2021 年度　修士論文

System-on-a-Chip を用いた
組み込みシステム開発基盤の構築と FPGA 間通信の実証
— 次世代高エネルギー実験制御系への応用

(Development of an embedded system using System-on-a-Chip
and demonstration of communication between FPGAs
— Application to a control system
of next generation high-energy physics experiments)

東京大学大学院理学系研究科物理学専攻
素粒子物理国際研究センター
石野研究室

博士課程（前期課程）2 年
学籍番号 35–206023

岡崎健人

2022 年 1 月 6 日

概要

2027 年から開始される高輝度 LHC-ATLAS 実験において、ミューオントリガーエレクトロニクスはすべて刷新される。ミューオントリガーシステムの後段回路 Sector Logic（SL）は大規模 FPGA を搭載し、送信されたすべてのヒット信号を用いてオンライン演算や読み出しがなされる。SL の制御系高度化のため、Zynq UltraScale+ MPSoC を搭載した市販のメザニンカード（Mercury XU5）を導入する。Zynq MPSoC の OS として CentOS 7 を採用し、ボード上の素子の制御を行う。また ATLAS control network（ATCN）と Endcap SL のインターフェースとなる。さらに AXI Chip2Chip を用いた高速シリアル通信で大型 FPGA のレジスタ制御や読み出しデータのダンプを行うことで、トリガーや読み出しのロジックに関わるパラメータの設定や、コミッショニングやキャリブレーション時のテストリードアウト環境のコンフィグレーション、フロントエンド FPGA のリモートでの操作を行う。

本研究ではまず Zynq MPSoC で SD カード起点で CentOS 7 を起動し、基本的なファームウェア開発を行なった。これにより組み込みシステムとしての Zynq MPSoC 開発の技術基盤を確立した。また 2021 年 10 月に完成した Endcap SL 第 1 試作機上の Zynq MPSoC で CentOS 7 を起動し、クロックジェネレータや温度センサなどの素子をソフトウェアから操作できることを確認し、それらの検証の中で Endcap SL 第 1 試作機上の配線上の問題を指摘した。さらに Zynq MPSoC で Xilinx Virtual Cable という技術を使い、ネットワーク越しに Zynq MPSoC から大型 FPGA のプログラムやデバッグができることを示した。AXI Chip2Chip と Gigabit transceiver を用いた高速シリアル通信のリンクも確立し、Zynq MPSoC から大型 FPGA のメモリにアクセスできることを確認した。大型 FPGA のレジスタ制御に用いられる、AXI Chip2Chip とレジスタとのインターコネクトの開発も行なった。これにより HL-LHC に向けた Endcap SL の Zynq MPSoC 周りの開発基盤が整った。

本研究は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好例であり、本研究におけるボードデザインや、ソフトウェアやファームウェアの実装の知見および運用経験は、今後の物理実験における制御系のデザインにも活用できる。本研究では高エネルギー実験における制御系高度化のための 1 つの標準型を確立し、重要な技術基盤を築いた。

目次

図目次

表目次

コードブロック目次

第 1 章

高輝度 LHC-ATLAS 実験における Level-0 Muon Trigger System

1.1　ATLAS 検出器における TGC

欧州原子核研究機構（CERN）はスイスジュネーブ郊外に所在する世界最大規模の素粒子物理研究機関である。その地下およそ 100 m にはフランスとの国境をまたぐ形で Large Hadron Collider（LHC）が敷設されている。LHC は周長 26.66 km の陽子-陽子衝突型円形加速器で、2022 年から開始される運転（Run 3）での重心エネルギーは 13.6 TeV、瞬間ルミノシティはピークで $2.0 \times 10^{34}\,\mathrm{cm^{-2}\,s^{-1}}$ になる。

図 1.1 のように LHC の 4 つの衝突点にはそれぞれ Large Hadron Collider beauty（LHCb）、A Large Ion CollidEr（ALICE）、Compact Muon Solenoid（CMS）、そして A Toroidal Lhc ApparatuS（ATLAS）という粒子検出器が置かれている。

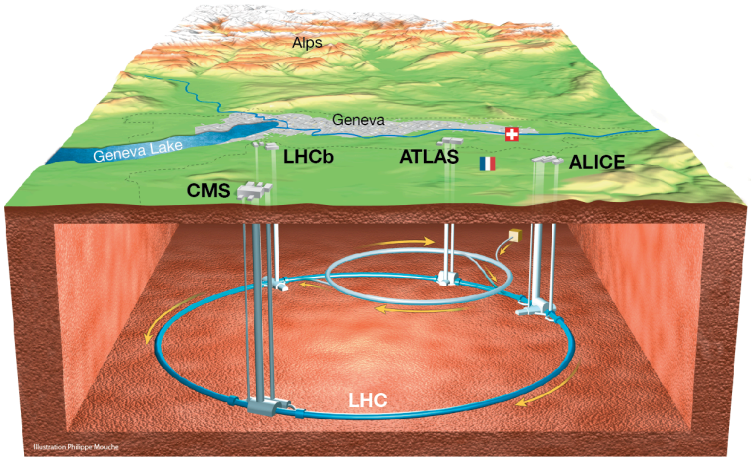
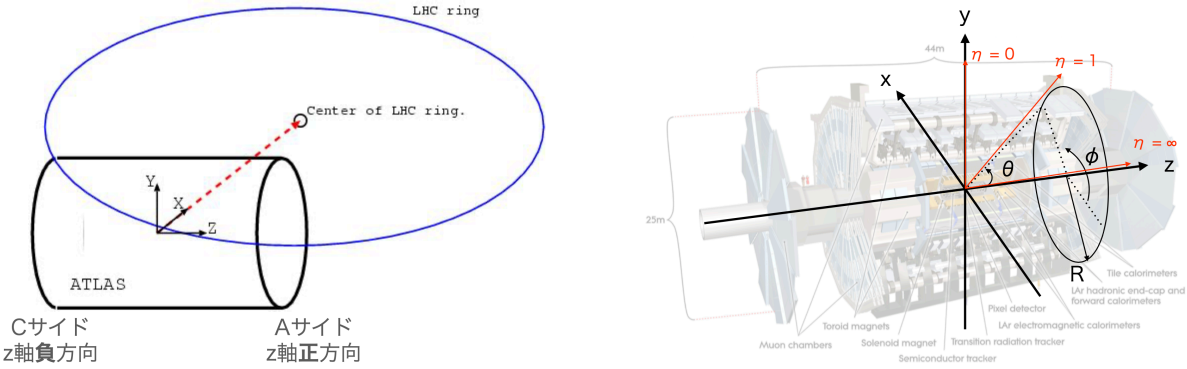


図 1.1　LHC と 4 つの検出器（Mouche 2014）

1.1.1　ATLAS 検出器

ATLAS 検出器は全長 44 m、直径 25 m の円筒状をしており、質量 7 kt の大型汎用検出器である。LHC によって 7 TeV まで加速された 10^{11} 個の陽子を含むバンチが 40.079 MHz で検出器中心で交差し（24.95 ns に 1 回のバ



(a) 直交座標系 (b) 円筒座標系

図 1.2 ATLAS における座標系の定義

ンチ交差頻度)、その中で生じる陽子陽子衝突の測定を行う。

陽子バンチ同士が衝突する ATLAS 検出器の中心を衝突点（Interaction Point、IP）とよぶ。ATLAS 実験においては 2 つの座標系、直交座標系と円筒座標系が用いられる。直交座標系では図 1.2(a) のように IP を原点とし、LHC の中心を向く方向に x 軸、鉛直上向きに y 軸をとり、右手系に従って z 軸をとる。また図 1.2(b) のように、z 軸からの距離を R、z 軸となす角を θ、x–y 平面となす角を ϕ とする円筒座標系を採用する。さらに θ を用いて

$$\eta \equiv -\ln\left[\tan\left(\frac{\theta}{2}\right)\right]$$

と定義される擬ラピディティという量も用いられる。

ATLAS 検出器は円筒形をしているが、円筒の側面をバレル部（barrel）、円筒の底面部をエンドキャップ部（endcap）という。本研究で取り扱うミューオントリガーは、バレル部とエンドキャップ部で異なる検出器、トリガーロジックを実装するが、おおよそ $|\eta| = 1$ のあたりがバレルシステムとエンドキャップシステムの境界である。また図 1.2(a) において $z > 0$ の側を A-side、$z < 0$ の側を C-side とよぶ。

1.1.2 Thin Gap Camber

ATLAS 検出器の両端には TGC（Thin Gap Chambers）が設置されている。これはエンドキャップ部のミューオントリガーを担当する検出器である。TGC はエンドキャップトロイド磁石の内側にある Endcap Inner（EIL4）と外側にある Big Wheel（BW）から構成される（図 1.3）。TGC は ATLAS 検出器の最外層に設置されており、荷電粒子の中で最も透過力の大きいミューオンを主に検出する。多層の位置情報を統合することで横運動量を概算し、あらかじめ設定した運動量閾値を超えたミューオンを含んだ事象をトリガーする。

TGC 検出器を構成する最小単位は、1 m から 2 m 四辺の台形型をした multi-wire propotional chamber（MWPC）のユニットである。その構造を図 1.4 に示す。直径 50 µm のアノードワイヤーが 1.8 mm 間隔で並んでおり、そのアノード面から 1.4 mm 離れてカソードストリップがある。これにより 2 次元的な読み出しができる。ガス増幅部には二酸化炭素と n-C_5H_{12}（n-ペンタン）がそれぞれ 55%, 45% の割合で混合されたガスが充填されている。アノードとカソード間には約 2.9 kV の電圧が印加されていて、電離した電子の増幅を促す。これにより約 3×10^5 のガスゲインが得られている。

A-side、C-side それぞれ 7 層のガス層で構成される TGC BW は、3 つのステーションに分かれて設置されており、IP に近い順に M1、M2、M3 と呼ばれている。M1 は 3 層、M2 と M3 はどちらも 2 層のガス層からなるため、M1 は doublet、M2 と M3 は triplet とも呼ばれる。それらの断面図を図 1.5 に示す。

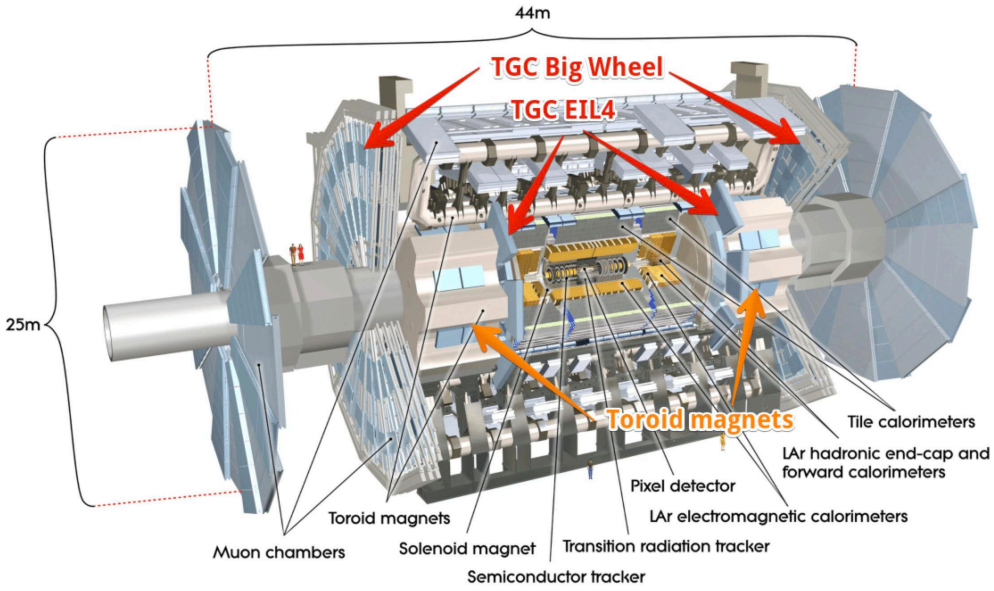


図 1.3 TGC Big Wheel と Endcap Inner

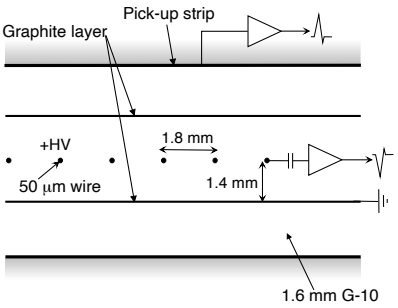


図 1.4 TGC の構造(The ATLAS Collaboration et al. 2008)

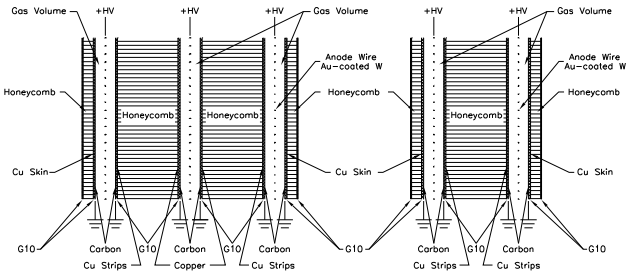


図 1.5 triplet と doublet の断面図(The ATLAS Collaboration et al. 2008)

TGC の $1.05 < |\eta| < 1.92$ の領域をエンドキャップ、$1.92 < |\eta| < 2.4$ の領域をフォーワードと呼んでいる。エンドキャップ領域を放射状に 48 分割、フォーワード領域を放射状に 24 分割した単位はトリガーセクターと呼ばれており、トリガー回路としてそれぞれ独立である。さらにエンドキャップ領域の 4 つのトリガーセクターとフォーワード領域の 2 つのトリガーセクターをあわせた単位を 1/12 セクターと呼ぶ。

1.2 Phase-II Upgrade

新物理探索とヒッグス粒子の精密測定のため、LHC は瞬間ルミノシティを $7.5 \times 10^{34}\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ に増強して高輝度 LHC（High-Luminosity LHC、HL-LHC）として運転を行う。このための検出器、トリガー読み出しシステムのアップグレードを Phase-II Upgrade という。アップグレードとオペレーションのロードマップを図 1.6 に示す。重心系エネルギーは 14 TeV になり、運転期間全体で $3000\,\mathrm{fb}^{-1}$ のデータを収集する。

Run 3 でパイルアップ数（1 回のバンチ交差での衝突回数）は 60 であるのに対して、高輝度化により HL-LHC では 200 になる(ATLAS Collaboration 2017)。このような発生粒子の密度が高い環境においても、新粒子探索や精密測定の性能を高く保って物理成果を上げるべく、ATLAS 検出器を構成する検出器や、Trigger and Data AQuisition（TDAQ）システムに関わるエレクトロニクスもアップデートする。

図 1.6　2021 年 1 月に更新された LHC/HL-LHC アップグレード計画(CERN 2021)

1.2.1　L0Muon トリガーシステム

HL-LHC での ATLAS 実験において、トリガーシステムは大きく分けて 2 段階に分かれる。TDAQ システムの全体像を図 1.7 に示す。初段は Level-0（L0）トリガーと呼ばれ、ここでハードウェアベースの高速な事象選別を行い、1 MHz のトリガーレート帯域に収まるように事象選別を行う。後段のトリガーシステムは Event Filter（EF）と呼ばれ、ソフトウェアによる精密な事象選別を行う。事象レートは最終的に 10 kHz まで落とされ、この選別をパスした事象の情報が CERN の Permanent Storage に保存される。なお現行（2022 年より開始する Run 3）ではハードウェアベースの初段トリガーレートが 100 kHz、ソフトウェアを用いた事象の記録レートが 1 kHz である。また初段トリガー判定に使える時間を 2.5 µs から 10 µs に拡張することでより洗練されたトリガーアルゴリズムを使えるようにする。

図 1.7 にあるように、L0 トリガーシステムは L0Muon、L0Calo、MUCTPI、CTP などから構成される。本論文は L0Muon（Level-0 Muon Trigger System）に関するものである。

HL-LHC における L0Muon は Endcap Sector Logic（Endcap SL）、Barrel Sector Logic（Barrel SL）、MDT Trigger Processor、NSW Trigger Processor から構成される。Monitored Drift Tubes（MDT）はミューオン検出器全域（$|\eta| < 2.7$）でミューオンの運動量を精密に測定する検出器である。また New Small Wheel（NSW）はエンドキャップトロイド磁石の内側に設置されたガス検出器であり、$1.3 < |\eta| < 2.7$ の範囲をカバーする。NSW Trigger Processor は NSW のヒット情報から飛跡再構成を行い飛跡情報を Endcap SL に送る。Endcap SL は他にも PS ボードから TGC のヒット情報、RPC（BIS78）からのヒット情報、タイルカロリメータからのエネルギー情報を受け取る。Endcap SL はこれらの情報からミューオンを検出し、飛跡再構成をして MDT Trigger Processor に渡す。MDT Trigger Processor では MDT からのヒット情報と Endcap SL からの再構成された飛跡情報をもとに、より精密な運動量測定を行い、その結果を Endcap SL に返す。最終的なミューオンの飛跡候補は Level-0 Muon Central Trigger Processor Interface（MUCTPI）へと送られる。バレル部でのトリガーは RPC、タイルカロリメータ、MDT Trigger Processor からの情報をもとに Barrel SL によって下され、ここでの飛跡情報も Level-0 MUCTPI へ送られる。

Front-End LInk eXchange（FELIX）は Level-0 Accept（L0A）信号を各検出器のフロントエンド回路に発行す

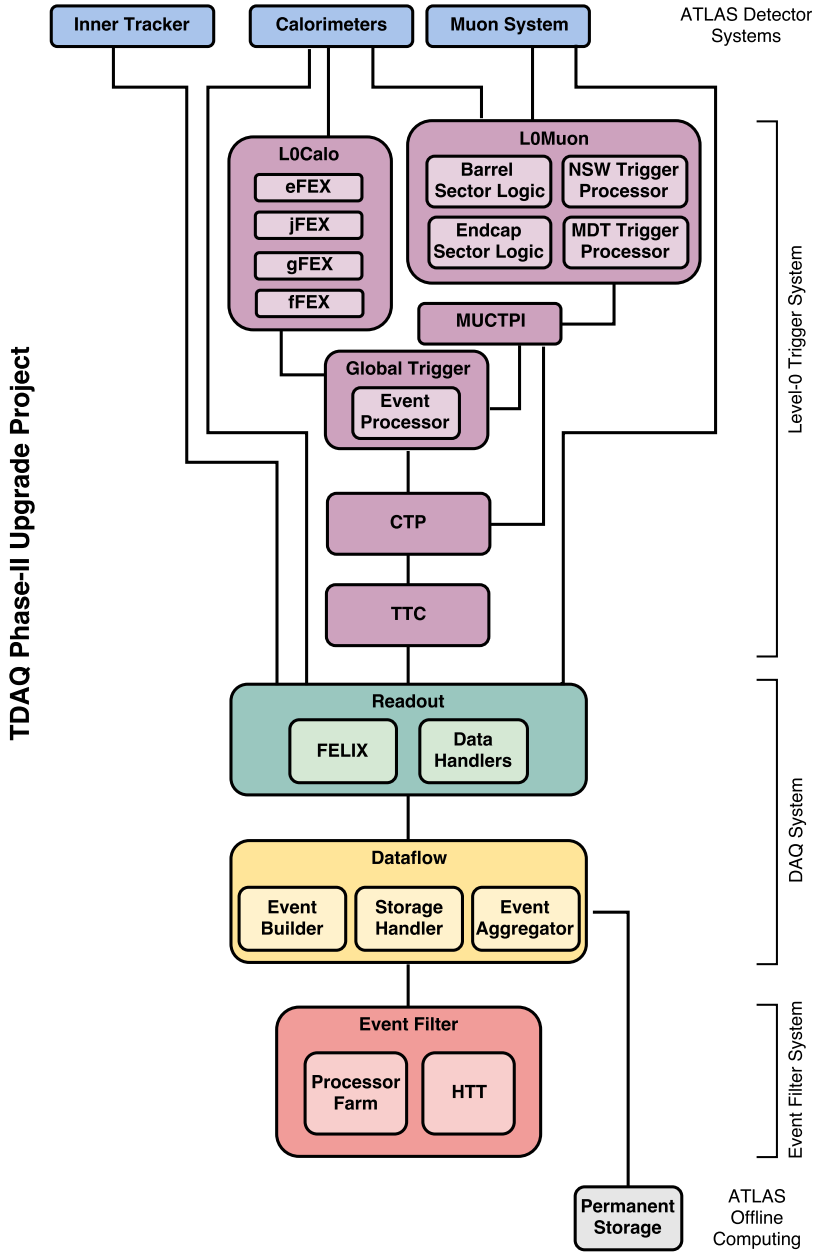


図 1.7 HL-LHC での ATLAS 実験における TDAQ システムの全体像(ATLAS Collaboration 2017)

る。L0A 信号で許可された BCID のイベントの情報は L0 バッファからソフトウェアベースの Level-1 トリガーシステムに読み出される。この段階でトリガーレートは 40 MHz から 1 MHz になる。

1.2.2 HL-LHC における TGC エレクトロニクス

図 1.9 に TGC エレクトロニクスシステムの概念図を示す。TGC 検出器から出力されるアナログ信号は、TGC ユニットの縁に設置された Amplifier-Shaper-Discriminator（ASD）ボード上の 4 つの ASIC によって増幅、波形整形、波高弁別されデジタル信号に変換される。このデジタル信号は Low Voltage Differential Signaling（LVDS）

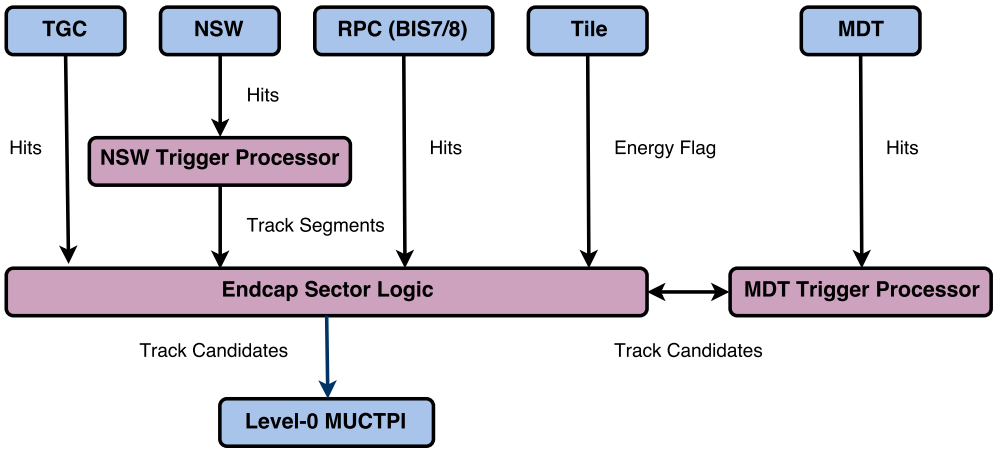


図 1.8 HL-LHC における endcap 部の L0Muon トリガーシステム（ATLAS Collaboration 2017）

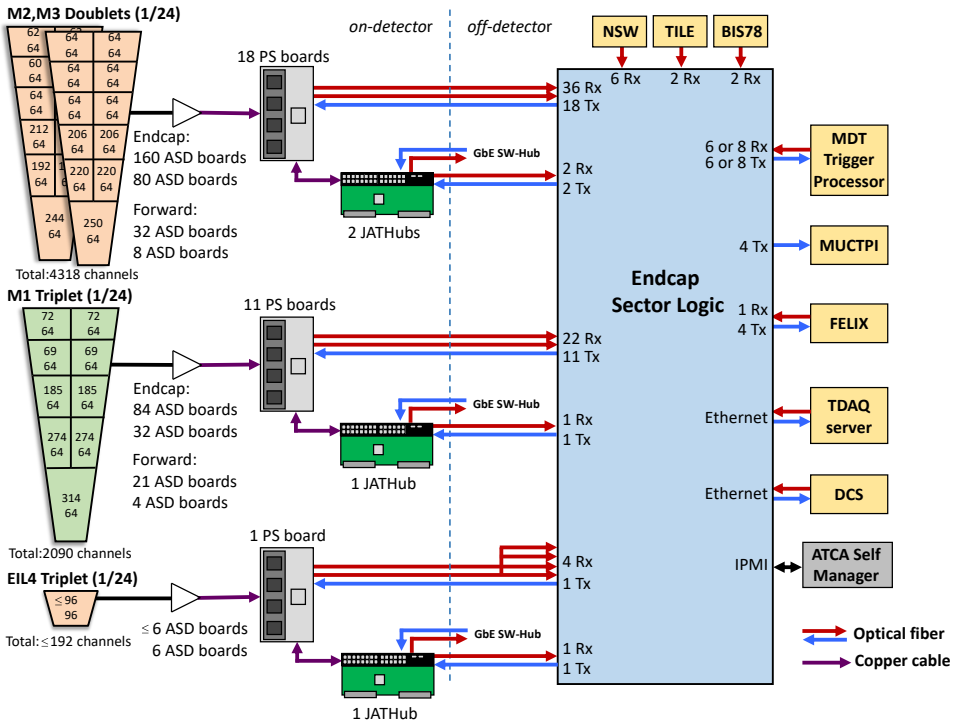


図 1.9 1/24 セクターをカバーする TGC エレクトロニクスシステムの概念図

で PS ボードに送られる。

PS ボード（図 1.10）には 8 つ[1]の Patch-Panel ASIC（PP ASIC）と 1 つの FPGA が搭載されている。ここでの目的はヒット信号のタイミング調整と BCID の割り振りを行って後段の Endcap SL に送ることである。

節 1.1.1 で説明したように陽子バンチ同士は 40.079 MHz で衝突しており、LHC の各システムはこれと同期した LHC クロックを使用している。しかし ASD からの LVDS 信号は、IP からの ToF（Time of Flight）の違いやケーブル長の違いにより、同じ衝突事象由来の信号でもタイミングが異なって到達する。PP ASIC はこのようなタイミ

[1] マザーボードとドーターボードにそれぞれ 4 つの PP ASIC が搭載されている。

ングの異なるヒット信号に遅延をかけることで、1 ns 以下の精度でタイミング調整を行い、LHC クロックと同期したヒット信号を出力する。さらにどの衝突事象由来の信号かを表す Bunch Crossing IDentification（BCID）も各信号に割り振る。

PS ボード上の FPGA は同ボード上の QSPI フラッシュメモリに格納された FPGA ビットストリームによりプログラムされる。8 つの PP ASIC によりタイミング調整されたヒット信号はこの FPGA 内の GTX トランシーバを用いて、高速シリアル通信によって後段の Endcap SL に転送される。60 m–100 m 離れた回路室にある Endcap SL とは SFP+ コネクタを介して光ケーブルで接続される。1 つの PS ボードに対して、1 バンチ交差あたりチャンネル数 32×8 PP ASICs $= 256$ のバイナリヒット情報と、64 bit の付属する他の情報の計 320 bit を Endcap SL に送るが、シリアル通信には 8B/10B エンコーディングが行われるため転送レートは $320\,\text{bit} \times 10/8 \times 40.079\,\text{MHz} = 16\,\text{Gbps}$ となる。このため 8 Gbps に対応した 2 本の光トランシーバを用いて Endcap SL にデータ転送する。Endcap SL からは同様にシリアル通信で PS ボードで用いる LHC クロックと PS ボード上 FPGA の制御信号（後述する Trigger, Timing, and Control（TTC）信号）が送られる。PS ボードは BW の 1/12 セクターに 4 つ（M1 で 2 つ、M2 と M3 で 2 つ）設置されている PS-Pack 内に収まっている。

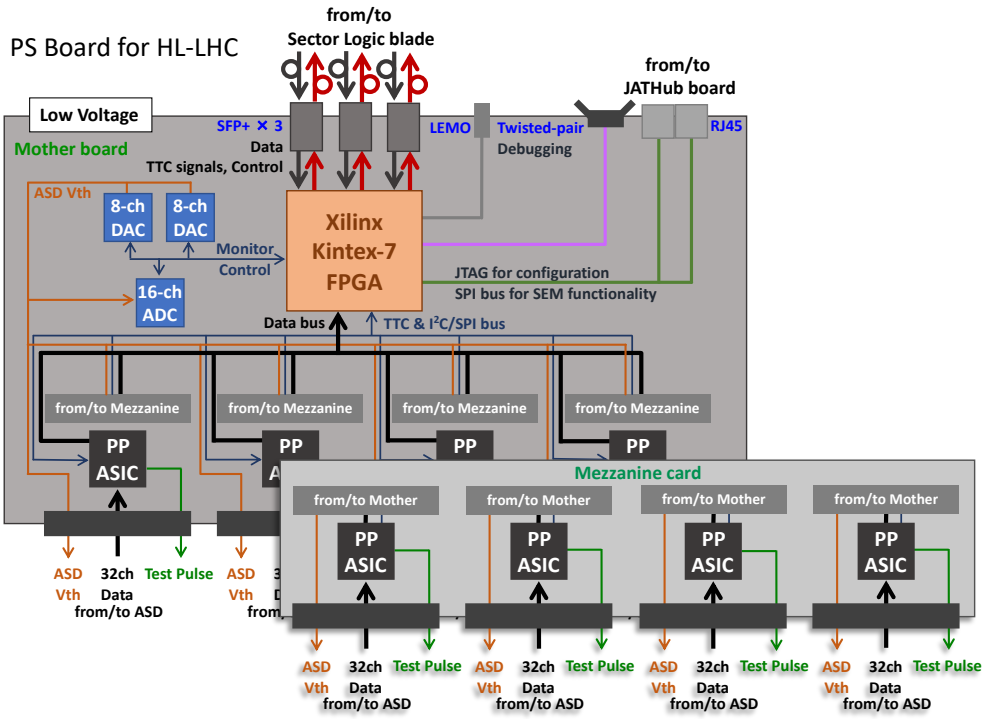


図 1.10　PS ボードの簡略化した回路図

運転中の ATLAS 実験ホールは高い放射線環境となるため、PS ボード上の FPGA 内で Single Event Upset（SEU、レジスタやメモリの bit がフリップすること）が発生する。PS ボード上の FPGA には SEU を自動回復させる仕組みが搭載されるが、それでも対応できない放射線損傷を検出し回復手続きを行うのが JTAG Assistance Hub（JATHub）である。JATHub には Zynq-7000 SoC がメインドライバとして搭載され、Linux が走り回路室と Ethernet 通信をする。また、この Linux で走るアプリケーションにより JTAG 線経由の PS ボード上の FPGA のプログラムが行われる。さらに隣の JATHub の SEU の監視と回復手続きをしたり、LHC クロックの位相をモニタする機能も持つ。JATHub は BW の 1/12 セクターの外側に設置された VME クレートに収まっている。

PS ボードを通過したすべてのヒット情報が Endcap Sector Logic（Endcap SL、図 1.11）に転送される。Endcap

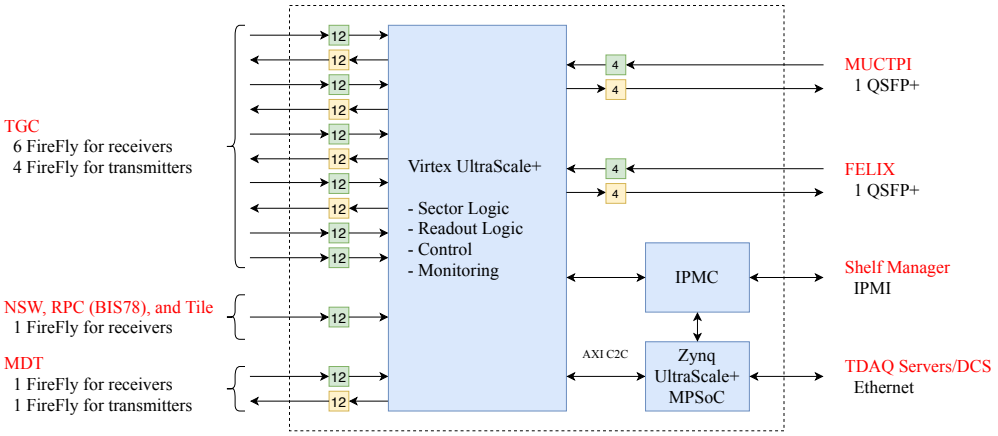


図 1.11 Endcap SL のダイアグラム

SL はまた RPC（BIS78）、NSW、タイルカロリメータ、MDT Trigger Processor（MDT TP）といったエンドキャップ部のミューオン検出器からもヒットのデータを受信し、飛跡情報を統合してトリガーを後段の Level-0 MUCTPI へ送信する。ここでの飛跡再構成には Xilinx Vertex UltraScale+ XCVU13P という大型の FPGA で、あらかじめ RAM に保存された飛跡のパターン（Look-Up Table、LUT）を照合することで、複雑な計算をすることなく高速な飛跡再構成を行う。図 1.12 のように、TGC BW のワイヤーの情報から $(\eta, \Delta\theta)$ を、ストリップの情報から $(\phi, \Delta\phi)$ をパターンマッチングアルゴリズムによって算出し、Coincidence Window（CW）を用いて p_{T} の概算をする。これを MDT TP に送信し、MDT からの情報を統合したより精密な p_{T} を受信する。最終的な飛跡情報を Level-0 MUCTP に送信し、Central Trigger Processor（CTP）でトリガー判定が下される。

実験ホールから離れた回路室には Advanced Telecommunication Computing Architecture（ATCA）クレートが 6 台設置されており、そこに合計で 48 台（1 台は 1/24 セクターを担当し、A-side と C-side の 2 サイド分）の Endcap SL が収まる。回路室の放射線レベルは低いため、加速器の運転中であってもここには人員のアクセスが可能である。さらに Endcap SL は FELIX から Trigger, Timing, and Control（TTC）信号（LHC クロックやリセット信号）を受け取り、PS ボードに LHC クロックを配布する。電源投入時には ATCA shelf manager が CERN が開発した IPMI Management Controller（IPMC）を通じてボードの制御やコンフィギュレーションを行う。Endcap SL にはまた Zynq UltraScale+ MPSoC が搭載された商用のメザニンカードを搭載し、Zynq MPSoC が XCVU13P を含めたボード上の素子の監視と制御を行う。

1.2.3 Endcap Sector Logic における Zynq MPSoC

Endcap SL 上には Zynq MPSoC が搭載され、同ボード上の監視や制御、XCVU13P を介した前段回路へのアクセスのためのインターフェイスとなる。その機能とそのためのパスを図 1.13 に示した。本論文はこの Zynq MPSoC に関するものである。

Zynq MPSoC は同ボードの大型 FPGA（XCVU13P）および温度センサや FireFly モジュール（光トランシーバ）などの素子の監視と制御を行う。特に XCVU13P の監視および制御には、AXI Chip2Chip という AXI4 プロトコルのブリッジを用いた高速シリアル通信で、メモリからのデータ読み出しやレジスタ制御を実現する。XCVU13P の JTAG 線を用いたプログラムや PS ボードの QSPI フラッシュメモリへのパラメータの書き込み、JATHub のブートファイルの書き込みなどの機能も持つ。Zynq MPSoC は Ethernet 通信により、TDAQ サーバや Data Control System（DCS）と Endcap SL のインターフェイスとなる。

Zynq MPSoC では Linux が走り、これらの監視や制御のためのアプリケーションが動作する。この Linux は

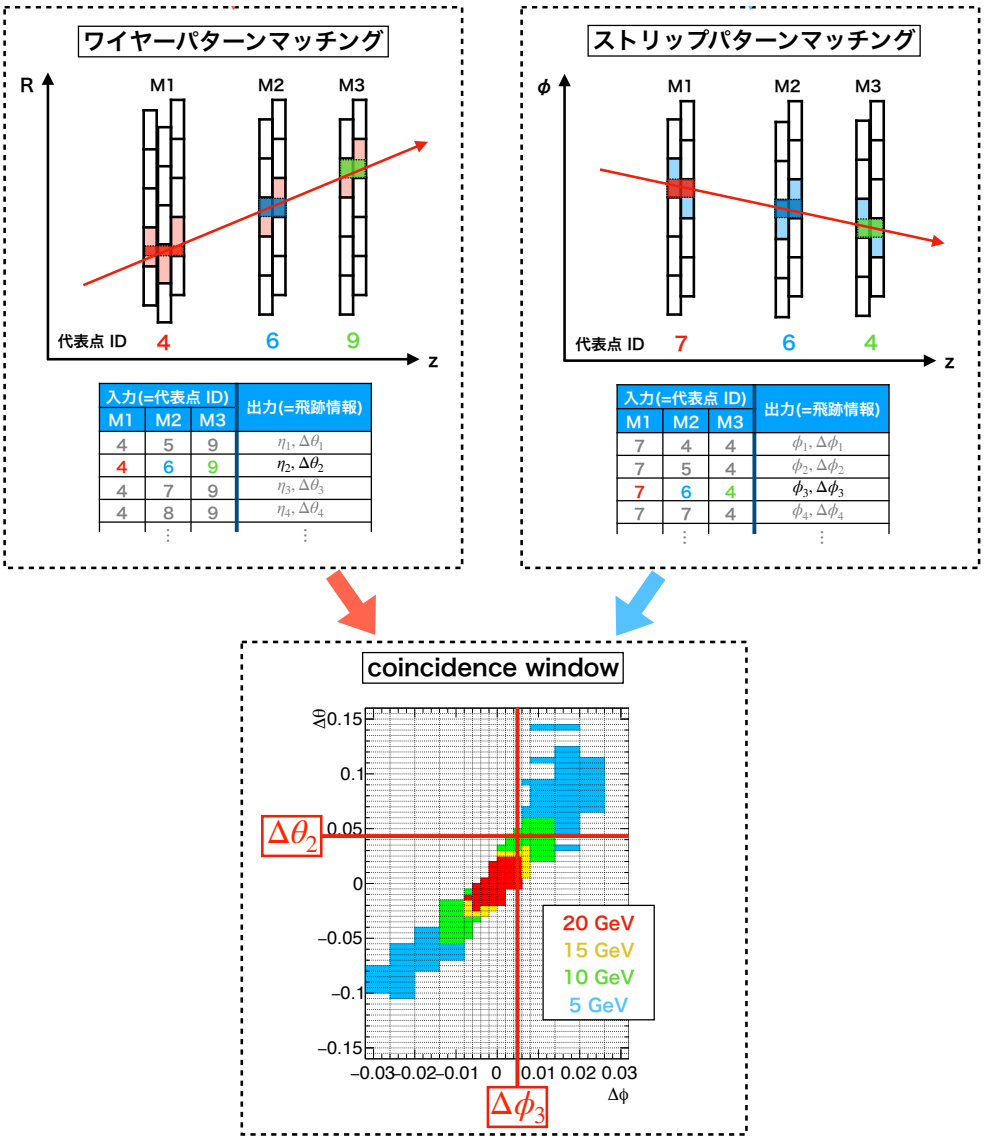


図 1.12　Endcap SL のパターンマッチングアルゴリズムと CW を用いた p_{T} の概算(三野 2020)。LUT を用いて飛跡の位置と角度を求め、角度情報から CW で p_{T} 閾値を求める。

microSD カードから起動する他、QSPI フラッシュメモリや Ethernet からも起動できる。本研究ではデモンストレーションするための OS として CentOS 7 を採用したが、HL-LHC プロジェクト全体で使用すべき OS については CERN の IT 部門で検討されている。

この Zynq MPSoC（XCZU-5EV-FSVC784-2-I）は市販されている Mercury XU5 というメザニンカードに搭載されたものを用いる。これによりボードの開発コストを下げ、アプリケーションやファームウェアの開発や試験が別のベースボードを用いて容易になり、さらにボードと切り離せることでメンテナンスの見通しをよくした。このメザニンカードおよび SoC は図 1.13 に示した用途や I/O 数、ATLAS 実験の DCS チームからの要求(Kopeliansky and Schlenker 2019)や ATLAS 実験 TDAQ サブシステムにおける SoC の要求(Kopeliansky and Vandelli 2019)を満たすように考慮して選定されている。さらに将来的に冗長性、拡張性をもってソフトウェア、ファームウェア開発ができるよう Zynq SoC ではなく現段階でミドルエンド以上の Zynq MPSoC を採用した。

