EFCL Winter School – Track 2
Customizing RISC-V Based Microcontrollers

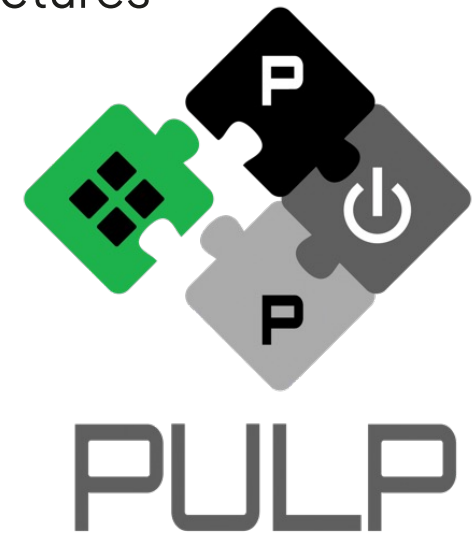
Lecture 1 – PULP Platform & PULPissimo architecture

Yvan Tortorella - yvan.tortorella@chips.it

We started >10 years ago (April 2013)

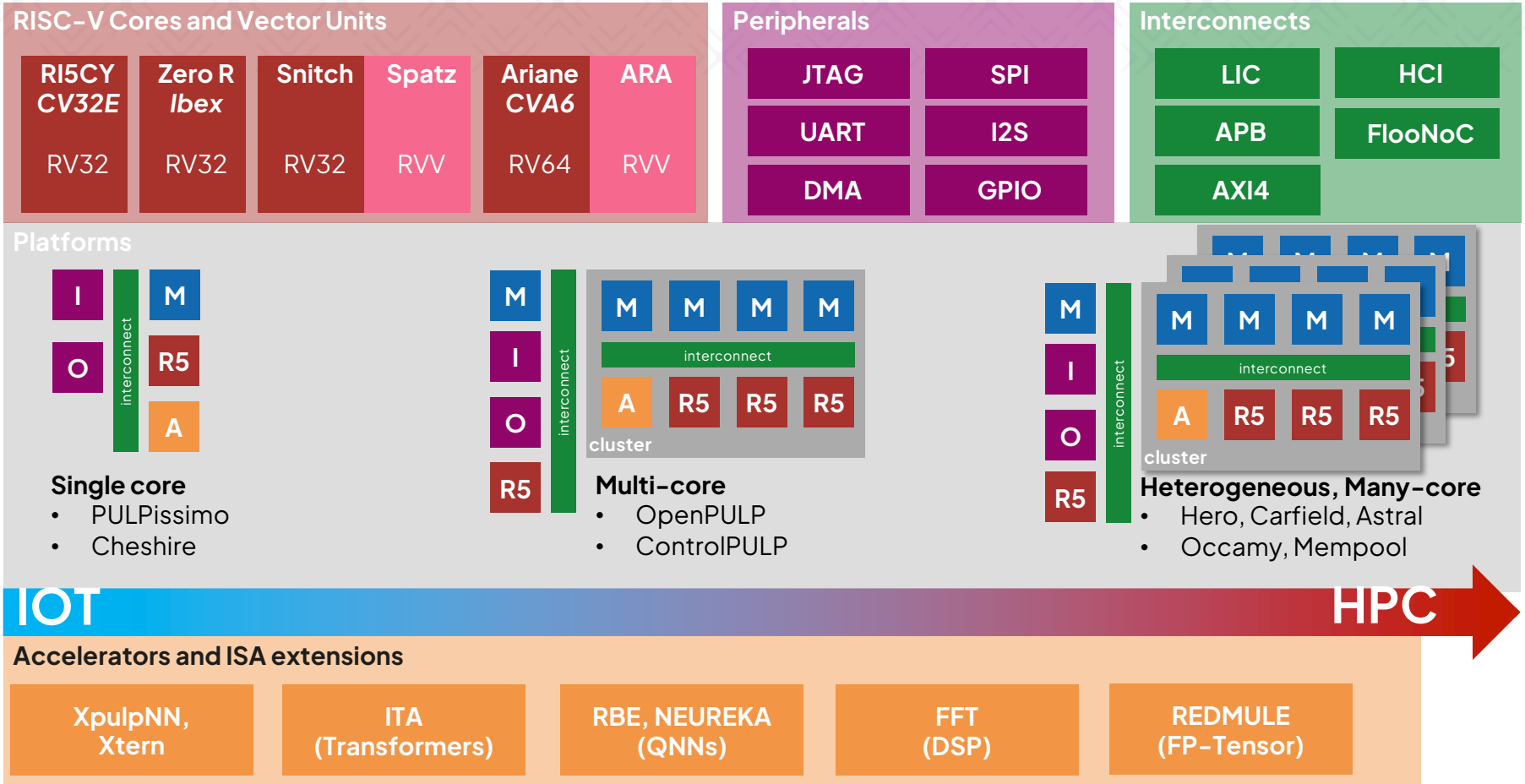
Chips-IT

- Born in April 2013, investigating new computing architectures
  - Efficient over a wide range (IoT <-> HPC)
- Key points
  - Parallel processing
  - Near threshold computing
  - Efficient switching between operating modes
  - Making best use of technology
  - Heterogeneous acceleration
- **Parallel Ultra Low-Power (PULP)** platform was born



The PULP Story

Chips-IT



Objective of this course

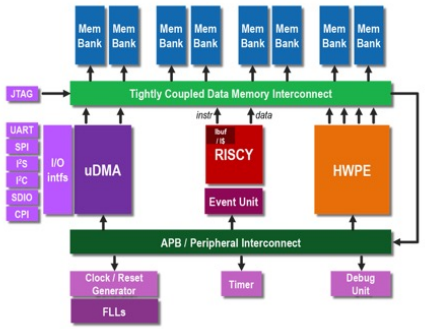
- We can not cover all the variety of PULP-based systems
  - *Host systems* (PULPissimo, Cheshire, ...)
  - *Programmable Accelerator systems* (PULP, Snitch, MemPool, ...)
- We focus on a "simple" microcontroller system and *customize* it
  - PULPissimo is a single-core MCU that we used as host system in most of our chips
  - Capable, programmable, and flexible...
  - ... but not particularly easy to use!

PULPissimo

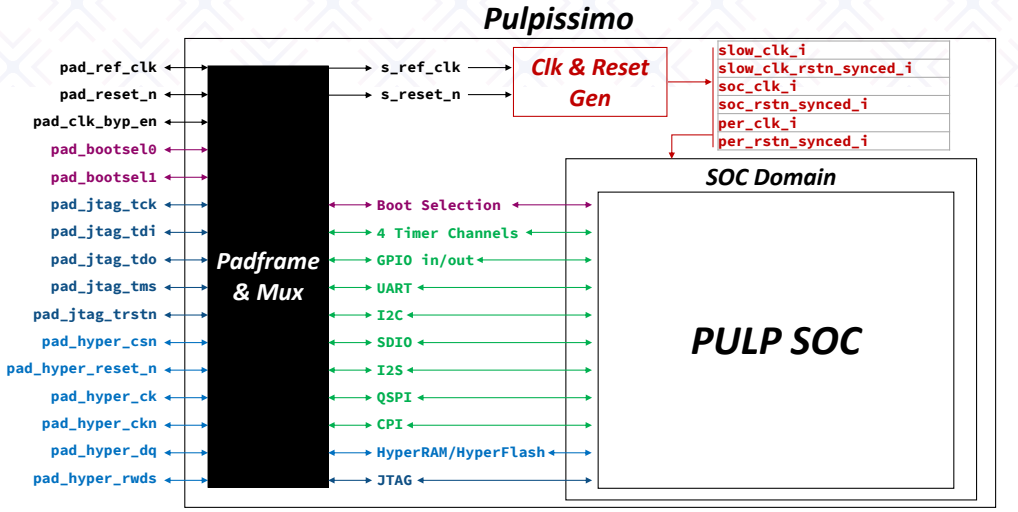
Citing

If you are using the PULPissimo IPs for an academic publication, please cite the following paper:

```
@INPROCEEDINGS{8640145,
  author={Schiavone, Pasquale Davide and Rossi, Davide and Pullini, Antonio and Di Mauro, Alfi
  booktitle={2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S
  title={Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX},
  year={2018},
  volume={},
  number={},
  pages={1-3},
  doi={10.1109/S3S.2018.8640145}}
```



Objective of this course




```
[06:36:36][ytortorella@lagrev5:/scratch2/ytortorella/ChipsIT/EfclWinterSchool2026/tests/pulpissimo]$ make help
"Using EFCL Winter School 2026 environment"
 PULPissimo Build & SIM Environment
Toplevel targets for building and simulating PULPissimo. Please check the make files in the subdirectories for additional targets..

Usage: make [TARGET [TARGET ...]] [ARGUMENT=VALUE [ARGUMENT=VALUE ...]]
Targets:
    help
        Get help for commands in this folder

    checkout
        Checkout all Bender IPs

    hw
        Re-generate generated hardware IPs

    bootrom
        Generate the boot rom

    padframe
        Generate the ASIC and simulation boot rom
        Generate the FPGA boot rom

    gpio-reconfigure GPIO=32
        Reconfigure number of GPIOs

        Params:
         - GPIO (example: 32) Number of GPIOs to reconfigure
```

- **We start by "dissecting" it into its constituent elements**
  - Understand what are the uDMA, the RISC-V core, interconnects, etc.
- **We learn *how* to use it in practice**
  - "I downloaded it from GitHub, but it doesn't work": let us fix that!
- **We augment PULPissimo in various ways**
  - Add hardware acceleration in the RISC-V core
  - Add separate cooperative HW accelerators
- **We deploy it on FPGA**

Prerequisites

Chips-IT

- **We assume three fundamentals**
  - Understanding of computer architecture (we'll *discuss* PULPissimo)
  - Capability to program in C (we'll *program* PULPissimo)
  - Familiarity with an HDL language (we'll *customize* PULPissimo)
- **We employ SystemVerilog**
  - Some of you probably know better VHDL (how many?)
  - Some of you might not have a strong background in HDLs in general (how many?)
  - See ***Teaching Aid****: Introduction to SystemVerilog for VHDL designers*
- **Hands-on are organized to reduce the need for previous knowledge**
  - Step-by-step guidance
  - For everything not on the guides, we have plenty of TAs!

More organization

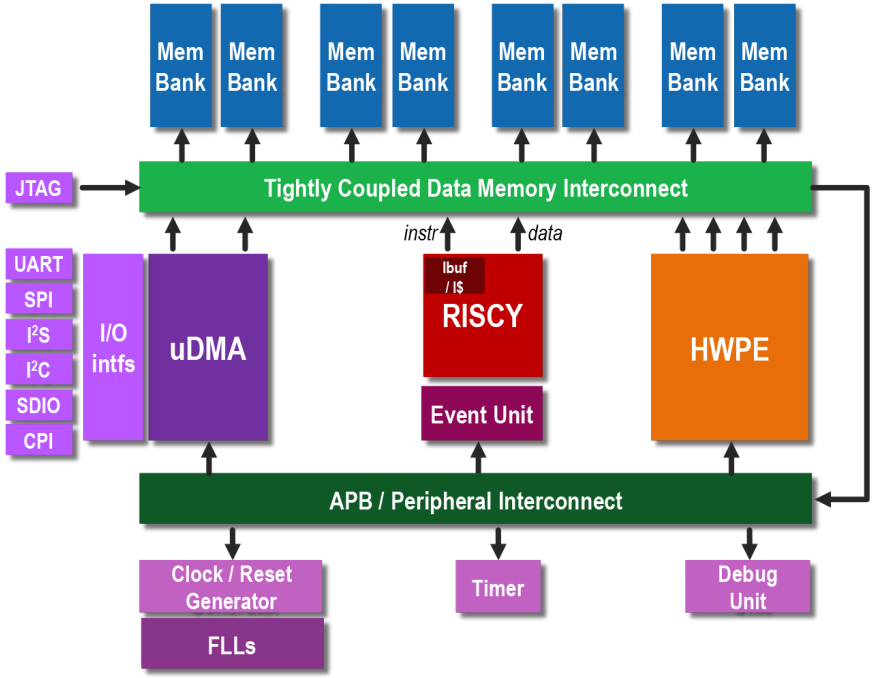
- Hands-on
  - Please organize in student pairs
  - It is recommended that at least one in the team has background in HDL
- Technical talks will take part in the E8 room (ground floor)
- Lessons, and hands on all in the D96.1 (lab room)
- You can bring home all code you produce
  - Exercises are also open-source on GitHub

Overview of the course

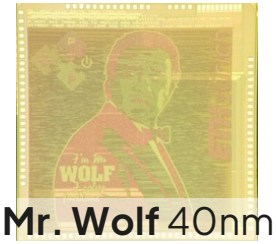
- **PULP platform & PULPissimo microcontroller architecture**
  - *Lecture*: PULPissimo microcontroller architecture
  - *Hands-on*: Introduction to the PULPissimo simulation environment
  - *Hands-on*: Introduction to the software environment and development of a FIR filter
- **Extending RISC-V cores with dedicated custom instructions**
- **Integrating cooperative HW Processing Engines / HWPEs**
- **Testing extended PULPissimo on FPGA**

So, what is PULPissimo?

