卒業論文

ハードウェア記述言語による
マルチサイクル/パイプラインマイクロプロセッサの設計

氏名 ： 大八木 睦
学籍番号 ： 2210980048-5
指導教員 ： 山崎 勝弘教授
提出日 ： 2002 年 2 月 18 日

立命館大学 理工学部 情報学科

内容梗概

本研究では、Verilog-HDL を用いて、マイクロプロセッサの最も基本的なアーキテクチャを理解する目的で、マルチサイクルマイクロプロセッサ、及びパイプラインマイクロプロセッサを設計を行った。マイクロプロセッサの設計においてはハードウェアを削減するため、MIPS のサブセットとして独自に 16 ビットの命令セットを作成した。

パイプラインマイクロプロセッサは、単一サイクルマイクロプロセッサのデータパスをパイプライン化することにより設計した。また、設計したマイクロプロセッサにおいて 3 つのテストパターンを作成し、Post-Map、及び Post-Place&Route の 2 種類のゲートレベルにおける動作検証を行った。

さらに単一サイクル、マルチサイクル、パイプラインの各マイクロプロセッサについて、ハードウェア規模を比較し、最短クロックサイクル、命令実行時間、CPI などの観点から性能比較とその考察を行った。

ハードウェア規模は、単一サイクルとマルチサイクルがほぼ同程度で、パイプラインは処理の複雑さから最も大規模となった。命令実行時間は、マルチサイクルが単一サイクルの 1.7 倍程度遅くなった。これは、命令セットの簡易性や、マルチサイクルの命令実行方式によって、制御ユニットやメモリユニットの遅延がボトルネックになり、最短クロックサイクルが単一サイクルの 3 分の 2 程度に制限されたからだと考えられる。一方、パイプラインは単一サイクルと比べてストール発生回数が少ない場合、3.4 倍の高速化が達成された。

目次

図目次

表目次

1　はじめに

1970 年代、集積回路が普及した時代に LSI が使用されていた製品は電卓、時計、ゲーム機といったパーソナル機器であった。1980 年代に入ると DRAM が普及し、以降の半導体設計規模が増大し始めた[3]。また、近年に入ってからの半導体技術の進歩はめざましく、1 チップに集積可能な論理回路の規模は飛躍的に増大し続けている[2]。1995 年には 10 万から 50 万ゲート規模であったものが、現在では約 10 倍の 100 万ゲートから 500 万ゲート規模に大規模化している。また SoC（System on Chip）と呼ばれるシステム自身を 1Chip 化する設計に移行しつつあるため、設計規模が益々大きくなっている[3]。

LSI 設計のシミュレーション技術は、1980 年代に入りゲートレベルシミュレーションが自動化され、それまでのボード上でのシミュレーションからコンピュータ上でのシミュレーションが可能になった。しかし、近年の設計規模の拡大により、従来の設計方法である回路図入力ではゲートレベルシミュレータを用いても大規模回路を要求仕様通りに短期間で開発することが困難になってきた。設計時間の短縮とコスト削減の必要性、また優れた論理合成ツールの出現によりゲートレベル設計から HDL（Hardware Description Language）記述設計へ変移している。HDL とは回路の動作や構造を記述するための言語で、回路設計では従来のように回路図入力ではなく HDL によって回路動作を記述する[5]。HDL には主に C 言語のような記述性を持つ Verilog-HDL と厳格な使用言語である VHDL がある。本研究では Verilog-HDL を用いて設計を行う。

同時に HDL の出現により、以前までのセル部から設計をはじめ、最終的にターゲットの製品を作成する方法（ボトムアップ 設計）から、機能の検証からアプローチする方法（トップダウン設計）に移行している。トップダウン設計では設計工程を機能検証工程とゲートへの具体化作業工程の二つに分けることにより、設計者は初めからタイミングなどの考慮をする必要がなく、機能設計に関するバグを早い段階で見つけることができ、開発期間の短縮に繋がっている[4]。

また、論理機能を焼き付けることで、その場ですぐに LSI として利用可能であるプログラム可能なゲートアレイ（FPGA：Field Programmable Gate Array）の登場でコスト削減、開発期間短縮化が目覚しいものとなってきている。

本研究室では、ハードウェア記述言語による FPGA 上への教育用マイクロプロセッサの実装[10]において、VHDL による CPU の設計が行われた。また C 言語からのハードウェア自動生成システムの構築[11]、VHDL 記述によるハードウェア設計[12]において HDL や FPGA に関する研究が行われ、成果を上げている。

以上のような背景を踏まえ、本研究では Verilog-HDL によるマイクロプロセッサの設計を行う。1971 年の世界初のマイクロプロセッサである 4004 の登場以来、マイクロプロセッサは飛躍的な進化を続けてきた。現在開発中のスーパースカラプロセッサ UltraSPARC-3 は動作周波数 600MHz、パイプライン段数最大 14 段、最大命令発行数が 6 とパソコン用プロセッサの 2 倍以上の性能を達成しようとしている[7]。

