MPSoCデバイスを用いた高輝度LHC-ATLAS実験トリガーエレクトロニクス制御の高度化

28th ICEPP Symposium 2022年02月20日

東京大学大学院 理学系研究科 物理学専攻 石野研究室
岡崎健人

本発表の構成

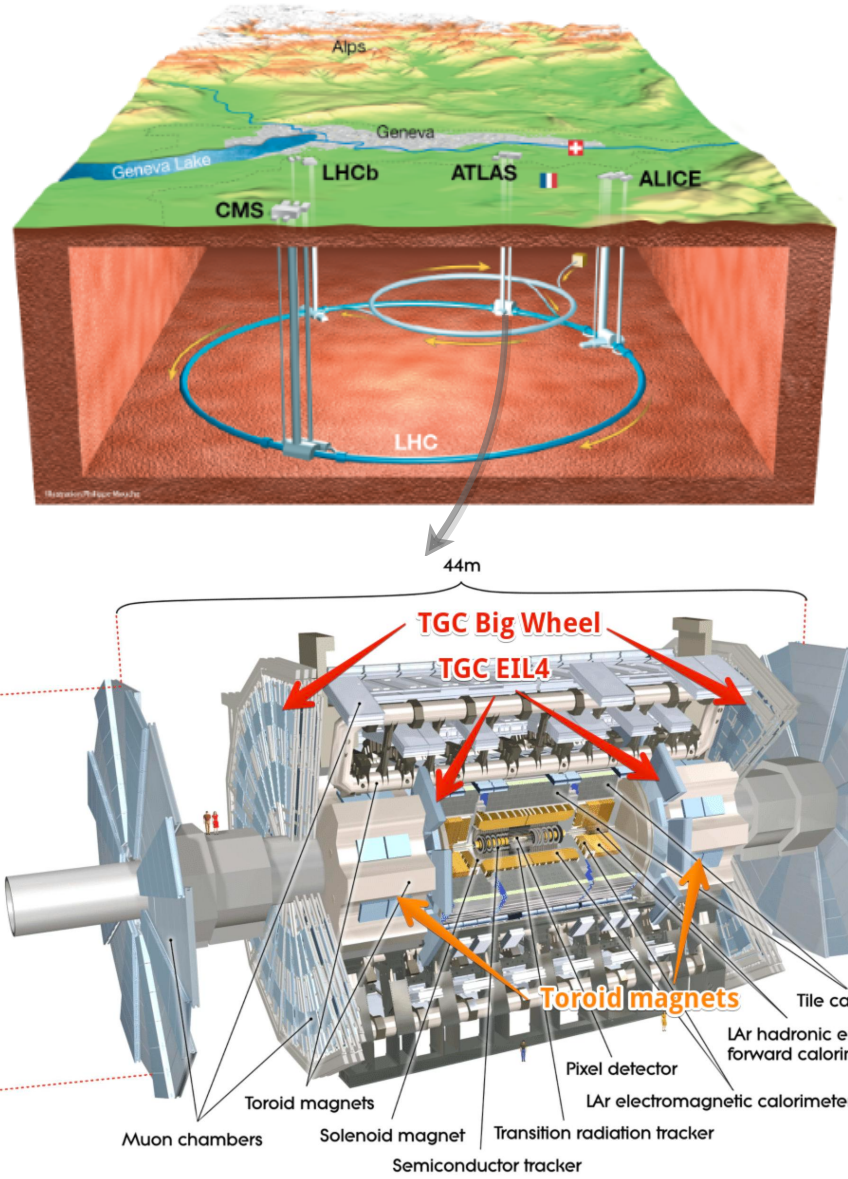
- 研究背景
- MPSoCでのソフトウェア/ファームウェア開発パスの確立
- Endcap SL第1試作機上での動作検証
- AXI Chip2Chipを用いたFPGA間シリアル通信
- まとめと今後の展望

研究背景

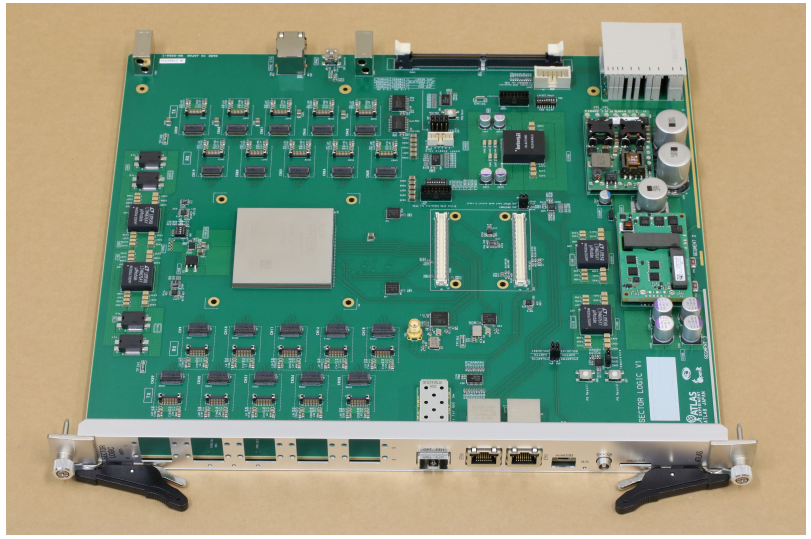
高輝度LHC-ATLAS実験

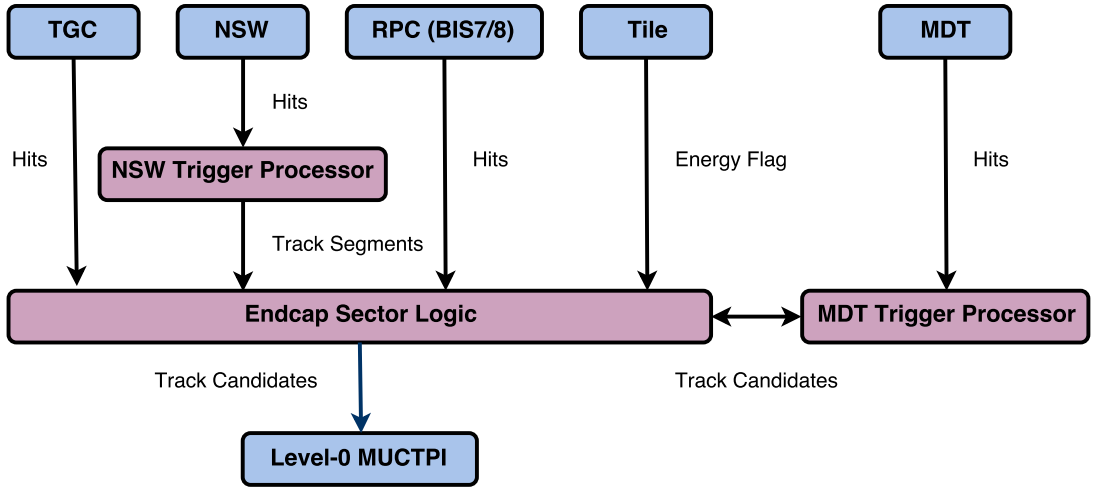
- LHCはCERNにある陽子-陽子衝突型円形加速器
- 瞬間最高ルミノシティを$7.5 \times 10^{34}\ \mathrm{cm}^{-2}\mathrm{s}^{-1}$に増強した**高輝度LHC**として運転（2027-）

- **ATLAS検出器**は標準模型の精密測定および新物理探索を目的とした大型汎用検出器
- **TGC検出器**はATLAS検出器のエンドキャップ部に設置され、ミューオントリガーを担当
- ATLAS検出器を構成する検出器、トリガー・読み出しのエレクトロニクスも一新



後段回路Endcap Sector Logic（Endcap SL）

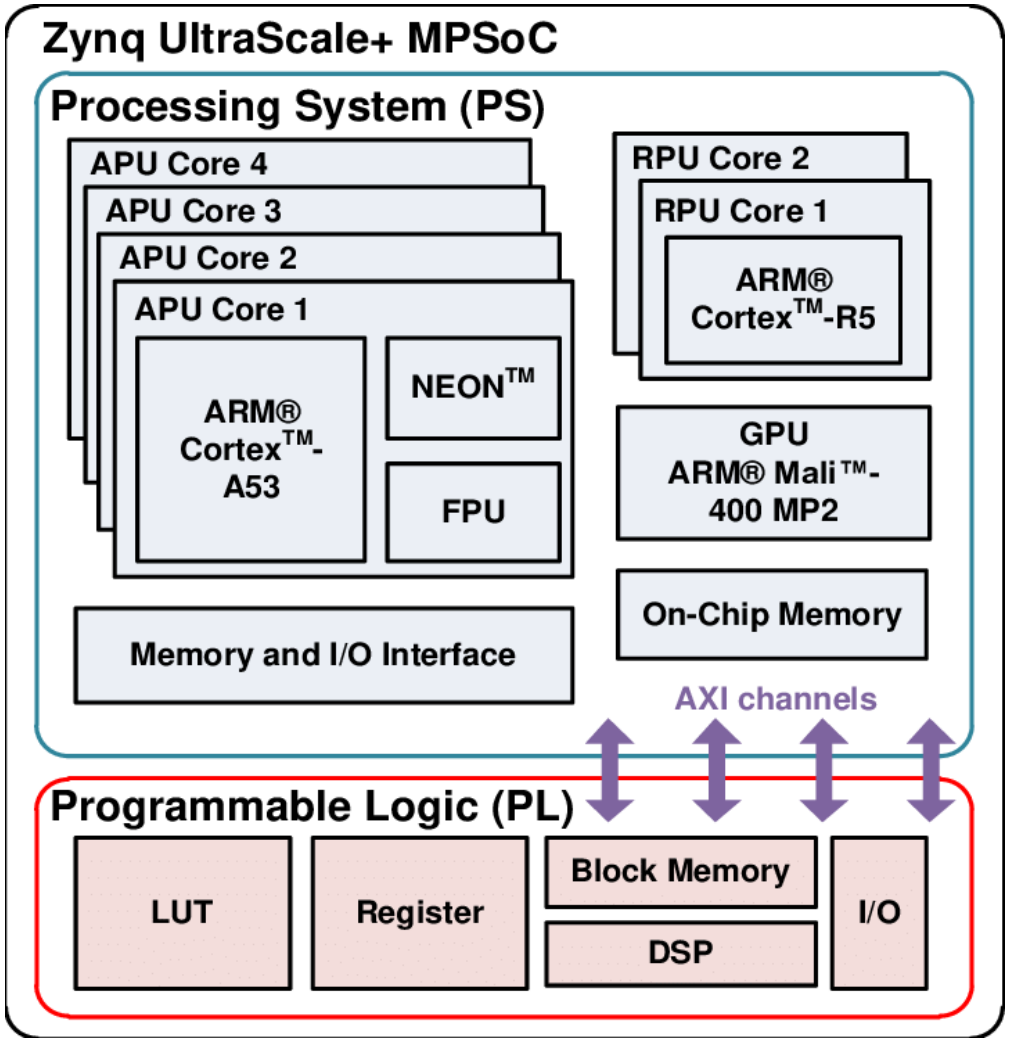




- TGCを含むエンドキャップ部ミューオン検出器群から生じるすべてのヒット信号を受信
- ボード上の大型FPGA（Xilinx Virtex UltraScale+ 13P、以降**XCVU13P**）で飛跡再構成を行い、後段のLevel-0 Muon Central Trigger Processor Interface（MUCTPI）に送信
- TGC前段回路へのLHC 40 MHzクロックや制御信号の配布

