金融商品取引アルゴリズムのハードウェア アクセラレーションに関する研究

北陸先端科学技術大学院大学
情報科学研究科

小林 弘幸

平成 30 年 3 月

課題研究報告書

金融商品取引アルゴリズムのハードウェア アクセラレーションに関する研究

1410752　　小林 弘幸

主指導教員　田中　清史
審査委員主査　田中　清史
審査委員　金子　峰雄
井口　寧

北陸先端科学技術大学院大学
情報科学研究科

平成 30 年 2 月

概 要

今日の金融商品の電子取引においては、売買注文の大部分はあらかじめプログラムされた取引アルゴリズムにより自動的に送信されている。売買注文は取引所に到達した順に売り注文と買い注文のマッチングが行われるため、電子取引に関わるシステムには可能な限り低遅延で通信と演算処理を行うことが要求される。通常の業務アプリケーションのネットワーク処理においては、電気信号による情報の受信時から起算して、情報を解釈・処理し、処理結果を送信する一連のプロセスにおいて、OS の TCP/IP プロトコルスタック及び通信アプリケーションの多重階層を経由するために相応の CPU 時間を必要とし、遅延が発生する最大の要因となっている。そこで本研究では金融市場における低遅延性の要求に応えるために、汎用プロセッサ上のソフトウェア処理に代えて、SoC FPGA 上に専用回路を構成しハードロジックにより処理を行うハードウェアアクセラレータを開発した。このアクセラレータを用いることで、FPGA 内で発生するレイテンシを 1 マイクロ秒未満に抑えることができ、サーバ側で計測される遅延値も 80 %程度削減されるという結果が得られた。

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景と目的

近年、ネットワーク処理や組込みシステム/自動運転の制御の目的で、リアルタイムで取得される情報に対して瞬間的に判断し低遅延で応答を返す専用ハードウェアを開発する動きがある [1]。ハードウェア処理を行うことで性能が向上するアプリケーション分野は多岐に渡るが、低遅延性を特に要求する領域の一つに金融商品取引がある。

2000 年前後から各国の主要取引所において株式などの金融商品が電子的に取引されるようになり、以降、迅速に取引関連情報を送受信する環境の整備が進み、現在では、主要な取引参加者は取引所のシステムが存在するデータセンター内にサーバを導入しており、情報更新に対して可能な限り高速に応答するシステムの開発競争が進んでいる [7]。[1]

本研究は、市場参加者が実際に発注に用いている逆指値という代表的な取引アルゴリズムを対象とし、ハードウェアアクセラレーションにより価格情報の受信から注文情報の送信までに要するレイテンシを低減することを目的とする。通信プロトコルは Ethernet、TCP/IP 及び、金融市場で最も一般的に用いられている FIX プロトコル [34] をターゲットとする。従来のソフトウェアを用いたネットワーク処理は、幾重にもわたる階層（割り込みハンドラ、Ethernet ドライバ、IP 層、TCP 層、ソケットシステムコール、FIX エンジン、アプリケーション）を経由することで必然的に遅延が発生している。本研究においては FPGA 上の専用回路を用いることで、この遅延を低減し通信を高速化することを意図している。一方、逆指値は非常にシンプルなロジックであり、取引アルゴリズムの演算処理の高速化を意図するものではない。

1.2　研究方法

本研究において、最初に取引所のマッチングシステムの挙動をシミュレートする取引所シミュレータ（サーバ）と、売買注文を送出するための発注クライアントの 2 つのソフトウェアを開発する。サーバとクライアントのマシンは 1 ギガビットイーサネットで接続され、FIX と呼ばれる通信プロトコル（OSI 参照モデルの L5-L7 に相当）を用いて相互に通信を行う。FIX 通信の実現にはフリーの FIX エンジン（クラスライブラリ）の

[1] 低遅延取引（Low Latency Trading)。ほぼ同じ意味で高頻度取引（High Frequency Trading : HFT) とも呼ばれ、高頻度取引や HFT の方が一般的だが、本研究においては一定期間に多くの通信を行う（スループット）ことよりも短い間隔で応答する（レイテンシ）ことに注目しているため、低遅延取引と呼んでいる。

QuickFIX を用いる。取引所シミュレータの開発には C#を用い、発注クライアントの開発には C++を利用する。発注クライアントは Linux（Ubuntu15.10) 環境で動作させるが、コードは Windows 上の Visual Studio 2017 で開発し、Linux の gcc でコンパイルしている。

次に、FPGA でアクセラレータを開発する。FPGA には、ハードコアプロセッサ及びギガビットイーサネットコントローラのハードマクロを備えている SoC FPGA である、Xilinx 社の Zynq を用いる。Zynq を搭載する FPGA ボードとしては Digilent 社の PYNQ-Z1 を利用する。ハードウェア設計には Vivado 2017.2 を利用し、FIX メッセージを処理するための FIX プロセッシングユニット（IP）を設計する。ハードウェア記述言語は Verilog HDL を使用する。Vivado で生成されるビットストリーム (.bit) とブートローダを合わせてブートイメージ (BOOT.bin) を作成するために Xilinx SDK を利用する。

また、受信フレーム及び送信フレームを格納するバッファを DRAM に替えて FPGA のエンベデッドメモリである Block RAM(以下 BRAM）上に設定するために、イーサネットコントローラのデバイスドライバ (xemacps) を修正する。ドライバを修正後に Linux のカーネル全体をコンパイルし直しイメージファイル (uImage) を作成し、BOOT.bin と共に SD カードのファイルを入れ替えてシステムを更新する。

その他、デバッグのために Linux から BRAM の Read/Write を行う用途で Python を利用する。Ethernet フレームの観測と遅延の実測にはパケットキャプチャツールの Wireshark を利用する。

1.3 本研究の貢献

FPGA を用いた低遅延取引に関する論文や報告書は幾つか存在するが、それらの研究成果は具体的な回路設計は開示されず、また、従来研究は高価な専用機材を用いているため、専門の事業者以外は手を出しにくい状況であった。本研究においては、安価な FPGA ボードとフリーソフトのみを利用してハードウェアアクセラレータを開発する方法を示している。近年の電子取引システムの高速化の恩恵を得ているのは一部の事業者のみとの批判もある [2] 中、一般投資家もあまねく低遅延取引が利用可能になるように技術の大衆化に貢献することを期待している。

また、アプリケーションプログラムの中で I/O のレイテンシにセンシティブな箇所の処理をハードウェアに委譲させる本研究の設計パターンは金融商品取引に限らず多様なアプリケーションにも適用可能で、SoC FPGA を利用してストリーム処理を高速化するための一手法を提示している。

1.4 本報告書の構成

本報告書では、以下の構成にてハードウェアアクセラレータの研究成果を記述する。

- 第 2 章　金融商品の低遅延取引、アルゴリズム取引に関わる研究について述べる。
- 第 3 章　本研究の前提知識としての FIX プロトコルに関して述べた後、取引所シミュレータと発注クライアント（ソフトウェア処理）について述べる。
- 第 4 章　ハードウェアアクセラレータの構成と開発方法について述べる。
- 第 5 章　作成したアクセラレータを用いたハード処理の回路遅延を算出する。また、トータル遅延を計測し、ソフトウェア処理の場合と比較する。
- 第 6 章　研究成果を実用化するための指針と他分野への発展性を述べ、本報告書のまとめを行う。

第2章　関連研究

2.1　低遅延取引にかかわる研究

ハードウェアアクセラレータを利用した低遅延取引の研究事例は2007年頃から存在する。代表的な研究としては、複数情報源からの価格情報を集約するサーバのストリームプロセッサとしてFPGAを用いることで遅延を26μs以下に短縮できることを実証したGareth W. Morris, David B. Thomas and Wayne Luk (2009) [4]、ネットワークプロトコルのデコード処理をFPGAにオフロードすることで発注判断の前処理部分を高速化できることを示したChristian Leber, Benjamin Geib, Heiner Litz(2011) [5]、FPGAを用いることで平均的に2.7μsで応答可能で、かつソフトウェア処理に比べてバラつきが小さいことを確認したRobin Pottathuparambil et al. (2011) [6]が挙げられる。

ベンダー企業の発行する資料や論文にも重要な研究が幾つか存在する。John W. Lockwood, Michaela Blott et al. (2012) [7]はFPGA搭載NICを用いて1μs（FPGA内部では200ns)の遅延で取引が可能であることを報告している。またこの論文の1～3章は低遅延取引におけるFPGAアクセラレータの利用方法と現状に関するサーベイとして利用できる。Finteligent Trading Technology Community (FTTC)によるリサーチレポート(2012)[8]には、ネットワーク処理に関しての遅延計測の具体的な方法が記載されている。ARGON DESIGN, ARISTA, FTTCによるリサーチレポート(2013) [9]は、価格情報メッセージの到達が完了する前に必要な情報が入手できた時点で注文情報の送信を始めるように回路を構成し、150nsの遅延で取引できることを報告している。井上(2012）[10]は日本の証券市場の実取引データを用いて株価の変化点検出等の複合イベント処理について、FPGA搭載NICを用いることでプロセッサよりも12.3倍の速度で処理できることを報告している。

「FPGAの原理と構成」天野編著(2016) [11]の7章6節はNIC（ASIC)+ CPUの通常構成、FPGA NIC ＋ CPUの構成、SoC FPGAのワンチップ構成の比較を行い、低遅延取引におけるSoC FPGAの優位性を指摘している。

2.2　アルゴリズム取引に関する参考文献

アルゴリズム取引について概略を理解するには情報処理学会誌「情報処理」（2012年9月号）特集「金融市場における最新情報技術」の1～4章 [12] [13] [14] [15]が利用できる。VWAP、Icebergなどのアルゴリズム取引戦略を解説している文献は多く存在するが、

論文では日本銀行金融研究所による杉原 (2011) [17]、書籍では Barry Johnson(2010) [16] が包括的である。

第3章　ソフトウェア処理

3.1　FIX プロトコル

FIX プロトコルは取引所からの価格情報や取引参加者からの注文情報といった、金融商品の電子取引に必要な種々の情報の送受信を文字列として統合的に扱うプロトコルである。非営利団体の FIX Trading Community により策定され、取引所、金融機関、機関投資家等の市場関係者の間の商取引情報の伝達手段として最も広範囲に用いられている。[1]

FIX メッセージは例えば以下のような形で表現される。

（買い注文を表す FIX メッセージ）

```
8=FIX.4.4_9=139_35=D_34=17_49=CLIENT1_52=20170512-17:54:56.404_
56=SIMULATOR_1=CLIENT1_11=101_21=1_38=2_40=2_44=1100_54=2_
55=MSFT_59=0_60=20170512-17:54:56_10=046
```

この FIX メッセージに相当する ASCII コードのバイナリに、TCP ヘッダ、IP ヘッダ、Ethernet ヘッダ、プリアンブル及び FCS を付加した Ethernet フレームが実際に送受信されることとなる。本研究では FPGA のハードロジックで注文を行う場合においても、FIX メッセージはソフトウェア（発注クライアント）で生成することとし、売買タイミング判断及び下位レイヤのヘッダの生成にハードロジックを活用している。FIX には複数のバージョン（4.0, 4.1, 4.2, 4.3, 4.4, 5.0(sp1, sp2)） が存在するが、最新バージョンである 5.0 よりも 4.4 が使われるケースが多いため、本研究では 4.4 をターゲットとする。

3.1.1　FIX メッセージの構造

FIX メッセージはタグと呼ばれるフィールドと値のペアから構成される。タグとタグ値は=で結ばれ、各タグ項目間は ASCII コード 0x01 の SOH(ヘッダ開始）で結ばれる。前述の FIX メッセージは表 3.1 のように分解できる。

上記のメッセージの場合、最初の 7 個のタグ (8,9,35,34,49,52,56) はヘッダ、最後のタグ（10）はフッタで、残りの 10 個のタグが本体である。ヘッダの中の最初の 3 個 (8,9,35) 及

[1]FIX 以外の通信方式としては取引所が独自に定める通信プロトコルを用いるケース（主に取引所-金融機関間）や、金融機関（証券会社・FX 取引業者・仮想通貨事業者）やベンダーが公開する API をクライアントが利用して取引を行うケース（金融機関-投資家間）がある。

表 3.1: FIX メッセージ例 (買い注文)

タグ番号	タグ名	値
8	BeginString	FIX.4.4
9	BodyLength	139
35	MsgType	D
34	MsgSeqNum	17
49	SenderCompID	CLIENT1
52	SendingTime	20170512-17:54:56.404
56	TargetCompID	SIMULATOR
1	Account	CLIENT1
11	ClOrdID	101
21	HandlInst	1
38	OrderQty	2
40	OrdType	2
44	Price	1100
54	Side	2
55	Symbol	MSFT
59	TimeInForce	0
60	TransactTime	20170512-17:54:56
10	CheckSum	046

びフッタ (10) の計 4 個のタグは全ての FIX メッセージにおいて必須である。タグ番号 8 は FIX のバージョンを表し、9 は FIX のメッセージ長（バイト数）、35 はメッセージタイプ、10 はチェックサムである。タグは FIX4.4 において（欠番も含め）956 番まで定義されているが、その中でどのタグのセットを用いるかはメッセージタイプによる。

FIX プロトコルの実装においては、各バージョンに含まれる全ての機能が実装される必要はなく、業務内容に応じた独自の機能項目を追加することもできる。[2] 本研究においても、サーバ・クライアントの双方のソフトウェアにおいて、取引に関する遅延の測定という目的に必要な項目のみを実装する。

3.1.2 メッセージタイプ

FIX メッセージは管理系メッセージ（FIX Administrative Message) と業務系メッセージ（FIX Application Message) に大別できる。管理系メッセージはログオン/ログアウトやハー

[2]通常、接続する業者間でサーバ側となる事業者が FIX の仕様の中でどの機能を利用するかを記載した仕様書を策定し、クライアント側となる事業者はその仕様を満たすように取引システムを構築する。

トビート、再送要求などの L5（セッション層）の処理に相当し、[3] 業務系メッセージは新規注文や価格情報配信などの実取引に関わる L7（アプリケーション層）のアクションに相当する。

FIX4.4 ではメッセージタイプは 0〜9, A〜Z, a〜z, AA〜AZ, BA〜BH の 96 種類が定義されているが、そのうち本研究に関わるもののみを表 3.2 に列挙する。「方向」は C(クライアント）と S(サーバ）のいずれから送信されるメッセージであるかを意味している。

表 3.2: メッセージタイプ

コード	方向	名称	意味
0	S → C → S	Heartbeat	通信生存確認
A	C → S → C	Logon	セッション確立
5	C → S → C	Logout	セッション終了
V	C → S	Market Data Request	価格情報要求
W	S → C	Market Data Snapshot Full Refresh	価格情報
D	C → S	New Order Single	新規注文
F	C → S	Order Cancel Request	注文取消/修正
8	S → C	Execution Report	注文受付/約定報告

この中でも本研究において特に重要なのは Market Data Full Refresh（W) と New Order Single(D) の 2 つのメッセージタイプで、クライアントに Market Data が到達してから New Order Single を送出するまでの時間をレイテンシの測定対象、ハードウェアアクセラレーションのターゲットとしている。[4]

3.1.3 FIX エンジン

FIX プロトコルを用いたアプリケーションを開発するにあたり、FIX エンジンと呼ばれる基本機能を備えたクラスライブラリが利用できる。本研究ではフリーソフトの QuickFIX という FIX エンジンを用いる [35]。QuickFIX は管理系メッセージを内包しており、管理系メッセージの送出、処理に関しては記述する必要がないため、業務系メッセージの送出と受信時の手続きをコーディングすることとなる。

QuickFix は C++/Java/.net/Go の 4 バージョンがあるが、この中で仮想取引所サーバは GUI 開発の容易さから C#(.net) を用いる（バージョンは QuickFIX/n 1.7.0)。発注用クライアントは Linux 環境で動作させることと、レイテンシ比較に用いるために実行速度に優れる C++を用いる（バージョンは QuickFIX 1.14.3)。

[3] 業務系メッセージの方が種類が多く、管理系メッセージは Heartbeat, Logon, TestRequest, ResendRequest, Reject, SequenceReset ,Logout の 7 種のみである。

[4] 実際の取引においては新規注文以外にも注文取消メッセージや注文訂正メッセージに関わる遅延も重要であるが、本研究で対象とする逆指値ロジックにおいては訂正・取消は自動的には行われないため、新規注文のみを考慮する。

3.2 取引所シミュレータ（サーバ）

本研究の取引対象となる金融商品は株式、債券、通貨、先物、オプション、暗号通貨など種別を問わないが、特に低遅延性が要求されるのは相対でなく取引所で行われる取引であるために、証券取引所（主に株式)、デリバティブ取引所（先物及びオプション）といった、公設の取引所での取引を念頭に置く。

証券取引所やデリバティブ取引所では、決められた取引セッション時間内において、取引参加者の買い注文と売り注文を集約し、売り注文価格≦買い注文価格となる注文対（買い及び売りの組）が出現したら即座に約定させている（新規注文に対して複数の既存注文が条件を満たす場合は価格→発注時間の優先順位で約定させる注文を決定する)。[5] この注文マッチングの機能をシミュレートする取引所シミュレータを実装する。取引所シミュレータはFIX通信のサーバ側の役割で、複数のクライアントと接続することができる。取引所シミュレータは各クライアントが接続するのを待機し、クライアント側が接続を開始する。

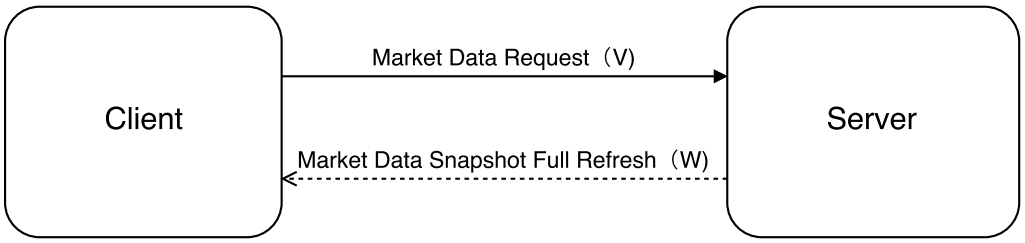
取引所シミュレータは、各クライアントからのNew Order Single(新規注文:D)、Order Cancel Request(注文取消:F)、Market Data Request(価格情報要求:V) のメッセージを受け入れ、新規注文と注文取消に対してはExecution Report(注文受付/約定報告メッセージ:8) を返し、価格情報要求に対してはMarket Data Snapshot Full Request(価格情報:W) を返す。

加えて、取引所シミュレータは各上場銘柄ごとに板情報（現在の注文状況）のリストを保有し、新規注文、注文取消メッセージを受け付ける際に板情報が更新される度に、価格情報要求を送信した全てのクライアントに対して価格情報（W) を送信する。また、新規注文の結果、売買が成立した場合、取引当事者双方のクライアント（新規注文及び既存注文の出し手）に対して注文受付/約定報告 (8) メッセージを送信する。以上をまとめたものが図3.1である。

