# プロセッサ搭載FPGAによる新しいデータ取得システムの開発

大阪大学大学院理学研究科物理学専攻博士前期課程
大西 裕二

2019/2/28

## 概 要

Zynq とは、一つのチップに FPGA（Field Programmable Gate Array）とプロセッサを搭載したデバイスのことである。Zynq を用いれば、FPGA でデジタル回路を構成することよるハードウェア開発だけでなく、プロセッサとして使用している CPU 上で Linux などの OS を稼働することにより、ソフトウェア開発も行うことができる。また、Zynq は、FPGA によるハードウェアと、プロセッサによるソフトウェアが、簡単にデータのやり取りを行えるという特徴を持つ。

従来のデータ収集システム（DAQ）開発では、FPGA を、検出器とコンピュータ間のインターフェースとして用いることがあった。この場合、FPGA 上でコンピュータの通信規格に従ったプロトコルによるデータ転送回路を開発しなければならない。一方、Zynq を用いた DAQ システムでは、FPGA 上にコンピュータへのデータ転送回路を開発しなくても、プロセッサを使用したソフトウェアでコンピュータへデータを転送できる。

本研究では、Zynq を用いた DAQ システムを開発し、どのくらいの性能があるかを調べた。まず、Zynq の FPGA 部から OS がアクセス可能なメモリーへのデータ転送レートの測定結果は、最大 400 MB/s のデータレートで転送できるという結果を得た。また、OS が、Gigabit Ethernet で接続されたコンピュータのハードディスクドライブ（HDD）へ、NFS (Network File System) という方法を使ってデータの書き込みを行い、その速度を測った結果、最大で約 28 MB/s のデータレートでデータを保存することができた。

調査の結果、NFS と HDD 書き込みの速度によって制限されているものの、Zynq を用いた DAQ システム全体で、最大約 34 MB/s のデータレートまでなら DAQ に用いることが可能であるという結論を得た。NFS 性能や書き込みパケット量の最適化を行うことにより、さらなる性能の向上が可能である。

最後にデモンストレーションとして、検出器からデータレート 20 MB/s でデータ読み出しを行い、エラーレートの上限値として $8.3 \times 10^{-12}$ を得、想定通りデータ収集ができることを確認した。

# 目 次

# 図 目 次

# 表 目 次

# 第1章　序論

本研究では、FPGA（Field Programmable Gate Array）とプロセッサを一つのチップに搭載した、Zynq という IC を用いて、新しい DAQ システムを開発し、どれだけの性能が得られるかを調査した。

本章では、まず 1.1 節でデータ取得（DAQ）システムについて述べる。次に、1.2 節で、DAQ システムで使われる FPGA というデバイスについて説明し、1.3 節で、FPGA を用いた DAQ システム開発の課題を述べる。1.4 節では、本研究では用いた Zynq というデバイスについて簡単に説明し、最後に、1.5 節で本研究の目的を述べる。

## 1.1　データ取得（DAQ）システム

DAQ とは Data AcQuisition（データ取得）の略で、検出器からの信号を読み出し、コンピュータへ保存することである。また、データ取得を行うシステムのことをデータ取得（DAQ）システムと呼ぶ。

DAQ システムは、図 1.1 に示すように、

- アナログ信号処理回路
- デジタル化回路
- デジタル信号処理回路
- データ転送回路

の 4 つの要素から構成される。アナログ信号処理回路で、検出器からのアナログ信号の増幅・波形整形などを行う。増幅・整形されたデジタル信号を、デジタル化回路でデジタル信号へ変換する。その後、デジタル信号処理回路で必要に応じた処理を行う。データ転送回路はコンピュータへデータを転送する。

近年では、特に、小さなスペースで多くの信号処理が必要な場合や、処理する信号が多い場合などに、ASIC（Application Specific Integrated Circuit）と呼ばれる専用の IC チップのフロントエンド回路[1]を用いることが多い。ASIC は、検出器からの信号を処理し、デジタル信号へ変換するために使われる。

[1]検出器すぐ近くの回路のこと。

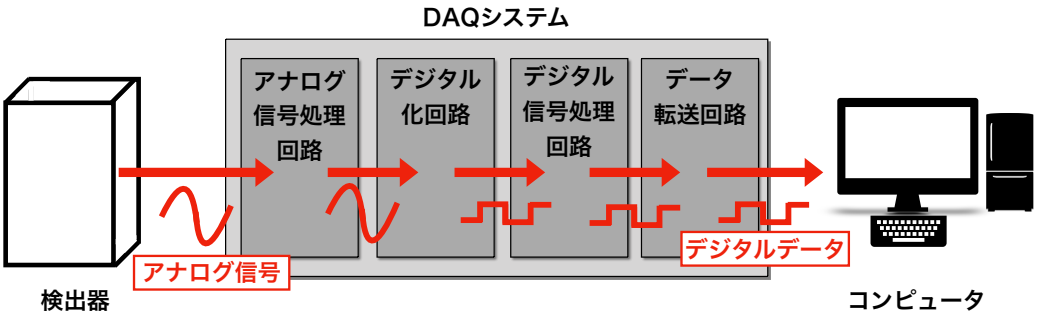


図 1.1: DAQ システムは、検出器からのアナログ信号を増幅・整形するアナログ信号処理回路、アナログ信号をデジタル信号へ変換するデジタル化回路、デジタル信号を必要に応じて処理するデジタル信号処理回路、デジタルデータをコンピュータへと転送するデータ転送回路の 4 つの要素からなる。

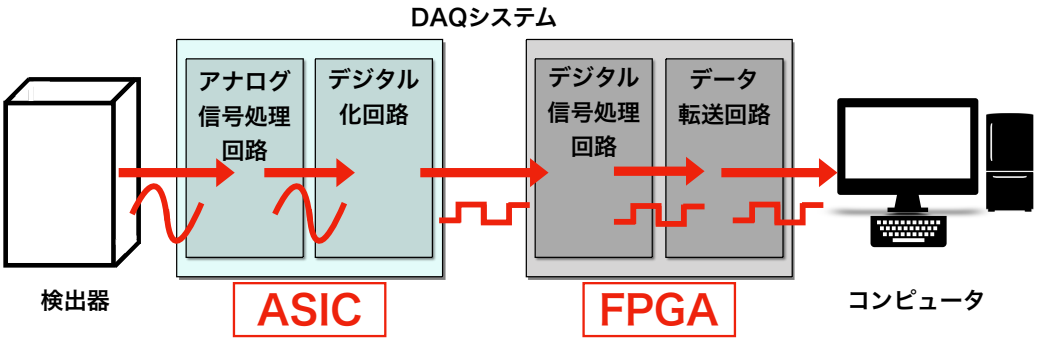


図 1.2: アナログ信号処理回路とデジタル化回路は、ASIC 上に実装することが多い。この時、デジタル信号処理回路とデータ転送回路は FPGA 上に開発する。

ATLAS 実験[2]を例に挙げると、ピクセルサイズ 50 $\mu$m $\times$ 250 $\mu$m、チャンネル数 26880、大きさ 20.2 mm$^2$ $\times$ 19.8 mm$^2$ のピクセルセンサーを読み出すために、FE-I4[3]という名前の ASIC が現在使われている。

図 1.2 に示すように、ASIC などのフロントエンド回路でデジタル化された信号を処理し、コンピュータへ転送するインターフェースとして、1.2 章で述べる FPGA というデバイスがよく使われる。

[2]欧州原子核研究機構（CERN）が、LHC（Large Hadron Collider）加速器を用いて行なっている実験の一つ。

[3]正式には、FE-I4b という名前である。本論文では、以降 FE-I4 と表記する。

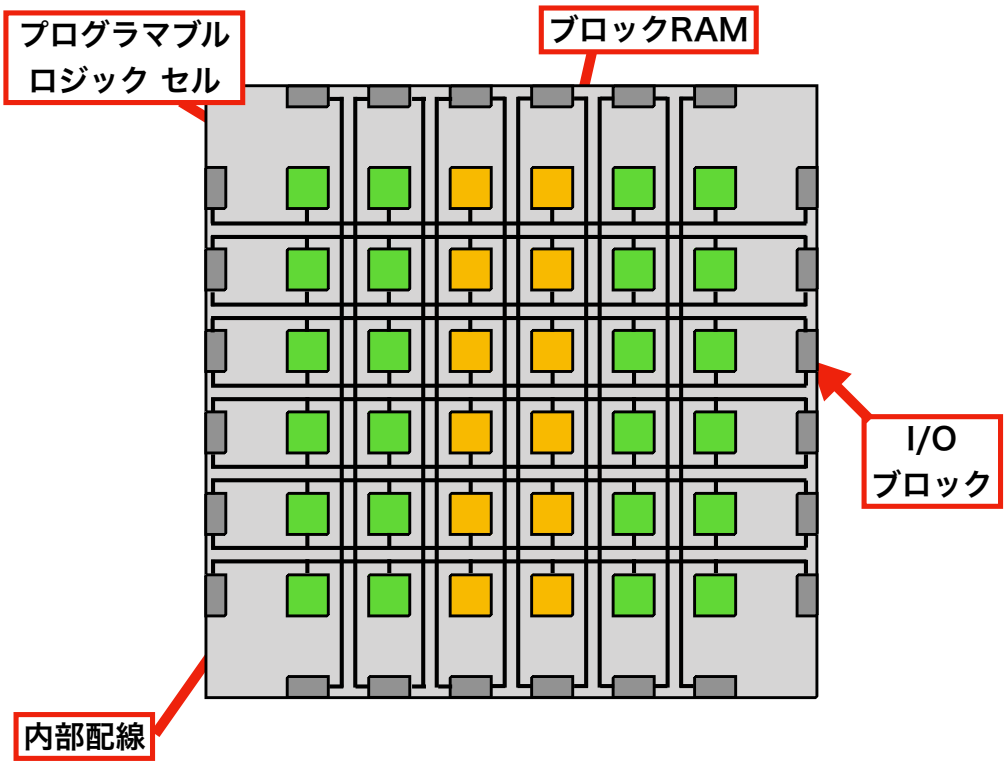


図 1.3: FPGA の構造。プログラマブル ロジック セルと呼ばれる回路素子の集まったブロックが内部配線で結合されている。そのほか、ブロック RAM と呼ばれるメモリ回路や、FPGA 外部と信号のやり取りを行う I/O という回路が存在する。

## 1.2 FPGA

FPGA とは、Field Programmable Gate Array の略で、論理回路のロジックや配線を自由に組み替えることができるプログラマブルロジックデバイスである。

FPGA の構成を図 1.3 に示す。回路素子の集まったブロック（プログラマブル ロジック セル）を内部配線で結合しており、回路素子や内部配線の接続を柔軟に変更できる。また、ブロック RAM と呼ばれる専用のメモリ回路が搭載されており、少量ではあるが、一時的に FPGA 内でデータを保存しておくことができる。FPGA チップ外部と FPGA 内の内部配線は、I/O ブロック[4]という回路を通して接続できる。

FPGA に実装する回路は、ハードウェア記述言語という、デジタル回路の動作仕様を記述する専用の言語を用いてプログラムする。FPGA は、プログ

[4]I/O は input / output の略

ラムされた回路情報を元に、回路素子や内部配線を組み替えて回路情報通りのハードウェアを実現する。この回路情報のことをファームウェアと呼ぶ。

さらに、用途によっては、IP Core（Intellectual Property Core）と呼ばれるファームウェアライブラリを使用して、開発にかかる時間を短縮することができる。

## 1.3 FPGA を使用した DAQ システム開発の課題

FPGA を DAQ システムで用いるためには、FPGA に実装するデジタル信号処理回路とデータ転送回路を開発する必要がある。特に、データ転送回路の開発には専門的な知識が必要であるほか、複雑なプロトコルに従った処理を FPGA 上で行う必要がある。

インターネット経由でデータを転送する例を考えると、比較的簡素な TCP/IP でも、TCP、IP、Ethernet と呼ばれる複数の通信規格（プロトコル）に従う階層構造となる。通常のコンピュータ間であれば、これらのプロトコルは、CPU 上で動く OS（Operating System）が管理しているため、データ転送を行うのに専門的な知識や技術を必要としない。

しかし、FPGA 上にコンピュータへのデータ転送回路を開発する場合は、これらのプロトコルを理解し、FPGA 上の回路を開発しなければならない。このように、データ転送回路の開発には、通信技術の専門知識と、FPGA に複雑な回路を作成する技術が必要となる。専門家が行うようなデータ通信回路の開発を避け、DAQ システムの開発を容易にすることが課題となる。

近年では、その解決策として SiTCP というものがある [1, 2]。SiTCP は、FPGA 上に構築した回路を用いて、TCP、IP、Ethernet プロトコルによる通信を行い、コンピュータへデータを転送する技術である。ライブラリとして提供されている SiTCP を用いることで、新しい DAQ システムへのデータ転送回路の実装が簡単になった。しかし、SiTCP を用いる場合、FPGA 外部の Ethernet 通信専用 IC ごとに、回路をカスタマイズしなければならない。

## 1.4 Zynq

上の問題を解決するため、本研究では異なるアプローチとして、Zynq という、FPGA とプロセッサを搭載した、新しいタイプの IC に着目した。

一般に、プロセッサとは CPU (Central Processing Unit) などの演算処理装置全般を指し、Zynq では CPU を使用している。CPU は、図 1.4 に示すように、核となる演算処理装置（プロセッサ コア）を中心とした周辺回路によって構成され、命令された演算を一つ一つ順番に処理（逐次処理）する。

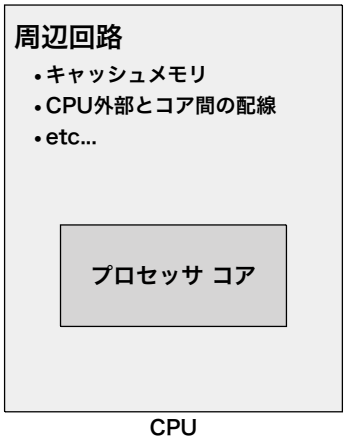


図 1.4: CPU の構造。CPU は、プロセッサコアを中心に、キャッシュコアや CPU 外部ープロセッサコア間の配線といった周辺回路によって構成されるチップである。

Zynq は、CPU を搭載しているため、CPU 上で OS を動かすことができる。さらに、チップ内部の配線を通して、FPGA 上のデータを、OS がアクセス可能なメモリへと転送することができる。

FPGA 上のデータを、OS がアクセス可能なメモリへ転送したのち、OS の機能を用いることによって、簡単にデータをコンピュータへ転送することができる。Zynq についての詳細は 2.1 章で説明する。

## 1.5 研究の目的と本論文の構成

本研究では、プロセッサ搭載 FPGA である Zynq を用いた、新しいタイプの DAQ システムを作った。この DAQ システムの開発には、OS の機能を用いてデータを転送するため、以下のメリットがある。

- 複雑な通信プロトコルを理解する必要がない。
- FPGA で複雑な回路を作成しなくてもよい。

そのため、従来の、FPGA のみを使用した DAQ システムと比べ、簡単に開発することができる。

一方、CPU はプロセスを逐次処理するため、以下のデメリットが存在する。

- レイテンシ[5]が大きい。
- レイテンシが不安定。

[5]通信要求から、実際の通信が始まるまでの遅延時間のこと。

本研究の目的は、Zynq を用いた DAQ システムの開発と、その DAQ システムのデータ転送速度、転送したデータのエラーレートを評価し、Zynq を用いた DAQ システムの可能性を調べることである。Zynq を利用した DAQ システムの研究例は少なく、その具体的な性能は、今後の DAQ 開発で Zynq を用いる上で、一つの指標となる。

本論文では、2 章で Zynq を用いた DAQ システムについて解説し、3 章で実際に開発した DAQ システムの性能を評価する。4 章では、評価結果をまとめ、性能改善案と応用例について述べる。最後に、5 章で結論を述べる。

# 第2章　Zynqを使った新しいDAQシステム

本章では、Zynq を使った DAQ システムについて解説する。はじめに本研究で用いた Zynq というデバイスについて述べ、次に新しい DAQ システムの概要を説明する。さらに、実際に開発した Zynq DAQ の全体構成のデザインについて述べた後、DAQ 上流側から FPGA、CPU という順番で細部のデザインを説明する。

## 2.1　プロセッサ搭載 FPGA Zynq

### 2.1.1　Zynq

Zynq は、FPGA とプロセッサを搭載した Xilinx 社の IC チップ製品群の総称であり、図 2.1 のように CPU をベースとした PS (Processing System) と従来の FPGA に相当する PL (Programmable Logic) が一つのチップの中に入っている。

そのため、DAQ システムの開発に Zynq を用いることによって、FPGA によるハードウェア開発だけでなく、プロセッサを用いたソフトウェア開発や既存のソフトウェア資源の活用が可能となる。

また、PS と PL はチップ内部で接続されているため、PS-PL 間での高速なデータ転送を簡単に行うことができる。

### 2.1.2　Zedboard

本研究では、 Avnet/Digilent 社がリリースしている Zynq 開発用ボード Zedboard（図 2.2）を用いた。

Zedboard は、以下の部品を搭載している。

- Zynq 製品群の一つである Zynq®-7020 チップ
- 512MB DDR3 メモリ[1]

[1]DDR3 メモリとは、Double-Data-Rate 3 メモリの略である。Double-Data-Rate とは、現在主流となっているメモリ規格で、クロックの立ち上がり/立ち下がりの両方を用いることにより、クロックの立ち上がりのみを用いる旧型のメモリの倍速でデータを転送できる。

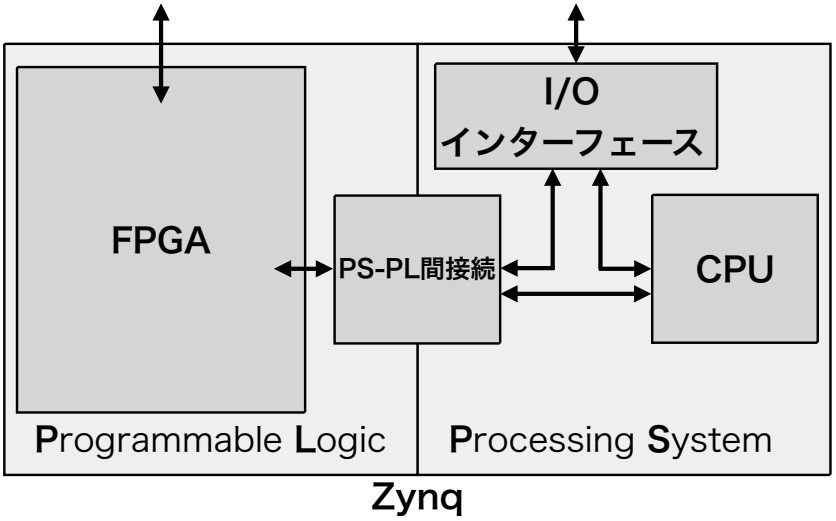


図 2.1: Zynq の構造。Zynq は、CPU とその周辺回路によって構成される PS と、FPGA を中心とした PL を持つ。PS の周辺回路は、主に Zynq 外部との間の I/O インターフェースによって構成される。PS と PL の間は内部接続を通して通信できる。

