Lab Instructions: Revision Control

Contents

[Introduction 2](#_Toc395023497)

[Lab Files 2](#_Toc395023498)

[GitHub for Windows 3](#_Toc395023499)

[Download and Installation 3](#_Toc395023500)

[Git: Mini-Tutorial 6](#_Toc395023501)

[Make Utility 8](#_Toc395023502)

[Simple Example 8](#_Toc395023503)

[More robust targets 8](#_Toc395023504)

[Multiple Targets 9](#_Toc395023505)

[Clean Targets 10](#_Toc395023506)

[Variables 10](#_Toc395023507)

[Lab 1 : Getting started with a basic RTL project 10](#_Toc395023508)

[Working in shells 10](#_Toc395023509)

[Setting up the working directory 11](#_Toc395023510)

[Lab Procedure 12](#_Toc395023511)

[Conclusion 12](#_Toc395023512)

[Lab 2 Using Revision Control with Managed IP 13](#_Toc395023513)

[Lab file 13](#_Toc395023514)

[Lab Procedure 13](#_Toc395023515)

[Conclusion 15](#_Toc395023516)

[Lab 3: Revision Control of a Block Design 15](#_Toc395023517)

[Lab files 15](#_Toc395023518)

[Lab Procedure 16](#_Toc395023519)

[Summary 17](#_Toc395023520)

[Packaged Custom RTL IP with IPI 17](#_Toc395023521)

[System Generator Projects 17](#_Toc395023522)

[HLS 17](#_Toc395023523)

[Other - TBD 17](#_Toc395023524)

[DCP 17](#_Toc395023525)

[EDK/XPS 17](#_Toc395023526)

[Appendix A: Solutions 17](#_Toc395023527)

# Introduction

This document includes the instructions for the Revision Control lab exercises. There is a total of TBD? labs:

1. RTL project
2. IP project
3. IPI project
4. TBD (The works? Add SysGen, HLS, CIP all in one lab? May not have time to create individual labs…)

We will be using the following tools for these labs:

* Git revision control
* MinGW make utility - included with Vivado
* Vivado 2014.2

Before the labs begin, there are some instructions on how to install Git for Windows. **Please try to install Git before the lab session in case there is limited bandwidth for downloads.** Also if you are new to revision control and particularly the Git revision control tool, please review the Git mini-tutorial to become familiar with the basic actions and commands. Finally if you are not familiar with the **make** utility, please review the introduction to make.

## Lab Files

The labs are created to run on Windows. Attendees can modify scripts and Makefiles to run on Linux if necessary.

Since revision control is essential to the labs, the labs will all use the same working directory and revision control repository.

### Files Provided with the Labs

The following files are provided with the labs, arranged in these directories:

* doc: lab instructions and presentation
* lab\_files: Use these as starting points or templates and complete them as necessary during the labs. Inside lab\_files are directories lab1, lab2, lab3, and lab? which contain the files for each lab.
* lab\_solutions: These contain completed versions of the files modified during the labs. Please do not jump directly to the lab\_solutions without first trying the labs. Notify your instructor if any lab instructions are unclear.

### Where to put the lab files

First create a “root” location where you will store the lab source files and working directory. For example:

* C:\tsc14\revision\_control\_labs
* C:\users\your\_username\revision\_control\_labs

This is where you will create the Git repository and where you will put the directories for the different types of lab files. Copy the files from lab\_files into this root directory. Example: lab1 contains these directories, and some of those directories contain files:

* lab1
  + bd
  + hdl
  + ip
  + scripts
  + xdc
  + work

Copy the lab1 contents (not lab1 itself) into your root location, for example C:\tsc14\revision\_control\_labs. Then when you begin Lab 1, your directory structure looks like this:

* C:\tsc14\revision\_control\_labs
  + bd
  + hdl
  + ip
  + scripts
  + xdc
  + work

Subsequent labs contain files that you add to these directories when you begin the lab. The lab2 directory contains a single file:

lab2\**scripts\ip.tcl**

so to begin lab2, you copy ip.tcl to:

C:\tsc14\revision\_control\_labs\**scripts\ip.tcl**

Do not overwrite the scripts directory with lab2\scripts. Subsequent labs follow similar patterns.

## GitHub for Windows

Git will be used as the revision control tool for these labs. It is free and provided as part of the GitHub installation.

