修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報システム学研究科ネットワークシステム学専攻　博士前期課程		
氏　　　名	佐藤　遼	学籍番号	1252018
論 文 題 目	カスタムプロセッサ構築用 FPGA プラットフォームの開発と評価		

要　　旨

プロセッサの性能向上は今なお求められており，更なる処理性能向上にはプロセッサアーキテクチャの改善が必要である．プロセッサアーキテクチャの研究ではアイディアの検証はソフトウェアシミュレーションによって行われることが多い．しかしながら，回路規模の増大や処理の複雑化によって評価にかかる時間が増大するという問題が顕在化している．

そこで注目すべき解決策が，Field Programmable Gate Array(FPGA）を用いたエミュレーション方法である．FPGA はハードウェアチップを設計するより，簡単に所望の回路を実現することができ，かつハードウェアの動作をソフトウェアよりも高速に模倣させることができる．また，何度でも内容を変更・修正できるため，動作確認のテストを容易に行うことができる．このため FPGA を用いたアーキテクチャ研究の高速化は有用な手段の一つと考えられる．

そこで本論文では ARM の ISA をベースとするカスタムプロセッサ構築用 FPGA プラットフォームを独自に実現することを最終的な目的として，プロセッサを独自に実装し，動作検証と評価を行う．

本稿ではカスタムプロセッサ構築用FPGA プラットフォームの実現を最終的な目的としてFPGAプラットフォームの実装と評価を行った．第一に，設計したプロセッサが想定通りに実装できていることを検証した．第二に，様々なプログラムに対してプロセッサが動作可能であることを確認した．第三に，シリアル通信でプロセッサのメモリにアクセスする機構を追加実装した．第四に，PCIe 通信でプロセッサの性能評価指数を出力する機構を追加実装した．最後に，シミュレータとFPGAで，プロセッサの検証にかかる実行時間を比較した．

これらの検証結果からFPGA による拡張性，高速動作性を確認し，カスタムプロセッサ構築用FPGAプラットフォームとして有用であることを確認した．加えて今後の研究の方向性として提案システムの改善点をまとめた．

平成２５年度修士論文

カスタムプロセッサ構築用
FPGA プラットフォームの開発と評価

大学院情報システム学研究科
情報ネットワークシステム学専攻

学 籍 番 号 :　1252018
氏 名:　佐藤 遼
主任指導教員: 吉永 努 教授
指 導 教 員 :　入江 英嗣 准教授
指 導 教 員 :　大坐畠 智 准教授
提出年月日 :　平成２６年１月２７日

(表紙裏)

目次

第 1 章　序論 **1**

第 2 章　関連研究 **3**

2.1 異種命令混合実行プロセッサ OROCHI 3
2.2 CoreSymphony 3
2.3 関連製品 4
2.4 その他の関連研究 5

第 3 章　提案システムの概要と実装手順 **6**

3.1 概要 6
3.2 実装手順 7
3.2.1 事前学習と準備 7
3.2.2 カスタムプロセッサの実装と検証 8
3.2.3 シリアル通信によるメモリアクセスの実装と検証 9
3.2.4 PCIe 通信による性能評価指数出力の実装と検証 10

第 4 章　実装したプラットフォームの検証と評価 **11**

4.1 様々なプログラムに対するカスタムプロセッサの動作検証 11
4.2 シリアル通信のメモリアクセス検証 27
4.3 PCIe 通信の性能評価指数出力検証 31
4.4 性能評価 34

第 5 章　結論 **35**

5.1 本研究のまとめ 35
5.2 今後の課題 36

参考文献 **38**

付録 **40**

図目次

3.1.1 提案システムの構成図 6
3.2.1 カスタムプロセッサブロック図 8
3.2.2 シリアル通信ブロック図/Buf 有り 9
3.2.3 シリアル通信ブロック図/Buf 無し 9
3.2.4 PCIe 通信ブロック図 10

4.1.1 Hillo シミュレーション結果 11
4.1.2 Hillo 全体波形図 12
4.1.3 Hillo 拡大波形図 12
4.1.4 Gusu シミュレーション結果 13
4.1.5 Gusu 全体波形図 14
4.1.6 Gusu 拡大波形図 14
4.1.7 Foradd シミュレーション結果 15
4.1.8 Foradd 全体波形図 16
4.1.9 Foradd 拡大波形図 16
4.1.10 Fibonacci シミュレーション結果 17
4.1.11 Fibonacci 全体波形図 18
4.1.12 Fibonacci 拡大波形図 18
4.1.13 FizzBuzz シミュレーション結果 19
4.1.14 FizzBuzz 全体波形図 20
4.1.15 FizzBuzz 拡大波形図 20
4.1.16 Sort シミュレーション結果 21
4.1.17 Sort 全体波形図 22
4.1.18 Sort 拡大波形図 22
4.1.19 Sosu シミュレーション結果 23
4.1.20 Sosu 全体波形図 24
4.1.21 Sosu 拡大波形図 24
4.1.22 Himeno シミュレーション結果 25
4.1.23 Himeno 全体波形図 26
4.1.24 Himeno 拡大波形図 26
4.2.1 Hillo メモリデータ入出力 27
4.2.2 Gusu メモリデータ入出力 27
4.2.3 Foradd メモリデータ入出力 28
4.2.4 Fibonacci メモリデータ入出力 28
4.2.5 FizzBuzz メモリデータ入出力 29

4.2.6 Sort メモリデータ入出力 29
4.2.7 Sosu メモリデータ入出力 30
4.2.8 Himeno メモリデータ入出力 30
4.3.1 PCIe 出力/Hillo 32
4.3.2 PCIe 出力/Gusu 32
4.3.3 PCIe 出力/Foradd 32
4.3.4 PCIe 出力/Fibonacci 32
4.3.5 PCIe 出力/FizzBuzz 33
4.3.6 PCIe 出力/Sort 33
4.3.7 PCIe 出力/Sosu 33
4.3.8 PCIe 出力/Himeno 33

表目次

3.1　シミュレーション環境 7

4.1　PCIe 通信出力詳細 31
4.2　動作周波数と使用リソース量 34
4.3　実行時間比較 34

ソースコード

4.1 Hillo 12
4.2 Gusu 13
4.3 Foradd 15
4.4 Fibonacci 17
4.5 FizzBuzz 19
4.6 Sort 21
4.7 Sosu 23
5.1 Himeno 40

第1章　序論

プロセッサの性能向上は今なお求められている．プロセッサの性能を向上させる方法には周波数の向上，アーキテクチャの改善およびマルチコア化がある．しかし熱や消費電力，配線遅延の相対的な増加によって，周波数の向上は限界に達してきている．またマルチコア化は，複数のタスクを同時に処理する事でシステム全体の処理性能を向上させる手法であり，シングルスレッドの処理性能を向上させるわけではない．そのためプロセッサの更なる処理性能向上にはプロセッサアーキテクチャの改善が必要である．特に，従来研究されてきたプロセッサ単体の計算処理性能を向上させる研究に加え，プリフェッチやキャッシュなどのプロセッサシステム全体を考慮したアーキテクチャの研究が重要になってきている．

プロセッサアーキテクチャの研究ではアイディアを検証し性能を評価する必要がある．時間的および金銭的コストの問題で，ソフトウェアシミュレーションによる動作検証及び性能評価が行われることが多い．しかしながら，プリフェッチやキャッシュの研究ではアプリケーションレベルでの評価が必要になるため，従来のアーキテクチャ単体のシミュレーションに比べて，評価にかかる時間が増大するという問題が顕在化している．

ソフトウェアによるシミュレーションに時間がかかるとはいえ，実際に研究のアイディアを実ハードウェアで作って検証・評価することは，金銭的にも開発コスト的にも現実的ではない．そこで注目すべき解決策が，Field Programmable Gate Array(FPGA) を用いたエミュレーション方法である．FPGA はアプリケーションに合わせた実ハードウェアをユーザが設計できるデバイスであり，実際にハードウェアチップを設計するよりは，簡単に所望の回路を実現することができ，かつハードウェアそのものであることから，ハードウェアの動作をソフトウェアよりも高速に模倣させることができる．また，何度でも内容を変更・修正できるため，動作確認のテストを容易に行うことができる．このため FPGA を用いたアーキテクチャ研究の高速化は有用な手段の一つと考えられる．

アーキテクチャの研究では Instruction Set Architecture(ISA) を決める方がやりやすい．パソコンでは x86，携帯機器では ARM，研究教育用では MIPS がよく用いられる．もちろん，オリジナルの ISA をベースにアーキテクチャの研究を行うという選択も考えられるが，今回は，ARM を対象にしたアーキテクチャの研究を想定する．その理由は，ARM は小型かつ消費電力あたりの処理性能が高いプロセッサであることから，携帯電話をはじめとして最近特に注目を集めているからである．すなわち，ARM の ISA をベースとすることで，新しい研究上のアイディアを実用的な環境で評価しやすいというメリットがあると考えられる．しかし，ARM は IP コアで商売を行っているため，ASIC 開発や FPGA 上に構成可能な RTL 記述が公開されていない．そのため，FPGA を用いて ARM をベースとしたアーキテクチャ研究を行うことができない．プロセッサアーキテクチャの研究で，あるアイディアを検証しようとしても，当然，ベースとなる ISA に沿った振る舞いをするプロセッサ部分のエミュレーションができなければ，その評価はできない．

そのため本研究では ARM の ISA をベースとするカスタムプロセッサ構築用 FPGA プラットフォームを独自に実現することを最終的な目的として，カスタムプロセッサを独自に実装し，動作検証

と評価を行う．

本論文の構成は次の通りである．まず，第二章で関連研究について述べる．次に，第三章では実装の対象であるプロセッサアーキテクチャについて実装までの手順を説明し，第四章で，実装した回路の動作検証の結果を，また，ソフトウェアシミュレータを使ってシミュレーションをするのにかかる時間と FPGA 上でエミュレーションするのにかかる時間を比較し，FPGA による高速化の有効性を示す．最後に本論文の成果をまとめ, 今後の課題を述べる．

第2章　関連研究

プロセッサの性能を向上させる方法のひとつとしてマルチスレッド・マルチコア化がある．関連研究 OROCHI は単一コアで複数命令セットを実行可能なプロセッサを，関連研究 CoreSymphony は複数のコアを協調動作させ，複数のキャッシュや演算器を利用し，逐次処理の高速化を実装している．以下に各関連研究の詳細を示す.

