# Common infrastructure F/W for Phase 2 hardware

***Alessandro Thea**, Tom Williams, Kristian Harder, Raghunandan Shukla, Greg Iles, Dave Newbold, Andy Rose*

*Workshop on CMS Tracker BackEnd Systems & DAQ for Phase-2*
*12 November 2018*

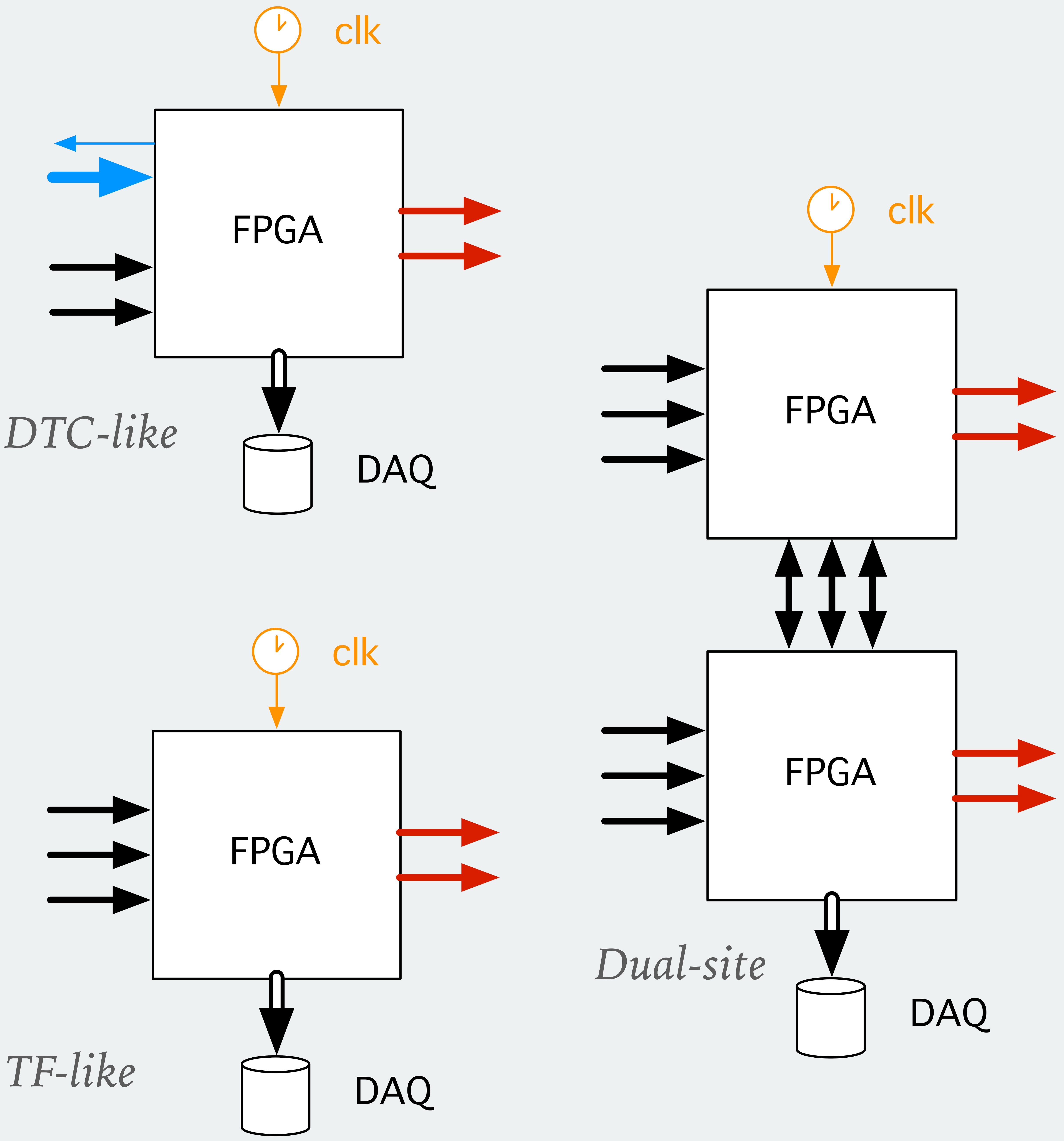
# INTRODUCTION

- Infrastructural firmware
  - Motivation for common implementation
  - Design considerations
  - EMP framework
- Firmware build tool
  - Motivation
  - IPBB
    - Dependency files
    - Vivado workflow
- Automated builds & tests

# INFRASTRUCTURAL FIRMWARE

# Context

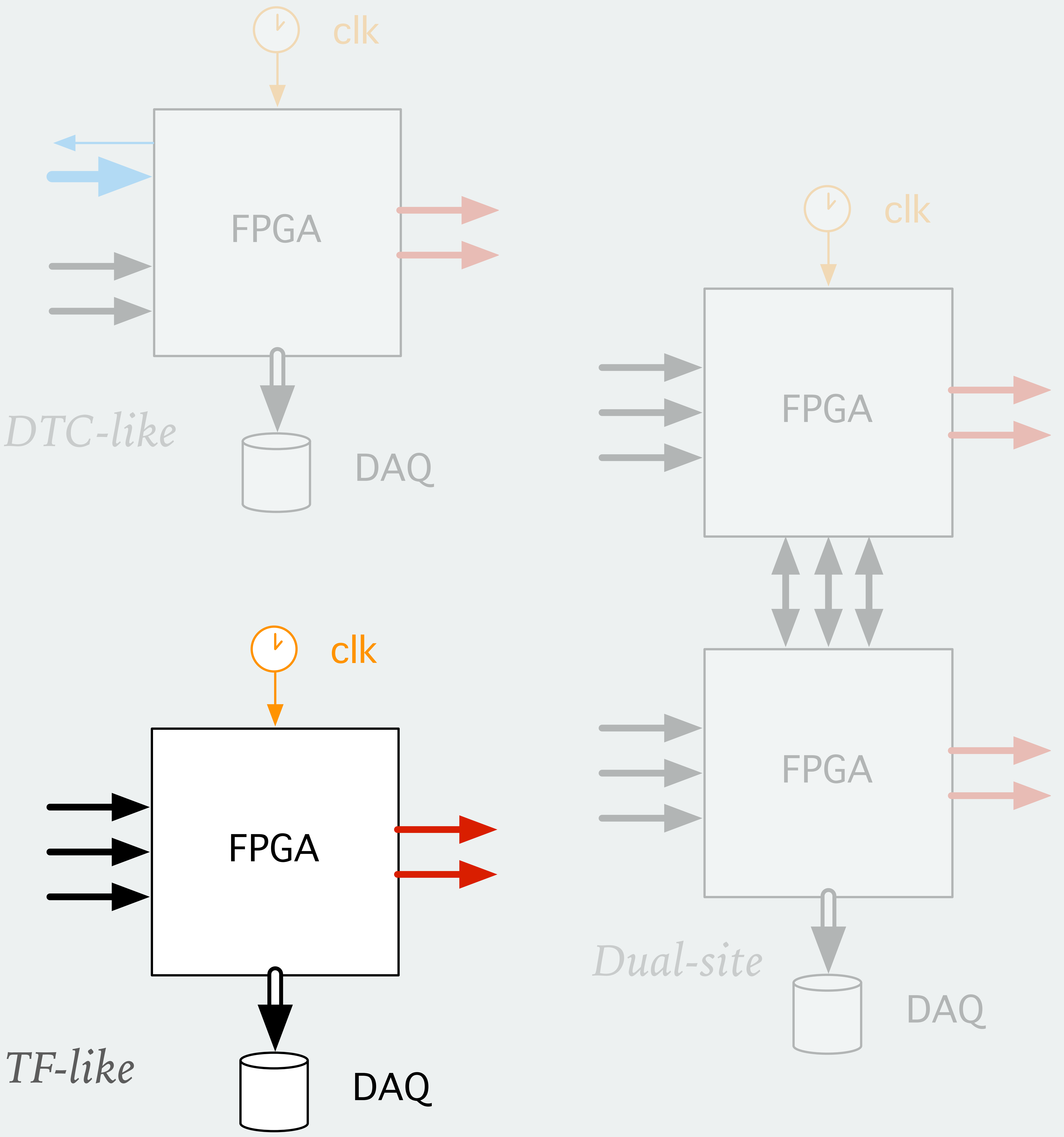
- Phase-2: replacement of most of the CMS backend electronics
  - Both within Tracker, L1-trigger, and other systems
  - Range of chips: KU(+), VU(+), …
- Underlying functionality: Many common requirements
  - Slow controls, TTC and clocking interface, Links (MGT-based, LpGBT, …), Readout & DAQ interface
- Lessons from phase-0 and phase-1 L1 trigger Upgrade
  - Writing firmware is hard
  - Debugging firmware is harder (significantly more so at P5)
- A CMS-wide common firmware approach has the potential to significantly reduce time and effort in commissioning
  - Solve a problem once, test the solution to death, use it everywhere




