Application of MPSoC on the ATLAS L0Muon Sector Logic blade:

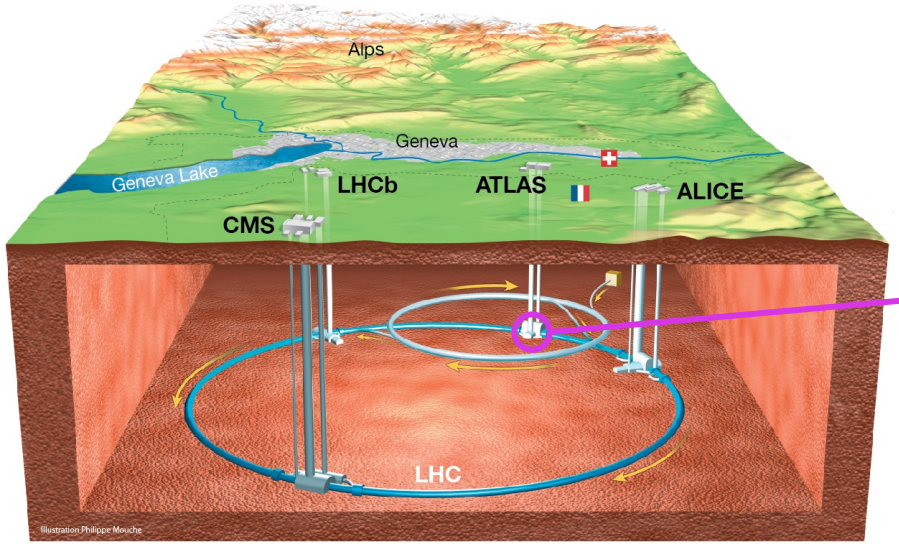
Control and single-board DAQ for commissioing

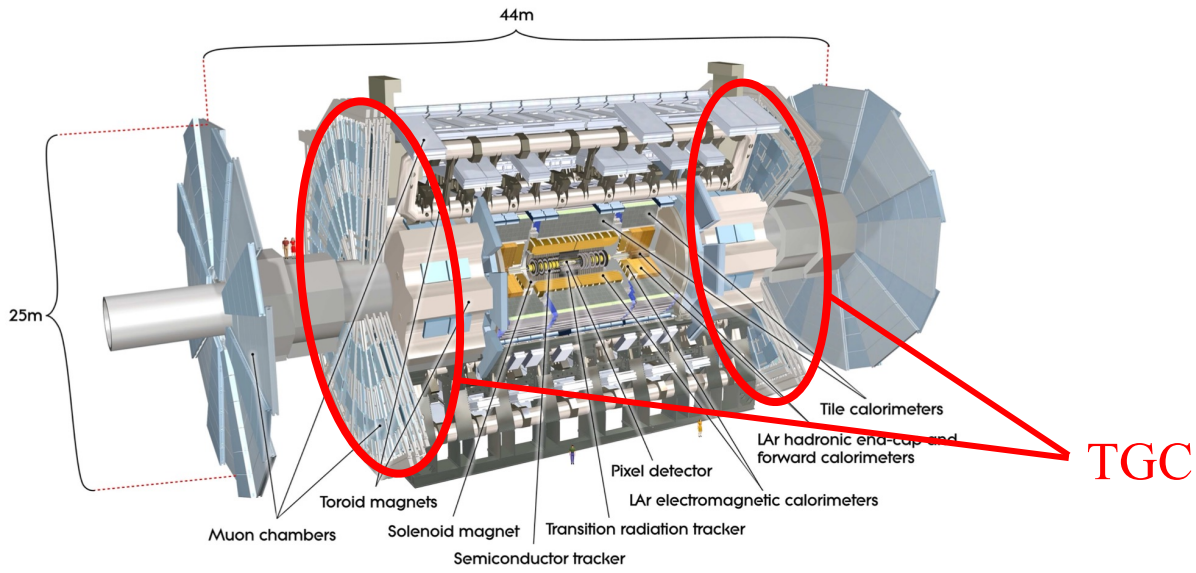
Akihiro Mishima, on behalf of TGC electronics group

The University of Tokyo, ICEPP

Phase-2 upgrade of ATLAS TGC

- TGC (Thin Gap Chamber)
  - Endcap muon trigger detector in $|\eta| > 1.05$ of ATLAS
  - Online muon reconstruction within the L0 latency budget (10 μs in total)
  - Renew Trigger & Readout electronics for phase-2 upgrade



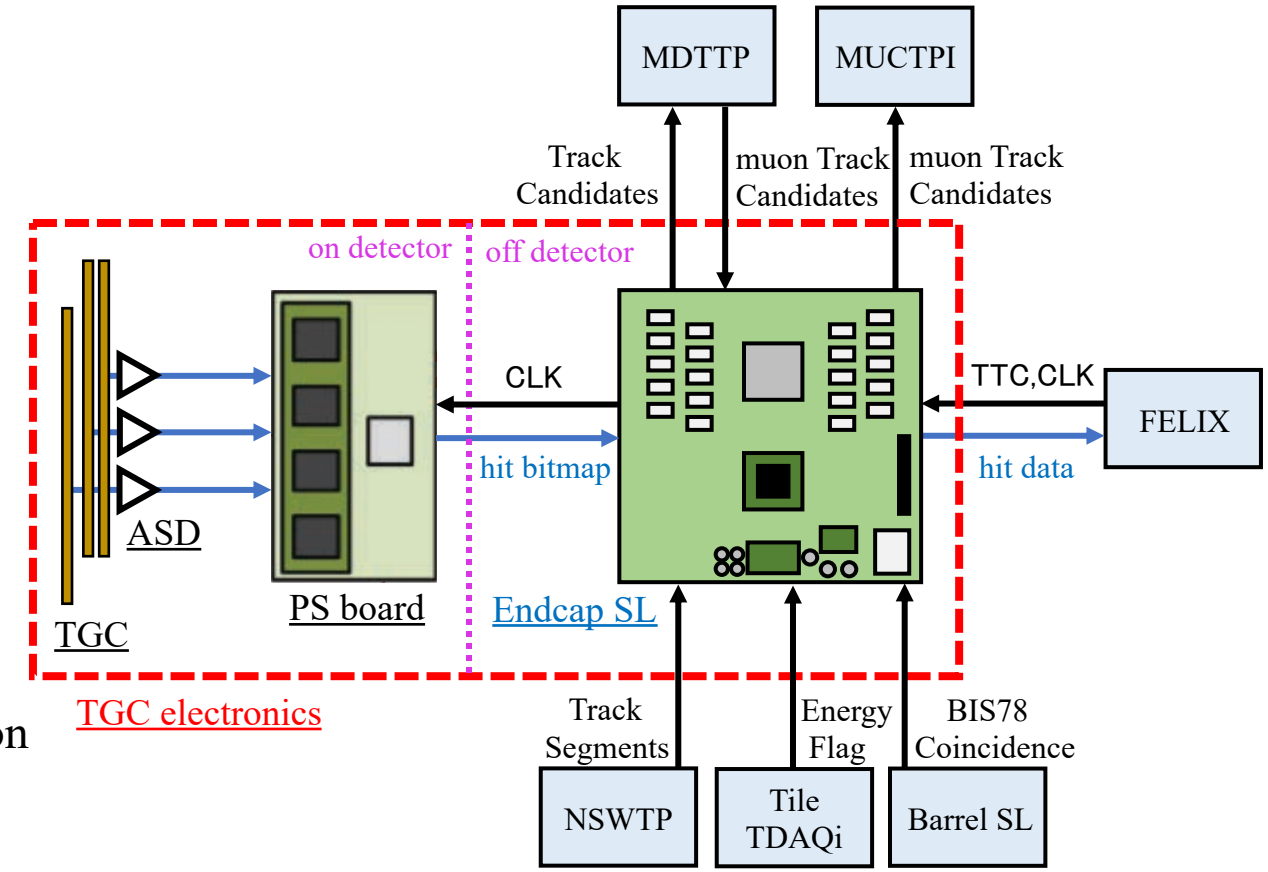


Phase-2 TGC electronics overview

- ASD (Amplifier – Shaper - Discriminator) (~23K boards)

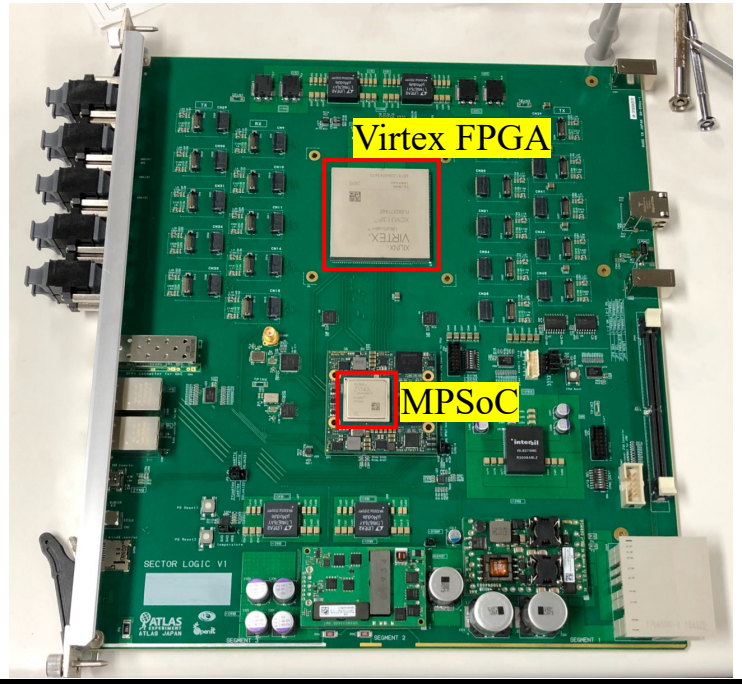
- PS board (Primary Processor board) (1434 boards)
  - Sends all hit bitmap to Endcap SL with fixed latency via optical links (8 Gbps x 2/PS board)

