Hardware/Software Co-Design with Gitlab CI

CROME: CERN RadiatiOn Monitoring Electronics

Amitabh Yadav, Hamza Boukabache, CROME Team
amitabh.yadav@cern.ch

SoC Interest Group Meeting
23/11/2022

Reference: https://codimd.web.cern.ch/s/7VHssRXbY#
Agenda

• Gitlab CI - what and why?
• CI and Docker Overview
• CROME CI Pipeline
• Key Features – Dependencies and CI Special Variables
• Building Xilinx Petalinux Image through CI
• Future Steps – Verification and Continuous Deployment.
Continuous Integration (CI)
Continuous Integration (CI) is the practice of continuously integrating and verifying the code changes automatically through CI pipelines.
Gitlab CI - what and why?

Continuous Integration (CI) is the practice of continuously integrating and verifying the code changes automatically through CI pipelines.

Advantages:

- Ensures successful HW and SW build/compilation.
- Automatic code quality and performance testing.
- Early detection of errors, bug tracking, reduced integration problems, and faster deployment.
- Reproducible builds using Docker containers [1]
- Code packaging and deployment.

CI Pipeline

- CI Pipeline:
  - A pipeline is a sequence of scripted steps that will be executed on the code in repository.
  - The steps are defined in the file `.gitlab-ci.yml` and is placed in the root of the project repository.
  - Gitlab detects the YAML file and initiates the GitlabCI script.
  - No modifications to the project repository.
  - At the end, any new files/outputs created during the process of execution of a CI script are called artifacts.

Image source: docs.gitlab.com
CI Pipeline - continued

- CI pipeline components - Jobs and Stages:
  - A pipeline is composed of independent **jobs** that execute scripts.
  - Jobs are grouped into **stages**.
  - Stages run in sequential order, but jobs within stages run in parallel.

- **Artifacts:**
  - Archive of generated output files and directories.
  - Accessible through GitLab UI or the API.
Gitlab CI Workflow

Image source: docs.gitlab.com
Gitlab CI Backend

- Gitlab Runner:
  - Runners are computers/virtual machines where we Gitlab CI scripts get executed.
  - For specialized/large software that require configurations of our own, we make use of CERN OpenStack virtual machine.
  - For this we need to install a runner on the system and register it as a gitlab-runner for our repository.

https://docs.gitlab.com/runner/install/
https://docs.gitlab.com/runner/register/
An example CI Pipeline
An example CI Pipeline

CROME Embedded Application
- pReset
- setPCAP
- ps2ppDump
- p2psDump
- CJB Embedded Application

ROMULUSlib

GNU GCC
- arm32 gcc
- Xilinx Petalinux/xtils

x86 executables,
Shared Library,
Configuration Files

arm32 executables,
Shared Library,
Configuration Files

HDL
- Vivado IPs
- Constraints

Xilinx Vivado
- Analysis
- Synthesis
- Implementation

BitStream and
Hardware Description
File
- ptTopLevel.bit
- ptTopLevel.xsa (hdf)

Xilinx Petalinux
- Petalinux-create
- Petalinux-config
- Petalinux-build
- Petalinux-package

BOOT.bin, image ub
roofs, boot.scr, uboot,
hd41
DTB
An example CI Pipeline

```plaintext
---
# list of stages for jobs, and their order of execution
- build
- test

# This job runs in the build stage, which runs first.
stage: build
before_script:
  - yum -y install doxygen
script:
  - gcc --version
  - cd /CPU/zyq/romulus1ib/
  - make RUN_ON_PC=1
  - echo "Building ROMULUS1lib for x86 architecture."
  - export LD_LIBRARY_PATH=/CPU/zyq/romulus1ib/
  - cd .../embededlinear_userspace_app
  - make RUN_ON_PC=1
  - echo "Embedded_Application and ROMULUS1lib Compiled Successfully"
artifacts:
  paths:
  - /CPU/zyq/romulus1ib/
  - /CPU/zyq/romulus1ib/.../embededlinear_userspace_app
  expire_in: 7d
---
```
An example CI Pipeline

