Continuous Integration (CI) of FPGA Designs for CMS

Topical Workshop on Electronics for Particle Physics (TWEPP) 2019
Robért Glein on behalf of the CMS Collaboration,
CU Boulder, September 3rd 2019
robert.glein@cern.ch
Problem: How to manage, build, and test FW designs?

• Precondition: The firmware (FW) code is versioned with *git* (or a similar versioning system)

• FW development
  • High degree of flexibility
  • Modular design often with various (*git*) submodules
  • Various tools for simulation (HDL), synthesis (netlist translation), implementation (place and route), and test of FPGA designs
  • Time intensive (functional) verification using various methods

• Solution: Continuous integration (CI) of FW
Outline

• Problem Description and Motivation
• Continuous Integration (CI) Overview
• Command Line Interface (CLI) FW Build Systems
• Continuous Integration / Continuous Deployment (CI/CD) Tools
• Demonstration
• Conclusion
Motivation of Continuous Integration for Firmware

- Widely used for software
- Especially useful with multiple developers working on the same or interconnected FW
- Automated check, build, simulation, and test from scratch
  - Uniform build, simulation, and test environment
  - An error can be traced to the exact commit
  - Covers specified functionalities via simulation and test
  - No risk of human error (changed constants, submodule not updated, specific tool not installed, …)
  - **Manual verification (simulation and HW) of FW is minimized**
    - Especially if different FPGAs and/or configurations are supported
- Requires a command line interface (CLI) build system
Continuous Integration (CI) Overview

- **Code:** .vhd, .v, .xcii, .bd, ...
- **Commit**
- **Simulation:** ModelSim, Vivado
- **Build:** synth, impl, bitgen
- **Continuous Deployment (CD):** Review, stage, deploy
- **Related code:** build, library, ...
- **Unit test**
- **HW test**
Continuous Integration (CI) Overview

- Code: .vhd, .v, .xci, .bd, ...
- Commit
- Simulation: ModelSim, Vivado
- Build: synth, impl, bitgen
- Continuous Deployment (CD): Review, stage, deploy
- Related code: build, library, ...
- Unit test
- HW test

- Manual code commit
Continuous Integration (CI) Overview

- Manual code commit
- Automated simulation (unit test), build, and HW tests
Continuous Integration (CI) Overview

- Manual code commit
- Automated simulation (unit test), build, and HW tests
- Manual review and automatic deployment
Outline

• Problem Description and Motivation
• Continuous Integration (CI) Overview
• Command Line Interface (CLI) FW Build Systems
• Continuous Integration / Continuous Deployment (CI/CD) Tools
• Demonstration
• Conclusion
Firmware CLI Build System: *Ruckus* by SLAC

- Well structured Makefile based CLI
- Supports incremental syn/impl feature (dcp), partial reconfiguration, and integration of HLS (all not tested by me)
- Some commands (right level of granularity):
  - `make depend`  Vivado Project Creation
  - `make sources`  Vivado Source Setup
  - `make xsim`  Vivado Simulation (CU contribution)
  - `make syn`  Vivado Synthesis Only
  - `make bit`  Vivado Build
  - `make interactive`  Vivado Interactive (CLI)
  - `make gui`  Vivado GUI
- Project website: [https://github.com/slaclab/ruckus](https://github.com/slaclab/ruckus)
- Documentation: [https://docs.google.com/presentation/d/1kvzXiByE8WlSo40Xd573DdR7dQU4BpDQGwEgNyeJjTI/](https://docs.google.com/presentation/d/1kvzXiByE8WlSo40Xd573DdR7dQU4BpDQGwEgNyeJjTI/)
Firmware CLI Build System: Investigated Alternatives

- **IP Bus Builder (IPBB)**
  - This *Python* tool is based on hierarchical dependency files *.dep* in cfg folders to include sources
  - Includes make-project, simulation, synth, impl, bit file creation, …

- **HDL Make**
  - Makefile based CLI but does not support block designs (recursively)
  - [https://www.ohwr.org/projects/hdl-make](https://www.ohwr.org/projects/hdl-make)

- **Vendor tools with CLI support (e.g. TCL)**
Outline

• Problem Description and Motivation
• Continuous Integration (CI) Overview
• Command Line Interface (CLI) FW Build Systems
• Continuous Integration / Continuous Deployment (CI/CD) Tools
• Demonstration
• Conclusion
**GitLab Continuous Integration / Continuous Deployment (CI/CD)**

