ECE 212 – Embedded Systems
Fall 2023

Introduction to the Git Version Control System

Revised August 31, 2023

1. Introduction

When creating hardware and software designs Electrical and Computer Engineers work with computer-based tools that represent different design objects as files stored on a computer system. Typically the files for each lab are stored in a directory with subdirectories for each type of file, as shown in Figure 1. Subfolders should be named as follows:

`hdl`	SystemVerilog and/or VHDL source code
`constraints`	FPGA constraint files
`reports`	Report files such as synthesis reports
`projects`	Vivado projects
`src`	C source code

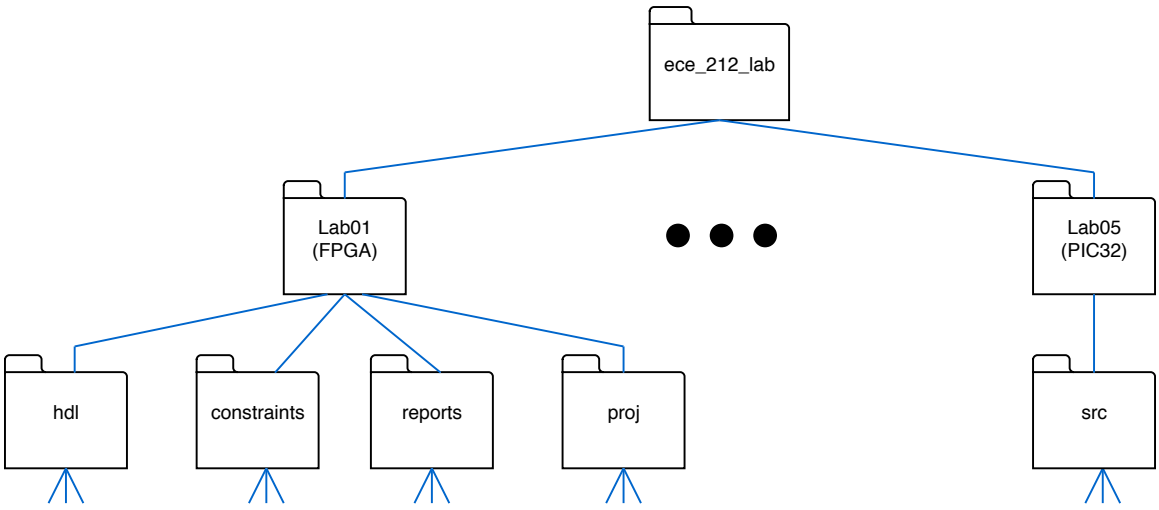

Figure 1 – Files in a Typical ECE 212 Project

Over the lifetime of a project, the number of these files grows and the files are modified from day to day. Even after a project is complete, it may be necessary to go back and make changes to correct bugs. As this process takes place, it becomes challenging to keep track of all the changes and to keep a good backup of the current design in case changes made in error must be undone. When multiple users are working with the files, this becomes even more difficult, since two people may inadvertently try to change the same file at the same time. Sometimes it is even necessary to backtrack to older version of the code.

Because this problem is particularly acute in software, software developers have developed tools for *version control* that address many of these issues. Some common version control tools are CVS, Subversion (often abbreviated as SVN), and Git.

Version control tools place a project's files into a *repository* and keep track of the changes made to these files by different users over time. Each time files are changed, these tools record the current version of the project. Version control tools also allow users to *roll back* to a previous version of the project if the changes that were made were unsuccessful. Finally, many version control tools provide users with an easy mechanism to back up project files so that a local computer crash does not result in lost files.

In ECE 212 we will use **Git** as our version control system. Git keeps track of changes to the project in a series of update steps called *commits*; you can think of each commit as a snapshot of all of the files in the current project. As files are added, removed, and changed, new commits are done creating a record of how the project files evolve over time. Most importantly, Git allows users to "roll back" changes to a project to restore a previous version if it turns out that recent changes were catastrophically wrong (for example, if your cat walked on the keyboard while you were getting coffee).

