修士論文

演算レベル並列処理用

マルチ ALU プロセッサの設計と実現

氏　　名	:	境　　直樹
学籍番号	:	6162110047-4
指導教員	:	山崎　勝弘　教授
提 出 日	:	2013 年 2 月 14 日

立命館大学大学院　理工学研究科　創造理工学専攻

内容梗概

本論文では、本研究室で開発しているハード/ソフト協調学習システムを参考として、新たにマルチ ALU プロセッサ(MAP)システムを開発した。システムでは演算レベル並列性の検証を主体とした検証フローを提案する。このフローを実現するためのツールとして、MAP アセンブラ、MAP シミュレータの開発を行い、マルチ ALU プロセッサの設計と検証について述べる。

MAP の設計では、プロセッサの特徴を設定し、命令セットを定義し、仕様に沿ってプロセッサを設計する。複数 ALU を有するプロセッサを特徴として、その特徴を活かせるようプロセッサを設計していく。レジスタファイルはすべての ALU で共有することで、複数同時演算を実現する。また 2 ポート BRAM を用いることにより命令フェッチの短縮を図ることが出来た。プロセッサデバッガを用いることにより MAP の実装が可能となり、FPGA ボード上で動作検証を行えた。検証データは研究室の 4 回生によって作成され、シミュレータと実機上で実行結果が得ることができた。また、演算レベル並列性を行い、プログラム内の静的な並列性と動的な並列性も検証した。どちらも多くの並列性を有している結果となり、特に MAP プロセッサに組み込んだ連鎖演算機能では、演算実行時平均して 46%の連鎖演算を行っていることが判明した。

多数の検証プログラムにより、FPGA ボード上での実行および演算レベル並列性の検証を行えた。MAP システムの検証フローに沿って演算レベル並列性を検出することができ、MAP システムが検証用システムとして利用価値を評価することができた。

目次

内容梗概 i
1. はじめに 1
2. マルチ ALU プロセッサ（MAP）システム 2
  2.1 システムの構成と機能 2
  2.2 MAP アセンブラ 3
  2.3 MAP シミュレータ 4
  2.4 プロセッサデバッガ 5
3. MAP アーキテクチャ 6
  3.1 命令セットアーキテクチャ 6
  3.2 MAP データパス 9
  3.3 MAP プログラミング 10
  3.4 並列演算と連鎖演算 11
4. 2ALUMAP の設計と検証 14
  4.1 2ALUMAP の設計 14
  4.2 論理シミュレーション 18
5. FPGA ボード上への実装と検証 25
  5.1 プロセッサデバッガとプロセッサモニタ 25
  5.2 各種プログラムの動作確認 27
6 並列性の評価と考察 33
  6.1 並列性の評価 33
  6.2 考察 34
7 おわりに 35

図目次

図 1：MAP システムの構成 2
図 2：MAP シミュレータの動作 4
図 3：MAP 命令セットアーキテクチャ 6
図 4：MAP のデータパス 9
図 5：MAP プログラム例 11
図 6：並列演算 12
図 7：連鎖演算 13
図 8：2ALUMAP の構成 14
図 9：レジスタファイルの入出力 15
図 10：レジスタファイルのデータ出力 16
図 11：レジスタファイルのデータ格納動作 16
図 12：PPU の構成 17
図 13：2 ポート BRAM 18
図 14：レジスタのシミュレーション波形 19
図 15：レジスタ内変化 19
図 16：並列実行判定のシミュレーション 20
図 17：連鎖演算判定のシミュレーション 21
図 18：1 命令読み出し 21
図 19：2 命令同時読み出し 22
図 20：プロセッサデバッガとプロセッサモニタの構成 26

表目次

表 1：MAP の命令一覧 8
表 2：命令別並列実行判定 20
表 3：単一実行結果 23
表 4：並列実行結果 23
表 5：連鎖演算結果 24
表 6：ΣN のシミュレーション結果 24
表 7：デバッグコマンド 25
表 8：MAP の設計規模と最大遅延 26
表 9：正数同士の乗算の実行結果 27
表 10：正数同士の除算の実行結果 28
表 11：ΣN の実行結果 28
表 12：N！の実行結果 28

表 13：素数の実行結果 29
表 14：最大公約数の実行結果 29
表 15：三角形の判別の実行結果 30
表 16：4 行 4 列の乗算の実行結果 30
表 17：BCD 加算の実行結果 31
表 18：2 点を通る直線の実行結果 31
表 19：バイトニックソートの実行結果 32
表 20：Booth アルゴリズムによる乗算の実行結果 32
表 21：2ALU の並列性 33

1. はじめに

半導体製造技術の発展により、LSI の小型化、高速化が可能となり、低消費電力化が進められてきた。近年では、GPU やマルチコアプロセッサなどのプロセッサ性能が急速に向上している。LSI 開発技術はハードウェアとソフトウェアに密接な関係があり、ハードとソフトの両方の知識が求められる。

我々はプロセッサ設計を中心として、ハードとソフト両方の知識を習得するためのハード/ソフト協調学習システム(HSCS)の研究を進めてきた[5]～[14]。HSCS を用いて最初に設計されたプロセッサが MONI[5]である。MONI をこのシステムの教育用プロセッサにし、シングルサイクル、マルチサイクル、パイプライン、スーパースカラプロセッサを設計した。次に、MONI を拡張した SOAR[9]、SARIS[13]を定義し、それらの設計を行ってきた。さらに、近年プロセッサ設計によく利用される ARM ライクプロセッサも設計し、プロセッサ設計時間などの視点から HSCS を評価した。次に、命令セットの異なるプロセッサの設計と FPGA ボード上での実装を可能とするために、プロセッサ設計支援ツールも作成してきた。ユーザが新しい命令を作成するための命令セット定義ツール、それに対応した汎用シミュレータと汎用アセンブラ、FPGA ボード上での実行を支援するプロセッサモニタとプロセッサデバッガも開発した。

本研究では、HSCS での研究をベースとして、演算レベル並列性の検証が可能な複数の ALU を有するマルチ ALU プロセッサ(MAP)の設計を行い、FPGA ボード上での動作検証と評価を行うことを目的とする。MAP ではデータ依存のない命令の並列演算と、データ依存のある命令の連鎖演算を可能として、高速化を図る。演算レベル並列性はソフトウェアとハードウェアで評価を行う。ソフトウェアでは、アセンブラによって静的な並列性を検出し、ハードウェアでは、プロセッサ内で動的な並列性を検出する。FPGA ボードで実行するために、プロセッサデバッガに MAP を結合させて実装する。

本研究では 2 個の ALU を有する MAP を設計し、論理シミュレーションを行い、FPGA ボード上で実行し、ソフトウェアとハードウェアで演算レベル並列性の検証を行う。

本論文では第 2 章で MAP システムについての概要、及び MAP アセンブラ[16]と MAP シミュレータ[17]について説明する。第 3 章で MAP プロセッサの命令セットアーキテクチャとデータパス、MAP プログラミングの記述方法の詳細を説明する。第 4 章で設計したプロセッサの性能と論理シミュレーションについて説明する。第 5 章では FPGA ボードに実装する際に使用するプロセッサデバッガとプロセッサモニタについて説明し、ボード上において各種プログラムの動作確認について説明する。そして、第 6 章では演算レベル並列性の評価と考察について述べる。

2. マルチ ALU プロセッサ（MAP）システム

2.1 システムの構成と機能

(1) システムの構成

演算レベル並列性を検証するために MAP システムを構築した。図 1 に MAP システムの構成を示す。

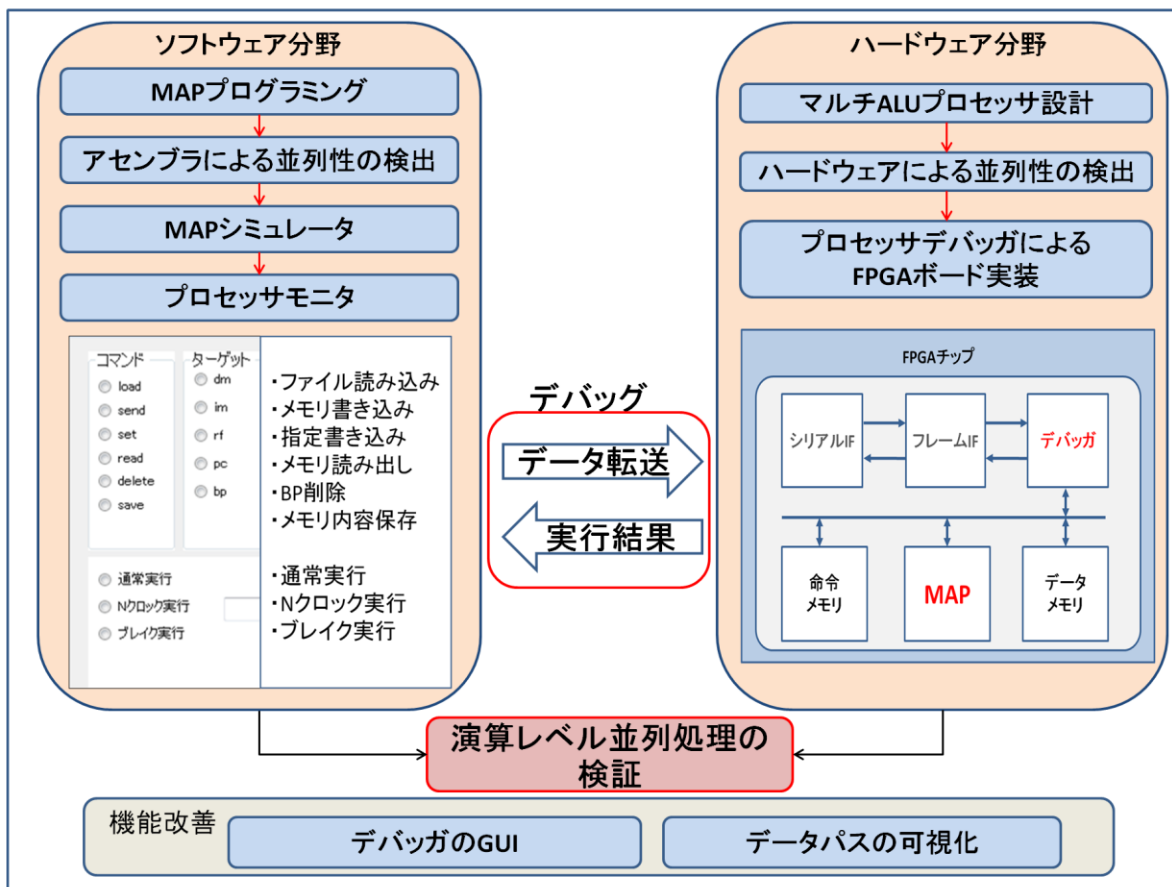

図 1：MAP システムの構成

MAP システムは、ソフトウェア分野とハードウェア分野に分かれている。ソフトウェア分野では、MAP プログラミングがあり、ここで作成したアセンブリプログラムをアセンブラに送る。アセンブラで生成した機械語を MAP シミュレータで実行する。シミュレーションで問題がなかったらプロセッサモニタで FPGA ボードにデータ転送し、デバッグする流れとなる。ハードウェア分野では MAP 設計を行い、プロセッサデバッガに MAP を組み込み、FPGA ボード上に実装する。プロセッサのデバッグはプロセッサデバッガ・モニタを使用する。プロセッサモニタから送られたコマンドを元にプログラムを実行し、実行結果をプロセッサモニタで受信し、PC の画面上に表示する。

(2)　MAP の設計と FPGA ボード上での実装

MAP の設計は Xilinx 社の ISE Design Suite 13.2 のツールを用いて Verilog-HDL で記述し、プロセッサを設計する。MAP を FPGA ボードに実装する際、プロセッサデバッガに接続する必要がある。また、パソコン操作にはプロセッサモニタを用いる。プロセッサモニタのユーザーインターフェースを GUI に変更することにより、ボード上でのデバッグ操作を簡略化して操作しやすくする。