2.1　異種命令混合実行プロセッサ OROCHI

異なる命令セットアーキテクチャのプロセッサを１チップに混載した商用マルチコア型プロセッサがあり，今後複数の命令セットを同時に実行できるプロセッサが徐々に一般化していくと考えられる．しかし複数種類のプロセッサコアを並置する方式を用いると，全体の回路規模が大きくなり，コア間通信のためにある程度複雑な調停機構が必要になる．そのため単一コアで同時実行するマルチスレッディングプロセッサ OROCHI を設計する．ここで OROCHI は奈良先端科学技術大学院大学が開発した異なる命令セットを同時に実行できるよう，一般的な SMT プロセッサを拡張したプロセッサモデルのことを指す，OROCHI は汎用命令セットにより記述されたソフトウェア資産をスーパスカラ方式により高速実行する部分と，マルチメディア処理用の命令セットに特化し，コンパイラ等によりスケジューリングが完了している命令列を VLIW 方式により効率よく実行する部分から構成される．そしてシミュレータの観点からソフトウェアシミュレーションと RTL ソースの記述量とシミュレーション実行速度について分析し，実用性の観点から FPGA と ASIC それぞれを対象とした論合成の結果から遅延時間や回路規模等についての評価している [1][2][3][4][5].

2.2　CoreSymphony

近年，１チップに複数のコアを集積する CMP が周流のアーキテクチャとなっている．CMP はスレッドレベル並列性を利用し，複数のスレッドを複数のコアで並列実行することで性能向上を得る．しかし並列プログラム中の逐次処理をなくすことは難しいため，CMP においても逐次処理の高速化が重要な課題となっている．そこで CoreSymphony という技術が提案されている．CoreSymphony は発行幅の狭いプロセッサコアをいくつか協調動作させることで，より発行幅の大きな仮想コアを形成する手法である．そのため，複数コアのキャッシュや演算器を利用することで，逐次処理を高速実行可能にしている．そして性能評価はシミュレータを利用し行われているが，ハードウェアリソース量の面から見ても標準的なアウトオブオーダ実行プロセッサの２倍程度のハードウェア量で実装可能であることを検証している [6][7][8][9][10].

2.3 関連製品

年々プロセッサに要求される回路規模は増大し，処理が複雑化している．そのためハードウェアとソフトウェアの開発をより密接に連携させる必要である．そこでハードウェア/ソフトウェア協調検証を行えるシステムが多く製品化されてきた．関連製品 Palladium XP もその内のひとつで 4 つのプラットフォームを実装しつつ，各プラットフォームはデータベースやフローにおいて互換性を持っている．そのためプラットフォーム間でデータの再コンパイルなどが不必要となっている．以下に各関連製品の詳細を示す

- Cadence Palladium XP

  ケイデンスの最先端名ハードウェア，ソフトウェアで実現した論理検証用コンピューティング・プラットフォームである．プロセッサ・ベースの計算エンジンと UXE(Unified Xccelerator Emulator) ソフトウェアにより，高速かつ柔軟な拡張性を実現するとともに従来のエミュレーションでは困難だった多様な用途に応えることができる [11]

- Showa&Sophia Technologies DS-5

  ARM プロセッサー搭載プラットフォーム向けの ARM 者推奨次世代開発ツールである．業界標準ともいえる Eclipse ベースの GUI で直感的な操作が可能となっている．サードパーティ性プラグ印との親和性も高く，Android SDK の ADT プラグインを追加インストールすることで java と C/C++のシームレスな開発が可能となる．各開発担当が同じ操作系を共有できるため交流も円滑に進む．[12]

- ALDEC HES-DVM

  HES-DVM は完全自動化機能とスクリプト環境を備えた SoC/ASIC デザインのハイブリッドバリデーション・検証環境である．ハードウェとソフトウェアの協調検証を可能にしていて，協調エミュレーションを活用することでハードウェアとソフトウェアの設計者は最新の FPGA テクノロジが利用可能となり，互いに同時並行で開発しながら RTL で開発・検証できる．[13]

2.4 その他の関連研究

プロセッサの性能向上が求められているため，性能検証の高速化は依然として求められている．性能検証の方法としてシミュレーション検証とエミュレーション検証がある．基本的にエミュレーション検証の方が高速であることがわかっている．そのためソフトウェアシミュレーションによる検証を高速化する様々な手法が提案されている．以下に各種関連研究の詳細を示す．

- ARM アーキテクチャ用仮想マシンモニタの実装

  今日，組み込み機器は広く普及し重要な要素となったいるが，その組み込み機器に要求される機能が非常に高度なものとなっている．また企業への新製品開発サイクル短縮への要求も同様に増加しており，これが原因となって組み込み機器製品の不具合や脆弱なセキュリティが問題となりかねない．それらを解決する手段として仮想マシンモニタ (VMM) の導入が考えられる．VMM はハードウェア上に複数の仮想マシンを構築できるため，セキュリティレベルによって個別の仮想マシンと OS を提供した場合，組み込み機器の安定性向上と強固なセキュリティの実現が可能となる．VMM は主に IA-32 アーキテクチャ専用だが，将来的な組み込み機器への利用を考え，ARM アーキテクチャ用の VMM を開発する [14].

- 組込みシステム向けマルチコア・プロセッサのためのソフトウェア開発支援

  高機能アプリケーションや汎用 OS を利用可能とするために，高性能・省電力型プロセッサへの需要が増大している．これに対して組込み機器向けプロセッサ・ベンダはマルチコア・プロセッサの開発を進めている．しかし，プロセッサのソフトウェア開発は煩雑なものとなりがちである．そこで携帯端末で広く使われている OMAP プロセッサを例に，ソフトウェア開発の問題点を明らかにし，解決し，これまで面倒だったマルチコアプロセッサを用いたシステム開発を可能とするランタイム環境を提案する [15].

第3章　提案システムの概要と実装手順

3.1　概要

前章で述べられている OROCHI や CoreSymphony は異種命令混合実行単一プロセッサであったりマルチコアプロセッサであるため，ARM の ISA ベースの単一コア・プロセッサの評価を得ることはできない．またハードウェア/ソフトウェア協調開発システムや OROCHI は独自のシミュレータもしくは Xilinx の ISE Design Suite(ISE) などの波形シミュレータを用いて性能評価を出力している．そこで FPGA 上で動作可能な ARM の ISA ベースの単一コア・プロセッサ (カスタムプロセッサ) とカスタムプロセッサのデータをやり取りする HostPC と FPGA 間のデータ通信機構を持つ FPGA プラットフォームを開発することでこれらの問題を解決できる．そのためカスタムプロセッサを実装しシリアル通信を用いたメモリアクセスや PCIe 通信による性能評価指数出力を追加実装することで，FPGA の拡張性を持ち実機に近い環境での動作検証と性能評価を得ることができるプラットフォームを開発する．以下の図 3.1.1 にシステム全体の構成図を示す．

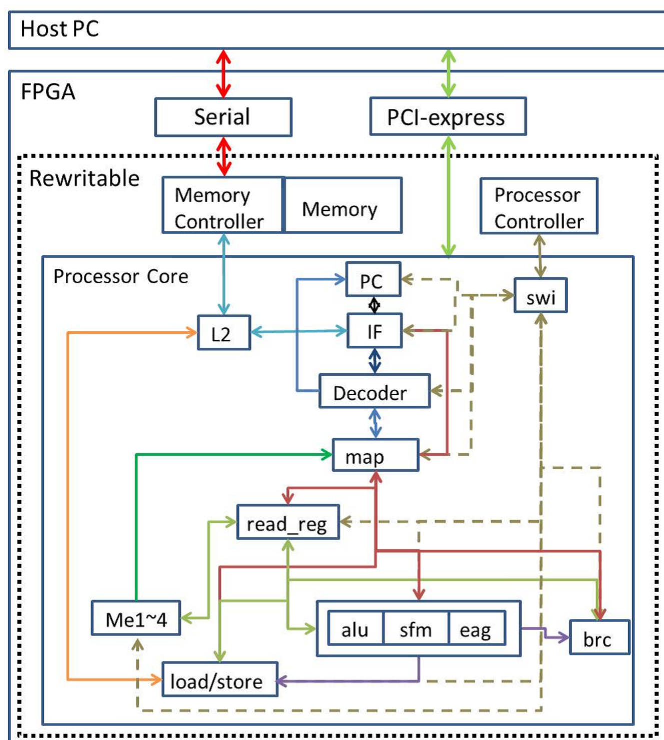

図 3.1.1: 提案システムの構成図

3.2 実装手順

プロセッサを実装するには，アーキテクチャの作成が必要不可欠である．しかし最初から全てを作るには命令セットの選択，デコーダや演算器などの機構，キャッシュやプリフェッチなどの理論と様々な事柄を理解し設計する必要がある．そのため多大な時間がかかってしまう．そこで，奈良先端科学技術大学院大学より異種命令セット同時実行プロセッサ OROCHI と OROCHI シミュレータ OSIM を参考にカスタムプロセッサを実装する．その後，シリアル通信，PCIe 通信を追加実装する．以降にカスタムプロセッサや各種通信機能を実装する際に行った手順を示す．

3.2.1 事前学習と準備

OROCHI シミュレータ OSIM の実行

事前準備として初めに OROCHI プロセッサの動作や出力結果等を調べるため，最終的に FPGA のエミュレーションと比較するため，OROCHI シミュレータ OSIM の動作を確認する．そこで OSIM の実行環境として以下の表 3.1 の Linux 環境を用意した．そしてソースコードと幾つかのサンプルプログラムの実行結果から OSIM の使用方法と構成を理解した．

表 3.1: シミュレーション環境

オペレーティングシステム (OS)	Windows XP Professional Version 2002 Service Pack2
CPU	Intel(R) Core(TM)2 Quad CPU Q9450 @2.66GHz
Memory	3.25GB RAM
仮想マシン	VMware Player Version 4.0.4.30409
仮想 OS	FreeBSD6.2R

OROCHI プロセッサの解析

次に，Verilog ソースの OROCHI プロセッサを解析・実行することで OROCHI の構成を理解し，カスタムプロセッサ構築のための知識を得た [4][5][16]．ここで，Verilog ソース解析から定義を書き換えることで様々な構成に変更可能であることやテストベンチを除くと主記憶，キャッシュやプロセッサ制御が FPGA 外にある構成で，論理合成には各機構を新しく追加する必要があることが確認できた．また実際に ISE でサンプルプログラムを実行し，構成の違いにより動作や出力結果にどのような差異が発生するかを確認した．

3.2.2 カスタムプロセッサの実装と検証

OROCHI の知識を元にカスタムプロセッサを実装し，ISE14.7 による動作検証を行った [17][18][19]. OROCHI の構成を変更するため定義の書き換えを行い単一アーキテクチャプロセッサへと分離し，OROCHI と同様のテストベンチの元でサンプルプログラムを実行した．その後，既存モジュールの書き換えやメモリ/プロセッサコントローラーや新規モジュールの追加を行い，論理合成可能なカスタムプロセッサを実装した．そしてプロセッサへの入力をクロック/リセットのみとしてテストベンチ新たに作成し，サンプルプログラムが同様に実行可能であることを確認した．

