Vivado Design Suite Tutorial

Revision Control Systems

UG1198 (v2015.4) November 18, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Changes
11/18/2015	2015.4	Initial release.

Table of Contents

Revision History 2
Revision Control 5
Introduction 5
Hardware and Software Requirements 5
Tutorial Lab Descriptions 5
Locating the Tutorial Design Files 6
Installing Tutorial Utilities 6
Lab 1: Simple RTL Project 9
Introduction 9
Lab Procedure 9
Conclusion 13
Lab 2: Simple IP Project 14
Introduction 14
Lab Procedure 14
Conclusion 17
Lab 3: Custom IP Project 18
Introduction 18
Lab Procedure 18
Conclusion 22
Lab 4: HLS-Based IP 23
Introduction 23
Lab Procedure 23
Conclusion 26
Lab 5: IP Integrator Block Design 27
Introduction 27
Lab Procedure 27
Conclusion 29

Lab 6: Top Level Integration 30
Introduction 30
Lab Procedure 30
Conclusion 32
Appendix A: Introduction to GIT 33
Git Overview 33
Conclusion 36
Appendix B: Introduction to the Make Utility 37
Make Utility Overview 37
A Simple Example 37
Multiple targets 38
All Targets 39
Clean Targets 40
Variables 40
Windows vs. Linux 40
Legal Notices 41
Please Read: Important Legal Notices 41

Introduction

The Xilinx® Vivado® Design Suite can work with a variety of revision control systems. The methodologies for source management and revision control can vary depending on the user and company preference, as well as the software used to manage revision control. This tutorial demonstrates some of the methods used for revision control and source file management using Git and make utilities that are included with the Vivado Design Suite software. For an introduction to the topic, and recommendations for source management and revision control, please refer to this link in the *Ultrafast Design Methodology Guide for the Vivado Design Suite* (UG949).

VIDEO: *You can also view the Using Vivado Design Suite with Revision Control quick take video to learn more about this topic.*

Hardware and Software Requirements

You will be using the following tools for these labs:

- Vivado 2015.x System Edition and SDK
- Git revision control – included with Vivado Design Suite
- MinGW make utility - included with Vivado System Edition

Please be sure to review the Appendices in this tutorial to familiarize yourself with setup and use of Git, and a brief introduction to the make utility.

Refer to the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)* for a complete list and description of the system and software requirements for the Vivado Design Suite.

Tutorial Lab Descriptions

This tutorial includes instructions for multiple Revision Control lab exercises. There are six labs addressing the following design and data types:

- Lab 1: Simple RTL Project
- Lab 2: Simple IP Project

- Lab 3: Custom IP Project
- Lab 4: HLS-Based IP
- Lab 5: IP Integrator Block Design
- Lab 6: Top-Level Integration

Locating the Tutorial Design Files

There are separate project files, design sources, and scripts for each of the labs in this tutorial. You can find the design files required for this tutorial in the associated reference design file.

1. Download the zipped reference file from the Xilinx website.
2. Extract the zip file contents into any write-accessible location on your hard drive, or network location.

The extracted source directory is referred to as `<Extract_Dir>` throughout this tutorial.

TIP: *You will modify the tutorial design data while working through this tutorial. You should use a new copy of the* ***<Extract_Dir>*** *directory each time you start this tutorial.*

The scripts provided in this tutorial are intended to run on Windows. You can modify scripts and Makefiles to run on Linux if necessary.

Installing Tutorial Utilities

Since revision control is the central concept for this tutorial, all six labs will use the same revision control repository and working directory. The `<Extract_Dir>` provides a root directory where you will store the tutorial source files, scripts, and working directory. For example:

```
C:\labs\revCtrl
```

1. Copy or extract the lab files into this directory. The following files are provided with the labs, organized in these directories:
   - `scripts`: Vivado run scripts and Makefiles
   - Design source code required for the labs is further organized into the following directories:
     - `hdl`: Verilog HDL including the top-level design
     - `xdc`: Xilinx Design Constraints for the top-level design
     - `hls`: C++ design
     - `dsp`: SysGen design

2. Create a working directory called `work` inside the root directory (`mkdir work`). The `work` directory is not under revision control, but this is where you will run Vivado and store files temporarily before checking work into the revision control system.

   You will be using the Windows command shell for these labs; this gives you the ability to:

   - Run Vivado Design Suite.
   - Run `make`.
   - Use Git commands for managing files under revision control.

TIP: *You can install GitHub for Windows to install a Git shell that you can use for these labs as an alternative to running Git from the Windows command line. GitHub for Windows can be downloaded from: https://desktop.GitHub.com.*

3. Open a Windows command shell, change to the `<Extract_Dir>` directory, and run the **env.bat** file in the scripts directory to set up the environment for running Vivado, `make` and Git commands:

   ```
   C:\labs\revCtrl>scripts\env.bat
   ```

   You will see some commands echoed to the shell as shown in the following figure:

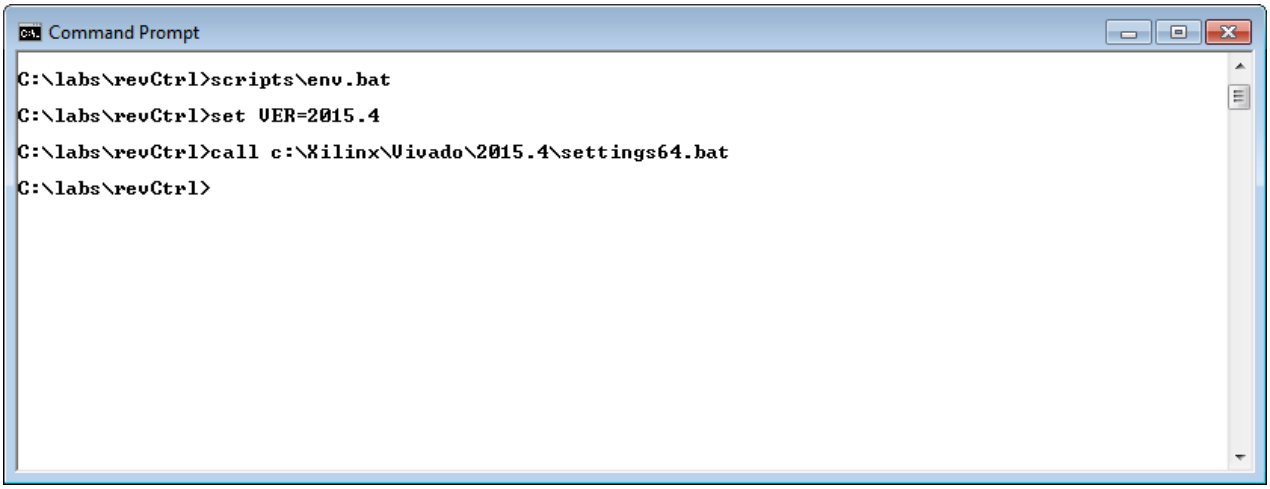

Figure 1: Setting up the shell environment

IMPORTANT: *If you did not install Vivado in the default location, modify the `env.bat` to match your environment.*

4. Examine the `env.bat` file to see that it adds the Minimalist GNU for Windows (MinGW) commands (`msys`, `git`, `vim`) to your path so you can run the `make` utility to call Makefiles in the Windows shell. It also sets command aliases to create Linux like commands such as `ls`, `cp`, `rm`, `mv`, and `find` for users who prefer working in a Linux like shell:

```
REM adding msys, git, and vim to the path env variable
SET PATH=C:\Xilinx\Vivado\%VER%\tps\win64\git-1.9.5\bin;%PATH%
SET PATH=C:\Xilinx\Vivado\%VER%\tps\share\vim\vim74;%PATH%
SET PATH=C:\Xilinx\Vivado_HLS\%VER%\msys\bin;%PATH%
REM some useful aliases to work better in linux
%SYSTEMROOT%\System32\doskey.exe ll=ls -altr $*
%SYSTEMROOT%\System32\doskey.exe vi=vim -N $*
%SYSTEMROOT%\System32\doskey.exe which=sh -c "which $*"
```

5. After calling the script, test your setup by verifying that you can run Vivado Design Suite, the `make` utility, and Git from the command line:

```
C:\labs\revCtrl>vivado -version
Vivado v2015.4 (64-bit)
SW Build 1389237 on Tue Nov 10 17:35:58 MDT 2015
IP Build 1389227 on Tue Nov 10 18:03:40 MDT 2015
Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

C:\labs\revCtrl>make
make: *** No targets specified and no makefile found.  Stop.

C:\labs\revCtrl>git --version
git version 1.9.5.msysgit.0
```

At this point, you have installed the tutorial scripts and design source files and setup the Git and `make` utilities. You are ready to begin the labs.