今回本研究では、マイクロプロセッサの基本的なアーキテクチャを理解するため、MIPS のサブセットとして独自に作成した命令セットにより、マルチサイクル方式のマイクロプロセッサ、およ

び 5 段パイプライン方式のマイクロプロセッサを設計する。回路設計の手順として、HDL による設計、機能シミュレーション、論理合成、Post-Map と Post-Place&Route の 2 種類のゲートレベルシミュレーションを行いそれぞれの結果を比較する。

はじめに第 2 章でハードウェア設計の概要について述べる。第 3 章では独自命令セットの仕様、マルチサイクルプロセッサの設計、及びゲートレベルシミュレーションによる動作検証について述べる。第 4 章ではパイプラインの設計について述べる。単一サイクルプロセッサのデータパスのパイプライン化から、データハザードや分岐ハザードの詳細、及びシミュレーションによる動作検証について述べる。第 6 章ではこれまで作成したマルチサイクルプロセッサと単一サイクルプロセッサ、及びパイプラインプロセッサのハードウェア規模や性能比較について述べる。第 6 章では現在までの成果と今後の課題を述べる。

2 ハードウェア記述言語によるシステム設計

2.1 ハードウェア記述言語

ハードウェア記述言語（HDL）はハードウェア動作を記述する言語である。回路規模が数十万ゲートを超すようになると、従来のような回路図を使用して設計をする方法では、デバッグが困難であり設計期間が非常に長くなってしまう。

HDL では回路図による設計より上位レベルでの設計が可能であるので、開発期間を短縮でき大規模な設計を行える。論理合成ツールを使用することで、目標とする性能や面積などの設計制約条件をゲート回路に自動生成できる[3][4]。HDL は他のプログラム言語にはない、時間の概念を持っている。ハードウェア内では入力から出力へ伝播する際に遅延が生じる。HDL は遅延を表現するための記述方法を持つ。

HDL の設計記述レベルにはゲートレベル、RTL（Register Transfer Level）、動作レベルの 3 つがある。それぞれの特徴を以下に示す。

- ゲートレベル
  - 回路表現の最下位レベル
  - RTL 記述から導かれた論理構造を保持する
  - 詳細なブール式構造や論理的な実現を表している
- RTL
  - 論理レベルより一段上の階層
  - レジスタを明確に定義し、レジスタ間に存在する組み合わせ回路を明確に記述
  - 論理合成できるレベル
- 動作レベル
  - RTL レベルよりさらに上位レベル
  - 仕様に対し、動作を思い浮かべながら記述する
  - レジスタなどを明確に記述する必要がない
  - 論理合成は不可

HDL には VHDL、Verilog-HDL、SFL（Structured Function Description Language）、UDL/I（Unified Design Language for Integrated circuit）のようにようにいくつかの種類がある[3][4]。その中でも、現在最も普及している VHDL と Verilog-HDL について述べる。

(1) VHDL

米国国防省が中心となって開発された多種多様な設計データを管理・保守するドキュメンテーション言語である。曖昧さを排除した厳格な仕様言語で、通信用 LSI やシステム設計などの開発に利用されている。複雑なデジタルシステムの設計、解析、シミュレーションを行うための幅広い言語構文を持っている。1987 年に IEEE（米国電気電子技術者協会）の認定を受けた。

(2) Verilog-HDL

当初、シミュレータ Verilog-XL の専用言語として開発され、その後 Verilog-HDL のソースが公開された。1993 年に OVI（Open Verilog International）によって標準化が開始され、1995 年に IEEE に認定された。C 言語をベースとしており、記述性に優れた言語で、シンプルで実際の設計に都合のいい言語でもある。また、シミュレーション用言語として開発された背景もあり、シミュレーション用記述が充実している[2]。

以下の図 1 の半加算器における VHDL と Verilog-HDL の記述例を図 2、3 に示す。

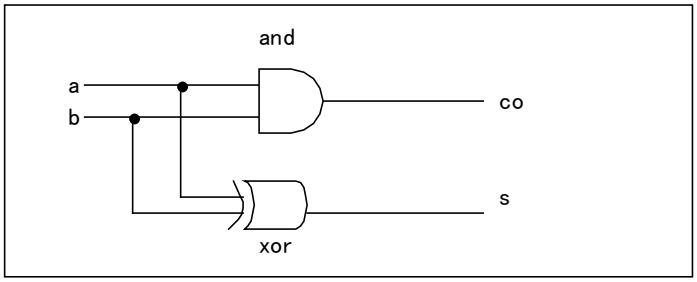

図 1 半加算器の論理回路図

```
1: library IEEE;                          --ライブラリ宣言
2: use IEEE.std_logic_1164.all;           --パッケージ呼び出し

3: entity half_addr is                    --論理回路名
4:         port(a,b : in std_logic;       --入力信号a,bを定義
5:                s,co : out std_logic);  --出力信号s,coを定義
6: end half_addr;
7:
8: architecture structure is              --half-addrの構造を定義
9:         s <= a xor b;                  --sにa xor bの結果を代入
10:        co <= a and b;                 --coにa and bの結果を代入
11: end dataflow;
```