以下の図 3.2.1 に実装したカスタムプロセッサのブロック図を示す．

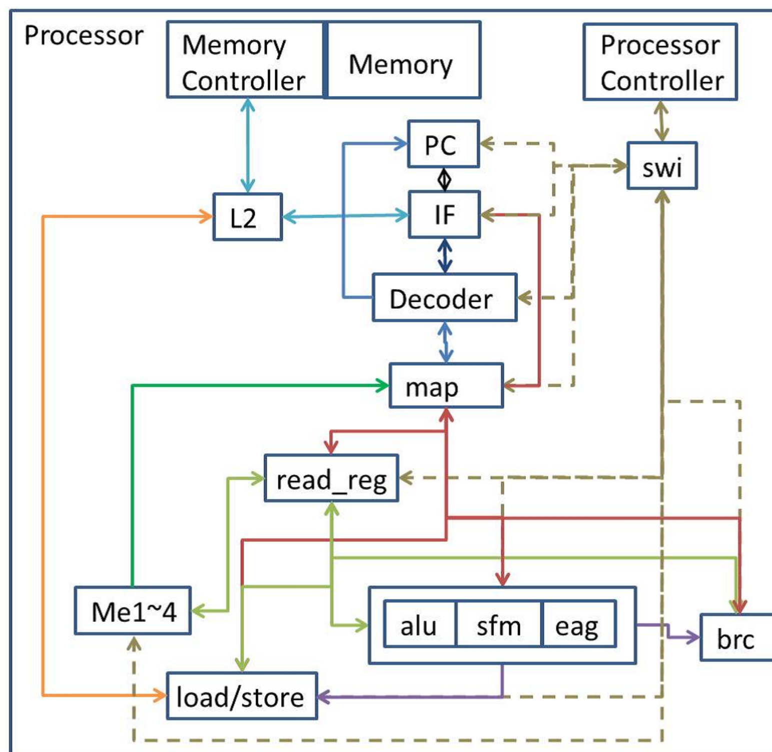

図 3.2.1: カスタムプロセッサブロック図

各種モジュールは以下の役割を行う．PC はプログラムカウンタを，IF は命令読み込みを，L2 は L2 キャッシュのアクセス制御を．decoder，map は命令解析と物理レジスタのマッピングを，read reg は各種演算器の利用状況管理と命令発行を，alu，sfm，eag，me1 は算術論理演算を，Load/Store と brc はロード・ストア命令と分岐命令を MemoryController と ProcessorController はメモリとプロセッサシステム全体の制御を担当する．

3.2.3　シリアル通信によるメモリアクセスの実装と検証

次に，カスタムプロセッサにシリアル通信機構を実装し，メモリに対してシリアル通信でデータのやり取りを可能にした [20][21]．その後，FPGA 評価ボード (ML605) 上でカスタムプロセッサとシリアル通信が正しく実機動作可能であることを確認した．前述のカスタムプロセッサは ISE 上でのシミュレーションによる動作確認は可能であるが，FPGA に実装したとしても外部と通信することができないため，プログラムデータを入力することも実行結果を出力することもできない．そこでデータ通信機能を追加実装する必要がある．しかしいきなりシリアル通信をを直接繋いでしまうと，回路のどの箇所で問題が発生しているか判別することができない．問題箇所を確認するためシリアル通信機構とカスタムプロセッサの間にバッファを接続し動作確認を行った．不必要なバッファとスイッチを排除し軽量化と手順の簡略化を行い，ML605 へ実装を行った．その結果，シリアル通信でメモリに直接プログラムを入力，実行後のメモリを出力として通信可能なことを確認した．なお，シリアル通信のデータ入力や出力は Cygwin 上で PySerial を使用した．以下の図 3.2.2 にバッファ有りのシリアル通信のブロック図を，図 3.2.3 に最終的に実装したバッファ無しのシリアル通信のブロック図を示す．

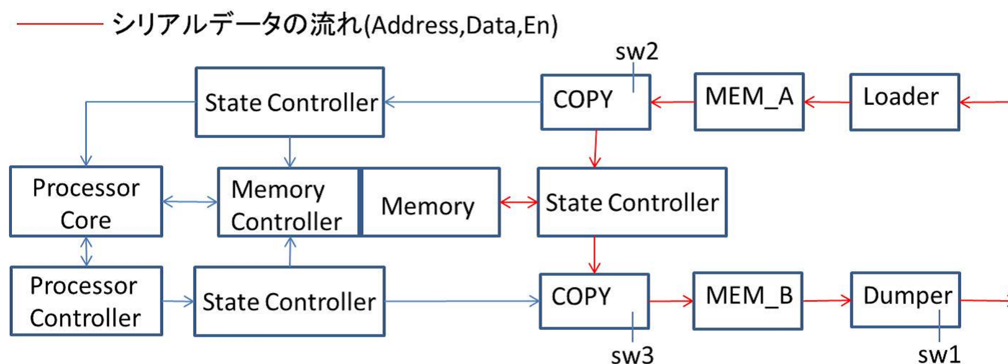

図 3.2.2: シリアル通信ブロック図/Buf 有り

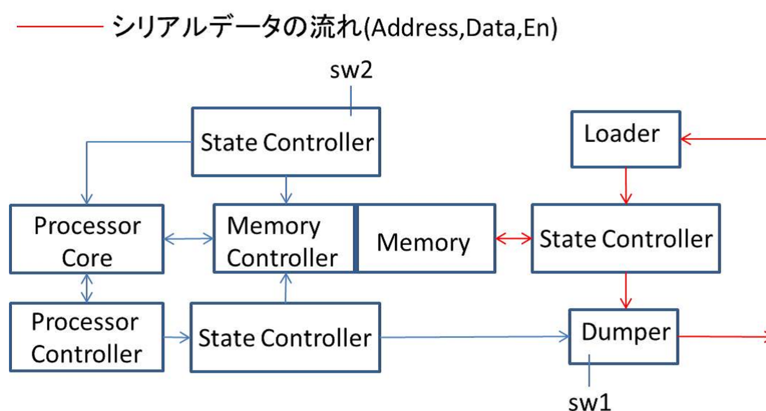

図 3.2.3: シリアル通信ブロック図/Buf 無し

3.2.4　PCIe 通信による性能評価指数出力の実装と検証

最後に，より高速な通信方法として PCIe 通信を実装した．前述のシリアル通信では直接メモリの読み書きを行えるため，メモリに記入されている事柄については確認することができるが，記入されていないことは確認できない．またシリアル通信でデータのやり取りを行うことは実行速度の面から見て，非常に遅く合理的ではない．そのためより高速な通信方法として PCIe が考えられた．そこで PCIe の IPcore を用いてプロセッサのサイクル数やキャッシュのヒット/ミス数などの性能評価指数を高速かつプロセッサの状態によらず，確認可能に設計し ML605 に実装した．その後，正しく結果が出力されていることを確認した．以下の図 3.2.4 に PCIe 通信のブロック図を示す図の Custom Processor 部分が今回実装した書き換え可能なカスタムプロセッサで，PCIe 通信の GPIO Port へ性能評価指数を出力している．

図 3.2.4: PCIe 通信ブロック図

第4章　実装したプラットフォームの検証と評価

4.1　様々なプログラムに対するカスタムプロセッサの動作検証

ISE14.7 を用いて，カスタムプロセッサ上で様々なプログラムをシミュレーション実行することで，各プログラムが実装したカスタムプロセッサで実行可能であることを確認する．またシミュレーション結果や出力される波形図からカスタムプロセッサが設計した通りに動作していることを検証する．以降に ISE の出力結果と各プログラムのソースコードを示す.

Hillo

Hillo は Hello を出力後，e を i に書き換えて出力するプログラムである．Hillo の ISE シミュレーション結果を図 4.1.1 に，ソースコードをソースコード 4.1 に全体波形図を図 4.1.2 に，一部拡大波形図を図 4.1.3 に示す.

まず図 4.1.1 を見ると，初めに Hello が出力され，その後 Hillo が出力されている．この結果とソースコード 4.1 から正しくプログラムが動作していることがわかる．次に図 4.1.2 を見ると，3 列目から順番に [242c，24e，14，3e，7，b, 6] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.1 の最後に同様の値が出力されている．最後に図 4.1.3 を見ると，最下段の信号 char の赤枠内に Hillo が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で Hillo が実行され，回路が想定通り動作していることがわかる.

```
Syscall : Puts
Hello

Syscall : Puts
Hillo

Syscall : Exit
cycle_counter                       :0x000000000000242c
run_cycle_counter(cycle - swi)      :0x000000000000024e
arm_terminal_instruction_counter    :0x0000000000000014
armcache_hit                        :0x0000003e
armcache_miss                       :0x00000007
op1cache_hit                        :0x0000000b
op1cache_miss                       :0x00000006
```

図 4.1.1: Hillo シミュレーション結果

ソースコード 4.1: Hillo

```
1 char a[1024] = "Hello\n";;
2 main(){
3     int i;
4     for (i=1; i<=1; i++){
5         puts(a);
6         a[1] = 'i';
7         puts(a);
8     }}
```

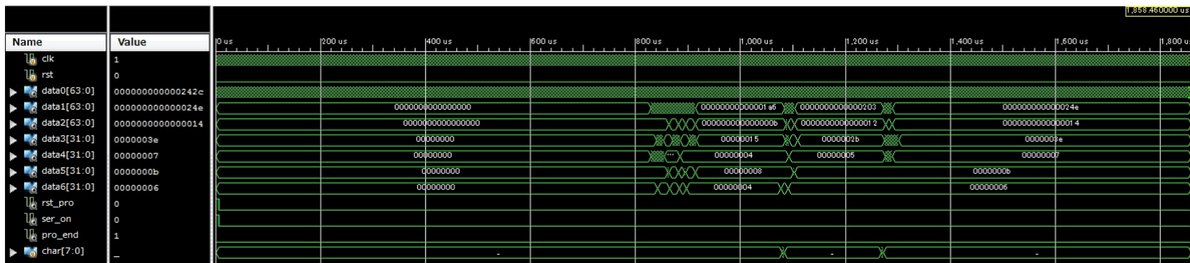

図 4.1.2: Hillo 全体波形図

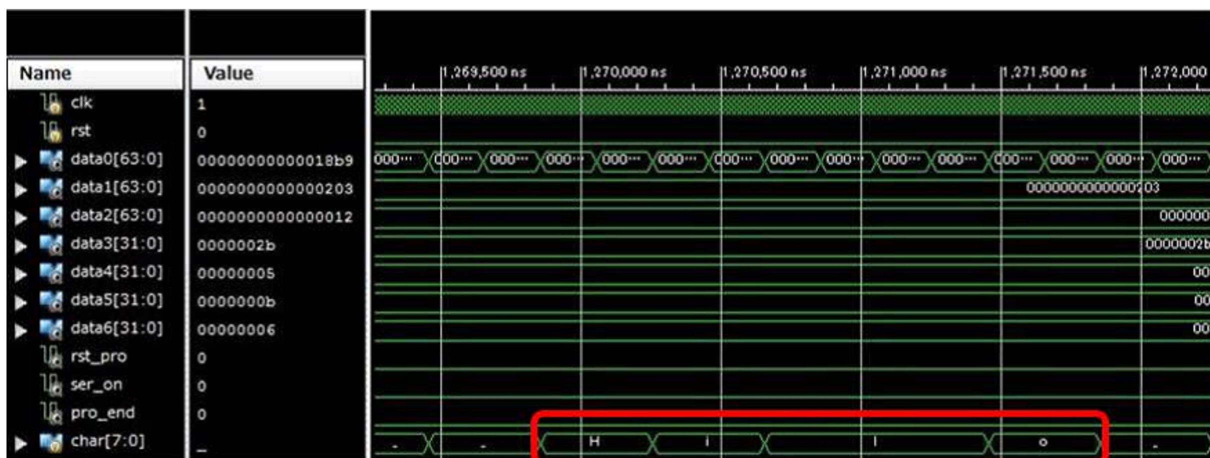

図 4.1.3: Hillo 拡大波形図

Gusu

Gusu は 100 までの偶数を出力するプログラムである．Gusu の ISE シミュレーション結果を図 4.1.4 に，ソースコードをソースコード 4.2 に全体波形図を図 4.1.5 に，一部拡大波形図を図 4.1.6 に示す.

