PROCEEDINGS OF SCIENCE

Continuous Integration of FPGA Designs for CMS

Robért Glein*
University of Colorado at Boulder, High-Energy Physics
E-mail: robert.glein@colorado.edu

Alexx Perloff
University of Colorado at Boulder, High-Energy Physics
E-mail: alexx.perloff@colorado.edu

Keith Ulmer
University of Colorado at Boulder, High-Energy Physics
E-mail: keith.ulmer@colorado.edu

Due to the high degree of flexibility when designing firmware for FPGAs, the build process and the designs themselves are vulnerable to errors. Continuous integration is a fast way to detect a majority of such errors. Additionally, simulations – using test methodologies for testbenches such as unit tests – and hardware tests can be automated. Continuous integration offers the benefits of reproducible results, reliable error detection, error tracing, avoiding human errors in the build process, and **minimizing the manual verification** of the firmware. Such an extensive and automated development procedure requires a slight increase in setup time and the need to use a comprehensive integration tool, such as the *GitLab's* CI/CD tools.

Topical Workshop on Electronics for Particle Physics TWEPP2019
2-6 September 2019
Santiago de Compostela - Spain

*Speaker.

PoS(TWEPP2019)028

1. Introduction

In the development of FPGA designs, comprehensive verification and testing is required. This is especially true for firmware developments with multiple contributors. The iterative firmware development process starts with the build environment and ends with the deployment of the firmware. Modular firmware designs are highly flexible, use various submodules and tools for simulation, synthesis (netlist translation), implementation (place and route), and test of FPGA designs. To manage this complexity, Continuous Integration (CI) offers an automated way to set up, build, verify, control, monitor, and deploy the FPGA implementation. CI is widely and successfully used in software development [1]. CI is based on uniform build, simulation, and test environments and ensures a correct build of the FPGA design. Additionally, it covers essential functionalities of the specified simulations and tests. Human errors in the build process are prevented, for example missing files, wrong tool versions, changed global constants, and wrong submodule commits. Another main benefit is that the manual verification of the firmware can be minimized, especially if different FPGAs and/or configurations are supported.

2. Continuous Integration Overview

The preconditions for CI of firmware are that the code is versioned with a system such as *git* and that the code can be built in an automated way. Figure 1 shows a schematic of the CI concept. First, we commit the code and related files manually, as is typical for a versioning system like *git*. Second, the CI system simulates, builds (synthesizes, implements, and generates the bit file), and optionally tests the hardware in an automatic manner following a setup script. In the system demonstrated here, a manual review of the CI results is used to ensure that all of the tests were successfully completed and to have a human-based check before release. After this review, the tool will automatically deploy the results. We recommend and will demonstrate the use of a Command Line Interface (CLI) build system. Although the system described here makes use of *GitLab's* CI tools, we will discuss alternative tools as well.

2.1 Command Line Interface Firmware Build Systems

Many CI tools exist that can handle a firmware development environment and also make use of a CLI. We investigated several such tools and list here that which we found could handle the *Xilinx Vivado* workflow: *Ruckus* [2], IP Bus Builder (IPBB) [3], HDL Make [4], and vendor tools with CLI support (e.g. Tool Command Language – TCL). All these tools provide abstraction of complex procedures to CLI commands using setup files. From this point on we will focus on development using the makefile-based CLI tool *Ruckus*, but the concept is similar for the other tools. Some *Ruckus* commands, including a short description of the purpose of the command, are:

- `make depend` — *Vivado* Project Creation (e.g. FPGA type)
- `make sources` — *Vivado* Source Setup (e.g. add *.vhd)
- `make xsim` — *Vivado* Simulation
- `make syn` — *Vivado* Synthesis
- `make bit` — *Vivado* Implementation and *Bitgen*
- `make interactive` — *Vivado* Interactive (TCL CLI)
- `make gui` — *Vivado* Graphical User Interface (GUI)