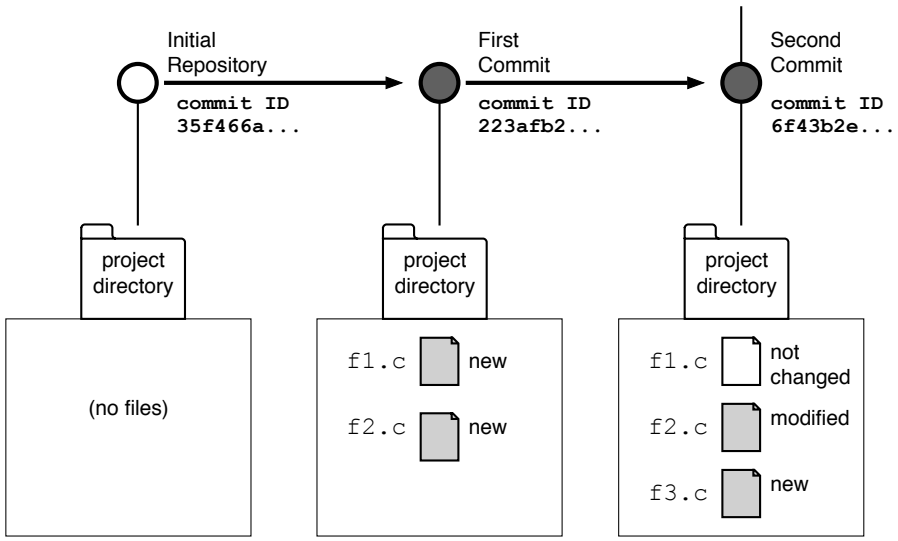

Figure 2 – Commits of a Project in the Git Version Control System

Figure 2 shows how a typical small project might evolve as successive versions are committed to the repository. When the repository is initially created, a project may not have any files. However, over time users will add files to the project, and when ready a commit is performed to create a snapshot. Each time a commit is done Git associates a *commit ID* with the commit – this is a 40-digit hexadecimal number that represents the current state of the repository. While this is not particularly user-friendly, Git allows you to refer to commit IDs using just the first few characters, so this is not as bad as it initially seems.

When used in its simplest mode you can think about each successive version as an element of a list. To make it easy to keep track of the most recent versions, Git allows users to refer to the most recent commit as **`HEAD`**.

Git can be used on a single computer, but to allow collaboration between multiple users a master repository on a server is often used. For example, GitHub provides a server on which you can create and store a master repository. It is used by developers of many

open-source projects. Lafayette uses a similar server that is local to the College called GitLab which is accessed in a similar fashion. Because Lafayette's GitLab server is firewalled and difficult to access remotely we will use GitHub.

Both GitHub and GitLab allow you to create a master repository on the remote server before *cloning* (i.e., copying) the project onto a local computer. As you make local changes to the project by adding and changing files, you can create new local versions of the project called *commits*. These changes can then be copied back to the master repository using a process called *pushing*. When multiple users are working with a project, they download recent changes from the master repository using a process called *pulling*.

In this tutorial we will walk you through the steps of creating a master repository on the GitHub server, cloning that repository to your local computer, making changes, committing the changes to create a new version, and then pushing these changes back the master repository.

2. Working with the Git Version Control System

2.1 Creating a Repository

To create a repository on the GitHub[1] server, do the following:

1. Use a web browser to visit `https://github.com`.
2. Create a new account by clicking on the "Sign Up for GitHub" button. Each student should create their own account.
3. One member of each group should create a new repository that will grant access to all lab group members and the instructor. To do this, click the "New" button.
4. Fill in the "Repository Name" field with the name of your project. Use a project name in the form `ece212_name1_name2…` where *`name1`*, *`name2`*, (and possibly *`name3`*) are the last names of the people in your lab group.
5. Fill in a brief description of your project in the "Project Description" file (this is optional, but a good idea).
6. Click the "Private" radio button – this repository should only be accessible to lab group members and the instructor.
7. Click the "Create repository" button at the bottom of the page.
8. Note that since the project is currently empty, this page will show setup instructions.