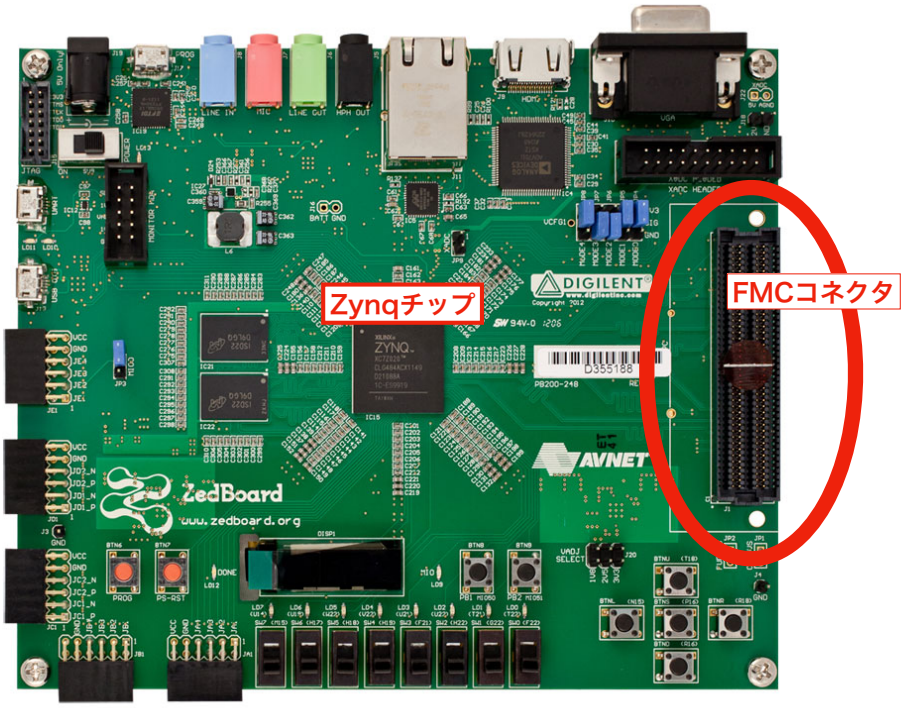


図 2.2: Zedboard の外観 [3]。赤丸で示した位置に FMC コネクタがある。

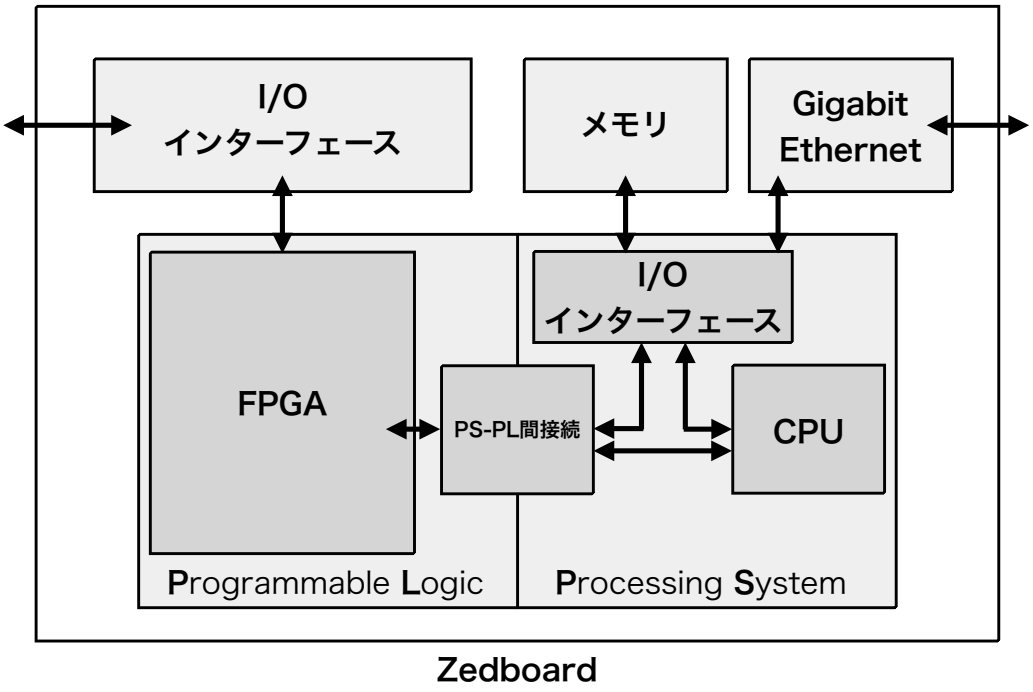


図 2.3: Zedboard の構造。Zedboard 内の配線を矢印で示している。

- Gigabit Ethernet インターフェース
- ボード外部との I/O インターフェース。本研究では、特に図 2.2 に示す FMC コネクタを使用する。

Zynq®-7020 の CPU は、デュアルコア CPU で、動作周波数は 667 MHz である。CPU の動作周波数や、FPGA の性能などの詳しい性能を表 2.1 に記す。

表 2.1: Zynq®-7020 の性能

| | | |
|---|---|---|
| CPU | プロセッサ コア | デュアル ARM Cortex-A9 |
| | 動作周波数 | 667 MHz |
| FPGA | プログラマブル ロジック セル | 85K 個 |
| | ブロック RAM （BRAM） | 4.9 Mbit |

Zynq®-7020 を含む、Zedboard 内の配線を図 2.3 に示す。Zedboard 上の機器と Zynq との接続には以下の 2 種類が存在する。

- Zedboard 外部との I/O インターフェース $\longleftrightarrow$ Zynq の FPGA 部

- メモリや Gigabit Ethernet コネクタ ⟷ Zynq の PS 部に配置されている I/O インターフェース

また、PS と PL を Zynq 内部で接続しているインターコネクトと呼ばれる接続は、AXI4(Advanced eXtensible Interface)[2]というプロトコルに従って PS-PL 間でデータのやり取りを行う。本研究では、PS-PL 間の接続を以下の二つの用途に使用する。

1. FPGA から、CPU がアクセス可能なメモリへデータの書き込みを行う。
   これを用いて、DAQ を行う際、FPGA 部で受信したデータを CPU が読み込む。

2. CPU から FPGA 上のレジスタの値を読み書きを行う。
   この方法で、FPGA 部に作成した回路のレジスタを読み書きし、DAQ システムのコントロールを行う。

それぞれ、以下に説明する DMA 転送、メモリマップド I/O という手法を用いて実現する。

#### DMA 転送

FPGA からメモリへデータを書き込む際には、DMA 転送という方法を用いる。DMA とは Direct Memory Access の略で、CPU は最初に DMA 転送開始命令を送るだけで、それ以降はデータ転送に関与しないメモリ読み書きの手法である。

図 2.4 に通常のメモリアクセスと DMA の違いを示す。DMA 転送では、通常のメモリアクセスに比べ高速にデータをメモリへ書き込むことができる。DMA 転送を制御する回路のことを DMA コントローラーと呼ぶ。本研究では、FPGA 上に DMA コントローラーを作成した。DMA コントローラーは、AXI4 規格の一種である AXI4-Stream 規格を使用して FPGA 上のデータをメモリへ転送する。

#### メモリマップド I/O

CPU から FPGA 上のレジスタを読み書きするために、メモリマップド I/O と呼ばれる方法を用いる。メモリマップド I/O を用いると、FPGA 上のレジスタが特定のメモリアドレスに割り当てられる。CPU は、そのメモリアドレスを読み書きすることによって、FPGA のレジスタの値を読み書きできる。本研究では、この方法で CPU から FPGA 上のレジスタを操作することによって、FPGA の動作を制御する。

[2]AXI4 とは ARM 社が提供する 4 世代目の AMBA （Advanced Microcontroller Bus Architecture）インターフェイス規格である。

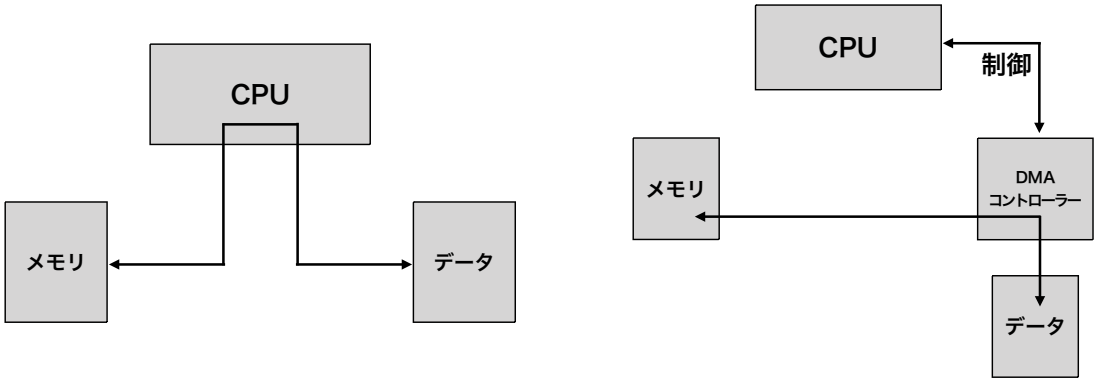


図 2.4: 左：通常のメモリアクセス。右：DMA によるメモリアクセス。通常のメモリアクセスでは、メモリへのアクセスを CPU が管理している。DMA によるメモリアクセスでは、最初に DMA コントローラーへ DMA 転送開始命令を送る以外に、CPU がメモリへのアクセスに関与しないため、高速にメモリへアクセスできる。

## 2.2 DAQ デザイン-全体構成-

本節では、実際に開発した DAQ のデザインについて解説する。図 2.5 に本 DAQ システムの全体構成を示す。本 DAQ システムでは、ASIC からやってくる信号を Zynq の FPGA 部で受信するデザインとする。受信したデータを FPGA 内で処理し、DMA 転送によってメモリへと書き込む。DMA 転送が終了すると、それを合図に CPU 上で動作する OS がメモリ上のデータにアクセスし、Gigabit Ethernet 接続を経由して、2.4.2 章で話す NFS という手段で後方のコンピュータへと転送する。コンピュータへと転送されたデータは、コンピュータ上で動く OS が管理し、ハードディスク（HDD）に書き込む。

本研究では、メモリマップド I/O を使用して、CPU から転送開始用の制御用レジスタを操作し、DMA 転送を開始する。また、CPU が、転送終了を知らせる制御用レジスタを読み続けることにより、DMA 転送の終了を検知できるようにした。DMA 転送が終了したことを CPU が認識したのち、NFS を使用して、メモリ上のデータをコンピュータへ転送する。データをコンピュータへ転送し終えたら、再度 CPU が DMA 転送を開始する命令を発行し、継続的にデータ取得を行う（図 2.6）。

## 2.3 FPGA 部のデザイン

Zynq の FPGA 部には、ASIC からのデータを処理するためのデータ処理回路と DMA 転送を行う回路（DMA 転送回路）を実装した。ASIC からの

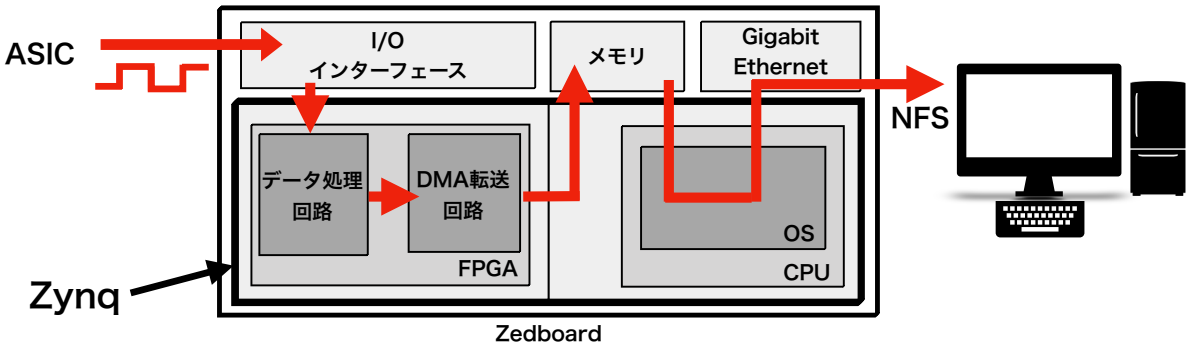


図 2.5: Zynq DAQ 全体構成。ASIC からのデータを、FPGA 内のデータ処理回路を通し、メモリへ DMA 転送する。その後、CPU 上で動く OS によって、NFS を使ってコンピュータへとメモリのデータを転送する。

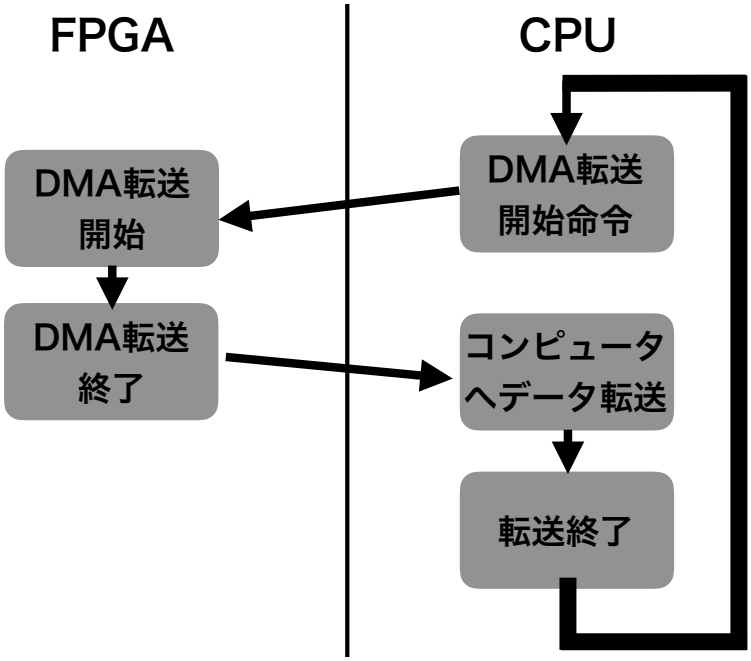


図 2.6: Zynq DAQ のプロセス。DMA 転送によって、メモリへとデータを転送し、その後、CPU が NFS を使用してコンピュータへデータを転送する。このプロセスを繰り返すことで、データを継続的に転送し続ける。

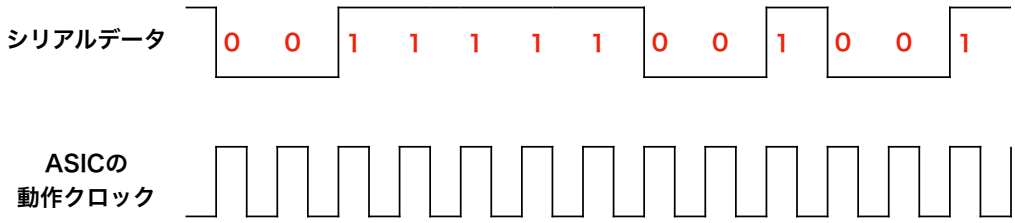


図 2.7: ASIC から出力されるシリアルデータ。シリアルデータは、ASIC のデータ出力回路を動かす動作クロックに同期している。

信号をデータ処理回路で DMA 転送可能な形へ整形し、その後 DMA 転送によってメモリへと直接書き込む。

本項では、まずデータ処理回路について説明し、その後に DMA 転送回路ついて説明する。

また、FPGA 上に作成した回路と、IP Core を使用した回路の一覧を表 2.2 に示す。以下でそれぞれの回路について説明する。

表 2.2: FPGA 上に作成した回路

| | | |
|---|---|---|
| データ処理回路 | サンプリング回路 | IP Core（Xilinx 社提供） |
| | シリアル-パラレル変換回路 | 自作 |
| | 非同期 FIFO | IP Core（Xilinx 社提供） |
| | プロトコル変換回路 | 自作 |
| DMA 転送回路 | DMA コントローラー | IP Core（Xilinx 社提供） |

### 2.3.1 データ処理回路

データ処理回路では、ASIC が出力する、図 2.7 のようなシリアルデータ[3]を受信し、DMA コントローラーの使用に必要なプロトコルへ変換する。この回路を、FPGA 上に作成した。

データ処理回路は、図 2.8 のように、以下の 4 つの要素で構成されている。

- サンプリング
- シリアル-パラレル変換
- 非同期 FIFO

[3]シリアルデータとは、1 クロックごとに 1bit だけ情報を送るデータ形式である。対義語として、1 クロックで複数 bit のデータを送るパラレルデータ形式がある。

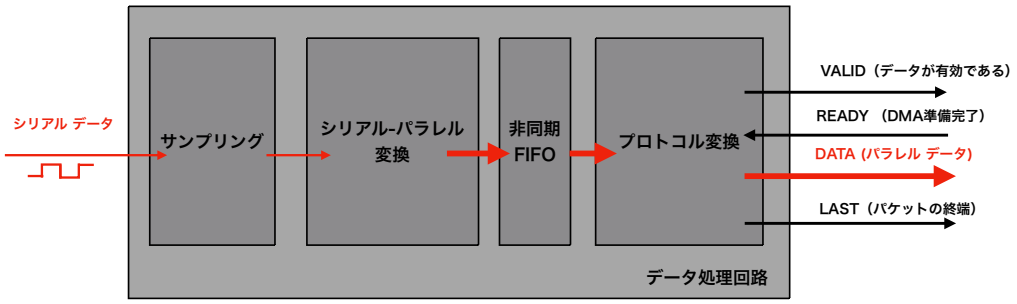


図 2.8: データ処理回路。サンプリング、シリアル-パラレル変換、非同期 FIFO、プロトコル変換の 4 要素から構成される。

- プロトコル変換

このうち、サンプリングを行う回路と、非同期 FIFO の回路には、Xillinx 社の提供する IP Core を用いた。

### サンプリング

サンプリングとは、外部からの入力信号を、FPGA の動作クロックに同期したデジタル信号に変換することである。図 2.9 のように、FPGA 内のサンプリングクロック[4]に同期したタイミングで、入力電圧が閾値よりも高いか低いかで High か Low かを判断し、FPGA の動作クロックに同期したデジタル信号に変換する。3.4.1 章で説明するように、本研究で用いた ASIC は、160 MHz の周波数でシリアルデータを出力する。そのため、サンプリングクロックの周波数は 160 MHz とした。

### シリアル-パラレル変換

サンプリングされた 160 MHz のシリアルデータを、図 2.10 のように、シフトレジスタというビットシフトを行う回路（シリアル-パラレル変換回路）を用いて、32 クロックに一回、32 bit のパラレルデータへ変換した。このクロックの時のパラレルデータのみを 2.3.1 章で説明する非同期 FIFO に入力する。そのため、Write enable という信号を、32 クロックに一回、32 bit のパラレルデータと同期したタイミングで出力するようにした。