(3)　MAP システムの目的

MAP システムでは、複数 ALU による演算レベル並列性の有効性の検証と並列プロセッサの設計能力の修得を目的としている。演算レベル並列性の検証では、MAP システムにおいて、二通りの検出方法をとる。一つ目はアセンブラによる静的並列性の検出である。これは、プログラムを上から下まで確認した際、プログラム自体にある並列性の可能性を見るものである。二つ目はハードウェアによる動的並列性の検証である。これは、実際にプロセッサを動作させた際に並列性があるかどうかを判断するものである。静的、動的の並列性を検出し、複数 ALU による演算レベル並列性を検証していく。

並列プロセッサの設計能力の修得では、ソフトウェア側で MAP のプログラミング、ハードウェア側で MAP の設計を行う。ユーザはプログラミングを通して MAP のアセンブリ言語を学んでいく。プログラムはアセンブラを通して、正しく記述できているか確認し機械語を生成する。生成した機械語は、シミュレータに通し、正しくプログラミングできているか確認し、プログラムのコーディングとデバッグを反復することによりアセンブリ言語を学習していく。また、シミュレータを可視化することにより命令ごとのデータの流れを理解する。MAP のハードウェア設計では、MAP 命令セットアーキテクチャと設計思想に沿ったプロセッサを設計する。ユーザは、あらかじめ用意されたモジュール構成を用いる。重要な部分または特徴ある部分を一から設計していき、プロセッサの構成と特徴を理解していく形で学習してもらう。設計したモジュール、プロセッサは論理シミュレーションで正しく動作できるかを確認下上で FPGA ボードに実装させ、プロセッサデバッガ・プロセッサモニタを用いて検証を行う。様々なプログラムを作成し実ボードで動作させることによりソフトウェアとハードウェアの学習を行う。

2.2　MAP アセンブラ[16]

MAP アセンブラはプログラマが記述したプログラムを機械語に変換し、プログラムにおける静的並列性を検出するアセンブラである。並列演算、連鎖演算、単一実行の検出を行い、機械語命令に変換する。また、検出時に演算順序や LD、ST 命令の順序を入れかえて、並列演算と連鎖演算の可能性を高められるように最適化することを検討している。ハードウェアによる並列演算の検出方法と、アセンブラによる検出方法を比較し評価を行う。

MAP アセンブラはアセンブリ言語で記述されたテキストファイルを入力とし、字句解析と構文解析を行い、機械語プログラムを出力する。まず Flex で字句解析を行い、ファイルのアセンブリ命令をトークンの並びに変換する。次に、Bison で構文解析を行い、トークン間の意味を解析していき、それに従って機械語を生成する。

2.3　MAP シミュレータ[17]

図 2 に MAP シミュレータの動作を示す。

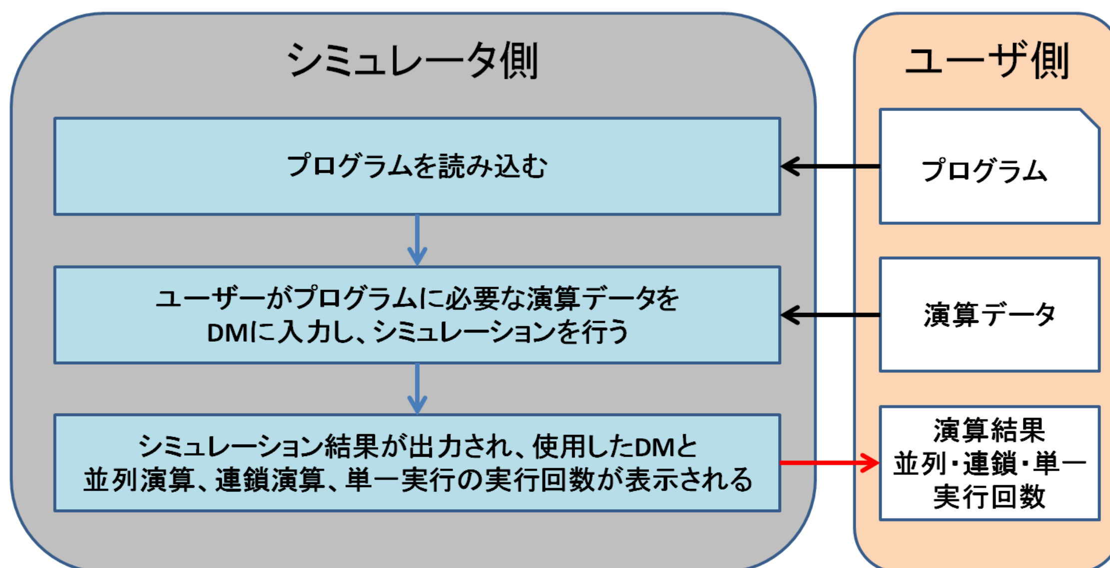

図 2：MAP シミュレータの動作

MAP シミュレータはプログラマが記述したアセンブリプログラムのデバッグ、および命令実行時の並列・連鎖性を調べる。MAP の複数 ALU による並列・連鎖演算、単一実行判別の有効性を示す役割を持つ。現在作成されたものは 2ALU のシミュレータである。図 2 に MAP シミュレータ動作を示す。ユーザが記述したプログラムをアセンブラに通し機械語変換したものを入力とする。入力後、ユーザはプログラムに必要な値を DM に設定する。次にファイルオープンを行い、IM にすべての機械語を格納する。命令実行時は IM から 1 命令分のコードを取り出し、命令の OP を解析する。取り出した命令の命令形式を判別し、それぞれの演算別に分岐させる。次にそれぞれの分岐先で命令を解析し、レジスタ、即値、シフト量、ファンクションを判別し実行する。その後、結果として DM や RF などに値を保存し、それを HALT まで演算を実行し続ける。最終的な演算結果が DM に格納される。

MAP シミュレータの並列・連鎖演算と単一実行判定は、２つの命令を比較し並列・連鎖性を判定する。アドレス番地の値の小さい方を上位命令、大きい方を下位命令とする。まず、上位命令の判別を行う。上位命令のオペコードを解析し、終了命令もしくは分岐命令であるかを判別し、もし終了命令や分岐命令であった場合は単一実行とする。上位

と下位にデータ依存がある場合は上位の演算結果を下位でも使用できる連鎖演算とする。上位と下位依存関係がない場合は同時実行とみなし並列演算とする。このようにして並列・連鎖演算と単一実行判定を行う。

2.4　プロセッサデバッガ[11],[12],[13]

FPGA ボードに実装する際に、HSCS で開発されたプロセッサデバッガ・モニタを用いる。プロセッサデバッガは、HSCS を利用して設計したプロセッサを、実機上で動作検証するためのハードウェア IP である。本モジュールと接続して論理合成をかけることで様々なプロセッサが FPGA に実装でき、ホスト PC と通信することで実行が可能である。プロセッサデバッガの利点は、FPGA ボードの LED や LCD、スイッチを使用せず、パソコン操作によって実機のデバッグを進められることである。HSCS 上で学生が作成したプロセッサを FPGA ボードに搭載するために欠かせない周辺モジュールを提供する。FPGA やホスト PC の環境によって発生する問題ではなく、プロセッサに関わる問題に集中し、自作プロセッサの実装と動作検証に専念できる。

3. MAP アーキテクチャ

3.1 命令セットアーキテクチャ

図 3 に MAP 命令セットアーキテクチャを示す。MAP には、Register 形式（R 形式)、Immediate 形式（I 形式)、Long 形式（L 形式)、Jump 形式（J 形式）の 4 つの命令形式を用意した。R 形式にはレジスタ間の演算を行う命令、レジスタ間の比較、シフト命令を定義する。I 形式にはレジスタ値と即値演算を行う命令、レジスタ値と即値比較を行う命令、LDHI、LDLI など命令を定義する。L 形式には条件分岐命令とメモリ・レジスタへのデータ転送命令を定義する。J 形式には無条件分岐命令の JUMP、プログラム終了命令となる HALT を定義する。L 形式と J 形式では DM へのアクセスと、PC に分岐する範囲を多く取ることで大規模な計算を行うことができる。

命令語長		32						命令種類
		6	5	5	5	5	6	
命令形式	R形式	Op	Rs	Rt	Rd	Shamt	Fn	レジスタ同士による演算
	I形式	Op	Rs	Rd	imm			レジスタと即値による演算
	L形式	Op	Rs	Rd	address/immediate			メモリアクセスや分岐命令
	J形式	Op	address					JUMP、HALT命令

図 3：MAP 命令セットアーキテクチャ

MAP 内では 1ALU を 32 ビットの命令で動作させ、演算データ幅も 32 ビット用意している。Op で命令を R 形式、I 形式、L 形式、J 形式と判断する。また R 形式においては、フィールド Fn で命令を詳細に識別しているため、Op1 つに対して 64 種類まで拡張が行える。I 形式と L 形式のフィールド imm で 16 ビット用意されており、－32468～32467 の値が設定できる。L 形式と J 形式では、プログラムカウンタやデータメモリにアクセスするフィールドを広く確保するために大きく長さをもつ。以下にアーキテクチャの各形式と各フィールドの意味を示す。

各フィールド

Op … オペコード、命令を識別

Fn … ファンクション、R 形式命令を詳細に識別

Rs … 演算レジスタ

Rt … 演算レジスタ

Rd … 演算結果を格納するレジスタ

Shamt　…　シフト量
Imm　…　即値
address　…　アドレス

アドレスの指定方式は、R 形式ではレジスタ直接、I 形式ではレジスタ即値、L 形式でメモリアクセスはベース相対アドレス、分岐命令は絶対アドレス、J 形式は絶対アドレスである。

表 1 に MAP の命令一覧を示す。

表 1：MAP の命令一覧

	命令	Op	Rs/Rt/Rd			Shamt	Fn	動作
R	NOP	000000	XXXX				000000	No operetion
	ADD	000001	Rs	Rt	Rd	X	000000	Rd = Rs+Rt
	SUB	000001	Rs	Rt	Rd	X	000001	Rd = Rs−Rt
	AND	000001	Rs	Rt	Rd	X	000010	Rd = Rs&Rt
	OR	000001	Rs	Rt	Rd	X	000011	Rd = Rs|Rt
	XOR	000001	Rs	Rt	Rd	X	000100	Rd = Rs^Rt
	NOT	000001	Rs	X	Rd	X	000101	Rd = !(Rs)
	SLT	000010	Rs	Rt	Rd	X	000000	(if Rs < Rt) Rd = 1
	SGT	000010	Rs	Rt	Rd	X	000001	(if Rs > Rt) Rd = 1
	SLE	000010	Rs	Rt	Rd	X	000010	(if Rs ≦ Rt) Rd = 1
	SGE	000010	Rs	Rt	Rd	X	000011	(if Rs ≧ Rt) Rd = 1
	SEQ	000010	Rs	Rt	Rd	X	000100	(if Rs = Rt) Rd = 1
	SNE	000010	Rs	Rt	Rd	X	000101	(if Rs ≠ Rt) Rd = 1
	SLL	000011	Rs	X	Rd	Shamt	000000	Rd = Rs << Shamt
	SRL	000011	Rs	X	Rd	Shamt	000001	Rd = Rs >> Shamt
	SRA	000011	Rs	X	Rd	Shamt	000010	Rd = Rs >>> Shamt
	JR	000100	Rs	X			000000	PC = Rs
I	ADDI	001000	Rs	Rd	Imm(16bit)			Rd = Rs+imm
	SUBI	001001	Rs	Rd	imm			Rd = Rs−imm
	ANDI	001010	Rs	Rd	imm			Rd = Rs&imm
	ORI	001011	Rs	Rd	imm			Rd = Rs|imm
	XORI	001100	Rs	Rd	imm			Rd = Rs^imm
	SLTI	001101	Rs	Rd	imm			(if Rs < imm) Rd = 1
	SGTI	001110	Rs	Rd	imm			(if Rs > imm) Rd = 1
	SLEI	001111	Rs	Rd	imm			(if Rs ≦ imm) Rd = 1
	SGEI	010000	Rs	Rd	imm			(if Rs ≧ imm) Rd = 1
	SEQI	010001	Rs	Rd	imm			(if Rs = imm) Rd = 1
	SNEI	010010	Rs	Rd	imm			(if Rs ≠ imm) Rd = 1
	LDHI	010011	Rs	Rd	imm			Rd = {imm,Rs[15:0]}
	LDLI	010100	Rs	Rd	imm			Rd = {Rs[31:16],imm}
L	BEQZ	100000	Rs	X	imm			(if Rs = 0) PC = imm
	BNEZ	100001	Rs	X	imm			(if Rs ≠ 0) PC = imm
	LD	100010	Rs	Rd	imm			Rd = DM[Rs+imm]
	ST	100011	Rs	Rd	imm			DM[Rs+imm] = Rd
	JAL	100100	X	Rd	imm			PC = imm:Rd = PC+4
J	JUMP	111110	Imm(26bit)					PC = imm
	HALT	111111	X					exit