stages:  # list of stages for jobs, and their order of execution
  - build
  - test
build-Software:  # This job runs in the build stage, which runs first.
  stage: build
  before_script:
    - yum -y install doxygen
  script:
    - gcc --version
    - cd CPU/zyq3/sw/ROMULUSlib/
    - make RUN_ON_PC=1
    - echo "Building ROMULUSlib for x86 architecture." 
    - export LD_LIBRARY_PATH="CPU/zyq3/sw/ROMULUSlib/"
    - cd .../embedded_linux_userspace_app
    - make RUN_ON_PC=1
    - echo "Embedded_Application and ROMULUSlib Compiled Successfully"

artifacts:
  paths:
    - CPU/zyq3/sw/ROMULUSlib/
    - CPU/zyq3/sw/Embedded_linux_userspace_app
  expire_in: 7d
An example CI Pipeline

```bash
# list of stages for jobs, and their order of execution
- build
- test

build-software:  # This job runs in the build stage, which runs first.
    stage: build
    before_script:
        - yum -y install doxygen
    script:
        - gcc --version
        - cd CMU/zynq/sw/ROMULUSlib/
        - make RUN_ON_PCIE=1
        - echo "Building ROMULUSlib for x86 architecture."
        - export ID_LIBRARY_PATH=CMU/zynq/sw/ROMULUSlib/
        - cd ../../embedded_linux_userspace_app
        - make RUN_ON_PCIE=1
        - echo "Embedded_Application and ROMULUSlib Compiled Successfully"
    artifacts:
        - CMU/zynq/sw/ROMULUSlib/
        - CMU/zynq/sw/embedded_linux_userspace_app
        - expire.in: 7d

# Build Hardware job defined to run in Docker container based on Vivado 2021.2 image
build.hw:
    stage: build
    image: gitlab-registry.cern.ch/cce/docker_build/vivado:2021.2
    script:
        - echo "Creating Project"
        - cd CMU/zynq/hw/
        - make create_project
        - echo "Vivado 2021.2 project created successfully."
        - make synthesis
        - echo "Vivado 2021.2 Project SYNTHESIS completed successfully."
        - make implementation
        - echo "Vivado 2021.2 Project IMPLEMENTATION completed successfully."
        - echo "Bitstream generated successfully."
    artifacts:
        - CMU/zynq/hw/outputfiles
```

23/11/2022
Amitabh Yadav, amitabh.yadav@cern.ch
An example CI Pipeline

In a simple `.gitlab-ci.yml` file, we define:

- The commands need to run in sequence (Stages) and those that need to run in parallel (Jobs).
- The scripts that need to be run.
- `before_script`, `after_script` and `variables`.
- Artifacts: the built files that need to be saved.
- Specification of whether to run the scripts automatically or trigger manually.
- Specification on which specific branch the pipeline should be executed automatically.

```yaml
stages: # list of stages for jobs, and their order of execution
  - build
  - test

build-software: # This job runs in the build stage, which runs first.
  stage: build
  before_script:
    - yum -y install doxygen
  script:
    - gcc --version
    - cd CMPP/zyng/sw/ROMULUS1ib/
    - make RUN_ON_PC=1
    - echo "Building ROMULUS1ib for x86 architecture."
    - export ID_LIBRARY_PATH=CMPP/zyng/sw/ROMULUS1ib/
    - cd ../embedded_linux_userspace_app
    - make RUN_ON_PC=1
    - echo "Embedded_Application and ROMULUS1ib Compiled Successfully"
  artifacts:
    paths:
      - CMPP/zyng/sw/ROMULUS1ib/
      - CMPP/zyng/sw/embedded_linux_userspace_app/
    expire_in: 7d

# Build Hardware job defined to run in Docker container based on Vivado 2021.2 image.
build_hw:
  stage: build
  image: github-registry.cern.ch/cec/docker_build/vivado/2021.2
  script:
    - echo "Creating Project"
    - cd CMPP/zyng/hw/
    - make create_project
    - echo "Vivado 2021.2 project created successfully."
    - make synthesis
    - echo "Vivado 2021.2 Project SYNTHESIS completed successfully."
    - make implementation
    - echo "Vivado 2021.2 Project IMPLEMENTATION completed successfully."
    - echo "Bitstream generated successfully."
  artifacts:
    paths:
      - CMPP/zyng/hw/outputfiles
```
Docker Executor

Using image keyword executes the CI jobs in Docker containers.

