Setting up development infrastructure for Petalinux projects and Zynq MPSoC/RFSoc based hardware utilizing continuous integration and deployment techniques

Michal Husejko
CERN SoC Interest Group Meeting
https://indico.cern.ch/event/1208190/
2022 NOVEMBER 23
Abstract

• In this tutorial we demonstrate how to setup basic Petalinux development and continuous integration and deployment (CI/CD) infrastructure for MPSoC/RFSoC based projects.

• We start by showing how to organize a workstation so that it could be used at the same time for interactive and batch (gitlab CI based) Petalinux compilation jobs.

• In the next step we extend the setup with an example RFSoC board to show how to perform continuous deployment of Petalinux images directly to the hardware utilizing network boot and how to execute and organize basic tests utilizing features of gitlab CI server.

• Tutorial relies on standard components which can be enabled in Petalinux/yocto (like docker and kubernetes) and provides low level information when necessary so that attendees could rather easily reuse all or part of the demonstrated content on their own premises.

i.e. in this presentation we mainly focus on Petalinux CI/CD
Agenda in pictures

- **Workstation (Vivado & Petalinux)**
  - Vivado
  - Petalinux
  - TFTP-NFS-JTAG
  - test1-hash
  - PowerOff

- **Network**
  - DHCP Srv.
  - TFTP Srv.
  - NFS exp.
  - Terminal/GUI (Interactive Work)
  - gitlab runner (CI/batch jobs)

- **Physical Connections**
  - PS eth
  - JTAG/UART over USB
  - USB

- **Optional Components**
  - NAS (optional)
  - gitlab server
Agenda in words

• Initial requirements.
• Setting up a workstation for Zynq development and (batch) CI jobs execution.
• Setting up Petalinux CI build.
• Basic Petalinux CI flow.
• Continuous Deployment to hardware (Xilinx XUP RFSoC 4x2 board).
• Gitlab CI support of junit reports, and coloring of merge requests
Gitlab CI – first step
What is required to enable gitlab CI jobs execution?

• Gitlab repository:
  • https://gitlab.cern.ch/
  • gitlab server at your Home Institute
  • home installation
  • ...

• Gitlab Runner(s) attached to your project (from gitlab web UI: Settings -> CI/CD -> Runners). Your own (project specific) or shared.

• .gitlab-ci.yml file controls what is happening on/with your runners when events related to your repository are occurring (push, merge, web [Run Pipeline], etc.)

• In this tutorial we will use a private runner - workstation connected to a self hosted gitlab server (VirtualBox) – just for fun and learning purposes.
Tutorial folder structure

• In this tutorial we utilize github RFSoC 4x2 repository which we extend with our own recipes, build scripts and gitlab CI control files.
  • Original repo available here: https://github.com/RealDigitalOrg/RFSoC4x2-BSP
  • The above repository contains Petalinux BSP file, but don’t utilize it – instead we build everything with our own recipes (for learning purposes).
• Above repo extended with:
  • Zynq PL code:
    • Makefile to execute Vivado.
  • Petalinux:
    • Recipes and scripts.
  • CI flow related
    • Scripts.
    • Tools.
Our gitlab CI control file (gitlab-ci.yml)

---

**Stages:**
- vivado
- petalinux
- prvron
- prog
- test
- pwroff
- tftp-cfg
- tftp-prog

---

**Vivado:**
```yaml
stages: vivado
  stage: vivado
  script:
    - bash
    - echo "$SRC_RIG_BSP"
    - /opt/xilinx/20202/Vitis/2020.2/settings64.sh
    - mkdir "$ART_STORAGE"
    - cd work
    - make xsa
    tags:
      - VIVADO
  rules:
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./basic."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./docker."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./sys."
```

**Petalinux:**
```yaml
stages: petalinux
  stage: petalinux
  script:
    - bash
    - echo "$SRC_RIG_BSP"
    - /opt/xilinx/20202/Vitis/2020.2/settings64.sh
    - make -j $NUM_JOBS
    - make -j $NUM_JOBS
    tags:
      - PETALINUX
  rules:
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./basic."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./docker."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./sys."
```

**Pvron:**
```yaml
stages: prvron
  stage: prvron
  script:
    - bash
    - echo "$SRC_RIG_BSP"
    - /opt/xilinx/20202/Vitis/2020.2/settings64.sh
    - make -j $NUM_JOBS
    - make -j $NUM_JOBS
    tags:
      - PRVRON
  rules:
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./basic."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./docker."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./sys."
```

