# EECS150 Fall 2013 Checkpoint: DVI Test Pattern

Authored by Austin Buchan
Prof. Ronald Fearing, GSIs: Austin Buchan, Stephen Twigg
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Due week of Oct. 24. Revision 1.0

## 1 Introduction

This checkpoint will familiarize your team with Git version control for HDL design, and the DVI output for displaying graphics on a monitor. In addition you will practice using ready/valid signals, and writing a simulation testbench for synthesizable code.

## 2 Prelab

There is no prelab checkoff for this checkpoint. It will help you get started in lab much faster if you can post your team information to Piazza before your lab according to the instructions in the next section.

## 3 Getting Started with Git

This section is mostly copied from the CS250 Lab 1 Handout.[1]

You will be using Git to manage your CS150 laboratory assignments. Please see the Git tutorial posted on the CS250 course website for more information about how to use Git with HDL.[2] In this class we will mainly support command-line interactions with git, but if you are familiar with other repository management software[3] feel free to use it.

Each student will have a private git repository hosted on github.com. If you don't already have a Github account, you will need to create one. Once you and your partner have an account, make a Piazza private post to instructors with the following information (one per team):

- "Team Info, Wed Section" as the post summary (replace Wed with the lab session you will be attending)
- Your names

[1]`http://www-inst.eecs.berkeley.edu/~cs250/fa13/handouts/lab1-gcd.pdf`
[2]`http://www-inst.eecs.berkeley.edu/~cs250/fa13/handouts/tut1-git.pdf`
[3]`http://git-scm.com/downloads/guis`

- cs150 login names
- GitHub account names

Once you do this, your GSI will create a private repository for you and you will be able to access the lab materials. You can access your private repo from anywhere by logging into GitHub.com, and checking https://github.com/EECS150/fa13_teamXX (with XX replacing your team number).

The project materials we provide will be hosted in a template repository. You will clone this template repository to a directory on the machine you're working on. Afterwards, you will set your remote repository to point at your private repository. This will create a local repository that is linked to two different remote repositories (one which is managed by the staff and is read only, while the other is your private repository). If any updates are made to the template repository, you should be able to easily merge the changes into your local repository.

The two remote repositories are named `template` and `origin`. `origin` points to your private repository and `template` points to the read-only staff account. If the provided project files are ever updated, a simple "`git pull template master`" should merge in these changes with your own local versions of the files.

After you have your team number from the GSI, you can initialize your repostory as follows. Make sure you replace fa13_teamXX with your team number:

```
mkdir fa13_teamXX
cd fa13_teamXX
git init
git remote add template https://github.com/EECS150/fa13_template
git remote add origin https://github.com/EECS150/fa13_teamXX
git pull template master
git pull origin master

  (do the following only once per team)

git push origin master
```

This procedure above needs to be done any time you initialize a new local copy of your repository. Note that the very last `push` only needs to be done once per team. Subsequent pushes from team members will be after you have made changes to the code. All team members should execute the other lines in their home directory when logged into the p380 machines so that everyone has their own local copy.

The Git tutorial listed above is a great resource, but here are a few major DOs and DON'Ts to keep your repository in a usable state:

## 3.1 Git DOs

- Add team member names and login info to the README.md file in the repo root directory.
- Commit early and often, with useful messages.
- Use `git status` often to make sure git has seen the changes you expect.

- Keep your repository organized in functional units (i.e. have a directory for each major component: dvi, sram, detector, etc.). "`git mv`" will help you move files while maintaining their version history.

- If there is some repository manipulation you don't know how to do, odds are there is already great documentation written for it. Your search path should be the tutorial, then internet, then GSIs.

### 3.2 Git DON'Ts

- Edit the same file from different repositories, especially in the same file position. If necessary, use frequent push/pull to avoid conflicts.

- Add large, generated content (bitfiles, coregen builds) to the repo. See the documentation on a "`.gitignore`" file to learn how to automatically exclude certain files from version control.

- Keep copies of slightly changed or old versions of files in the repo (this is what version control does for you!). "`git checkout, revert,` and `reset`" will let you use previous versions of files, and undo local changes.

## 4 DVI Test Pattern Generator

Now that you have a working version control system, you will begin working on the first component of the project. The DVI test pattern generator is not too technically difficult, but it will exercise your version control skills before you work on critical pieces.

