Chisel Release Process

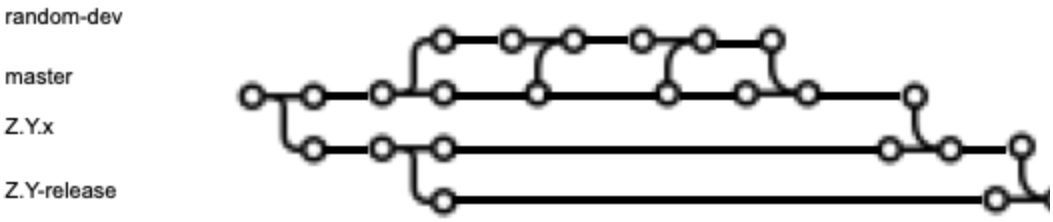
Branch Management

We assume that each submodule supports the following branch naming scheme:

master - main development branch - should always be buildable, may not be compatible with other master branches for several days

Z.Y.x branch - binary compatible with Z.Y-release branch, may be momentarily incompatible with related .x branches

Z.Y-release - under complete control of the release process. Always compatible with release-related branches of other repositories.



We make use of the fact that Git submodule support allows one to specify the branch to be used for each submodule. This allows us to deal with different branches for each submodule in each branch of the top level chisel-release repository.

The simplest case is the master branch of chisel-release. Its .gitmodule file looks like:

```
[submodule "chisel3"]
        path = chisel3
        url = https://github.com/freechipsproject/chisel3.git
        branch = master
[submodule "chisel-testers"]
        path = chisel-testers
        url = https://github.com/freechipsproject/chisel-testers.git
        branch = master
[submodule "firrtl"]
        path = firrtl
        url = https://github.com/freechipsproject/firrtl.git
        branch = master
[submodule "firrtl-interpreter"]
        path = firrtl-interpreter
        url = https://github.com/freechipsproject/firrtl-interpreter.git
        branch = master
[submodule "dsptools"]
        path = dsptools
        url = https://github.com/ucb-bar/dsptools.git
        branch = master
[submodule "treadle"]
        path = treadle
        url = https://github.com/freechipsproject/treadle.git
        branch = master
[submodule "diagrammer"]
        path = diagrammer
        url = https://github.com/freechipsproject/diagrammer
        branch = master
[submodule "chisel-testers2"]
        path = chisel-testers2
```

```
        url = https://github.com/ucb-bar/chisel-testers2.git
        branch = master
[submodule "chisel-template"]
        path = chisel-template
        url = https://github.com/freechipsproject/chisel-template.git
        branch = master
[submodule "chisel-tutorial"]
        path = chisel-tutorial
        url = https://github.com/ucb-bar/chisel-tutorial.git
        branch = master
```

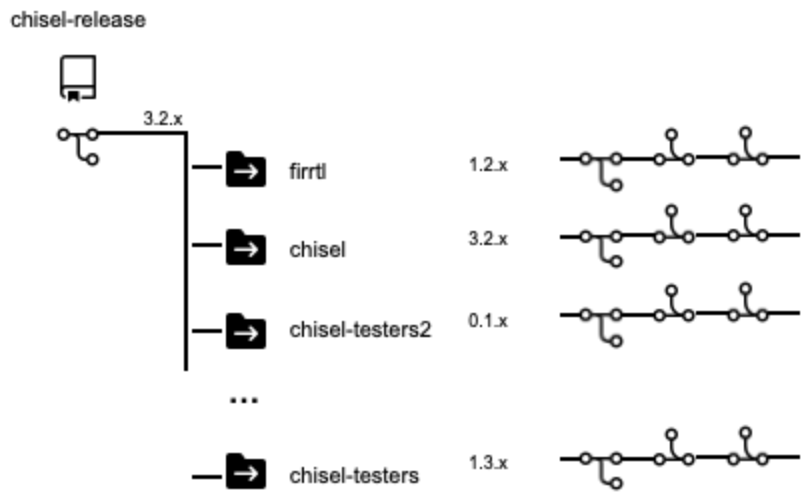
This indicates that the master branch of chisel-release uses the master branch of each submodule[1].