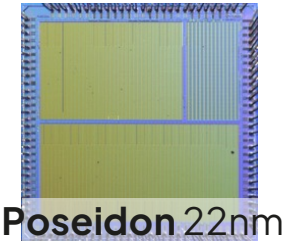
- Complete advanced single-core MCU System
  - 1 RISC-V core: **RI5CY / CV32** or **Ibex**
  - Multi-banked data memory
    - usually 4 word-interleaved banks = 128 bit of parallelism
  - "Private" memories + boot ROM
    - privileged access from core
  - An autonomous I/O DMA called **uDMA**
  - A set of peripherals
    - Clock / Reset generators
    - Timers
    - Debug Unit
  - Possibly, one or more **accelerators**
    - Within PULPissimo (**HWPE** – see Wed lecture)
    - AXI-coupled outside PULPissimo (e.g., the PULP cluster)



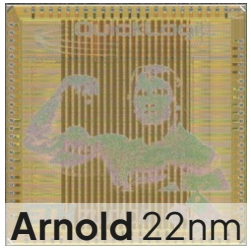
PULPissimo-based silicon



Mr. Wolf 40nm



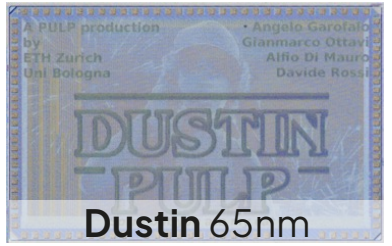
Poseidon 22nm



Arnold 22nm

Rosetta 65nm

Xavier 65nm



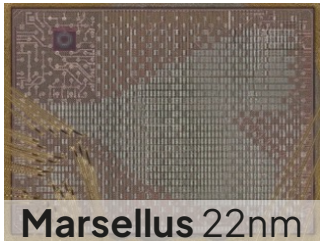
Dustin 65nm

Vega 22nm

Darkside 65nm

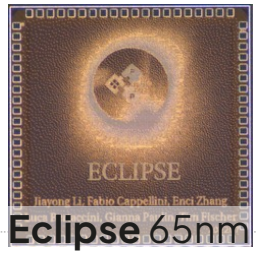
Echoes 65nm

Kraken 22nm



Marsellus 22nm

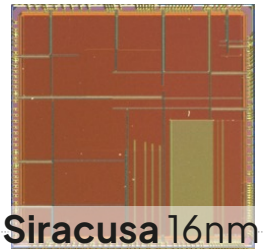
Cerberus 65nm



Eclipse 65nm

Kairos 65nm

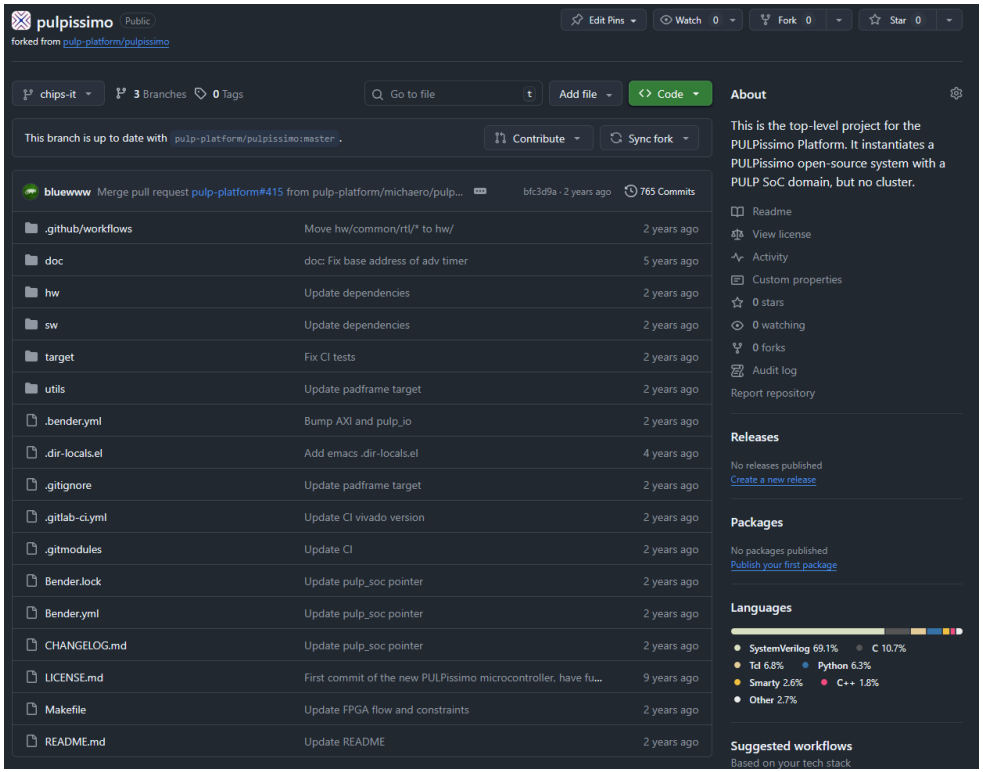
Trikarenos 65nm



Siracusa 16nm

...

The entry point for PULPissimo



- https://github.com/pulp-platform/pulpissimo
  - Does it contain everything? **No**
  - Like almost all PULP designs, PULPissimo makes extensive IP reuse and is built from a large collection of dependencies, each in its Git repository
- The PULPissimo GitHub contains
  - The testbench and simulation infrastructure
  - The top-level
  - Scripts for booting, FPGA
  - A bit of docs

Populating your PULPissimo

Chips-IT

```
[ytortorella@lagrev5 pulpissimo]$ make checkout
./bender checkout
  Checkout common_verification (https://github.com/pulp-platform/common_verification.git)
   Cloning common_verification (https://github.com/pulp-platform/common_verification.git)
  Checkout tech_cells_generic (https://github.com/pulp-platform/tech_cells_generic.git)
   Cloning tech_cells_generic (https://github.com/pulp-platform/tech_cells_generic.git)
  Checkout common_cells (https://github.com/pulp-platform/common_cells.git)
   Cloning common_cells (https://github.com/pulp-platform/common_cells.git)
  Checkout fpu_div_sqrt_mvp (https://github.com/pulp-platform/fpu_div_sqrt_mvp.git)
  # ... many others
```

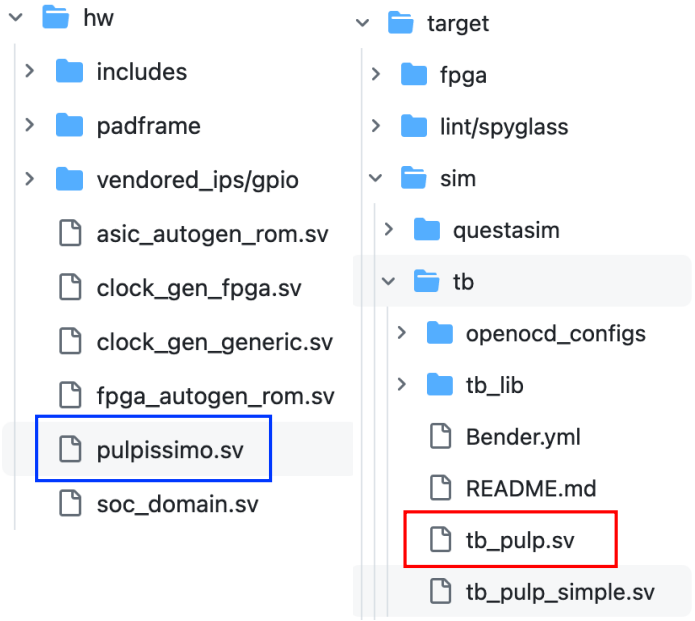
Populating your PULPissimo

Chips-IT

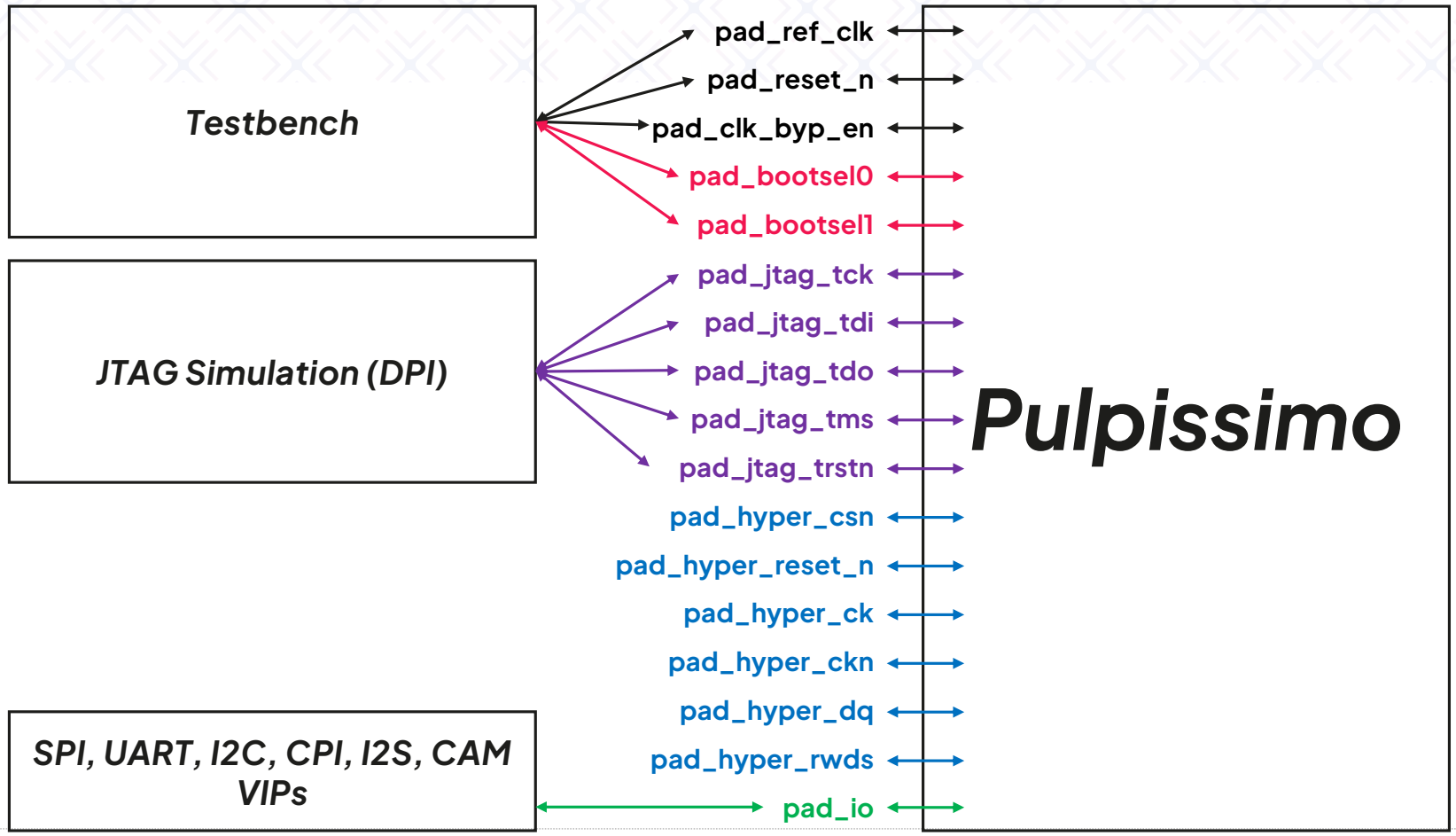
```
[ytortorella@lagrev5 pulpissimo]$ ls -l .bender/git/checkouts/
total 760
drwxr-xr-x+ 5 fconti users  9 May  8 23:48 adv_dbg_if-1ca7cdfb84c8f5ca
drwxr-xr-x+ 3 fconti users  7 May  8 23:48 apb2per-940361c093488815
drwxr-xr-x+ 5 fconti users  8 May  8 23:48 apb_adv_timer-8bef6a5bc0cb1116
drwxr-xr-x+ 4 fconti users  9 May  8 23:48 apb-e376e45e644c1b22
drwxr-xr-x+ 5 fconti users 10 May  8 23:48 apb_fll_if-836c625eb108998a
drwxr-xr-x+ 5 fconti users  9 May  8 23:48 apb_gpio-59a7da5ce725eeaa
drwxr-xr-x+ 4 fconti users  8 May  8 23:48 apb_interrupt_cntrl-637bd323fb264bca
drwxr-xr-x+ 4 fconti users  9 May  8 23:48 apb_node-e76108db35262d0b
# ... many others
```

PULPissimo testbench and top

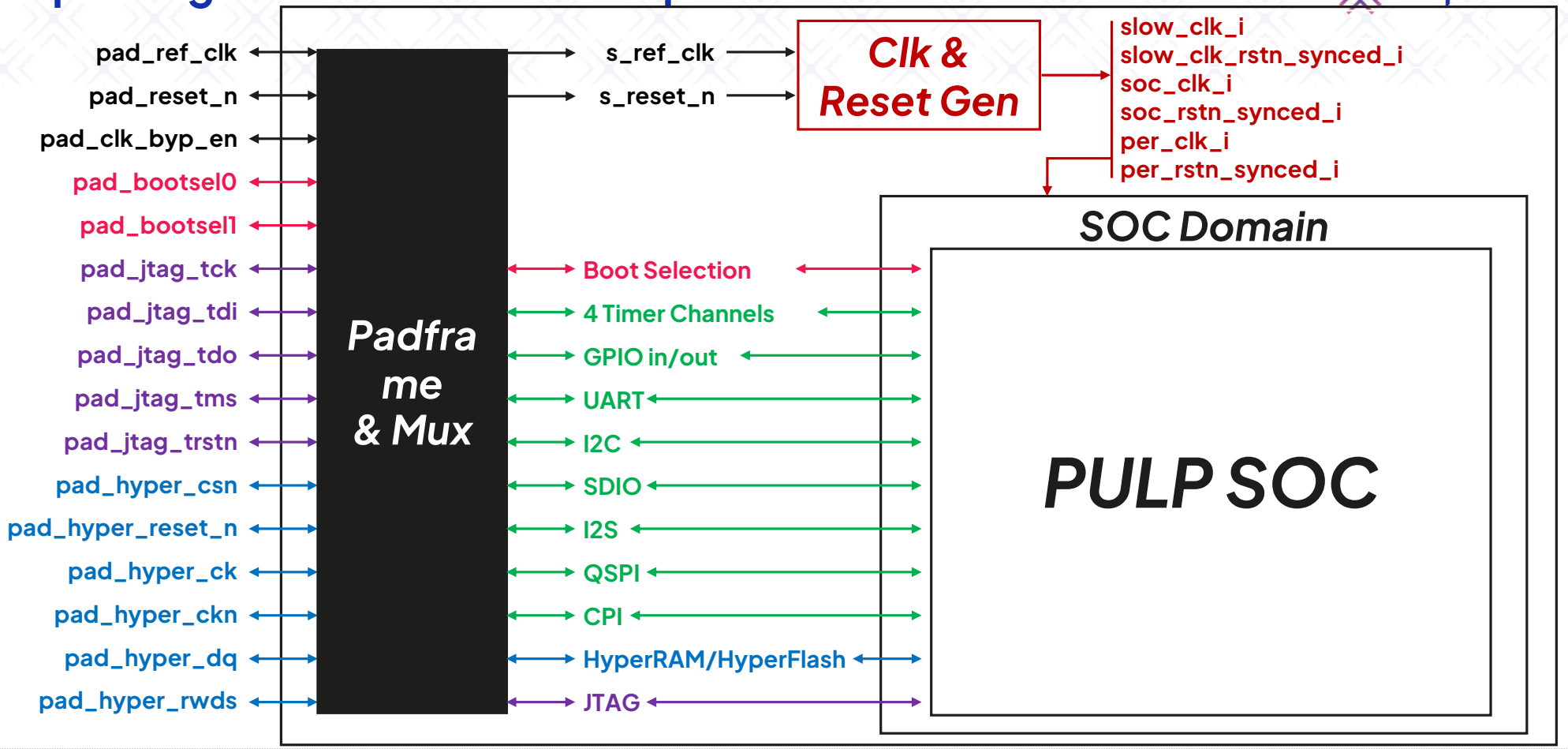
- **Simulation top-level**
  - the **testbench** in **target/sim/tb/tb_pulp.sv**
  - the PULPissimo **top** in **hw/pulpissimo.sv**
- **The PULPissimo top-level exposes a chip-like interface**
  - Only pads you would find in an actual PULPissimo chip
  - IO (with alternate modes) + CLK + RST + Boot mode sel