X は DON'T CARE を意味しており使用しないビット分だけ 0 が入る。

3.2 MAP データパス

MAP は複数 ALU を有する 32 ビットのプロセッサである。院生が設計したプロセッサを、FPGA チップ上で実際に動作させ、大学教育でプロセッサ設計能力を習得することが目標である。MAP のデータパスを図 4 に示す。命令メモリとデータメモリが分離したハーバードアーキテクチャである。MAP の特徴は以下の 5 点に要約できる。

① ALU 数は 2、4、8 … 複数 ALU
② レジスタ共有 … 全 ALU による共有
③ PPU（Parallel Processing Unit） … 並列連鎖単一処理判定部
④ ハーバードアーキテクチャ … 命令メモリとデータメモリの分離
⑤ バイトアドレス方式 … バイト毎のアドレッシング方法

例えば、レジスタ間演算命令では、命令メモリ(IM)から制御ユニット(CU)にオペコードとファンクションコードを送る。また IM から命令分のオペランドのアドレスと、演算結果を格納するアドレスをレジスタファイル(RF)に出力する。ALU は、CU から演算命令の種類を受け取り、演算データの演算を行い、指定したアドレスに演算結果を格納する。PC は加算器によって 8 加算する。

図 4 : MAP のデータパス

3.3　MAP プログラミング

(1)　MAP プログラミング

MAP には 37 個の命令があり、演算、ロード、ストア、分岐などを順に並べて逐次プログラムをユーザが記述する。プログラマは並列演算や連鎖演算を考慮せずに 1 演算ずつ記述していく。MAP 並列実行には並列演算と、ALU 間で依存関係があり連鎖させて実行できる連鎖演算がある。MAP は実行時に 2 演算ずつフェッチして並列、連鎖、単一のみを判定する。

(2)　擬似命令

MAP の 37 個の命令に、擬似命令として COPY と CLEAR の二種類を用意する。

COPY	$1	$2	…	$1 に$2 の内容をコピー。	ADDI	$1	$2	0
CLEAR	$1		…	$1 の内容を初期化。	SUB	$1	$1	$1

これにより、プログラマは理解しやすく、記述も容易にできるようになる。

(3)　命令記述例

表 1 の命令を用いて例のように記述していく。

例	R 形式命令	加算	ADD	$1	$1	$1
	I 形式命令	減算	SUBI	$1	$1	1
	L 形式命令	条件分岐	BNEZ	$1	LOOP	
	L 形式命令	LD	LD	$1	0 [$0]	
	J 形式命令	HALT	HALT			
	L 形式命令	JAL	JAL	$1	WORK	
	R 形式命令	JR	JR	$1		

命令の種類に応じて、格納及び呼び出しレジスタ、即値、ジャンプラベルを記述する。これらを組み合わせることによりプログラムが構成される。

(4)　メモリアクセス

基本的に$0 をベースレジスタとして扱う。

例　LD　$1　8 [$0]

命令“LD　$1　8 [$0]”では、命令のアドレス部は「8」なので、プログラムの先頭を 0 番地とすると、8 番地のところに求めるデータが格納されていることになる。また、プログラムの先頭アドレス、すなわちベースレジスタ 0 に格納されている値は 0 番地になっています。したがって、求める値が可能されている有効アドレスは、ベースレジスタ 0 の値 ＋ 命令で指定されたアドレス部の値 ＝ 8 となります。

(5)　プログラム例

図 5 に MAP プログラム例を示す。

```
        SUB   $0  $0      $0  } 連鎖演算
        LD    $1  0[$0]
        SUB   $3  $3      $3  } 並列演算
        LD    $2  4[$0]
LOOP:   SUBI  $2  $2      1   } 連鎖演算
        SGTI  $4  $2      0
        ADD   $3  $3      $1  } 並列演算
        BNEZ  $4  LOOP
        ST    $3  8[$0]       ・・・単一実行
        HALT
```

図 5 : MAP プログラム例

このプログラムは整数同士による掛け算 A×B のプログラムである。アルゴリズムでは B の値を 1 ずつ引いていき、引いた回数だけ、A を足している。例では、並列演算が 2 回、連鎖演算が 2 回、単一実行が 1 回の動作が検出できる。これは実際にプログラムを実行したのではなく、このプログラムが演算レベル並列性を含んでいるのか表したものである。実際に動作を行うならばこのプログラムは条件分岐により LOOP 内で複数回演算を行う。LOOP 内に並列演算、連鎖演算を含むため、B の値が大きいほど並列演算、連鎖演算の演算回数が増加し、演算レベル並列性が高まる。

3.4　並列演算と連鎖演算

(1)　並列演算

ALU 並列演算は、依存関係のない複数の演算を、1 命令サイクルで同時実行する。命令メモリからフェッチしてきた命令の判別を行い、並列実行できるか判断する。また、命令間のオペランドに依存関係がないか判断し、なければ並列演算を行う。並列演算のデータパスを図 6 に示す。

例　　足し算　A=B+C

　　　足し算　D=E+F

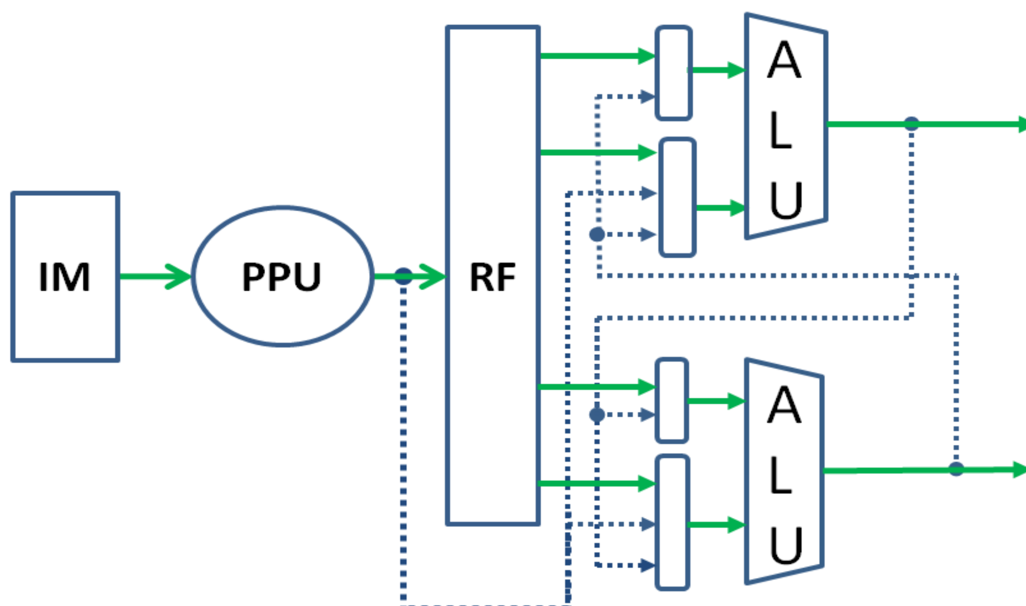

図 6：並列演算

例の場合、図 6 のように IM から二つの足し算の命令をフェッチする。フェッチ後、PPU に命令が流れる。PPU に入力されると命令の判別を行い、足し算と足し算なので並列実行できると判断する。また、それぞれの足し算で使用するオペランドが重複していないため、依存関係がないと判断でき、並列演算できる。RF から演算に使用するデータを取り出し、各 ALU に流す。各 ALU では独立して演算を行うことで 2 命令を同時に実行し、演算終了後レジスタに格納する。

(2)　連鎖演算

ALU 連鎖演算では、ある ALU での演算結果を他の ALU の入力とすることにより、依存関係のある 2 つの演算を 1 命令サイクルで実行する。連鎖演算のデータパスを図 7 に示す。

例　　足し算　A=B+C

　　　引き算　D=A-E

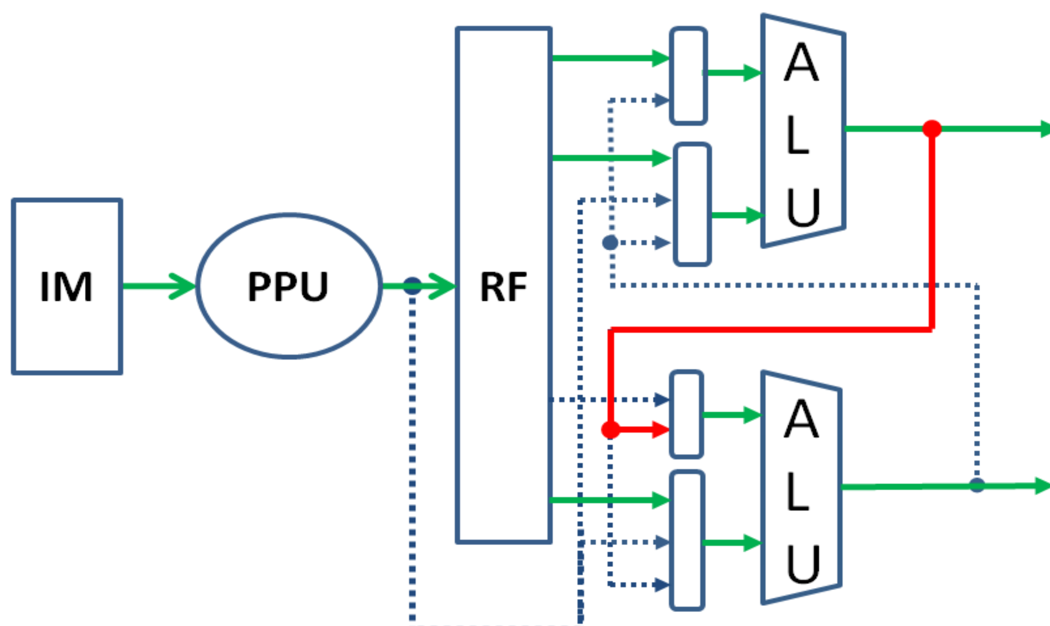

図 7：連鎖演算

2 命令間で依存関係がある場合、図 7 のように上の ALU の結果を下の ALU の演算に組み込むようにしている。上下の ALU の結果をレジスタファイルに格納する。