Slightly more interesting is the 3.2.x branch of chisel-release:

```
[submodule "chisel3"]
        path = chisel3
        url = https://github.com/freechipsproject/chisel3.git
        branch = 3.2.x
[submodule "chisel-testers"]
        path = chisel-testers
        url = https://github.com/freechipsproject/chisel-testers.git
        branch = 1.3.x
[submodule "firrtl"]
        path = firrtl
        url = https://github.com/freechipsproject/firrtl.git
        branch = 1.2.x
[submodule "firrtl-interpreter"]
        path = firrtl-interpreter
        url = https://github.com/freechipsproject/firrtl-interpreter.git
        branch = 1.2.x
[submodule "dsptools"]
        path = dsptools
        url = git@github.com:ucb-bar/dsptools.git
        branch = 1.2.x
[submodule "treadle"]
        path = treadle
        url = https://github.com/freechipsproject/treadle.git
        branch = 1.1.x
[submodule "diagrammer"]
        path = diagrammer
        url = https://github.com/freechipsproject/diagrammer
        branch = 1.1.x
[submodule "rocket-chip"]
        path = rocket-chip
        url = https://github.com/chipsalliance/rocket-chip.git
        branch = 1.2.x
[submodule "chisel-testers2"]
        path = chisel-testers2
        url = https://github.com/ucb-bar/chisel-testers2.git
        branch = 0.1.x
```

[1] The master branch includes the chisel-template and chisel-tutorial submodules although these submodules are not published. Their inclusion is purely to facilitate testing, although current code does not take advantage of this.

Here, the submodule branches are the compatible .x branches[2].



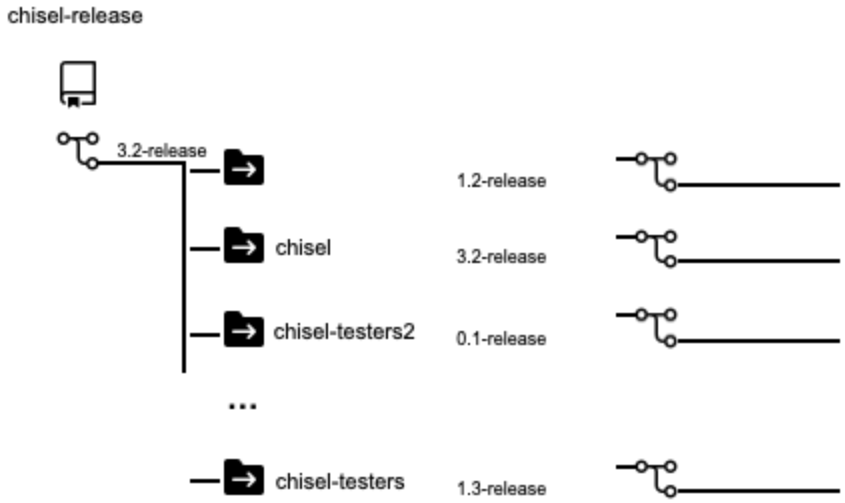
The 3.2-release version of .gitmodules is:

```
[submodule "chisel3"]
        path = chisel3
        url = https://github.com/freechipsproject/chisel3.git
        branch = 3.2-release
[submodule "chisel-testers"]
        path = chisel-testers
        url = https://github.com/freechipsproject/chisel-testers.git
        branch = 1.3-release
[submodule "firrtl"]
        path = firrtl
        url = https://github.com/freechipsproject/firrtl.git
        branch = 1.2-release
[submodule "firrtl-interpreter"]
        path = firrtl-interpreter
        url = https://github.com/freechipsproject/firrtl-interpreter.git
        branch = 1.2-release
[submodule "dsptools"]
        path = dsptools
        url = https://github.com/ucb-bar/dsptools.git
        branch = 1.2-release
[submodule "treadle"]
        path = treadle
        url = https://github.com/freechipsproject/treadle.git
        branch = 1.1-release
[submodule "diagrammer"]
        path = diagrammer
        url = https://github.com/freechipsproject/diagrammer
        branch = 1.1-release
[submodule "rocket-chip"]
        path = rocket-chip
        url = https://github.com/chipsalliance/rocket-chip.git
        branch = 1.2-release
[submodule "chisel-testers2"]
        path = chisel-testers2
```

[2] For historical reasons, rocket-chip was included as a submodule for the 3.2 .x and -release branchess.

```
        url = https://github.com/ucb-bar/chisel-testers2.git
        branch = 0.1-release
```

Here, the submodule branches are the compatible -release branches.