# Common architectures (1)

(Some of the) Repeating back-end electronics pattens:

- DTC-like
  - FE control and trigger optical links
  - TP links to L1-trigger
  - DAQ - Bandwidth intense and large fluctuations
- TF-like
  - Large optical I/O
  - DAQ - Bandwidth intense
  - Sophisticated algorithms
- One or many FPGA sites
  - intra-FPGA links




# Common architectures (2)

(Some of the) Repeating back-end electronics pattens:

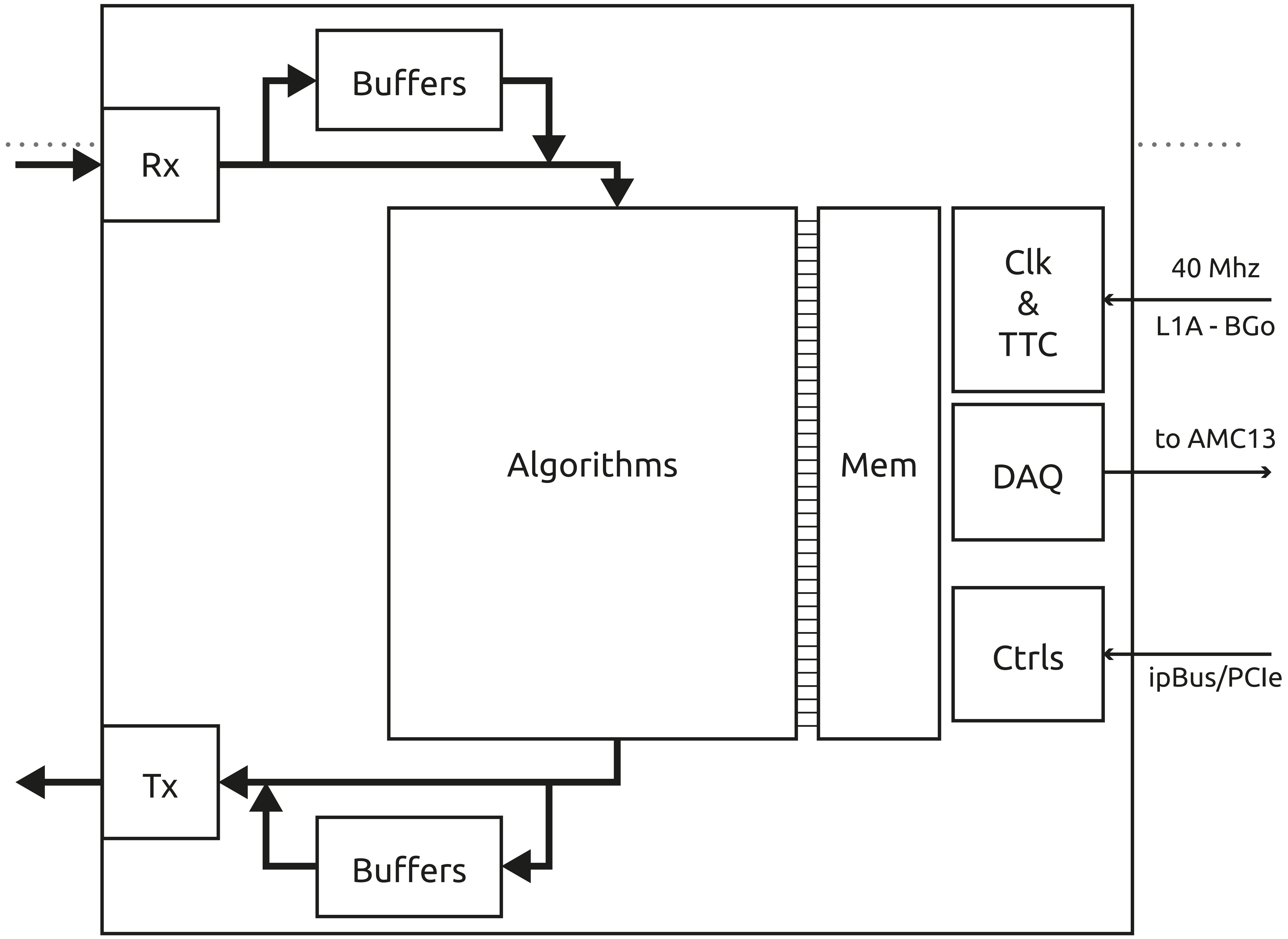
- Many common requirements
- Different mixture of key elements

Track-Finder/L1 Processor architecture

- Ideal for framework development
- Relatively self-contained
  - uniform I/O optical links, chaining and loopback possible
- Immediately applicable for algorithm development
- Builds on the L1-trigger Phase-1 experience
-

# TP Design considerations

- By cleanly separating 'physics algorithm' (e.g. particle reconstruction & ID) from infrastructural elements, e.g. …