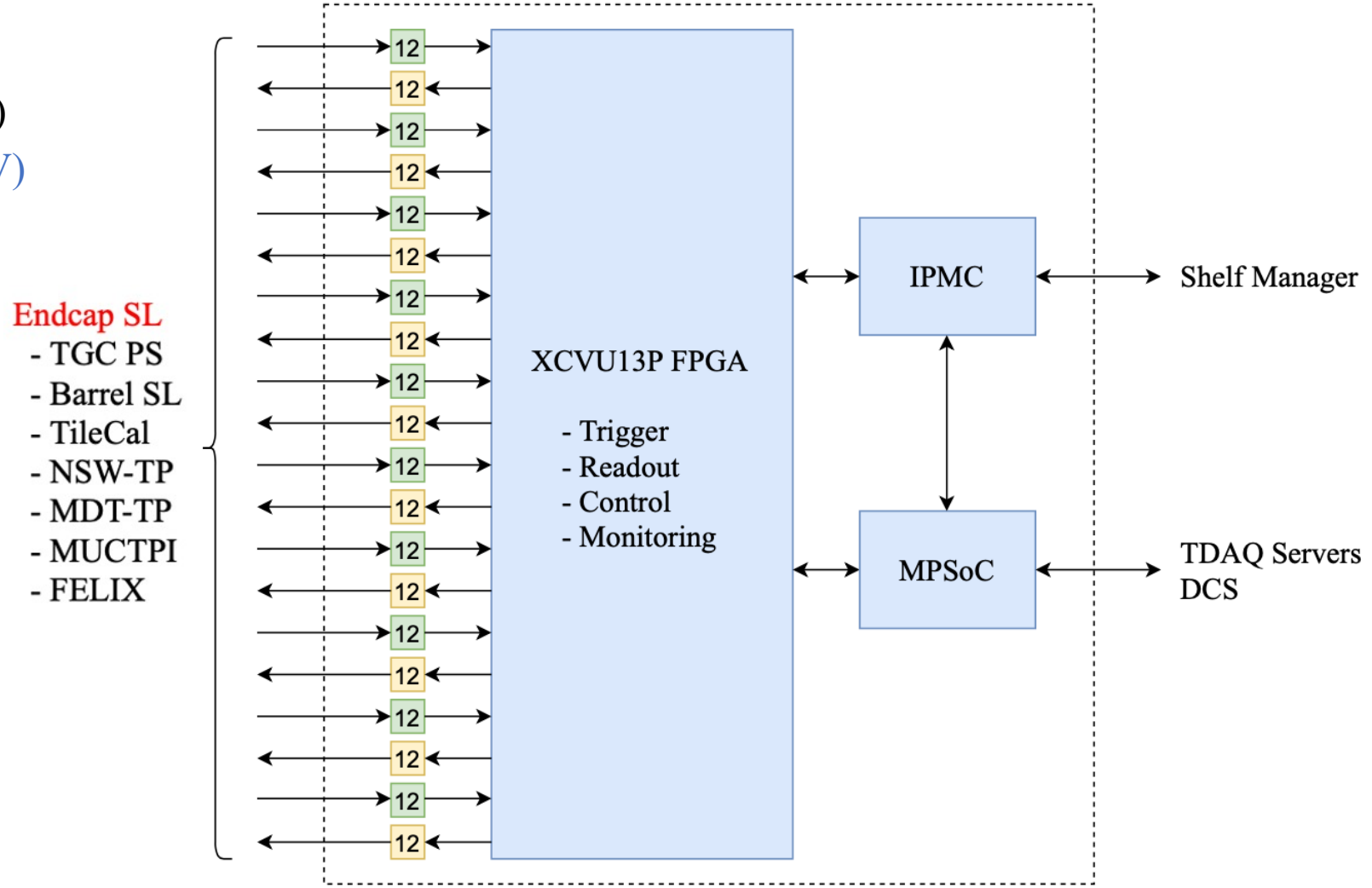
- Endcap SL (Endcap Sector Logic) (48 boards)
  - Performs muon track reconstruction and estimates p_T using hit signal from TGC BW
  - Improves p_T determination using information from other detectors (NSW, RPC BIS78, Tile calo, EIL4 TGC)
  - Sends Track Candidates to MDTTP
  - Receives muon Track Candidates with precise p_T and position from MDTTP
  - Sends muon Track Candidates to MUCTPI
  - Readouts hit data for each triggerd event to FELIX via optical links (9.6 Gbps x 4 available/SL board)



Endcap SL overview

- There are 48 boards in Endcap
- Each of them contains:
  - Virtex Ultrascale+ FPGA (XCVU13P)
  - Zynq Ultrascale+ MPSoC (XCZU5EV) (on Mercury XU5)
  - IPMC
  - FireFly TX (12 ch) x10
  - FireFly RX (12 ch) x10

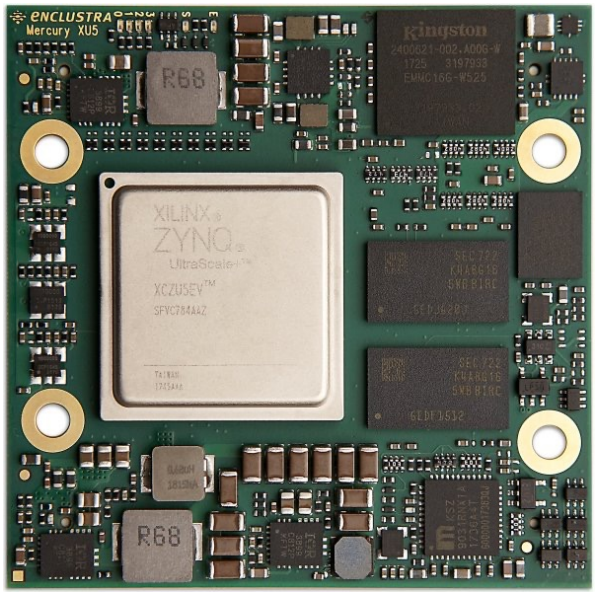


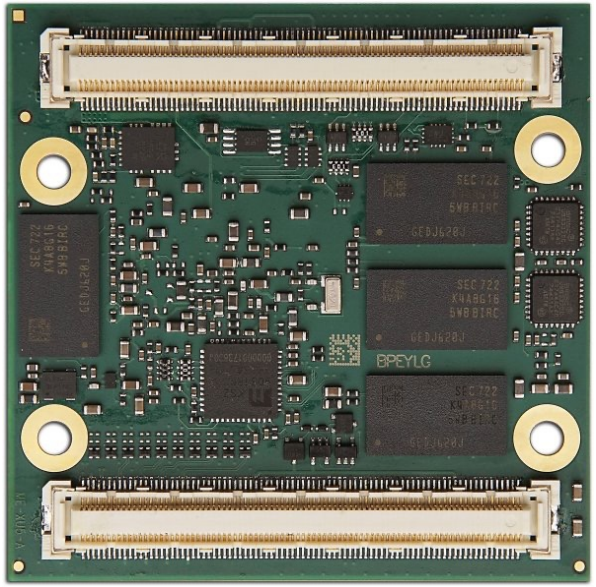


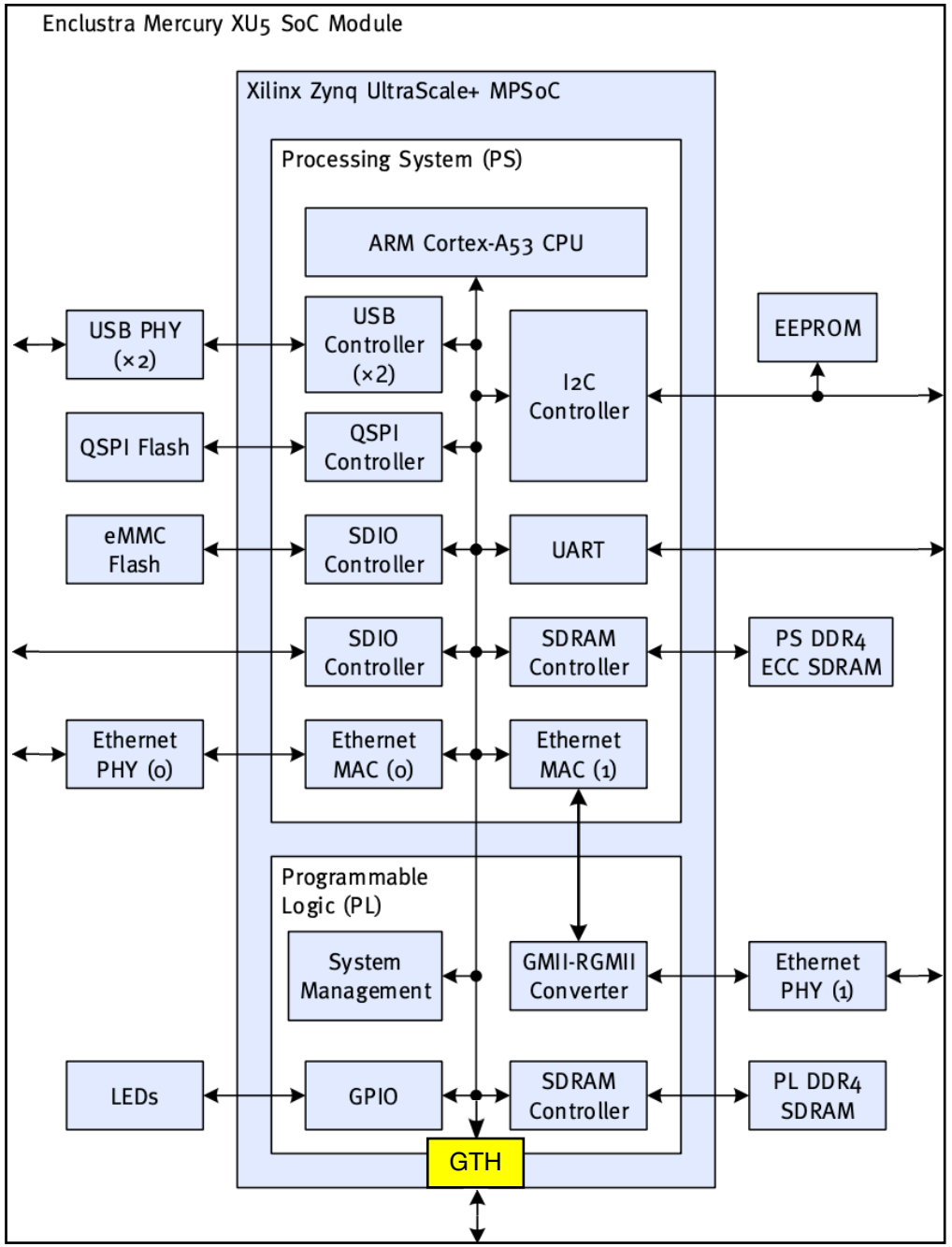
Mercury XU5 mezzanine

- The board has Mercury XU5 mezzanine distributed by Enclustra
  - Mezzanine: ME-XU5-5EV-2I-D12E
  - MPSoC tip: XCZU5EV-SFVC784-2-I