**Prog:**
```yaml
stages: prog
  stage: prog
  script:
    - bash
    - echo "$SRC_RIG_BSP"
    - /opt/xilinx/20202/Vitis/2020.2/settings64.sh
    - make -j $NUM_JOBS
    - make -j $NUM_JOBS
    tags:
      - PROG
  rules:
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./basic."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./docker."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./sys."
```

**TFTP-NFS-JTAG:**
```yaml
stages: tftp-jtag
  stage: tftp-jtag
  script:
    - bash
    - echo "$SRC_RIG_BSP"
    - /opt/xilinx/20202/Vitis/2020.2/settings64.sh
    - make -j $NUM_JOBS
    - make -j $NUM_JOBS
    tags:
      - TFTP-JTAG
  rules:
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./basic."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./docker."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./sys."
```

**PowerOff:**
```yaml
stages: poweroff
  stage: poweroff
  script:
    - bash
    - echo "$SRC_RIG_BSP"
    - /opt/xilinx/20202/Vitis/2020.2/settings64.sh
    - make -j $NUM_JOBS
    - make -j $NUM_JOBS
    tags:
      - POWEROFF
  rules:
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./basic."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./docker."
    - if "$SCI_PIPELINE_SOURCE" == "web" & & "$SRC_RIG_BSP" == "./sys."
```

---

Run Pipeline
1. Navigate to Pipelines view of your repository (CI/CD -> Pipelines) and click “Run Pipeline”

2. Launch a “web” pipeline (with default values from branch - freshest branch and ZYNQ_BSP=k8s) by clicking “Run Pipeline”.
Setting up a workstation for Zynq development and gitlab CI jobs execution.
Setting up a workstation for Zynq development and CI jobs execution

• Overview of HW and SW used for this tutorial
• Installing Vivado and Petalinux 2020.2
• Setting up a „Service account”
• A not so „Basic” gitlab runner installation
• RFSoC net boot services: DHCP, TFTP, and NFS
HW and SW used for testing

• Intel (Skull Canyon) NUC workstation:
  • Will be attached to our gitlab repository as a project specific runner.
  • Used for local compilations and gitlab-runner (shell executor), OS: Ubuntu LTS 18.04.6, 32 GB RAM, Intel i7-6770HQ (4c/8t).
  • Configured with NFS export, TFTD, and DHCP servers.
  • USB dongle with JTAG and internal network connected to RFSoC 4x2 board.
  • Petalinux 2020.2 (no BSP file used).

• RFSoC 4x2 board:
  • Configured for JTAG boot, no SD card used/inserted.
  • Powered from power outlet controlled over Ethernet.
Installing Vivado and Petalinux 2020.2

• Instructions provided by Xilinx:

• Don’t forget to install dependency packages listed by Petalinux:
  • https://www.xilinx.com/support/answers/72950.html
  • https://www.xilinx.com/support/answers/73296.html

• Configure licensing if running with non Webpack devices:
  • Webpack Features: https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html#webpack
  • Webpack Devices: https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html#architecture

• Tutorial references tools installed into /opt/Xilinx/v2020 folder
Setting up a „Service account” on the workstation

• We will add a „Service account” on our workstation.
  • Local home directory (/home/soc-usr)
  • This account will be used to run CI jobs, place TFTP and NFS images into service folders, and communicate with hardware over password less ssh.

• add soc-usr to sudoers
  • enable passwordless sudo (visudo - > soc-usr ALL=(ALL) NOPASSWD: ALL)
  • we will use sudo access to unpack rootfs and to be able to re-start TFTP and NFS services.
  • You could limit sudo access only to the commands which are necessary in your scripts.

• generate private+public key (ssh-keygen -t rsa -b 4096)
  • We will inject public key into Petalinux rootfs so that we can easily communicate with RFSoC board and to execute scripts remotely.
Default gitlab-runner installation

• Instructions provided on this website:
  • https://docs.gitlab.com/runner/install/linux-repository.html

• By default – installs gitlab-runner into default folder (/home/gitlab-runner), uses single config file for all registered projects, and executes gitlab CI jobs with gitlab-runner account (also added to sudoers).