[4]サンプリングクロックとは、サンプリングを行う回路を動かす動作クロックのこと。

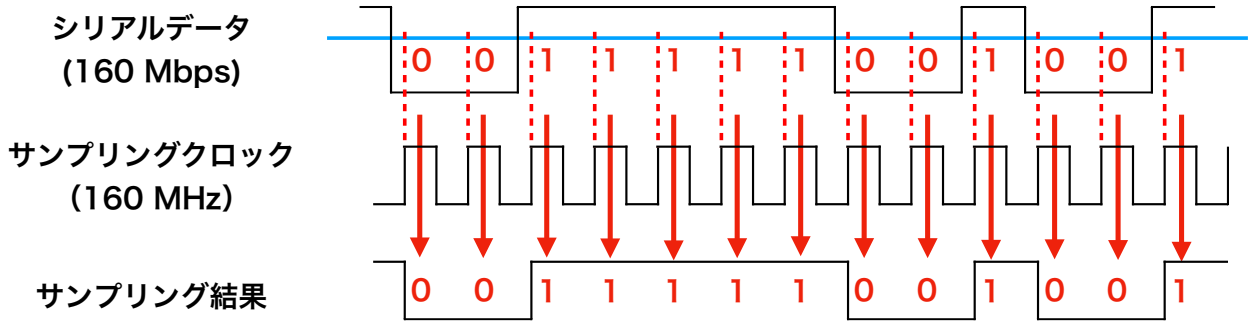


図 2.9: サンプリングの様子。サンプリングクロックが High になるごとに、サンプリング対象の信号電圧が閾値を超えているかを判断し、FPGA 内へサンプリング結果を入力する。青線は、信号電圧が High か Low かを判断する閾値である。

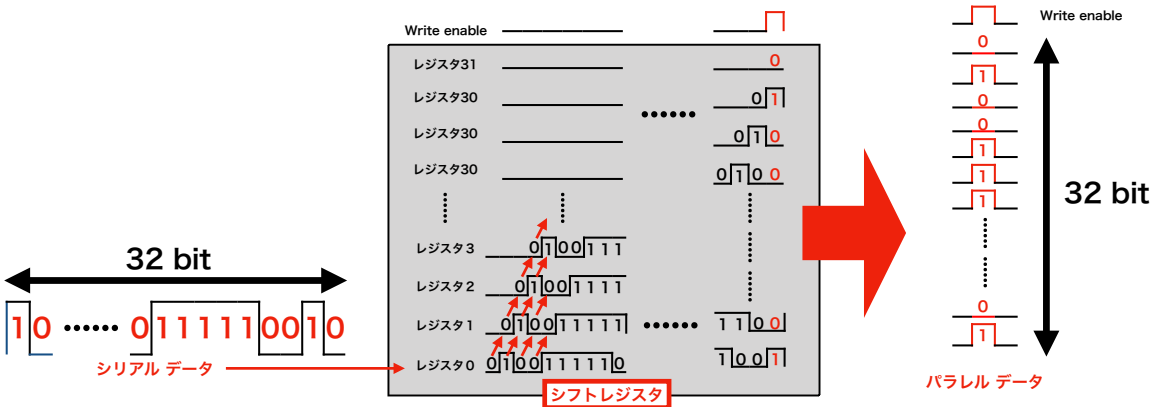


図 2.10: シリアル-パラレル変換回路の描像。シフトレジスタと呼ばれる、ビットシフトを行う回路を使ってシリアル入力をパラレル出力に変換する。本研究では、32 bit 分のシリアルデータを、32 bit のパラレルデータに変換した。

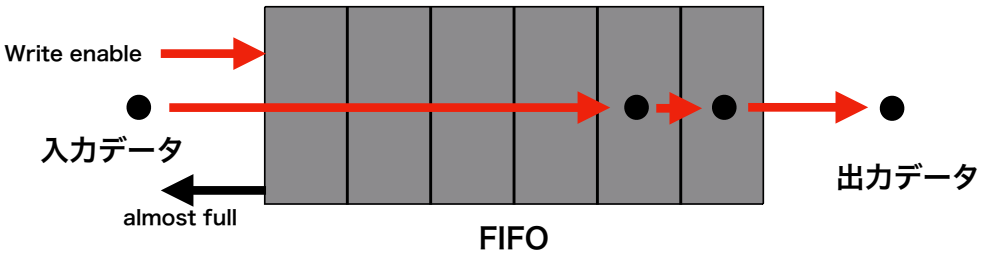


図 2.11: FIFO。データを貯めるバッファの一種。バッファに入った順番に出力される。

### 非同期 FIFO

次に、パラレル信号へ変換されたデータを FIFO (First In First Out) と呼ばれるバッファへ入力する。FIFO とは、図 2.11 のように、最初に入ったデータが最初に出てくるタイプのバッファのことである。FIFO は、Write enable 信号と同時に入力されたデータのみ入力を受け付ける。また、FIFO の中のデータ量が、FIFO 容量上限近く（容量の約 99.9%）になると、almost full 信号を出力する。

今回は、特に非同期 FIFO と呼ばれる FIFO を用いた。非同期 FIFO は、データ入力とデータ出力に異なる周波数の動作クロックを用いて、デジタル信号を搬送するクロック周波数を変更することができる。

本研究で非同期 FIFO を用いた理由は以下のクロック変更、一時データ保存である。

- クロック変更
  160 MHz の動作クロックに同期して伝送されるパラレルデータを、100 MHz で動作する、コントローラー（2.3.2 章 DMA 転送回路の項目で説明）へ入力する。このために、本研究では、非同期 FIFO の入力クロックを 160 MHz、出力クロックを 100 MHz とした。

- 一時データ保存
  NFS を用いてコンピュータへデータを転送している間、DMA 転送は止まる。この間 FIFO を使い、新しく入力されたデータを破棄せずに保存し、データの損失を防ぐ。FIFO の大きさは、FPGA に搭載されている BRAM のサイズ（表 2.1）と、拡張性を考慮して 256 kB（=2048 kbit）とした。

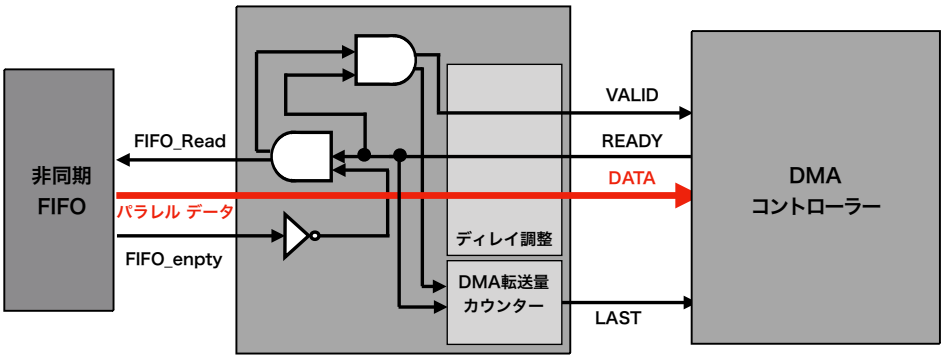


図 2.12: FIFO 内のデータを、DMA コントローラーへ入力可能な AXI4 Stream プロトコルへ変換する回路。DMA コントローラーから READY 信号を受け取ると、FIFO の中身を DATA 信号として、VALID 信号と一緒に DMA コントローラーへ送信する。VALID 信号と READY 信号をモニタすることで、何 Byte のデータを DMA 転送したかカウントし、転送量が 120 kB になると LAST 信号を送信する。

#### プロトコル変換

データ処理回路の最後に、図 2.12 のようなプロトコル変換回路を作成した。この回路で、FPGA へ入力されたデータから、DMA コントローラーを使用するために必要な VALID、DATA、LAST 信号を 120 kB パケットで生成する。

DMA 転送開始命令が DMA コントローラーに書き込まれると、DMA コントローラーは READY 信号をプロトコル変換回路に送信する。するとプロトコル変換回路は、図 2.13 のように、FIFO に存在しているデータを、VALID 信号とセットで DMA コントローラーへ入力する。

プロトコル変換回路は DMA コントローラーへの転送量を計測し、120 kB 送信したタイミングで LAST 信号を送信する。すると、DMA コントローラーは READY 信号を Low にする。これにより、データパケットを一区切りし、DMA 転送を終了する。この 120 kB という値は、3.2 章での DMA 転送の評価を元に決定した。DMA 転送が終了すると、再度 DMA 転送開始命令が書き込まれるまで、DMA コントローラーは停止する。

### 2.3.2 DMA 転送回路

本研究では、データ処理回路にて処理されたデータを、32 本の信号線を使い、100 MHz のクロックを使って DMA 転送する。ここで、100 MHz を用いた理由は、Zynq 上の PS が PL へ供給するクロックのデフォルト設定が

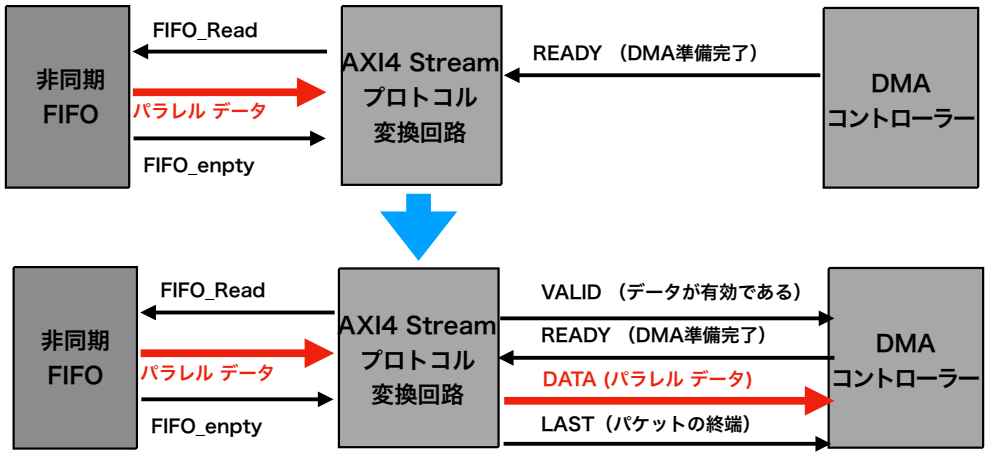


図 2.13: データ処理回路と DMA コントローラーの間のプロトコル。READY 信号は DMA コントローラーが転送準備可能であることを示し、VALID 信号は、同時に送る DATA 信号が有効なデータであることを示す。LAST 信号は、データパケットの境界を示し、DMA 転送を終了する合図となる。

100 MHz だったからである。CPU と FPGA の同期などを考える必要のない、確実に使用可能なデフォルト設定を用いた。計算上、100 MHz で 32 bit 転送すれば、最大 400 MB/s（=3.2 Gbit/s）の速度で FPGA 上のデータをメモリへと書き込むことができる。

DMA 転送を行うための回路として、Xillinx 社の提供する DMA コントローラーという IP Core を FPGA 部に配置した。DMA コントローラーには、メモリマップド I/O によって CPU から値を読み書きできる、制御用レジスタが用意されている。本研究では、DMA 転送開始命令を書き込む、DMA 転送が終了したことを確認する、などの用途で使用した。

また、DMA コントローラーの動作には、制御用レジスタとは別に、以下の 4 つの信号が必要である。

- DATA 信号：転送するデータそのものである。
- VALID 信号：入力されるデータが有効であることを示す。
  READY 信号が有効な時に、VALID 信号と DATA 信号が DMA コントローラーに入力されると、DMA 転送が行われる。
- READY 信号：DMA コントローラーが転送準備完了したことを表す。
- LAST 信号：データパケット[5]の最後を示す。

図 2.14 のように、DMA コントローラーの制御用レジスタへ DMA 開始命令を入力すると、READY 信号が有効になり、DMA 転送可能な状態になる。

[5] 一回の転送で送るデータの塊のこと。

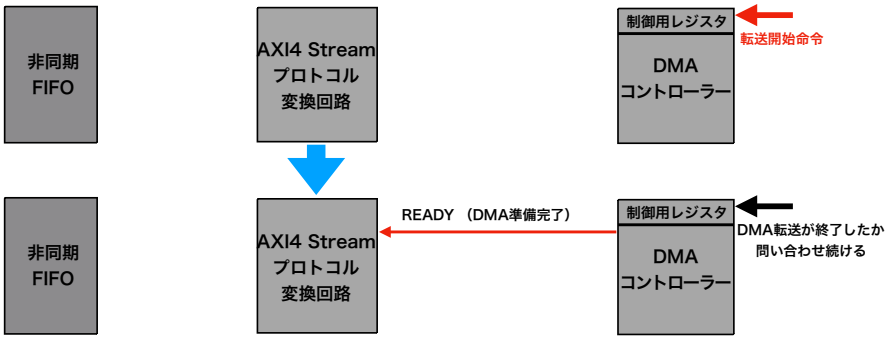


図 2.14: 制御用レジスタへ DMA 開始命令を入力すると、READY 信号が有効になる。

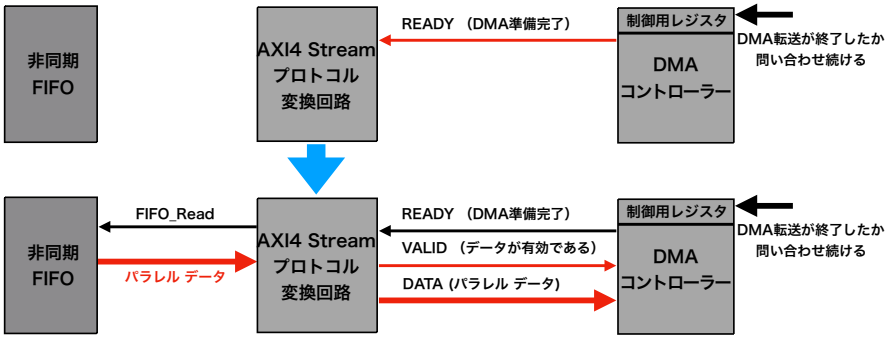


図 2.15: READY 信号が有効になると、プロトコル変換回路が FIFO からデータを取り出し、DMA コントローラーへ VALID とともに送信する。

READY 信号が有効になると、図 2.15 のように、プロトコル変換回路が FIFO からデータを取り出し、DMA コントローラーへ VALID 信号とともに送信する。図 2.16 のように、VALID 信号、DATA 信号だけでなく、同時に LAST 信号が入力された時に、DMA 転送が終了する。この時、READY 信号も無効になるだけでなく、制御用レジスタに DMA 転送終了のフラグが立つため、CPU はそのフラグを読み続けることにより、DMA 転送が終了したことを知ることができる。

## 2.4 CPU 部の OS とそのデザイン

本 DAQ システムにおける CPU の役割は、メモリマップド I/O を通して DMA コントローラの制御を行うことと、コンピュータにデータを転送することの二つである。

今回、CPU 上で動作する OS に下記で説明する Petalinux OS[4] を採用

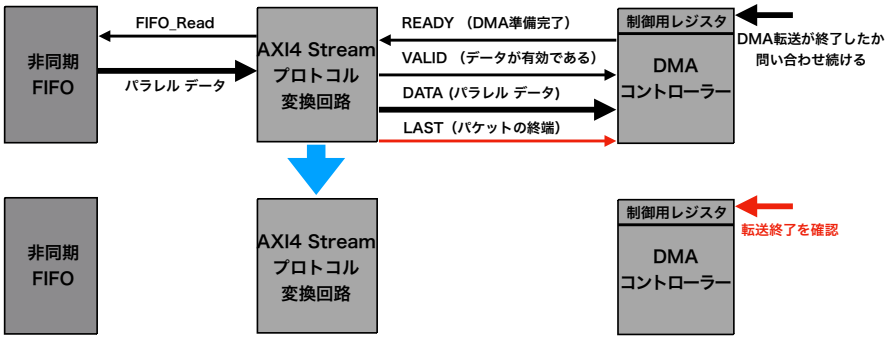


図 2.16: VALID 信号、DATA 信号と同時に LAST 信号が DMA コントローラーへ入力されると、DMA 転送が終了し、制御用レジスタに DMA 転送終了のフラグが立つ。

し、データ転送先のコンピュータでは Linux OS[6]を動作させた。本研究では、Petalinux OS-Linux OS 間で NFS (Network File System) を使用してデータ転送を行なった。

### 2.4.1 PetaLinux OS

PetaLinux OS とは、Xilinx 社の提供する PetaLinux Tools というツールによって生成されるカスタム Linux OS のことである。PetaLinux Tools には、開発した Zynq ハードウェア情報を取り込んで、それを反映した設定を行なう機能がある。本研究では、特に以下の簡便性に着目して PetaLinux OS を使用した。

1. OS が使用可能なメモリ領域を、DDR3 メモリ上に確保する設定を自動で行う。
2. DMA コントローラーの制御レジスタを、メモリマップド I/O 経由で操作できるように、特定のメモリアドレスに自動でマッピングする。
3. NFS や C 言語ソフトウェアを実行するのに必要なソフトウェア資源を、OS 本体を格納した image ファイルに同梱できる。
4. FPGA の回路情報と OS のブートローダー[7]をまとめた boot ファイルを作成できる。
5. 3. で作成した boot ファイルを使用すると、PetaLinux OS の起動を行う前に、boot ファイルに含まれる FPGA の回路情報を元に、FPGA 上にデジタル回路を構築する。

[6]本研究では、特に CentOS 7 という Linux distribution を使用した。
[7]OS を起動するためのプログラムのこと。

### 2.4.2 NFS

NFS (Network File System) とは、複数のコンピュータがファイルを共有する仕組みのことであり、NFS サーバーの HDD を NFS クライアントがネットワーク越しにマウントし、手元にあるかのように扱うことができる。

本研究では、Linux OS を NFS サーバーとし、PetaLinux OS を NFS クライアントとする。NFS を用いることによって、PetaLinux OS は Linux OS を通して、コンピュータの HDD を使用することができる。

図 2.17 に、NFS を用いたデータ転送のプロセスを示す。Zynq の CPU 上で動作する PetaLinux OS が、Linux OS に対して、データ書き込み要求と共にデータを送信する。Linux OS は、NFS を通じてデータを受信すると、受信したデータを HDD へ書き込む前に、メモリ上でバッファリングする。データをバッファに書き終えると、実際に HDD にはデータが書かれていなくても、データを受け取ったことを知らせる応答を PetaLinux OS へ返す。HDD への書き込み速度は、少量の書き込みを繰り返すより、大量に一回で書き込む方が無駄がなく高速である。そのため、Linux OS は書き込むデータを一定量バッファリングしたのち、バッファの中身をまとめて HDD へ書き込む。このバッファリングの量によって HDD 書き込みの速度に違いがあるため、Linux OS がバッファリングする量はデータ取得速度に影響する。

