Git for Hardware Designers

Jeffery Scott, Sanjeev Singh
Juniper Networks
2530 Meridian Parkway Suite 2092
Durham NC 27713

***Abstract*- A modern version control system is needed to support the dynamic workflow of Agile hardware design. With the increasing complexity and reduced time for development, front end RTL designers, are forced to support multiple versions of the design and coordinate changes with multiple teams. This requires them to constantly time share their attention across many different contexts. This kind of dynamic workflow is hard to support with conventional centralized version control systems. Commonly designers end up maintaining separate workspaces to support these independent development threads. However this is not a scalable model and makes tracking and merging hard.**

In this paper, we propose using a popular open source distributed version control system, Git, for hardware development. Git is a robust version control system used by some of the largest open source projects. However it is not yet a popular choice for hardware design houses. In this paper we outline some of the challenges faced with the current flow that can be solved using Git.

On a clean slate setup, we would recommend starting with Git. There are many commercial companies offering training and support to enable it. However if companies have already invested heavily in centralized version control, then various intermediate workflows are possible. Designers can start by using distributed version control system (DVCS) for their local work. Then they can export to the central repositories using custom adapter scripts.

I. Introduction

Version control system is an integral part of every development workflow. Traditionally, hardware design houses have been using a central version control system with a single monolithic project centric repository. However there are serious limitations that these impose on the hardware team. A popular alternative used in the software community to alleviate these issues, is a distributed version control system (DVCS) like Git [1]. Git is the defacto standard for open source DVCS, proven to work over large collaborative projects. It is well documented and there are many resources like [2] that provide training and support for companies migrating to it.

In this paper we imagine a scenario, of a team, working on a hardware project like a complex ASIC. The ASIC consists of many hundreds of blocks with a large team of front end designers, verification, emulation and back end folks working on it together.

We list the major features a version control system needs to provide for a hardware designer to work in this environment. An example scenario is proposed where such a feature would be required. These are currently not being handled efficiently with the traditional approach. We then provide a conceptual solution using Git and some recommendations on how it could be implemented.

II. Private Repositories

A. Requirement

RTL designers require entirely isolated development environment. They need their own repositories complete with its own history and branch structure. Changes done to these private repositories do not create clutter in the shared repository.

B. Example Scenario

Designers are constantly balancing various parameters like area, timing, power, new functionality etc. For this they need the ability to experiment with different approaches, without the risk of affecting other team members. These changes may not be trivial and also being worked on in a time shared fashion with the mainstream development. Having a private repository, provides the ability to treat this as a private independent development environment, where one could commit intermediate code and branch without worrying about affecting others. On completion, if unsuccessful these changes can be cleanly dropped, or if successful, merged back to the main branch to be shared with others.

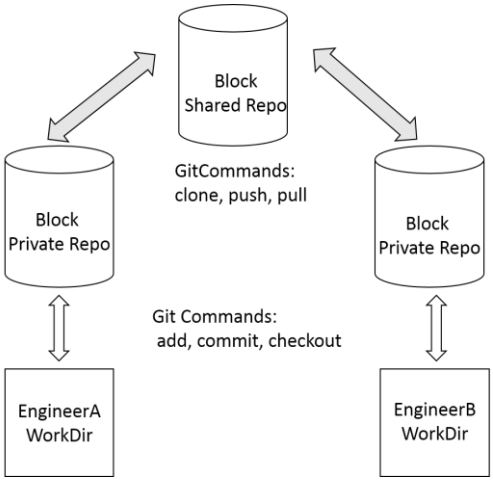

Figure 1 Shared Repository Model

C. Git Solution

Git is a designed to work with many different repositories. When designers clone a repository, they do not just copy the latest snapshot of the files over. They in fact, mirror the entire repository. Hence they have a full history of the project available on their local workspace. From Git perspective, their local repository is equivalent to the remote repository. Any merges between the two is simply a form of repository to repository communication.

This is fundamentally different from traditional centralized version control system (CVCS). All centralized version systems assume a central repository (on a server) where everyone commits their changes. All commits appear on the central repository.

