ASIC-Strength Verification in a Fast-Moving FPGA World

Bryan Murdock
Fusion-io
February 7, 2013

***Abstract*—FPGA design and verification have some key differences from the typical ASIC project. FPGA designs are fast-moving, ever changing, and can have very short release cycles. Techniques that help keep code quality high and verification effective are explained and discussed including: General Automation, the benefits of the Python programming language for use in automation, Continuous Integration, Distributed Version Control, Strategies for organizing, building, and testing a highly configurable code base, Web-based Code Reviews, and Register Generation.**

I. Introduction: FPGA vs. ASIC

Over the last few years Fusion-io's small, fast-moving FPGA team has moved away from the typical ad-hoc style of FPGA testing and towards full ASIC-strength verification. Here are some of the characteristics of the fast moving FPGA world that make traditional ASIC-style verification challenging:

A. Short release cycles

New features are defined, implemented, and released to customers all in a matter of a few months. Verification has to be able to quickly go from feature-definition to feature-fully-verified during that same short time period. Long drawn-out verification planning, verification environment bring-up and verification environment debug for each new feature is not an option. The verification team has to be nimble and quick to react.

B. Features are mixed and matched

Various sets of features are combined together into various FPGA builds for various products that are all released at about the same time. In other words, there isn't a single ASIC to focus verification efforts on. Instead there are many FPGA builds to verify, all concurrently. The design is configurable and our verification code-base has to be flexible and configurable as well.

C. Very small team

Though it has grown rapidly, the team started out microscopically small compared to other well-known ASIC shops. There isn't an endless supply of people to get all this done so we have to find ways to multiply our efforts.

Solid design verification is critical, and to accomplish that the realities of the fast-paced FPGA world need to be addressed head on. The approaches and tools discussed in this paper helped our small team to rapidly increase the quality of our design verification while meeting the demands of our business. They should be helpful to any FPGA team looking to better their verification, as well as to ASIC teams looking to become more agile and adaptable.

II. General Automation

The first thing we did at Fusion-io to improve our verification was to automate critical processes and make them easily repeatable. Usually at the start of an FPGA design, especially if the design targets a new FPGA or requires a new version of the tools, engineers spend a lot of time clicking around the GUI. That's great for exploring and learning the capabilities of the tools, but once we settle into a routine and start doing the same sequence of clicks over and over to get things done, we recognize we are well into the time when we should be automating those actions with scripts. We live by the adage that anything in the design and verification process that you do over and over can and should be automated. Automation reduces the chance for human error, makes critical processes easily repeatable, and saves a lot of time over the long run.

Automation should go all the way, don't automate just part of the process and leave manual steps for humans to carry out. At a bare minimum you should have three scripts that allow you to do each of these by issuing a single simple command:

- run a test and get a pass/fail report at the end
- run a suite of tests and get a pass/fail report at the end for each test
- produce a bitstream for your target FPGA

If you have those three things automated then you are well on your way towards ASIC-strength verification. All of those operations should be easy to do over and over. Given the same version of code and command-line arguments, they should produce the same results every time, for anybody that runs them. If Alice runs the foo test with seed 1234 on revision 5678 of the code and it fails, she should be able to give that information to Bob and it should be no problem for him to re-run the test on his workstation and get the exact same result. This helped us a lot as our team grew rapidly. The processes to get basic stuff done that new team members had to learn were simple.

Once we had that bare minimum automated we started looking for more to automate[1]. Some of the questions you

[1] See [1] for a lot of insights into what you should shoot for in your automation. It's software centric, but there are some good universal principles in there.

can ask yourself to help with this are, once you produce a bitstream, do you load it onto a board and test it? Can the steps to do that be automated? When you run a suite of simulations do you manually open log files and classify test failures into groups? Could that be automated? When you run those test suites, do you collect code coverage? Do you merge coverage from multiple test runs together? Can all that be automated? What about debugging tasks? Are there repetitive steps that can be automated? When you build bitstreams or run simulations, do you ever look at how long each step takes or how much memory it used? Can a report of that be automatically generated?