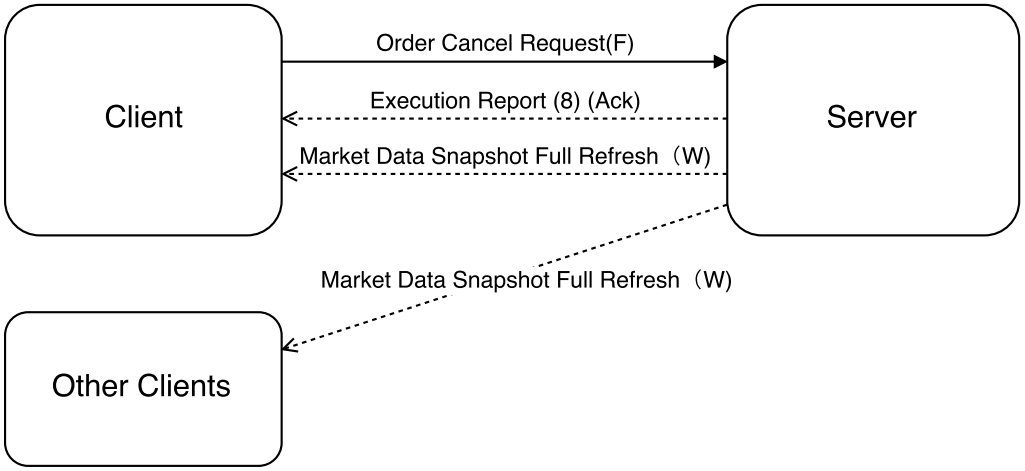
取引所シミュレータは注文状況、約定結果や取引のログを表示するために図3.2のGUIを持つ。

[5]板寄せ方式 (Open and Closing Auctions) と呼ばれる、一定期間内に出された注文を（即時約定させずに）期間終了後にまとめて約定させる仕組みもあり、取引所によっては各取引セッションの最初と最後は板寄せでマッチングが行われているが、各国の主要取引所における取引の大半は即座に約定する方式（ザラバ方式；Continuous Auction method）によって行われている。

１）サーバに価格情報要求が到達した時



２）サーバに注文取消が到達した時



３）サーバに新規注文が到達した時

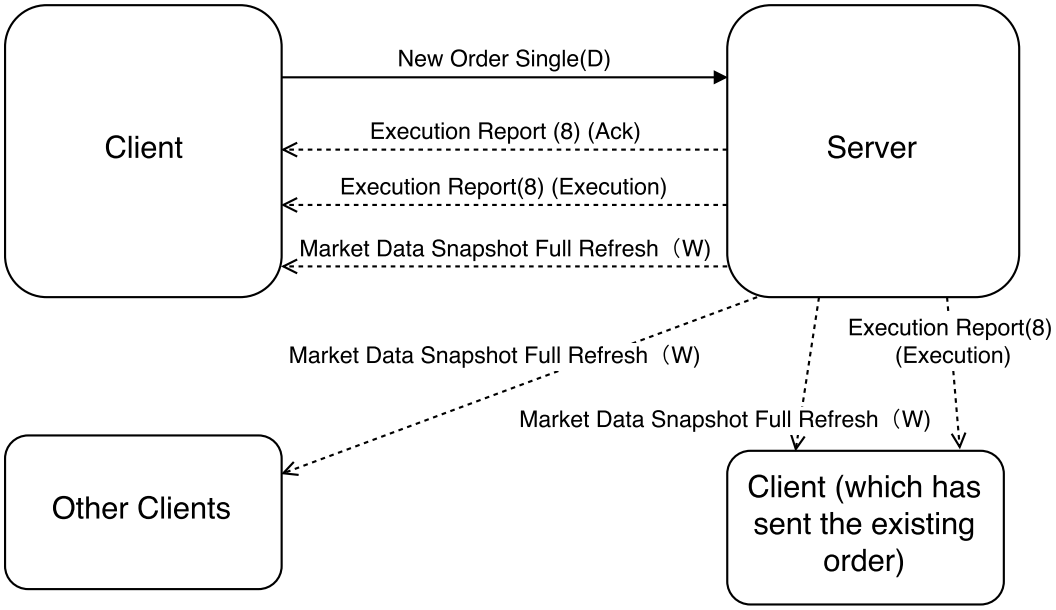


図 3.1: 取引所シミュレータの挙動

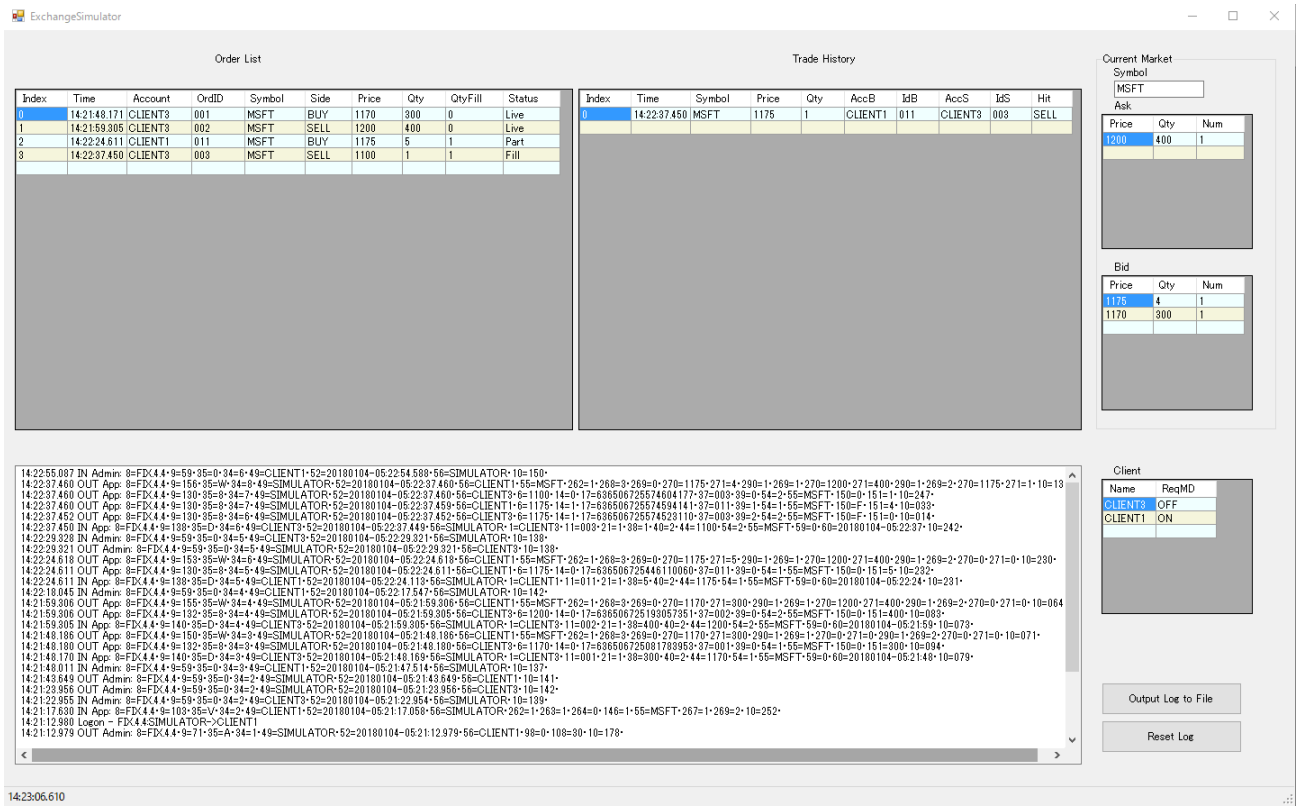


図 3.2: 取引所シミュレータ

3.3 発注クライアントと逆指値

発注クライアントは図 3.4 のような CUI アプリケーションで、コマンドを入力することで取引所シミュレータに対し新規注文 (D)、注文取消 (F)、価格情報要求 (V) のメッセージを送信できる。また、取引アルゴリズムを登録することで、取引所シミュレータから価格情報更新（W) メッセージを受信時に、あらかじめ決められた条件を満たした場合、新規注文（D) を自動的に送信することができる（図 3.3 参照)。[6]

本研究においては、この取引アルゴリズムとして、最もシンプルな「逆指値」を設定している。逆指値は価格があらかじめ設定された数値以上になった場合に買い注文を出す、あるいは設定された数値以下になった場合に売り注文を出すロジックである。逆指値はストップオーダーとも呼ばれ、リスクを限定的にする目的でたびたび使われる。例えば株価が 1000 円の時に株式を購入した投資家は、損失を投下資金の 20%までに抑えたい場合、株価が 800 円以下になった場合に自動的に売却するように逆指値を設定しておけば、価格変動により生ずる損失額に上限を設けることができる。[7] [8]

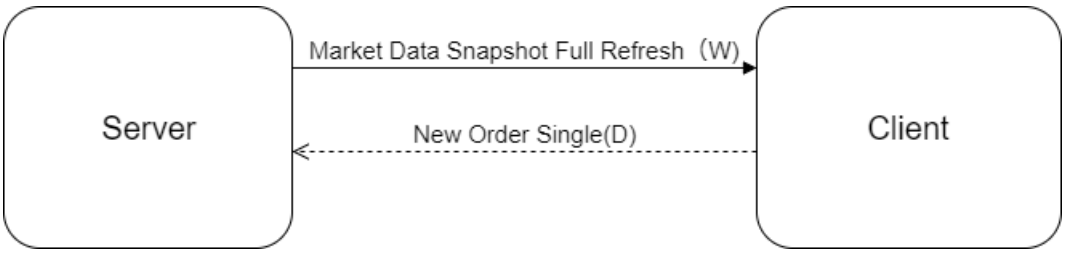


図 3.3: 発注クライアントのアルゴリズム取引

[6]一般の取引アルゴリズムにおいては、価格情報がトリガになるケース以外にも、自注文の約定メッセージ受信、タイマイベント、ニュース等の市場外の情報受信など様々な情報更新がトリガとなり注文を送出するが、本研究においては価格情報更新のみをトリガとしている。

[7]逆指値注文 (Stop Order) を取引仕様として受け付ける取引所も存在するが、例えば東京証券取引所では逆指値注文を受け入れていない。多くのオンライン証券会社は「逆指値」を投資家に提供しているが、これは各証券会社のサーバ側で逆指値注文がトリガ条件達成時に取引所に指値/成行注文を送信するようなシステムになっている。

[8]逆指値注文を利用するのは主に個人投資家で、市場状況を常に監視せずともあらかじめ設定した価格到達時に注文を執行できることがメリットとなる。一方、機関投資家は多種類の銘柄に対して大量の注文を送信するために VWAP 等のより複雑な取引アルゴリズムを利用するケースが多い。

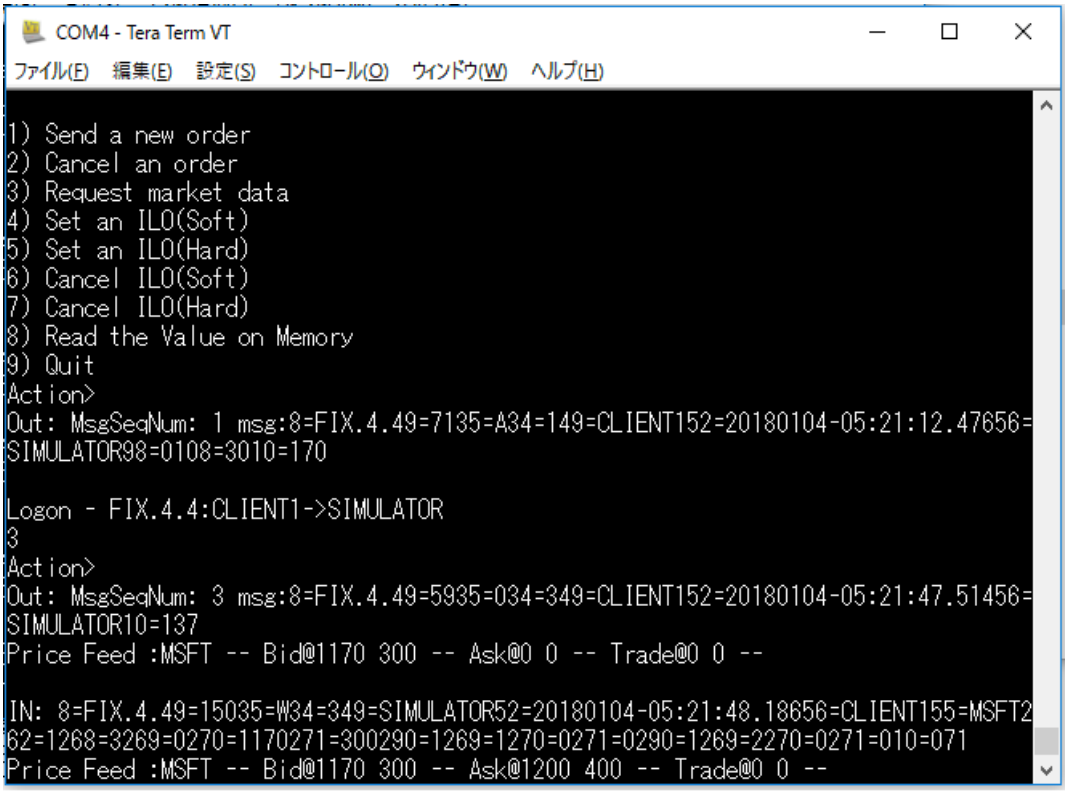


図 3.4: 発注クライアント

3.4　システム全体像

取引所シミュレータを起動する仮想取引所サーバ（Windows PC）と発注クライアントを起動するクライアントマシン（PYNQ-Z1 上の Linux）は図 3.5 のようにイーサネットケーブル（1000BASE-T）で接続され、FIX メッセージをやり取りする。時刻計測はサーバ側で行う。

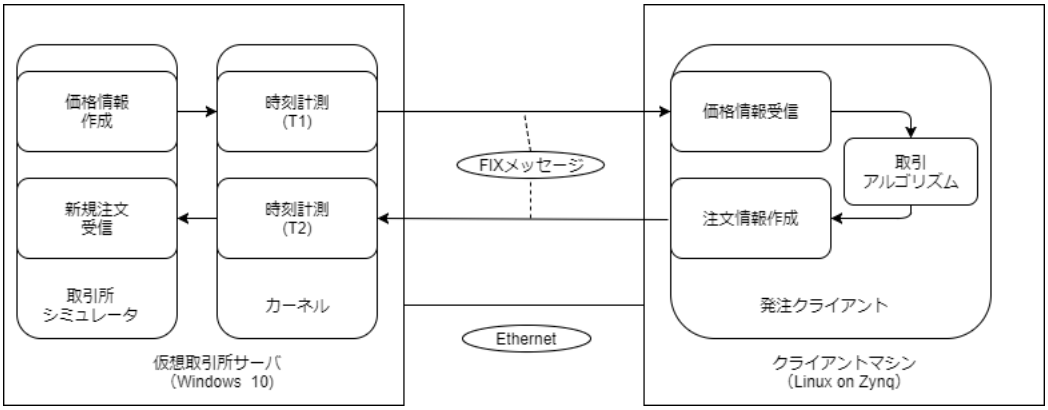


図 3.5: システム全体像

第4章　ハードウェア処理

4.1　FPGA

本節ではハードウェア設計に利用するFPGAデバイスとバスについて述べる。

4.1.1　Zynq

本研究においては、前章で述べた発注クライアントソフトウェアを用いた方式がスタート地点であるため、OS（Linux) を動かすCPUとFPGAが1チップのSoC FPGA構成が利用しやすい。[1] そこで、XilinxのSoC FPGAのZynq（Zynq-7000 All Programmable SoCシリーズ）を用いる [40] [33]。市販のZynqボードの中で、Digilent社のPYNQ-Z1を選択したが、価格がアカデミック版で＄65と最も安価であることに加え、Linux（Ubuntu）が起動するイメージファイルが提供されており、開発が容易であることによる [36]。PYNQ-Z1に搭載しているFPGAの型番はxc7z020clg400-1である。

ZynqはPS部（CPU、イーサネットコントローラなどハードIP）とPL部（FPGA) で構成されるが、PYNQ-Z1などの安価なボードにおいてはEthernetとの接続はPS部に限られている。そこで、PL部にBRAMを作成し、Ethernetから受信する情報、及びPL部からEthernetに送信すべき情報はこのBRAMに格納する。

ZynqのPS部にはCPU（ARM Cortex-A9のデュアルコア) とギガビットイーサネットコントローラ（Cadence Gigabit Ethernet MAC）が存在し、PYNQ-Z1ボード上にDDR3 DRAM(IS43TR16256A-125KBL) が存在する。通常の構成において、イーサネットコントローラは受信したフレームをDRAMに転送し、CPUがDRAM上のフレームを読む。送信時はCPUが送信すべきフレームをDRAMに書き込み、イーサネットコントローラがDRAM上のフレームを読む。一方、本研究ではドライバを改造し、フレームの転送先をPL部にあるBRAMに変更することで、PL部のユーザ回路からも送受信するフレームデータにアクセス可能にしている。

また、PYNQ-Z1ボードにはPHY(Realtek RTL8211E-VL PHY) が搭載されており、ZynqのPS部とRGMIIで接続されている。PHYはRJ-45コネクタと接続しており、RJ-45に

[1] SoC FPGAを利用しない構成では、Microblaze等のソフトコアプロセッサを利用する方法、FPGAとプロセッサは別チップの構成としてPCI expressで接続する方法も考えられるが、一般にソフトコアよりSoCのハードコアプロセッサの方が動作周波数が高く、また現時点において1Gbps以上のイーサネットコネクタを備える安価（50,000円以下）なXilinx社のFPGAボードはZynq製品に限られるため、ソフトコアプロセッサは利用していない。同様に安価なボードではPCIe接続の端子が無い為に選択肢から外れる。

LAN ケーブル（Cat5e 以上）を繋ぐことで、1000BASE-T の通信を行うことができる。イーサネットコントローラは GMII 接続のため、PS 部に GMII と RGMII を変換するアダプタが搭載されている。Cortex-A9 の動作周波数は最大 650MHz、DDR3 メモリコントローラは最大 525MHz である。

4.1.2 AXI バス

PS 部、BRAM、FIX プロセッシングユニット（自作 IP）は AMBA4 AXI4 (Full) バスで接続される [38] [41]。そこで FIX プロセッシングユニットは AXI に準拠する形で I/O を記述する必要がある。AXI はマスタとスレーブに分かれるが、FIX プロセッシングユニットをマスタ、BRAM をスレーブとし、FIX プロセッシングユニットから書込（Write)、読込（Read) の指示を発行する。AXI のデータ幅は 8, 16, 32, 64, 128, 256, 512, 1024 から選べるが、本研究においては可能な限り早急にデータの読み書きを行うために、最大の 1024 ビット (128 バイト）に設定する。

PS 部内部の CPU との接続は AMBA3 AXI（データ幅は 32 ビット)、イーサネットコントローラとの接続は AMBA2 AHB/APB [39] でそれぞれバージョンが異なるが、AHB/APB と AXI3 は PS 部の Interconnect で、AXI3 と AXI4 は PL 部の Interconnect で変換される。

4.2 アクセラレータ全体像

本節では本研究で作成するアクセラレータの回路構成、ステートマシン、BRAM のアドレス設定に着目し、アクセラレータの全体像を述べる。

4.2.1 回路構成

作成した回路の全体図を図 4.1 に示す。実際に設計する回路（IP) は npu_0（FIX プロセッシングユニット）で、この FIX プロセッシングユニットが AXI バス経由で BRAM と接続している。PS 部（AXI3) は AXI Interconnect の内部にあるプロトコルコンバータで AXI4 に変換し、PL 部の BRAM と接続している。外部 I/O は LED 出力 (LD0,LD1,LD2,LD3) とスイッチ (BTN0)・トグルスイッチ (SW0, SW1) に接続している。