Introduction

You will start in Lab 1 with a very simple design, called `top`, to see how revision control works with the Vivado Design Suite. The goal is to identify the minimum set of files needed to recreate the `top` design and place those files under revision control.

TIP: *You will use the project-based flow for simplicity throughout this tutorial, but you can also apply the concepts discussed here to non-project flows. For more information on project and non-project design modes refer to this link in the Vivado Design Suite User Guide: Design Flows Overview (UG892).*

The following files are needed to regenerate the top project and the bitstream for the design. These files should be placed under revision control:

- scripts: `setup_simple.tcl`, `compile.tcl`, `Makefile_simple`
- hdl: `top_simple.v`, `threeflop.v`
- xdc: `top.xdc`, `top_io_simple.xdc`

Lab Procedure

1. From within the command shell you setup in Installing Tutorial Utilities, navigate to the `scripts` directory and view the file `setup_simple.tcl`.

   This script creates a Vivado project and adds the source files to that project:

```
# get the directory where this script resides
set thisDir [file dirname [info script]]
# source common utilities
source -notrace $thisDir/utils.tcl

set hdlRoot ../hdl
set xdcRoot ../xdc

# Create project
create_project -force top ./top/ -part xc7k325tffg900-2

# Set project properties
set obj [get_projects top]
set_property "board_part" "xilinx.com:kc705:part0:1.2" $objset_property
"simulator_language" "Mixed" $obj
set_property "target_language" "Verilog" $obj
```

```
add_files -norecurse $hdlRoot/top/top_simple.v
add_files -norecurse $hdlRoot/threeFlop/threeFlop.v
add_files -norecurse $xdcRoot/top.xdc
add_files -norecurse $xdcRoot/top_io_simple.xdc

# If successful, "touch" a file so the make utility will know it's done
touch {.setup.done}
```

2. From the <Extract_Dir>, change to the work directory where you will create and manage your design projects:

```
C:\labs\revCtrl>cd work
```

3. From the command prompt launch the Vivado Design Suite and source the setup_simple.tcl script to create the project and add the design files:

```
C:\labs\revCtrl\work>vivado -mode batch -source
..\scripts\setup_simple.tcl
```

Notice the script uses add_files to add the source files to the project. This adds the specified file to the project by referencing it from its current location, in a library of source files for instance. If you use the import_files command instead, the file is copied and imported into the local project directory structure. This results in a new copy of the source file, which you must manage under revision control. Managing multiple copies of the same source file can create revision control and design management issues. When working with revision control systems, it is simpler to keep source files outside of the Vivado project directory structure. It is not recommended to put the Vivado project directory in revision control.

TIP: *For more information on referencing design files or copying them into your project, refer to this link in the Vivado Design Suite User Guide: System-Level Design Entry (UG895).*

The setup_simple.tcl script also "touches" or creates and empty file called .setup.done. This file is a status file used by the make utility to determine if the setup step is complete. The timestamp on the file can also be used to determine when the setup step was performed relative to some of the other steps in the flow.

4. From the scripts directory, inspect the contents of the compile_simple.tcl script:

```
# get the directory where this script resides
set thisDir [file dirname [info script]]
# source common utilities
source -notrace $thisDir/utils.tcl

# Create project
open_project ./top/top.xpr

# Implement and write_bitstreamlaunch_runs impl_1
wait_on_run impl_1
open_run impl_1

write_bitstream top.bit
```

```
# If successful, "touch" a file so the make utility will know it's done
touch {.compile.done}
```

The `compile_simple.tcl` script opens the top project and runs synthesis and implementation, and generates a bitstream for the design. The two files that are significant in this step are the bitstream file, `top.bit`, and the status file, `.compile.done`, which is created at the end of the script. The timestamp on the `.compile.done` file is used by `make` to manage dependencies. If `.compile.done` file is missing, or is dated earlier than `.setup.done`, the compile target is made.

5. Copy the make file `Makefile_simple` from the scripts directory into the work directory and rename it to `Makefile`.

6. Examine the contents of the `Makefile` script:

```
# these are the sources - everything depends upon them
RTL=../hdl/top/top.v ../hdl/threeFlop/threeFlop.v
XDC=../xdc/top.xdc ../xdc/top_io_simple.xdc

# These are all the compilation targets, starting with "all"
all : setup compile

# Setup the top level project
setup : .setup.done
.setup.done : $(RTL) $(XDC)
   cmd /c "vivado -mode batch -source ../scripts/setup_simple.tcl -log
setup.log -jou setup.jou"

compile : .compile.done
.compile.done : .setup.done
   cmd /c "vivado -mode batch -source ../scripts/compile_simple.tcl -log
compile.log -jou compile.jou"

# delete everything except this Makefile
clean :
   find . -not -name "Makefile*" -not -name "." | xargs rm -rf
```

The four targets contained within the Makefile are: `all`, `setup`, `compile`, and `clean`. Two of the targets require the scripts you previously ran to create the Vivado Design Suite project and generate the bitstream for the design: `setup_simple.tcl` and `compile_simple.tcl`. Please see Appendix B: Introduction to the Make Utility for more information on `make` and the Makefile syntax. Notice that the compile target depends on `.compile.done`, which depends on `.setup.done`, which in turn depends on the `$(RTL)` and `$(XDC)` variables.

Using variables in the Makefile allows the RTL and XDC files to be specified in a single location, and the variables to be reused as needed. This also makes the overall Makefile more readable, and more easily edited. The syntax for referencing a variable in the Makefile, such as the RTL variable, is $(RTL).

The first character of a line containing a Makefile command is the tab character. This is the default character that the `make` utility uses to identify rules.

CAUTION! *The whitespace at the beginning of each command must be a tab character, not spaces. Sometimes copying and pasting text in an editor will cause a tab character to be replaced with multiple spaces. This can cause errors in your Makefile.*

The entire build process can be managed using `make` since it describes the dependency relationships and the "rules" to make the targets.

7. Experiment with the `make` utility using the Makefile, and generate a few targets. Clean the work directory then make the `setup` target:

```
C:\labs\revCtrl\work>make clean
C:\labs\revCtrl\work>make setup
```

TIP: *Because the compile target depends on the setup target, if you make the compile target before the setup target, the make utility makes the setup target first.*

8. Make the `compile` target, followed by `all`:

```
C:\labs\revCtrl\work>make compile
C:\labs\revCtrl\work>make all
```

Because the 'all' target includes both setup and compile, which are already made, the make utility reports, "`Nothing is to be done for all.`"

The top project is made, and ready to be checked into the Git repository.

9. From the work directory, change to the root directory, or `<Extract_Dir>`:

```
C:\labs\revCtrl\work>cd ..
```

10. Create the Git repository in the root folder:

```
C:\labs\revCtrl>git init
```

11. Check the status of the repository:

```
C:\labs\revCtrl>git status
```

It should report a newly created repository with untracked files.