- Enclustra provides the reference design for the mezzanine: https://github.com/enclustra/Mercury_XU5_PE1_Reference_Design

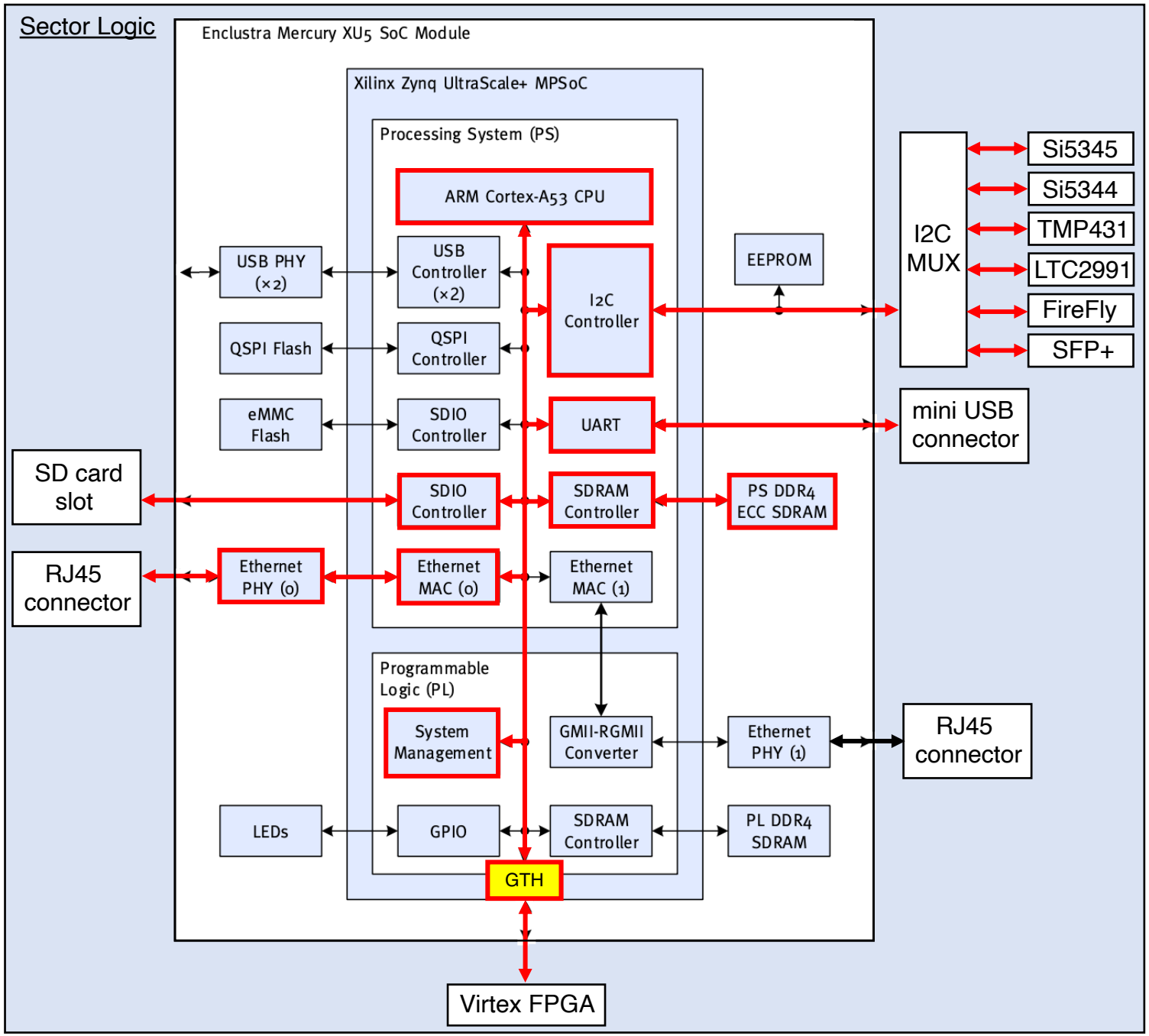






Mercury XU5 IO

- On the SL board, following components are used:
  - SDIO Controller
  - Ethernet MAC & PHY
  - I2C Controller
  - UART
  - SDRAM Controller & PS DDR4
  - GTH

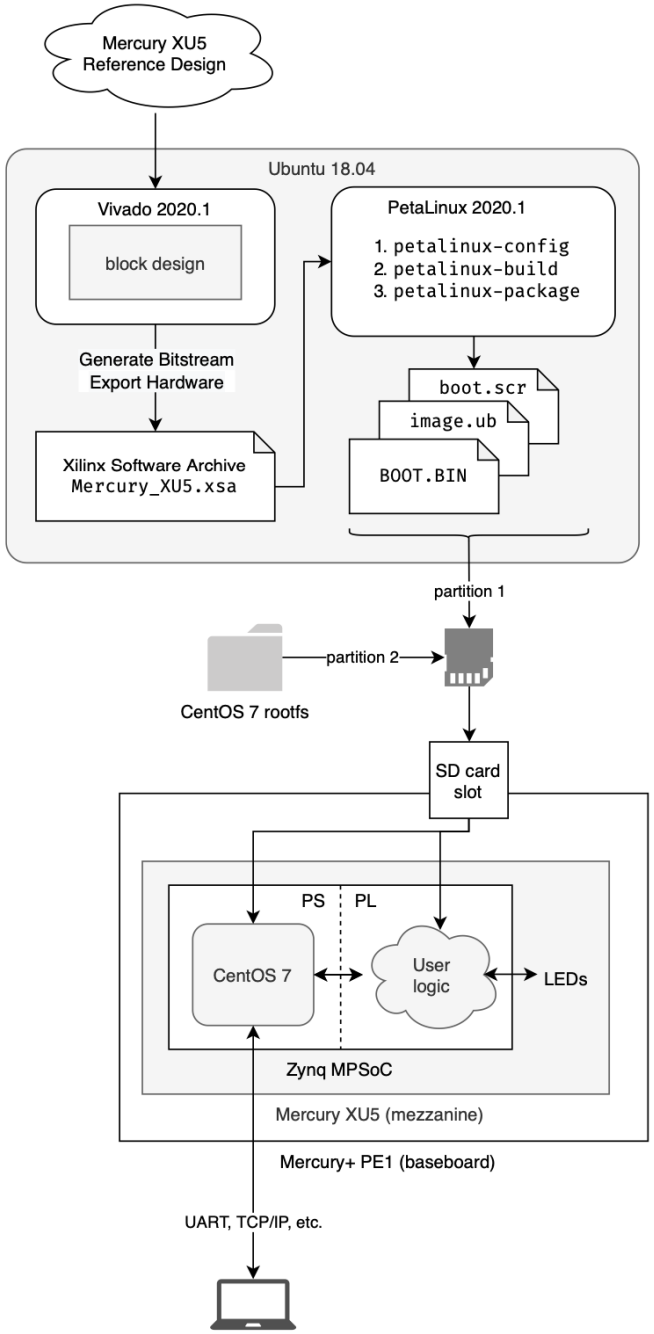


Development flow

- Create Block Design in Vivado 2020.1 taking advantage of the Mercury XU5 Reference Design

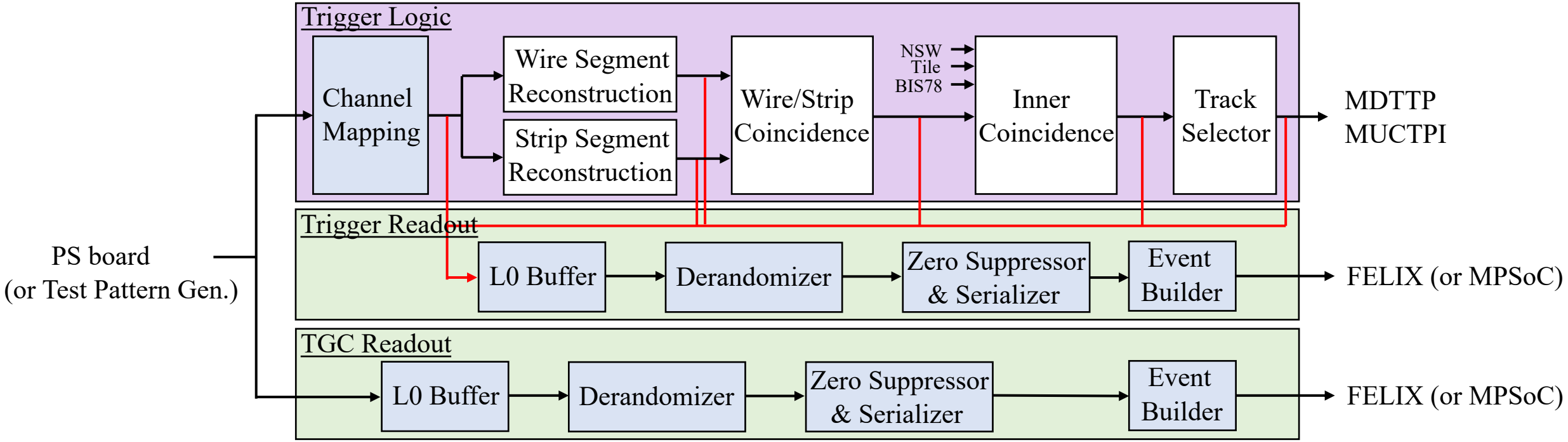
- Generate Bitstream and Export Hardware in the form of Xilinx Software Archive (.xsa)
- Using .xsa file as input of the PetaLinux 2020.1, perform config, build, and package of the boot files
  - boot.scr
  - image.ub
  - BOOT.BIN
  - uEnv.txt
    → Environment variables for U-Boot are set in the uEnv.txt