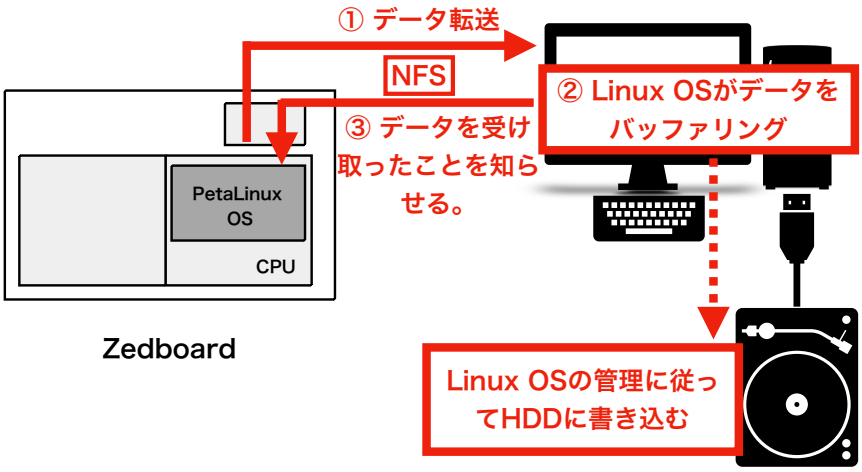


図 2.17: NFS を用いたデータ転送。PetaLinux OS は、NFS を経由してデータを Linux OS へ転送する。Linux OS は HDD 書き込みを高速に行うため、すぐには HDD へデータを書き込まず、一旦メモリに用意したバッファへとデータを格納する。このバッファへデータを格納すると、Linux OS は、Petalinux OS へ、データを受け取ったことを知らせる。

# 第3章　Zynq DAQの性能評価

本章では、以下の各要素の性能評価を通して、Zynq を用いた DAQ システムの可能性を研究した。DAQ 最下流のコンピュータ側から上流に向かって順に、NFS データ転送、DMA 転送、データ処理回路を評価する。今回 DAQ 下流側から順に評価をする理由は、システム上流にあるデータ処理回路を評価するために、データを保存する機能が必要だったからである。最後に、総合評価として実際に ASIC からのデータのデータを取得し、コンピュータへと正しくデータが保存されていることを確認する。

## 3.1　NFSデータ転送の評価

本節では、PetaLinux OS – Linux OS 間での NFS データ転送速度について評価する。

2.4.2 章で説明したように、HDD へデータを書き込む前に OS によるデータのバッファリングが行われる。このバッファリングの量によっては、実際に HDD にデータを書き込む時に CPU 稼働率が急上昇してしまうことがある。コンピュータがこのタイミングで PetaLinux OS からデータを受信すると、OS が HDD への書き込みを優先し、PetaLinux OS へデータを受け取ったことを知らせるまでに大きな遅延が生じることがある。

この問題を解決するため、Linux OS のバッファリングサイズを変更しながら NFS 転送にかかった時間を測定し、最適なバッファリングサイズを決定した。

### 3.1.1　評価方法

はじめに、Petalinux OS でメモリ上に 120kB のデータを用意し、このデータを、NFS を利用して Linux OS へと転送した。

しかし、Linux OS のバッファリングサイズが転送データ量 120 kB を上回る場合は、一回転送しただけでは HDD には書き込まれず、メモリにバッファリングされるだけである。そこで、NFS を用いて、10000 回連続でデータを繰り返し転送した。繰り返し転送している最中に、バッファリングしたデータ量がバッファリングサイズを上回ると、Linux OS は HDD へデータを書き込み始める。Linux OS は、HDD へデータを書いている最中でも、PetaLinux OS

から送られてきたデータを、メモリの別の領域へバッファリングする。そのため、NFS でのデータ転送と、HDD へのデータ保存を並行して行うことができる。

この繰り返し転送 1 サイクルにかかった時間（図 2.17 の③から、次のサイクルの③までの時間）を測定し、NFS データ転送の速度と安定性を評価した。

同時に、Linux OS のバッファリングサイズを変更しながら測定を繰り返し、最適なバッファリングサイズを調査した。Linux OS のバッファリングサイズは、dirty bytes という名前の kernel parameter を変更することにより設定できる。

### 3.1.2 結果

測定結果のヒストグラムの一部を図 3.1 に示す。横軸は 120 kB を NFS 転送する 1 サイクルにかかった時間を表す。dirty bytes の値が小さいほど、データ転送にかかる時間が長い。

横軸に dirty bytes の値、縦軸に 1 サイクルにかかった時間の 10000 回の平均をプロットしたものが図 3.2 である。dirty bytes の値が 120 kB よりも大きいと、120 kB を転送するのにかかる時間が平均で 4300 $\mu$s で安定していることがわかる。

図 3.2 で示した平均値と、一サイクルの転送にかかった時間からデータ転送レートを計算したものを図 3.3 に示す。dirty bytes の値が十分大きい場合、28 MB/s のデータ転送レートで NFS データ転送ができることがわかった。

本研究では、データ書き込み速度が最大で約 100 MB/s の HDD 用いたため、Gigabit Ethernet の転送速度が約 120 MB/s であることを考えると、本来は、最大で約 100 MB/s のデータ転送速度が得られるはずである。しかし、今回の結果では 28 MB/s の転送速度となった。さらに、dirty bytes パラメータが十分大きい時の、NFS 転送するのにかかった時間の平均値 4300 $\mu$s に対して、倍である 8600 $\mu$s を閾値とし、それ以上時間がかかった事象数（外れ値）と dirty bytes パラメータの関係を、図 3.4 に示す。測定の結果、16 MB〜256 MB の範囲で外れ値の数が最小になっており、最も安定してデータ転送を行えることがわかる。

以上の結果から、NFS を用いると、HDD へデータを保存しながら、28 MB/s のレートでデータを転送できることがわかった。また、HDD 書き込みを行う前のバッファリングサイズは 16 MB〜256 MB が適切である。バッファリングサイズが小さい方が、頻繁にデータを HDD へ保存できるため、本研究ではバッファリングサイズとして 16 MB を採用した。

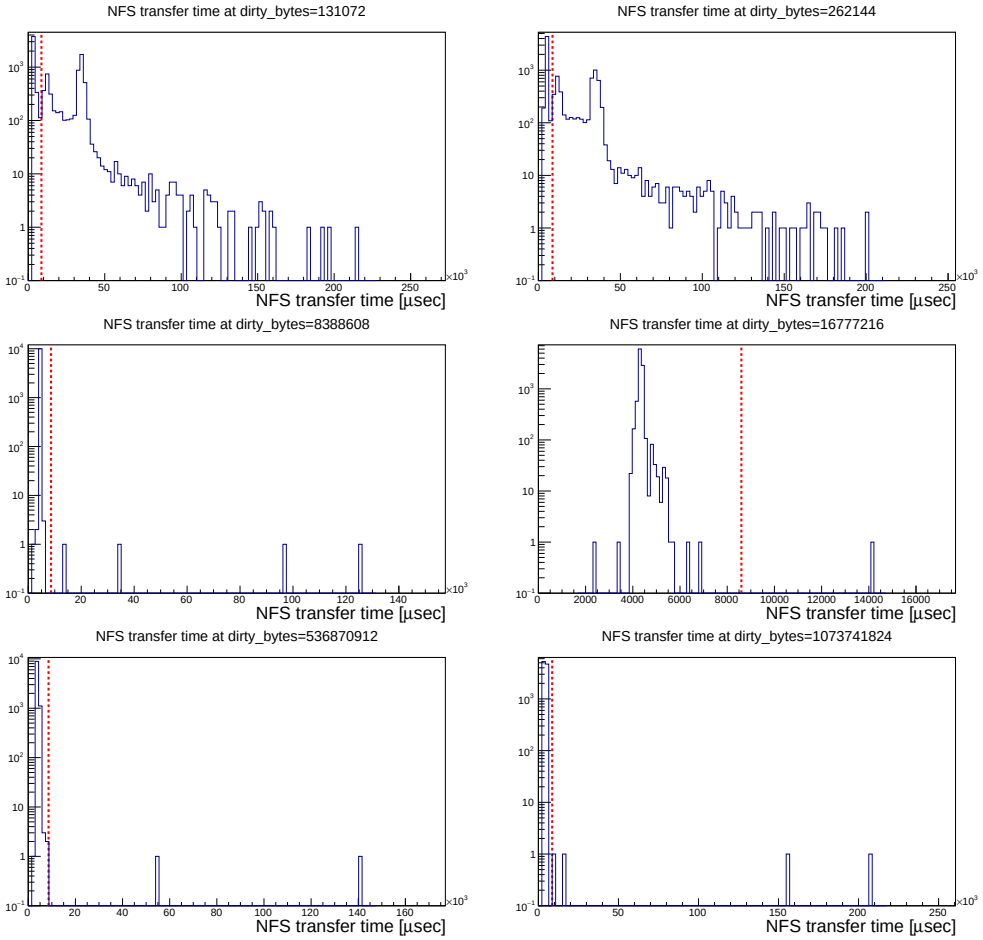


図 3.1: NFS 転送にかかった時間のヒストグラム。横軸は 120 kB を NFS 転送する 1 サイクルにかかった時間を表す。6 つのヒストグラムは、それぞれヒストグラムの上に示した dirty bytes という、Linux OS が HDD へデータを書き込む前に行うバッファリングのサイズを変えながら測定した結果である。ここで、dirty bytes の単位は byte である。図中の赤点線は、横軸 8600 $\mu$s を表す。この値より大きく転送に時間がかかったものを外れ値として数える。

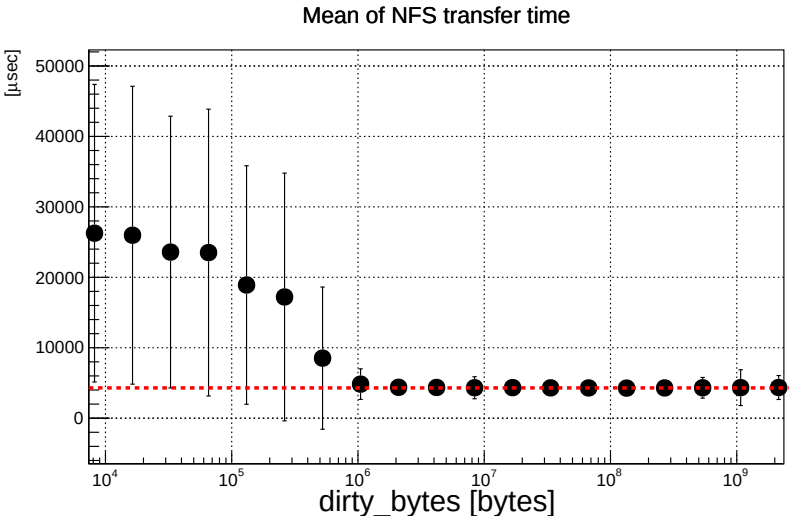


図 3.2: NFS 転送するのにかかった時間の平均値と dirty bytes の関係。赤点線は、縦軸 4300 $\mu$s を表す。縦棒は、図 3.1 での RMS（Root Mean Square：二乗平均平方根）を表す。

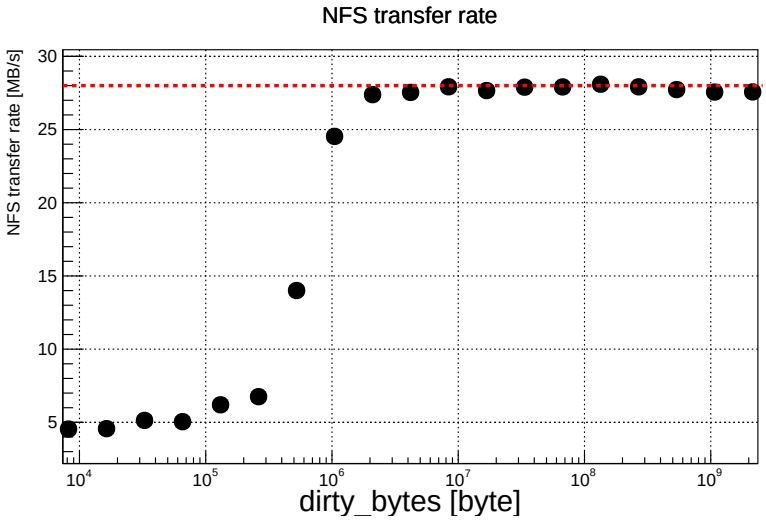


図 3.3: 10000 サイクル測定で調べたデータ転送速度と dirty bytes の関係。赤線は、縦軸 28 MB/s を表す。

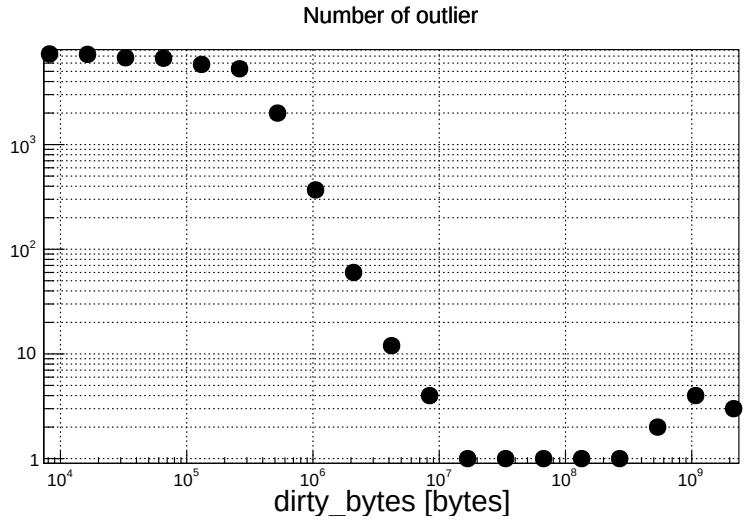


図 3.4: 10000 回データを転送した時に、転送かかった時間が 8600 $\mu$s の閾値より長かった事象数。横軸は dirty bytes パラメータの値。

## 3.2 DMA 転送の評価

本節では、Zynq の FPGA 部にあるデータをメモリへと転送する DMA 転送について評価する。FPGA 上のデータをメモリに DMA 転送し、転送開始から転送終了までにかかった時間を測定した。

2.3.2 で説明したように、理想的には 400 MB/s の速度で DMA 転送ができる。しかし、一回の転送データ量が小さいと、DMA コントローラーの制御にかかる時間が無視できなくなる。

そこで、転送データ量を変化させながら DMA 転送時間を測定し、実際にデータを取得するときに用いるデータ転送量を決定する。

### 3.2.1 評価方法

まず、FPGA 上に、DMA コントローラーから READY 信号を受け取ると、DATA 信号と VALID 信号を DMA コントローラーへ送信する回路（DMA 評価用回路）を作成した。DMA 評価用回路は、ある回数 VALID 信号と DATA 信号を送信すると、LAST 信号を DMA コントローラーへ出力する。

図 3.5 に、DMA 転送を評価するための手順を示す。PetaLinux OS から、2.1.2 章で述べたメモリマップド I/O を通して、DMA コントローラーへ DMA 転送開始命令を送る。転送開始命令を受けた DMA コントローラーは、先ほど用意したデータをメモリ上の設定されたアドレスへと書き込み始める。DMA 転送が終了すると、転送が無事終了したこと意味する制御用レジスタの値が High になる。PetaLinux OS は、メモリマップド I/O を通して、そのレジス

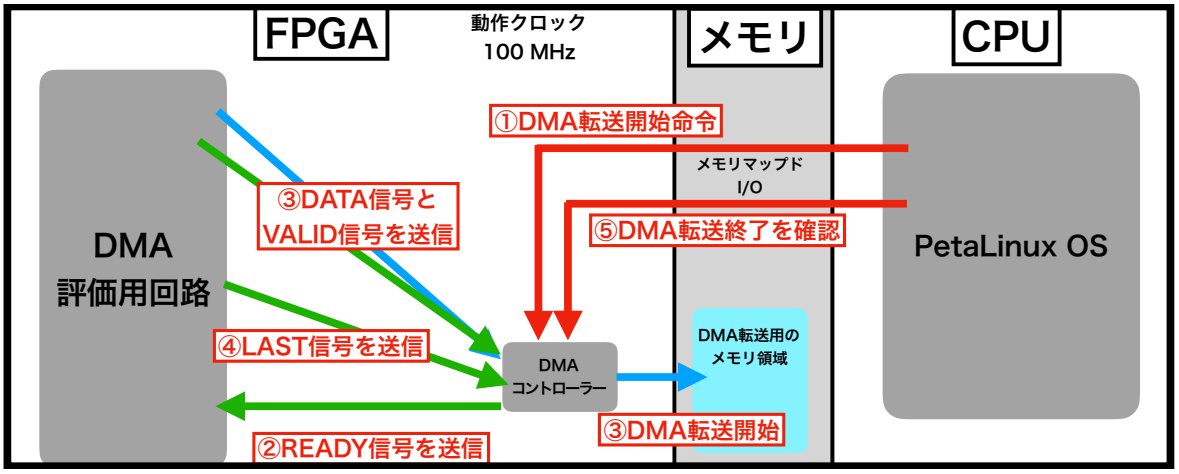


図 3.5: DMA 転送を評価するための手順。赤矢印は PetaLinux OS から行う操作を表す。青矢印は、転送するデータの流れを意味する。緑矢印は、それぞれ VALID 信号、LAST 信号、READY 信号を表す。①から。⑤までが 1 サイクルであり、⑤の後はもう一度①に戻る。

タの値が High になるまで制御用レジスタを読み続けることにより、DMA 転送が終了したことを知ることができる。

今回、図 3.5 の①直前から⑤直後までの時間を繰り返し測定した。この時間には、実際の転送時間だけでなく、制御用レジスタを読み書きする時間も含まれている。

また、一回の転送データ量（DMA 評価用回路が LAST 信号を出力するまでに送信する、VALID 信号と DATA 信号の回数）を変更しながら同様の測定を行い、DAQ に最も適した一回の転送データ量を調べた。

### 3.2.2 結果

一例として、一回の転送データ量が 120 kB の時の、DMA 転送にかかった時間（PetaLinux OS が DMA 転送開始命令を送る直前から、DMA 転送が終了したことを知るまでの時間）をヒストグラムにしたものが図 3.6 である。

図 3.7 に、一回の転送データ量ごとの DMA 転送にかかった時間の平均値をプロットした。横軸は一回の転送データ量、縦軸は DMA 転送にかかった時間を表したグラフであり、縦軸の誤差棒は DMA 転送にかかった時間の標準偏差を表す。図 3.7 左から、一回の転送データ量が大きい範囲では、測定時間が転送データ量の一次関数になっていることがわかる。図 3.7 右のグラフは、転送データが少ない範囲を拡大したものである。一回の転送にかかる時間が小さい時は、丸め誤差[1]があるため、一次関数からずれ、階段状の形が

[1]小数点以下が切り捨てられることによる誤差

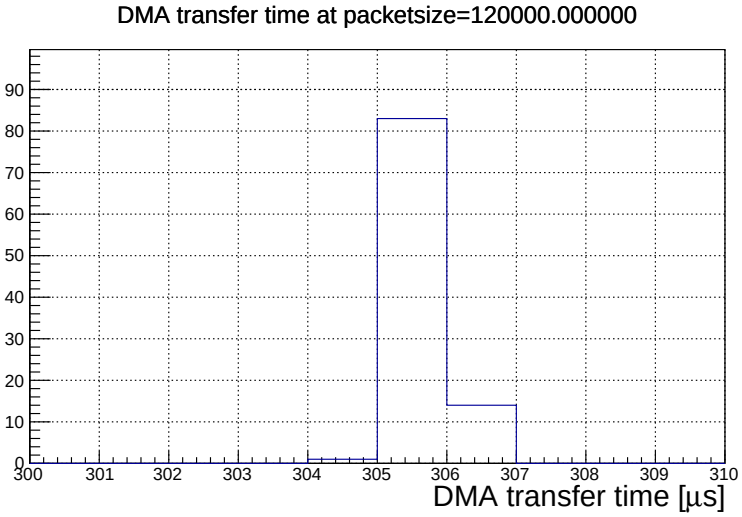


図 3.6: 一回の転送データ量が 120 kB の時の、転送にかかった時間のヒストグラム。横軸は DMA 転送にかかった時間である。縦軸の誤差棒は DMA 転送にかかった時間の標準偏差を表す。