まず図 4.1.4 を見ると，2，4，6，…98，100 と出力されている．この結果とソースコード 4.2 から正しくプログラムが動作していることがわかる．次に図 4.1.5 を見ると，３列目から順番に [215e4, 133f6，34c9，3f60，b0，13c4，27e] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.4 の最後に同様の値が出力されている．最後に図 4.1.6 を見ると，最下段の信号 char の赤枠内に 98 が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で Gusu が実行され，回路が想定通り動作していることがわかる．

```
Syscall : Write
2
Syscall : Write
4
Syscall : Write
6
Syscall : Write
8
Syscall : Write
10
Syscall : Write
12
Syscall : Write
14
Syscall : Write
16
Syscall : Write
18
Syscall : Write
20
Syscall : Write
22
Syscall : Write
24
Syscall : Write
26
Syscall : Write
28
Syscall : Write
30
Syscall : Write
32
Syscall : Write
34
Syscall : Write
36
Syscall : Write
38
Syscall : Write
40
Syscall : Write
42
Syscall : Write
44
Syscall : Write
46
Syscall : Write
48
Syscall : Write
50
Syscall : Write
52
Syscall : Write
54
Syscall : Write
56
Syscall : Write
58
Syscall : Write
60
Syscall : Write
62
Syscall : Write
64
Syscall : Write
66
Syscall : Write
68
Syscall : Write
70
Syscall : Write
72
Syscall : Write
74
Syscall : Write
76
Syscall : Write
78
Syscall : Write
80
Syscall : Write
82
Syscall : Write
84
Syscall : Write
86
Syscall : Write
88
Syscall : Write
90
Syscall : Write
92
Syscall : Write
94
Syscall : Write
96
Syscall : Write
98
Syscall : Write
100
Syscall : Exit
cycle_counter                       :0x00000000000215e4
run_cycle_counter(cycle - swi)      :0x00000000000133f6
arm_terminal_instruction_counter    :0x00000000000034c9
armcache_hit                        :0x00003f60
armcache_miss                       :0x000000b0
op1cache_hit                        :0x000013c4
op1cache_miss                       :0x0000027e
```

図 4.1.4: Gusu シミュレーション結果

ソースコード 4.2: Gusu

```
1 #include <stdio.h>
2 main() {
3 int i;
4     for (i=1;i<=100;i++){
5         if(i%2==0) printf("%d␣",i);
6     }}
```

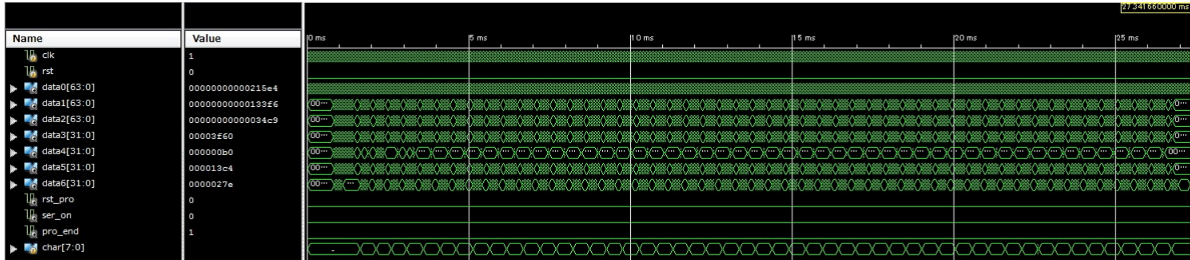

図 4.1.5: Gusu 全体波形図

図 4.1.6: Gusu 拡大波形図

Foradd

Foradd は 1〜99 までの数を合計値を出力するプログラムである．Foradd の ISE シミュレーションした結果を図 4.1.7 に，ソースコードをソースコード 4.3 に全体波形図を図 4.1.8 に，一部拡大波形図を図 4.1.9 に示す.

まず図 4.1.7 を見ると，4950 と出力されている．この結果とソースコード 4.3 から正しくプログラムが動作していることがわかる．次に図 4.1.8 を見ると，３列目から順番に [2d1a，dc2，11b，e4，36，61，e] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.7 の最後に同様の値が出力されている．最後に図 4.1.9 を見ると，最下段の信号 char の赤枠内に 495…が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で Foradd が実行され，回路が正常に動作していることがわかる．

```
Syscall : Write
4950
Syscall : Exit
cycle_counter                        :0x0000000000002d1a
run_cycle_counter(cycle - swi)       :0x0000000000000dc2
arm_terminal_instruction_counter     :0x000000000000011b
armcache_hit                         :0x000000e4
armcache_miss                        :0x00000036
op1cache_hit                         :0x00000061
op1cache_miss                        :0x0000000e
```

図 4.1.7: Foradd シミュレーション結果

ソースコード 4.3: Foradd

```c
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int i;
    int kekka=0;
    for (i=0; i<100; i++){
        kekka = kekka + i;}
    printf("%d",kekka);
    return 0;}
```

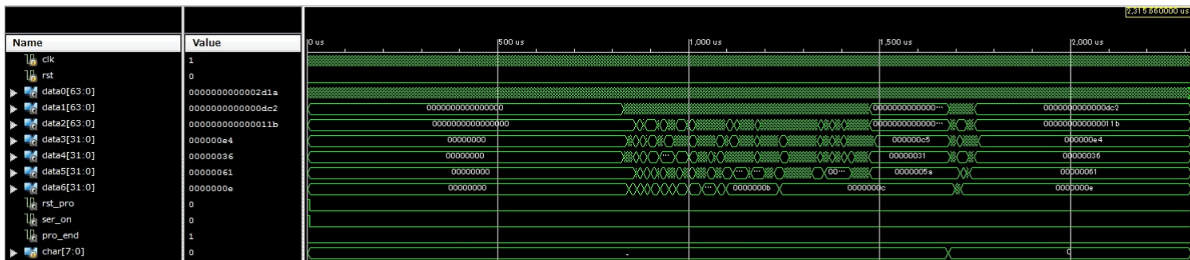

図 4.1.8: Foradd 全体波形図

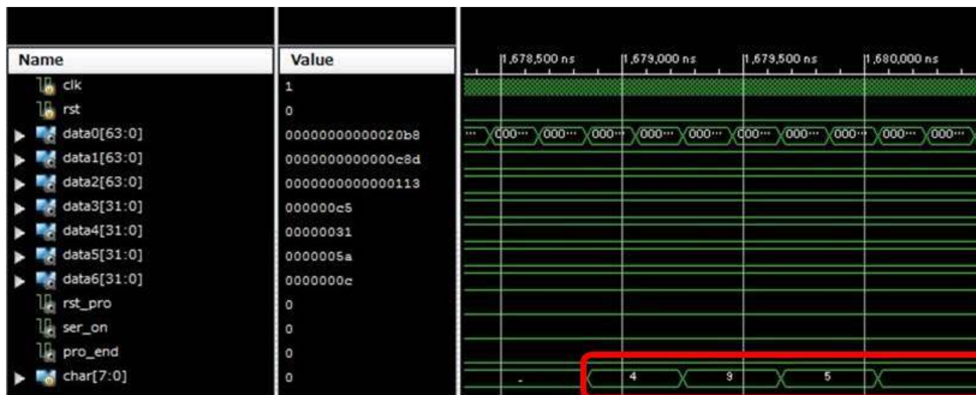

図 4.1.9: Foradd 拡大波形図

Fibonacci

Fibonacci はフィボナッチ数列を 10 項目まで出力するプログラムである．Fibonacci の ISE シミュレーション結果を図 4.1.10 に，ソースコードをソースコード 4.4 に全体波形図を図 4.1.11 に，一部拡大波形図を図 4.1.12 に示す．

まず図 4.1.10 を見ると，0，1，1，2，3，5，8，13，21，34 と出力されている．この結果とソースコード 4.4 から正しくプログラムが動作していることがわかる．次に図 4.1.11 を見ると，３列目から順番に [2d1a，dc2，11b，e4，36，61，e] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.10 の最後に同様の値が出力されている．最後に図 4.1.12 を見ると，最下段の信号 char の赤枠内に 34 が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で Fibonacci が実行され，回路が正常に動作していることがわかる．

```
Syscall : Write
0
Syscall : Write

1
Syscall : Write

1
Syscall : Write

2
Syscall : Write

3
Syscall : Write

5
Syscall : Write

8
```

```
Syscall : Write

13
Syscall : Write

21
Syscall : Write

34
Syscall : Exit
cycle_counter                      :0x0000000000007eed
run_cycle_counter(cycle - swi)     :0x0000000000003bc7
arm_terminal_instruction_counter   :0x000000000000096a
armcache_hit                       :0x00000abe
armcache_miss                      :0x00000048
op1cache_hit                       :0x000003a1
op1cache_miss                      :0x00000072
```

図 4.1.10: Fibonacci シミュレーション結果

ソースコード 4.4: Fibonacci

```
1 #include <stdio.h>
2 int main(void){
3     int a,b=0,c=1,i;
4     for(i=1;i<=10;i++){
5         printf("%d\n", b);
6         a = b+c;
7         b = c;
8         c = a;}
9     return 0;}
```

図 4.1.11: Fibonacci 全体波形図

図 4.1.12: Fibonacci 拡大波形図

FizzBuzz

FizzBuzz は 15 までの数値の内，3 の倍数に Fizz を 5 の倍数に Buzz を 15 の倍数に FizzBuzz を出力するプログラムである．FizzBuzz の ISE シミュレーション結果を図 4.1.13 に，ソースコードをソースコード 4.5 に，全体波形図を図 4.1.14 に，一部拡大波形図を図 4.1.15 に示す．