2.2 Adding a README file

It is good practice when using Git to include a README file in the root directory of the project that summarizes the purpose of the project and provides simple directions for how to use it. While this can be done on a local computer, you can also add this through the GitHub server. README files in Git are typically written in a simple markup language called "markdown" with an extension ".md" that allows simple formatting such as headings; thus the name this file is `README.md`.

[1] GitLab uses a similar interface with some slight differences.

To add this this file to your repository, click on the README hyperlink on the project page, enter your description into the text box that is displayed, and press the "Commit Changes" button at the bottom of this page.

2.3 Giving Other Users Access to the Repository

If you set up your repository as a private project then you will need to add other users (e.g., lab partners and the instructor) who will be working with the repository. These users will also need accounts on the GitHub (see above). Once you are sure they have done this, do the following:

1. Click on the "Settings" link at the top of the project page.
2. Click the "Collaborators" link on the Settings Menu and click the "Add People" button to add other lab group members and the instructor.
3. Follow the instructions to add lab partners and the instructor as collaborators.

3. Creating a Clone Repository on a Local Computer

In order to work on a project, you will need to create a copy of the master repository on GitHub on a lab computer or your personal laptop. This process is called *cloning* the repository. The idea is that you will work locally on this project by adding files (and later, modifying existing files). Later we will discuss how to "push" these changes back to the master repository on the GitHub server.

1. Open a "Git CMD" command prompt window from the "start" menu by navigating to start->All apps->Git->Git CMD (you may also want to right-click on this menu item and select "Pin to Start" for faster access in the future). If using Linux, open a "terminal" window.
2. The command prompt window accepts commands for working with files and directories as well as invocations of the Git command. Some common commands are listed below in Table 1.
3. Navigate to a directory that will house your repository using the "`cd`" command. We recommend avoiding directories on network drives (on a Windows machine, use a directory on the "C" drive).
4. If this is your first time using Git on this computer, set up your username and email with the following commands:

   ```
   git config --global user.name username
   ```

   ```
   git config --global user.email email
   ```

5. Use a web browser to navigate to the GitHub server and open the page for your project. Copy the URL for the repository from the field at the top of this page. It should look something like:
   `https://github.com/username/projname.git`
6. Clone the repository by typing the following command in the terminal window (paste the URL from the previous step in the place of *`repository_URL)`*:

   ```
   git clone repository_URL
   ```

This will create a new directory called *projname* in the current working directory. Change to this directory and list the files (if this is a brand new project, there may not be any files present initially).

Command (Linux equiv.)	Description
`dir` `(ls)`	List all files in a current directory or matching a particular description (`e.g., ls *.sv`)
`cd` *dir_name*	Change the current working to *dir_name*. Convenient shortcut: ".." refers to the enclosing directory of the current working directory (e.g., `cd ..`)
`cd` `(pwd)`	Print the current working directory
`mkdir` *dir_name*	Create directory *dir_name* within the current working directory
`del` *file_name* `(rm)`	Delete (remove) *file_name*. Warning: this is permanent!
`copy` *src_file dst_file* `(cp)`	Create a copy of *src_file* named *dst_file*

Table 1 – Some Basic Command Prompt Commands (and Linux Equivalents)

4. Making Changes in a Repository

4.1 Adding Files

If you followed the preceding steps, you just created a *mostly* empty[2] directory file called *projname*. You can now move into this directory and start adding and changing files using an editor, development environment, or other tool. For example, in Figure 1 the user has created two new files called `f1.sv` and `f2.sv`.

Once you have completed these changes, it takes two steps to include these new files in a repository. First, you must *stage* the new files for inclusion in the project using the `git add` command, which has the form:

```
git add filename(s)
```

Once you are ready to finalize these new files as part of your project, the next step is to *commit* these changes:

[2] I say *mostly* empty because you may have already added a `README.md` file in Section 3.1; in addition, Git stores configuration information in a hidden subdirectory called `.git`.

```
git commit -m "message describing this commit"
```

This will create a new version of the project. Each time a commit occurs, Git will print out a status message that includes a version ID string looking something like:

```
[master 9bcd877]
```