見えている。図 3.7 右の y 軸切片は、DMA 転送で実際にデータを転送しなくてもかかる時間を表す。したがって、DMA コントローラーの制御用レジスタを読み書きするのにかかった時間は、3〜4 $\mu$s 程度である。

図 3.8 は、転送したデータ量を DMA 転送にかかった時間で割って、転送速度を計算したものである。横軸は図 3.7 同様に一回の転送データ量であり、縦軸は転送速度を示す。一回の転送データ量が大きくなるにつれて、制御用レジスタを読み書きするのにかかる時間が相対的に小さくなるため、理論値である 400 MB/s に近づいて行く。図 3.8 からは、100 kB($\sim 10^5$ Byte) 付近から DMA 転送速度が理論値に漸近していることがわかる。

一回の転送データ量が小さければ小さいほどデータ損失に強いと考え、理論値に漸近し始める 120 kB を実際の Zynq DAQ における一回の転送データ量とした。

## 3.3 データ処理回路の評価

本節では、データ処理回路について、以下の２項目を評価をする。

1. データ転送速度
   データ取得可能なデータ転送速度。

2. エラーレート
   外部から入力したデータを取得した時に、入力データと異なった値を誤って取得した割合。

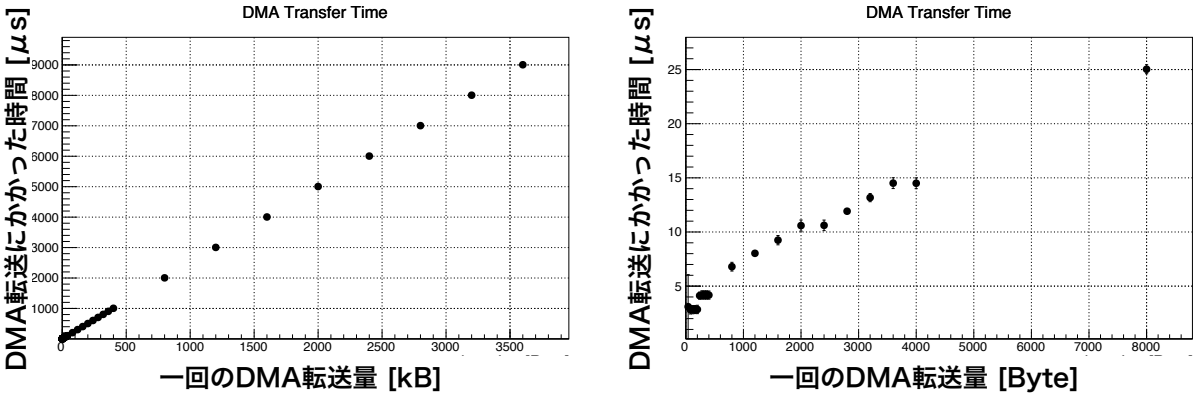


図 3.7: 左図は、一回の転送データ量を変えながら、DMA 転送にかかった時間を測定したプロット。横軸は一回の転送データ量、縦軸は転送にかかった時間を表す。一回の転送データ量が十分大きな範囲では、転送時間が転送量に比例している。右図は、左図の横軸が小さな範囲を拡大したもの。DMA コントローラーの制御を行う時間を含んでいるため、y 軸切片は 0 ではない。

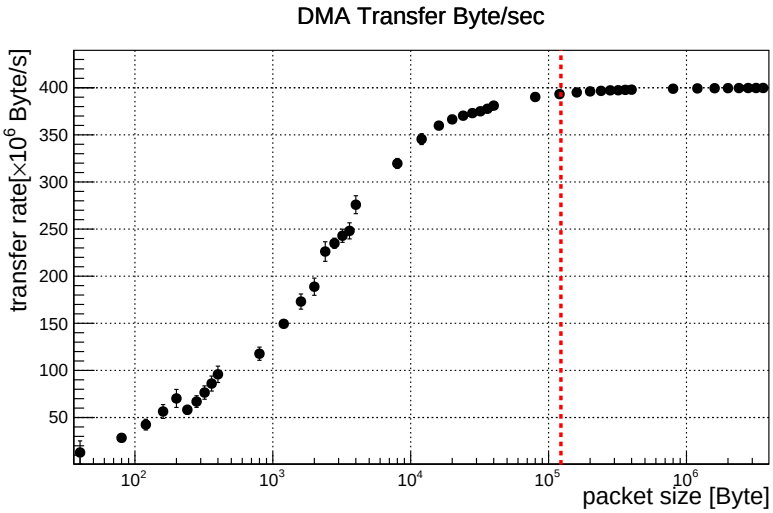


図 3.8: 一回の転送データ量ごとに DMA 転送にかかった時間を表すプロット。横軸は一回の転送データ量、縦軸は DMA 転送速度を表す。赤点線は、実際の Zynq DAQ に採用した、一回の転送データ量 120 kB を示す。

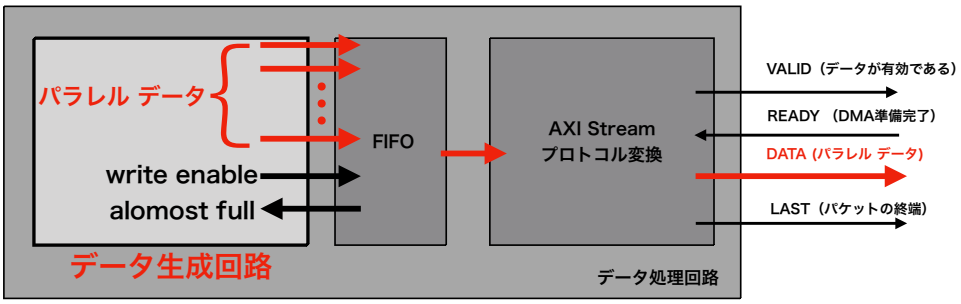


図 3.9: 図 2.8 の非同期 FIFO へ、あるデータレートでパラレルデータを次々と入力し、疑似的に入力データレートを変えながらデータ取得を行う。

データ転送速度の評価では、図 3.9 のように、FPGA 上で、あるデータレートでパラレルデータを生成し、それを次々に非同期 FIFO へ入力していった。データ生成回路は、write enable 信号と一緒にパラレルデータを FIFO へ送信する。また、飽和している FIFO にさらにデータを書き込もうとすることを防ぐため、FIFO が飽和直前であることを示す almost full 信号が High の時は、write enable 信号とパラレルデータを送信しないようにした。

生成するデータのデータレートを変えながら、図 3.10 のように、2.3.1 章で説明したプロトコル変換回路と DMA 転送回路、2.4.2 章で述べた NFS を通してデータ取得した。図 3.5 の①直前から⑤直後までの時間を繰り返し測定することにより、データ取得可能なデータ転送速度を評価した。

エラーレートの評価では、図 3.11 のように、FPGA 内で、あるパターンのシリアルデータを生成し、Zedboard 外部へと出力する。外部へ出力した信号をケーブルでループバックさせて、疑似的に ASIC から送られてきたデータとみなし、2.3.1 章で説明したデータ処理回路と DMA 転送回路、2.4.2 章で述べた NFS を通してコンピュータに保存する。コンピュータに保存されたデータのパターンを、生成したデータのパターンと照らし合わせ、データのエラーレートを評価する。このセットアップでは、FIFO が almost full 信号を出力しているときもデータが次々と送られてくるため、FIFO が受け止められなかったデータは、受信できずに損失してしまう。そのため、FIFO に入力されたデータに対してのみ、エラーレートを評価した。

次節では、実際の評価方法について説明する前に、項目 2 のエラーレートを評価を行うために必要な Zedboard 外部と FPGA のインターフェースについて説明する。その後、それぞれの項目の評価方法を説明し、評価結果をまとめる。

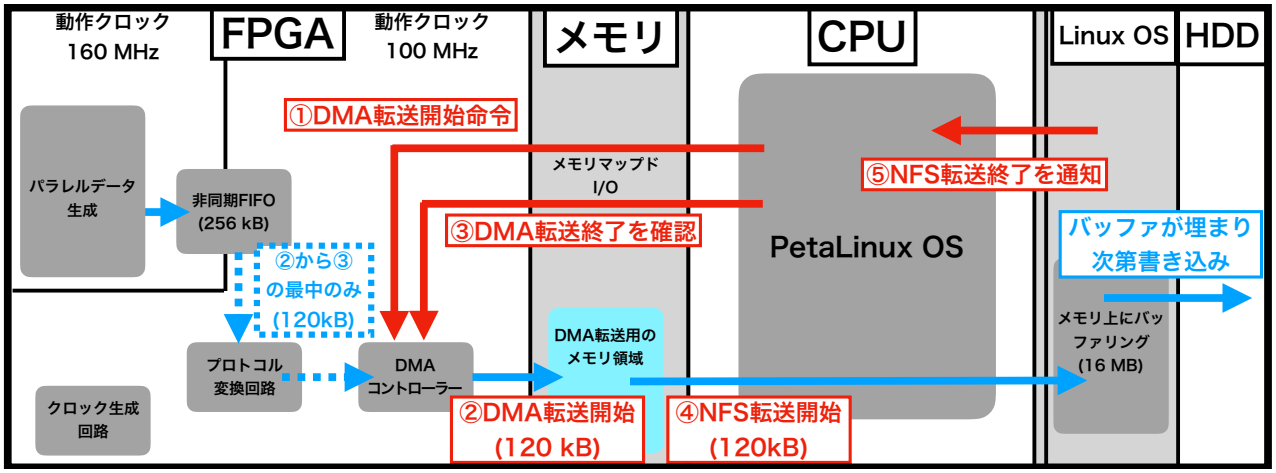


図 3.10: 生成したパラレルデータをデータ取得するプロセスの簡略図。赤矢印は、データ取得を行う時の手順を表す。①から⑤までが 1 サイクルであり、⑤の後はもう一度①に戻る。青矢印は取得するデータの流れを表す。青点線矢印は、DMA 転送の最中（②と③の間）のみデータが流れることを意味する。

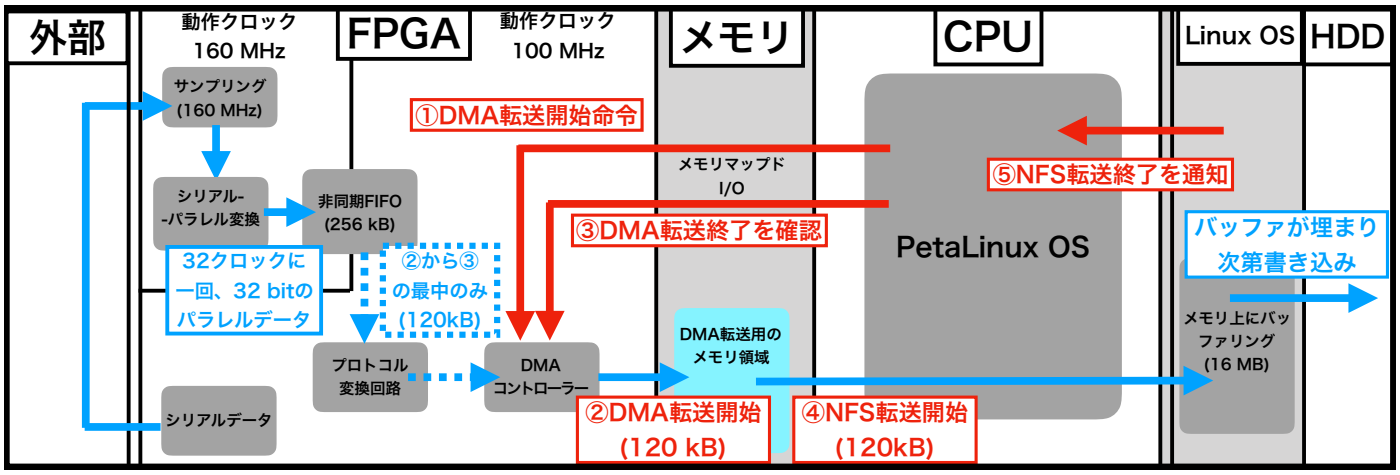


図 3.11: 生成したシリアルデータを、外部からループバックさせ、データ取得するプロセスの簡略図。赤矢印は、データ取得を行う時の手順を表す。①から⑤までが 1 サイクルであり、⑤の後はもう一度①に戻る。青矢印は取得するデータの流れを表す。青点線矢印は、DMA 転送の最中（②と③の間）のみデータが流れることを意味する。

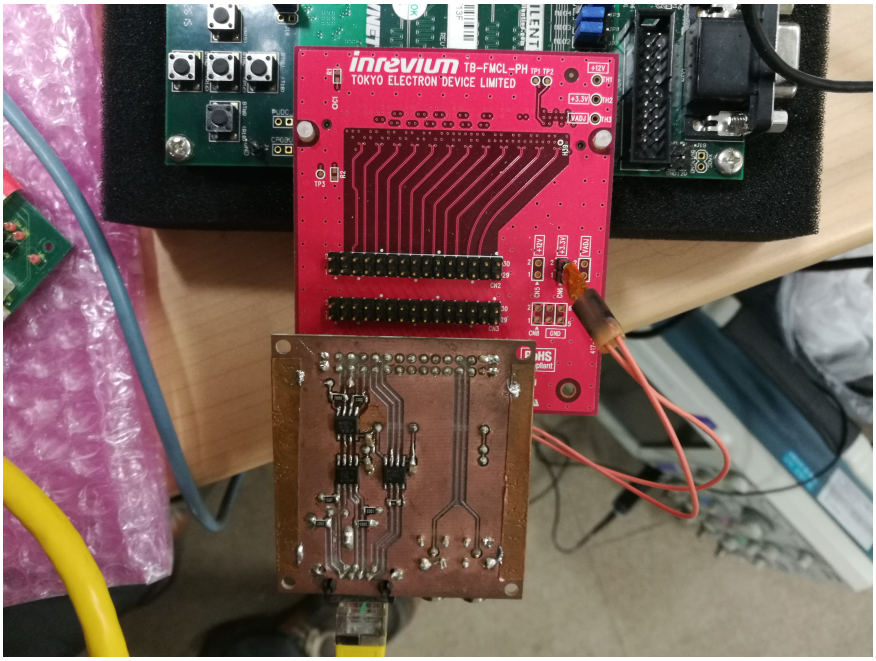


図 3.12: FPGA メザニンカード (FMC) とインターフェースカード [5] を Zedboard に繋いだ写真。赤色の拡張カードが FPGA メザニンカード (FMC) であり、その下につながっているのがインターフェースカードである。

### 3.3.1 インターフェース

本研究では、エラーレートの評価を行うために、Zedboard 外部と FPGA 間で信号をやり取りをする必要がある。そこで、Zedboard 外部と FPGA 間のインターフェースとして、Zedboard 上の FMC コネクタを使用した（図 2.2）。FMC コネクタからさらに Zedboard 外部へ出力するために、図 3.12 のように、FPGA メザニンカード (FMC) とよばれる拡張カード[2]と、FMC 規格の配線をイーサネットコネクタに接続するインターフェースカード [5]（図 3.13）を用いた。

このインターフェースカードは、3.4 章で使用する ASIC、FE-I4 を読みだすために作られたものである。図 3.14 のように、FE-I4 からのデータ、FE-I4 を制御するコマンド、FE-I4 へ供給するクロックの 3 種類の差動信号[3]を、RJ45 コネクタ[4]を使って、Ethernet ケーブルから FPGA メザニンカード (FMC) につなぐ。FPGA メザニンカード (FMC) は、それらの信号を Zedboard の FMC コネクタに接続する。

[2]東京エレクトロンデバイス製 TB-FMCL-PH
[3]差動信号とは、二本の信号線で一つの信号を伝送する方式のこと。電圧が正負反転した二つの信号線の電圧差を使ってデジタル信号を伝送する。電圧差を用いるため、コモンモードノイズに強い。
[4]RJ45 とは、Ethernet ケーブルで使用されているコネクタの規格のこと。

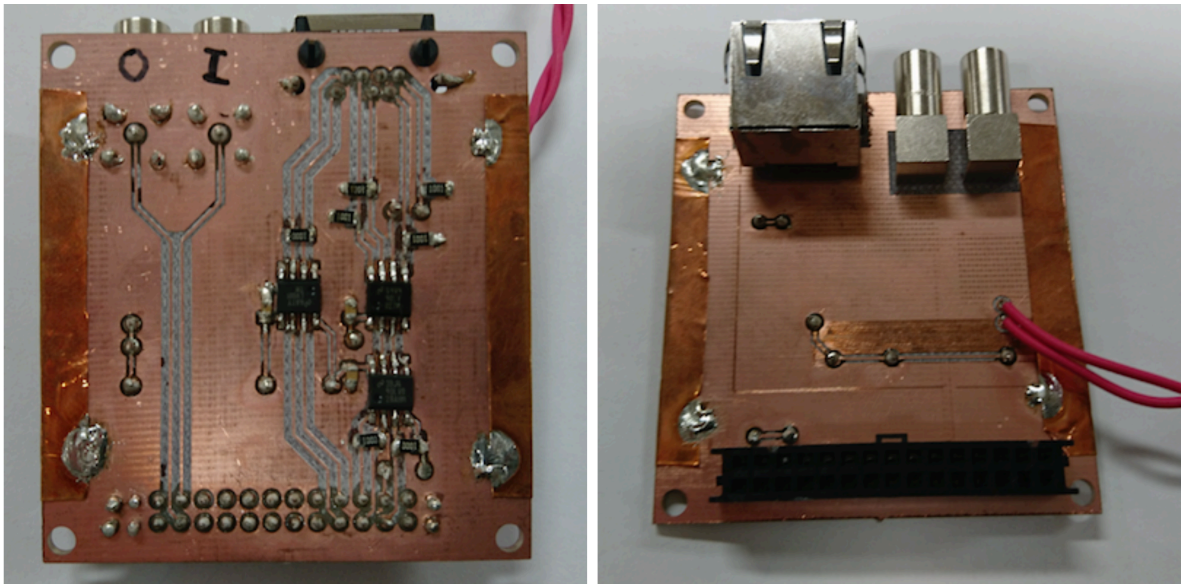

図 3.13: インターフェースカードの写真 [5]。左：裏面。右：表面。

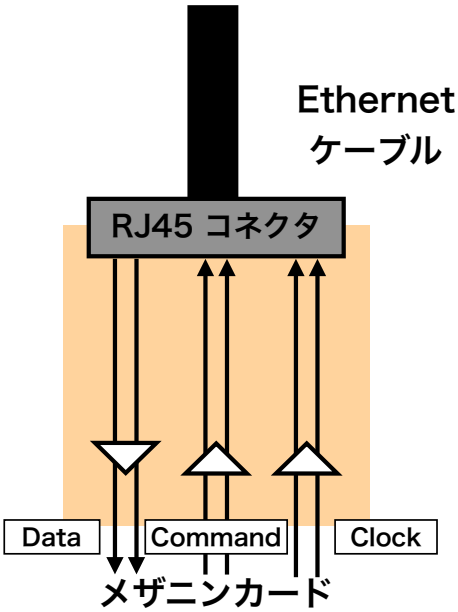


図 3.14: インターフェースカードは、FPGA からの差動信号（コマンドとクロック）と、ASIC からの出力データを Ethernet ケーブルに束ねる役割を果たす。図中に示す三角形は、ASIC 独自の信号規格と、LVDS という信号規格を変換する IC を表している。