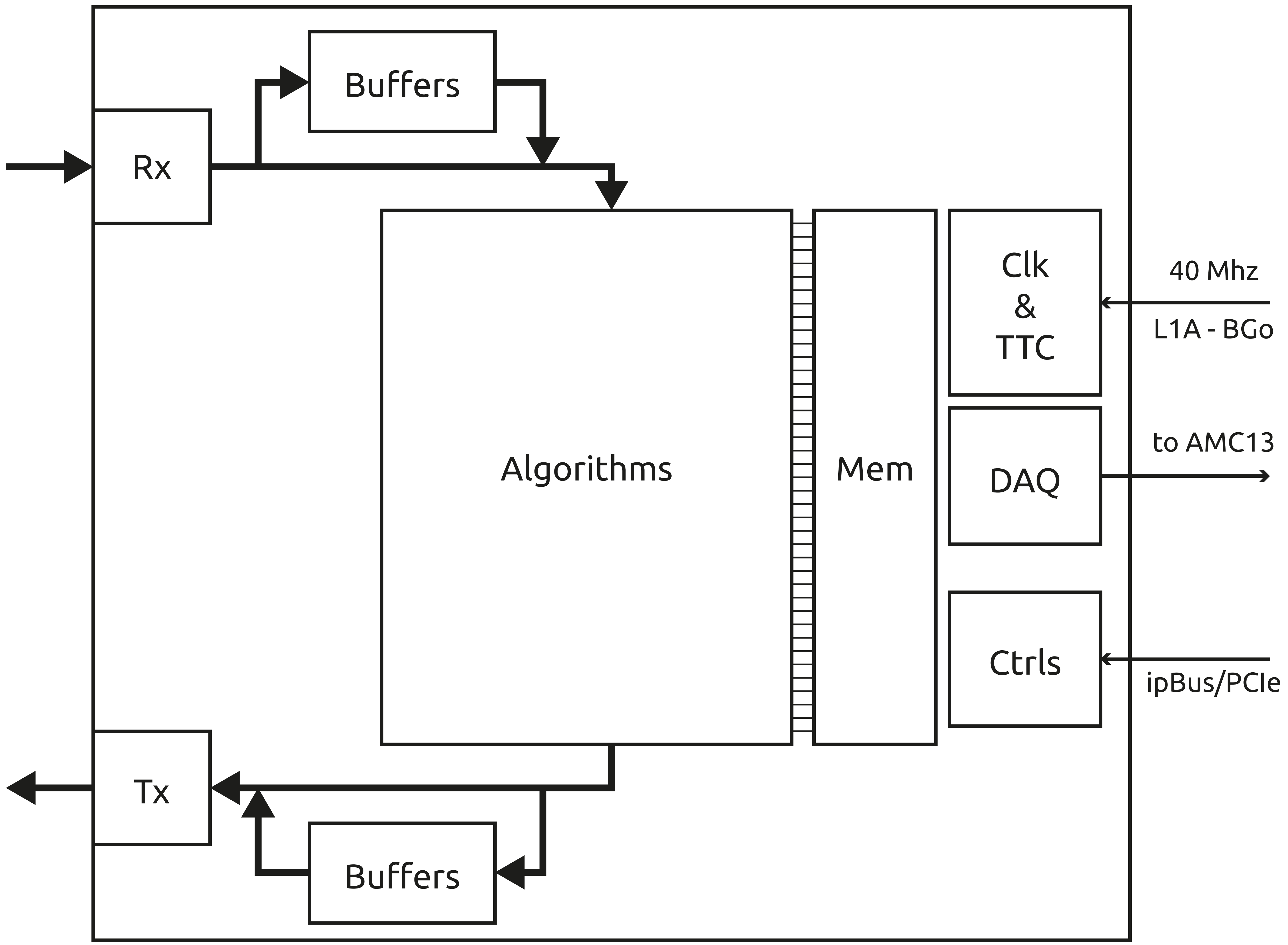



- … can re-use same implementation of in frastructure with wide range of 'physics algorithm' firmware
  - i.e. TTC, clocking; playback/capture/latency buffers, readout, links
  - **Avoid re-inventing** — and **re-debugging** — the wheel
    - Ensuring **reproducible behaviour** on all platforms, **at all stages**

# TP Design considerations (2)

- By using device-agnostic interfaces …
    - Between algorithm & infrastructure (e.g. array of rx/tx data bus)
    - Between different infrastructural elements
  - **With configuration parameterised by build-time constants**
    - E.g. **multiplicity & throughput of I/O channels**
    - Build-time constants = generics, or constants in VHDL packages

- … can minimise the amount of source code that must be changed between
  - Different packages
  - Different FPGAs
    - Even different series, e.g. Ultrascale & Ultrascale+
  - Different boards




# EMP FRAMEWORK (1)

**EMP** = **E**xtensible, **m**odular data **p**rocessor framework (*working title*)

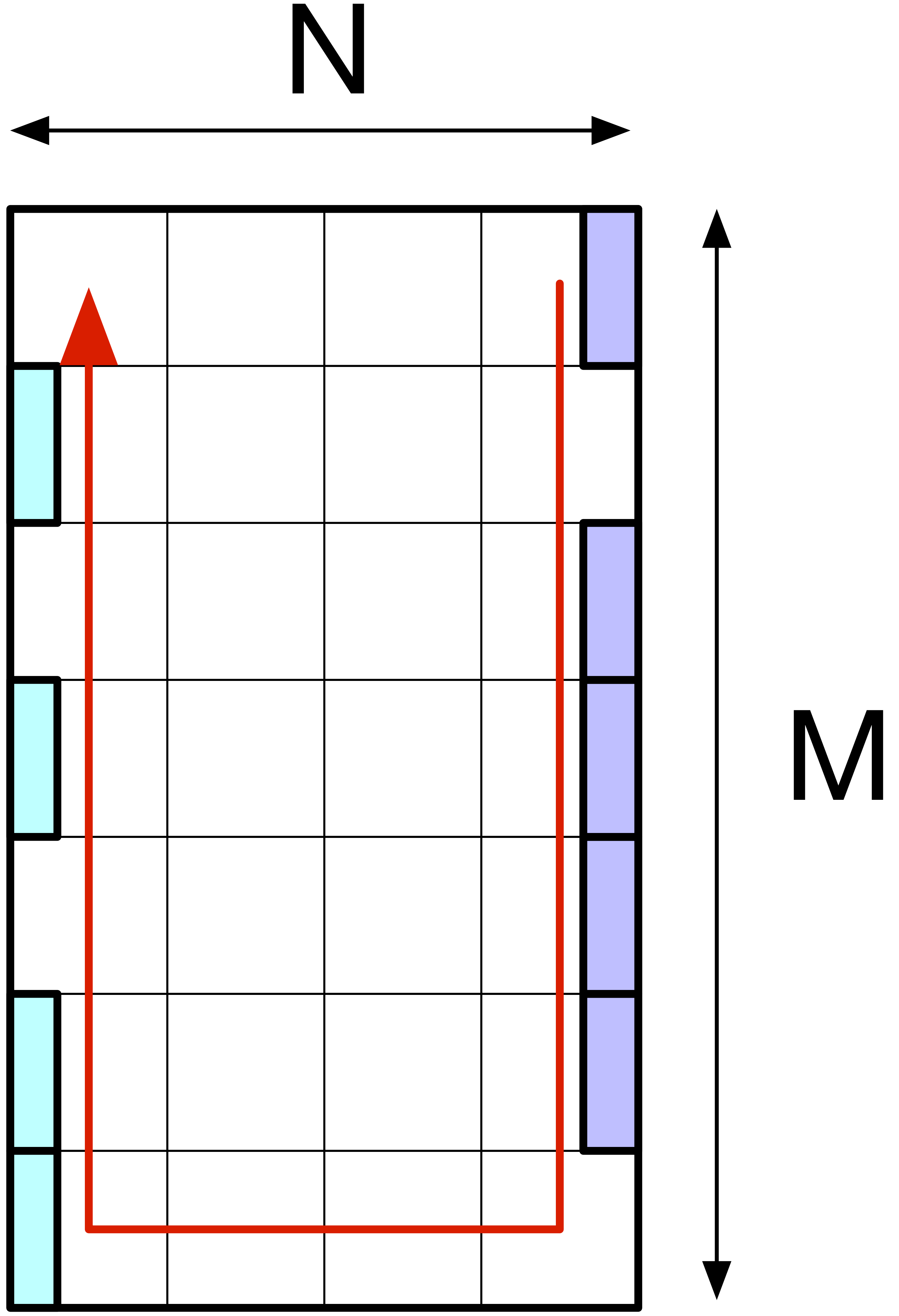
- **Clean separation** of 'physics algorithm' (the "**payload**") & infrastructure
  - Using **device-agnostic interface**