D. Recommendation

As outlined in [3] the "Shared Repository" model is recommended for teams working on private projects. This organization is shown in **Figure** 1**.**

Technically, Git does not attach any semantic value to any repository. However it is useful to create a shared repository in a central area that is the "blessed" repository. Designers can continue developing on their local repositories but treat this central repository (commonly referred to as "origin") as the repository for sharing code with other members of the team and for maintaining the production ready version of the code.

III. COLLABORATIVE WORKFLOW

A. Requirement

RTL designers need to collaborate with different members of the hardware team. Changes for one sub team should not affect other teams. The designer needs to treat these as independent line of development and then merge them into the main trunk manually.

B. Example Scenario

In our scenario, a block designer may be working with

- Design Verification Teams (DV)
    - Block Design Verification (BlockDV)
    - Chip Level Verification (ChipDV)
    - Emulation Team (EmulationDV)
- Formal Verification Team (FV)
- Physical Design Team (PD)
    - Synthesis/Timing/Floor planning

Each of these teams has different requirements. Even though serial workflow would be easiest for the designer, it is not always expeditious and the best use of resources.

A verification team requires the functionality to be working while the PD team may just require that the design be "Lint" clean. Hence a timing fix for the PD team need not wait for the change to be vetted by the verification teams. A designer would like the change to be released to PD team, without exposing it to the DV team. After looking at the results from PD, if the timing fix looks promising, the designer may put in the effort to make it regression clean. If the timing fix does not work, then the change can be quietly discarded and other experiments tried.

In a similar vein the changes required for formal verification are very different from other requests. Formal tools are generally limited by capacity. They may want to reduce the design size by changing the memory sizes to minimum value, replacing some rtl logic with simplified models. Experiments required to keep making forward progress should not affect other team views.

Hardware development also requires many different teams to be working together on the same facet of the design. For example a block being developed, is simultaneously being verified in block level test bench, full chip test bench and also on the emulation platform. Each of these environments, have different intent. A block level environment is focused on intra block details, while a top level environment is focused on inter block interactions and the emulation environment is concerned with system level features. Hence there is a natural order in which design changed must be verified. Unless the block level regressions are clean, the new code should not be released to the top level and only when the top level smoke tests pass; it should be released to emulation. Circumventing this, causes all the teams to trip on the same bugs, increasing closure time and frustration.

C. Git Solution

Git solution is to use "branches" for all these independent line of development. Git allows entire branches to be shared across repositories. In general, Git provides 2 types of branches. "Local" branches are used for development work while "Remote" branches allow collaboration. Designer needs to manually push their commits to the remote branch for it to be shared. Similarly other team members need to manually pull upstream commits into their local repository for accessing it.

In CVCS based environments, branching is not as light weight operation as it is in Git. Branches are more cumbersome and difficult to track and merge back. Hence they are considered as "advanced" topics and used occasionally for capturing a large code changes. Difficulties like these, sometimes force the designers to create multiple copies of the workspace that they share with different sub teams. Copying files between shared folders is difficult to track and merge. Hence it should be discouraged.

In Git branches are an integral part of everyday workflow and so designed to be lightweight and quick to create, rename, merge and delete. Conceptually Git treats branches as pointers to specific commits. Since each commit knows it parent(s), Git can trace the commit history backward to reconstruct what's in the branch. Using this Git can automatically employ several different merge strategies.

- Fast Forward Merge
  If Git detects that you are merging back into the original branch and that there have been no new commits in the original branch, it simply "fast forwards" the branch pointer. No change is done to the repository as shown below in **Figure 2.** In the figure, circles represent the commits and the arrows reflect the parent pointer maintained by Git for all commits. The merge simply causes the master tip to move to the "Branch_x" tip.