GitHub.com also serves as the remote repository for the Vivado Tcl Store so you are encouraged to become familiar with it, although its use will not be covered in these labs.

### Download and Installation

See the Windows GitHub introduction page here:

[https://windows.GitHub.com](https://windows.github.com)

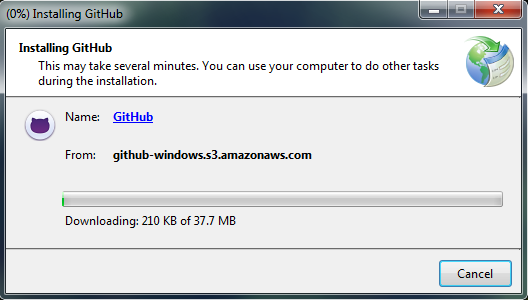
From there, click on

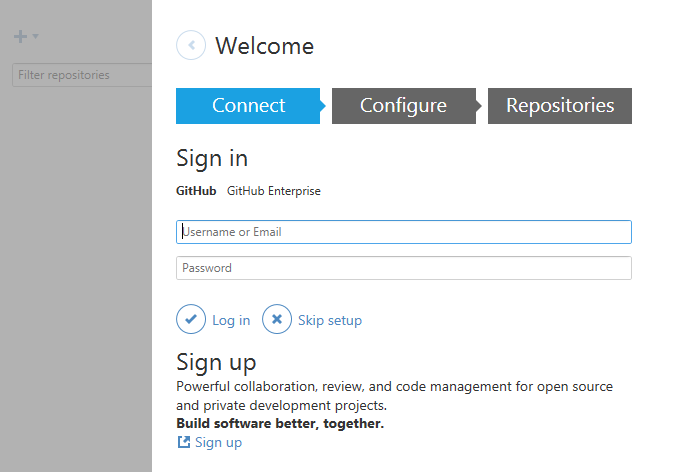


to download the setup program, or go to this URL:

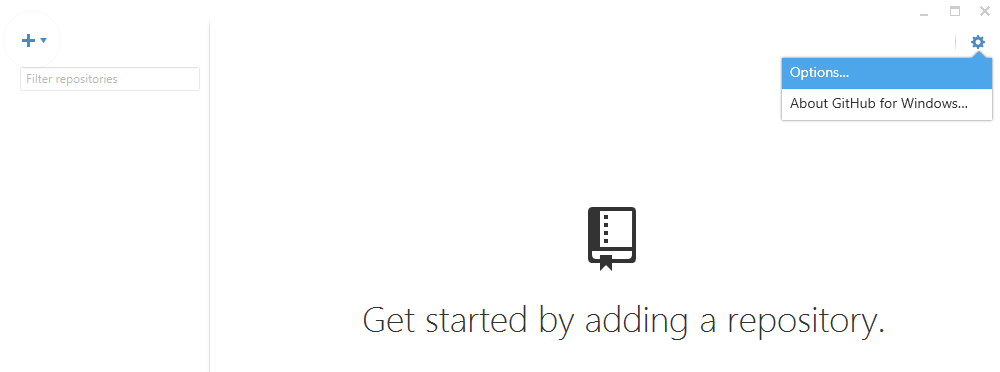
[https://GitHub-windows.s3.amazonaws.com/GitHubSetup.exe](https://github-windows.s3.amazonaws.com/GitHubSetup.exe)

Once GitHubSetup.exe is downloaded, run it to install GitHub.



Once the installation is complete you will see the Welcome page. Here you can **Log in** if you already have a GitHub account, or click the **Sign up** link at the bottom to create a new account. Or you may just continue without logging in by clicking on **Skip setup**. 