まず図 4.1.13 を見ると，1，2，Fizz，…，13，14，Fizz,Buzz と出力されている．この結果とソースコード 4.5 から正しくプログラムが動作していることがわかる．次に図 4.1.14 を見ると，３列目から順番に [a7c1，50bf，c4c，e0f，48，51c，93] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.13 の最後に同様の値が出力されている．最後に図 4.1.15 を見ると，最下段の信号 char にの赤枠内 Fizz,Buzz が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で FizzBuzz が実行され，回路が正常に動作していることがわかる．

```
Syscall : Write
1
Syscall : Write

2
Syscall : Write

Fizz
Syscall : Write

4
Syscall : Write

Buzz
Syscall : Write

Fizz
Syscall : Write

7
Syscall : Write

8
Syscall : Write

Fizz
```

```
Syscall : Write

Buzz
Syscall : Write

11
Syscall : Write

Fizz
Syscall : Write

13
Syscall : Write

14
Syscall : Write

Fizz,Buzz
Syscall : Exit
cycle_counter                     :0x000000000000a7c1
run_cycle_counter(cycle - swi)    :0x00000000000050bf
arm_terminal_instruction_counter  :0x0000000000000c4c
armcache_hit                      :0x00000e0f
armcache_miss                     :0x00000048
op1cache_hit                      :0x0000051c
op1cache_miss                     :0x00000093
```

図 4.1.13: FizzBuzz シミュレーション結果

ソースコード 4.5: FizzBuzz

```c
1  #include <stdio.h>
2  int main(void){
3      int i;
4      for (i = 1; i <= 15; i++) {
5          if (i % 3 == 0 && i % 5 == 0)
6              printf("Fizz,Buzz\n");
7          else if (i % 3 == 0)
8              printf("Fizz\n");
9          else if (i % 5 == 0)
10             printf("Buzz\n");
11         else
12             printf("%d\n", i);}
13     return 0;}
```

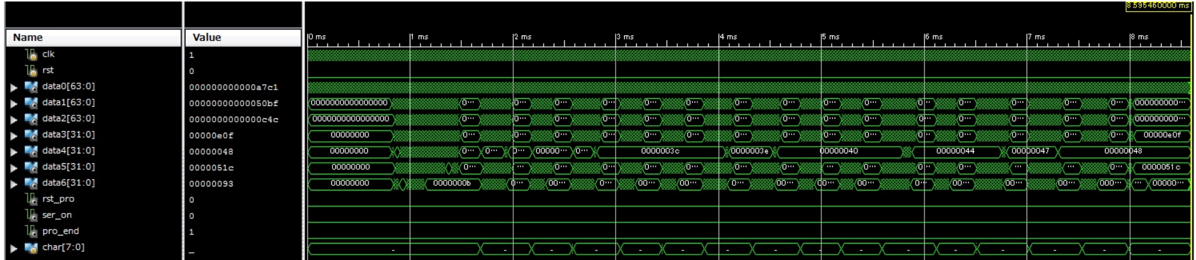

図 4.1.14: FizzBuzz 全体波形図

図 4.1.15: FizzBuzz 拡大波形図

Sort

Sort はランダムに並んだ数値を一度出力し，バブルソート後ソートされた数値を出力するプログラムである．Sort の ISE シミュレーション結果を図 4.1.16 に，ソースコードをソースコード 4.6 に，全体波形図を図 4.1.17，一部拡大波形図を図 4.1.18 に示す．

まず図 4.1.16 を見ると，80，5，36，…，9，1，78 と出力された後，1，5，9，…，78，80，100 と出力されている．この結果とソースコード 4.6 から正しくプログラムが動作していることがわかる．次に図 4.1.17 を見ると，３列目から順番に [fd99，83ed，17de，47ab，94，8e7，102] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.16 の最後に同様の値が出力されている．最後に図 4.1.18 を見ると，最下段の信号 char の赤枠内に 80 が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で Sort が実行され，回路が正常に動作していることがわかる．

```
Syscall : Write
80,
Syscall : Write
5,
Syscall : Write
36,
Syscall : Write
23,
Syscall : Write
12,
Syscall : Write
100,
Syscall : Write
45,
Syscall : Write
9,
Syscall : Write
1,
Syscall : Write
78,
Syscall : Write

Syscall : Write
1,
Syscall : Write
5,
Syscall : Write
9,
Syscall : Write
12,
Syscall : Write
23,
Syscall : Write
36,
Syscall : Write
45,
Syscall : Write
78,
Syscall : Write
80,
Syscall : Write
100,
Syscall : Write

Syscall : Exit
cycle_counter                        :0x000000000000fd99
run_cycle_counter(cycle - swi)       :0x00000000000083ed
arm_terminal_instruction_counter     :0x00000000000017de
armcache_hit                         :0x000047ab
armcache_miss                        :0x00000094
op1cache_hit                         :0x000008e7
op1cache_miss                        :0x00000102
```

図 4.1.16: Sort シミュレーション結果

ソースコード 4.6: Sort

```
1  #include <stdio.h>
2  int main(){
3      int data[MAX] = { 80,5,36,23,12,100,45,9,1,78 };
4      int n,i,w;
5      for ( i=0; i<MAX; i++ ){
6          printf("%d,",data[i]); }
7      printf("\n");
8      for ( n=MAX; n>1; n-- ){
9          for ( i=0; i<n-1; i++ ){
10             if ( data[i]>data[i+1] ){
11                 w=data[i];
12                 data[i]=data[i+1];
13                 data[i+1]=w;}}}
14     for ( i=0; i<10; i++ ){
15         printf("%d,",data[i]);}
16     printf("\n");}
```

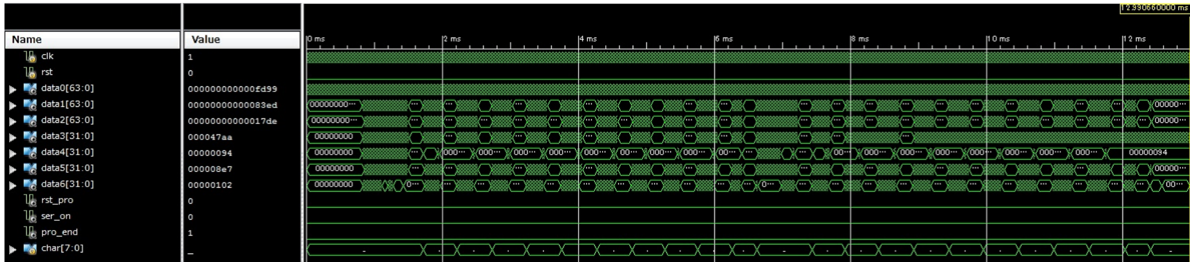

図 4.1.17: Sort 全体波形図

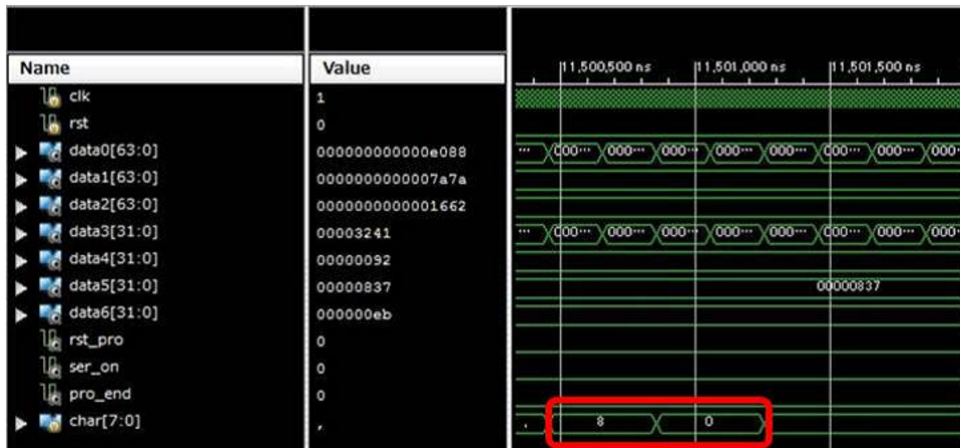

図 4.1.18: Sort 拡大波形図

Sosu

Sosu はエラトステネスの篩で 50 までの素数を出力するプログラムである．Sosu のソースコードをソースコード 4.7 に，ISE のシミュレーション結果を図 4.1.19 に，全体波形図を図 4.1.20 に，一部拡大波形図を図 4.1.21 に示す．

まず図 4.1.19 を見ると，2，3，5，7，…41，43，47 と出力されている．この結果とソースコード 4.7 から正しくプログラムが動作していることがわかる．次に図 4.1.20 を見ると，３列目から順番に [c913，6eb4，1364，15ea，6b，6b4，e2] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.19 の最後に同様の値が出力されている．最後に図 4.1.21 を見ると，最下段の信号 char にの赤枠内 4…が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で Sosu が実行され，回路が正常に動作していることがわかる．

```
Syscall : Write
 2
Syscall : Write
 3
Syscall : Write
 5
Syscall : Write
 7
Syscall : Write
 11
Syscall : Write
 13
Syscall : Write
 17
Syscall : Write
 19
Syscall : Write
 23
Syscall : Write
 29
Syscall : Write
 31
Syscall : Write
 37
Syscall : Write
 41
Syscall : Write
 43
Syscall : Write
 47
cycle_counter                       :0x000000000000c913
run_cycle_counter(cycle - swi)      :0x0000000000006eb4
arm_terminal_instruction_counter    :0x0000000000001364
armcache_hit                        :0x000015ea
armcache_miss                       :0x0000006b
op1cache_hit                        :0x000006b4
op1cache_miss                       :0x000000e2
```

図 4.1.19: Sosu シミュレーション結果

ソースコード 4.7: Sosu

```
1  #include <stdio.h>
2  #include <math.h>
3  #define NUM 50
4  int main(void){
5      unsigned i, j;
6      unsigned sq_num = (int)sqrt((double)NUM);
7      unsigned prime[NUM];
8      for (i = 0; i < NUM; i++)
9          prime[i] = 1;
10         prime[0] = 0;
11     for (i = 1; i < sq_num; i++) {
12         if (prime[i] == 1)
13             for (j = (i+1); (i+1) * j <= NUM; j++)
14                 prime[(i+1) * j - 1] = 0;}
15     for (i = 0; i < NUM; i++)
16         if (prime[i] == 1)
17             printf("%3d", i + 1);
18     putchar('\n');
19     return (0);}
```