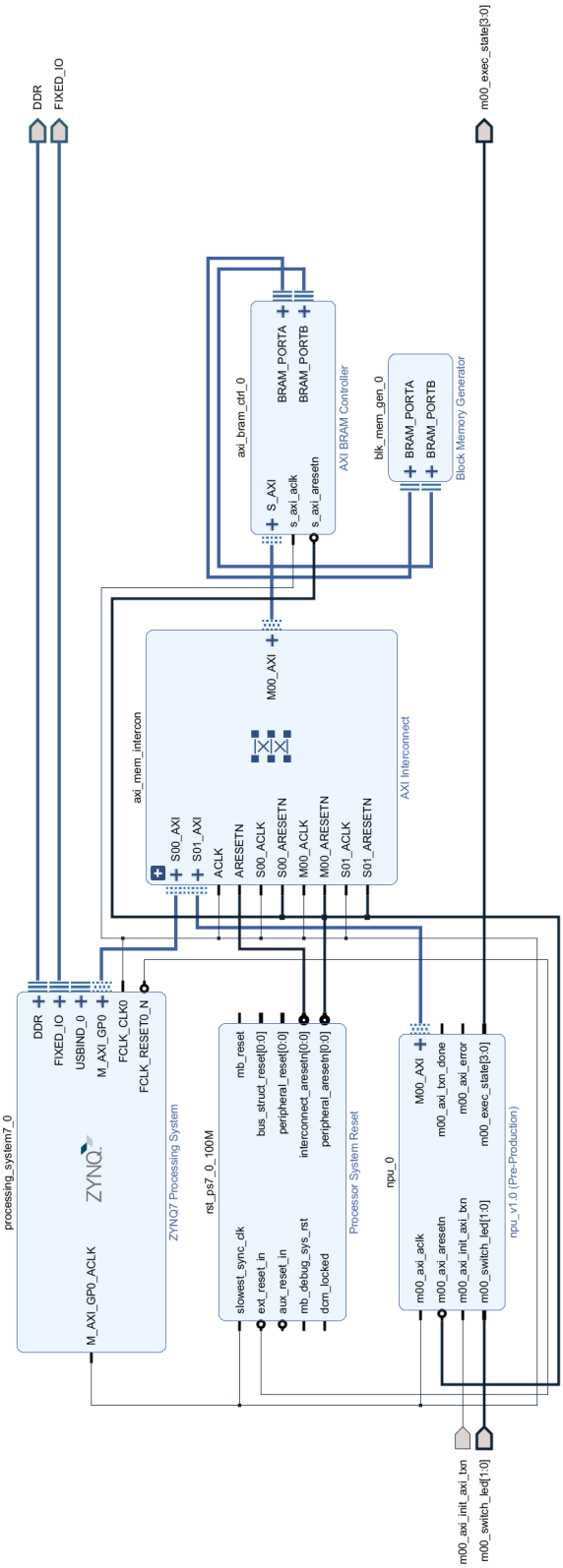

DDR
FIXED_IO
m00_exec_state[3:0]
axi_bram_ctrl_0
S_AXI
s_axi_aclk
s_axi_aresetn
BRAM_PORTA
BRAM_PORTB
Axi BRAM Controller
blk_mem_gen_0
BRAM_PORTA
BRAM_PORTB
Block Memory Generator
axi_mem_intercon
S00_AXI
S01_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
S01_ACLK
S01_ARESETN
M00_AXI
AXI Interconnect
processing_system7_0
DDR
FIXED_IO
USBIND_0
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N
M_AXI_GP0_ACLK
ZYNQ7 Processing System
rst_ps7_0_100M
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
Processor System Reset
npu_0
M00_AXI
m00_axi_txn_done
m00_axi_error
m00_exec_state[3:0]
m00_axi_aclk
m00_axi_aresetn
m00_axi_init_axi_txn
m00_switch_led[1:0]
npu_v1.0 (Pre-Production)
m00_axi_init_axi_txn
m00_switch_led[1:0]

図 4.1: PL 部回路構成

4.2.2　ステートマシン

FIX プロセッシングユニットは 3 つの「状態」を持ち、この状態を順序回路の制御に用いている。状態遷移図を表したものが、図 4.2 である。

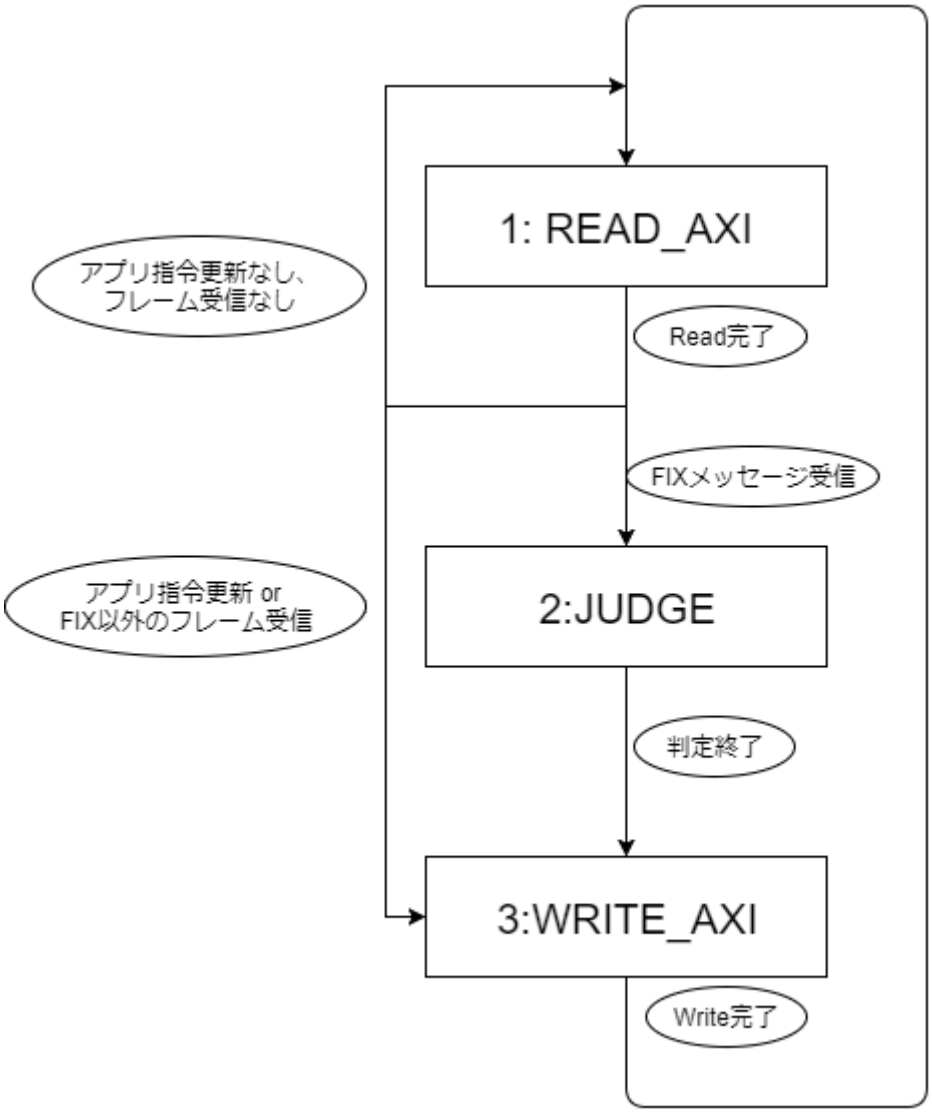


図 4.2: ステートマシン

4.2.3 BRAM

FPGA の内部メモリとして BRAM が利用できる。本研究で用いる Zynq には 140 × 36Kb の BRAM が搭載されている。BRAM は論理合成時にレジスタの実装として推論される他、BRAM Controller [42] [43] を利用し、物理アドレスを割り振ることで、明示的に PS 部や他の IP からアクセスできる。

本研究では Linux 上のソフトウェア（Ethernet ドライバ及び発注クライアントアプリ）と PL 部のデータの受け渡し用に BRAM の特定アドレス (0x4000_0000-0x4000_0FFF)(4KB) を確保している。上記のアドレスにアクセスすることで CPU からも FIX プロセッシングユニットからもデータを Read/Write できる。具体的な仕様を表 4.1 に示す。(アドレスは全て 0x4000_0000 からのオフセットを意味する。以降の表でも同様）

表 4.1: BRAM アドレス設定

先頭	最終	長さ (Byte)	内容	ハード	ドライバ	アプリ
0x0000	0x00FF	256	送信バッファ	W	R	-
0x0100	0x017F	128	各種フラグ格納領域	R/W	R/W	R/W
0x0180	0x027F	256	アプリからの指令領域	R	-	W
0x0280	0x037F	256	受信バッファ	R	W	-
0x0380	0x0FFF	3200	受信バッファ	-	W	-

このような配置にした主な理由は AXI のバースト転送において、連続したアドレス領域の昇順アクセスであれば Read 及び Write が高速に行えるという事情による。本研究の対象とする FIX メッセージは受信：価格情報（メッセージタイプ W）と送信：新規注文（メッセージタイプ D）の 2 つで、共に通常 256 バイト以下[2] であり、送信バッファとアプリからの指令領域は 256 バイト以下に限定できるが、一般のプロトコルでは 1518 バイトまでのフレームを受信しうるため、受信バッファは最後に配置し、そのうちの先頭の 256 バイトのみ FIX プロセッシングユニットから読むことにする。

このアドレス設定にすることで、Write のアクセス範囲は 0x0000-0x017F の 3 × 128 バイト、Read のアクセス範囲は 0x0100-0x037F の 5 × 128 バイトとなり、Write では 3、Read では 5 のバースト長を設定することで、それぞれ 1 回ずつの命令発行で全データを読み書きすることが可能になる。

各種フラグ格納領域（0x0100 からの 128 バイト）のうち、実際に情報のやり取りを行うのは先頭の 22 バイトである。詳細は表 4.2 に示す。

この中で特に重要なフラグは Ethernet ドライバ、発注クライアントアプリとやりとりをする「受信フレーム到達」「フレーム処理完了」「アプリ指令更新」「累積送信回数」である。

[2]FIX の仕様上、銘柄名やクライアント名などの可変長項目が含まれるため、256 バイトを超えるメッセージが送受信される可能性もある。そこで 256 バイト以下に収まるように FIX で交換される各項目の文字列数に上限を設けている。詳細は表 4.9 参照。

表 4.2: BRAM アドレス設定 各種フラグ

先頭	最終	長さ (Byte)	内容	ハード	ドライバ	アプリ
0x0100	0x0103	4	受信フレーム到達	R	W	-
0x0104	0x0107	4	フレーム処理完了	W	R	-
0x0108	0x010B	4	アプリ指令更新	R	-	W
0x010C	0x010C	1	セグメント長	W	-	-
0x010D	0x010D	1	アプリ指令設定済	W	-	-
0x010E	0x010E	1	累積送信回数	W	-	R
0x010F	0x0110	2	累積 FIX 処理回数	W	-	-
0x0111	0x0112	2	累積フレーム処理回数	W	-	-
0x0113	0x0115	3	累積 AXI Read 回数	W	-	-

FIX プロセッシングユニットは殆どの時間において READ_AXI 状態であるが、「受信フレーム到達」時には JUDGE 状態または WRITE_AXI 状態に遷移する。受信フレーム到達後は必ず「フレーム処理完了」フラグを書き込み、ドライバに通知する（ハードウェアで発注を行う時は 0xFF、行わない場合は 0xAA を返す）。

ハードウェア発注が行われた場合は「累積送信回数」がインクリメントされるため、アプリ側はハードに設定した注文が執行されたことを認識できる。

「アプリ指令更新」フラグの値が変更された場合も、WRITE_AXI 状態に遷移して「アプリ指令設定済」フラグが ON になったことを通知する書込みを行う。

4.3　受信・送信フレーム

本節においては、BRAM に書き込まれる受信フレームと送信フレームについて述べる。

価格情報を表すフレームの受信時に、当該フレームから価格情報を抜き出し、設定されている逆指値と比較を行い、発注する場合に FIX メッセージに TCP、IP、MAC のヘッダを付加し BRAM 中の送信バッファに書き込んでいる。ここで付加される各ヘッダは受信した価格情報フレームから作成する必要がある。前章で示したソフトウェアを利用し、逆指値執行時の価格情報受信と新規注文送信に相当する Ethernet フレームのペアをパケットキャプチャツールを用いて取得し、価格情報から新規注文を作成するような回路を設計する。以下では具体的な Ethernet フレームを用いて方法を述べる。

4.3.1　L2(MAC）

データリンク層（L2) で付加される L2 ヘッダは Ethernet-Ⅱ で定義されている（表 4.3、表 4.4 参照)。

表 4.3: L2 ヘッダ 受信

先頭	最終	長さ (Byte)	内容	値
0x0280	0x0285	6	宛先 MAC アドレス	00 0a 35 00 01 22
0x0286	0x028B	6	送信元 MAC アドレス	22 2c 56 dc 98 0a
0x028C	0x028D	2	タイプ	08 00

表 4.4: L2 ヘッダ 送信

先頭	最終	長さ (Byte)	内容	値
0x0000	0x0005	6	宛先 MAC アドレス	22 2c 56 dc 98 0a
0x0006	0x000B	6	送信元 MAC アドレス	00 0a 35 00 01 22
0x000C	0x000D	2	タイプ	08 00

「タイプ」は上位レイヤのプロトコル (0800 は IPv4) を表すため固定値で、送信フレームの L2 ヘッダは単純に受信フレームの L2 ヘッダの宛先と送信元の MAC アドレスを入れ替えて作成できる。なお、Ethernet-Ⅱ ではこの 3 項目以外にもプリアンブルと FCS も定義しているが、この 2 項目はイーサネットコントローラが処理し、PL 部には届かないため、対象としない。

4.3.2 L3(IP)

ネットワーク層（L3) で付加される L3 ヘッダは RFC791 で定義される IP(v4) である（表 4.5、表 4.6 参照)。

表 4.5: L3 ヘッダ　受信

先頭	最終	長さ (Byte)	内容	値
0x028E	0x028E	1	バージョン& IP ヘッダ長	45
0x028F	0x028F	1	パケット優先度	00
0x0290	0x0291	2	パケット長	00 de
0x0292	0x0293	2	識別番号	21 50
0x0294	0x0295	2	分割フラグ	40 00
0x0296	0x0296	1	TTL（パケット寿命）	80
0x0297	0x0297	1	上位プロトコル	06
0x0298	0x0299	2	チェックサム	00 00
0x029A	0x029D	4	送信元 IP アドレス	c0 a8 0b 06
0x029E	0x02A1	4	宛先 IP アドレス	c0 a8 0b 18

表 4.6: L3 ヘッダ　送信

先頭	最終	長さ (Byte)	内容	値
0x000E	0x000E	1	バージョン& IP ヘッダ長	45
0x000F	0x000F	1	パケット優先度	00
0x0010	0x0011	2	パケット長	00 cb
0x0012	0x0013	2	識別番号	56 6a
0x0014	0x0015	2	分割フラグ	40 00
0x0016	0x0016	1	TTL（パケット寿命）	40
0x0017	0x0017	1	上位プロトコル	06
0x0018	0x0019	2	チェックサム	4c 54
0x001A	0x001D	4	送信元 IP アドレス	c0 a8 0b 18
0x001E	0x0021	4	宛先 IP アドレス	c0 a8 0b 06

この中で、バージョン& IP ヘッダ長は 45（IPv4, 20 バイト)、パケット優先度は 00（デフォルト)、分割フラグは 40（分割禁止)、TTL は 40（64 回)、上位プロトコルは 06(TCP) に固定しても差し支えない。また、識別番号は分割するときのみ意味を持つために任意の番号に設定できる。送信元及び宛先の IP アドレスは MAC アドレス同様単純に入れ替えて作成できる。残りのパケット長とチェックサムに関しては回路上で計算が必要となる。

4.3.3 L4(TCP)

トランスポート層（L4）で付加される L4 ヘッダは RFC793 で定義される TCP である（表 4.7、表 4.8 参照）。TCP は表 4.7 で示される 20 バイトは必須だが、それ以外にもオプション項目を持つ場合がある。今回使用した環境ではデフォルトでオプション項目の Timestamps が有効であったが、Timestamps の値の生成にはデジタル回路内で時刻を管理することが要求される。TCP 通信に Timestamps の項目は必須ではないため、本研究では Timestamps を無効にし、オプション項目を持たないように設定している。[3]

表 4.7: L4 ヘッダ　受信

先頭	最終	長さ (Byte)	内容	値
0x02A2	0x02A3	2	送信元ポート番号	13 89
0x02A4	0x02A5	2	宛先ポート番号	cf 1f
0x02A6	0x02A9	4	シーケンス番号	75 17 94 f7
0x02AA	0x02AD	4	確認応答番号	c9 ce e0 fe
0x02AE	0x02AF	2	TCP ヘッダ長＆コントロールビット	50 18
0x02B0	0x02B1	2	ウィンドウサイズ	08 01
0x02B2	0x02B3	2	チェックサム	98 3f
0x02B4	0x02B5	2	緊急ポインタ	00 00

表 4.8: L4 ヘッダ　送信

先頭	最終	長さ (Byte)	内容	値
0x0022	0x0023	2	送信元ポート番号	cf 1f
0x0024	0x0025	2	宛先ポート番号	13 89
0x0026	0x0029	4	シーケンス番号	c9 ce e0 fe
0x002A	0x002D	4	確認応答番号	75 17 95 ad
0x002E	0x002F	2	TCP ヘッダ長＆コントロールビット	50 18
0x0030	0x0031	2	ウィンドウサイズ	02 2d
0x0032	0x0033	2	チェックサム	b3 68
0x0034	0x0035	2	緊急ポインタ	00 00

トランスポート層においてもポート番号は送信元及び宛先を入れ替えて作成できる。オプションを利用しない場合の TCP ヘッダ長は 50（20 バイト）となる。コントロールビットはコネクションの状態を制御する 6 つのフラグで構成されるが、今回はこのうち PUSH(速やかにデータをアプリケーションに渡す）と ACK（確認応答フラグ）の 2 つのビットを

[3] クライアント側の Linux 端末で ”sudo sysctl -w net.ipv4.tcp_timestamps=0” のコマンドを入力することで Timestamps を無効にできる。

立てるため、コントロールビットは 18 となる。URGENT（緊急フラグ）は立てず、緊急ポインタは読まれない（ここでは 00 00 に設定する）。

ウィンドウサイズの項目は、本来は利用可能な受信バッファのサイズを通知する必要があるが、このサイズは PL 部で管理していない。この項目はフロー制御に用いられ、確認応答を待たずに受け取れるデータサイズを 0〜65,535 で示すものであるが、保守的に小さめの数値を設定すれば問題が起こることはない。次回クライアントからサーバに何らかの Ethernet フレームが送信される際に本来の数値に更新される。

送信すべきシーケンス番号は直近に受信した確認応答番号に一致するので計算は必要ないが、送信すべき確認応答番号は受信したシーケンス番号に受信したペイロードの長さ（バイト）を加えたものであり、回路上で計算する必要がある。チェックサムもネットワーク層同様の手順で計算する。

4.3.4 FIX(価格情報受信）

価格情報の FIX メッセージは表 4.9 のフォーマットで与えられる。受信 FIX メッセージの開始アドレスは 0x02B6 である。L2、L3、L4 ヘッダ（54 バイト）を合わせたフレームデータが 256 バイト以内に収まるように FIX メッセージの長さは 202 バイト以内に抑える必要がある。そこで各項目に文字数の上限を設定している。