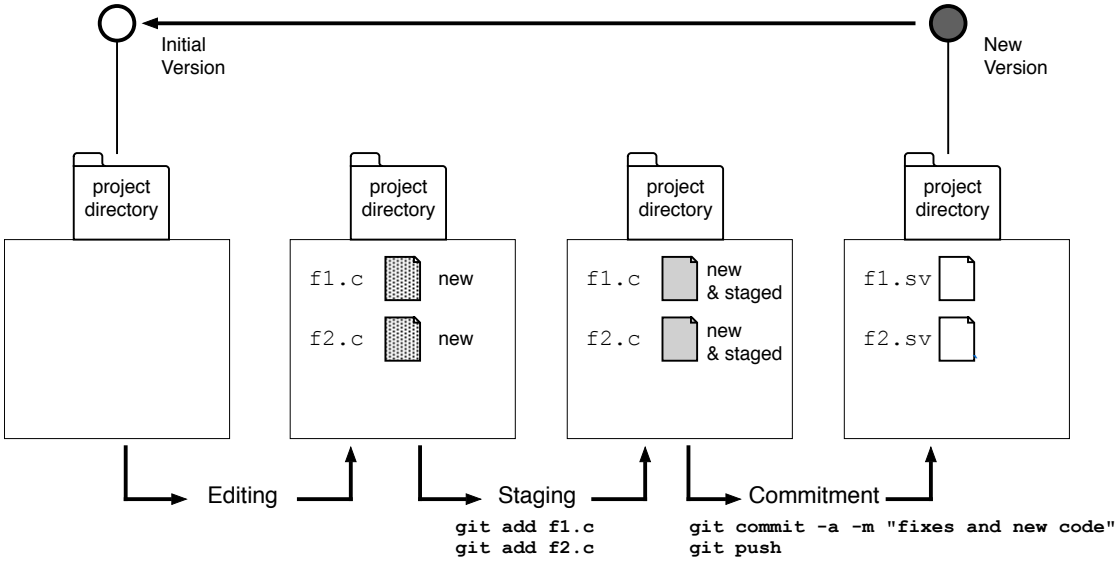

Figure 3 – Adding New Files to a Repository

The 7-digit hexadecimal number in this ID is actually the first 7 digits of a 40-digit *hash code* that uniquely identifies this version. These IDs can be useful if you need to revert back to a previous version.

Figure 3 shows how the two new files `f1.sv` and `f2.sv` are first staged and then committed to create a new version.

4.2 Changing and Removing Files

When additional changes are made to a project, they are staged and committed to the repository in a similar way. New files must be explicitly staged using the `git add` command. Modified files may also be staged using the `git add` command, but if the following commit is done using the `-a` option Git automatically stages modified files for you. Similarly, if you need to remove a file you may do so explicitly using the `git rm` command or else just delete the file in the usual way (e.g. the `rm` command in the terminal window or dragging the file to the trash in the GUI) followed by a commit using the `-a` option.

For example, Figure 4 shows "Version A" of a project consisting of three files `f1.c`, `f2.c`, and `f3.c`. While editing the user creates a new file `f4.c` and deletes file `f2.c`. The user then invokes a commit with the `-a` option which automatically stages the removal of `f2.c` and the modification of `f3.c`. This creates "Version B" as shown.