TIP: *It is a good idea to run `git status` after each step to ensure the repository state is as expected.*

12. Check in the following directories: (never check in `work`):

```
C:\labs\revCtrl>git add dsp hdl hls scripts xdc tb
C:\labs\revCtrl>git status
C:\labs\revCtrl>git commit -m "Initial Top Project checkin"
```

Note*: When running `git status` above, notice what the repository looks like with pending changes before running the commit process.*

13. After all the files are successfully checked-in, clean the work directory:

```
C:\labs\revCtrl>cd work
C:\labs\revCtrl\work>make clean
```

14. You can also remove the Makefile at this time, as it is only used for Lab 1.

Conclusion

This first lab illustrates the process of managing design files under revision control. In this lab you:

- Worked with Tcl scripts to create a simple Vivado design suite project and generated the bitstream for the design.
- Used a Makefile to define targets and dependencies, and to run those scripts to build the targets.
- Created a Git repository to check-in the recommended files into revision control.
- In the following labs, you will build on the concepts learned here to explore recommendations for managing different types of design data under revision control.

Introduction

The goal of this lab is to become familiar with the process of managing standalone IP from a managed IP project under a revision control system. You will generate the output products for an AXI IIC Bus Interface IP using a scripted flow and place the resulting files under revision control, where you can reuse them in other designs.

For more information on using IP from the Vivado IP Catalog, and the managed IP flow, refer to the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Lab Procedure

1. From within a command shell that you configured as described in Installing Tutorial Utilities, navigate to the `scripts` directory and inspect the `ip.tcl` script:

```
# get the directory where this script resides
set thisDir [file dirname [info script]]
# source common utilities
source -notrace $thisDir/utils.tcl

set ipDir ./ip

create_project -force managed_ip_project $ipDir/managed_ip_project -part
xc7z020clg484-1 -ip
# Set project properties
set obj [get_projects]
set_property "board_part" "xilinx.com:zc702:part0:1.0" $obj
set_property "simulator_language" "Mixed" $obj
set_property "target_language" "Verilog" $obj
# set_property coreContainer.enable 1 $obj

set_property target_simulator XSim [current_project]
create_ip -name axi_iic -vendor xilinx.com -library ip -module_name
axi_iic_0 -dir $ipDir
set_property -dict [list CONFIG.AXI_ACLK_FREQ_MHZ {100}] [get_ips
axi_iic_0]

generate_target all [get_files  axi_iic_0.xci]

export_ip_user_files -of_objects [get_files axi_iic_0.xci] -no_script -
ip_user_files_dir $ipDir -force -quiet

create_ip_run [get_files -of_objects [get_fileset sources_1] [get_files
*/axi_iic_0.xci]]
launch_run  axi_iic_0_synth_1
```

```
wait_on_run axi_iic_0_synth_1

export_simulation -of_objects [get_files axi_iic_0.xci] -force -quiet

# If successful, "touch" a file so the make utility will know it's done
touch {.ip.done}
```

This script creates a managed IP project, instantiates an AXI IIC IP into the project using the `create_ip` Tcl command, generates the output products for the IP using the `generate_target` Tcl command, enables the core container feature, and launches synthesis for the IP. For more information about the core container feature, see this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896).

TIP: *To generate a script like this, you could begin by using the Vivado IDE to walk through the steps, and then use the Tcl commands that are written to the `vivado.jou` journal file as the starting point for your script. For more information, see the Vivado Design Suite User Guide: Using Tcl Scripting (UG894).*

Notice the `set_property -dict` command customizes the IP in the managed IP project. The `-dict` option applies the arguments as name-value pairs like the Tcl dict structure: a series of property names and property values.

In the script above, you can see that the managed IP project references the `$ipDir` variable, which is defined as `./ip` in the `work` directory. Inside the `ip` directory is where the `managed_ip_project`, and `axi_iic_0` IP instance will be found.

2. Change to the `work` directory and copy the file Makefile from the scripts to the `work` directory.

   ```
   C:\labs\revCtrl>cd work
   C:\labs\revCtrl\work>cp ../scripts/Makefile .
   ```

   This `Makefile` is used for the remaining labs of the tutorial, and is therefore more complicated than the `Makefile_simple` used in Lab 1.

3. Review the `Makefile` to become familiar with its content and conventions.

   Notice the use of variables to define source files for the project, as well as defining the OS in use, and various tool options. There are many more targets defined in this `Makefile` including: all, setup, compile, sim, bd_gen, ip, cip, hls, and clean.

4. From within the `work` directory, make the AXI IIC IP target, using the command:

   ```
   make ip
   ```

   Just as in Lab 1, the script here creates an empty file, `.ip.done`, to indicate the successful completion of the script for the make utility. Since `.ip.done` is a single file created at the end of IP generation, `make` can check this single file to determine if dependencies exist or need to be updated.

For other designers to use the customized AXI IIC IP, and the generated output products, in their designs, all of the output products from the `axi_iic_0` instance in the managed IP project are needed. These are the files that you will place under revision control. However, if only the customized IP is reused, and the output products will be regenerated, then only the XCI file, `axi_iic_0.xci`, needs to be placed under version control. The rest of the required output products can be regenerated.

TIP: *Labs #3 and #4 in the Vivado Design Suite Tutorial: Designing with IP (UG939) provide an example of adding existing IP XCI files into a new design, with and without the supporting output products. IP with the required output products can be used as configured in a new design. IP without the required output products will become locked, and will need to be updated to the latest version in the Vivado IP Catalog in order to regenerate the required output products. For more information on why IP can become locked, refer to this link in the Vivado Design Suite User Guide: Designing with IP (UG896).*

Now you will place the customized IP, and its generated output products under revision control. In the lab `root` directory, or `<Extract_Dir>`, you will see a directory called `ip`. In the `work` directory there should also be a directory called `ip`, which is the managed IP project containing two sub-directories: `axi_iic_0` and `managed_ip_project`.

To add the IP directory to the revision control repository, copy the `axi_iic_0` directory from the `work/ip/` directory to the `<Extract_Dir>/ip/` directory as the design sources are preserved outside of the working directory. Do not copy the `managed_ip_project` as it is not recommended to put Vivado projects into revision control systems.

> ***Note:*** *To use the core container feature, edit the `ip.tcl` file. Uncomment out the `set_property coreContainer.enable 1 $obj` line by removing the preceding `#`. This creates an `axi_iic_0.xcix` file under revision control, not the `axi_iic_0` directory.*

5. From the root directory, copy the `work/ip/axi_iic_0` folder to the `<Extract_Dir>/ip` folder:

```
C:\labs\revCtrl\work>cd ..
C:\labs\revCtrl>cp -r work/ip/axi_iic_0 ip/axi_iic_0
```

6. Next, add the `<Extract_Dir>/ip` directory (and all the IP instances contained within it) to the Git revision control systems:

```
C:\labs\revCtrl>git add ip
C:\labs\revCtrl>git commit -m "AXI IIC IP Checkin"
```

7. Check the status of the Git repository:

```
C:\labs\revCtrl>git status
```

Note that the `work` directory should still be reported as untracked.

TIP: *It is a good idea to run* `git status` *after each step to ensure the repository state is as expected.*

Conclusion

This lab extends the basic concepts of scripting and revision control to cover generated IP.

Introduction

The goal of this lab is to become familiar with managing custom, user-defined IP under revision control. Custom IP is different from the standard IP provided in the Vivado IP Catalog. There is a different set of design files to manage in this case.

The three types of files needed to reuse a custom IP include the: design source files (for example HDL, XDC), the `component.xml` that defines the customizable IP parameters, and the `xgui` directory and contents that define the IP symbol.

In this lab, you package the BFT example design as custom IP, and place the results into the revision control system for source management. You then add this user-defined IP to an IP repository for use in other designs. For more information on packaging custom IP see the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118).

