GitLab-CI for FPGA development at LHCb
The EP/LBC (Online) group
GitLab-CI
(Continuous Integration)
GitLab-CI at LHCb (FPGA development only)
Our environment

- **FPGA firmware**
  - Multiple repositories (submodules)
  - Languages: VHDL, Verilog, TCL
  - Toolchain: Questa, Quartus (proprietary)
  - CI: shell runners

- **Low-level software**
  - Multiple repositories (independent)
  - Languages: C, C++, Python
  - Toolchain: GCC, Python...
  - CI: shared runners

- **SCADA middleware**
  - One repository
  - Languages: CTRL (Siemens proprietary)
  - Toolchain: PVSS (proprietary)
  - CI: dind? [wip]

- Each has its own CI pipelines, but in the end we also have to test the integration of the different pieces working together.
Target hardware

We receive data from the detectors (proprietary rad-hard protocols over optical fiber), process it and emit it in a “COTS-friendly” format. We have to produce different firmware (sometimes more than one) for each device and for each sub-detector.

AMC40 (Legacy)
- GBT
- Front-end rad-hard optics

PCle40 (Production)
- GBT
- Back-end high-speed links
- PCle

10GbE
FW synthesis combinatorics

Control+Data “MiniDAQ”

Control Plane

Data Plane

TEST

SciFi/FF SciFi/FV

MUON

VELO

RICH/LO RICH/HO

CALO

UT/x3 UT/x4 UT/x5

Control Plane

Data Plane

Test Plane

SciFi Plane

MUON Plane

VELO Plane

RICH Plane

CALO Plane

UT Plane

Control Plane

Data Plane

Test Plane

SciFi Plane

MUON Plane

VELO Plane

RICH Plane

CALO Plane

UT Plane

Control Plane

Data Plane

Test Plane

SciFi Plane

MUON Plane

VELO Plane

RICH Plane

CALO Plane

UT Plane

Single-feature images

Timing Distribution

AMC40

PCIe40v1

PCIe40v2.x
Firmware repository

- FPGA firmware split across different responsibility areas
- Different developers / institutes work in parallel on their respective component
- Each component is its own repository, versioned using git tags and semantic versioning
- One upstream repo tracks the global state of the firmware, each component is a git submodule
- Python scripts to manage submodules for users with little git experience
- CI runs EDA design flow against the upstream repo
- Submodules can trigger builds in the upstream repo for quick testing
Firmware integration flow

**DEVELOPER**

1. Push changes to one or more components
2. Click a manual action in GitLab to build new changes (or wait for the nightly build)
3. When satisfied, run a script that will generate tags and submit the new components upstream (a merge request is automatically created)

```bash
./scripts/git/release --minor --merge mycomponent/
```

**MAINTAINER**

1. Receive a merge request (a pipeline is automatically started)
2. If the change is minor, can decide to automatically merge when the pipeline completes
   
   ![Pipeline #12399611 running for 5a733d30.](image)

3. If not, wait for pipeline to publish firmware RPMs, go to a development machine, run `yum install lhcb-pcie40-firmware-merge-<MR#>` and test it
4. Push fixes to merge request until it can be accepted
5. Once merged, click a manual action in GitLab to automatically create and publish a new tag

![create_tag](image)
Firmware pipeline

- Executed nightly, or when users send changes upstream, or when manually triggered
- ~30 compilations (for now)
- Most compilations can take several hours (between 4 and 10 depending on complexity) but only a few minutes if cache is still valid
- Custom scripts to avoid a rebuild whenever possible (EDA tools have poor dependency tracking)

(30m ~ 20h)
Firmware lessons learned

- **Resources**
  Describe simulation and compilation resources available in terms of gitlab-runner tags, e.g. `lhcb-daq40-firmware-quartus161-32G`, `lhcb-daq40-firmware-cache`, `lhcb-daq40-firmware-publish`.

- **Simulations**
  Pool resources from other institutes within the LHCb collaboration.

- **Concurrent jobs**
  Use `rsync(1)` and `flock(1)` to prevent Quartus cache corruption.

- **Long-running jobs**
  Write cron scripts to keep idle Quartus jobs alive (as gitlab-runner spontaneously terminates them after 1 hour).

- **Dependencies**
  Keep Quartus project reports, write scripts to compare git histories between rebuilds.

- **Merges**
  Automatically create MR based on branch name, reuse build cache after merge, fast-forward upstream merges.

- **Branches**
  Branches with the same name across components are triggered together (using GitLab CI variables).

- **Deployment**
  Distribute firmware, WinCC components, software etc. as RPMs and write tools to install/program automatically.