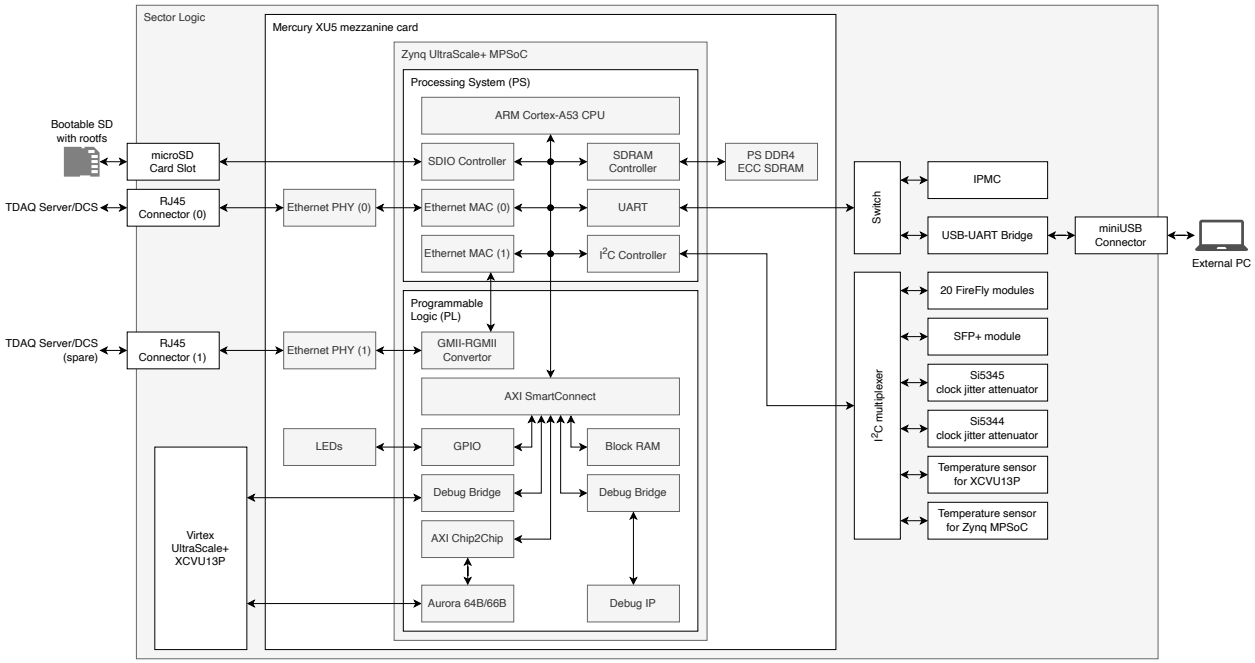


図 1.13　Endcap SL における Zynq MPSoC の機能の概略図

1.3　研究目的

節 1.2.3 で説明したように、Endcap SL 上の Zynq MPSoC は HL-LHC において重要な役割を担っている。本番で用いられる Mercury XU5 上の Zynq MPSoC の機能開発のため、Linux を走らせ基本的なファームウェア開発のための開発フローの確立が最優先事項である。Zynq MPSoC は組み込みシステムのコンパイルまでの手続きが煩雑で、開発の初期のハードルが高くなることも多い。さらに 2021 年 10 月に完成した Endcap SL 第 1 試作機上で正常に Linux が起動し、ボード上の素子との通信などの機能が使えることを確認する。また AXI Chip2Chip を使った高速シリアル通信といった、高エネルギー実験のエレクトロニクス制御のための機能実装と、そのデモンストレーションをするためのファームウェアの開発も行う。

市販品のメザニンカードと高エネルギー実験のためのフルカスタムのエレクトロニクスを組み合わせたシステムを使った初の運用例を確立し、次世代高エネルギー実験の基盤技術を切り開くことが本研究の目的である。

1.4　本論文の構成

第 1 章では HL-LHC での Zynq MPSoC の役割と、そのために必要な背景知識について説明した。第 2 章では Zynq MPSoC のソフトウェアやファームウェア開発に必要な要素や手法を詳細に説明し、実機を用いて基本的な開発パスが確立したことを述べる。第 3 章では Endcap SL 第 1 試作機上の Zynq MPSoC を使って行った動作検証について説明し、第 4 章で FPGA 間高速シリアル通信のためのファームウェア開発現状について述べる。最後に第 5 章では本論文のまとめと今後の展望について説明する。本論文にとって本質的ではないが重要な情報は付録に掲載した。

第 2 章

Zynq MPSoC を用いた組み込みシステムの開発

Mercury XU5 の Zynq MPSoC は Endcap SL や PS ボードの制御や監視を担う。そのために Zynq MPSoC で走る適切なアプリケーションソフトウェアやファームウェアが開発されるが、そのためにはまず組み込みソフトウェア、ファームウェアの横断的な開発基盤の確立が必要となる。ここでは ZCU102 評価ボードや Mercury+ PE1 ベースボードに載せた Mercury XU5 メザニンカードといった様々なプラットフォームで Linux を SD カードからブートさせるまでを実現し、ファームウェアの合成、ソフトウェアのビルド、ブートイメージのパッケージングを含む基本的な開発環境を構築した。これにより本番で動作するアプリケーションやファームウェアの開発および試験が容易にできるようになっただけでなく、これらの経験によって 2021 年 10 月に完成した Endcap SL 試作機のためのソフトウェア、ファームウェアの開発を素早く行うことができ、Zynq MPSoC 周辺機能の検証をスムーズに進めることができた。

節 2.1 では Zynq MPSoC とそれが搭載された Mercury XU5 メザニンカードについて紹介する。節 2.2 では Zynq MPSoC でソフトウェアとファームウェアがどのように起動されるかを説明し、それを元に必要な開発要素およびその作成方法を節 2.3 に示す。ZCU102 評価ボードと Mercury XU5 での Zynq MPSoC 開発例を節 2.4 と節 2.5 に説明する。また PetaLinux を用いた開発パスの参考のため、Zynq MPSoC が搭載された Ultra96-V2 というシングルボードコンピュータを用いた、FPGA による物体認識処理の高速化も試みた。Ultra96-V2 の開発については付録 B で説明している。

2.1 Zynq UltraScale+ MPSoC

プロセッサやメモリ、I/O ポートなど、システムにとって必要なパーツを 1 つのチップとして搭載したものを System-on-a-Chip（SoC）という。特に複数のプロセッサを搭載した SoC を MultiProcessor System-on-a-Chip（MPSoC）という。Xilinx® 社は Zynq®-7000 SoC や Zynq UltraScale+™ MPSoC を提供している。以降、Xilinx 社の Zynq UltraScale+ MPSoC を本稿では単に Zynq MPSoC と呼ぶことにする。

Zynq MPSoC の基本的なパーツを示したブロックダイアグラムを図 2.1 に示す。ここに示されているように、Zynq MPSoC は大きく分けて Processing System（PS）と Programmable Logic（PL）の 2 つのブロックから成り立っている。

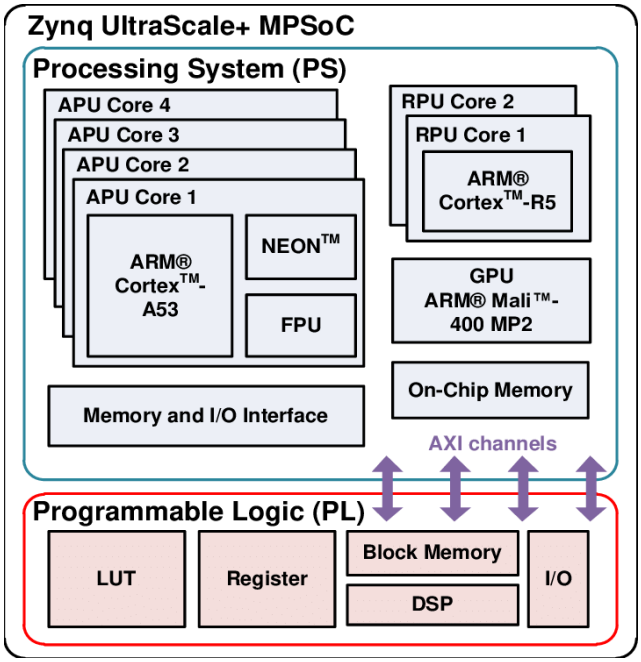


図 2.1　Zynq UltraScale+ MPSoC のブロックダイアグラム（Dzemaili 2021）

2.1.1　Processing System（PS）

Zynq MPSoC の PS には、Arm® 社のデュアルまたはクアッドの 64-bit Cortex®-A53 とデュアル Cortex-R5 が搭載されている。Cortex-A53 がデュアルであるかクアッドであるかや、それらの他に GPU も搭載しているかなどによって CG、EG、EV といった Zynq MPSoC のファミリの違いとして分類される。

デュアルまたはクアッド Arm Cortex-A53 プロセッサと 2 レベルのキャッシュなどをまとめた単位のことを Application Processing Unit（APU）と呼んでいる。APU は PS の主要部分であり、U-Boot や Linux カーネルなどのソフトウェアは基本的に APU 上で動作する。またデュアル Arm Cortex-R5 プロセッサとその周辺機能をまとめた単位を Real-time Processing Unit（RPU）という。RPU は処理時間に特定の制限時間があるリアルタイムアプリケーション[1] に使用される。

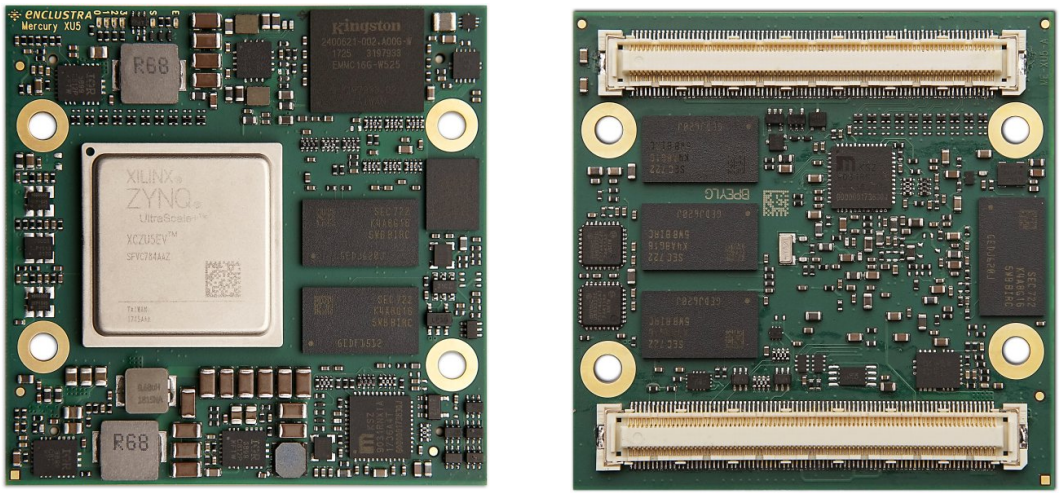
APU や RPU の他に、PS には Xilinx 社による MicroBlaze™ プロセッサを用いた Platform Management Unit（PMU）が備わっている。これは APU や RPU とは独立して動作する。PMU は電源投入やシステムのリセット、電源の管理、エラー通知、プロセッサ間割り込みなどの安全機能を担う。

さらにセキュアなブート、温度管理、暗号化キーの管理などのセキュリティー面の機能を担う Configuration Security Unit（CSU）が組み込まれている。

2.1.2　Programmable Logic（PL）

Zynq SoC や Zynq MPSoC はプロセッサ部（PS）とは別に内部に Field Programmable Gate Array（FPGA）を搭載しており、この部分は Programmable Logic（PL）と呼ばれている。CG、EG、EV デバイスによって違いがあるものの、基本的に PL には LUT の他に Block RAM や UltraRAM などのメモリ、高速通信のための Gigabit

[1] たとえば、車のエアバッグは車両が衝突してから発動するまでの時間に厳しい制約がある。またペースメーカなどの医療機器では、シンプルな機能であっても人命に関わるものなので、処理時間が保障されている必要がある。



(a) 表面 (b) 裏面

図 2.2 Mercury XU5 メザニンカード（Enclustra 2019c）

Transceiver（GT）などが備わっている。

PL をプログラムすることで利用できるようになる機能のことを、本論ではファームウェア[2]と呼称することにする。

図 2.1 にあるように、PS と PL 間のデータ転送には AXI4 が採用されている。Xilinx 社は FPGA の内部バスとしても AXI4 を積極的に採用しており、Vivado で使える IP のインターフェイスとして広く用いられている。内部バスとして AXI4 を用いることの利点として、Xilinx 社は次のことを挙げている（Xilinx 2017）。

- AXI4 プロトコルを理解するだけで多くの IP の動作を理解でき生産性を上げることができる。
- AXI4 プロトコルには 3 種類のプロトコルが存在するため、用途に合わせてフレキシブルに応用できる。
- Xilinx 社だけでなく多くの IP ベンダーは AXI4 をサポートしているため、可能性を広げることができる。

2.1.3 Mercury XU5 と Mercury+ PE1

スイスのチューリッヒに所在する Enclustra 社は、Xilinx 社や Intel 社の FPGA を搭載したハードウェアを販売している。Enclustra 社の販売している Mercury XU5（図 2.2）は Zynq MPSoC を搭載した、51.5 mm × 54.0 mm 四辺のメザニンカードである。Mercury XU5 には Zynq MPSoC の他に DDR4 SDRAM、Kingston 製の eMMC（embedded MultiMediaCard）、2 つの Gigabit Ethernet PHY、2 つの USB PHY、QSPI フラッシュメモリ、ユーザ LED などが搭載されている（図 2.4(a)）。裏面にはベースボードに接続するためのコネクタが 2 列ある。

Mercury メザニンカードのベースボードとして、Enclustra 社は Mercury+ PE1 を販売している。Mercury+ PE1 には PE1-200/300/400 の 3 種類のハードウェアがあるが、Mercury+ PE1-300 を開発に用いた。Mercury+ PE1-300 の top view を図 2.3 に、ハードウェアのブロックダイアグラムを図 2.4(b) に示す。Mercury+ PE1 には Mercury メザニンカードのためのコネクタはもちろん、microSD、microUSB、Ethernet、JTAG、DC 電源などのためのコネクタや、ユーザ LED、ピンヘッダ、ボタン、クロックジェネレータなど多数の機能が備わっている。Mercury メザニンカードを Mercury+ PE1 にマウントすることで、シングルボードコンピュータとして利用で

[2] 一般的に「ファームウェア」というのは ROM に書かれた書き換えできないプログラムのことを指すことが多いが、本論文では特に FPGA に組み込んだ機能のことを単に「ファームウェア」と呼ぶ。

きる。

Zynq MPSoC を直接ボードに搭載するのではなくメザニンカードに搭載されたものを使用することで、ボードの開発コストを下げることができる。特に DDR 通信をするメモリチップ周りの配線は高速アクセスが必要で、設計の制約が多いため一般に困難な箇所である。フルカスタムな実験では、既存のメザニンカードを使うことでこのような設計をメインボードとデカップルし、開発の難易度を下げるのは合理的である。また、高エネルギー実験の制御系のデザインでは、実験ごとの特殊な制約もゆるく、同様のデザインを異なるエレクトロニクスで再利用することも多い。その意味でも市販品のメザニンカードを活用し、メインボードのデザインから分離することは合理的である。さらに運用中に Zynq MPSoC に問題が発生した場合など、交換が容易になる。ルートファイルシステム（root filesystem、rootfs）や FPGA ビットストリームをメザニンカードではなくボード本体に搭載した SD カードに格納しておくことで、メザニンカードを交換しても以前と変わらない環境をすぐに再現できる。このように商用のメザニンカードを使用して、開発コストを下げ、メンテナンスの見通しをよくする。

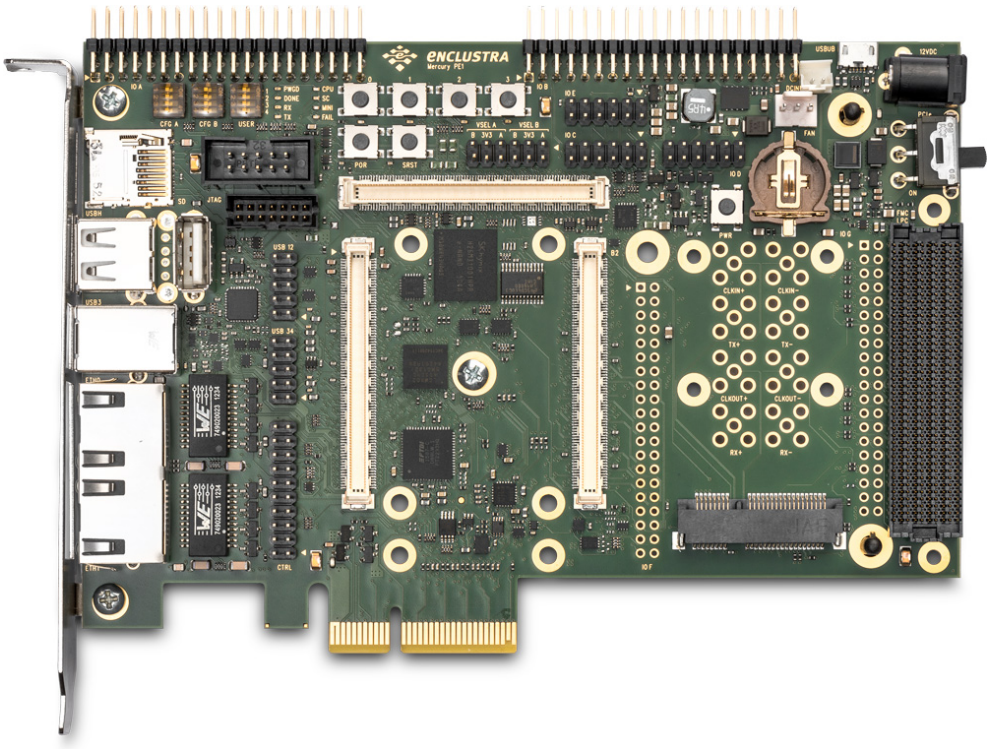
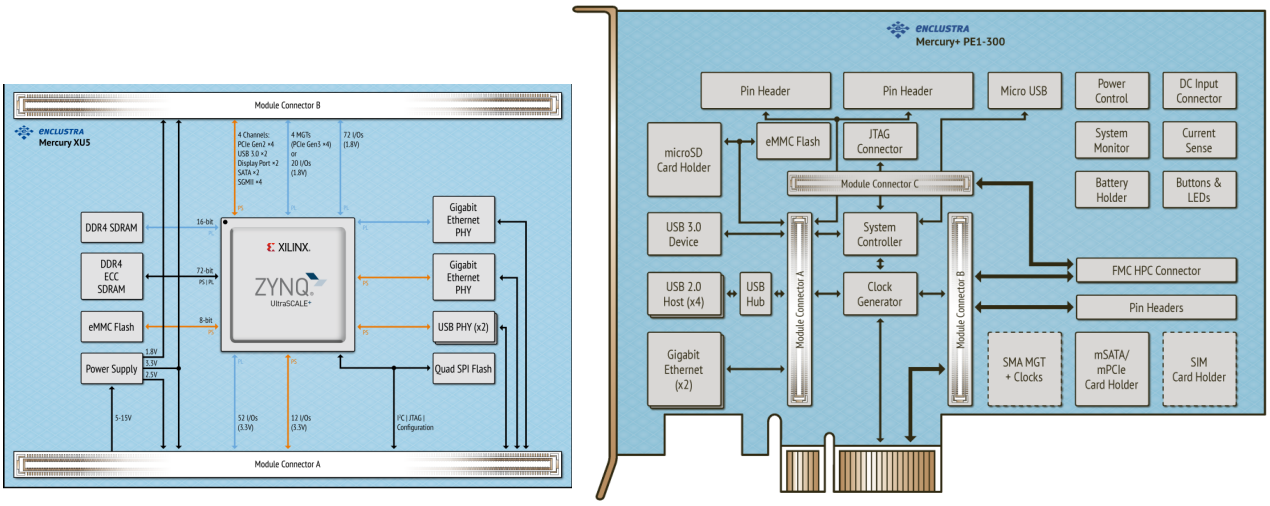


図 2.3　Mercury+ PE1 ベースボード（Enclustra 2019b）

2.2　Zynq MPSoC におけるブートシーケンス

Zynq MPSoC は eMMC、MMC（SD カード）、QSPI フラッシュメモリ、JTAG、USB などからのブートをサポートしている。ブートプロセスは複数の段階から成り、PMU と CSU で管理および実行される。Zynq MPSoC で開発やデバッグを行う際、ブートシーケンスを理解しておく必要があるのでここでその説明をする。

ブートシーケンスの例を図 2.5 に示す。POR（Power On Reset）時、つまり電源を投入するとまず PMU が起動し、PMU ROM に格納されているプリブートタスクを実行してシステムをセットアップする。これには CSU の解放が含まれている。PMU は CSU の解放後、PMU サーバーモードに移行し電力の監視を行う。



(a) Mercury XU5 (Enclustra b) (b) Mercury+ PE1-300 (Enclustra a)

図 2.4 Mercury XU5 と Mercury+ PE1-300 のハードウェアブロックダイアグラム

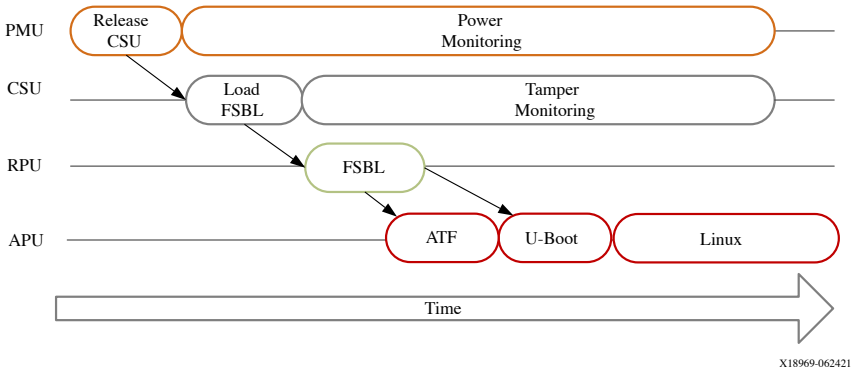


図 2.5 ブートシーケンスの例(Xilinx 2021b)

PMU によってリセットから解放された CSU は On Chip Memory（OCM）、APU、RPU を初期化し、First Stage Boot Loader（FSBL）の OCM へのロード、およびオプションの PMU Firmware（PMUFW）の PMU RAM へのロードを行う。PMUFW はユーザーがカスタマイズ可能で、電力のほかに温度管理やエラー通知などを行わせることもできる。FSBL は APU または RPU で、PMUFW は PMU で、それぞれパラレルに動作する。

FSBL は Arm Trusted Firmware（ATF）に制御を移す。ATF は Arm アーキテクチャで動作するソフトウェアのためのセキュアな機能をまとめたものである。ATF は U-Boot などの Second Stage Boot Loader に制御を移す。U-Boot はデバイスツリーの情報、カーネルイメージ、もし U-Boot に rootfs もパッケージされていればそれを RAM にロードする。rootfs が外部（SD カード第 2 パーティションなど）にある場合はそれを利用する。デバイスツリーとは、Linux カーネルが利用可能なハードウェアのコンポーネントをツリー状に記述したデータのことである。ブートイメージに FPGA ビットストリームファイルを含む場合、U-Boot はこれを RAM にロードし、RAM にロードされた FPGA ビットストリームに従って PL をプログラムする。U-Boot がブート時に読み込む環境変数と実行するスクリプトをまとめたファイルがあるならば、それを読み込み、実行する。最後に U-Boot は RAM にロードした Linux カーネルに制御を移す。Linux カーネルはデバイスツリーの情報をもとにハードウェアのデバイスドライバを呼び出す。こうして起動した Linux は対称型マルチプロセッシングで動作するため複数の

Cortex-A53 プロセッサを利用する。

2.3　Zynq MPSoC の開発手順

2.3.1　開発環境と使用するツール

一般に Zynq SoC のソフトウェアやファームウェアの開発には Windows[3] または Ubuntu などの Linux がインストールされたコンピュータを使用する。Xilinx 社は Vivado、Vitis IDE、PetaLinux といった、Zynq SoC の開発に必要なツールを提供している。

Vivado® Design Suite は、Xilinx 社製の FPGA の Hardware Description Language（HDL）によるデザインの作成から論理合成（synthesis）、配置配線（implementation）、FPGA ビットストリームの生成と書き込み、デバッグまでを行うのに用いられる。Vivado ではファームウェアを作成する際、HDL によるコーディングのほかに、GUI でダイアグラムを作ることでデザインを作成する方法があり、この場合 IP やモジュールのインスタンス化やそれら同士の接続が容易になる[4]。こうして作られるダイアグラムをブロックデザインと呼ぶ。

Zynq SoC の PS で動作するソフトウェア[5]は Vitis™ Integrated Development Environment（IDE）[6] で作成することができる。さらにそのソフトウェアが走るプラットフォーム（APU か RPU か、APU の中でも 4 つのうち何番目の Arm Cortex-A53 を使用するかなど）を選択できたり、JTAG からのソフトウェアのブートを行うこともできる。ソフトウェア開発のため、バージョン 2019.1 までは Xilinx Software Development Kit（SDK）が用いられていたが、バージョン 2019.2 以降は Xilinx SDK は Vitis Unified Software Platform に統合され、Vitis IDE と呼ぶようになっている。

Xilinx 社の Zynq SoC で動作する Linux カーネルをクロスコンパイル[7] するには PetaLinux ツール[8] を用いる。PetaLinux でも FSBL や ATF などのベアメタルソフトウェアをコンパイルできる。したがって本研究では Vitis IDE を用いる機会はあまりなかった。PetaLinux では次のようなことができる。

- カーネルモジュールの選択、作成
- U-Boot の環境変数の設定
- デバイスツリーのコンパイル
- 組み込みアプリケーションのクロスコンパイル
- Linux カーネルのコンパイル
- ブートイメージのパッケージ
- カーネルイメージのパッケージ（ユーザが操作することなく自動的に行われる）

特に Zynq SoC が搭載された Enclustra 社製のデバイスを使って開発する際、PetaLinux の代わりに Enclustra

[3] Vivado は自動的にディレクトリやファイルを生成するのだが、Windows ではパス名の最大長が 260 文字という制約があるため、不必要なエラーに悩まされることが多々あった。また PetaLinux は Linux しかサポートしていないので、ホスト OS として Windows を用いている場合、わざわざ仮想化マシン等を使って Linux をゲスト OS としてインストールするという環境構築の手間が増える。以上のことからホスト OS として Linux を選択することを勧める。

[4] FPGA 上で動作する、ある機能をひとまとまりにしたものをモジュールといい、特に Vivado で使用可能な Xilinx が提供しているモジュールを IP という。一般に IP はユーザが設定をカスタマイズできる。モジュールを実際に使うようにすることをインスタンス化やインスタンシエートといい、そうして使えるようになったものをインスタンスという。

[5] ここでソフトウェアというのは FSBL などのベアメタルソフトウェアから OS の上で走るアプリケーションまでを含む。

[6] Vitis は/váɪtɪs/のように発音される。

[7] 対象のプラットフォームで動作するソフトウェアを、異なるプラットフォームでコンパイルすることをクロスコンパイルという。

[8] 名称に Linux がついているが、これは OS ではなくあくまでクロスコンパイラであることに注意。また PetaLinux を用いてビルドした Linux のことを PetaLinux と呼んでいる人もいるため文脈による判断が必要な場合がある。研究室内ではそのような Linux は、Zynq Linux と呼び区別していた。

Build Environment（EBE）を使用する選択肢もある。EBE でも PetaLinux と同じように使用するカーネルモジュールの選択やコンパイルができるが、本研究室や日本 ATLAS グループでは PetaLinux を用いた開発経験が豊富であるため、積極的に EBE を使わなければいけない動機は今のところない。

2.3.2 ブートに必要なファイルと作成方法

節 2.2 に登場したファイルと制御の流れを図 2.6 にまとめた。節 2.2 で説明したように、リセット後は FSBL と PMUFW が起動し、制御は FSBL、ATF、U-Boot と遷移していく。U-Boot が Linux カーネルを起動して Linux がブートする。図 2.6 において点線で示したファイルはオプションである。実線の矢印はその先にあるファイルをメモリにロードして制御を移すことを表しており、点線の矢印はその先にあるファイルの情報を参照することを示している（ただし devicetree blob は U-Boot によってメモリにロードされ、Linux カーネルはそのロードされた情報を参照する）。`boot.scr` は U-Boot のための実行スクリプトであり、U-Boot はこれを参照して Linux カーネルをロードする。PetaLinux 2020.1 以降のバージョンでは、`boot.scr` を使う。また `uEnv.txt` は U-Boot の環境変数の設定が書かれたファイルであり、これはユーザが編集可能である。`uEnv.txt` を使うならば、U-Boot がそれを読み込むよう PetaLinux で適切に設定をする。図 2.6 では、これらのファイルを 2 つのファイルにパッケージングしている。ここでは PMUFW、FSBL、ATF、U-Boot、FPGA ビットストリームを `BOOT.BIN` というバイナリファイルに、Linux カーネル、および rootfs は `image.ub` にパッケージされている例を示した。このパッケージの仕方はあくまで一例である。デバイスごとに異なる `system.dtb` を用意したいが、デバイスごとにコンパイルすることが困難であるという場合などは、このように `image.ub` から `system.dtb` を分離しておくことができる。ただし `boot.scr`、`uEnv.txt` は独立なファイルであり、`BOOT.BIN` と並列に保存する。

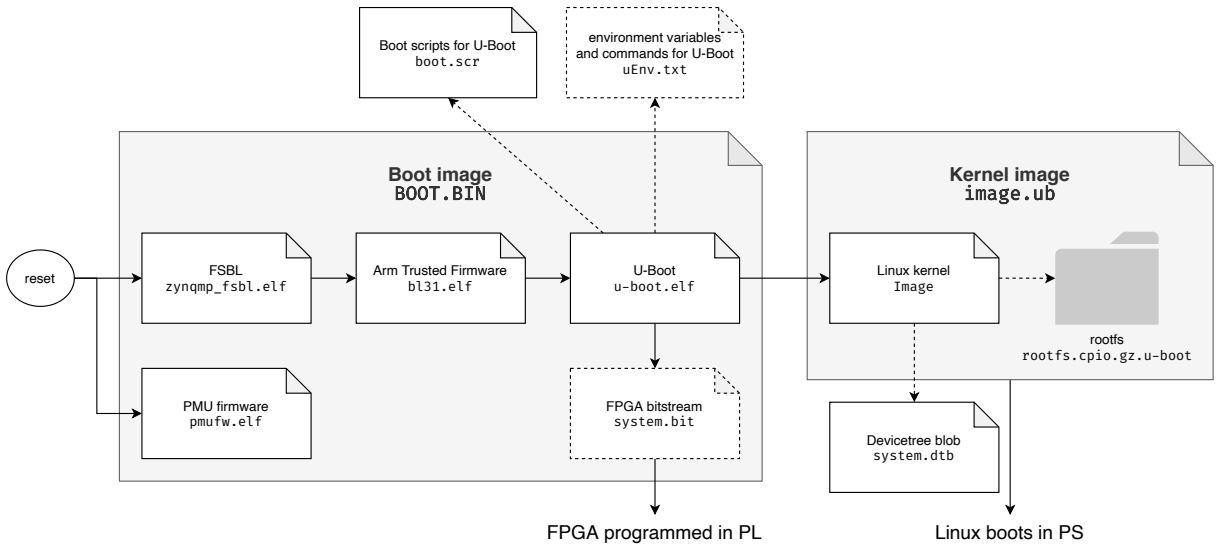


図 2.6 Zynq MPSoC のブートに必要なファイルと制御の流れの例 1。ここでは rootfs を `image.ub` に含め、devicetree blob は含んでいない。

図 2.6 では、rootfs（`rootfs.cpio.gz.u-boot`）は PetaLinux でコンパイルしたもので、`image.ub` にパッケージされている例を示した。しかし Linux カーネルが参照する rootfs のロケーションは、PetaLinux により他にも設定可能である。たとえば SD カードの第 2 パーティションに CentOS 7 の rootfs を用意し Linux カーネルがそこを参照するよう設定すれば、OS として CentOS 7 を使うことができる。そのような例を図 2.7 に示した。ここでは rootfs を `image.ub` に含めていない。また `system.dtb` は `image.ub` にパッケージされている。

章 3 で説明する Mercury XU5 と Endcap SL ボードでの開発には、図 2.7 のようなコンフィギュレーションを

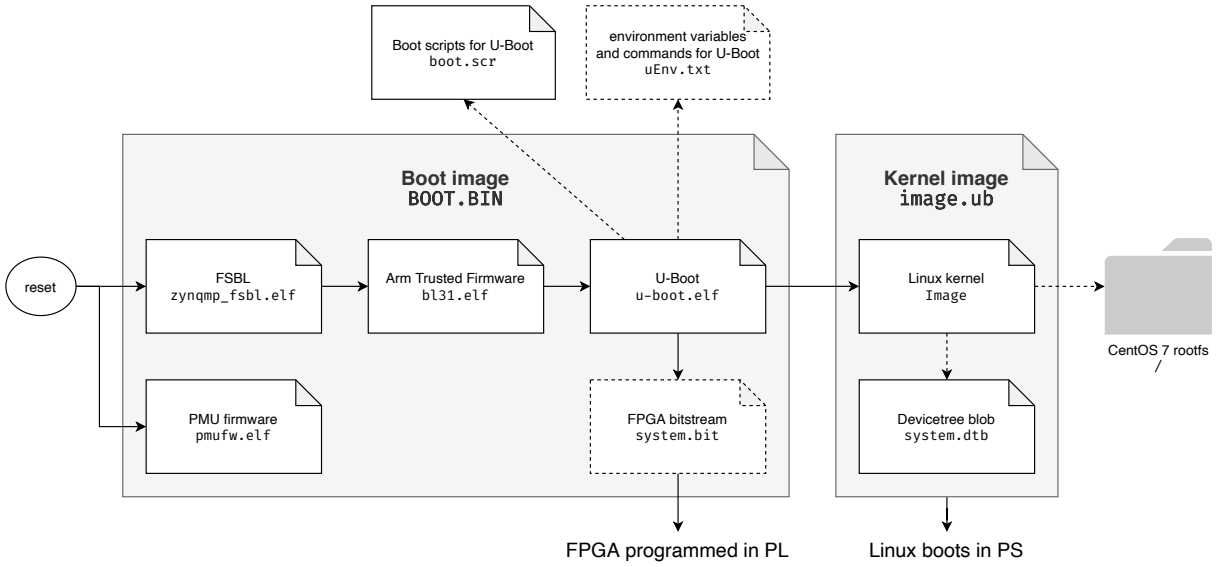


図 2.7 Zynq MPSoC のブートに必要なファイルと制御の流れの例 2。rootfs はブートイメージ、カーネルイメージとは別に用意している。また devicetree blob は `image.ub` に含めてパッケージングしている。Endcap SL ではこのコンフィギュレーションを採用した。

採用した。各コンピューティングデバイスは実験全体の方針に合わせて OS が一元管理されるべきであるということと、Linux ベースの OS を使用すべきという各サブシステムからの要請から、OS としては CentOS を採用する。さらに 48 台の Endcap SL のブートに用いるファイルは同一で良いため、できるだけコンパクトにまとめられた図 2.7 のようなコンフィギュレーションが好ましい。ただし IP アドレスといった Ethernet の設定は `uEnv.txt` で指定するため、このファイルだけは各 Endcap SL ボードで異なる。

以上のように、Zynq MPSoC のブートのために用意しなければならないファイルはいくつかある。そのようなファイルの作成に必要な材料、用いるツール、生成物を表 2.1 にまとめた。なお XSA というファイルは直接ブートには関係しないが、ハードウェアやファームウェアの情報を Vivado から開発フロー後段のツールに伝えるのに必要となる。XSA の説明は節 2.3.3 で行う。

表 2.1　ブートに用いるファイルと作成方法

Component	Input	Tools used	Output file
FPGA ビットストリーム	HDL files (`*.v`, `*.vhd`, `*.sv`, etc.), constraints files (`*.xdc`, `*.tcl`)	Vivado	`system.bit`
XSA	`system.bit`	Vivado	`*.xsa`
FSBL	fetched source code & `*.xsa`	Vitis/PetaLinux/EBE	`zynqmp_fsbl.elf`
PMUFW	fetched source code & `*.xsa`	Vitis/PetaLinux/EBE	`pmufw.elf`
ATF	fetched source code & `*.xsa`	Vitis/PetaLinux/EBE	`bl31.elf`
devicetree blob	`*.dts`, `*.dtsi`	PetaLinux/EBE	`system.dtb`
U-Boot	fetched source code & user's configuration	PetaLinux/EBE	`u-boot.elf`
Linux カーネル	fetched source code & user's configuration	PetaLinux/EBE	`Image`
rootfs	user's configuration	PetaLinux/EBE	`rootfs.cpio.tar.gz`
CentOS 7 rootfs	(can be downloaded from a mirror site)	–	`/`
Boot script file	user's configuration	PetaLinux/EBE	`boot.scr`
U-Boot env file	–	–	`uEnv.txt`
Boot image	`zynqmp_fsbl.elf`, `pmufw.elf`, `bl31.elf`, (`system.bit`)	PetaLinux/EBE	`BOOT.BIN`
Kernel image	`Image`, (`system.dtb`, `rootfs.cpio.tar.gz`)	PetaLinux/EBE	`image.ub`

2.3.3　一般的な開発の流れ

開発ツールのバージョンは Vivado 2020.1、Vitis 2020.1、PetaLinux 2020.1 を使うことを想定する。表 2.1 に挙げた各要素について、一般的な Zynq MPSoC の開発の流れについて説明する。

まず Vivado を立ち上げ、HDL のコーディングやブロックデザインで PL のためのファームウェアを作成する。FPGA 外部との物理制約を記述するために、拡張子 `.xdc` や `.tcl` の制約ファイル（constraints file）を作成する。ブロックデザインの場合は Vivado が自動的に HDL ファイルを作成している。それが完成したら論理合成、配置配線、FPGA ビットストリーム（`*.bit`）の生成を行う。この間に error や critical warning が出た場合は適宜対処する。FPGA ビットストリームの生成が完了したら、FPGA ビットストリームを含んだ Xilinx Software Archive（XSA）ファイル[9] を出力しておく。これにより開発フロー後段の Vitis IDE や PetaLinux にハードウェアや FPGA の情報を伝えられる。なおこの時点で開発用コンピュータと開発ボードを Xilinx Platform Cable USB II などで繋げば、FPGA ビットストリームを用いて PL をプログラムできる。

XSA を Vitis IDE に読み込ませ、FSBL などのベアメタルソフトウェアの作成をする。この時点で、ここで作成したベアメタルソフトウェアは JTAG などを使ってメモリにロードすれば Zynq MPSoC で実行可能になる。ただし Linux をビルドする立場では、Vitis IDE での作業は不要である。

U-Boot や Linux カーネルといった Linux に関連したソフトウェアのクロスコンパイルには PetaLinux を用いる。その場合ファームウェアは U-Boot から PL にプログラムされる。新規 PetaLinux プロジェクトを作成し、XSA を読み込ませて Linux カーネルや U-Boot の設定などを行い、`petalinux-build` というコマンドでビルドする。ビルドが終わったらブートイメージをパッケージングし、`BOOT.BIN` を作成する。`image.ub` は PetaLinux での設定にしたがって自動的にパッケージされる。

ここでは特に SD カードからのブートについて説明する。ボード上のブートモードを指定するジャンパーピンや DIP スイッチを SD カードに対応する設定にすることにより、電源投入時に SD カードにあるブートファイルを参照する。さらに PetaLinux でのコンフィギュレーション時に指定することで、ブートデバイスの優先順位は `boot.scr` に記述される。SD カードのパーティションを 2 つに区切って、第 1 パーティションに `BOOT.BIN`、`image.ub` などのイメージを置き、第 2 パーティションに rootfs を構築しておく。その SD カードをボードの SD カードスロットに挿入し、電源を投入すれば節 2.2 や節 2.3 で説明したようなフローで PL の FPGA はプログラムされ、PS では Linux が走るようになる。ボードと開発用コンピュータを USB で接続し、UART 通信でコンソールを見れれば、PS で Linux が起動する様子が確認できる。起動しなかった場合でも、ブートシーケンスのどこで問題が発生したかを確認できる。

開発を進める上でファームウェアの変更やカーネルモジュールの追加などは頻繁に行う必要があるが、その度に電源を切って SD カードを抜き挿しするのは面倒である。これを避けるためにはカーネルイメージや FPGA ビットストリームのファイルを更新する操作を、ネットワーク経由で Linux から行えばよい（東田 2019）。Linux カーネルの実態はメモリにロードされたプログラムであり、元となるファイルを変更しても、その時点で走っているプログラムやファームウェアには影響は及ばない。変更を反映させるには Linux をリブートさせればよく、これはリモートでも実行可能である。ただし Linux の起動前で問題が発生したときには、直接 SD カードに書き込み直す復旧の手続きが必要になる。

[9] Vivado 2019.2 以前のバージョンでは拡張子 `.hdf` の Hardware Description File が出力されていた。

2.4　ZCU102 評価ボードでの開発例

Xilinx 社の ZCU102 評価ボード（Rev 1.1）には Zynq UltraScale+ MPSoC（XCZU9EG-2FFVB1156E）が搭載されている。この評価ボードを用いた開発の流れを以下で説明する。

ここでは PS 上で CentOS 7 を走らせ、PL には Block RAM（BRAM）と AXI GPIO を使った単純なファームウェアを実装した。AXI GPIO の信号はボード上の 8-bit LEDs に繋がっていて、PL から操作できる。さらに PS と PL を接続すれば、CentOS 7 上で動くプログラムから LED を制御できるようになる。開発には Vivado 2020.1 と PetaLinux 2020.1 を使用した。

2.4.1　Vivado を用いたファームウェアの作成

テンプレートとして ZCU102 を選択し、新規 Vivado プロジェクトを立ち上げた。新しいブロックデザインを開いた。

たとえば BRAM と AXI GPIO の IP ブロックをブロックデザインに追加し、AXI GPIO の IP ブロックをダブルクリックして [Board interface]の設定を [dip switches 8bits]から [led 8bits]にした。ここで PS も全体のハードウェアデザインの一部であり、PS の IP ブロックをブロックデザインに組み込める。PS の IP コアをダブルクリックすれば、PS から使用するペリフェラルの設定もここでできる（図 2.8）。さらに [Address Editor]のタブから、PL 内のモジュールの物理アドレスを指定することができる（図 2.9）。これにより PL 内の機能がアドレス空間に確保され、PS で走る Linux はここで指定したアドレスにアクセスすることにより、PL 内のモジュールを操作できる。PL 外部とのインターフェイスポートを作成し、然るべく配線することで[10]、最終的に図 2.10 のようなブロックデザインが完成した。PS と PL 間のインターフェイスには節 2.1.2 で説明した AXI というプロトコルが採用されている。図 2.10 中の AXI SmartConnect IP は AXI のマルチプレクサとして用いられている。これは AXI のマスターとスレーブを最大 16 : 16 で接続でき、データ幅の異なる AXI チャンネルも変換してくれる。