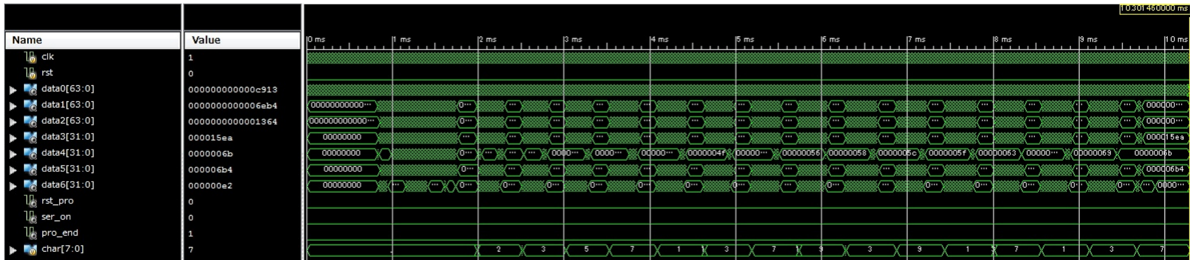

図 4.1.20: Sosu 全体波形図

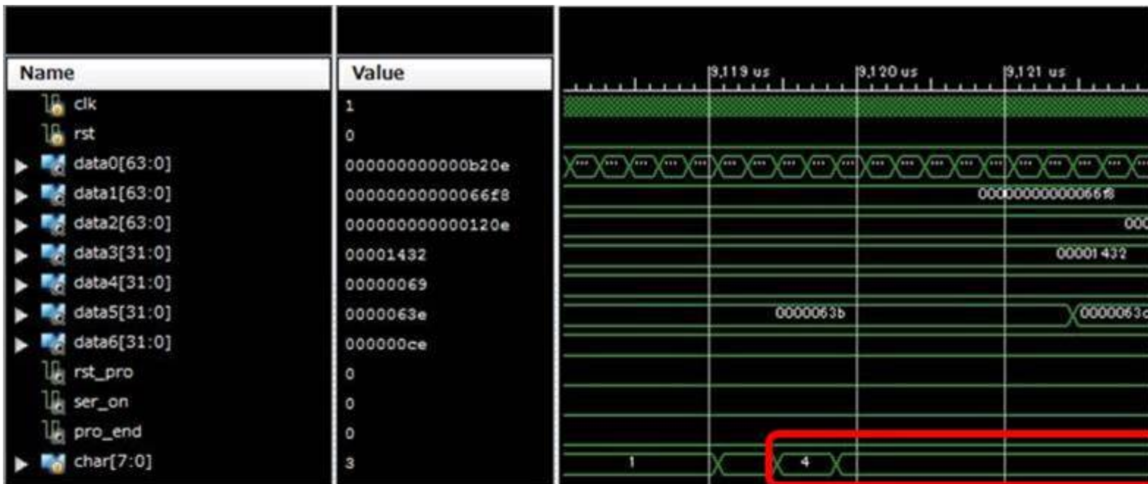

図 4.1.21: Sosu 拡大波形図

Himeno

Himeno は姫野ベンチマークの行列計算 (5 × 5 × 9) を指定回数行うプログラムである．Himeno の ISE のシミュレーション結果を図 4.1.22 に，全体波形図を図 4.1.23 に，一部拡大波形図を図 4.1.24 に示す．Himeno のソースコードは非常に長文のため付録に添付する．

まず図 4.1.19 を見ると，mimax=5 mjmax=5 mkmax=9 ⋯ Process end と出力されている．この結果とソースコード 5.1 から正しくプログラムが動作していることがわかる．次に図 4.1.20 を見ると，３列目から順番に [d12a9，cc186，4aecb，5c305，446，8b43，1c8] と出力されている．これはプログラム実行後に出力される性能評価指数を表していて，図 4.1.19 の最後に同様の値が出力されている．最後に図 4.1.21 を見ると，最下段の信号 char の赤枠内に Process end が出力されている．これは出力結果を格納している信号であり，他の値も同様に出力されている．これらの結果からカスタムプロセッサ上で Himeno が実行され，回路が正常に動作していることがわかる．

```
Syscall : Write
mimax = 5 mjmax = 5 mkmax = 9

Syscall : Write
imax = 4 jmax = 4 kmax =8

Syscall : Write
 Start rehearsal measurement process.

Syscall : Write
 Measure the performance in 10 times.


Syscall : Write
process end

Syscall : Exit
cycle_counter                      :0x00000000000d12a9
run_cycle_counter(cycle - swi)     :0x00000000000cc186
arm_terminal_instruction_counter   :0x000000000004aecb
armcache_hit                       :0x0005c305
armcache_miss                      :0x00000446
op1cache_hit                       :0x00008b43
op1cache_miss                      :0x000001c8
```

図 4.1.22: Himeno シミュレーション結果

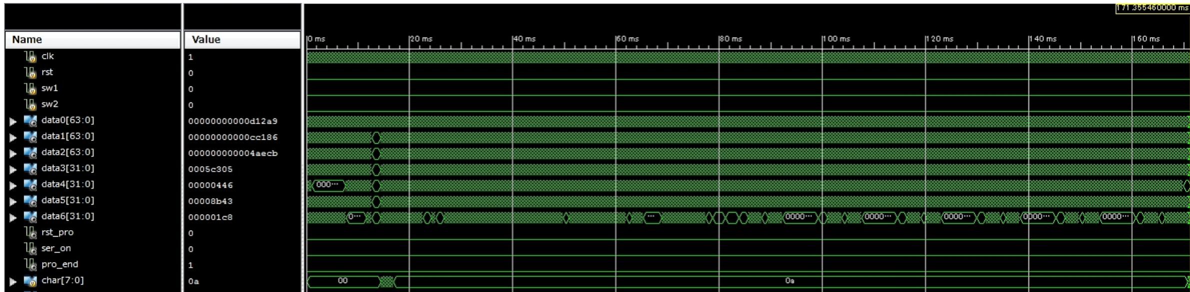

図 4.1.23: Himeno 全体波形図

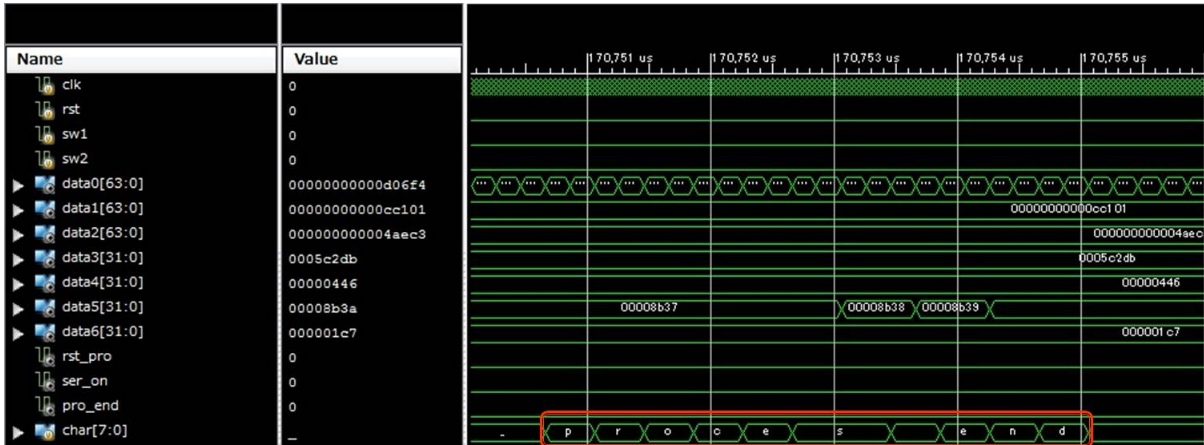

図 4.1.24: Himeno 拡大波形図

4.2 シリアル通信のメモリアクセス検証

カスタムプロセッサに追加実装したシリアル通信の入力データとプログラム実行後の出力データを比較し，二つの差異を見ることでシリアル通信が正しく実装されていることを確認する．図 4.2.1 から図 4.2.8 に各プログラムごとのデータの入出力の結果とデータ差異を示す．各図は上が入力データを下が出力データを表している．また，赤く塗られている部分が上下のデータで差異がある部分である．

各プログラムが正しく動作していることは確認されているため，どの図においても最後に出力した結果がメモリに格納されていることがわかる，もし，シリアル通信の設計が正しくできていない場合，アドレスのずれやビット欠けによるデータの欠落などが起こる．しかし，どの図をとってみてもアドレスのずれやデータの欠落はしていない．この結果から入力データと出力データが正しくやり取りされていることが証明でき，シリアル通信を含む回路全体が正しく実装されていることがわかる，

Hillo.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
0001F260	0C	00	0D	00	0E	00	0F	00	10	00	00	00	48	65	6C	6C	............Hell
0001F270	6F	0A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	o...............

output-Hillo.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
0001F260	0C	00	0D	00	0E	00	0F	00	10	00	00	00	48	69	6C	6C	............Hill
0001F270	6F	0A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	o...............

図 4.2.1: Hillo メモリデータ入出力

Gusu.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF720	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF730	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF740	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF750	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF760	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

output-Gusu.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	00	00	00	00	00	00	00	31	30	30	00	00	00	..........100...
000FF720	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF730	03	00	00	00	6C	F7	FF	07	00	00	00	00	00	00	00	00	....l...........
000FF740	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF750	D8	5D	01	00	00	00	00	00	00	00	00	00	18	00	01	00	ﾘ]..............
000FF760	00	00	00	00	00	00	00	00	00	00	00	00	31	30	30	20	100

図 4.2.2: Gusu メモリデータ入出力

Foradd.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF720	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF730	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF740	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF750	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF760	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF770	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

output-Foradd.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	00	00	00	00	00	00	00	00	00	00	34	39	35	495
000FF720	30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0...............
000FF730	00	00	00	00	0C	10	00	00	70	F7	FF	07	00	00	00	00	........p.......
000FF740	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF750	00	00	00	00	D8	5D	01	00	00	00	00	00	18	00	01	00	....ﾘ]..........
000FF760	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF770	34	39	35	30	00	00	00	00	00	00	00	00	00	00	00	00	4950............

図 4.2.3: Foradd メモリデータ入出力

Fibonacci.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF720	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF730	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF740	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF750	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF760	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

output-Fibonacci.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	33	34	00	00	00	00	00	00	00	00	00	00	00	...34...........
000FF720	00	00	00	00	00	00	00	00	03	00	00	00	64	F7	FF	07	............d...
000FF730	37	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	7...............
000FF740	00	00	00	00	00	00	00	00	D8	5D	01	00	37	00	00	00	........ﾘ]..7...
000FF750	00	00	00	00	00	00	00	00	00	00	00	00	18	00	01	00	....
000FF760	00	00	00	00	33	34	0A	00	00	00	00	00	00	00	00	00	....34..........