- Documentation: [https://docs.gitlab.com/ee/ci/](https://docs.gitlab.com/ee/ci/) and [https://docs.gitlab.com/ee/ci/yaml/](https://docs.gitlab.com/ee/ci/yaml/)

- Can be used without any source files in the *GitLab* repository (e.g. all code located in GitHub)

Source: [https://docs.gitlab.com/ee/ci/README.html](https://docs.gitlab.com/ee/ci/README.html)
GitLab CI/CD
Investigated Alternatives

- [https://about.gitlab.com/devops-tools/](https://about.gitlab.com/devops-tools/)
- **CircleCI**: [https://circleci.com](https://circleci.com)
  - Supposed to have the best GitHub integration
  - After a glimpse, I did not find runner options (execute a script on any workstation)
- **Jenkins**: [https://jenkins.io](https://jenkins.io)
  - Solid CI tool but not that powerful
- **Travis**, …
Gitlab CI/CD
Runner (Executing Workstation)

- [https://docs.gitlab.com/ee/ci/runners/README.html](https://docs.gitlab.com/ee/ci/runners/README.html)
- Run the code defined in .gitlab-ci.yml (next slide)
- Isolated (virtual) machine: Pick up jobs through the coordinator (Gitlab server)
- Types:
  - Shared: Similar requirements between multiple projects (e.g. Python, GCC, …)
  - Specific: Special requirements (e.g. HW test workstation with connected boards)
  - Group: Multiple projects under one group (e.g. Xilinx tools like Vivado (HLS))
- GitLab server and runner are connected via token(s)
- gitlab-runner binary runs as gitlab-runner user on the workstation and can be configured (e.g. number of parallel jobs)
- Docker support for the build environment
Gitlab CI/CD
Configuration File: .gitlab-ci.yml

- [https://docs.gitlab.com/ee/ci/yaml/](https://docs.gitlab.com/ee/ci/yaml/)
- Highly configurable with a lot of options. Examples:

```yaml
variables:
  VIVADO_PATH_SH: '/opt/Xilinx/Vivado/${VIVADO_VERSION}/settings64.sh'

stages:
  - project-prepare
  - project-make

.job_template: &template_base
tags:
  - xilinx-tools
before_script:
  - source $VIVADO_PATH_SH

project-prepare: # Actual job
<<: *template_project_prepare # Calls *template_base (not shown)
variables:
  VIVADO_VERSION: "2019.1"
```

09/03/19 Continuous Firmware Integration
Gitlab CI/CD
Configuration File: .gitlab-ci.yml

- [https://docs.gitlab.com/ee/ci/yaml/](https://docs.gitlab.com/ee/ci/yaml/)
- Highly configurable with a lot of options. Examples:

```yml
variables:
  VIVADO_PATH_SH: '/opt/Xilinx/Vivado/$(VIVADO_VERSION)/settings64.sh'

stages:
  - project-prepare
  - project-make

.job_template: &template_base
  tags:
    - xilinx-tools
  before_script:
    - source $VIVADO_PATH_SH

project-prepare: # Actual job
  <<: *template_project_prepare # Calls *template_base (not shown)
  variables:
    VIVADO_VERSION: "2019.1"
```

09/03/19 Continuous Firmware Integration
**Gitlab CI/CD**  
Configuration File: `.gitlab-ci.yml`

- [https://docs.gitlab.com/ee/ci/yaml/](https://docs.gitlab.com/ee/ci/yaml/)
- Highly configurable with a lot of options. Examples:

```yaml
variables:
  VIVADO_PATH_SH: '/opt/Xilinx/Vivado/${VIVADO_VERSION}/settings64.sh'

stages:
  - project-prepare
  - project-make

.job_template: &template_base
tags:
  - xilinx-tools
before_script:
  - source $VIVADO_PATH_SH

project-prepare: # Actual job
  <<: *template_project_prepare # Calls *template_base (not shown)
variables:
  VIVADO_VERSION: "2019.1"
```

09/03/19

Continuous Firmware Integration
**Gitlab CI/CD**

**Configuration File: .gitlab-ci.yml**

- [https://docs.gitlab.com/ee/ci/yaml/](https://docs.gitlab.com/ee/ci/yaml/)
- Highly configurable with a lot of options. Examples:

```yaml
variables:
  VIVADO_PATH_SH: '/opt/Xilinx/Vivado/${VIVADO_VERSION}/settings64.sh'

stages:
  - project-prepare
  - project-make

.job_template: &template_base
  tags:
    - xilinx-tools
  before_script:
    - source $VIVADO_PATH_SH

project-prepare: # Actual job
  <<: *template_project_prepare # Calls *template_base (not shown)
  variables:
    VIVADO_VERSION: "2019.1"
```
Gitlab CI/CD
Configuration File: .gitlab-ci.yml