It takes time to write scripts for all of that, but it also takes time to manually do everything. The time it takes to manually click GUI's or generate reports is time that could be spent writing better tests and finding more bugs. Automating critical processes is the first step we took to improving our design verification, and it's hard to envision working any other way now.

III. PYTHON

Once we decided to automate repetitive tasks our team needed to choose a scripting language. As we thought it through we considered past experience with other languages (Perl and shell-scripts, mainly) and the characteristics of the scripts we knew we'd be writing. Considering what slowed us down in the past we realized some things. We don't work on automation scripts every day. The code doesn't stay in active memory in our brains. We'd often spend a day or two updating a particular script and then not look at it again for weeks. Our brains would page that script out of active memory and page in our testbench or design code in its place. The second characteristic of these scripts is that they are never done. As with all software, we would need to continually make little changes and updates to our scripts here and there, forever. We would need to be able to quickly read and re-understand the code for these scripts and make changes to them, over and over. Of course any scripting language would also need to be high-level and ideally have a multitude of libraries so we don't have to re-invent solutions for common problems. As we considered these requirements, spoke with the experienced software developers at our company, and looked at various scripting language options we realized that the features of Python met all these challenges well.

Python is the most readable scripting language out there. It reads like the pseudo-code examples you see in textbooks. Unlike Perl or Bash, it eschews symbols and uses plain English words wherever possible[2]. English stays in your working memory, abbreviations using $%&*(#@) do not.

Here are just some basic examples of Python to prove my point, there are many more examples online:

```
# get the length of a list, and a string
some_list = ['a', 'b', 'c']
list_length = len(some_list)

some_string = 'foobarbaz'
string_length = len(some_string)
```

[2] Yet Python only has about 30 reserved words, unlike, say, SystemVerilog's 220 or so.

```
# define and call a function:
def print_list(list_input):
    for l in list_input:
        print l

# it can handle lists of varying types:
print_list(['one', 'two', 'three'])
print_list([1, 1, 2, 3, 5])
```

The other advantage of Python is that it is natively object oriented. Many of the same coding patterns that we use in verification languages like SystemVerilog and C++ work just as well (or better) in Python. This leads to less cognitive load when switching between writing your testbenches and writing your scripts. One way to illustrate this is with a Function Object Design Pattern example, also known as the Functor Design Pattern [2]. Just the name scares some people, but if you are familiar with the Universal Verification Methodology (UVM), a uvm_sequence is a functor [3]. Here is code for a Functor that produces incrementing integers, starting with the integer you seed it with. First in SystemVerilog:

```
class count_from;
   int count;

   function new(int n);
      count = n;
   endfunction

   function int body();
      return count++;
   endfunction
endclass

module top;
   initial begin
      count_from cf1;
      count_from cf2;
      cf1 = new(5);
      cf2 = new(100);
      repeat(5) begin
         $display("cf 1: %0d", cf1.body());
         $display("cf 2: %0d", cf2.body());
      end
   end
endmodule
```

And that gives you this output:

```
cf 1: 5
cf 2: 100
cf 1: 6
cf 2: 101
cf 1: 7
cf 2: 102
cf 1: 8
cf 2: 103
cf 1: 9
cf 2: 104
```

SystemVerilog has no syntax to make a class object callable like a function, so I used the same convention that the uvm_sequence does and just gave it a function named body.

Here is a Python example that gives the same output:

```
class CountFrom:
    # constructor:
    def __init__(self, count):
        self.count = count

    # makes this class callable like
    # a function:
    def __call__(self):
        current_count = self.count
        self.count += 1
        return current_count

cf1 = CountFrom(5)
cf2 = CountFrom(100)
for i in range(5):
    print "cf 1: {0}".format(cf1())
    print "cf 2: {0}".format(cf2())
```

Pretty similar to the SystemVerilog, but like a good scripting language it's 10 lines shorter with no need to declare variables ahead of time. Now, this might seem like an obscure example, but it shows that even doing this somewhat rare object-oriented thing can translate well between your verification language and your scripting language. We have found that we really use many of the same design skills in both languages and that reinforces those skills in both languages. We have gained insight while solving problems with Python that have helped us to better solve problems with SystemVerilog or C++, and our Python scripts have benefited from our various team members' SystemVerilog and C++ experience as well.