Currently based on the Trigger Processor architecture

**Implementation & design:**
*builds on phase-1 'MP7' framework*

# EMP framework (2) – Key Features

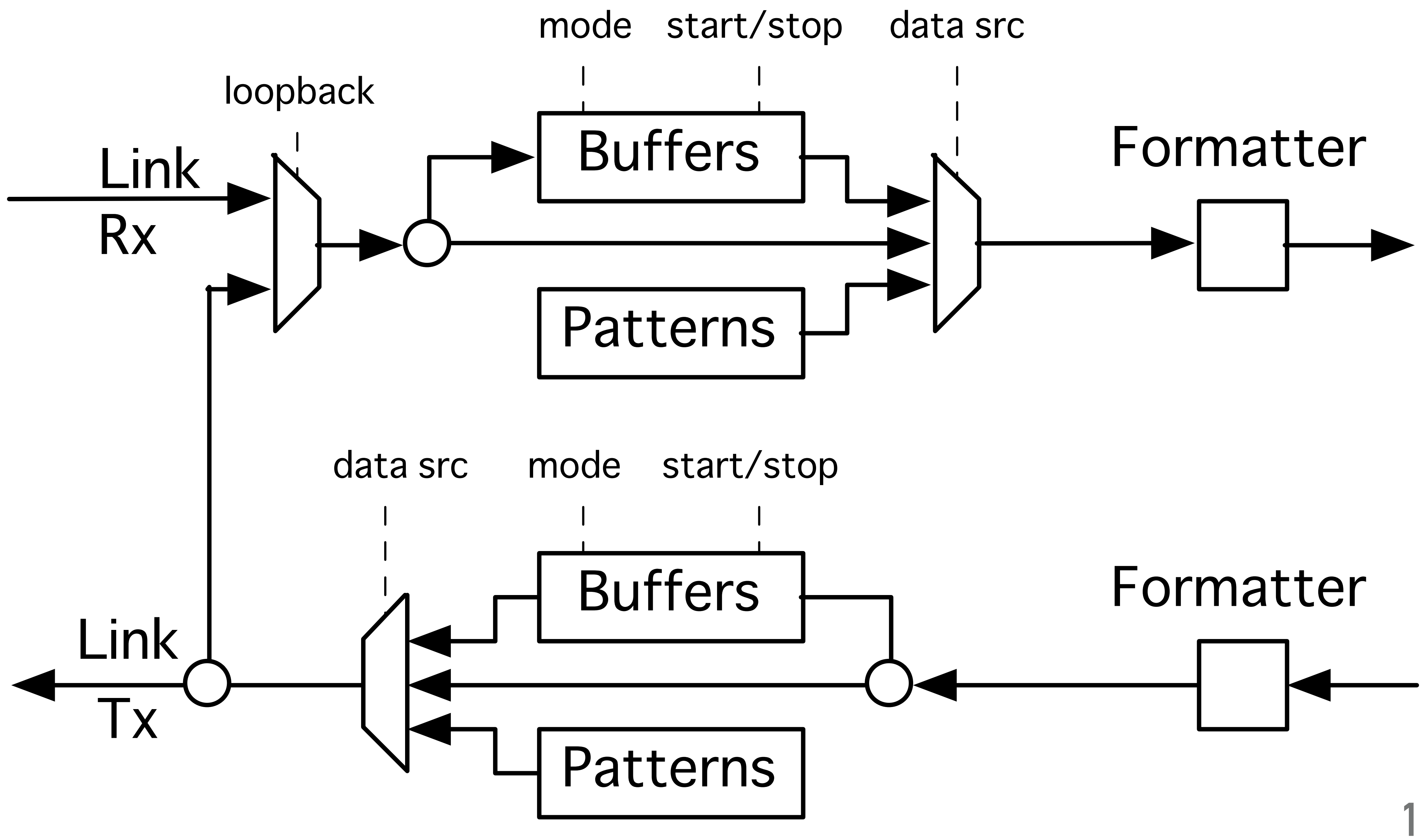
- Infrastructure constrained to 'edge' of chip
  - 'Centre' of chip free for 'physics algorithm' FW
  - **Realistic routing & testing of algo FW without using physical links** (even before link FW ready)
- Customising the payload (i.e. physics algo)
  - **Only need to write the source code for your trigger algorithm**
    - **Reference generic top-level entity** for that board in build tool config. file



# EMP FRAMEWORK (3) – KEY FEATURES

- Functionality
  - **Configuration, clock management, TTC**
    - Configurable clocking infrastructure (n*LHC clock)
    - TTC command injection
    - TTC history capture
    - Internal random trigger generators
  - **I/O buffers**
    - Read & write through control bus
    - **Reliable playback/capture synchronisation** across the chip
    - If wanted, can **instantiate only those buffers** that **match layout of MGTs** on any particular FPGA / package

*per-channel buffers*
*4 channels per region*

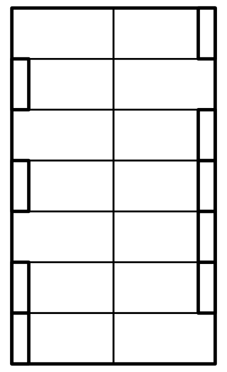


# EMP framework (4) – Natively Supported FPGAs

**Vivado:** *2017.4 & 2018.1*

- Devices
  - Implemented & tested:
    - **KU115** — HiTech Global K800 & MPUltra
    - **VU9P** - VCU118
    - Both with test 'null algo' payload, and resource-hungry 'physics algo' payload from track finder
  - Planned: KU15P
  - **Easily portable** between different FPGAs & packages
    - With minimal code duplication
    - E.g. Ultrascale vs Ultrascale+: Only duplicate modules that directly instantiate family-specific primitives