PULPissimo testbench and top

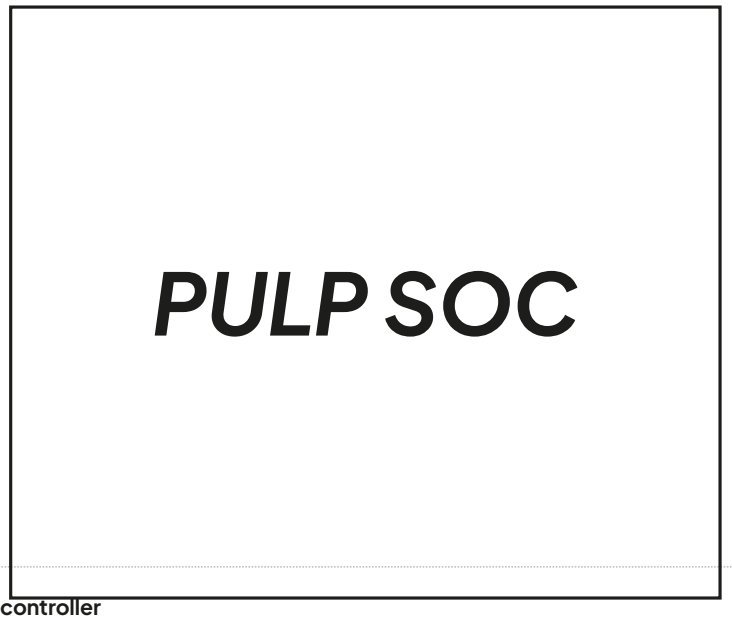


Opening the PULPissimo top-level box



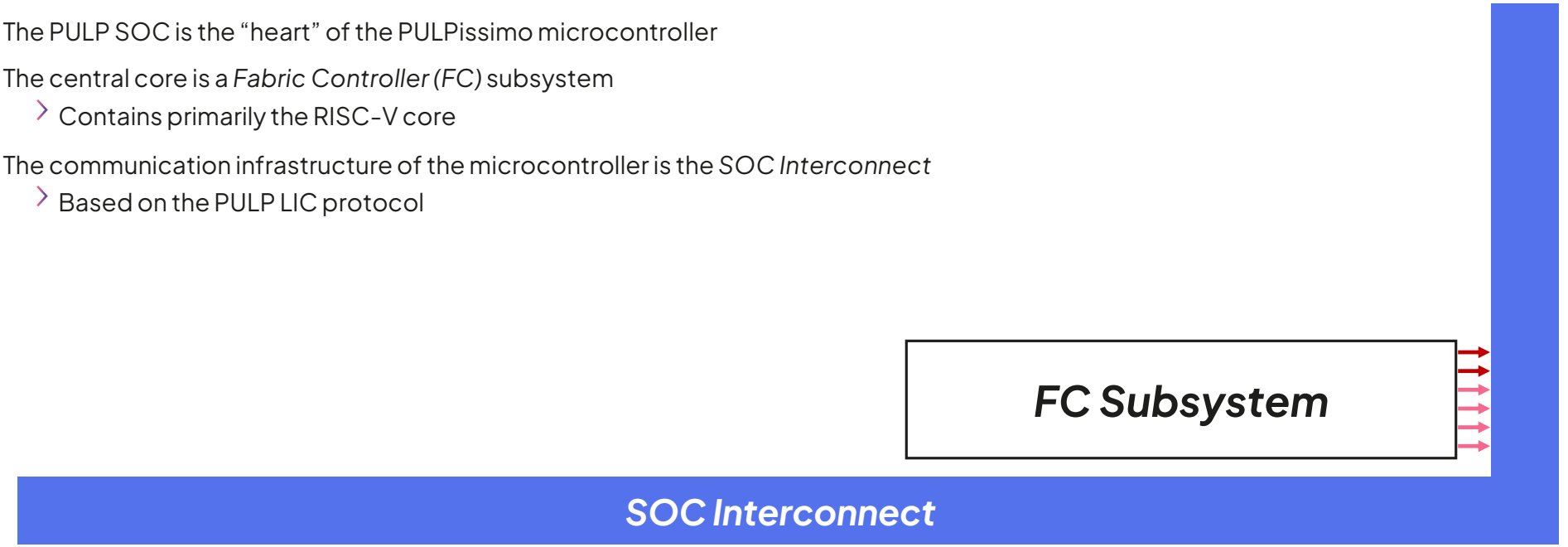
Opening the PULPissimo top-level box

The PULP SOC is the “heart” of the PULPissimo microcontroller



Opening the PULPissimo top-level box

- The PULP SOC is the “heart” of the PULPissimo microcontroller
- The central core is a *Fabric Controller (FC)* subsystem
  - Contains primarily the RISC-V core
- The communication infrastructure of the microcontroller is the *SOC Interconnect*
  - Based on the PULP LIC protocol



PULP Logarithmic Interconnect (LIC) protocol

Chips-IT

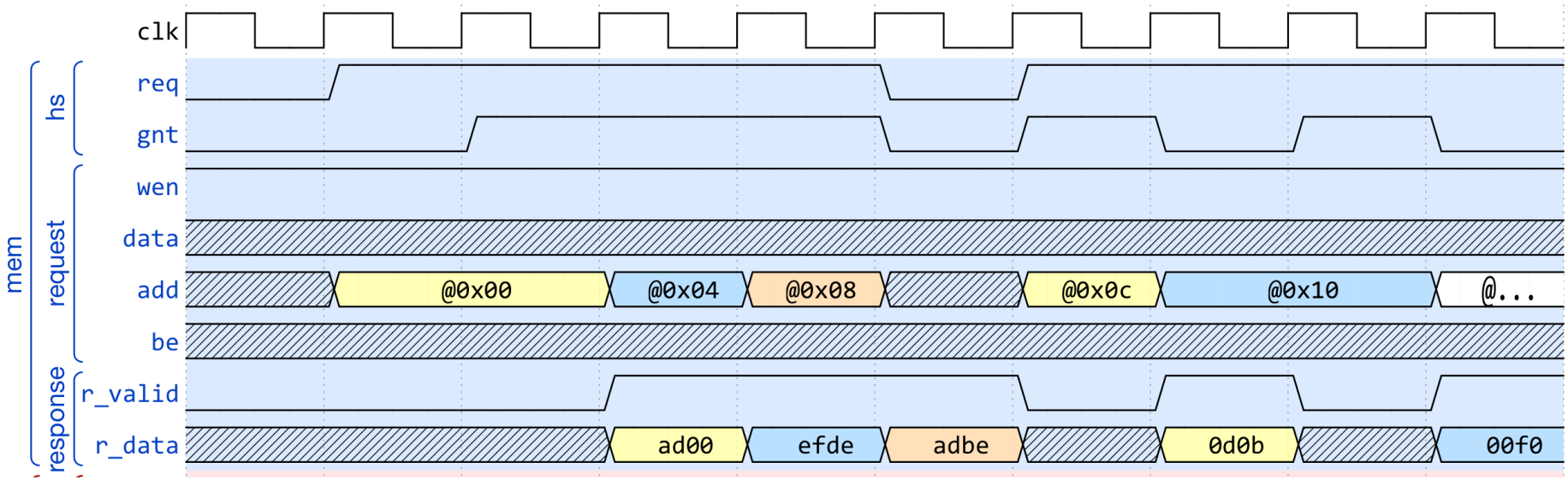
Protocol meant for use through low-latency, high-bandwidth crossbars.

- **fully synchronous**, and **pipelined**
- every transfer takes (at least) two cycles in two phases: ***REQUEST*** and ***RESPONSE***
- not a standard, some IPs define their own version with stricter rules:
  https://ibex-core.readthedocs.io/en/latest/03_reference/load_store_unit.html#protocol
  https://hwpe-doc.readthedocs.io/en/latest/protocols.html#hwpe-mem
- OpenHW Group's OpenBus Interface (OBI) is a very similar, more strictly standardized protocol
  https://github.com/openhwgroup/obi

	Name	Bits	Direction	Meaning
request channel	**req**	1	Initiator to Target	Valid request on the bus
	wen	1	Initiator to Target	1 for read, 0 for write
	data	32	Initiator to Target	Write data payload
	addr	32	Initiator to Target	Memory map address
	be	4	Initiator to Target	Byte enable
response channel	**gnt**	1	Target to Initiator	1 if request accepted by the target
	r_valid	1	Target to Initiator	Valid response on the bus
	r_data	32	Target to Initiator	Read data payload

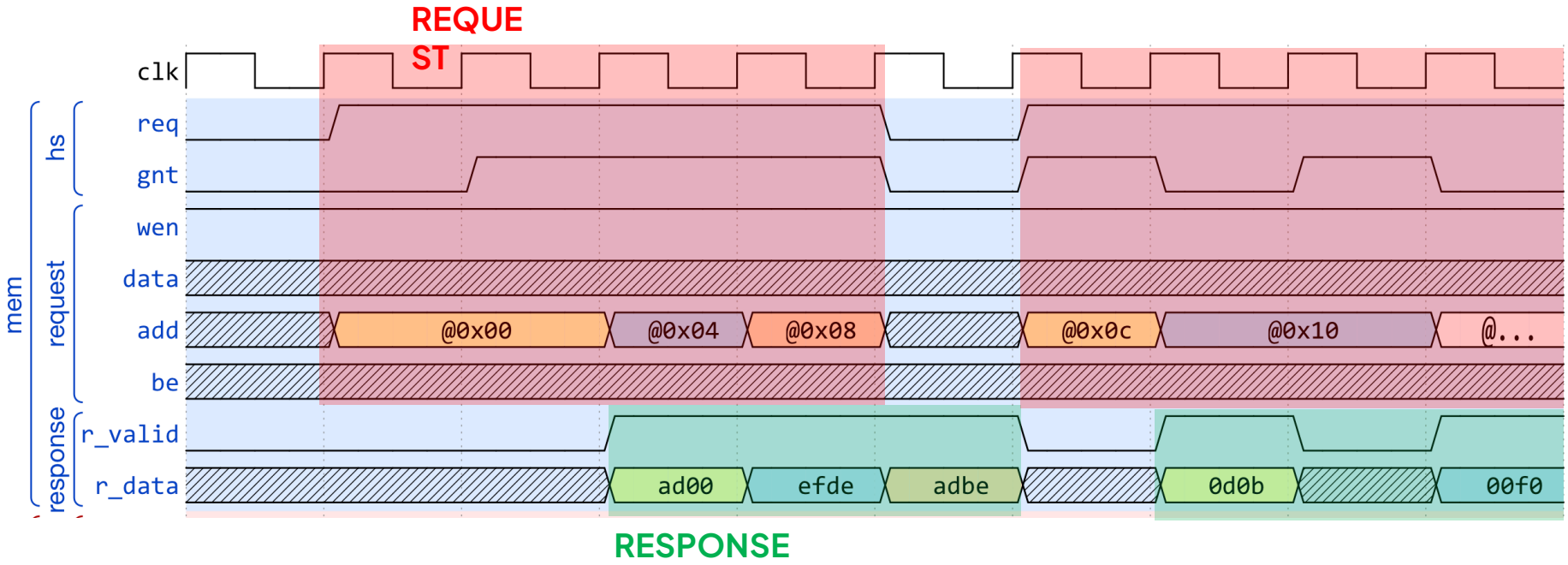
PULP LIC (log-interconnect)

- Still simple (no bursts, no out-of-order outstanding transactions) but faster than APB



PULP LIC (log-interconnect)

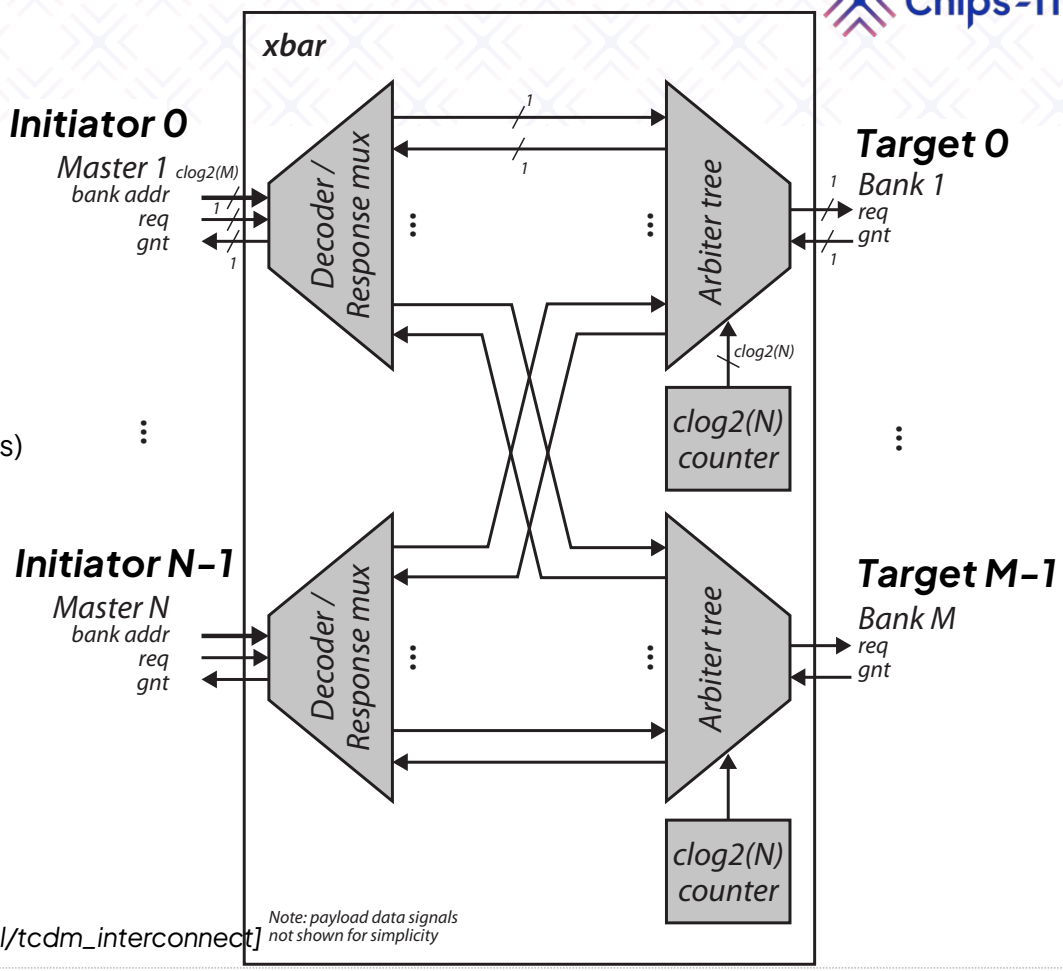
- Still simple (no bursts, no out-of-order outstanding transactions) but faster than APB