Lab Procedure

1. From within a command shell that you configured as described in Installing Tutorial Utilities, navigate to the `scripts` directory and inspect the `cip.tcl` script:

```
# get the directory where this script resides
set thisDir [file dirname [info script]]
# source common utilities
source -notrace $thisDir/utils.tcl

# create the directories to package the IP cleanly
if {![file exists ./cip]} {
   file mkdir ./cip
}
if {![file exists ./cip/bft]} {
   file mkdir ./cip/bft
}
if {![file exists ./cip/bft/bftLib]} {
   file mkdir ./cip/bft/bftLib
}
foreach f [glob ../hdl/bft/*.v*] {
   file copy -force $f ./cip/bft/
}
foreach f [glob ../hdl/bft/bftLib/*.vhdl] {
   file copy -force $f ./cip/bft/bftLib/
}
```

```
# Create project
create_project -force bft ./bft/ -part xc7z020clg484-1
add_files -norecurse [glob ./cip/bft/*.v*]
add_files -norecurse [glob ./cip/bft/bftLib/*.vhdl]
set_property library bftLib [get_files */round_*.vhdl]
set_property library bftLib [get_files */bft_package.vhdl]
set_property library bftLib [get_files */core_transform.vhdl]

update_compile_order -fileset sources_1
update_compile_order -fileset sim_1

ipx::package_project -root_dir ./cip/bft

# If successful, "touch" a file so the make utility will know it's done
touch {.cip.done}
```

Notice in the `cip.tcl` script, the directory structure for containing the BFT source files is created in the `./cip` folder, and the files are copied from the HDL library into the `cip` directory for synthesis and simulation of the custom IP.

2. Change directory to `work` and make the AXI IIC IP target, using the following commands:

```
C:\labs\revCtrl>cd work
C:\labs\revCtrl\work>make cip
```

The `cip.tcl` script creates an empty file, `.cip.done`, to indicate the successful completion of the script for the make utility. The new custom IP has been created and packaged.

In Lab 4, you will add this custom IP to a block design. Test the IP at this time to ensure it will work as expected by adding it to a simple project.

3. From the `work` directory, launch the Vivado Design Suite in interactive mode:

```
C:\labs\revCtrl\work>vivado
```

4. After Vivado opens, select **Create New Project** on the Getting Started page.
5. Click **Next** in the New Project wizard.
6. On the Project Name page, specify the Project Name and Location:
   a. Project name: **`project_1`**
   b. Project Location: ***`<Extract_Dir>/work`***
   c. Check the **`Create project subdirectory`** box.
   d. Click **Next**.
7. On the Project Type page, select **RTL project**, enable **Do not specify sources at this time**, and click **Next**.
8. On the Default Part page, select the device **xc7z020clg484-1**, and click **Next**.

9. Review the New Project Summary page to ensure it reflects your choices, and click **Finish**.

   The Vivado Design Suite creates the `project_1`, and opens it in the Vivado IDE.

10. Under the Flow Navigator, select the **IP Catalog** to open the Vivado IP Catalog.

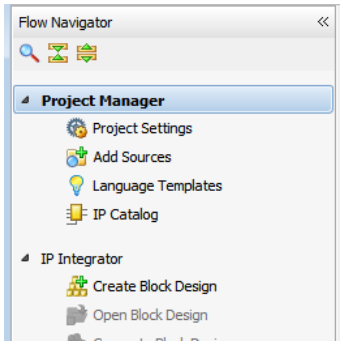

Figure 2: Select IP Catalog

11. In the IP Catalog window, right-click and select **Add Repository**.

    A Repositories file browser opens, letting you select folders to add to the IP repository that the Vivado IP Catalog uses to locate IP.

12. Select the `<Extract_Dir>/work/cip/bft` folder to add, and click **Select**.

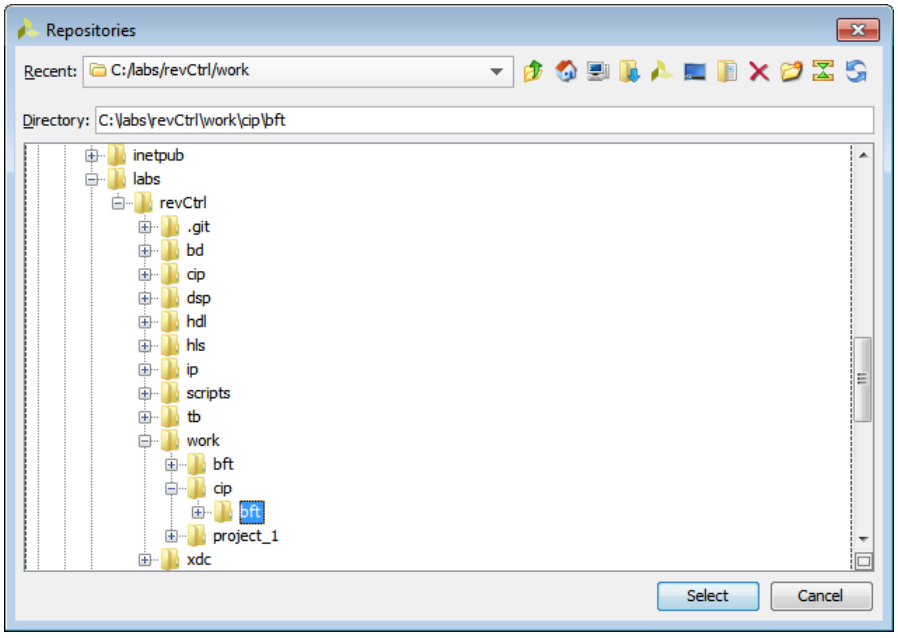

Figure 3: Add IP Repository

The Add Repository confirmation dialog box opens, showing that the selected folder and one IP and zero interfaces will be added to the IP Catalog.

13. Click **OK**.

    You should now see the `bft_v1_0` IP shown under the User Repository heading in the IP Catalog, as shown in the following figure.

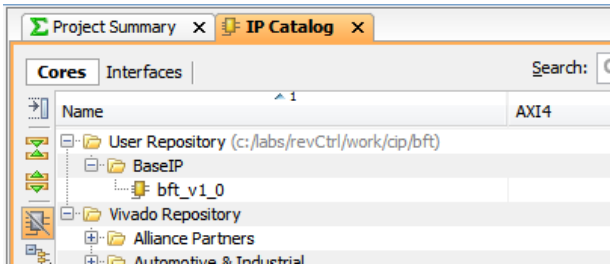

Figure 4: User Repository

14. From the Flow Navigator select **Create Block Design**.
15. In the Create Block Design dialog box, specify **design_bft** for the design name, and click **OK**.
16. In the Vivado IP integrator design canvas, select **Add IP**, and search for the **bft** IP as shown in the following figure.

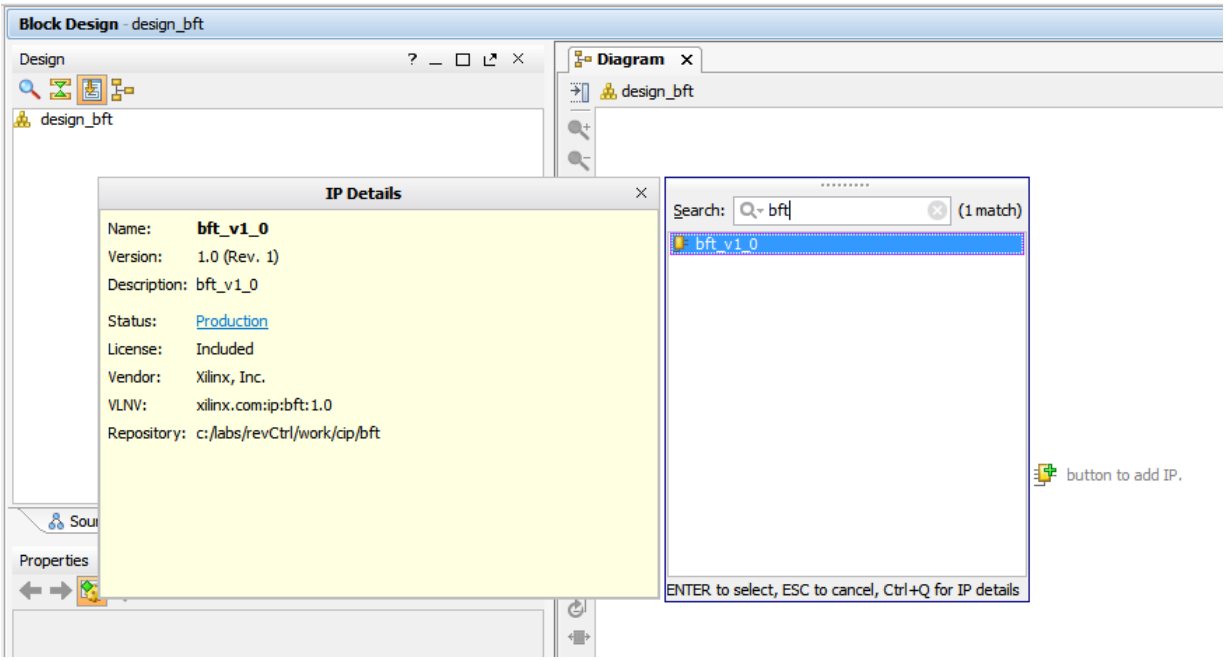

Figure 5: Add BFT IP

The bft IP symbol is added to the block design. Other IP can be added to the block design, and connected to the BFT IP as needed. Having validated the BFT IP to this degree, you can exit Vivado without saving the block design or the project.