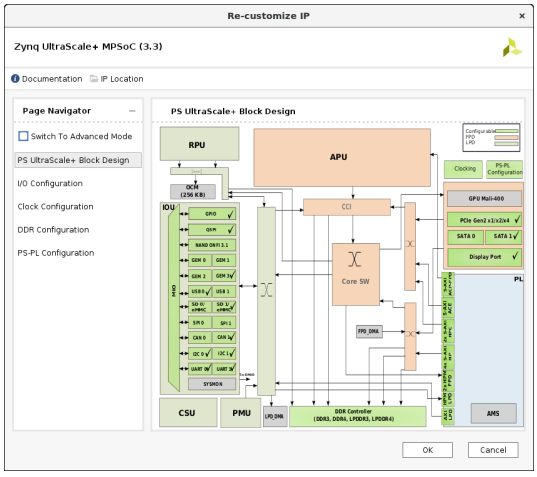


図 2.8　Zynq MPSoC PS の設定画面の例。左のタブから、どの MIO をどのペリフェラルに割り当てるかなどを設定できる。

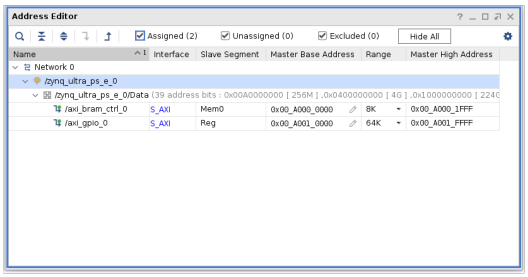


図 2.9　[Address Editor]のタブの例

ブロックデザインもまた 1 つの大きなモジュールであり、実際に使えるようにするにはこれをインスタ

[10] Vivado には auto connection の機能があり、それを使うこともできる。

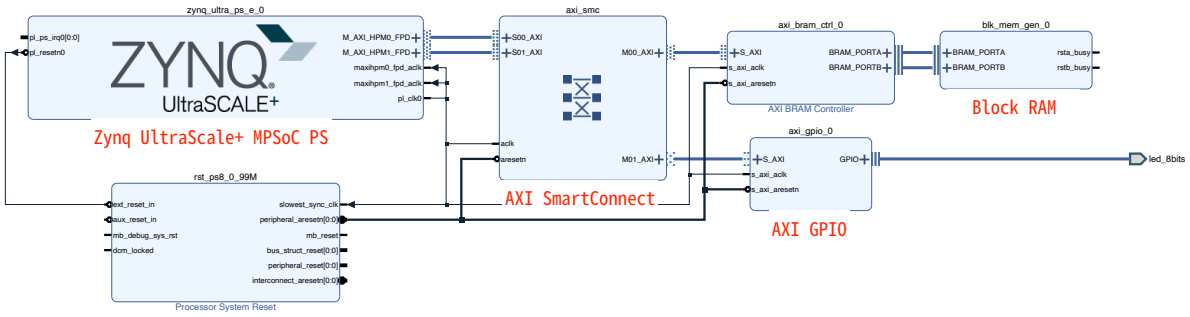


図 2.10 BRAM と AXI GPIO LED を用いたブロックデザインの例

ンシエートする必要がある。Vivado ではブロックデザインを保存して、[Sources]タブ上で右クリックをし[Create HDL wrapper]を選択すれば、そのブロックデザインインスタンシエートした HDL ファイルが得られる。この操作は「ラッパーをかける」と呼ばれており、ブロックデザインのファイル名が`design_1.bd`（HDL ファイル`design_1.v`は Vivado によって自動生成されている）ならば、ラッパーをかけたものは`design_1_wrapper.v`として得られる。`design_1.v`には Zynq MPSoC PS や BRAM などのモジュールがインスタンシエートされたモジュール`design_1`が記述されており、`design_1_wrapper.v`ではそのモジュール`design_1`がインスタンシエートされる。また LED に繋がるインターフェイスポート（図 2.10 の led_8bits）のように FPGA 外部との接続が必要ならば、FPGA Bank のピン配置などを指定した制約ファイルを作成する必要がある。

[Flow Navigator]から [Generate Bitstream]を実行すれば、論理合成、配置配線、FPGA ビットストリームの生成までを Vivado が自動的に行ってくれる。FPGA ビットストリームの生成が成功したら、[Export] → [Export Hardware]から、FPGA ビットストリームを含んだ XSA ファイルを出力する。

2.4.2 PetaLinux でのブートファイル作成

行った詳細な作業やコマンドについては付録 A でまとめることにして、ここでは PetaLinux でできることを大まかに記述する。

PetaLinux でプロジェクトを立ち上げ、Vivado で生成した XSA ファイルを読み込ませて基本的な設定を行う。これにはブートイメージやカーネルイメージなどのパッケージングの仕方、それらをどこに格納しておくかというブートモードの設定（ZCU102 では JTAG、QSPI、SD の選択肢がある）、SD カードは何番を使うか、Linux 起動時の Ethernet の IP アドレスの設定などが含まれる。

またデバイスドライバを追加したい場合など、rootfs や Linux カーネル、U-Boot の個別な設定もできる。自作の組み込みアプリケーションやカーネルモジュールのもこの時点加えることができる[11]。

デバイスツリーの編集もこの時点で行う。デバイスツリーについてはソースコードを Device Tree Source（DTS）といい、コンパイルしたものを Device Tree Blob（DTB）という[12]。またデバイスツリーのコンパイラは Device Tree Compiler（DTC）といい、PetaLinux は内部でこれを用いている。ここではあらかじめ提供されている ZCU102 のデバイスツリーを使用するよう設定した。カスタムボードを使用する場合や、PL に置いたモジュールを Userspace I/O から使えるようにする場合などはその情報をデバイスツリーに加える必要がある。それには

[11] アプリケーションについては OS 上でも開発可能で、必ずしもこの段階で加える必要はない。後述するように、Endcap SL のアプリケーションは主に CentOS 7 上で開発した。

[12] DTS の拡張子は`.dts`または`.dtsi`、DTB は`.dtb`である。

ユーザ編集用の DTS があるのでそれを編集する。

必要なコンフィギュレーションが済んだら、`petalinux-build` というコマンドですべてのクロスコンパイルを行う。完了したら `BOOT.BIN` のパッケージングを行う。`image.ub` のパッケージングはユーザのコンフィギュレーションに応じて自動的になされている。

2.4.3 SD カードからのブート

第 1 パーティションは FAT32、第 2 パーティションを EXT4 でフォーマットした SD カードを用意し、第 1 パーティションに `BOOT.BIN`、`image.ub`、`boot.scr` をコピーした。第 2 パーティションには CentOS 7 の rootfs を展開した[13]。

使用した ZCU102 評価ボードを図 2.11 に示す。SD カードスロットに SD カードを挿入し、ブートモードを指定するスイッチ SW6 を適切に変更した。またデータ転送用の microUSB ケーブルで開発用コンピュータと繋ぎ、ブート時のメッセージを UART 通信で確認できるようセットアップした[14]。SW1 がオフになっていることを確かめ、12 V 電源を ZCU102 の 6 ピン Molex コネクタに接続した。SW1 をオンにしたところ、制御が FSBL、ATF、U-Boot、そして Linux カーネルと移ってゆくのがコンソール画面で確認でき、最終的に CentOS 7 のログインメッセージまで到達した。

デバイスツリーは ZCU102 のテンプレートを使うよう PetaLinux で設定したため（コードブロック A.1）、LAN ケーブルを挿せば Ethernet は使用可能な状態であった。CentOS 7 にログインし、パッケージのアップデートや必要なパッケージのインストールを行えば、Zynq MPSoC の CentOS 7 で動作するソフトウェアの開発ができるようになった。

2.4.4 PL のロジックを PS から操作

図 2.9 にあるように、PL 内の BRAM と AXI GPIO にはそれぞれ物理アドレスを割り振られている。このメモリアドレスにソフトウェアから `/dev/mem` を開きアクセスすることで、BRAM や LED を操作することができた。詳細な説明と用いた C 言語で書いたソースコードは付録 C に残しておく。

2.5 Mercury XU5 と Mercury+ PE1 での開発パスの確立

ここでは Mercury+ PE1 ベースボードに載せた Mercury XU5 メザニンカードでソフトウェアとファームウェアを開発したことについて記す。将来的に Mercury XU5 は Endcap SL に搭載されて運用が行われるが、ここでの経験の蓄積により、Endcap SL 第 1 試作機での開発が円滑に進んだ。また本番の運用で Zynq MPSoC で動作させるソフトウェアやアプリケーションなどは Mercury XU5 と Mercury+ PE1 の組み合わせで行うことで開発が容易になる。

基本的には ZCU102 と同様に、Vivado 2020.1 と PetaLinux 2020.1 を用いた。またクロスコンパイラとして PetaLinux の代わりに EBE を用いた開発パスも実現した。

Enclustra 社は 2020 年 12 月に Mercury XU5 と Mercury+ PE1 のためのリファレンスデザイン（Koeppel et al. 2020）を公開した。これを用いることでゼロから Vivado プロジェクトを作成する必要はなくなり、Vivado を用いた開発が非常に容易になった。さらに Endcap SL の Mercury XU5 とのコネクタ周りの配線は Mercury+ PE1 を

[13] 北陸先端科学技術大学院大学のミラーサイト http://ftp.jaist.ac.jp/pub/Linux/CentOS-vault/altarch/7.4.1708/isos/aarch64/ から `CentOS-7-aarch64-rootfs-7.4.1708.tar.xz` をダウンロードし、それを第 2 パーティションに解凍した。

[14] 開発用コンピュータのホスト OS が Windows ならば Tera Term を使うことになるだろう。ホスト OS が Ubuntu などの Linux ならば、`screen` コマンドを用いて `sudo screen /dev/ttyUSB 115200` のように実行すればよい。

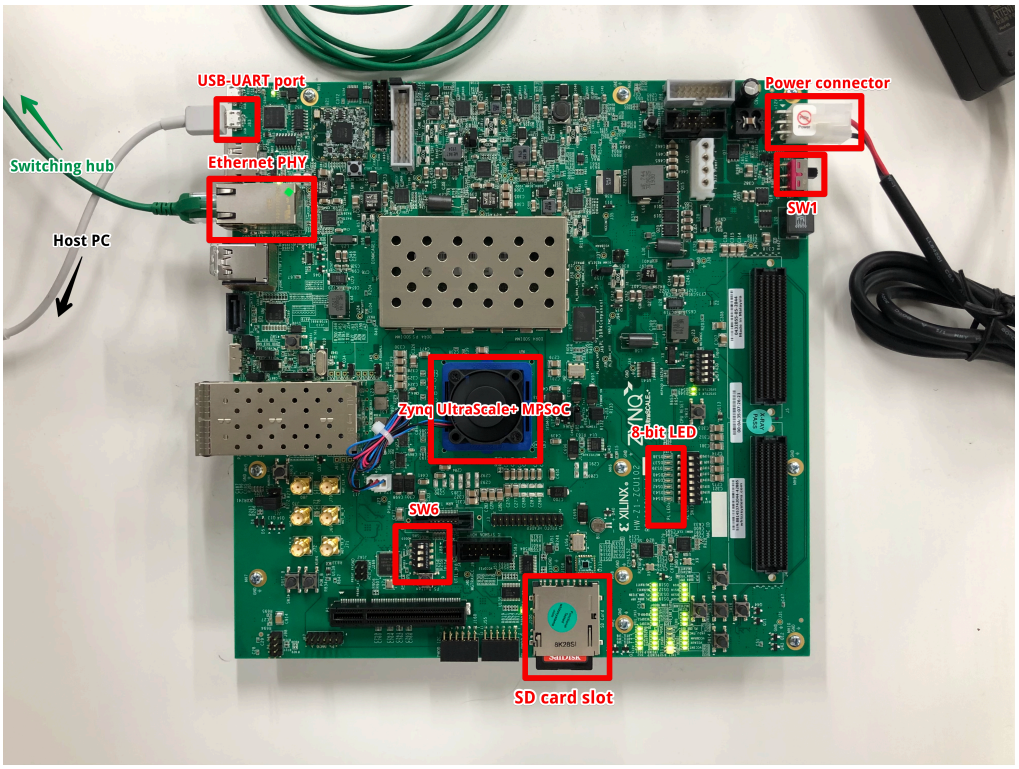


図 2.11　ZCU102 評価ボード

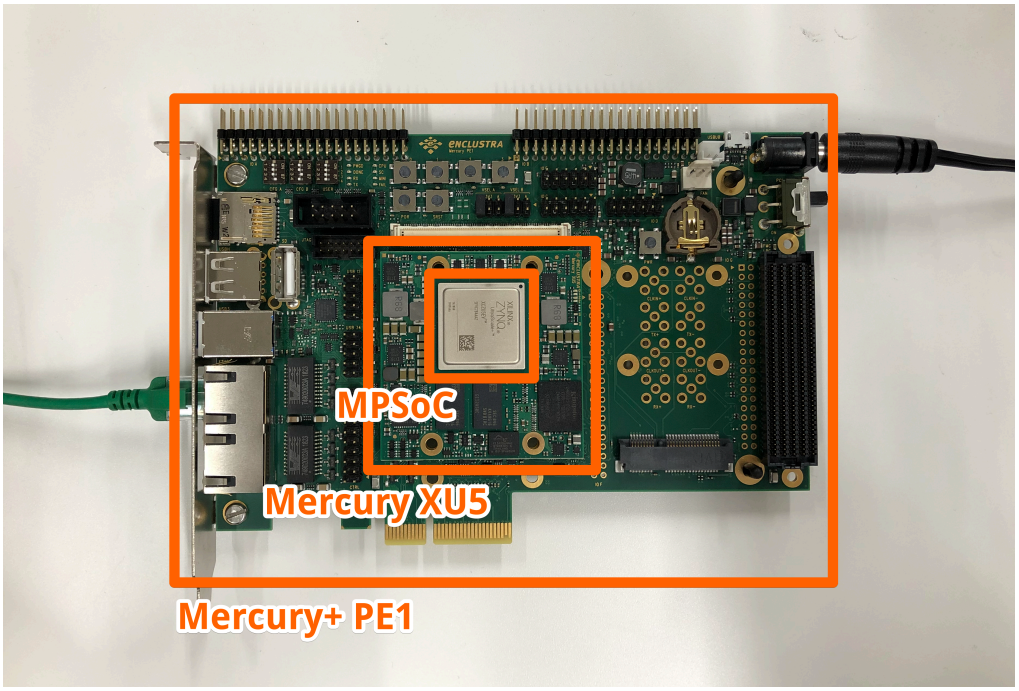


図 2.12　Mercury+ PE1 にマウントさせた Mercury XU5

参考にして設計されているため、このリファレンスデザインを起点にして開発パスを確立することで Endcap SL での Linux のブートや機能開発といった拡張を自然に行うことができた。リファレンスデザインのブロックダイアグラムを図 2.13 に示す。

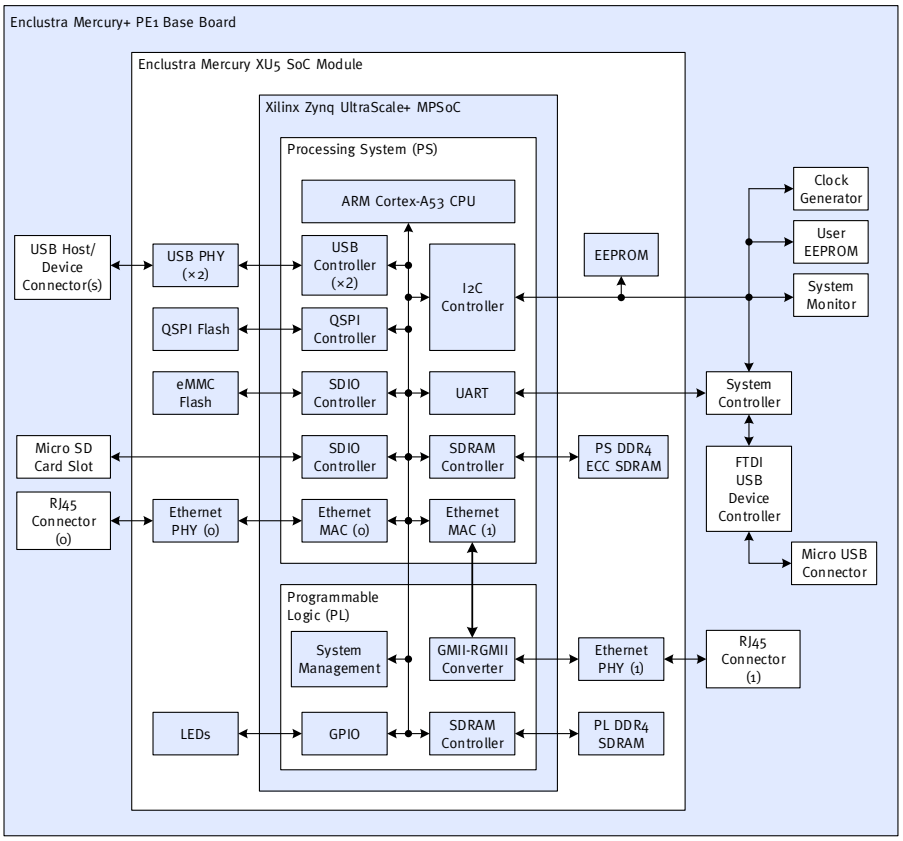


図 2.13 Mercury XU5 PE1 リファレンスデザインのブロックダイアグラム（Koeppel et al. 2020）

2.5.1 Vivado と PetaLinux を用いたパス（リファレンスデザインの利用）

新しく開拓した Vivado 2020.1 と PetaLinux 2020.1 による Mercury XU5 の開発パスを図 2.14 に示す。

HDL ファイルが共有されていれば、Vivado のプロジェクトやブロックデザインは Tcl ファイルで他人と共有が可能である。リファレンスデザインの Tcl ファイルに書かれたデバイスのプロパティを、Mercury XU5 と Mercury+ PE1 に変更した。この Tcl ファイルを Vivado の [Tcl Console]で `source` することにより、リファレンスデザインのプロジェクトを再現した。

このリファレンスデザインはそのままでも実機上で動作することを確かめた。さらにここでは自分の作成したファームウェアが実際に動作するかを確認するため、ZCU102 の場合と同様に BRAM を追加した。また提供された HDL のコードでは、Mercury XU5 上の 2 つの LED が 3 Hz と 6 Hz でブリンクするようになっていた。ここではこれを変更し、AXI GPIO で PS と接続して、ユーザが Linux から操作できるようにした。

[Generate Bitstream]を実行して論理合成、配置配線、FPGA ビットストリームの生成を行なった。こうして FPGA ビットストリームを含んだ XSA を出力し、PetaLinux の入力ファイルとした。

PetaLinux では ZCU102 と同様にコンフィギュレーションを行なった。ただしリファレンスデザインでの PS の設定では Mercury XU5 上の eMMC と SD カードをどちらも使うようになっており、Linux のデバイスファ

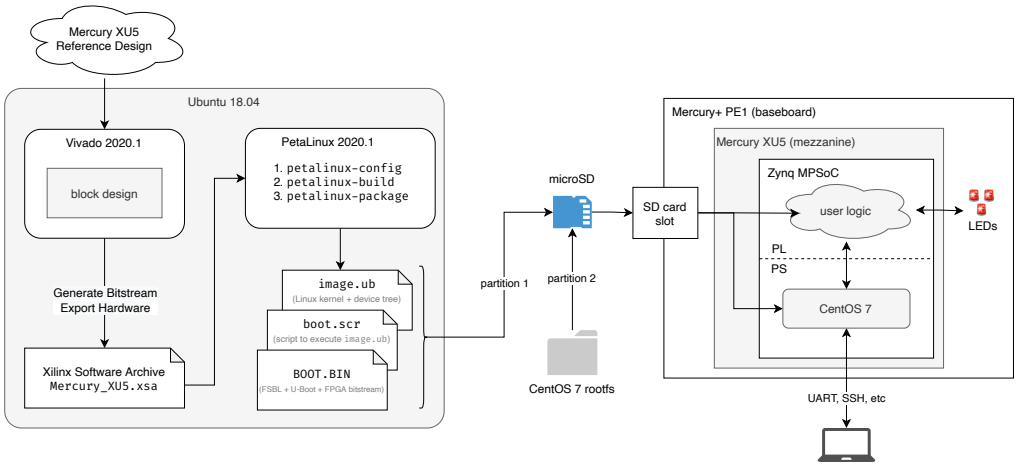


図 2.14　Vivado と PetaLinux を用いた Mercury XU5 の開発フロー

イル `/dev/mmcblk0` は eMMC、`/dev/mmcblk1` が SD カードに対応している[15]。したがってブートファイルは `/dev/mmcblk1p1` を、rootfs は `/dev/mmcblk1p2` を参照するように PetaLinux でコンフィギュレーションする必要があった。

ソフトウェアのビルドとパッケージングを行い、microSD カードの第 1 パーティションに `BOOT.BIN`、`image.ub`、`boot.scr` をコピーし、第 2 パーティションに CentOS 7 の rootfs を展開した。ブートモードを指定する DIP スイッチを適切に設定して microSD カードを Mercury+ PE1 の SD カードスロットに挿入した。UART 通信でコンソール画面を確認できるよう、microUSB ケーブルを開発用コンピュータを接続した。12 V 電源プラグを挿入し電源をオンにしたところ、正常に SD カードブートが完了した。Mercury+ PE1 の RJ45 コネクタに LAN ケーブルを接続すれば Ethernet が使え、SSH によるログインも可能となった。

2.5.2　PetaLinux の代用として EBE を用いたパス

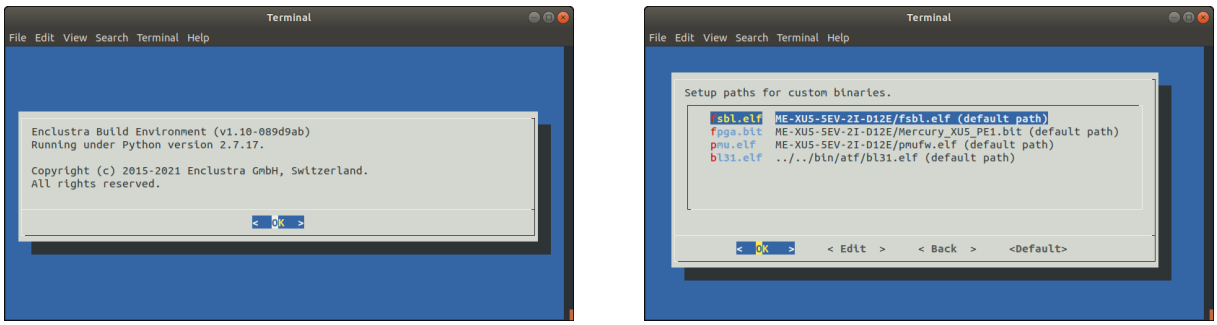
クロスコンパイラとして PetaLinux ではなく EBE (Enclustra 2019a)を用いてビルドを行うことも可能である。`git clone` したリポジトリで、`build.sh` というスクリプトを走らせることで EBE が起動する（図 2.15(a)）。

ここで用いるハードウェアやブートモード、ビルドするものを選択し進んでゆく。ブートモードは SD カードブートを選択した。途中で FSBL、FPGA ビットストリーム、PMUFW、ATF を指定する段階がある（図 2.15(b)）。ここで `(default path)` を指定すれば、リファレンスデザインであらかじめ生成されたものが使用できる。

デフォルトでは `out_<timestamp>_<module>_<board>_<bootmode>` のような名前のディレクトリに `boot.bin`、`Image`、`devicetree.dtb`、`uboot.scr` が生成される。これらを microSD カードの第 1 パーティションにコピーし、第 2 パーティションに CentOS 7 の rootfs を展開した。この方法でも Mercury XU5 で CentOS 7 を起動することができた。

図 2.15 の段階で、FSBL、FPGA ビットストリーム、PMUFW、ATF のバイナリファイルは `(default path)` を指定していた。これによりこれらの 4 ファイルはリファレンスデザインからプリジェネレートされたものを使用していた。しかし、これらを変更することでユーザが作成したファームウェアを展開することも可能であった。たとえば FPGA ビットストリームを Vivado 2019.1 で、FSBL を Xilinx SDK 2018.3 で生成したものを EBE で選択し、開発した例を図 2.16 に示す。ただし Vivado 2019.1 を用いているため XSA ではなく HDF ファイルを使って

[15] SD カードが `/dev/mmcblk1` ならば、その第 1 パーティションは `/dev/mmcblk1p1`、第 2 パーティションは `/dev/mmcblk1p2` ... のように Linux では識別される。



(a) EBE welcome screen　　(b) バイナリのカスタマイズ画面

図 2.15　EBE によるコンフィギュレーション

いる。また Vitis IDE ではなく Xilinx SDK を用いているが、Vitis IDE でも FSBL などのベアメタルソフトウェアを作成できるため、Vivado 2020.1、Vitis IDE 2020.1 を用いてもこの開発フローは成功すると考えられる。

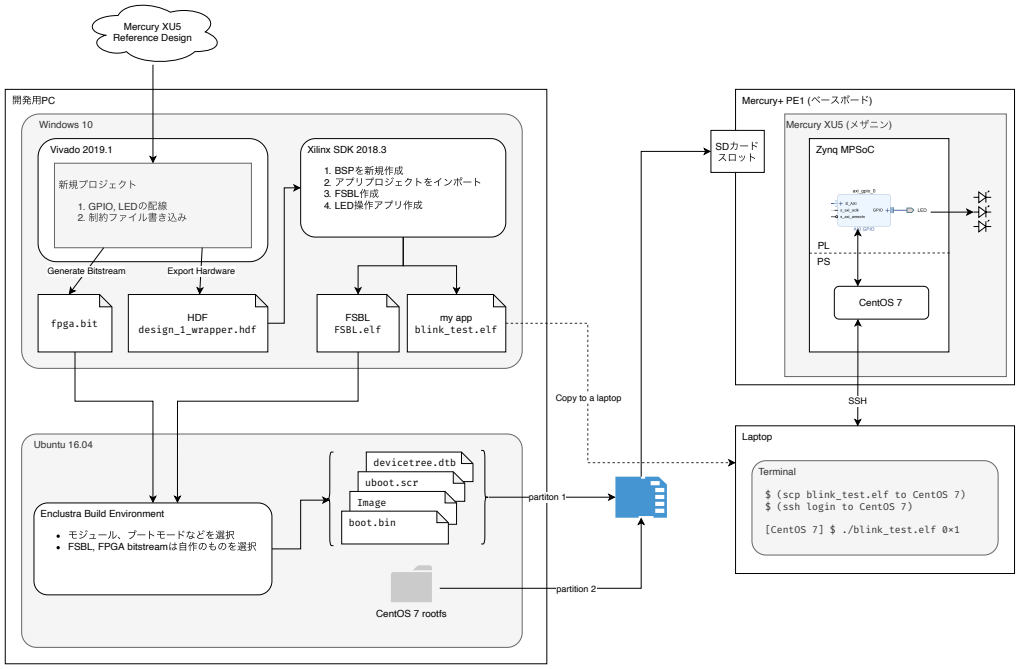


図 2.16　EBE を用いた Mercury XU5 開発パスの例

第 3 章

Endcap Sector Logic 第 1 試作機上の MPSoC の動作検証

2021 年 10 月 18 日、2 台の Endcap Sector Logic 第 1 試作機が茨城県つくば市の高エネルギー加速器研究機構（KEK）に納品された。私は KEK に赴き、Mercury XU5（Zynq MPSoC の FPGA のタイプは XCZU-5EV-FSVC784-2-I）を Endcap SL にマウントさせた際のペリフェラルの試験および Xilinx Virtual Cable をはじめとするいくつかの機能の開発を行なった。以下にその詳細を述べる。

開発環境は Vivado 2020.1 と PetaLinux 2020.1 である。詳細な手順や用いたソースコードなどは https://gitlab.cern.ch/atlas-tdaq-l0tgc/soc/l0muesl に残す予定である。Zynq MPSoC と XCVU13P に実装した最終的なファームウェアのブロックデザインは付録 G にある。

図 3.1　Endcap SL 第 1 試作機にマウントした Mercury XU5 の様子。中央に見える銀色の素子が Zynq MPSoC で、その上側に見える大きな銀色の素子が Virtex UltraScale+ XCVU13P。

3.1 準備

事前準備として、Mercury XU5 と Endcap SL の回路図を確認した。Mercury XU5 のコネクタピンを介して Zynq MPSoC と Endcap SL は接続されるが、それらの信号線の機能や対応関係と、初期試験時の注意事項などをまとめたスプレッドシートを作成した。たとえば UART の信号は、接続先を miniUSB コネクタにするか IPMC にするかをボード上のジャンパーピンで選択できるようになっており（図 F.2 のコネクタピン A105 および A107 を参照）、初期のテストではデバッグ用に開発用コンピュータと接続させるため、ジャンパーピンの 2と 3を接続すればよいことを確認した。他にも Zynq MPSoC のブートモードを指定するためのジャンパーピン（図 F.2 のコネクタピン A112 および A126 を参照）も搭載されているので、どちらもオープンにしておけばよいことを事前に確認した。初期試験に必要なソフトウェアやファームウェアといった開発項目もこの段階でまとめた。

また初期のファームウェアを作成するために Mercury XU5 のリファレンスデザイン（Koeppel et al. 2020）をコピーした。リファレンスデザインのダイアグラム（図 2.13）の Mercury+ PE1 の部分を Endcap SL に置き換えたときに変更すべき点がないかあらかじめリストアップするつもりで双方の回路図を確認したが、実際には Endcap SL 作成時に開発がスムーズに行えるよう Mercury+ PE1 の回路図を参照しその機能をできるだけ取り込んだため、リファレンスデザインのほとんどの部分をそのまま使えることがわかった。PS の設定はそのまま利用し、PL のデザインは Endcap SL 第 1 試作機との接続に合わせて修正する計画にした。

図 3.1 のように Mercury XU5 を Endcap SL にマウントした後、電源を投入すると PS のブートが完了していないことを示す LED が点灯し、これをもって Mercury XU5 に電力が供給されていることを確認した。

3.2 CentOS 7 の SD カードブート

Mercury XU5 のリファレンスデザイン（Koeppel et al. 2020）を起点に、Endcap SL で走る CentOS 7 の構築を行なった。図 2.13 にあるように、リファレンスデザインとして必要な PS のペリフェラルの設定と、PL のデザイン例は Enclustra 社から提供されている。最初の試験としては PS の設定をそのままコピーし、PL のロジックは消去して、新たに GPIO LED と BRAM を追加した。また制約ファイルもリファレンスデザインを参照し、PL で用いる I/O に応じて修正した。Vivado で XSA ファイルを出力し、PetaLinux で適切な設定をしたのちカーネルイメージとブートイメージのビルドを行なった。FAT32 でフォーマットした microSD カードの第 1 パーティションに `BOOT.BIN`、`image.ub`、`boot.scr` をコピーし、EXT4 でフォーマットした第 2 パーティションには CentOS 7 の rootfs を展開した。この microSD カードを Endcap SL の SD カードスロットに挿入した。また OS のブートモードを指定するジャンパーピンを SD カードブートモードに設定し、電源を投入した。miniUSB で開発用コンピュータと接続し、シリアル通信（UART）でコンソール画面を開いて、FSBL や U-Boot の段階のブートメッセージを確認した。

この時 U-Boot は Linux カーネルを mmcblk1（microSD カード）に見つけることができずに通り過ぎ、次に mmcblk0（メザニンカード上の eMMC）を探したが当然存在しないので見つけられず、制御を Linux カーネルに移すことができなかった。Endcap SL の回路図を確認すると、SD カードスロット（図 3.2）から出ている Card Detection（CD）の信号 SDIODETが Mercury XU5 とのコネクタピン A81 に入っていたが（図 F.2）、Mercury XU5 側では、A81 は Zynq MPSoC の FPGA Bank と接続されており、CD の MIO に対応するコネクタピンは A98 であった（図 F.1 を参照）。Endcap SL 上では A98 は何にも接続されていなかった。すなわち Endcap SL 上では、この時点では SDIODETの信号は FPGA に入っていて、CD が確立されていないことから U-Boot はこれを見て SD カードが挿さっていないと勘違いし、eMMC に Linux カーネルを探しに行きストップしたと考えられる。

この問題に対しては、Vivado での PS の設定で MIO45 を用いた CD の手続きを無効化することで対処した

(図 3.3)。すなわち U-Boot は CD 信号線のレベルによらず、SD カードが挿さっているものと解釈して、Linux カーネルを SD カードに探しに行きブートを開始する。ブートファイルを作成し直して、SD カードをアップデートしたところ、無事 CentOS 7 が起動した。

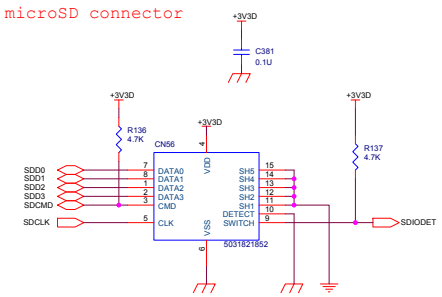


図 3.2　microSD カードスロット

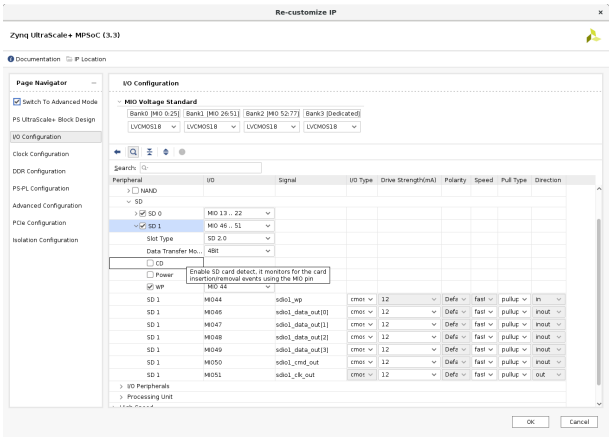


図 3.3　Vivado で CD の機能を無効化

後日、図 3.4 のように A81 と A98 のピンを導線でジャンパー接続したところ CD を有効にしたままでも SD カードブートが成功した。

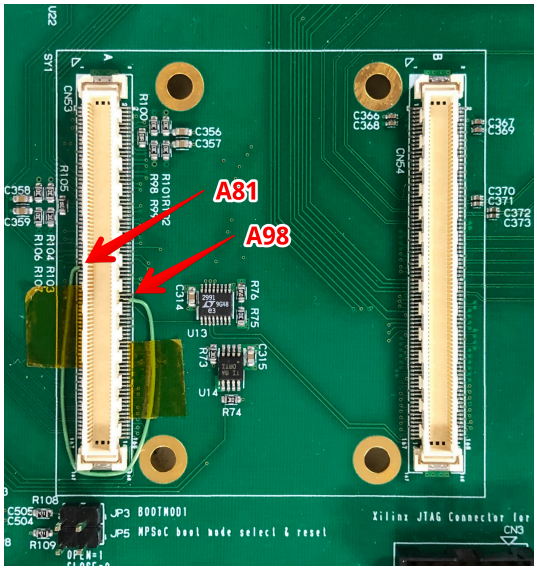
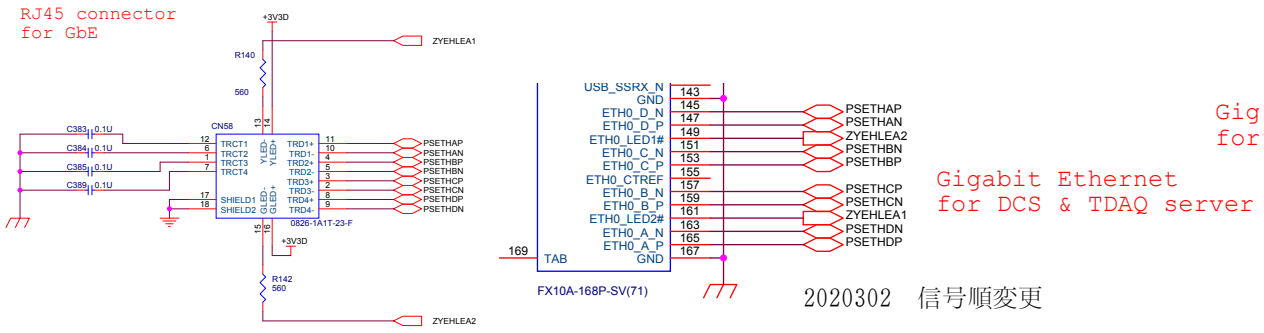


図 3.4　メザニンとのコネクタピン A81 と A98 を接続した様子

3.3　Gigabit Ethernet の開通

Mercury XU5 のリファレンスデザインを参考にして、Endcap SL 上で Gigabit Ethernet を開通させた。図 2.13 にあるように、Mercury XU5 には Ethernet PHY が 2 つ搭載されていて、1 つは PS 領域の MIO に接続されている。もう 1 つは PL 領域のユーザ I/O に接続してあり、EMIO を介して PS の Gigabit Ethernet Controller に接続されている。PL 内で GMII-to-RGMII convertor を介して PS と接続すれば使用可能である。Endcap SL にはこれ



(a) Endcap SL 上の ETH0 用 RJ45 コネクタ　　(b) Endcap SL 上の Mercury XU5 とのコネクタ A の ETH0 に関する部分

図 3.5　Endcap SL 上の ETH0 用 RJ45 コネクタと Mercury XU5 とのコネクタ A の ETH0 に関する部分。(a) RJ45 コネクタから 4 本の信号線が Mercury XU5 とのコネクタに延びている。(b) Mercury XU5 から見て A、B、C、Dのピンに D、C、B、Aの順番でスワップされている。

ら 2 つの Ethernet のインターフェイスとして RJ45 コネクタが 2 つ搭載されている。

3.3.1　スワップされた配線の問題と PS Ethernet（ETH0）の開通

図 2.13 のようなリファレンスデザインの PS の設定を用いたので、そのままでも PS Gigabit Ethernet（ETH0）は使えるはずであったが、Endcap SL 第 1 試作機で試験したところ想定通りに動作しなかった。

図 3.5(a) の RJ45 コネクタを介した LAN ケーブルからの 4 線の信号 A、B、C、D（さらにそれぞれ Pと Nの極性があるため合計で 8 本）は、図 3.5(b) または図 F.2 に示したように、Endcap SL 上では Mercury XU5 とのコネクタ A145 から A165 の奇数列に接続されている。ここでは Mercury XU5 から見て A、B、C、Dのピンに対して D、C、B、Aの順番で接続されており、これが動作不良の原因であった[1]。

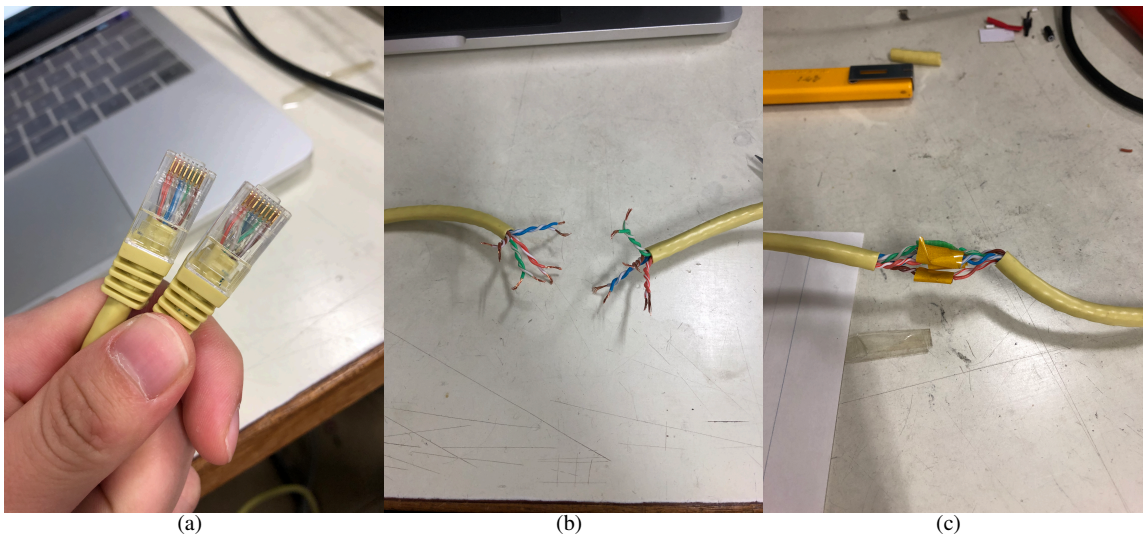
そこで応急処置的に、Aと D、Bと Cを LAN ケーブル内でスワップさせておいたものを自作した（図 3.6）。混線や断線がないことを両端のコネクタで確認したのち、Endcap SL と開発用コンピュータを接続したところ、無事に PS Gigabit Ethernet のリンクが確立し、SSH によるログインなどが可能となった。後日、正式な道具を用いてこのようにスワップさせた LAN ケーブルをいくつか作製した。Endcap SL 第 2 試作機ではこの配線を修正することにし、第 1 試作機についてはスワップさせた独自の LAN ケーブルを用いてこれに引き続く試験を行った。