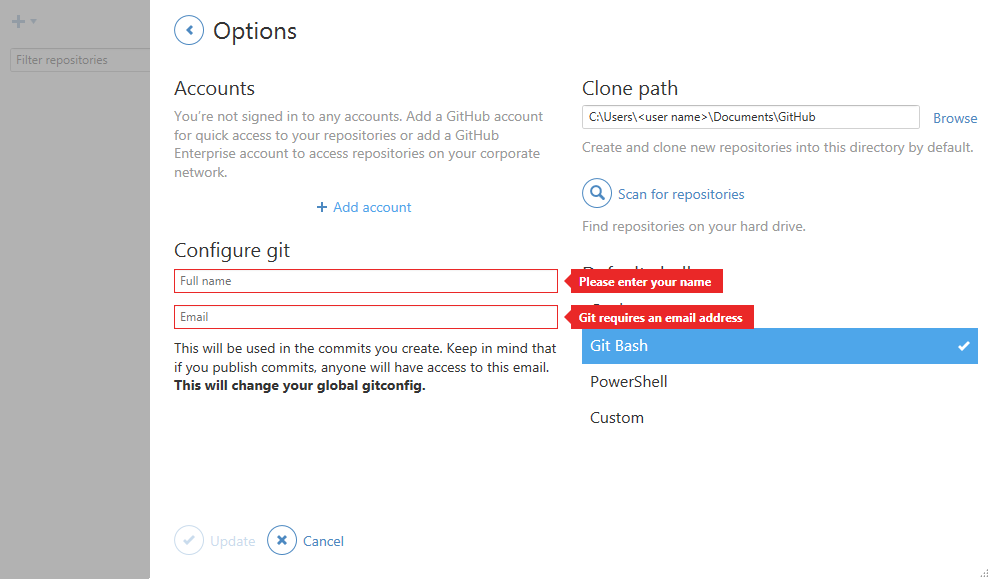
The next page shows a start page where no repositories have been set up, which should be typical for most. We will skip adding repositories for now and instead right-click on the blue Settings icon in the upper right corner and choose **Options…**



Here we will configure the following:

1. In the lower right, choose Git Bash as the default shell instead of PowerShell.
2. Enter your name and email which Git uses to track changes.

Once finished, click on the Update button at the bottom to save changes.



Now you may exit the GitHub GUI. We will be using the Git Shell for revision control in all labs.

### Git: Mini-Tutorial

We will now run through a mini-tutorial to introduce you to Git and its most commonly used commands.



To launch Git, double-click the Git Shell icon to launch the Git Shell. It should open a command shell with a bash prompt ($). Try the following command sequence which will create a file, check it in, modify it, check it in again, then remove it, and finally restore it. The commands are in bold, followed by a brief explanation. If you are new to Git, please try this exercise before the lab session.

1. **cd** Go to your home directory.
2. **mkdir git\_temp** We will create a repository here and delete it later.
3. **cd git\_temp**
4. **git status** This reports the current repository status - there is no repository at this point.
5. **git init** This creates a repository. Notice the prompt changes and displays a trailing “(master)” which indicates the master branch is active.
6. **git status** Should report nothing to commit.
7. **echo example1 > file.txt** Create a file with contents of “example1”
8. **git status** Should report file file.txt as “untracked.”
9. **git add file.txt** Stage file.txt for committing to the repository. Note that all files in the current directory and sub-directories can be checked in using **git add .** or **git add \***
10. **git status** Should report file file.txt as a new file.
11. **git commit** This should open a file where you can enter comments about the check in. You can also use the **-m** option with a string argument for the comment.
12. **git status** Should now report nothing to commit. The repository is up-to-date.
13. **echo example2 > file.txt**  File modification: overwrite file.txt with different contents.
14. **git status** The file file.txt is modified.
15. **git add file.txt** Stage file.txt for committing to the repository. Notice that this command was recommended by the prior **git status** command.
16. **git commit -m "changed the file"** Commit file.txt.
17. **git status** Should now report nothing to commit.
18. **git rm file.txt** Remove the file from disk and stage it to be removed from the repository. The file file.txt is gone.
19. **git status**  Shows that file.txt is ready to be deleted on the next commit. Assume we realize this was a mistake and we really didn’t want to delete the file. Git suggests a command to unstage the file for deletion.
20. **git reset HEAD file.txt** Unstage the file for deletion.
21. **git status** Git once again recommends a command to discard changes in the working directory.
22. **git checkout -- file.txt** Undo the deletion.
23. **git status** All back to normal.
24. **git rm file.txt** Remove the file from disk and stage it to be removed from the repository.
25. **git commit -m “deleted the file”** Commit the file deletion. The file file.txt is now gone! Realizing it was a mistake, we will now attempt to restore it.
26. **git log --summary** See summary of check-ins:

$ git log --summary

commit 853dbbc3fd3d06812b8868cf0afd0425645e24f3

Author: Your Name <yourname@xilinx.com>

Date: Mon Aug 4 15:51:57 2014 -0700

deleted the file

delete mode 100644 file.txt

commit fba22d51e92f7d19641131853123163e94619918

Author: Your Name <yourname@xilinx.com>

Date: Mon Aug 4 15:47:49 2014 -0700

changed the file

commit 77f73e2cd21de802ff2629df73ebfca91813b9c0

Author: Your Name <yourname@xilinx.com>

Date: Mon Aug 4 15:39:17 2014 -0700

created the file

create mode 100644 file.txt

1. **git checkout fba22d51e92f7d19641131853123163e94619918 -- file.txt** Restore the file. The commit string (long string of numbers) is used to choose the desired version of the file. In this case we restored the previous version. **In your case, use the string corresponding to the one in red above.**
2. **cat file.txt** This should display the file contents from the previous checkin.
3. **git commit -m "restored the file"** Restore file.txt back into the repository.
4. **git status** All should be back to normal.

When finished you can return to your home directory and remove the practice directory.

1. **cd; rm -rf git\_temp**

This should give you an idea of how the common commands work. In particular we will be using these commands frequently in the labs:

* git status
* git add
* git commit

## Make Utility

This section is an introduction to the UNIX **make** utility. Please review this section if you are new to make. The make utility is widely used to build projects using “Makefiles.” Makefiles contain all the “rules” (commands) needed to build “targets,” usually outputs such as netlists or bitstreams. The rules also describe certain dependencies based on file timestamps to determine which commands must be run and which can be skipped because their output is up-to-date.

### Simple Example

Here is a simple example of a Makefile. An RTL design consists of a single Verilog file top.v and is synthesized to create a checkpoint top.dcp.

top\_synth.dcp : top.v

cmd /c "vivado -mode batch -source run\_synth.tcl"

Target: **top\_synth.dcp** is a target (an output), that depends on **top.v**. If top.v has a modification time that is more recent than top\_synth.dcp, then top\_synth.dcp needs to be regenerated, otherwise it is up to date.

Rule: The rule to make top\_synth.dcp is fairly simple. The target top\_synth.dcp depends on top.v, and if it is out of date, run Vivado in batch mode with a script **run\_synth.tcl** that presumably runs synthesis and writes top\_synth.dcp. Note that wrapping vivado in a **cmd /c** call is only required on Windows.

A note about the syntax: the whitespace at the beginning of each command must be a tab character, not spaces.

To build the top\_synth.dcp target, run make like this from a shell prompt:

make -f Makefile top\_synth.dcp

Assuming top\_synth.dcp is out-of-date, older than top.v, vivado is launched with the batch script to regenerate top\_synth.dcp. Additional notes:

* If the Makefile resides in the current directory, and it has the name **Makefile** then the -f option is not necessary.
* If the target is not specified, make will just build the first target in the Makefile which is top\_synth.dcp.

For this simple example, the same results can be generated by running only **make** without any options.

### More robust targets

Assume that we may not know the output of a target, for example we don’t know the name of the file, or even what file is generated by run\_synth.tcl. We can create a symbolic target instead of relying on the output file.

# This is a comment

synth : .synth.done

.synth.done : top.v

cmd /c "vivado -mode batch -source run\_synth.tcl"

And in the Vivado run script, the .synth.done file is **touch**ed after a successful run:

read\_verilog top.v

synth\_design -top top

write\_checkpoint -force top\_synth.dcp

touch .synth.done

The touch command is a Tcl proc that emulates the UNIX touch command, which creates an empty file with a timestamp of the end of the Vivado run script. This proc will be provided with the lab files.

When make is run:

1. It builds the first target **synth** which depends on the file **.synth.done**.
2. The file .synth.done does not yet exist, so make looks for the rules to build .synth.done.
3. The file .synth.done is built running vivado in batch mode, followed by the **touch** proc.
4. The touch proc creates an empty file .synth.done.

When make is subsequently run:

1. If .synth.done exists, make compares its timestamp to that of top.v. If .synth.done is older, then it is out-of-date and the synth target is regenerated.
2. If .synth.done exists, and is not out-of-date, then make does nothing for synth.
3. If .synth.done does not exist, for example if it was cleaned from the current directory, then the synth target is made.