図 4.2.4: Fibonacci メモリデータ入出力

```
FizzBuzz.bin
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
000FF710  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF720  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF730  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF740  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF750  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF760  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF770  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
output-FizzBuzz.bin
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
000FF710  00 00 00 00 00 00 00 31 34 00 00 00 00 00 00 00   .......14.......
000FF720  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF730  68 F7 FF 07 00 00 00 00 00 00 00 00 00 00 00 00   h...............
000FF740  00 00 00 00 00 00 00 00 00 00 00 00 D8 5D 01 00   ............ﾘ]..
000FF750  0F 00 00 00 00 00 00 00 00 00 00 00 18 00 01 00   ................
000FF760  00 00 00 00 00 00 00 00 46 69 7A 7A 2C 42 75 7A   ........Fizz,Buz
000FF770  7A 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00   z...............
```

図 4.2.5: FizzBuzz メモリデータ入出力

```
Sort.bin
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
000FF6E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF6F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF700  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF710  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF720  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF730  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF740  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF750  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF760  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF770  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
output-Sort.bin
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
000FF6E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 31 30   ..............10
000FF6F0  30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   0...............
000FF700  00 00 00 00 03 00 00 00 40 F7 FF 07 00 00 00 00   ........@.......
000FF710  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF720  00 00 00 00 D8 5D 01 00 00 00 00 00 01 00 00 00   ....ﾘ]..........
000FF730  05 00 00 00 09 00 00 00 04 00 00 00 74 F7 FF 07   ............t...
000FF740  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF750  00 00 00 00 00 00 00 00 D8 5D 01 00 00 00 00 00   ........ﾘ]......
000FF760  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000FF770  00 00 00 00 0A 00 00 00 00 00 00 00 00 00 00 00   ................
```

図 4.2.6: Sort メモリデータ入出力

Sosu.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF650	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF660	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF670	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF680	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF690	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF6A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

output-Sosu.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF650	00	00	00	34	37	00	00	00	00	00	00	00	00	00	00	00	...47..........ｰ
000FF660	00	00	00	00	00	00	00	00	01	00	00	00	A4	F6	FF	07	............､...
000FF670	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF680	00	00	00	00	00	00	00	00	D8	5D	01	00	30	00	00	08	........ﾘ]..0...
000FF690	00	00	00	00	01	00	00	00	01	00	00	00	00	00	00	00	....
000FF6A0	01	00	00	00	20	34	37	00	00	00	00	00	00	00	00	00	.... 47.........

図 4.2.7: Sosu メモリデータ入出力

Himeno.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF720	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF730	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF740	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF750	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF760	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF770	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000FF780	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

output-Himeno.bin

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	0123456789ABCDEF
000FF710	00	00	00	31	30	00	00	00	00	00	00	00	00	00	00	00	...10...........
000FF720	00	00	00	00	00	00	00	00	0A	00	00	00	64	F7	FF	07	............d...
000FF730	E8	CC	3C	00	00	00	00	00	00	00	00	00	00	00	00	00	靂<.............
000FF740	00	00	00	00	00	00	00	00	D8	5D	01	00	0A	00	00	00	........ﾘ]......
000FF750	00	00	00	00	00	00	00	00	00	00	00	00	18	00	01	00	....
000FF760	00	00	00	00	70	72	6F	63	65	73	73	20	65	6E	64	0A	....process end.
000FF770	00	70	65	72	66	6F	72	6D	61	6E	63	65	20	69	6E	20	.performance in
000FF780	31	30	20	74	69	6D	65	73	2E	0A	0A	00	00	00	00	00	10 times........

図 4.2.8: Himeno メモリデータ入出力

4.3 PCIe 通信の性能評価指数出力検証

実装した PCIe 通信と動作検証用のプログラムを使用し，性能評価指数がプロセッサの状態にかかわらず出力できることを確認する．また，PCIe 通信を用いた性能評価指数の出力結果と ISE14.7 のシミュレーションの出力結果を比較し，動作を確認する．以降の図 4.3.1 から図 4.3.8 に各プログラムごとの性能評価指数の出力を示す.

各プログラムに対する PCIe の出力結果と ISE14.7 のシミュレーション結果を比較すると，各所に一致もしくは類似する数値が入っていることから PCIe 通信によって正しく性能評価指数が出力されていることがわかる．出力結果を比較して数値が違う箇所は動作環境の違いによるものであると考えられる．シミュレーション検証に比べてエミュレーション検証が実機に近い環境であることやシミュレーションとエミュレーションによる不定値の扱いなどの点から PCIe 通信側の出力結果がより正しい出力と考えられる．出力結果の行番号と詳細内容を以下の表 4.1 にまとめる．

表 4.1: PCIe 通信出力詳細

行番号	出力詳細
0	ボードサイクル数
1，2	実行サイクル数
3，4	実行サイクル数 – システムコールサイクル数
5，6	ARM 命令実行数
7	命令キャッシュ/ヒット数
8	命令キャッシュ/ミス数
9	データキャッシュ/ヒット数
10	データキャッシュ/ミス数
13	プロセッサリセット
14	シリアル通信 on/off
15	プログラム終了フラグ

```
GPIOs
 [0] 29fbf04b
 [1] 00000000
 [2] 0000242c
 [3] 00000000
 [4] 0000024e
 [5] 00000000
 [6] 00000014
 [7] 0000003e
 [8] 00000007
 [9] 0000000b
 [10] 00000006
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000001
 [15] 00000001
```

図 4.3.1: PCIe 出力/Hillo

```
GPIOs
 [0] 246b4b1f
 [1] 00000000
 [2] 000215ea
 [3] 00000000
 [4] 000133fb
 [5] 00000000
 [6] 000034c9
 [7] 00003f0d
 [8] 000000c7
 [9] 000013c4
 [10] 0000027c
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000001
 [15] 00000001
```

図 4.3.2: PCIe 出力/Gusu

```
GPIOs
 [0] 6aba2ca5
 [1] 00000000
 [2] 00002d1a
 [3] 00000000
 [4] 00000dc2
 [5] 00000000
 [6] 0000011b
 [7] 000000e4
 [8] 00000036
 [9] 00000061
 [10] 0000000e
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000000
 [15] 00000001
```

図 4.3.3: PCIe 出力/Foradd

```
GPIOs
 [0] 05127fc6
 [1] 00000000
 [2] 00007eed
 [3] 00000000
 [4] 00003bc7
 [5] 00000000
 [6] 0000096a
 [7] 00000aba
 [8] 00000049
 [9] 000003a1
 [10] 00000072
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000000
 [15] 00000001
```

図 4.3.4: PCIe 出力/Fibonacci

```
GPIOs
 [0] c2b97296
 [1] 00000000
 [2] 0000a7c1
 [3] 00000000
 [4] 000050bf
 [5] 00000000
 [6] 00000c4c
 [7] 00000e0b
 [8] 00000049
 [9] 0000051c
 [10] 00000093
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000001
 [15] 00000001
```

図 4.3.5: PCIe 出力/FizzBuzz

```
GPIOs
 [0] 565bebaf
 [1] 00000000
 [2] 0000fd90
 [3] 00000000
 [4] 000083e4
 [5] 00000000
 [6] 000017de
 [7] 00004799
 [8] 00000095
 [9] 000008e6
 [10] 00000102
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000001
 [15] 00000001
```

図 4.3.6: PCIe 出力/Sort

```
GPIOs
 [0] ab2824c9
 [1] 00000000
 [2] 0000c913
 [3] 00000000
 [4] 00006eb4
 [5] 00000000
 [6] 00001364
 [7] 000015d6
 [8] 00000070
 [9] 000006b4
 [10] 000000e2
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000001
 [15] 00000001
```

図 4.3.7: PCIe 出力/Sosu

```
GPIOs
 [0] 14cf44bc
 [1] 00000000
 [2] 000d125e
 [3] 00000000
 [4] 000cc13b
 [5] 00000000
 [6] 0004aecb
 [7] 0005c297
 [8] 00000446
 [9] 00008b4c
 [10] 000001c9
 [11] 00000000
 [12] 00000000
 [13] 00000000
 [14] 00000001
 [15] 00000001
```

図 4.3.8: PCIe 出力/Himeno

4.4 性能評価

第一に今回実装した FPGA プラットフォームが十分な拡張性を持っていることを確認する．論理合成の結果からプラットフォームの各機構の動作周波数と使用リソース量を以下の表 4.2 に示す

表 4.2: 動作周波数と使用リソース量

	動作周波数	BRAM	LUTs	Registers
Processor	5MHz	63%	34%	6%
Serial	10MHz	1%	1%	1%
PCIe	100MHz	3%	1%	1%

この結果から新たに機能拡張を行うのに十分なリソース量が残されていることがわかる．

第二に作成したプラットフォームのエミュレーションにかかる時間を評価する．まず各プログラムの FPGA 実行時間は以下の式によって得ることができる．

$$\text{実行時間 } T[\mathrm{s}] = \frac{\text{サイクル数}}{\text{プロセッサの動作周波数 [Hz]}}$$

なお，サイクル数は前述の PCIe 出力結果から確認でき，動作周波数はプロセッサ設計時に行う設定から 5[MHz] であるため，FPGA 実行時間は下式のようになる

$$\text{実行時間 } T[\mathrm{s}] = \text{サイクル数} \times 200[\mathrm{ns}]$$

次にソフトウェアシミュレータのシミュレーションにかかる時間を評価する．OSIM でシミュレーション開始から終了までの時間を１０回測定し，その平均を OSIM 実行時間とする．以上から求められる各プログラムの FPGA 実行時間と OSIM 実行時間を表 4.3 に示す.