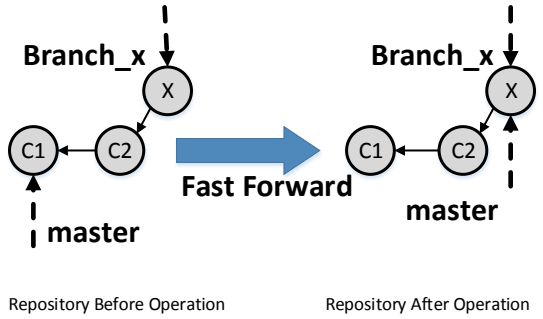

Figure 2 Fast Forward Merge

- Recursive Merge
  This is used when both the branches have commits that are not in the other branch. Git creates a merge commit that has two parent commits which are the latest commit in each branch. This is known as 3-way merge since Git uses 3 commits to create the merge commit: the 2 latest commit and their common ancestor. A conceptual view of the commits before and after the merge is shown below in **Figure 3.**

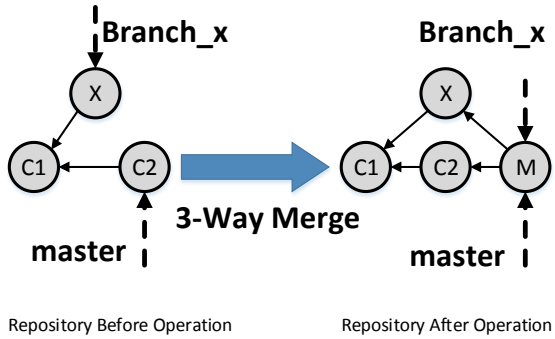

Figure 3 Recursive Merge

- Rebase
  Git rebase allows the user to take a series of commits on a branch and replay them on top of another commit. This prevents cluttering up the commit history with various merge commits and gives a linear development history when the changes are merged. In **Figure 4** the state of the repository is shown when the master branch is rebased on "Branch_x".

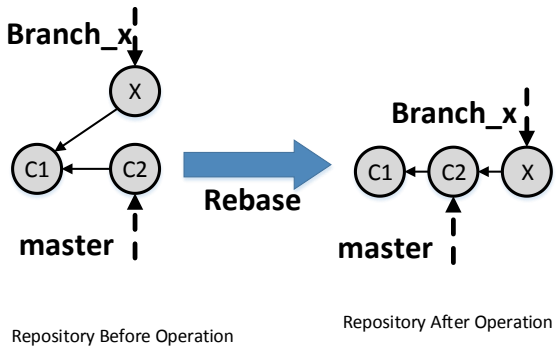

Figure 4 Rebase Operation

D. Recommendation

Technically Git treats all branches as equivalent. But it is useful to have recommended development workflow for managing branches as outlined in [4].

We recommend creating an infinitely living branch on the central server called "origin/develop" that is the state of top of tree. All regressions are run on the tip of this branch and once passing, will be a candidate for making a release. For sharing code between the different sub teams, we recommend creating feature (also called topic) branches. These branches would eventually be deleted either because it is merged back into the develop branch or they are discarded. For example a designer may have many feature branches related to all the independent work being done.

- origin/timing_fix_x
  Used to work with the PD team in trying various RTL changes to fix the timing.
- origin/bug_fix_PR100
  Used to work with the BlockDV team to fix the bugs being seen on the block level regressions.

A possible timeline of the repository is shown below in **Figure 5**. The designer created the above mentioned feature branches and then merged them back to the origin/develop branch once they were successfully complete. The develop branch continued to move forward with other commits while these were being independently worked on.

As part of good practice, feature branches should not be very long running. Designers should frequently sync their feature branches with the develop branch so that they do not stray too far from it. By using the "rebase" strategy, the designers can apply their changes on top of the develop branch, ensuring a linear commit history. For important features it is a good practice to not use the "fast forward" strategy for merging their changes. Fast forward as shown in **Figure 2** does not create a commit history. This makes it harder to trace back when the feature got merged in the develop branch.

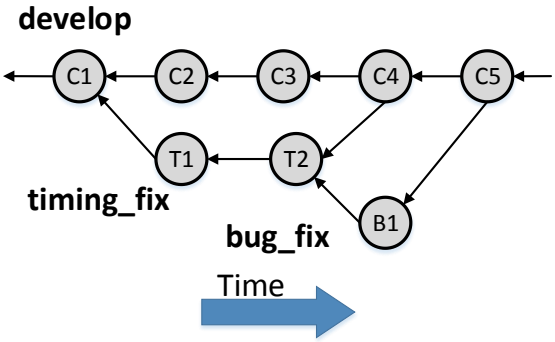

Figure 5 Development Flow

IV. ROBUST RELEASE FLOW

A. Requirement

Hardware development is an iterative process. Hence it needs to make several intermediate releases before the final release. All the release should be tracked cleanly with their own commit logs.