### Multiple 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, if we added another target for bitstream generation, our Makefile becomes:

all : synth bitstream

synth : .synth.done

.synth.done : top.v

cmd /c "vivado -mode batch -source run\_synth.tcl"

bitstream : .bitstream.done

.bitstream.done : top.v

cmd /c "vivado -mode batch -source run\_bitstream.tcl"

Here we added the bitstream target which has similar rules to synth. The first target is **all** with the dependencies of **synth** and **bitstream**. Running make without any targets will run **all** by default which in turn runs **synth** and **bitstream**.

### Clean Targets

Since make is normally run in a “working directory” where tools are run and outputs are generated, the relevant outputs and scripts are copied or moved from the working directory to a known location and checked in. Afterwards the working directory is typically “cleaned” of tool outputs using a **clean** make target. Example:

clean :

rm -rf \*.log \*.jou project\_\* .\*

This simply removes all log, jou, project, and target files (such as .synth.done). There are many other ways to clean the working directory. We will be using clean targets in the labs.

### Variables

Variables can be used in Makefiles to improve readability and simplify maintenance. Here is an example. Note how the variable is dereferenced using $:

**SRCS** = ../srcs ../xdc ../scripts

synth : .synth.done

.synth.done : $(**SRCS**)

cmd /c "vivado -mode batch -source run\_synth.tcl"

These simple examples should cover all the basics to get you started using make in these labs. See the Web for more info about make.

# Lab 1 : RTL Project Scripts

In this lab we will start with a very basic set of files and introduce you the basic processes involved in creating a project script for Vivado and maintaining files under revision control. This includes creating:

1. Vivado run scripts to create and run projects
2. A Makefile that calls the run scripts to build projects
3. A Git repository that keeps files under revision control

Please review the introductions to Git and make if they are not familiar to you.

## Working in shells

We will use two different command shells for the labs:

* Windows command shell - to run Vivado and make
* Git bash shell - for everything else

Upon opening a Windows command shell, use the **env.bat** script to set up Vivado to run in the shell:

call <path to scripts>\env.bat

This also adds the MinGW UNIX-like commands to your path so you can run the **make** utility to call Makefiles in the Windows shell. You can also run other common UNIX commands such ls, cp, rm, mv, and the UNIX find command within the Windows command shell.

Test the make command. In the Windows command shell, run **make test**. It should fail with a message similar to this:



If you get a different message or experience some other difficulty, ask your instructor for assistance.

To launch the Git shell, double-click the icon as described earlier during the Git introduction:



The Git shell functions just like a bash shell and is pre-configured to run git commands. On Windows the pathname to the **C:\** drive is **/c/**, so for example the folder **C:\tsc14** is accessed using **/c/tsc14**.

## Lab Setup

As described in the introduction, we will start with a few files in a simple directory structure and add to it as the design grows with each successive lab. In the lab1 directory you’ll find these files to begin:

* bd (empty directory)
* ip (empty directory)
* work (empty directory)
* hdl
  + threeFlop
    - threeFlop.v
* xdc
  + top.xdc
* scripts
  + setup.tcl
  + Makefile
  + env.bat
  + utils.tcl

Copy these files to your root directory such as C:\tsc14\revision\_control\_labs. We will call this the root root directory. Inside root we will place the Git repository and a work folder to hold Vivado results. The lab directory hierarchy should now resemble:

* (root directory)
  + bd
  + ip
  + hdl
  + xdc
  + scripts
  + work

When working on files in this lab, it is recommended to keep script files in the **script** directory and test them by running vivado or make in the **work** directory. To do this copy the Makefile into the work directory for editing and test it by calling **make** on the command line. When finished the Makefile can be checked back in to the scripts directory.