この中で重要な項目は Symbol(55)(銘柄コード) と MDEntryPx(270) である。MDEntryPx（価格）は Bid(買注文)、Offer(売注文)、Trade（取引) の 3 種類があるが、本研究の実装では FIX メッセージ中の 3 番目（最後）の MDEntryPx の項目が直近の Trade（取引) 価格を意味する。本研究では取引価格のみを用いるため、「Symbol」及び「3 番目の MDEntryPx」の 2 項目の値を取得するように回路を構成する。

4.3.5 発注クライアントアプリから渡されるデータ仕様

発注クライアントのソフトウェアからハードに送信すべき FIX メッセージ及び送信条件を渡すためのアドレス仕様が表 4.10 である。このように、送信すべき FIX メッセージ及び送信対象銘柄、トリガー条件の価格と売買フラグをアプリからハードウェアに渡している。

4.3.6 FIX(新規注文送信）

新規注文に関しては FIX クライアントのソフトウェアが生成し、FPGA に渡すため FIX メッセージを生成するための回路は作成しない。こちらも 202 バイト以内に抑える必要があるが、表 4.9（価格情報）の制限をかければ新規注文メッセージの文字数には余裕があるため表 4.11 では割愛している。送信バッファ内の FIX メッセージの開始アドレスは 0x0036 である。

表 4.9: FIX メッセージ（Market Data Snapshot Full Refresh)

タグ番号	タグ名	値	文字数
8	BeginString	FIX.4.4	7
9	BodyLength	159	3
35	MsgType	W	1
34	MsgSeqNum	22	1 ～ 6
49	SenderCompID	SIMULATOR	1 ～ 10
52	SendingTime	20180112-23:28:59.432	21
56	TargetCompID	CLIENT1	1 ～ 10
55	Symbol	MSFT	1 ～ 4
262	MDReqID	1	1
268	NoMDEntries	3	1
269	MDEntryType	0	1
270	MDEntryPx	1170	1 ～ 6
271	MDEntrySize	296	1 ～ 4
290	MDEntryPositionNo	1	1
269	MDEntryType	1	1
270	MDEntryPx	1200	1 ～ 6
271	MDEntrySize	497	1 ～ 4
290	MDEntryPositionNo	1	1
269	MDEntryType	2	1
270	MDEntryPx	1170	1 ～ 6
271	MDEntrySize	1	1 ～ 4
10	CheckSum	046	3

表 4.10: アプリからの指令項目

先頭	最終	長さ (Byte)	内容	値	値の意味
0x0180	0x0180	1	ID	F1	
0x0181	0x0184	4	OrderID	30 31 30 31	0101
0x0185	0x0188	4	銘柄名	4d 53 46 54	MSFT
0x0189	0x0189	1	売 or 買	32	1(Buy)/2(Sell)
0x018A	0x018E	5	トリガー価格	00 00 11 80 00	1180
0x018F	0x018F	1	パディング	00	
0x0190	0x0259	最大 202	FIX（新規注文)	(次項参照)	

表 4.11: FIX メッセージ（New Order Single)

タグ番号	タグ名	値
8	BeginString	FIX.4.4
9	BodyLength	140
35	MsgType	D
34	MsgSeqNum	20
49	SenderCompID	CLIENT1
52	SendingTime	20180112-23:28:59.089
56	TargetCompID	SIMULATOR
1	Account	CLIENT1
11	ClOrdID	0102
21	HandlInst	1
38	OrderQty	2
40	OrdType	2
44	Price	1100
54	Side	2
55	Symbol	MSFT
59	TimeInForce	0
60	TransactTime	20180112-23:28:59
10	CheckSum	086

4.4 FIX プロセッシングユニット

本節では FIX プロセッシングユニットを構成する配線及びレジスタを記し、レジスタの更新ルールについて述べる。本研究において、FIX プロセッシングユニットの I/O の主要部分は AXI である。そこで、開発に当たっては Vivado で作成される AXI4 ペリフェラルのサンプルを利用する。[4]

その結果、配線とレジスタの一部はサンプル回路に由来し、一部は独自に設計されたものとなっている。そこで表に「設計」の項目を設け、サンプル由来の配線及びレジスタには S(Sample)、本研究で独自に追加された配線及びレジスタは F(FIX Processing Unit) と記載し、区別している。

4.4.1 外部配線

FIX プロセッシングユニットの外部インターフェースである配線を表 4.12 にまとめる。「値/アサイン」の列は固定値を取る項目は値も付記し（値は 16 進数）、それ以外の出力配線は割り当てられるレジスタを記入している。M_AXI で始まる 44 本の配線は AXI4(Full) バスの構成要素だが、本研究において大半の項目は固定値である。固定値以外の項目では VALID, READY, LAST 関連のレジスタの ON,、OFF のタイミングはサンプル回路に記されているために、実際の設計では入力データ M_AXI_RDATA を受け取る方法と、出力データを M_AXI_WDATA(axi_wdata レジスタ) に送るタイミングが主要な設計項目となる。

[4]Vivado の画面から [Tools] → [Create and Package New IP] → [Create a new AXI4 peripheral] を選択後、Interface Type : Full, Interface Mode : Master のインターフェース 1 個のみを持つ IP を作成。

表 4.12: 外部配線

I/O	設計	ビット幅	名称	役割	値/アサイン
I	S	1	INIT_AXI_TXN	開始信号	-
I	F	2	SWITCH_LED	LED 出力切替	-
O	S	1	ERROR	エラー出力	-
O	F	4	EXEC_STATE	状態変数 LED 出力	-
I	S	1	M_AXI_ACLK	クロック信号	-
I	S	1	M_AXI_ARESETN	リセット信号	-
O	S	1	M_AXI_AWID	書込み ID	0
O	S	32	M_AXI_AWADDR	書込みアドレス	40000000
O	S	8	M_AXI_AWLEN	書込みバースト長（バースト回数-1)	02 (3 回)
O	S	3	M_AXI_AWSIZE	書込みバーストサイズ（データ幅の対数）	7 (128Byte)
O	S	2	M_AXI_AWBURST	書込みバーストタイプ	1 (INCR)
O	S	1	M_AXI_AWLOCK	書込み排他制御	0 (Normal)
O	S	4	M_AXI_AWCACHE	書込みメモリタイプ	2
O	S	3	M_AXI_AWPROT	書込みプロテクションタイプ	0
O	S	4	M_AXI_AWQOS	書込み QoS	0
O	S	1	M_AXI_AWUSER	ユーザ定義信号	1
O	S	1	M_AXI_AWVALID	書込みアドレス Valid 信号	axi_awvalid
I	S	1	M_AXI_AWREADY	書込みアドレス Ready 信号	-
O	S	1024	M_AXI_WDATA	書込みデータ	axi_wdata
O	S	128	M_AXI_WSTRB	書込みストローブ（ビットマスク)	全ビット 1
O	S	1	M_AXI_WLAST	書込みバースト最終フラグ	axi_wlast
O	S	1	M_AXI_WUSER	ユーザ定義信号	0
O	S	1	M_AXI_WVALID	書込み Valid 信号	axi_wvalid
I	S	1	M_AXI_WREADY	書込み Ready 信号	-
I	S	1	M_AXI_BID	応答 ID	0
I	S	2	M_AXI_BRESP	書込み状態応答	0 (OKAY)
I	S	1	M_AXI_BUSER	ユーザ定義信号	0
I	S	1	M_AXI_BVALID	書込み応答 Valid 信号	-
O	S	1	M_AXI_BREADY	書込み応答 Ready 信号	axi_bready
O	S	1	M_AXI_ARID	読込みアドレス ID	0
O	S	32	M_AXI_ARADDR	読込みアドレス	40000100
O	S	8	M_AXI_ARLEN	読込みバースト長	04 (5 回)
O	S	3	M_AXI_ARSIZE	読込みバーストサイズ	7 (128Byte)
O	S	2	M_AXI_ARBURST	読込みバーストタイプ	1 (INCR)
O	S	1	M_AXI_ARLOCK	読込み排他制御	0
O	S	4	M_AXI_ARCACHE	読込みメモリタイプ	2
O	S	3	M_AXI_ARPROT	読込みプロテクションタイプ	0
O	S	4	M_AXI_ARQOS	読込み QoS	0
O	S	1	M_AXI_ARUSER	ユーザ定義信号	1
O	S	1	M_AXI_ARVALID	読込み Valid 信号	axi_arvalid
I	S	1	M_AXI_ARREADY	読込み Ready 信号	-
I	S	1	M_AXI_RID	読込み ID	0
I	S	1024	M_AXI_RDATA	読込みデータ	-
I	S	1	M_AXI_RRESP	読込み状態応答	0 (OKAY)
I	S	1	M_AXI_RLAST	読込みバースト最終フラグ	-
I	S	1	M_AXI_RUSER	ユーザ定義信号	0
I	S	1	M_AXI_RVALID	読込み Valid 信号	-
O	S	1	M_AXI_RREADY	読込み Ready 信号	axi_rready

4.4.2 内部配線

FIX プロセッシングユニットにおいて、ロジックの見通しを良くするために表 4.13 に示す 18 個の内部配線を定義している。

各配線の割り当てに関しては付録のリスト 6.15 に掲載している。

表 4.13: 内部配線

設計	ビット幅	名称	役割
S	1	wnext	WRITE 時のネクストデータ要更新フラグ
S	1	rnext	READ 時のネクストデータ到達フラグ
S	1	init_txn_pulse	開始信号立ち上がりパルス
F	1	reading	READ 中フラグ
F	1	writing	WRITE 中フラグ
F	1	state_changing	状態遷移フラグ
F	16	segment_len_in	入力パケットのセグメント長
F	16	segment_len_out	出力パケットのセグメント長
F	32	tcp_sum	TCP 層チェックサム用合計値
F	32	ip_sum	IP 層チェックサム用合計値
F	112	header_frame	Ethernet ヘッダ
F	160	header_ip	IP ヘッダ（チェックサム計算前）
F	160	header_tcp	TCP ヘッダ（チェックサム計算前）
F	32	price	入力 FIX メッセージの価格
F	160	header_ip_c	IP ヘッダ（チェックサム計算後）
F	160	header_tcp_c	TCP ヘッダ（チェックサム計算後）
F	432	header	L2-L4 の全ヘッダ
F	1024	reader	AXI READ のデータリーダ

4.4.3 レジスタ

FIX プロセッシングユニットは 64 種類のレジスタを持つ（表 4.14、表 4.15 参照）。レジスタの更新規則である always 構文は 14 個存在し、各レジスタがどの更新規則で更新されるかを「更新 No.」の項目で示す。

表 4.14: レジスタ（1）

設計	ビット幅	名称	更新 No.	役割
S	1	init_txn_ff	1	開始信号パルス
S	1	init_txn_ff2	1	開始信号パルス（1 クロックのラグ）
S	1	axi_awvalid	2	WRITE アドレス Valid 信号
S	1	axi_wvalid	3	WRITE データ Valid 信号
S	1	axi_wlast	4	WRITE 最終データフラグ
S	1024	axi_wdata	5	WRITE データ
S	1	axi_bready	6	WRITE 完了メッセージ受信信号
S	1	axi_arvalid	7	READ アドレス Valid 信号
S	1	axi_rready	8	READ データ 受信信号
S	1	burst_write_active	9	一連のバースト WRITE 中フラグ
S	1	burst_read_active	10	一連のバースト READ 中フラグ
S	1	start_single_burst_write	11	WRITE 開始シグナル
S	1	start_single_burst_read	11	READ 開始シグナル
F	2	state	11	状態変数
F	2	state_bfr	11	1 クロック前の状態変数
F	1	softd_set	11	逆指値設定済みフラグ
F	8	counter_send	11	累積送信回数
F	16	counter_packet	11	累積パケット処理回数
F	16	counter_fix	11	累積 FIX メッセージ処理回数
F	24	counter_read	11	累積 READ 回数
F	1	refresh	11	READ 関連レジスタリセット
F	1	refresh_all	11	レジスタリセット

表 4.15: レジスタ（2）

設計	ビット幅	名称	更新 No.	役割
F	1	read_done	12	READ 完了フラグ
F	1	packet_fix	12	受信フレームが FIX メッセージ
F	4	counter_r	12	READ 状態のクロックカウンタ
F	1	packet_received	12	新規に Ethernet フレーム受信フラグ
F	8	softd_number	12	アプリ指令番号
F	1	softd_updated	12	アプリ指令更新フラグ
F	80	fix_judge	12	(デバッグ用)
F	1	err_s	12	READ 中のエラーフラグ
F	4	cur_no	12	アプリ指令番号
F	32	order_id	12	アプリ指令オーダー番号
F	32	price_trigger	12	逆指値トリガー価格
F	1	side_b	12	逆指値の Buy/Sell フラグ
F	32	symbol_target	12	逆指値対象銘柄
F	8*256	fix	12	送信予定の FIX メッセージ
F	16	msg_len	12	送信予定 FIX メッセージの長さ
F	48	mac_from	12	受信フレームの Ethernet ヘッダ項目
F	48	mac_to	12	受信フレームの Ethernet ヘッダ項目
F	32	ip_from	12	受信パケットの IP ヘッダ項目
F	32	ip_to	12	受信パケットの IP ヘッダ項目
F	16	packet_len	12	受信パケット長（IP ヘッダ項目）
F	16	port_from	12	受信セグメントの TCP ヘッダ項目
F	16	port_to	12	受信セグメントの TCP ヘッダ項目
F	32	msgno_to	12	受信セグメントの TCP ヘッダ項目
F	32	sequence_no	12	受信セグメントの TCP ヘッダ項目
F	18*64	fix_s1	12	FIX メッセージ合計値（1 段目）
F	21*8	fix_s2	12	FIX メッセージ合計値（2 段目）
F	8*256	fix_in	12	受信した FIX メッセージ
F	1	judged	13	判定済みフラグ
F	1	send_ok	13	判定結果の送信 OK フラグ
F	4	counter_j	13	JUDGE 状態のクロックカウンタ
F	1	err_j	13	JUDGE 中のエラーフラグ
F	32	onehot_s	13	FIX 内の Symbol の開始位置検索用
F	64	onehot_p	13	FIX 内の Price の開始位置検索用
F	5	index_s	13	Symbol データの開始位置
F	6	index_p	13	Price データの開始位置
F	32	fix_symbol	13	受信 FIX メッセージの銘柄名
F	64	fix_price	13	受信 FIX メッセージの価格データ
F	32	fix_sum	13	FIX メッセージの合計値（TCP 層チェックサムの計算用）
F	16	checksum_ip	13	IP 層のチェックサム
F	16	checksum_tcp	13	TCP 層のチェックサム
F	8*256	messages	13	出力フレーム
F	4	counter_w	14	WRITE 状態のクロックカウンタ
F	1	written	14	WRITE 完了フラグ

4.4.4 レジスタ更新ルール

レジスタの更新はM_AXI_ACLKの立ち上がりエッジで同期的に行われる。更新規則1〜10はサンプル由来のレジスタ値の更新に関するルールで、更新規則5（送信バッファaxi_wdataの更新）以外はサンプル回路に即している。更新規則11は状態遷移とAXI通信を開始するシグナルの発生に関わる更新規則で、更新規則12〜14はそれぞれREAD_AXI、JUDGE、WRITE_AXIの状態に固有の処理である。更新ルール1〜14を実装したVerilog HDLのコードは付録に掲載している。

これらのレジスタ更新規則の中で実現方法に検討の余地がある「TCPチェックサムの計算方法」と「FIXメッセージから銘柄・価格を抽出する方法」の2項目について詳述する。

TCPプロトコルのチェックサム計算

TCPプロトコルのチェックサムの計算は、疑似ヘッダ、TCPヘッダ、TCPペイロード（FIXメッセージ）に関して、16ビットずつで区切られた各項目を全て加算して得られる。特にFIXメッセージ（最大202バイト＝101個）の和を計算する必要があり、1クロックサイクルの計算は困難である。FIXプロセッシングユニットではリスト4.1に示すfunction文で24ビット8入力の和演算ブロックを作成し、3クロックサイクルでFIXメッセージのチェックサム計算用合計値を計算している。

リスト4.1: 和演算ブロック

```
1     function[23:0] sum(input[23:0] in0, in1, in2, in3, in4, in5, in6, in7);
2     begin
3         sum=((in0+in1)+(in2+in3))+((in4+in5)+(in6+in7));
4     end
5     endfunction
```

FIXメッセージからの情報抽出

FIXは可変長の文字列に情報を載せるタイプのプロトコルであるため、READ後にビット列から必要なデータを拾い上げる必要がある。本研究では4.3.4節で述べたようにSymbol(銘柄名）と(3番目の)MdEntryPx（取引価格）を取得する。

FIXメッセージではタグ番号と値の間は0x3D(”=”)、タグ値と次のタグ番号の間は0x01(”･”で表す)で区切られているため、Symbol(タグ番号55)のデータ項目の直前4文字は必ず”･55=”の文字列となり、逆に”･55=”の後は必ずSymbolのデータ項目が続く。

そこで、onehotレジスタを用意して、FIXメッセージのi番目以降4文字が”･55=”の場合にonehot[i]=1、それ以外の場合はonehot[i]=0となるように決めれば、onehotは1ビットのみが1の値を取る。また、そのビット位置+4がSymbolデータの開始位置となる。

取引価格の取得も同様の手順で可能だが、価格(MDEntryPx：タグ番号270)の場合、同一タグ番号のデータが3つあるためにもう一工夫必要である。そこで、MDEntryPxの直前の項目(MDEntryType)に着目し、MDEntryPxが取引価格を意味するのはMDEntryTypeが

2 の場合という事実を用いる。すなわち、取引価格データを取得するためには、”2・270=” を探せばよい。

回路上は、リスト 4.2 のように、1 クロック目に onehot を作成し、2 クロック目に onehot からビット位置 index の取得、 3 クロック目に fix_symbol(銘柄名) の取得を行っている。

また、onehot レジスタは FIX メッセージの最大数 (202）用意すれば足りるが、上記の処理を行う回路を合成すると、「202 個× 8bit の入力データから index の候補を出力するブロックを 202 個作成し、202 個の候補を統合して index を出力する回路」が生成され、回路規模が膨大になる。加えて経路段数が増えるために配線遅延が生じ、最大遅延パスの遅延も 1 クロックの長さ (20ns) を大きく超えてしまう。

そこで、表 4.9 の文字数の上限値の設定を利用し、各項目が取りうる範囲に制限して探索するようにする。すなわち、Symbol に関しては開始位置の最小値が 64、最大値が 87 で 24 通りの可能性があり、MDEntryPx に関しては開始位置の最小値が 125、最大値が 168 で 44 個の可能性があるので、onehot_s レジスタは 24 個、onehot_p レジスタは 44 個用意し、この範囲を探索すればよい（実装上は拡張性のため多少バッファを持たせそれぞれ 32 個、64 個用意している）。

リスト 4.2: FIX メッセージから情報を拾い上げる回路

```
case (counter_j)
    0: begin
        for(i=0;i<32;i=i+1) begin
            if(fix_in[i+60]==8'h01 && fix_in[i+61] == 8'h35
                && fix_in[i+62] == 8'h35 && fix_in[i+63] == 8'h3D)
                onehot_s[i] <= 1'b1;
        end
        for(i=0;i<64;i=i+1) begin
            if(fix_in[i+119] == 8'h32 && fix_in[i+120] == 8'h01
                && fix_in[i+121] == 8'h32 && fix_in[i+122] == 8'h37
                && fix_in[i+123] == 8'h30 && fix_in[i+124] == 8'h3D)
                onehot_p[i] <= 1'b1;
        end
    end
    1: begin
       for(i=0;i<32;i=i+1) begin
            if(onehot_s[i] == 1'b1) index_s <= i;
        end
        for(i=0;i<64;i=i+1) begin
            if(onehot_p[i] == 1'b1) index_p <= i;
        end
    end
    2: begin
        for(i=0;i<4;i=i+1)
            fix_symbol[31-8*i -: 8] <= fix_in[index_s+i+64];
        for(i=0;i<8;i=i+1)
            fix_price[63-8*i -: 8] <= fix_in[index_p+i+125];
    end
endcase
```