### 4.1 Conditional Compilation Macros

You should first examine `hardware/src/FPGA_TOP_ML505.v`. Notice the lines:

```
`define DVI_ENABLE
`ifdef DVI_ENABLE
  ...
`else
  assign DVI_D = 0;
  ...
`endif
```

The `` `define `` macro behaves much the same way as in C/C++. When a symbol is defined, the code in the `` `ifdef `` block will be compiled. If the line with the `` `define `` is commented out, the `` `else `` block is compiled. This is useful when you want to disable structures to save compile time, but they are connected to top-level outputs. Supplying the default assignments will prevent the compiler from erroring on undriven outputs. We suggest adding your hardware test modules to `FPGA_TOP_ML505.v` in this manner so they are easily removed/substituted for other modules as the project progresses.

### 4.2 DVI Interface

We have adopted the physical DVI interface from previous years of CS150. The `DVI` module handles communicating with the Chrontel chip and initializing its control registers via an I2C bus to set the desired color format and update rate.

The driver is currently configured to 800 x 600 x 72Hz x 3 bytes representing red, green, and blue pixel values on the `video` input. You should not need to worry about changing this initialization, since grayscale can be emulated by putting the same value on each color channel. The chip assumes that every valid value clocked in to the chip corresponds to an output pixel. A valid pixel is determined via the standard synchronous ready/valid interface discussed in discussion and class.

The internal counters will automatically reset to the top-right corner of the screen when 480,000 pixels are read, so you need only supply a constant stream of pixels to keep the DVI driver happy.

### 4.3 Pattern Generator

You will produce a test pattern generator that verifies proper writing of the video output. This pattern will consist of a repeated pattern of colored rectangles with different width and height that alternates the color palette every second. To verify proper axis output, we require that the pattern have wider than tall rectangles, and the color palette be distinguishable between the two phases.

Additionally, the colors should not change in the middle of a frame. An example pattern is shown in Fig. 1, but the actual dimensions and color are up to your artistic license. Finally, you will need to worry about proper treatment of the `video_ready` and `video_valid` signals.

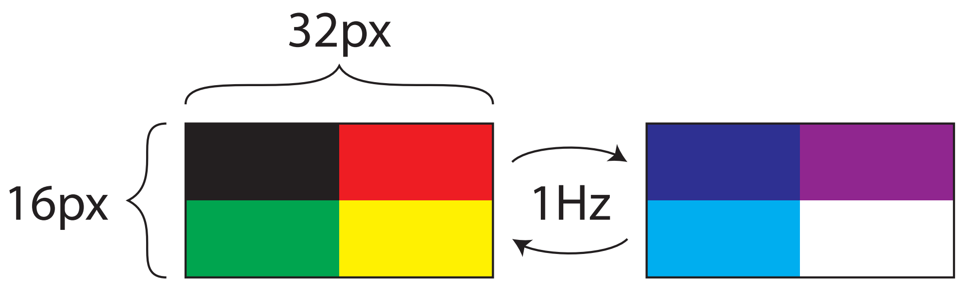

Figure 1: Example color test pattern.

### 4.4 Simulation

To exercise the Pattern Generator, and practice modular coding practices, you will write a testbench for the Pattern Generator. This testbench should either mimic the timing of the Chrontel chip, or alternatively can randomly stall the Pattern Generator by de-asserting the `video_ready` signal. It must verify that the correct pattern of pixels is received. To show this is working, include a (normally commented) line of code in the Pattern Generator that injects an error in the pixel stream. For checkoff we will want to see this in a simulation trace output or ModelSim waveform.

One partner should have the first add and commit of PatternGenerator.v, and the other should do this for PatternGeneratorTest.v. Try agreeing on the pattern specification beforehand, and then writing the generator and testbench separately before synchronizing the repos and testing.

The /hardware/sim directory structure has been slightly modified from that in lab3. To play nicer with version control, we have moved the test.do and test.input files to /hardware/sim/test/src and /hardware/sim/tests/data respectively. To run a test file, you need to create a symbolic link from the tests directory to the .do file in the src directory:

```
cd /hardware/sim/tests
ln -s src/test.do test.do
```

The symbolic links are ignored by version control with a .gitignore file in the tests directory. This way the source and test vectors for all of your tests can be committed when what they execute changes, but changing which tests are run does not appear as a change to your repository. This will be useful when team members are testing different portions of the design, and do not want to overwrite the selection of tests that their partner is running.

### 4.5 Checkoff Goals

By the week that this checkpoint is due, your team should be able to show:

- At least one commit to the repository from each team member.
- PatternGenerator.v with error injection
- PatternGeneratorTest.v with random or periodic assertion of video_ready
- A ModelSim waveform showing correct behavior of Ready/Valid signals.
- The test pattern on the external monitor connected to the FPGA board.