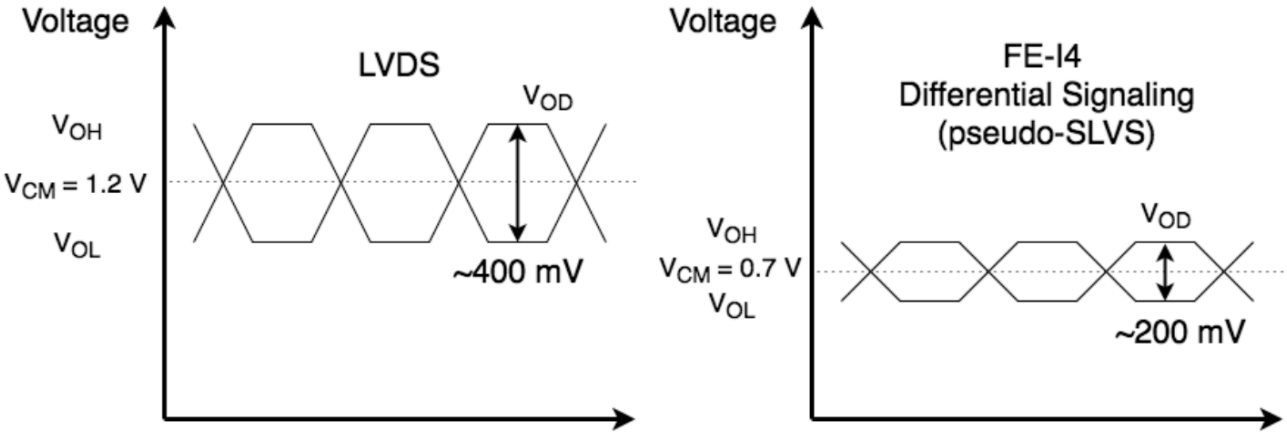


図 3.15: 差動電圧の説明図 [5]。左図は、FPGA の入出力に使用する LVDS 規格。右図は、FE-I4 独自の信号規格。

また、FE-I4 は図 3.15 右のような、独自の信号規格を採用している。インターフェースカードには、FE-I4 独自の信号規格と、図 3.15 左に示す LVDS という信号規格を変換する IC が使用されている。

### 3.3.2 データ転送速度の評価方法

図 3.10 で示したように、FPGA 上で、あるデータレートでパラレルデータを生成し、それを次々に非同期 FIFO へ入力していった。生成したパラレルデータは、本来のシリアル-パラレル変換回路の出力と同じ 32 bit 幅とした。非同期 FIFO に入力したデータを、2.3.1 章で説明したプロトコル変換回路で 120 kB ごとに区切り、DMA 転送を行った。今回、PetaLinux OS が DMA 転送開始命令を DMA コントローラーに送る直前から、次の DMA 転送開始直前までの時間を、繰り返し計測した。転送データ量と、転送するのにかかった時間から、実際にデータを転送できるデータレートを計算した。また、非同期 FIFO への入力レートを変えながらデータ転送速度を測定し、プロトコル変換回路、DMA 転送回路、NFS を用いたデータ取得システムの最大データ転送速度を評価した。

### 3.3.3 データ転送速度の評価結果

例として、生成したデータを FIFO へ 20 MB/s の入力レートで入力した時、120 kB のデータ転送にかかった時間を図 3.16 に示す。この時、120 kB のデータ転送に、平均で（5999±4）$\mu$s かかっていることがわかる。この結果からデータ取得レートを求めると、（20.00±0.01）MB/s であった。

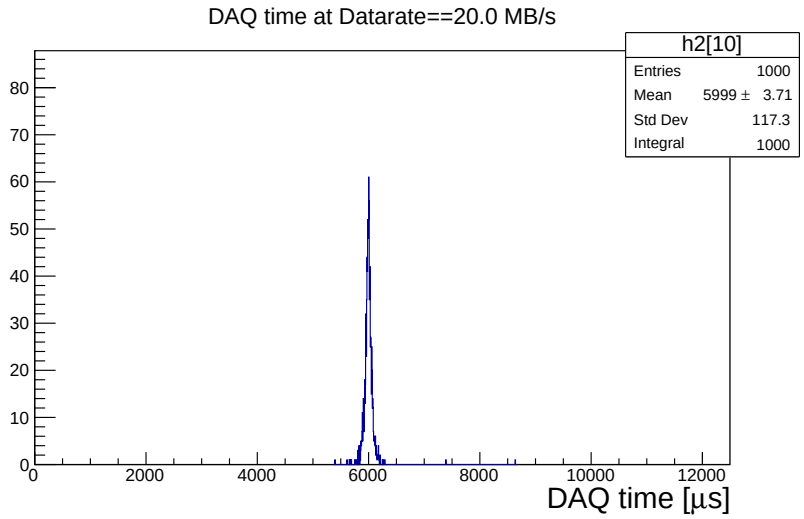


図 3.16: FIFO への入力レートが 20 MB/s の時に、120 kB のデータ転送にかかった時間のヒストグラム。横軸は、生成したデータ 120 kB を転送するのにかかった時間を表す。

FIFO への入力データレートを変えながら、実際のデータ取得レートを測定した結果を図 3.17 に示す。グラフから、34 MB/s 以上のデータレートでデータを入力しても、34 MB/s 以上の速度でデータを取得することはできないことがわかる。34 MB/s 未満の入力データレートに対しては、DAQ の実行データレートと入力データレートが一致している。

以上のことから、プロトコル変換回路、DMA 転送回路、NFS を用いたデータ取得システムは、最大で 34 MB/s の入力データレートでデータ取得ができることがわかった。

しかし、データ転送レート 34 MB/s は、3.1 章で調べた NFS 転送の速度 28 MB/s を上回っている。3.1 章では、Linux OS は、NFS によって Zynq からデータを受け取り続けていた。一方で、今回は DMA 転送と NFS 転送を順番に行なっているため、DMA 転送が行われている間に、Linux OS が NFS で受け取ったデータを HDD に書き込む準備を行うことができる。このことが、3.1 章を上回る結果を得た原因になりうると考えている。

### 3.3.4 エラーレートの評価方法

まず、インターフェースカードのクロック配線を用いて、Zynq の FPGA 部で生成した 40 MHz のクロック信号を、Zedboard 外部へ 3.3.1 章で説明した LVDS という差動信号の規格で出力した。この時、図 3.18 のように、インターフェースカードからは、3.3.1 章で説明した FE-I4 独自の信号規格で出力

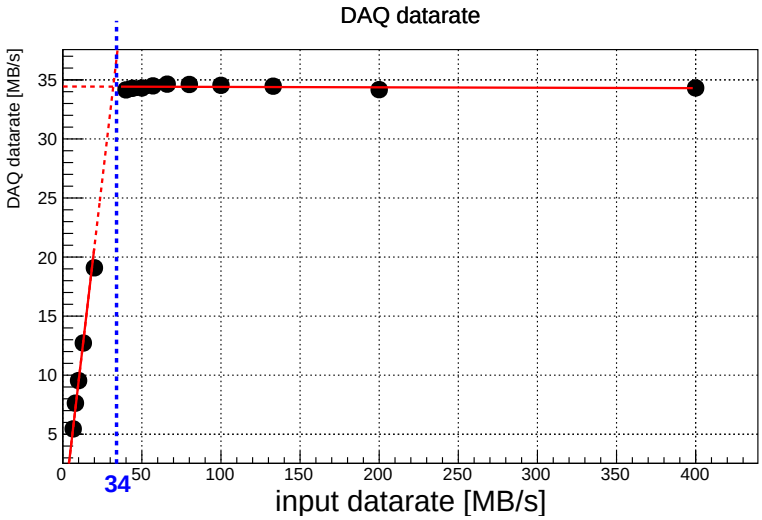


図 3.17: DAQ の実行データレートを、入力データレートごとに測定したグラフ。横軸：生成したパラレルデータのデータレート。縦軸：DAQ の実行データレート。赤線は、それぞれ横軸 0〜35、35〜400 MB/s の範囲で Fitting した結果である。赤点線は、それぞれの Fittig 結果を外挿したものである。青点線は、横軸が 34 MB/s であることを表し、二つの赤点線の交点を通る。

される。

外部へ出力したクロックは、図 3.19 のように、Ethernet ケーブルの配線を分離し、ループバックさせてデータを伝送する配線に入力し、擬似的に外部からやってきたデータとみなして扱った。その時の実際の様子を図 3.20 の写真に示す。

ループバックして FPGA に入力した 40 MHz クロックを、擬似的に 160 MHz のシリアルデータとみなし、図 3.21 のように、データ処理回路にて 160 MHz のクロックでサンプリングした。この時、40MHz のクロックを 160 MHz でサンプリングすると、160 MHz のクロックに同期して 1100 のパターンを繰り返すシリアル信号に変換される。今回、40 MHz のクロックを 160 MHz のシリアルデータとみなして 160 MHz でサンプリングした理由は、ループバックした配線を 3.4 章で使用する FE-I4 に取り替えるだけで、全く同じ FPGA 上の回路で FE-I4 のデータ取得を行うことができるからである。

サンプリング結果として得られた 160 MHz のシリアルデータは、2.3.1 章で説明したシリアル-パラレル変換回路によって、32 クロックごとに、32 bit のパラレルデータに変換される。160 MHz の動作クロックに同期して、32 クロックに一回 32 bit で出力されるパラレルデータを、非同期 FIFO を用いて、DMA 転送回路を動かす 100 MHz の動作クロックに同期させる。その後 DMA 転送を行い、データを PetaLinux OS がアクセス可能なメモリへ転送する。最後に、NFS を用いてコンピュータへデータを転送する。DMA 転送

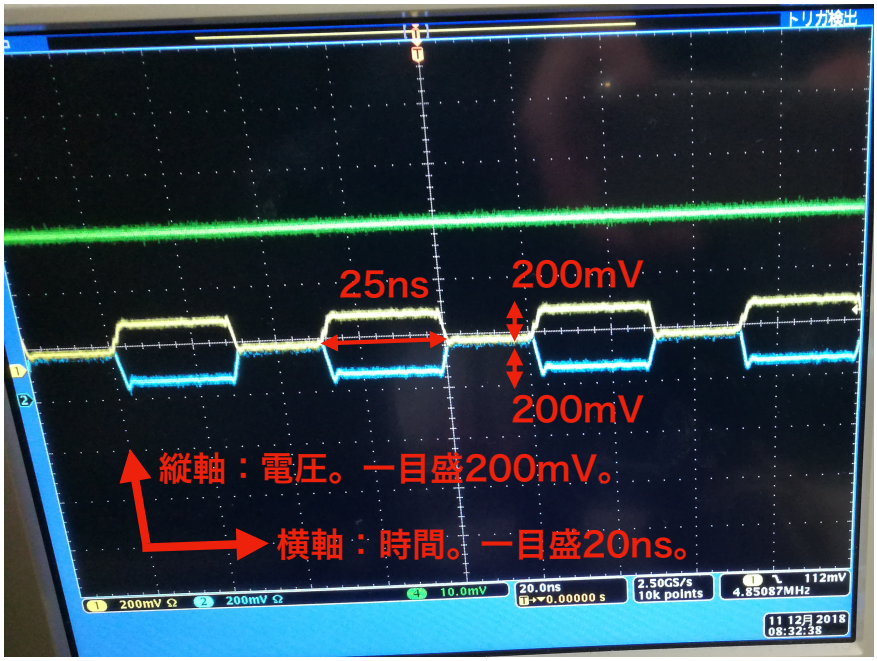


図 3.18: Zedboard から出力した 40 MHz 差動クロックをオシロスコープで見た様子。写真では、二つの作動信号のグラウンドレベルをずらしているが、図 3.15 右に示す FE-I4 独自の信号規格となっている。

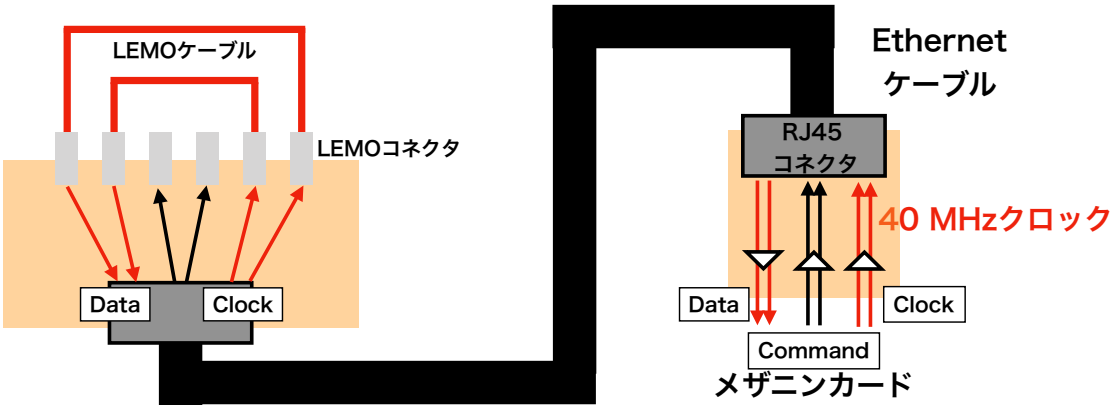


図 3.19: 40 MHz のクロックを、クロック配線から Zedboard 外部へ出力し、データ配線へ繋ぎ、Zedboard へループバックする。ループバックする際は、RJ45 コネクタと LEMO コネクタを使用して、Ethernet ケーブルの配線を LEMO ケーブルへ変換するボード [5] を用いた。

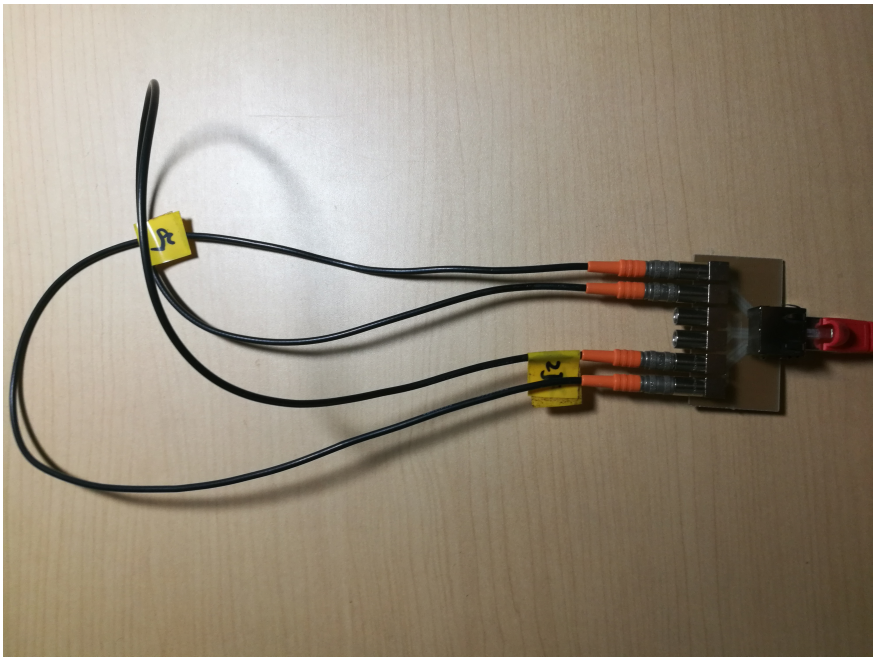

図 3.20: Ethernet ケーブルの配線を分離し、クロックをデータの配線に入力する様子。

をし、NFS を用いてコンピュータへデータを転送する、というサイクルを繰り返すして、FIFO に入ったデータを断続的にデータ転送する。

サンプリングの際、図 3.22 のように、サンプリングクロックがサンプリングデータのエッジ[5]と重なると、間違ったサンプリング結果になることがある。そのため、ループバックケーブルの長さを調節し、データ信号のエッジがサンプリングクロックと重ならないように、データ信号の位相をずらした。ループバックケーブルの長さを変えながらデータを取得し、コンピュータに保存されたデータのパターンを、生成したデータのパターンと照らし合わせ、エラーレートを評価した。

### 3.3.5 エラーレートの評価結果

データ転送のクオリティは、ビットエラーレート（BER：Bit Error Rate）という指標によって評価できる。

$$\mathrm{BER} \equiv \frac{\text{Number of Bit Errors}}{\text{Number of Transfer bits}}$$

ビットエラーの数が 0 である場合、Appendix A の方法に従って、以下のビットエラーレートの上限値を求めることができる。ここで、C.L. は信頼水準の

[5]デジタル信号の立ち上がりや立ち下りのこと

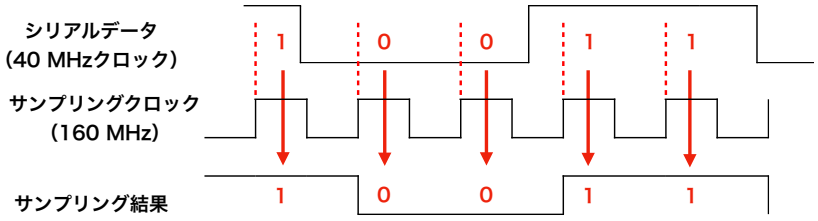


図 3.21: 40 MHz クロックを擬似的に 160 Mbps のシリアルデータとして扱って 160 MHz クロックでサンプリングする様子。

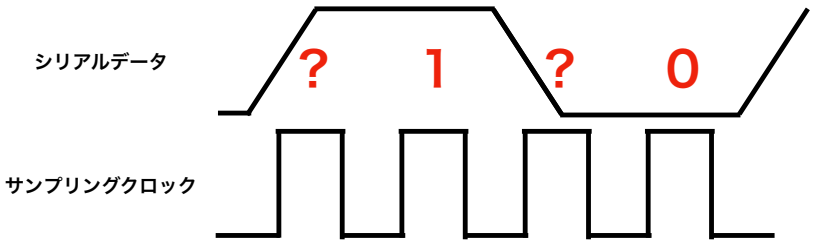


図 3.22: 良くないサンプリングの例。サンプリングクロックと、サンプリング対象のエッジのタイミングが一致すると、デジタル信号の位相揺らぎなどの影響でサンプリング結果が不定になる。

ことを表す。

$$\mathrm{BER}_{\mathrm{upperlimit}} \equiv \frac{-\ln(1 - C.L.)}{\text{Number of Transfer bit}}$$