The core language of Python is great, but what about libraries? One motto of Python is, "batteries included." The standard library that comes with your Python installation has already solved many of the problems we have encountered while writing scripts. There are modules for parsing command-line arguments and options (optparse, or argparse in newer version of Python), interacting with the shell (system, os, shutil, subprocess), for handling temporary files (tempfile), for logging stuff while your script runs (logging), regular expressions (re), sending mail (smtplib), interfacing a database (psycopg2), and so on. Finally, outside of the standard library there are libraries and frameworks that makes it easy to write simple web applications, which we have used with great success in order to generate and publish reports (django, flask, or bottle). More information about all of these libraries can by found with your favorite Internet search engine.

Using Python has made our scripts very readable and easily maintainable. A wide number of people have been able to understand the code in our scripts and to continually edit, upgrade, and add features to them as needed.

IV. Continuous Integration

With new features being added to design and verification code at a rapid pace, many different configurations to test, and a release looming just around the corner (especially if there is **always** a release just around the corner) how could we make sure the code base stayed stable? Continuous integration is the answer at Fusion-io. Continuous Integration (often shortened to, CI) is a term software engineers use to describe systems that monitor your code repository and automatically run a suite of tests whenever new changes are added. If it sounds simple that's because it is, but it has been very effective at keeping our code base free of bugs. Once we automated the running of tests and builds as described above, it was trivial to have a CI system run those for us.

Fusion-io's software department already had a CI system up and running when us hardware people were ready to start doing our own automated testing and builds. It was easy for them to allocate a part of the system for us to use. That is probably the case at many companies and it gives you an instant benefit when you bring it up. I won't say too much more on CI because there is a great paper on the subject from last year titled, *A 30 Minute Project Makeover Using Continuous Integration*, that gives more details than a single section of this paper can [4]. Continuous Integration is a simple way to catch bugs early and keep them out of your design.

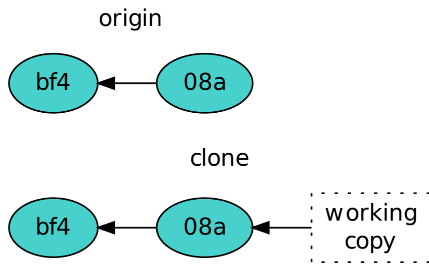
V. Distributed Version Control

One challenge we had at Fusion-io as the team grew was that we started to have many lines of development going on at once including maintenance of old FPGA bitstream releases, multiple new features for future releases, and multiple releases in progress simultaneously. Just having more engineers working on the same codebase creates more possibility for conflicts and integration pains. While developing complicated features engineers like being able to check in their incremental work and not wait until everything is done and tested, but checking unfinished work into mainline causes problems for others on the team. Branches help with that, of course, but existing free version control tools like CVS and Subversion just didn't support branching and merging smoothly and seamlessly. These issues were starting to slow us down and cause us to spend time solving problems with communication and coordinating work. Solving those problems are not where we wanted to be spending our time on a day-to-day basis.

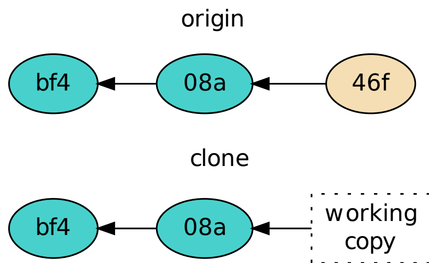
Better branch and merge support seemed to be issue number one. Expensive commercial tools exist that support those operations more robustly than CVS and Subversion, however at the time there were some new free and open-source tools that also reportedly handled branching and merging very well. These tools take a different approach from the likes of CVS, Subversion, Clearcase, Perforce, etc., which all use a centralized source code repository and server. The new tools use a distributed model and are referred to collectively as Distributed Version Control Systems (DVCS's). Git and Mercurial are the two most well known of these newer open-source DVCS's. As we investigated DVCS's we learned that they have other benefits that arise from their distributed nature. The licensing cost of these tools (there is none) is great too.