図 2 半加算器の VHDL による記述

VHDL では最初の 1 行目と 2 行目でライブラリ宣言とパッケージ呼び出しが必要である。これは C 言語の include 文のようなものである。VHDL では外部とのインターフェース部であるポート宣言を entity 部で、記述する論理回路の構造と architecture 部で記述する。

```
1: module(a, b, s, co);                 //ポートリスト宣言
2:         input a,b;                   //入力信号a,bを定義
3:         output s, co;                //出力信号s,soを定義
4:
5:         assign s = a ^b;             //sにa xor bの結果を代入
6:         assign co = a & b;           //coにa and bの結果を代入
7: endmodule
```

図 3 半加算器の Verilog-HDL による記述

Verilog-HDL では 1 行目に外部とのポート宣言を行い、2,3 行目で入力信号と出力信号の定義を行う。そして、続いて論理回路の構造を示す。Verilog-HDL では VHDL のようなパッケージ呼び出しは不要である。

2.2 LSI 設計手法について

設計手法には主にボトムアップ設計とトップダウン設計がある。それぞれの特徴を以下に示す。

- ボトムアップ設計
  - 回路図入力でゲートレベルシミュレータで設計を行う
  - テクノロジの設計を最初に決めなければならない
  - 各ブロックの設計が完成しないと全体の検証がはじめられない
  - デバッグが困難
  - 設計の最終段階まで製品の完全な動作検証ができない

- トップダウン設計
  - HDL と論理合成ツールにより普及
  - 設計の初期の段階で仕様の致命的な欠陥を発見することができる
  - 設計の遅いブロックに全体の設計期間が引っ張られない
  - 論理合成により時間短縮が出来、設計者は使用の検討/検証に専念できる
  - 設計データを簡単に再利用可能

ボトムアップ設計の欠点を克服するため、現在はトップダウン設計手法へと移行しつつある。トップダウン による LSI 設計は機能検証の工程とゲート回路の実現化の工程に分けることができる。機能検証はロジック部の設計と開発を HDL により行う工程であり、ゲート回路の実現化はフィジカル部すなわちよりチップに近い場所の設計を行う工程である。

トップダウン設計のフローチャートを図 4 に示す。

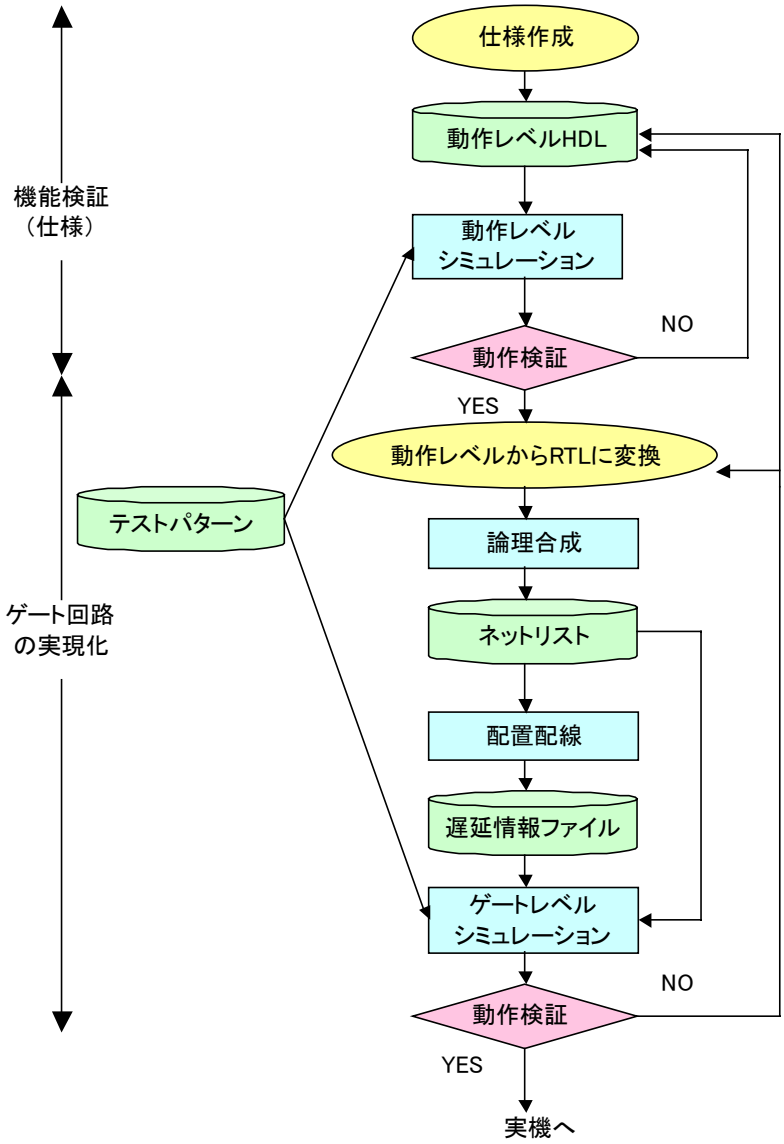

図 4 トップダウン設計のフローチャート

トップダウン設計では HDL で書かれた設計ユニットにおいて動作が仕様を満たしてるかを動作レベルシミュレーションで確認する。この場合、具体的なタイミングなどの考慮をしなくてもよいので、機能設計に関するバグを早期に発見することが可能である。動作レベルシミュレーションで動作を確認した後、合成ツールによって論理合成を行う。そして、設計回路を最適化しゲートレベルシミュレーションでタイミング動作の確認を行う[4]。