Crossbar interconnects

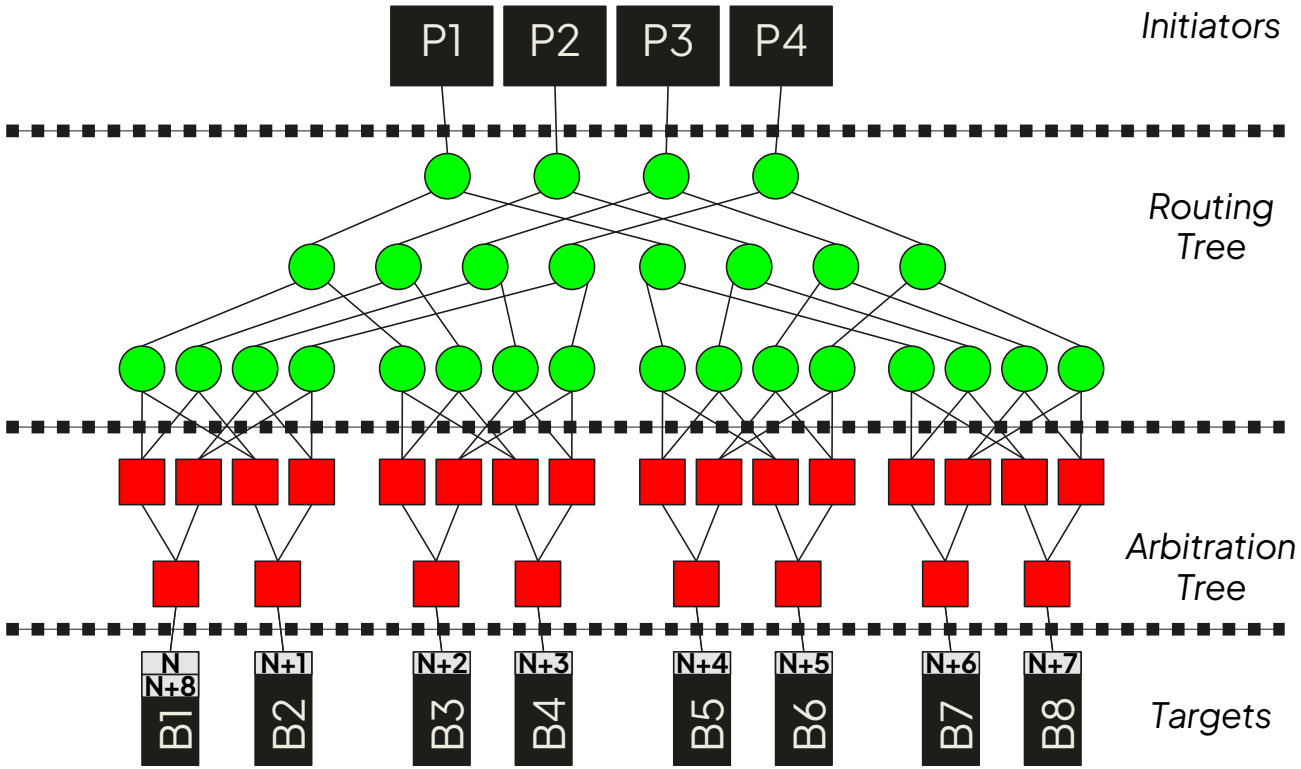
In a **full xbar** (**crossbar**):

- each initiator is **routed** to a target using the transaction **address**
- each target needs an ***arbitration tree*** to decide (arbiter) which initiator gets access
- **round-robin priority** (starvation free)
- **no extra collisions** introduced by xbar (only target conflicts)

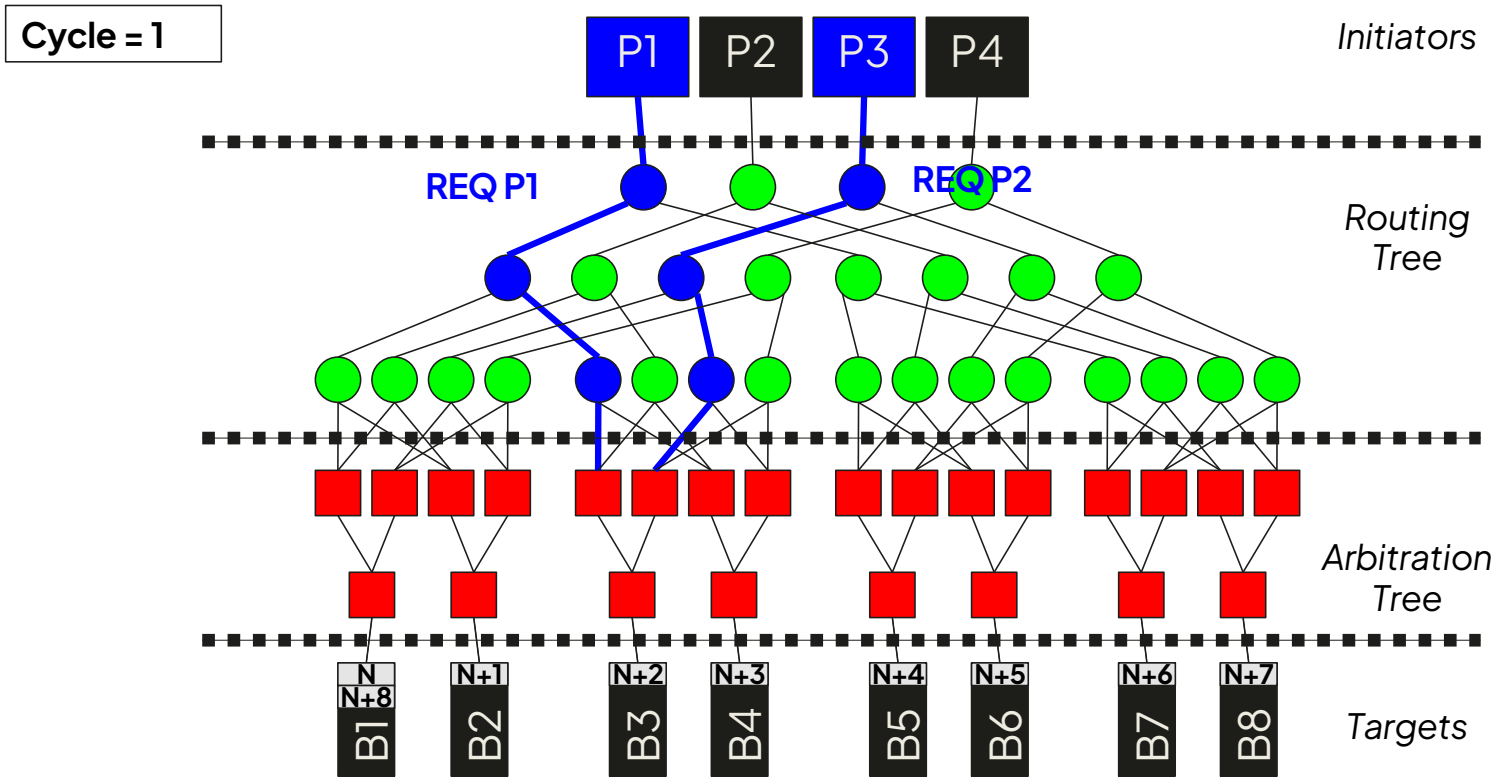


[https://github.com/pulp-platform/cluster_interconnect/tree/master/rtl/tcdm_interconnect]

