BaseJump STL Contributor's Guide (2021)

Thank you for stepping forward to help contribute to BaseJump STL! Together, we will transform how hardware is designed!

We have had both local contributors at UW / UCSD and also remote student contributors from institutions across the world. Beyond learning how to design hardware well, and helping to transform the way that the world design hardware, we also want to help our contributors out. In many cases, we give follow-on recommendations to both companies and graduate school. In fact, we have brought on several contributors into master's and PhD programs over the year. Additionally, your code will get taped out and used in our chip prototypes, in the latest process generations such as TSMC 16.

Generally, for remote interns, we require a 40 hours commitment for at least 12 weeks. In several cases, students have continued into the school year with a more advanced project that leverages their work on BaseJump STL because they wanted to see their stuff in silicon.

Here is more info:

General

- You will work on our project that involves building open source hardware in SystemVerilog / Verilog.
- All of your code must be made available for others to use, free of use, via an open source license of Prof. Michael Taylor's choice, in this case Apache 2.0.
- You must give detailed weekly reports, as detailed below.
- You must work autonomously and without reminder from us.
- If you do a great job and complete your tasks, Prof Taylor and/or team will be happy to write you a recommendation letter for grad school or employer.

Resources

Make sure to study these items before beginning. This will give you more information on the high level goals of the project, and for the code base.

1. The BaseJump STL DAC Paper.
2. The BaseJump STL DAC Slides.
3. The BSG SystemVerilog Style Guide.
4. https://github.com/bespoke-silicon-group/basejump_stl

5. bjump.org
6. If you are new to git, see https://try.github.io/ for a tutorial.

Weekly Reports

You should give a detailed report emailed to professor taylor, containing:

1. What you planned to do for this week (and why)
2. What you actually did, and what challenges you overcame, how you overcame them, and what insights you got.
3. What you plan to do for next time (and why)

Having this text will improve your technical writing, improve communication with us, and provide defacto documentation of your progress on the project.

Getting Help

For your project, in order of preference
1) try things out yourself,
2) look up on internet,
3) the basejump-stl@googlegroups.com mailing list
4) prof.taylor@gmail.com, via email (or the graduate student if one was assigned to you)

The Project

The goal of the project is to build an open-source equivalent of the Standard Template Library (in C++) for hardware design. This way, instead of rewriting and redebugging the same code over and over, we can reuse a bunch of well-designed hardware components. Because of this, good, clean style is important. You **should focus on mimicking the style currently in use (naming, spacing, etc)**, and should try never to write the same code twice -- factor similar code into modules with clean, easy-to-explain interfaces and reuse them. You must follow these coding guidelines for naming and style.

Remember that when you do a pull request to the github repo, you are presenting your work to others, and if it is unnecessarily sloppy, then they will be annoyed with you. Be sure to make it your best work before you do a pull request.

Your initial project will be focused on improving the quality of the existing modules. After you have grasped our coding style, testing methodology, and understand what modules we provide, and contributed to our testing, then we will continue onto a project that makes use of the

BaseJump STL (aka bsg_ip_cores) library. For example one of our previous students (Bandhav) did this, and ended up designing a manycore processor.

The basic structure of the repository is that different modules that people may want to use are organized by type into directory. In the top-level "testing" directory, we plan to mirror the directory structure of the top-level of the repo, so that ideally each file has a test.

So for instance, the directory `basejump_stl/testing/bsg_misc/bsg_popcount` is intended to test the file `basejump_stl/bsg_misc/bsg_popcount.v`. You can look at that test as an example.

For writing tests, we have a lot of helpful modules that already exist in the BaseJump STL source base, that you should use. This way we can use our own philosophy of reuse even in testing! As you develop tests, you will come up with your own ideas about what helper modules we should use, and we can add those to the repo as well. After you have written tests and have a sense of our hardware design philosophy, then you will be contributing synthesizable code to the repository as well. **Some of your code is likely to end up in a chip that we design in the near future!**

To run your tests, we recommend that you use Verilator. This will be scripted with a Makefile (see the example in the project above), in order to try out different variations in parameters. All of your code must be synthesizable for ASIC, unless the code is specifically for a testbench.

Another option, but less recommended, is Vivado (from Xilinx). They only charge money for device support (ie. you want to synthesize your design for a specific FPGA) but we really just want the RTL simulator and waveform viewer so there is no need to target a specific FPGA. You can check it out here https://www.xilinx.com/support/download.html. I think you might need to create a Xilinx account but that is free. Select the Web Pack License. You must still run everything via makefile and command line; GUIs are the antithesis of automation.

Using GNU Make

In our testing infrastructure, we make use of GNU make (http://www.gnu.org/software/make/). You will need to decode some of our makefiles so here is some useful information.

There are manuals there. Make is a declarative language -- you tell it what file you want to build, and then it combines rules in the makefile to generate it. There is a link to a webpage that has the entire manual on one webpage (http://www.gnu.org/software/make/manual/make.html), which is good for searching through to look up specific features. You can read the manual to learn about make basics, like how to write rules and build things. And then look up some of the advanced features that I used by searching through the manual.

