Prework: Revision Control Labs

Contents

[Introduction 1](#_Toc395362604)

[GitHub for Windows 1](#_Toc395362605)

[Download and Installation 2](#_Toc395362606)

[Git: Mini-Tutorial 5](#_Toc395362607)

[Introduction to the Make Utility 7](#_Toc395362608)

[Simple Example 7](#_Toc395362609)

[More robust targets 7](#_Toc395362610)

[Multiple Targets 8](#_Toc395362611)

[Clean Targets 9](#_Toc395362612)

[Variables 9](#_Toc395362613)

[Windows vs. Linux 9](#_Toc395362614)

[Working in shells 9](#_Toc395362615)

# Introduction

This document provides an introduction to the utilities used in the Revision Control lab exercises. For the labs we will be using the following software:

* Vivado **System Edition** with SDK
* Git revision control
* MinGW make utility - included with Vivado

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.
* If you are not familiar with the **make** utility, please review the introduction to make.

# 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)

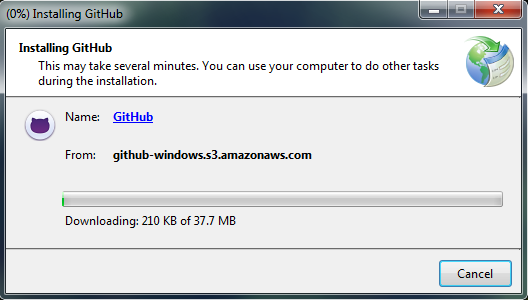
From there, click on

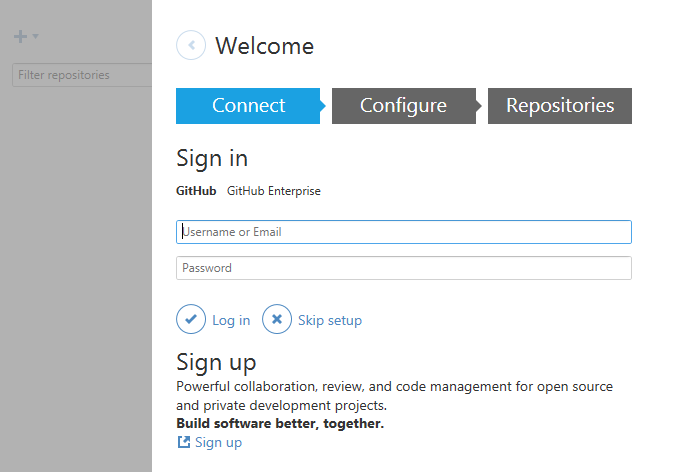


to download the setup program, or go to this URL:

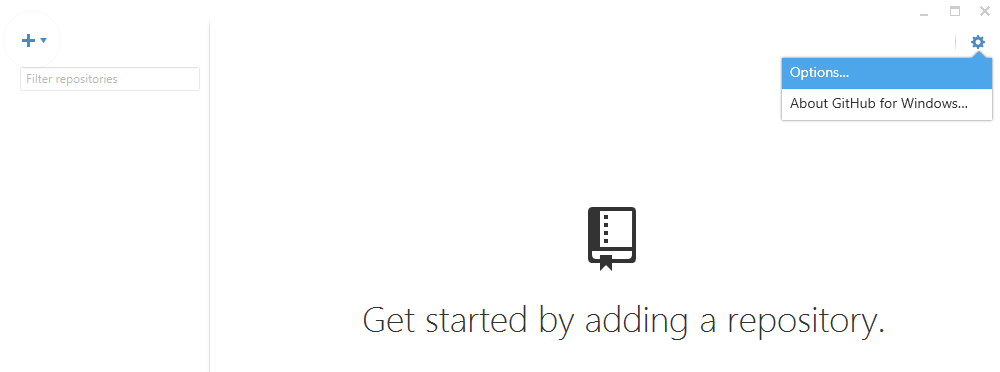
[https://GitHub-windows.s3.amazonaws.com/GitHubSetup.exe](https://github-windows.s3.amazonaws.com/GitHubSetup.exe)

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**. 

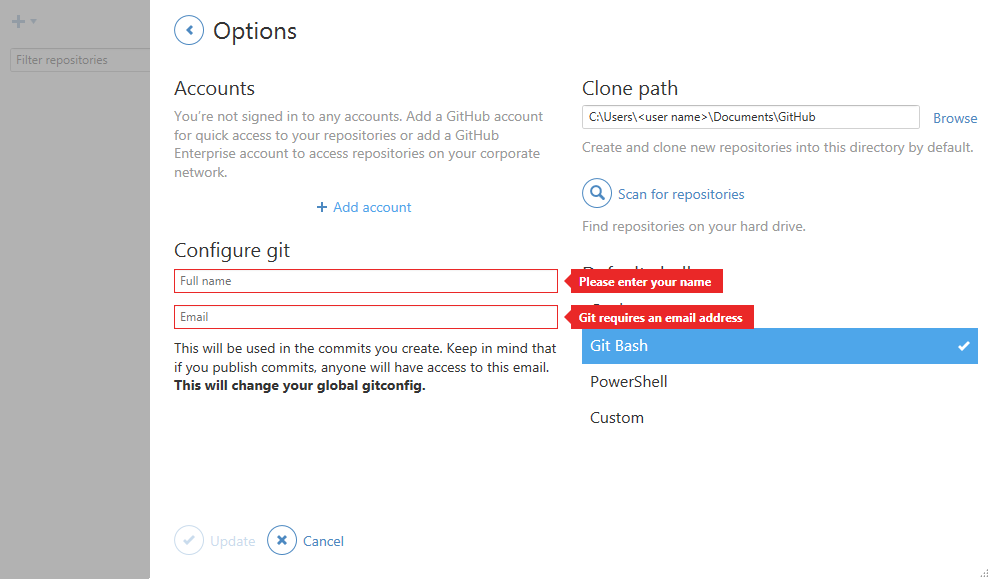
The next page shows a start page where no repositories have been set up, which should be typical for most. We will skip adding repositories for now and instead right-click on the blue Settings icon in the upper right corner and choose **Options…**



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@company.com>

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

deleted the file

delete mode 100644 file.txt

commit fba22d51e92f7d19641131853123163e94619918

Author: Your Name <yourname@company.com>

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

changed the file

commit 77f73e2cd21de802ff2629df73ebfca91813b9c0

Author: Your Name <yourname@company.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 repository state with 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

# Introduction to the 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.

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 : .synth.done .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"

## Windows vs. Linux

The labs designed to run on Windows but the Makefiles used are mostly portable. The primary difference is the way vivado is called:

* Windows: cmd /c "vivado -mode batch -source run\_synth.tcl"
* Linux: vivado -mode batch -source run\_synth.tcl

The pathname hierarchy separator is a forward slash, same for both Windows and Linux. Examples:

* vivado -mode batch -source ./run\_synth.tcl
* vivado -mode batch -source ../scripts/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.

# 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

A recommendation is to create a .bat script that can be sourced upon opening a new Windows cmd shell. The file env.bat will be provided with the lab files. Its contents are:

call c:\Xilinx\Vivado\2014.4\settings64.bat

SET PATH=C:\Xilinx\Vivado\_HLS\2014.4\msys\bin;%PATH%

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:

C:\training\revCtrl\work>make test

make: \*\*\* No rule to make target `test'. Stop.

C:\training\revCtrl\work>

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:\Xilinx** is accessed using **/c/Xilinx**.