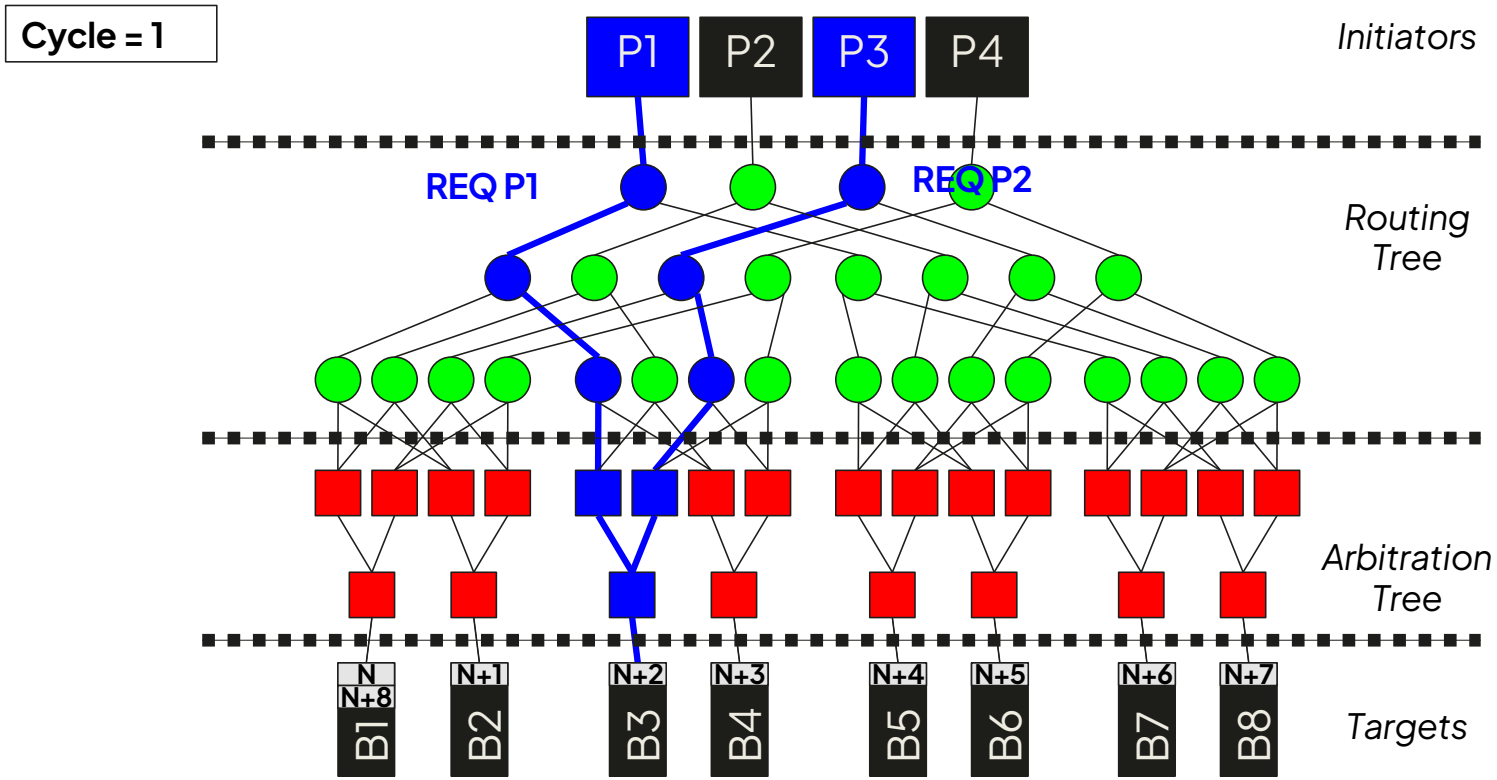
PULP LIC on a xbar



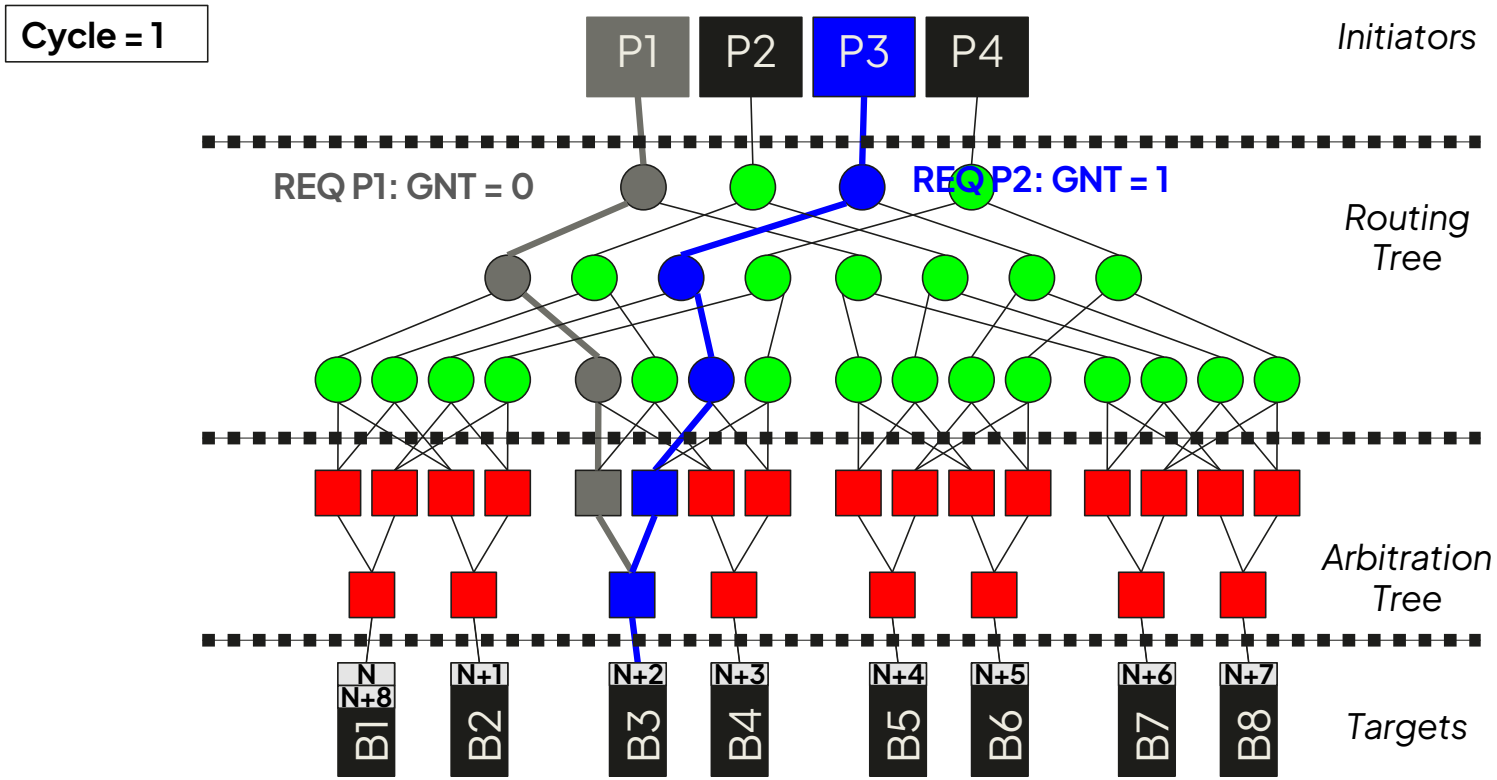
PULP LIC on a xbar



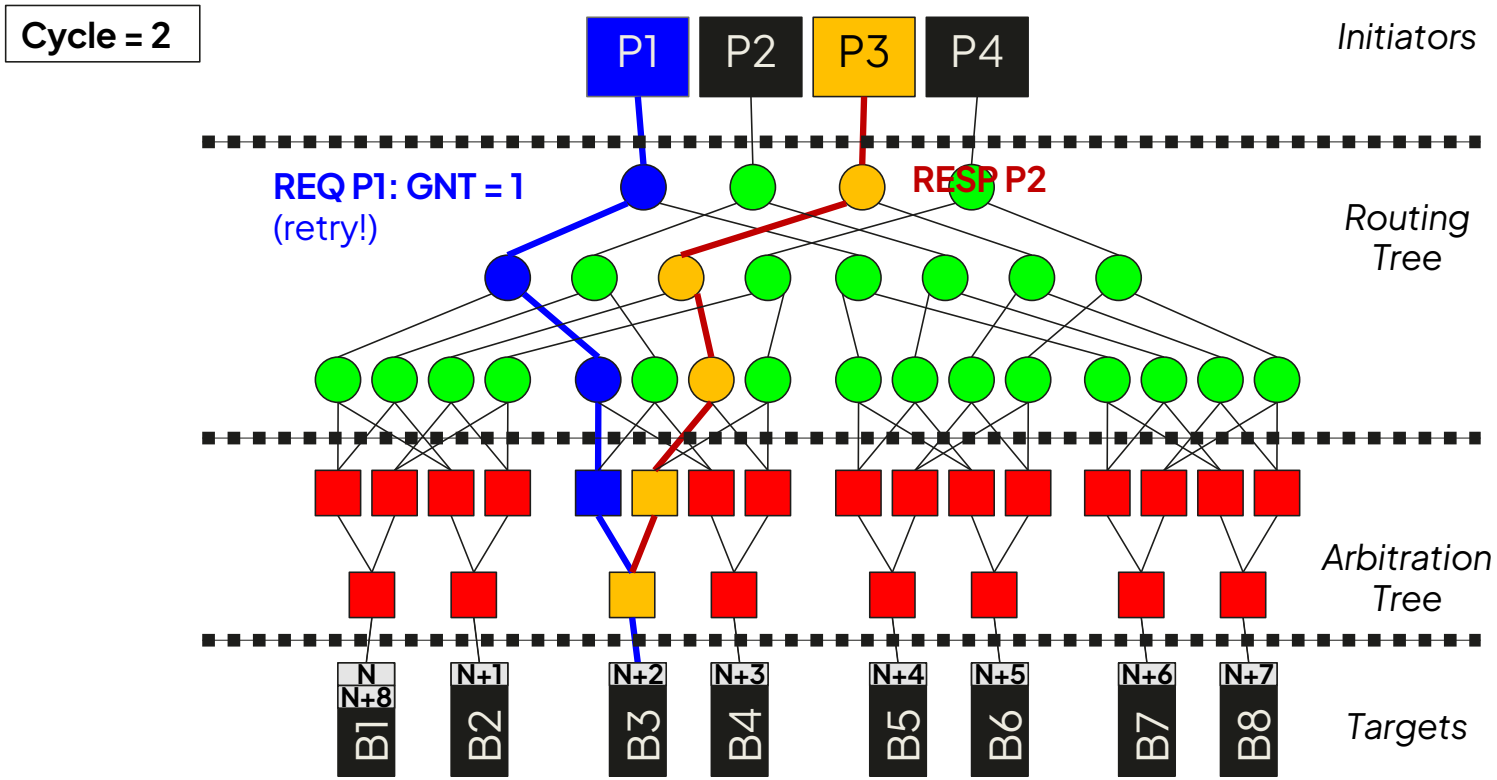
PULP LIC on a xbar



PULP LIC on a xbar

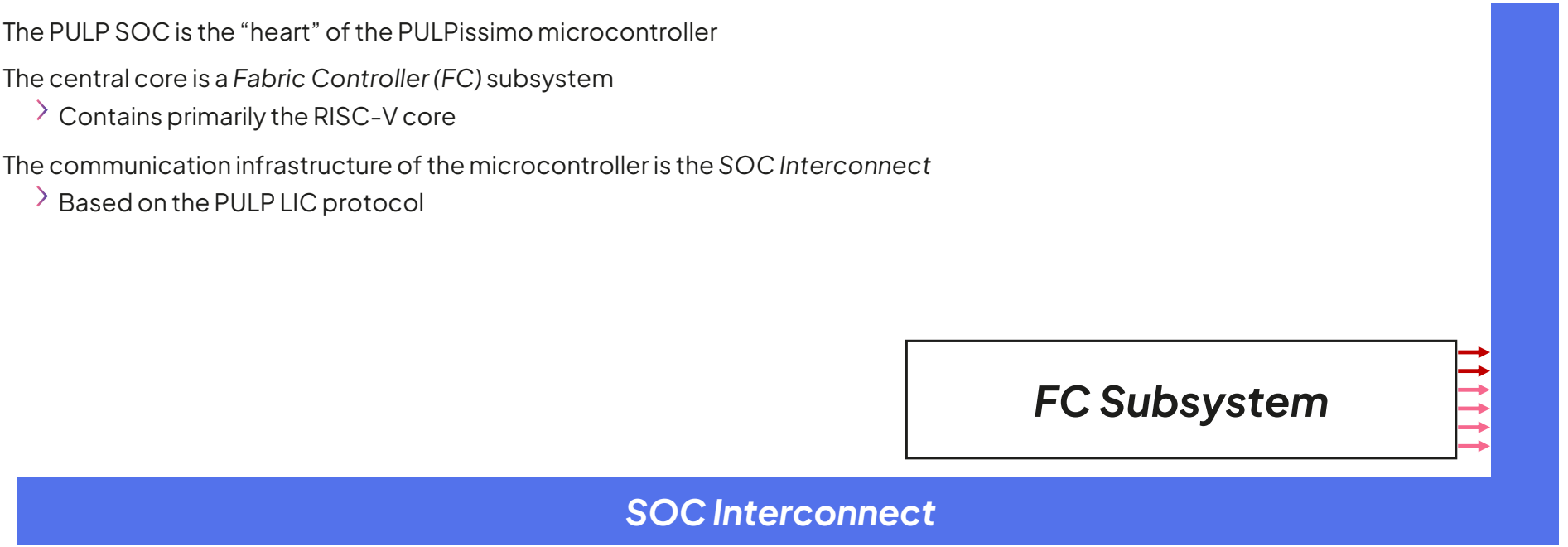


PULP LIC on a xbar



Opening the PULPissimo top-level box

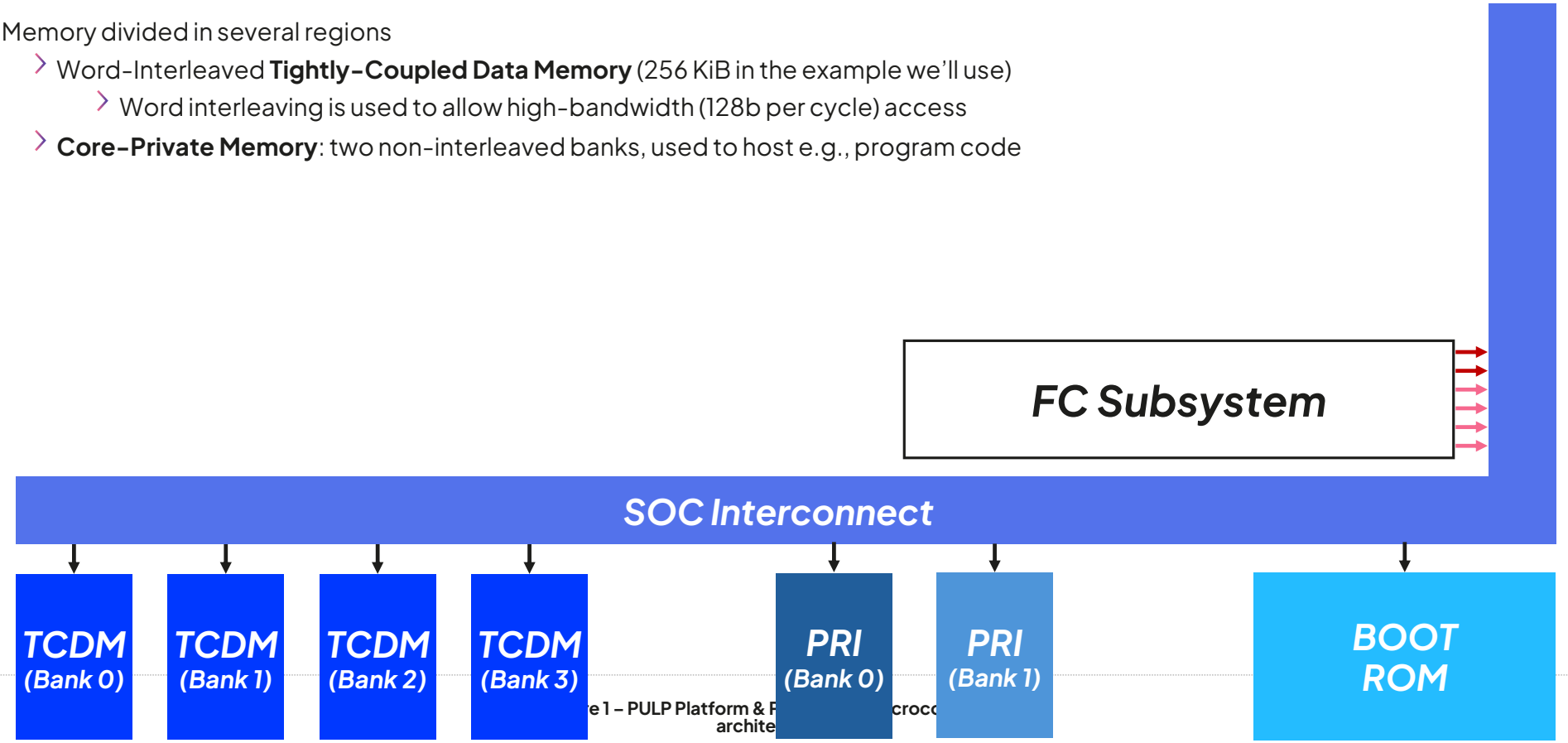
- The PULP SOC is the “heart” of the PULPissimo microcontroller
- The central core is a *Fabric Controller (FC)* subsystem
  - Contains primarily the RISC-V core
- The communication infrastructure of the microcontroller is the *SOC Interconnect*
  - Based on the PULP LIC protocol



Opening the PULPissimo top-level box

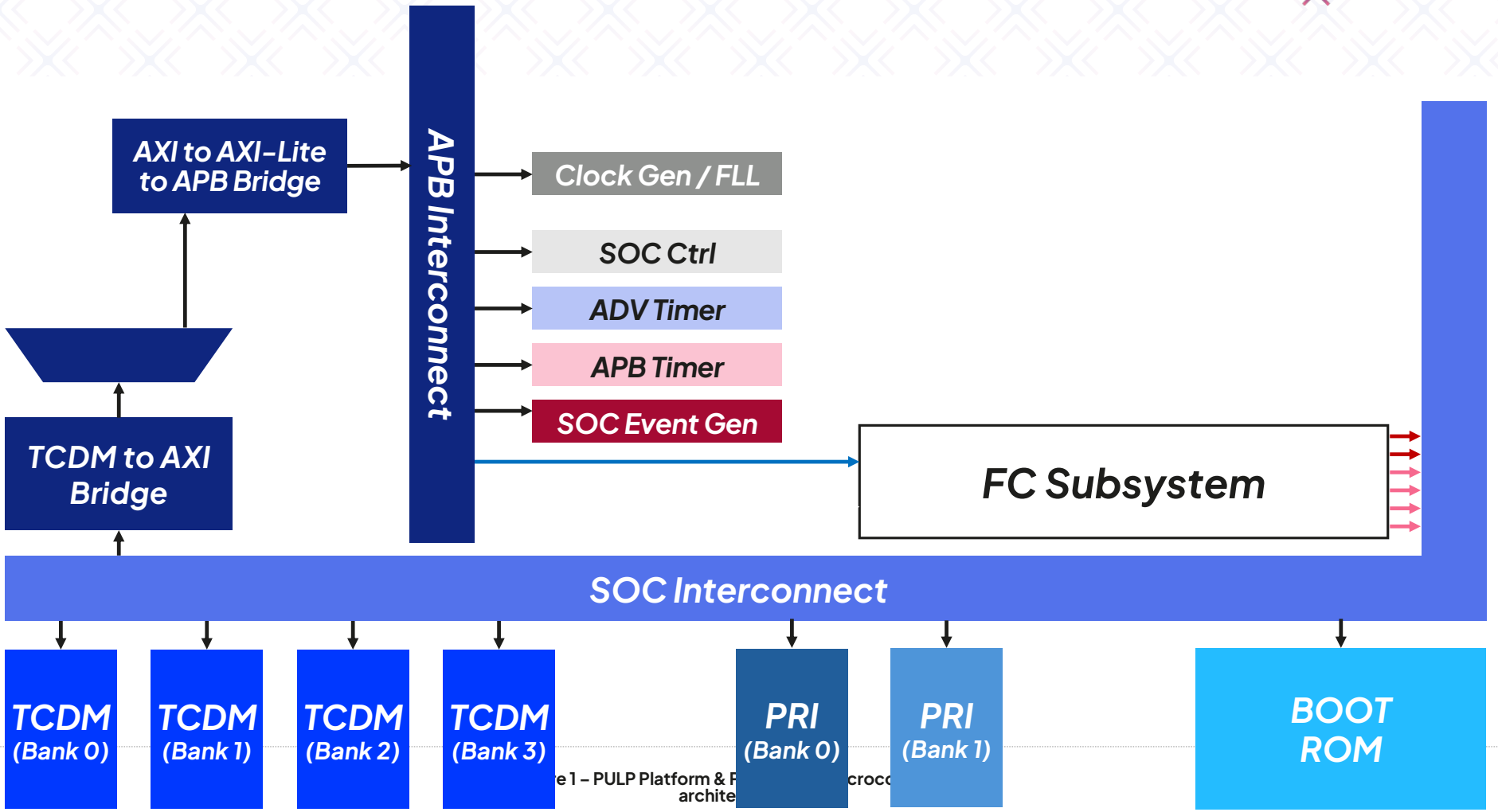
Chips-IT

- Memory divided in several regions
  - Word-Interleaved **Tightly-Coupled Data Memory** (256 KiB in the example we'll use)
    - Word interleaving is used to allow high-bandwidth (128b per cycle) access
  - **Core-Private Memory**: two non-interleaved banks, used to host e.g., program code

