CMS HGCAL and gitlab pipelines

SoC Interest Group Meeting

Arnaud Steen, on behalf of the HGCAL

May 3, 2022
CMS Endcap calorimeter for phase 2

- Electromagnetic calorimeter (CE-E): Si, Cu/CuW/Pb absorbers, 26 layers (double sided cassettes), 25.5 $X_0$ & $\sim 1.7 \lambda$
- Hadronic calorimeter (CE-H): Si, and scintillator, steel absorbers, 21 layers, $\sim 9.5 \lambda$
Use of SOC in HGCAL: single module/ROC test system

- Hexa-controller test system with silicon module
  - Custom board hosting a Trenz TE0820 module with a ZYNQ UltraScale+
  - "Trophy" board carrying power and signals from/to HGCROCs
  - Silicon module with embedded ROCs

- Same test system for single ROC testboard

- Very similar test system for tile-module
Use of SOC in HGCAL: ECON-T testing

- Concentrator ASICs in HGCAL
  - ECON-T: collect and filter trigger primitives from HGCROC and transmit them to lpGBTs
  - ECON-D: collect DAQ data from HGCROC and transmit them to lpGBTs

![Image of test board with labels: FPGA, Individual power domains, ASIC power @ 1.2 V]
Use of SOC in HGCAL : v2/3 system tests

- **V2 system test:**
  - ZCU102 as back-end (fast command, link capture, lpGBT control)
  - Hexacontroller (with Trenz) as ECON emulators

- **V3 system test:**
  - ZCU102 as back-end
  - Hexacontroller (not in the picture) will be used as ECON-D emulators as it is not yet available
Next uses of SOC in HGCAL: robot for testing HGCROC

• ≈ 120k HGCROC to test during production
• 2 robots will have each 5 single ROC testers
Next uses of SOC in HGCAL: multi-module test system

- Si-modules will be tested inside a cold box ($\approx -30^\circ C$) after assembly
- 6 Si module assembly centers will be equipped with such system
- Hexaboards (30k) will be also tested with such system in 1 or 2 labs
Use of SOC in HGCAL

Similar use of TE0820 module and ZCU102:
- root and boot partitions placed in SD card
- using centos 7
- firmware loaded "manually" by the user with a python script (≈ re-writting of the fpgautil.c : a simple command line tool to load FPGA)
- Xilinx IPs:
  - Direct I2C for ROC configuration + ADC (ROC power consumption, DC levels) readout on the "trophy" board and single ROC socket board
  - Direct GPIO to control signals like resets for the ROCs, enable/power good for LDOs
- Custom IPs → AXI lite and AXI full. Using uio driver: "uio-pdrv-genirq" and uhal library (ipbus-software):
  - Control of the registers of fast command block, link capture block ...
  - Readout of FIFOs
  - Control of lpGBT registers
- SOMs registered on network and use DHCP

Running test in practice:
- ssh connection in the ZYNQ
- load the PL for the system at hand
- start the SW
Firmware and software in HGCAL test systems

- **Common firmware blocks for several test systems:**
  - Same fast command block for single module/ROC test system and V2/3 system test
  - Same link capture block for
    - single module/ROC test system capturing ROC data (trigger primitive or DAQ data)
    - V2/3 system capturing ECON data
  - "uio-pdrv-genirq" to have interface with AXI registers and software

- **Common software using a custom ipbus-software version to memory map the AXI registers and read the FIFOs.** Was inspired by Dan Gastler’s presentation (ApolloUpdate slides), from which we added:
  - Interrupt signal handler
  - "Non-incremental" block read (to read FIFOs)
Example: single module/ROC test system

- DAQ flow overview

  - Synchronization of the software by using zmq library: [https://zeromq.org/](https://zeromq.org/)
  - Configuration using yaml format: [https://yaml.org/](https://yaml.org/)

---

**Remote PC**
- Yaml Files (initLD.yaml)
  - Rep/Req socket:
    - send config (HGROC config)
    - read back config

**ZYNQ PS**
- ZMQ master (full_test.py, python)
  - Rep/Req socket:
    - send config (L1A type, BX ...)
    - start run
    - check if run finished

**ZYNQ PL**
- I2C server (zmq_server.py, python)
- ROC config, with I2C
- Event building
  - Data serialization

**Remote PC**
- I2C server (zmq_server.py, python)
- Rep/Req socket:
  - send config (HGCROC config)
  - read back config