To understand the benefits of a DVCS and use it successfully we have found that it helps to know some details of how it works. With distributed version control every developer gets a copy of the whole repository, called a clone. There is no centralized repository required, but our team designates one clone of the repository to be the central one used for collaboration. When you make code changes you commit them to the repository just like with centralized systems, but the commit is local and only exists in your clone of the repository. That means you don't need a network connection at all to do basic version control operations like committing code, viewing the log, and diffing your files against the history. To share

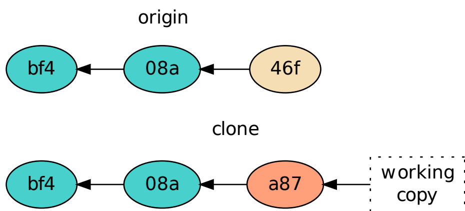
changes with others you “push” or “pull” them from one repository clone to another. At this point, some diagrams might help. To start, you’ll see the designated central repository and your clone with your working copy. The version numbers are a cryptographic hash to make sure every version is unique amongst all clones in the universe (abbreviated to a 3-digit hex number for brevity). More recent versions refer (or point) back to their parent (older) versions. Here’s how it looks:



With a busy team, while you are working on your changes usually someone else will commit and push their own changes to the original repository, creating a new version there:

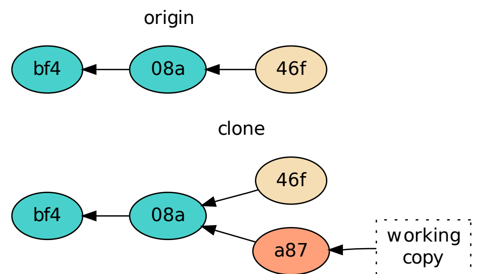


When you are satisfied with your own changes you commit them to your repository creating a new version there:

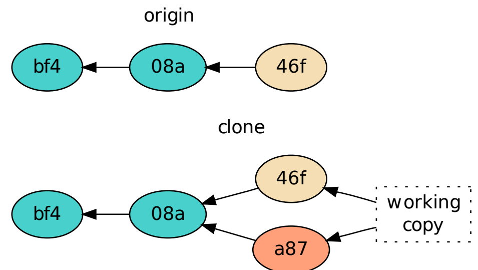


Note that at this point your changes are a commit in your repository but nobody else has a copy of that commit. You are free to delete or edit that commit before making it public. Our team has found this to be infinitely useful. You can easily try out new changes or experiments and take full advantage of version control to checkpoint your work and go back to previous checkpoints while testing and experimenting. You can then edit, re-arrange, or combine those checkpoint commits however you like before sharing them with the team (if you decide to share them at all). It’s very liberating.

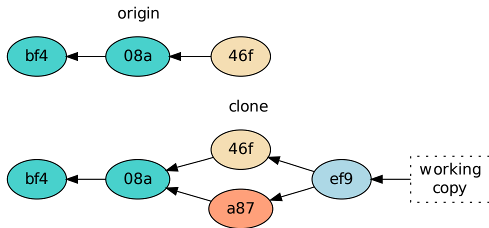
Continuing with our example, once you are happy with your changes you will want to share them with the team. Before you push them to the original repository, you should merge your co-workers changes with yours. Do a pull (or fetch, in some DVCS’s) to bring the change from the original repository into yours:



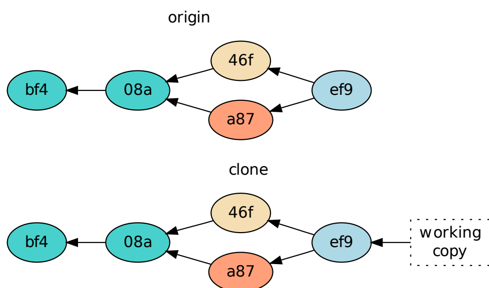
Note that your original changes (version a87) are untouched. Now you can merge your changes together with your co-worker’s in your working copy:



If things don’t look good you can abort the merge and your working copy will go back to being based on version a87. If it does look good you can commit the merge:



Now you can push your commits (a87 and ef9) back to the original repository where your teammates can pull them. Here's how it looks after the push:



One big benefit to this workflow is that you never have to merge other engineers' changes with yours before you commit. That is different from centralized tools that often require you to merge others changes into your working copy before you can commit it. They don't provide a way for you to checkpoint your work before doing that merge.

Note also that the branches in the repository history that are shown in the examples above can be given descriptive names and can be used for much more than one changeset. We regularly use new feature development branches, release branches, and bugfix branches to do our development.

As mentioned above, the two most popular and high-quality open source DVCS's are Git and Mercurial[3]. We chose Mercurial because we found it to be much more user friendly than Git, and because it and its plugins are written in Python, which we are already familiar with (not that we have needed to edit the source code, but reading it has provided some insights). Git runs slightly faster than Mercurial, but not enough to outweigh the advantages we saw with Mercurial. Both Git and Mercurial borrow ideas heavily from each other and they are looking more and more alike as the two projects progress. Both are well documented, have lots of enthusiastic users, and have free code hosting and bug-tracking available online for small teams [5], [6]. There are free tools to easily convert your whole repository history from just about any version control system to either Git or Mercurial. It's hard to go wrong with either one.

[3]Others DVCS's include bzr, fossil, and veracity.

The ability to make and edit local commits and the excellent branching and merging that these tools provide makes sharing changes among many engineers much more safe and easy. Using a DVCS, our team spends less time worrying about revision control and collaboration issues and more time developing and verifying great hardware.

VI. A Configuration-Driven Build System

When designing and developing for FPGA targets it became advantageous for our team to have one code base from which we could build many unique bitstreams. Reasons for this included the desire to target more than one FPGA part, vendor, and speed-grade, and the desire to create builds with different sets of features[4]. When the amount of options that had to be selected to produce the various bitstreams grew large, it became quite a challenge to manage. Maintaining configurations wasn't too difficult at first. We've all heard of `` `ifdef ``, and the UVM has its configuration database for configurable testbenches, but when using those techniques alone things quickly got out of hand.

You might, as we did at Fusion-io, see this evolve slowly. Maybe first the desire for two different clock speeds comes up so that you can use the low clock speed to get a bitstream produced more quickly for functional testing. You add a simple `` `ifdef `` to the code and a command-line switch to the build script and you are good to go. Then someone points out that you should be simulating the two clock speeds as well, so you add the same command-line switch to your simulation script. Next someone wants a build that supports software ecc instead of a hardware corrector. You add another `` `ifdef `` to your code and another switch to your build and simulation scripts. Then a request for shallower on-chip buffers comes in. Another `` `ifdef `` and another switch are added. If you are careful the default simulation and build script invocations don't change, and still look simple like this:

```
$ sim-script testname
$ build-script
```

Maybe those invocation still produce the high-speed, hardware ecc, deep buffer build since it was the original configuration, but suddenly you have a lot more possible command-line invocations:

```
$ build-script --slow-clock
$ build-script --soft-ecc
$ build-script --small-buffers --slow-clock
$ build-script --soft-ecc --small-buffers
$ # and so on
```

Not to mention the similar simulation script invocations as well. Things are getting out of hand with only three options, and our codebase quickly accumulated many more than three. With all those possible combinations of options a question quickly arose for us in verification, are those all valid combinations that need to be supported and tested? Maybe no product will ever ship with shallow buffers and software ecc. Where is that recorded so that all parties that need it, including your scripts, have access to that information?