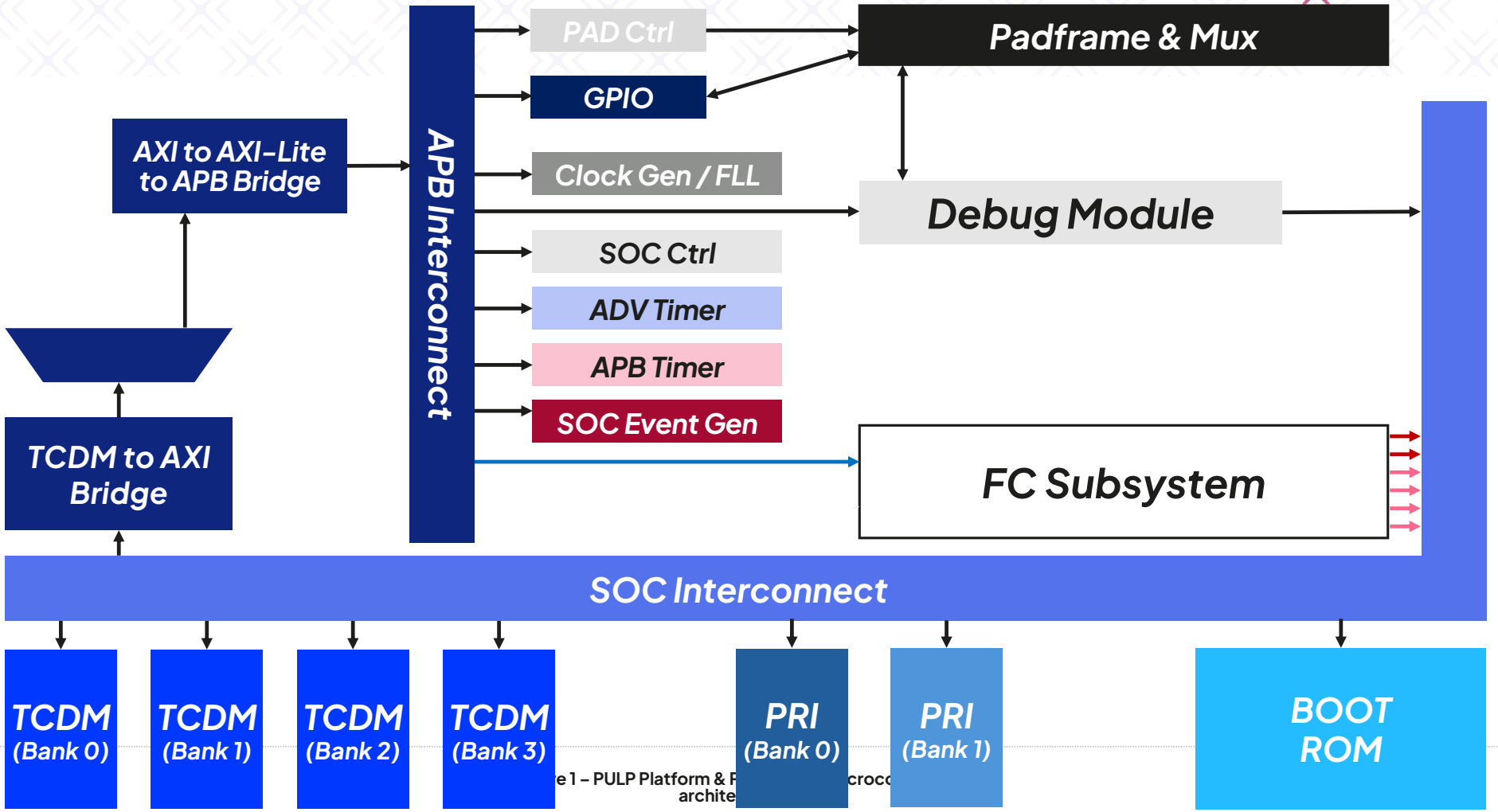


PULP SOC

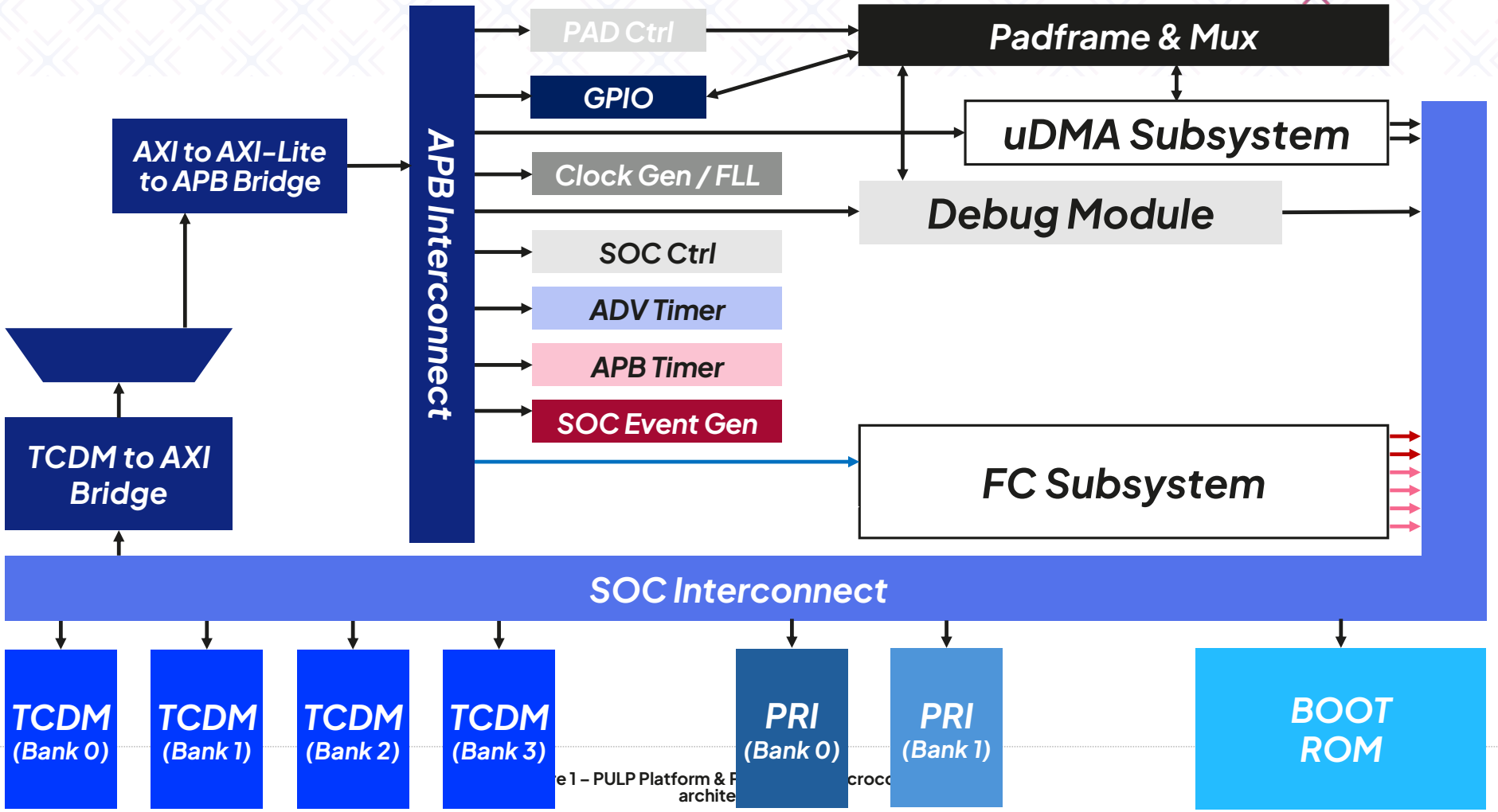
Chips-IT



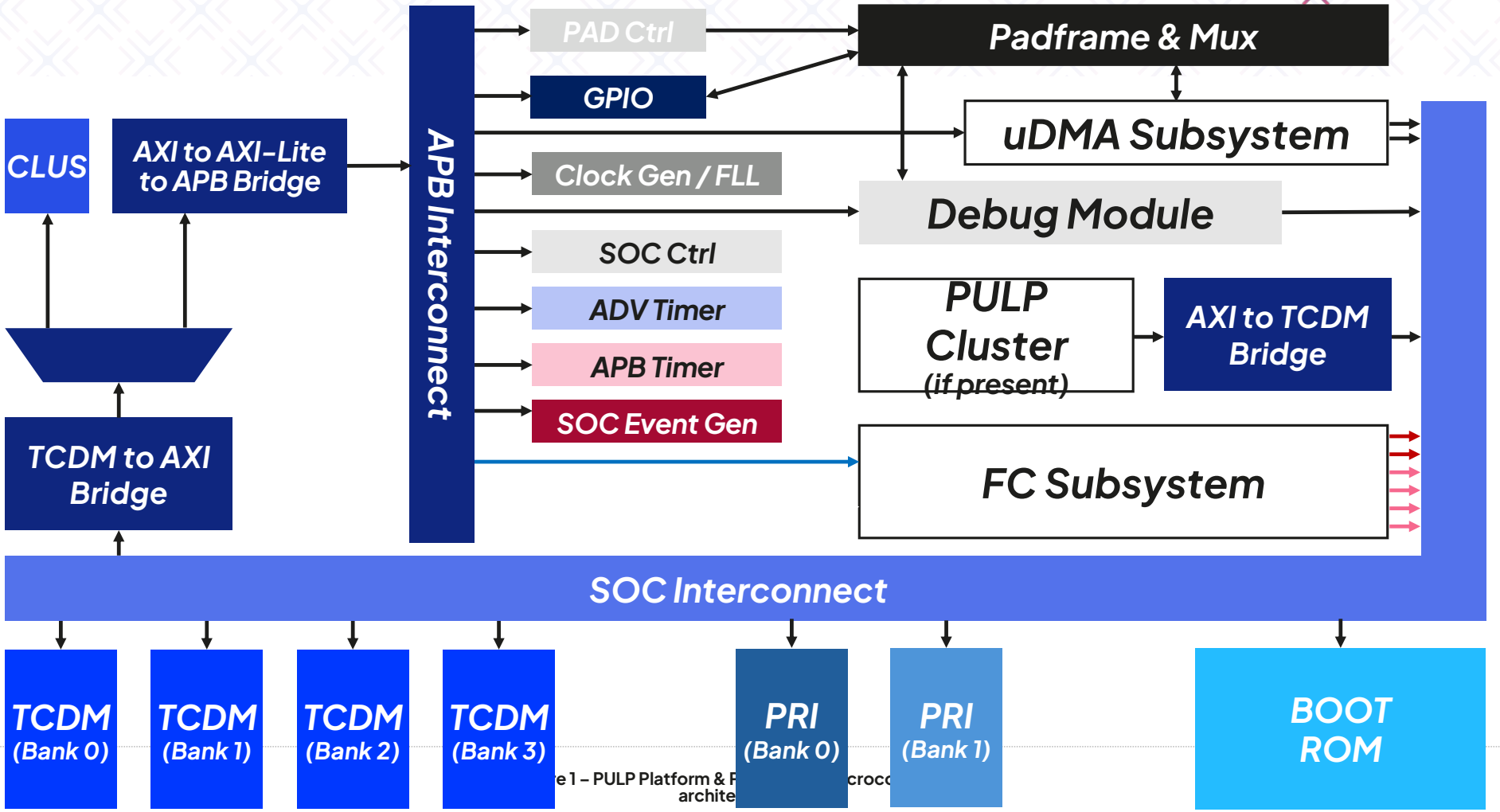
PULP SOC


Chips-IT
PAD Ctrl
GPIO
Padframe & Mux
AXI to AXI-Lite to APB Bridge
APB Interconnect
Clock Gen / FLL
Debug Module
SOC Ctrl
ADV Timer
APB Timer
SOC Event Gen
TCDM to AXI Bridge
FC Subsystem
SOC Interconnect
TCDM (Bank 0)
TCDM (Bank 1)
TCDM (Bank 2)
TCDM (Bank 3)
PRI (Bank 0)
PRI (Bank 1)
BOOT ROM

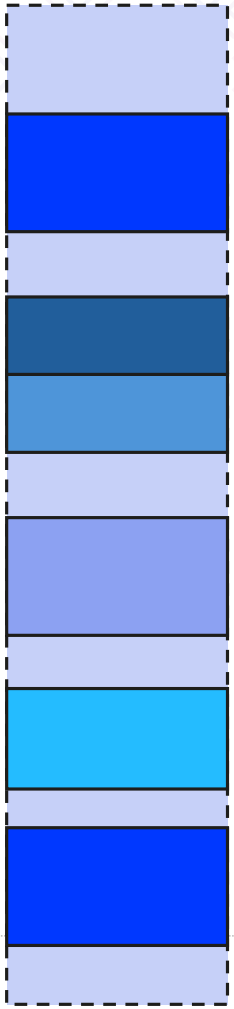
PULP SOC


Chips-IT
PAD Ctrl
GPIO
Clock Gen / FLL
SOC Ctrl
ADV Timer
APB Timer
SOC Event Gen
APB Interconnect
AXI to AXI-Lite to APB Bridge
TCDM to AXI Bridge
Padframe & Mux
uDMA Subsystem
Debug Module
FC Subsystem
SOC Interconnect
TCDM (Bank 0)
TCDM (Bank 1)
TCDM (Bank 2)
TCDM (Bank 3)
PRI (Bank 0)
PRI (Bank 1)
BOOT ROM

PULP SOC


Chips-IT
PAD Ctrl
GPIO
Clock Gen / FLL
SOC Ctrl
ADV Timer
APB Timer
SOC Event Gen
APB Interconnect
Padframe & Mux
uDMA Subsystem
Debug Module
PULP Cluster (if present)
AXI to TCDM Bridge
FC Subsystem
CLUS
AXI to AXI-Lite to APB Bridge
TCDM to AXI Bridge
SOC Interconnect
TCDM (Bank 0)
TCDM (Bank 1)
TCDM (Bank 2)
TCDM (Bank 3)
PRI (Bank 0)
PRI (Bank 1)
BOOT ROM

TCDM (Interleaved Memory)



0x1c09_0000

TCDM (Interleaved Memory)