4.5 シミュレーション

本節ではFIXプロセッシングユニットを開発するためのシミュレーション方法について述べる。

4.5.1 シミュレーション用回路

デジタル回路設計にはシミュレーションによる検証が欠かせないが、本研究ではPS部のプロセッサがBRAMにEthernetフレームを書き込んでおり、PS部の挙動はシミュレートできないため、PS部の代わりにBRAMにEthernetフレームを書き込むシミュレーション用IP(myip_sim_writer)を作成する。

シミュレーション用回路を図4.3に示す。この回路ではZynqチップに実装するBRAM(BRAM0)以外に、ドライバによって書き込まれるEthernetフレームのサンプルを格納するためのBRAM(BRAM1)と、発注クライアントアプリによって書き込まれるアプリ指令データのサンプルを格納するためのBRAM(BRAM2)を持つ。

BRAM1、BRAM2はAXI経由でなく直接アドレス指定でアクセスできるStand Aloneモードに設定し、それぞれEthernetフレームとアプリ指令データをあらかじめテキストファイル(.coe)から読み格納している。シミュレーション用IPはBRAM1とBRAM2から読んだデータをAXI4 LiteでBRAM0に書き込んでいる。[5] また、シミュレーション用回路はシステムのクロックをPS部から取得できないため、Clocking Wizard IPを用いて生成している（クロック周波数は50MHz)。

なお、このシミュレーションは全て動作レベル（Behavioral Simulation)で行っている。

4.5.2 シミュレーション用IP

シミュレーション用のIPはM_AXI_ARESETN（負論理）のリセットで開始し、BRAM1からEthernetフレーム、BRAM2からアプリ指令データを読み、IP内のレジスタに格納する。両方のデータを読み終えたら準備が完了し、外部出力信号TXN_DONEをONにして通知する。

準備完了後、外部入力信号のINIT_AXI_TXNがONになる時に、アプリ指令データをBRAM0（0x0180 〜 0x027F)に書込み、書込みが完了すると、アプリ指令更新フラグ（0x0108）に通知(0xFE）する。同様に、外部信号のSWITCH_WRITE_PACKETINがONになる時に、EthernetフレームをBRAM1(0x0280〜 0x037F)に書き込み、書込み完了後、受信フレーム到達フラグ（0x0100）に通知(0xFF）する。

[5] AXI4 LiteはFullと異なりバースト転送機能がなく、データ幅も最大64ビットで書込みに時間がかかるが、シミュレーション用IPはレイテンシにセンシティブでないために実装が簡単なLite版を選択している。

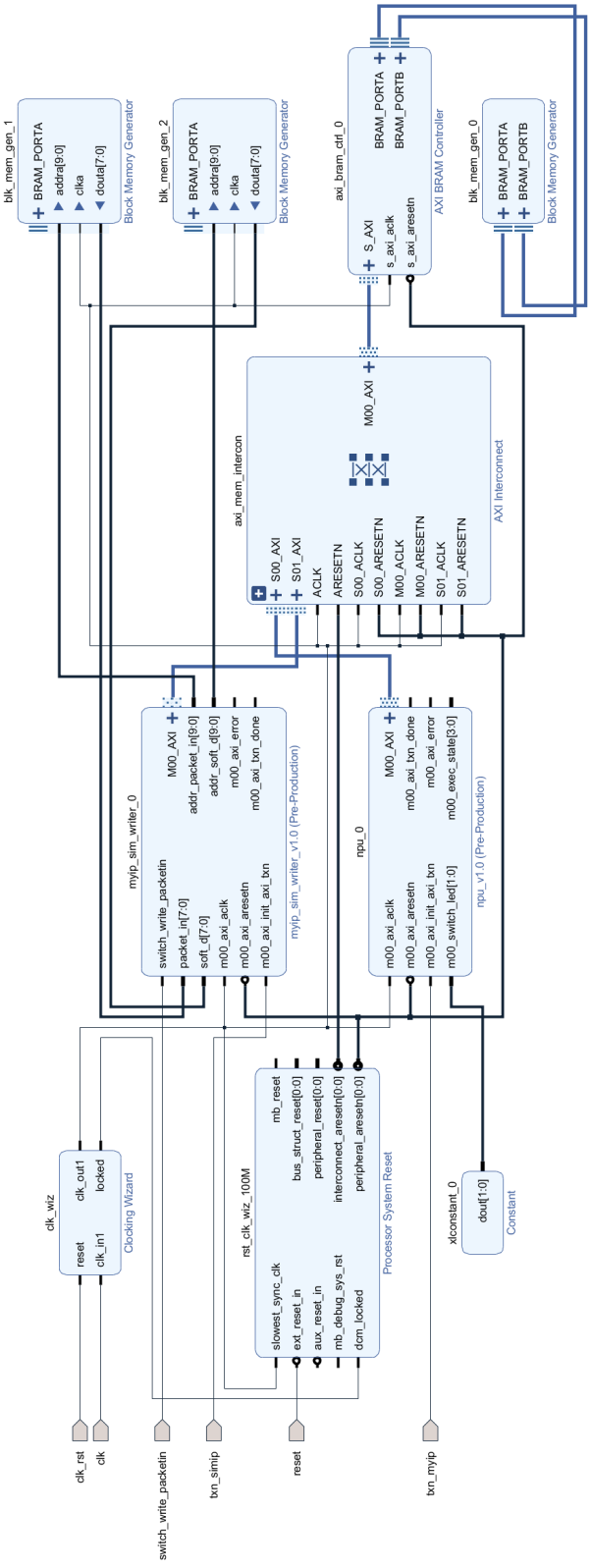


図 4.3: シミュレーション用回路

4.5.3 テストベンチ

以上のシミュレーション用の回路と IP を用いて、FIX プロセッシングユニットが正常稼働することを確認するためのテストベンチをリスト 4.3 に示す。

リスト 4.3: テストベンチ

```
module sim(
    );

    reg clk, clk_rst, reset;
    reg txn_simip, txn_myip;
    reg switch_write_packetin;

    parameter STEP=20;
    design_sim_wrapper design_sim_i
        (.clk(clk),
        .clk_rst(clk_rst),
        .reset(reset),
        .txn_simip(txn_simip),
        .txn_myip(txn_myip),
        .switch_write_packetin(switch_write_packetin)
        );

    always #(STEP/2) clk=~clk;

    initial begin
        clk=0; clk_rst=0; txn_simip=0; txn_myip=0; reset=0;
        switch_write_packetin=0;

        #(STEP*10) clk_rst=1;
        #(STEP) clk_rst=0;
        #(STEP*89) reset=1;
        #(STEP*905)
        //20ns
        txn_simip=1;
        #(STEP*5) txn_simip=0;
        #(STEP*995)
        //40ns
         // #(STEP*17)

        switch_write_packetin=1;
        #(STEP*10)  switch_write_packetin=0;
        #(STEP*990)
        //60ns (1st complete)

        txn_simip=1;
        #(STEP*5) txn_simip=0;
        #(STEP*995)
        //80ns
        switch_write_packetin=1;
        #(STEP*10) switch_write_packetin=1;
        #(STEP*990)

        //100ns
        txn_myip=1;
        #(STEP*5) txn_myip=0;
        #(STEP*995)

        //120ns (2nd complete)
        $finish;
    end
endmodule
```

4.5.4　タイミングチャート

リスト 4.3 のテストベンチを用いてシミュレーションを行って得られたタイミングチャートが図 4.4、図 4.5、図 4.6 である。

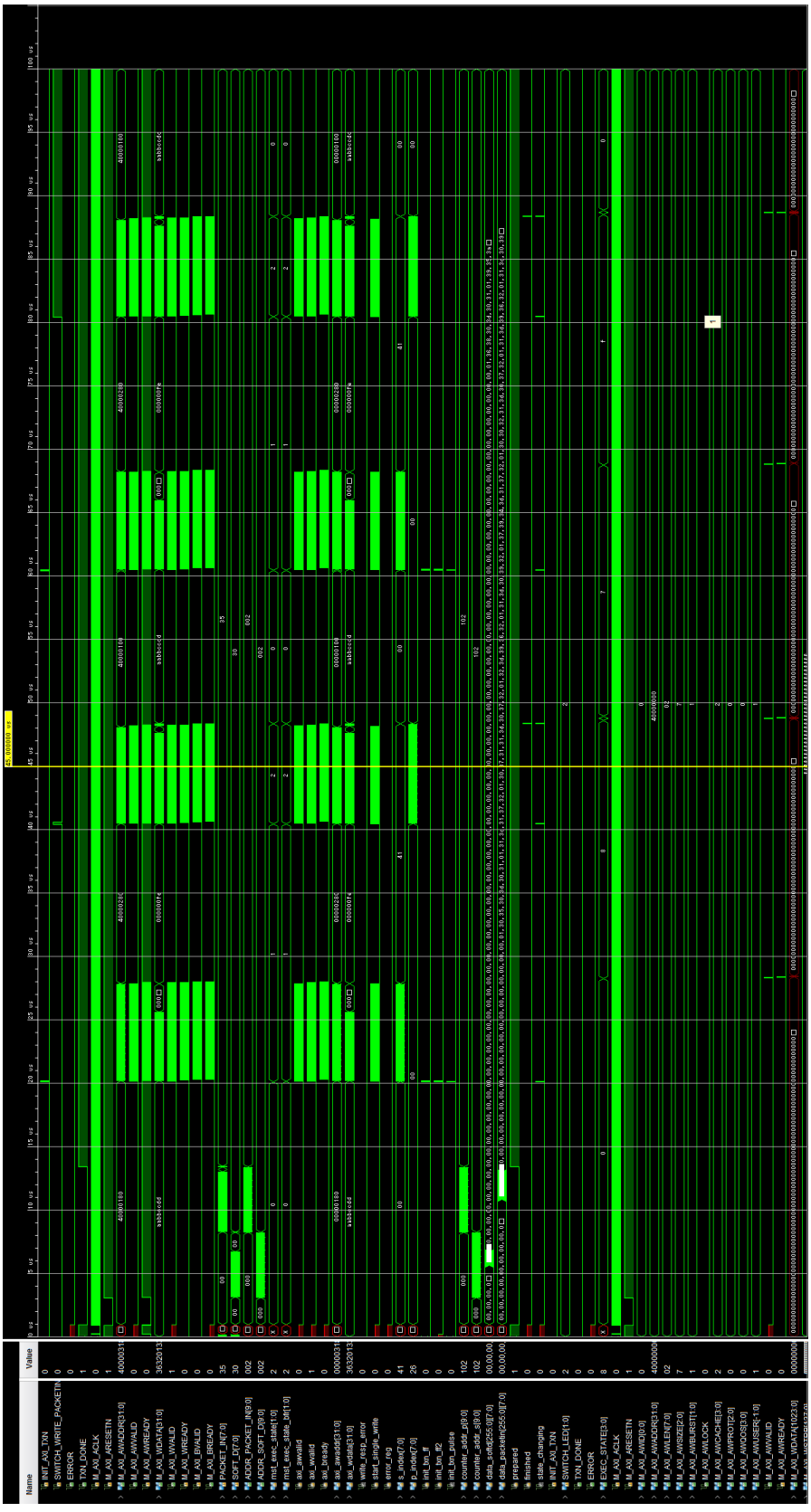


図 4.4: シミュレーション結果（1）

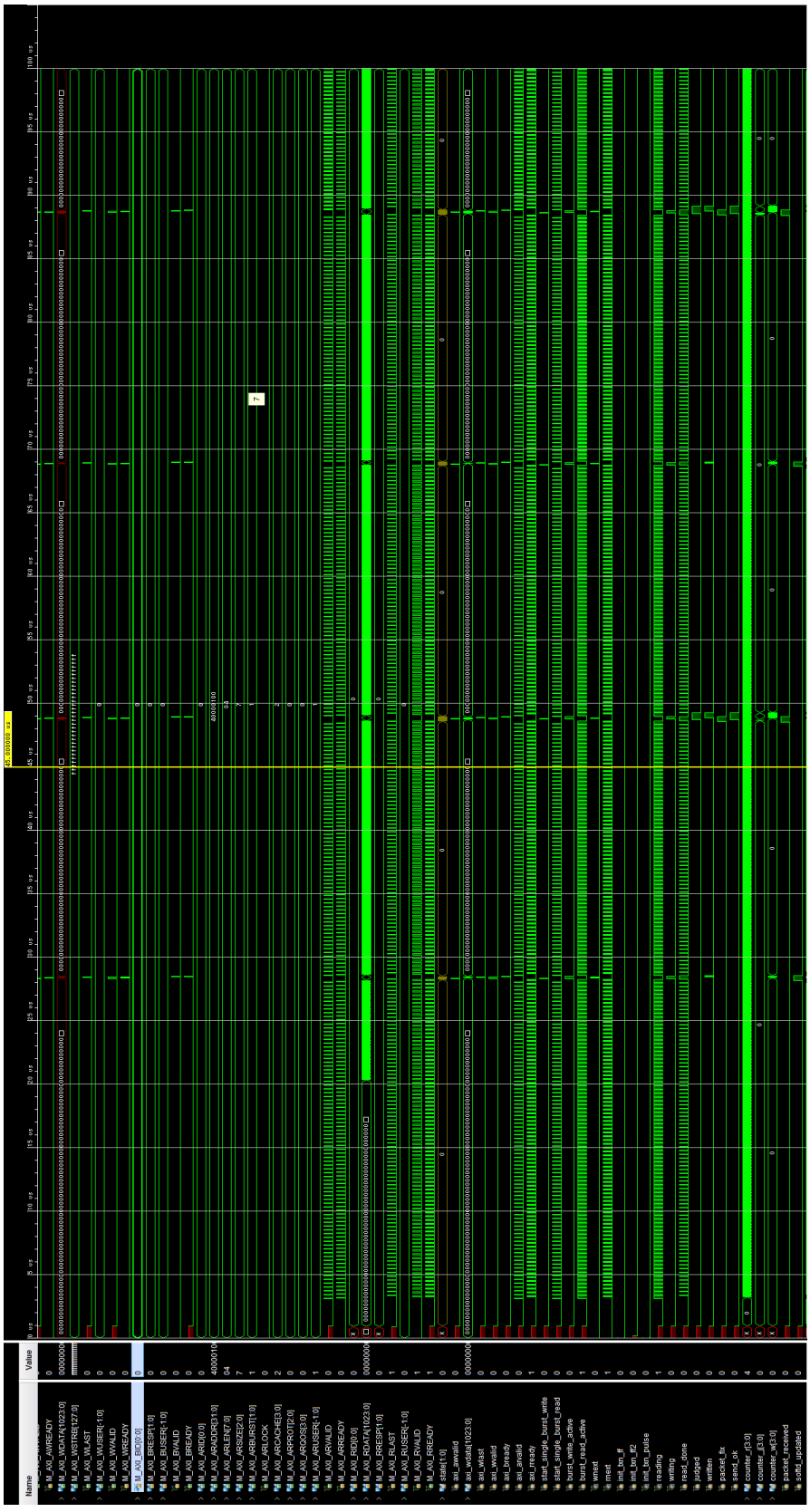


図 4.5: シミュレーション結果（2）

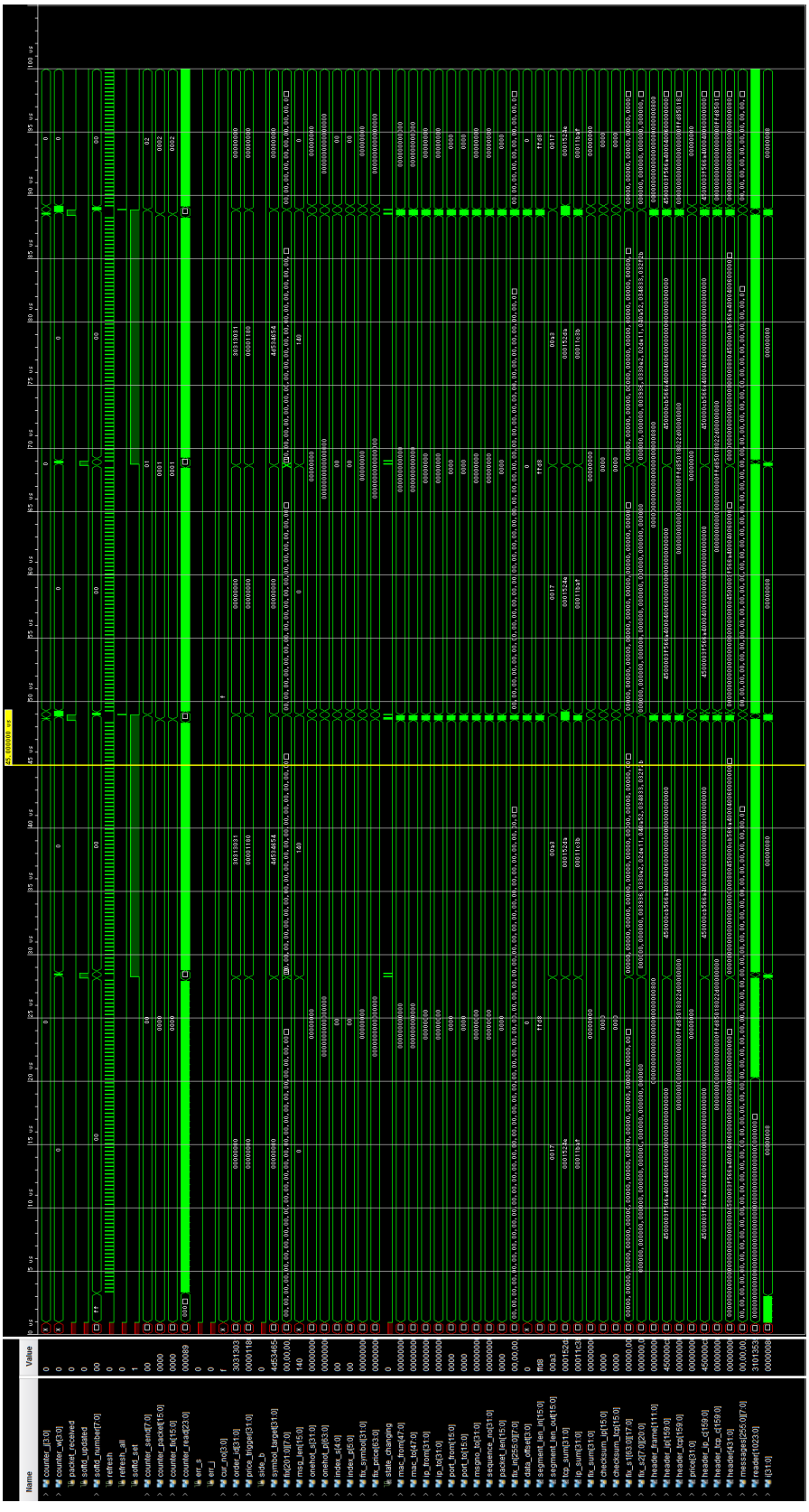


図 4.6: シミュレーション結果（3）

4.6　合成結果

設計したアクセラレータ（図 4.1 の回路) を論理合成し、配置配線を行った。各資源の利用率は表 4.16 に示すような数値になった。

表 4.16: PL 部資源利用率

Resource	Utilization	Available	%
LUT	11650	53200	21.90
LUTRAM	125	17400	0.72
FF	18737	106400	17.61
BRAM	32	140	22.86
IO	7	125	5.60
BUFG	1	32	3.13