KEK に申請して固定の IP アドレスを取得し、Zynq MPSoC をネットワークに接続した。IP アドレスや MAC アドレスは、UART 通信でログインし、CentOS 7 の NetworkManager Text User Interface（`nmtui` コマンド）を用いることで設定した[2]。また PetaLinux でのコンフィギュレーションの段階で、これらの設定を行うこともできる。また後述するように `uEnv.txt` を用いた IP アドレスと MAC アドレスの指定も可能で、ETH1 の開通後に `uEnv.txt` を使って設定することにした。

ETH0 が開通したことにより、パッケージのアップデートや、リモートでのアクセスもできるようになった。

[1] Mercury XU5 に搭載されている 2 つの Ethernet PHY（KSZ9031RNX、https://ww1.microchip.com/downloads/en/DeviceDoc/00002117G.pdf）には、LAN ケーブルのストレートとクロスオーバーのタイプの差を吸収するべく auto MDI/MDI-X の機能がついている。これにより A-B、C-D間のスワップは正しくマップされ直すが、このようなスワップ（A-D、B-C）には対応していなかった。

[2] NIC に関する設定ファイルは、Linux では `/etc/sysconfig/network-scripts/` 配下に `cfg-eth0` のようなファイル名で生成されている。CentOS 7 ではこれらのファイルを直接編集するよりも、`nmtui` を用いて設定することが推奨されている。



(a) (b) (c)

図 3.6 内部で信号線をスワップさせた LAN ケーブルを作製した。(a) 一般的にストレートケーブルであるかクロスオーバーケーブルであるかは、両端のコネクタを見て、中の導線の色の並びを確認すればわかる。Endcap SL に用いられている RJ45 コネクタは、コネクタの端から端までを順に A、B、C、Dと定義していたので、ここでは A-D、B-Cのスワップをすべく最両端の色同士（橙色と茶色）、その 1 つ内側の色同士（青色と緑色）をスワップさせることにした。(b) LAN ケーブル中央付近で切断し、被覆した様子。(c) 橙色と茶色、青色と緑色をハンダで接続し、触れ合わないようにビニールテープを巻いておいた。後日正式な道具を用いて、このようにスワップさせた信号線をかしめた独自の LAN ケーブルを作成した。

3.3.2 PL Ethernet（ETH1）の開通

Ethernet PHY 1 は PL 領域の EMIO と接続されており、PL 内で GMII-to-RGMII convertor を用いることにより使用できる。Xilinx 社は Vivado の IP として GMII-to-RGMII convertor を提供しているが、Mercury XU5 には使えず、Enclustra 社が提供するモジュールを使う必要があった(Koeppel et al. 2020)。 再び FPGA ビットストリームを生成し、XSA ファイルを出力、PetaLinux にてブートファイルをビルドし直し起動した。

ETH1 が開通したことを開発用コンピュータとの Peer-to-Peer 通信ができることをもって確認した。ただし ETH1 の基盤上の配線も節 3.3.1 の ETH0 と同様に問題があったため、中の配線をスワップさせた独自の LAN ケーブルを使った。しかし、`uEnv.txt` を使わないまま ETH1 が開通すると、ETH0 の MAC アドレスや IP アドレスがランダムに上書きされてしまうことがわかった。これを避けるため ETH0/1 の MAC アドレスは `uEnv.txt` に記述して設定するようにした[3]。U-Boot が `uEnv.txt` を参照するように `boot.scr` を変更する必要があるが、`boot.scr` はバイナリを含むファイルであるため、単純にテキストエディタなどで編集しただけでは機能しないことがわかっている。これを変更するには PetaLinux プロジェクトの `project-spec/meta-user/recipe-bsp/u-boot/u-boot-zynq-scr/boot.cmd.default.initrd` を編集してビルドををすれば適切な `boot.scr` が生成される。

適切な `uEnv.txt` を準備することで、ETH0 に関しては起動時に指定の IP アドレスが設定された。ETH1 に関しては MAC アドレスは `uEnv.txt` に書かれたように設定され、IP アドレスは CentOS 7 の rootfs から `nmtui` などのコマンドを使って設定した。

[3] FSBL やデバイスツリーで MAC アドレスや IP アドレスを指定しても、U-Boot はそれらを上書きしてランダムに割り振ってしまう。しかし U-Boot は `uEnv.txt` の環境変数を参照するので、MAC アドレスや IP アドレスはここに書くのが一番確実であると考えられている。

3.4 I²C 通信プロトコルによる clock jitter attenuator の設定

SL には Si5345 jitter attenuating clock multiplier（図 3.7）が搭載されており、これがボード上の Zynq MPSoC や XCVU13P のシステムクロックやリファレンスクロックの起点となっている。2027 年からの本番の運用では TTC から配布され再構成した 40 MHz の LHC クロック（図 3.7 のポート 0 の GTRECLKP/N）が主な入力となるが、オンボードの 40.079 MHz テスト用水晶発振器のクロック（ポート 1）や外部クロック（ポート 2）も利用可能である。

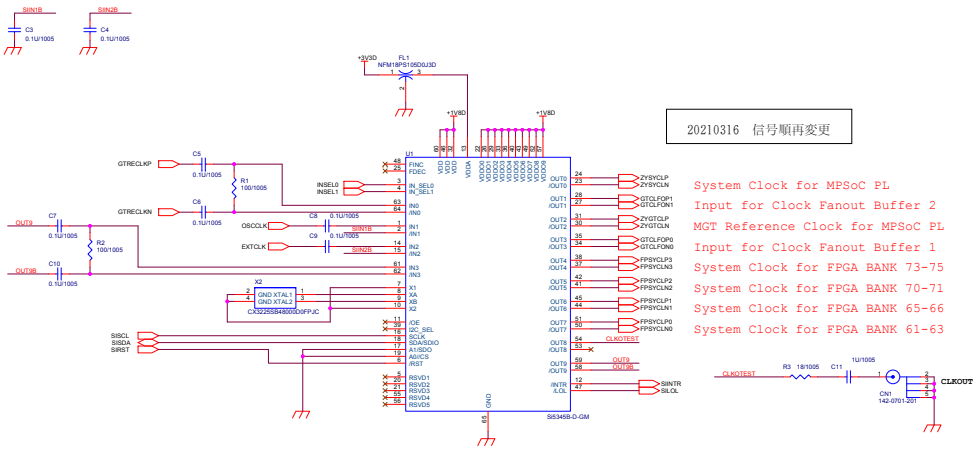


図 3.7 Endcap SL 上の Si5345 clock jitter attenuator

Zynq MPSoC PS の I²C 用 MIO は、図 3.8 の TCA9548A I²C channel multiplexer（I²C MUX）を介して Si5345 と I²C の信号線で繋がっている。この I²C MUX は他にも FireFly モジュール、SFP+ コネクタ、Zynq MPSoC と XCVU13P の温度センサ、Si5344 とも接続されている。ユーザは Zynq MPSoC PS で走るソフトウェアからこれらの I²C デバイスにアクセスし、それらのステータスの監視やコンフィギュレーションを行う。Linux からの I²C デバイスの詳細なアクセス方法は付録 D に譲り、ここでは 40.079 MHz のテスト用水晶発振器の入力クロックから XCVU13P の MGT Bank に供給する 240.474 MHz のクロックを出力ポート 1 と 3 から出力する設定について、簡単に述べる。

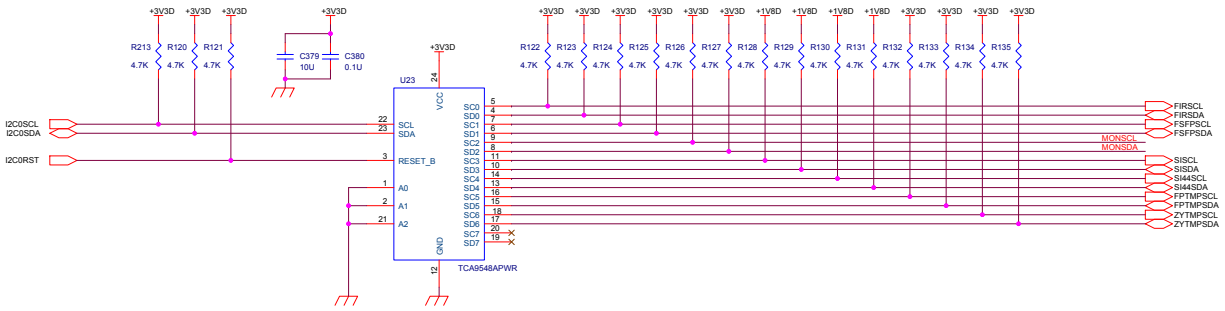


図 3.8 Endcap SL 上の TCA9548A I²C channel multiplexer

まず PetaLinux のプロジェクトで必要な I²C デバイスドライバが有効になっていることを確認した。I²C MUX についての情報を付録 D のコードブロック D.1 のようにデバイスツリーに記述して、Linux カーネルが I²C MUX

のデバイスを検知できるよう設定した。ブートファイルのコンパイルを行ったところ、CentOS 7 から Si5345 のデバイスファイルが `/dev/i2c-4` として見えていることをブートログや `i2c-tools` というパッケージを用いて確認した。`/dev/i2c-4` は外部デバイスである I^2C ポートへの入出力のためのスペシャルファイルであるが、自動的に割り当てられるスペシャルファイル名は PS の再起動などで変わらないことを確認した。

Si5345 はその内部に多数のレジスタをもっており、I^2C でレジスタの値を読み書きすることでコンフィギュレーションや監視ができるようになっている。そのレジスタ数は膨大であるため[4]、SkyWorks 社はユーザフレンドリーな Si53xx 設定用アプリケーション ClockBuilder Pro [5] を提供している。このアプリケーションを用いて入力ポート 1 の 40.079 MHz のクロックを受け、出力ポート 1 と 3 から 240.474 MHz のクロックを生成するよう設定した。さらに ClockBuilder Pro には、このような設定を実現するのにコンフィギュレーションしなければならないレジスタのアドレスと書き込むべき値が書かれたリストを、を CSV 形式やヘッダーファイルの形式（`.h`）で出力する機能がある。

Si5345 のデバイスファイルが `/dev/i2c-4` であることがわかり、レジスタのリストも手に入れたので、このヘッダーファイルをインクルードして I^2C で書き込むプログラムを C 言語で作成した[6]。これを Zynq MPSoC で走る CentOS 7 上で開発とコンパイルし実行したところ、設定通りに出力クロックが出ていることをオシロスコープのプローブを当てて確認した。

3.5 Xilinx Virtual Cable を使った FPGA のデバッグおよびプログラム

Xilinx Virtual Cable（XVC）の機能を用いることで、リモートでの Zynq MPSoC 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ようになった。

XVC は TCP/IP 通信を JTAG の信号に変換することで、ネットワーク越しに Xilinx 社のデバイスをプログラムやデバッグする機能のことである。通常、Xilinx 社の FPGA をプログラムしたりデバッグしたりするには、開発用コンピュータと対象のデバイスを専用の JTAG ケーブルや USB ケーブルで繋いで行うが、XVC を用いることでそのような手間が省け、インターネットを経由し世界中どこからでも FPGA のデバッグコアインターフェースにアクセスすることができるようになる。

XVC の機能を実装するにあたって必要な要件として次のことが挙げられる。

- Zynq SoC と対象の FPGA のテストアクセスポートが JTAG の 4 線で接続されていること
- Zynq SoC で XVC サーバが起動していること
- 開発用コンピュータと Zynq SoC がネットワーク的に接続可能であること
- その開発用コンピュータに Vivado がインストールされていること

Endcap SL 上で Zynq MPSoC と XCVU13P は JTAG の 4 線で接続されており（図 F.3 の B75、B77、B81、B83 に接続されている 4 本）、また節 3.3 で述べたように Ethernet のパスが開通したので、あとは PL 内で TCP/IP の信号を JTAG 信号に変換する準備をし、Zynq MPSoC で動作する XVC サーバアプリケーションを作成すればよかった。

同研究室内で Zynq-7000 デバイスを用いた XVC の経験があり（東田 2019; 田中 2021）、基本的な開発の流れや使用方法はこれらを参考にした。ただし先行研究では Xilinx 社の提供しているモジュールを変更したものを用いて

[4] 「ページ」が 9 あり、それぞれにレジスタが 256 個あるため単純計算で $9 \times 256 = 2304$ のレジスタがある。ただし read-only のレジスタも含まれているし、そのすべてが使用できるわけではない。ページについては付録 D.4 を参照のこと。

[5] `https://www.skyworksinc.com/en/application-pages/clockbuilder-pro-software` から入手できる。ただし Windows しかサポートしていないため注意。開発用コンピュータのホスト OS が Linux であれば Wine を使うことでエミュレートできるかもしれない。

[6] ただし Si5345 特有のアクセスの仕方があったため、その説明は付録 D.4 に書いておく。

おり、Vivado で使用できる IP は使っていなかった。さらに Zynq MPSoC と JTAG で接続された FPGA をプログラムする手法はまだあまり知られていないことからここにその知見を残しておく。

3.5.1 Zynq MPSoC PL のデバッグ

まず XVC で Zynq MPSoC が自分の PL にある FPGA を、ILA などを用いてデバッグできるような仕組みを開発した。ここでいうデバッグとは、通常 JTAG ケーブルを用いて利用するデバッグコア（ILA、VIO、IBERT など）を用いたデバッグのことを指す。本節で述べるように、Debug Bridge という IP コアをカスケード接続し、PS でのソフトウェアのコマンドをデバッグコアインターフェイスに接続し実現する。Xilinx 社は Zynq MPSoC 用の XVC のチュートリアル(anunesgu 2021) を提供しているので、基本的にはそれをフォローした。

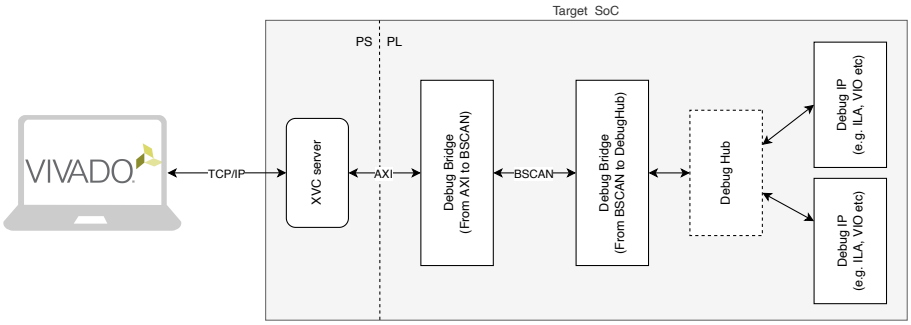


図 3.9 Zynq MPSoC の PL をデバッグする際の XVC の構成

Zynq MPSoC が自身の PL をデバッグする際の構成図を図 3.9 に示す。ブロックデザインに Debug Bridge という IP を 2 つ追加し、1 つ目は [From AXI to BSCAN]、2 つ目は [From BSCAN to DebugHub] というモードにした。1 つ目の AXI のポートは PS と繋がっている AXI SmartConnect に繋ぎ、1 つ目と 2 つ目の BSCAN のポートを接続した。1 つ目の Debug Bridge は PS から来る XVC コマンドを AXI4-Lite で受け取り、JTAG Boundary Scan（BSCAN）というインターフェイスで Debug Hub にそれを伝える。Debug Hub[7]というのは FPGA の BSCAN と Integrated Logic Analyzer（ILA）や Integrated Bit Error Ratio Test（IBERT）などのデバッグコアのインターフェイスとなる Vivado コアである。今回はテスト用に PS が出す 100 MHz の PL クロックで動作する 6-bit counter の出力を図 3.10 のように ILA で確認するようにした。ちなみに [From AXI to BSCAN]モードと [From BSCAN to DebugHub]モードの 2 つの Debug Bridge に対して、ILA などのデバッグコアは複数あってよいことが経験的にわかっている。

このブロックデザインを保存し、FPGA ビットストリームと XSA を生成した。

以降の操作の詳細は付録 E に書き、ここでは大まかな流れについて説明する。Linux が Userspace I/O（UIO）として Debug Bridge を検出できるよう、デバイスツリーにその情報を追加し、UIO を使えるようにデバイスドライバを追加した。`bootargs`（OS 起動時、U-Boot のカーネルコンソールに渡される環境変数）も編集し、ビルドを行った。CentOS 7 を起動すると、`/dev/uio0` が `debug_bridge` に割り当てられた。

XVC サーバアプリケーションのソースコードを準備し（詳細は付録 E.2）、それを Zynq MPSoC の CentOS 7 に転送してデバッグやコンパイルを行った。これを実行させた状態で、開発用コンピュータの Vivado で Hardware Manager を開き、Zynq MPSoC の IP アドレスと XVC サーバ用のポートを入力したところ、`debug_bridge_0` が

[7] Debug Hub コアはユーザ定義ではない層でインスタンシエートされ、2 つ目の [From BSCAN to DebugHub]モードの Debug Bridge と ILA はその層で Debug Hub コアを介して接続されるため、これらの接続はブロックデザインには表されない。

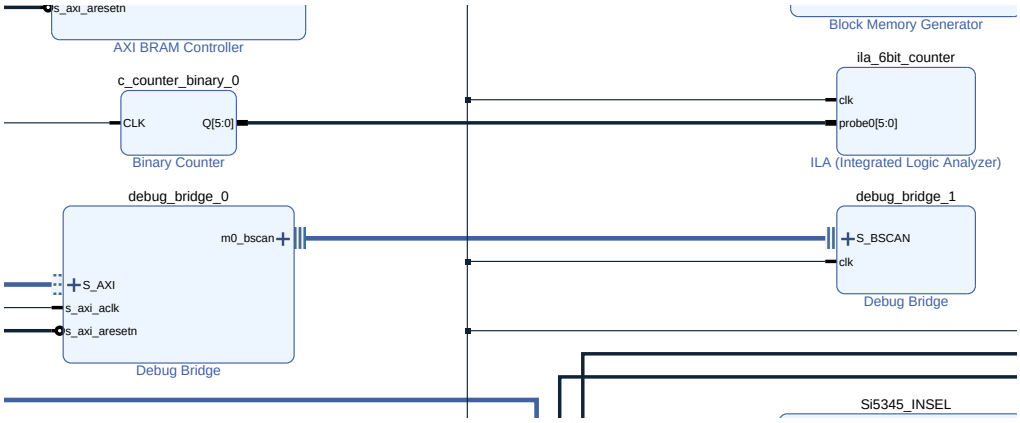


図 3.10　PL デバッグ用に置いた 2 つの Debug Bridge コア。ILA は 6-bit counter の出力をプローブするようにした。

検出された。さらに ILA のプローブファイル[8] を指定したところ、図 3.11 のように 6-bit counter の出力をリモートでプローブできるようになった。

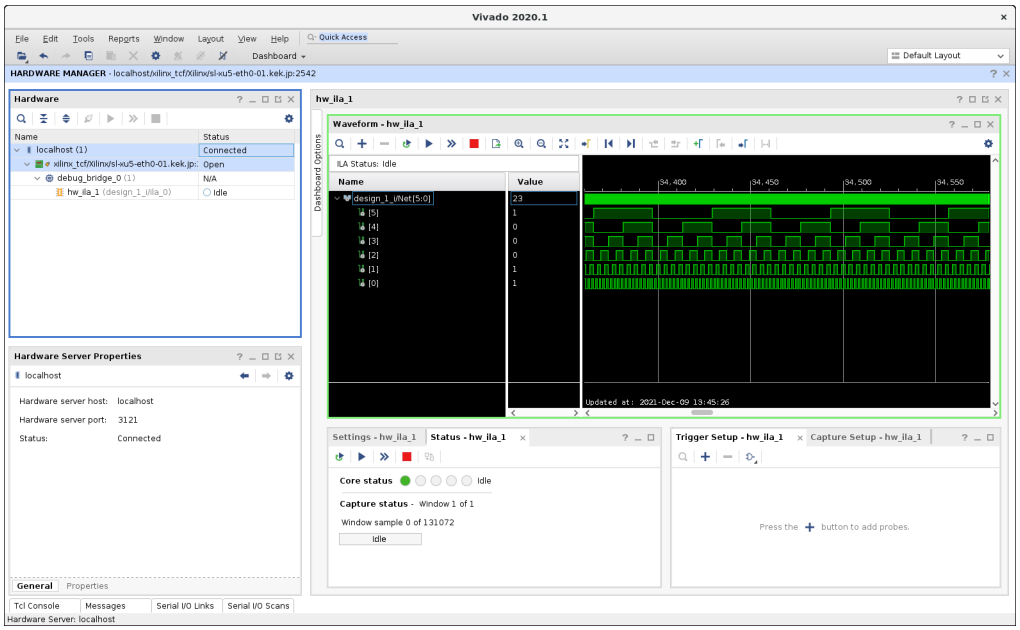


図 3.11　XVC を用いて Zynq MPSoC の PL に組み込んだ 6-bit counter をプローブしている様子

この応用例を持って、リモートによる Zynq MPSoC の PL をデバッグするデモンストレーションに成功した。

3.5.2　XCVU13P のデバッグおよびプログラム

Zynq MPSoC と JTAG で接続されたターゲット FPGA デバイスを、XVC でデバッグおよびプログラムするには節 3.5.1 と同様に Debug Bridge を用いる。今回は図 3.12 のように Debug Bridge を [From AXI to JTAG]モードで Zynq MPSoC の PL に仕込んでおく。

通常 FPGA をプログラムしたりデバッグコアにアクセスするには、現場で Xilinx Platform Cable USB II など

[8] Vivado が自動的に `<Vivado project directory>/<project>.runs/impl_1/` に拡張子 `.ltx` のプローブファイルを生成している。

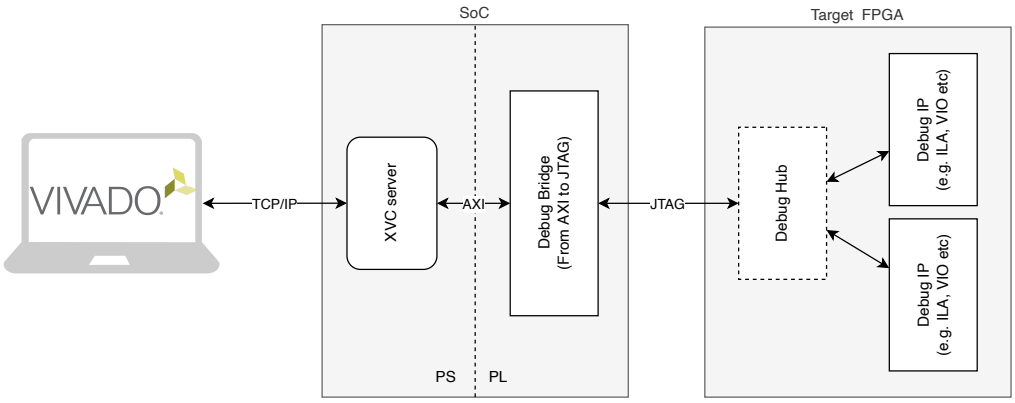


図 3.12 [From AXI to JTAG]モードでの Debug Bridge 使用例

を用いた JTAG 線経由で行う。これをネットワーク経由で行うために Zynq MPSoC で走る XVC サーバを踏み台とする。TCP/IP で受信した情報をもとに外部の 4 本の信号線を介し、JTAG プロトコルにしたがってターゲット FPGA のテストアクセスポートにアクセスする。したがって Debug Bridge のマスターインターフェイスは JTAG となり、PS と PL は AXI で接続されているので、スレーブインターフェイスは AXI となるのが合理的である。

そのように Zynq MPSoC のブロックデザインに Debug Bridge を [From AXI to JTAG]モードで追加し、Vivado で XSA を出力した。PetaLinux でブートファイルをビルドし直して CentOS 7 を起動し、`/dev/uio1` に XCVU13P 用の Debug Bridge を確認した[9]。

XVC サーバアプリケーションのソースコードの `/dev/uio0` を `/dev/uio1` に修正してコンパイルした。それを実行して開発用コンピュータの Vivado で Hardware Manager を開き、Zynq MPSoC の IP アドレスと XVC サーバ用のポートを入力すると `xcvu13p_0` が検出された。Debug Bridge の [From AXI to JTAG]モードの使用用途として FPGA のプログラムは想定していないが(Xilinx 2018b)、[Program Device]を実行すれば正常にプログラムができた（図 3.13)。

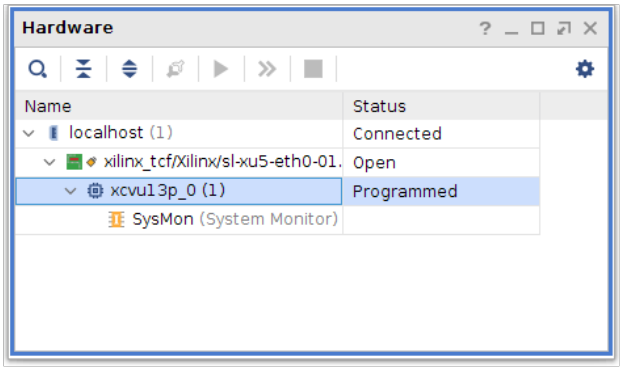


図 3.13 XVC での XCVU13P のプログラムに成功した様子

今や Endcap SL の Zynq MPSoC は、権限を持っていれば世界中どこからでもアクセスができ、Zynq MPSoC の PL や XCVU13P をデバッグしたりプログラムできる状態である。たとえば東京から KEK の Zynq MPSoC にアクセスして XVC サーバを起動し、XCVU13P の GTY のリンクをテストする IBERT の試験もできるようになっ

[9] この番号の順番は自明ではないが、経験的に [Address Editor]（図 2.9）で物理アドレスが割り振られた順番になっているように思われる。

た。その様子を図 3.14 に示す。

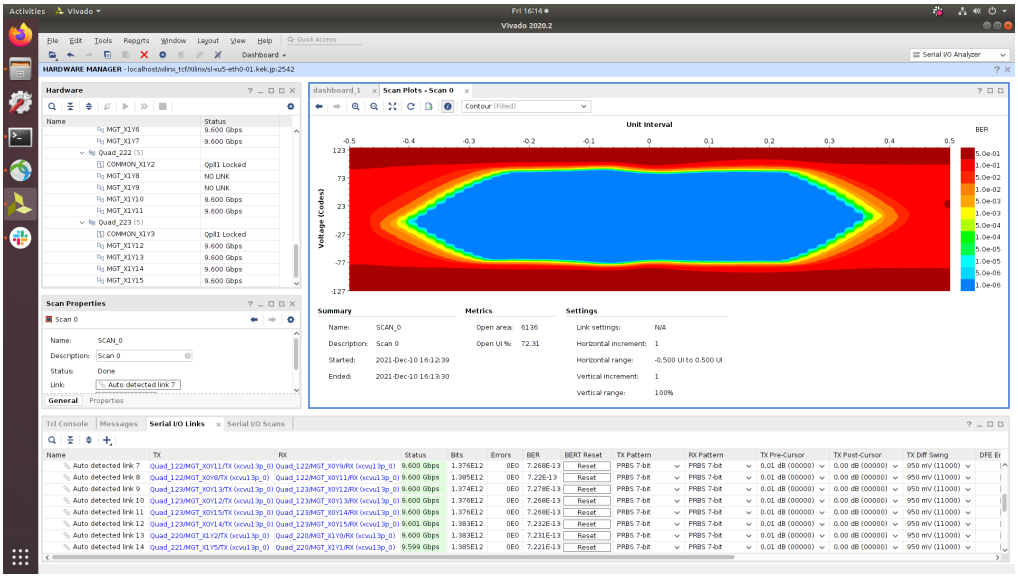


図 3.14　リモートで IBERT 試験を行っている様子

この応用例をもって、Zynq MPSoC を踏み台にしたリモートでの別の FPGA のデバッグおよびプログラムを行うことのデモンストレーションに成功した。

3.6　その他の細かな確認事項

3.6.1　JTAG からの Zynq MPSoC PL のプログラム

通常、Xilinx 社の FPGA をプログラムしたりデバッグしたりするには、Xilinx Platform Cable USB II などを開発用コンピュータと対象のデバイスに接続して行う。Endcap SL にも Zynq MPSoC 用に JTAG コネクタが搭載されており、ここに Xilinx Platform Cable USB II を接続することで、ローカルでの PL のデバッグやプログラムを行うことができる。

これを行ったところ、図 3.15 のように正常に PL がプログラムされたことを確認した。

なお、ソフトウェアに関しては Mercury XU5 は eMMC、QSPI、SD カードからのブートのみサポートしており、JTAG からのブートをサポートしていない。

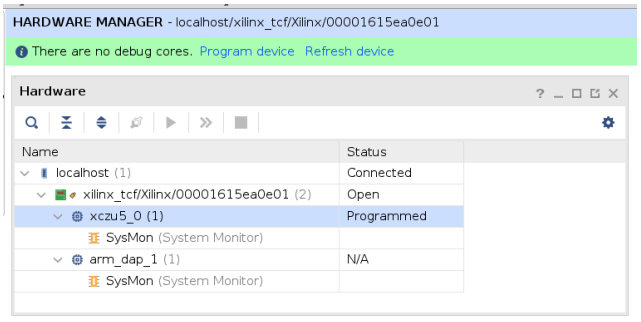


図 3.15　JTAG からローカルに Zynq MPSoC の PL をプログラムした様子

3.6.2 I^2C での温度センサの確認

TGA9548A I^2C MUX（図 3.8）のポート 5 から XCVU13P 用の温度センサ、ポート 6 から Zynq MPSoC 用の温度センサのための I^2C 線が出ている。これらは同じ温度センサ[10] で、レジスタ `0x00` と `0x15` の値を I^2C で読むことで、local temperature がわかるようになっている。

節 D.3 と同様にこれらを I^2C-read するプログラムを作成し、実行したところ、正しく温度が測れているように見えた。その結果は Vivado の Hardware Manager のシステムモニタで測った温度とコンシステントであった。

3.6.3 Zynq MPSoC からの Si5345 入力クロックの指定

コネクタピン B103 と B107 から出ているそれぞれ INSEL0と INSEL1という信号線（図 F.3 を参照）が Si5345 の IN_SEL[1:0]のポートにつながっており（図 3.7 を参照）、これにより Zynq MPSoC から Si5345 の入力クロックを選択できるようになっている。SL では入力ポート 0 が TTC 信号をソースとする 40 MHz、入力ポート 1 がボード上の発信機をソースとする 40 MHz、入力ポート 2 が LEMO コネクタからの外部入力の、最大 3 つの入力ソースの選択が可能である。ClockBuilder Pro でのコンフィギュレーションで入力クロックが単一の場合、これらのポートは使用しないように設定されるが、複数の場合はこれらのポートから入力クロックを指定する必要がある。

今回は 2 bit 幅の出力を持つ GPIO を用いて、PS からこれを操作できるようにした。Si5345 は、IN_SEL[1:0]で指定された入力クロックが有効でない場合、元の出力を保持し続ける holdover という機能が備わっている。そのため現在の試験のセットアップでは PS から IN_SEL[1:0]を変えても出力周波数が変更するのを確認できなかった。しかし Si5345 の IN_SEL[1:0]に対応するピンに直接プローブを当て、確かにソフトウェアから指定した通りに電圧が変わっていることを確認した。

[10] TMP431、https://www.ti.com/lit/ds/symlink/tmp432.pdf

第 4 章

AXI Chip2Chip を用いた FPGA 間シリアル通信用ファームウェアの開発

4.1　目指す完成形

HL-LHC での Zynq MPSoC の役割の 1 つとして、XCVU13P 内のレジスタを読み書きすることにより、回路の動作モードを設定したりリアルタイムで回路の状態を確認するというものがある。Zynq MPSoC（XCZU-5EV-FSVC784-2-I）には GTH トランシーバ（最大レーンレート 16.375 Gbps）が、XCVU13P には GTY トランシーバ（最大レーンレート 32.75 Gbps[1]）が搭載されており、これらの Multi Gigabit Transceiver（MGT）を使った高速シリアル通信によって、Zynq MPSoC から XCVU13P のレジスタ制御を実現する。

レジスタ制御機構の完成模式図を図 4.1 に示す。図中の AR、AW、W、R、Bの矢印は AXI4 のチャンネルを、Tの矢印は AXI4-Stream を表している。ただし AXI4 では信号線を機能ごとにまとめたものをチャンネルと呼ぶ。AXI4 プロトコルについては付録 H を参照のこと。Zynq MPSoC の PS で走る CentOS 7 は PL に置かれた AXI Chip2Chip master と AXI4 でやりとりができる。節 4.2.2 で説明する Aurora 64B/66B を使うことで AXI4-Stream 形式のデータを高速シリアル通信の形式にエンコードし、XCVU13P にシリアル線を用いて情報を伝える。XCVU13P 側では情報を Aurora 64B/66B でデコードし AXI Chip2Chip slave が AXI4 に復元する。それぞれのレジスタに命令を振り分けるには register interface を用いる。

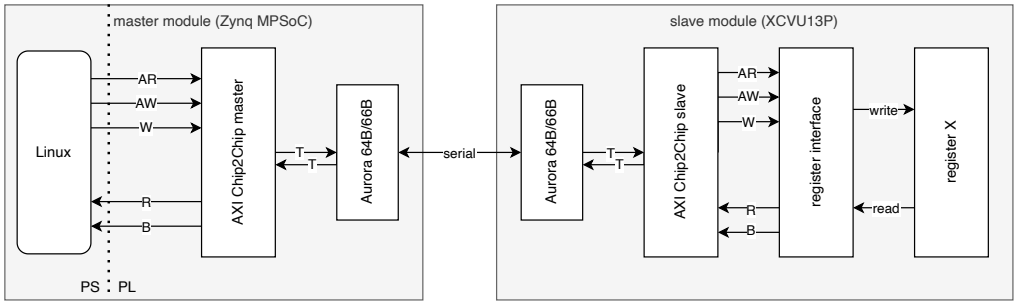


図 4.1　レジスタ制御の完成模式図

このような仕組みにより、Linux で走っているアプリケーションから XCVU13P のレジスタを制御できるようになる。

[1] ただしこれはスピードグレードの選択による。選択可能なスピードグレードの中で最大のレーンレートが示した 32.75 Gbps である。

4.2 AXI Chip2Chip

4.2.1 AXI Chip2Chip を使うことの利点

PS と PL 間のインターフェイス、また Xilinx 社の FPGA 内のバスとして AXI4 が用いられていることは節 2.1.2 で説明した。1 つの Zynq のプロセッサをマスターとし、複数の FPGA デバイスをスレーブとして通信を行うのに AXI4 プロトコルを用いるのは自然な発想である。しかし付録 H にあるように AXI4 には多ビットの信号線が多くあり、そのすべてを Zynq の I/O ピンに割り当てるのは現実的ではない。

そのような複数 FPGA 間の、シームレスな AXI4 ブリッジング機能を提供するのが AXI Chip2Chip[2]である（Xilinx 2020）。これにより少ないピン数で FPGA 間の AXI4 接続ができるようになり、マスターデバイスはスレーブデバイス内の回路をあたかも自分のものであるように操作できるようになる。仮想的に AXI4 をデバイス間で用いることで、AXI4 のバースト転送やアウトオブオーダー転送も行うことができる。

さらに AXI Chip2Chip は FPGA 間のインターフェイスとして SelectIO™ と Aurora の 2 種類から選択可能である。ここではシリアル線を用いて FPGA 間通信を行うため、Aurora インターフェイスを採用する。

4.2.2 AXI Chip2Chip と Aurora コア

AXI Chip2Chip にはマスターとスレーブのモードがあり、2 つのデバイスのうち片方をマスター、もう片方をスレーブとして使用する。

AXI Chip2Chip の PHY タイプは次の 4 種類がある。

- SelectIO SDR（single data rate）
- SelectIO DDR（double data rate）
- Aurora 8B/10B
- Aurora 64B/66B

マスターとスレーブデバイスで同じ PHY タイプを使用しなければならない。

Aurora 8B/10B および Aurora 64B/66B は Gigabit Transceiver（GT）を用いて高速シリアル通信を行うための IP コアである。PHY タイプとして Aurora を選択した場合、AXI Chip2Chip の IP だけでなく Aurora IP もインスタンシエートする必要がある。AXI Chip2Chip と Aurora 間のインターフェイスには AXI4-Stream が使われる。XCVU13P の GTY モジュールは Aurora 8B/10B のインスタンシエートをサポートしないため、今回 Zynq MPSoC と XCVU13P の接続には Aurora 64B/66B を用いた。

AXI Chip2Chip と Aurora コアのクロックとリセットの接続を図 4.2 に示す。左のブロックがマスターデバイス、右のブロックがスレーブデバイスを表しており、AXI Chip2Chip と Aurora の IP とこれらの動作に必要なクロックとリセット信号が描かれている。Aurora 内の GT に用いる GT リファレンスクロックは外部から供給する。AXI Chip2Chip IP に aurora_pma_init_inを供給することにより aurora_pma_init_outが発生し、これが Aurora IP の pma_init入力となる。また図 4.2 には描かれていないが AXI Chip2Chip IP の aurora_reset_pb信号を Aurora の reset_pbに入力する。init_clkの 6 周期以上に相当する時間 aurora_pma_init_inをアサートし、ディアサートすることで、Aurora IP のリセットシーケンスに沿ったタイミングで pma_initと reset_pbを発行する。後述する AXI Chip2Chip IP のサンプルデザインでは init_clkの 16 周期分だけ aurora_pma_init_inをアサートしていた。また lane_up[m-1:0]は各レーン（トランシーバ）が正常に初期化されたことを表し、channel_upはすべ

[2] C2C と略記されることが多い。

てのレーン（チャンネル）の初期化が完了し、データの送受信の準備が整っていることを示すポートである。他にも mmcm_not_lockedや link_reset_outなどのデバッグ用のポートがあるが、詳細は Xilinx (2018a)や Xilinx (2020)を参照されたい。

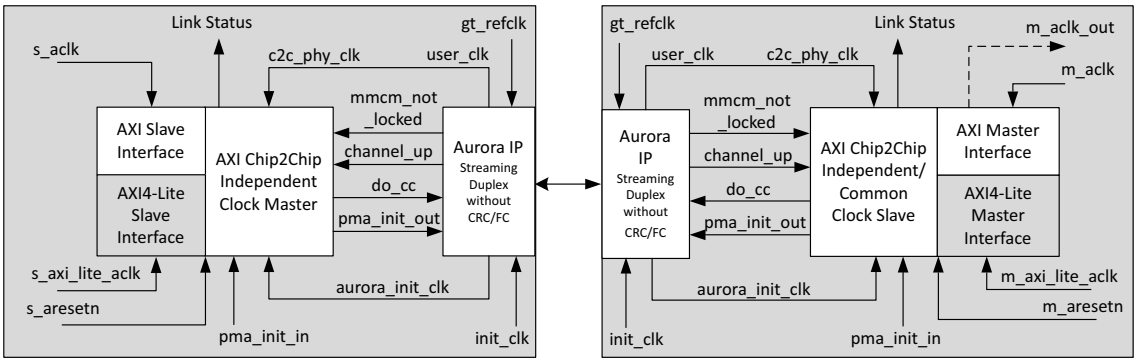


図 4.2 AXI Chip2Chip と Aurora コアのクロックとリセットの接続(Xilinx 2020)

4.3 Endcap SL 上でのファームウェアの作成

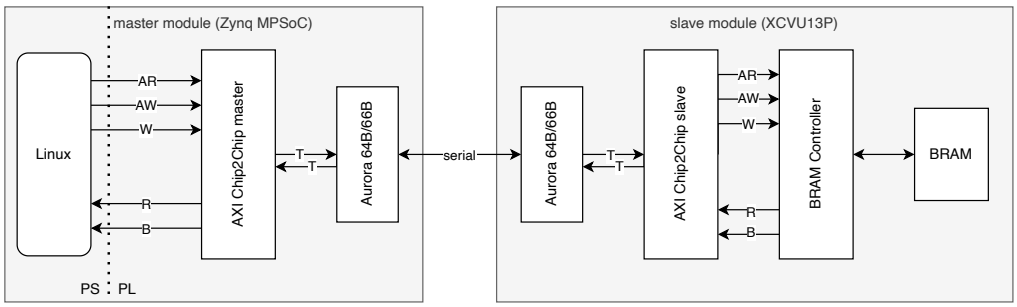


図 4.3 AXI Chip2Chip を用いた FPGA 間通信のデモンストレーションのため構築したファームウェアのダイアグラム

BRAM Controller と BRAM は Vivado の IP として使用可能であり機能することがわかっているため、ここではまず AXI Chip2Chip を用いた FPGA 間通信のデモンストレーションのため図 4.3 のように AXI Chip2Chip slave のスレーブとして BRAM を置いておいた。この XCVU13P に置いた BRAM に Zynq MPSoC の CPU からアクセスできることをもって AXI Chip2Chip を用いた FPGA 間通信ができることを確認した。

Endcap SL 第 1 試作機上では、AXI Chip2Chip を用いた高速シリアル通信のための信号線が、0と 1の 2 組設けられている（それぞれに TX と RX、さらに P と N の極性があるため合計で 8 本ある）[3]。0の方のシリアル線 4 本を用いて、Zynq MPSoC と XCVU13P 間のシリアル通信ができるか試したところ、リンクの確立とデータの転送ができることを確認した。ここでは Zynq MPSoC をマスターとし、XCVU13P をスレーブとする AXI Chip2Chip を用いた通信をデモンストレーションをするためのファームウェアの開発について述べる。