0x1c01_0000

0x1c00_8000

Core-privileged Memory Bank 1

0x1c00_0000

Core-privileged Memory Bank 0

0x1a40_0000

SOC Peripherals

0x1a10_0000

0x1a04_0000

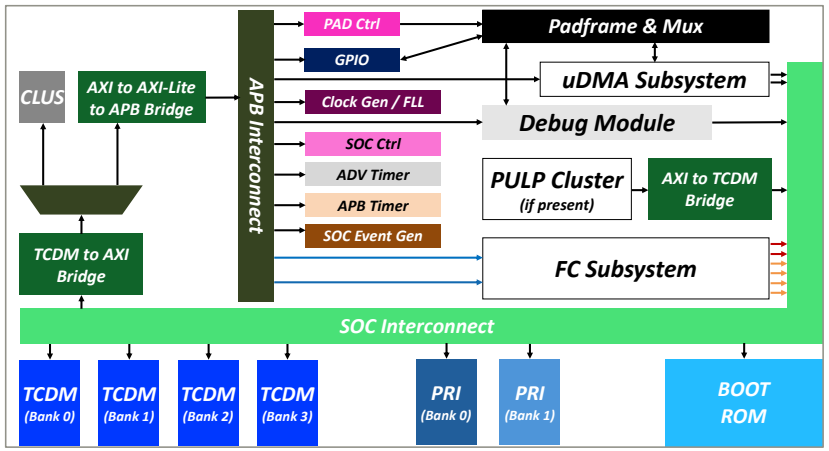
Boot ROM

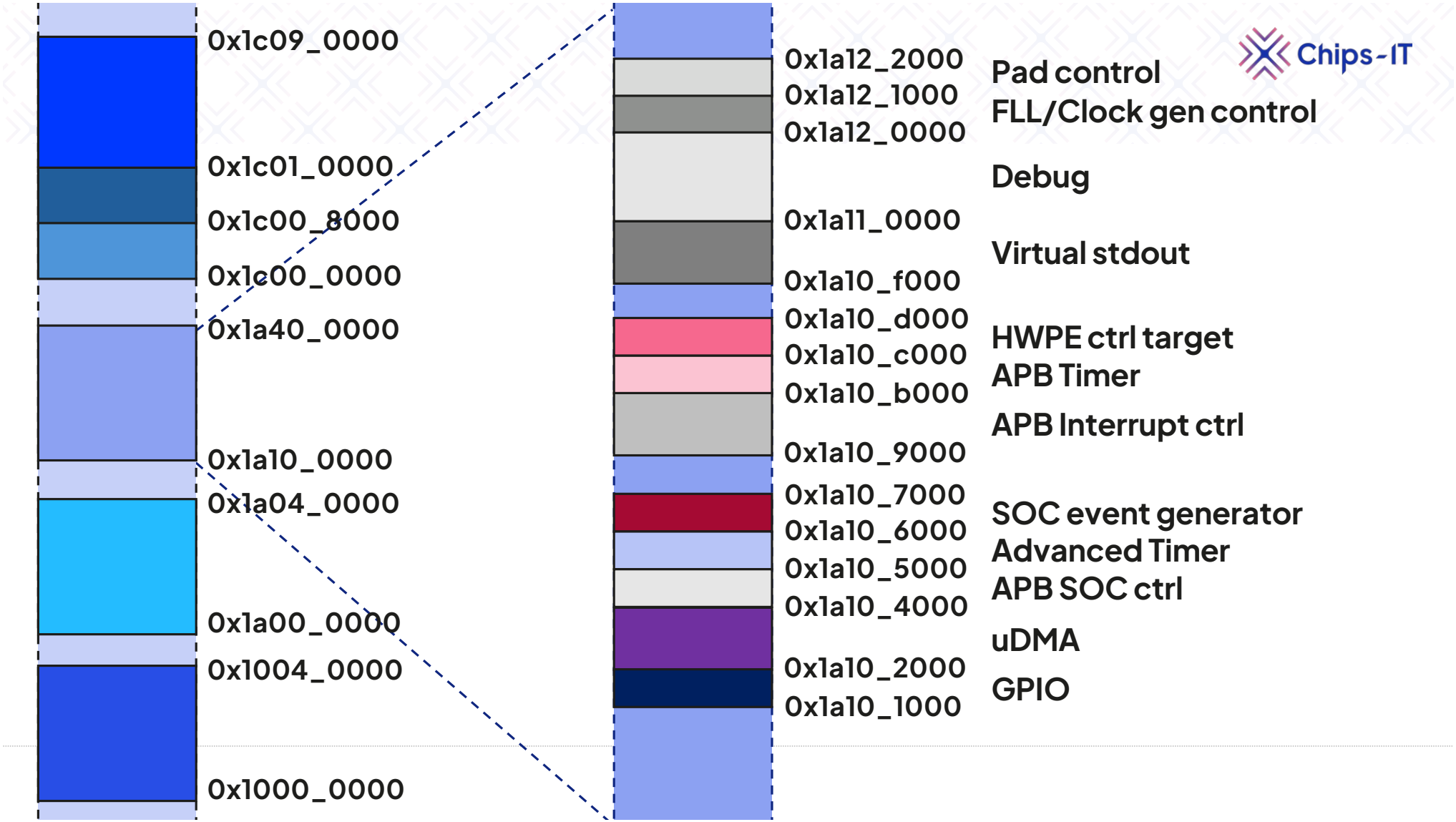
0x1a00_0000

0x1004_0000

PULP Cluster (if present)

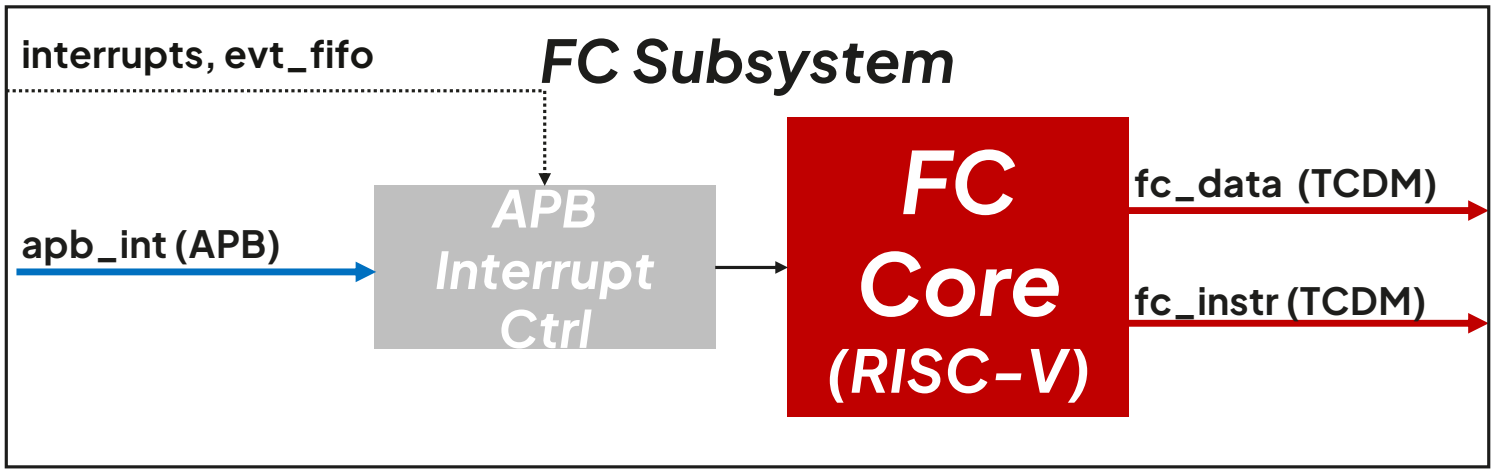
0x1000_0000




0x1c09_0000
0x1c01_0000
0x1c00_8000
0x1c00_0000
0x1a40_0000
0x1a10_0000
0x1a04_0000
0x1a00_0000
0x1004_0000
0x1000_0000
0x1a12_2000
0x1a12_1000
0x1a12_0000
0x1a11_0000
0x1a10_f000
0x1a10_d000
0x1a10_c000
0x1a10_b000
0x1a10_9000
0x1a10_7000
0x1a10_6000
0x1a10_5000
0x1a10_4000
0x1a10_2000
0x1a10_1000
Pad control
FLL/Clock gen control
Debug
Virtual stdout
HWPE ctrl target
APB Timer
APB Interrupt ctrl
SOC event generator
Advanced Timer
APB SOC ctrl
uDMA
GPIO
Chips-IT

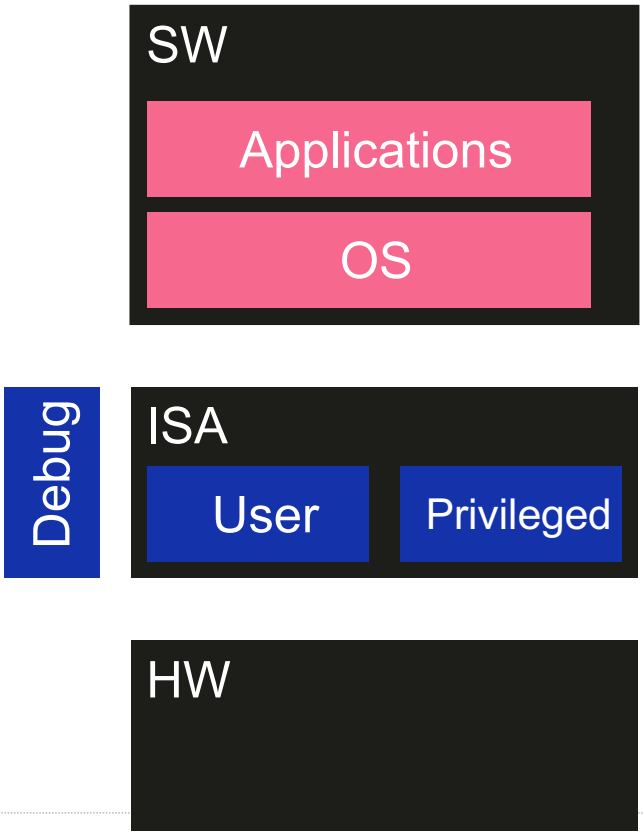
Fabric Controller Subsystem

- The *Fabric Controller* subsystem contains the RISC-V core of PULPissimo
- Three available microarchitectures, all based on the RISC-V ISA
  - RI5CY / CV32E40P
  - Ibex
  - **CV32E40X** (we'll use this latter in this course)

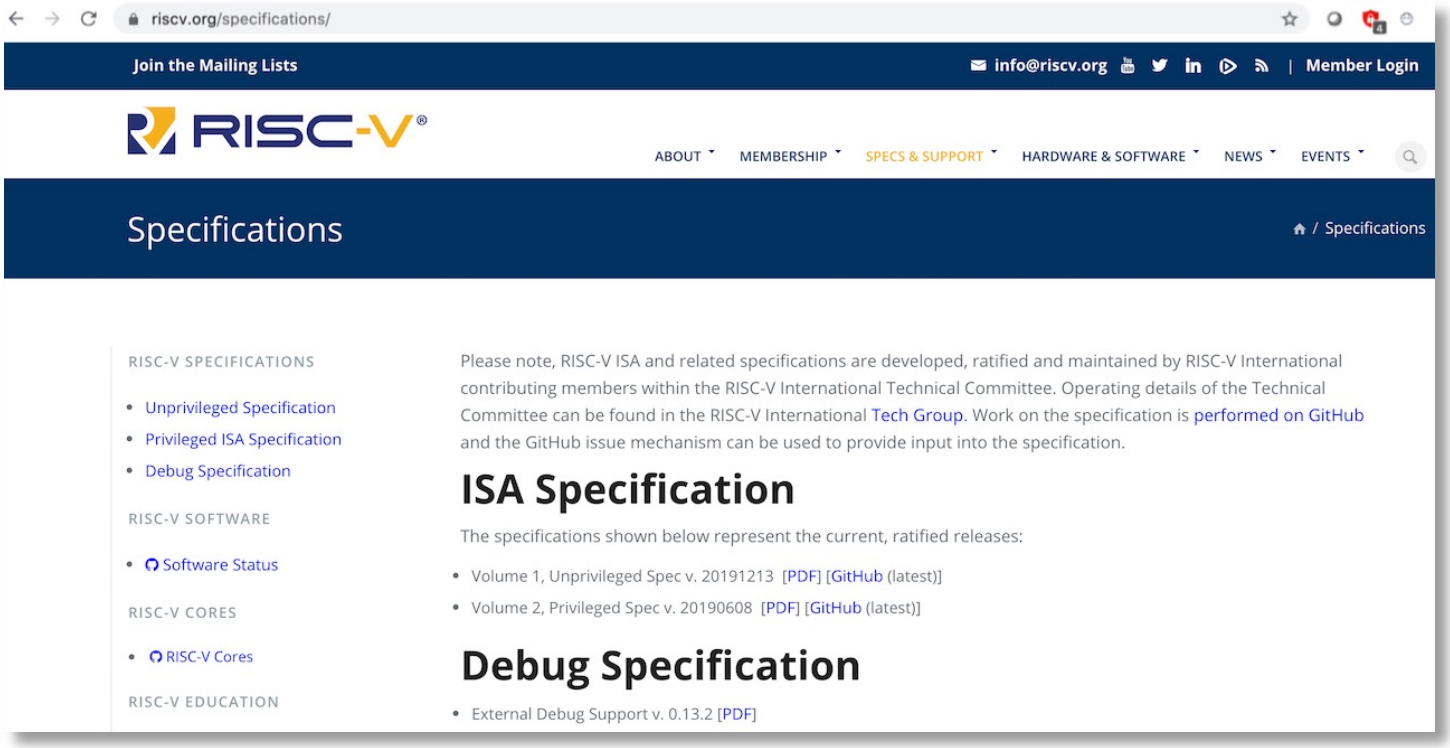


RISC-V Instruction Set Architecture

- Started by UC-Berkeley in 2010
- Contract between SW and HW
  - Partitioned into user and privileged spec
  - External Debug
- Standard governed by RISC-V foundation
  - ETHZ is a founding member of the foundation
  - Necessary for the continuity
- Defines 32, 64 and 128 bit ISA
  - No implementation, just the ISA
  - Different implementations (both open and close source)
- At ETHZ+UNIBO we specialize in efficient implementations of RISC-V cores



RISC-V maintains basically a PDF document



ISA defines the instructions that processor uses

Chips-IT

C++ source #1

```
1  void gemm_bin(int M, int N, int K, float ALPHA,
2          char  *A, int lda,
3          float *B, int ldb,
4          float *C, int ldc)
5  {
6      int i,j,k;
7      for(i = 0; i < M; ++i){
8          for(k = 0; k < K; ++k){
9              char A_PART = A[i*lda+k];
10             if(A_PART){
11                 for(j = 0; j < N; ++j){
12                     C[i*ldc+j] += B[k*ldb+j];
13                 }
14             } else {
15                 for(j = 0; j < N; ++j){
16                     C[i*ldc+j] -= B[k*ldb+j];
17                 }
18             }
19         }
20     }
21 }
22
```

RISC-V rv64gc clang (trunk) (Editor #1, Compiler #1) C++

RISC-V rv64gc clang (trunk) | Compiler options... | Output... | Filter... | Libraries | Add new... | Add tool...

```
63  .LBB0_7:                                # in Loop: Header=BB0_6 D
64          ld      a0, -56(s0)
65          lw      a1, -76(s0)
66          lw      a2, -60(s0)
67          mul     a1, a1, a2
68          lw      a2, -72(s0)
69          addw    a1, a1, a2
70          slli    a1, a1, 2
71          add     a0, a0, a1
72          flw     ft0, 0(a0)
73          ld      a0, 0(s0)
74          lw      a1, -68(s0)
75          lw      a3, -64(s0)
76          mul     a1, a1, a3
77          addw    a1, a1, a2
78          slli    a1, a1, 2
79          add     a0, a0, a1
80          flw     ft1, 0(a0)
81          fadd.s  ft0, ft1, ft0
82          fsw     ft0, 0(a0)
```

C++ program translated to RISC-V instructions defined by ISA.