## Lab Procedure

1. Complete the Tcl script named **setup.tcl** to recreate a Vivado project for the **threeFlop** design. Recall the ways to create a project script such as write\_project\_tcl or reviewing .jou commands for the equivalent GUI actions. Make sure:
   1. The touch proc is called at the end of the script to create a Makefile target - This tells **make** that the setup script ran successfully. If setup fails then no target file is generated.
   2. The sources are referenced remotely, not added to the project.
   3. When testing the script, run Vivado in the **work** directory.
2. Create a new Tcl script called **compile.tcl.**
   1. It should open the threeFlop project and generate a bitstream.
   2. Use the touch proc to create a target file **.compile.done** similar to the target file used for the setup target.
   3. Test the script to ensure it creates the Vivado project and Makefile target as expected.
3. Complete the Makefile to build a bitstream using setup and compile targets.
   1. The setup target has been completed as an example.
   2. Add a target **compile** that calls compile.tcl to generate the bitstream.
   3. Add a target **clean** that removes all the files in the current directory which is expected to be the **work** directory.
      1. Note that UNIX commands are used to remove files.
      2. Pay special attention to the Makefile target files which begin with a dot character.
   4. Test the script to ensure it creates the bitstream and Makefile target as expected.
4. Check the files into the Git repository.
   1. Create the Git repository in the root folder using the **git init** command.
   2. Check the status using the **git status** command. It should reflect a newly created repository with **Untracked files**.
   3. Check in the **hdl**, **scripts**, **xdc, bd, and ip** directories, but not work. The commands to use are:
      1. **git add <filename(s) or directory name(s)>**  to stage files for a commit.
      2. **git commit -m “Comments on the checkin”**

It is good to run git status after each step to ensure the project state is as expected.

**Note:** Git does not support adding empty directories to a repository. The empty bd and ip directories each contain a hidden, untracked file with a special name .git.ignore.

1. Once all has been checked in successfully, clean the work directory.

## Conclusion

This first lab has taken you through the complete process of:

1. Creating Tcl scripts to create and run Vivado projects.
2. Preparing and calling a Makefile to run those scripts to build projects.
3. Creating and using a Git repository to keep recommended files under revision control.

We will build on the concepts learned in this lab to learn how to handle different types of design data under revision control.

# Lab 2 Using Revision Control with Managed IP

This lab covers the process of generating a simple managed IP AXI IIC and placing it under revision control. The lab objectives include:

* Creating a Managed IP project for AXI IIC.
* Create a script to fully generate the IP with OOC flow.
* Update the Git repository with the IP products and updated scripts into revision control.
* Iterate and change an IP customization option and update the Git repository as needed.

## Lab file

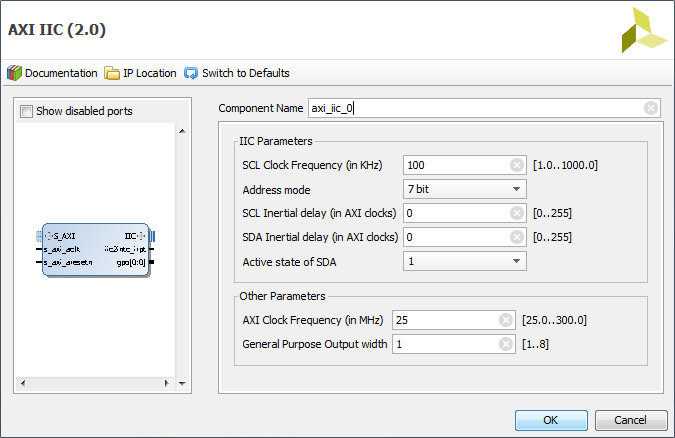
Only one lab file is needed to get started:

* scripts/ip.tcl: This is a template file where you will add the Tcl required to build the IP from scratch.

We will be editing the ip.tcl script to generate the IP from scratch, updating the Makefile to include the IP as a target, and checking the IP output products into the ip directory.

## Lab Procedure

1. Copy the lab file ip.tcl to your scripts directory.
2. Create the AXI IIC IP in the work directory.
   1. Target part is xc7z020clg484-1.
   2. The target language is Verilog.
   3. Use the default settings for the IP. The component name can be left as is: **axi\_iic\_0**.



1. Complete the **ip.tcl** Tcl script that recreates the IP from scratch, all the way to output product generation.
   1. Avoid using absolute paths so that the IP can be recreated in a different location in the future. Note the suggestion of storing locally in the **ip** directory.
   2. Make sure the utils.tcl script is sourced and use the touch proc to create a file **.ip.done** that can be used for a Makefile target similar to **compile** and **setup**.
   3. Since ip.tcl may be called in a sequence with other scripts, remember to wait on a run before creating .ip.done.
2. Add the IP target to the Makefile:
   1. Add the new target called **ip** that regenerates the axi\_iic\_0 output products from scratch.
   2. Add the rule to create the ip target. Assume that the IP will be generated unconditionally when the ip target is made.
   3. The other targets remain unchanged for this lab.
3. Test the Makefile by making the **clean** target followed by **ip**.
4. Check that the IP output products are in the new directory **ip** which should be located in **work**. The ip directory should contain the axi\_iic\_0 directory and everything below it.
5. If satisfied, copy the necessary files to a new directory **ip** which is in the root directory, at the same level as work. Recall the recommendation of files to check in for Managed IP: the IP directory and everything underneath.
6. Check in the IP, the Makefile and the new script:
   1. Use **git status** to check the project status.
   2. Use **git add** to add the ip directory contents and changed files.
   3. Use **git status** to check the project status again.
   4. Use **git commi**t to check in the files.
7. Once the IP is under revision control, make a change. Change the AXI Clock Frequency from 25 MHz to **100 MHz**. The equivalent IP property is CONFIG.AXI\_ACLK\_FREQ\_MHZ.