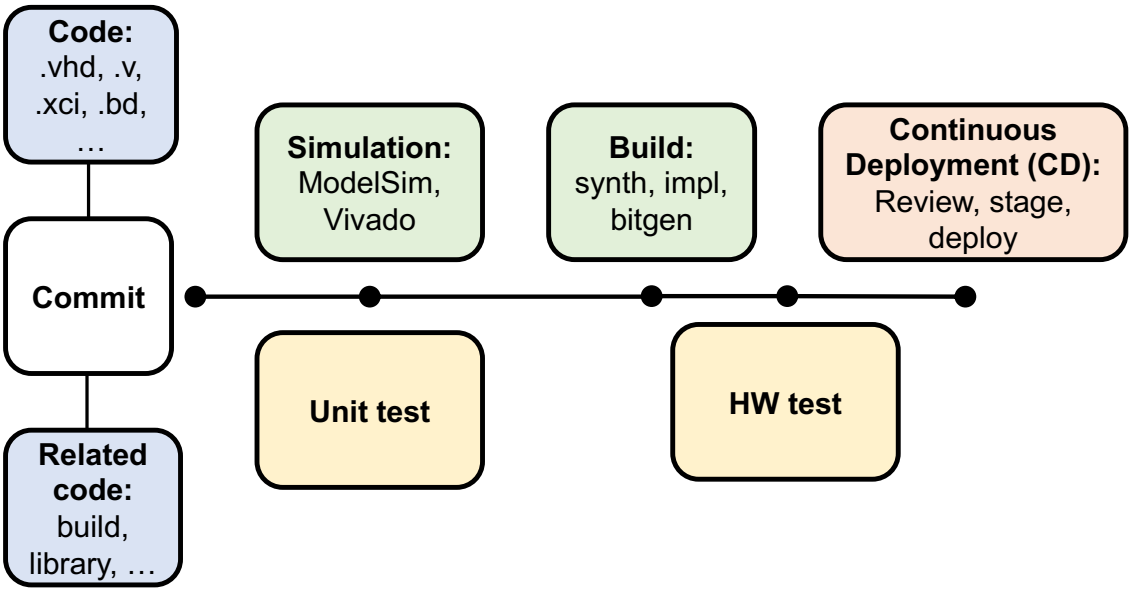

Figure 1: General overview of continuous integration for firmware.

These commands provide a good granularity – which will be later mapped to stages – to simplify the CI setup. While the *Ruckus* CLI tool was developed by a team at *Stanford University* [2], we have now extended the toolset to include the *Xilinx Vivado* simulation (`xsim`) workflow.

2.2 Continuous Integration Tools

In contrast to the just described CLI tools, this subsection discusses CI tools, which orchestrate the CLI tools. There are a few CI tools that can be used for firmware development, many of which are surveyed in [5]. We investigated *GitLab* Continuous Integration / Continuous Deployment (CI/CD) [6], *Jenkins* [7], *CircleCI* [8], and *Travis* [9]. Our CI tool requirements are: *git* compatibility, support of custom runners, support of the entire design flow (plan, set up, build, verify, control, monitor, and deploy), and configuration management. Because *GitLab* CI/CD is the tool that best fulfills our needs, we made the best experiences with, and is most the comprehensive [5], we have chosen to proceed using that tool. However, the workflow would be similar if using the other CI/CD programs.

GitLab CI/CD is configured using a `.gitlab-ci.yml` file, which is committed to the repository. Among other things, this file specifies variables, stages, job templates, and jobs, the last of which can call upon other scripts contained within the repository. The `.gitlab-ci.yml` file is executed by a *GitLab* runner (a workstation or a server) as a pipeline when new code is pushed to the *git* repository. The runner is an isolated (virtual) machine with installed software (e.g. *Xilinx Vivado*) that picks up jobs from the coordinator (*GitLab* server). It can be specified as a shared, group, or specific runner. For security purposes, the runner and coordinator are connected via a token, allowing only authenticated programs to be sent to the runner. The *gitlab-runner* executable is configurable at the user level for things such as number of parallel jobs, type of jobs, etc. The CI/CD service provided by *GitLab* is compatible with container based systems such as *Docker*, allowing for user customization of the build environment. One other feature worth mentioning is

that *GitLab* allows for the mirroring of external *git* repositories. This then allows the *GitLab* CI/CD service to be used while not relying on *GitLab* as the main repository hosting service.