# EMP FRAMEWORK (5) – EXAMPLE IMPLEMENTATION



**Key**
Datapath
Infra (excl. PCIe)
PCIe xdma
TTC

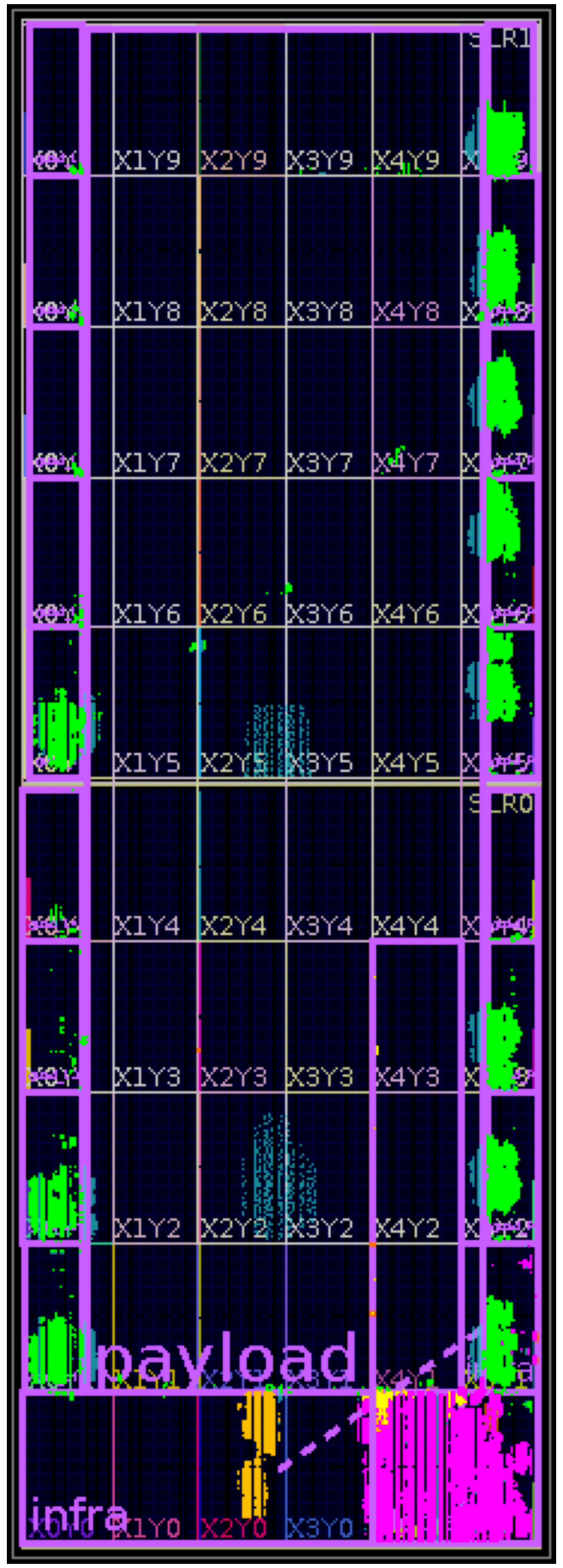
## Device declaration

```
----------------------------------------------------------------
package emp_device_decl is

  constant BOARD_DESIGN_ID : std_logic_vector(7 downto 0) := X"01";

  constant N_REGION       : integer := 18;

  constant N_REFCLK     : integer := 10;
  constant CROSS_REGION : integer := 8;

  constant IO_GT_REGIONS : io_gt_array(0 to N_REGION - 1) := (
    0  => io_gt,
    1  => io_gt,
    2  => io_gt,
    --3  => io_gt,
    4  => io_gt,
    5  => io_gt,
    6  => io_gt,
    7  => io_gt,
    8  => io_gt,
    --11  => io_gt,
    --12  => io_gt,
    13 => io_gt,
    16 => io_gt,
    17 => io_gt,
    others  => io_no_gt
  );

end emp_device_decl;
----------------------------------------------------------------
```

## Project declaration

```
-- mgt -> chk -> buf -> fmt -> (algo) -> (fmt) -> buf -> chk -> mgt -> clk -> altclk
-- reserve 0 and 19 for infrastructure
constant REGION_CONF : region_conf_array_t := (
  0  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 4, 5),  -- 232
  1  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 4, 5),  -- 231
  2  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 2, 3),  -- 230
  4  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 2, 3),  -- 6 / 112
  5  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 2, 3),  -- 5 / 113
  6  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 0, 1),  -- 4 / 114
  7  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 0, 1),  -- 3 / 115
  8  => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 0, 1),  -- 3 / 115
  ---- Cross-chip
  13 => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 8, 9),  -- 2 / 116
  16 => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 6, 7),  -- 1 / 117
  17 => (no_mgt, u_crc32, buf, no_fmt, buf, u_crc32, no_mgt, 6, 7),  -- 0 / 118

  others => kDummyRegion
  );
```

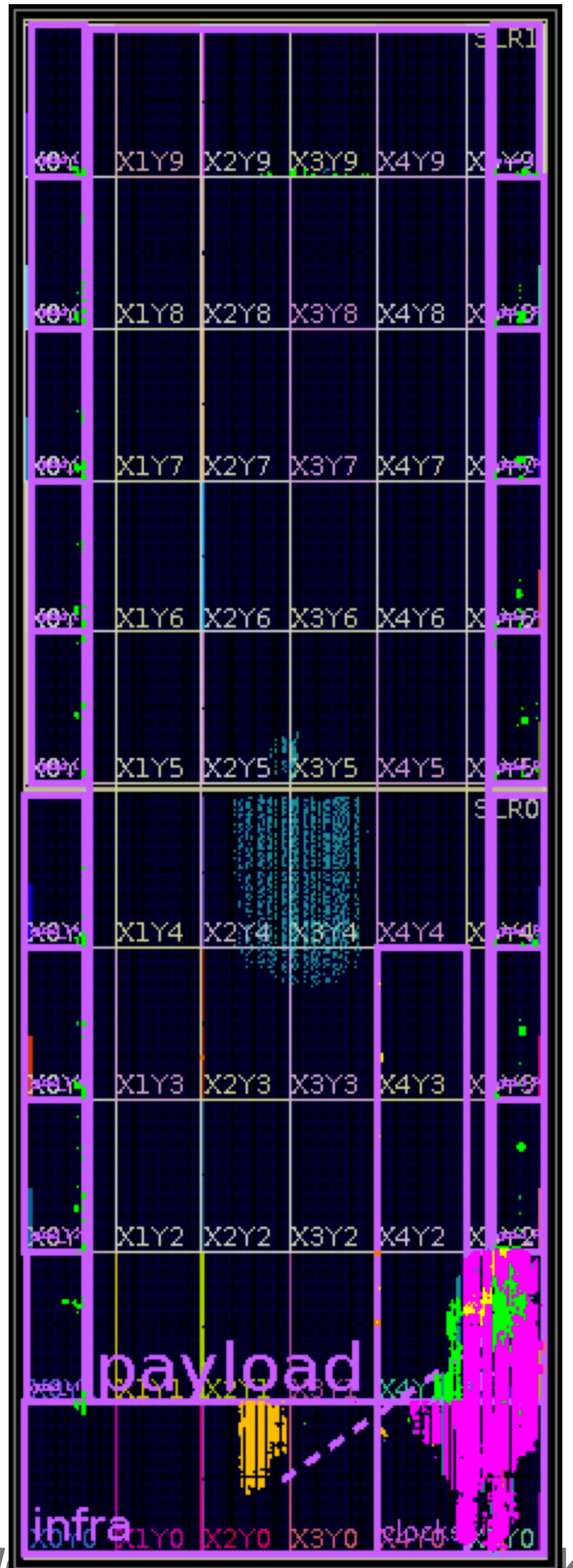
**K800: 44 channel**



# EMP FRAMEWORK (6) – EXAMPLE IMPLEMENTATION

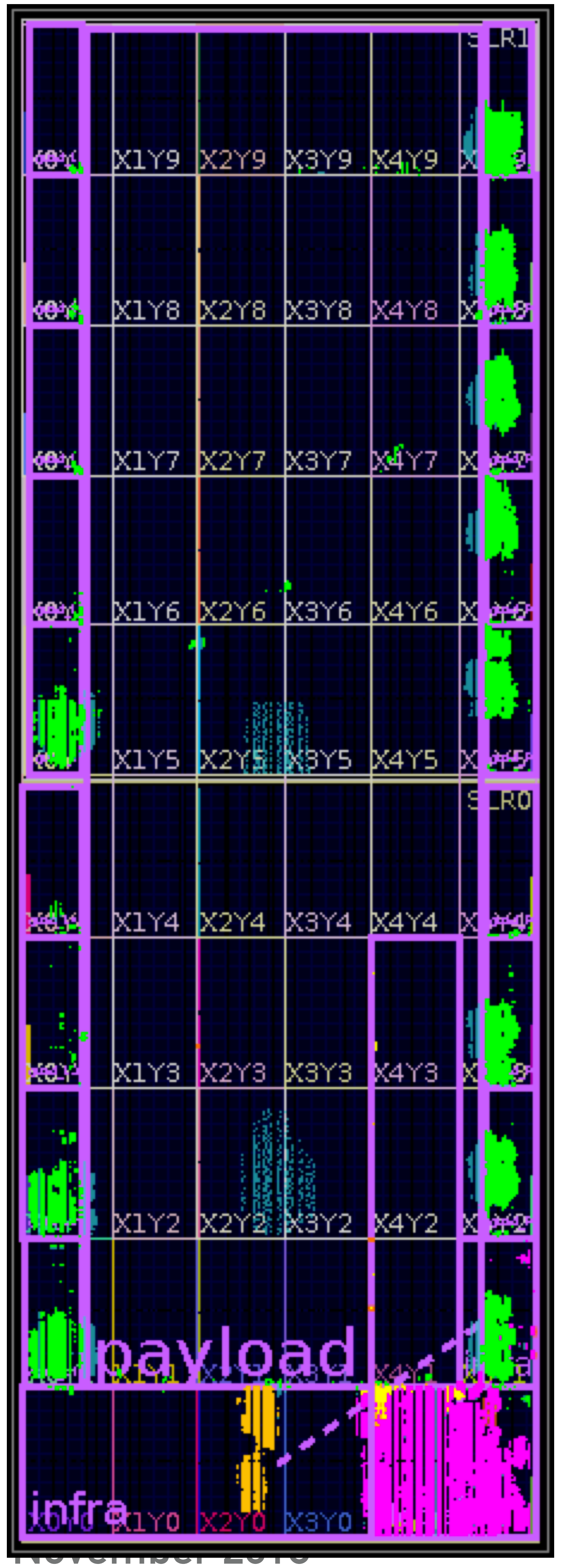
Key
Datapath
Infra (excl. PCIe)
PCIe xdma
TTC