例の場合、図 7 のように IM から二つの足し算の命令をフェッチする。フェッチ後、PPU に命令が流れる。PPU に入力されると命令の判別を行い、足し算同士なので並列実行できると判断する。また例において、足し算で格納するオペランド A と引き算で使用するオペランド A が重複しているため、依存関係があると判断でき、連鎖演算できる。RF から演算に使用するデータを取り出し、上の ALU にデータを流す。上の ALU で演算を行い、演算結果を下の ALU に流す。下の ALU 演算に必要なデータを用いることで 2 命令を同時に実行する。実行後上下の ALU の結果をレジスタファイルに格納する。この一連の流れを一サイクルで行うことを連鎖演算とする。

4. 2ALUMAP の設計と検証

4.1　2ALUMAP の設計

複数 ALU における演算レベル並列性をハードウェアで検証するため Verilog-HDL で設計していく。演算レベル並列性を確かめるため、今回の設計は 2 個の ALU を有する MAP の設計を行う。図 8 に今回設計する 2ALUMAP の構成を示す。MAP の設計構成は、機能ごとにモジュールを作成し、設計した各モジュールをトップモジュールで統合されている。そのため各モジュールでの動作確認が容易となり、全体構築及びデバッグが容易になる。作成するモジュールは MAP のトップモジュール及び、プログラムカウンタ、レジスタファイル、2 個の ALU、コントロールユニット、符号拡張、Parallel Processing Unit(PPU)、14 個のマルチプレクサ、と ISE CORE Generator で生成した IM と DM である。

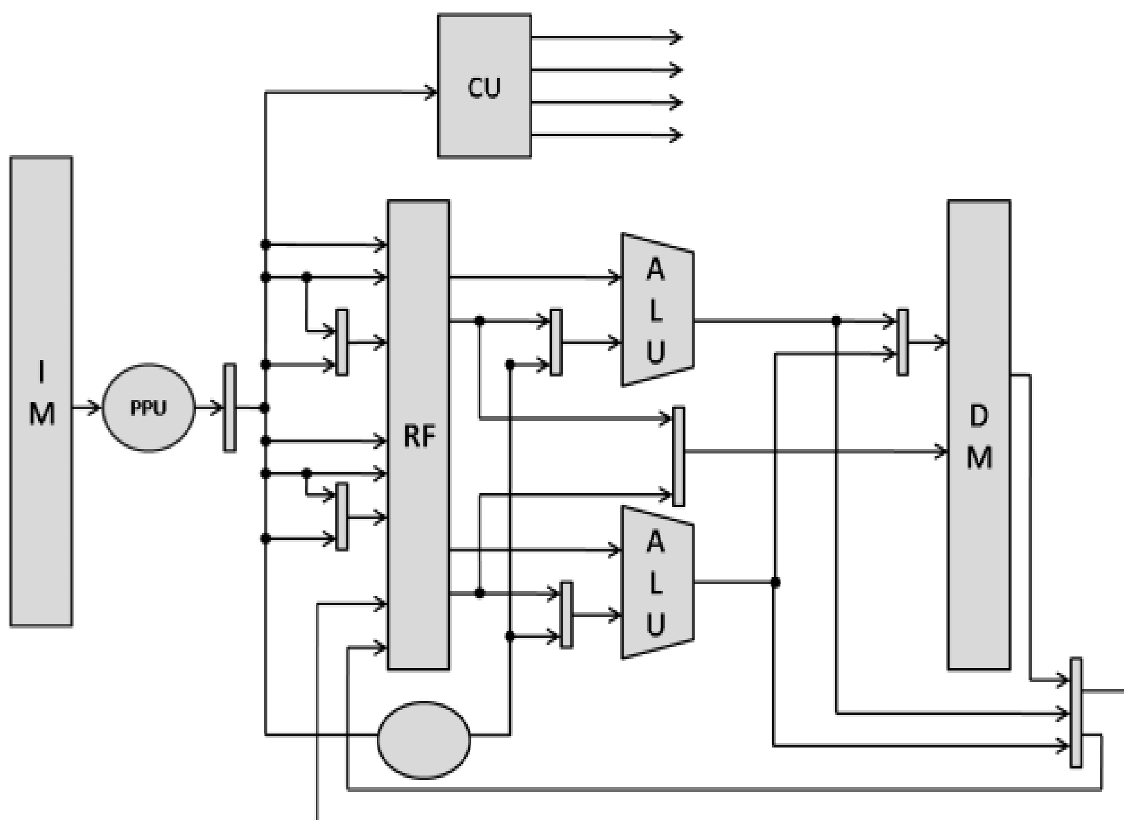

図 8：2ALUMAP の構成

3.2 に記述している MAP の特徴の中でも、②のレジスタ共有、③の PPU は MAP のプロセッサ設計においても重要な部分である。またメモリ設計において 2 ポート BRAM を用いて設計する。そのため、これらの構成と動作原理について以下に記述する。

(1) レジスタファイル

レジスタファイルは 2 個の ALU で共有する。そのため、ALU ごとの入出力が必要である。図 9 にレジスタファイルの入出力を示す。入力の種類は、読み出し用アドレスポートの RF_ADDR00 、RF_ADDR10、RF_ADDR01、RF_ADDR11、書き込み用アドレスポートの RF_INADDR0、RF_INADDR1、書き込み用データポートの RF_INDATA0、RF_INDATA1、ライトイネーブルの RF_WE0、RF_WE1、クロックの CLK、リセットの XRST である。出力は RF_OUT00、RF_OUT10、RF_OUT01、RF_OUT11 の四つであり、レジスタ内のデータを出力する。

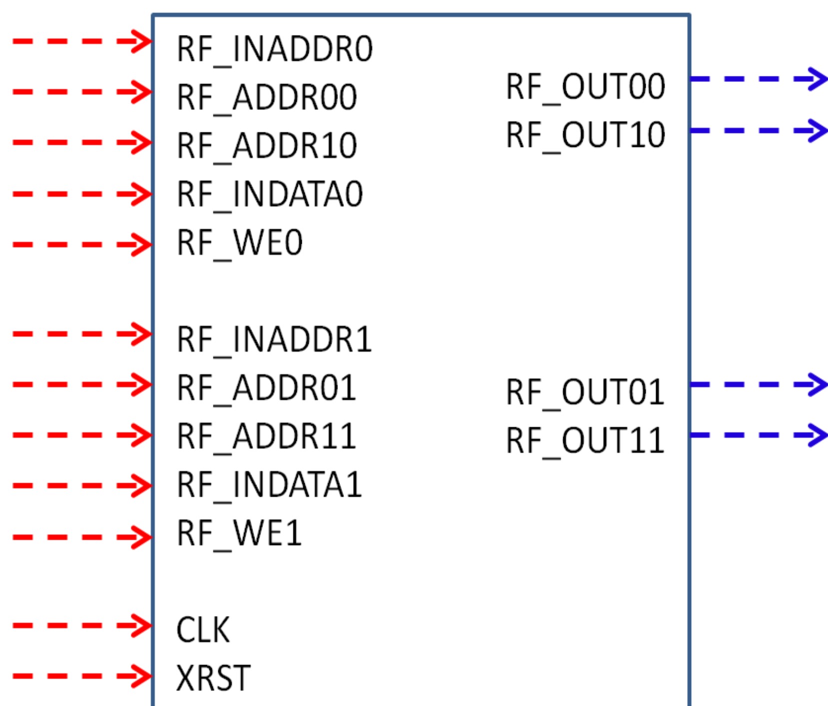

図 9：レジスタファイルの入出力

MAP において複数 ALU が一つのレジスタファイルを共有しているのは演算において重要な項目である。このレジスタファイルでは 4 ポート同時読み出しと 2 ポート同時書き込みが行われる。図 10 にレジスタファイルのデータ出力動作を示し、図 11 にレジスタファイルのデータ格納動作を示す。

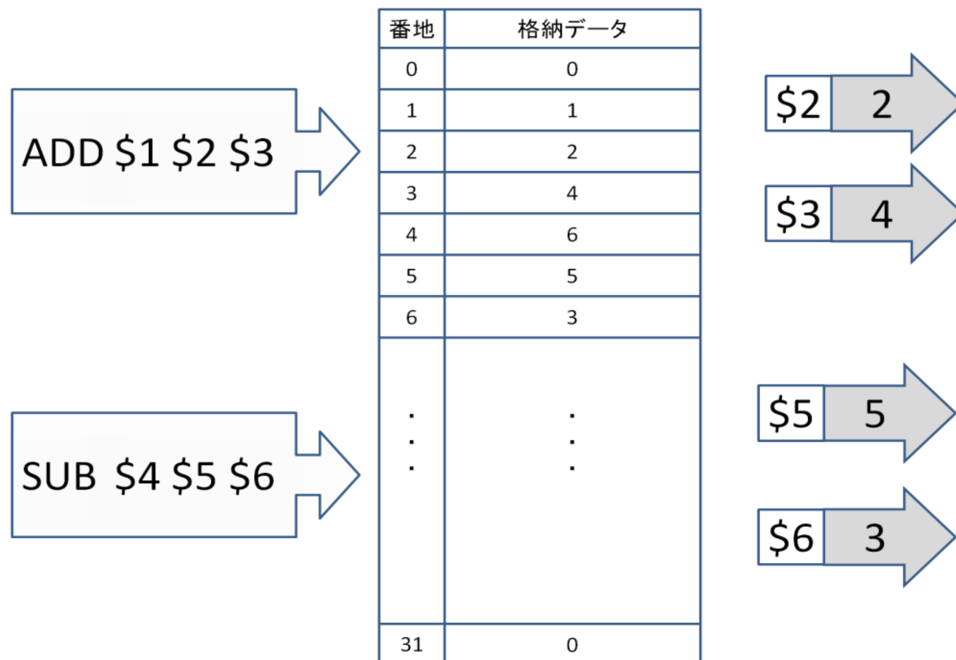

図 10：レジスタファイルのデータ出力動作

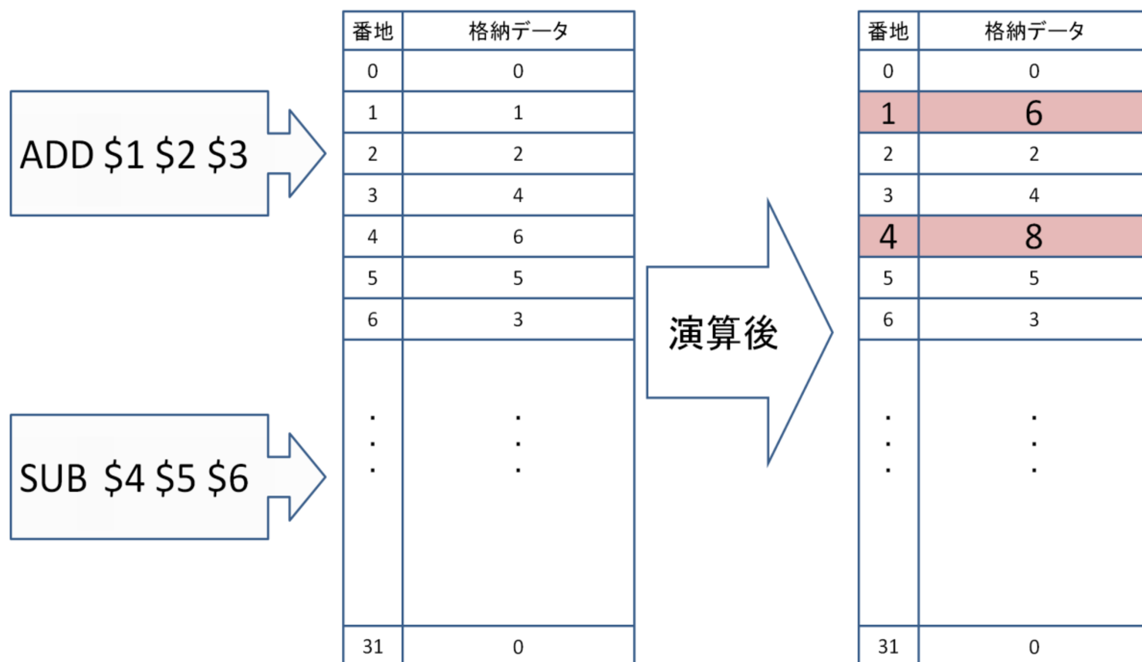

図 11：レジスタファイルのデータ格納動作

図 10 のようにレジスタファイルは命令に応じて必要なデータを出力する。また、出力ポートはどれも独立しており、複数ポートが同じ読み出しアドレスを指定しても、動作に影響は出ない。演算が終了し、データが格納されるときは元のデータを上書きする。図 11 のように、異なる書き込みアドレスを指定しても動作に影響はない。しかし、同じ