- Place boot files in partition-1 of SD card, and place CentOS 7 rootfs in partition-2
- Configuration will be automatically done after the power on
- We can access the CentOS via the TCP/IP
- Applications can be developed using compilers (e.g. gcc) on the CentOS



Endcap SL firmware development background

- The Readout firmware has been implemented and the Trigger firmware is being integrated in the Virtex FPGA
- The control path are being implemented



- Initial commissioning of the SL board is ongoing at the moment

→ Control path and Readout chain has been established taking advantage of the MPSoC

MPSoC studies

- XVC
- Control path
- Single-board DAQ

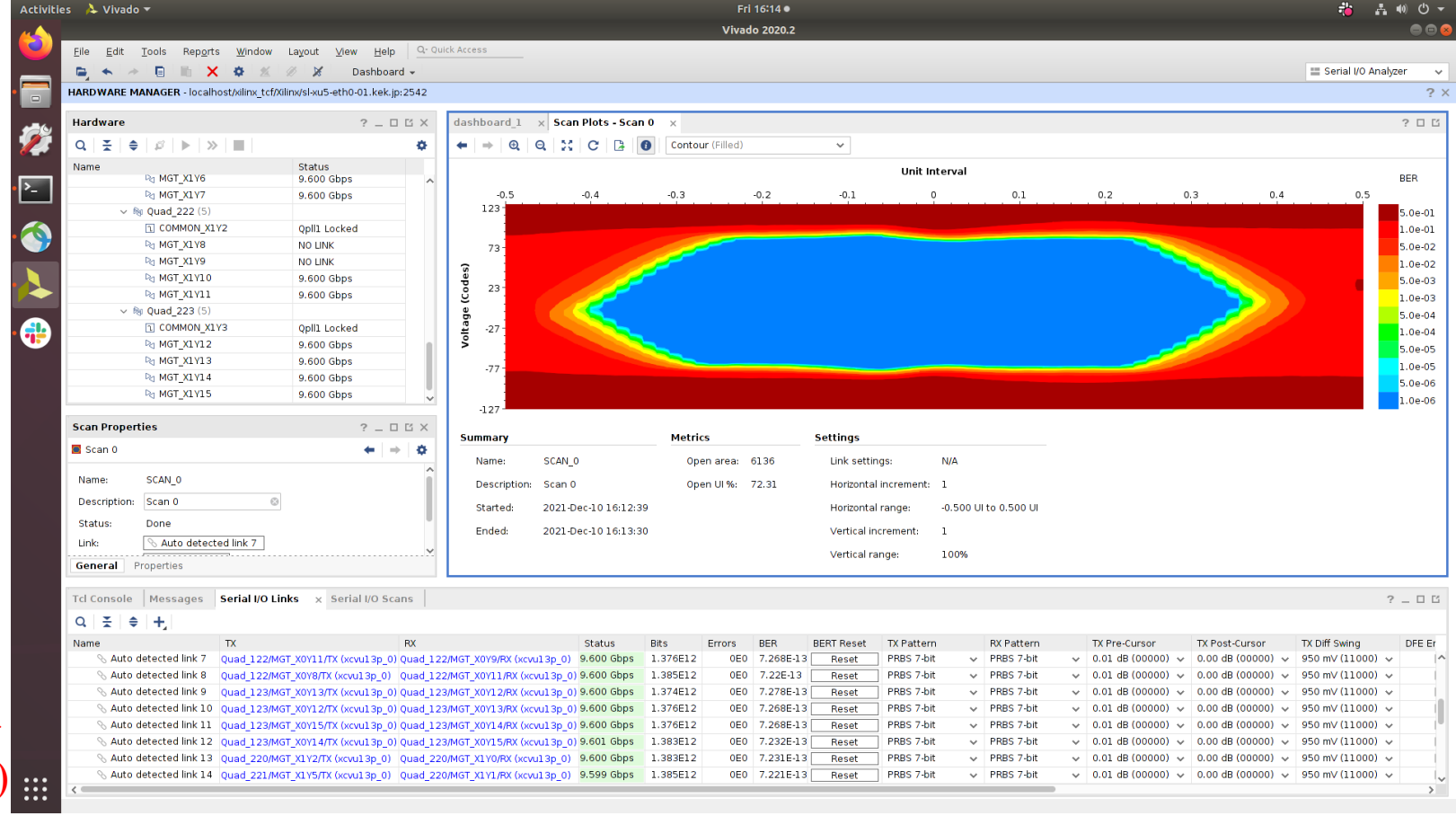
XVC

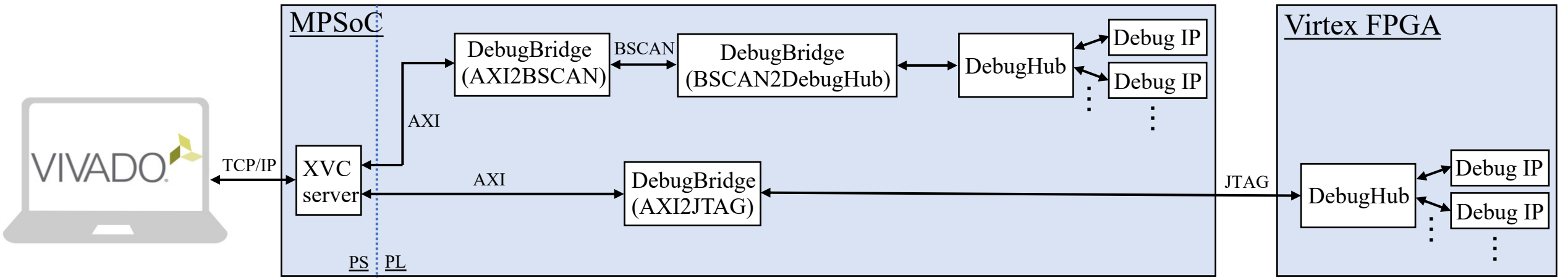
- Using Xilinx Virtual Cable (XVC), debug (ILA, VIO, etc.) of MPSoC can be performed via TCP/IP

- MPSoC and FPGA on the SL board are connected with JTAG wires

  → Debug and configuration of the Virtex FPGA can be performed via TCP/IP

Monitor GUI of IBERT by remote →
(accessing from Tokyo to SL@KEK (~100 km))





XVC server setting

- We are using the example code of the XVC server application provided by Xilinx:
  - https://github.com/Xilinx/XilinxVirtualCable

- DebugBridge can be easily accessed via the device file /dev/mem, but we prefer the access is protected in the User space IO (UIO) to avoid access by mistake

- Need to edit the sample code, device tree, and PetaLinux settings so that CentOS can detect the DebugBridge as UIO

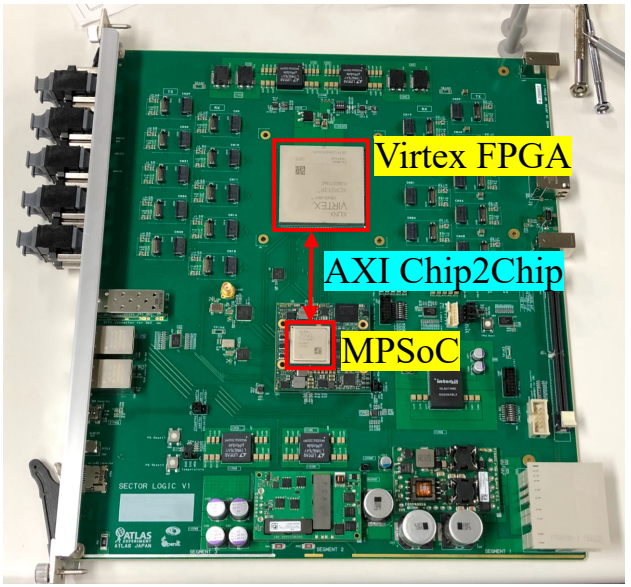
  (→ Technical details will be in the backup slides)

  → DebugBridge is successfully detected as UIO:

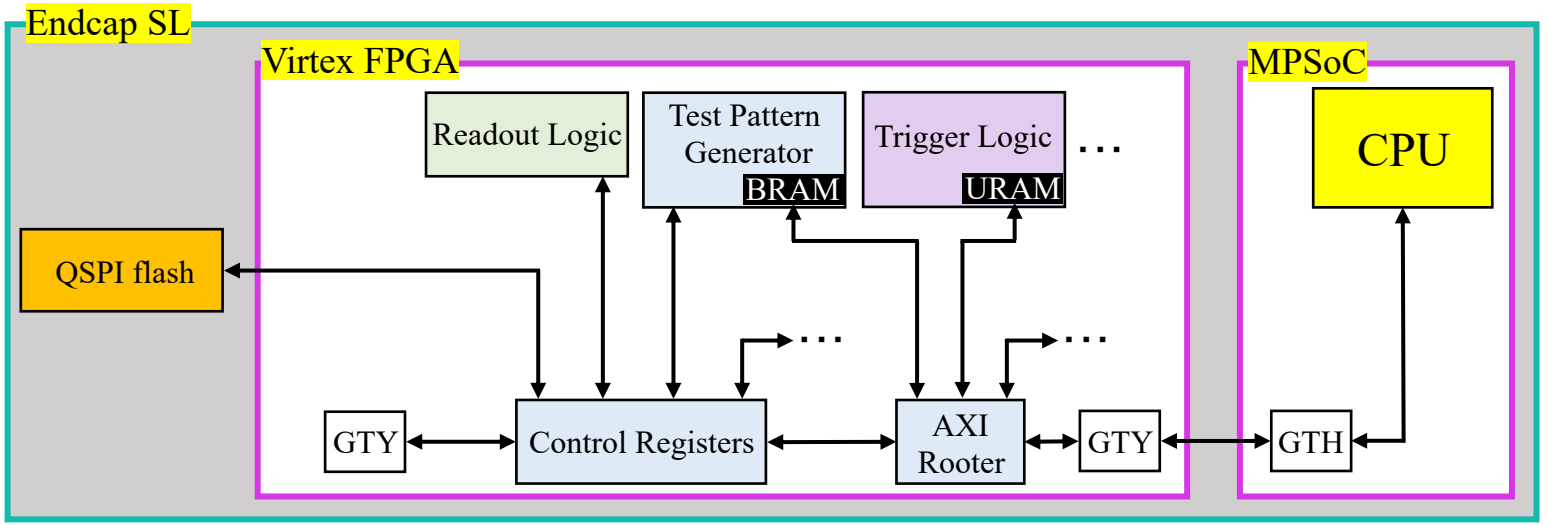
```
$ cat /sys/class/uio/uio0/name
debug_bridge
```

Control path

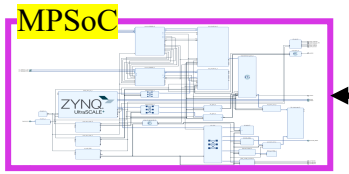
- Registers in the Virtex FGPA can be controlled from MPSoC taking advantage of the AXI Chip2Chip



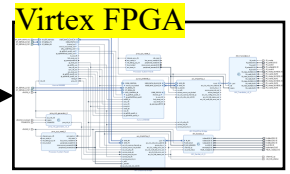
- For now, following functions, for example, have been implemented:
  - Trigger test pattern
    - Test pattern can be fed into the readout chain and trigger logic
    - Writing ‘1’ to a certain register triggers the output of the Test Pattern Generator
    - Registers for specifying which pattern will be output have also been implemented
  - Perform bit-banging into the QSPI flash
    - QSPI flash is used to store the FPGA bitstream and parameters for initial configuration
    - Data transcation in the SPI protocol can be performed via the FPGA registers
  - GTY reset
    - Writing ‘1’ to a certain register triggers the reset sequence of the GTY channels
- Following functions are to be implemented:
  - Rewrite the test patterns in BRAM
  - Write patterns in URAM for trigger logic
  - Reset readout buffers
  - etc.



Control path - MPSoC Block Design



AXI C2C



to FPGA

from FPGA

3.33 Gbps
(50 MHz x 66b)

Aurora 64B66B

AXI4 Stream

AXI4

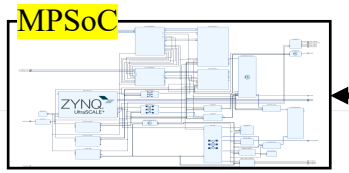
AXI SmartConnect

AXI4

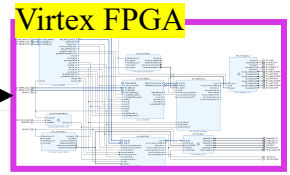
AXI Chip2Chip Bridge
(Master)

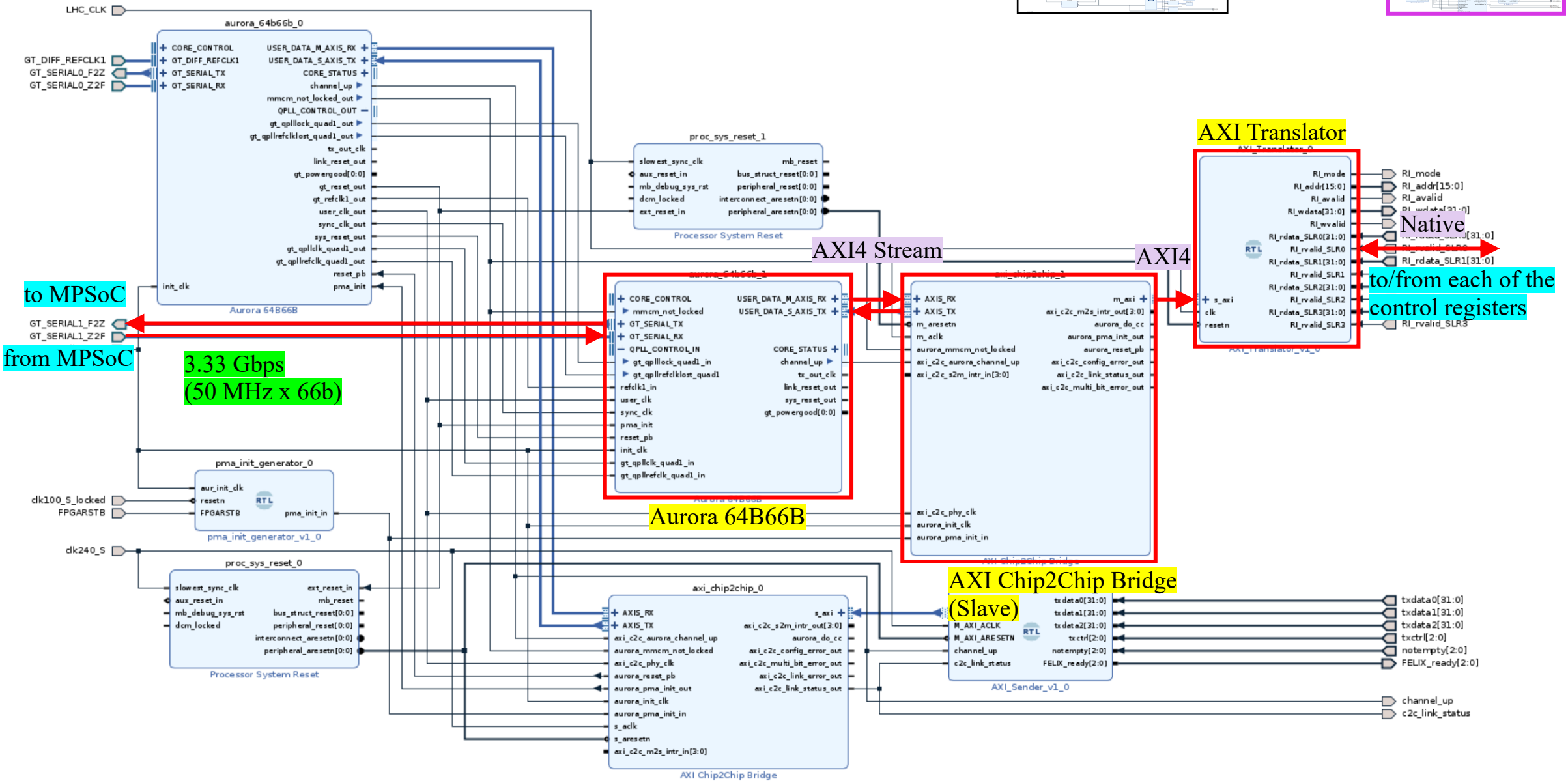
Zynq Ultrascale+ MPSoC
(Processing System)

Control path - Virtex FPGA Block Design



AXI C2C





Control path - Software operation

- On the MPSoC side, the AXI Chip2Chip Bridge is assigned to the address space of the CentOS:

Address Editor

/zynq_ultra_ps_e

/zynq_ultra_ps_e/Data (39 address bits : 0x00A0000000 [256M] ,0x0400000000 [2M] ,0x

			Base Address	Depth	High Address
/axi_chip2chip_1	s_axi	Mem0	0x00_A000_0000	64K	0x00_A000_FFFF

- Each destination address of the control registers are assigend within the address range (Base Address-High Address)
- CentOS has a device file /dev/mem for all physical address spaces available to the kernel

→ Opening the /dev/mem, the modules in PL can be accessible from the PS

- Open the /dev/mem

```
fd = open( "/dev/mem", O_RDWR );
```

- Get the address pointer to the PL module via the /dev/mem

```
ptr = mmap( NULL, page_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, page_addr );
```

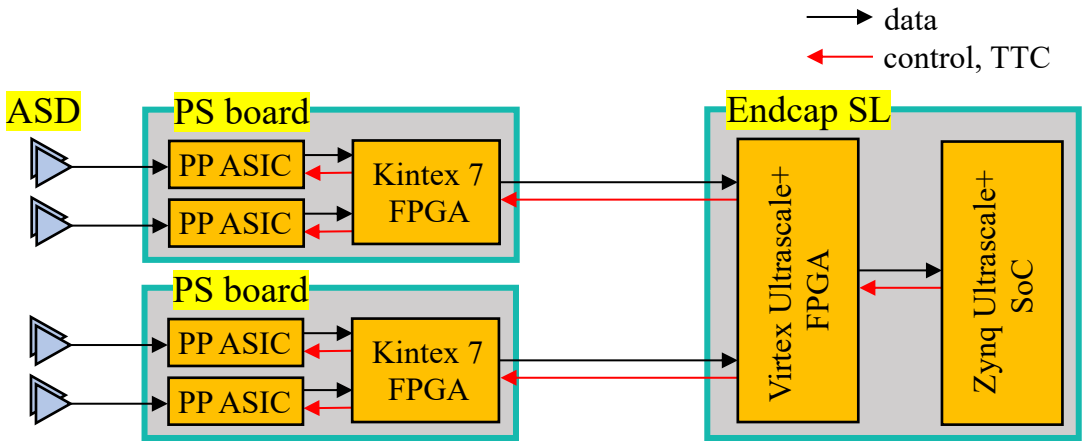
(→ Sample C code is in the backup slide)