[4]This can also happen with ASIC projects where you have the ASIC build target and an FPGA emulation build target.

We found that what really needed to happen was to somehow encapsulate all these options into some sort of collection that has a name. We call these collections of options, Builds. An easy way to implement Builds is with simple verilog files in a special directory of your codebase. For example, here are three Builds (three files) for three of the possible configurations mentioned above (with a configurable bus-width thrown in too, for good measure):

```
// build: functional
`define FUNCTIONAL_BUILD 1
`define CLOCK_FREQUENCEY_MHz 100
`define BUS_WIDTH 32
`define BUFFER_DEPTH 16
// hardware ecc
```

```
// build: small
`define SMALL_BUILD 1
`define CLOCK_FREQUENCEY_MHz 700
`define BUS_WIDTH 16
`define BUFFER_DEPTH 4
`define SOFT_ECC 1
```

```
// build: full
`define FULL_BUILD
`define CLOCK_FREQUENCEY_MHz 700
`define BUS_WIDTH 32
`define BUFFER_DEPTH 16
// hardware ecc
```

Notice that we didn't just have true/false for each of those (or defined and not defined), we used actual numbers where the design could be parameterized. That would open up all kinds of unconstrained build combinations if you just passed a clock frequency (for example) on the command-line of your script, but we have encapsulated the clock frequencies, bus widths, etc. that we plan to test and ship in these Build definitions. With Builds defined and in place, your build and sim script invocations become:

```
$ sim-script functional testname
$ build-script functional
```

or:

```
$ sim-script small testname
$ build-script small
```

or:

```
$ sim-script full testname
$ build-script full
```

Simple and clear. The scripts can do the magic of supplying the Build file specified on the command-line to the simulation and build tools.

The Builds don't have to be simple verilog files. The number of build options we were dealing with got so large that we actually wrote a fairly sophisticated web-based tool with a SQL database back-end to manage them all. How you structure it depends on how many Builds you are managing, whether you want Builds to be able to inherit from each other, whether Builds should support multiple names/aliases per Build, collaboration needs with other teams in the company, and so forth. However you do it, once it's set up and working you should have one place to specify valid Builds and your script invocations will become much simpler.

Now that specifying the possible configurations has been cleaned up it's time to look at the design code. If you proliferate `` `ifdef `` constructs or generate blocks in your design code to implement the various options the code will get very messy and hard to read. Fortunately, there are other ways to make design code configurable.

The two best ways to make design code configurable while preserving readability is to use parameters and modules. Parameters are nothing new and I won't go into detail here. They are what you would use for something like the buffer size and bus width options mentioned above. The use of modules could apply to the ecc example above. To switch between ecc features you create two different modules with the same name, each in their own file. Maybe you call the module, ecc. If you follow the common rules of One Module Per File and Give The File The Same Name As The Module then you'll need to keep the two `ecc.v` files in two different directories. Now each Build consists of one of the above verilog files (with the SOFT_ECC define no longer necessary) plus a file list. The file lists would look something like this:

```
// file list for build: functional
top.v
fifo.v
bus.v
ecc/ecc.v
```

```
// file list for build: small
top.v
fifo.v
bus.v
software_ecc/ecc.v
```

```
// file list for build: full
top.v
fifo.v
bus.v
ecc/ecc.v
```

Again, your simulation and build scripts supply the correct file list according to the Build specified on the command-line. Now, in a higher-level module you simply instantiate the ecc module, with no `` `ifdef `` needed. This keeps the code very clean and readable. With parameters, modules, and careful creation of the file list your code can be very configurable and largely `` `ifdef `` free.