3. Continuous Integration Example

In this section we present a firmware design workflow for the Global Track Trigger (GTT) FPGA board of the CMS level 1 trigger [10]. Figure 2 shows a specialization of Figure 1, including commands from the CLI build system *Ruckus* from Section 2.

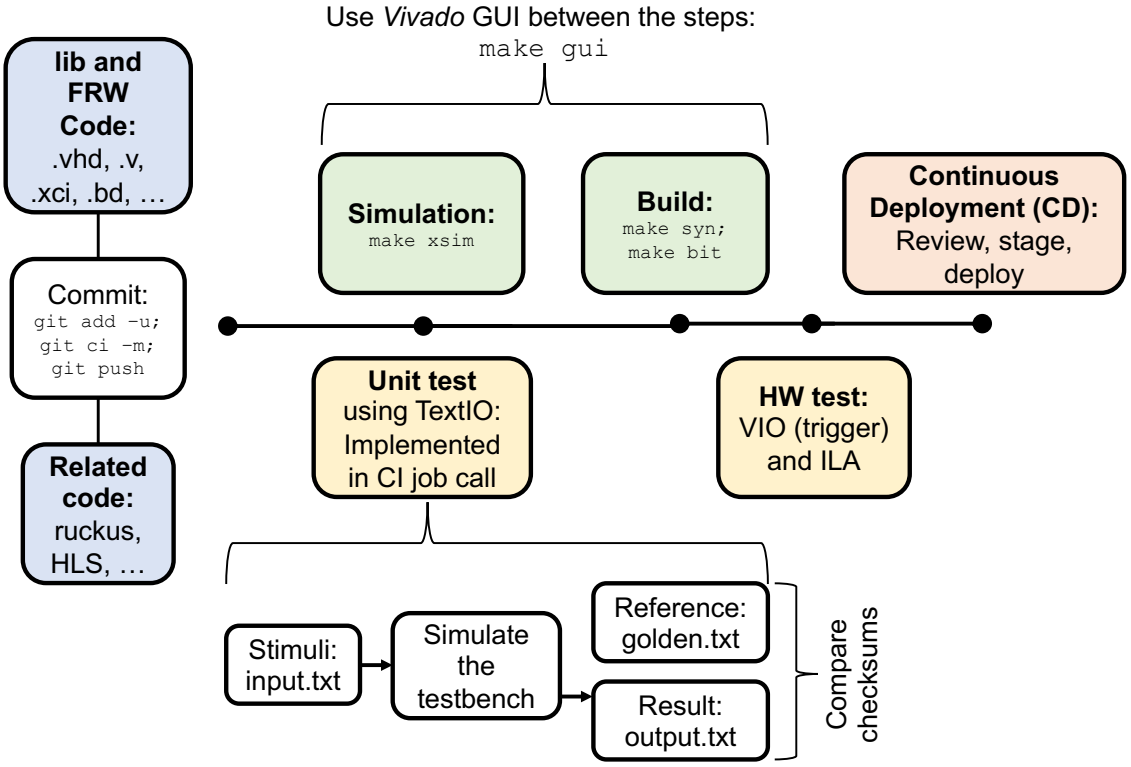

Figure 2: Overview of applied continuous integration using a command line interface tool for firmware builds. The command `make gui` is for manual debugging and not part of the continuous integration.

The input stimuli required by the Hardware Description Language (HDL) simulation are generated by a *Python* script from within a unit test. These stimuli are fed into a testbench using the *TextIO* library, at which point the *Vivado* HDL simulation generates the results. The checksum for the results is calculated and compared to a golden reference to determine if the test failed or passed. By generating the *input.txt* and *output.txt* on the fly, which both consisting of many lines of code depending on the simulation time and time resolution, we reduce the amount of data that has to be stored in the repository. In this CI example, we set up several HDL simulations, which are both performed and evaluated in parallel. The synthesis stage – followed by the build stage – will only be triggered if the simulation stage is successful. In the example setup, the hardware testbench is instrumented with Virtual Input/Output (VIO) modules and Integrated Logic Analyzer (ILA) modules provided by *Xilinx*. Figure 3 shows a screenshot of the CI pipeline. Note that due to the lack of a dedicated FPGA board for hardware testing, that portion of the CI workflow has been skipped.