- [https://docs.gitlab.com/ee/ci/yaml/](https://docs.gitlab.com/ee/ci/yaml/)

- Highly configurable with a lot of options. Examples:

```
variables:
  VIVADO_PATH_SH: '/opt/Xilinx/Vivado/${VIVADO_VERSION}/settings64.sh'

stages:
  - project-prepare
  - project-make

.job_template: &template_base
  tags:
    - xilinx-tools
  before_script:
    - source $VIVADO_PATH_SH

project-prepare: # Actual job
  <<: *template_project_prepare # Calls *template_base (not shown)
  variables:
    VIVADO_VERSION: "2019.1"
```
CI Development I

**lib Code:**
- .vhd, .v,
- .xci, .bd, ...

**Commit:**
- git add -u;
- git ci -m;
- git push

**FRW Code:**
- .vhd, .v,
- .xci, .bd, ...

**Commit:**
- git add -u;
- git ci -m;
- git push

- **Use Vivado GUI between the steps:**
  - make gui

- **Simulation:**
  - make xsim

- **Build:**
  - make syn;
  - make bit

- **Related code:**
  - ruckus,
  - HLS, ...

- **Continuous Deployment (CD):**
  - Review, stage,
  - deploy

- **Unit test using TextIO:**
  - Implemented in CI job call

- **Stimuli:**
  - input.txt

- **Simulate the testbench**

- **Reference:**
  - golden.txt

- **Result:**
  - output.txt

- **HW test:**
  - VIO (trigger)
  - and ILA

- **Compare checksums**
CI Development II:
https://gitlab.cern.ch/GTT/fpga/-/jobs

- Template based CI stages/jobs
- Required SW is tagged

<table>
<thead>
<tr>
<th>Status</th>
<th>Job</th>
<th>Pipeline</th>
<th>Stage</th>
<th>Name</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬠ failed</td>
<td>#3964704</td>
<td>master → 0cf84b18</td>
<td>build-impl</td>
<td>build-impl</td>
<td>01:38:51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#817421 by</td>
<td></td>
<td></td>
<td>1 hour ago</td>
</tr>
<tr>
<td>⬠ passed</td>
<td>#3964703</td>
<td>master → 0cf84b18</td>
<td>build-synth</td>
<td>build-synth</td>
<td>00:39:14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#817421 by</td>
<td></td>
<td></td>
<td>2 hours ago</td>
</tr>
<tr>
<td>⬠ passed</td>
<td>#3964702</td>
<td>master → 0cf84b18</td>
<td>project-sim</td>
<td>project-sim-mux64</td>
<td>00:02:06</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#817421 by</td>
<td></td>
<td></td>
<td>3 hours ago</td>
</tr>
<tr>
<td>⬠ passed</td>
<td>#3964701</td>
<td>master → 0cf84b18</td>
<td>project-sim</td>
<td>project-sim</td>
<td>00:03:20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#817421 by</td>
<td></td>
<td></td>
<td>3 hours ago</td>
</tr>
<tr>
<td>⬠ passed</td>
<td>#3964700</td>
<td>master → 0cf84b18</td>
<td>project-doxygen</td>
<td>project-doxygen</td>
<td>00:05:27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#817421 by</td>
<td></td>
<td></td>
<td>3 hours ago</td>
</tr>
<tr>
<td>⬠ passed</td>
<td>#3964699</td>
<td>master → 0cf84b18</td>
<td>project-make</td>
<td>project-make</td>
<td>00:02:41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#817421 by</td>
<td></td>
<td></td>
<td>3 hours ago</td>
</tr>
<tr>
<td>⬠ passed</td>
<td>#3964698</td>
<td>master → 0cf84b18</td>
<td>project-prepare</td>
<td>project-prepare</td>
<td>00:00:27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#817421 by</td>
<td></td>
<td></td>
<td>3 hours ago</td>
</tr>
</tbody>
</table>
CI Development III: Build Including Parallel Simulations


```yaml
project-sim:
  <<: *template_project_sim
  variables:
    VIVADO_VERSION: "2018.3"
    VIVADO_PROJECT_SIM: tb_top
  before_script:
    - source $VIVADO_PATH_SH # needed because template before_script is replaced
    - cd framework/hdl/tb/vcu118/
    - python ut_generator.py 0.001 1000000
    - cd ../.....
  after_script:
    - export HASH="$(md5sum ./framework/hdl/tb/vcu118/output_text.txt | awk '{ print $1 }')" # calculate the md5sum
    - if grep -q $HASH ./framework/hdl/tb/vcu118/golden_hash.txt; then echo "Hash found: $HASH"; else echo "No hash found" && exit 1; fi # check if hash is found

project-sim-mux64:
  <<: *template_project_sim
  variables:
    VIVADO_VERSION: "2018.3"
    VIVADO_PROJECT_SIM: tb_mux64
```