Test Design

Each module in BaseJump STL has input and output signals, as well as input parameters. We need to make sure that our tests cover both the important variations in wired inputs and outputs, as well as variations in parameters. The popcount test I have linked to above gives an example of testing both of these.

Ensure both module and testing code are environment agnostic. As BaseJump STL is designed to be imported by other projects, we strive to avoid namespace pollution, committing local paths, or depending on any non-standard environment variables.

Each test should have comments that explain how the test works. Let's have a section in the comment blocks, called "TEST RATIONALE" that discusses what cases the test should cover, in English. Here is an *example* rationale:

// TEST RATIONALE

// 1. STATE SPACE

The output of the function is undefined if there is more than a single 1 bit set in the function, so that fortunately limits the state space that needs to be tested, and means that we can exhaustively test the function, simply by testing the following inputs: all zeros, and the value 1 shifted from 0 to width_p bits. Clearly, we should test both output values.

// 2. PARAMETERIZATION

The parameter width_p determines the behavior of the function in a significant way, because it is written as a divide-and-conquer recursive algorithm. Significantly, in the cost, the case (width_p=1) is a special case, and power of two widths and non-power of two-widths are handled with different clauses. So a minimal set of tests might be width_p=1, width_p=4, and width_p=5,6,7. However, since there are relatively few cases, an alternative approach is to test width_p=1..512, which gives us brute force assurance.

B. For the parameterization, it is somewhat tricky because we want to run these tests in an automatic fashion. In academia, and in industry, people very widely use gmake. We will make use of it. All of your work should be scripted using Makefiles -- no manual typing (except "make") should be required, so that we can repeatedly re-rerun things.

Test Coverage

Moving forward, we expect that new modules that are committed should also have tests that achieve near 100% line and toggle coverage. Moreover, they should pass the BaseJump STL synthesis tests. See our BSG Coverage Based Methodology document on the right way to do verification:

Merging and naming of modules with like behavior

- for debugging, we want to maintain the invariant that every input, and every output is explicitly listed when people instantiate a module, to prevent bugs.
- for code clarity, we don't want people to have specify null input values to ports if they are not using them (e.g. if they want a d-flip flop, don't make them put a .reset(1'b0),.en(1'b1)).
- we want basejump to provide portable interfaces so that people don't have to change their verilog code to move it from one foundry to the next -- they should just have to modify the /hard library
- if a module generally corresponds to hardened macros, then we want to make sure to have interfaces that will map efficiently onto those structures across the dominant IP vendors (e.g., TSMC, ARM, Xilinx, Altera)
- we want the module names to be different if there are significant difference in cost for implementing each variant. So for example, the write_bit versions of RAMs are not supported by Xilinx FPGAs, so we don't want to make the write bit versions the single unified interface to a ram -- because people may use them casually and not realize that they will have a horrendous cost when mapped to FPGA.
- we want a module name to correspond to a unique idea
  - if there are some minor variants that have little ramification for area/performance/energy, and for which there are no established terminology, then it is preferable to merge them and have a parameter select between the modes (ideally, the obvious default mode, and the slightly less obvious non-default mode.

How to Ramp Up

The intent here is to model how you should approach collaboration in any open source project.

1. Read all of the items labeled resources -- the BaseJump STL DAC paper, slides, and the BSG SystemVerilog style guide.
2. Start looking at the BaseJump STL modules on github. Familiarize yourself with what is there and what might be missing, and how it is organized.
3. Start with this modest task that will help you get experience with the code base and our style guide. Look through the code and find violations of the style guide, and try to do a

few pull requests to clean up any issues you identify. Make sure the requests are minimal and do not have unnecessary reformatting of white space.

4. Also look through the testing directory and try to identify a few modules or functionality that have not been tested. Check to see if issues have been filed for that module, and if not, file an issue, referencing the file that was not testing. File an issue for each untested thing.
5. Look through the issues that have already been filed for BaseJump STL, and see if you can do pull requests that fix some of the easy ones. If you have a direct question about the issue, you can post it on the issue itself in github. If you are asking more for clarifications about coding style or contributing, then use the basejump-stl@googlegroups.com mailing list.
6. Write a test for one of the modules that has not been tested. Look at the other code to see how tests are setup. Achieve 100% toggle and line coverage.
7. Once you have built up some credibility by fixing some issues on the existing code, now it is time to create some new modules. To minimize wasted time, clear the idea ahead of time by either asking for suggestions on the mailing list (if you want, you can say what kind of module you are interested in) and getting a consensus, or if you have a concrete idea, write an email to the mailing list with the title RFP: module_name (e.g. RFP: bsg_decoder_thermometer) and a body that describes the module and the interface (port names and types, and parameter names) you believe it should have. Then folks can tell you if it would be useful or is redundant before you charge off and do it. Or they can tell you that somebody is already working on it and help you avoid duplication of effort. If you have other proposed changes that are not just new modules, you can also use RFP: <change>. Simply bug or issue fixes do not require unless there is some ambiguity.

Thanks in advance for your great contributions!