This allows us to write generic scripts to deal with each submodule that avoids having to encode a specific branch in the script.

A shell command to update each submodule to the head of its associated branch is:

```
git submodule foreach \
  'sbranch=$(git config -f $toplevel/.gitmodules submodule.$name.branch);
   git fetch origin $sbranch && git checkout $sbranch && git pull
  '
```

A shell command to merge the .x branches into the -release branches (assuming we're on the release branches and without committing the merge) is:

```
git submodule foreach \
  'sbranch=$(git config -f $toplevel/.gitmodules submodule.$name.branch);
   xbranch=$(echo $sbranch | sed -e 's/-release/.x/');
   git merge --no-ff --no-commit $xbranch;
  '
```

Typically, not all the submodules incorporated in chisel-release should participate in the foreach loops. For example, rocket-chip is a submodule of the 3.2 .x and -release branches, but should no longer be included in general submodule expressions. chisel-template and chisel-tutorial are included in the master branch of chisel-release, but they should not be included in general submodule expressions. We implement this by wrapping the foreach expression in a shell conditional. For example, to eliminate rocket-chip from the 3.2 submodule processing:

```
git submodule foreach \
  'if [ $name != "rocket-chip" ]; then
   sbranch=$(git config -f $toplevel/.gitmodules submodule.$name.branch);
   git fetch origin $sbranch && git checkout $sbranch && git pull
   ; fi
  '
```

To do the same thing for chisel-tutorial and chisel-template in the master branch:

```
git submodule foreach \
  'if [ $name != "chisel-template" -a $name != "chisel-tutorial" ]; then
   sbranch=$(git config -f $toplevel/.gitmodules submodule.$name.branch);
   git fetch origin $sbranch && git checkout $sbranch && git pull
   ; fi
  '
```

Periodic Testing (07/27/20)

1. clone chisel-release repository

```
git clone https://github.com/ucb-bar/chisel-release.git
```

2. clone chisel-repo-tools repository

```
git clone https://github.com/ucb-bar/chisel-repo-tools.git
```

3. update master branch

3.1. checkout master branch

```
git checkout master
git pull
```

3.2. populate submodules

```
git submodule update --init --recursive
```

3.3. position submodules at head of master branches

```
make pull
```

3.4. run tests

```
make -j 8 clean install test
```

3.5. update top level

```
git add -u
```

3.6. commit

```
git commit -m "Bump master branches"
```

3.7. push[3]

```
git push
```

[3] Note: we don't modify the submodules so we don't need to (and we shouldn't) push them.

Snapshot (from master) Preparation Details (07/27/20)

This assumes that changes in master need to be manually merged with the Z.Y.x current release branch.

1. clone chisel-release repository

```
git clone https://github.com/ucb-bar/chisel-release.git
```

2. clone chisel-repo-tools repository

```
git clone https://github.com/ucb-bar/chisel-repo-tools.git
```

3. update master branch

   3.1. setup virtualenv

   source ~/.virtualenvs/cit3/bin/activate

   3.2. cd chisel-release

   3.3. checkout master branch

```
git checkout master
git pull
```

   3.4. populate submodules

```
git submodule update --init --recursive
```

   3.5. position submodules at head of master branches

```
make pull
```

   3.6. update top level

```
git add -u
```

   3.7. commit

```
git commit -m "Bump versions"
```

   3.8. push[4]

```
git push
```

[4] Note: we don't modify the submodules so we don't need to (and we shouldn't) push them.