• We will replace default gitlab-runner account with our „service account” and register multiple services each with different control files.
  • Unique control files per each service – control of jobs concurrency (shell executor)
  • Multiple services – each with its own control file, and its own control of work/execution folder (SATA vs. NVMe Gen4, RAID vs. splitting storage traffic).
  • NOTE: You can use multiple services pattern for sharing a single computing node among many users, having some control over QoS (concurrency) and accounting (unique user names)
Not so basic gitlab runner installation

• We will create 4 gitlab-runner services:
  • gitlab-runner-vivado (concurrency 4 jobs).
  • gitlab-runner-petalinux (concurrency 1 job).
  • gitlab-runner-sim (concurrency 8 jobs).
  • gitlab-runner-prog (concurrency 1 job).

• Each service to be executed with our service account „soc-usr”.
  • Password less sudo, private+public ssh key. Temporary storage on NVMe drive.
  • Example:
    • sudo gitlab-runner install -n gitlab-runner-petalinux -d "/opt/gitlab-ci-tmp/petalinux" -c "/home/soc-usr/.gitlab-runner/config-petalinux.toml" -u soc-usr
    • sudo service gitlab-runner-petalinux restart

• We register runners with our gitlab repository as shell executors:
  • Example:
    • gitlab-runner register -c /home/soc-usr/.gitlab-runner/config-petalinux.toml
Our workstation in our gitlab repository

• Our single workstation with 4 gitlab-runner services.
• Each service with:
  • More meaningful account
  • Behind a single tag
  • With job concurrency control and
  • Full storage location control
DHCP for internal network

• The DHCP server will serve internal network 10.5.5.x (Eth from USB dongle, directly connected to RFSoC board)
• sudo service isc-dhcp-server start
TFTP to serve Linux kernel

• sudo service tftpd-hpa start
• Service tied to dongle eth interface (/etc/default/tftpd-hpa)
• Folder to store Petalinux generated Image:
  • /tftpboot
  • soc-user has write permissions to that folder - used by CI to deploy images.
NFS export to serve Linux root file system

• NFS export:
  /tftpboot/nfsroot *(rw,sync,no_root_squash,no_subtree_check,crossmnt)
• sudo service nfs-kernel-server start
• Petalinux bootargs:
  • console=ttyPS0,115200n8 earlyprintk ip=hcp root=/dev/nfs rootfstype=nfs
    nfsroot=10.5.5.1:/tftpboot/nfsroot,port=2049,nfsvers=3,tcp rw
  • soc-user uses sudo permissions to unpack Petalinux generated rootfs.tar.gz into /tftpboot/nfsroot

Kernel command line: console=ttyPS0,115200n8 earlyprintk ip=hcp root=/dev/nfs rootfstype=nfs nfsroot=10.5.5.1:/tftpboot/nfsroot,port=2049,nfsvers=3,tcp rw

VFS: Mounted root (nfs filesystem) on device 0:12.
Forcing NFS server to work with specific version.

- sudo vim /etc/default/nfs-kernel-server
  - Was: # RPCNFSDCOUNT=8
  - Is: RPCNFSDCOUNT="8 --no-nfs-version 4"

- sudo cat /proc/fs/nfsd/versions
  -2 +3 -4 -4.0 -4.1 -4.2
Zynq PL Vivado CI flow

Not so important for this presentation
„Vivado” stage
Zynq PL related elements

• gitlab-ci.yml (Vivado stage)
• Makefile (Vivado project mode flow commands)
• Original github Zynq PL design kept as a Vivado/IPI exported design.
• We just add a makefile:
  • make xsa
make xsa

• ... but this tutorial utilizes XSA (with bit file) available from the github repo.
Basic Petalinux CI flow
„Petalinux” stage
We have three different “BSP” available

- **basic:**
  - Really basic BSP with TFTP+NFS+JTAG boot.

- **docker:**
  - Basic + docker

- **k8s:**
  - Basic+docker+kubernetes

Selection of the BSP to build is done using ZYNQ_BSP variable.

- Example to build k8s:
  - Run pipeline with ZYNQ_BSP=k8s
1. Setup environment

2. Use the XSA available in the repo.

3. Create basic petalinux project (no BSP used)

4. Copy pre-generated project-spec files (kernel and rootfs configs)
   - generation explained on next slides

5. Adjust settings using perl script and echo commands:
   - inject branch name and short githash as a RFSoC hostname
   - add download/sstate/sstate_local cache repositories

6. Apply recipe to add soc-usr local account, inject public key, and
   create custom application which we will use during CD/test phase.