K800: 4 channel




Change value of one constant in VHDL package
emp_project_decl

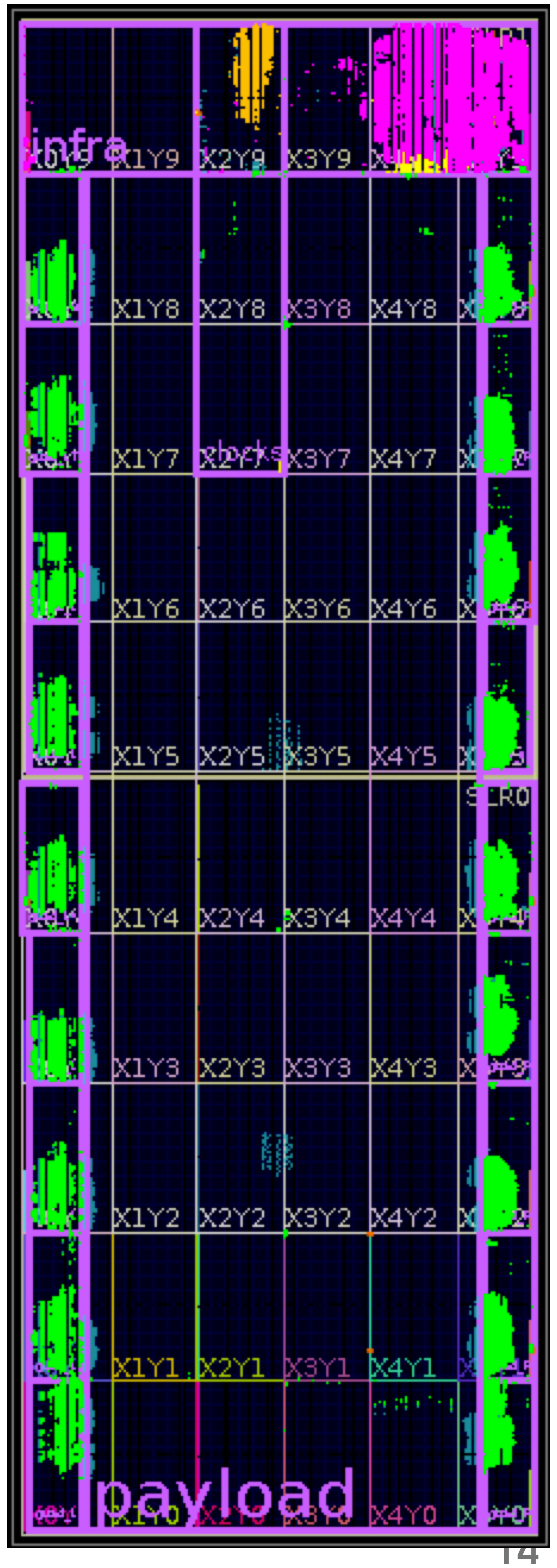
K800: 44 channel




Reference different device-specific VHDL package & area constraints

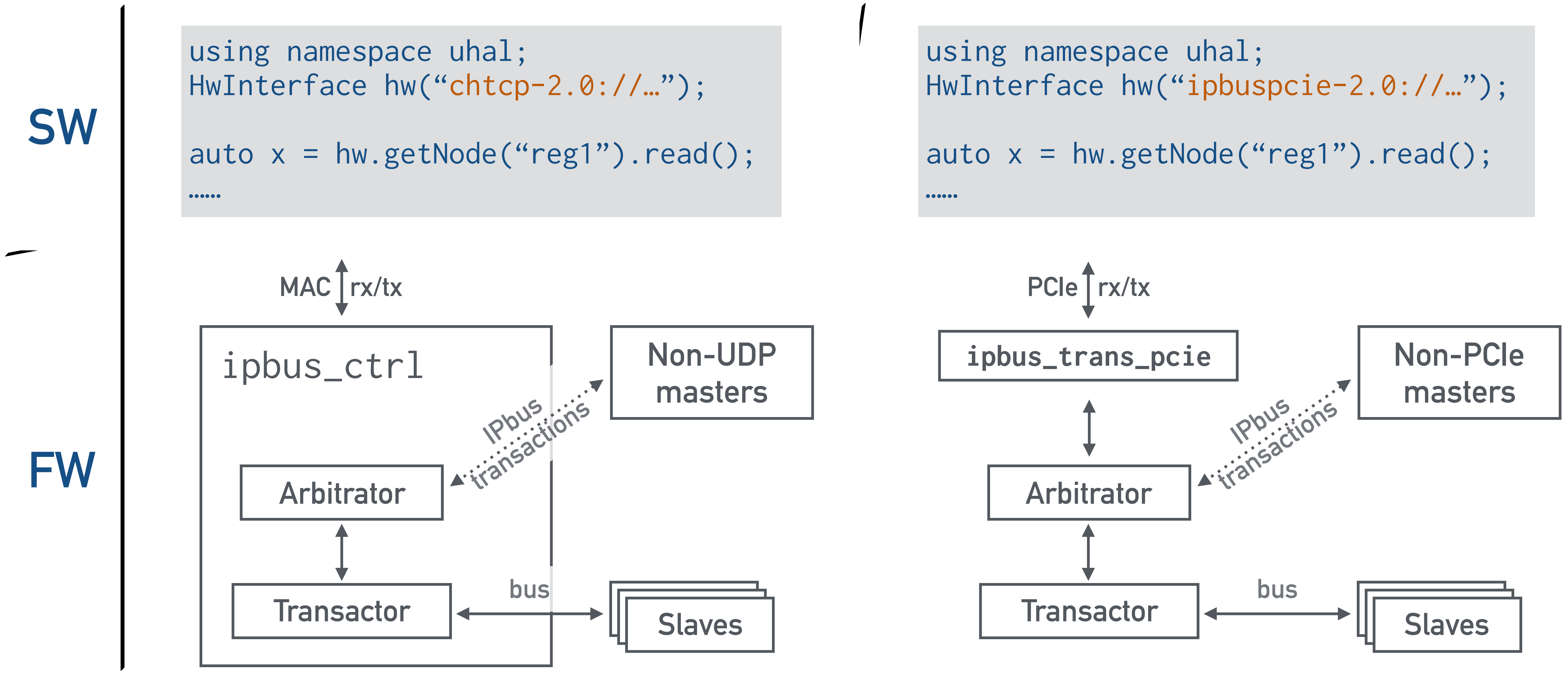
Change value of one constant in VHDL package
emp_design_decl