書き込みアドレスの場合は下の命令の結果を優先して書き込む仕様である。これは、命令の順序により、下の命令のほうが新しい命令と見なしているためである。このようにしてレジスタファイルの動作を制御している。ALU を増やす場合、レジスタファイルのポート数を増やす必要があり、書き込み動作の制御も増やす必要がある。

⑵ PPU

PPU では並列演算、連鎖演算、単一実行を検出している。図 12 に PPU の構成を示す。

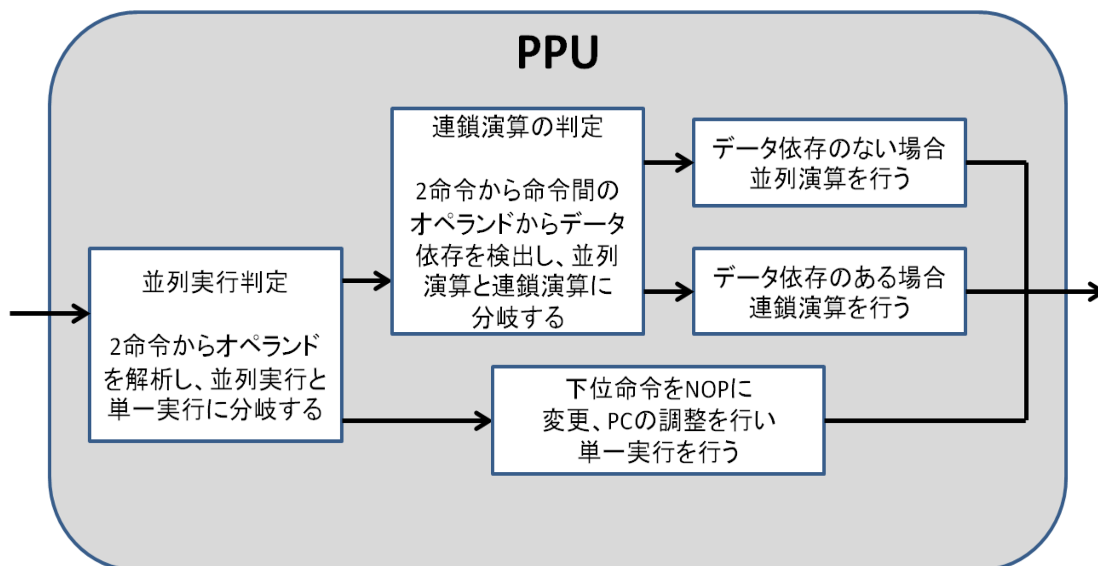

図 12 : PPU の構成

若いメモリアドレスから取り出したものを上位命令、もう一方を下位命令とする。PPU 入力は取り出した 2 命令が入る。並列実行判定で、並列実行にできない場合、単一実行となるので一方の命令を NOP に変更し、動作しないようにする。上位命令で条件分岐や無条件分岐命令の場合、下位命令の演算により、プログラムが誤動作を起こす場合があるため、単一実行にする必要がある。また、上位命令がロード命令で、下位命令にデータ依存がある場合も単一実行にする。単一実行の場合、NOP 命令は実行されないため、プログラムカウンタに一命令分の 4 だけ加算されるようにする。これらより、データ依存、制御依存を緩和する。並列実行できるならばそのまま連鎖演算の判定に進む。

連鎖演算の判定では、上位命令と下位命令の間にデータ依存があるかを判定する。連鎖演算判定で命令内のレジスタ指定フィールドから、一方の ALU の演算結果を他方の ALU で演算を行うか判断する。つまり、演算に使われるレジスタが格納するレジスタと等しければ連鎖演算が可能と判断する。連鎖演算が可能であると判断したとき、ALU の前にあるマルチプレクサ(MUX)で ALU の結果を演算に使用できるように指定する。データ依存がない場合、並列演算とみなし、命令を同時に実行する。

(3) 2 ポート BRAM

FPGA ボードに実装する際、プロセッサの命令メモリとデータメモリは BRAM を用いる。MAP を動作する際、一命令ずつ取り出すと大幅に時間がかかる。そのため、ポート数を増やすことにより、速度向上を図る。BRAM の設計は ISE Core Generator を使用して、生成する。図 13 に 2 ポート BRAM を示す。

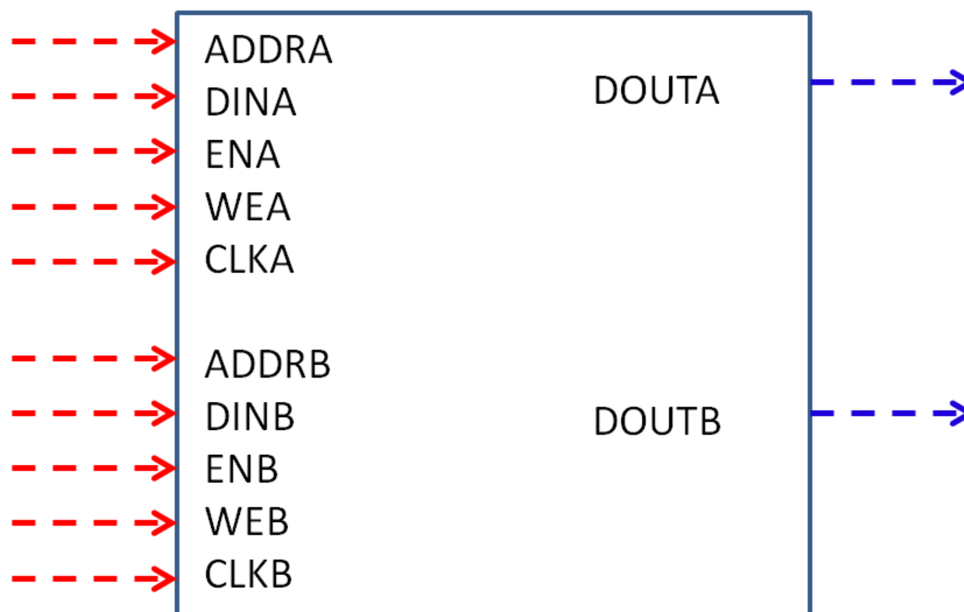

図 13 : 2 ポート BRAM

入出力は、読み出し書き込みアドレス(ADDRA、ADDRB)、書き込みデータ(DINA、DINB)、イネーブル信号(ENA、ENB)、ライトイネーブル信号(WEA、WEB)、クロック信号(CLKA、CLKB)、読み出しデータ(DOUTA、DOUTB)の 6 種類である。ポートごとに信号が独立しており、一方が書き込み動作でもう一方が読み出し動作でも正常に実行できる。また、クロックもポートごとに割り振られているため、タイミングをずらした制御も可能である。

4.2 論理シミュレーション

Verilog-HDL で設計したものを ISE の Isim を用いてシミュレーションを行う。

(1) レジスタファイル

アドレス、データを設定したテキストファイルをレジスタに読み込み、シミュレーション実行する。図 14 にレジスタのシミュレーション波形、図 15 にレジスタ内変化を示す。

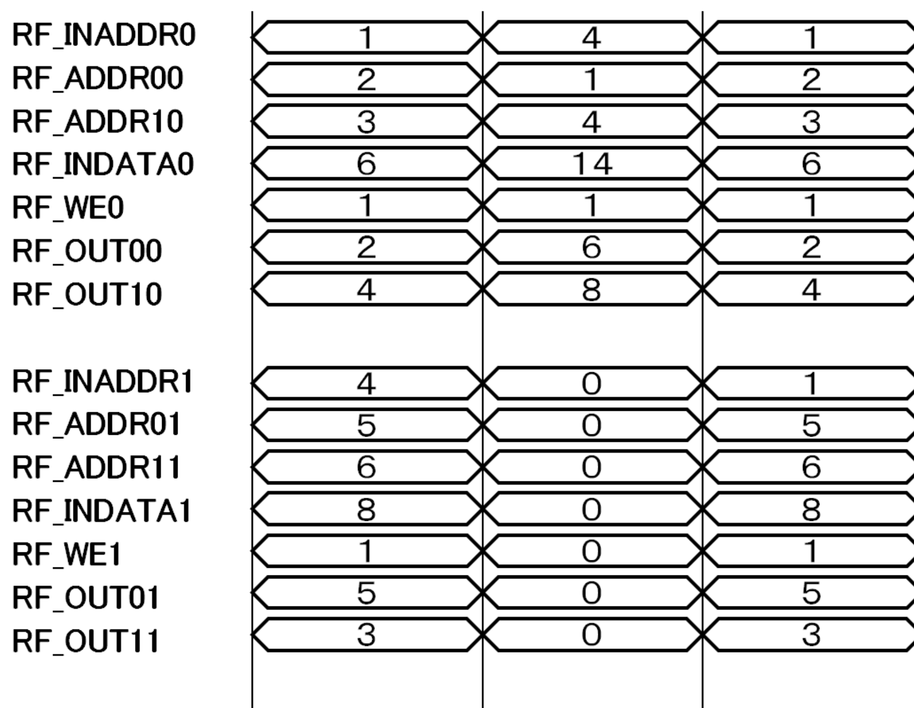

図 14：レジスタのシミュレーション波形

番地	初期データ	一回目	二回目	三回目
0	0	0	0	0
1	1	6	6	8
2	2	2	2	2
3	4	4	4	4
4	6	8	14	14
5	5	5	5	5
6	3	3	3	3
・ ・ ・	・ ・ ・	・ ・ ・	・ ・ ・	・ ・ ・
31	0	0	0	0

図 15：レジスタ内変化

シミュレーションの入力に応じて、レジスタ内が変化している。WE の立ち上がりがないときは、書き込みデータが入力されても格納されず、同じ番地にデータを格納するときも、下の命令の結果を仕様どおり優先して格納している。

(2) PPU

並列実行と連鎖演算の判定部が分かれているので、それぞれシミュレーションを行う。図 16 に並列実行判定のシミュレーションを示す。

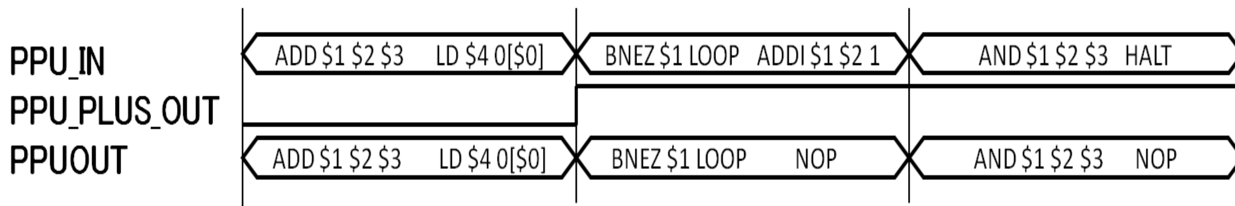

図 16：並列実行判定のシミュレーション

並列実行判定のシミュレーションでは、入力に応じて並列実行と単一実行に区別する。並列実行の場合は、そのままデータを流す。単一実行では下位命令を NOP に変更している。また、次の命令をフェッチする際に調整する信号(PPU_PLUS_OUT)を出力している。表 2 に命令別並列実行判定を示す。