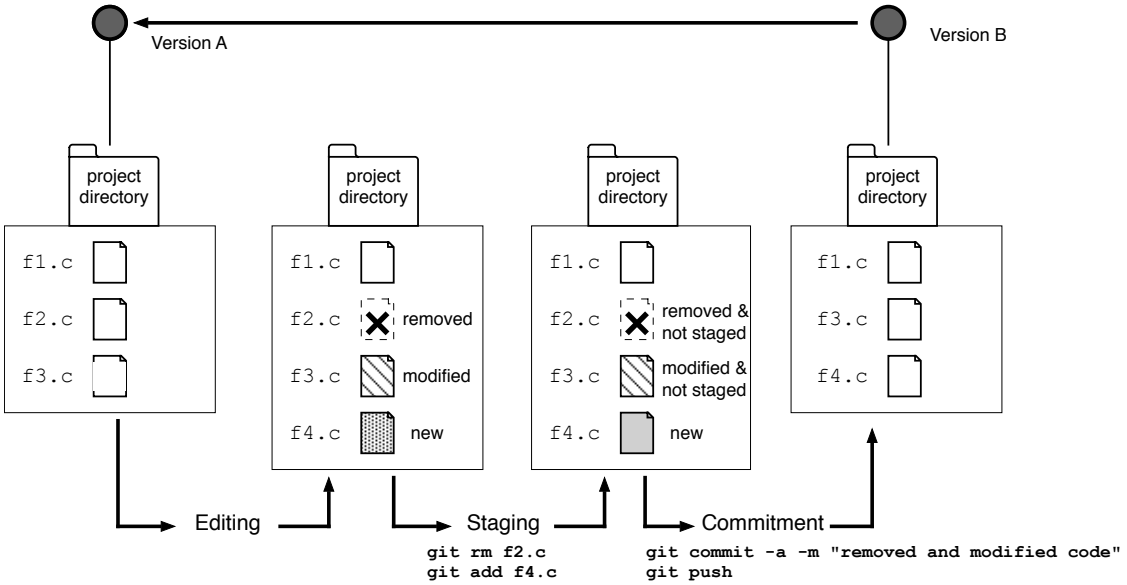

Figure 3 – Modifying and Removing Files

5. Undoing Changes

It is an unfortunate fact that often when changes are made the resulting project may be damaged and the changes must be removed. Git has a different procedure for doing this depending on whether the changes have been committed or not.

5.1 Reset – Removing Uncommitted Changes

Imagine a scenario where you have made several changes without knowing exactly what you are doing (this is especially likely when you are working while tired or stressed). For example, you may be trying to modify a program and somehow you have goofed it up to the point where it won't even compile and you'd like to start over with a clean copy. So long as you have not yet done a commit, you can restore the current version of the project using the command:

```
git reset --hard
```

This will restore the repository to the last committed version (called the HEAD revision). Go home, get some rest, and start over.

5.2 Revert – Removing Committed Changes

OK so in this case you made changes, did a commit, and later discovered that you made some bad mistakes that can't be easily fixed. You can undo a commit and go back to the previous version using the command:

```
git revert HEAD~
```

Now you can pretend that the mistakes never happened and start over.

6. Push: Uploading Changes to the Remote Repository

The commands in the previous section introduce modifications and new versions in the local repository on the user's computer. None of these changes are reflected in the master version of the project on the GitHub server that you cloned in Section 3. The process of updating the project on the server is called *pushing* changes to the server repository. This is done for the first time using the command

```
git push –u origin master
```

for later pushes, you can just use:

```
git push
```

Note: it is important to keep in mind that other users may also be pushing changes to the repository. For this reason it is advisable to perform a pull of any changes from other users (as described in the next section) before pushing changes.

7. Pull: Downloading Changes from the Remote Repository

It is often the case that you'd like to work on your project on a different computer or pass your project on to different users such as your lab partner. In order to do this, push any changes you made on your current computer to the master repository and then clone the project on the new computer as described in Section 3. Once this has happened, you can bring these changes back to the original computer by *pulling* these changes from the GitHub server. This is done using the command:

```
git pull
```

8. Advanced Use: Branches and Merging

When used as described in this document, the progression of commits can be visualized as a long string of changes, as shown in Figure 2. However, when multiple users are working with a project Git supports the concept of *branches* in which users working on different parts of the program are developing changes separately.

Git always maintains at least one branch called `master`. However, the project developers may wish to add a separate branch in which new features are added to a different design. Once these new features have been implemented and completely tested the two branches can be merged back together. A new branch is created using the command:

```
git branch branchname
```

In order to actually work with the new branch, it is necessary to *check out* that branch using the command:

```
git checkout branchname
```

A convenient shorthand is to use the -b option of the checkout command, which combines the two commands:

```
git checkout -b branchname
```

Once the new branch is created, any changes and commits moving forward will be associated with the new branch, while the `master` branch remains unchanged. This allows changes to be tested while keeping a stable version. Once the changes in the branch have been tested and debugged, they can be merged back into master by checking out the master and using the `git merge` command as follows:

```
git checkout master
git merge branchname
```

Once the branch has been merged, it can be deleted with the command:

```
git branch -d branchname
```

As an example, Figure 4 shows the steps needed to create a new branch called **`newfeature`** that might be used to add a new feature to a program. While this feature is being developed and debugged, the master branch maintains a stable version of the software. Once development is complete, the new branch is merged back into the master branch.