It is expected that the next release to be as good as the previous release but with more functionality. Hence the code must pass all the previous and new tests before it can be released. This process takes time, effort and usually a dedicated resource to do it. During this time while the release is being prepared, the rest of the team continues to work on the next set of features without interrupting the release in progress.

B. Example Scenario

Assume that when all the required feature tests are passing in the develop branch, the "release master" decides to do a release. As part of the release process, a more rigorous test suite needs to run to ensure that none of the previous features were broken. During this time, designers should continue to work on the next feature set by committing to the develop branch. It is possible that the release process may discover some critical bugs during or in the worst case, after the release is done.

In such scenario, it should be possible to fix the code base without restarting the entire release process or waiting for the next scheduled release.

C. Git Solution

Git idiomatic solution is to use a release branch for making releases. We recommend using the "origin/master" as the release branch. Commits on this branch reflect the releases done by the projects. These should be tagged as per the release conventions.

To solve many common release flow issues, Git provides certain unique commands that make the job easier. For example

- Cherry Pick Commits
  Sometimes the branches are so different that merge is not an option. In that case, one can use Git "cherry-pick" to pick specific commits from other branches and then commit them to the local branch as one commit.
- Assigning Blame
  Many bugs can be traced back to certain lines in code. Using Git "blame" one can quickly find out when that problem line was introduced in the repository and use that as the starting point of the debug.
- Find Bad Commit
  Git "bisect" command can help the designer to search through the repository to find the commit that has the bug in it. It works by repeatedly dividing the commit history between the "good" commit and the "bad" commit to find the commit that introduced the bug.

D. Recommendation

As mentioned in Section III, the branch "origin/develop" represents the top of tree. When this branch reflects all the required features for making a release, a temporary "release" branch is created for preparing for the release. The team continues to work on the next set of features by using the "origin/develop" branch. It is possible that bug fixes might be done on this "release" branch before the code can be released into the "origin/master" branch. If required, these fixes should be also merged back into the "origin/develop" branch so that they are part of the next release. A possible timeline using this flow is shown below in **Figure 6.**

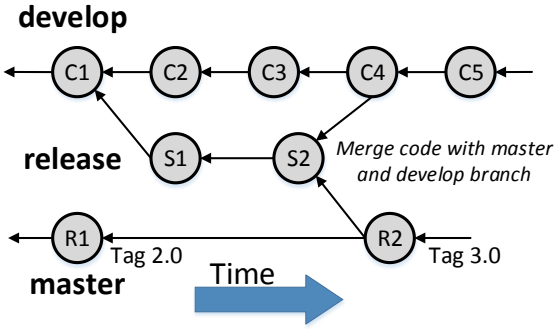

Figure 6 Release Flow

To handle unplanned release, hotfix branches can be used. These can be branched from the current release or earlier releases as required. Once the fixes are done, the changes can be merged back into the development branch. An example flow is shown in Figure 7. A hot fix branch is created to provide a fix for release 2.0. This is released as 2.1. The fixes done are then merged back into the "develop" branch so that they are part of the next release 3.0.

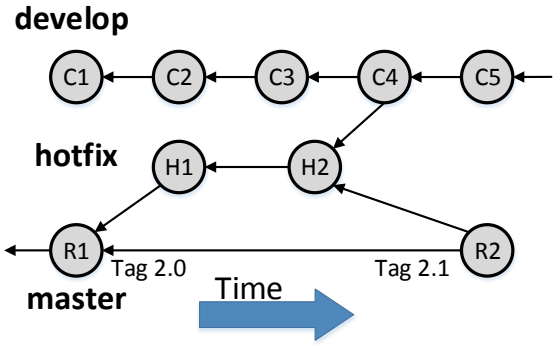

Figure 7 Hotfix Flow

Traditional CVCS flows use tagging to achieve this. However tags do not allow the independence that a branch provides.