4. update .x branch

4.1. checkout .x branch

```
git checkout <<major>>.<<sub-major>>.x
git pull
```

4.2. populate submodules

```
git submodule update --init --recursive
```

4.3. position submodules at head of .x branch

```
make pull
```

4.4. update top level[5]

```
git add -u
```

4.5. commit

```
git commit -m "Bump versions"
```

5. merge master branches into .x branches

5.1. merge individual .x branches with their respective master

```
git submodule foreach '
  if git diff --cached --quiet; then git merge --no-ff --no-commit master;
  fi
'
```

5.2. verify

```
git status -b -uno --ignore-submodules=untracked
python $VERSIONING verify
```

6. test merged configuration

6.1. verify verilator

```
verilator -version
```

6.2. verify yosys

```
yosys -V
```

[5] In principle, there will be no changes if this is the same copy of the repository from which the previous snapshot was released.

6.3. verify z3

```
z3 --version
```

6.4. run tests

```
make -j 8 clean install test >& make-clean-install-test.out
```

6.5. check test results

```
# see if any failures happened, if so debug away
grep 'ests: succeeded' make-clean-install-test.out
tail -100 make-clean-install-test.out
grep '\[error\]' make-clean-install-test.out
```

7. commit merges

7.1. commit each submodule[6]

```
git submodule foreach 'if git diff --cached --quiet ; then echo skipping ;
else git commit --no-edit ; fi'
```

7.2. push the submodules

```
git submodule foreach 'git push'
```

7.3. add updated submodules

```
git add -u
```

7.4. commit top level

```
git commit -m "Release Z.Y.X prep"
```

7.5. push top level

```
git push
```

[6] Commit the submodule if it has staged changes

Release Process Details (06/25/20)

1. clone chisel-release repository

```
git clone https://github.com/ucb-bar/chisel-release.git
```

2. clone chisel-repo-tools repository

```
git clone https://github.com/ucb-bar/chisel-repo-tools.git
```

Set up the python virtual environment for the release and define the PYTHONPATH and VERSIONING environment variables. And setup python virtual environment
source ~/.virtualenvs/cit3/bin/activate

3. update .x branch

3.1. checkout .x branch

```
git checkout Z.Y.x
git pull
```

3.2. populate submodules

```
git submodule update --init --recursive
```

3.3. position submodules at head of .x branches

```
mkdir stamps
make pull
```

3.4. update top level

```
git add -u
```

3.5. commit

```
git commit -m "Bump versions"
```

3.6. push[7]

```
git push
```

[7] Note: we don't modify the submodules so we don't need to (and we shouldn't) push them.

4. update -release branch

4.1. checkout -release branch

```
git checkout Z.Y-release
git pull
```

4.2. populate submodules

```
git submodule update --init --recursive
```

4.3. position submodules at head of branch

```
make pull
```

4.4. update top level[8]

```
git add -u
```

4.5. commit

```
git commit -m "Bump versions"
```

4.6. push

```
git push
```

5. bump -release branch versions

5.1. user $VERSIONING script to bump versions (USE ONLY ONE OF THESE)

5.1.1. date-stamped SNAPSHOT

```
python $VERSIONING -s "20200630" write
python $VERSIONING -s "" write
```

5.1.2. bump major version

```
python $VERSIONING bump-maj
```

5.1.3. prepare major release candidate

```
python $VERSIONING -r "RC<<candidate-number>>" write
```

5.1.4. new release after candidates

```
python $VERSIONING -r "" write
```

5.1.5. bump minor version

```
python $VERSIONING bump-min
```

[8] In principle, there will be no changes if this is the same copy of the repository from which the previous minor version was released.

5.2. verify version bumps

```
git diff --submodule=diff
```

The only differences should be the version.yml file and the versions in the submodule build.sbt and build.sc files.