2.3 FPGA

FPGA（Field Programmable Gate Array）はプログラム可能なゲートアレイで、論理機能を焼き付けることで、その場で LSI として利用可能である。従来、設計した LSI の動作確認は実際に LSI に焼くか、シミュレーションに頼っていた。しかし、実際に LSI には時間とコストがかかり、大規模な回路になるとシミュレーションでは時間がかかりすぎて実用で気ではなかった。FPGA の出現によりこの問題は解決された。設計した LSI の構成データをその場ですぐに FPGA に焼きつけることで動作検証を行うことが出来、コストと時間の削減に大いに貢献した。

2.4 プロセッサアーキテクチヤの分類

マイクロプロセッサは段階的に発展をしてきた。

第 1 期は 1971 年からの 6 年間で、マイクロプロセッサの黎明期で、世界初のマイクロプロセッサ 4004 が誕生した。第 2 期は 1977 年からの 9 年間で、マイクロプロセッサの発展時期である。16 ビットマイクロプロセッサ 8080 などが開発され（1978 年）、命令とシステムのアーキテクチャの争いとなった。第 3 期は 1987 年からの 9 年間でワークステーションとパソコン用プロセッサの性能競争であり、コンピュータで開発された性能向上技術のプロセッサへの導入であった。性能競争は Pentium Pro の登場(1995 年)でますます激しくなった。第 4 期は 1995 年からのマルチメディア時代で、オブジェクト指向技術はマルチメディア時代のプロセッサ開発に重要な考え方となった[7]。

このように様々なプロセッサが開発されて行く過程で、現在までに高速化のためのプロセッサアーキテクチャも提案されている。以下に、主なプロセッサアーキテクチャを示す。

(1) 単一サイクル方式

1 命令を 1 クロックサイクルで行う形式。すべての命令はクロックエッジから命令を開始し、次のクロックエッジで命令を完了する。処理がもっとも長い命令にあわせてクロックサイクルを設定するため効率が悪く、現代のマイクロプロセッサのアーキテクチャとして使用されることはまずない。

(2) マルチサイクル方式

ひとつの命令を複数のステップに分け、それぞれのステップが 1 クロックサイクルを占める。1 クロックサイクルを短くすることが可能で、処理を高速化できる。また、1 命令につき同じ機能ユニットを 2 回以上使用できるため、ハードウェアコストの削減にもなる。マルチサイクル方式の詳細は 3 章で述べる。

(3) パイプライン方式

ひとつの命令を複数ステップ（ステージ）に分け、連続した命令の各ステージをのを少しずつずらして同時並行的に実行する実現方式。命令のスループットが増大し、命令の全体のクロックサイクル数が大幅に減少する。しかし、パイプラインステージを増やせば増やすほど高速化が期待できるというわけではない。実際には次のクロックサイクルで次の命令が実行できないという現象（パイプラインハザード）が起こるためである。
パイプラインハザードには大きく分けて構造ハザード、制御ハザード、分岐ハザードの 3 つがある。これらのハザードを解決するためにはフォワーディング、ストール、分岐予測といった方法がある[1]。フォワーディング、ストール、分岐予測の詳細については 4 章で述べる。

(4) スーパースカラ方式

プロセッサ内のユニットを多重化して、各パイプラインステージのそれぞれで複数の命令が流れるようにする実現方式。命令を複数個フェッチし、同時に実行できる命令を動的スケジューリングにより探し出し、複数の命令を同時に実行する。命令の実行順序はプログラムに掛かれている順序とは異なったものになる[8]。

(5) VILW（Very Long Instruction Word）方式

1 つの命令語長を 256～1024 ビットと長く作り、その中に複数の命令を格納しておきそれらを全て同時に実行する方式。常に決まった数の命令がパイプラインに投入され同時に実行される。各命令には依存関係がないように事前にコンパイラによって最適化しておくか、同時に実行できる命令がないときは NOP（No Operation）を埋めておく。依存関係がないのでスーパースカラのような動的スケジューリングではなく、静的スケジューリングを行うのでハードウェアを簡素化することができる[9]。

3 マルチサイクルマイクロプロセッサの設計

3.1 命令セットアーキテクチャ：MONI

本研究では独自に命令セットの作成を行った。(以下 MONI 命令セットと呼ぶ) 32 ビットの MIPS 命令のビット幅を 16 ビットに半減することで、ハードウェアを削減し、無駄のない設計を行える。MONI 命令セットのフォーマットを表 1 に示す。

表 1 命令フォーマット

命令形式/フィールド長	4	3	3	3	3
R 形式	op	rs	rt	rd	funct
I 形式	op	rs	rt	address	
J 形式	op	taeget address			