表 4.3: 実行時間比較

プログラム名	FPGA[ms]	OSIM[ms]
Hillo	1.86	35500
Gusu	27.40	36335
Foradd	2.31	35562
Fibonacci	6.50	36651
FizzBuzz	8.59	37133
Sort	13.0	36674
Sosu	10.3	36841
Himeno	171	48206

この結果から FPGA の実行時間が OSIM の実行時間に比べて最小で 282 倍，最大で 19086 倍高速であることがわかる．また論理合成の結果から FPGA プラットフォーム全体の動作周波数 5[MHz] に対して，最大動作周波数は 24.639[MHz] とであるため，今回の回路構成でも更に約 5 倍の高速化が望める．

第5章　結論

5.1　本研究のまとめ

本稿ではカスタムプロセッサ構築用FPGAプラットフォームの実現を最終的な目的としてFPGAプラットフォームの実装と評価を行った．命令セットには組み込み用プロセッサの業界標準であるARM命令セットを採用した．

第一に，XIlinx Design Suite14.7のシミュレータであるISimを使用し，設計したカスタムプロセッサが想定通りに実装できていることを検証した．

第二に，シミュレータ上で単純な文字の出力のみだけでなく，数字の出力・単純な演算・繰り返し処理・複雑な演算・それらの組み合わせなどの様々なプログラムに対して設計したカスタムプロセッサが動作可能であることを確認した．

第三に，シリアル通信でメモリへの読み書きする機構を追加し，直接メモリに読み書き可能であること，またシリアル通信を含めてFPGA上で様々なプログラムに対してカスタムプロセッサが正しく動作することを確認した．

第四に，より高速な通信方法としてPCIe通信を追加することで，カスタムプロセッサの状態に依らず，かつ高速に性能評価指数を出力できることを確認した．

最後に，FPGAの使用リソースの観点から十分な拡張性を持っていることを検証した．またシミュレータとFPGAのそれぞれで，設計した回路のプログラム実行にかかる時間を求めた．その結果，拡張に対して十分なリソースが確保されていること，シミュレータを使用した動作検証と比べて，FPGAを使用した動作検証の方がおよそ数百倍〜数万倍高速であることを確認した．これはプロセッサアーキテクチャを研究する上でFPGAを用いた高速化が有用であることを意味する．

これらの結果からこのFPGAプラットフォームは拡張性，高速動作性，様々なプログラムを動作させることができるプロセッサを保持し，カスタムプロセッサ構築用のプラットフォームとして有用であることが確認できた．

5.2　今後の課題

第一に，シミュレータと FPGA のそれぞれで設計した回路の実行時間を比較しているが，今回はプログラムの実行時間のみの比較であり，通信時間を除外している．理由はシリアル通信ではデータのやり取りに膨大な時間がかかるためである．その改善の一環として PCIe による性能評価指数の検出を行っているが，メモリへの読み書きは依然としてシリアル通信によって行われている．より高速な動作のために，シリアル通信で行われているメモリへの読み書きを PCIe 通信，若しくは他の高速な通信方法に置き換えること．

第二に，PCIe の IPcore と作成したプロセッサの伝送遅延と動作周波数の関係からプロセッサ側の動作周波数を非常に押さえている．そのため，FPGA の実行時間が非常に遅くなっている．なので，プロセッサ本体の回路構成の変更や IP コアを使わず通信方法を確立することでプロセッサを含む全体の性能を向上させること．

第三に，今回の検証では FPGA 上の動作検証と通信方法の確立は行っているが他の動作検証評価は行っていない．そこで，現回路構成以外の新しい回路構成を実装し，正しく動作するかを確認すること．

以上の三点が今後の課題としてあげられる．

謝辞

本研究を進めるにあたり，ご指導をいただいた修士論文指導教員の吉永努教授，吉見真聡助教に感謝いたします．また常日頃から助言や励まし，心配のお言葉をいただいた吉永・入江研究室の皆様に感謝します.

参考文献

[1] 小島知也, 中島康彦. OROCHI 評価用集中命令ウィンドウ型スーパスカラの設計. 奈良先端科学技術大学院大学情報科学研究科, 2006.

[2] 吉村和浩, 中田尚, 中島康彦. 異種命令 SMT プロセッサ OROCHI の実装と分析. 奈良先端科学技術大学院大学情報科学研究科, 2008.

[3] 山原幹雄, 中田尚, 中島康彦. 異種命令混在実行プロセッサにおけるプロセススケジューリング手法. 奈良先端科学技術大学院大学, 2008.

[4] 小島知也. 異種命令セット同時実行プロセッサ OROCHI における命令分解機構の設計と評価. 奈良先端科学技術大学院大学情報科学研究科, 2007.

[5] 市來亮人. SMT プロセッサ向けの正確な記憶下位層モデルの構築. 奈良先端科学技術大学院大学情報科学研究科, 2009.

[6] 若杉祐太, 坂口嘉一, 三好健文, 吉瀬謙二. CoreSymphony アーキテクチャの高効率化. 東京工業大学大学院情報理工学研究科, 2009.

[7] 坂口嘉一, 松村貴之, 永塚智之, 吉瀬謙二. CoreSymphony 実現に向けたコアアーキテクチャの検討. 東京工業大学大学院情報理工学研究科, 2011.

[8] 永塚智之, 坂口嘉一, 松村貴之, 吉瀬謙二. CoreSymphony の実現に向けた高性能フロントエンドアーキテクチャ. 東京工業大学工学部情報工学科, 2011.

[9] 坂口嘉一. CoreSymphony アーキテクチャの実装に関する研究. 東京工業大学大学院情報理工学研究科, 2012.

[10] 上野貴廣. CoreSymphony における命令ステアリングの高性能化. 東京工業大学工学部情報工学科, 2012.

[11] Cadence. *Palladium XP*. http://www.cadence.co.jp/topics/2010/palladium_xp.html.

[12] Sohwa&Sophia Technologies. *DS-5*. http://www.ss-technologies.co.jp/service/arm/ds5/index.html.

[13] ALDEC. *HES-DVM*. http://www.aldec.com/jp/products/emulation/hes-dvm.

[14] 鈴木章浩, 及川修一. ARM アーキテクチャ用仮想マシンモニタの実装. 筑波大学, 2010.

[15] 高橋清隆, 柴山悦哉. 組込みシステム向けマルチコア・プロセッサのためのソフトウェア開発支援. 東京工業大学大学院情報理工学研究科, 2007.

[16] 小林優. ハードウェア記述言語の速習＆実践 入門 Verilog-HDL 記述. CQ 出版社, 2001.

[17] David Money Harris, Sarah L Harris. *Digital Design and Computer Architecture*. 翔泳社, 2009.

[18] Steve Furber. ARM プロセッサ. CQ 出版社, 1999.

[19] ARM Information Center. RealView Compilation Tools アセンブラガイド, 2014.

[20] 中野巧. VHDL によるマイクロプロセッサ設計入門. CQ 出版社, 2002.

[21] 三好健文. Inter face ソフトウェア技術者のための FPGA 入門. CQ 出版社, 2009.

付録

ソースコード 5.1: Himeno

```c
#include <stdio.h>
#define MIMAX 9
#define MJMAX 9
#define MKMAX 17

double second();
float jacobi();
void initmt();
double fflop(int,int,int);
double mflops(int,double,double);

static float p[MIMAX][MJMAX][MKMAX];
static float a[4][MIMAX][MJMAX][MKMAX],
             b[3][MIMAX][MJMAX][MKMAX],
             c[3][MIMAX][MJMAX][MKMAX];
static float bnd[MIMAX][MJMAX][MKMAX];
static float wrk1[MIMAX][MJMAX][MKMAX],
             wrk2[MIMAX][MJMAX][MKMAX];

static int imax, jmax, kmax;
static float omega;

int main(){
    int i,j,k,nn;
    float gosa;
    double cpu,cpu0,cpu1,flop,target;

    target= 60.0;
    omega= 0.8;
    imax = MIMAX-1;
    jmax = MJMAX-1;
    kmax = MKMAX-1;

    /*
     * Initializing matrixes
     */
    initmt();
    printf("mimax = %d mjmax = %d mkmax = %d\n",MIMAX, MJMAX, MKMAX);
    printf("imax = %d jmax = %d kmax =%d\n",imax,jmax,kmax);

```

```
41      nn= 10;
42      printf("␣Start␣rehearsal␣measurement␣process.\n");
43      printf("␣Measure␣the␣performance␣in␣%d␣times.\n\n",nn);
44
45      gosa= jacobi(nn);
46      flop= fflop(imax,jmax,kmax);
47
48      printf("measurement␣process␣end\n");
49      return (0);}
50
51  void initmt(){
52      int i,j,k;
53
54      for (i=0 ; i<MIMAX ; i++)
55        for (j=0 ; j<MJMAX ; j++)
56          for (k=0 ; k<MKMAX ; k++){
57            a[0][i][j][k]=0.0;
58            a[1][i][j][k]=0.0;
59            a[2][i][j][k]=0.0;
60            a[3][i][j][k]=0.0;
61            b[0][i][j][k]=0.0;
62            b[1][i][j][k]=0.0;
63            b[2][i][j][k]=0.0;
64            c[0][i][j][k]=0.0;
65            c[1][i][j][k]=0.0;
66            c[2][i][j][k]=0.0;
67            p[i][j][k]=0.0;
68            wrk1[i][j][k]=0.0;
69            bnd[i][j][k]=0.0;}
70
71      for (i=0 ; i<imax ; i++)
72        for (j=0 ; j<jmax ; j++)
73          for (k=0 ; k<kmax ; k++){
74            a[0][i][j][k]=1.0;
75            a[1][i][j][k]=1.0;
76            a[2][i][j][k]=1.0;
77            a[3][i][j][k]=1.0/6.0;
78            b[0][i][j][k]=0.0;
79            b[1][i][j][k]=0.0;
80            b[2][i][j][k]=0.0;
81            c[0][i][j][k]=1.0;
82            c[1][i][j][k]=1.0;
83            c[2][i][j][k]=1.0;
84            p[i][j][k]=(float)(i*i)/(float)((imax-1)*(imax-1));
85            wrk1[i][j][k]=0.0;
86            bnd[i][j][k]=1.0;
87          }}
88
89  float jacobi(int nn){
90      int i,j,k,n;
```

```
91      float gosa, s0, ss;
92
93      for (n=0 ; n<nn ; ++n){
94          gosa = 0.0;
95
96      for (i=1 ; i<imax−1 ; i++)
97          for (j=1 ; j<jmax−1 ; j++)
98              for (k=1 ; k<kmax−1 ; k++){
99                  s0 = a[0][i][j][k] ∗ p[i+1][j ][k ]
100                     + a[1][i][j][k] ∗ p[i ][j+1][k ]
101                     + a[2][i][j][k] ∗ p[i ][j ][k+1]
102                     + b[0][i][j][k] ∗ ( p[i+1][j+1][k ] − p[i+1][j−1][k ]
103                                       − p[i−1][j+1][k ] + p[i−1][j−1][k ] )
104                     + b[1][i][j][k] ∗ ( p[i ][j+1][k+1] − p[i ][j−1][k+1]
105                                       − p[i ][j+1][k−1] + p[i ][j−1][k−1] )
106                     + b[2][i][j][k] ∗ ( p[i+1][j ][k+1] − p[i−1][j ][k+1]
107                                       − p[i+1][j ][k−1] + p[i−1][j ][k−1] )
108                     + c[0][i][j][k] ∗ p[i−1][j ][k ]
109                     + c[1][i][j][k] ∗ p[i ][j−1][k ]
110                     + c[2][i][j][k] ∗ p[i ][j ][k−1]
111                     + wrk1[i][j][k];
112
113                 ss = ( s0 ∗ a[3][i][j][k] − p[i][j][k] ) ∗ bnd[i][j][k];
114
115                 gosa+= ss∗ss;
116                 wrk2[i][j][k] = p[i][j][k] + omega ∗ ss;
117             }
118     for (i=1 ; i<imax−1 ; ++i)
119         for (j=1 ; j<jmax−1 ; ++j)
120             for (k=1 ; k<kmax−1 ; ++k)
121                 p[i][j][k] = wrk2[i][j][k];
122     } /∗ end n loop ∗/
123     return(gosa);
124 }
125
126 double fflop(int mx,int my, int mz){
127     return((double)(mz−2)∗(double)(my−2)∗(double)(mx−2)∗34.0);
128 }
```