XCVU13P 側のファームウェアのブロックデザインは図 G.1 にある。XCVU13P 側では AXI Chip2Chip slave と Aurora 64B/66B をインスタンシエートし、両者の対応するポートを接続した。AXI Chip2Chip slave の先には BRAM を置いておき、これを Zynq MPSoC の PS から操作することを試みた。また Aurora 64B/66B と AXI

[3] 図 F.3 のメザニンとのコネクタ列 B の 45、47、51、53 から出ている C2CRXがついた信号線が XCVU13P から見て受信の信号。48、50、54、56 から出ている C2CTXがついた信号線が XCVU13P から見た送信の信号。

Chip2Chip slave にはデバッグ用のポートがあり、Concat IP を使ってそれらをまとめ、ILA でプローブできるようにしておいた。

Zynq MPSoC 側でも XCVU13P と同じように、AXI Chip2Chip master と Aurora 64B/66B をインスタンシエートし、デバッグ用のポートは ILA でプローブするように設定した。Zynq MPSoC 側のブロックデザインは図 G.2 にある。AXI Chip2Chip master の AXI4 の入力は、AXI SmartConnect を介して PS と接続した。すなわち AXI Chip2Chip master は AXI GPIO や BRAM と同列に PS と接続されており（図 4.4）、また [Address Editor]で物理アドレスを割り当てることができた（図 4.5）。これにより PS から AXI GPIO や BRAM をするのと同様に、付録 C のようなプログラムでソフトウェアから AXI Chip2Chip master に情報を送り込むという使い方を想定した。AXI Chip2Chip master には `0x80030000` から 64 kB のメモリが割り当てた。これに対応して、XCVU13P 側の BRAM は幅 32 bit、深さ 64 kB と設定した。

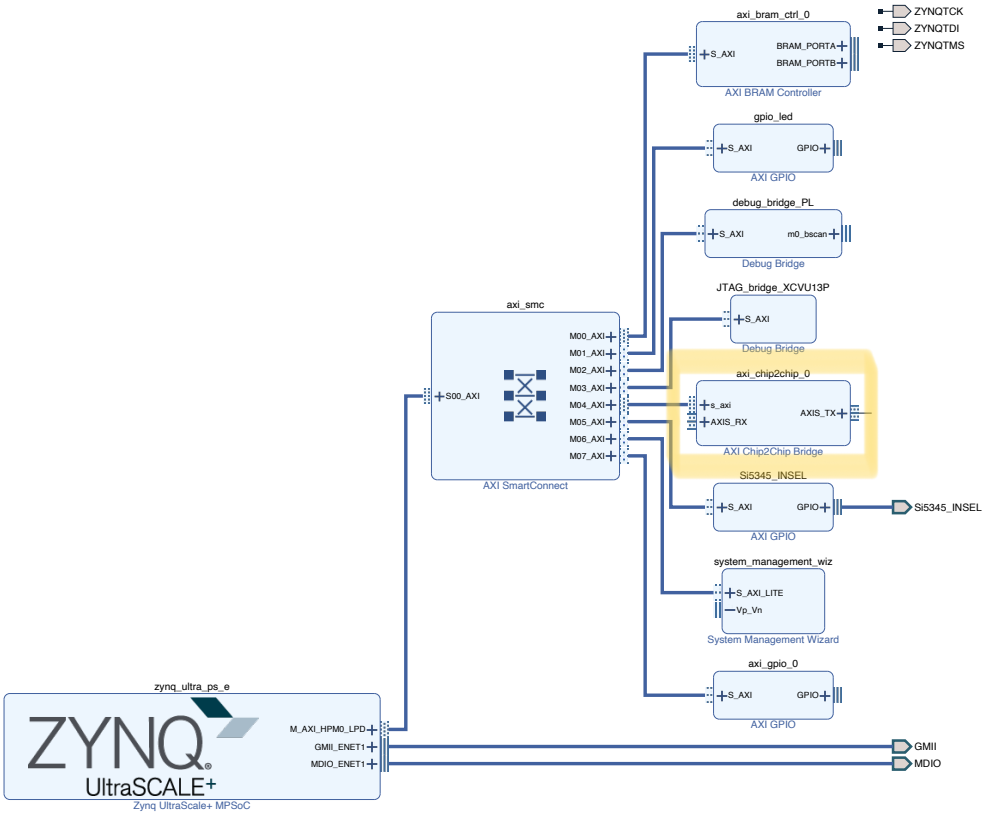


図 4.4 AXI Chip2Chip master は AXI SmartConnect を介して PS と接続できる

図 4.6 に使用したクロックと信号線、制約ファイルに書いた FPGA ピンを示した。マスター/スレーブともに、MGT Bank に供給する GT リファレンスクロックは 125 MHz の差動クロックを用いた。本番の運用では、Si5344 jitter attanuator のポート 3 を介した 125 MHz クロック発信器からのものをリファレンスクロックとして用いるが、ここでは節 3.4 で使用経験のある Si5345 を用いた。ClockBuilder Pro で Si5345 のポート 2 と 3（それぞれ Zynq MPSoC と XCVU13P の GT リファレンスクロックに対応）からボードの発振器をソースとして 125 MHz を出すようコンフィギュレーションし、節 3.4 と同様に PS で走るソフトウェアから I^2C で Si5345 を操作した。図 4.6 にあるように、ポート 3 から出る差動クロックは Si53302 clock fanout の素子に入る。そのポート 8 から出る差動クロックを XCVU13P の GT リファレンスクロックとして用い、ポート 0 から出るものを FPGA 内のシステムクロックとして用いた。FPGA 内ではシステムクロックから Clocking Wizard を用いて 100 MHz に分周した。さら

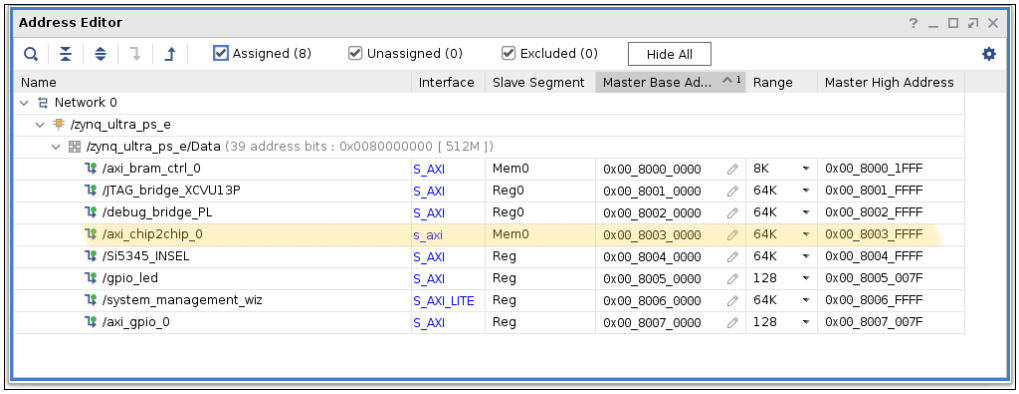


図 4.5 [Address Editor]では AXI Chip2Chip master に物理アドレスが割り当てられていた

にこの 100 MHz で動作する、パルス信号を生成するモジュール（図 G.1 のブロックデザイン中の`heartbeat`モジュール）を作成し、この信号を Endcap SL 上のデバッグ用 LED に接続した。この仕組みにより、fanout の素子から出る 125 MHz が有効であることを LED の点灯で確認できるようにした。

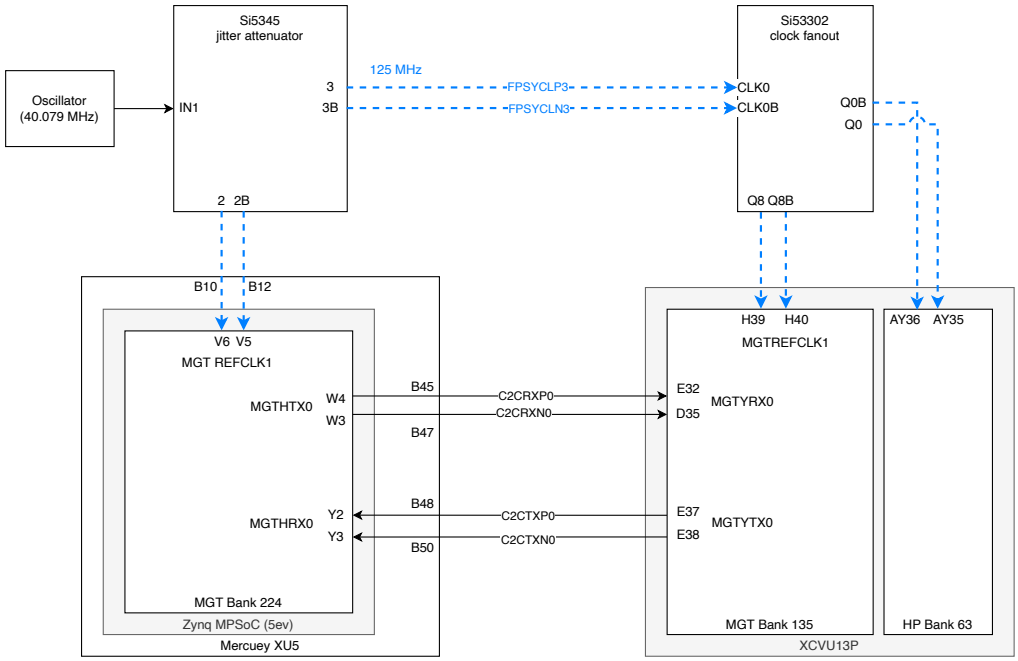


図 4.6 Endcap SL 第 1 試作機で AXI Chip2Chip の試験をする際使用したクロックとピン

Aurora 64B/66B のリセットには、init_clkの 16 周期だけ aurora_pma_init_inをアサートするモジュール（`pma_init_generator`）を作成し、これを PS から AXI GPIO を介して操作できるようにした。

4.4 Endcap SL での高速シリアル通信試験の結果

まず Zynq MPSoC の CentOS 7 から Si5345 で 125 MHz を生成するプログラムを走らせた。次に XVC サーバを起動し、XCVU13P をリモートでプログラムしたところ、デバッグ用 LED が点灯した。これにより XCVU13P が正常にプログラムされたことと Si53302 fanout から正常に 125 MHz の差動クロックが出力されていることを確認した。また Zynq MPSoC の PL をプログラムするために、OS をリブートした。なお OS をリブートしても電力が供

給されている限り、Si5345 は出力を維持する。以上により、2 つのファームウェアがプログラムされ、XVC で Zynq MPSoC と XCVU13P に仕込んだ ILA が確認できる状態になった。ソフトウェアから`pma_init_generator`に命令を送ることで Zynq MPSoC と XCVU13P 間のリンクをリセットした。デバッグ用のポートを ILA で確認すると、Zynq MPSoC と XCVU13P の両方で lane_up[0:0]や channel_upがアサートされたことなどから、シリアルリンクが確立されたことを確かめた。XCVU13P のデバッグポートを ILA で確認した際の様子を図 4.7 に示す。またその結果をまとめた表 4.1 に、Zynq MPSoC 側でのプローブ結果を表 4.2 に示す。

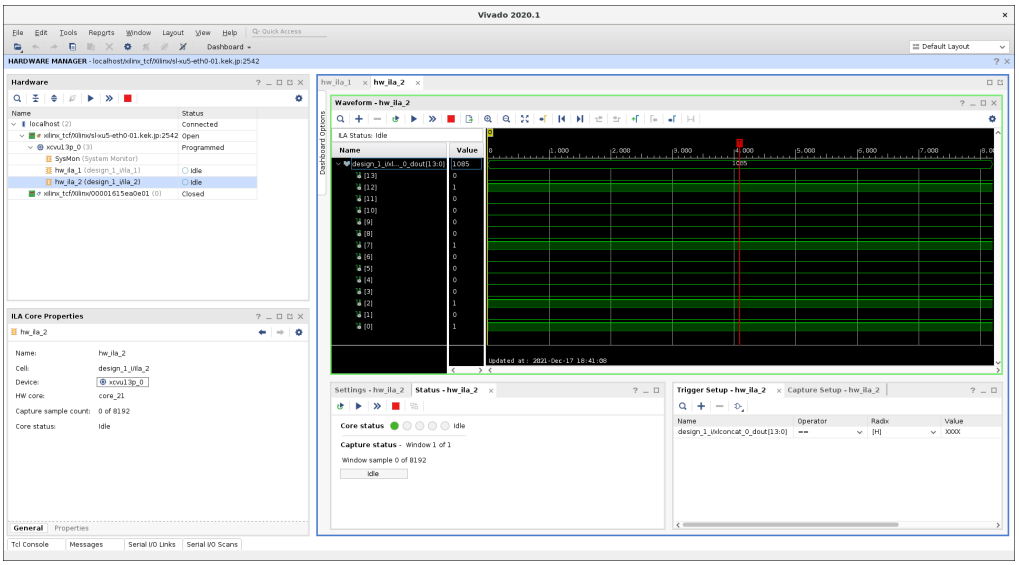


図 4.7 XCVU13P 側の AXI Chip2Chip と Aurora 64B/66B のデバッグポートをモニタし、リンクの確立を確認した

この状態でメモリのアドレス `0x80030000` にアクセスしたところ（付録 C を参照）、問題なく BRAM に PS のソフトウェアからアクセスできた。ただしマスター側とスレーブ側で [ID Width]と [WUSER Width]を一致させる必要がある。

このように、Endcap SL 第 1 試作機の Aurora のシリアル線がハードウェア的に機能し、また AXI Chip2Chip を用いたシリアル通信によるデータ転送ができることを確認した。実際のレーンレートやエラー率などの検証はこれから行う。

4.5 register interface の作成

Vivado では AXI Chip2Chip のサンプルデザインとテストベンチシミュレーション（図 4.8）が利用可能である。Traffic Generator は適当なデータパターンを生成し、ARチャンネル、AWチャンネル、Wチャンネルでデータを AXI Chip2Chip に送信するもので、Traffic Checker は同じ規則で参照データを生成し、Rチャンネルと Bチャンネルで AXI Chip2Chip から返された応答が合致しているかを検知するものである。このテストベンチシミュレーションの動作については付録 I で述べる。図 4.1 のような機構を実現するため、図 4.8 のテストベンチシミュレーションを編集し、図 4.9 のようなテストベンチシミュレーションを作成した。図 4.8 のスレーブ側にあった Traffic Generator と Traffic Checker の代わりにレジスタを複数置き、AXI Chip2Chip slave とレジスタとの間の仲介として register interface を作成した。register interface は AXI4 で命令を受け取り、対象のレジスタにデータの読み書きをするものである。register interface の作成には AXI4 の知識が必要であったので、AXI4 プロトコルについての説明を付録 H に記しておく。

register interface の開発には Verilog-HDL を用いたが、Verilog-HDL では多次元配列のワイヤが使えない。そ

表 4.1 XCVU13P のプローブ結果

ILA bit	Debug port	Value
13	axi_c2c_multi_bit_error_out	0
12	axi_c2c_link_status_out	1
11	axi_c2c_config_error_out	0
10	aurora_pma_init_out	0
9	aurora_pma_init_in	0
8	aurora_reset_pb	0
7	gt_powergood	1
6	gt_reset_out	0
5	sys_reset_out	0
4	link_reset_out	0
3	gt_qpllrefclklost_quad1_out	0
2	gt_qplllock_quad1_out	1
1	mmcm_not_locked_out	0
0	channel_up	1
	13-bit value	0x1085

表 4.2 Zynq MPSoC のプローブ結果

ILA bit	Debug port	Value
18	gt_powergood[0:0]	1
17	gt_reset_out	0
16	sys_reset_out	0
15	link_reset_out	0
14	gt_qpllrefclklost_quad1_out	0
13	gt_qplllock_quad1_out	1
12	mmcm_not_locked_out	0
11	soft_err	0
10	hard_err	0
9	gt_pll_lock	1
8	lane_up[0:0]	1
7	channel_up	1
6	aurora_pma_init_in	0
5	aurora_pma_init_out	0
4	aurora_reset_pb	0
3	axi_c2c_multi_bit_error_out	0
2	axi_c2c_link_error_out	0
1	axi_c2c_config_error_out	0
0	axi_c2c_link_status_out	1
	19-bit value	0x42381

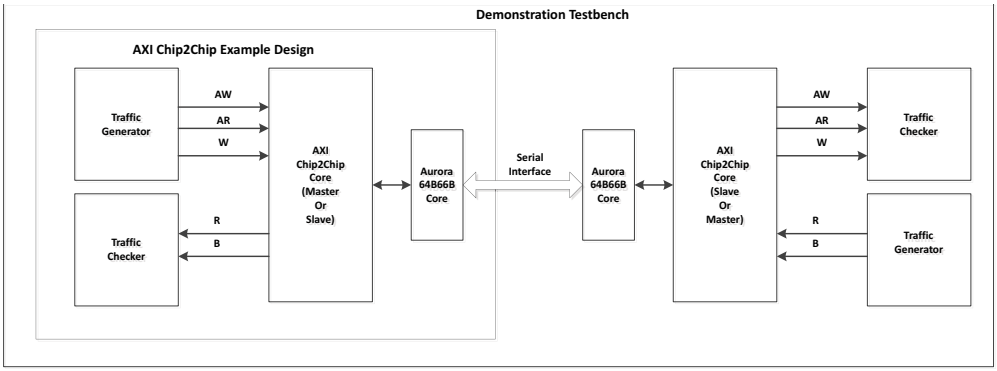


図 4.8 AXI Chip2Chip のテストベンチシミュレーション(Xilinx 2020)

のため図 4.10 のように、ここではレジスタの数に対応した読み書きの信号線 rdata_i、wdata_i（10 個のレジスタがあるならば i $= 0, \ldots, 9$）を用意した。

4.5.1 register interface 内部のステートマシン

register interface 内で走るステートマシンを以下のように設計した。

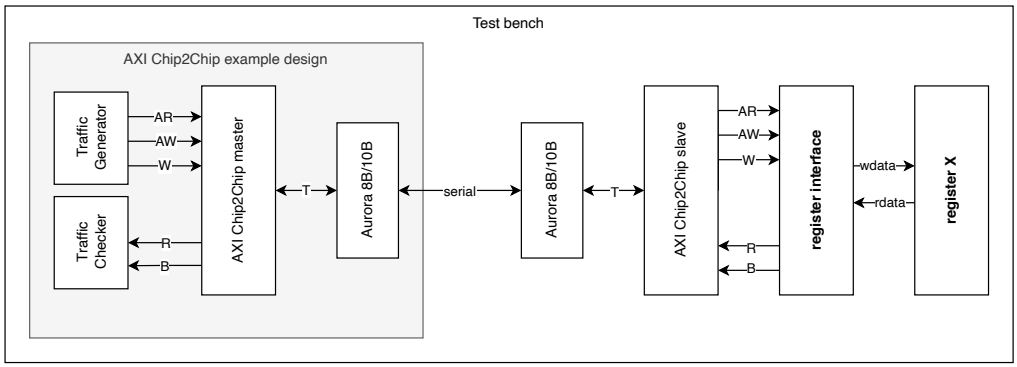


図 4.9 register interface を加えたテストベンチシミュレーション

図 4.10 register interface とレジスタとの接続にはそれぞれ独立な信号線を用意した。

読み出しに関するステートマシンを表した表したタイミングチャートを図 4.11 に示す。レジスタからのデータの読み出しについては、次のような 3 つのステートを遷移するようにした。

RESET ARESETnがディアサートされている間このステートに移る。レジスタ変数を初期化し、IDLE ステートに移る。

IDLE ARREADYをアサートしておき、AXI Chip2Chip からの AEVALIDがアサートされれば ARADDRと ARIDを受け取る。それらをそれぞれ `tmp_raddr` と `tmp_rid` というレジスタ変数に格納し、TRANSFER ステートに移る。

TRANSFER `tmp_raddr` の値に対応したレジスタから rdata_iを用いてデータを読み出し、結果を AXI Chip2Chip に返す。その際には RIDとして `tmp_rid` を返すことで、AXI Chip2Chip との整合性をとる。RRESPとしては基本的に `0b00`（OK）を返し、レジスタが存在しないアドレスであれば `0b10`（SLVERR）を返す。ステートを IDLE にして次の指令を待つ。

書き込みについても同様のステートマシンを作成した。この書き込みステートマシンを表したタイミングチャートを図 4.12 に示した。各ステートでは次のような動作をする。

RESET ARESETnがディアサートされている間このステートに移る。レジスタ変数を初期化し、IDLE ステートに移る。

IDLE AWREADYと WREADYをアサートして待機する。AXI Chip2Chip からの AWVALIDと WVALIDがアサートされれば、AWADDR、AWID、WDATAをそれぞれ `tmp_waddr`、`tmp_wid`、`tmp_wdata` というレジスタ変数に格納する。ステートを TRANSFER に移す。

TRANSFER `tmp_waddr` に対応したレジスタへの信号 wdata_i（$i = 0, \ldots, 9$）を `tmp_wdata` にする。BID を `tmp_wid` にし、BRESPは `tmp_waddr` が妥当であれば `0b00`（OK）、レジスタが存在しな

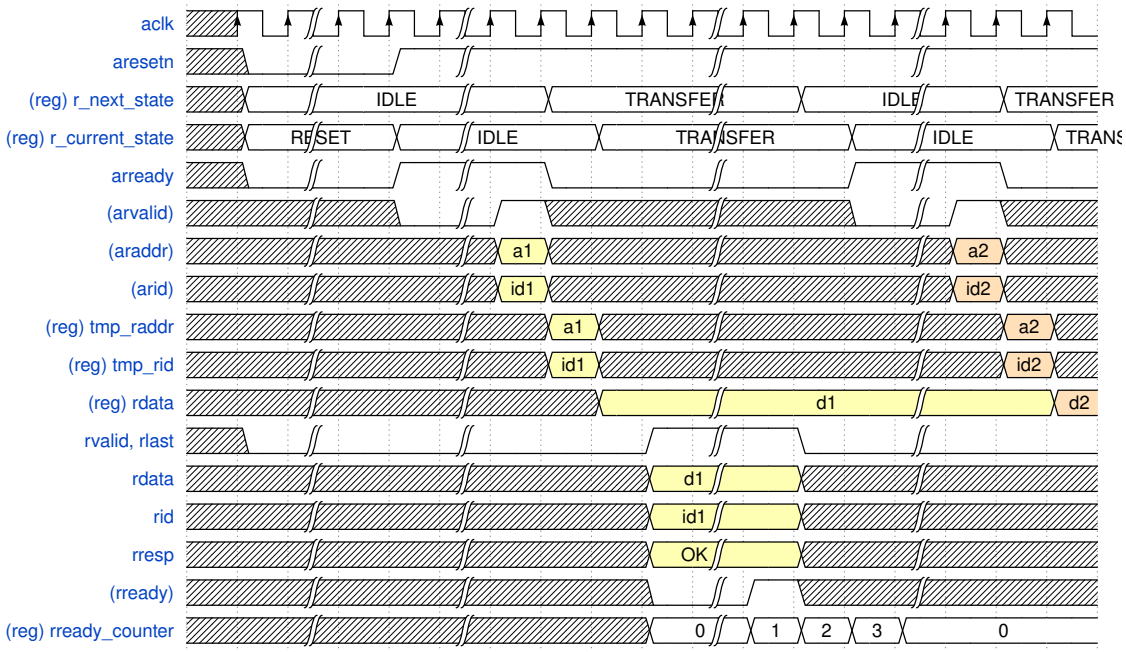


図 4.11 register interface の読み出しに関するステートマシンのタイミングチャート。AXI Chip2Chip からの入力信号は括弧で囲み、AXI Chip2Chip への出力には何もつけていない。またレジスタ変数には (reg) をつけた。(reg) rdata はレジスタからのデータを格納するレジスタ変数である。

いアドレスであれば `0b10`（SLVERR）にする。それと同時に BVALIDをアサートしておき、BREADYもアサートされれば IDLE ステートに戻る。

これら読み出しと書き込みのステートマシンは独立に動作するようにしているが、AXI Chip2Chip から読み出しと書き込みの命令が同時に来たときの排他的処理の機構はまだ設けていない。ただしテストベンチシミュレーションの運用においては、同時に読み出しと書き込みすることはなかったため問題なく動作した。そのような場合の対処の実装は今後の課題である。

4.5.2 register interface を用いたテストベンチシミュレーションの実行

まず 10 個のレジスタを用意し、それらを register interface と接続した。図 4.9 のマスター側の Traffic Generator と Traffic Checker をアドレス 0 から 9 にアクセスするように編集した。テストベンチシミュレーションを実行したところ、10 個のレジスタにデータが正しく書き込まれ（図 4.13）、さらにマスター側でも書き込んだデータが正常に読み出されていることを確かめた。

また 4 つのレジスタに、表 4.3 のようなより実用的な機能を持たせて試験をした。 マスター側からは 32 bit のラ

表 4.3 4 つのレジスタに与えた役割

レジスタのインスタンス名	役割	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 に配布する。
`register_1`	status register	固定の値 `0xEE1234EE` 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する。

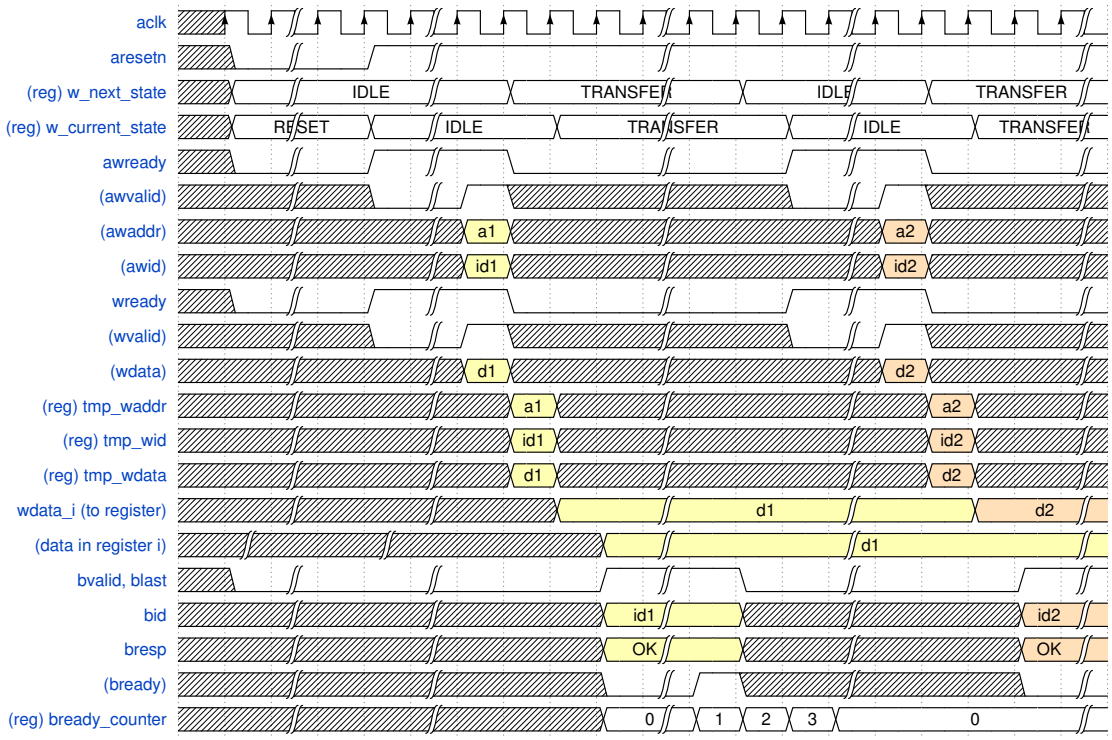


図 4.12　register interface の書き込みに関するステートマシンのタイミングチャート。AXI Chip2Chip からの入力信号は括弧で囲み、AXI Chip2Chip への出力には何もつけていない。またレジスタ変数には (reg) をつけた。(reg) rdata はレジスタからのデータを格納するレジスタ変数である。

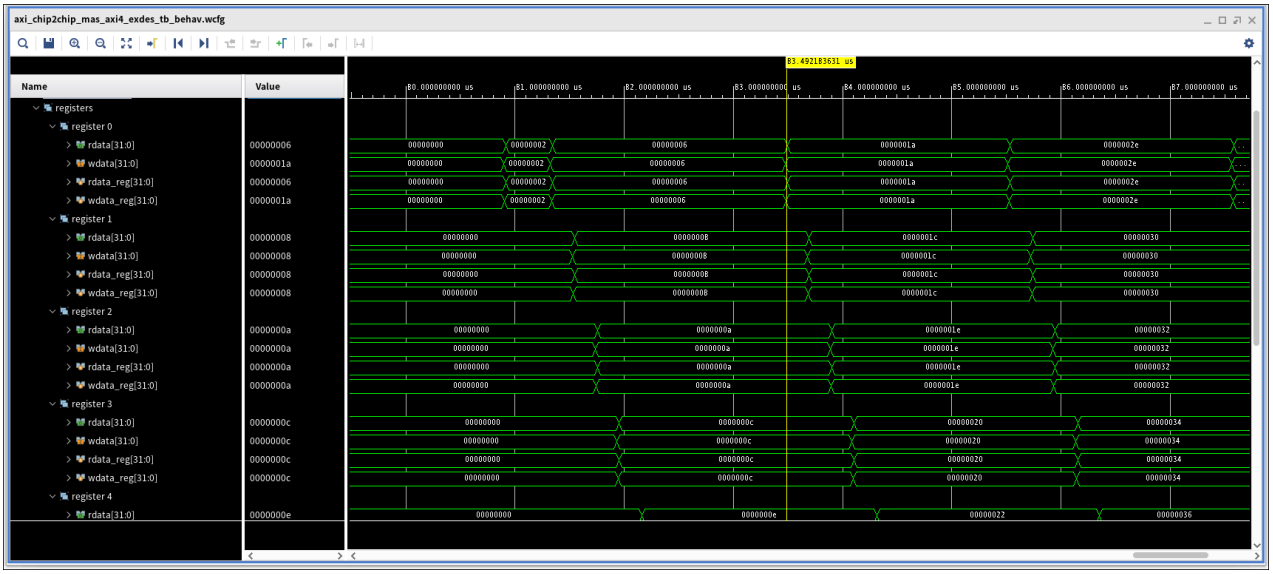


図 4.13　10 個のレジスタに正しく書き込めた様子。マスターから、2 ずつインクリメントしたデータをレジスタ 0 からレジスタ 9 まで順に書き込んだ。_regがついているのはレジスタ内部で用いているレジスタ変数。

ンダムなデータを生成した。テストベンチシミュレーションを実行すると、図 4.14 のようになった。reset register に奇数のデータが書き込まれたとき、reset register と control register の値がリセットされ `0x00000000` になっている。また status register は読み出し専用で、正常に `0xEE1234EE` を出し続けている。また pulse register に奇数データが書き込まれたとき、1 クロック分だけ値が変化してその後 `0x00000000` にリセットされていることがわかる。これはパルス的に変化する信号を出すのに使用できる。

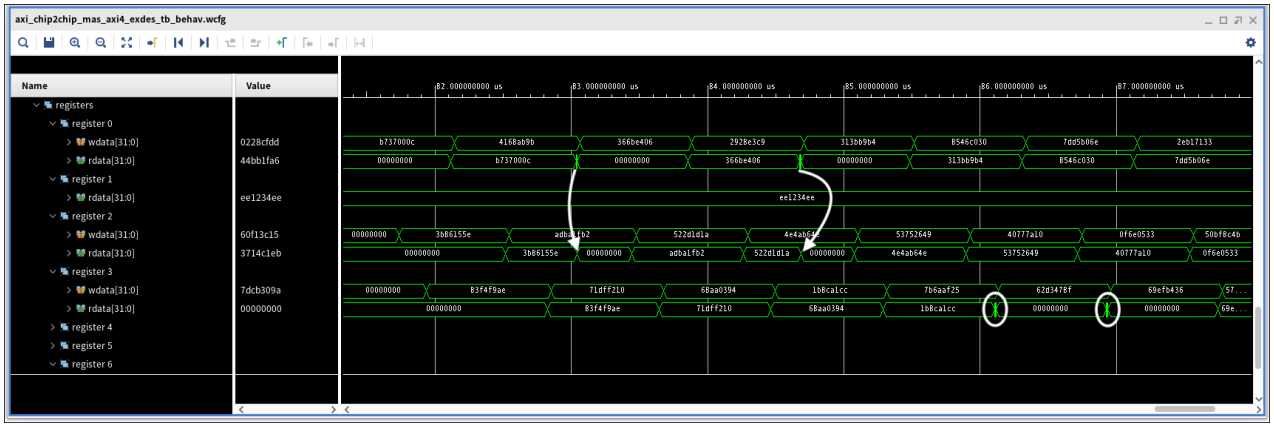


図 4.14　4 つのレジスタに表 4.3 のような役割を持たせたときの挙動

節 4.3 では Endcap SL 第 1 試作機を用いた高速シリアル通信が確立したが、その前に図 4.9 のファームウェアを実機で試験するため、ZCU102 評価ボードを用いたループバック回路の作成を試みており、その詳細は付録 J で述べる。今後、図 4.9 の機構を Endcap SL 第 1 試作機で実装する予定である。

4.6　本章のまとめ

ここでは物理層のトランシーバを使ったファームウェアを実装し、AXI Chip2Chip を用いたシリアル通信するデモンストレーションを Endcap SL 第 1 試作機上で成功させた。また register interface の開発とテストベンチシミュレーションを用いた動作検証も完了した。今後はこれらを統合し、Endcap SL 第 1 試作機上で図 4.9 のようなレジスタ制御の機構を実現する。

第 5 章

まとめと今後の展望

本研究では Zynq UltraScale+ MPSoC で CentOS 7 という通常の Linux をブートできることを確認し、基本的なソフトウェアやファームウェアの開発パスを開拓した。Endcap SL 第 1 試作機に Mercury XU5 メザニンカードをマウントし、そこでも microSD カード起点の CentOS 7 のブートができたことを示した。また Endcap SL 上のクロックジェネレータや温度センサなどの素子を Zynq MPSoC から監視、制御できることを確認した。さらに Zynq MPSoC で Xilinx Virtual Cable が使え、Zynq MPSoC のファームウェアのデバッグや、XCVU13P のプログラムやデバッグをネットワーク越しに行うことができることを示した。AXI Chip2Chip のブリッジング機能を用いた FPGA 間の高速シリアル通信ではリンクを確立させ、Zynq MPSoC から XCVU13P の BRAM にアクセスすることができた。また Zynq MPSoC を用いたソフトウェアやファームウェアの開発や試験のため ZCU102 評価ボードでの開発環境の整備も行なった。

今後は AXI Chip2Chip を用いたシリアル信号線のレーンレートやビットエラー率などの詳細を調査する予定である。また今回は 2 つあるシリアル線のうち 1 つを用いて検証を行なったが、もう片方の検証も行ったり、もう 1 つのクロックジェネレータが動作するかなど、Endcap SL 第 1 試作で行うべき試験はまだ残っている。さらに XCVU13P のレジスタ制御や、Endcap SL を PS ボードと接続して読み出しデータのダンプパスを確立し、2027 年の HL-LHC のための準備を進める。

付録 A

典型的な PetaLinux での作業の詳細

```
source <path to>/petalinux/2020.1/settings.sh # Activate PetaLinux 2020.1
```

で PetaLinux 2020.1 を使える状態にする[1]。

```
petalinux-create --type project --name <project name> --template zynqMP
cd <project name>
```

で PetaLinux でプロジェクトを立ち上げ、それに入る。

Vivado で生成した XSA ファイルを読み込ませて基本的な設定を行うには

```
petalinux-config --get-hw-description=<path to the dir of XSA>
```

を実行すれば、図 A.1 のような画面に切り替わり、ここで基本的なコンフィギュレーションを行う。XSA の更新がないならば、`get-hw-description` のオプションはなくてよい。

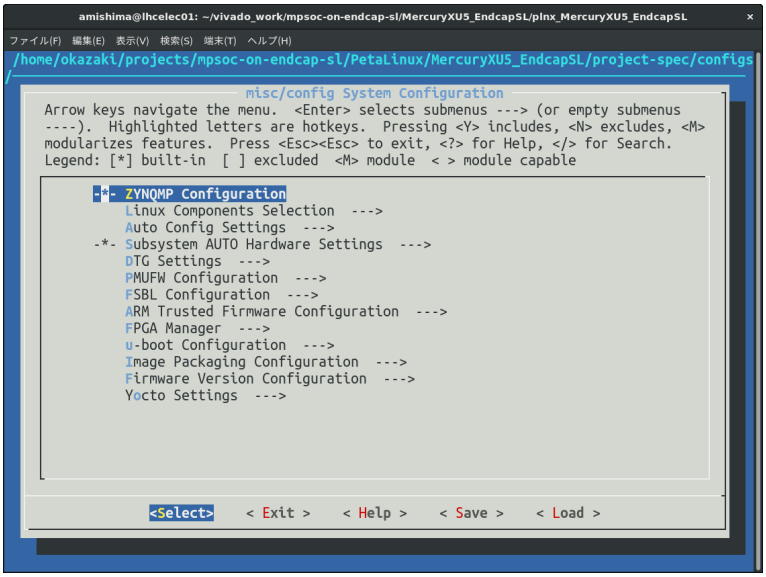


図 A.1 `petalinux-config` 画面

[1] ログイン時に自動的に PetaLinux 2020.1 を使える状態にするように `$HOME/.bashrc` に書いておいたり、`alias` を作成すると便利である。

ZCU102 に関してはコードブロック A.1 に設定した。これは図 2.7 のようなパッケージングをし、SD カード第 1 パーティションに`BOOT.BIN`、`image.ub`、`boot.scr`を置き、第 2 パーティションの rootfs を使用するような設定である。

```
Subsystem AUTO Hardware Settings
  ---> Advanced bootable images storage Settings
    ---> boot image settings           ---> image storage media (primary sd)
    ---> u-boot env partition settings ---> image storage media (primary sd)
    ---> kernel image settings         ---> image storage media (primary sd)
    ---> jffs2 rootfs image settings   ---> image storage media (primary flash)
    ---> dtb image settings            ---> image storage media (from boot image)
DTG Settings
  ---> (zcu102-rev1.0) MACHINE_NAME
  ---> Kernel Bootargs
    ---> [ ] generate boot args automatically (console=ttyPS0,115200
    clk_ignore_unused root=/dev/mmcblk0p2 rootfstype=ext4 rw rootwait earlyprintk)
    user set kernel bootargs
Image Packaging Configuration
  ---> Root filesystem type (EXT4 (SD/eMMC/SATA/USB))
  ---> (/dev/mmcblk0p2) Device node of SD device
```

コードブロック A.1　ZCU102 で行った設定の例

ユーザ編集用の DTS は

```
<PetaLinux project>/project-spec/meta-user/recipies-bsp/device-tree/files/system-user
    .dtsi
```

であるので、必要ならばそれを編集する。

```
petalinux-build
```

というコマンドでカーネルや FSBL、デバイスツリー、rootfs などがビルドされ、`<PetaLinux project>/images/linux/`の下にできる。

```
cd images/linux
petalinux-package --boot --fsbl zynqmp_fsbl.elf --fpga system.bit --pmufw pmufw.elf
    --u-boot --force
```

これで`BOOT.BIN`がパッケージされる。`force`オプションは初回ならばつけなくていいが、2 回目以降は上書きするのに必要なオプションである。

付録 B

Ultra96-V2 を用いた開発パスの確認と物体検出処理の高速化

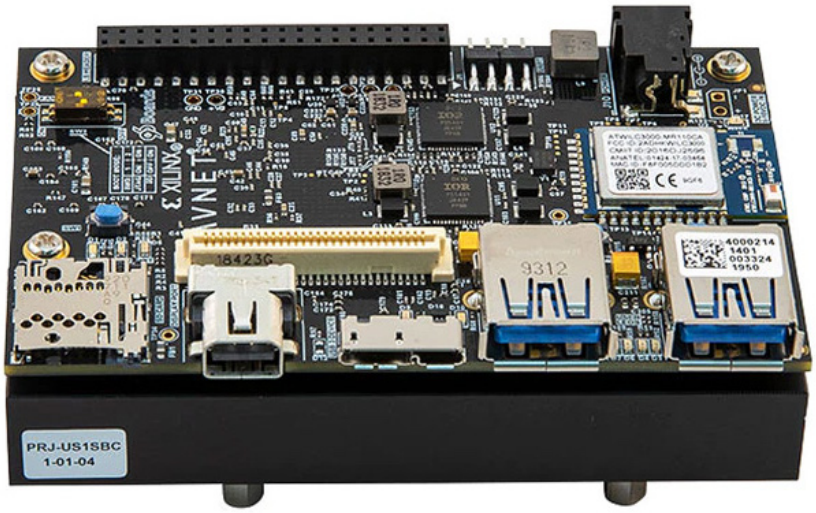


図 B.1　Ultra96-V2 シングルボードコンピュータ（Avnet）

Avnet® 社が製造している Ultra96-V2 シングルボードコンピュータ（図 B.1）には、Xilinx 社製の Zynq UltraScale+ MPSoC（ZU3EG）が搭載されている。また Enclustra 社製の Mercury XU5 と Mercury+ PE1 よりもコミュニティが厚く資料も豊富であった。Mercury XU5 と Mercury+ PE1、ZCU102 評価ボードを用いた開発パスを開拓する前に、この Ultra96-V2 を用いて PetaLinux を使った開発パスの確認と、ファームウェアの作成練習を行った。ファームウェア作成練習の題材としては、YOLOv3 という物体検出に使われるニューラルネットワークの行列演算を FPGA で行い、物体検出処理の高速化を試みた。これについては奥畑 (2019)を参考にしたが、これは Ultra96-V2 ではなく Ultra96 を用いたものであった。具体的には次のようなことを行った。