**Remote PC**
- DAQ client (zmq_client.c++)
  - Push/Pull socket
  - Binary data
  - Storage disk

**Remote PC**
- Storage disk
- DAQ client (zmq_client.c++)
  - Push/Pull socket

**Remote PC**
- DAQ Server (zmq-server, c++)
  - Event building
  - Data serialization

**Remote PC**
- I2C master (full_test.py, python)
  - Rep/Req socket:
    - send config
    - start run
    - stop

**Remote PC**
- I2C bus
- Python I2C
- Fast control
  - Clock and fast commands @ 320 MHz
  - elinks @ 1.28 Gb
  - 2 DAQ links per ROC
  - 4 (or 2 for HD board) TRG links per ROC

**Remote PC**
- Link capture & BRAM (or FIFO)

**Remote PC**
- Hexaboard or Single ROC
  - ROC config, with I2C
Gitlab chain for HGCAL firmware

- Last year (link):
  - vivado running in dedicated docker runner launched with gitlab pipeline
  - artifacts with .bit and .dtsi files, to be downloaded as zipped file

- device tree compiler used locally to create the .dtbo file before loading the FPGA
Gitlab chain for HGCAL firmware

Update since last year:

- Device tree compiler run in the gitlab pipeline to create the .dtbo and add it to the artifacts
- Configurable (depending on the design) .xml files for ipbus-software added to the artifacts
- Artifacts packaged inside RPM
- RPMs upload to a yum repository (hosted on a eos website).

To install/update the FW:

```
yum install -y hexaboard-hd-tester-v1p1-trophy-v2
```
Gitlab chain for HGCAL software for single module/ROC tester

Before having gitlab CI/CD for the SW
- DAQ server and DAQ client (c++) needed to be compile (using cmake tool) from source code
- DAQ server and client depend on several pre-requisites : ipbus-software (only server), zmq, yaml-cpp, boost ...

Gitlab CI/CD for HGCAL software
- pipeline to build the software for aarch64 (server) and x86_64 (client)
- split the pipeline into several steps and save container image after installing pre-requisite:
  1. Build container image with centos7 and with installing pre-requisites
  2. Build container image : compile cppzmq latest versions
  3. Build container image (only done for aarch64): compile HGCAL ipbus-software version + create and save RPM on the yum repository
  4. Use 3rd image (resp. 2nd image) image to compile the DAQ server (resp. client) + create and save RPMs on the yum repository. Submodules (python SW) are also packaged inside the RPMs.
Gitlab chain for HGCAL software: docker build template

- Docker build template being re-used in steps 1, 2 and 3

```yaml
.build_template:
  stage: build
  image: docker
  tags:
    - docker-privileged
  services:
    - docker:dind
  before_script:
    - docker run --rm --privileged aptman/qus --static -- -p ${TGT_ARCH} # only aarch64
    - docker login -u ${CI_REGISTRY_USER} -p ${CI_REGISTRY_PASSWORD} ${{CI_REGISTRY}}
  script:
    - docker build
      ${CI_PROJECT_DIR}
      --file ${CI_PROJECT_DIR}/${CONTEXT_DIR}/Dockerfile
      --tag ${REG_SLUG}:latest
      --tag ${REG_SLUG}:${CI_COMMIT_REF_NAME}
      --build-arg CI_COMMIT_REF_NAME=${CI_COMMIT_REF_NAME}
    - docker push --all-tags ${REG_SLUG}
```
Gitlab chain for HGCAL software: dockerfiles

- Step 1: starting from centos 7 image + install pre-requisite

```bash
FROM arm64v8/centos:7
RUN yum install -y epel-release centos-release-scl-rh
    && yum update -y
    && yum install -y
    cmake
    cmake3
    zeromq
    zeromq-devel
    libyaml
    libyaml-devel
    yaml-cpp
    yaml-cpp-devel
    boost
    boost-devel
    python3
    python3-devel
    python3-dev
    autoconf-archive
    pugixml
    pugixml-devel
    make
    gcc-c++
git
rpm-build
devtoolset-10
&& yum clean all
```

- Step 2: compile and install cppzmq (from a fork of cppzmq in which we added the CI pipeline)

```bash
FROM gitlab-registry.cern.ch/hgcal-daq-sw/docker-images/centos7/centos7-aarch64:latest
ADD ./ /home/centos7-with-cppzmq
ENV CPPZMQ_PATH="/home/centos7-with-cppzmq"
ENV BUILD_DIR="/home/centos7-with-cppzmq/build"
RUN mkdir -p ${BUILD_DIR} && cd ${BUILD_DIR} && cmake .. / && make -j`nproc` && make install && cpack3 && ls ${BUILD_DIR}
```