Zynq UltraScale+ MPSoC

- SLには**Zynq MPSoC**（MultiProcessor System-on-a-Chip）を搭載し、制御系の中枢を担う
  - Processing System（**PS**）：64-bit quad Arm Cortex-A53などを用いたプロセッサ部
  - Programmable Logic（**PL**）：UltraScale+ FPGA部



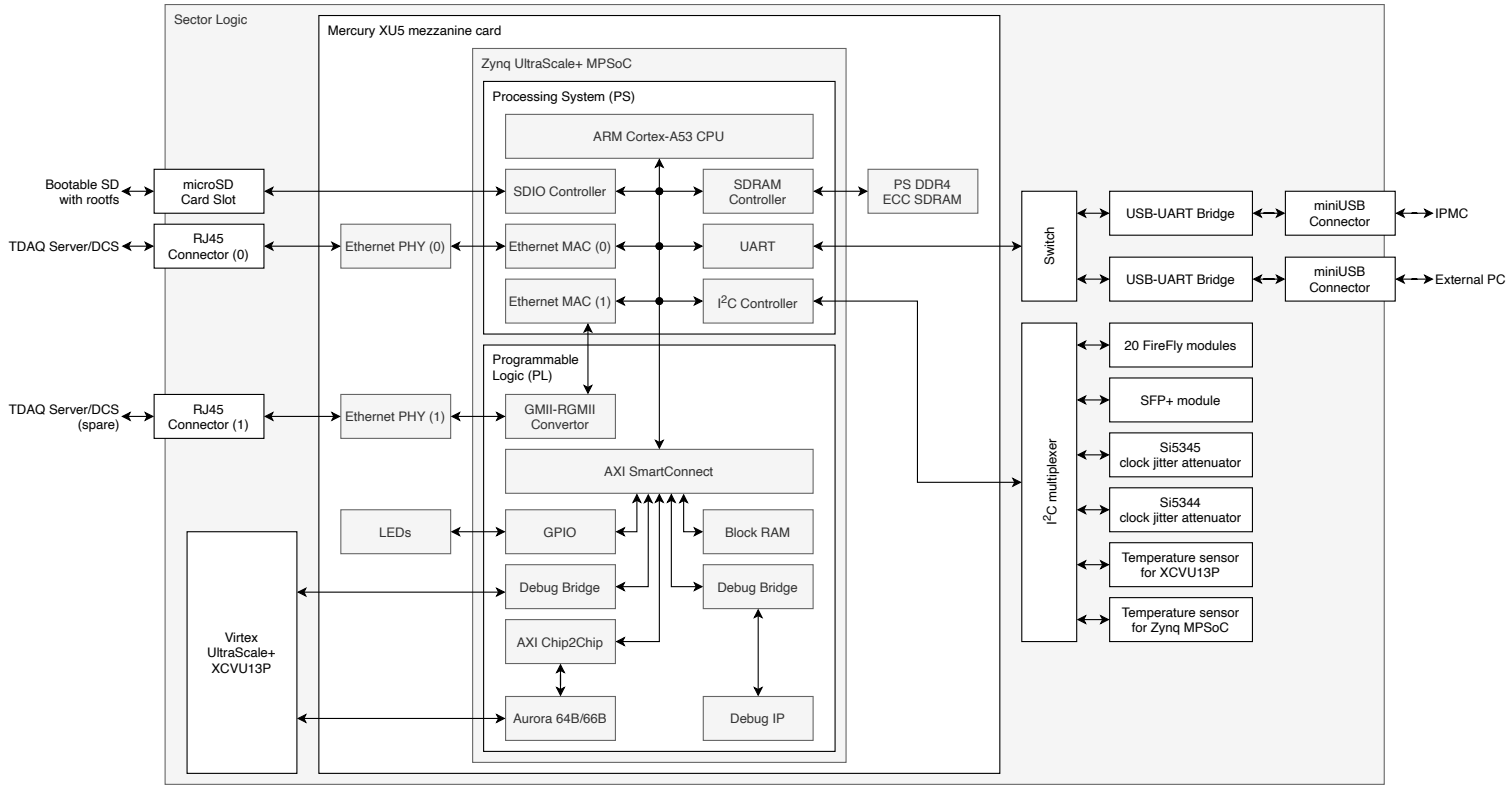
Mercury XU5メザニンカード

- Zynq MPSoC搭載のメザニンカード（**Mercury XU5**）をSLで使用する
  - PS/PL用のDDR4 SDRAM
  - 2 Gigabit Ethernet PHYs
- 市販のメザニンカードを高エネルギー実験で使用する利点
  - MPSoC周りの設計を省略し、ボードの開発コストを下げる
  - 異なるベースボードでもメザニンカードの知見や開発物を共有
  - メザニンカード上で問題が発生した場合、交換が容易

Endcap SL上の Zynq MPSoCの機能

PSではLinuxが走り、次のようなアプリケーション・ファームウェアが動作する

- 外部ネットワークとSLとのインターフェイス
- XCVU13Pのプログラム
- ボード上の素子の監視と制御
- XCVU13Pのレジスタ操作による回路のパラメータ設定、フロントエンドFPGAへのアクセス

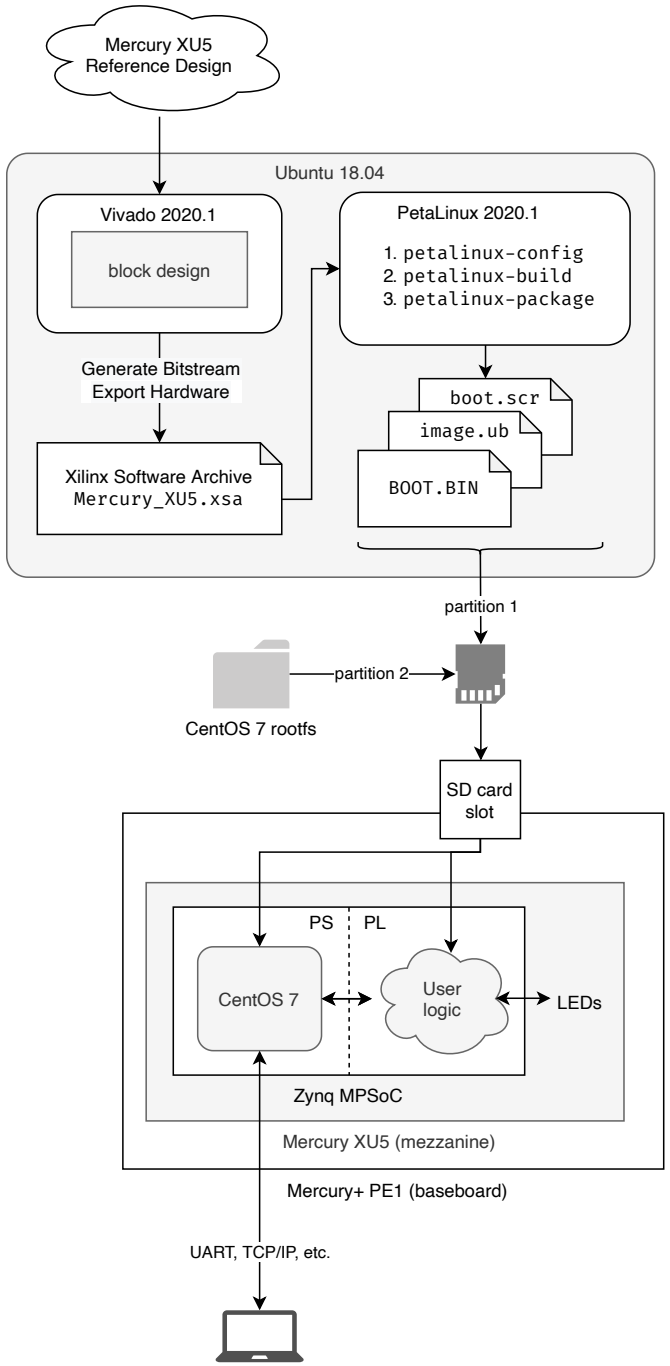


研究目的

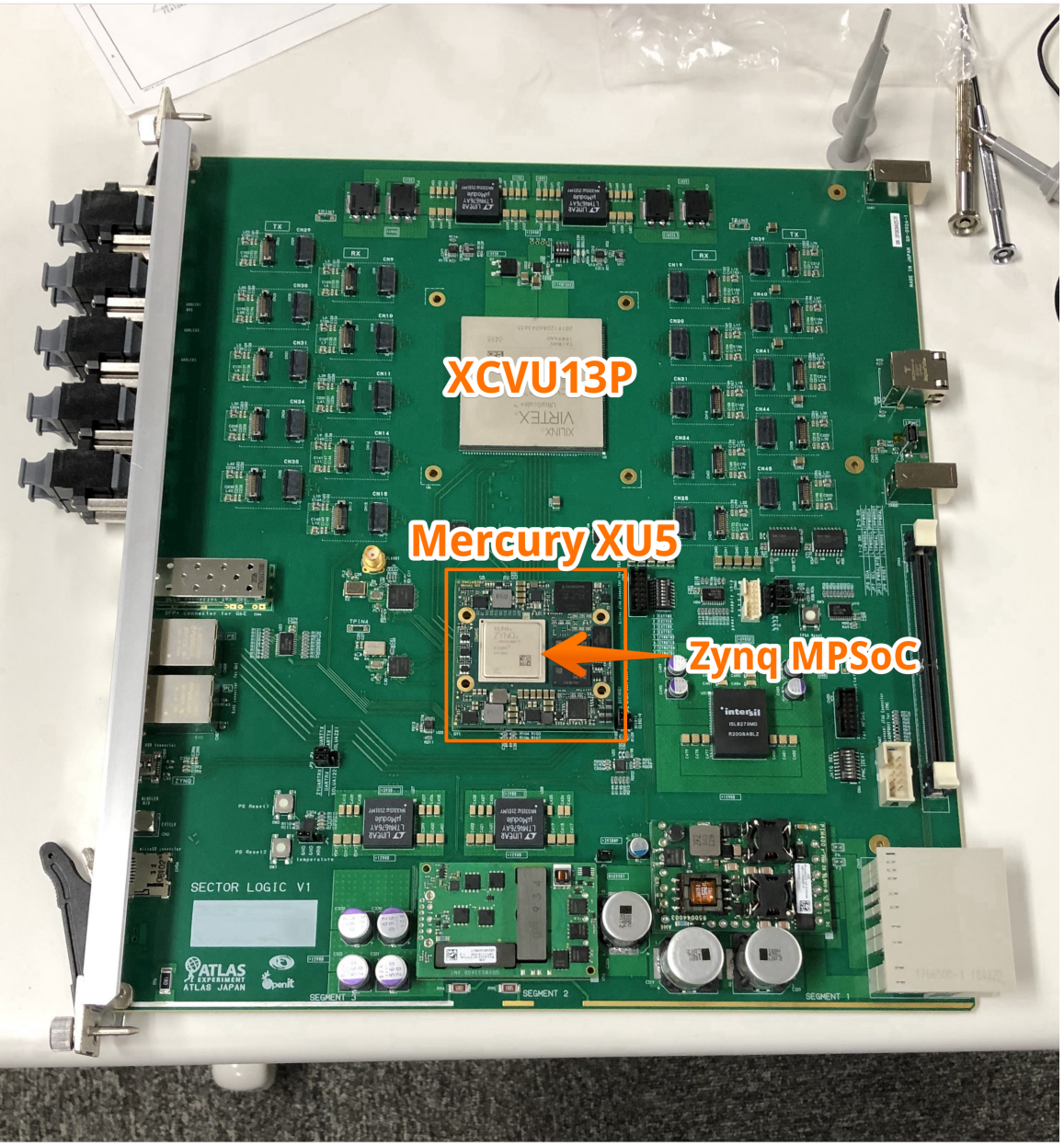
- Mercury XU5上のZynq MPSoCでソフトウェア、ファームウェア開発フローの確立
  - 一般に、単純なFPGAに比べてCPUとFPGAが一体となったZynq SoCの開発は複雑で困難
  - 開発パスを完成させ、その手法をグループで共有することは価値がある
- Endcap SL第1試作機上での動作検証ならびに機能実装
- AXI Chip2Chipを用いたFPGA間シリアル通信のデモンストレーション