7. Build Petalinux project

8. Store artifacts
More details on simple petalinux flow

- Get XSA file from storage
- Create basic Zynq(-7000) project using “zynqMP” template – no BSP used
  - `petalinux-create --type project --template zynqMP --name peta20202`
- Ingest XSA file into basic project
  - `petalinux-config --project peta20202 --get-hw-description=./..../sw/design_1_wrapper.xsa --silentconfig`
- Configure newly created project to match your (CI) needs:
  - details on the next slides – 3 different methods to configure Petalinux from lowest to highest complexity
- Apply new config:
  - `petalinux-config --project peta20202 --silentconfig`
- Build the project:
  - `petalinux-build --project peta20202`
- Store artifacts in the storage (to be used by programming stage)
  - In principle push whole ./peta20202/images/linux repo to the storage.
Initial project-spec configs – generate on your workstation using menuconfig then push to git and use as a baseline

- DTG Settings -> Kernel Bootargs -> generate boot args automatically []
  - Disable
- DTG Settings -> Kernel Bootargs -> user set kernel bootargs
  - earlycon console=ttys0,115200n8 clk_ignore_unused earlyprintk rootwait root=/dev/nfs rw
  - nfsroot=10.5.5.1:/tftpboot/nfsroot, port=2049, nfsvers=3, tcp ip= dhcp
- Image Packaging Configuration -> Root filesystem type -> (NFS)
  - Chose NFS
- Enable TFTP boot
- Image Packaging Configuration -> Location of NFS root directory (/tftpboot/nfsroot)
- Image Packaging Configuration -> tftpboot directory (/tftpboot)
- Firmware Version Configuration -> (MK_CONFIG_SUBSYSTEM_HOSTNAME) Hostname
- Firmware Version Configuration -> (MK_BSP_NAME) Product name
- Firmware Version Configuration -> (1.00) Firmware Version

- Store config in repository and then apply within CI flow as a base line – from our yaml file:
  - cp ../petalinux_bsp/${MK_BSP_NAME}/sw/project-spec/configs/config peta20202/project-spec/configs/config
Adjust settings using perl script and echo commands

- Two examples below:

  `perl -i -pe 's/\bMK_CONFIG_SUBSYSTEM_HOSTNAME\b/${MKEXPECTED_HOSTNAME}/g' ./peta20202/project-spec/configs/config`

  `echo "DL_DIR = "/home/soc-usr/ycache/v20202/downloads" >> peta20202/project-spec/meta-user/conf/petalinuxbsp.conf`
Firmware Version Configuration ->
(MK_CONFIG_SUBSYSTEM_HOSTNAME) Hostname (perl)

• Hostname name used to help visualize traceability – inject branch name and git sha into it (replace MK_CONFIG_SUBSYSTEM_HOSTNAME project name)
Apply recipe to add soc-usr local account, inject public key, and create custom application (1/3)

• mkdir -p ./peta/project-spec/meta-user/recipes-core/images/
• cp ${MCI_FLOW_ROOT_DIR}/recipes/recipes-core/images/petalinux-user-image.bbappend ./peta7/project-spec/meta-user/recipes-core/images/

# petalinux-image-minimal.bbappend content

inherit extrausers

EXTRA_USERS_PARAMS = "\n  usermod -P * root; \n  useradd -P * soc-usr; \n  usermod -aG docker soc-usr; \n  ""
Apply recipe to add soc-user local account, inject public key, and create custom application (2/3)

- petalinux-create --project peta20202 --type apps --template install --name mhcicd --enable --force
- rm ./peta/project-spec/meta-user/recipes-apps/mhcicd/files/*
- rm ./peta/project-spec/meta-user/recipes-apps/mhcicd/mhcicd.bb
- cp ${MCI_FLOW_ROOT_DIR}/recipes/recipes-apps/mhcicd/files/id_rsa.pub ./peta/project-spec/meta-user/recipes-apps/mhcicd/files
- cp ${MCI_FLOW_ROOT_DIR}/recipes/recipes-apps/mhcicd/mhcicd.bb ./peta/project-spec/meta-user/recipes-apps/mhcicd
Apply recipe to add soc-usr local account, inject public key, and create custom application (2/3)