動作周波数 50MHz(20ns/clock）に対して Worst Negative Slack(WNS) は 3.745ns であった。

アクセラレータの回路を配置した結果、デバイスの利用状況は図 4.7 のようになった。

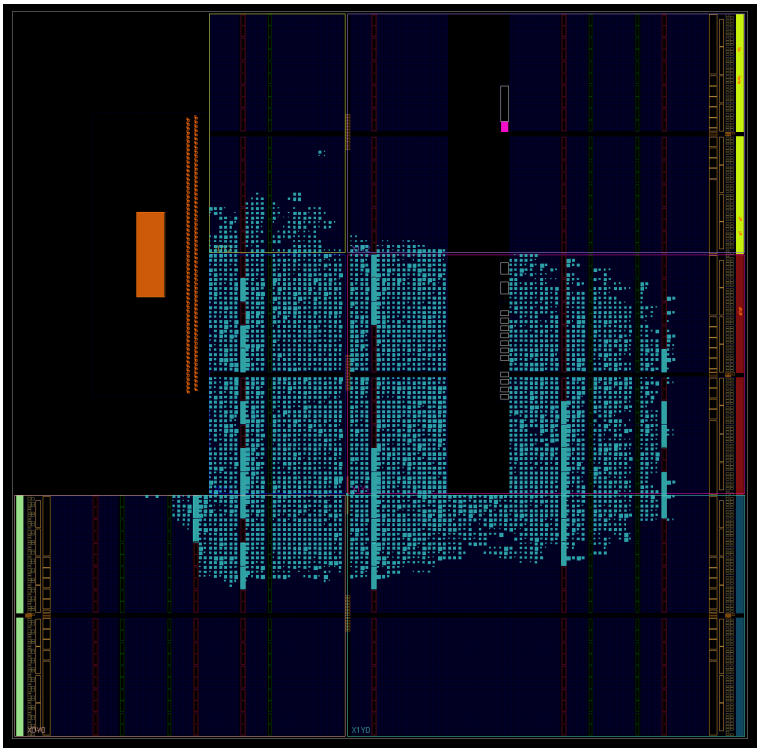


図 4.7: 回路配置結果

4.7　ソフトウェアの修正

Zynq の PL 部を利用するために、発注クライアントアプリとネットワークのデバイスドライバを修正する必要がある。本節ではこのソフトウェアの修正に関して記述する。

4.7.1　発注クライアントアプリの修正 - BRAM へのアクセス

発注クライアントから BRAM にアプリ指令データを書き込むためにソフトウェアを書き換える必要がある。ユーザ空間において、C/C++では mmap システムコールを使い物理アドレスを unsigned char 型のポインタに直接マッピングすることができる。具体的なコードをリスト 4.4 に示す。このコードでは、0x0180〜0x027F にアプリ指令データ（表 4.10 参照）を書き込み、最後に 0x0108 にアプリ指令更新フラグを書き込んでいる。

リスト 4.4: BRAM に書き込むための C++コード

```
1  int fd;
2  void *mem_map;
3  volatile unsigned char *m;
4
5  fd = open("/dev/mem", (O_RDWR | O_SYNC));
6  mem_map = mmap(NULL, 0x1000, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, 0x40000000);
7  m = (unsigned char*)mem_map;
8
9  for (i = 0; i < 16; i++)
10 {
11   m[i+384] = sc[i];
12 }
13
14 int size = orderHILO.toString().size();
15
16 for (i = 0; i < size; i++)
17 {
18   m[i + 384 + 16] = orderHILO.toString()[i];
19 }
20
21 m[256+8] = (unsigned char) (n % 254 + 1); //01_FE
```

4.7.2　発注クライアントアプリの修正 - FIX メッセージのアップデート

FIX メッセージの作成は発注クライアントアプリに委ねているが、アプリは一度作成した FIX メッセージを BRAM に書きこむだけでなく、そのメッセージが送信されるか取り消されるまで、適宜更新し続ける必要がある。その理由は、FIX メッセージの中に、MsgSeqNum（メッセージ通番）、SendingTime（送信時刻）の項目があることに起因する。

MsgSeqNum は FIX メッセージが送信される度に＋ 1 されるため、例えば、前回送信されたメッセージの MsgSeqNum が 100 の時点で逆指値をハードに設定する場合、MsgSeqNum=101 の FIX メッセージを BRAM に書き込むが、その後アプリが MsgSeqNum=101, 102, 103, ... のメッセージを送信することがある。サーバ側は MsgSeqNum の番号順に届くことを期待するため、設定された逆指値が有効な FIX メッセージであり続けるために

は、FIX メッセージが送信される度に、その都度 BRAM に格納される FIX メッセージが更新される必要がある。

そこで、発注クライアントを修正し、BRAM 上に逆指値が設定されている場合は、FIX メッセージを送信する直前に BRAM の累積送信回数 (0x010E) を読み、累積送信回数が+1 されていない（＝設定された逆指値が現在も有効である) 場合、新しい MsgSeqNum の値で再度 FIX メッセージを作成し、BRAM を更新することにしている。

なお、FIX エンジンによって 30 秒に一度ハートビートメッセージが送られているので、上記の BRAM 更新を導入することで SendingTime のずれも最大でも数十秒程度に収まり、サーバ側は正常なメッセージと認識する。[6]

4.7.3 ネットワークドライバの改造 - BRAM へのアクセス

本研究ではイーサネットコントローラが受信したデータを PL 部の BRAM に送り、FIX プロセッシングユニットからアクセスできるようにしている。また、Linux の TCP/IP プロトコルスタックが管理する通常の送信処理に加え、BRAM の指定領域に送信フレームが書き込まれた時にも送信するようにしている。この仕様を実現するためにデバイスドライバを修正する必要がある。本研究では、イーサネットコントローラのデバイスドライバ (xemacps) のソースコード (xilinx_emacps.c) を改造する。[7]

まず、ドライバを初期化する時に呼び出される関数 xemacps_descriptor_init 内において、受信フレームの転送先アドレスを BRAM 内のアドレス (0x0280〜) に変更し、このアドレスを配列 (xmap) にマッピングする。受信時のフレームの転送先メモリアドレスは DRAM 内に存在する受信バッファディスクリプタによって管理されており、イーサネットコントローラは（受信バッファでなく）受信バッファディスクリプタのリストのベースアドレスを保持している。[8] そこで、受信バッファディスクリプタが参照しているアドレスを BRAM の受信バッファ領域の先頭アドレス (0x0280) に設定する。ドライバはカーネル内で起動しており mmap システムコールは使えないため、ioremap_nochache 関数を用いて BRAM のアドレスにマッピングする。

次に、Ethernet フレームを受信した時に発生するハードウェア割り込みの手続きの中で呼ばれる関数 xemacps_rx 内において、FPGA が生成した注文の送信処理を行うように変更する。送信に関しても、送信バッファのアドレスは各送信バッファディスクリプタが管理しており、イーサネットコントローラは送信バッファディスクリプタリストのベースア

[6]仮にサーバ側の実装が時刻のずれをより厳しく管理するようなものであっても、発注クライアント側で FIX メッセージ送信時に加え、1 秒おきに BRAM を更新するタイマイベントを設定する等の方法で容易に対応可能である。

[7]PYNQ の Linux イメージファイルがデフォルトで用いているデバイスドライバ (macb) ではなく xemacps を利用する。起動時のデバイスドライバを変更するには、デバイスツリーのソースファイル devicetree.dts を修正し、コンパイルして得られるデバイスツリーファイル devicetree.dtb を利用する。

[8]受信バッファディスクリプタの個数はデフォルトでは 256 に設定されているが、BRAM を利用するという目的においては 1 個あれば十分であり、受信バッファディスクリプタ数は 1 に設定している。

ドレスを保持している。[9]

xemacps_rx 関数を改造するための詳細な手順を示す。

1. 受信フレーム到達フラグ (0x0100) に 0xFF を書き込む。

   → PL 部が Ethernet フレームの処理を開始する。

2. フレーム処理完了フラグ (0x0104) の値を読む。

   - 0xAA の場合、PL 部は送信すべきフレームを生成していないため 5. に進む。
   - 0xFF の場合、3. に進みハードウェア生成注文の処理を行う。
   - それ以外の値の場合は 0xAA または 0xFF が書き込まれるまで待つ。

3. 送信バッファディスクリプタの参照アドレスを BRAM 内の送信バッファ領域の先頭アドレス (0x0000) に指定する。また、バッファ長をフレームサイズに設定する。

4. イーサネットコントローラ内のネットワークコントロールレジスタの start_tx（送信開始）フラグを立てる（xemacps_write 関数）。

   →イーサネットコントローラが送信フレームを取得し、送信を開始する。

5. フレーム処理完了フラグ (0x0104) の値をクリア（0x00) する。

6. Linux が受信処理を行えるように BRAM の受信バッファからソケットバッファに Ethernet フレームをコピーする。

7. 以下、通常の受信処理を行う。

このようにネットワークドライバを改造することで、イーサネットコントローラは Ethernet フレームの受信時に BRAM に DMA 転送できる。その後の処理はドライバ内で行われており、ソフトウェアの関与はゼロではないが、xemacps_rx が呼ばれるハードウェア割り込みコンテキスト内で完結するため、ネットワーク処理に要する遅延の大部分が削減できる。

4.7.4 ネットワークドライバの改造 - TCP 及び FIX の整合性確保

本研究において、Linux 上で起動する発注クライアントがサーバとセッションを張り通信をしている中、PL 部の FIX プロセッシングユニットが独自に Ethernet フレームを生成し、そのフレームをサーバに送信している。そのために Linux の TCP 層で管理しているシーケンス番号、及び FIX エンジンで管理している MsgSeqNum 番号の整合性が取れなくなり、サーバ側のソフトウェアにエラーと判断され通信セッションが切断されてしまう。

[9]送信バッファディスクリプタはデフォルトで 256 個存在するが、BRAM から送信するために 1 個だけ利用し、残りの 255 個は通常の OS 経由の送信用に確保している。

この整合性問題を解決するために、逆指値注文を BRAM に登録する際に、発注クライアントのメモリにも同様の逆指値注文を登録して、PL 部が生成する新規注文メッセージを送信することとなった場合、発注クライアントからも同一の新規注文のダミーメッセージを生成するようにする。そして、二重に注文が送出されるのを防ぐために、Ethernet ドライバの送信処理を行う関数 xemacps_start_xmit を修正し、PL 部から送出されたフレームのシーケンス番号と同一のシーケンス番号のフレームに関しては破棄し、実際に送信されないようにしている。このようにダミーメッセージを生成し、送信の直前で破棄することで、Linux の TCP 層及び FIX エンジンで管理されるメッセージの整合性を担保している。[10]

以上の方法でドライバのソースコード (xilinx_emacps.c) を変更し、Linux 全体を再コンパイルしてイメージファイル uImage を再作成する。

[10] ダミーメッセージの生成によらず Linux の TCP 層のソースコードを変更するのが本来の修正方法に思えるが、ドライバでなく Linux カーネル本体を修正する必要があることと、表面的なにシーケンス番号を変更するだけでは整合性問題は解決せず（PL 部生成メッセージに対する Ack がサーバ側から送られてきたときにクライアントの TCP 層は通信異常と判断してしまう）、修正が困難であったために断念した。

第5章　遅延の計測と考察

5.1　回路遅延

FPGA のデジタル回路で発生する遅延は、「受信バッファに価格情報フレームデータが書き込まれた瞬間から送信バッファに新規注文フレームデータが書き込まれる瞬間までのクロックサイクル」に動作周期（動作周波数の逆数）を掛けて算出できる。

PL 部がデータを読んでから書き込むまでのクロックサイクルは基本的には毎回同一値であるが、FIX プロセッシングユニットはパケット処理時以外は常に READ 状態で、BRAM に対して READ 命令を発行し続けている。一回の AXI READ 命令の開始から終了までには 15 クロックを要するため、その中のどのタイミングで受信バッファ書込み完了フラグがセットされるかによって必要なクロックサイクルに差異が生じる。

実際に、1 クロックずつ書込み完了フラグをセットするタイミングをずらして調べてみたところ、AXI READ に必要なクロックサイクル数は最小で 12、最大で 26 であった (一様分布なので平均は (12+26)/2=19 クロック)。また、発注すべきか判定する JUDGE フェイズに要するクロックサイクルは 5 で、BRAM に全データが書き込まれるのに必要なクロックサイクルは 8 であった。よって、最小で 12+5+8=25、最大で 26+5+8=39、平均で 19+5+8=32 となる。動作周波数は 50MHz に設定しており、動作周期は 20ns である。よって、回路遅延は平均で 32*20=640ns、 最大で 39*20=780ns と算出できる。

5.2　トータル遅延の計測

通信の相手側が価格情報メッセージを送信してから新規注文メッセージを受信するまでのトータル遅延に関しても、アクセラレータを用いた場合と、従来のソフトウェア処理により注文メッセージを送出した場合とを比較し、レイテンシ削減効果を確かめた。計測方法は、サーバ上で動作するパケットキャプチャツール（Wireshark) のタイムスタンプによるもので、サーバが価格情報メッセージを送信した時刻と新規注文メッセージを受信した時刻の差として測定できる遅延である。

比較対象として、PYNQ-Z1 ボード以外にも、実際の運用環境に近づけるために Linux(Fedora20) 及び Windows7 マシンでのソフトウェア処理も計測した（表 5.1 参照）。[1]

表 5.1: トータル遅延 (μs)

統計量	本研究	PYNQ-Z1	Fedora20	Windows7
サンプル数	52	53	51	54
平均値	148	1573	684	6420
中央値	139	1276	680	6329
最大値	254	6911	794	29676
最小値	89	1161	577	2533
標準偏差	41	1119	56	3385

平均値及び中央値を比較すると、本研究で設計したハードウェアアクセラレータを用いると、同じ PYNQ-Z1 ボードのソフトウェア処理の 9 割の遅延を削減でき、Fedora マシン上のソフトウェア処理と比べても 8 割程度の遅延削減効果があることが確かめられる。また、この計測値にはサーバマシン内部で生じている遅延も含まれ、クライアントの責任領域で生ずる遅延の削減率に関する下限値と解釈できる。[2]

5.3 受信→判断→送信までの処理の流れ

ソフトウェア処理とハードウェア処理の遅延に関して考察を加えるために、イーサネットケーブルから届いた受信信号に対してアプリケーションプログラムが判断を行い、ケーブルに送信信号を送出するまでの流れを本研究の環境（1 ギガビットイーサネット, PYNQ-Z1, Linux, QuickFIX) を前提にまとめた。以下の経路は基本的に最短の場合で、プロセスやバッファの状態によってはすぐに先に進めず Wait 状態になり更に遅延が生ずるケースもある。

[1]Linux マシンのスペックは CPU:Corei7-3770 3.4GHz, Ethernet Controller: RealTek 8169, OS: Fedora 20 64bit, Kernel: 3.11.10。

Windows マシンのスペックは CPU:Corei7-2600 3.4GHz, Ethernet Controller: RealTek 8168, OS: Windows7 Ultimate SP1 64bit。

[2]ソフトウェア実行において PYNQ-Z1 の遅延が Fedora マシンの倍近い数値になっているのは主に CPU の動作周波数の差異（PYNQ-Z1(CortexA9): 650MHz, Fedora マシン (Corei7-3770):3.4GHz）によると考えられる。また、Windows7 の遅延値が際立って高いが、必ずしも WindowsOS 自体の性質ではなく、QuickFIX の Windows 版実装固有の現象と考えられる。実際に Windows7 マシンで別の通信アプリケーションプログラムの遅延を計測したところ 1 ミリ秒以下であった。

5.3.1 ソフトウェア処理の場合

ハードの受信処理

1. LAN ケーブルから送られてきた電気信号が RJ-45 コネクタ（LAN ポート）に到達し、信号が PHY チップに送られる。
2. PHY は受信した信号を復号し、誤り訂正を施す。得られたフレームを RGMII で Zynq の PS 部に送る。
3. Zynq の PS 部に GMII to RGMII アダプタが存在し、RGMII の信号を GMII に変換し、イーサネットコントローラに渡す。
4. イーサネットコントローラは FCS の確認後、受信したフレームを PS 部の DRAM コントローラ経由で DRAM（システムメモリ領域）の受信バッファに DMA 転送する。
5. イーサネットコントローラは DMA 転送後に受信完了割り込みを生成し、割り込みコントローラ（GIC) に通知する。
6. 割り込みコントローラは CPU の 2 つのコアから割り込みをかけるコアを選択し、割り込み禁止状態になければプロセッサに割り込みを通知する。
7. 割り込み信号を受け取ったプロセッサはハードウェア割り込みハンドラを起動させる。

Linux カーネルの受信処理

1. ハードウェア割り込みハンドラが Ethernet ドライバの xemacps_rx 関数を呼び出す。この関数内でシステムメモリ内の受信バッファからカーネルのソケットバッファにフレームをコピーする。
2. ハードウェア割り込みハンドラはパケットをキューに格納し、ソフト割り込みを発生させ、処理を終了する。
3. ソフト割り込みハンドラが起動し、キューからパケットを呼び出し IP 層の処理に移る。
4. IP 層ではヘッダのチェックサムを確認し、パケットが断片化されていない場合は TCP 層の処理に移る。
5. TCP 層でも種々の確認・処理を行い、最終的にアプリケーション領域のバッファにフレームを書き込む。

アプリケーションの処理

1. FIX エンジンが受信用のスレッドを用意し、その受信用スレッドはアプリケーション用バッファ領域をポーリングで読み続ける。バッファにソケット情報が書き込まれた時に FIX エンジンは処理を開始し、FIX アプリケーションプログラムの受信イベント (fromApp) を発生させる。

2. FIX アプリケーションのユーザロジックが FIX メッセージを処理し、発注判断を行い、発注する場合は FIX エンジンで定義されている送信関数 (sendToTarget) を呼び出す。

3. FIX エンジンが send システムコールを発行する。

Linux カーネルの送信処理

1. send システムコールが呼び出されカーネルモードに移行する。TCP 層の sendmsg 関数を呼び出し、ソケットバッファを作成し、メッセージをコピーする。

2. TCP 層は tcp_transmit_skb 関数にソケットバッファを渡し、TCP ヘッダを作成し、IP 層の ip_queue_xmit 関数を呼び出す。

3. IP 層は IP ヘッダを作成し、ARP 解決を行い、データリンク層の dev_queue_xmit 関数を呼び出す。

4. データリンク層はパケットをデバイスの送信キューに入れ、qdisc_run 関数を実行する。

5. キューからパケットを取り出し、ネットデバイスの hard_start_xmit メソッドを呼び出す。この結果 Ethernet ドライバの xemacps_start_xmit 関数が呼ばれる。

6. xemacps_start_xmit 関数内でイーサネットコントローラ内のネットワークコントロールレジスタの start_tx（送信開始）ビットを立てる。

ハードの送信処理

1. イーサネットコントローラは APB バス経由でドライバの命令を受け取り、start_tx ビットが ON になると、DRAM 内の送信バッファから DRAM コントローラ経由でフレームを取得し、コントローラ内の TX パケットバッファに格納する。