市販のメザニンカード+フルカスタムボードを使った新しい高エネルギー実験制御系のデザインを確立し、その基盤技術を築く

MPSoCでのソフトウェア/ファームウェア開発パスの確立



- Mercury XU5とその評価ボードMercury+ PE1を用いた開発パスの実現を最初のステップとした
- Mercury XU5 PE1 reference designを活用し、ハードウェアの設定を簡略化した
- VivadoでPLの設定、FPGA bitstreamを出力
- PetaLinuxでブートファイルのクロスコンパイル、パッケージング
- SDカードの作成
  i. ブートファイル（`BOOT.BIN`, `image.ub`, `boot.scr`）
  ii. CentOS 7のルートファイルシステム
- CentOS 7のブートに成功し、ソフトウェアからLEDなどを操作できることを確かめた



Endcap SL第1試作機上での動作検証

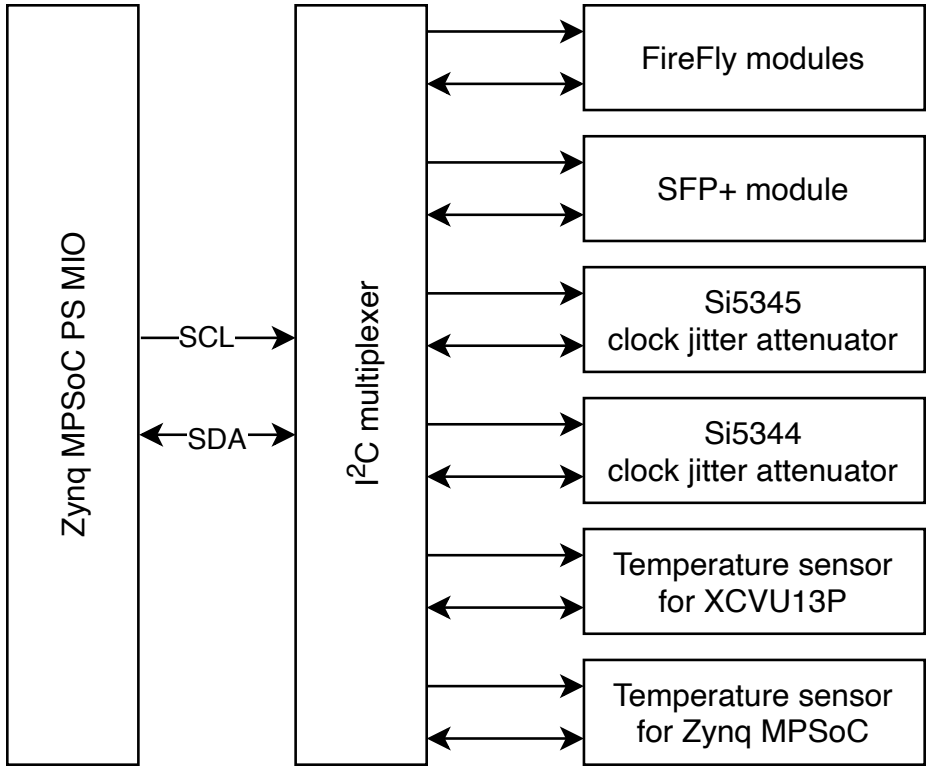
1. CentOS 7のブート
2. Ethernetの開通
   - ネットワーク経由でログインしアプリケーションを開発するのに必要
3. LinuxからI^2Cデバイスを操作
4. Xilinx Virtual Cable
   - MPSoC PLのデバッグ
   - XCVU13Pのデバッグおよびプログラム

CentOS 7のブート

- Endcap SLはMercury+ PE1を参考にして設計されており、確立したパスで開発できる
- SLボード上の配線に間違いがあったが、Vivadoでのハードウェアの設計を変更すればCentOS 7が起動した

Ethernetの開通（PS/PL Gigabit Ethernet）

- MPSoCはEthernet経由でコマンドの受信やデータベースの参照をするためのインターフェイスとなる
- SLボード上の配線に間違いがあったが自作のLANケーブルで対処した
- IPアドレス、MACアドレスの設定にはCentOS 7の`nmtui`コマンドや`uEnv.txt`で設定できた
- Endcap SL第2試作機では修正する予定

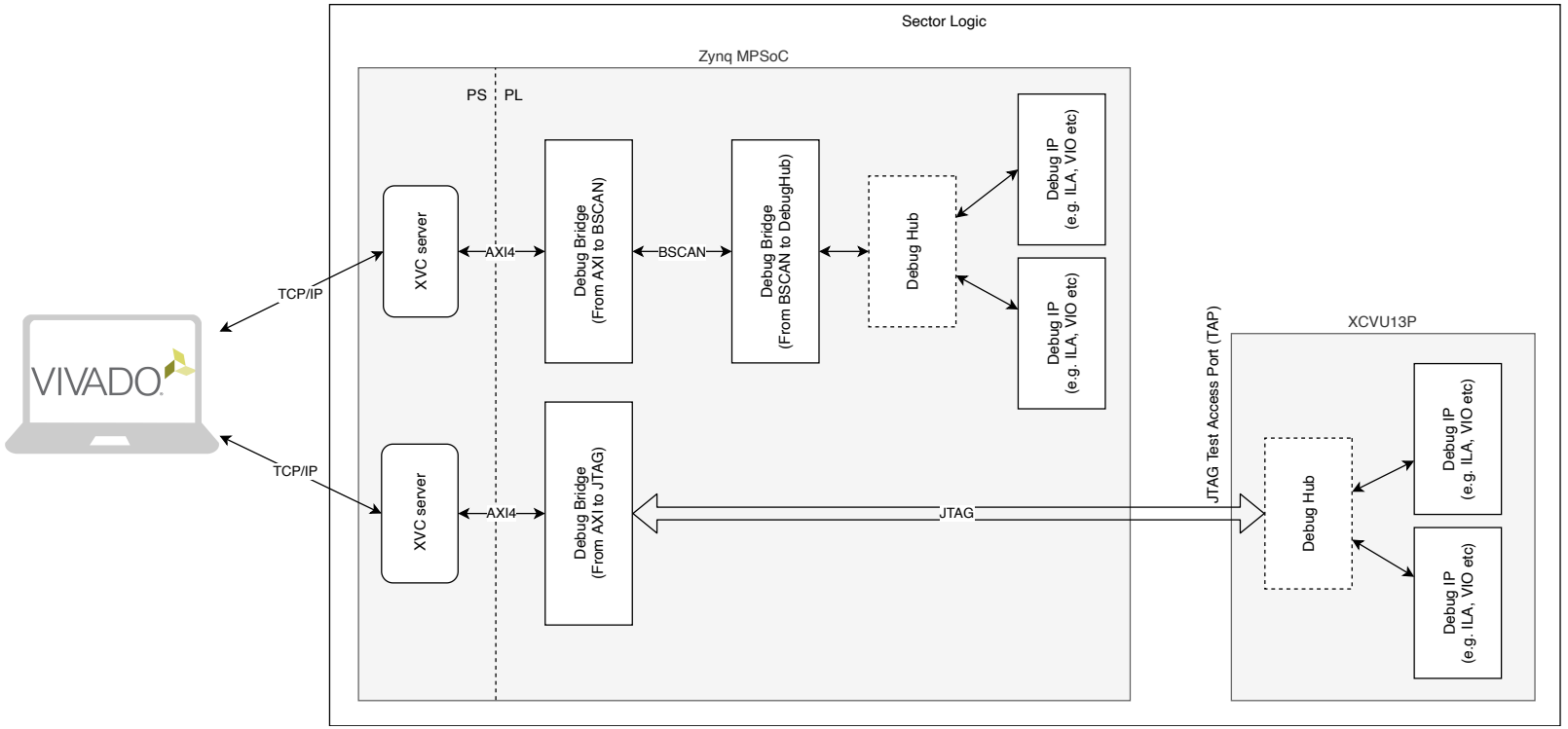


LinuxからI²Cデバイスを操作

- MPSoCはI²C multiplexer（I²C MUX）を介してclock attenuator、温度センサ、SFP+モジュールなどの操作、監視ができる
- PetaLinuxで生成したLinuxカーネルに付属したデバイスドライバとCのライブラリを使えば、`/dev/i2c-*`ファイルを`open`し読み書きすることでI²Cによるデバイスの制御、監視ができることを確認した
- SLに限らずLinuxからハードウェアを制御をする好例

Xilinx Virtual Cable（XVC）

- TCP/IP通信で送られるコマンドをJTAG信号に変換することで、Xilinx社のデバイスをプログラムやデバッグする機能



- ネットワーク経由でZynq MPSoCにアクセスし、PLのデバッグや、XCVU13Pのデバッグおよびプログラムができることを確認した

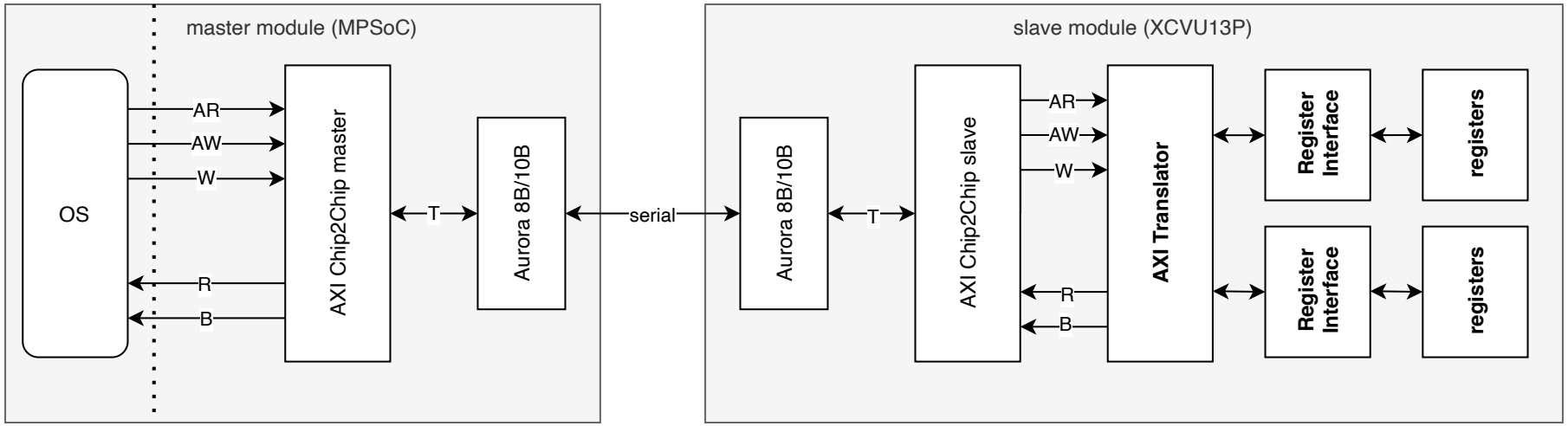
AXI Chip2Chipを用いたFPGA間シリアル通信

- 高輝度LHC-ATLAS実験では、Zynq MPSoCがXCVU13P内のレジスタを読み書きすることにより、回路の動作モードの設定、ステータスの確認を行う
- 実験で制御すべきレジスタが、Linuxが参照できるメモリ空間にマップされていることで、アプリケーションから容易に操作できる