V. BLOCK CENTRIC REPOSITORIES

A. *Requirement*

Hardware blocks should be able to be reused across multiple designs. The repositories should be self-contained and dependencies to other blocks minimized. Checkout time, build time should be optimum and the commit logs should be clear and relevant.

The top level design should be viewed as a composite repository of block level repositories. A dependency management tool should be used to track the version dependencies of the sub projects and the parent project.

B. *Example Scenario*

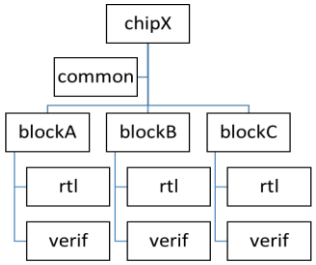

Figure 8 Traditional Project Repository

Assume a block being used in chip 'X' project with a single repository structure like shown in **Figure 8**. Once chip 'X' is taped out, one of the block (say blockA) with some new features is being used in chip 'Y'. The designer would like to add these incremental changes in the same environment and treat the new features as a new release. But with the repository structure in **Figure 8**, it is hard to figure out the dependencies of the block. Copying blockA files to the new chipY repository is not a good idea since it causes code duplication. Any bugs found in blockA, may need to be applied manually to both the versions of the block.

C. *Git Solution*

Git provides the concept of "submodules'" to create "parent child" relation between repositories. It allows us to divide the monolithic repository into components which are separate Git repositories. Each of these components hold one set of coherent files.

Git by itself does not provide any dependency management system between the parent and the sub projects. Use an external tool similar to "Maven" or "RubyGems" for this purpose.

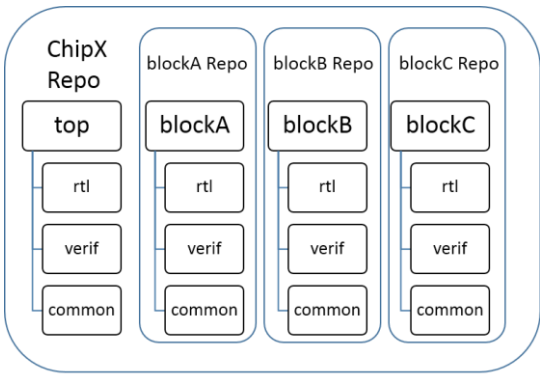

Figure 9 Recommended Repository Layout

D. *Recommendation*

As outlined in [5] Git fundamentally looks at the whole repository and carries the entire information around. So Git will scale very badly if there is one whole repository.

We recommend the following approach for dividing the repository into multiple repositories and using submodules to create composite repositories.

- Create a repository for each conceptual group. .
- Create separate repositories for large binary files for example layout related database, timing reports etc.

Following this recommendation we can re organize our project repository to be more block centric. This is shown in **Figure 9**. Each block can be made its own repository. The chip level repository includes the block repositories as submodules. This clearly exposes the external dependencies that the block has and make moving to other projects painless.

VI. SUMMARY

There are big benefits to using Git for hardware development. For teams moving from CVCS to Git, a summary of the pros and cons is shown in **Table 1**.

Table 1 Pros/Cons of Using Git

Feature	**Pros**	**Cons**	**Comment**
Private Repositories	Provide isolated environments to experiment without cluttering up the central repository.	Require infrastructure support for backups and secure access to the central repository.	With cloud computing this is not an issue.
Robust Collaborative and Release flows	Branch with advanced merging and sharing features allow more flexible flows.	Require training and support for the new workflows.	Many commercial companies offering such training.
Block Centric Repositories	By providing a repository for each block, promote IP centric workflows. This reduces dependencies making block reuse much easier.	Reorganizing a chip centric repository into multiple block level repositories is not a trivial task.	Help reuse in the long run.

VII. REFERENCES

[1] Git [Online]. Available: http://www.scm-git.com
[2] Getting Git Right: Available https://www.atlassian.com/git/
[3] Github: Available http://help.github.com
[4] A Successful Git branching model by Vincent Driessen . Available: http://nvie.com/posts/a-successful-git-branching-model/
[5] Git repository size. Available: http://stackoverflow.com/questions/984707/what-are-the-file-limits-in-git-number-and-size/984973#984973