2. イーサネットコントローラ内の MAC トランスミッタがフレームに FCS とプリアンブルを付加して GMII バスに送信する。

3. PS 部の GMII to RGMII アダプタで RGMII の信号に変換され、フレームが PHY に送られる。

4. PHY はフレームを 1000BASE-T で伝送するために 8B1Q4 符号化方式により符号化し、1 バイトを 5 値データに変換する。この 5 値データを 4D-PAM5 多値変換方式で 4 対のツイストペアケーブルに乗せて 125Mbaud でベースバンド伝送する。

5.3.2 ハードウェア処理の場合

ハードの受信処理

（4 以外はソフトウェア処理と同じ。）
4'. イーサネットコントローラは FCS の確認後、受信したフレームを PL 部の BRAM 内の受信バッファに DMA 転送する。

Linux カーネル受信処理→ FPGA → Linux カーネル送信処理

1. ハードウェア割り込みハンドラが Ethernet ドライバの xemacps_rx 関数を呼び出す。この関数内で PL 部の受信フレーム到達フラグを 0xFF にし、FIX プロセッシングユニットの処理が開始される。

2. FIX プロセッシングユニットは BRAM からフレームを読み、発注判断を行い、発注する場合は BRAM の送信バッファにフレームを書き込み、フレーム処理完了フラグを 0xFF にする。

3. xemacps_rx 関数に処理が戻り、イーサネットコントローラ内のネットワークコントロールレジスタの start_tx（送信開始）ビットを立てる。

ハードの送信処理

（1 以外はソフトウェア処理と同じ。）
1'. イーサネットコントローラは APB バス経由でドライバの命令を受け取り、start_tx ビットが ON になると、PL 部の BRAM 内の送信バッファからフレームを取得し、コントローラ内の TX パケットバッファに格納する。

5.4 考察

アプリケーションソフトウェアを用いた通常のネットワーク通信において、受信したフレームデータはアプリケーションのユーザロジックに届くまで、前節で示したような多重

階層を経ることにより不可避的に遅延が発生してしまう。具体的には、カーネルからユーザランドへの遷移、ユーザランドからカーネルへの遷移が最低 1 回は発生し、コンテキストスイッチのオーバヘッドが見込まれる。加えて、ソフトウェア割り込みで処理されるため、CPU 時間の割り当てに不確実性があり、他のプロセスの影響も受けやすい。

本研究の方法では一部にソフトウェアを介在させているが、カーネル内のハードウェア割り込みコンテキストでの処理に限られるため、ソフトウェアで発生する遅延の大部分を削減できると考えられる。

第6章　研究の発展性とまとめ

6.1　実用化に向けて

本節では本研究で設計したハードウェアアクセラレータを実際の低遅延取引環境で利用するために考慮すべき点に関して触れる。

6.1.1　FIX 以外のプロトコルの実装

本研究では通信プロトコルとして FIX をターゲットとしたが、FIX は情報を文字列として表現して ASCII コードを送受信するためにメッセージが長くなってしまうという弱点があり、今日の多くの取引所の取引システムでは FIX 以外の独自の通信プロトコルを定義している。[1]

そのため実用的なシステム構築には対応するプロトコルの数だけ仕様を理解し回路を設計することが必要となるが、これらの通信プロトコルの多くは文字列ではなく（TCP/IP と同じように）定められた各項目順のビット列であるために、特に受信部の設計は FIX に比べて容易である。

6.1.2　一般の取引アルゴリズムの実装

本研究ではネットワーク処理に要する遅延の計測・削減を目標に研究を行ったため、逆指値という最も単純な取引アルゴリズムのみに対応しており、またハードウェアに設定できる注文は 1 件だけで実用性は乏しい。

しかしながら、裁定取引アルゴリズム、VWAP 取引等の注文執行系アルゴリズム、将来価格を機械学習や統計分析で予測するタイプのアルゴリズムなど任意の取引アルゴリズムについて、SoC FPGA の PS 部でアプリケーションソフトウェアが OS の TCP/IP プロトコルスタックを利用して通信を行いながら、レイテンシにセンシティブな発注判断箇所のみを PL 部に置く本研究のアプローチで低遅延化することができ、より複雑な取引アルゴリズムを実装する際もシステムアーキテクチャのプロトタイプとして利用できる。

[1] 例えば東京証券取引所の取引システム Arrowhead は独自の通信プロトコルを定義しており、大阪取引所のシステム J-Gate は他の取引所（NASDAQ OMX）が定義した通信プロトコルを利用している。

6.1.3 情報源と発注先が異なる場合

本研究では通信相手の取引所シミュレータに対して、一つの TCP セッションで送信/受信を行っていたが、取引所からの価格情報受信と注文情報送信が別系統になる構成もある。また、取引アルゴリズムによってはニュースフィードなどの取引所外の情報を利用するケースもある。そのような、送信先とは異なる情報源のメッセージに起因する発注を行う場合、システム設計を少し変える必要がある。

特に問題となるのが、TCP ヘッダ内の「シーケンス番号」及び「確認応答番号」の 2 項目で、本研究では受信した情報から送信する情報を作成しているが、受信と送信の相手方が異なる場合は受信情報を用いることができないため、OS カーネル内部（デバイスドライバまたは TCP プロトコル処理部）を修正し、ソフトウェアで Ethernet フレームを送信する度に PL 部にシーケンス番号と確認応答番号の情報を書き込むようにして、PL 部はその情報を利用して TCP ヘッダを作成する必要がある。

6.1.4 トータル遅延の更なる減少

本研究においては安価で容易に入手可能な機材と環境のみを用いて低遅延取引を実現するための、いわば試作機を作成することが目的であったが、予算が許せば SoC FPGA を用いてより低遅延な通信を実現できる。

具体的には、ギガビットイーサネットコントローラのハードマクロを利用する代わりに、PL 部のイーサネット用 IP を利用し、Ethernet フレームの受信時にプロセッサを介在させずにハードウェア処理で完結させてしまえば遅延は小さくなると考えられる。また、1 ギガビット通信では 256 バイトのフレームの送信及び受信にそれぞれ最低 2μs を要するが、10 ギガビットで通信を行えば最低 200ns になり、更に遅延は小さくなると考えられる。

また、このような数十 μs 以下の遅延の測定には、通信相手側のソフトウェアで計測する場合、相手側で発生する遅延が支配的になってしまうため、Ethernet ケーブルや回路基盤上を流れる電気信号をロジックアナライザや LAN アナライザといった計測用機材で測定することも必要である。

6.2 ネットワーク処理一般の高速化

金融商品取引に限らず、一般の分散処理システムにおいても、各ノード内の演算性能以上にネットワーク性能がボトルネックとなるケースが散見される。OS を経由せず直接ネットワーク越しのノードのメモリにアクセスする既存の技術として、RDMA（Remote Direct Memory Access) が挙げられるが、そのような高速通信が利用可能なのは大規模クラウドサービスや HPC 等の一部の環境に限られ、いわゆるエッジ側デバイスでは OS のプロトコルスタックを経由してアプリケーションプログラムが通信データを処理する従来手法が主流である。

そこで、ネットワークを介した処理のレスポンスタイムを改善するために FPGA を用いてアプリケーション層の処理の一部をハードウェア化する本研究のアプローチは、小規模データセンターや IoT システムのエッジサーバ、組み込み機器における通信の高速化の目的でも適用しうると考える。

6.3 まとめ

本研究では金融商品の低遅延取引の実現をテーマに、ネットワーク経由で発生するデータをリアルタイムでストリーミング処理するためのハードウェアアクセラレータを設計した。SoC FPGA を利用し、OS の TCP/IP プロトコルスタックや FIX エンジンといったソフトウェア資産を利用しつつ、レイテンシにセンシティブな処理のみを FPGA 上のデジタル回路に配置した。具体的には、逆指値という最もシンプルな取引アルゴリズムを実装し、FPGA 内部で生じる遅延が最大でも 780ns という結果が得られた。サーバ側で計測されるトータルの遅延に関しても、同じ FPGA ボード上のソフトウェア処理と比べて 90%、Linux PC のソフトウェア処理と比べても 80%の遅延削減が確認できた。

参考文献

[1] 高性能半導体「データフロープロセッサー (DFP)」. https://www.denso.com/jp/ja/news/media-center/press-kits/tms2017/TMS_DFP_JPN.pdf (2017 年 12 月 27 日閲覧)

[2] 祝迫得夫. 日本における高頻度取引（High Frequency Trading）の現状について. 第 1 期 JSDA キャピタルマーケットフォーラム研究論文. p27-40. 2017.

[3] Australian Securities & Investments Commision. Review of high-frequency trading and dark liquidity. 2015.

[4] Gareth W. Morris, David B. Thomas and Wayne Luk. FPGA accelerated low-latency market data feed processing. In 17th IEEE Symposium on High Performance Interconnects 2009.

[5] Christian Leber, Benjamin Geib, Heiner Litz. High Frequency Trading Acceleration using FPGAs. In FPL 2011.

[6] Robin Pottathuparambil et al. Low-latency FPGA Based Financial Data Feed Handler. IEEE International Symposium on Field-Programmable Custome Computing Machines. 2011.

[7] John W. Lockwood et al. A Low-Latency Library in FPGA Hardware for High-Frequency Trading(HFT) In 2012 IEEE 20th Annual Symposium on High-Performance Interconnects

[8] Finteligent Trading Technology Community (FTTC), Research Report: 10GbE Low Latency Networking Technology Review. 2012.

[9] ARGON DESIGN, ARISTA, finteligent. Research Report: The Arista 7124FX Switch as a High Performance Trade Execution Platform. 2013.

[10] 井上浩明. 特集, 金融市場における最新情報技術：FPGA による金融業務アクセラレーション-複合イベント処理を題材に-. 情報処理. 2012, vol. 53, no. 9, p. 921-926.

[11] 天野英晴編 (2016) 『FPGA の原理と構成』 オーム社.

[12] 水田孝信. 特集, 金融市場における最新情報技術：金融の役割と情報化の進展. 情報処理 2012, vol. 53, no. 9, p. 890-897.

[13] 中山慎一郎、長山昌平、鳥海不二夫．特集，金融市場における最新情報技術：システムトレードによる自動取引．情報処理．2012, vol. 53, no. 9, p. 898-903.

[14] 尹照元、松井宏樹．特集，金融市場における最新情報技術：アルゴリズム・トレードの現状と今後の展開．情報処理．2012, vol. 53, no. 9, p. 904-909.

[15] 小林賢一、百石弘澄．特集，金融市場における最新情報技術 : 株式売買システム”arrowhead”を取り巻く市場環境の変化について．情報処理．2012, vol. 53, no. 9, p. 910-914.

[16] Barry Johnson. Algorithmic Trading & DMA. 4 Myeloma Press. 2010

[17] 杉原慶彦．取引コストの削減を巡る市場参加者の取組み：アルゴリズム取引と代替市場の活用．　金融研究. 2011.4, p29-87

[18] 庄子裕明（2008）『FIX 入門』　メタビット出版サービス.

[19] 石田修、瀬戸康一郎 (2005) 『改訂版 10 ギガビット Ethernet 教科書』　インプレス R&D.

[20] Charles E. Spurgeon, Joann Zimmerman(2016)『詳説イーサネット第 2 版』　O'Reilly

[21] インターフェース編集部 (2006) 『Ethernet のしくみとハードウェア設計技法—プロトコルの詳細からネットワーク対応機器の作成まで 』　CQ 出版社.

[22] みやたひろし (2017)『パケットキャプチャの教科書』　SB クリエイティブ

[23] 高橋浩和、小田逸郎. 山幡為佐久 (2006)　『Linux カーネル解読室』　ソフトバンククリエイティブ.

[24] 平田豊 (2011) 『Linux カーネル解析入門〔増補版〕』　工学社.

[25] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman(2005) 『第 3 版 LINUX デバイスドライバ』　O'Reilly.

[26] 宗像 尚郎/海老原 祐太郎 (2016) 『動くメカニズムを図解&実験! Linux 超入門 』　CQ 出版社.

[27] FPGA マガジン編集部 (2013) 『FPGA マガジン No.3 高速 Ethernet × FPGA』　CQ 出版社.

[28] FPGA マガジン編集部 (2016) 『FPGA マガジン No.12 ARM コア FPGA × Linux 初体験』　CQ 出版社.

[29] 岩田利王、横溝憲治 (2016)『FPGA パソコン ZYBO で作る LinuxI/O ミニコンピュータ』　CQ 出版社.

[30] 鈴木量三朗、片岡啓明 (2014)『ARM Cortex-A9 × 2! Zynq でワンチップ Linux on FPGA』　CQ 出版社.

[31] 小林優 (2016)『FPGA プログラミング大全 Xilinx 編』　秀和システム.

[32] 宇野俊夫 (2006)『組込み I/O インタフェース基礎講座』　翔泳社.

[33] Louise H. Crockett et al. The Zynq Book. Strathclyde Academic Media. 2014

[34] http://www.fixtradingcommunity.org/ FIX Trading Community. (2017 年 12 月 27 日閲覧)

[35] http://www.quickfixengine.org/ QuickFIX (2017 年 12 月 27 日閲覧)

[36] http://www.pynq.io/ PYNQ:PYTHON PRODUCTIVITY FOR ZYNQ (2017 年 12 月 27 日閲覧)

[37] Xilinx Python productivity for Zynq(Pynq) Documentation Release 1.01. 2017. (2017 年 12 月 27 日閲覧)

[38] ARM. AMBA AXI and ACE Protocol Specification. ARM IHI 0022D(ID102711). 2003.

[39] ARM. AMBA Specification (Rev 2.0). ARM IHI 0011A. 1999.

[40] Xilinx. Zynq-7000 All Programmable SoC Technical Reference Manual. UG585(v1.12.1). 2017

[41] Xilinx. Vivado Design Suite AXI Reference Guide. UG1037 (v4.0). 2017.

[42] Xilinx. Block Memory Generator v8.3. PG058. 2017.

[43] Xilinx. AXI Block RAM(BRAM) Controller v4.0. 2015.

[44] Andrew Putnam et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on.

[45] 低レイテンシー・クラウド・ネットワークの設計 https://www.arista.com/assets/data/pdf/CloudNetworkLatency_jp.pdf (2018 年 1 月 23 日閲覧)

[46] 高レスポンスやビッグデータ処理が要求される新たなアプリケーションの開拓を推進する「エッジコンピューティング構想」を策定 http://www.ntt.co.jp/news2014/1401/140123a.html (2018 年 1 月 23 日閲覧)

謝辞

本研究を進めるにあたり、デジタル回路設計の基本からご指導を賜り、また進捗がなく自分自身研究の継続を諦めかけた時にも見放さずに、貴重な時間を割いて研究構想の実現にご助力いただきました田中清史准教授に心から感謝いたします。また、副テーマのご指導を頂きました白井清昭准教授、課題研究計画発表審査で研究の方向性をご指導いただきました金子峰雄教授、井口寧教授、実務家の視点からの貴重なアドバイスを頂きました田中研究室東京サテライトの皆様に感謝いたします。

付録

リスト 6.1: レジスタ更新規則 1

```
always @(posedge M_AXI_ACLK) // 1
  begin
    // Initiates AXI transaction delay
    if (M_AXI_ARESETN == 0 )
      begin
        init_txn_ff <= 1'b0;
        init_txn_ff2 <= 1'b0;
      end
    else
      begin
        init_txn_ff <= INIT_AXI_TXN;
        init_txn_ff2 <= init_txn_ff;
      end
  end
```

リスト 6.2: レジスタ更新規則 2

```
  always @(posedge M_AXI_ACLK)   //2
  begin

    if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 )
      begin
        axi_awvalid <= 1'b0;
      end
    // If previously not valid , start next transaction
    else if (~axi_awvalid && start_single_burst_write)
      begin
        axi_awvalid <= 1'b1;
      end
    /* Once asserted, VALIDs cannot be deasserted, so axi_awvalid
    must wait until transaction is accepted */
    else if (M_AXI_AWREADY && axi_awvalid)
      begin
        axi_awvalid <= 1'b0;
      end
    else
      axi_awvalid <= axi_awvalid;
    end
```

リスト 6.3: レジスタ更新規則 3

```
  always @(posedge M_AXI_ACLK) // 3
  begin
    if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 )
      begin
        axi_wvalid <= 1'b0;
      end
    // If previously not valid, start next transaction
    else if (~axi_wvalid && start_single_burst_write)
```

```
        begin
          axi_wvalid <= 1'b1;
        end
      /* If WREADY and too many writes, throttle WVALID
      Once asserted, VALIDs cannot be deasserted, so WVALID
      must wait until burst is complete with WLAST */
      else if (wnext && axi_wlast)
        axi_wvalid <= 1'b0;
      else
        axi_wvalid <= axi_wvalid;
    end
```

リスト 6.4: レジスタ更新規則 4

```
    always @(posedge M_AXI_ACLK) // 4
    begin
      if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 || state_changing)
        begin
          axi_wlast <= 1'b0;
        end
      // axi_wlast is asserted when the write index
      // count reaches the penultimate count to synchronize
      // with the last write data when write_index is b1111
      // else if (&(write_index[C_TRANSACTIONS_NUM-1:1])&& ~write_index[0] && wnext)
      else if ((counter_w == 3 - 2 ) && wnext) //バースト長3:
        begin
          axi_wlast <= 1'b1;
        end
      // Deassrt axi_wlast when the last write data has been
      // accepted by the slave with a valid response
      else if (wnext)
        axi_wlast <= 1'b0;
      else
        axi_wlast <= axi_wlast;
    end
```

リスト 6.5: レジスタ更新規則 5

```
        always @(posedge M_AXI_ACLK) begin // 5
            if (state == WRITE_AXI) begin
                if(state_changing) begin
                    for (i=0;i<128;i=i+1) begin
                        axi_wdata[i*8+7 -: 8] <= messages[i];
                    end
                end
                else if (wnext) begin
                    case (counter_w)
                        0: begin
                            for (i=0;i<128;i=i+1) begin
                               axi_wdata[i*8+7 -: 8] <= messages[i+128];
                           end
                        end
                        1:   begin
                        //axi_wdata <= writer3;
                            axi_wdata[31 : 0] <= 32'h00000000; //ドライバ(FF→ハード)
                            axi_wdata[63 : 32] <= send_ok ? 32'hffffffff : 32'haaaaaaaa;
                            axi_wdata[95 : 64] <= 32'h00000000; //アプリ→ハード
                            axi_wdata[103 : 96] <= segment_len_out[7:0];
                            axi_wdata[111 : 104] <= {7'b0000000, softd_set};
                            axi_wdata[119 : 112] <= counter_send;
                            axi_wdata[127 : 120] <= counter_fix[15:8];
                            axi_wdata[135 : 128] <= counter_fix[7:0];
                            axi_wdata[143 : 136] <= counter_packet[15:8];
                            axi_wdata[151 : 144] <= counter_packet[7:0];
```

```
                            axi_wdata[159 : 152] <= counter_read[23:16];
                            axi_wdata[167 : 160] <= counter_read[15:8];
                            axi_wdata[175 : 168] <= counter_read[7:0];
                            axi_wdata[255 : 176] <= fix_judge;
                            axi_wdata[1023 : 256] <= 0;
                        end
                        default:
                            axi_wdata <= 1024'b0;
                    endcase
                end
            end
            else
                axi_wdata <= 1024'b0;
        end
```

リスト 6.6: レジスタ更新規則 6

```
    always @(posedge M_AXI_ACLK) // 6
    begin
      if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 )
        begin
          axi_bready <= 1'b0;
        end
      // accept/acknowledge bresp with axi_bready by the master
      // when M_AXI_BVALID is asserted by slave
      else if (M_AXI_BVALID && ~axi_bready)
        begin
          axi_bready <= 1'b1;
        end
      // deassert after one clock cycle
      else if (axi_bready)
        begin
          axi_bready <= 1'b0;
        end
      // retain the previous value
      else
        axi_bready <= axi_bready;
    end
```