ループバックケーブル長を変えて、サンプリングするデータの位相を 160 MHz サンプリングクロック約 1 周期分（6.25 ns）ずらしながら、データのエラーを測定した結果を表 3.1 に示す。一列目は、ループバックケーブルによるディレイ時間、二列目はビットエラー数、三列目はトータルの転送ビット数を表す。表 3.1 から、ループバックケーブルによるディレイが、1 ns 付近のところ

表 3.1: ループバックケーブルごとのビットエラー数

| ディレイ | ビットエラー数 | 転送ビット数 |
|---|---|---|
| 0.5 ns | 0 | 3.7 Gbit |
| 0.9 ns | 366304 | 0.17 Gbit |
| 1.0 ns | 1 | 2.2 Gbit |
| 1.5 ns | 0 | 1.7 Gbit |
| 2.0 ns | 0 | 3.4 Gbit |
| 2.5 ns | 0 | 4,5 Gbit |
| 3.0 ns | 0 | 23.1 Gbit |
| 3.5 ns | 0 | 2.1 Gbit |
| 4.0 ns | 0 | 2.3 Gbit |
| 4.5 ns | 0 | 2.9 Gbit |
| 5.0 ns | 0 | 6.1 Gbit |
| 5.5 ns | 0 | 7.4 Gbit |
| 6.0 ns | 0 | 3.9 Gbit |
| 6.5 ns | 0 | 4.4 Gbit |

でビットエラーの数が大きくなっているため、このディレイでは、先述した正しくサンプリングできない位相となっていると判断し、この位相を避けた。

例として、ケーブルディレイが　3.0 ns の時の測定結果でビットエラーレートの上限を計算する。信頼水準 95%（C.L.=0.95）を仮定すると、約 23 GBit の転送中ビットエラーが 0 だったため、

$$\mathrm{BER}_{\mathrm{upperlimit}} = 1.3 \times 10^{-10}$$

となる。

この結果から、データ処理回路は、10GBit 程度のデータであれば、外部からの信号をビットエラーなく、サンプリングできることがわかった。この時、DMA 転送したデータ数と実際に FIFO から出力されたデータ数が一致することを DMA 転送毎に確認し、DAQ を行う際にデータの欠損が起きていな

いことを保証した。

## 3.4 総合評価

本節では、Zynq を用いた DAQ システムを使って ASIC からのデータの取得を行い、開発した Zynq を用いた DAQ システムが、実際にデータ取得に使用することができることを確認する。また、実際に ASIC のデータ取得を行なった時の、データ損失率、エラーレートを評価する。

### 3.4.1 読み出し対象

本研究では、データを読み出す対象として、ATLAS 実験で使われている FE-I4 という ASIC を使用した。この ASIC は、ATLAS 検出器のうち、内部飛跡検出器として使われるピクセルサイズ 50 $\mu$m × 250 $\mu$m、チャンネル数 26880 のピクセルセンサーを読み出すために開発されたものである。

FE-I4 は、図 3.23 のように、外部から 40 MHz の差動信号を動作クロックとして受けて動作し、160 Mbit/s の速度でデータを差動信号として出力する。また、差動コマンドの配線から特定のパターンのシリアル信号を入力し、FE-I4 のコントロールを行う。

FE-I4 は待機状態ではデータを出力しないが、外部からコマンドを入力して初期化すると、Idle state に移行して 160 Mbit/s で表 3.2 に示す Idle パターンの信号を出力する。FE-I4 の出力データは、8 bit のデータを 10 bit で送る

表 3.2: FE-I4 の Idle パターン [6]

| | 8 bit | 10 bit (RD-) | 10 bit (RD+) |
|---|---|---|---|
| Idle パターン | 001 11100 | 001111 1001 | 110000 0110 |

8b10b という方式でコーディングされている。今回の Idle パターンの場合、表 3.2 の RD−,RD+ を交互に出力する。本研究ではこの Idle パターンをデコードせずに、データとして取得した。読み出したデータを、RD− と RD+ を合わせた合計 20 bit 単位で解読した。

### 3.4.2 評価方法

本研究では、Zynq DAQ システムから差動コマンドを送信する機能をまだ実装していない。そのため、今回は既存の FE-I4 読み出し DAQ システム [5] を用いて FE-I4 へコマンドを送り初期化を行った。

FE-I4 を初期化に成功した後、既存の DAQ システムと FE-I4 の間の接続に用いた Ethernet ケーブルを、既存の DAQ システムから外す。外した Ethernet

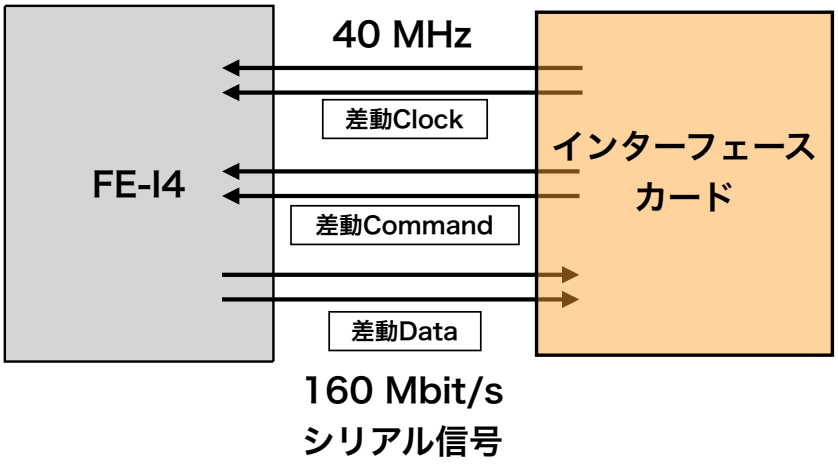


図 3.23: FE-I4 の入出力。

ケーブルを、Zedboard を用いた DAQ システムに接続し、Zynq DAQ システムに FE-I4 の Idle 信号を差動信号として入力した。

図 3.24 は、Zynq を用いた FE-I4 読み出しの全体像である。FPGA 上で生成した 40 MHz のクロックを、差動信号へ変換し、インターフェースカードを介して FE-I4 へ供給した。FE-I4 の出力する 160 Mbit/s シリアルデータは、インターフェースカードを経由して、Zynq 上の FPGA に入力した。この時、40 MHz のクロックをつくる回路と、差動信号の入出力を行う回路には、Xilinx 社の提供する IP Core を用いた。

Zynq を用いた DAQ システムの、データ取得プロセスを図 3.25 に簡単に示す。FPGA へ入力されたシリアルデータは、まず 160 MHz でサンプリングした。サンプリングされたシリアル信号を、シリアル-パラレル変換回路で、32 クロックごとに 32 bit のパラレルデータへ変換した。パラレルデータに変換したのち、256 kB の容量を持つ非同期 FIFO にパラレルデータを入力した。このとき、FIFO の almost full 信号は無視して入力したため、FIFO が飽和しているときに新たに入力したデータは損失してしまう。プロトコル変換回路は、PetaLinux OS が DMA 転送回路に転送開始命令を送ると、FIFO の中にあるデータを取り出しつつ、DMA 転送を行うプロトコルに変換し、DMA 転送回路に入力する仕様にした。

PetaLinux OS が DMA 転送回路に転送開始命令を送ると、プロトコル変換回路を経由して、FIFO の中のデータを OS がアクセス可能なメモリへ 120 kB

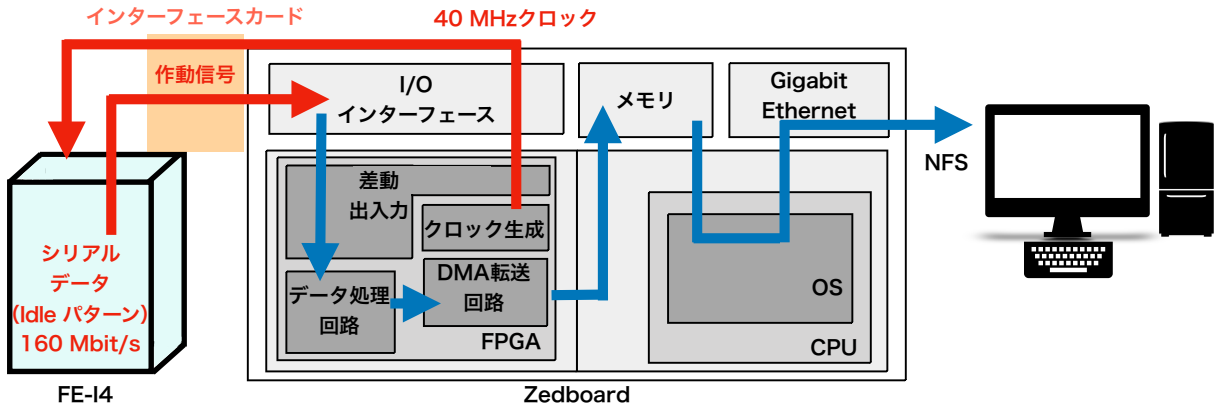


図 3.24: Zynq を用いた FE-I4 読み出しの全体像。FPGA 上で生成した 40 MHz のクロックを動作クロックとして、FE-I4 へ供給する。Zynq は、FE-I4 からの 160 Mbit/s のシリアルデータを FPGA で受け取る。FPGA と Zedboar 外部との信号は、差動信号で送受信を行う。

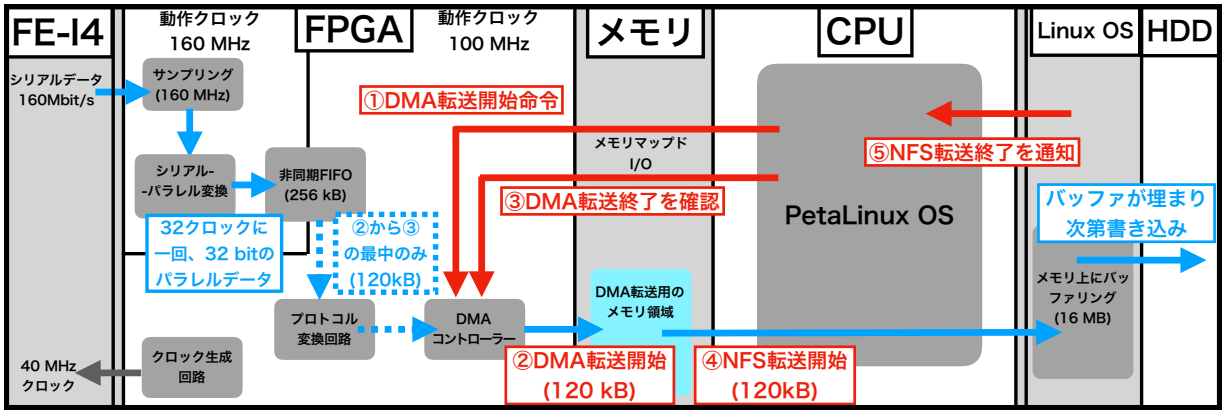


図 3.25: Zynq を用いた DAQ システムの、データ取得プロセスの簡略図。赤矢印は、データ取得を行う時の手順を表す。①から⑤までが 1 サイクルであり、⑤の後はもう一度①に戻る。青矢印は取得するデータの流れを表す。青点線矢印は、DMA 転送の最中（②と③の間）のみデータが流れることを意味する。

ずつ DMA 転送した。一回の DMA 転送量の 120 kB は、3.2 章の結果をもとに決定した。

PetaLinux OS は、DMA 転送回路の制御用レジスタを読み続けて、DMA 転送が終了したことを認識する。DMA 転送が終了すると、PetaLinux OS は NFS を用いてメモリ上のデータを LLinux OS へ転送する。この時、3.1 章の結果から、Linux OS が HDD へデータを書き込む前に行うバッファリングのサイズは 16 MB とした。

一回データ取得を行なっただけでは 120 kB しかデータを取得しないため、Linux OS のバッファリングのサイズ 16 MB に届かず、HDD にデータが保存されない。そこで、PetaLinux OS が Linux OS に NFS でデータを転送し終わると、DMA 転送開始命令を DMA 転送回路に送り、もう一度 DMA 転送を再開する。

一回のデータ取得が終了するともう一度 DMA 転送からやり直す、というサイクルを続けると、累計転送データ量が Linux OS のバファリングサイズを超えたところで、Linus OS がバッファリングしたデータを HDD へ書き込み始める。そのままデータ取得を続けると、データを HDD へ保存しながらのデータ取得が行える。

今回、NFS 転送終了から、次のデータ取得サイクルの NFS 転送終了までの時間（図 3.25 の①直前から⑤直後までの時間のこと。２サイクル目以降は、⑤直後から次のサイクルの⑤直後までの時間に当たる。）を繰り返し計測し、転送データ量 120 kB ごとに、データ取得に要した時間を測定した。１サイクルの転送データ量を、データ取得に要した時間で割ることで、Zynq DAQ システムが FE-I4 を読み出す時の、データ取得レートを評価した。

また、コンピュータへ保存されたデータを表 3.2 の Idle パターンと照らし合わせ、Zynq DAQ システムによる FE-I4 読み出しのエラーレートを評価した。

### 3.4.3 結果

FE-I4 の Idle 信号 120 kB のデータを取得するのにかかった時間を、図 3.26 に示す。120 kB のデータ取得に、平均で（6161±3）$\mu$s かかっていることがわかる。この結果からデータ取得レートを求めると、（19.94±0.01）MB/s であった。

今回、出力データレート 20 MB/s の FE-I4 を読みだしているため、データの損失がなければ、Zynq を使ってのデータ読出し速度も 20 MB/s になるはずであった。しかし実際には、入力データに対して、1-((19.94±0.01) MB/s)/(20 MB/s) =（0.30±0.05）%のデータ損失があったことになる。

図 3.26 を見ると、データ取得に、平均の 30 倍以上の時間がかかっている時があることがわかる。

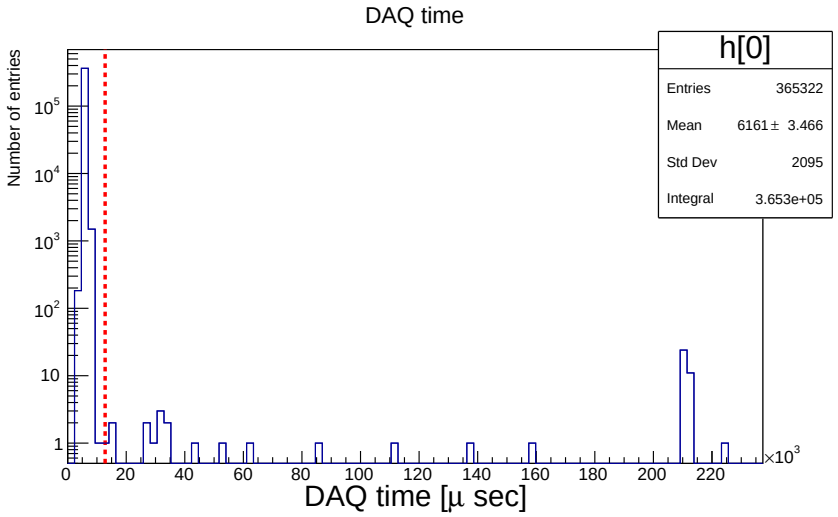


図 3.26: 1 サイクル 120 kB のデータ取得にかかった時間のヒストグラム。横軸は、120 kB の Idle 信号を転送するのにかかった時間を表す。赤点線は 256 kB の FIFO が FE-I4 のからのデータで飽和するのにかかる時間（12800 $\mu$s）を示す。

本来、2.3.1 章で述べたように、DAQ システムがデータをコンピュータへ送っている間 (図 3.25 の①、④、⑤の期間のこと)、図 3.25 にある 256 kB の非同期 FIFO がデータの受け皿となる。そのため、データ取得 1 サイクルにかかる時間が、FIFO が溢れない程度であればデータの損失は起こらない。

しかし、FE-I4 のデータレート 160 MBit/s と、FIFO の大きさ 256 kB から計算すると、FIFO がデータの受け皿として機能できるのは、最大でも 12800 $\mu$s である。そのため、データ取得 1 サイクルに 12800 $\mu$s 以上かかると、FIFO が溢れてデータが損失すると考える。

そこで、図 3.26 のうち、データ取得にかかった時間が 12800 $\mu$s 未満のものを選別し、図 3.27 に示す。図 3.27 に示すイベントでは、FIFO が飽和せず、データが損失しない。この時、120 kB のデータ取得に、平均で（6138.0±0.3）$\mu$s かかっていることがわかる。この結果からデータ取得レートを求めると、（20.02±0.01）MB/s であった。

以上のことから、FIFO が飽和してしまっていることが、（0.30±0.05）%のデータを損失している原因であることがわかる。

最後に、読みだした FE-I4 のアイドル信号を、表 3.2 に示す Idle パターンと照らし合わせ、エラーレートを評価する。

図 3.25 の、非同期 FIFO 直後にあるプロトコル変換回路で、FIFO から何 byte 取り出して DMA 転送を行うかを決定している。本研究では 120 kB ごとに DMA 転送を行なった。そのため、メモリマップド I/O を通してプロトコル変換回路のレジスタを読むことで、DMA 転送回路に入力したデータ数

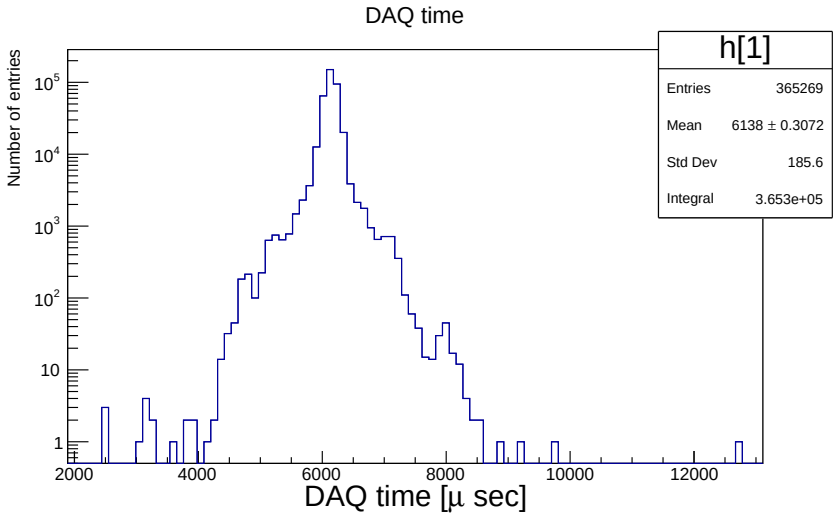


図 3.27: 1 サイクル 120 kB のデータ取得にかかった時間が 12800 $\mu$s 未満のものを選別したヒストグラム。横軸は、120 kB の Idle 信号を転送するのにかかった時間を表す。

がわかる。

さらに、DMA 転送を行うデータの最後に任意のデータパターンを付与した。DMA 転送後に、メモリにあるデータを、DMA 転送先アドレスの始点から、付与したデータパターンまでのデータ数を数えることで、FIFO から取り出したデータを欠けることなく転送できていることを保証した。

最終的に、データ取得レートを評価する際に取得したデータを、表 3.2 の Idle パターンと照らし合わせ、データにエラーがあるかどうか確認した。その結果、取得したデータ 360 GBit は、すべて Idle パターンと一致することを確認した。