- Linux のブート
  - 提供されている Linux イメージの SD カードブート（節 B.1.1）
  - PetaLinux の BSP を用いた Linux のブート（節 B.1.2）
  - Vivado の BDF を用いたファームウェアの作成（節 B.1.3）
- ニューラルネットワークの行列演算をハードウェア化（節 B.2）

開発環境には Vivado 2019.2 と PetaLinux 2019.2 を用いた。

B.1　Linux のブート

B.1.1　提供されている Linux イメージの SD カードブート

まず Ultra96-V2 の箱に同封されていた「Ultra96-V2 Quick-Start」というインストラクション[1]をフォローした。これは提供されているイメージファイルを microSD カードに焼き、電源を ON にすることですぐに Linux を起動できるというものだった。

まず提供されているイメージファイルをダウンロードした。これを Etcher というアプリケーションを用いて、microSD に展開し、ブータブル SD を作成した。Ultra96-V2 ボードに SD を挿入し、DIP スィッチの 1 が OFF、2 が ON になっていることを確かめた。電源のプラグを挿し（電源ジャックの近くにある LED が緑色に光る）、SW4 を押して離すと Linux が起動した。Ultra96-V2 ボードには Wi-Fi チップが搭載されていて、無線 LAN でアクセスできる[2]。IP アドレスは `192.168.2.1` が割り振られるようになっていた。また無線 LAN 接続可能な開発用コンピュータからは、Ultra96-V2 デバイスが `Ultra96-V2_<MAC address>` という名前で検出されたので、これに接続した（図 B.2）。

図 B.2　開発用コンピュータから Wi-Fi で Ultra96-V2 を `Ultra96-V2_<MAC address>` として検出

ブラウザから `http://192.168.2.1` で接続すれば、図 B.3 のウェブアプリから LED の制御をすることができた。またウェブアプリから外部の Wi-Fi ルータに接続するよう設定すれば、インターネットにも接続できた。

SW4 を押して離す、またはコマンド `shutdown -h now` や `poweroff` でシャットダウンした。より詳細にはこれは Ultra96-V2 のホームページ（Avnet）の [Technical Documents] → [Ultra96-V2 Getting Started Guide]から入手可能な Getting Started Guide を参照するとよい。

B.1.2　PetaLinux の BSP を用いた Linux のブート

このような Xilinx 社製の SoC が搭載されたボードでは、そのベンダーは PetaLinux の Board Support Package（BSP）を提供していることが多い。BSP を用いることで、リファレンスデザインの PetaLinux プロジェクトをすぐに作成できる。Ultra96-V2 の場合、Ultra96-V2 のホームページ（Avnet）の [Reference Design] → [PetaLinux Board Support Packages]より [Ultra96-V2 – PetaLinux 2019.2 BSP]

[1] これは Ultra96-V2 のホームページ（Avnet）の [Technical Documents] → [Ultra96-V2 Quick-Start Card]からも入手可能。
[2] ただしブート時にログを見てデバッグする必要があるならば、UART 通信用のモジュールを別途購入する必要がある。

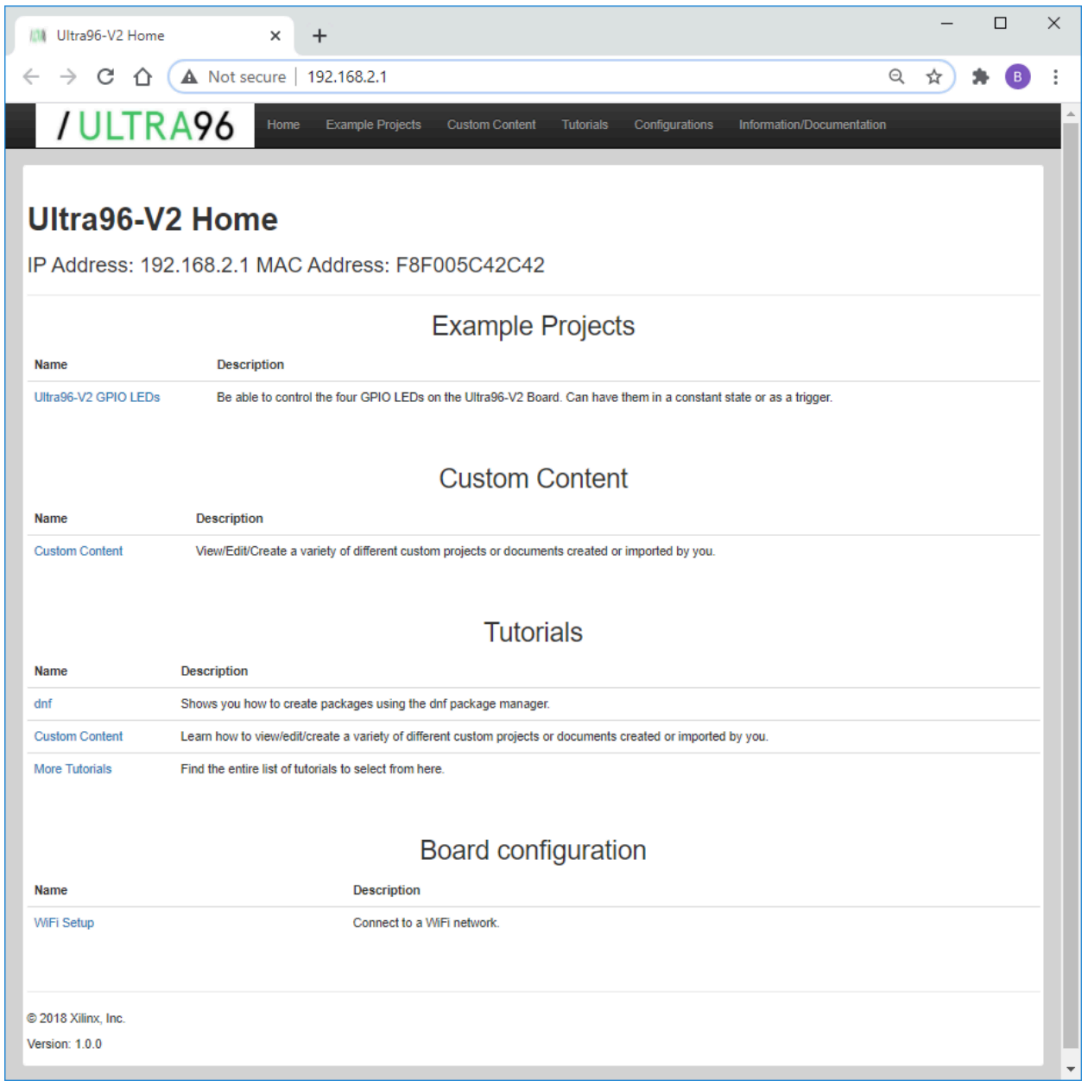


図 B.3 Ultra96-V2 のウェブアプリの例

（`ultra96v2_oob_2019_2.bsp`[3]）をダウンロードし、次のように PetaLinux プロジェクトの作成、ビルド、パッケージングを行った。

```
petalinux-create -t project -s <path_to>/ultra96v2_oob_2019_2.bsp
cd ultra96v2_oob_2019_2
petalinux-build
cd images/linux
petalinux-package --boot --fsbl zynqmp_fsbl.elf --fpga system.bit --pmufw pmufw.elf
    --u-boot
```

これでブートイメージ`BOOT.BIN`がパッケージングされた。microSD の第 1 パーティションに`BOOT.BIN`と`image.ub`をコピーし[4]、第 2 パーティションに`rootfs.tar.gz`を展開した。SD カード

[3] oob は out-of-box のことで、「製品などを入手後にすぐ使える」という意味らしい。
[4] PetaLinux 2019.2 を用いていたため、`boot.scr`は用いずにブートする。`boot.scr`が必要となるのはバージョン 2020.1 から。

を安全に取り外し、Ultra96-V2 ボードに挿入して電源を入れ、Wi-Fi で接続すると、節 B.1.1 と同じ Linux が起動していることを確認できた。

B.1.3 Vivado の BDF を用いたファームウェアの作成

ここでは自分で回路を作成し、Linux 起動時に PL をプログラムして使えるようにした。簡単な回路として、BRAM を追加し、PS から BRAM の内容を読み書きする。開発の概略図は図 B.4 の通り。

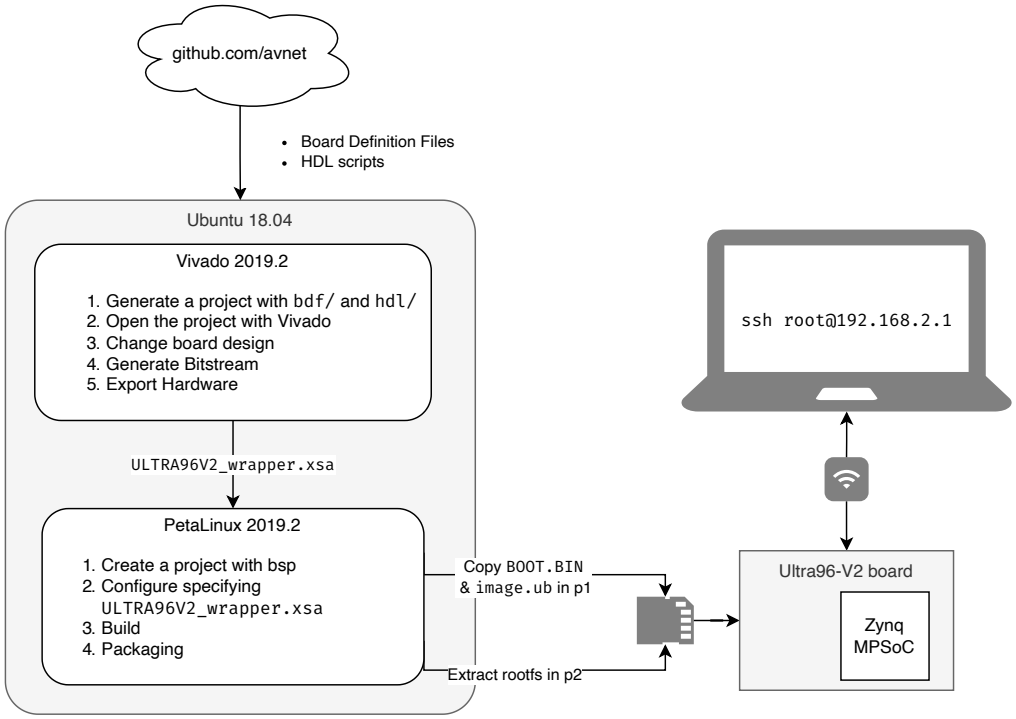


図 B.4 Ultra96-V2 の開発フロー

BSP から作成した PetaLinux のプロジェクトディレクトリ `ultra96v2_oob_2019_2/` の中の `hardware/ULTRA96V2_2019_2/` には Vivado のプロジェクトファイル一式が揃っていた。しかしたとえば `hardware/ULTRA96V2_2019_2/ULTRA96V2.xpr` を Vivado でオープンして FPGA ビットストリームを生成しようとしても、Avnet 社が作成したモジュールなどの情報がないため成功しなかった。

そこで、Avnet の GitHub のリポジトリから Board Definition Files（BDF）と HDL ファイルの一式を `git clone` して、Vivado プロジェクトを作成した（コードブロック B.1）。BDF というのは、Vivado のプロジェクトを作成するときに便利なテンプレートのようなものである。また HDL ファイルには回路の情報が書いてあり、Tcl ファイルを `source` することで回路を生成できる。

```
git clone -b master https://github.com/avnet/bdf.git
git clone -b 2019.2 https://github.com/avnet/hdl.git
cd hdl/Scripts/
source ~/Xilinx/Vivado/2019.2/settings64.sh # Activate Vivado 2019.2
vivado -mode batch -source ./make_ultra96v2_oob.tcl
```

コードブロック B.1 GitHub から BDF と HDL ファイルの一式をクローンして Vivado プロジェクトを作成

コードブロック B.1 最後のコマンドは Vivado のバッチモード（CUI）で`make_ultra96v2_oob.tcl`をソースにすることを意味している。このコマンドは`hdl/Scripts/`から実行しなければならない。また GitHub からクローンしてきた 2 つのディレクトリ`bdf/`と`hdl/`は同じディレクトリにいるような構成にしなければならない。`make_ultra96v2_oob.tcl`の中で、`bdf/`の Ultra96-V2 のものを相対パスで参照するようになっているからだ。`bdf/`と`hdf/`には関係ないボードのファイルが含まれているので、それらは削除してよい。

Vivado による`source`コマンドの実行が終了したら、`hdl/Projects/ultra96v2_oob/ULTRA96V2_2019_2/`に Vivado のプロジェクトファイル一式がある。この中の`ULTRA96V2.xsa`を PetaLinux で持ちいれば、これまでと同様にリファレンスデザインの PL が展開される。また Vivado のプロジェクトは`ULTRA96V2.xpr`である。このブロックデザインを見ると、リファレンスデザイン（oob）のものになっていた（図 B.5）。ここでは練習のために これらのモジュールをすべて消し、1 から PL のデザインを作った。

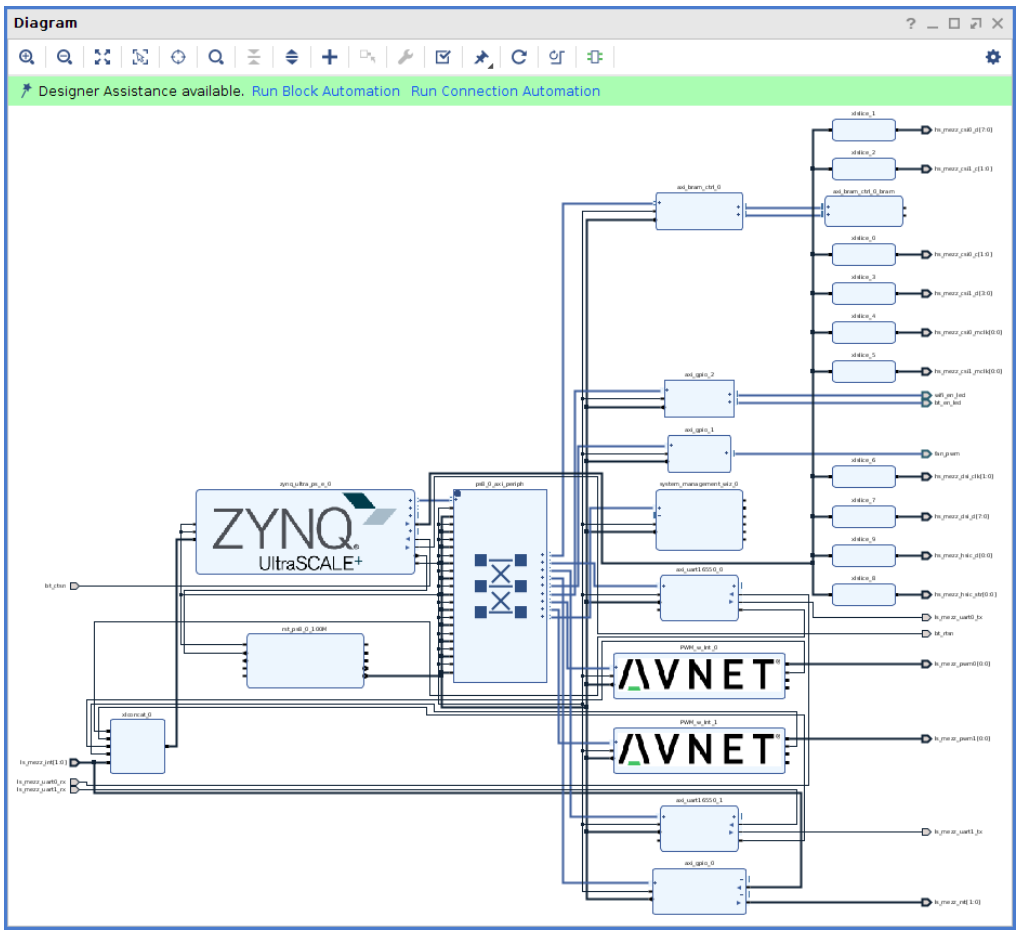


図 B.5　OOB のブロックデザイン。これをすべて消して、1 からデザインを作成した。

まず [Add IP]から Zynq UltraScale+ MPSoC の IP ブロックを追加した。このブロックをダブルクリックし、[PS-PL Configuration] → [PS-PL Interfaces] → [Master Interface] → [AXI HPM0 FPD]にチェックを入れた。pl_clk0のピンと maxihpm0_fpd_aclk、maxihpm1_fpd_aclkを線で繋いだ。AXI BRAM Controller と Block Memory Generator の IP を追加した。BRAM にはポート A とポート B があるが 1 つ消してそれらをつなぎ、[Run Block Automation] → [Select All] → [OK]をクリックすると図 **??**のようなブロックデザインが完成した。また [Address Editor]のタブを開いて [Auto Assign Address]をクリックして、axi_bram_ctrl_0の物理アドレスを`0xA0000000`として割り振った（図 B.7）。

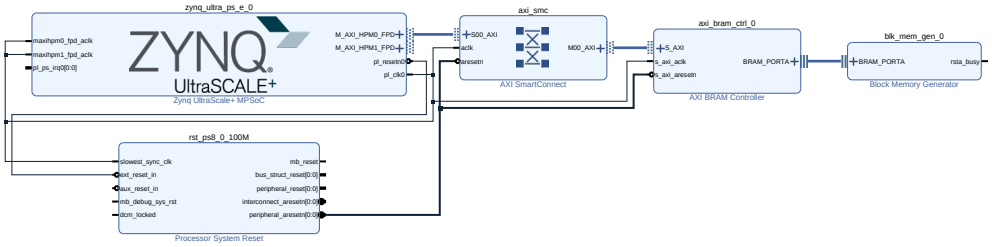


図 B.6 新たに作成した BRAM のブロックデザイン

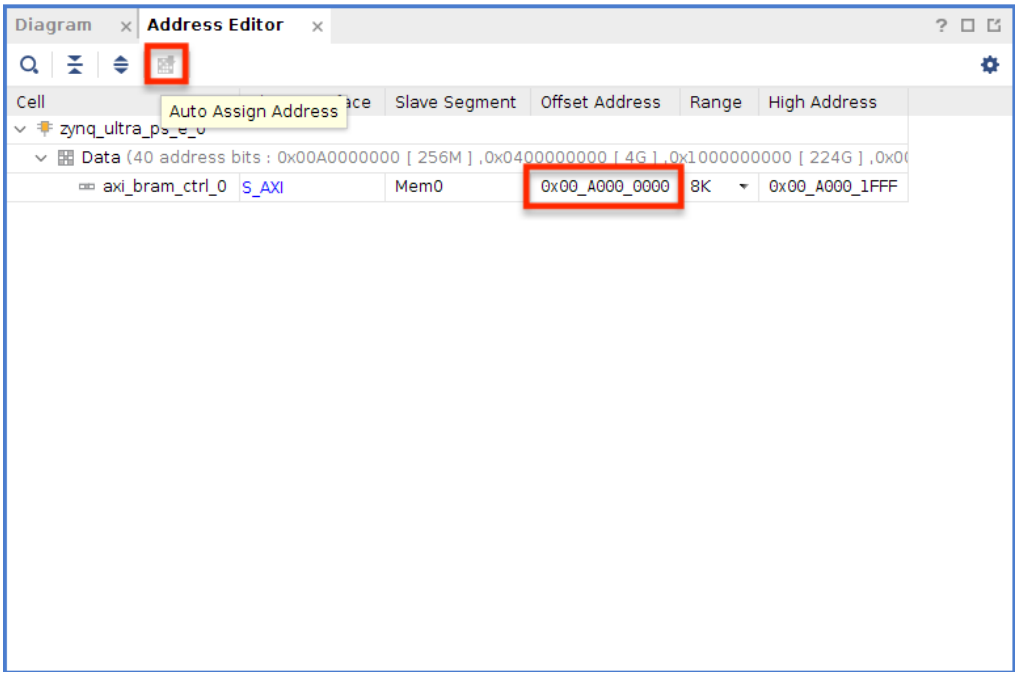


図 B.7 BRAM のアドレス割り振り

これで [Flow] → [Generate Bitstream]をクリックして、ビットストリームを生成する。処理が終わったら [File] → [Export] → [Export Hardware]をクリック、[Include Bitstream]にチェックを入れて、OK をクリックした。Vivado プロジェクトディレクトリに`ULTRA96V2_wrapper.xsa`が生成された。

Vivado を終了し、節 B.1.2 の PetaLinux のプロジェクトディレクトリに話を移す。ここでは`ULTRA96V2_wrapper.xsa`があるディレクトリを指定して`petalinux-config`を行い、ビルドした。

```
cd <path to>/ultra96v2_oob_2019_2/
petalinux-config --get-hw-description=<path to>/hdl/Projects/ultra96v2_oob/
    ULTRA96V2_2019_2/ --oldconfig
petalinux-build
cd images/linux
petalinux-package --boot --fsbl zynqmp_fsbl.elf --fpga system.bit --pmufw pmufw.elf
    --u-boot --force
```

あとは節 B.1.2 と同じように SD の第 1 パーティションに`BOOT.BIN`と`image.ub`をコピーし、第 2 パーティションに`rootfs.tar.gz`を展開した。Ultra96-V2 ボードに SD を挿して起動してログインしたところ、BRAM

に割り当てた物理アドレス`0xA0000000`の値を読み書き可能なことを確認した。`rootfs.tar.gz`では`devmem`コマンドが使えるので、それを用いた。

```
root@ultra96v2_oob_2019_2# devmem 0xa0000000
0x00000000
root@ultra96v2_oob_2019_2# devmem 0xa0000000 32 0xdeadbeef
root@ultra96v2_oob_2019_2# devmem 0xa0000000
0xDEADBEEF
```

B.2　ニューラルネットワークの行列演算をハードウェア化する

YOLOv3 (Redmon and Farhadi 2018)は入力された画像から物体の位置と種類を検出する機械学習モデルである。ここでは行列演算 IP を用いた回路を作成し、Ultra96-V2 のハードウェアで物体検出を行うことを試みた。

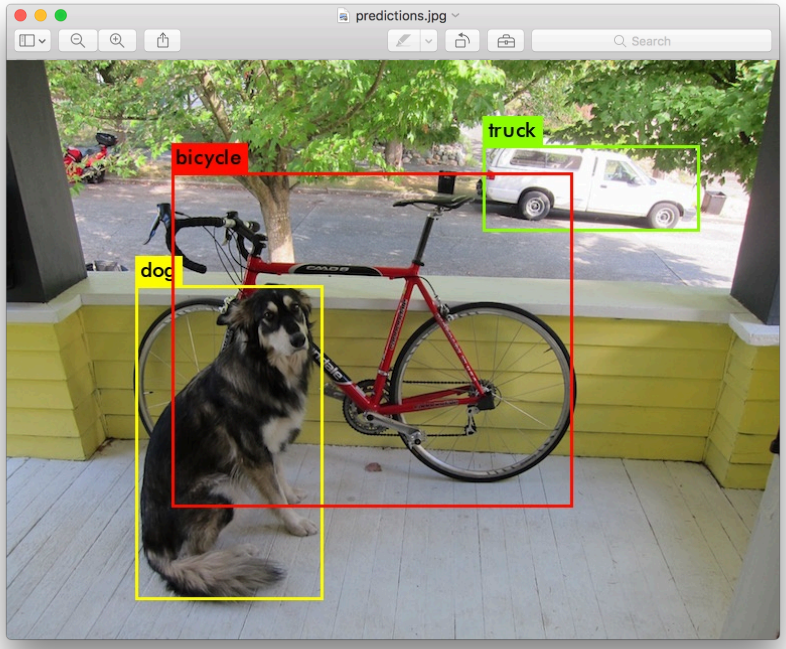


図 B.8　YOLOv3 を用いて画像の物体検出をした例

B.2.1　行列演算 IP の作成

32 × 32 の行列演算 IP として、Xilinx Application Note 1170 (Daniele Bagni et al. 2016)の HLS_accel を活用した。これは 32 × 32 の行列 A と B の要素を AXI4-Stream で受け、それらの積 $C = AB$ を AXI4-Stream で出力するものである。このリファレンスデザインをダウンロードし解凍すると`xapp1170_2015v4/`が入手できた。この IP は Zynq-7000 SoC のものなので、Zynq MPSoC 用に高位合成するためのスクリプト`xapp1170_2015v4/empty/hls/run_hls_script.tcl`を編集した。前半はすべてコメントアウトし、後半をコードブロック B.2 のように編集した。

```
############################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 2012 Xilinx Inc. All rights reserved.
############################################################

open_project hls_wrapped_mmult_prj
set_top HLS_accel
add_files mmult_accel.cpp -cflags "-DDB_DEBUG"
add_files -tb mmult_test.cpp -cflags "-DDB_DEBUG"
open_solution "solution0"
# Change below according to Ultra96-V2 board
set_part {xczu3eg-sbva484-1-i}
create_clock -period 10 -name default
set_directive_inline "mmult_hw"
set_directive_pipeline -II 1 "mmult_hw/L2"
set_directive_array_partition -type block -factor 16 -dim 2 "mmult_hw" a
set_directive_array_partition -type block -factor 16 -dim 1 "mmult_hw" b
csim_design -clean
#-setup
csynth_design
# Add the line below to use HLS_acces as an IP in Vivado block design
export_design -rtl verilog -format ip_catalog
close_project
quit
```

コードブロック B.2　編集した`run_hls_script.tcl`。コメントアウトした前半は省略してある。

編集後、このスクリプトを実行して C 言語によるソースコードから Vivado の IP を生成した。

```
source <path to>/Vivado/2019.2/settings64.sh # activate Vivado 2019.2
cd <path_to>/xapp1170_2015v4/empty/hls/
vivado_hls -f run_hls_script.tcl
```

完了すると、HLS_accel が

```
xapp1170_2015v4/empty/hls/hls_wrapped_mmult_prj/sokution0/impl/ip/
```

配下にできた。

生成した Vivado の IP をブロックデザインに組み込めるようにするため、Vivado プロジェクトで [Tools] → [Settings...] → [IP] → [Repository]から

```
xapp1170_2015v4/empty/hls/hls_wrapped_mmult_prj/sokution0/impl/ip/
```

を選択し、[Apply]をクリックした。新しいブロックデザインを作成し、[Add IP]から HLS_accel を検索してこれを追加した。また Zynq UltraScale+ MPSoC の IP も追加し、ブロックをダブルクリックして次のように設定した。

- [PS-PL Configuration] → [PS-PL Interfaces]
  - [Master Interface]
    - ☑ [AXI HPM0 FPD]
    - ☑ [AXI HPM0 LPD]
  - [Slave Interface]
    - ☑ [AXI HPC0 FPD]
- [Clock Configuration] → [Output Clocks] → [PL Fabric Clocks]
  - ☑ [PL0] ☑ [PL1] ☑ [PL2] ☑ [PL3]
- [I/O Configuration] → [Low Speed] → [I/O Peripherals] → [I/O Peripherals]
  - [UART]
    - [UART0] → ☑ [MODEM]
    - [UART1] → ☑ [MODEM]
  - [GPIO] → ☑ [GPIO EMIO]

追加した HLS_accel は AXI4-Stream でデータを入出力するが、PS のインターフェイスは AXI4 であるため直接は接続できない。そこで AXI Direct Memory Access（AXI DMA）を介して AXI4 に変換し、PS に接続した。AXI DMA の設定は [Enable Scatter Gather Engine]のチェックを外してシンプル DMA モードした。シンプル DMA モードというのは、転送するメモリ領域の開始アドレスと長さを AXI DMA にセットして実行するものである。最終的に図 B.9 のような回路を作成した。これで FPGA ビットストリームを生成し、XSA を出力した。

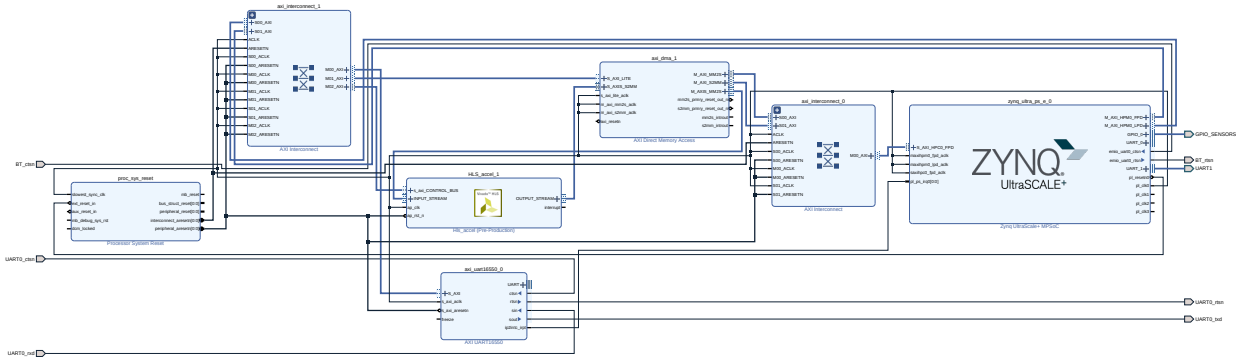


図 B.9　HLS_accel をブロックデザインに追加し、AXI DMA を介して PS と接続した

B.2.2　カーネルモジュール u-dma-buf の利用

シンプル DMA モードにしたので、Linux のユーザ空間から DMA 転送するのに連続したメモリ領域を確保する必要がある。それをするのに u-dma-buf[5] を用いた。これは UIO を使ってユーザ空間でデバイスドライバを動かす場合の DMA バッファを提供する。

まず PetaLinux のプロジェクトディレクトリに `cd` し、カーネルモジュールの雛形を作成した。

[5] https://github.com/ikwzm/udmabuf

```
petalinux-create -t modules --name u-dma-buf --enable
```

これで`project-spec/meta-user/recipes-modules/u-dma-buf/`にテンプレートができる。この配下の`files/u-dma-buf.c`を GitHub から`git clone`した`u-dma-buf.c`と差し替え、モジュールをビルドする。

```
cd project-spec/meta-user/recipes-modules/u-dma-buf/files/
git clone -b u-dma-buf-master https://github.com/ikwzm/udmabuf.git
cp udmabuf/u-dma-buf.c .
petalinux-build -c u-dma-buf
```

AXI DMA と HLS_accel を UIO を使ってアクセスできるよう、コードブロック B.3 のようにデバイスツリー[6]を編集した。

```
/include/ "system-conf.dtsi"
/ {
    chosen {
        bootargs = "console=ttyPS0,115200 earlyprintk uio_pdrv_genirq.of_id=generic-
    uio rootfstype=ext4 root=/dev/mmcblk0p2 rw rootwait";
        stdout-path = "serial0:115200n8";
    };
    u-dma-buf@0 {
        compatible = "ikwzm,u-dma-buf";
        device-name = "u-dma-buf0";
        size = <0x080000>; // 0x080000 Bytes = 524288 Bytes = 512 KiB
        dma-coherent;
    };
    u-dma-buf@1 {
        compatible = "ikwzm,u-dma-buf";
        device-name = "u-dma-buf1";
        size = <0x080000>; // 0x080000 Bytes = 524288 Bytes = 512 KiB
        dma-coherent;
    };
};

&axi_dma_1 {
    compatible = "generic-uio";
    dma-coherent;
};
&HLS_accel_1 {
    compatible = "generic-uio";
};
```

[6] `project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi`

コードブロック B.3　編集したデバイスツリー

```
petalinux-build
```

ですべてをビルドし、これまで通りパッケージングをして SD カードの作成をし、Ultra96-V2 を起動した。

B.2.3　行列演算 IP のテスト

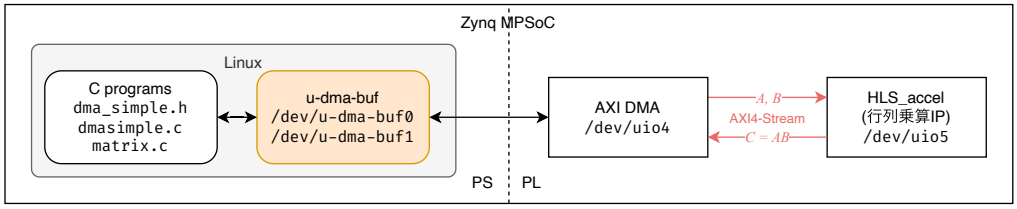


図 B.10　HLS_accel を用いた行列演算のためのソフトウェア、ファームウェア構造

Linux を起動すると、`/dev/uio4` に HLS_accel が、`/dev/uio5` に AXI DMA が割り当てられた。また u-dma-buf が `/dev/u-dma-buf0` と `/dev/u-dma-buf1` に割り当てられていた。2 つあるのはデータの入力と出力のためである。

```
root@ultra96v2-oob-2019-2:~# ls /dev/uio
uio0  uio1  uio2  uio3  uio4  uio5

root@ultra96v2-oob-2019-2:~# cat /sys/class/uio/uio*/name
axi-pmon
axi-pmon
axi-pmon
axi-pmon
HLS_accel
dma
```

HLS_accel にデバイスファイルを割り当てているのは、使用開始時に制御インターフェイスのオフセット 0 のアドレスに `0x81` を書く必要があるためである。`/sys/class/u-dma-buf/u-dma-buf*/size` には 524288 （これは 10 進数で、デバイスツリーで指定した `0x8000` に対応）が書かれており、`/sys/class/u-dma-buf/u-dma-buf*/phys_addr` にはメモリの開始アドレスが書かれていた。

```
root@ultra96v2-oob-2019-2:~# cat /sys/class/u-dma-buf/u-dma-buf*/size
524288
524288

root@ultra96v2-oob-2019-2:~# cat /sys/class/u-dma-buf/u-dma-buf*/phys_addr
0x000000005fd00000
0x000000005fd80000
```

u-dma-buf のカーネルモジュールは `/lib/modules/4.19.0-xilinx-v2019.2/extra/u-dma-buf.ko` であるので、これをモジュールとしてロードした。

```
# insmod u-dma-buf.ko u-dma-buf0=1048576 u-dma-buf1=1048576
insmod: ERROR: could not insert module u-dma-buf.ko: File exists
```

このエラーはもうロード済みだったことを示しており、無視して良い。

さて HLS_accel を使って行列演算ができるか、動作確認をした。コードブロック B.4 は HLS_accel での行列計算の結果を単純にソフトウェアでの計算結果と比較するプログラム `matrix.c` の一部である。ここでは最初に行列 `a`, `b` の要素を `intake_buf.buf` にコピーしている。DMA への入力 `intake_buf.buf` と出力 `outlet_buf.buf` のアドレスを、`dma_setup` 関数で設定している。`dma_start` で転送サイズを書いて転送が開始される（1 要素 4 B なのですべての要素数の 4 倍）。`outlet_buf.buf` への DMA 転送が完了するまで `dma_wait_irq` で待つ。転送が完了したら `outlet_buf.buf` から要素を `c` に取り出して、ソフトウェアで計算した `cref` と値を比較している。詳細な説明は奥畑 (2019)を参照されたい。

```
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        ((float *)(intake_buf.buf))[i * n + j] = a[i * n + j];  // Copy matrix a
    }
}
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        ((float *)(intake_buf.buf))[i * n + j + m * m] =
            b[i * n + j];  // Copy matrix b
    }
}

// Set the addresses of AXI DMA input intake_buf.buf and output outlet_buf.buf
dma_setup(dma_regs, intake_buf.phys_addr, outlet_buf.phys_addr);

// Start DMA transfer (i.e. starting matrix calculation process)
dma_outlet_start(dma_regs, (m * m * 4));
dma_intake_start(dma_regs, (m * m * 2 * 4));

// Wait for the transfer
dma_wait_irq(dma_regs);
dma_clear_status(dma_regs);

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        // Copy values in DMA buffer into the matrix c
        c[i * n + j] = ((float *)(outlet_buf.buf))[i * n + j];
    }
```

```
}

// Software-based calculation of matrix cref
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        float sum = 0;
        for (t = 0; t < k; t++) { sum += a[i * k + t] * b[t * n + j]; }
        cref[i * n + j] = sum;
    }
}

// Compare c and cref
int cnt_ok = 0;
int cnt_ng = 0;
for (i = 0; i < m * m; i++) {
    if (c[i] == cref[i]) {
        cnt_ok++;
    } else {
        printf("NG: hw %f, sw %f\n", c[i], cref[i]);
        cnt_ng++;
    }
}

// Print the result of comparison above
printf("OK: %d, NG: %d, Total: %d\n", cnt_ok, cnt_ng, (m * m));
```

コードブロック B.4　HLS_accel による行列計算結果をソフトウェアでの計算結果と比較するプログラム `matrix.c` の一部

これをコンパイルし実行すると、HLS_accel で計算した値とソフトウェアで計算した値が正しく一致したことが確かめられた。

```
# gcc -O0 matrix.c dma_simple.c
# ./a.out
OK: 1024, NG: 0, Total: 1024
```

B.2.4　YOLOv3 の高速化

HLS_accel を用いて YOLOv3 での物体検出処理がどのくらい高速化できるのかを試した。

まず https://github.com/pjreddie/darknet.git のコミットハッシュ `f6d8617` のものを `git clone` して、HLS_accel を使わずに処理をした。

```
# First, logged in to Linux running on Ultra96-V2
git clone -b f6d8617 https://github.com/pjreddie/darknet.git
cd darknet
```

```
make
wget https://pjreddie.com/media/files/yolov3.weights # Download pre-trained weights
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg # Detect
```

モデルが形成され、検出が行われる。出力結果の画像は`predictions.png`に反映された。

```
data/dog.jpg: Predicted in 149.737438 seconds.
dog: 100%
truck: 92%
bicycle: 99%
```

この処理を 10 回行い実行時間の平均をとると 149.86 s であった。

次に FPGA の HLS_accel を使うように、`git clone`したリポジトリ`darknet/`の以下のソースコードを編集した。どのように編集したかも奥畑 (2019)を参照されたい。

- `src/convolutional_layer.c`
- `examples/detector.c`
- `includes/darknet.h`
- `Makefile`

また`dma_simple.c`と`dma_simple.h`を`src/`に配置して、コンパイルし検出処理を実行した。

```
make
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

10 回処理時間の測定を行ったところ平均は 148.15 s となり、処理は 2 s ほど早くなった（図 B.11）。

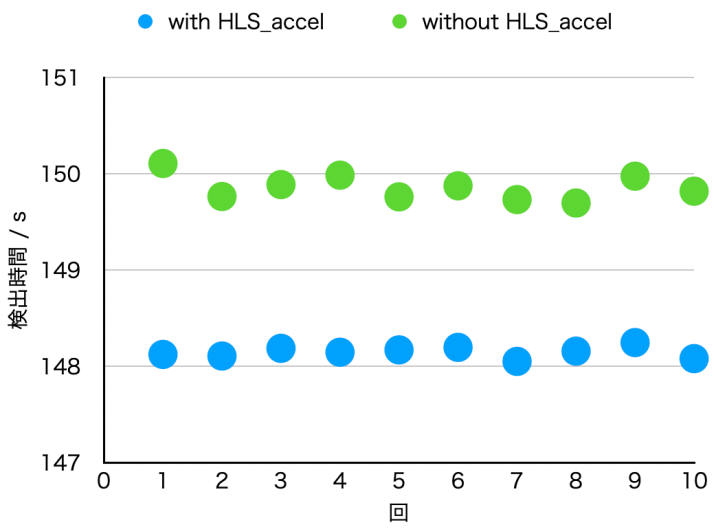


図 B.11　物体検出処理時間の結果。ハードウェアベースの HLS_accel を用いた処理の方が 2 s ほど早くなった。

革新的に高速化ができたとは言えないが、HLS_accel が正しく動作しハードウェアベースの処理ができたと考えられる。さらなる高速化のための手法として挙げられるのは、

- 行列のサイズを 32×32 から 64×64 に大きくする

- クロック周波数（ここでは 100 MHz）を上げる
- データ転送時のデータ幅（ここでは 32 bit）を拡張しサイクル数を減らす

などが考えられる。

付録 C

Linux からのメモリアクセスによるモジュールの操作

Ubuntu ならば`devmem`というコマンドを使うことで、ユーザ空間から任意のメモリにアクセスすることができるが、CentOS 7 は Ubuntu とは系統が異なるためか、そのようなコマンドは存在していないようだった。しかし CentOS 7 にも`/dev/mem`という、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が存在する。これを`open`し`mmap`することで、Ubuntu の`devmem`と同様の機能を持つソフトウェアを作成することができた。その例をコードブロック C.1 に示す。ソースコードの初めに`define`している`PHYS_ADDR`で、図 2.9 の [Address Editor]で割り振った物理アドレスを指定している。

```
#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 0xA0000000

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   = 0xdeadbeef;

    // 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;
}
```

コードブロック C.1　`devmem` と同様の機能を持つ C 言語によるプログラム

たとえばこれを BRAM に対して実行し成功すれば