- ZynqのPS-PL間インターフェイスとして**AXI4プロトコル**が採用されている
- またVivadoで使用可能なIPの多くはAXI4をサポートしている
- FPGA間の通信にもAXI4を用いるのが合理的

- **AXI Chip2Chip**は複数FPGA間のシームレスなAXIブリッジとなる
- **Aurora**をインターフェイスとして採用すれば、物理層のGigabit Transceiverを用いた高速シリアル通信が可能

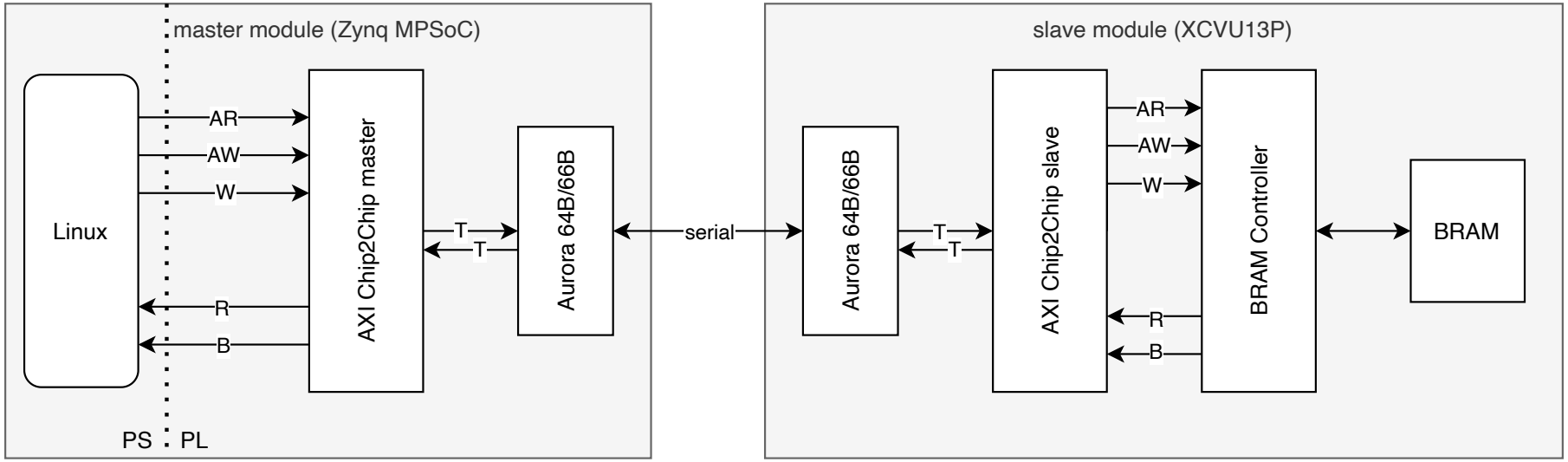
AXI Chip2Chipを用いたレジスタ制御機構のデザイン



- AXI4（`AR`, `AW`, `W`, `R`, `B`）、AXI4-Stream（`T`）
- チップ間通信のためAXI Chip2Chipを用い、Aurora 64B/66Bでエンコード・デコードしチップ間のシリアル通信を実現する
- **AXI Translator**はAXI4コマンドを解釈して各**Register Interface**に配布し、各レジスタに読み書きを行う

AXI Chip2Chipを用いたシリアル通信の開通

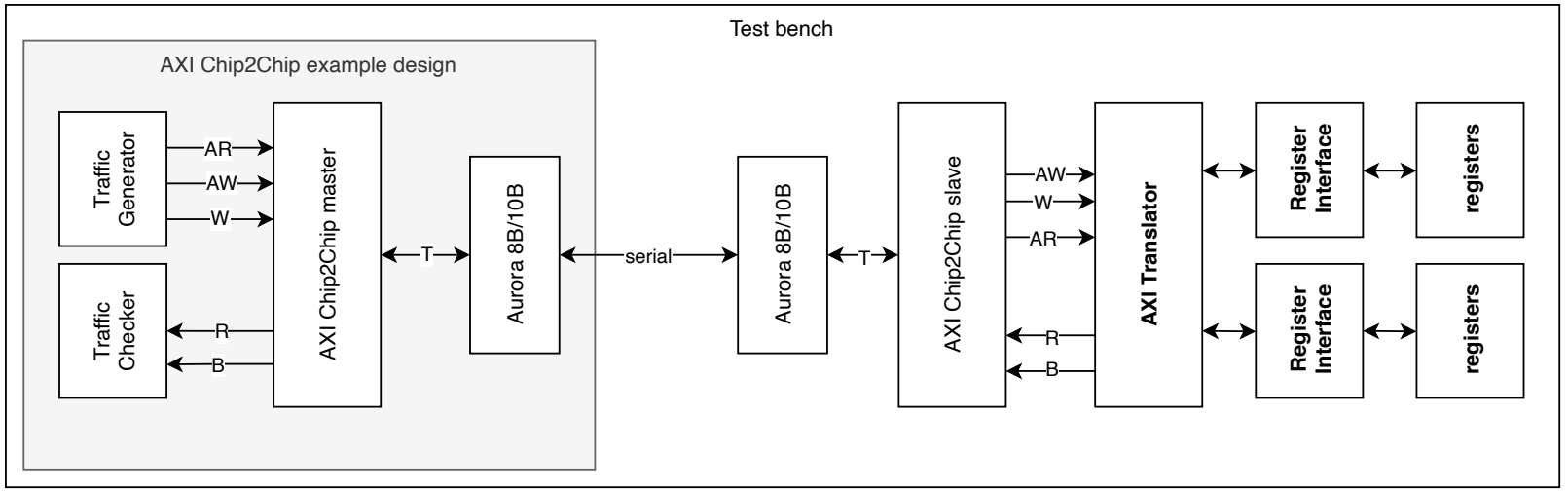
- まずAXI Chip2Chipを用いたシリアル通信の試験のため、VivadoのIPとして使用可能で経験のあるBlock RAM（BRAM）をスレーブ側に置いておいた
- このBRAMはAXI Chip2Chip masterの使用可能なメモリ空間に対応した幅と深さをもつ



- AXI Chip2ChipとAurora 64B/66Bを然るべく結線し、デバッグ用のポートをプローブすることで、シリアルリンクの確立を確認した
- MPSoCで走るCentOS 7のソフトウェアからアクセスに成功した

レジスタ制御機構の開発

- AXI4プロトコルを理解し、AXI Chip2Chip slaveのAXI4による命令を各レジスタに配布するAXI TranslatorとRegister Interfaceを開発
- AXI Chip2Chipのサンプルデザインを活用し、このレジスタ制御機構が正常に動作することをテストベンチシミュレーションで確認



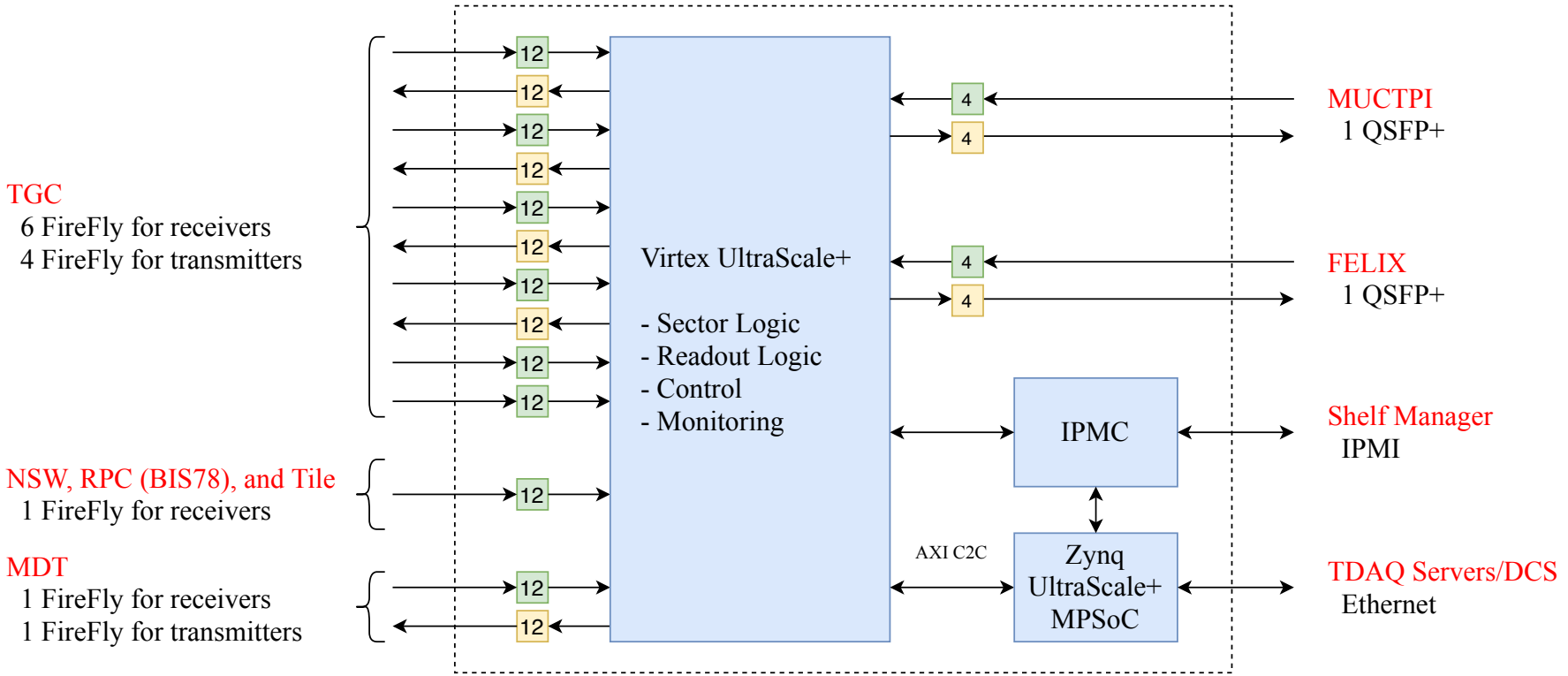
- AXI Chip2Chipを用いてシリアル通信できることがわかり、レジスタ制御機構も完成したので、今後はこれらを統合しレジスタ制御機構用のファームウェアを作成する