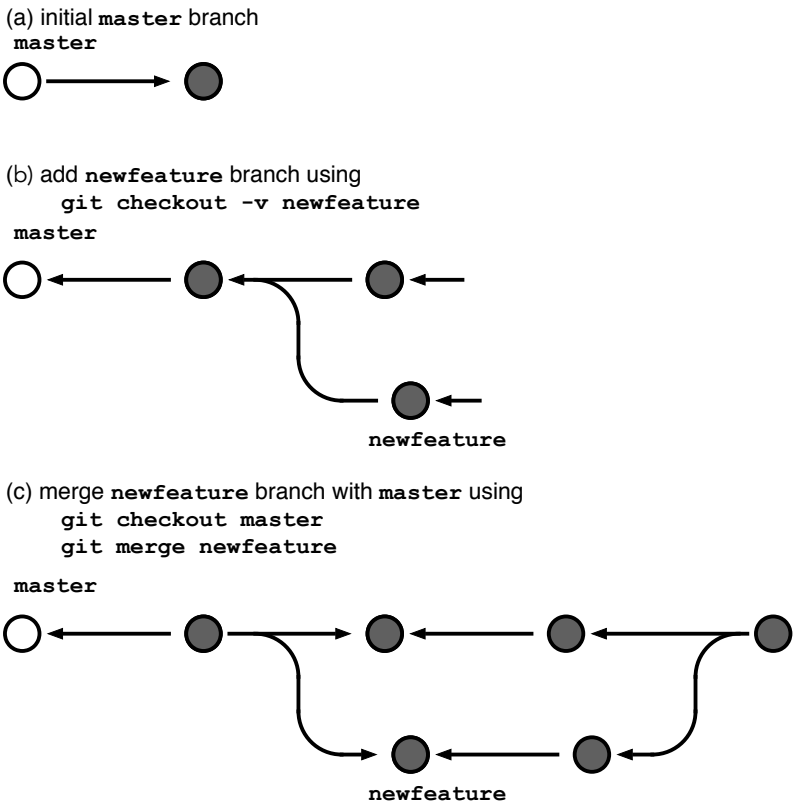

Figure 4 – Branching and Merging Example

9. Glossary

repository (repo)	A directory containing project files along with hidden information to keep track of previous versions of the project. Repositories may be local or remote.
cloning	The process of copying an existing project repository from a remote server onto a local computer. Usually done once to create

	the local repository; after that updating is done by pushing and pulling.
commit	The process of updating changed project files to form a new version of the project in the local repository.
push	The process of uploading project changes in the local repository to the master repository on a remote server (e.g., GitHub)
pull	The process of downloading project changes from the master repository on a remote server (e.g, GitHub) to a local repository.
branch	An alternate stream of versions that is used to make changes while maintaining a stable master version.
merge	The process of combining branches together.

10. Summary

This document has provided an overview of the basic mechanics of using Git for version control. Once a repository has been created and cloned (see Section 2) you will primarily use the following commands to interact with Git:

Command	Interpretation
`git add` *`filename(s)`*	Stage a new (or modified) file for inclusion in the next commit.
`git rm` *`filename(s)`*	Delete one or more files from the repository
`git mv` *`filename newname`*	Change the name of a file
`git mv` *`filename(s) directory`*	Move one or more files to another directory in the repository
`git commit -a -m "`*`commit msg`*`"`	Stage modified and removed files and commit these along with any staged new files. Creates a new version.
`git push`	Push changes up to and including the current version to the GitHub server
`git pull`	Pull changes from the GitHub server
`git branch` *`bname`*	Create a new branch called *`bname`*
`git checkout` *`bname`*	Switch from the current branch to branch *`bname`*
`git merge` *`bname`*	Merge branch *bname* with the currently checked out branch.

For More Information

For more information about git, see the book *Pro Git* by Scott Chacon and Ben Straub. It is available for free online at: https://git-scm.com/book/en/v2