```
$ gcc -std=c99 -Wall -g -O0 devmem.c -o devmem
$ sudo ./devmem
offset = 0, data=0x0
offset = 0, data=0xdeadbeef
```

のように値を読み書きできた。

付録 D

Linux からの I^2C デバイス操作方法

Zynq MPSoC で走る Linux から I^2C のペリフェラルを操作する方法を確認したので、大事な技術としてここに記しておく。大まかに次のような手順で開発を進める。

1. PetaLinux で I^2C デバイスドライバを有効にする
2. 回路図などから操作したい対象の I^2C デバイスの I^2C スレーブアドレスを確認しておく
3. デバイスツリーに I^2C デバイスの情報を追加する
4. ブートログや `i2c-tools` を用いて、対象の I^2C デバイスのデバイスファイルを特定する
5. 対象のデバイスファイルを `open` して値を読み書きする C 言語のアプリケーションを作成し実行する

1、2、3 については節 D.1 で、4 と 5 についてはそれぞれ次の節 D.2 と D.3 で説明する。

D.1 PetaLinux での準備

1 については

```
petalinux-config -c u-boot
```

で

```
Device Drivers ---> I2C support
```

とナビゲートし、図 D.1 のようにデバイスドライバを有効化する。通常デフォルトで有効化されているので、それを確かめる。

2 の段階において、知っておくべきなのは I^2C MUX と対象の I^2C デバイスのスレーブアドレスである。たとえば Endcap SL に搭載されている Si5345 について、図 3.7 の回路図を見れば、ハードウェアで A0、A1が Low になっていることがわかる。このことと Si5345 のデータシートにより（図 D.2)、スレーブアドレスは `0b1101000` すなわち `0x68` であることがわかる。Endcap SL に搭載されている I^2C MUX についても同様に、A0、A1、A2がすべて Low になっていることから、スレーブアドレスは `0x70` であると決定できる。

3 についてはボード上に搭載されている I^2C デバイスによって書き方は変わるが、Endcap SL 上の I^2C MUX についての記述はコードブロック D.1 のようにした[1]。Endcap SL 上の I^2C MUX は ZCU102 上のものと同じであるため、ZCU102 のデバイスツリーを参照した。ここで I^2C スレーブアドレスを入力しておく。

[1] https://mjmwired.net/kernel/Documentation/devicetree/bindings/clock/silabs,si5341.txt に Si534x のデバイスツリーの書き方が示されているので、これを参考にした。

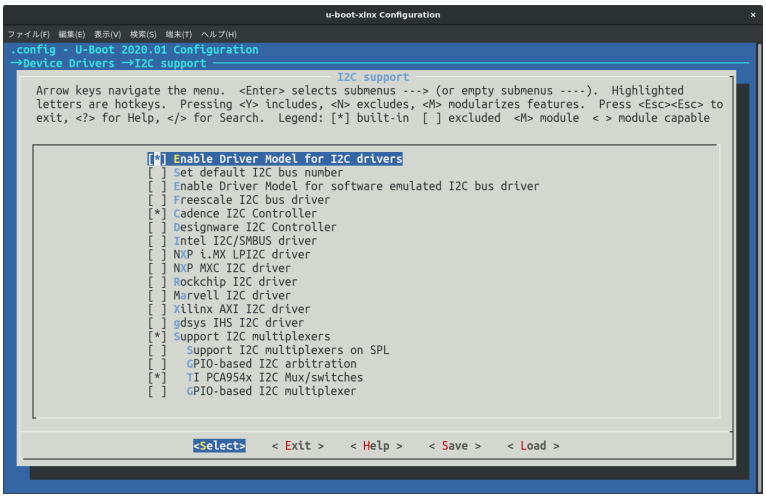


図 D.1　I²C デバイスドライバを追加

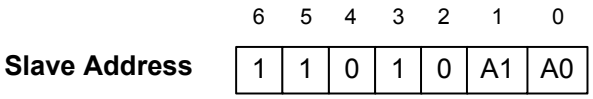


図 D.2　Si5345 の I²C スレーブアドレスはこのように決まっている（Silicon Labs 2018）。

```
&i2c0 {
    i2c-mux@70 { /* U23 */
        compatible = "nxp,pca9548";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0x70>;
        i2c@3 {
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <3>;
            si5345b: clock-generator@68 { /* U1 */
                compatible = "silabs,si5345";
                reg = <0x68>;
                #clock-cells = <2>;
                clocks = <40000000>; // 40 MHz
                clock-names = "in0";
                out@0 { reg = <0>; };
                out@1 { reg = <1>; };
                out@2 { reg = <2>; };
                out@3 { reg = <3>; };
                out@4 { reg = <4>; };
```

```
                out@5 { reg = <5>; };
                out@6 { reg = <6>; };
                out@7 { reg = <7>; };
                out@8 { reg = <8>; };
            };
        };
        i2c@4 {
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <4>;
            si5344b: clock-generator@68 { /* U2 */
                compatible = "silabs,si5344";
                reg = <0x68>;
                #clock-cells = <2>;
                clocks = <40000000>; // 40 MHz
                clock-names = "in0";
                out@0 { reg = <0>; };
                out@1 { reg = <1>; };
                out@2 { reg = <2>; };
                out@3 { reg = <3>; };
            };
        };
        /* port 7 unconnected */
    };
};
```

コードブロック D.1　Endcap SL 上の I²C MUX（図 3.8）についてのデバイスツリーの記述

D.2　ブートログや I²C Tools を用いた I²C デバイスの確認

ブートファイルを作成し CentOS 7 を起動すれば、Linux が I²C デバイスを検出したことをコードブロック D.2 のようにブートログから確認できる。コードブロック D.2 からは 8 チャンネルの I²C MUX が `i2c-0` として検出され、I²C スレーブアドレスが `0x70` であることも確認できた。さらに `/dev/i2c-*` というデバイスファイルが `/dev/i2c-0` から `/dev/i2c-8` まで現れる（コードブロック D.3）。

```
$ dmesg | grep -e i2c -e I2C
[    2.821281] i2c /dev entries driver
[    3.648600] i2c i2c-0: Added multiplexed i2c bus 1
[    3.682426] i2c i2c-0: Added multiplexed i2c bus 2
[    3.693040] i2c i2c-0: Added multiplexed i2c bus 3
[    3.698014] i2c i2c-0: Added multiplexed i2c bus 4
[    3.702981] i2c i2c-0: Added multiplexed i2c bus 5
[    3.707866] i2c i2c-0: Added multiplexed i2c bus 6
```

```
[    3.744817] i2c i2c-0: Added multiplexed i2c bus 7
[    3.749750] i2c i2c-0: Added multiplexed i2c bus 8
[    3.754542] pca954x 0-0070: registered 8 multiplexed busses for I2C switch pca9548
[    3.762121] cdns-i2c ff020000.i2c: 400 kHz mmio ff020000 irq 32
```

コードブロック D.2 ブートログから I²C デバイスが検出されていることの確認

```
$ ls /dev/i2c-*
/dev/i2c-0  /dev/i2c-2  /dev/i2c-4  /dev/i2c-6  /dev/i2c-8
/dev/i2c-1  /dev/i2c-3  /dev/i2c-5  /dev/i2c-7
```

コードブロック D.3 `/dev/i2c-*` というデバイスファイルを 9 つ検出

さて `i2c-tools` はコマンドラインから I²C デバイスにアクセスするためのパッケージである。CentOS 7 には

```
sudo yum install -y i2c-tools
```

を実行すればインストールされるはずである。たとえば `i2cdetect` というコマンドを `-l` オプションとともに使えば OS が検出している I²C デバイスの一覧を確認できる。

```
$ sudo i2cdetect -l
i2c-3   i2c         i2c-0-mux (chan_id 2)               I2C adapter
i2c-1   i2c         i2c-0-mux (chan_id 0)               I2C adapter
i2c-8   i2c         i2c-0-mux (chan_id 7)               I2C adapter
i2c-6   i2c         i2c-0-mux (chan_id 5)               I2C adapter
i2c-4   i2c         i2c-0-mux (chan_id 3)               I2C adapter
i2c-2   i2c         i2c-0-mux (chan_id 1)               I2C adapter
i2c-0   i2c         Cadence I2C at ff020000             I2C adapter
i2c-7   i2c         i2c-0-mux (chan_id 6)               I2C adapter
i2c-5   i2c         i2c-0-mux (chan_id 4)               I2C adapter
```

コードブロック D.4 `i2cdetect -l` で使用可能な I²C デバイスを確認できる

Si5345 の I²C 信号線は I²C MUX のポート 3 から延びているため、コードブロック D.4 の結果から `/dev/i2c-4` がそのデバイスファイルであると考えられる。`i2cdetect -y -r <i2cbus>` を実行すればその I²C デバイスのアドレスをスキャンして、表を出力する。ここで `<i2cbus>` は I²C デバイスに付けられた数字であり、今回の場合は `4` となる。出力結果をコードブロック D.5 に示す。返された表の数字はプローブした結果なんらかの反応があったアドレスを表しており、`UU` は反応があったがドライバなどによって使用されていることを意味している。ここでは `68` は Si534 のスレーブアドレス `0x68` に一致し、`UU` がある `0x70` は I²C MUX のスレーブアドレスに一致する。したがって Si5345 のデバイスファイルは `/dev/i2c-4` であると決定できる。

```
$ sudo i2cdetect -y -r 4
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: UU -- -- -- -- -- -- --
```

コードブロック D.5　`i2cdetect -y -r 4` の出力結果

`i2cdump -y <i2cbus> <address>` を実行すればその I²C デバイスのレジスタの値をダンプして表示してくれる。たとえば `i2cdump -y 4 68` を実行すると結果はコードブロック D.6 のようになり、何らかのデータが入っていることが確認できる。

```
$ i2cdump -y 4 0x68
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 05 00 45 53 01 03 10 07 00 00 00 68 d9 00 0f ff    ?.ES????...h?.?.
10: 00 df ff 2f ff 03 02 dc ff 00 00 f6 00 00 00 00    .?./.???...?....
20: 00 0f 00 fe 0b 00 00 00 00 00 00 02 00 00 00 00    .?.??......?....
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
40: 04 00 00 00 00 0c 00 00 00 00 00 00 00 00 00 00    ?....?..........
50: 0f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ?...............
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
b0: 00 00 00 00 00 00 00 01 00 c3 31 7b 07 00 00 00    .......?.?1{?...
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
e0: 00 00 03 00 00 01 00 00 00 00 00 00 00 00 00 00    ..?..?..........
f0: 00 00 00 00 00 00 0d 03 00 2f ff 00 00 00 0f 00    ......??./....?.
```

コードブロック D.6　`i2cdump` コマンドでレジスタの内容をダンプした結果

D.3　ソフトウェアからの I²C デバイスの操作方法

`/dev/i2c-*` デバイスファイルをインターフェイスとして Linux のユーザスペースから I²C デバイスを操作することができる。これには i2c-dev というカーネルモジュールを追加しておく必要がある。START コンディションや STOP コンディションといった I²C 特有の操作はこのカーネルモジュールが自動的に行ってくれる。

プレーンな I²C 通信をするには、`ioctl()` 関数を `I2C_SLAVE` モードで使用して I²C スレーブアドレスをセットし、`read()` / `write()` 関数を用いてレジスタアドレスやデータを読み書きすればよい。I²C-read/write をする関数の例をそれぞれコードブロック D.7 とコードブロック D.8 に示す。ただしエラー処理は省略している。

```
#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;
}
```

コードブロック D.7 I²C-read 関数の例

```
#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;
}
```

コードブロック D.8　I^{2}C-write 関数の例

これらを用いて、たとえば 0xAA のレジスタの値は

```
uint8_t data = 0;
i2c_read(0xaa, &data);
printf("data 0x%02x\n", data);
```

とすれば取得でき、0xBB のレジスタに 0xBB を書き込みたければ

```
i2c_write(0xaa, 0xbb);
```

とすればよい。

D.4　Si5345 の操作方法（ページの概念）

Si5345 のレジスタアクセスは少し特殊であったため記しておく。Si5345 のレジスタアドレスは 16 bit であるが、その上位 8 bit は「ページ」を表している。つまりたとえばあるレジスタのアドレスが 0x051A ならば、「ページ 0x05 の 0x1A 番目のレジスタ」と解釈する。

各ページの 0x01 番目のレジスタは PAGE register と呼ばれており、最初にここにアクセスしたいレジスタのページを書き込む必要がある。たとえばアドレス 0x051A のレジスタを I^{2}C-read する場合、

1. （現在どのページにいるかはわからないが）アドレス 0x01 のレジスタに 0x05 を書く（ページ 0x05 に移動）
2. アドレス 0x1A のレジスタを読む

のように手順を踏む。同様にアドレス 0x051A のレジスタに 0x03 というデータを I^{2}C-write したいならば、

1. （現在どのページにいるかはわからないが）アドレス 0x01 のレジスタに 0x05 を書く（ページ 0x05 に移動）
2. アドレス 0x1A のレジスタに 0x03 というデータを書く

という手順を踏む必要がある。

付録 E

XVC 開発の詳細

E.1　PetaLinux での操作

Linux が Userspace I/O（UIO）として Debug Bridge を検出できるよう、デバイスツリーに以下を追加した。ここの `debug_bridge_0` というのは図 3.10 のブロックデザイン内の名前と同じにしなくてはならない。

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

コードブロック E.1　`debug_bridge_0` の情報をデバイスツリーに追加

また UIO を使えるように

```
petalinux-config -c kernel
```

において

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

と進み図 E.1 のようにデバイスドライバを追加する。さらに

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

をアンチェックする。

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

において、Kernel bootargs に `uio_pdrv_genirq.of_id=generic-uio cpuidle.off=1` を追加する。もし `uEnv.txt` を使って bootargs を設定しているならば、そこにも追加しておく。

こうして `petalinux-build` を行う。

CentOS 7 を起動したら、UIO に `debug_bridge` に割り当てられていることを確認する（コードブロック E.2）。

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

コードブロック E.2　CentOS 7 から `debug_bridge` が `/dev/uio0` に割り当てられている様子

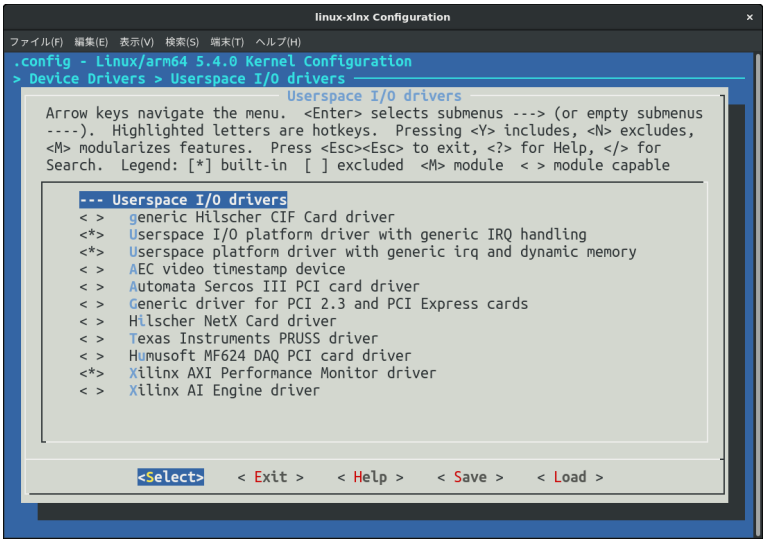


図 E.1　UIO を使うためのデバイスドライバを追加

E.2　XVC サーバアプリケーションの開発

もし参照のチュートリアル（anunesgu 2021）のように PetaLinux プロジェクトで XVC サーバアプリケーションをコンパイルしたならば、それは PetaLinux が生成する rootfs に組み込まれる。すなわち実行ファイルが `/usr/bin/` に置かれた状態で生成される。しかし今回は Zynq MPSoC にアクセスできる状態であり、PetaLinux の rootfs ではなく CentOS 7 のものを使用している。そこで PetaLinux の段階では XVC サーバアプリケーションはコンパイルせず、Zynq MPSoC 上の CentOS 7 でコンパイルした。完成したら実行ファイルを `/usr/bin/` に置けば、PetaLinux が生成する rootfs と同じくコマンドとして利用できるようになる。

XVC サーバアプリケーションのソースコードは、Xilinx 社が GitHub で提供しているもの（Xilinx 2021a）を利用した。それにデバッグオプションや使い方のメッセージ（コードブロック E.3）を表示するよう変更した。ただしソースコード内で `debug_bridge` が `/dev/uio0` に割り当てられていることをハードコードしている。`xvcserver.c` を Zynq MPSoC の CentOS 7 に転送し、そこでデバッグやコンパイルを行った。

```
$ sudo ./xvcserver0

Xilinx Virtual Cable server starting.

---- Instruction ----
0. Make sure you're going to open /dev/uio0. If needed,
   change the variable UIO_PATH in the source code.
1. Check my IP address first.
2. Open Hardware Manager in Vivado in your computer.
3. "Open Target" > "Open New Target" > "Local server"
   > "Add Xilinx Virtual Cable".
4. Enter hostname: <my IP address>, port: 2542.
5. Enjoy!
```