まとめと今後の展望

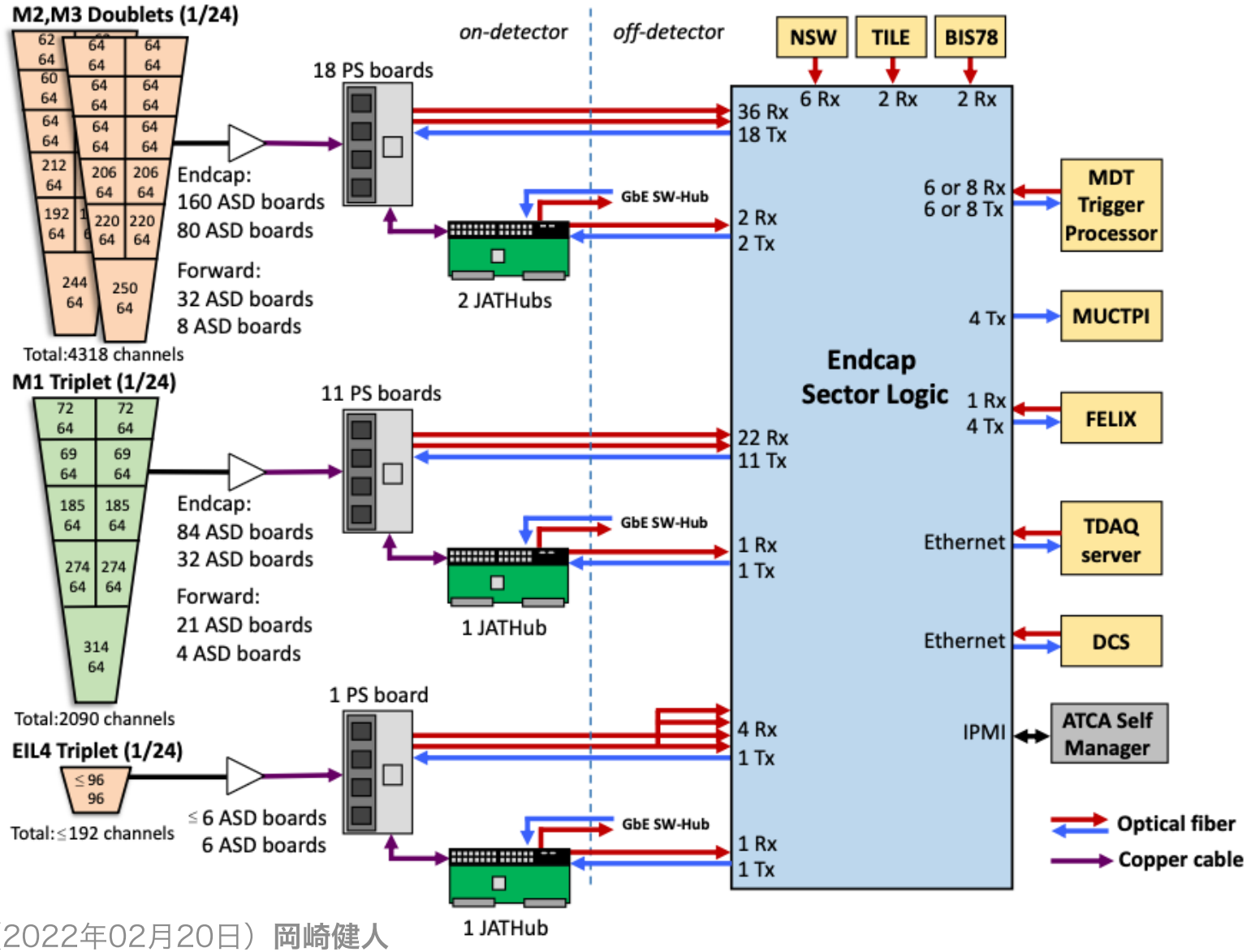
- 比較的安価な市販品のメザニンカードを、実験のためのカスタムボードに搭載し運用に成功した
  - Zynq MPSoC on Mercury XU5でのLinuxの起動、アプリケーションとファームウェアの開発基盤を確立
  - Endcap SL第1試作機を用いてLinuxを起動し、ペリフェラルの動作確認、最低限必要な機能開発を行い、Endcap SL制御のための開発環境を整えた
  - AXI Chip2Chipを用いたシリアル通信のデモンストレーションに成功し、新たなFPGA間通信の手法を開拓した
- 今後の展望
  - AXI Chip2Chipの機構とregister interfaceを統合し、XCVU13Pのレジスタ制御を実現する
  - テストパターンを用いたXCVU13Pでのトリガー・読み出しパスを確立し（MPSoCから確認）、前段回路と統合
  - （Endcap SLの動作試験の残り）