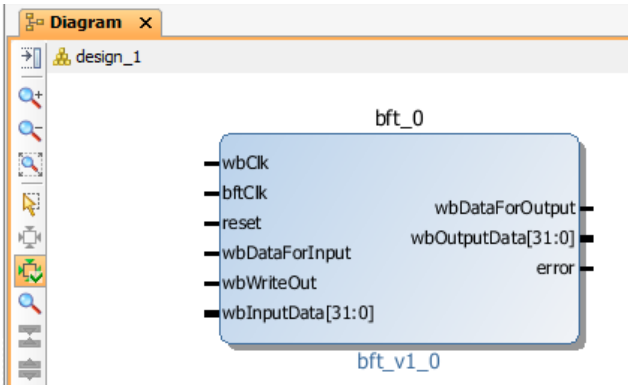

Figure 6: BFT IP

You will now add the files for the custom BFT IP to the revision control system. As you did in Lab 2, you will copy the custom IP into the `<Extract_Dir>/cip` folder to add the files to revision control.

17. From the root directory, copy the `work/cip/bft` folder to the `<Extract_Dir>/cip` folder:

```
C:\labs\revCtrl\work>cd ..
C:\labs\revCtrl>cp -r work/cip/bft cip
```

18. Next, add the `<Extract_Dir>/cip` directory (and all the IP instances contained within it) to the Git revision control systems:

```
C:\labs\revCtrl>git add cip
C:\labs\revCtrl>git commit -m "BFT Custom IP Checkin"
```

19. Check the status of the Git repository:

```
C:\labs\revCtrl>git status
```

Note that the `work` directory should still be reported as untracked.

Conclusion

This lab extends the basic concepts of scripting and revision control to cover a custom IP built from the BFT example design.

Introduction

The goal of this lab is to become familiar with packaging a Vivado High-Level Synthesis (HLS) project into a custom IP. The HLS `rgb_mux` design is synthesized and packaged as custom IP. You will then place it under revision control.

The IP generation includes scripting the Vivado HLS run to synthesize it and generate the custom IP from the C++ source files. You can then add the custom IP to the project IP Catalog repositories to be used in other designs.

Lab Procedure

1. From within a command shell that you configured as described in Installing Tutorial Utilities, navigate to the `scripts` directory and inspect the `hls.tcl` script.

   This script use the Vivado HLS feature, synthesizes the rgb_mux design, and packages the results as a custom IP. Notice the HLS command `export_design`, in the `hls.tcl` script, packages the design as custom IP.

   Vivado HLS automatically records a script for recreating a design project. In the `<Extract_Dir>/hls` folder, the header of the `script.tcl` file shows that it was automatically created by the tool. This automatically created script would be a good place to start in creating a custom script for HLS.

2. Use make to generate `rgb_mux` in the `work` directory:

   ```
   C:\labs\revCtrl>cd work
   C:\labs\revCtrl\work>make hls
   ```

   Upon completion, note that the `rgb_mux` custom IP is located in the `work/rgb_mux/solution_zc702/impl/ip` folder. Just like the BFT custom IP in Lab 3, you use the newly-created `rgb_mux` IP in a block design. Test the IP by adding it to the IP repository and placing it in a block design.

3. Open the test project, `project_1`, that you created in Lab 3:

   ```
   C:\labs\revCtrl\work>vivado project_1/project_1.xpr
   ```

4. Under the Flow Navigator, select **IP Catalog,** right-click in the IP Catalog window and select **Add Repository**.

   A Repositories file browser opens, letting you select folders to add to the IP repository that the Vivado IP Catalog uses to locate IP.

5. Select the `<Extract_Dir>/work/cip/rgb_mux` folder to add, and click **Select**.

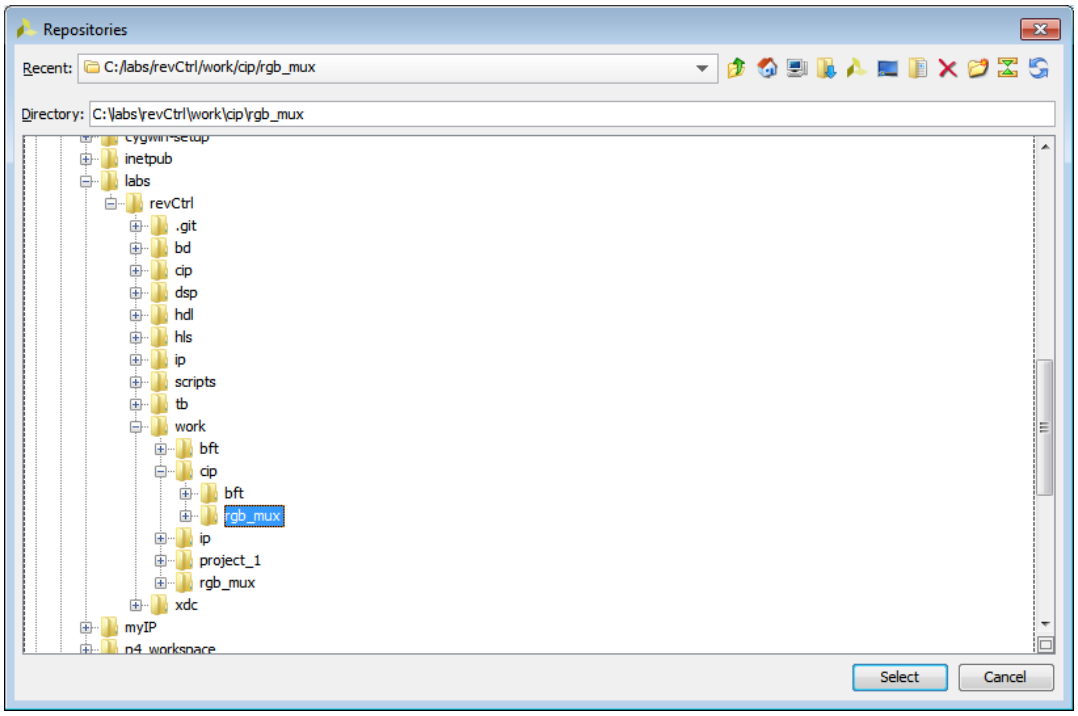

Figure 7: Add IP Repository

The Add Repository confirmation dialog box opens, showing that the selected folder and one IP and zero interfaces will be added to the IP Catalog.

6. Click **OK**.

You should now see the `rgb_mux` IP shown under the User Repository heading in the IP Catalog, as shown in the following figure.

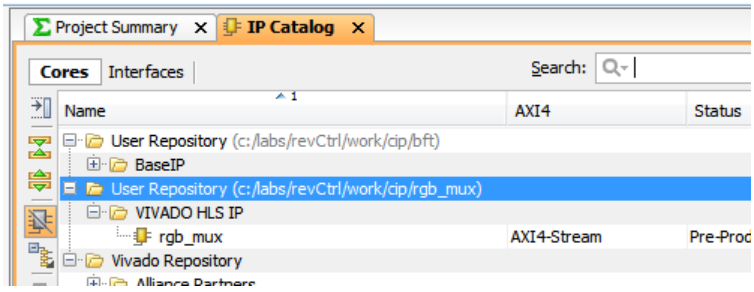

Figure 8: User Repository

7. From the Flow Navigator select **Create Block Design**.
8. In the Create Block Design dialog box specify **design_rgb** for the design name, and click **OK**.
9. In the Vivado IP integrator design canvas, select **Add IP**, and search for the **rgb_mux** IP as shown in the following figure.