前節と同様にビットエラーレートの上限値を計算すると

$$\mathrm{BER}_{\mathrm{upperlimit}} = 8.3 \times 10^{-12}$$

が得られた。

以上の結果をまとめると、本研究で開発した Zynq DAQ システムを用いて、出力データレート 20 MB/s の FE-I4 を読み出したところ、データ取得レート（19.94±0.01）MB/s、入力データ損失が、(0.30±0.05）%、ビットエラーレートの上限 $8.3 \times 10^{-12}$ の結果を得た。

このデータ損失は、データ取得を繰り返す際、平均の数十倍もの時間を要することがある、ということに原因がある。また、例として FE-I4 を 1000 秒間読み出す場合に、合計で約三秒間データ取得できない時間が存在してしまうことを意味する。このことは、外部メモリなどを用いて、FIFO の容量を

大きくすることにより対処可能である。

# 第4章　結果のまとめと考察

本研究では、Zynq を用いた DAQ システムを開発し、その DAQ システムのデータ転送速度、転送したデータのエラーレートを評価した。これまでの結果を表 4.1 にまとめる。

表 4.1: 本研究の結果

| | FE-I4 | FPGA | メモリ | CPU | HDD |
|---|---|---|---|---|---|
| メモリ ⟶ HDD | | | 28 MB/s | | |
| FPGA ⟶ メモリ | | 400 MB/s | | | |
| FPGA ⟶ HDD | | 34 MB/s | | | |
| FE-I4 ⟶ HDD | 19.94 MB/s | | | | |

まず、3.1 章では、NFS という手段を用いて PetaLinux OS から Linux OS へデータ転送を行い、その転送速度を測った。その結果、最大 28 MB/s のデータレートでデータ転送を行った。

次に、3.2 章で、Zynq の FPGA 部から、CPU がアクセス可能なメモリへデータを転送した。その転送速度を測ったところ、最大 400 MB/s という結果を得た。

3.3 章では、DMA 転送、NFS 転送を通して、FPGA 上のデータを HDD に保存し、データ取得レートを測定した。その結果、最大で 34 MB/s のデータレートでデータ取得ができるという結果を得た。この結果は、3.1 章の結果が 28 MB/s であったのに対して矛盾する。本研究では、3.1 章よりも早い速度で、データ取得が行えたことの原因特定に至っていない。また、Zedboard 外部からの信号をデータ取得した時のビットエラーレートを測定した。その結果、ビットエラーレートの上限値 $\mathrm{BER}_{\mathrm{upperlimit}} = 1.3 \times 10^{-10}$ を得た。

3.4 章では、実際に実験で使われてる ASIC のデータを読出し、Zynq を用いた DAQ システム全体の総合評価を行った。出力データレート 20 MB/s の ASIC の信号を読出し、データ取得レート 19.94 MB/s、データ損失率 0.3%、ビットエラーレートの上限値 $\mathrm{BER}_{\mathrm{upperlimit}} = 8.3 \times 10^{-12}$ でデータ取得を行なった。

本来、Gigabit Ethernet の転送速度（120 MB/s）と、データを保存する HDD の書き込み速度（今回用いたものは 100 MB/s）による制限だけでは、最大で 100 MB/s のデータ取得レートが達成できるはずである。しかし、今

回の研究結果では、最大 34 MB/s の速度でしかデータ取得を行えなかった。

3.1 章の測定結果からわかるように、NFS 転送の遅さがデータ転送速度にかける制限は大きく、今回の、Zynq を用いて NFS でデータ転送を行うシステムのデータ取得レートは、ほとんど NFS の速度により制限されている。

## 4.1 今後の課題

本研究で開発した Zynq を用いた DAQ システムは、NFS 転送によって大きく制限されているため、Linux OS へのデータ転送速度を改善すれば、さらに高速なデータ読出しができると考える。そこで、NFS 転送速度を向上させる方法について以下の 2 点を議論する。

1. キャッシュメモリの使用
   本研究ではデータを RAM メモリに DMA 転送していたところを、キャッシュメモリという、より高速にアクセスできるメモリデバイスに DMA 転送する。

2. 64 bit CPU を使用
   今回研究で用いた 32 bit CPU を搭載した Zynq を、64 bit CPU を搭載した Zynq へ変更する。

#### キャッシュメモリの使用

まず、一つ目のキャッシュメモリを用いる方法を述べる。本研究で用いた DMA 転送は、FPGA から、CPU からアクセス可能な、DDR3 という規格の RAM メモリへデータを転送した。キャッシュメモリとは、容量が小さいものの、RAM メモリよりも CPU に近く、さらにアクセス速度も大幅に速いメモリである。

少し特殊な手順が必要であるが、Zynq 製品は、DMA 転送でキャッシュメモリにデータを書き込むことができる。当然、RAM メモリよりも読出し速度が格段に早いので、メモリ上のデータを NFS 転送するとき、転送速度を向上を見込める。

ただし、元々のメモリ読出し速度が速い[1]ため、この方法による劇的な性能向上は難しい。

#### 64 bit CPU を使用

次に、64 bit CPU を用いる方法について述べる。本研究で用いた Zynq の CPU は、表 2.1 に載せたデュアル ARM Cortex-A9 であり、これは 32 bitCPU[2]で

[1] 一般的な DDR3 メモリは遅くても数 GB/s の転送速度
[2] 32 bit CPU とは、CPU が扱うことのできる bit 幅が 32 bit の CPU のことである。

ある。そのため、今回用いた PetaLinux OS も、NFS 転送も、全て 32bit 環境で動作している。

NFS 転送の性能に、この 32 bit 環境であることが影響している。NFS は現在 version4 が最新であるが、32 bit 環境では、version2 相当の性能しか発揮できない。実際に、今回使用した NFS では、2GB 以上のデータを一回で転送することができなかった。この転送限界は、NFS version2 にあったもので、32 bit 幅のメモリアドレスしか扱えないことが原因であった。転送限界と同様に、NFS 転送のパケットサイズも 32 bit 環境では、NFS version2 相当の 8kB が上限である可能性がある。

もしこの予想が正しければ、一回で 120 kB を NFS 転送するとき、8 kB のパケットに分けて複数回送ることになり、転送のオーバーヘッド[3]が相対的に大きくなり、大幅な転送速度の低下が起きる。

そこで、今回用いた Zynq®-7000 シリーズではなく、Zynq UltraScale+などの、64 bit CPU を搭載したモデルの Zynq を用いれば、32 bit CPU による強力な制約を克服することができるはずである。

## 4.2 Zynq DAQ の応用

Zynq を使用した DAQ システムの大きな特徴は、

- DAQ の過程で CPU を経由することができる
- メモリマップド I/O によって、CPU から FPGA の操作が高速かつ簡単に行えること

の二点である。これらの特徴を組み合わせて応用すると、例えば、高速なデータ読出しの際に、CPU 上で簡単な解析を行い、必要なデータを選別して、データサイズを減らすことができる。本研究で使用した Zynq に搭載されている CPU は、32 bit CPU で、動作クロックが 667 MHz なので、32bit のデータに対し条件判断処理をするのに 100 クロックかかったと仮定すると、1 秒間に約 25MB のデータに対して条件判断処理をすることができる。つまり、25 MB/s 未満の入力データレートの DAQ ならば、すべてのデータに対し、簡単な条件処理を行うことができることを意味する。

当然、100 クロック以上かかるような重い計算を行うと、処理可能なデータレートは下がっていく。その場合、FPGA 内でデータのヒストグラム化などを行なって軽量化したものを、定期的に CPU でも解析する、などを行えば、より高速なデータレートに対してもデータの軽量化を図ることができる。

ハイエンドモデルの Zynq 製品では、動作クロックが 1.5GHz のクワッドコア CPU を使っている。つまり、本研究で用いたものの倍以上速く計算が

[3]オーバーヘッドとは、実際のデータ転送にかかる時間とは別に、プロトコルの処理などにかかった時間のことである。

でき、さらに倍の数の並列処理を行うことができる。これだけの性能があれば、件判断処理をするのに 100 クロックかかったとしても、50MB/s 近くの高データレートでもすべてのデータに対し、簡単な条件処理を行うことができる可能性がある。

# 第5章　結論

従来のDAQシステムにおいて、FPGAは検出器からのデータを受け取り、コンピュータへ転送を行うインターフェース役を担うことが多かった。この役目を果たすためには、コンピュータとの通信プロトコルに従ってデータ転送するデジタル回路をFPGA上に開発することが必須であった。

本研究では、Zynqを用いて、コンピュータへのデータ転送回路をFPGAで開発する必要のない、新しいDAQシステムを開発し、その性能評価を行なった。

その結果、NFSの速度によって制限されているものの、Zynqを用いたDAQシステムは、最大約34 MB/sのデータレートまでならDAQが可能であるという結論を得た。ただし、この値はNFS転送速度の評価結果である28 MB/sに矛盾する。このことは、データを受け取るコンピュータが、DMA転送が行われている間に、受け取ったデータを整理できることに原因があるのではないかと考えている。キャッシュメモリを使用したり、64 bit CPUを搭載したZynq製品を用いることで、さらなる性能の向上が可能である。

最後に、FE-I4という出力データレートが20 MB/sのASICを用いて、Zynq DAQが実際のデータ読み出しに使えることを確認した。この時、データ取得レート（19.94±0.01）MB/s、データ損失率（0.30±0.05）%、ビットエラーレートの上限値 $\mathrm{BER}_{\mathrm{upperlimit}} = 8.3 \times 10^{-12}$ であった。

本研究の結果、Zynqを用いることで、簡単にコンピュータへデータ転送ができるDAQシステムが開発できた。その性能も、最大読出し速度34 MB/sを達成し、実際に現役のASICのデータを読み出すことができることを確認した。

# 参考文献


[1] ”SiTCP” 内田智久 , IPNS, KEK http://research.kek.jp/people/uchida/technologies/SiTCP/

[2] ”SiTCP 説明書” 内田智久 Electronics system group, IPNS, KEK
http://research.kek.jp/people/uchida/technologies/SiTCP/doc/SiTCP.pdf

[3] ”ZedBoard” Xilinx, Inc.
https://japan.xilinx.com/products/boards-and-kits/1-8dyf-11.html

[4] ”Peta Linux” Xilinx, Inc.
https://japan.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[5] 澤田 恭範（2018）大阪大学大学院理学研究科物理学専攻博士前期課程
修士論文 HL-LHC ATLAS 実験に向けた シリコンピクセル検出器用 データ収集システムの開発

[6] The FE-I4B Integrated Circuit Guide - CERN Indico
https://indico.cern.ch/event/261840/contributions/1594374/attachments/462649/641213/FE-I4B_V2.3.pdf

# 謝辞

大学院博士前期課程の２年間、多くの方のお世話になりました。この場を借りて、お礼申し上げたいと思います。

山中卓教授には、物理学のみならず、研究者としての在り方そのものを学びました。研究に臨む姿勢をはじめ、修士論文の書き方や、学会での発表資料の作り方等多くのことをご指導いただきました。ありがとうございました。

南條創准教授には、研究の方向性や、研究の段取りなど多くの事柄について多くのアドバイスをいただきました。特に、本論文の作成にあたり、非常に多くの助言をいただきました。本当に感謝しております。

廣瀬穣助教授には、FPGA を用いた回路構成について多くの意見をいただきました。FPGA に関する知識や経験が足りずに困っていた時、エキスパートである廣瀬穣助教授がいたことで非常に助けられました。ありがとうございました。

高エネルギー加速器研究機構（KEK）の花垣和範教授には、本研究から離れた視点からの助言をいただきました。ありがとうございました。

KEK の外川学准教授には、学部生時代に、検出器の扱いをはじめとした実験の基礎を教わりました。感謝しております。

そのほか、多くの ATLAS 共同研究者の方々から助言や助力をいただいたこと、感謝の念に堪えません。

研究室の先輩である矢島和希さんには、使用する機器の扱いや、研究に関する知識や経験に大いに助けられました。特に、FPGA の開発に関する助言をいただけたことは本当に助かりました。また、本研究で用いた FE-I4 を動作させる際にも助力をいただきました。本当にありがとうございました。

山中卓研究室の研究員である清水信宏さん、小寺克茂さんには、その豊富な経験に基づく助言を多くいただきました。

山中卓研究室の卒業生である、澤田恭範さん、山元大生さん、佐藤友太さん、西宮隼人さんには、な事務手続きや、講義に関する助言をいただきました。

同期の原宜広君、真利共生君には、学会発表などでの資料作りの時に、個人的に助言をもらうなど大いに助けられました。

後輩の大杉真優さん、山家谷昌平君、WICKREMASINGHE LAKMIN 君、沖本直哉君、白石諒太君には、その研究に打ち込む姿勢に良い刺激をもらいました。

秘書の藤阪千衣さんには、備品の購入や必要書類の提出など、事務的に手

続きに関して非常にお世話になりました。
　最後に、これまで支えてくださった両親に深謝申し上げます。本当にありがとうございました。

# 付 録A　Bit Error Rate

ここでは、3.3.5 章で使用したビットエラーレート（BER：Bit Error Rate）の上限値を導出する。

ビットエラーレートとは、受信したデジタル信号の論理（0 か 1 か）を誤って受信してしまう割合を表す指標である。論理を誤って受信してしまうことをビットエラーと呼ぶ。

ビットエラーレートは以下の式のように定義され、誤って受信したデータ数を総受信データ数で割ることによって計算できる。

$$\mathrm{BER} \equiv \frac{\text{Number of Bit Error}}{\text{Number of Transfer bits}}$$

ビットエラーの数が 0 である場合、ビットエラーレートは 0 となるが、ある信頼水準におけるその上限値を計算によって求めることができる。

## A.1　ビットエラーレートの上限

ビットエラーが起きる確率が小さいと仮定すると、ビットエラー数の確率分布はポアソン分布[1]となる。ポアソン分布の確率密度関数 $P_{x,\mu}$ は、以下の式で表され、その概形は図 A.1 のようになる。

$$P(x,\mu) \equiv e^{-\mu}\frac{\mu^x}{x!} \tag{A.1}$$

ここで、x はビットエラー数を表し、$\mu$ は以下の式で定義され、ビットエラー数の期待値を表す。

$$\mu \equiv BER \times \text{Number of Transfer bit} \tag{A.2}$$

次に、信頼水準を C.L. と書き、以下のように定義する。

$$\mathrm{C.L.} \equiv \sum_{x=N_{Err}}^{x=\infty} P(x,\mu) \tag{A.3}$$

[1]ビットエラー数の確率分布は、原理的には二項分布であるが、発生確率が小さいときの二項分布は、ポアソン分布に近似できる。

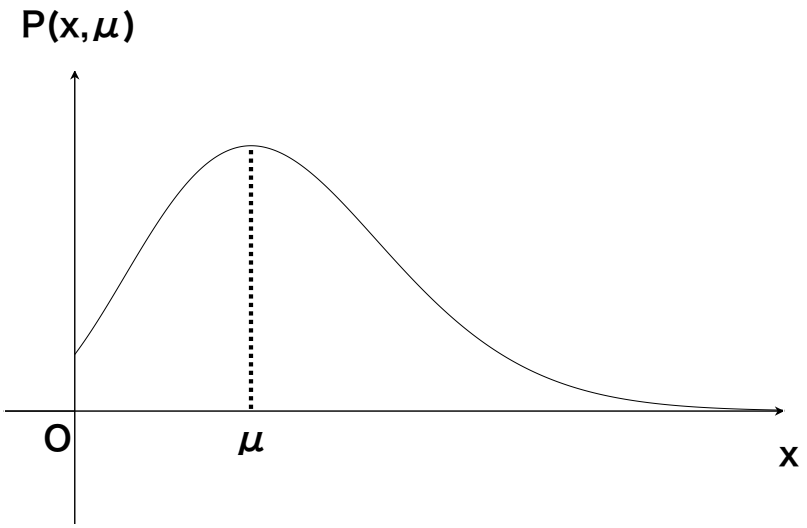


図 A.1: ポアソン分布の概形

信頼水準は、 図 A.2 に示すように、確率密度関数を $x > N_{Err}$ の範囲で積分したものである。つまり、$x > N_{Err}$ となる確率を意味する。

式 A.3 から、$x < N_{Err}$ となる確率は、

$$1 - \text{C.L.} = \sum_{x=0}^{x=N_{Err}} P(x, \mu) \tag{A.4}$$

となる。

式 A.4 は、式 A.1 から

$$1 - \text{C.L.} = \sum_{x=0}^{x=N_{Err}} e^{-\mu} \frac{\mu^x}{x!} = e^{-\mu} \sum_{x=0}^{x=N_{Err}} \frac{\mu^x}{x!} \tag{A.5}$$

と書ける。

式 A.5 を整理して

$$e^{-\mu} = \frac{1 - \text{C.L.}}{\sum_{x=0}^{x=N_{Err}} \frac{\mu^x}{x!}} \tag{A.6}$$

とする。

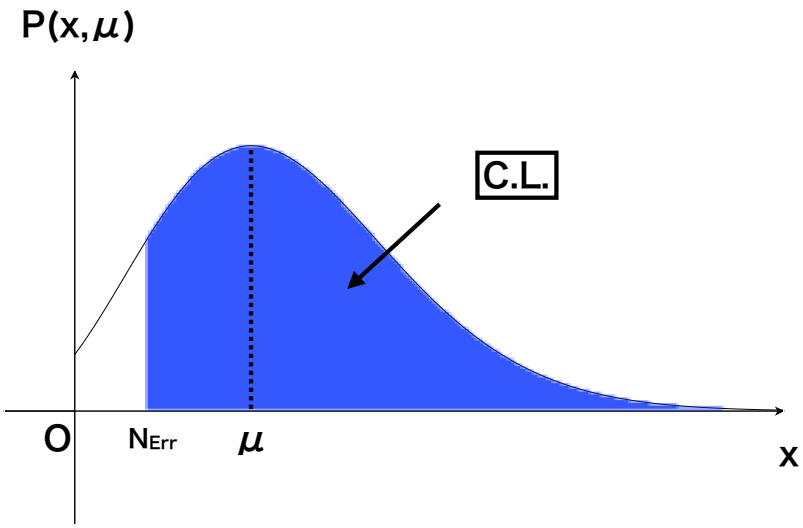


図 A.2: 信頼水準の定義。青色の領域の面積は信頼水準を意味する。

式 A.6 両辺の自然対数を取ると、

$$\mu = -\ln(1 - \mathrm{C.L.}) + \ln\left(\sum_{x=0}^{x=N_{Err}} \frac{\mu^x}{x!}\right) \tag{A.7}$$

となり、式 A.2 を用いて計算すると、

$$\mathrm{BER} = \frac{-\ln(1 - C.L.) + \ln\left(\sum_{x=0}^{x=N_{Err}} \frac{\mu^x}{x!}\right)}{\text{Number of Transfer bit}} \tag{A.8}$$

となる。

測定されたビットエラー数が 0 の時、式 A.8 に $\mathrm{N_{Err}} = 0$ を代入することで、式 A.9 が導かれる。

$$\mathrm{BER_{upperlimit}} \equiv \frac{-\ln(1 - C.L.)}{\text{Number of Transfer bit}} \tag{A.9}$$

式 A.9 は、ビットエラーレートの上限値を表す。