以下にラベルの詳細を示す。

op ： opcode。命令ソースコード。命令形式の判定。４ビット
rs ： 第１のソース・オペランドのレジスタ。３ビット
rt ： 第２のソース・オペランドのレジスタ。３ビット
rd ： デスティネーションレジスタ。結果を収める先。３ビット
funct ： 機能コード。R 形式命令で実際の ALU の機能を明確にする。３ビット
address ： rs に対する offset 値
target address ： メモリ番地を示す即値アドレス

表 2 に MONI 命令セットの全命令の概要を示す。

表 2 MONI 命令セット

命令形式	命令内容	オペランド	概要
R 形式命令	算術演算命令	add,sub,and,or	rd ← rs 演算子 rt
		big	if(rs > rt) then rd ← rs else if(rs < rt) then rd← rt
		slt	if(rs <rt) then rd ←1 else rd ←1
I 形式命令	データ転送	lw	rt ← memory[rs ＋ address]
		sw	memory[rs ＋ address] ← rt
	条件分岐命令	beq	if(rs = rt) then pc ← address
J 形式命令	ジャンプ命令	j	pc ← address
HALT 形式命令	HALT 命令	halt	プログラムの停止

3.2　マルチサイクルマイクロプロセッサのアーキテクチャ

図 5 にマルチサイクルマイクロプロセッサのデータパスを示す。

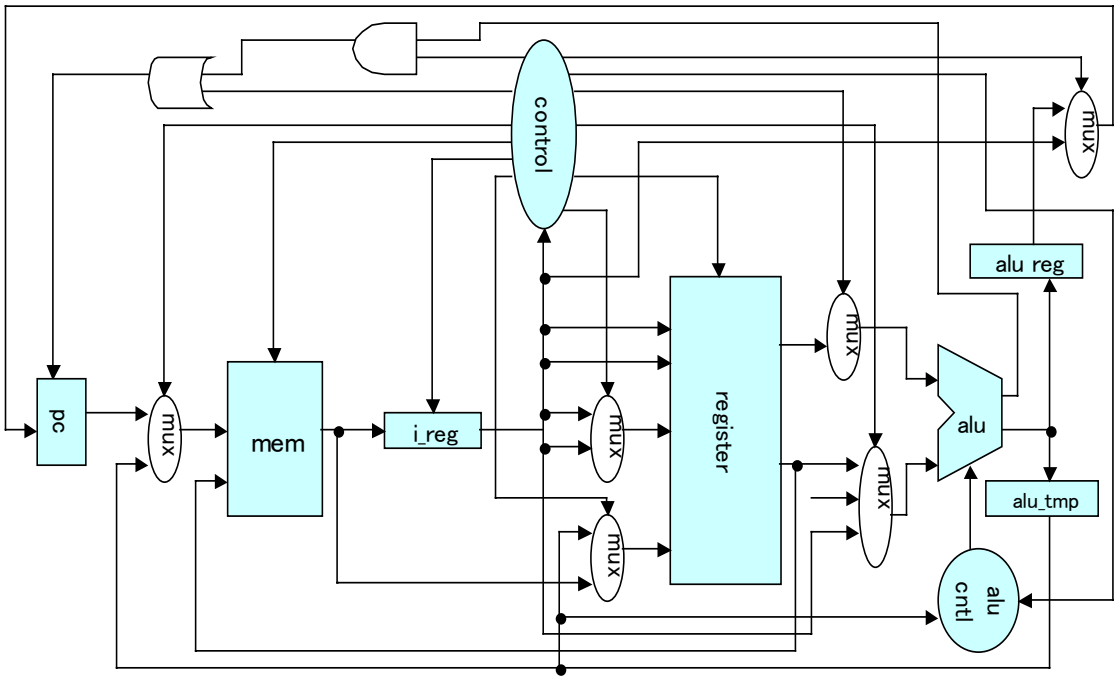

図 5 マルチサイクルマイクロプロセッサのデータパス

マルチサイクルマイクロプロセッサは以下の機能ユニットで構成されている。それぞれの機能を述べる。

- プログラムカウンタ(pc)　・・・現在実行中の命令のメモリアドレスを格納する
- メモリ(mem)　・・・命令とデータを格納する
- 命令レジスタ(i_reg)　・・・現在実行中の命令を格納する
- 制御ユニット(control)　・・・命令のステップごとの制御を行う
- レジスタ(register)　・・・変数などを格納する
- ALU(alu)　・・・算術演算および、ゼロ判定を行う
- ALU 制御 (alu control)　・・・ALU の演算を決定する制御を行う
- ALU レジスタ(alu_reg)　・・・ALU の演算結果を一時保存する
- ALU レジスタ（alu_tmp）　・・・ALU の演算結果を一時保存する

マルチサイクル方式では複数のサイクルに渡ってひとつの命令を実行するため、現在実行中の命令を命令レジスタ(i_reg)に保存しておく必要がある。命令レジスタは次の命令が実行を開始する時に書き換えられる。