Back-up Slides

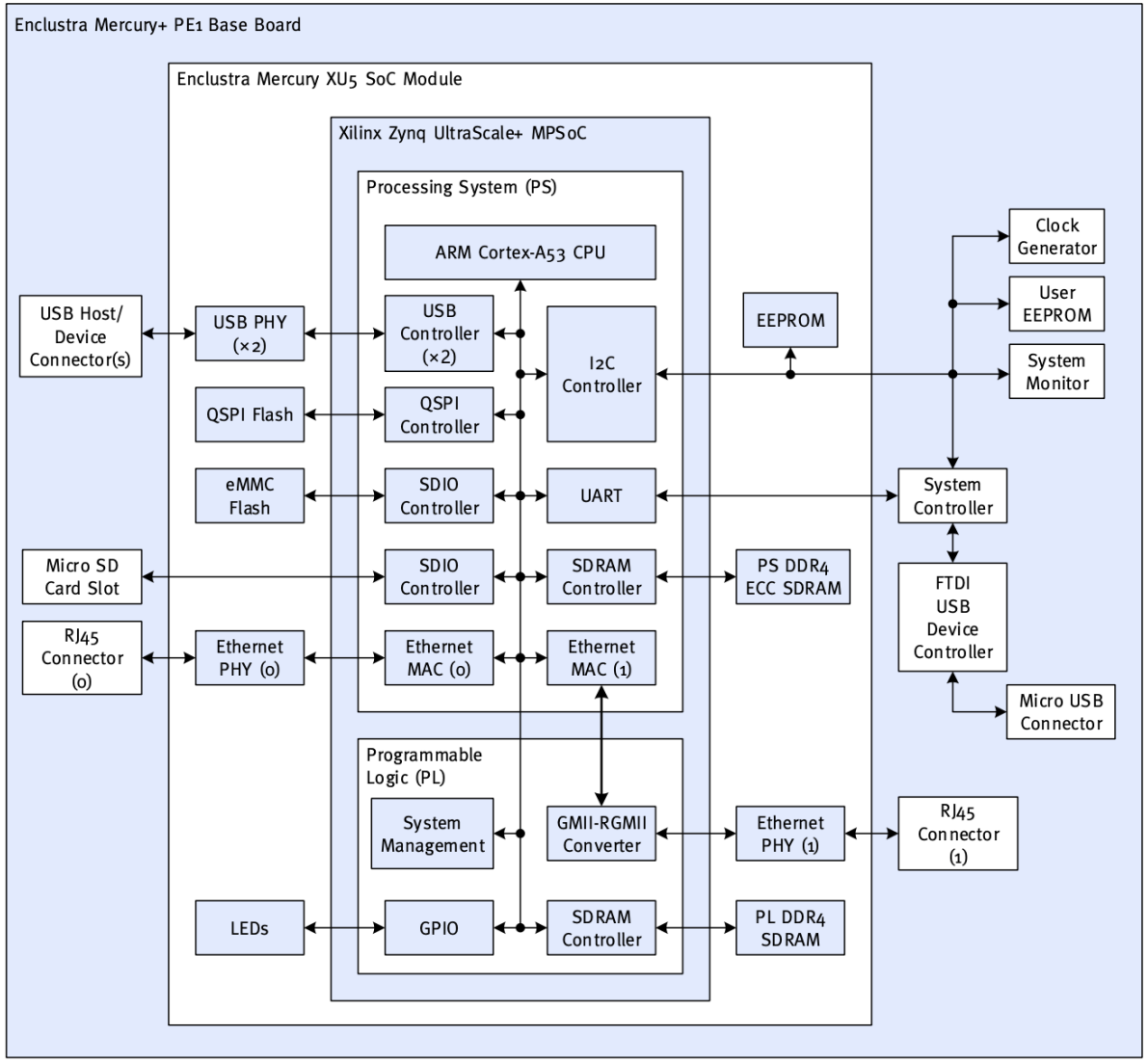
Sector Logic Interface



TGCエレクトロニクス

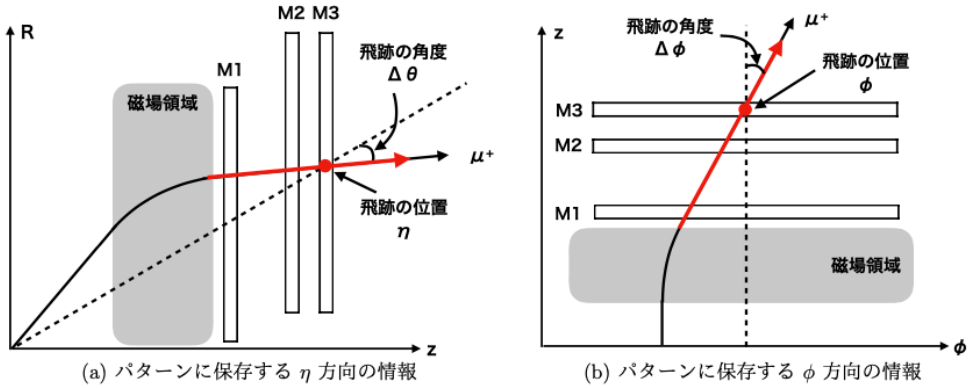


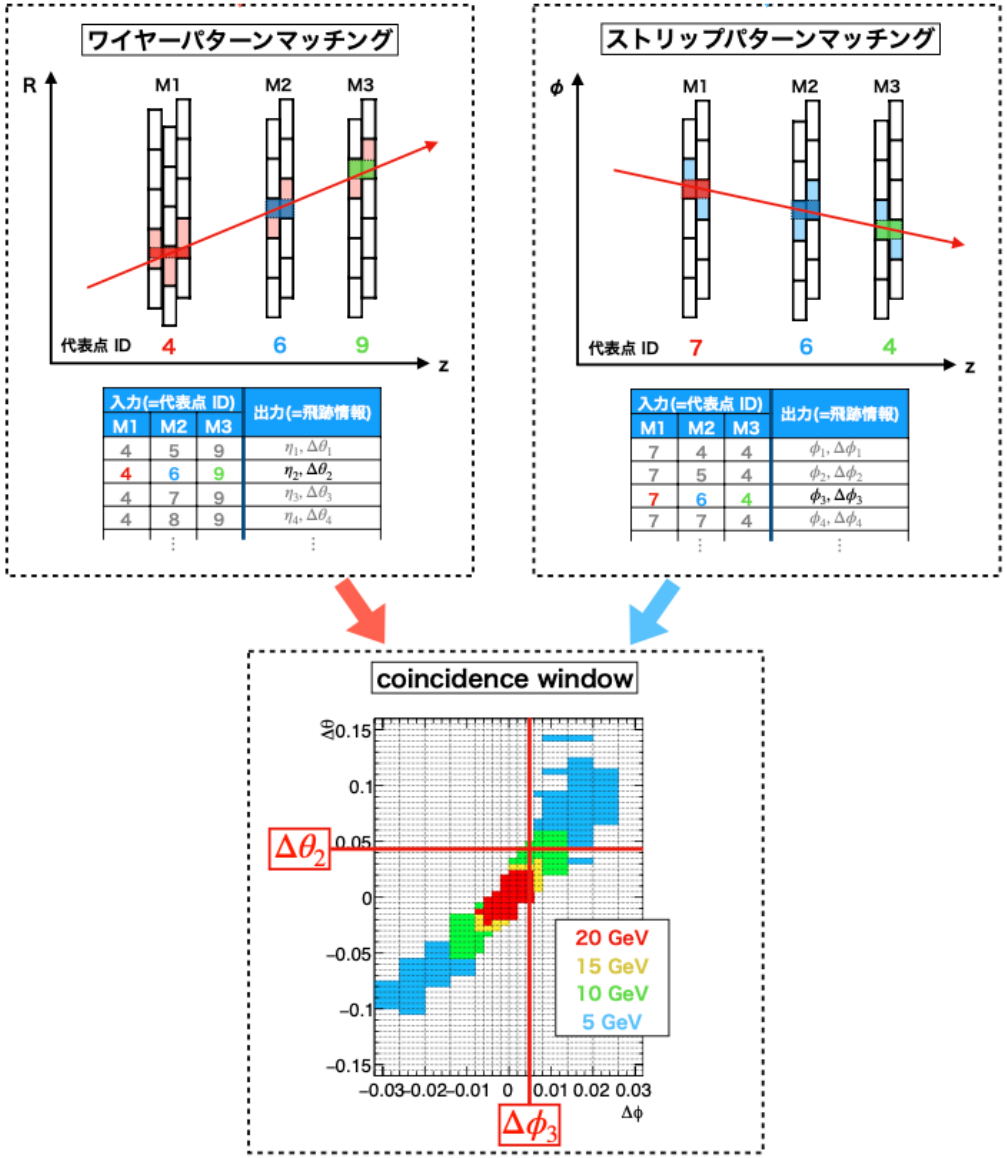
Mercury XU5 PE1 Reference Design



Endcap SLでの飛跡再構成

- ヒットのデータから飛跡の位置と角度を計算する
  - ワイヤーの情報から$(\eta, \Delta\theta)$を、ストリップの情報から$(\phi, \Delta\phi)$を算出
  - あらかじめ対応表（Look-Up Table）を準備しておくことで高速処理を実現（**パターンマッチングアルゴリズム**）
- Coincidence Windowを用いてp_{T}を概算する





参考文献：三野裕哉，高輝度LHC ATLAS実験に向けた初段ミューオントリガーアルゴリズムの開発およびハードウェアへの実装，修士論文，2019

LinuxからPLのモジュールを操作

- 物理アドレスを割り当てられたPLのモジュールはPSからアクセス可能
- Ubuntuでは`devmem`コマンドでユーザ空間から任意のメモリにアクセス可能
- CentOSではLinuxカーネルが使用できるすべての物理アドレス空間のキャラクターデバイス`/dev/mem`が存在
- この`/dev/mem`を`open`してPSメモリにマップし、読み書きを行うことでPLモジュールの操作が可能
- あるいは、PLモジュールをUserspace I/O（UIO）として使用するようデバイスツリーに記述しておけば、Linuxから`/dev/uio*`として使用が可能

`/dev/mem` を `open` してメモリにマップし、読み書きを行うプログラム例

```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;
}
```

I2C通信のためのプログラムを作成

START conditionやSTOP conditionといったI2C特有の操作はカーネルモジュールi2c-devが自動的に行ってくれる。

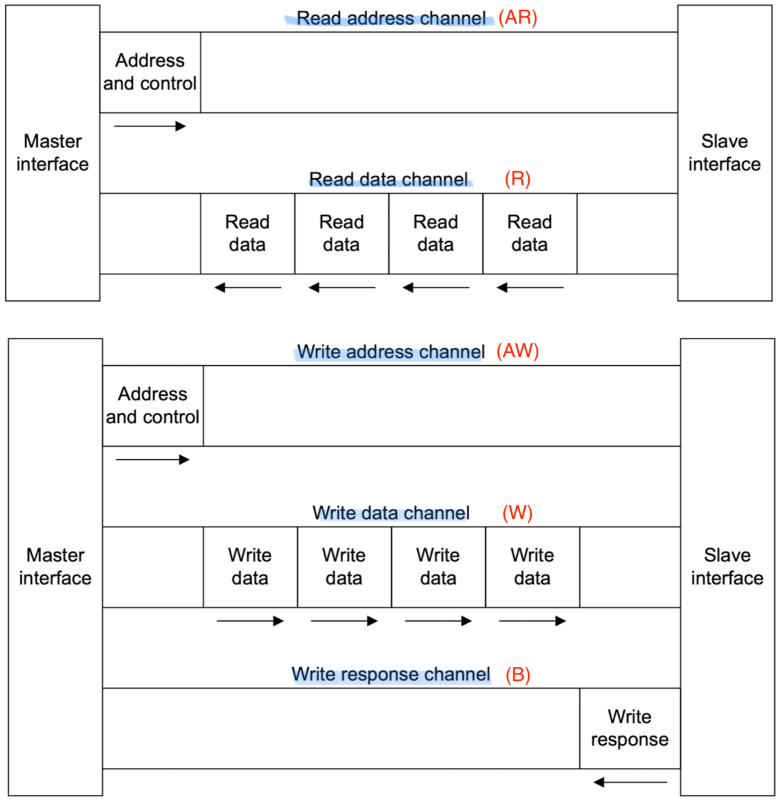
I2C-readのための関数

```
#include <linux/i2c-dev.h>  // I2C_SLAVE
#include <stdint.h>         // uintX_t
#include <sys/ioctl.h>      // ioctl()

static const char* dev_file =
    "/dev/i2c-4";  // Si5345 (U1) on Endcap SL, connected from the port 3 on
                   // pca9548 (U23)
static const uint8_t dev_addr = 0x68;  // Si5345 slave address

int i2c_read(uint8_t reg_addr, uint8_t* pdata) {
    // Open the I2C device file
    int fd = open(dev_file, O_RDWR);

    // Set I2C slave address
    int ret = ioctl(fd, I2C_SLAVE, dev_addr);

    // I2C read
    write(fd, &reg_addr, 1);  // set register address
    read(fd, pdata, 1);       // read its data

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

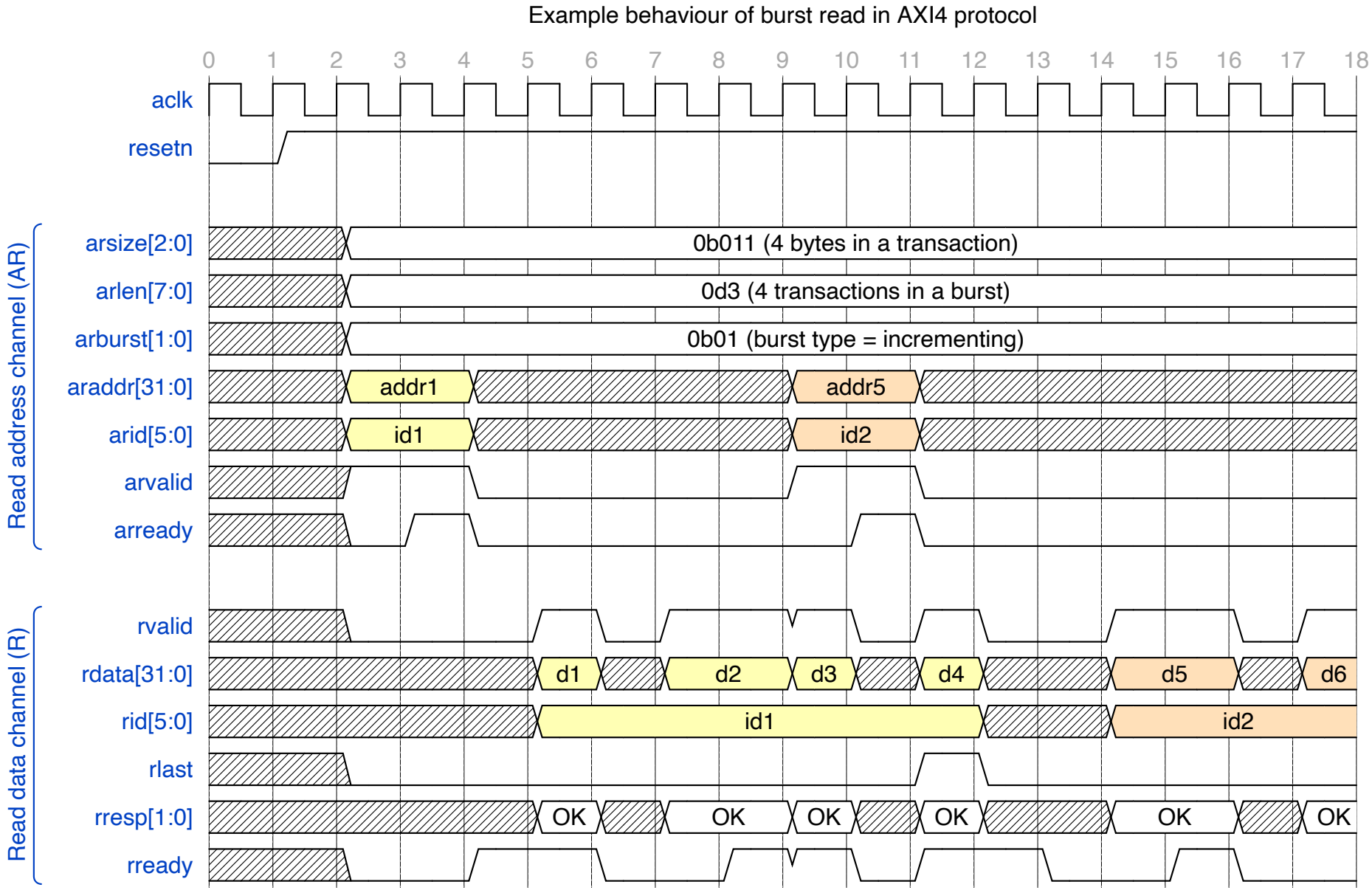
I2C-writeのための関数

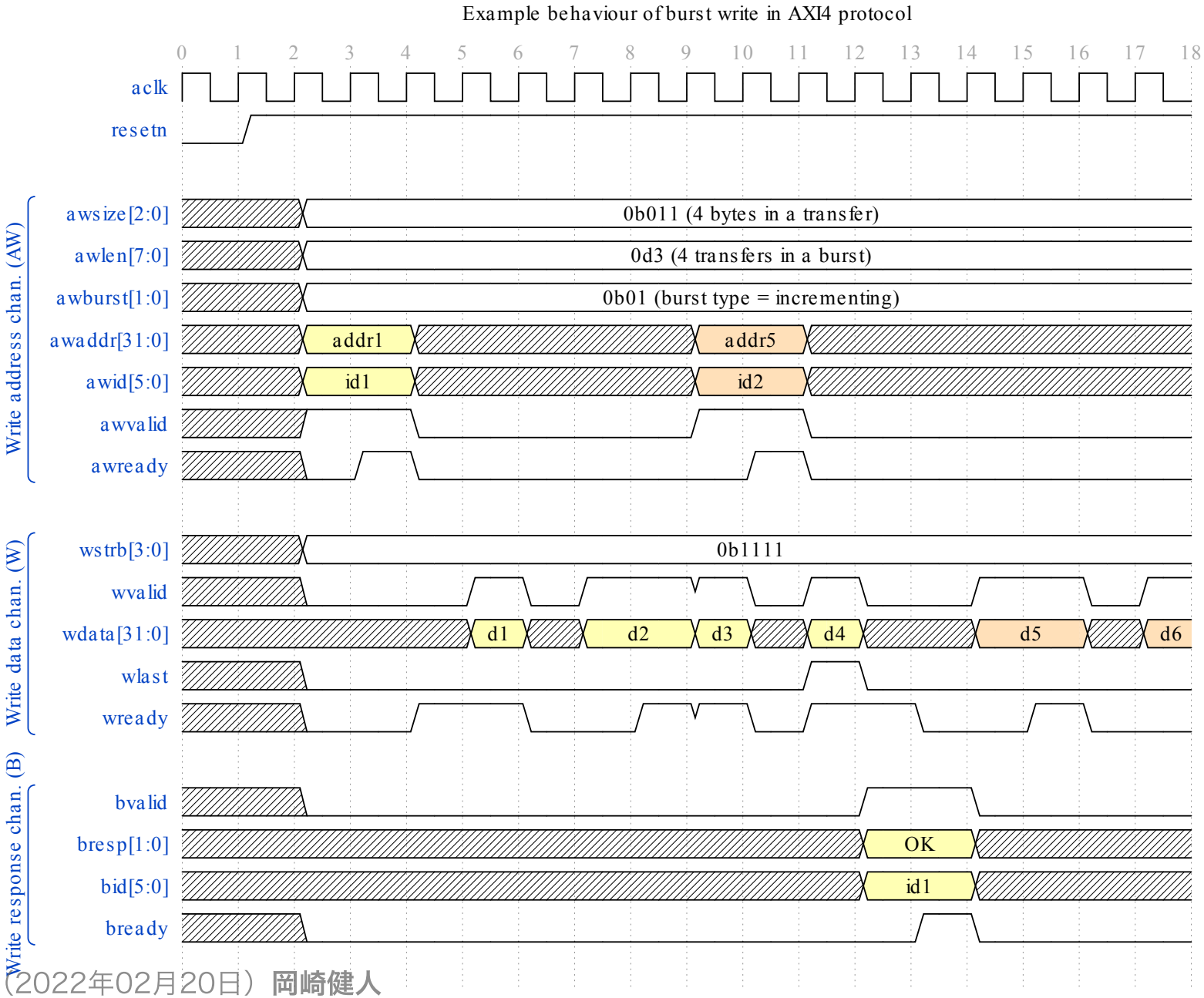
```
#include <linux/i2c-dev.h>  // I2C_SLAVE
#include <stdint.h>         // uintX_t
#include <sys/ioctl.h>      // ioctl()