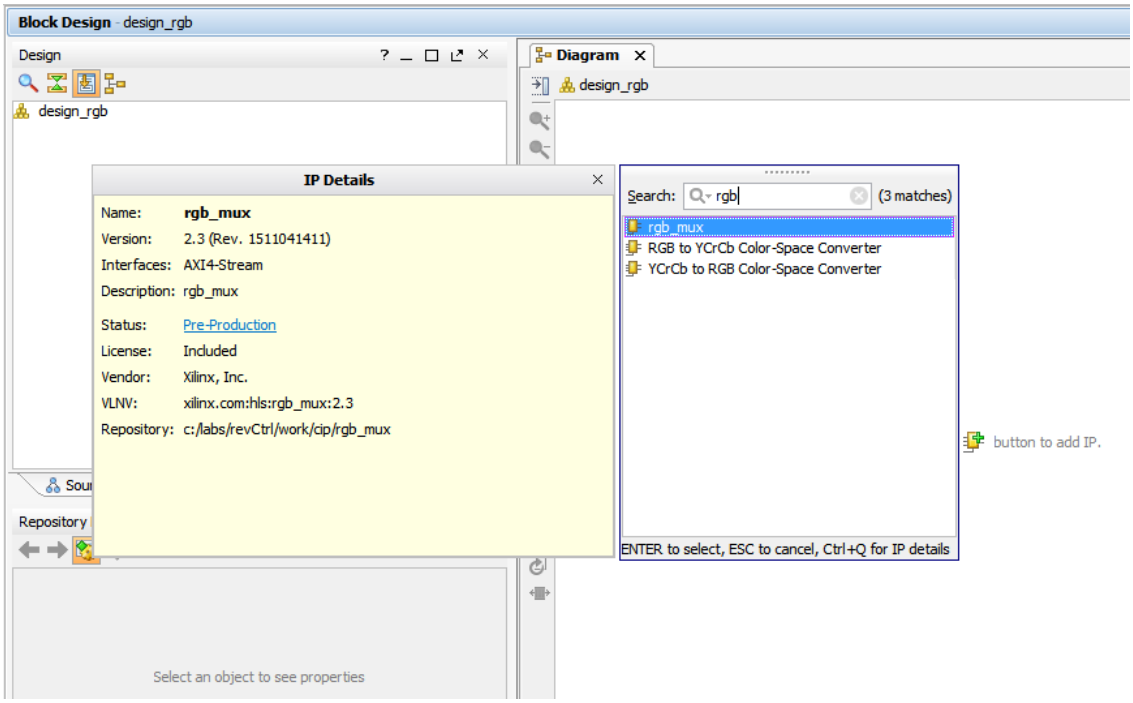

Figure 9: Add BFT IP

The `rgb_mux` IP symbol is added to the block design. Having validated the IP to this degree, you can exit the Vivado IDE without saving the block design or the project.

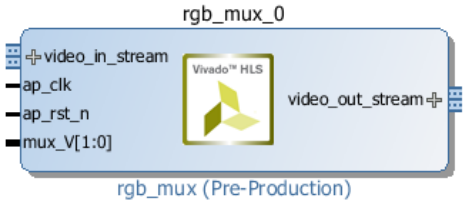

Figure 10: BFT IP

Now add the files for the custom `rgb_mux` IP to the revision control system. As you did in Lab 3, copy the custom IP into the `<Extract_Dir>/cip` folder to add the files to revision control.

10. From the root directory, copy the `work/cip/rgb_mux` folder to the `<Extract_Dir>/cip` folder:

```
C:\labs\revCtrl\work>cd ..
C:\labs\revCtrl>cp -r work/cip/rgb_mux cip
```

11. Next, add the `<Extract_Dir>/cip` directory (and all the IP instances contained within it) to the Git revision control systems:

```
C:\labs\revCtrl>git add cip
C:\labs\revCtrl>git commit -m "RGB_MUX HLS IP Checkin"
```

12. Check the status of the Git repository:

```
C:\labs\revCtrl>git status
```

    Note that the `work` directory should still be reported as untracked.

Conclusion

This lab illustrates that creating custom IP from HLS is a straightforward process that can be entirely scripted and generated using make, and the results managed under revision control for reuse in other designs.

Introduction

In this lab, you will focus on an IP integrator block design: automating the generation of the `zynq_bd` block design, using scripts, and placing the needed files under revision control. The `zynq_bd` block design uses the IP created in Labs 3 and 4, as well as other IP from the IP Catalog.

Output products are generated for a block design, similar to an IP, so that the block design can be reused in other designs.

Lab Procedure

The block design generation may take a bit longer to run than the previous labs. Use the Makefile to generate the block design, and then proceed with the next steps while the Vivado® IDE runs in the background.

1. From within a command shell that you configured as described in Installing Tutorial Utilities, change directory to `work` and make the block design:

   ```
   C:\labs\revCtrl>cd work
   C:\labs\revCtrl\work>make bd_gen
   ```

TIP: *If your working directory is becoming cluttered, you can also clean it:* `make clean`

2. From within the `<Extract_Dir>/scripts` folder, familiarize yourself with the `bd_gen.tcl` script which generates the `zynq_bd` block design that can be reused in another design.

   The block design includes many IP blocks, including the custom IP generated in prior labs. The Tcl commands in the script that add the custom IP to the Vivado IP Catalog are as follows:

   ```
   set_property ip_repo_paths {../cip/bft ../cip/rgb_mux} [current_fileset]
   update_ip_catalog
   ```

The zynq_bd block design is actually generated by another script, bd_cip.tcl, which is sourced from bd_gen.tcl. Notice that bd_gen.tcl provides some logic for supporting earlier versions of the Vivado tool:

```
if {$currVer eq "2015.3"} {
   source $thisDir/bd_cip_2015_3.tcl
} else {
   # this script will only work with 2015.4
   source $thisDir/bd_cip.tcl
}
```

This is because block design Tcl scripts are generally valid for the specific release they are written for, because the parameters and configuration of block designs are release specific. For more information on writing block design Tcl see this link in the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994).

The output products of the zynq_bd block design, needed to support the reuse of the clock design in other designs, are generated by the following command sequence:

```
# Prepare as out-of-context run
create_fileset -blockset -define_from zynq_bd zynq_bd
# Generate output products similar to IP
generate_target all [get_files */zynq_bd.bd]
```

After the block design has been made, examine the work directory to see two new directories created in the process: zynq and zynq_bd. Inspect the contents of each of these directories; note that zynq contains a Vivado Design Suite project, zynq.xpr, which was used to create the block design, zynq_bd. To manage the source files for the block design, place the zynq_bd folder under revision control. Only zynq_bd contains the relevant files to reuse the generated block design.

3. Launch the Vivado IDE on the zynq project:

```
C:\labs\revCtrl\work>vivado zynq/zynq.xpr
```

You mightnotice that the zynq project does not contain any top-level design, or other design sources beyond the zynq_bd. This is because the project was created solely to generate the output products required to support the reuse of the zynq_bd block design.

4. Open and view the block design. You can export the block design as a Tcl script using the **File > Export > Export Block Design** command, or the following Tcl command from the Tcl Console:

```
write_bd_tcl -force
C:/labs/revCtrl/work/zynq/zynq_bd.tcl
```

The block design Tcl script contains all the Tcl commands to recreate the block design from scratch. It is a convenient way to manage block designs under revision control. In fact, it is the basis for the bd_cip.tcl script used in this lab.