The Docker executor divides the job into multiple steps:

- **Prepare**: Create and start the services.
- **Pre-job**: Clone, restore cache and download artifacts from previous stages.
- **Job**: User build. This is run on the user-provided Docker image.
- **Post-job**: Create cache, upload artifacts to GitLab.

*pre-job and post-job are run on a special docker container based on Alpine Linux.

Prebuilt docker images for Vivado, Petalinux, QuestaSim, ModelSim, Doxygen etc. are available through Gitlab Container Registry at SY-EPC-CCE’s repository[1]


---

23/11/2022
Amitabh Yadav, amitabh.yadav@cern.ch
CROME CI Pipeline
CROME CI Pipeline

merge request pipelines runs when a merge request is open for the branch.

```
stages:
  - pre-checks
  - build

workflow: # dictates pipeline behaviour
rules:
  - if: $CI_PIPELINE_SOURCE == 'merge_request_event'
```
CROME CI Pipeline: Syntax Check and Linting

```yaml
# List of stages for jobs, and their order of execution
- pre-checks
- build

# dictates pipeline behaviour
rules:
- if: $CI_PIPELINE_SOURCE == 'merge_request_event'
  then:
    stage: pre-checks
    image: gitlab-registry.cern.ch/cce/docker_build/vivado:2018.1
  always:
    stage: build
    image: gitlab-registry.cern.ch/cce/docker_build/vivado:2018.1

before_script:
  - sudo yum -y install autoconf automake
  - cd $CMAKE_BINARY_DIR
  - make -j

- check_vhdl_syntax
- check_linting
```

**before_script:** Use for initial setup of docker container and resolve missing dependencies. Upon final testing this can be packaged in the original docker image.
CROME CI Pipeline: Syntax Check and Linting

allow_failure: allows for a CI job to fail and continue executing the pipeline.

artifacts: built files and executables that are passed on to the next job.
CROME CI Pipeline: Build Stage

```
build_hw:
  stage: build
  image: gitlab-registry.cern.ch/cce/docker_build/vivado:2018.1
  allow_failure: false
  before_script:
    - sudo yum -y install autoconf automake
  script:
    - cd CMPU/zynq/hw/
    - autoreconf -i
    - mkdir -p build
    - cd build
    - ../configure --enable-triplification --enable-backend=ion --enable-sem=repair
    - make implementation
    - echo "Vivado 2018.1 Project IMPLEMENTATION completed successfully."
  artifacts:
    paths:
      - CMPU/zynq/hw/build/CMPU.hdf
```
CROME CI Pipeline: Build Stage

build_hw:
  stage: build
  image: gitlab-registry.cern.ch/cce/docker_build/vivado:2018.1
  allow_failure: false
  before_script:
    - sudo yum -y install autoconf automake
  script:
    - cd CMGPU/zyng/hw/
    - autoreconf -i
    - mkdir -p build
    - cd build
    - ../configure --enable-triplication --enable-frontend=ion --enable-sem=repair
    - make implementation
    - echo "Vivado 2018.1 Project IMPLEMENTATION completed successfully."
  artifacts:
    paths:
      - CMGPU/zyng/hw/build/CMGPU.hdf