MPUltra: 72 channel



# EMP FRAMEWORK (7) – CONTROL LAYER

- Slow control
  - IPbus, extended to support PCIe as transport layer
  - Internal bus & high-level SW agnostic to link/transport layer protocol
    - Can **easily adapt above PCIe-based designs to Ethernet/UDP** — and potentially other link/transport layer protocols — by **only changing 2 VHDL files**, *without re-writing any software that controls the firmware framework*



# EMP framework (8) – Testbench

- So far, focussed on implementing firmware in real devices
  - But, during development, it can be very useful to simulate (algorithm) firmware in a testbench …
- General-purpose payload testbench
  - Allows you to **inject/capture data in simulation** using the **same file format for input/output data as with real hardware**

    $\Longrightarrow$ **Quasi-transparent switch between** testing algorithms in **HW and simulation**

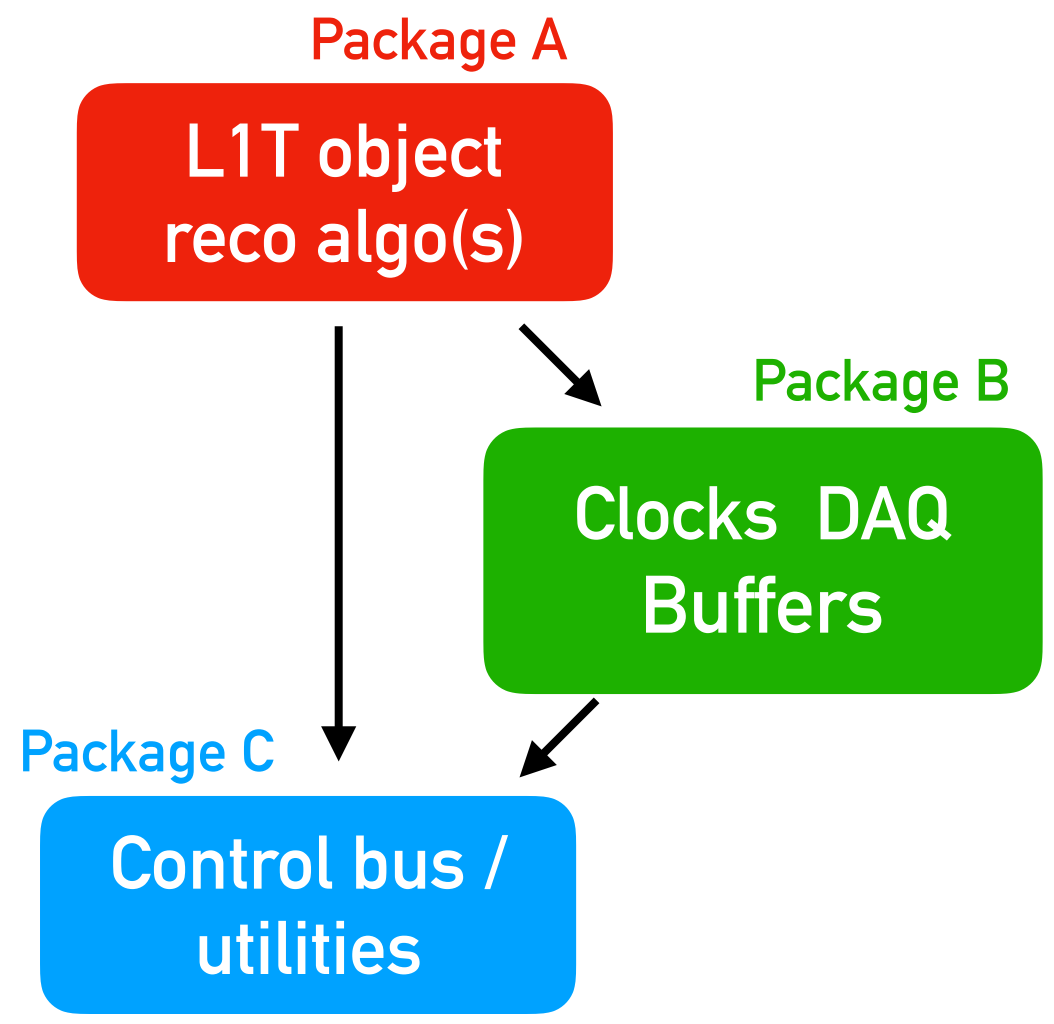
    ```
    vsim -c -Gsourcefile=ttbarPU200_26_5_0.txt -Gsinkfile=output.txt work.top -do "run 3000ns;"
    ```

  - Implementation bypasses full framework simulation for extra speed
- Documentation
  - For access, subscribe to 'emp-fwk-users' e-group
  - https://gitlab.cern.ch/p2-xware/firmware/emp-fwk/wikis/home
    - Explains how to integrate in custom 'physics algo' payload & how to play data through payload using associated software

# Firmware build tool

# Introduction (1)

- Modern FPGAs
  - Large amount of resources; substantial firmware projects
  - Single bitfile: Hundreds of source code files




- Common problems to solve $\Longrightarrow$ share code
  - Minimise time spent debugging (difficult & time-consuming)
  - Avoid code (and bug) duplication
  - Result: Re-usable “components”, organised in “packages”

**Component**

- Self-consistent set of sources, cores & config files
  - *with well-defined dependencies*
- Entities used together to achieve a **single/common high-level goal**
- E.g. clocks; TTC; I/O buffers; links

**Package**

*essentially a “firmware library”*

- Collection of components (and optionally top-level projects) that …
  - *Belong to the **same context***
  - *Have the **same release cycle***
- Naturally corresponds to 1 git repo

# INTRODUCTION (2)

- Code from multiple sources → 1 bit file
- Workflow needs to **ensure reproducibility** in this **multi-institute, multi-developer scenario**
  - Consistent results **essential for debugging**
  - **'Ideal' workflow** would allow people to avoid …
    - Having to manually resolve avoidable complications when migrating from version A.B to X.Y of some FW component
    - …
  - **In order to maximise time spent …**
    - Developing & implementing trigger algorithms
    - Testing new X Gbps HW/FW
    - Working on physics analysis
    - …

# Build tool: Design goal

- Ability to easily
  - **Assemble a working project** from its components, and **build firmware** using a **command-line interface**
    - Vivado, ModelSim, …

```
< checkout repo1 vX.Y >
<checkout repo2 vA.B >
< setup vivado project >
<build bit file>
```

  - … but also, **switch to IDE GUI at any step** in the process
  - Connect components to the control bus
    - … AND move them to a different address by only updating 1 file

- From source code to bitfile (or sim), workflow should be
  - **Simple & reproducible**, under **any circumstances**

# BUILD TOOL: IPBB

https://github.com/ipbus/ipbb

- IPBB = **IPbus Build tool**
  - A command-line firmware project management & build tool
  - Developed to achieve design goals from last few slides
    - i.e. setting up & building complex FW projects should be simple & reproducible, under all circumstances
  - Not IPbus specific
  - Integrated with **Vivado** & **ModelSim/QuestaSim**
    - Dependency resolution
    - Project creation, synthesis, implementation & bitfile generation
    - IPcore generation
  - Integrated with **git** & **SVN**
  - Added feature for IPbus-based designs
    - Auto-generation of IPbus address decoder logic

**Technical details**

- Builds on concepts from ProjectManager.py
- Implemented in Python (2.7)
- Supported OS: Linux