5. Now add the files for the `zynq_bd` block design to the revision control system. As you did in prior labs, copy the required files into the `<Extract_Dir>/bd` folder to add the files to revision control.
6. Exit Vivado to continue.
7. From the root directory, copy the `work/zynq_bd` folder to the `<Extract_Dir>/bd` folder:

   ```
   C:\labs\revCtrl\work>cd ..
   C:\labs\revCtrl>cp -r work/zynq_bd bd
   ```

8. Next, add the `<Extract_Dir>/bd` directory, and all its files, to the Git revision control systems:

   ```
   C:\labs\revCtrl>git add bd
   C:\labs\revCtrl>git commit -m "ZYNQ_BD Block Design Checkin"
   ```

9. Check the status of the Git repository:

   ```
   C:\labs\revCtrl>git status
   ```

   Note that the `work` directory should still be reported as Untracked.

Conclusion

This lab demonstrates the automated process for creating a block design in the Vivado IP integrator feature, and placing the contents under revision control. The block design is more complex than the custom IP from the prior labs, as the block design can contain multiple IP, and requires a block design description. Output products are generated for the block design in a manner similar to that of an IP and the output products can be placed under revision control to be reused in other designs.

Introduction

This final lab pulls the many elements of the prior labs together. The top design here contains:

- The `zynq_bd` block design, which contains the IP and custom IP from the earlier labs.
- Several Verilog RTL modules including the top-level design.
- A System Generator (SysGen) design generated standalone and integrated as a design checkpoint (DCP) file into the design.

Just as in the prior labs, you will automate the generation of the top-level design, and place necessary files under revision control.

Lab Procedure

The generation of the top-level design for this lab is automated using Vivado scripts and the Makefile.

1. In the work directory, open the Makefile in a text editor for viewing.

   The targets defined in the Makefile for the top-level design include `all` and `compile`.

2. In the Makefile, find the boolean variable `REUSEGOLDEN`.

   When `REUSEGOLDEN` is `TRUE`:

   - The dependencies of the setup target in the Makefile include only the `$(IP)` and `$(BD)` sources. If the sources are not out of date, then the generated, golden results are used.
   - The `setup.tcl` script is called with the option **-**`tclargs reuseGolden`**.**

   When `REUSEGOLDEN` is `FALSE`:

   - The dependencies of the setup target include not only the `$(IP)` and `$(BD)` sources but also the `done` target files. It is assumed that the generated results are never stored and always generated from the source files.
   - The `setup.tcl` script is called without the option **-**`tclargs reuseGolden`**.**

3. Look at the scripts folder, and examine the `setup.tcl` file for the `reuseGolden` argument:

```
if {[llength $argv] > 0 && "$argv" eq "reuseGolden"} {
   set reuseGolden 1
} else {
   set reuseGolden 0
}
```

   Without the `reuseGolden` argument, the `setup.tcl` script looks for the `ip` and `bd` in the local working directory, `work`. When `reuseGolden` argument is present, the script looks in the `<Extract_Dir>` root directory, which is the Git repository, for the versions of the files which are under revision control.

4. In the `work` directory, edit the Makefile to change the value of `REUSEGOLDEN` to `TRUE`. Save and close the file.

```
REUSEGOLDEN = TRUE
```

5. Clean the work directory to remove all files but the Makefile:

```
C:\labs\revCtrl\work>make clean
```

6. In the `work` directory, make the top-level design (this may take a few minutes):

```
C:\labs\revCtrl\work>make all
```

   In the Makefile, setting `REUSEGOLDEN` to `TRUE` results in potentially faster compile times, due to the fact that the top-level design is using sources from the revision control system, and does not need to regenerate the required files.

7. Once the generation of the top design is complete, explore the contents of the work directory.

   Notice that just the top design and its associated files are in the work directory. This is because the `BFT` and `RGB_MUCX` custom IP, the `ZYNQ_BD` block design, and the other design sources are taken from the revision control system. If you had set `REUSEGOLDEN` to `FALSE`, then you would also expect to see the designs and data files for those objects

   Optionally, try setting the `REUSEGOLDEN` variable to `FALSE` in the Makefile, and see in detail how the make process differs from the previous run.

8. You can also open the top project in the Vivado IDE and view the results:

```
C:\labs\revCtrl\work>vivado top/top.xpr
```

Conclusion

This lab covers the various design file types most likely encountered in real designs, and illustrates how the entire flow can be managed with Vivado Tcl scripts and make. Variables can be used to control the make process as you saw with the different values of `REUSEGOLDEN` and the `tclargs` option.

Using revision control systems, you can choose to reuse as many design sources as possible, or to regenerate as much of the design as needed. In some cases it makes sense to place the source file and all generated output products under revision control, because the customization is very specialized, and the design files won't change once generated. However, in other cases placing just the source file under revision control, and regenerating the output products as needed for each new design can reduce the managed files, and allow for re-customization of the IP or block design in future designs. The approach you choose should depend on your needs, and the design data you are managing.

These concepts are applicable to all types of designs created using Vivado, and other design tools such as simulation. Although this tutorial used Git for revision control, the general strategies also apply to other revision control systems which have similar features.

Git Overview

Git is a free, open source, revision control system that you can find at git-scm.com. You can also find documentation at this link.

The following is a small tutorial to introduce you to Git, and its most commonly used commands. If you are new to Git, please use this exercise to familiarize yourself with the utility before proceeding to the labs. This tutorial requires a command shell that you configured as described in Installing Tutorial Utilities.

TIP: *Although Git is included with the Vivado Design Suite, GitHub Desktop is an alternative option and not required for the labs. GitHub provides a command shell that you can instead of the Windows Command Prompt shell. It can be downloaded from: https://desktop.github.com/. Documentation and installation instructions can be found at this link.*

From a properly configured command shell, use the following command sequence to create a file, check it into the Git repository, modify it, check-in the updated file, delete the local file, and restore the file from the vault.

1. Go to your home directory, or a directory to complete this tutorial. Create a directory for the Git repository at this location (you will delete it later). Change directory to the Git repository, and report the current status of the repository. Git returns a fatal message, that the current directory is not a repository:

   ```
   cd <home>
   mkdir git_temp
   cd git_temp
   git status
   ```

2. Create the Git repository at the current location, and report the status again. This time Git reports nothing to commit:

   ```
   git init
   git status
   ```

3. Create a file, write the string “example1”, and report the status to see there is nothing to commit. Git, however, reports one new file that could be added to be tracked:

   ```
   echo example1 > file.txt
   git status
   ```

4. Stage the `file.txt` for committing to the repository:

```
git add file.txt
```

TIP: *Note that you can stage all files in the current directory and sub-directories by using wildcards like* `git add .` *or* `git add *`

5. Report status, reports `file.txt` as a newly added file that needs to be committed. Commit the file to the repository:

```
git status
git commit
```

   The `commit` command opens a VIM editor for you to record some comments about the check in. You can also use the `-m` option with a string argument to provide the argument automatically.

6. Modify the `file.txt` with different contents, "example2". Report status now shows that `file.txt` has been modified:

```
echo example2 > file.txt
git status
```

   Notice that the `status` command also recommends adding the updated file to the repository.

7. Stage `file.txt` for committing to the repository, and commit the file with the comment included:

```
git add file.txt
git commit -m "changed the file"
```

8. List the current directory to see the `file.txt` file is there:

```
ls
```

9. Remove the file from local disk, and stage it to be deleted from the repository. List the current directory to see the file is removed. The Git status command shows that that `file.txt` is ready to be deleted on the next commit:

```
git rm file.txt
ls
git status
```

   Notice that the status command also suggests a command to restore the removed file, just in case removing it was a mistake.

10. Unstage the `file.txt` file for deletion from the repository. The status report suggests a command to restore the removed file to the local disk:

```
git reset HEAD file.txt
git status
```

11. Undo the removal of the file from the local disk. List the current directory to show the `file.txt` is restored. The Git status command shows nothing to report:

```
git checkout -- file.txt
git status
```

12. Repeat the removal of the file from the local disk, and also stage it to be deleted from the repository. The status command shows that that `file.txt` is ready to be deleted on the next commit. Commit the deletion to remove the file from the repository:

```
git rm file.txt
git status
git commit -m "deleted the file"
```

    The file `file.txt` is now gone from the local disk and the Git repository. However, there is still a chance to restore it.