ALU レジスタ（alu_reg）は実行中の命令が分岐命令で分岐条件が成立した場合に、分岐先のアドレスの計算結果を次のクロックサイクルで使用するために必要なユニットである。

また、もうひとつの ALU レジスタ(alu_tmp)はステップ 3 で出力された ALU の演算結果をステップ 4 で使用する時までに更新されてしまわないように設置した一時レジスタである。計算された値をクロックの立下り時に書き込み、次のクロックの立ち上がりでその値を使用する。

3.3 命令の実行と制御

マルチサイクル方式のマイクロプロセッサは、

1. 命令フェッチ
2. 命令デコードとレジスタのフェッチ、及び分岐先アドレスの計算
3. メモリアドレスの計算及び演算の実行
4. メモリアクセス
5. メモリ読み出し

の各ステップを 1 クロックサイクルで行う方式である。
表 3 に命令の各ステップについて述べる。

表 3 マルチサイクルの実行ステップ

step	R 形式	ロード	ストア	条件分岐	ジャンプ
1	i_reg ← mem[pc] pc ← pc + 1				
2	rs ← i_reg[11 : 9] rt ← i_reg[8 : 6] alu_reg ← pc + addr -- 分岐先アドレスの計算				
3	alu_out ← rs 演算子 rt	alu_out ← rs + addr		pc ←alu_reg	pc ←tar_addr
4	register[i_reg[5:3]] ← alu_out	mem_data ← mem[alu_out]	mem[alu_out] ← register[rt]		
5		register[i_reg[8:6]] ← mem_data			

注：alu_out は ALU の計算結果、mem_data はメモリから読み出された値を示す

全ての命令においてステップ 1 の命令フェッチと、ステップ 2 の命令デコードとレジスタのフェッチは共通している。

命令のフェッチ(ステップ 1)ではメモリから命令を読み出し、命令レジスタに格納する。これは上述したように、マルチサイクル方式では実行中の命令が終わるまで、その命令を保持する必要

があるからである。さらにプログラムカウンタを次の命令のアドレスを指すように繰り上げる。

命令のデコードとレジスタのフェッチ(ステップ 2)では、読み出すレジスタのレジスタ番号を振り分ける。また、分岐先のアドレスを計算する。この段階では現在実行中の命令がどの命令かはまだわかっていない。しかし、たとえ分岐命令でなくても分岐先アドレスを計算しておく。分岐先アドレスを前もって計算しておいても実害はないからである。

この先から各命令によって処理の仕方が変わってくる。まず、R 形式の命令では、ステップ 2 で読み出された二つのレジスタの内容が ALU によって計算される。そしてステップ 4 でその結果がレジスタに格納される。

ロード命令とストア命令ではステップ 3 でメモリの読み出し先や書き込み先のアドレスを計算する。その後、ロード命令はステップ 4 でメモリから計算されたアドレスを読み出し、ステップ 5 でレジスタに格納する。ストア命令ではステップ 4 で、レジスタから読み出した値を計算したメモリアドレスの場所に格納する。

また、分岐命令ではステップ 2 で計算した分岐先アドレスを、ステップ 3 で次のプログラムカウンタに代入する。

ジャンプ命令ではステップ 3 で次のプログラムカウンタにジャンプ先のアドレスを格納するのみである。

3.4 シミュレーションによる動作検証と考察

本研究では XILINX 社の FoundationISE4.1 を用いて HDL 設計を行い、MTI 社の ModelSimXE によるゲートレベルシミュレーションを行った。また、ゲートレベルシミュレーションを行う際の FPGA は「Virtex2」を想定した。

ゲートレベルシミュレーションには以下の 2 種類がありそれぞれのレベルにおいてシミュレーションを行った。

- Post-Map シミュレーション

実装する回路を FPGA 上の回路資源に対応付ける際のシミュレーション。FPGA に各 CLB を配置する前のゲートレベルシミュレーションである。

- Post-Place&Route シミュレーション

配置配線後のゲートレベルシミュレーションで、FPGA にダウンロードした場合にもこの結果と同様の性能が期待できる。FPGA の各 CLB 間の遅延や、スイッチングの遅延も含まれるゲートレベルシミュレーションである。

シミュレーションを行う際に使用するテストパターンとして「1 から 100 までの和」、「5 つ数の中の最大値」、及び「ユークリッド互助法による 2 数の最大公約数」を求めるプログラムを作成し、設計したマイクロプロセッサの性能を評価するために、命令実行時間、クロック数、CPI の測定を行った。以下の表 4 にそれぞれのテストパターンで与えたプログラムの命令数を示す。

表 4　テストプログラムの命令数

1 から 100 までの和	5 つ数の中の最大値	2 数の最大公約数
407	38	70

マルチサイクルマイクロプロセッサのゲートレベルシミュレーションにおける最短クロックサイクルと各種遅延を表 5 に示す。

表 5　遅延と最短クロックサイクル

	最短クロックサイクル	制御ユニット安定遅延	メモリ読み出し遅延
Post-Map	24	6～7	1～2
Post-Place&Route	54	13～15	15～16

(単位：ns)