5.3. commit version bumps

```
git submodule foreach '
  git add -u && git commit -m "Bump version strings." '
```

5.4. add updated branches and update version.yml

```
git add -u
```

5.5. commit updated branches and update version.yml

```
git commit -m "Bump version strings."
```

6. merge .x branches into -release branches

6.1. run merge script, you may need to loop on this step while fixing any conflicts in the submodules (this will typically be in the build.* files)

```
./merge-release-with-x.sh
```

6.2. verify

```
git status -b -uno --ignore-submodules=untracked
```

7. test release configuration

7.1. verify verilator

```
verilator -version
```

7.2. verify yosys

```
yosys -V
```

7.3. verify z3

```
z3 --version
```

7.4. run tests

```
make -j 8 clean install test >& make-clean-install-test.out
```

7.6. check test results

```
# see if any failures happened, if so debug away
grep 'ests: succeeded' make-clean-install-test.out
tail -100 make-clean-install-test.out
grep '\[error\]' make-clean-install-test.out
```

8. commit merges

8.1. commit each submodule[9]

```
git submodule foreach '
if git diff --cached --quiet ; then echo skipping ; else
git commit --no-edit
fi
'
```

8.2. add updated submodules

```
git add -u
```

8.3. commit top level

```
git commit -m "Release Z.Y.X"
```

9. push release

9.1. push submodules

```
git submodule foreach '
  git push
'
```

9.2. push top level

```
git push
```

10. publish release

10.1. publish signed

```
make +publishSigned
```

[9] Commit the submodule if it has staged changes

10.2. login to oss.sonatype.org

10.3. select staging repositories

10.4. select edu.berkeley.cs

10.5. select individual staging repository items & verify content

10.6. get sha of top level chisel-release commit

```
git log -1
For example:
2b1450bea5f6f87d99b1a07fe18f54190189c470
```

10.7. select all staging repository items & click Close

10.8. paste sha, version, and date into dialog box & click confirm

- 10.8.1. Example
- 10.8.2. 6b6f24376c3ac044c16bcc7e0b7e7b5ed282a5c0
- 10.8.3. 3.4.0-RC1
- 10.8.4. 20200816
- 10.8.5.

10.9. select individual staging repository item and monitor actions (requires clicking on Refresh)

10.10. if errors

10.10.1. diagnose error

10.10.2. delete (control click) failing artifacts (on Sonatype)

10.10.3. fix errors

10.10.4. push fixed submodules

10.10.5. close fixed submodules (on Sonatype)

10.11. select all staging repository items & click Release

11. tag submodule release branches[10]

```
 git submodule foreach '
 rbranch=$(git config -f $toplevel/.gitmodules submodule.$name.branch);
 xbranch=$(echo $rbranch | sed -e "s/-release/.x/");
 eval git tag $(../genTag.sh $xbranch)
'
>After running above (assuming all looks good) change the echo to an eval and
re-run.

git submodule foreach 'git describe'
git submodule foreach 'git push origin $(git describe)'
```

12. tag top level branch

```
eval git tag $(./genTag.sh Z.Y.x vZ.Y.X)
git describe
git push origin $(git describe)
```

13. publish SNAPSHOT version[11]

In general, whenever a fixed release is published (either date-stamped SNAPSHOT or major or minor release, a corresponding non-date-stamped SNAPSHOT should also be published. To do this, temporarily change the published version for each submodule (but not its dependencies). This ensures that builds using the SNAPSHOT will be reproducible, at least until the next SNAPSHOT is published.

13.1. checkout -release branch

```
git checkout Z.Y-release
```

[10] To test this to verify the tag contents; replace the `eval` with an `echo`.
[11] We publish an un-date-stamped SNAPSHOT corresponding to the most recently published version.

```
git pull
```

13.2. populate submodules

```
git submodule update --init --recursive
```

13.3. temporarily set root versions for each submodule

```
python $VERSIONING -s "" --onlyroot write
```

13.4. verify the (temporary) changes

```
git diff --submodule=diff
```

13.5. clean and install

```
make -j 8 clean install
```

13.6. publishSigned[12]

```
make +publishSigned
```

13.7. undo version changes

```
git checkout version.yml
python $VERSIONING write
```

or

```
git checkout version.yml
git submodule foreach 'git checkout $(git ls-files --modified)'
```

14. generate change log

15. publish version changelogs to GitHub

[12] Publishing SNAPSHOTs doesn't currently require any activity on Sonatype.