# Dependency (.dep) files

*Describe relations between sources, cores, constraints & address tables*

- Parsed by build tool ⟹ list of all files required to build bit file / run sim
- For example, top-level dep file in package A, using components from packages B & C …

**Package A**

Top-level dep file

```
src top.vhd
src entity1.vhd entity2.vhd

include package-B:component1
include package-B:component2

include package-C:myComponent
```

**Package-B**

component1/firmware/cfg/component1.dep

```
src someEntity.vhd
src anotherEntity.vhd

src --cd ../ucf area_constraints.tcl
```

component2/firmware/cfg/component2.dep

```
src entityA.vhd
src entityB.vhd
src anotherEntity.vhd
src somePackage.vhd
```

**Package-C**

```
src anEntity.vhd
src anotherEntity.vhd
src myPackage.vhd
```

Crucially, source code files implementing any component only need to be listed once, in that component, not in any other compnents/packages

# IPBB: Vivado workflow

- Example: Building emp-fwk K800 ‘null algo’ design

```
ipbb init phase2-work
cd phase2-work
ipbb add git https://github.com/ipbus/ipbus-firmware -b v1.2
ipbb add git https://:@gitlab.cern.ch:8443/p2-xware/firmware/emp-fwk.git -b v1.1.1
ipbb add git https://:@gitlab.cern.ch:8443/cms-cactus/firmware/mp7.git -b ephemeral/phase2-vA

ipbb proj create vivado k800_72channel emp-fwk:projects/examples/k800 -t top_full.dep
cd proj/k800_72channel

ipbb vivado project
```

  - Newly-created project accessible with Vivado
    - At this point, user can choose to open Vivado GUI & work from there …
    - Or, build from command line (e.g. for ‘blind’ or automated builds)

```
ipbb vivado synth -j4 impl -j4
ipbb vivado package
```

Run synthesis & implementation

Packages bit file in tarball with address tables & build metainfo

  - Behind the scenes, “ipbb vivado” commands use Vivado TCL interface

# Automated builds & tests

# AUTOMATED BUILDS & TESTS (1)

- Last decade: Growth in automation tools for developers, e.g:
  - Jenkins; Gitlab CI

- Widely used in software world
  - e.g. L1T online software
    - Until YETS 17/18: Nightly builds (cron job, running custom Python script)
    - Since YETS 17/18: Gitlab CI

- Why automate builds & tests?
  - Verifies build + test successful, without risk of human error
    - E.g. Automated build will fail if you forget to commit crucial new file

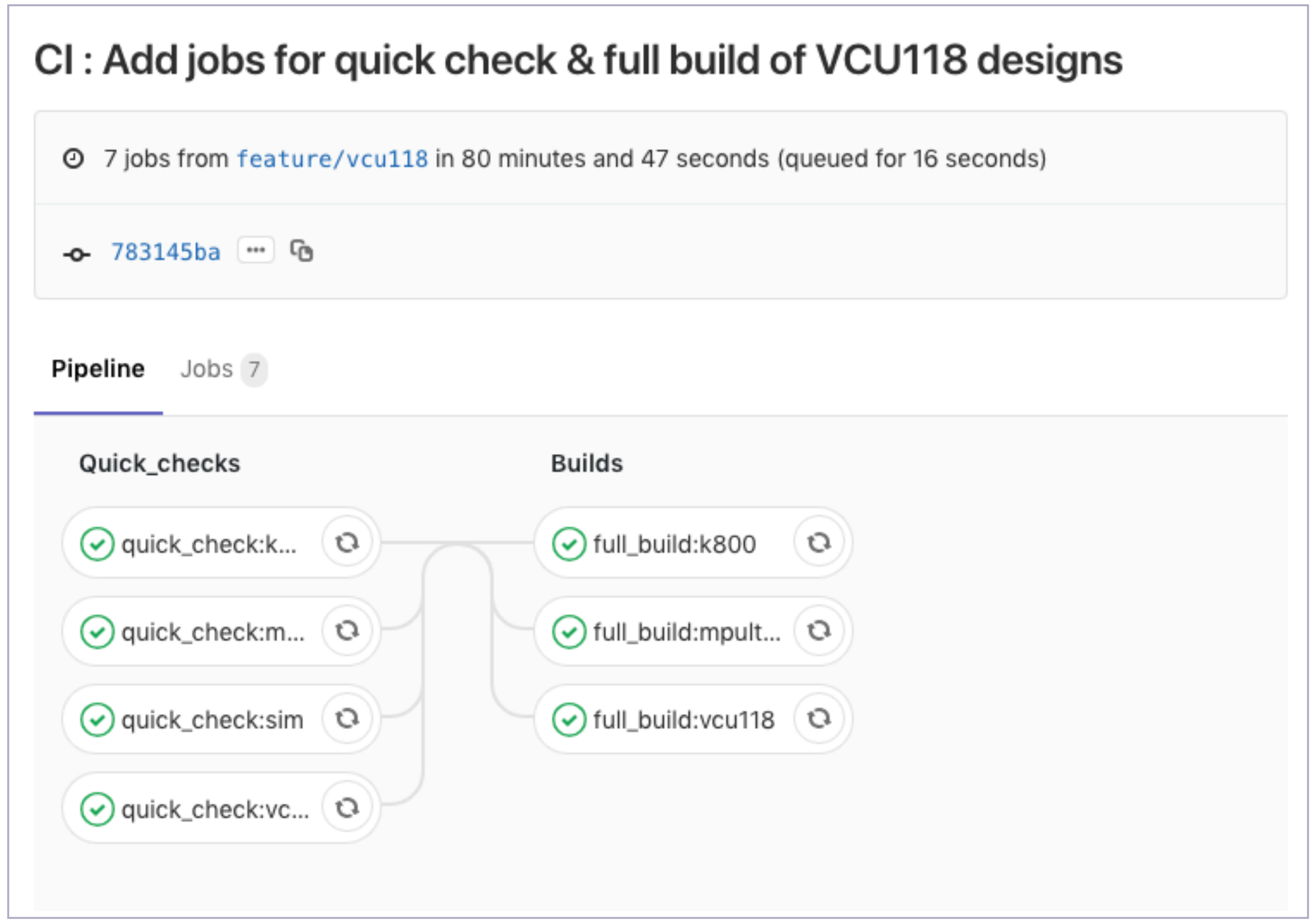
  - Minimises amount of time spent manually verifying firmware
    - Esp. important when building for **multiple platforms** (e.g. FW for multiple chips)

# AUTOMATED BUILDS & TESTS (2)

- Firmware
  - Build tools require license
  - Synthesis & implementation for resource-intensive designs
    - Can take very long time & have high memory usage

Neither of these differences precludes automated builds (& tests) from being useful

- EMP framework
  - Using Gitlab CI, with 2-stage pipeline
    1. Dependency & syntax check in Vivado/ModelSim for each design
       - Quick; run on all commits
    2. Build 'null algo' designs
       - Takes longer; only run on-demand (manually triggered)



# Summary

- Infrastructural firmware
  - Common implementation: Avoid re-designing the wheel
    - Portable between **different FPGAs & packages** with **minimal code duplication**
  - EMP framework
    - Clocking & I/O buffers highly configurable, using build-time constants
    - Two KU115 designs (K800) one VU9P (VCU118) implemented; KU15P planned
- Firmware build tool
  - Useful, esp. if reusing common FW components in different designs
  - Goal: *Ensure 'firmware build/sim' workflow is simple & reproducible, under any circumstances*
  - IPBB
- Automated builds & tests
  - Very useful, in particular when targeting multiple platforms/configurations