→ Virtex FPGA registers can be mapped into the MPSoC PS address space and can be accessed handily

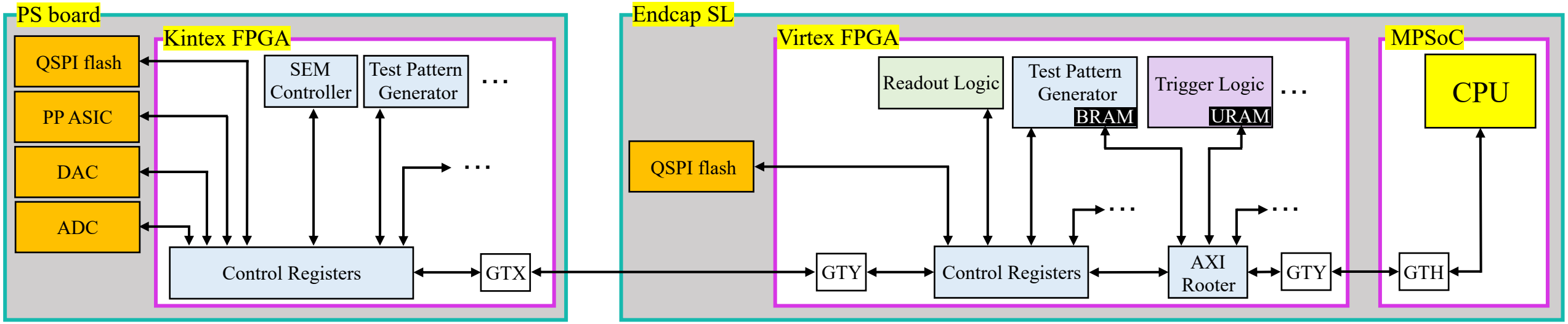
- The control path will be interfaced to the ATLAS TDAQ software
  - State control of the whole TGC electronics can be performed centrally
  - Configuration paramters to write in the control registers will be managed by the data base in ATLAS (OKS)

Application: Commissioning with the frontend electronics

- The control path can be used to control and monitor the frontend electronics with some extensions
  - Set delay parameters in PP ASIC
  - Trigger test pulse for ASD
  - Monitor SEU flag from PS board
  - etc.



- We are planning to conduct a commissioning with the frontend electronics
  - The main objective is to demonstrate the full readout chain and control path



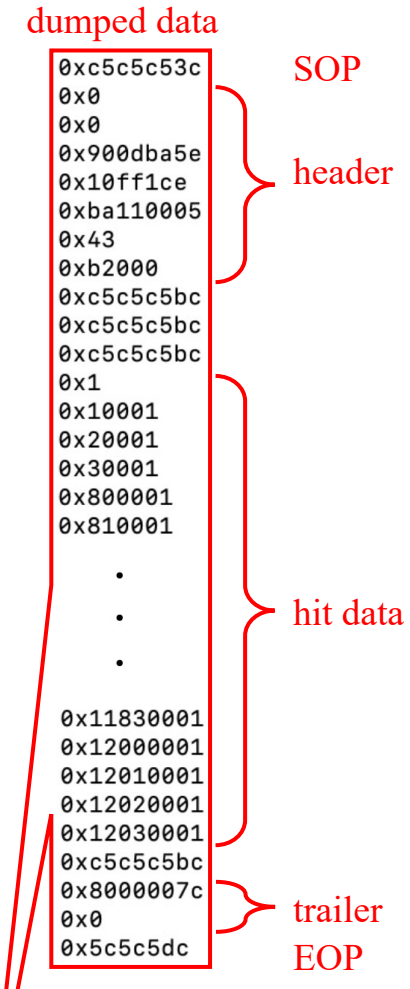
Single Board DAQ for initial board-level test

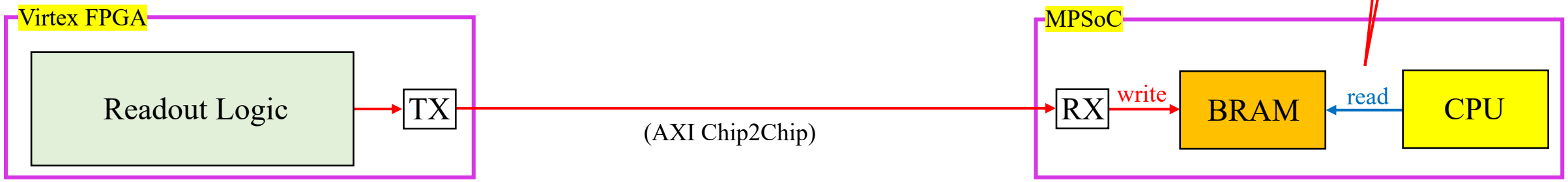
- The output of the Readout will be sent to FELIX via optical links
  → However, FELIX board is not in our possession at the moment
  → We have established Single Board DAQ using MPSoC and AXI Chip2Chip

- The output of the Readout firmware can be dumed into BRAM in MPSoC, and can be processed by CPU
  → Single board test framework is achieved and Readout chain is working well

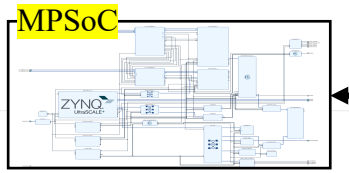
- The size of BRAM will be limited by the resource constraint of the MPSoC PL
  - The limit will be about 32 bit width x 320K depth (corresponding to approx. 500 events)
  - We may use the PS DDR4 via DMA to reserve larger storage capacity for the future commissioning

- There are three fibers supposed to be connected to FELIX, so the serialization of these into one AXI lane is performed in the Virtex FPGA

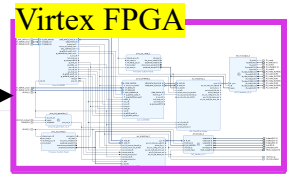




Readout path - Virtex FPGA Block Design



AXI C2C



from MPSoC

to MPSoC

15.84 Gbps
(240 MHz x 66b)

AXI4 Stream

Aurora 64B66B

AXI Chip2Chip Bridge
(Master)

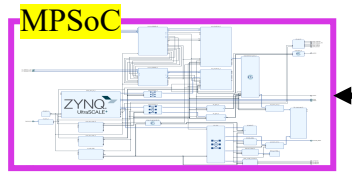
AXI4

Native

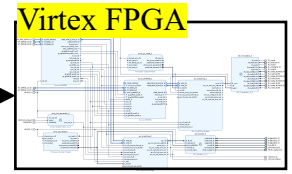
to/from Readout Logic

AXI Sender

Readout path - MPSoC Block Design



AXI C2C



to FPGA

from FPGA

15.84 Gbps
(240 MHz x 66b)

AXI4 Stream

Aurora 64B66B

AXI Chip2Chip Bridge
(Slave)

AXI4

AXI Channel Interrupter

AXI BRAM Controller

AXI4

Port A

Port B

Block Memory Generator

Zynq Ultrascale+ MPSoC
(Processing System)

AXI4

AXI4

AXI BRAM Controller

AXI SmartConnect

Single board DAQ - Software operation

- As the AXI Chip2Chip case, the BRAM is assigned to the address space of the CentOS:

→ From a software perspective, both the control path and the single board DAQ can be handled uniformly by just changing the destination address to access.

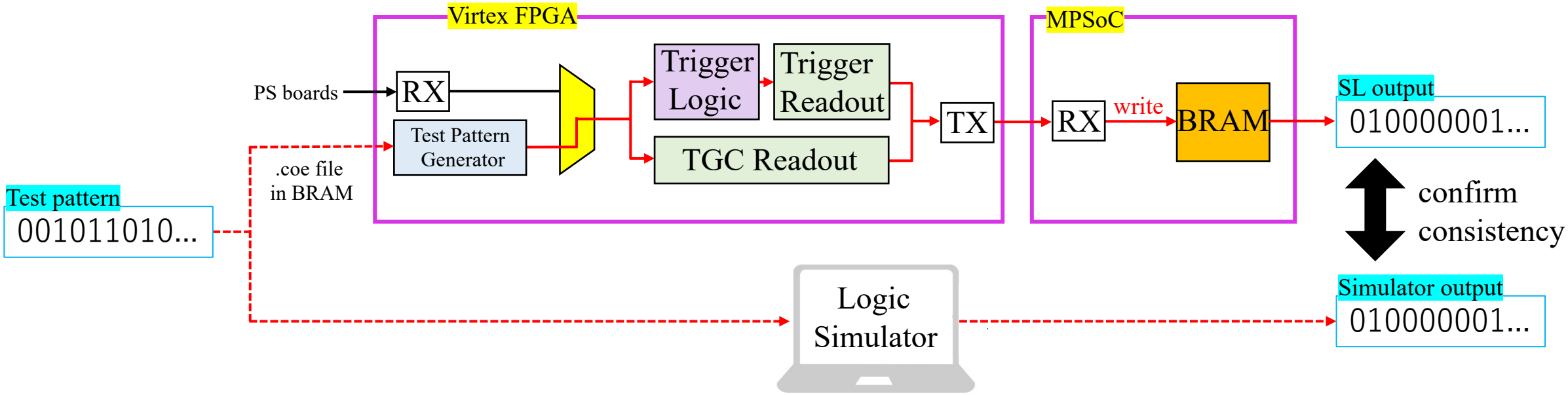
- The data taking operation example is as follows:
  1. Clear data in the BRAM
  2. Speciy the test pattern ID and trigger the test pattern via the Control path
  2.1. The corresponding event data is readout and written in the BRAM in a certain latency
  3. Readout the event data from the BRAM into a binary file
  4. Repeat 2. & 3. until the BRAM is filled, and back to 1.