リスト 6.7: レジスタ更新規則 7

```
    always @(posedge M_AXI_ACLK) // 7
    begin

      if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 )
        begin
          axi_arvalid <= 1'b0;
        end
      // If previously not valid , start next transaction
      else if (~axi_arvalid && start_single_burst_read)
        begin
          axi_arvalid <= 1'b1;
        end
      else if (M_AXI_ARREADY && axi_arvalid)
        begin
          axi_arvalid <= 1'b0;
        end
      else
        axi_arvalid <= axi_arvalid;
    end
```

リスト 6.8: レジスタ更新規則 8

```
    always @(posedge M_AXI_ACLK) // 8
    begin
      if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 )
        begin
          axi_rready <= 1'b0;
        end
      // accept/acknowledge rdata/rresp with axi_rready by the master
      // when M_AXI_RVALID is asserted by slave
      else if (M_AXI_RVALID)
        begin
           if (M_AXI_RLAST && axi_rready)
            begin
              axi_rready <= 1'b0;
            end
           else
             begin
               axi_rready <= 1'b1;
             end
        end
      // retain the previous value
    end
```

リスト 6.9: レジスタ更新規則 9

```
    always @(posedge M_AXI_ACLK) // 9
    begin
      if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
        burst_write_active <= 1'b0;

      //The burst_write_active is asserted when a write burst transaction is initiated
      else if (start_single_burst_write)
        burst_write_active <= 1'b1;
      else if (M_AXI_BVALID && axi_bready)
        burst_write_active <= 0;
    end
```

リスト 6.10: レジスタ更新規則 10

```
    always @(posedge M_AXI_ACLK) // 10
    begin
      if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
        burst_read_active <= 1'b0;

      //The burst_write_active is asserted when a write burst transaction is initiated
      else if (start_single_burst_read)
        burst_read_active <= 1'b1;
      else if (M_AXI_RVALID && axi_rready && M_AXI_RLAST)
        burst_read_active <= 0;
      end
```

リスト 6.11: レジスタ更新規則 11

```
    always @ (posedge M_AXI_ACLK) begin // 11
        if (M_AXI_ARESETN == 1'b0) begin
            state <= READ_AXI;
            state_bfr <= READ_AXI;
          start_single_burst_write <= 1'b0;
            start_single_burst_read  <= 1'b0;
            refresh <= 1'b0;
            refresh_all <= 1'b0;
            softd_set <= 1'b0;
            counter_fix <= 16'h0000;
```

```
            counter_packet <= 16'h0000;
            counter_send <= 8'h00;
            counter_read <= 24'h000000;
        end
        else begin
            if (state == READ_AXI) begin
                if (read_done && ~refresh_all && ~refresh) begin
                    if (softd_updated)
                        softd_set <= 1'b1;
                    if (packet_received)
                        counter_packet <= counter_packet + 1;

                    if (packet_fix) begin
                        state <= JUDGE;
                        counter_fix <= counter_fix + 1;
                    end
                    else if (softd_updated || packet_received)
                        state <= WRITE_AXI;
                    else begin
                        refresh <= 1'b1;
                        counter_read <= counter_read + 1;
                    end
                end
                else if(~axi_arvalid && ~burst_read_active && ~start_single_burst_read)
                    start_single_burst_read <= 1'b1;
                else
                    start_single_burst_read <= 1'b0;
            end
            else if (state == JUDGE) begin
                if (judged)
                    state <= WRITE_AXI;
            end
            else if (state == WRITE_AXI) begin
                if (~writing && written) begin
                    if(send_ok) begin
                        softd_set <= 1'b0;
                        counter_send <= counter_send + 1;
                        refresh_all <= 1'b1;
                    end
                    else
                        refresh <= 1'b1;
                    state <= READ_AXI;
                end
                else if (~axi_awvalid && ~start_single_burst_write && ~burst_write_active)
                    start_single_burst_write <= 1'b1;
                else
                    start_single_burst_write <= 1'b0; //Negate to generate a pulse
            end
            if (refresh == 1'b1)
                refresh <= 1'b0;
            if (refresh_all == 1'b1)
                refresh_all <= 1'b0;
            state_bfr <= state;
        end
    end
```

リスト 6.12: レジスタ更新規則 12

```
    always @ (posedge M_AXI_ACLK) begin // 12
        if (M_AXI_ARESETN == 1'b0 || init_txn_pulse ==1'b1 || refresh_all) begin
            softd_number <= 8'hFF; // 0xFF:reset  0x01-0xFE, when the number changes, updated
            cur_no <= 4'hF;
            order_id <= 32'h00000000;
            symbol_target <= 32'h00000000;
            side_b <= 1'b0;
```

```
8               price_trigger <= 32'h00000000;
9               msg_len <= 8'h00;
10              for (i=0;i<202;i=i+1) fix[i] <= 8'h00;
11              for(i=0;i<64;i=i+1) fix_s1[i] <= 0;
12              for(i=0;i<8;i=i+1) fix_s2[i] <= 0;
13              err_s <=1'b0;
14
15              counter_r <= 8'h00;
16              read_done <= 1'b0;
17              packet_received <= 1'b0;
18              softd_updated <= 1'b0;
19
20              packet_fix <= 1'b0;
21              mac_from <= 48'h000000000000; mac_to <= 48'h000000000000;
22              packet_len <= 16'h0000;
23              ip_from <= 32'h00000000; ip_to <= 32'h00000000;
24              port_from <= 16'h0000; port_to <= 16'h0000;
25              sequence_no <= 32'h00000000; msgno_to <= 32'h00000000;
26              data_offset <= 4'h0;
27              for (i=0;i<256;i=i+1) fix_in[i] <= 8'h00;
28          end
29          else if (refresh) begin
30              counter_r <= 8'h00;
31              read_done <= 1'b0;
32              packet_received <= 1'b0;
33              softd_updated <= 1'b0;
34              fix_judge <= 0;
35              packet_fix <= 1'b0;
36              mac_from <= 48'h000000000000; mac_to <= 48'h000000000000;
37              packet_len <= 16'h0000;
38              ip_from <= 32'h00000000; ip_to <= 32'h00000000;
39              port_from <= 16'h0000; port_to <= 16'h0000;
40              sequence_no <= 32'h00000000; msgno_to <= 32'h00000000;
41              data_offset <= 4'h0;
42              for (i=0;i<256;i=i+1) fix_in[i] <= 8'h00;
43           end
44           else if (state == READ_AXI && rnext) begin
45               case (counter_r)
46                   0:  begin
47                       if (M_AXI_RDATA[7:0] == 8'hff)
48                           packet_received <= 1'b1;
49                       if (M_AXI_RDATA[71:64] != 8'h00 && M_AXI_RDATA[71:64] != softd_number)
50                           softd_updated <= 1'b1;
51                       softd_number <= M_AXI_RDATA[71:64];
52                   end
53                   1:  if (softd_updated) begin
54                           cur_no <= reader[1023-8 -: 4]; // M_AXI_RDATA[15:12];
55                           order_id <= reader[1023-2*8 -: 32]; //M_AXI_RDATA[47:16];
56                           symbol_target <= reader[1023-6*8 -: 32];
57                           if (reader[1023-10*8 -: 8] == 8'h31)
58                               side_b <= 1'b1;
59                           else if (reader[1023-10*8 -: 8] == 8'h32)
60                               side_b <= 1'b0;
61                           else
62                               err_s <= 1'b1;
63                           price_trigger <=reader[1023-11*8 -: 32];
64                           msg_len <= reader[1023-28*8-4 -: 4] * 100
65                                           + reader[1023-29*8-4 -: 4] * 10
66                                           + reader[1023-30*8-4 -: 4];
67                           for(i=0; i<128-16; i=i+1) begin
68                               fix[i] <= reader[1023-(16+i)*8 -: 8];
69                           end
70                       end
71                   2:  if (softd_updated) begin
72                           for(i=0; i<90; i=i+1) begin
73                               fix[i+112] <= reader[1023-8*i -: 8];
```

```
                    end
                end
            3:  begin
                if (packet_received) begin
                    mac_from <= reader[1023 -: 48];
                    mac_to <= reader[1023-6*8 -: 48];
                    packet_len <= reader[1023-16*8 -: 16];
                    ip_from <= reader[1023-26*8 -: 32];
                    ip_to <= reader[1023-30*8 -: 32];
                    port_from <= reader[1023-34*8 -: 16];
                    port_to <= reader[1023-36*8 -: 16];
                    sequence_no <= reader[1023-38*8 -: 32];
                    msgno_to <= reader[1023-42*8 -: 32];
                    data_offset <= reader[1023-46*8 -: 4];
                    for (i=0;i<128-54;i=i+1) begin
                        fix_in[i] <= reader[1023-(54+i)*8 -: 8];
                    end

                    if( reader[1023-54*8 -:80] == 80'h383d4649582e342e3401) begin
                        packet_fix <= 1'b1;
                    end
                    fix_judge <= reader[1023-54*8 -: 80];
                end
                if (softd_updated) begin
                    for(i=0;i<50;i=i+1) begin
                        fix_s1[i] <= {fix[i*4+0], fix[i*4+1]} + {fix[i*4+2], fix[i*4+3]};
                    end
                    fix_s1[50] <= {fix[200],fix[201]};
                end
            end
            4:  begin
                if (packet_received) begin
                    for (i=0;i<128;i=i+1) begin
                        fix_in[i+74] <= reader[1023-i*8 -: 8];
                    end
                end
                if (softd_updated) begin
                    for(i=0;i<8;i=i+1) begin
                        fix_s2[i] <= sum(fix_s1[i*8+0], fix_s1[i*8+1], fix_s1[i*8+2],
                                         fix_s1[i*8+3], fix_s1[i*8+4], fix_s1[i*8+5],
                                         fix_s1[i*8+6], fix_s1[i*8+7]);
                    end
                end
                read_done <= 1'b1;
            end
            default:
                ;
        endcase
        counter_r <= counter_r + 1;
    end
end
```

リスト 6.13: レジスタ更新規則 13

```
always @ (posedge M_AXI_ACLK) begin // 13
    if(M_AXI_ARESETN == 1'b0 || init_txn_pulse ==1'b1 || refresh == 1'b1) begin
        judged <= 1'b0;
        counter_j <= 1'b0;
        send_ok <= 1'b0;
        fix_symbol <= 32'h00000000;
        fix_price <= 64'h0000000000000000;
        err_j <= 1'b0;
       fix_sum <= 32'h00000000;
       onehot_s <= 32'h00000000;
       onehot_p <= 48'h000000000000;
```

```
12              index_s <= 5'h0;
13              index_p <= 6'h0;
14              checksum_ip <= 16'h0000; checksum_tcp <= 16'h0000;
15
16               for (i=0;i<256;i=i+1) begin
17                   messages[i] <= 8'h00;
18               end
19          end
20          else if (state == JUDGE) begin
21              case (counter_j)
22              0: begin
23                  //価格情報
24                  if (fix_in[19] != 8'h57 && fix_in[18] != 8'h57)
25                      judged <= 1'b1;
26                  else begin
27                      for(i=0;i<32;i=i+1) begin
28                          if(fix_in[i+60]==8'h01 && fix_in[i+61] == 8'h35
29                            && fix_in[i+62] == 8'h35 && fix_in[i+63] == 8'h3D)
30                              onehot_s[i] <= 1'b1;
31                      end
32                      for(i=0;i<64;i=i+1) begin
33                          if(fix_in[i+119] == 8'h32 && fix_in[i+120] == 8'h01
34                             && fix_in[i+121] == 8'h32 && fix_in[i+122] == 8'h37
35                             && fix_in[i+123] == 8'h30 && fix_in[i+124] == 8'h3D)  //   2.270=
36                              onehot_p[i] <= 1'b1;
37                      end
38                  end
39                  fix_sum <= sum(fix_s2[0], fix_s2[1], fix_s2[2], fix_s2[3],
40                                  fix_s2[4], fix_s2[5], fix_s2[6], fix_s2[7]);
41              end
42              1: begin //ここでindex_s,が個生成されてしまうのでできるだけ減らすindex_pn
43                 for(i=0;i<32;i=i+1) begin
44                      if(onehot_s[i] == 1'b1) index_s <= i;
45                  end
46                  for(i=0;i<64;i=i+1) begin
47                      if(onehot_p[i] == 1'b1) index_p <= i;
48                  end
49              end
50              2: begin //ここでも取りうるの数だけ回路ができてしまうのでなるだけ減らすfix_in
51                  for(i=0;i<4;i=i+1)
52                      fix_symbol[31-8*i -: 8] <= fix_in[index_s+i+64];
53                  for(i=0;i<8;i=i+1)
54                      fix_price[63-8*i -: 8] <= fix_in[index_p+i+125];
55
56                  checksum_ip <= ~(ip_sum[15:0]+ip_sum[31:16]);
57                  checksum_tcp <= ~(tcp_sum[15:0]+tcp_sum[31:16]);
58
59              end
60              3: begin
61                  if(fix_symbol == symbol_target && price != 32'hffffffff
62                      && price != 32'h00000000) begin
63                      if ((price>=price_trigger && side_b) || (price<=price_trigger && !side_b))
64                          send_ok <= 1'b1;
65                  end
66                  else
67                      err_j <= 1'b1;
68
69                   for (i = 0; i < 54; i = i + 1) begin
70                     messages[i] <= header[431- 8*i -: 8];
71                  end
72                  for (i = 0; i < 202; i= i + 1) begin
73                     messages[i + 54] <= fix[i];
74                  end
75
76                  judged <= 1'b1;
77              end
```

```
78              default:
79                  ;
80              endcase
81              counter_j <= counter_j + 1;
82          end
83      end
```

リスト 6.14: レジスタ更新規則 14

```
1       always @ (posedge M_AXI_ACLK) begin // 14
2           if(M_AXI_ARESETN == 1'b0 || init_txn_pulse ==1'b1 || refresh == 1'b1) begin
3               counter_w <= 8'h00;
4               written <= 1'b0;
5           end
6           else if (wnext && state == WRITE_AXI) begin
7               if (counter_w == 2)
8                   written <= 1'b1;
9               counter_w <= counter_w + 1;
10          end
11      end
```

リスト 6.15: 配線 assign

```
1       assign EXEC_STATE = SWITCH_LED[1] ? (SWITCH_LED[0] ? counter_read[23:20]
2                ? {softd_set, counter_send[0], counter_fix[0], counter_packet[0]}) : 4'b0000;
3       assign ERROR = err_s || err_j;
4
5       assign init_txn_pulse = (!init_txn_ff2) && init_txn_ff;
6       assign wnext = M_AXI_WREADY & axi_wvalid;
7       assign rnext = M_AXI_RVALID && axi_rready;
8
9       assign segment_len_in = packet_len - 40;
10      assign segment_len_out = msg_len + 23;
11
12      assign header_frame[15:0]=16'h0800;
13      assign header_ip[159:144]=16'h4500;
14      assign header_ip[127:112]=16'h566a; //(identification No.)
15      assign header_ip[111:96]=16'h4000;
16      assign header_ip[95:80]=16'h4006; //8086 TTL
17      assign header_ip[79:64] = 16'h0000; //Dummy
18      assign header_tcp[63:48]=16'h5018; //No options
19      assign header_tcp[47:32]=16'h022d; //(window size)
20      assign header_tcp[31:16] = 16'h0000; //Dummy
21      assign header_tcp[15:0]=16'h0000;
22
23      assign header_frame[111:64] = mac_to;
24      assign header_frame[63:16] = mac_from;
25      assign header_ip[63:32] = ip_to;
26      assign header_ip[31:0]=ip_from;
27      assign header_tcp[159:144] = port_to;
28      assign header_tcp[143:128]=port_from;
29      assign header_tcp[127:96] = msgno_to;
30
31      assign header_ip[143:128]=msg_len+63;
32      assign header_tcp[95:64]=sequence_no+ segment_len_in; // add bytes of Fix payload
33
34      assign header_ip_c[159:80]=header_ip[159:80];
35      assign header_ip_c[79:64]=checksum_ip;
36      assign header_ip_c[63:0]=header_ip[63:0];
37      assign header_tcp_c[159:32]=header_tcp[159:32];
38      assign header_tcp_c[31:16]=checksum_tcp;
39      assign header_tcp_c[15:0]=header_tcp[15:0];
40
41      assign header[431:320]=header_frame;
```

```
42      assign header[319:160]=header_ip_c;
43      assign header[159:0]=header_tcp_c;
44
45      assign ip_sum = header_ip[159:144] + header_ip[143:128] + header_ip[127:112]
46          + header_ip[111:96] + header_ip[95:80] + header_ip[63:48]
47          + header_ip[47:32] + header_ip[31:16] + header_ip[15:0];
48
49      assign tcp_sum = (header_ip[63:48] + header_ip[47:32] + header_ip[31:16]
50          + header_ip[15:0] + 16'h0006 + (msg_len+43) + header_tcp[159:144]
51          + header_tcp[143:128] + header_tcp[127:112] + header_tcp[111:96] + header_tcp[95:80]
52          + header_tcp[79:64] + header_tcp[63:48] + header_tcp[47:32] + header_tcp[15:0])
53          + fix_sum;
54
55      genvar k;
56      generate
57          for (k=0;k<128;k=k+1) begin: Reader
58              assign reader[k*8+7 : k*8] = M_AXI_RDATA[1023-k*8 : 1016-k*8];
59          end
60      endgenerate
61
62      assign state_changing = (state != state_bfr);
63      assign reading = burst_read_active || start_single_burst_read;
64      assign writing = burst_write_active || start_single_burst_write;
65
66      assign price = mkprice(fix_price);
```

リスト 6.16: function 文

```
1       function[31:0] mkprice(input[63:0] fix_price);
2       begin
3           if (fix_price[55:48] == 8'h01)
4               mkprice = {28'h0000000, fix_price[59:56]};
5           else if (fix_price[47:40] == 8'h01)
6               mkprice  = {24'h000000, fix_price[59:56], fix_price[51:48]};
7           else if (fix_price[39:32] == 8'h01)
8               mkprice = {20'h00000, fix_price[59:56], fix_price[51:48], fix_price[43:40]};
9           else if (fix_price[31:24] == 8'h01)
10              mkprice = {16'h0000, fix_price[59:56], fix_price[51:48], fix_price[43:40],
11                fix_price[35:32]};
12          else if (fix_price[23:16] == 8'h01)
13              mkprice = {12'h000, fix_price[59:56], fix_price[51:48], fix_price[43:40],
14                fix_price[35:32],fix_price[27:24]};
15          else if (fix_price[15:8] == 8'h01)
16              mkprice = {8'h00, fix_price[59:56], fix_price[51:48], fix_price[43:40],
17                fix_price[35:32],fix_price[27:24],fix_price[19:16]};
18          else if (fix_price[7:0] == 8'h01)
19              mkprice = {4'h0, fix_price[59:56], fix_price[51:48], fix_price[43:40],
20                fix_price[35:32],fix_price[27:24],fix_price[19:16],fix_price[11:8]};
21          else
22              mkprice = {fix_price[59:56], fix_price[51:48], fix_price[43:40],fix_price[35:32],
23                fix_price[27:24],fix_price[19:16],fix_price[11:8],fix_price[3:0]};
24      end
25      endfunction
26
27      function[23:0] sum(input[23:0] in0, in1, in2, in3, in4, in5, in6, in7);
28      begin
29          sum=((in0+in1)+(in2+in3))+((in4+in5)+(in6+in7));
30      end
31      endfunction
```