static const char *dev_file =
    "/dev/i2c-4";  // Si5345 (U1) on Endcap SL, connected from the port 3 on
                   // pca9548 (U23)
static const uint8_t dev_addr = 0x68;  // Si5345 slave address

int i2c_write(uint8_t reg_addr, uint8_t data) {
    // Open the I2C device file
    int fd = open(dev_file, O_RDWR);

    // Set I2C slave address
    int ret = ioctl(fd, I2C_SLAVE, dev_addr);

    // I2C write
    uint8_t packet[2];
    packet[0] = reg_addr;
    packet[1] = data;
    write(fd, packet, 2);

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

AXI4 protocol



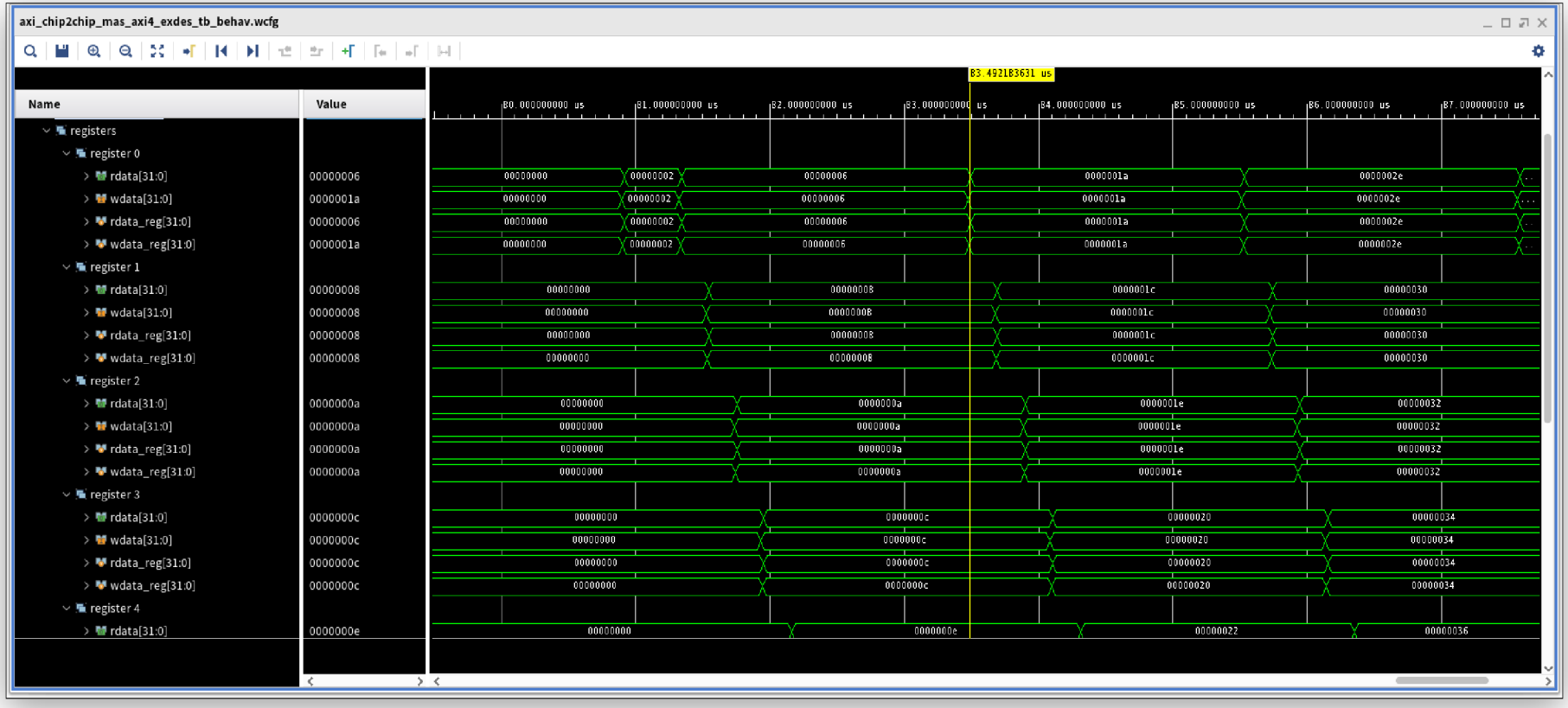
- AXI4では、複数の信号線を用途ごとにまとめたチャンネルという概念がある
  - `AR`, 読み出しアドレスチャンネル
  - `R`, 読み出しデータチャンネル
  - `AW`, 書き込みアドレスチャンネル
  - `W`, 書き込みデータチャンネル
  - `B`, 書き込み応答チャンネル
- 各チャンネルは`VALID`と`READY`信号を持ち、独立に動作できる
- すべてにトランザクションを識別するIDのための信号があるため、マスターとスレーブで整合性を保つことができる


Example behaviour of burst read in AXI4 protocol
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
aclk
resetn
Read address channel (AR)
arsize[2:0]
0b011 (4 bytes in a transaction)
arlen[7:0]
0d3 (4 transactions in a burst)
arburst[1:0]
0b01 (burst type = incrementing)
araddr[31:0]
addr1
addr5
arid[5:0]
id1
id2
arvalid
arready
Read data channel (R)
rvalid
rdata[31:0]
d1
d2
d3
d4
d5
d6
rid[5:0]
id1
id2
rlast
rresp[1:0]
OK
OK
OK
OK
OK
OK
rready


Example behaviour of burst write in AXI4 protocol
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
aclk
resetn
Write address chan. (AW)
awsize[2:0]
0b011 (4 bytes in a transfer)
awlen[7:0]
0d3 (4 transfers in a burst)
awburst[1:0]
0b01 (burst type = incrementing)
awaddr[31:0]
addr1
addr5
awid[5:0]
id1
id2
awvalid
awready
Write data chan. (W)
wstrb[3:0]
0b1111
wvalid
wdata[31:0]
d1
d2
d3
d4
d5
d6
wlast
wready
Write response chan. (B)
bvalid
bresp[1:0]
OK
bid[5:0]
id1
bready