- When the BRAM is full, the data sent from the Virtex FPGA is truncated on a event-by-event basis
- We have also implemented an operation mode where test patterns and TTC keep being generated autonomously with some rate (e.g. 1 kHz) using counter circuits in the Virtex FPGA (the control path can switch the op. mode)
- The binary files fetched in the 3. are processed in the MPSoC PS in the current commissioning
  → File forwarding path out of the MPSoC is to be implemented for the future test (via SD card, Ethernet, etc.)

Single board commissioning

- The single board DAQ and readout firmware have been validated
  → To validate the Trigger firmware, a simulator emulating the Trigger Logic is being developed

- Test patterns are used as the input of the firmware
  - We have developed Test Pattern Generator emulating the input bitmaps from the PS boards
  - Test patterns are stored in BRAMs
  - The BRAMs are initilaized with .coe file in the current design, but it will be reconfigurable from MPSoC
- TTC is emulated in the Virtex FPGA



Summary

- Developed XVC path
  - Enables debug of MPSoC PL via TCP/IP
  - Enables debug and configuration of the Virtex FPGA on the SL via TCP/IP and JTAG path

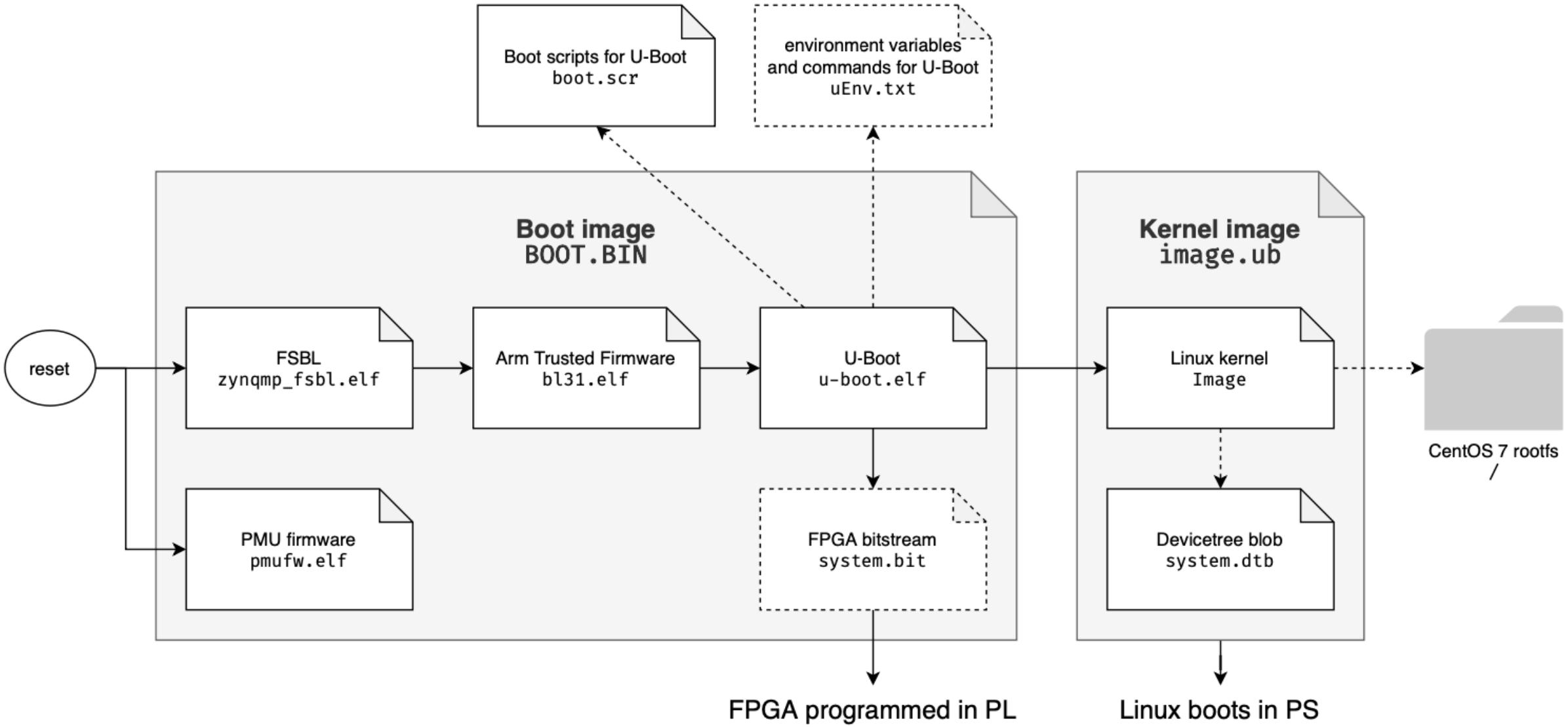
- Implemented control path using AXI Chip2Chip
  - Registers in the Virtex FPGA on the SL can be controlled from applications on CentOS
    - Test pattern control and QSPI bit-banging have already implemented
    - Futher functions are to be added on this infrastructure
  - Frontend electronics can also be controlled via the Virtex FPGA and optical links

- Designed and implemented single-board DAQ system
  - The outputs of the Readout Logic can be dumped into RAM of MPSoC utilizing the AXI Chip2Chip
  - Readout and processing can be performed by applications on CentOS
  - Readout chain of the SL has been demonstrated

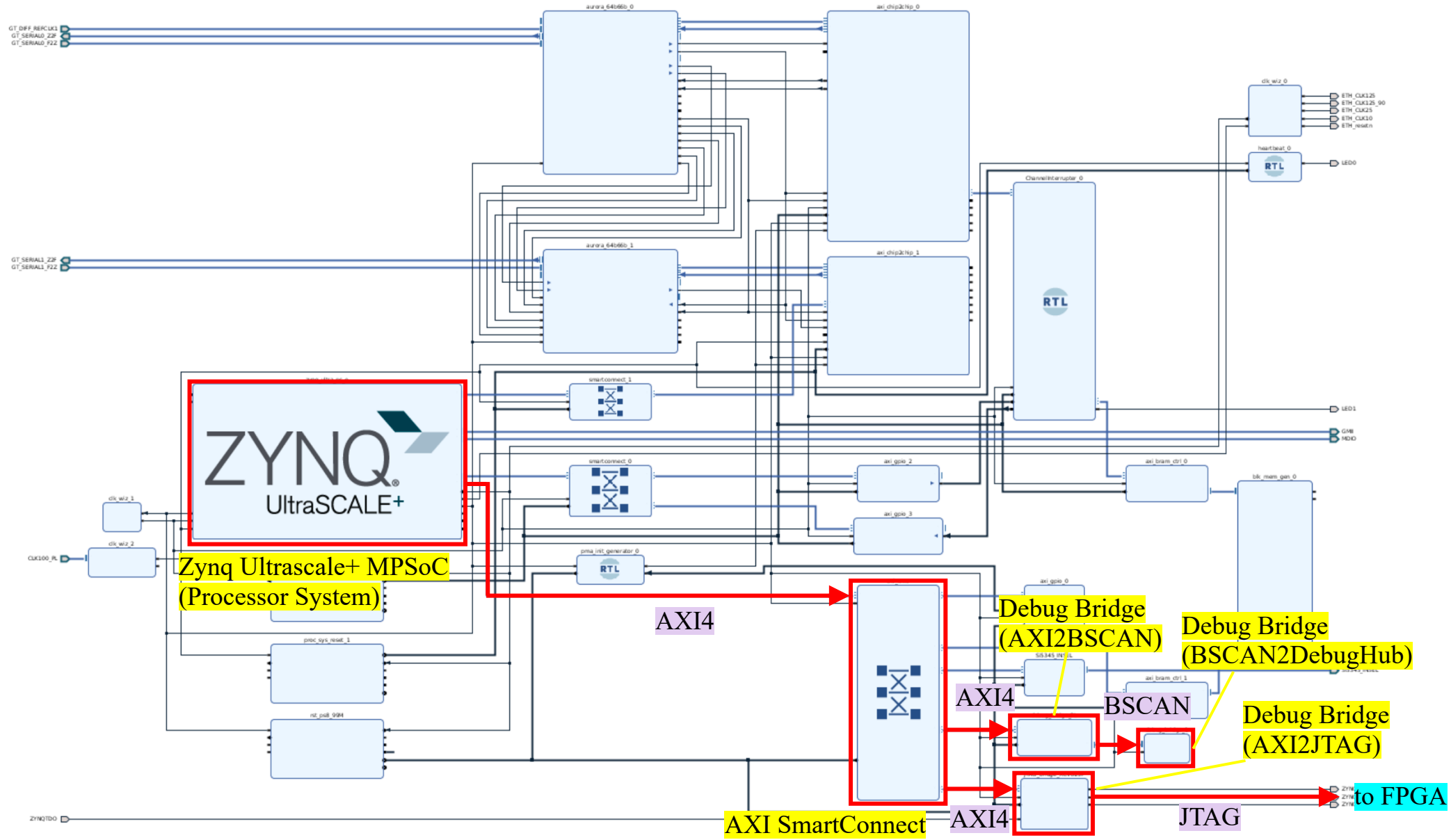
→ We can conduct electronics commissioing using the control path and single board DAQ
  - Readout firmware development and validation have been done
  - Trigger firmware integration and validation are now ongoing
  - Commissioing with the frontend electronics are being planned

Backup

Boot sequence



XVC – MPSoC Block Design



Device tree and PetaLinux setting for XVC (1)