This will run on ANY RISC-V implementation

COMPILER EXPLORER

RISC-V Ecosystem

Chips-IT

- Binutils – upstream
- GCC – upstream
- LLVM – upstream
- Simulator:
  - ”Spike” - reference
  - QEMU, Gem5
- OpenOCD
- OS
  - Linux, sel4, freeRTOS, zephyr
- Runtimes
  - Jikes, Ocaml, Go
- SW maintained by different parties
  - Binutils and GCC by SiFive (UCBerkeley start-up)

RISC-V ISA is divided into extensions

I	Integer instructions (frozen)
E	Reduced number of registers
M	Multiplication and Division (frozen)
A	Atomic instructions (frozen)
F	Single-Precision Floating-Point (frozen)
D	Double-Precision Floating-Point (frozen)
C	Compressed Instructions (frozen)
X	Non Standard Extensions

- **Kept very simple and extendable**
  - Wide range of applications from IoT to HPC
- **RV + word-width + extensions**
  - RV32IMC: 32bit, integer, multiplication, compressed
- **User specification:**
  - Separated into extensions, only **I** is mandatory
- **Privileged Specification (WIP):**
  - Governs OS functionality: Exceptions, Interrupts
  - Virtual Addressing
  - Privilege Levels

Work continues on new RISC-V extensions

- **Foundation members work in task-groups**
- **Dedicated task-groups**
  - Formal specification
  - Memory Model
  - Marketing
  - External Debug Specification
- **ETH Zurich also contributes**
  - Bit manipulation
  - Packed SIMD, DSP

Q	Quad-precision Floating-Point
L	Decimal Floating Point
B	Bit Manipulation
T	Transactional Memory
P	Packed SIMD
J	Dynamically Translated Languages
V	Vector Operations
N	User-Level Interrupts

What is so special about RISC-V

- It is FR
  - E
- It is a m
  - S
    N
  - R
  - M
- Reserv
- Open

RISC-V base ISAs have either little-endian or big-endian memory systems, with the privileged architecture further defining bi-endian operation. Instructions are stored in memory as a sequence of 16-bit little-endian parcels, regardless of memory system endianness. Parcels forming one instruction are stored at increasing halfword addresses, with the lowest-addressed parcel holding the lowest-numbered bits in the instruction specification.

We originally chose little-endian byte ordering for the RISC-V memory system because little-endian systems are currently dominant commercially (all x86 systems; iOS, Android, and Windows for ARM). A minor point is that we have also found little-endian memory systems to be more natural for hardware designers. However, certain application areas, such as IP networking,

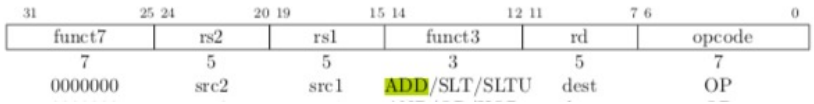
The FREEDOM in RISC-V is implementation

- **You can access all ISAs without (many) restrictions**
  - SW tools need to be developed so that they can generate code for that ISA
- **Most ISAs are closed. Only specific vendors can implement it**
  - To use a core that implements an ISA, you have to license/buy it from vendor
  - Open source SW (for the ISA) is possible but **building HW is not allowed**

RISC-V

Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



ARM

C2.9 ADD

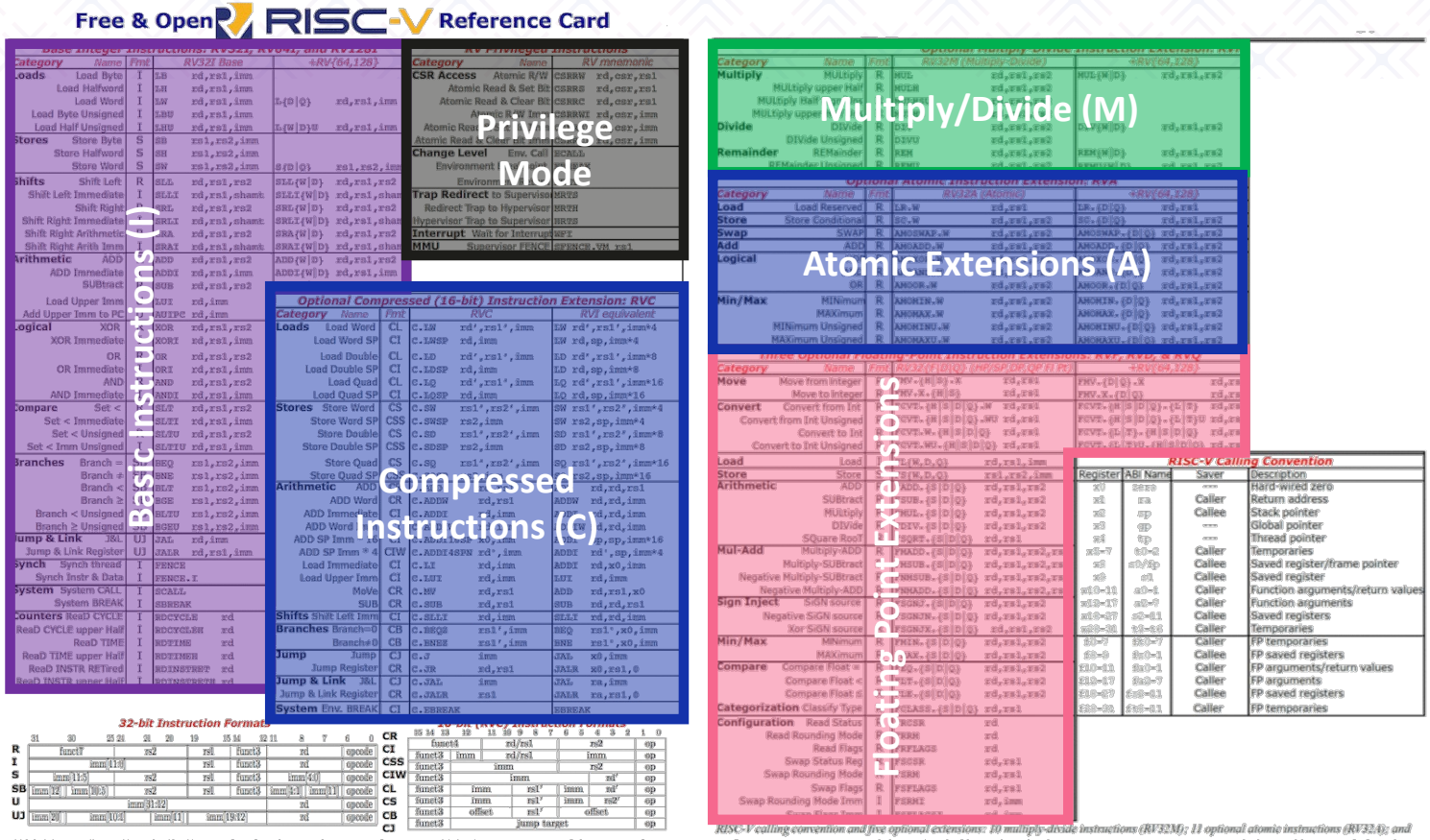
Add without Carry.

Syntax

```
ADD{S}{cond} {Rd}, Rn, Operand2
ADD{cond} {Rd}, Rn, #imm12 ; T32, 32-bit encoding only
```

RISC-V: all in one page

Chips-IT



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, {} means set, so L{D|Q} is both LD and LQ. See risc.org. (5/21/15 revision)

RISC-V compiler & PULPissimo runtime

- **(Many) more details on the RISC-V ISA in tomorrow's lecture on ISA extensions**
- **For now we'll focus on the essential SW architecture for PULPissimo**
  - *Compiler*: we adopt LLVM v18, but any baseline RISC-V **RV32IMC** compiler will do as we use a 32-bit core (CV32E40X)
    - GCC is also a possible option
  - *Runtime environment*: we use the minimal PULP runtime, a very simple SDK that can be used to write small bare-metal test programs (not suitable for application development)
    - Linker script + crt0.s + Makefile rules to easily build code for PULPissimo

A look at crt0.s

- Code @ address `instr_base`
  - **`_start`**: initial operations of the microcontroller
    - sets up the stack pointer
    - clears bss
    - calls the **`main`** function

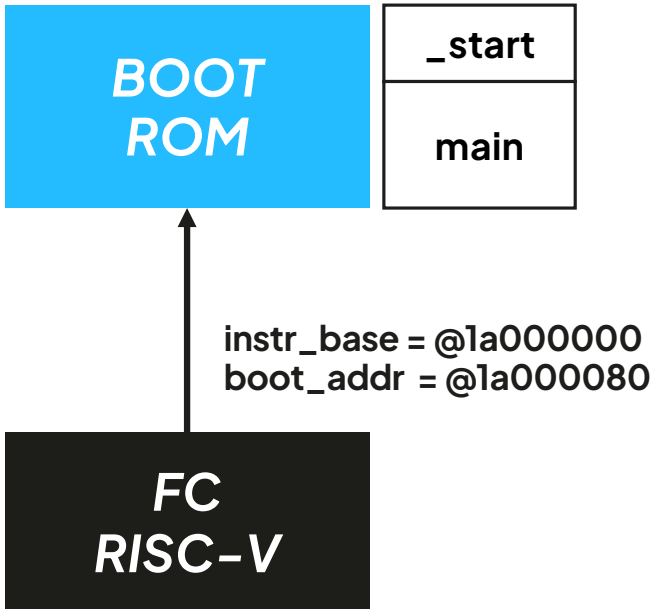
```
#include "config.h"
#include "archi/pulp.h"

.section .text

_start:
/* initialize global pointer */
.option push
.option norelax
1:auipc gp, %pcrel_hi(__global_pointer$)
addi gp, gp, %pcrel_lo(1b)
.option pop

/* init stack pointer */
la sp, __stack_top

/* clear the bss segment */
la t0, __bss_start
la t1, __bss_end
1:
sw zero,0(t0)
addi t0, t0, 4
bltu t0, t1, 1b

/* call main */
li a0, 0 /* a0 = argc */
addi a1, sp, 4 /* a1 = argv */
li a2, 0 /* a2 = envp = NULL */
call main
tail exit
```

A look at crt0.s

- Code @ address `instr_base`
  - **`_start`**: initial operations of the microcontroller
    - sets up the stack pointer
    - clears bss
    - calls the **`main`** function
  - *interrupt vector table* from `0x0` to `0x80`
    - in default crt0.s, all but one point to a **`default_handler`**, which just returns
    - at address `0x80` the initial reset vector: a simple `jal` to the **`_start`** symbol
  - **`loop`**: when the program returns from **`main`**, it will end in this infinite loop
  - **`change_stack`**: move to another stack pointer (e.g., in boot from SPI Flash)

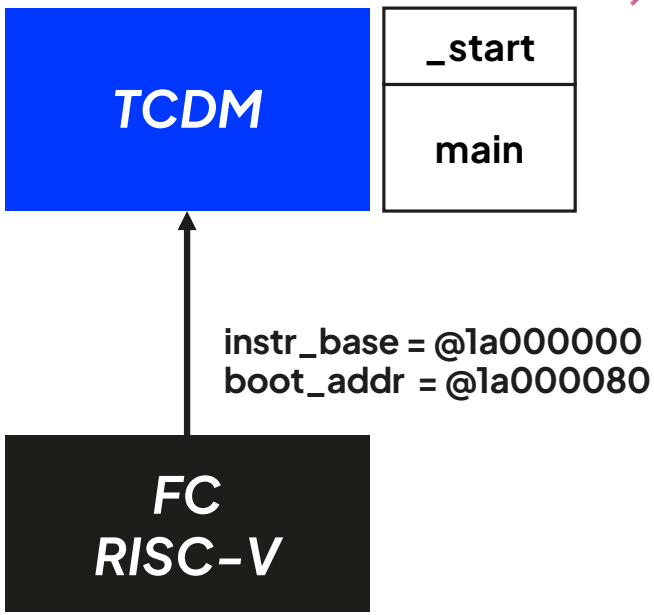
```
.global change_stack
change_stack:
        mv sp, a2
        jr a1

.section .vectors, "ax"
.option norvc;
.org 0x0
jal x0, default_handler
/* ... */
jal x0, default_handler

/* reset vector */
.org 0x80
jal x0, _start
.org 0x84

loop:
        j loop

default_handler:
        mret
```

PULPissimo boot procedure (ROM + Flash)



- **Boot from the boot ROM**
  - Start from **`_start`** symbol
  - Jump to a **`main`** within the boot ROM itself
    - copy program code from SPI Flash, HyperFlash, etc. to TCDM
    - jump to the entry point (using **`change_stack`**)

PULPissimo boot procedure (JTAG / simul. preload)



- **Boot from the boot ROM**
  - Start from **`_start`** symbol
  - Jump to a **`main`** within the TCDM
  - This is what we use in our RTL simulation!

And now... practice!

Chips-IT

- **The hands-on material + the Teaching Assistants will lead you in your first experiments on the PULPissimo platform**
  - Introduction to the PULPissimo simulation environment
    - You will download and learn to build and use the PULPissimo environment using a simple application a-la-helloworld
    - Running an helloworld already uses much of the functionality presented today, and a bit more (especially from the I/O viewpoint)
  - Develop a simple application (FIR filter) and evaluate its performance
    - If you are rusty with C, a good occasion to get productive again!
    - Good preparation for what we do in the next few days
    - Test a few useful facilities: disassembly, tracing, waveform debugging

Thank you

Chips-IT