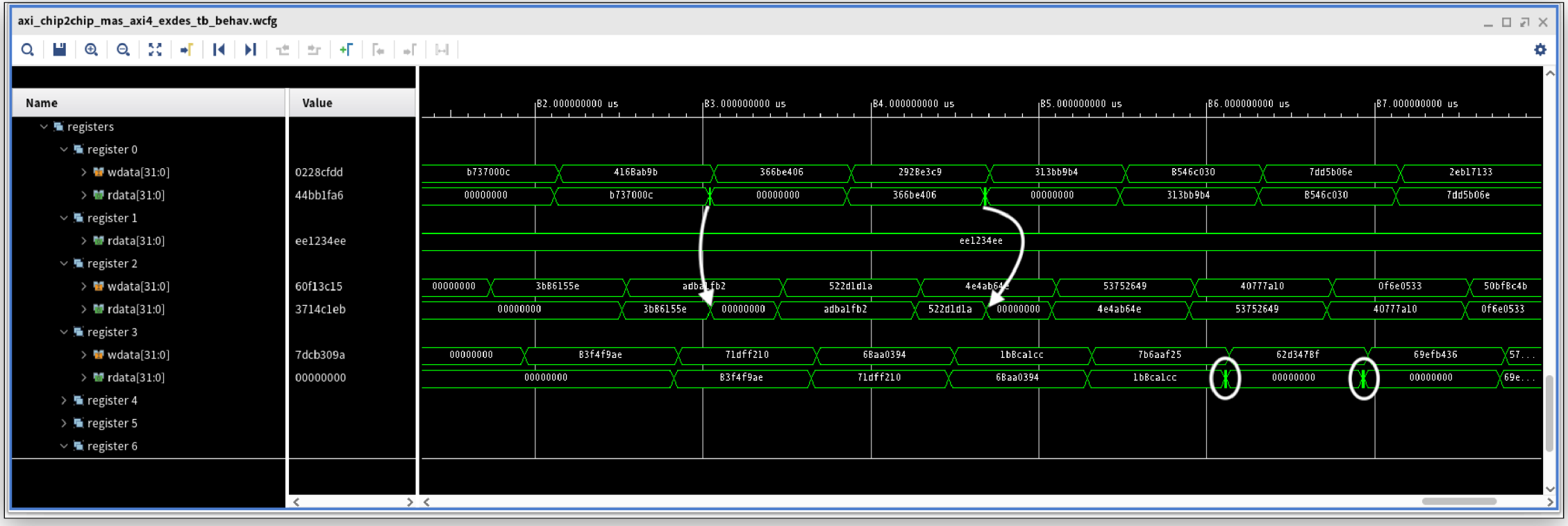
テストベンチシミュレーションの結果

- 10個のレジスタを用意し、それらに正しく書き込まれ、マスター側でも書き込まれたデータを読み出せていることを確認した



- 次の4つのレジスタを用意し、動作が正しく行われていることを確認した

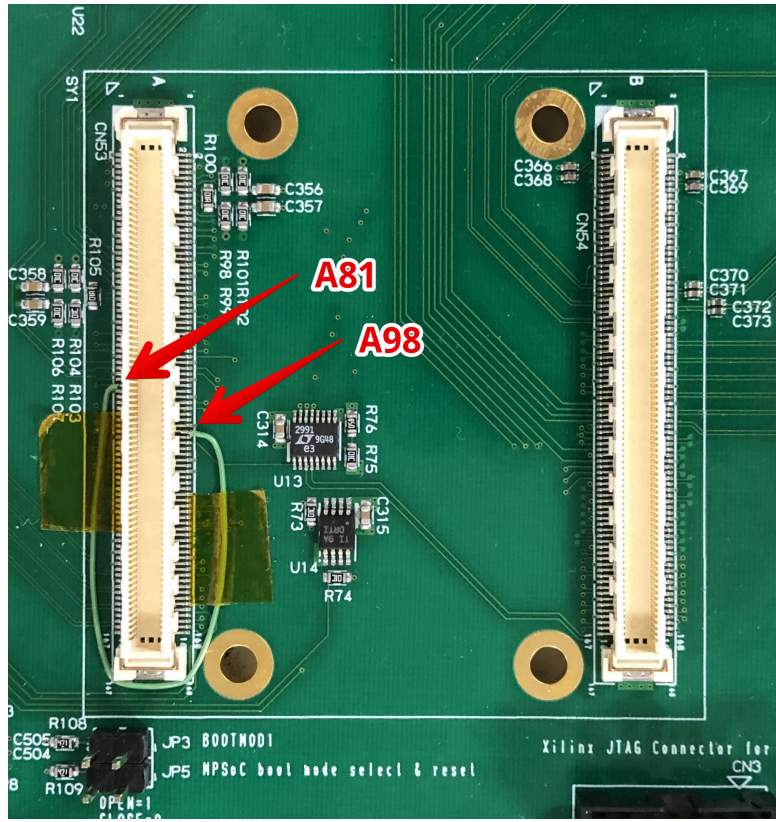
instance	roll	動作
`register_0`	reset register	自身の値の0ビット目をリセット信号として自身とcontrol registerに配布
`register_1`	status register	固定の値 `0×ee1234ee` を出し続ける
`register_2`	control register	マスターにより値が単純に読み書きされる
`register_3`	pulse register	自身の値の0ビット目を自身のリセット信号として使用



- マスターからランダムなデータを生成
- このテストベンチシミュレーションによりregister interfaceが正常に動作することを確認した

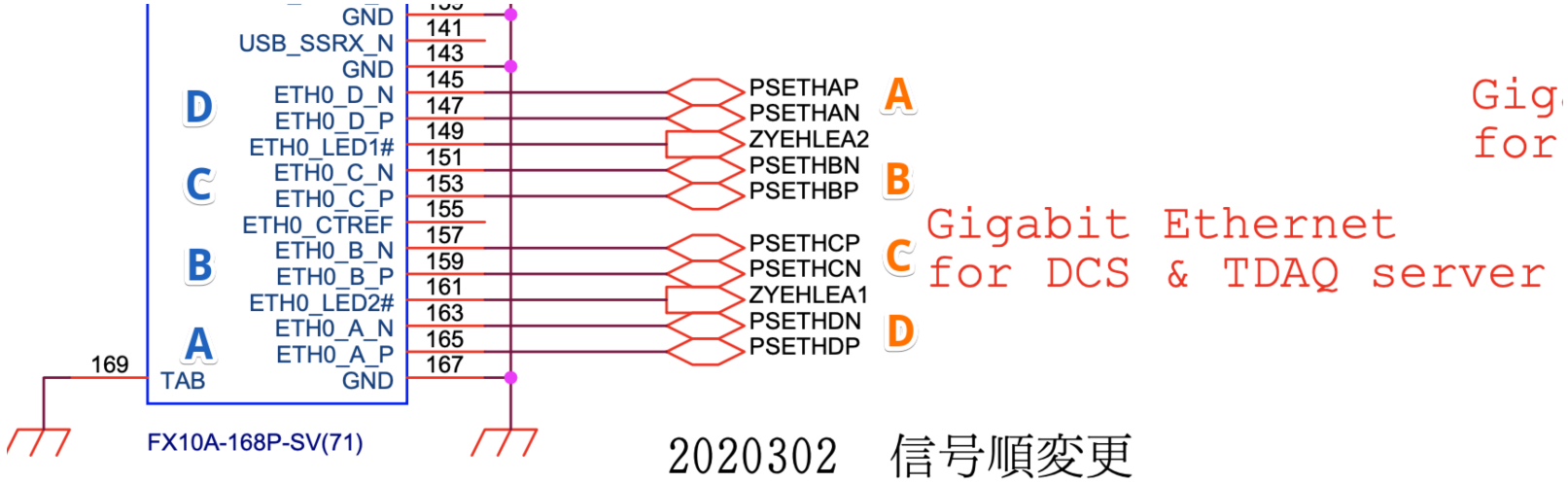
CentOS 7のブート

- Endcap SLはMercury+ PE1を参考にして設計されており、確立したパスで開発できる
- ブートファイルの準備
  - PSの設定はMercury XU5 PE1 reference designを参照
  - PLには自作の簡単なロジックを構築
  - PetaLinuxでブートファイルを作成しSDにコピー
- Card Detection（CD）信号線が正しいコネクタピンに接続しておらず、U-Bootでストップ
- コネクタピン同士をジャンパー接続、あるいはCDを使わないようにすれば、CentOS 7が起動



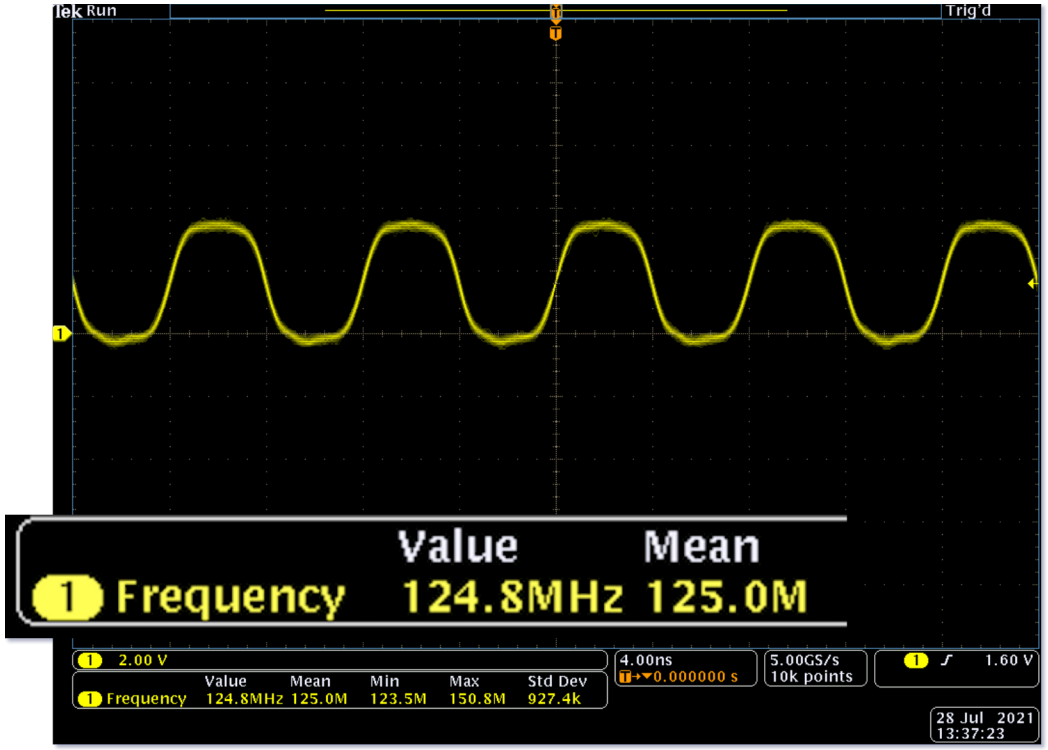
Ethernetの開通（PS/PL Gigabit Ethernet）

- MPSoCはEthernet経由でコマンドの受信やデータベースの参照をするためのインターフェイスとなる
- Mercury XU5 & PE1でEthernetは確立していたためEndcap SLでも同様に開通できるはずであったが、想定通りに動作しなかった
  - SL側で(A, B, C, D)とすべきところを(D, C, B, A)とスワップして配線されていた



- スワップし直したLANケーブルを自作したところ開通した
  - Endcap SL第2試作機では修正する予定
- IPアドレス、MACアドレスの設定にはCentOS 7の `nmtui` コマンドや `uEnv.txt` で設定できた

同じ技術で、ZCU102評価ボード上のSi570クロックオシレータのレジスタを操作し、125 MHzを生成



Zynq MPSoCとXCVU13Pの温度センサをレジスタをI^2C-readすることで温度を確認

```
(sl-xu5-01) temperature $ sudo ./zynqmp --time=10
Local temperature around ZynqMP.
  0.0 sec: temp = 30.5625 DegC
  0.5 sec: temp = 30.5625 DegC
  1.0 sec: temp = 30.5625 DegC
  1.5 sec: temp = 30.5625 DegC
  2.0 sec: temp = 30.5625 DegC
  2.5 sec: temp = 30.5625 DegC
  # この辺で卓上扇風機をOFFにした
  3.0 sec: temp = 30.6875 DegC
  3.5 sec: temp = 30.6250 DegC
  4.0 sec: temp = 30.6875 DegC
  4.5 sec: temp = 30.6875 DegC
  5.0 sec: temp = 30.7500 DegC
  5.5 sec: temp = 30.7500 DegC
  6.0 sec: temp = 30.8125 DegC
  6.5 sec: temp = 30.8750 DegC
  7.0 sec: temp = 30.8750 DegC
  7.5 sec: temp = 30.9375 DegC
  8.0 sec: temp = 31.0000 DegC
  8.5 sec: temp = 31.0625 DegC
  9.0 sec: temp = 31.0625 DegC
  9.5 sec: temp = 31.1250 DegC
```

たとえば、KEKにあるEndcap SLに東京からアクセスし、FireFly（光通信モジュール）のIBERT試験ができるようになった