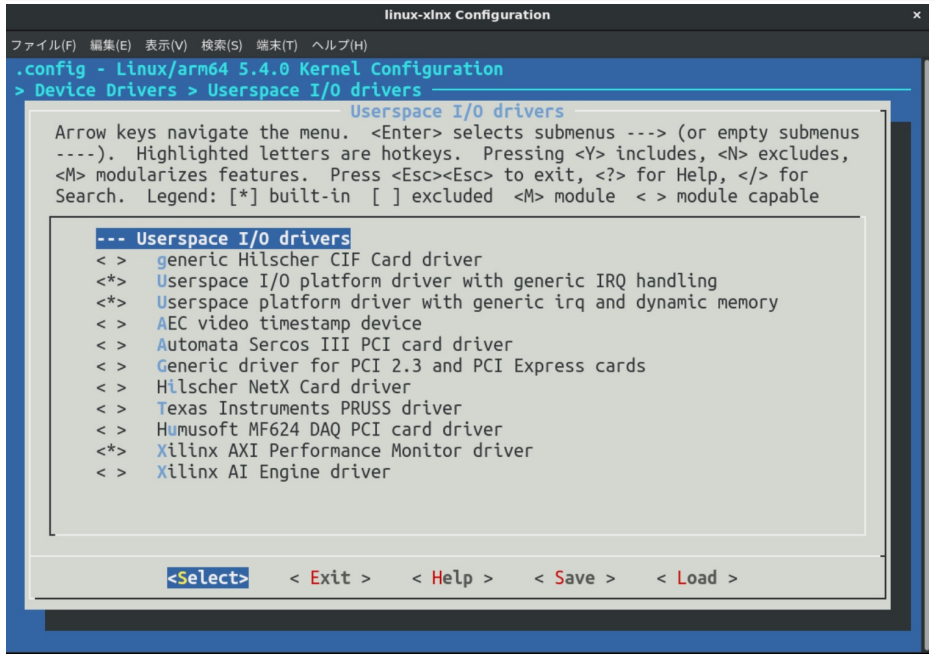
- Edit the device tree so that the CentOS can detect the DebugBridge as UIO

```
// Allow the Debug Bridge to communicate in the Linux uio space.
&debug_bridge_0 {
  compatible = "generic-uio";
};
```

- Configure the PetaLinux settings as follows:

```
petalinux-config -c kernel

Device Drivers ---> Userspace I/O drivers
```




```
CPU Power Management ---> CPU Idle ---> [ ] CPU idle PM suppor
```

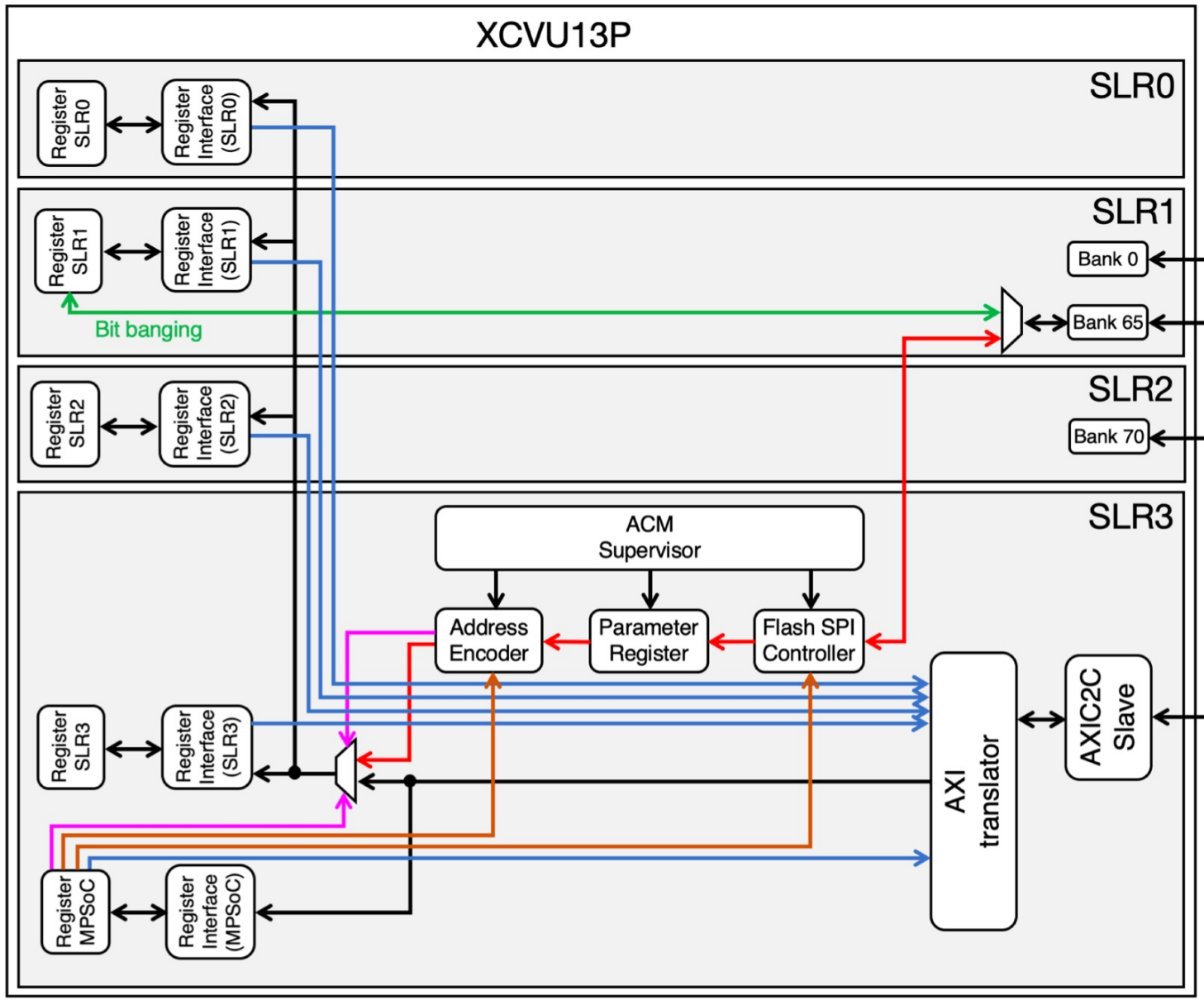
Device tree and PetaLinux setting for XVC (2)

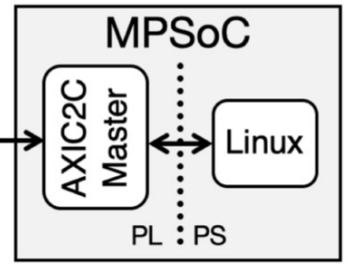
- Configure the PetaLinux settings as follows:

```
petalinux-config --get-hw-description=<path ot XSA file>
```

  - Add “uio_pdrv_genirq.of_id=generic_uio cpuidle.off=1” into the Kernel bootargs
    - If you set bootargs in uEnv.txt, also add the sentence to the text

Control registers interface

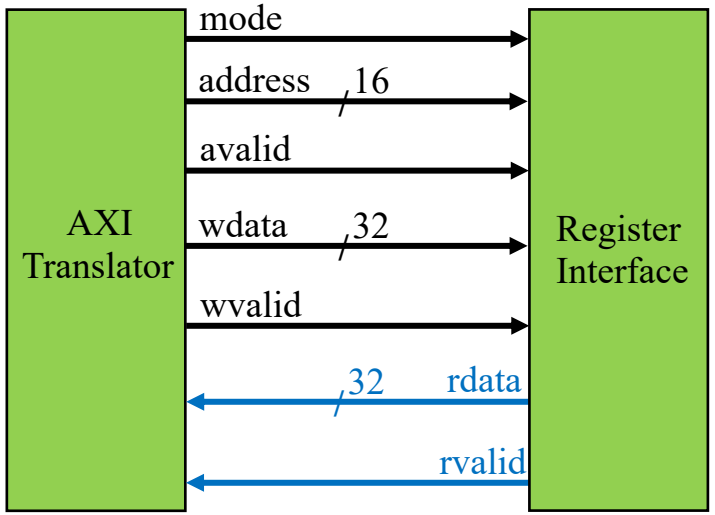




QSPI0

QSPI1

200 MHz quartz



C code sample for opening /dev/mem and accessing the PL

```c
#include <fcntl.h>      // open() etc.
#include <stdint.h>     // uintX_t etc.
#include <stdio.h>      // printf() etc.
#include <sys/mman.h>   // mmap() etc.
#include <sys/stat.h>   // open() etc.
#include <sys/types.h>  // open() etc.
#include <unistd.h>     // sysconf() etc.

#define PHYS_ADDR 0×A0000000

int main() {
    int                 fd;
    uint32_t            page_size = sysconf(_SC_PAGESIZE);
    uint32_t            phys_addr, page_addr, page_offset;
    volatile uint32_t* ptr;

    // Open /dev/mem with read-write mode.
    fd = open("/dev/mem", O_RDWR);  // File Descriptor. fd ⩾ 0 if successful
    if (fd < 0) {
        printf("Error opening /dev/mem\n");
        return -1;
    }

    // Initialize ptr
    phys_addr   = PHYS_ADDR;
    page_addr   = (phys_addr & ~(page_size - 1));
    page_offset = phys_addr - page_addr;
    ptr         = mmap(NULL, page_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
                       page_addr);
    if (ptr == MAP_FAILED) {
        printf("Failed to map physical address.\n");
        close(fd);
        return -1;
    }
    ptr += page_offset;

    uint32_t offset = 0;
    uint32_t data   = 0×deadbeef;

    // Read
    printf("offset = %d, \tdata = 0x%x\n", offset, *(ptr + offset));

    // Write
    *(ptr + offset) = data;

    // Read again
    printf("offset = %d, \tdata = 0x%x\n", offset, *(ptr + offset));

    close(fd);
    return 0;
}
```