1. Then regenerate the IP, update scripts, and check in files as needed. You may need to clean the work directory first.
2. Once all has been checked in successfully, clean the work directory.

## Conclusion

In this lab you have covered how to manage IP under revision control. You should now be comfortable performing the following:

* Creating and modifying Tcl scripts to fully generate the IP from scratch.
* Using a Makefile to build IP.
* Checking in IP files under revision control with Git.
* Modifying IP and updating the Git repository.

# Lab 3: Revision Control of a Block Design

This lab focuses on block designs from IP Integrator. In terms of revision control, block designs are somewhat similar to IP as they have associated output products. Block design management may also include recreating the block design itself from scratch. In this lab you will cover:

1. Use of write\_bd\_tcl to generate a Tcl script to recreate a block design.
2. Creating a script to generate the block design output products.
3. Using a Makefile to build a block design output products from scratch.
4. Checking in the block design under revision control.

## Lab files

The lab files contain the following new files to get started:

* scripts/bd\_gen.tcl: This is a template file where you will add the Tcl required to build the block design from scratch.
* zynq\_bd\_project: contains a project with the block design used for the lab

This lab is a bit more complex than previous labs. This lab sets up a scenario where a customer has a block design inside a Vivado project that needs reused, and placed under revision control. The original block design is a local source in a project in zynq\_bd\_project.xpr. The sequence of updating the block design is:

* The block design is modified in IP Integrator in zynq\_bd\_project.xpr.
* The block design is saved and packaged into a separate source directory containing block designs. Recall the recommendation for block designs is to check in the directory containing the .bd file: that directory and all files and directories underneath.
* Other designs instantiate the block design which implies that those designs depend on the corresponding .bd file.
* When the block design is updated and saved, the .bd file changes and the entire block design package must be regenerated from scratch and checked back into the bd directory.

## Lab Procedure

1. Copy zynq\_bd\_project into the work directory and open the project from work. The block design should look similar to the following diagram.



1. Use write\_bd\_tcl to write a script that creates the block design. Name the file **bd.tcl** and place it in the source directory called **bd** inside the root directory.
2. Create a Tcl script **bd\_gen.tcl** in the **scripts** directory that generates the block design output, similar to an IP. Some hints:
   1. Check what Tcl commands are issued when launching **Generate Block Design**.
   2. Review the IP generation script ip.tcl from Lab 2 which uses a similar process to generate the IP.
   3. Note that a project is required to hold the generated block design, although the project itself may not be useful for revision control.
   4. Include the creation of a Makefile target when finished.
3. Next update the Makefile.
   1. Include a target **bd\_gen** that results in the entire block design being generated from scratch.
   2. In this lab we will assume that a final block design has been handed to us, so we do not need to monitor the original zynq\_bd\_project as a dependency. But we may choose to modify the block design in the future. It is now a self-contained package.
4. Recall the recommendation of files to check in for block designs, the block design directory and everything underneath it. Check in the recommended files to the bd directory. Check in the necessary script files and the Makefile. Hint: you do not need to check in zynq\_bd\_project.
5. Once all has been checked in successfully, clean the work directory.

## Summary

This lab demonstrates how IP Integrator Block Designs can be managed using revision control. Some key points to remember:

* A block design .bd file can be generated using the write\_bd\_tcl command.
* The generate\_target command generates block design output products from a .bd file, similar to Managed IP.

# Packaged Custom RTL IP with IPI

# System Generator Projects

# HLS

# Other - TBD

## DCP

## EDK/XPS

# Appendix A: Solutions