```
----------------------
```

コードブロック E.3　XVC サーバ起動時にはインストラクションメッセージを表示させるようにした

付録 F

回路図

ここでは説明上必要だった回路図を掲載しておく。

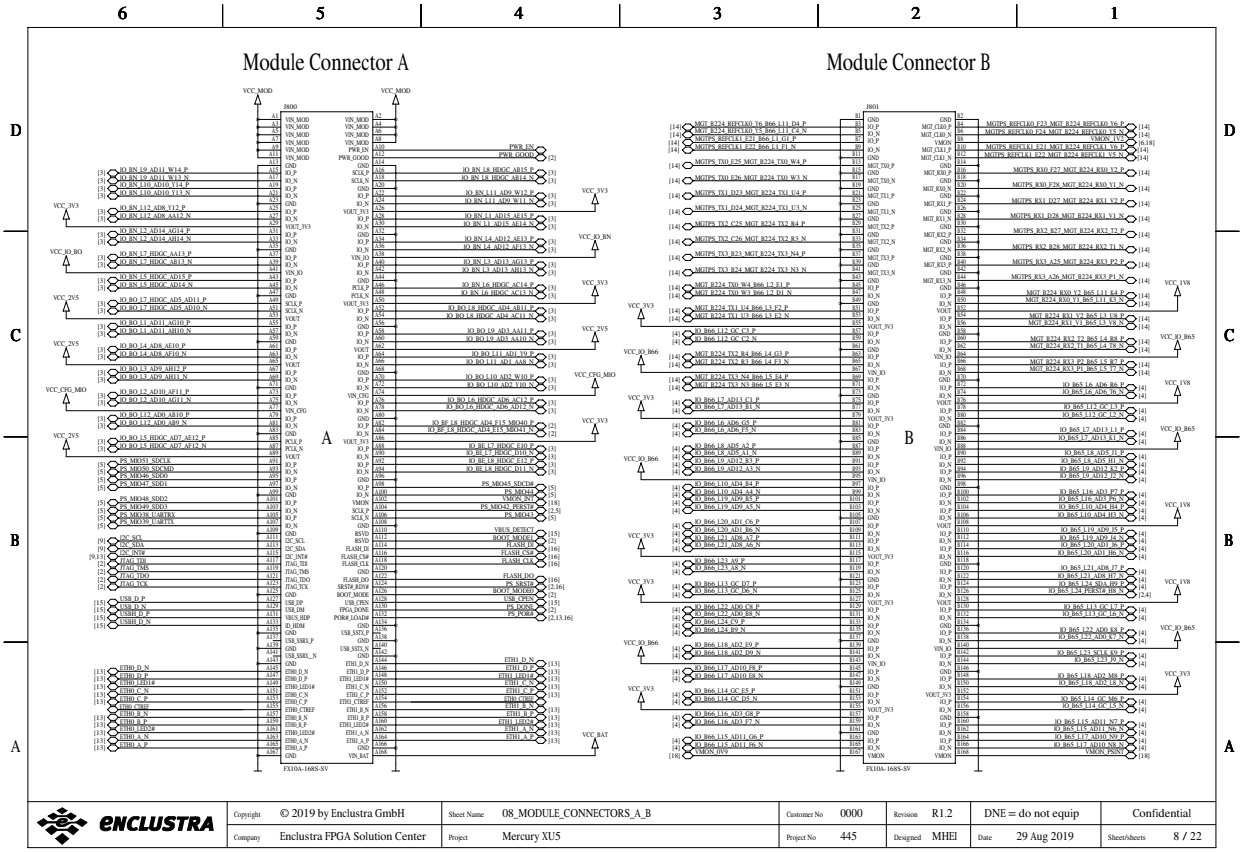


図 F.1　Mercury XU5 のコネクタ A と B

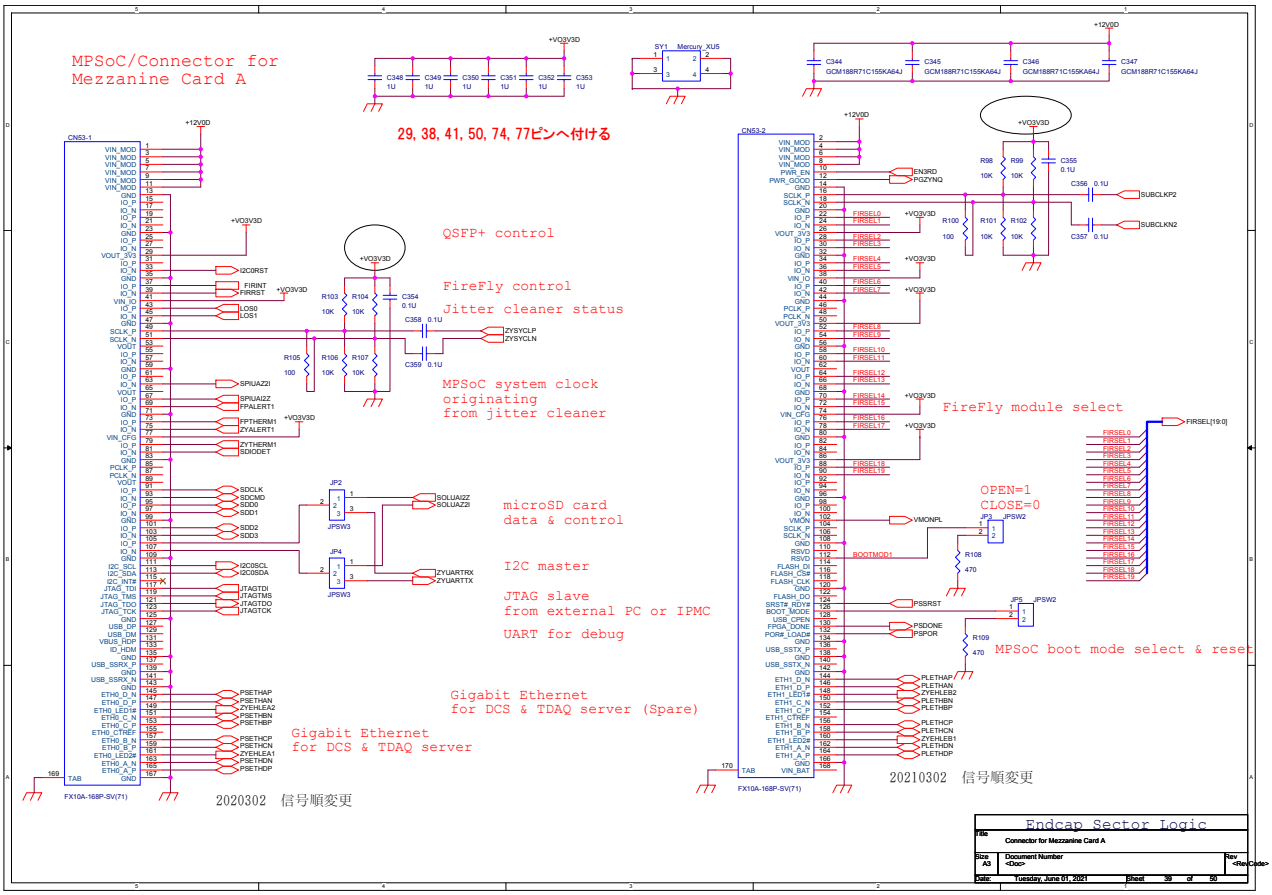


図 F.2 Endcap SL 上の Mercury XU5 とのコネクタ A

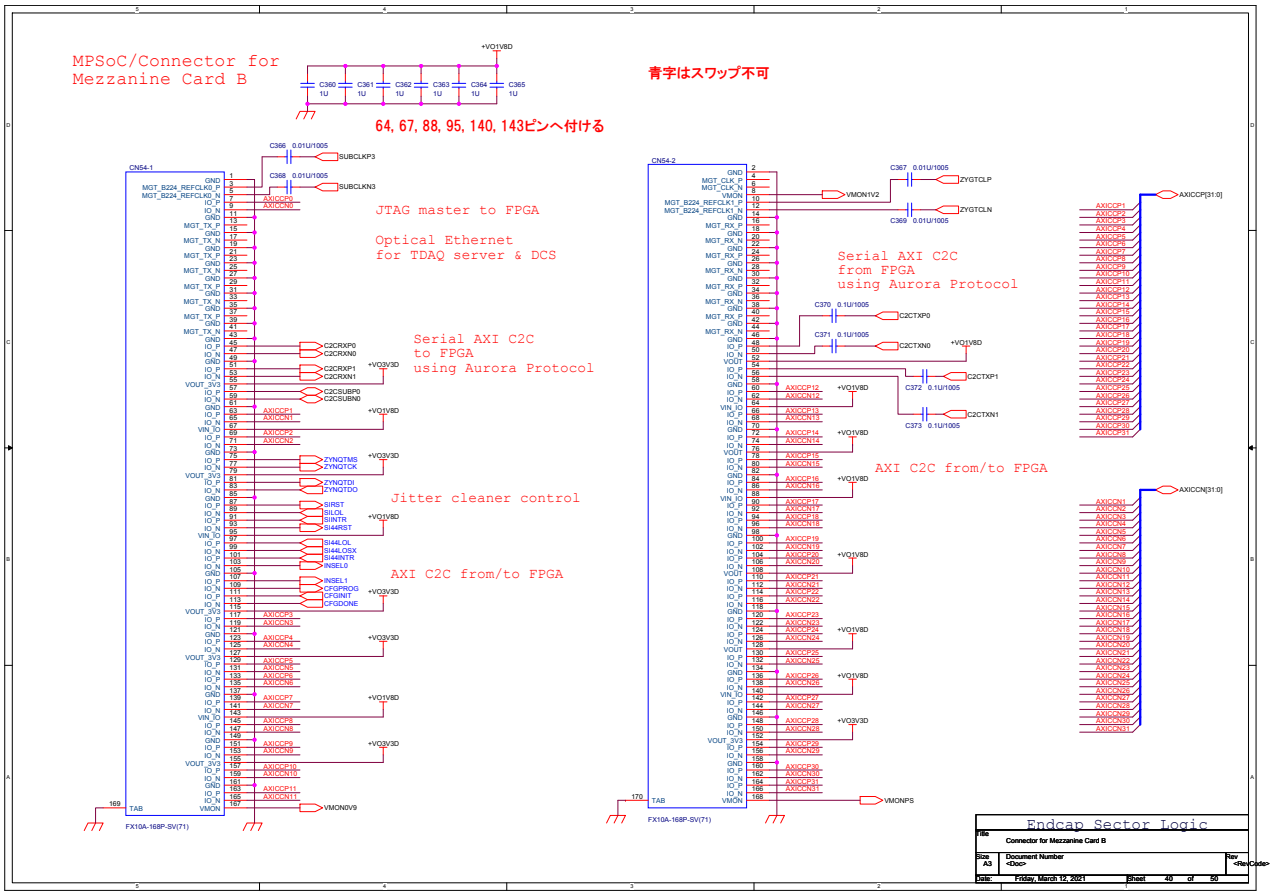


図 F.3　Endcap SL 上の Mercury XU5 とのコネクタ B

付録 G

Endcap SL 第 1 試作機で使用したファームウェアのブロックデザイン

Endcap SL に実装したファームウェアのブロックデザインを掲載しておく。

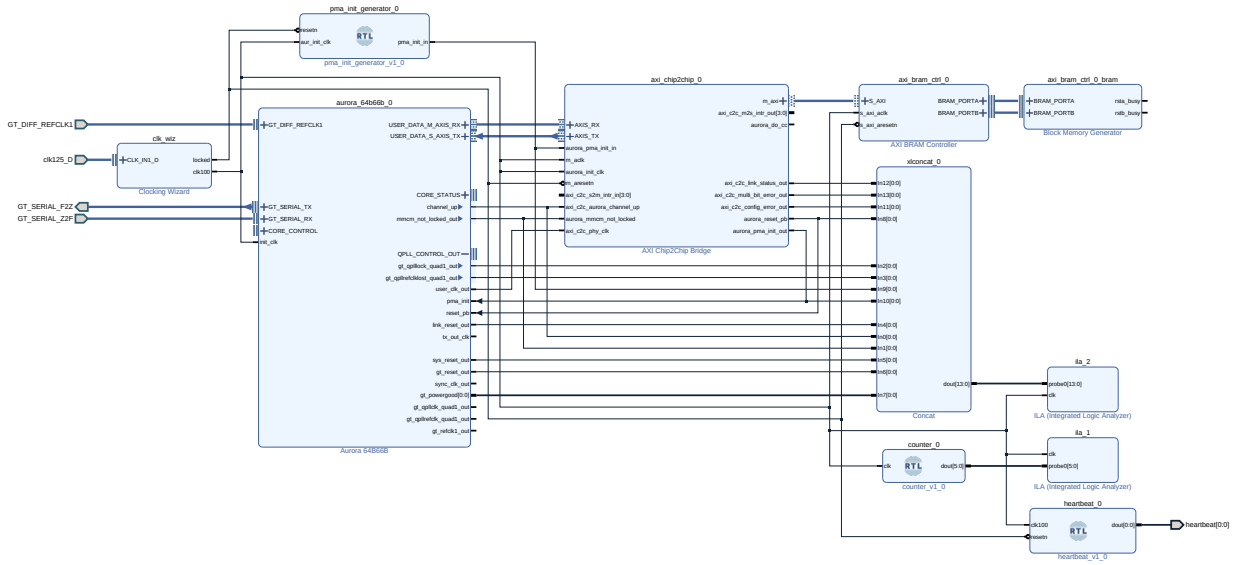


図 G.1　XCVU13P 側のブロックデザイン（2022 年 1 月 28 日現在）

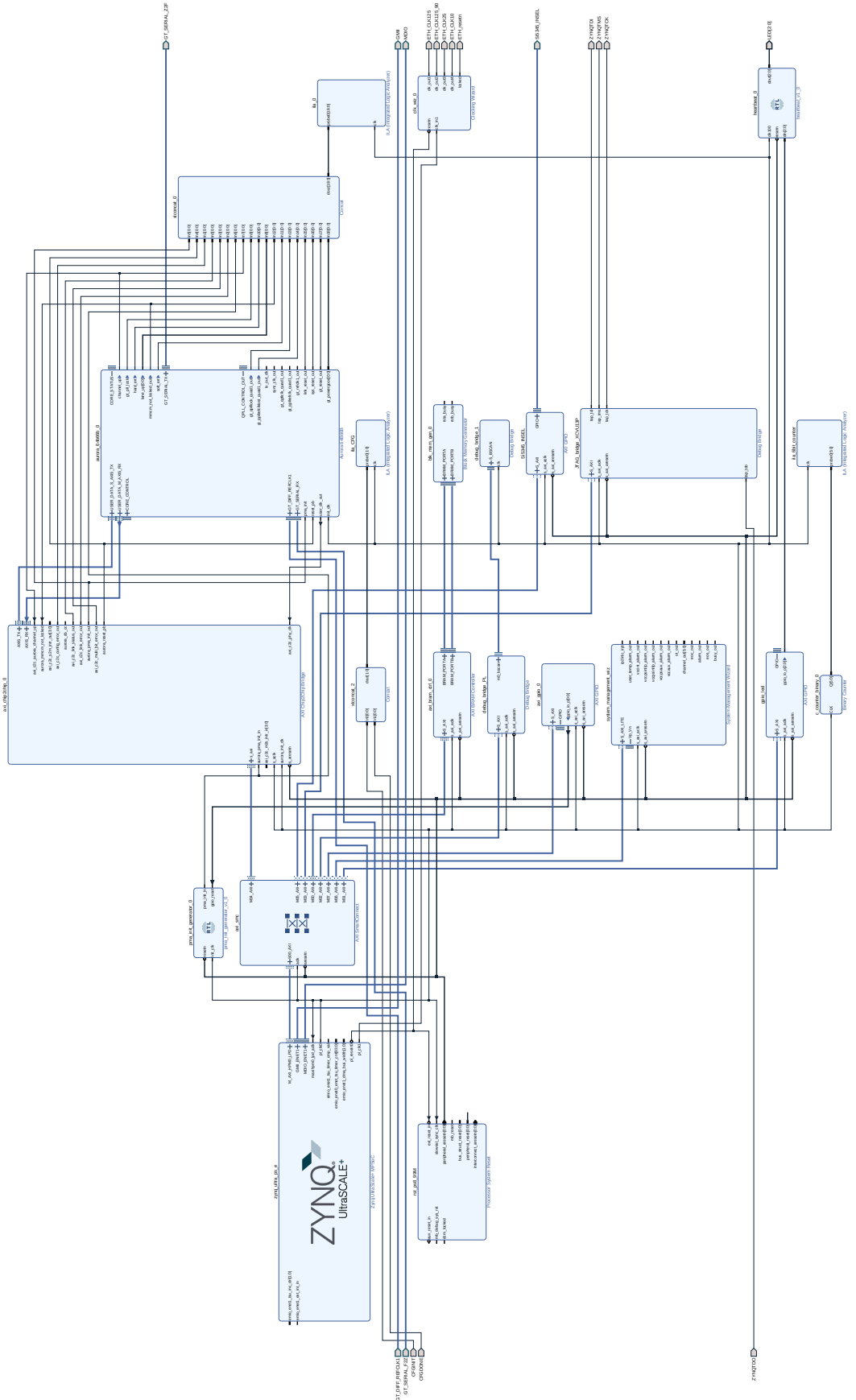


図 G.2 Zynq MPSoC 側のブロックデザイン（2022 年 1 月 28 日現在）。RGMII-GMII convertor はこれの外側にある。

付録 H

AXI4 プロトコル

AXI Chip2Chip を用いたファームウェアを開発するのにあたって、AXI4 プロトコルを理解することが必要であった。ここでは簡単にその説明をする。

Advanced eXtensible Interface 4（AXI4）は ARM® 社による第 4 世代の Advanced Microcontroller Bus Architecture（AMBA®）インターフェースプロトコルの 1 つである。基本的にマスターとスレーブを 1 対 1 で接続する規格だが、図 H.1 のように[1] interconnect を使えば複数のマスターとスレーブに拡張できる。たとえば Vivado で使用可能な AXI SmartConnect IP は最大 16 対 16 のマスターとスレーブの接続をサポートしている。

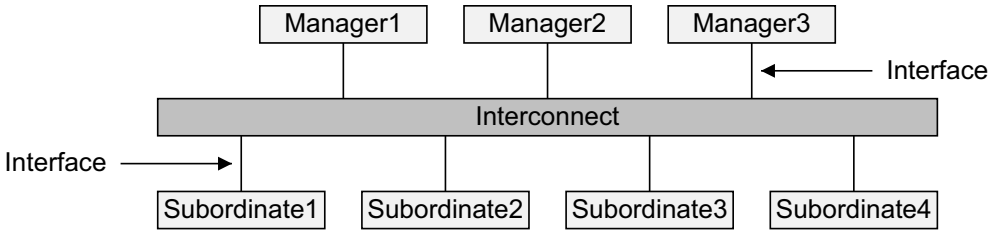


図 H.1　Interconnect を介した複数のマスターとスレーブの接続（ARM 2021）

AXI4 は AXI4-Full、AXI4-Lite、AXI4-Stream の 3 種類のプロトコルの集合である。AXI4-Full と AXI4-Lite は memory-mapped プロトコル（transaction 中にメモリ空間のアドレスの情報を含む）だが、AXI4-Stream はそうではない。それぞれ次のような特徴を持つ。

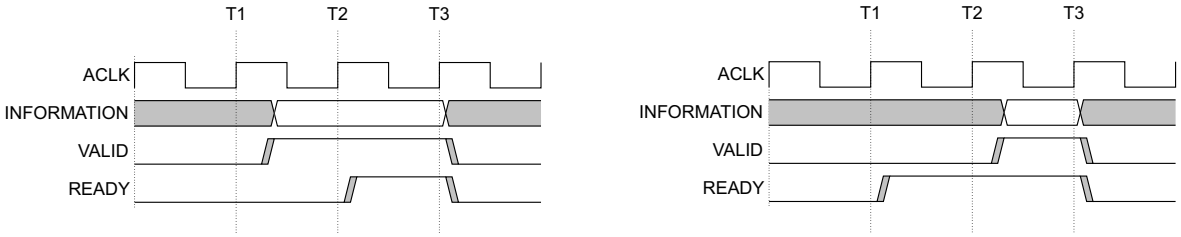
AXI4-Full	バースト転送を基調とした高いスループットを持つ memory-mapped プロトコル
AXI4-Lite	バースト転送はできないものの軽量な memory-mapped プロトコル
AXI4-Stream	高速にデータを流す軽量なプロトコル

AXI4-Full のことを単に AXI4 と呼ぶことが非常に多く、本論文でもそのように言葉を使っている。
なお単一のデータの転送を transfer とし、複数のデータ転送のことを transaction と呼ぶように定義しておく。

H.1　基本的なコンセプト

グローバルシグナルとしてクロック線 ACLKとリセット線 ARESETnがある。接頭辞の Aは AXI プロトコルに関連していることを示す。また ARESETnの接尾辞 nは active-low であることを表している。

[1] ARM (2021)ではバージョン H.b から master/slave という語の代わりに manager/subordinate という単語を使うようになった。



(a) VALID before READY　　(b) READY before VALID

図 H.2　AXI4 におけるハンドシェイク（ARM 2021）

すべてのプロトコルには共通して VALID、READY、そして何らかの情報を示すバス（ここではその信号名を仮に INFORMATIONとする）の信号線がある。マスターは INFORMATIONを送信する際、または有効な情報が含まれている間ずっと VALIDをアサートし、スレーブにそのことを伝える。ただし ARESETnがディアサートされている時は、マスターは VALIDをアサートすることは許されていない。またスレーブは情報を受け取る余裕がある間 READYをアサートする。すなわちマスターとスレーブ間で transfer が行われるのは ACLKの立ち上がりで VALIDと READYがともにアサートされている時である。両者はそれをもって transfer が完了したことを知る。

このようなハンドシェイクの例を図 H.2 に示した。図 H.2(a) ではマスターが先に INFORMATIONを準備し VALIDを立てている。図中の T2 の時点ではスレーブは READYを立てていないため、transfer は行われず、マスターはその出力状態を維持しなければならない。T2 の直後でスレーブが READYをアサートしているが ACLKの立ち上がりが来るまで transfer は完了しない。T3 で VALIDと READYがどちらもアサートされているため、ここの時点で transfer が完了する。一方図 H.2(b) ではスレーブが待機状態であるため READYを最初に立てている。図 H.2(a) と同様に T3 で transfer が完了する。

AXI4 ではマスターは一度 VALIDをアサートしたら transfer が完了するまでそれを維持しなければならず、また INFORMATIONの値も変更してはならない。ただしスレーブは READYをアサートしても、VALIDがディアサートされている状態であればそのアサートを待たずにディアサートできる。

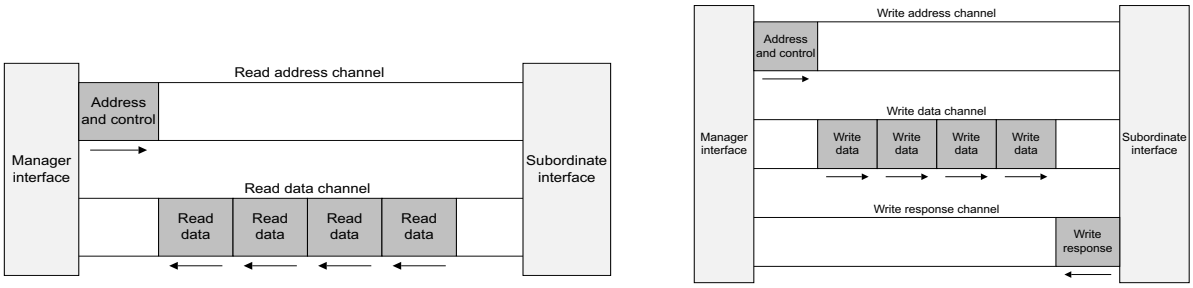
さらに AXI4 では、マスターは VALIDを立てるのに READYを待ってはいけないという約束がある。このことを統一しないと両者は互いの信号がアサートされるまで待ち合うというデッドロックの問題が発生するからである。ただしスレーブは VALIDのアサートを待って READYを立てても構わない。

H.2　AXI4-Stream

AXI4-Stream は AXI4 プロトコルのうち最もシンプルなプロトコルである。各信号線名には AXI4-Stream であることを示すため、TVALIDや TREADYのように接頭辞 Tをつける。アドレスといった概念が必要でなく、マスターからスレーブに一方的にデータを高速に流すのに AXI4-Stream が用いられる。

AXI Chip2Chip の実装では AXI Chip2Chip と Aurora 間の通信で送信用と受信用で 2 つの AXI4-Stream チャンネルが使われる。その信号は TVALID、TREADY、TDATAから構成されていた。TDATAは多ビットの信号線である。それらの transfer の行われ方は節 H.1 で説明した通りである。

ただし AXI4-Stream で必須の信号線は TVALIDで、その他はオプションであるという特徴がある。たとえば Aurora のデータ送信用チャンネルでは TVALIDと TDATAのみで TREADYは省略されている。これは Aurora の AXI4-Stream のスレーブとなるモジュールがすべての transfer を受け取れる前提であるか、AXI4-Stream マスターとしての Aurora がバックプレッシャを意図的に受け取らないようにしていると考えられる。また TDATAを省略し



(a) 読み出し　　(b) 書き込み

図 H.3　AXI4 の transaction（ARM 2021）

た場合は TKEEPといった別の信号が必要になるが、AXI Chip2Chip と Aurora の実装では TDATAを省略することはないため、詳細は ARM (2010)に譲る。

H.3　AXI4-Lite と AXI4

AXI4-Lite と AXI4-Full は memory-mapped プロトコルである。すなわち transaction にはターゲットのアドレスの情報が含まれる。

H.3.1　チャンネル

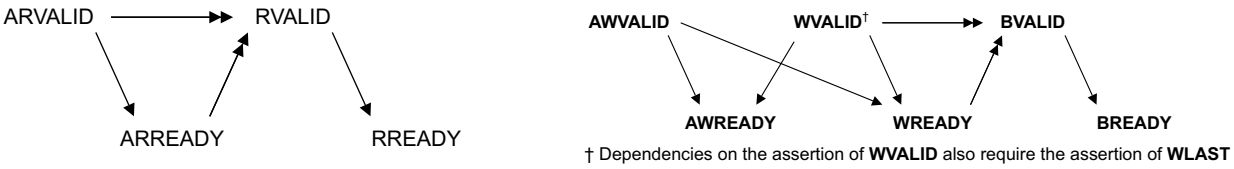
AXI4-Lite と AXI4 では信号線を役割ごとにグループ化したチャンネルという概念を持っている。チャンネルは 5 種類ある。各チャンネルはそれぞれ独自に VALIDと READYの信号を持っているため、ある程度互いに独立して transaction を行うことができる。5 チャンネルを一覧にしたものが表 H.1 である。

表 H.1　AXI4-Lite と AXI4 で使われる 5 つのチャンネル

Channel	Signal prefix	Data source	日本語での呼ばれ方
Read address channel	AR	Master	読み出しアドレスチャンネル
Read data channel	R	Slave	読み出しデータチャンネル
Write address channel	AW	Master	書き込みアドレスチャンネル
Write data channel	W	Master	書き込みデータチャンネル
Write response channel	B	Slave	書き込み応答チャンネル

AXI4 におけるそれぞれのチャンネルの役割を示したのが図 H.3 である。図 H.3(a) のようにデータの読み出し時はマスターが ARチャンネルを使ってターゲットのアドレスをスレーブに伝える。スレーブはそれに対応したデータを Rチャンネルを用いてマスターに返す。これには読み出しデータの他に読み出しの成否の情報（読み出し応答、read response）も含まれる。ただし図 H.3 では AXI4 に特有なバースト転送（burst transaction）が行われているため、1 つの読み出しアドレスの transfer に対して複数のデータが返送されている。バースト転送については節 H.3.3 で後述する。図 H.3(b) は書き込み transaction の構造を表している。マスターは AWチャンネルと Wチャンネルを使ってそれぞれ書き込む対象のアドレスとデータをスレーブに送信する。スレーブはその成否を Bチャンネルを用いてマスターに返す。

スレーブはマスターから読み出しアドレスを受け取らないと、読み出しデータと読み出し応答の transaction を開始できない。同様に書き込みアドレスと書き込みデータを受け取らないと、書き込み応答の transaction を開始でき



(a) 読み出し (b) 書き込み

図 H.4 各チャンネル間のハンドシェイクの依存性(ARM 2021)。X→Yは、Xは Yを待ってのアサートをしてはならないが、Yは Xを待たずにアサートしても構わないことを表している。X⇒Yは、Yは Xを待ってアサートしなければならないことを表している。

ない。このことと、節 H.1 で説明した VALIDがアサートされるのに READYのアサートを待ってはいけないというルールから、各チャンネルの VALIDと READYの間には図 H.4 のような時間的依存性が発生することが理解できる。

H.3.2 信号線

AXI4 と AXI4-Lite の主要な信号線を表 H.2 にまとめた。ただし必須の信号線とオプションの信号線は区別しておらず、AXI Chip2Chip のサンプルデザインで使われていたものをまとめている。AXI4 と AXI4-Lite に共通して存在する信号は **Common**、AXI4 のみに存在する信号は AXI4 only と書き込んでいる。このように AXI4 は AXI4-Lite を包含している関係にある。どちらにも存在しない信号は–と書き込んでいる。たとえば ARチャンネルは読み出しアドレスをマスターからスレーブに伝えるためのものなので、ARDATAという信号は存在しない。ここで AXI4 only と書かれている信号はほとんど節 H.3.3 で後述するバースト転送に関わるものである。

表 H.2 AXI4 と AXI4-Lite の信号線

Channel / Signal	Write address AW-	Read address AR-	Write data W-	Write response B-	Read data R-
-VALID	**Common**	**Common**	**Common**	**Common**	**Common**
-READY	**Common**	**Common**	**Common**	**Common**	**Common**
-ID	AXI4 only	AXI4 only	–	AXI4 only	AXI4 only
-USER	AXI4 only	AXI4 only	AXI4 only	AXI4 only	AXI4 only
-ADDR	**Common**	**Common**	–	–	–
-PROT	**Common**	**Common**	–	–	–
-LEN	AXI4 only	AXI4 only	–	–	–
-SIZE	AXI4 only	AXI4 only	–	–	–
-BURST	AXI4 only	AXI4 only	–	–	–
-DATA	–	–	**Common**	–	**Common**
-STRB	–	–	**Common**	–	–
-LAST	–	–	AXI4 only	–	AXI4 only
-RESP	–	–	–	**Common**	**Common**

表 H.2 で **Common** となっている信号について簡単に説明する。すなわち AXI4-Lite の説明でもある。VALIDと READYは 5 つのチャンネルすべてで用いられている。ARADDRと AWADDRはターゲットのアドレスを転送す

るための信号である。幅 3 bit の ARPROTと AWPROTはプロテクションタイプをマスターが指定するためのもので、Vivado では値 `0b000` が推奨されている。WDATAについてはマスターが送信する書き込みデータを、RDATAはスレーブが送り返す読み出しデータを表す。WSTRBはストローブ信号で、マスターが有効なバイトレーンを指示する。すなわち WSTRB[n]が 1 ならば WDATA[(8n+7):(8n)]が有効であることを意味する。WDATAの幅が 32 bit ならば WSTRB幅はその 1/8 倍の 4 bit である。BRESPと RRESTはそれぞれ書き込みと読み出しの成否をスレーブがマスターに伝えるものである。読み出し transaction についてはスレーブが VALIDをドライブするため読み出しデータと一緒に読み出し応答をマスターに送ることができる。しかし書き込みの transaction についてはアドレスとデータのチャンネルどちらも VALIDはマスターがハンドルしているため書き込み応答専用のチャンネルが必要である。BRESPと RRESTはどちらも幅 2 の信号で、通常アクセスしたら `0b00` となる。上位 1 bit が `0b1` であればエラーを表す。詳細は ARM (2021)を参照のこと。

H.3.3　バースト転送

AXI4-Full はバースト転送をベースとしたプロトコルである。マスターは制御情報と最初のアドレスだけを指定し、そのあとのアドレスはスレーブ側が責任を持つ。

図 H.5 に読み出し transaction のバースト転送の例を示した。ただし ACLKや VALIDなどは省略している。1 transfer は 1 つのデータについての transaction であるとすると、図 H.5 では 4 つの transfer を 1 burst としている。このように 1 burst あたりいくつの transfer を含んでいるかを burst length と呼ぶ。また 1 回の transfer でのデータサイズを burst size といい、この場合 4 B である。

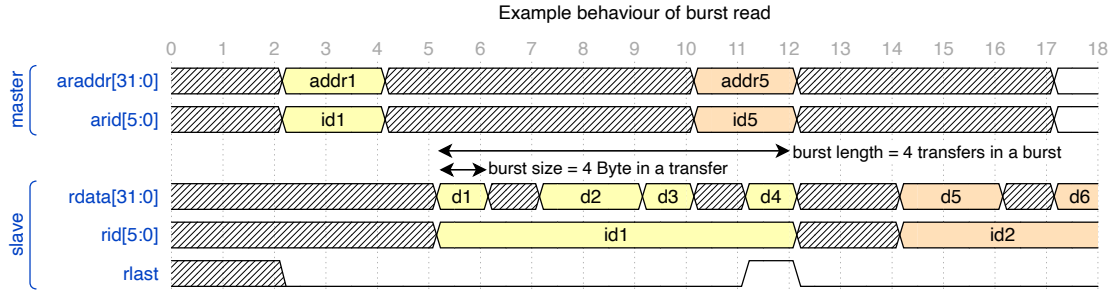


図 H.5　AXI4 burst read

図 H.5 で、マスターは ARチャンネルを使ってアドレス addr1 と transaction ID id1 をスレーブに渡している。スレーブは addr1 から始まる 4 つのアドレスのデータを連続してマスターに伝えている。その際マスターから渡された transaction ID を RID信号を使って付している。最後に burst の終わりを示す RLASTをアサートしてバースト転送が完了する。

どのようにアドレスをインクリメントさせるかや、burst length や burst size の値はマスターが専用の信号を使ってスレーブに指示する。AxLEN[7:0]は burst length を、また AxSIZE[2:0]は burst size をそれぞれ次のように指定するものである（ただし xは Rまたは W）。

$$(\text{burst length}) = 1 + \text{AxLEN[7:0]},$$
$$\log_2 (\text{burst size}) = \text{AxSIZE[2:0]}.$$

またバースト開始時の最初のアドレス（starting address）は

$$(\text{starting address}) = \text{AxADDR}$$

となる。AxBRST[1:0]はバーストタイプを指定するためのものである。バーストタイプは 3 種類あり、それぞれ次のようになっている。

FIXED 同じアドレスにアクセスし続ける。`0b00` で指定する。
INCR アドレスをインクリメントする。ただしインクリメントの幅は burst size である。`0b01` で指定する。
WRAP INCR バーストとほぼ同様だが、upper wrap boundary に達したら lower wrap boundary に戻る。`0b10` で指定する。

である。

バーストタイプが FIXED の場合、i 回めの transfer のアドレスはすべて

$$\text{addr}_i = (\text{starting address})$$

である。バーストタイプが INCR または WRAP の場合、

$$(\text{aligned address}) = \left\lfloor \frac{(\text{starting address})}{(\text{burst size})} \right\rfloor \times (\text{burst size})$$

として、

$$\text{addr}_i = \begin{cases} (\text{starting address}) & \text{for } i = 1, \\ (\text{aligned address}) + i \times (\text{burst size}) & \text{for } i \geq 2 \end{cases}$$

となる。特に WRAP バーストの場合、

$$\begin{aligned} (\text{transaction size}) &= (\text{burst length}) \times (\text{burst size}), \\ (\text{lower wrap boundary}) &= \left\lfloor \frac{(\text{starting address})}{(\text{transaction size})} \right\rfloor \times (\text{transaction size}), \\ (\text{upper wrap boundary}) &= (\text{lower wrap boundary}) + (\text{transaction size}) \end{aligned}$$

として、$\text{addr}_i \geq (\text{upper wrap boundary})$ となるならば $\text{addr}_i = (\text{lower wrap boundary})$ とする。

次のような注意点がある。

- transaction size は 4 kB を超えないようにしなければならない。
- INCR バーストについては burst length は最大 256 まで取れるが、その他のバーストタイプについては最大 16 までである。特に WRAP バーストについては、burst length は 2、4、8、16 のいずれかでなければならない。

-USERはユーザが定義した信号として使うことができる。

WLASTと RLASTは burst の最後の transfer であることを示すのに使われる。

ARID、AWID、RID、BIDを使って各バーストに transaction ID を付けることによって、マスターとスレーブでどのデータがどのバーストに関連しているかを共有することができる。ただし WDATAの順序は AWADDRと同じであるため、WIDはその冗長性により AXI4 から廃止されている。AXI3 と AXI4 のインターフェイスを接続する場合は WIDの代替として WUSERを使えばよい。transaction ID によりマスターはスレーブからの応答を待たずに次のバーストを開始できたり、準備のできたデータから処理を行うようなアウトオブオーダー転送も可能である。たとえばマスターが id1、id2、id3 の順番でバーストを開始したとき、スレーブは id1、id3、id2 のように応答しても両者は問題なくデータとアドレスを取り扱える。これにより処理の時間短縮も可能だと考えられる。

その他のオプションの信号線については ARM (2021)を参照のこと。

最後に読み出し、書き込みバースト転送の例をそれぞれ図 H.6 と図 H.7 に示す。

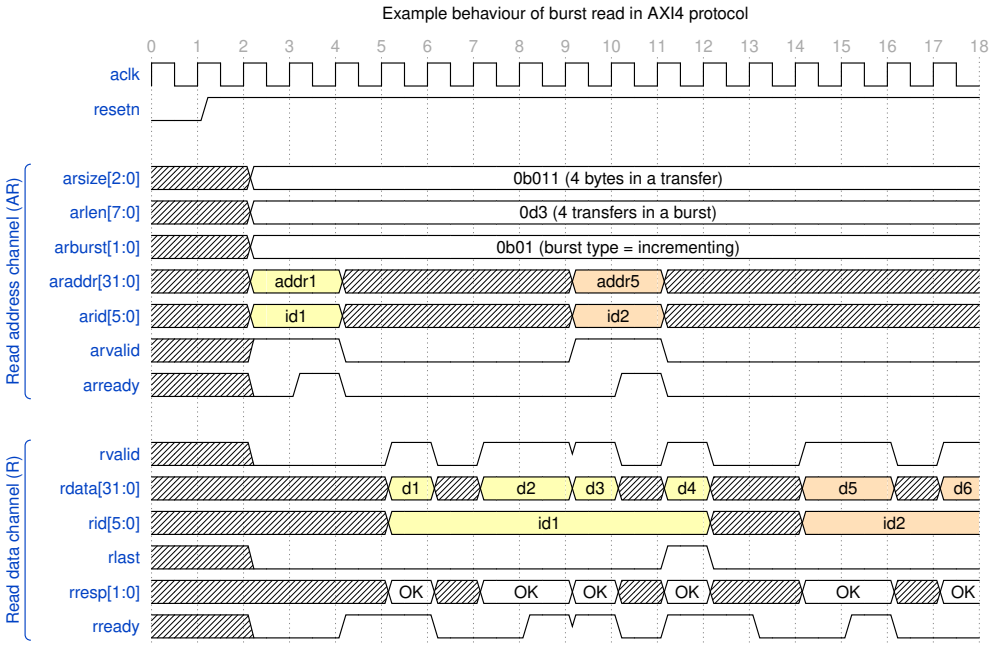


図 H.6　読み出しバースト転送の例

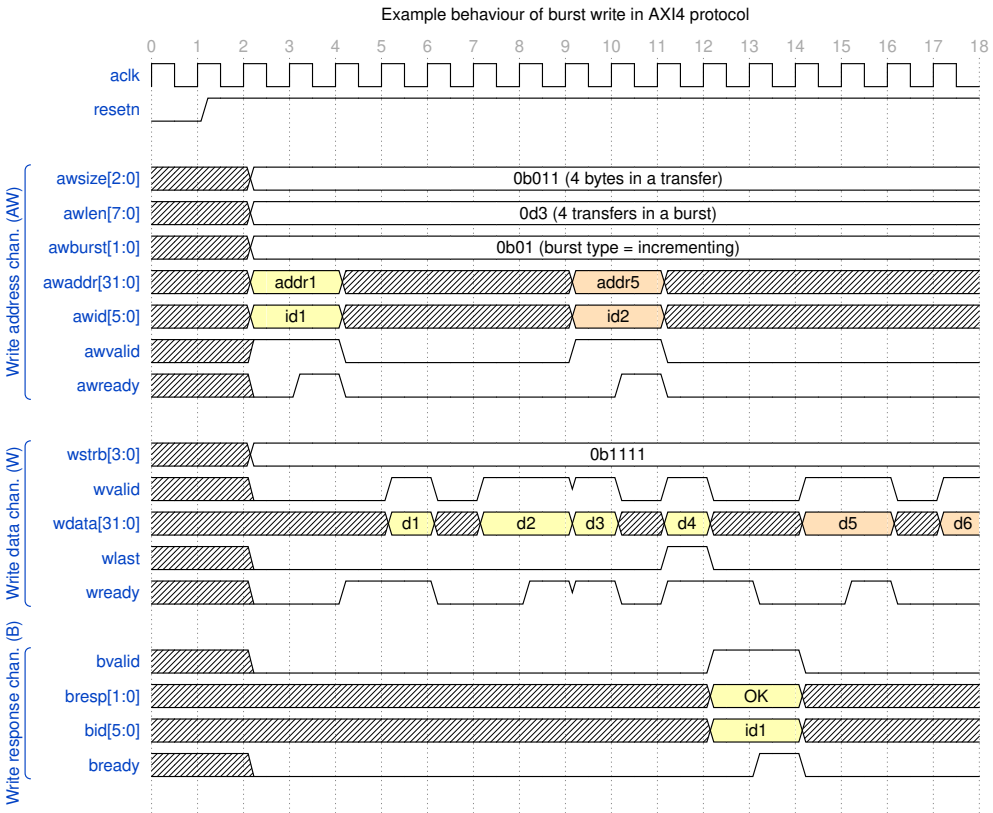


図 H.7　書き込みバースト転送の例

付録 I

AXI Chip2Chip のテストベンチシミュレーションの実行

Vivado では AXI Chip2Chip のサンプルデザインが利用可能である。そのサンプルデザインは図 I.1 のようになっており、AXI Chip2Chip と Aurora のインスタンスと Traffic Generator と Traffic Checker というモジュールが使われている。SelectIO または Aurora の選択や、Aurora 8B/10B または Aurora 64B/66B の選択などは、サンプルデザインを生成するときに選択できる。ここではインターフェイスとして Aurora 8B/10B を用いた。Traffic Generator は適当なデータパターンを生成し、ARチャンネル、AWチャンネル、Wチャンネルでデータを AXI Chip2Chip に送信する。Traffic Checker は同じ規則で参照データを生成し、Rチャンネルと Bチャンネルで AXI Chip2Chip から返された応答が合致しているかを検知するものである。

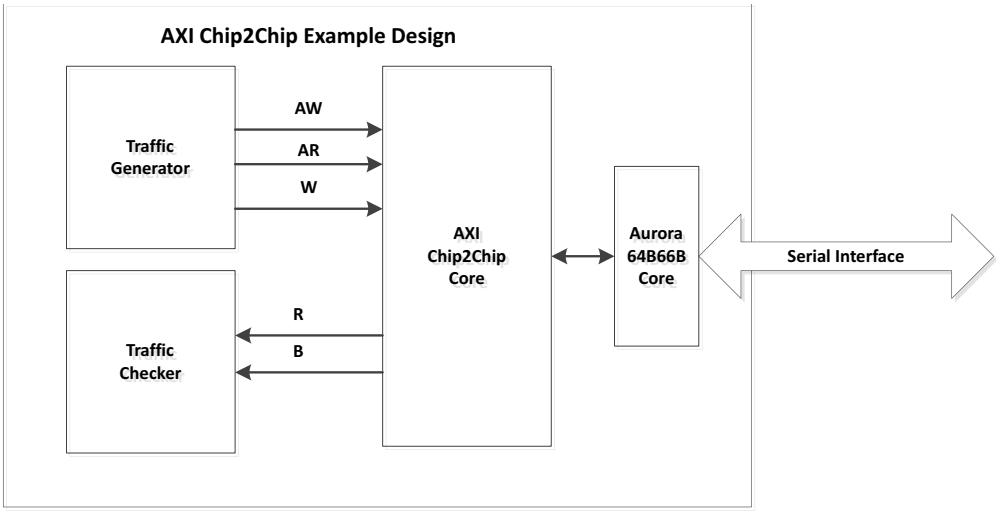


図 I.1　AXI Chip2Chip のサンプルデザイン（Xilinx 2020）

さらにサンプルデザインには図 4.8 のテストベンチシミュレーションが含まれている。マスター側にはサンプルデザインが、スレーブ側ではラッパーをかけていないサンプルデザインが用いられている。このテストベンチシミュレーションの動作は、まずマスターとスレーブ間のリンクの確立を確認したのち両者の間でデータ転送を行い、期待されているデータが送られてきたことを Traffic Checker が判別して、その結果を出力するというものであった。

提供されているテストベンチシミュレーションはそのままでは動作せず、若干のデバッグ[1] をすることでシミュレーションが完了した。さらにマスター側の Traffic Generator を編集し、すべての WID[5:0]で書き込みデータ WDATA[31:0]が `0xDEADBEEF` になるようにしたところ、スレーブ側の Traffic Checker でもそれが見えていることを確認した（図 I.2）。またスレーブ側の Traffic Generator で RID[5:0]が `0x39` のときに読み出しデータ RDATA[31:0]が `0xDEADBEEF` になるようにしたところ、これもマスター側の Traffic Checker で見えていることを確認した（図 I.3）。

図 I.2　マスターが書き込むデータをすべて `0xDEADBEEF` にしたとき、スレーブがそれを受け取る様子

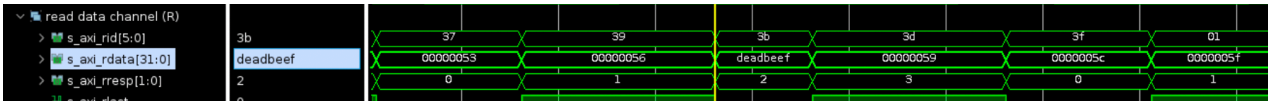


図 I.3　RIDが `0x39` のときマスターがデータ `0xDEADBEEF` を受け取る様子

[1] スレーブ側の `aurora_8b10b_0_partner` というモジュールではなく、マスター側に合わせて `aurora_8b10b_0` を使うようにした。

付録 J

ZCU102 評価ボードを用いたループバック試験

図 4.9 のファームウェアを実機で試験するため、図 J.1 のように ZCU102 評価ボードでループバック試験をすることを考えた。

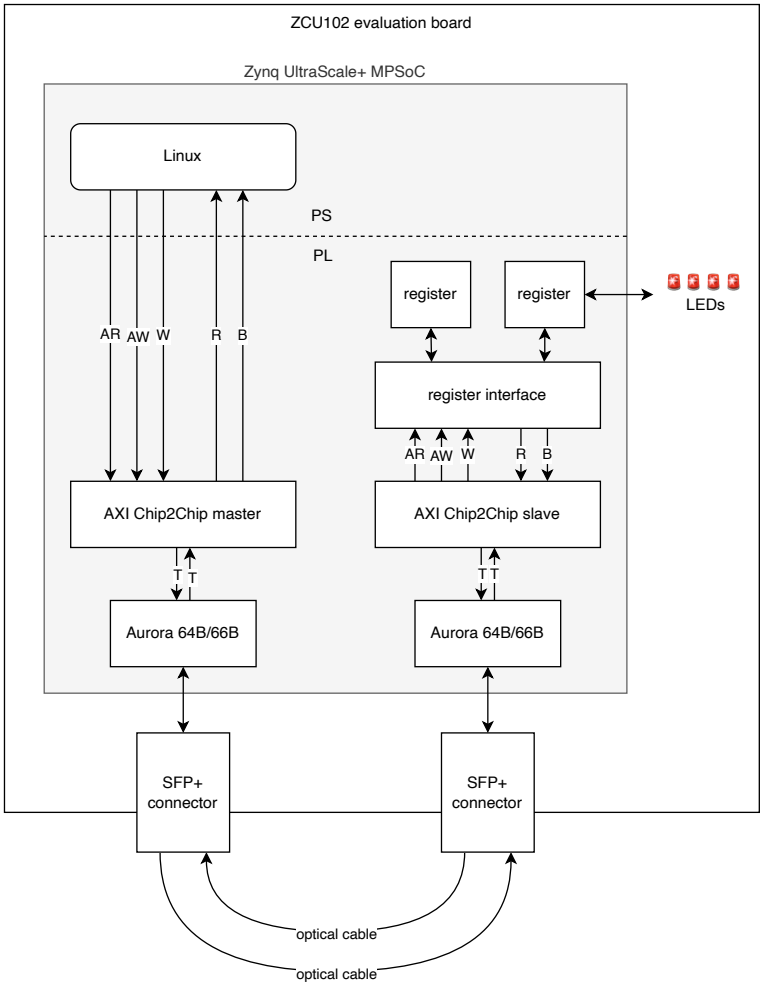


図 J.1　ZCU102 評価ボードでのループバック試験

ZCU102 には 2 × 2 の SFP/SFP+ クアッドコネクタが搭載されており、SFP0（right-top connector）と SFP1

（right-lower connector）を使用する。これらの SFP+ クアッドコネクタは Zynq MPSoC の GTH トランシーバと接続されており、Quad 230 という GT Bank を使う（図 J.2）。

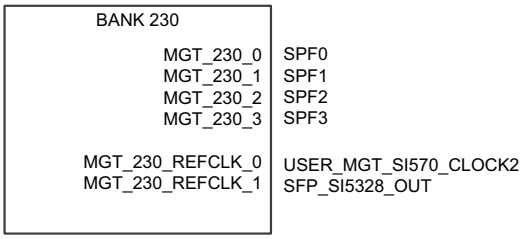


図 J.2　ZCU102 の Bank 230（Xilinx 2019）

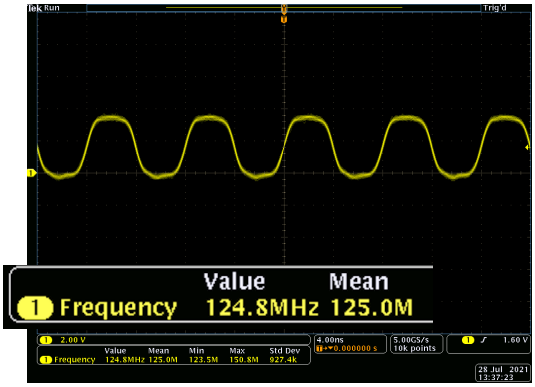


図 J.3　オシロスコープで 125 MHz を確認

MGT_230_REFCLK_0には ZCU102 上の Si570 jitter attenuator からのクロックを入れる。これはデフォルトで 156.25 MHz を出力するが、Si5345 と同様に Linux から I^{2}C でコンフィギュレーションできる。またコンフィギュレーションすべきレジスタはレジスタ 7 から 18 と、135 と 137 のみで、Si5345 に比べて圧倒的に少ない。ここでは 125 MHz を出力するようレジスタの値を計算し、節 3.4 と同様に Linux から I^{2}C で操作した。オシロスコープで Si570 の出力をプローブし、図 J.3 のように 125 MHz が出力されていることを確認した。

MGT_230_REFCLK_1には Si5328 jitter attenuator からの出力を用いる。FPGA によって出力された差動リカバリクロックが Si5328 に入り、その出力が MGT_230_REFCLK_1に入る。Si5328 はまた PS からの I^{2}C 操作でコンフィギュレーションできるが、これはまだ実行していない。

J.1　BRAM-C2C interconnect の構想

今後 ZCU102 でのループバック試験を行うが、ここでも節 4.4 のように AXI SmartConnect を介した PS からのアクセスした際に問題が発生するようであれば、PS から BRAM への読み書きは経験があるため、代替手法として BRAM を介して PS から AXI Chip2Chip master に情報を送ることを考案した。また BRAM と AXI Chip2Chip master を介するものとして BRAM-C2C interconnect の機構も設計した。その場合のファームウェアのヒエラルキーは図 J.4 のようになる。

表 J.1 に示すような BRAM のアドレスマップを決めておく。ユーザ（ソフトウェア）は BRAM の指定のオフセットに ARADDRや WDATAなどを書いておき、OPERATIONのアドレスに `0b01`（読み出し）、`0b10`（書き込み）、または `0b11`（読み書き両方）のオペレーションを書くことで、自動的に動作が開始されるようにした。また BRAM-C2C interconnect は、動作が開始されれば STATUSのアドレスにそれを表すデータを書いておき、ソフトウェアはそれを見て BRAM を操作しない仕組みにした。

BRAM-C2C interconnect は次のようなステートマシンで動作する。

1. レジスタ変数の初期化を行う。
2. BRAM の OPERATIONのアドレスに書かれたデータを読み、動作を開始する。STATUSのアドレスに動作が開始されたことを表すデータを書く。
3. オペレーションに応じて ARチャンネル、AWチャンネル、Wチャンネルの情報を BRAM から取得する。
4. AXI Chip2Chip master と AXI4 の transaction を開始して情報を渡す。

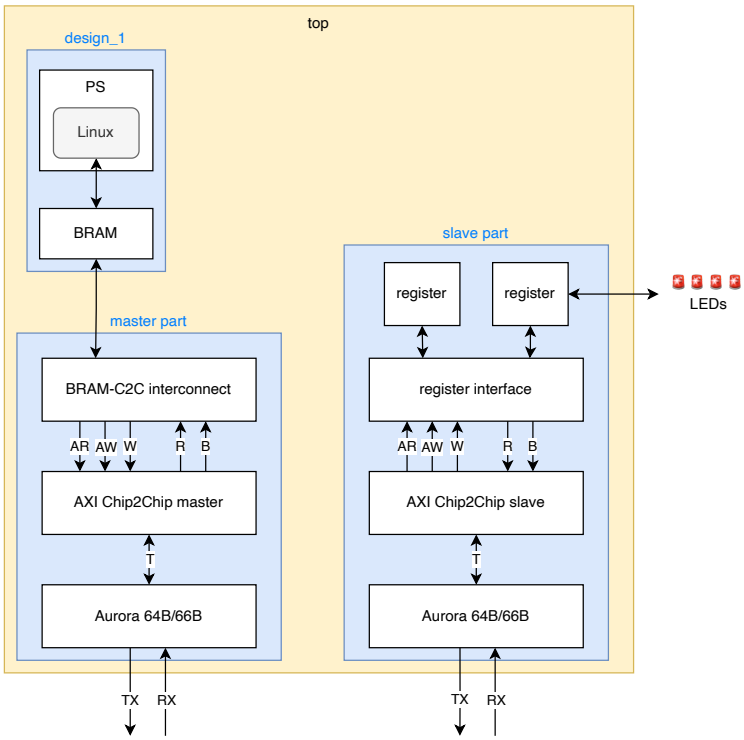


図 J.4　BRAM-C2C interconnect を組み込んだループバックファームウェア

5. AXI Chip2Chip master から transaction の結果を受け取る。
6. 受け取った結果を BRAM に書き込み、STATUSのデータをクリアする。また OPERATIONのアドレスに `0b00` を書いて 2 のステートに戻る。

BRAM-C2C interconnect を作成し、CentOS 7 で動作するソフトウェアも開発したが、BRAM-C2C interconnect が正しく情報を取得せず、レジスタに読み書きができなかった。原因を調査すると、PS（ソフトウェア）側からとロジック側からでは BRAM のアクセスの仕方が次のように異なることが判明した。

- PS から見てオフセット x にあるデータは、ロジック側から $4x$ としてアクセスすることで共有できる。
- ロジック側から書き込んだ 32 bit のデータは、PS からは下位 8 bit しか観測できない。

この検証結果から BRAM のアドレスマップを表 J.1 のように修正した。これを受けてソフトウェアとファームウェアも修正する予定である。

表 J.1　新たに作成すべき BRAM アドレスマップ

AXI4 commands	width/bit	BRAM address	Address offset
OPERATION	2	0×4	0
STATUS	2	1×4	1
ARADDR	32	(2 to 9) $\times 4$	2 to 9
ARID	6	(10 to 11) $\times 4$	10 to 11
ARLEN	8	(12 to 13) $\times 4$	12 to 13
ARSIZE	3	14×4	14
ARBURST	2	15×4	15
RDATA	32	(16 to 23) $\times 4$	16 to 23
RID	6	(24 to 25) $\times 4$	24 to 25
RRESP	2	26×4	26
AWADDR	32	(27 to 34) $\times 4$	27 to 34
AWID	6	(35 to 36) $\times 4$	35 to 36
AWLEN	8	(37 to 38) $\times 4$	37 to 38
AWSIZE	3	39×4	39
AWBURST	2	40×4	40
WDATA	32	(41 to 48) $\times 4$	41 to 48
WUSER	4	49×4	49
WSTRB	4	50×4	50
BID	6	(51 to 52) $\times 4$	51 to 52
BRESP	2	53×4	53

謝辞

本研究室の石野雅也教授と奥村恭幸准教授のご尽力により、新型コロナウイルスの蔓延によるリモートワークでもそれらの影響を感じることなく研究をすることができました。指導教官である石野雅也教授には毎週の研究室会議などで的確なアドバイスやコメントをいただきました。この論文の添削もかなり細かくアドバイスをいただき、心より感謝しております。また Mercury+ PE1 を破損させてしまった時や、KEK で急に LAN ケーブルのかしめ工具が必要になった時など、素早く対応していただきありがとうございました。京都のアパート探しについての相談でもお世話になりました。また奥村恭幸准教授には、研究室会議で毎週必ずコメントや次にすべきことのアドバイスなどをいただき、自分の研究を円滑に進めることができました。VHDL や C/C++ の書き方、オシロスコープの使い方など、テクニカルな面でも勉強させていただきました。また研究室に Ubuntu サーバを構築していただき Vivado と PetaLinux が使えるようになったことで、大幅な時間短縮になりました。お 2 人には大変お世話になりました。誠にありがとうございました。

研究室内で Phase-II の学生として一緒だった、先輩の田中碧人さんと同期の青木匠くんにも大変お世話になりました。田中さんには PetaLinux の使い方をレクチャーしてもらいました。また参考になるソースコードをいただいたり、その他にもさまざまな細かいアドバイスをいただきました。また青木は Vivado の使い方が僕より上手で、ファームウェアのエラーで数日悩んでいる時に彼に相談すれば、一発で解決して驚いた[1]のをよく覚えています。Endcap SL で CentOS 7 からの I^2C の操作がうまくいかなかった時など、困難に直面したときは親身になって一緒に考えてくれました。そのおかげで I^2C の信号線をオシロスコープでプローブしてみようということになり、それによって僕のコーディングに誤りがあることがわかりました。後輩の三島章熙くんは AXI Chip2Chip に関してお手伝いをしてくれました。彼と議論していて AXI SmartConnect を使えば AXI Chip2Chip が PS から使えることに気づいたし、実際に初めて AXI Chip2Chip を使ったシリアル通信を試験したのは彼でした。お互い KEK に出張するのが初めてで、一緒に作業ができて楽しかったです。また慶應義塾大学から一緒だった同期の林雄一郎くんにも長い間お世話になりました。彼と ICEPP の見学会に来たのがきっかけでこの研究室に入ることにしたし、日吉の喫茶店で協力して院試の勉強をしたのはいい思い出です。就職活動もお互い情報共有してやってきました。博士進学するとのことなので、道は違えどお互い頑張りましょう。

ICEPP の他の学生やスタッフにもお世話になりました。研究室での会議では鋭い質問をいただき、自分では当たり前だと思っていたが実はよくわかっていなかったことなどをあぶり出すことができました。半分オンラインでの CERN 夏の学校では様々な興味深い講義を提供していただきありがとうございました。特に CERN 現地のスタッフの方々に中継で ATLAS 検出器を見させていただいたことで、そもそも ATLAS が本当に存在することを初めて実感しましたし、現場の雰囲気も感じることができました。

大学院まで支えてくれた両親と親戚にも深く感謝をしています。2021 年 9 月に亡くなった祖父は、これまで常に僕の学業を応援してくれていました。彼にこの修士論文を見せられなかったことや、晩年会いに行けなかったことが悔やまれます。他にもアルバイト先の方々や友達にも多くの面でお世話になりました。

多くの方々に助けられ、この修士論文が書けたのだと改めて思います。皆さま、どうもありがとうございました。

[1] 生年月日が 1 日違いなのも驚きました。

引用文献

[1] anunesgu 2021 “Designing an XVC project for Remote Debugging of Zynq UltraScale+ devices,” https://support.xilinx.com/s/article/974879.

[2] ARM 2010 *AMBA 4 AXI4-Stream Protocol Specification*, URL: https://developer.arm.com/documentation/ihi0051/a/.

[3] ——— 2021 *AMBA AXI and ACE Protocol Specification*, URL: https://developer.arm.com/documentation/ihi0022/hc/.

[4] ATLAS Collaboration 2017 “Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System,” URL: https://cds.cern.ch/record/2285584.

[5] Avnet *Ultra96-V2 product page*, URL: https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/.

[6] CERN 2021 “The HL-LHC project,” https://hilumilhc.web.cern.ch/content/hl-lhc-project.

[7] Daniele Bagni, A., Di Fresco, J., and Noguera, F. M. V. 2016 *A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS (XAPP1170 v2.0)*, URL: https://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf.

[8] Dzemaili, N. 2021 “A reliable booting system for Zynq Ultrascale+ MPSoC devices,” URL: https://cds.cern.ch/record/2763095, Presented 17 Mar 2021.

[9] Enclustra 2019a “Enclustra Build Environment,” https://github.com/enclustra-bsp/bsp-xilinx.

[10] ——— 2019b *Mercury+ PE1 Base Board User Manual*, URL: https://www.enclustra.com/en/products/base-boards/mercury-pe1-200-300-400/.

[11] ——— 2019c *Mercury XU5 SoC Module User Manual*, URL: https://www.enclustra.com/en/products/system-on-chip-modules/mercury-xu5/.

[12] ——— *Mercury+ PE1-200/300/400 product page*, URL: https://www.enclustra.com/en/products/base-boards/mercury-pe1-200-300-400/.

[13] ——— *Mercury XU5 product page*, URL: https://www.enclustra.com/en/products/system-on-chip-modules/mercury-xu5/.

[14] Koeppel, G., Duerner, D., and Ruder, A. 2020 “Mercury XU5 PE1 Reference Design,” https://github.com/enclustra/Mercury_XU5_PE1_Reference_Design.

[15] Kopeliansky, R. and Schlenker, S. 2019 “DCS for TDAQ subsystems: Requirements Document for HL-LHC,” URL: https://edms.cern.ch/document/1992002/4.

[16] Kopeliansky, R. and Vandelli, W. 2019 “SoC for TDAQ subsystems: Requirements Document for HL-LHC,” URL: https://edms.cern.ch/document/2188675/1.

[17] Mouche, P. 2014 “Overall view of the LHC. Vue d’ensemble du LHC,” URL: https://cds.cern.ch/record/1708847, General Photo.

[18] Redmon, J. and Farhadi, A. 2018 “YOLOv3: An Incremental Improvement,” *arXiv:1804.02767*, URL: https://pjreddie.com/darknet/yolo/.

[19] Silicon Labs 2018 *Si5345, Si5344, Si5342 Rev. D Family Reference Manual*, URL: https://twiki.nevis.columbia.edu/twiki/pub/ATLAS/AnalogTestboard/Si5345-44-42-D-RM.pdf.

[20] The ATLAS Collaboration, Aad, G., Abat, E. et al. 2008 “The ATLAS Experiment at the CERN Large Hadron Collider,” *Journal of Instrumentation* **3**, No. 08, S08003–S08003, DOI: 10.1088/1748-0221/3/08/s08003.

[21] Xilinx 2017 *Vivado Design Suite AXI Reference Guide (UG1037 v4.0)*, URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[22] ——— 2018a *Aurora 64B/66B v11.2 LogiCORE IP Product Guide (PG074)*, URL: https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b/v11_2/pg074-aurora-64b66b.pdf.

[23] ——— 2018b *Debug Bridge v3.0 LogiCORE IP Product Guide (PG245 v3.0)*, URL: https://www.xilinx.com/support/documentation/ip_documentation/debug_bridge/v3_0/pg245-debug-bridge.pdf.

[24] ——— 2019 *ZCU102 Evaluation Board User Guide (UG1182 v1.6)*, URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[25] ——— 2020 *AXI Chip2Chip v5.0 LogiCORE IP Product Guide (PG067)*, URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_chip2chip/v5_0/pg067-axi-chip2chip.pdf.

[26] ——— 2021a “XilinxVirtualCable,” https://github.com/Xilinx/XilinxVirtualCable.git.

[27] ——— 2021b *Zynq UltraScale+ MPSoC Software Developer Guide (UG1137 v2021.2)*, URL: https://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx2021_2/ug1137-zynq-ultrascale-mpsoc-swdev.pdf.

[28] 奥畑宏之 2019 『Interface 2019 年 1 月号小型リアルタイム組み込み人工知能』, CQ 出版社.

[29] 三野裕哉 2020 「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, 京都大学大学院理学研究科物理学・宇宙物理学専攻, URL：https://www-he.scphys.kyoto-u.ac.jp/theses/master/mino_mt.pdf.

[30] 田中碧人 2021 「System-on-a-Chip を用いたエレクトロニクス制御回路の開発 – 高放射線環境での大規模システムへの応用」, 修士論文, 東京大学大学院理学系研究科物理学専攻, URL：https://www.icepp.s.u-tokyo.ac.jp/download/master/m2020_tanaka.pdf.

[31] 東田旺大 2019 「大規模高エネルギー実験における電子回路モジュール制御への system-on-a-chip デバイスの応用研究」, 修士論文, 東京大学大学院理学系研究科物理学専攻, URL：https://www.icepp.s.u-tokyo.ac.jp/download/master/m2018_higashida.pdf.