表 2：命令別並列実行判定

上位 ALU	下位 ALU	並列動作		上位 ALU	下位 ALU	並列動作
R 演算	R 演算	○		R シフト	L 分岐	○
	R 比較	○			L メモリ	○
	R シフト	○		I 形式	R 演算	○
	I	○			R 比較	○
	L 分岐	○			R シフト	○
	L メモリ	○			I	○
R 比較	R 演算	○			L 分岐	○
	R 比較	○			L メモリ	○
	R シフト	○		L 分岐	すべて	×
	I	○		L メモリ	R 演算	○
	L 分岐	○			R 比較	○
	L メモリ	○			R シフト	○
R シフト	R 演算	○			I	○
	R 比較	○			L 分岐	○
	R シフト	○			L メモリ	○
	I	○		J 形式	すべて	×

表 2 のように、上位命令が L 分岐や J 形式の場合は制御ハザードを回避するため並列動作しない。命令別意列実行判定では、仕様どおりに設計できていることが確認できる。

加えて、下位命令から上位命令に連鎖はしないよう制御している。プログラムは逐次で書かれているため、下位命令の結果を上位命令に連鎖させてしまうとプログラムが正常に動作しない場合が考えられるためである。

次に連鎖判定のシミュレーションである。図 17 に連鎖演算判定のシミュレーションを示す。

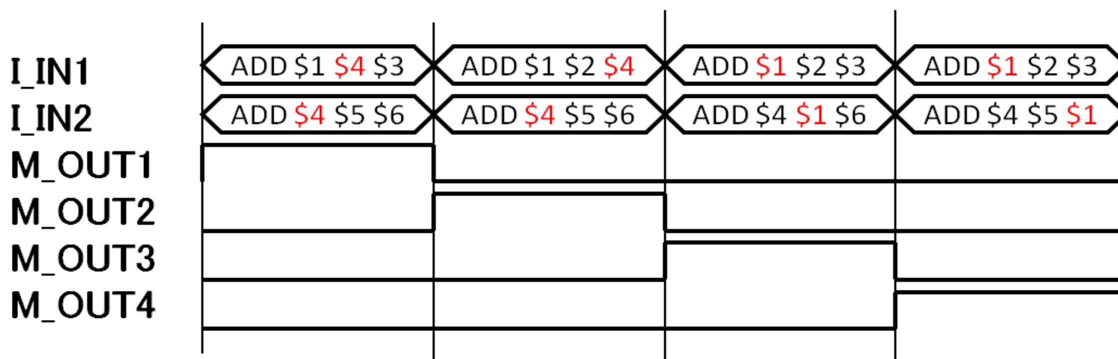

図 17：連鎖演算判定のシミュレーション

並列実行判定のシミュレーションでは、入力された命令間のデータ依存関係を検出し、依存関係がいれば、図 17 のように M_OUT＊を立ち上げる。各 M_OUT によって、連鎖演算のデータパスが制御される。データ依存の検出では、命令間のオペコードとオペランドを比較する。命令によってオペランドが変わってくるため制御する必要がある。これら並列実行と連鎖演算の判定部が動作することによって、並列演算、連鎖演算、単一実行が正常に動作できる。

(3) 2 ポート BRAM

2 ポートを用いることで、命令フェッチの時間短縮を図る。1 ポートでは 1 クロックに対して 1 命令をフェッチしていた。図 18 に 1 命令読み出しを示す。

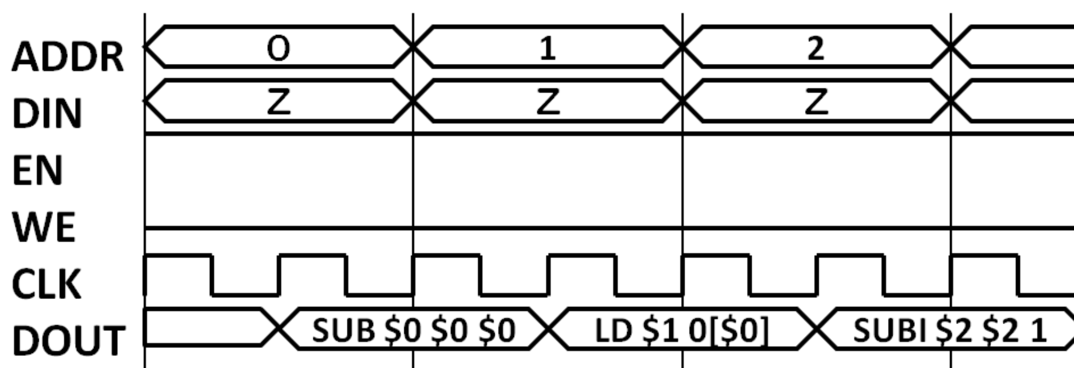

図 18：1 命令読み出し

1 ポートで複数命令を取り出す場合時間が命令数に応じて増えていく。これを解決するために 2 ポートで設計した。図 19 に 2 ポートのシミュレーションを示す。

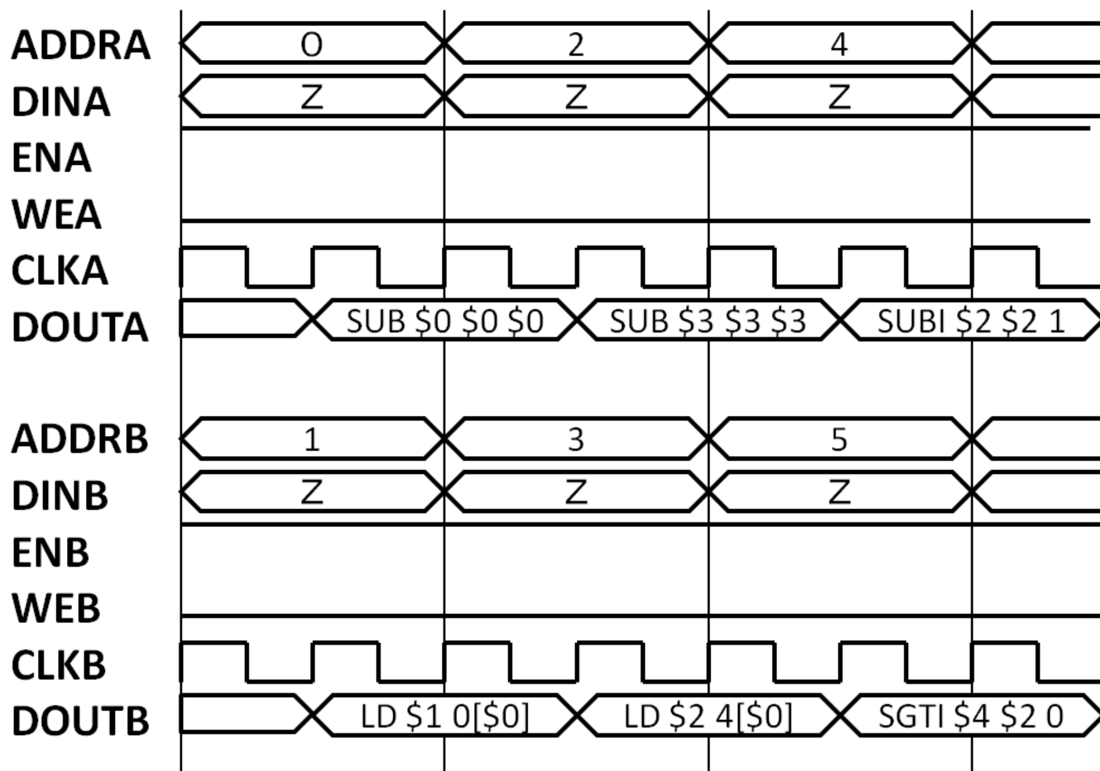

図 19：2 命令同時読み出し

2 ポートの場合、図 19 のように 1 クロックで 2 命令取り出すことができる。BRAM の設計は ISE Core Generator を使用して生成する。シミュレーションを行う際＊＊＊.coe ファイルを作成し、ISE Core Generator で作成時に読み込ませて生成を行う。こうすることではじめから BRAM 内にデータが格納された状態でシミュレーションでき、検証が容易になる。BRAM の特性上、図 18 と図 19 のようにデータの取り出す際、一クロック遅れて出力される。

(4) MAP

作成したモジュールをトップモジュールで統合し、配線を行うことにより MAP 全体を構築する。シミュレーション方法としてはあらかじめ BRAM 内にデータを設定しておき、段階に分けて動作確認を行う。一段階目は、上位 ALU のみ命令を実行させ、下位 ALU は NOP とし実行しない。こうすることにより、各命令が正しく演算できているか確認できる。二段階目は、並列演算のパターンを実行させる。各 ALU が正しく演算し、各命令

が実行されているか確認する。三段階目は連鎖演算のパターンを実行させる。上位命令の結果を下位命令に連鎖させ正しい演算が実行されているか確認する。最終段階では、簡単なプログラムを実行させる。プログラムが正常に実行し、最後に HALT 命令で停止信号 FIN が立ち上がるようにプロセッサが動作しているが確認する。表 3 に単一実行結果を示し、表 4 に並列演算結果を示し、表 5 に連鎖演算結果を示す。

表 3 : 単一実行結果

上位 ALU	単一実行
R 演算	○
R 比較	○
R シフト	○
I 形式	○
L 分岐	○
L メモリ	○
J 形式	○

表 4 : 並列実行結果

上位 ALU	下位 ALU	並列演算		上位 ALU	下位 ALU	並列演算
R 演算	R 演算	○		R シフト	L 分岐	○
	R 比較	○			L メモリ	○
	R シフト	○		I 形式	R 演算	○
	I	○			R 比較	○
	L 分岐	○			R シフト	○
	L メモリ	○			I	○
R 比較	R 演算	○			L 分岐	○
	R 比較	○			L メモリ	○
	R シフト	○		L 分岐	すべて	×
	I	○		L メモリ	R 演算	○
	L 分岐	○			R 比較	○
	L メモリ	○			R シフト	○
R シフト	R 演算	○			I	○
	R 比較	○			L 分岐	○
	R シフト	○			L メモリ	○
	I	○		J 形式	すべて	×

表 5：連鎖演算結果

上位 ALU	下位 ALU	連鎖演算		上位 ALU	下位 ALU	連鎖演算
R 演算	R 演算	○		R シフト	L 分岐	○
	R 比較	○			L メモリ	○
	R シフト	○		I 形式	R 演算	○
	I	○			R 比較	○
	L 分岐	○			R シフト	○
	L メモリ	○			I	○
R 比較	R 演算	○			L 分岐	○
	R 比較	○			L メモリ	○
	R シフト	○		L 分岐	すべて	×
	I	○		L メモリ	R 演算	×
	L 分岐	○			R 比較	×
	L メモリ	○			R シフト	×
R シフト	R 演算	○			I	×
	R 比較	○			L 分岐	×
	R シフト	○			L メモリ	×
	I	○		J 形式	すべて	×

単一実行ではすべての命令が正常に動作した。表 4 は表 2 と同様の結果が得られ、これにより並列演算では PPU が正しく動作できたことが確認できた。表 5 では表 4 と違い L メモリとの連鎖演算ができていない。これは MAP の特性上メモリアクセスし取り出したものはレジスタに格納されるまで使用できず、連鎖演算が行うことができないためである。それ以外の連鎖演算が可能であり動作構成としては正しく動作できている。これらを踏まえて、プログラムを実行する。サンプルプログラムとしてΣN を実行する。表 6 にΣN のシミュレーション結果を示す。