- [https://gitlab.cern.ch/GTT/fpga/pipelines/1056751](https://gitlab.cern.ch/GTT/fpga/pipelines/1056751) (visualization)
Outline

• Problem Description and Motivation
• Continuous Integration (CI) Overview
• Command Line Interface (CLI) FW Build Systems
• Continuous Integration / Continuous Deployment (CI/CD) Tools
• Demonstration
• Conclusion
CI Demonstration

- [https://gitlab.cern.ch/GTT/fpga/pipelines](https://gitlab.cern.ch/GTT/fpga/pipelines)
- Access the *GitLab* repository
- Rerun implementation
- Watch the CI doing its job
- Explain different CI features
  - Pipeline
  - Jobs: Log and artifacts
  - Charts
CI Conclusion

- Faster and reproducible results
- Easier troubleshooting (an error can be traced to the exact commit)
- **Manual verification of FW is minimized**
- Using *GitLab* CI/CD
  - Git-based, fully featured platform for SW and FW development
  - FW requires CLI build system
  - Support for the entire FPGA design process
    - Plan, create, verify, package, release, configure, and monitor
- What happens if you do not use CI?
  - If you forget to test the entire chain, your algorithm embedded in the infrastructure may cause errors (timing, not tested function, …)
  - Errors may be hidden and pile up until the system breaks down
CI Conclusion

• Faster and reproducible results
• Easier troubleshooting (an error can be traced to the exact commit)
• Manual verification of FW is minimized
• Using GitLab CI/CD
  • Git-based, fully featured platform for SW and FW development
  • FW requires CLI build system
  • Support for the entire FPGA design process
    • Plan, create, verify, package, release, configure, and monitor

• What happens if you do not use CI?
  • If you forget to test the entire chain, your algorithm embedded in the infrastructure may cause errors (timing, not tested function, …)
  • Errors may be hidden and pile up until the system breaks down
  • DON’T LET THIS HAPPEN TO YOU! ➔ USE CI/CD!
Back Up
https://about.gitlab.com/devops-tools/
(From the *Gitlab* perspective)
Track Stand-Alone Object (Global Track Trigger) in CMS: Overview

- **Track Interface Rx**
  - Track FIFOs (2 Events)
  - 96 b Converter
  - FIFOs + optional conv.
  - 18x64 b

- **Algorithm**
  - MET (HLS) @ $f_{clk, MET}$
  - Jet (Verilog) @ $f_{clk, Jet}$
  - Vertex (HLS) @ $f_{clk, Vertex}$

- **Object Interface Tx**
  - Converter with FIFOs
  - vld / rdy

- **FW Modules**
  - FW Modules for TMUX=1
  - FW Modules for TMUX=7
  - FW Modules for TMUX=13

- **Keywords**
  - Rx1, Rx2, ..., Rx18 → TMUX=1
  - Rx19, Rx20, ..., Rx36 → TMUX=7
  - Rx37, Rx38, ..., Rx54 → TMUX=13

- **GTY**
  - 9x3 TSA_Seri; 9x3 Muon; 4xVtx PF; 2xJF&MET GT; 1 DAQ

- **TF** @ 25 Gbps

- **Timeline**
  - 09/03/19
  - Continuous Firmware Integration
Doxygen: Automated Generation of Code (e.g. HDL) Documentation

Top Entity Reference

Top entity with a FPGA-independent interface. More...

Inheritance diagram for top:

```
  p_T_lut_V_0_rom     p_T_lut_V_1_rom     eta_lut_V_0_rom     eta_lut_V_1_rom
    |            |            |            |            |
   /            /            /            /
p_T_lut_V_0   p_T_lut_V_1   eta_lut_V_0   eta_lut_V_1
    |            |            |            |            |
   /            /            /            /
  p_T          eta          decipher_v2018_2  z_cascade
    |            |            |                |
   /            /            /                /
track_gen_bram track_conv  jet_finding_top2  ila_jet_finding
    |            |            |                |
   /            /            /                /
  top          top          top              top
```

09/03/19

Continuous Firmware Integration