When you do find that you need to use `` `ifdefs `` or generate blocks to configure your design code it is important to use good naming for the `` `define ``'s or parameters that drive these. Once you have Builds with names for your collection of features it will become very tempting to use Build names for `` `define ``'s. This happened to us when a new Build was defined that used a new feature for the first time. Let's say the Build was called Napoleon. If the new feature that Napoleon uses is software ecc, you might be tempted to write code like this:

```
module top;
   input pin1;
`ifdef NAPOLEON_BUILD
   input special_soft_ecc_pin;
`endif
   input pin2;
   output pin3;
   // etc.
endmodule
```

That works fine until some other Build is defined that also uses software ecc. What you really want is to make sure and use the feature name for your `` `define ``, so the above code looks like this instead:

```
module top;
   input pin1;
`ifdef SOFTWARE_ECC
   input special_soft_ecc_pin;
`endif
   input pin2;
   output pin3;
   // etc.
endmodule
```

Then both Builds can reference the SOFTWARE_ECC feature and the code will read more clearly.

We've seen that Builds can map collections of options to `` `define ``'s, parameters, and a file list. What do Builds mean for verification? Testbench code obviously needs to be informed of the same `` `define ``'s and parameters and the same file list. Even though a Build specifies the design code configuration, which is all compile-time stuff, a single Build might support various run-time options that also need to be tracked and tested. You probably want to keep Builds focused on compile-time stuff only (stuff that ultimately affects synthesis output). Run-time options that affect test stimulus, prediction, and responses need to be dealt with in addition to Builds in the world of verification. Going into more detail on that is beyond the scope of this paper, but be aware that similar methods of encapsulation can be used. Also keep in mind that verification code can use facilities for run-time configuration of the code[5] and clumsier compile-time constructs like `` `ifdef `` and parameters can often be avoided altogether.

It's easy to not think very hard about how to approach the task of making your code configurable because it is usually a need that grows little by little. Using the power of encapsulation helped our team deal with the complexities without falling into the pit of hard to maintain code and scripts with an endless number of command-line arguments. Encapsulating and simplifying build configurations allows us to easily manage multiple FPGA targets and use a single code base to develop and verify them all.

VII. WEB-BASED CODE REVIEWS

The best way to get bugs out of your design is to catch them before they ever get in. One of the best ways to do that is to make sure all code gets looked over by more than just the person that wrote it. Code reviews are a great idea in theory, but can seem difficult and impractical for a fast-moving FPGA team. At Fusion-io, once we had more than a couple people on the team we really wanted to do code reviews but found it hard to get two or more engineers in the same room together to go over source code. Individuals didn't want to interrupt their co-workers every time they had code to check-in. Even when we did schedule code review meetings, it was difficult in a meeting setting for reviewers to concentrate and understand the code they were being asked to review. Also, engineers had to manually take notes of what the reviewers suggested during the meeting, and then go back and implement any changes they had suggested. After those changes were made, then what? Should we schedule yet another review meeting to review the updates? Web-based code reviews alleviated all of those problems for us.

We use a tool named Review Board for web-based code reviews [7]. It is free and open source and runs on a web server inside your company. When you have code ready for review you create a review request on the Review Board server. To do this there is a simple command-line tool that automatically takes a diff from your working copy and uploads it to the Review Board server. You then go to the resulting web page for your review request and select who you would like to review your code (and you can set up groups to make this simpler). Review Board then sends each invitee a simple email with a link to the review request web page. Now they can all review the code on their own computer, at their own desk, at a time that is convenient for them.

Fig. 1. Review Request Example

On the review request web page each reviewer will see

[5] $test$plusargs and $value$plusargs which, incidentally, the UVM makes use of for things like UVM_TESTNAME and uvm_set_config_int

a nice side-by-side diff of the code changes. Figure 1 shows what this looks like. Reviewers can click on the line numbers in the middle of the diff and a text entry box will appear. They can type in comments they have about that line of code right there. When they are happy with the comments they have entered they click the Publish Review button and an email is sent to the review request submitter and to the other reviewers. Everyone can then reply to their comments right on the review request page (we have had many productive conversations on Review Board). If you make more changes to your code in response to the helpful review comments you can update the diff on the review request with the same command-line tool and the reviewers will again be notified. Once reviewers are satisfied with the code under review they can click on the "Ship It!" link to let you know. It always feels good to get a "Ship It!" email from Review Board.