- Step 3: compile HGCAL version (to have AXI over UIO over uhal) ipbus-software

```bash
FROM gitlab-registry.cern.ch/hgcal-daq-sw/docker-images/centos7-with-cppzmq/centos7-aarch64:latest
# This is where the COPY/ADD would go to get the git repo
ADD ./ /home/ipbus-software
# This is where the building process would go
ENV ORIGINAL_PATH="/home/ipbus-software"
ENV LONG_ENOUGH_PATH="/home/ipbus-software_________
RUN lscpu
    && mv ${ORIGINAL_PATH} ${LONG_ENOUGH_PATH}
    && mkdir -p ${ORIGINAL_PATH}
    && cd ${LONG_ENOUGH_PATH}
    && export CPPLUS_INCLUDE_PATH=${CPPLUS_INCLUDE_PATH}/usr/include/python3.6m/
    && make -k -j`nproc` Set-uhal
    && make install -j`nproc` Set-uhal
    && export PACKAGE_RELEASE_SUFFIX=hgcal_v0_0_0
    && make -k Set-uhal PACKAGE_RELEASE_SUFFIX=${PACKAGE_RELEASE_SUFFIX} rpm
    && cp find .../*_rpmrpm" ${ORIGINAL_PATH}
    && ls ${ORIGINAL_PATH}
```

- Need ≈ 30 mins on shared runner → important to have this step separated
Gitlab chain for HGCAL software: last step

- Build DAQ server for aarch64:

```bash
build_aarch64:
  stage: build
  # to be used if docker-arm shared runner is not available anymore
  tags:
    - docker-privileged
  services:
    - hypriot/qemu-register:latest
  # to be used if docker-arm shared runner is available (it is faster)
  tags:
    - docker-arm
  image:
    name: gitlab-registry.cern.ch/hgcal-daq-sw/ipbus-software/centos7-aarch64:latest
  script:
    - ls $(PWD)
    - sed -i '/\{REG_MAPS\}/g' /opt/hexacore/\$\{CI_COMMIT_REF_NAME\}/etc/g2c/Translator.py
    - export BUILD_DIRS=$(PWD)/build
    - mkdir $(BUILD_DIR)
    - cd $(BUILD_DIR)
    - echo "BRANCH = $CI_COMMIT_REF_NAME"
    - scl enable devtoolset-10 'cmake -DBUILD_CLIENT=ON -DROOT_INCLUDE_DIRS=/usr/include/root
    - DBRANCH_NAME=$CI_COMMIT_REF_NAME ..;/ make -j" nproc" make install ; cpack'
    - mkdir -p $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
    - cp *.rpm $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
    - echo $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
    - ls $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
  artifacts:
    paths:
      - $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
```

- Build DAQ client for x86_64:

```bash
build_x86_64:
  stage: build
  image:
    name: gitlab-registry.cern.ch/hgcal-daq-sw/docker-images/centos7-with-cppmxg/centos7-x86_64:latest
  script:
    - ls $(PWD)
    - export BUILD_DIR=$(PWD)/build
    - mkdir $(BUILD_DIR)
    - cd $(BUILD_DIR)
    - scl enable devtoolset-10 'cmake -DBUILD_CLIENT=ON -DROOT_INCLUDE_DIRS=/usr/include/root
    - DBRANCH_NAME=$CI_COMMIT_REF_NAME ..;/ make -j" nproc" make install ; cpack'
    - mkdir -p $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
    - cp *.rpm $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
    - echo $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
    - ls $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
  artifacts:
    paths:
      - $(CI_PROJECT_DIR)/$(CI_OUTPUT_DIR)
```

- CI/CD pipeline

<table>
<thead>
<tr>
<th>Build</th>
<th>Deploy</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_aarch64</td>
<td>deploy-eos</td>
</tr>
<tr>
<td>build_x86_64</td>
<td></td>
</tr>
</tbody>
</table>
Summary and next plans

- Using gitlab pipelines and RPMs for software and firmware building and deployment. Helpful for:
  - FW/SW developers as it gives quick confirmation if a commit is OK
  - FW/SW developers as it garanties that the users are using right version of FW/SW and all their submodules
  - Users: "yum install ..." without having to compile from source is easier and more convenient. It avoid issues with different pre-requisite versions ...

- Issue in getting Trenz modules with infinite delivery dates (9.9.9999)

- Considering using Kria modules instead of Trenz
  - Move from Vivado 2019.2 to 2021.2

- Will then have Trenz, Kria and ZCU102 systems
  - Plan to automatize PetaLinux build