode FPGA design engineer have a set of HDL files, constraints files and scripts in a repository instead of having a Vivado project. This scripts may use source files from any convenient location of the repository and can instruct Vivado to create all temporary files in a specified directory. This directories can be easily excluded from adding to a repository. As a result only meaningful changes of source files will be visible in SCCS. Additionally, as source files and scripts are all what is needed for your work in this mode, we can be sure, that all important information is on SCCS and this information remains consistent when we switch branches.
As each script can create its own working directory with files for its work and reports, they do not interfere with each other. It is possible to run multiple scripts simultaneously even if they use the same source code files. This allow running multiple different tests or different synthesis processes at once. The author usually uses a script which launches all the tests at the end of a day. The other script, which checks all the reports for presence of errors, is used at mornings.
Non-Project mode also allows us to run multiple synthesis and P&R scripts simultaneously. It allows us to try different settings and optimization strategies. It also allows assembly of some small chunk of design to check its timings quickly.
A way to build a randomized test using a random seed for RNG, which allows you to reproduce faulty tests, will be shown in the next section of the article.
It is important to understand that work in Non-Project Mode does not mean that we cannot take advantage of GUI of Vivado. An engineer working in Non-Project Mode can open Vivado in GUI

mode to look at results of work of scripts. It is much more convenient to study faulty critical paths in GUI then looking at text reports. It is also possible to add a *start_gui* command at any place in a script to have following commands executed with GUI on. For example, the author uses a tcl variable to determine if he wants a test being executed in a command line mode or if he wants to view waveforms in a GUI.
Let us consider examples of a test script and of a synthesis script.

3.1 A test script

Here we will show how to built a basic test script.
At the beginning of the script it is convenient to define all constants, like names of files or a name of a working directory. Such constants are the parts of script that are most likely to be changed, and it is much more convenient to do if we have them collected together. It also improves readability.
So, at first we define a name of a working directory. I prefer to start names of working directories for tests starting with the prefix *sim_*. This name must be unique for each script so that we can run multiple scripts simultaneously. Then, it is convenient to define a variable, which determines if we are going to run this test in a command line mode or with GUI. This allows us to routinely run it without GUI and easily swap to GUI if the test fails and we need to debug it.
We also define a name of a top module of our testbench and a list of RTL source files.

```
set simdir ./sim_OurTestbench
set visual 0
set tst OurTestbench
set slist [list \
   ./src/SourceFile1.vhd \
   ./src/OurTestbench.vhd \
]
```

As these files will be compiled in the order we list them, make sure that you list files with packages before files where these packages are used. If we use IP cores then for each core we should also add *<corename>_stub.vhdl* (or .v) and *<corename>_sim_netlist.vhdl* (or .v) files in the same list (in this order). These two files you can find in the directory where you generated the core.
After declaring target device using *set_part -quiet <device_p/n>* command we define that we want to run Vivado in multiple threads.

```
set_param general.maxThreads 8
```

Create a directory for files for Vivado and for results of simulation and switch directory so that all files are generated there.

```
file mkdir $simdir
cd $simdir
set srcdir ./..
```

Load all sources in the memory.

```
foreach a $slist {exec cmd /c xvhdl $srcdir/$a}
```

We generate a random seed for the test from two random arguments passed to the script. We use two different random numbers because we use Windows' *%RANDOM%* variable as a source of initial seed and it has a small range of possible values. By combining two of them we can make wider range of possible seeds. We also print out these random numbers so that we could repeat the same test again if it fails with this particular seeds.

```
set seedf "seed="
set seeda1 [lindex $argv 0]
set seeda2 [lindex $argv 1]
```

```
puts seeda1:$seeda1
puts seeda2:$seeda2
set seeda [expr $seeda1 * 32768 + $seeda2]
if {$seeda == 0} { set seeda 1}
puts seeda:$seeda
```

Suppose the testbench in our example has two generic parameters: *seed* and *Encoder*. (Only integer type of parameters passed from TCL scripts to VHDL code is supported at the moment by Vivado.)
Elaborate the testbench with the determined settings:

```
exec xelab -s ${tst}Enc --debug typical -generic_top $seedf$seeda \
     -generic_top "Encoder=1" work.$tst
```

Then we run the test:

```
if [ expr $visual == 1] {
   start_gui
   xsim -log sim_${tst}Enc.log -view ../${tst}Enc.wcfg ${tst}Enc
} else {
   exec cmd /c xsim -R -onerror quit -log report_${tst}Enc.log ${tst}Enc
}
```