マルチサイクルマイクロプロセッサは、ステップ 1 でのクロックの立ち上がりから制御ユニットの出力が安定するまでの遅延と、制御ユニットの出力によってメモリから読み出される値の遅延とが、クロックサイクルに大きく影響していると考えられる。ステップ 1 でのクロックの立ち上がりで命令フェッチの制御線が出力される。その制御線の出力に従いメモリからは命令が読み出され、同じクロックの立下りで命令レジスタに書き込まれる。このため、クロックの立ち上がりから同じクロックの立下りまでにメモリから読み出される値が安定していないと、正しい値が命令レジスタに書き込まれないと考えられる。

2 種類のゲートレベルシミュレーションを比べると、実際に FPGA の CLB 間の遅延やスイッチングの遅延がある Place&Route シミュレーションの方が制御ユニット安定遅延、メモリ読み出し遅延ともに大きくなり、その結果最短クロックサイクルも長くなった。

従って、本研究で設計したマルチサイクルマイクロプロセッサではクロックが 1 になっている時間が制御ユニットからの出力遅延とメモリ読み出し遅延を加えたもの以上のクロックサイクルで正常な動作が期待できる。

また、ゲートレベルシミュレーションによって得られた各種結果を以下の表 6 に示す。

表 6　マルチサイクルマイクロプロセッサのシミュレーション結果

	実行時間(ns)		クロック数	CPI
	Map	Place&Route		
1 から 100 の和	32428	77058	1427	3.51
5 つの数の最大値	3552	7992	148	3.90
2 数の最大公約数	5880	13230	245	3.50

（実行時間：ns、クロック数：回）

マルチサイクルマイクロプロセッサは、1 つの命令を複数のクロックサイクルで実行するため、CPI は 3.5～3.9 の間の数値が得られた。5 つの数の最大値の CPI が 3.9 と 3 つのテストパターン中最大なのは、命令列中にローと命令を多く使用しているからだと考えられる。設計したマルチサイクルマイクロプロセッサは、条件分岐とジャンプ命令に 3 クロックサイクル、算術論理演算命令とストア命令に 4 クロックサイクル、ロード命令に 5 クロックサイクルを要する。すなわち、ロード命令を多用するほど、CPI が大きくなるということがわかる。

また、実行時間の考察は 5 章の「プロセッサの性能比較と考察」で行う。

4　パイプラインマイクロプロセッサの設計

4.1　単一サイクルマイクロプロセッサデータパスのパイプライン化

パイプラインマイクロプロセッサを設計する際、単一サイクルマイクロプロセッサのデータパスを利用する。図 6 に、元となる単一サイクルマルチプロセッサのデータパスを示す。

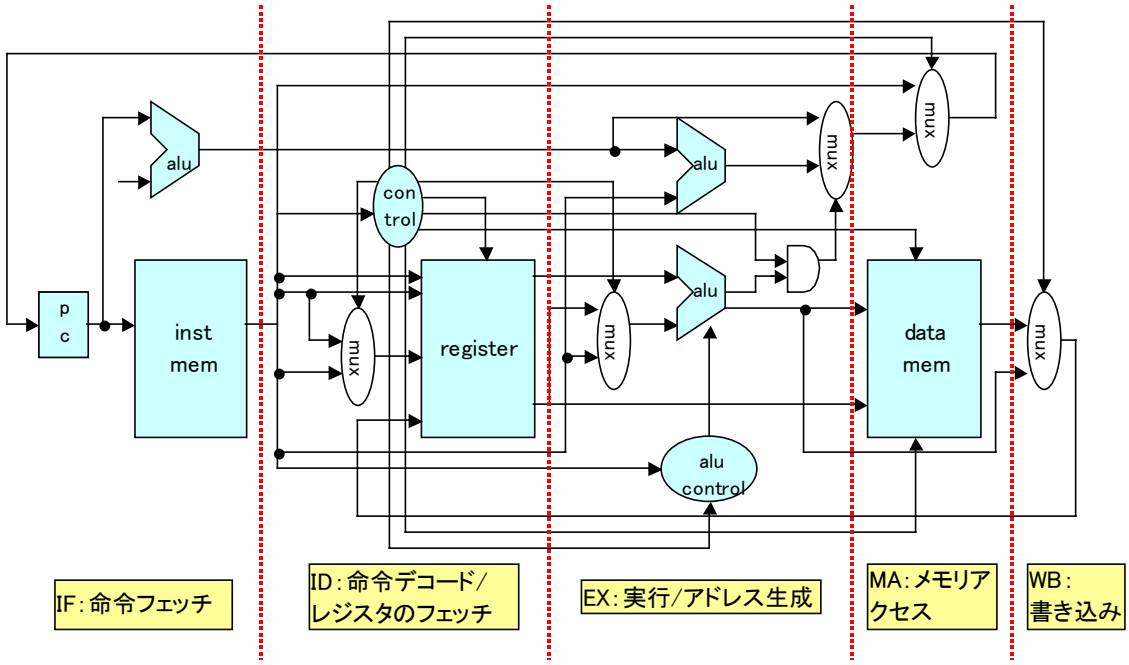

図 6 単一サイクルマイクロプロセッサのデータパス

図 6 のように、単一サイクルマイクロプロセッサのデータパスを 5 つのステージに分け、パイプラインマイクロプロセッサを実現する。

以下、単一サイクルマイクロプロセッサのデータパスのパイプライン化を順を追って説明する。

4.1.1　パイプラインレジスタの追加

2 章でも述べたようにパイプライン方式のマイクロプロセッサとは、複数の命令の各ステージを少しずつずらして実行することにより、同時並列的に命令を実行するものである。