13. View the summary log of check-ins:

```
git log --summary
```

    This returns a summary report similar to the following:

```
commit b7909b897e33a82e3d1328268b9cb69578cbae5a
Author: Your Name <yourname@company.com>
Date:   Thu Nov 5 18:11:36 2015 -0800

    deleted the file

 delete mode 100644 file.txt

commit d18d97a437fe8874d31d08461fa0bcf660301ad2
Author: Your Name <yourname@company.com>
Date:   Thu Nov 5 18:05:49 2015 -0800

    Changed the File

 create mode 100644 file.txt

commit 77f73e2cd21de802ff2629df73ebfca91813b9c0
Author: Your Name <yourname@company.com>
Date:   Thu Nov 5 18:05:49 2015 -0800

    created the file

 create mode 100644 file.txt
```

    By examining the log file, you can see the internal transaction ID of the various Git transactions committed to the repository. These transactions includes creating, updating, and deleting files.

14. Use the appropriate transaction ID to restore the updated version of the file:

```
git checkout d18d97a437fe8874d31d08461fa0bcf660301ad2 -- file.txt
```

    ***Note*:** Your transaction ID will be different.

15. Examine the contents of the `file.txt` file. This should display the updated file contents, "example2".
16. Report the status of the repository to see that the restored file is available to be committed to the repository. Commit the `file.txt` back into the repository. Report the status again to see that there is nothing to report.

```
git status
git commit -m "restored the file"
git status
```

Conclusion

This should give you an idea of how the common commands work. You will be using the following commands most frequently in the labs:

- `git add`
- `git commit`
- `git status`

However, you can also see from this small exercise that there are many powerful features of Git available for revision control and source file management. When you have finished you can return to your *<home>* directory and remove the practice directory.

```
cd <home>
rm -rf git_temp
```

Appendix B: Introduction to the Make Utility

Make Utility Overview

This is an introduction to the GNU `make` utility that is installed with the Vivado Design Suite. Please review this section if you are new to make. You can find a description of the make command syntax at the following URL: www.gnu.org/software/make/manual/make.html

The `make` utility is widely used to build projects using "Makefiles." Makefiles contain all the "rules", or commands, needed to build "targets" such as Vivado synthesis or implementation outputs, netlists, and device bitstreams. The Makefile rules can also describe process dependencies, based on the timestamps on files, to determine which commands make must run and which can be skipped because the dependency has been satisfied.

You can get a quick view of the various commands offered by the `make` utility using the following command:

```
make -h
```

TIP: *This tutorial requires a command shell that you configured as described in Installing Tutorial Utilities.*

A Simple Example

The Makefile consists of rules with the following structure:

```
target : dependency
   command 1
   command 2
   …
   …
```

`target`: is the name of the object to be created, or made.

`dependency`: A required file or object that must be available and current for the rule to be satisfied, and the target made.

`commands`: One or more commands to be run by the make utility to create the target. Commands can be listed on a single line or on multiple lines.

CAUTION! *The whitespace at the beginning of each command must be a tab character, not spaces. Sometimes copying and pasting text in an editor will cause a tab character to be replaced with multiple spaces. This can cause errors in your Makefile.*

In the following simple example of a Makefile, the `top_synth.dcp` is the target to be made. The target has a dependency on a single Verilog file, `top.v`, and when the dependency is satisfied, the target is made. The command to make the target uses Vivado synthesis, run through the `run_synth.tcl` script to create the design checkpoint.

```
top_synth.dcp : top.v
   cmd /c "vivado -mode batch -source run_synth.tcl"
```

To build the `top_synth.dcp` target, run make from a shell prompt using the following command:

```
make -f <file> top_synth.dcp
```

TIP: *`-f <file>` specifies the name of a file that defines the make rules. If this file is named `Makefile`, and can be found in the current working directory, the file does not need to be specified. If the target is not specified, make will just build the first target in the Makefile.*

If `top_synth.dcp` is out-of-date, older than `top.v`, Vivado synthesis is launched with the Tcl script to regenerate the design checkpoint.

Based on this simple example, where `top_synth.dcp` is the first (or only) target, and the make file is called Makefile, the target can also be made by running the following command:

```
make
```

Multiple targets

If you do not know the output of a target, for example you don't know the name of the file that will be produced, or you don't know what file is generated by the `run_synth.tcl` script, you can create a symbolic target.

In the following rule, the synth target is dependent on the `.synth.done` file, and the `.synth.done` target is dependent on the `top.v` file.

```
synth : .synth.done

.synth.done : top.v
   cmd /c "vivado -mode batch -source run_synth.tcl"
```

In the `run_synth.tcl` script, the `.synth.done` file is created after a successful run:

```
read_verilog top.v
synth_design -top top
write_checkpoint -force top_synth.dcp
touch .synth.done
```

TIP: *The `touch` command creates an empty file that is time-stamped after the `top_synth.dcp` is created.*

When you run `make synth`:

1. It examines the `synth` target, which depends on the file `.synth.done`.
2. Because `.synth.done` does not yet exist, make examines the rule to build that target.
3. The file `.synth.done` is made by running Vivado synthesis using the `run_synth.tcl` script.
4. The touch command creates the empty `.synth.done` file.

When `make` is subsequently run:

5. If `.synth.done` exists, make compares its timestamp to that of `top.v`. If `.synth.done` is older, then `top.v` has been updated and synthesis needs to be run again.
6. If `.synth.done` exists, and is not out-of-date, then make does nothing for the synth target.
7. If `.synth.done` does not exist, if it was cleaned or removed from the current directory for example, then the synth target is made as discussed above.

All Targets

When a Makefile contains multiple targets, it is common to include a target called `all` that generates all targets in the Makefile. For example:

```
all : synth bitstream

synth : .synth.done

.synth.done : top.v
   cmd /c "vivado -mode batch -source run_synth.tcl"

bitstream : .synth.done .bitstream.done

.bitstream.done : top.v
   cmd /c "vivado -mode batch -source run_bitstream.tcl"
```

In this Makefile, the bitstream target has been added, which has rules that are similar to the synth target. In addition, the first target is `all` with dependencies on both `synth` and `bitstream`.

Running make without specifying a target will run `all` by default, which in turn makes both `synth` and `bitstream`.

Clean Targets

The make utility is usually run in a working directory, where different tools are run and outputs are generated. The required outputs and scripts are copied or moved from the working directory, and checked-in to the revision control repository.

After check-in, the working directory is typically "cleaned" of files using a `clean` target. For example:

```
clean :
   rm -rf *.log *.jou project_* .*
```

This removes all log, jou, project, and target files (such as `.synth.done`). There are many other ways to clean the working directory. The Makefile used in the labs of this tutorial include the `clean` target.

Variables

You can declare variables in Makefiles to improve readability and simplify maintenance. For example:

```
SRCS = ../srcs ../xdc ../scripts
synth : .synth.done

.synth.done : $(SRCS)
        cmd /c "vivado -mode batch -source run_synth.tcl"
```

TIP: *You must use of the '$' character to reference the variable in the Makefile, after it has been declared.*

Windows vs. Linux

The labs in this tutorial are designed to run on Windows, but the Makefiles used are portable. The primary difference is the way the Vivado tool is launched:

- Windows:
  `cmd /c "vivado -mode batch -source run_synth.tcl"`
- Linux:
  `vivado -mode batch -source run_synth.tcl`

Note that the Windows command uses the Windows Command Prompt, `cmd /c`, to launch the tool, and the actual command must be enclosed in quotes. The Linux command can be directly called.

The pathname hierarchy separator is a forward slash, `/`, for both Windows and Linux:

```
vivado -mode batch -source ../scripts/run_synth.tcl
```

Please Read: Important Legal Notices

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

© Copyright 2015 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.