After the first test finishes, we want to repeat elaborate-simulate sequence for a different value of the Encoder parameter:

```
exec xelab -s ${tst}Dec --debug typical -generic_top $seedf$seeda \
    -generic_top "Encoder=0" work.$tst
if [ expr $visual == 1] {
   start_gui
   xsim -log sim_${tst}Dec.log -view ../${tst}Dec.wcfg ${tst}Dec
} else {
   exec cmd /c xsim -R -onerror quit -log report_${tst}Dec.log ${tst}Dec
}
```

Close Vivado when the test is finished.

```
if [ expr $visual == 0] {
    exit
}
```

In case of visual set to 0, results of simulation will be saved at *report_OurTestbenchEnc.log* and *report_OurTestbenchDec.log* files.
This script is launched using the a *sim_OurTestbench.bat* file with a following content:

```
TITLE %~n0
vivado -mode tcl -tempDir tmp -source sim_OurTestbench.tcl -log vivado_sim_OurTestbench.log -tclargs %RANDOM% %RANDOM%
```

If we need to repeat the test with the same seed, we take *seed1* and *seed2* values from the log file and insert them in place of the two *%RANDOM%* arguments.

3.2 A synthesis and P&R script

Let us discuss a script which generates an .mcs file.
At first we define working directory, environment and lists of source files. Make sure that files with packages are defined before files that use them.

```
set synthdir ./Synth_OurDesign
set topmodule TopVCU118Module
set part xcvu9p-flga2104-2L-e
set VHDL_sources [list \
    ./src/I2C/I2C.vhd \
    ./src/TopVCU118Module.vhd \
]
set ips [list \
    ./src/ip/ip1/ip1.xci \
    ./src/ip/ip2/ip2.xci \
    ./src/ip/clk_0/clk_0.xci \
]
set constraints [list \
    ./src/VCU118TopClocks.xdc \
    ./src/VCU118TopPins.xdc \
    ./src/VCU118Conf.xdc \
    ./src/I2C.xdc \
]
```

Let us make it multithreaded:

```
set_param general.maxThreads 8
```

Create a working directory and load sources in memory:

```
file mkdir $synthdir
cd $synthdir
set srcdir ./..
set outdir ./results
file mkdir $outdir
set_part -quiet $part
foreach a $VHDL_sources {
    read_vhdl $srcdir/$a
}
foreach a $ips {
    read_ip $srcdir/$a
}
foreach a $constraints {
    read_xdc $srcdir/$a
}
```

We are free to set different parameters of synthesis.

```
set_param synth.elaboration.rodinMoreOptions "rt::set_parameter synRetiming true"
synth_design -top $topmodule -part $part
```

After synthesis we save a checkpoint so that we can load it if we want to rerun implementation without rerunning synthesis.

```
write_checkpoint -force $outdir/post_synth
```

Also, we generate some reports we need using commands *report_clocks, report_clock_interaction, report_timing_summary*.

After optimization of a synthesized design we place and route it. We are also free to run different optimizations on different stages of the process. Creating checkpoints after each step is not necessary, but could be useful.

```
opt_design
place_design
```

```
phys_opt_design -retime
write_checkpoint -force $outdir/post_place
route_design
write_checkpoint -force $outdir/post_route

report_timing_summary -file $outdir/post_route_timing_summary_unc -report_unconstrained
report_drc -file $outdir/post_route_drc
report_utilization -file $outdir/utilization
```

If we have ChipScope module installed in our design it is handy to use the following command.

```
write_debug_probes -force $outdir/probes.ltx
```

With this file ChipScope viewer automatically groups signals into buses and assigns correct names to them.

And finally, we create our bitfile and .msc file.

```
write_bitstream -force $outdir/$topmodule.bit

file copy -force $outdir/$topmodule.bit ./
write_cfgmem -force -format MCS -size 128 -interface BPIx16 -loadbit "up 0x00000000
$topmodule.bit" $outdir/$topmodule.mcs
exit
```

4 Conclusion

Non-Project Mode is a great instrument when you work in a big project. It is fully compatible with source code control systems and this fact means that it is easy to manage your code even if you have several people changing it. Using Non-Project Mode it is easy to run multiple randomized tests of your design at the same time without copying it. It is also possible to run multiple synthesis, place and route processes whether you want to try different optimizations or you just want to asses timings of several modules.

5 Further reading

1. Vivado Design Suite Tcl Command Reference Guide UG835
2. Vivado Design Suite User Guide: Design Flows Overview UG892
3. Video: Using the Non-Project Batch Flow
4. Vivado Design Suite Designing with IP tutorial Lab 4