The depicted pipeline is specified by the `.gitlab-ci.yml` configuration file. The results of CI pipelines can be accessed in form of artifacts, in our case a bit file, and log files. In other projects we extended the CI of firmware by combining it with CI of *Vivado* High-Level Synthesis (HLS), which generates HDL code from high-level code such as C/C++. It is easier to setup the CI for *Vivado* HLS because we can take advantage of some of the built-in TCL scripts for the stages: build-csim, build-csynth, build-cosim, and build-export.

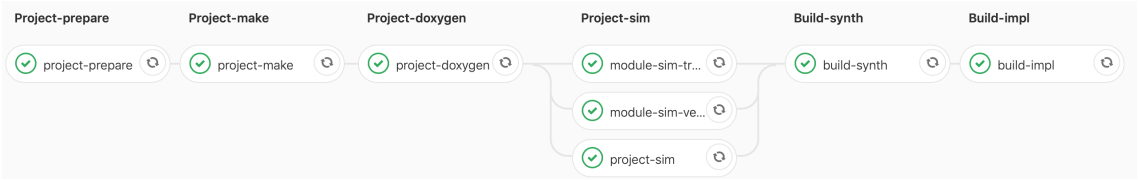

Figure 3: Screenshot of a successfully build *GitLab* CI/CD pipeline (stages and jobs) of an FPGA design. The *Project-sim* stage consists of three parallel simulation jobs.

We encountered different types of errors, whereby CI saved us time due to the early detection compared to a non-automated build and verification flow: Functional errors in simulation and hardware test (e.g. erroneous algorithms), erroneous build environment (e.g. wrong tool version), erroneous HDL module interfaces (e.g. changed width), wrong submodule commit, and timing closure errors.

4. Conclusion

CI for firmware builds offers results that are easier to reproduce compared to firmware project without CI. Errors are exposed faster and the troubleshooting is made easier by tracing an error to the exact commit. The manual verification of the firmware is minimized by setting up automated simulations and hardware tests. By using *GitLab* CI/CD developers can take advantage of a *git*-based, fully featured platform for software and firmware development. For many users, a CLI-based build system will help to simplify the setup of the CI project. The entire FPGA design process is supported by *GitLab* and *GitLab* CI/CD, including planning (organizing and tracking project progress), creating, verifying, packaging, releasing, configuring, and monitoring.

References

[1] N. Forsgren, D. Smith, J. Humble, and J. Frazelle, "State of DevOps 2019," Tech. Rep., 2019. [Online]. Available: https://services.google.com/fh/files/misc/state-of-devops-2019.pdf

[2] L. Ruckmann, "TID-AIR Electronics: An Introduction to Ruckus and SURF," Oct. 2018. [Online]. Available: https://docs.google.com/presentation/d/1kvzXiByE8WISo40Xd573DdR7dQU4BpDQGwEgNyeJjTI/edit

[3] "IPBB primer - IPbus SW v2.6.4, FW v1.5 documentation," 2018. [Online]. Available: https://ipbus.web.cern.ch/ipbus/doc/user/html/firmware/ipbb-primer.html

[4] "Hdlmake Wiki," 2019. [Online]. Available: https://www.ohwr.org/project/hdl-make/wikis/home

[5] "DevOps Tools Landscape | GitLab," 2019. [Online]. Available: https://about.gitlab.com/devops-tools/

[6] "GitLab CI/CD," 2019. [Online]. Available: https://docs.gitlab.com/ee/ci/

[7] "Jenkins," 2019. [Online]. Available: https://jenkins.io/

[8] "CircleCI," 2019. [Online]. Available: https://circleci.com/

[9] "Travis CI," 2019. [Online]. Available: https://travis-ci.org/

[10] CMS Collaboration, *The Phase-2 Upgrade of the CMS L1 Trigger Interim Technical Design Report*, Sep. 2017. [Online]. Available: https://cds.cern.ch/record/2283192