表 6：ΣN のシミュレーション結果

データメモリ番地	ΣN
0	$A_{16} = 10_{10}$
4	$37_{16} = 55_{10}$

データメモリには 0 番地に $000A_{16}$ の値を入れてプログラムを実行した。ΣN では、データメモリの 0 番地を N の値とし、総和を求めてデータメモリ 4 番地に結果を返す。実行結果として $0037_{16} = 55_{10}$ の正しい結果が得られた。

5. FPGA ボード上への実装と検証

5.1　プロセッサデバッガとプロセッサモニタ

(1)　プロセッサデバッガ・モニタの構成

プロセッサデバッガは、HSCS を利用して設計したプロセッサを、実機上で動作検証するためのハードウェア IP である。そもそも、プロセッサは単独では FPGA に実装しても動作しないモジュールである。プロセッサを動作させるためには、チップの電源が投入されてから切断されるまでの間に、命令メモリとデータメモリに命令とデータを格納し、外部からの制御信号によってプログラムを実行し、実行終了後に結果をメモリから読み出す必要がある。プロセッサデバッガでは、プロセッサを FPGA 上で動作させるために必要なモジュールとして、命令メモリ・データメモリ・システムシーケンサ・DMA コントローラ・バスコントローラを組み込み、プロセッサを検証するための周辺モジュールとして提供している。

プロセッサモニタは、プロセッサ設計の学習で利用するソフトウェアアプリケーションである。HDL によって設計したプロセッサを FPGA 上で動作検証する際に、ホスト PC 上からデバッグコマンドを入力し、プロセッサデバッガへ指示を与える。表 7 にデバッグコマンドを示す。表 7 の他に停止(halt)、リセット(rst)、終了(exit)、help のコマンドがある。テストの流れとしては、load でファイルをロードし、send でファイルを書き込む。これにより命令メモリやデータメモリにデータを書き込むことができる。set でプログラムカウンタの値を 0 にし､run で命令メモリに書き込まれたプログラムを実行する。read により値が正しく書きこまれているか、プログラムが正常に実行したか確認することもできる。

表 7 : デバッグコマンド

コマンド	ターゲット	意味
send	dm/im/rf	メモリ、レジスタの書き込み
read	dm/im/rf/pc	メモリ、レジスタの読み出し
save	dm/im/rf/pc/bp	メモリ、レジスタの内容を保存
load	dm/im/rf/bp	ファイルからロード
set	pc/bp	PC、ブレイクポイントの設定
del	bp	ブレイクポイントの削除
list/init	dm/im/rf/pc/bp	メモリ、レジスタの表示/初期化
run		通常実行
run clk N		Nクロック実行
run bp		ブレイク実行

dm:データメモリ　im:命令メモリ　rf:レジスタファイル　pc:プログラムカウンタ
bp:ブレイクポイント

(2)　MAP の実装

設計した MAP を FPGA ボード上に実装を行う。実装には HSCS で開発されたプロセッサデバッガ・モニタを用いる。プロセッサデバッガの実装対象として Xillinx 社の Spartan-3 & 3E Starter Kit ボードを使用する。Spartan-3A Starter Kit に実装できるように環境を整える。表 8 に MAP の設計規模と最大遅延を示す。

表 8：MAP の設計規模と最大遅延

モジュール	スライス数	LUT 数	FF 数	最大遅延(ns)
TOP	4593	8821	1057	46.324

プロセッサの FPGA 使用率はスライス数が 78%、LUT 数が 74%、FF 数が 8%となった。最高動作周波数は 31.052MHz である。このプロセッサをプロセッサデバッガに接続し、実行を行う。図 20 にプロセッサデバッガとプロセッサモニタの構成を示す。

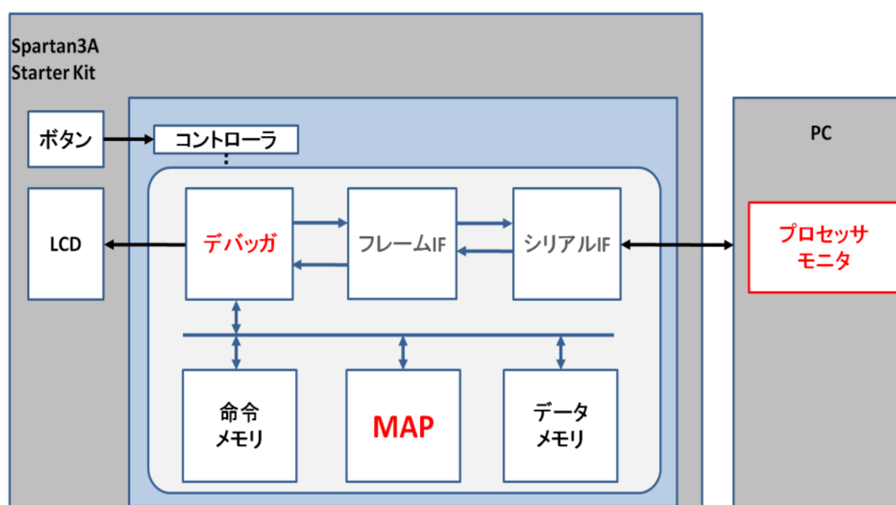

図 20：プロセッサデバッガとプロセッサモニタの構成

プロセッサデバッガにデバッグ対象のプロセッサを搭載するためには、インタフェースに準拠しなければならない。プロセッサデバッガで用意しているプロセッサの接続先は、データメモリ（DM）、命令メモリ（IM）、レジスタファイル（RF）、プログラムカウンタ（PC）である。DM と IM はプロセッサの外部との接続であり、必ず接続しなければならない。

今回 MAP をプロセッサデバッガに接続する場合、変更すべき箇所がある。Spartan-3A Starter Kit へ対応させるための pin 配置の変更、2 ポート BRAM への変更と命令語長の違いによるプロセッサデバッガ内のバス幅の調節、プロセッサモニタのフレームの調節が必要である。

以前まで、プロセッサデバッガは Spartan-3E に対応していたが、Spartan-3A がより良い実装環境のため変更を行う。pin 配置はクロック信号、LED、LCD 表示、ボタン入

力、rs232c 通信の五つである。rs232 通信で、実装環境の変化により、PC との通信ケーブルをメス - オスケーブルからメス - メスケーブルに変更する。そのため、使用する対応デバイスも変更し、pin 配置も対応させる。

プロセッサデバッガに接続されている IM、DM を ISE Core Generator を使用して、2 ポート BRAM を生成する。対応前のプロセッサデバッガにおいて、IM と DM は 1 ポート、データ幅 16bit で構成しているが、データ幅を MAP に対応し、2 ポート、データ幅 32bit で構成する。デバッグの際、メモリ内のデータを取り出す必要がある。そのため、2 ポートの場合は 1 ポート分のみデバッガからの制御信号、読み出し書き込みデータが送受信できるよう設計する。データ幅が変更になるので、プロセッサデバッガ内のデータ幅を変更する必要がある。MAP とプロセッサデバッガの接続部、MAP と IM、DM の接続部に注意して変更を行う。これにより、MAP をプロセッサデバッガに対応させることができる。

また、従来のプロセッサに比べ MAP の命令語長が 16bit から 32bit に変化し、レジスタファイル数の増可している。従来のプロセッサモニタでは対応ができないため、データ通信幅などのフレーム変更する必要がある。固定された値を MAP に対応させ、プロセッサモニタ側から正常にデバッグできる環境を整える。

これらの設定を行うことにより、MAP を FPGA ボード上に実装させ、PC 側からデバッグを行うことができる。

5.2　各種プログラムの動作確認

プログラムの動作確認ではプロセッサモニタからプログラムを転送し,DM へ使用するデータを設定する。PC 側からのコマンドでプロセッサを実行させ、DM 内の演算結果をプロセッサモニタで確認する。各種プログラムを FPGA ボードで実行し、動作確認を行うことにより MAP が正しく動作できているか確認を行う。

(1)　正数同士の乗算

C=A×B の計算を行う。A を 0 番地、B を 4 番地に設定し、C を 8 番地に格納する。表 9 に正数同士の乗算の実行結果を示す。

表 9 : 正数同士の乗算の実行結果

番地	test1	test2
0	5	1
4	4	10
8	*20*	*10*

表 9 より、C=A×B の計算が正しく実行され、DM に正しい演算結果が格納されているのが確認できた。

(2)　正数同士の除算

C=A%B の計算を行う。A を 0 番地、B を 4 番地に設定し、C を 8 番地、余りを 12 番地に格納する。表 10 に正数同士の除算の実行結果を示す。

表 10：正数同士の除算の実行結果

番地	test1	test2
0	20	15
4	4	6
8	*5*	*2*
12	*0*	*3*

表 10 より、test1 では割り切れる計算の値を、test2 では余りの出る計算の値を設定し実行させる。実行結果より、C=A%B の計算が正しく実行され、DM の 8 番地と 12 番地に正しい演算結果が格納されているのが確認できた。

(3)　ΣN

ΣN の計算を行う。N を 0 番地に設定し、ΣN を 4 番地に格納する。表 11 にΣN の実行結果を示す。

表 11：ΣN の実行結果

番地	test1	test2
0	5	10
4	*15*	*55*

表 11 より、ΣN の計算が正しく実行され、DM に正しい演算結果が格納されているのが確認できた。また、test2 では論理シミュレーションと同様の結果が得られた。

(4)　N！

N！の計算を行う。N を 0 番地に設定し、N！を 4 番地に格納する。表 12 に N！の実行結果を示す。

表 12：N！の実行結果

番地	test1	test2
0	5	7
4	*120*	*5040*

表 12 より、N！の計算が正しく実行され、DM に正しい演算結果が格納されているのが確認できた。

(5)　素数

素数の計算を行う。判定する値を 0 番地に設定し、判別結果を 4 番地に格納する。表 13 に素数の実行結果を示す。

表 13：素数の実行結果

番地	test1	test2
0	13	51
4	*0*	*1*

表 13 より、test1 の 13 では素数であり、DM の 4 番地に 0 がそのままの状態である。test2 の 51 では 3 や 17 で割り切れることにより、素数でないと判断され DM の 4 番地に 1 が格納されている。実行結果より素数の計算が正しく実行され、DM に正しい演算結果が格納されているのが確認できた。

(6)　最大公約数

A と B の最大公約数 C の計算を行う。A を 0 番地、B を 4 番地に設定し、最大公約数 C を 8 番地に格納する。表 14 に最大公約数の実行結果を示す。

表 14：最大公約数の実行結果

番地	test1	test2
0	5	10
4	20	13
8	*5*	*1*

表 14 より、test1 のとき、0 番地の 5 が最大公約数となり、DM に 8 番地に 5 が格納される。test2 のとき、10 と 13 共に 1 より大きい値の公約数がないため DM に 1 が格納される。計算が正しく実行され、DM に正しい演算結果が格納されているのが確認できた。

(7)　三角形の判別

三角形の判別の計算を行う。三辺を辺の短い順に 0 番地、4 番地、8 番地に設定し、判別結果を 12 番地に格納する。表 15 に三角形の判別の実行結果を示す。

表 15：三角形の判別の実行結果

番地	test1	test2	test3	test4	test5
0	1	5	2	3	3
4	2	5	5	4	4
8	5	5	5	5	6
12	*1*	*2*	*3*	*4*	*5*

表 15 より、test1 のとき三角形が成立しないので DM の 12 番地に 1 が格納されている。test2 のとき正三角形が成立するので DM に 2 が格納されている。test3 のとき二等辺三角形が成立するので DM に 3 が格納されている。test4 のとき直角三角形が成立するので DM に 4 が格納されている。test5 のときその他の三角形が成立するので DM に 5 が格納されている。計算が正しく実行され、DM に正しい判別結果が格納されているのが確認できた。