build_sw:
  stage: build
  image: gitlab-registry.cern.ch/cce/docker_build/petalinux:2018.1
  allow_failure: false
  before_script:
    - sudo yum -y install autoconf automake
  script:
    - source /opt/Xilinx/petalinux/settings.sh
    - cd CMGPU/zyng/sw/cromeSuite
    - autoreconf -i
    - mkdir -p build
    - cd build
    - ../configure --host arm-linux-gnueabihf
    - make -j8
    - echo "Embedded_Linux_Userspace Application and ROMULUSlib Compiled Successfully"
  artifacts:
    paths:
      - CMGPU/zyng/sw/ROMULUSLib/LibROMULUS.*
      - CMGPU/zyng/sw/cromeSuite/build/cromeApp
      - CMGPU/zyng/sw/cromeSuite/build/preset
      - CMGPU/zyng/sw/cromeSuite/build/setPCAP
      - CMGPU/zyng/sw/cromeSuite/apps/cromeApp/local_parameters.dat
      - CMGPU/zyng/sw/cromeSuite/apps/cromeApp/romulus_parameters.dat
      - CMGPU/zyng/sw/Embedded_Linux_Userspace_app/doc
Key Features

needs is used needs to execute jobs out-of-order.
Key Features

**needs** is used to execute jobs out-of-order.

**variables** can be defined in `.gitlab-ci.yml` or in the project settings. They can be made global or local to a job and are used in similar way as shell variables.
Key Features

**needs** is used needs to execute jobs out-of-order.

**variables** can be defined in `.gitlab-ci.yml` or in the project settings. They can be made global or local to a job and are used in a similar way as shell variables.

**when** is used within a job to specify if the job needs manual intervention to start.

- Stages such as `formal_verification` for license availability reasons.
- Linux image build.
# CROME CI Pipeline: Artifacts

<table>
<thead>
<tr>
<th>Status</th>
<th>Pipeline Description</th>
<th>Trigger Details</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>passed</td>
<td>Dummy commit</td>
<td>417bb2e06</td>
<td>✔️ ✔️</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>passed</td>
<td>Dummy commit</td>
<td>417bb2e06</td>
<td>✔️ ✔️</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>passed</td>
<td>cosim/README.md, updated prepare_sdcard, minor impr...</td>
<td></td>
<td>✔️ ✔️</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>failed</td>
<td>swc: WIP change of file structure</td>
<td></td>
<td>✔️ ✗ ✔️</td>
</tr>
</tbody>
</table>

**Download artifacts**

- build_swarchive
- build_swarchive
- check_linting:archive
- check_linting:junit
- check_linting:codequality
Building Linux Images through Gitlab CI

- Currently testing: Building of Embedded Linux Image along with bit bake bootscript applications.

Amitabh Yadav, amitabh.yadav@cern.ch
Building Linux Images through Gitlab CI

- Currently testing: Building of Embedded Linux Image along with bit bake bootscript applications.
- kernel and hw configuration through --silentconfig option on Xilinx/Petalinux 2021.2

**build_linux:**
```
stage: build.cronix
needs: []
image: gitlab-registry.cern.ch/cce/docker_build/petalinux:2021.2
script:
  - echo "Run cronix21 Build Petalinux Image"
  - cd CPU/zyg/zynq
  - petalinux-create --type project --template zynq --name cronix21
  - cd cronix21
  - petalinux-config --get-hw-description /mdf/ --silentconfig
  - petalinux-config -c kernel --silentconfig
  - petalinux-build
when: manual
allow_failure: true
rules:
  - if: "$CI_COMMIT_BRANCH == "gitlabci2021.2"
artifacts:
  when: on_success
  paths:
    - images/linux/
  expire_in: 7d
```

*in development*
Conclusion

- CI can be efficiently adopted for heterogeneous development for SoCs.
- We have currently deployed Gitlab CI in development branch is currently used for HDL linting using vsg docker image.
- Gitlab CI is proving elemental during migration of our Xilinx HDL codebase + IPs from version 2018.1 to 2021.2
- Has helped in resolving dependency issues through GNU Autotools and Docker containers to execute CI jobs.

**In the works is Gitlab CI for:**
- Formal Verification scripts to be added to the CI pipelines to be executed automatically.
- Continuous Deployment through gitlab-runner through ssh into devices.