ChangeLog Details (06/25/20)

For running on mac use `brew tap mongodb/brew`

1. checkout version

```
git checkout Z.Y-release
git pull
git submodule update --init --recursive
git submodule foreach 'git fetch origin'
```

2. populate database with pull requests/issues[13]

```
export GHRPAT=XXX
where XXX is your GITHUB token
git submodule foreach '
  cd .. &&
  python $PYTHONPATH/repoissues2db/repoissues2db.py -r $name -s 2020-04-01
'
```

3. verify version tag selection is accurate[14]

```
git submodule foreach '
  branch=$(sh ../major-version-from-branch.sh) &&
  tags=($(git tag -l --sort=v:refname |
  grep -v SNAPSHOT |
  tail -n 2));
  echo ${tags[0]} .. ${tags[1]}
'
```

4. generate git log one-liners

```
git submodule foreach '
  branch=$(sh ../major-version-from-branch.sh) &&
  tags=($(git tag -l --sort=v:refname |
  grep -v SNAPSHOT |
  tail -n 2));
  echo ${tags[1]};
  git log --oneline ${tags[0]}..${tags[1]} > releaseNotes.${tags[1]}
'
```

5. generate changelog

```
git submodule foreach '
  branch=$(sh ../major-version-from-branch.sh) &&
  tags=($(git tag -l --sort=v:refname |
```

[13] Database population is version agnostic and only required once regardless of the number of version changelogs to be generated.

[14] This and the following step's usage of major-version-from-branch.sh may require some sed massaging if we need to quote the embedded '.' in the version string for use in grep's regex.

```
  grep -v SNAPSHOT | tail -n 2));
  echo ${tags[1]};
  python $PYTHONPATH/gitlog2releasenotes/gitlog2releasenotes.py -b git-$name
releaseNotes.${tags[1]}
' > changelog.txt
```

6. edit changelog

   Manually edit the changelog.txt file
   Goal is one line per PR

7. update github

   Select repo
   Releases
   Draft new release
   Add version number
   Add description, either
   - list of PR's or
   - 'Bump dependencies'

Example
go to https://github.com/freechipsproject/firrtl
click on release
click on draft new release
paste in text
set tag to 1.4.0
save it.

Edit data from changelog.txt from step 6 above.
Fiddle with it
Save draft -- gives chance for others to review
click `This is pre-release`

8.

Publishing unstamped SNAPSHOTs

The goal of publishing periodic releases of the Chisel repositories is to offer clients a compatible collection of cooperating libraries that can be used in reproducible builds. A release (and its dependencies) are designed to be stable and unchanging. If you build with the 3.2.6 release of chisel3, that code (and its firrtl dependencies) will not change.

To support reproducible builds, we publish date-stamped SNAPSHOTs (chisel3_2.12-3.2-20191106-SNAPSHOT.jar), release candidates (chisel3_2.12-3.2.0-RC2.jar), and minor releases (chisel3_2.12-3.2.0.jar). These are guaranteed to be stable and unchanging.

Some clients (particularly CI[15] workflows) may want something more current, without having to update their build scripts as new releases are published. To that end, we publish an un-date-stamped SNAPSHOT corresponding to each release. It's basically the same code as the latest date-stamped SNAPSHOT or minor release, but published with a generic version (i.e., 3.2-SNAPSHOT).

There are two ways to accomplish this.

The first (the more complicated but more correct) way, is to replace the version number of each submodule with its generic, un-date-stamped SNAPSHOT version, and re-publish the release. The code will have the same dependencies as the original, but will be published (and accessible) under the generic SNAPSHOT name. These version changes are temporary and should be discarded after publishing.

That is, after publishing 3.2.6, we use version tooling to change the top-level versions of each submodule to the equivalent un-date-stamped SNAPSHOT. We don't change the default versions of the dependencies, so the code is built with and uses the same dependencies as the equivalent 3.2.6 release. Think of it as a trivial name change: the jar chisel3_2.12-3.2.6.jar is also published as chisel3_2.12-3.2-SNAPSHOT.jar.