(8) 4 行 4 列の乗算

4 行 4 列の乗算の計算を行う。4 行 4 列の被乗数を 0 番地から 3C 番地に、4 行 4 列の乗数を 40 番地から 7C 番地に設定し、演算結果を 80 番地から BC 番地に格納する。表 16 に 4 行 4 列の乗算の実行結果を示す。

表 16：4 行 4 列の乗算の実行結果

番地	0	4	8	C
0	2	2	2	2
1	1	1	1	1
2	2	2	2	2
3	1	1	1	1
4	1	2	1	2
5	1	2	1	2
6	1	2	1	2
7	1	2	1	2
8	*8*	*16*	*8*	*16*
9	*4*	*8*	*4*	*8*
A	*8*	*16*	*8*	*16*
B	*4*	*8*	*4*	*8*

表 16 より、4 行 4 列の乗算の計算が正しく実行され、DM の 80 番地から BC 番地に正しい演算結果が格納されているのが確認できた。

(9)　BCD 加算

BCD 加算 C=A+B の計算を行う。A を 0 番地、B を 4 番地に設定し、C を 8 番地に格納する。表 17 に BCD 加算の実行結果を示す。

表 17 : BCD 加算の実行結果

番地	test1	test2
0	5	5
4	2	7
8	*7*	*12*

表 17 より、test1 では 16 進数の計算結果でも 10 進数表記と同じであり、test2 では 16 進数表記の場合 C_{16} であるが、BCD 加算により 12_{16} で格納された。計算が正しく実行され、DM に正しい演算結果が格納されているのが確認できた。

(10)　2 点を通る直線

2 点を通る直線の計算を行う。(X_1、Y_1)を 0 番地、4 番地に設定し、(X_2、Y_2)を 8 番地、12 番地に設定する。直線の方程式 Y=aX＋b の a、b を 16 番地、20 番地に格納する。表 18 に 2 点を通る直線の実行結果を示す。

表 18 : 2 点を通る直線の実行結果

番地	test1	test2
0	2	2
4	3	1
8	3	1
12	4	2
16	*1*	*−1*
20	*1*	*3*

表 18 より、正しい実行結果が得られた。test2 では傾きが負の数の場合でも計算が正しく実行され、DM に正しい判別結果が格納されているのが確認できた。

(11)　バイトニックソート

バイトニックソートの計算を行う。固定数のソートを行うもので 0 番地から 28 番地に値を設定し、ソート結果を格納する。表 19 にバイトニックソートの実行結果を示す。

表 19：バイトニックソートの実行結果

番地	test1	ans1	test2	ans2
0	7	1	7	−4
4	2	2	2	1
8	4	3	−4	2
12	3	4	3	3
16	5	5	5	5
20	8	6	8	6
24	6	7	6	7
28	1	8	1	8

表 19 より、0 番地から昇順にソートされている。また、test2 において負の数が設定される場合でも計算が正しく実行され、DM に正しい判別結果が格納されているのが確認できた。

(12)　Booth アルゴリズムによる乗算

Booth アルゴリズムによる掛け算 C＝A×B を行う。A を 0 番地、B を 4 番地に設定し、C を 8 番地に格納する。表 20 に Booth アルゴリズムによる乗算の実行結果を示す。

表 20：Booth アルゴリズムによる乗算の実行結果

番地	test1	test2	test3	test4
0	4	−4	4	−4
4	3	3	−3	−3
8	*12*	*−12*	*−12*	*12*

表 20 より、正しい実行結果が得られた。正負の数の場合でも計算が正しく実行され、DM に正しい判別結果が格納されているのが確認できた。

以上のプログラムの実行結果から、MAP を FPGA ボードに実装でき、正常に動作できることを確認した。

6　並列性の評価と考察

6.1　並列性の評価

MAP において並列性の評価を行う。表 21 に 2ALU の並列性を示す。静的な並列性では、並列演算が平均して 27%占めており、連鎖演算は 28%占めている。プログラム内における並列性の可能性としては 55%も占めており、演算レベル並列性の有効性があるといえる。また動的な並列性では、並列演算が平均して 27%占めており、連鎖演算は 46%占めている。プログラムの実行時における演算レベル並列性は平均して 73%のあり、複数 ALU の有効性があるといえる。静的、動的並列性を見ても、連鎖演算の有効性は大きいと言える。データ依存の命令が多く含まれ、連鎖演算を除くと単一実行となり、大きく並列性を損ねてしまう。

2ALU における演算レベル並列性は、複数 ALU を用いることの有効性を示すことができたといえる。

表 21：2ALU の並列性

プログラム		掛け算	割り算	ΣN	N!	素数判定	最大公約数	三角形判別	4行4列の乗算	BCD加算	直線	bytonic	ブース掛け算		平均
動的	並列演算	49	49	0.1	52	5	8	41	45	0	36	15	28		27
	連鎖演算	49	48	99	46	57	46	39	34	33	29	17	56		46
	単一実行	2	3	0.9	2	38	46	20	21	67	35	68	15		27
静的	並列演算	18	25	33	54	25	20	21	33	0	40	22	33		27
	連鎖演算	36	25	18	18	50	30	33	9	38	31	11	45		28
	単一実行	46	50	49	38	25	50	46	58	62	29	67	22		45

%で表示

6.2 考察

MAP システムを作成し、MAP を FPGA ボードに実装することができた。実装するに当たって、プロセッサデバッガ、プロセッサモニタの変更を行う必要があったが、データ幅や、メモリ空間の調整を行うことで対応できることがわかった。演算レベル並列性の検証において、静的な並列性はアセンブリを用いずに検証しており、動的な並列性はシミュレータ上で検証している。静的な検証ではアセンブラに、並列性の検出機能を実装させ、検証時間を短縮させる必要がある。また、動的な演算レベル並列性の検証において、ハードウェアによる実機上の検出を行うことにより、正しい並列性の結果が得られる。

今回検証できなかったが、MAP の連鎖演算機能の有効性を検証する必要がある。連鎖演算を行う場合動作周波数が低下する。連鎖演算の有無による動作周波数と実装規模の変化を検証し、比較する必要がある。連鎖演算の効果は、表 21 で示すとおり非常に高い。しかし、動作周波数が連鎖演算の効率よりも性能が下がるものであってはいけないと考える。また、ALU 数を増やした MAP を設計し、4ALU での演算レベル並列性を検証すべきと考える。

MAP プログラミングでは学習者がアセンブラ、シミュレータを用いてプログラムのデバッグを簡単に行うことができた。これを用いることで FPGA ボードでの検証がスムーズに行うことができ、実験環境が向上した。またプロセッサモニタにおいて、GUI を用いた検証が可能になり、デバッグ環境も向上することができた。今後はシミュレータの可視化を行い、プロセッサの動作学習や、並列演算の学習に利用できるようにしたい。

7　おわりに

本論文では、本研究室で開発してきた HSCS をもとに、新たらしい MAP システムを開発した。ソフトウェア側におけるプログラミング、MAP アセンブラ、MAP シミュレータ、プロセッサモニタ、ハードウェア側における MAP 設計、プロセッサデバッガの並列性の検証システムを構築した。また、マルチ ALU プロセッサ MAP のデータパス、命令セット、プログラミング、並列実行について述べた。MAP では、ユーザは演算の逐次プログラムを記述し、実行時に 2 命令ずつフェッチして、並列演算、連鎖演算を判定して実行する。また、MAP アセンブラにも並列性検出機能をもたせ、並列性検出をハードとソフトいずれで行うのが良いかを評価する予定である。

FPGA ボード上で動作させるためのプロセッサデバッガについても述べた。MAP の HDL 記述と論理合成、論理シミュレーションを行い、Spartan3A Starter Kit へプロセッサデバッガを移植した。2ALU の MAP を Spartan 3A 上で実動作させ、プロセッサデバッガを用いてプログラムの検証を行える環境を整えた。

MAP システムを用いて、ソフトウェア、ハードウェアにおける並列性の検証が必要である。現在、アセンブラには並列性の検証機能がないため自動的に検出されず、PPU における検出は実装されているが、実機上における演算レベル並列性がどの程度か検証できない。ハードウェア・ソフトウェア両面の検証機能の向上が課題である。また、今回設計した MAP は 2ALU で設計されたものであり、演算レベル並列性の検証するにあたって ALU を増加した 4ALU の MAP の設計が更なる課題である。

謝辞
本研究の機会を与えてくださり、ご指導を頂きました山崎勝弘教授に深く感謝いたします。また、本研究に関して様々な相談に乗って頂き、貴重な助言を頂いた孟助手はじめ、様々な面で貴重な助言や励ましを下さった研究室の皆様に深く感謝いたします。

参考文献

[1] David A.Patterson, john L.Hennessy 著，成田光彰　訳：コンピュータの構成と設計(上)(下)，日経 BP 社，1999.

[2] 小林優：入門 Verilog-HDL 記述，CQ 出版，2001.

[3] 並木英明, 前田智美, 宮尾正大 : 実用入門ディジタル回路と Verilog-HDL, 技術評論社, 1996.

[4] 坂井修一　著，コンピュータアーキテクチャ，電子情報通信学会、コロナ社，2004.

[5] 池田修久 : ハード/ソフト・コラーニングシステム上での FPGA ボードコンピュータの設計と実装，立命館大学理工学研究科修士論文，2004.

[6] 池田，他 : ハード/ソフト・コラーニングシステムにおける FPGA ボードコンピュータの設計，情報処理学会，第 66 回全国大会論文集，5T-5，2004.

[7] 大八木，他：ハード/ソフト・コラーニングシステムにおけるアーキテクチャ選択可能なプロセッサシミュレータの設計, 情報処理学会, 第 66 回全国大会論文集, 5T-6, 2004.

[8] 古川達久：マルチサイクル・パイプライン方式による教育用マイクロプロセッサの設計と検証，立命館大学理工学部卒業論文，2003 .

[9] 難波、中村、小柳、山崎：マイクロプロセッサの設計と検証に基づいたハード/ソフト・コラーニングシステムの拡張、FIT2005、情報処理科学技術レターズ、C-001、pp.33-36、2005.

[10] Hoang Anh Tuan, K. Nakamura, S. Namba, K. Yamazaki and S. Oyanagi:A FPGA Based hardware/soufware Co-learning System, 信学技報、RECONF2005-48、2005.

[11] 難波、中村、Tuan、山崎、小柳：ハード/ソフト協調学習のための命令セット定義ツールとプロセッサデバッガの開発，FIT2006, N-009，2006.

[12] 難波、志水、山崎、小柳：プロセッサ設計支援ツールの設計・実装とハード/ソフト協調学習システムの評価、FIT2007、情報科学技術レターズ、LC002、pp.39-42、2007.

[13] 井手、志水、山崎、小柳：学生によるプロセッサ設計実験に基づいたハード/ソフト協調学習システムの評価、FIT2008、C-009、2008.

[14] PISHVA JOHN CYRUS P、井手、山崎：ハードソフト協調設計のためのコンパイラ学習システムの設計と実現、情報処理学会関西支部大会　ものづくり基盤コンピューティングシステム研究会、A-10、2010.

[15] 境、山崎：FPGA ボード上でのマルチ ALU プロセッサの設計と実装、情報処理学会関西支部大会　ものづくり基盤コンピューティングシステム研究会、A-02、2011.

[16] 田中　亮佑：マルチ ALU プロセッサにおけるアセンブラの設計と試作（Ⅰ）、立命館大学理工学部卒業論文、2012.

[17] 高松　良太：マルチ ALU プロセッサにおけるシミュレータの設計と試作、立命館大

学理工学部卒業論文、2012.
[18] 境　直樹：MAP 仕様書、2011.
[19] 石川　陽章：マルチ ALU プロセッサのデバッガ用 GUI の設計と検証、立命館大学理工学部卒業論文、2013.
[20] 杵川　大智：マルチ ALU プロセッサの並列プログラミングと動作検証、立命館大学理工学部卒業論文、2013.