SUMMARY = "Simple mhcicd application"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
SRC_URI = "file://id_rsa.pub"
S = "${WORKDIR}"
USER="soc-usr"
do_install() {
    install -d ${D}/home/${USER}/.ssh/
    install -m 0755 ${S}/id_rsa.pub ${D}/home/${USER}/.ssh/
    install -m 0755 ${S}/id_rsa.pub ${D}/home/${USER}/.ssh/authorized_keys
FILES_${PN} += "/home/${USER}/.ssh/*"
Apply recipe to add soc-usr local account, inject public key, and create custom application (3/3)

- petalinux-create --project peta --type apps --template install --name trojan --enable -force
- rm ./peta/project-spec/meta-user/recipes-apps/trojan/files/*
- cp ${MCI_FLOW_ROOT_DIR}/recipes/recipes-apps/trojan/files/trojan ./peta/project-spec/meta-user/recipes-apps/trojan/files

```
#!/bin/sh

echo "been here. Tony Halik"
```
Enable docker and kubernetes (k8s)

- CONFIG_YOCTO_MACHINE_NAME="docker-zynqmp-generic"
Speed up the Petalinux compilation

- echo "DL_DIR = "/home/soc-usr/ycache/v20202/downloads" >> peta20202/project-spec/meta-user/conf/petalinuxbsp.conf

- echo "SOURCE_MIRROR_URL = "file:///home/soc-usr/ycache/v20202/downloads" >> peta20202/project-spec/meta-user/conf/petalinuxbsp.conf

- echo "SSTATE_DIR = "/home/soc-usr/ycache/v20202/sstate_local" >> peta20202/project-spec/meta-user/conf/petalinuxbsp.conf

```
# Local sstate feeds settings
#
CONFIG_YOCTO_LOCAL_SSTATE FEEDS_URL="/home/soc-usr/ycache/v20202/sstate_aarch64_2020.2/aarch64"
CONFIG_YOCTO_NETWORK_SSTATE FEEDS=y
```
Continuous Deployment to hardware
Power outlet controlled over Ethernet

- **Netio PowerPDU 4PS**
  - Around 220 CHF on galaxus.ch
- Commands send using curl
  - More sophisticated APIs available.
  - JSON and status checking would be better.
- Command constructed using gitlab CI
  Variables (Settings->CI/CD->Variables)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETIO_IPADDR</td>
<td>************</td>
</tr>
<tr>
<td>NETIO_OUTPUT</td>
<td>************</td>
</tr>
<tr>
<td>NETIO_PASS</td>
<td>************</td>
</tr>
</tbody>
</table>
AdAstra:

stage: pwron

needs: ["Petalinux"]

script:
- bash
  - curl "${NETIO_IPADDR}/netio.cgi?pass=${NETIO_PASS}&${NETIO_OUTPUT}=0"
  - sleep 5s
  - curl "${NETIO_IPADDR}/netio.cgi?pass=${NETIO_PASS}&${NETIO_OUTPUT}=1"
  - sleep 10s
  - sudo service isc-dhcp-server restart
  - sudo service nfs-kernel-server restart
  - sudo service tftpd-hpa restart
  - sudo service isc-dhcp-server status
  - sudo service nfs-kernel-server status
  - sudo service tftpd-hpa status

tags:
- PROG

rules:
- if: "${CI_PIPELINE_SOURCE} == "web"
  when: manual
Programming flow (1/4)

- Executed automatically after power ON.
- Extract content of ./peta20202/images/linux from storage
- Populate /tftpboot and /tftpboot/nfsroot with content from above
  - Image.ub -> /tftpboot
  - rootfs -> unpack to /tftpboot/nfsroot
Deploy to the DANGER-ZONE

TFTP-NFS-JTAG:

- stage: prog
- needs: ["AdAstra"]

```
script:
  - bash
  - . /opt/xilinx/v20202/petalinux/settings.sh
  - sudo /tftpboot/clean.sh
  - tree -L 2 /tftpboot
  - cd ${ART_STORAGE}/peta20202/images/linux
  - cp -R Image system.dtb rootfs.tar.gz pxelinux.cfg/ /tftpboot/
  - sudo /tftpboot/fillnfs.sh
  - tree -L 2 /tftpboot
  - cd ${ART_STORAGE}/peta20202/images/linux
  - petalinux-boot --jtag --uboot --fpga --bitstream system.bit
```

environment:

- name: DANGER-ZONE
- tags:
  - PROG
rules:

- if: "${CI_PIPELINE_SOURCE} == "web"

# when: manual
Programming flow (2/4)

• Remaining necessary contents of the ./peta20202/images/linux pushed over jtag:
  • `petalinux-boot --jtag --uboot --fpga --bitstream system.bit`
Programming flow (3/4)

- Push button image redeployment with full gitlab hash traceability
Programming flow (4/4)

• Push button image redeployment with full gitlab hash traceability
Basic testing
Four example tests (trivial)

- Check if the git hash injected into petalinux images matches pipeline commit hash
  - Parse result of “hostname” command (we injected hostname and short git hash into ./peta20202/project-spec/configs/config file at build time).
- Execute basic script and check if the returned value matches expected response.
  - Parse result returned by our “trojan” command (we can adjust the message to inject errors: modify the file, commit && push, and observe test passing/failing).
- Check docker version and compare against expected value.
- Display kubectl version.
- All tests executed through password less ssh.
test1-hash:
  stage: test
  needs: 
    - [TFTP-NFS-STAT]
  script:
    - bash
    - export RESP=$(ssh -q -o StrictHostChecking=no -o UserKnownHostsFile=/dev/null 10.5.5.2 'hostname')
    - echo $RESP
    - export TST1=$(CI_COMMIT_REF_SLUG)
    - echo $TST1
    - export TST2=$(CI_COMMIT_SHA:0:16)
    - echo $TST2
    - export EXPECTED=$TST1-$TST2
    - echo $EXPECTED
    - if [[ "$RESP" =~ "$EXPECTED" ]]; then echo "Test OK"; else echo "Test NOT passed" && exit 1; fi
  rules:
    - if: "SCM_PIPEDLINE_SOURCE == "web""
      allow_failure: true

$ export RESP=$(ssh -q -o StrictHostChecking=no -o UserKnownHostsFile=/dev/null 10.5.5.3 'hostname ')
$ echo $RESP
f600662-cf6ba9f0
$ export TST1="f600662"
$ echo $TST1
f600662
$ export TST2=$(CI_COMMIT_SHA:0:16)
$ echo $TST2
cf6ba9f0
$ export EXPECTED=$TST1-$TST2
$ echo $EXPECTED
f600662-cf6ba9f0
$ if [[ "$RESP" =~ "$EXPECTED" ]]; then echo "SUCCESS" > status.log; cat status.log; else echo "FAILER" > status.log; cat status.log; exit 1; fi
SUCCESS
Job succeeded
from our recipes/recipes-apps/trojan/files/trojan
Lets inject some problems – Tony Montana back in Town!

• **Modify** `recipes/recipes-apps/trojan/files/trojan`
• Commit && rebuild
Gitlab CI test tab

- Lets use gitlab CI server backend to organize our testing reports.
- We will use junit reporting supported by gitlab CI.
- Each test (we have 4 of them) generates report.xml (in junit format).
- Junit report stored as an artifact – all per job reports combined into a single table.
Combined test report and error details
Fix the tst2
Gitlab environments – tracability of deployments to the DANGER-ZONE
3 times a charm – fix the errors
Gitlab Environments – tracing deployments

<table>
<thead>
<tr>
<th>Status</th>
<th>ID</th>
<th>Trigger</th>
<th>Commit</th>
<th>Job</th>
<th>Created</th>
<th>Deployed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>#23</td>
<td><code>mih-dev/002-g</code></td>
<td>7e2910ad</td>
<td>TFTP-NFS-JTA...</td>
<td>18 minutes ago</td>
<td>2 minutes ago</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fix the docker version</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUCCESS</td>
<td>#22</td>
<td><code>mih-dev/002-g</code></td>
<td>3eb24d31</td>
<td>TFTP-NFS-JTA...</td>
<td>42 minutes ago</td>
<td>26 minutes ago</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Attempt to fix the error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUCCESS</td>
<td>#21</td>
<td><code>mih-dev/002-g</code></td>
<td>e9d77487</td>
<td>TFTP-NFS-JTA...</td>
<td>50 minutes ago</td>
<td>49 minutes ago</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>docker test should work</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Utilize test reports on merge requests
Next steps

• Clean the code and release it on gitlab.cern.ch
• Extend tutorial with a k3s cluster built out of the workstation (primary controller/tainted) and the dev kit (computing node).