The second (the simpler but potentially less correct) way, is to publish the .x branch that was merged into the -release branch. In principle, these should contain the same code as the -release branch, but with the generic SNAPSHOT versions. Additionally, its dependencies will be the generic SNAPSHOT versions, so should they change, it will as well.

This means that you cannot use un-date-stamped SNAPSHOTs in builds you expect to be reproducible, but if your intention is simply to test against an internally consistent suite of Chisel code, representing the most recently published version of a major release, SNAPSHOTs should allow you to do that.

[15] Continuous Integration

Troubleshooting

- what did I do
  Keep a log (bash history or Emacs shell buffer) of every command you issue. Capture output of long running steps (clean, install, test, +publishSigned) for later analysis in case of errors.
- where am I
  Verify that you're in the directory you think you should be in. I occasionally find I'm up or down a level from where I thought I was.
- what's changed
  Become familiar with the `git status` command, especially `git status -b -uno --ignore-submodules=untracked`. I use this often to confirm that what I expect to have changed matches reality. Before changing the state of the world, think about how you would revert that change, and how you would verify that the results of the change are what you expected.
- publishSigned fails
  Verify your Sonatype credentials are correct. To test the signing code without interacting with Sonatype, you can do a `publishLocalSigned`
- sonatype closing fails
  Check the actions associated with the failing staged repository. There should be an indication of which specific action failed. Delete the failing repository (all Scala versions), fix the error (typically in build.sbt) and retry `+publishSigned` for the failing repository.
- build fails
  Typically some API has changed without updating downstream dependencies. Try to reproduce the problem in the master or Z.Y.x branch. This should be brought to the attention of the developers and fixed in the master or Z.Y.x branch.
- test fails or hangs
  Determine if this is due to the testing infrastructure (Java, ScalaTest, parallelism). Try to reproduce the problem in the master or Z.Y.x branch
- git clone/checkout fails
  - filesystem case insensitive
  - submodule push missing
    The top level module refers to a commit in a submodule that hasn't been pushed to GitHub. If you can't find the commit, you'll need to reproduce it (which may be impossible), or generate a new release.
- git push fails
  Someone has pushed another version. This should never happen on the Z.Y-release branches, since only the release process is permitted to push changes to these branches, and you shouldn't be pushing to any other submodule branch during the release process.

- make push fails

`make push` requires a clean working tree. Stash or undo any changes to the top level and submodule directories and do a
`git pull && git submodule update --init --recursive`.

- no tags are found when trying to generate changelog
  The goal is to find the last two non-SNAPSHOT tags and the commits between them. The logic trying to restrict which kinds of tags to consider may be too restrictive, or may be making incorrect assumptions about the relationship between branches and tags.

Authentication

cat ~/.sbt/1.0/sonatype.sbt

GIT considerations

A recent change to `git` has caused the following messages to occur when running `git pull` see: StackOverflow: warning-pulling-without-specifying-how-to-reconcile
I have run the following command
`git config pull.rebase false ; git submodule foreach 'git config pull.rebase false'` in the master branch of chisel-release on 10/22/2020

`

Appendix - Python Virtual Environment

We use a python virtual environment in which to run the various python repository tools. This currently requires Python 3.6 or 3.7 and can be configured as follows:

1. install the generic python virtualenv
2. mkdir ~/.virtualenvs
3. virtualenv -p python3 ~/.virtualenvs/cit3[16]
4. pip install --upgrade pip
5. pip install --upgrade setuptools
6. source ~/.virtualenvs/cit3/bin/activate
7. pip install -r requirements.txt[17]

If you're running python 3.6, you need to install dataclasses (which is built-in in 3.7)

7.1. pip install dataclasses

To activate the virtual environment:

1. source ~/.virtualenvs/cit3/bin/activate
2. set environment variables
   2.1. export PYTHONPATH=/Users/chick/Adept/dev/release-generators/chisel-repo-tools/src
   2.2. export VERSIONING=/Users/chick/Adept/dev/release-generators/chisel-repo-tools/src/versioning/versioning.py
3.

[16] The name is historic and stems from "continuous integration and test" using python 3.
[17] requirements.txt