- **Traceability**
  Store git version information in the FPGA image automatically, and write tools to extract this info in the field.
Software pipeline

- Low-level software and kernel drivers for our custom hardware
- Main platform is CERN Centos7
- We still build for SLC6 and i386 for legacy platform support
- Default system compiler + GCC7 for extra diagnostics

Build
- make-cc7-dim
- make-cc7-disa...
- make-cc7-driver
- make-cc7-hwloc
- make-cc7-mon
- make-doc
- make-slcc6-dim
- make-slcc6-disa...
- make-slcc6-driver
- make-slcc6-hwloc

Package
- dkms-cc7
- dkms-slcc6
- rpm-cc7-amc40
- rpm-cc7-daq40
- rpm-cc7-pcie40
- rpm-doc
- rpm-slcc6-amc40
- rpm-slcc6-daq40
- rpm-slcc6-pcie40

Install
- install-cc7
- install-slcc6

Publish
- publish-unstable

(~10m)
WinCC-OA “pipeline”

- Program FPGA
- Subscribe
- Reload
- Read information
- Check PRBS
- Check RxCounters
- Check RxReady
- Configure
- Start Run
- Check Triggers
- Stop Run
- TAP report
Final remarks

- Overall, our experience with GitLab has been very positive!
  - GitLab REST API is powerful enough to let us customize the CI flow to our needs

- I’m slowly converting our group to use GitLab CI
  - Some “devops” attitude is required, but we make the system as easy to use as possible
  - Some of these tools are still unfamiliar outside the domain of software engineering
  - Integrating with EDA and SCADA toolchains requires extra effort

- Work in progress
  - Integrate simulation output validator in pipeline
  - Automate hardware-in-the-loop integration tests

- Eventually
  - Deploy/rollback using GitLab environments
  - Publish metrics and monitor hardware performance in a live system (Grafana/Kibana)
DIY (Do It Yourself)

You will need:

1. **GitLab CI**
   - Centrally provided by IT-CDA
   - Includes shared Docker runners

2. Your toolchain packaged in a way that GitLab can use (Quartus/Vivado/etc.)
   - Could be centrally provided by IT-CDA?
   - Or: build your own Docker images
   - Or: set up private runners on a physical or virtual machine (e.g. on CERN OpenStack)

3. A way to share artifacts across pipelines (if you want to reuse builds)
   - E.g.: some NFS/SMB filesystem running on CERN OpenStack
   - Or: GitLab CI provides an internally managed cache since version 9.0
     - We have no experience with it however, and our cache is currently around 200 GB

4. Patience to write and debug lots of scripts! (it pays off, eventually)
   - In addition to TCL, we use python a lot
     - Python-gitlab for GitLab API integration
     - GitPython to automate GIT repository manipulation
Thank you for your attention
GitLab issues

   StuckCiJobsWorker wrongly detects, cancels 'stuck' builds when per-job timeout is more than an hour
   - Some of our jobs can easily require several hours
   - We have less machines than we have jobs
   - For now, we set up cronjobs running in background that use the GitLab API to find jobs that have been killed for inactivity and resuscitate them

   Relative submodule link to a nested project fails to resolve
   - Our firmware flow relies on submodules
   - A user should be able to click on a submodule and open the corresponding repository at the correct commit state
   - It’s already fixed on gitlab.com but not yet in the instance deployed at CERN
FPGA development 101

- Circuit is defined in a Hardware Description Language
- Simulator + testbench + input vectors reproduce behavior of device and allow some form of debugging
- Compiler toolchain maps specification into device-specific gate array configuration and interconnection
  - Lots of TCL scripts
  - 48 GB RAM!! (vendor recommendation)
  - As many GHz as you can afford
  - Still very time consuming
- Workarounds
  - Distribute different compilations to different machines
  - Reuse artifacts aggressively

- Simulation (~minutes)
- Analysis & Synthesis (~hour)
- Place & Route (~hours)
- Configure device (~seconds)
- Integration (software and middleware)
Firmware pipeline - stages

Jobs in “Prepare” stage

- create_mr
  - Users run a script to submit changes upstream
  - Pipeline picks up the new branch and creates a merge request automatically

- cache_master
  - Previous compilations are preserved and cached
  - Custom TCL script during FPGA synthesis checks project against git history for modifications

- clean_mr
  - Ensure the merged branch is deleted
  - Update cache for master branch with last compilation

Jobs in “Tag” stage

- create_tag
  - Manual job
  - When maintainer wants to create a new release, a tag is created according to our naming convention

- clean_tag
  - Remove project cache for tag once firmware has been released
  - Every cache amounts to tens of gigabytes