Code reviews with an online code review tool have made a big difference for our team. Before we started using Review Board nobody was sure if they had time to stop work and review someone else's code. We are very interested in what our teammates are committing to our repository and now that it's convenient for us we spend quality time reviewing code changes. Code reviews have helped us catch bugs before they even made it to our main code repository.

VIII. Future Work: Registers

There are other areas of the design and verification process that our team is in the process of automating and improving. Control and Status Registers (CSRs, or just, registers) are the next time sink that we are tackling. Currently everything we do with registers is done manually by hand. We code all the verilog by hand, we choose addresses and make sure they don't conflict manually, decoding logic is written by hand, documentation is manually created and maintained, software header files, verification, you get the idea. Our talented engineers can handle all this work and we haven't had any major problems with registers, but it does take up time that could be better spent on other things.

We have been eyeing and evaluating software tools to automate all the chores related to registers for a while and we recently adopted one, CSRCompiler from Semifore, Inc. Now we can write a register specification in a formal language and from this spec, CSRCompiler will generate register verilog code, register decode verilog code, documentation in various formats, header files that define constants for register addresses and fields within registers in both the verilog and c languages, verification code (uvm_reg), and so forth. This will make creating, changing, and verifying registers much quicker and easier. Another benefit we are noticing is that it will make it easier for us to improve our registers (that's the changing part mentioned in the previous sentence). With everything automatically synchronized, using this tool will make it much easier to experiment with different hierarchies and groupings of registers, different addresses and address spaces, and even different names for registers and fields, all so we can find what works best. Instead of needing to manually edit everything from verilog to software header files in order to make a change you can simply make the change in the source specification and re-run CSRCompiler.

All of this automation reduces the chances for bugs to get into the FPGA design and into the software interacting with the FPGA. It also gives our verification efforts a huge boost. We will be able to spend a lot less time manually dealing with register and field addresses and more time creating better tests.

IX. Conclusion

Keeping up with a fast-moving, ever changing design can be hard for a verification team. Using the tools, processes, and techniques introduced in this paper my team has been able to increase the quality of our design verification and continue to react quickly and effectively to the needs of our most demanding, fast-paced customers, as well as win new high-value deals for us and our shareholders. These same tools, processes, and techniques will be a benefit to any team that is facing the same challenges.

Acknowledgment

I'd like to thank Eric Decker, who was definitely the biggest supporter of the movement towards ASIC-strength verification and is co-innovator in some key areas presented above as well.

About the Author

Bryan Murdock is a Senior Verification Engineer at Fusion-io. He has over 10 years of experience working on ASIC and FPGA verification and writing embedded software. Bryan has a Bachelor of Science in Computer Engineering from Brigham Young University. He blogs occasionally at http://bryan-murdock.blogspot.com, enjoys skiing on the Greatest Snow on Earth 20 minutes away from the Fusion-io Salt Lake City, UT office. He can be reached by email at bmurdock@fusionio.com.

References

[1] R. Hymas. (2013) Build and Release Management Series. [Online]. Available: http://thoughts.rockhymas.com/post/12014744973/build-and-release-management-series

[2] Wikipedia. (2012) Function object — wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Function_object&oldid=521332561

[3] M. Graphics. (2012) Tour/Sequences — Demo Tour of the UVM/OVM Cookbook. [Online]. Available: https://verificationacademy.com/cookbook/Tour/Sequences

[4] J. Gray and G. McGregor, "A 30 Minute Project Makeover Using Continuous Integration," in *DVCon 2012 Proceedings*, 2012.

[5] Atlassian. (2013) Bitbucket. [Online]. Available: https://bitbucket.org/

[6] GitHub Inc. (2013) Github. [Online]. Available: https://github.com/

[7] Beanbag, Inc. (2013) Review Board. [Online]. Available: http://www.reviewboard.org/