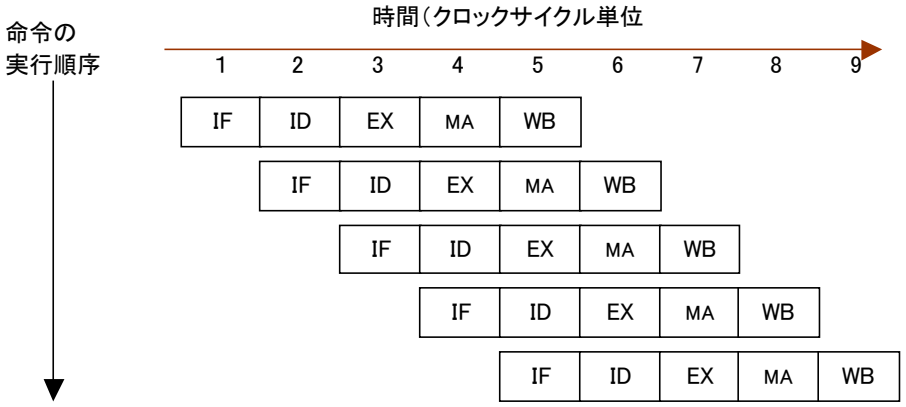

図 7 パイプラインマイクロプロセッサの命令実行の様子

図 7 のようにパイプライン方式では、1 クロックサイクルで 1 つのステージを実行する。そのため、命令メモリから読み出した命令を残りの 4 つのステージで利用するためには、その値をレジスタに保持しておく必要がある。なぜなら、1 つ目の命令が ID ステージに移ると同時に、次の命令が IF ステージに投入されるからである。このことは、パイプラインの各ステージに当てはまる。

図 8 のように単一サイクルのデータパスにパイプラインレジスタを追加する。四つのパイプラインレジスタは前後のステージの名前を付ける[1]。

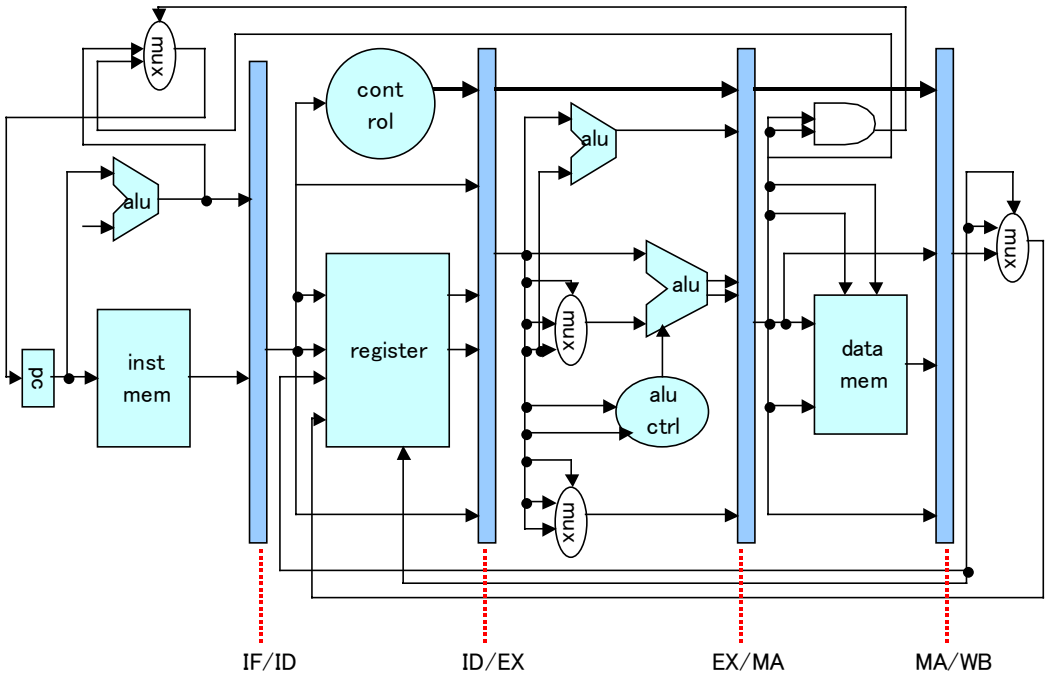

図 8 パイプラインレジスタを追加した単一サイクルデータパス

パイプラインレジスタは次のステージへと移って行くデータを全て記憶できるだけの幅が必要である。また、後のクロックで別の命令が必要とする可能性のあるデータを全て書き込まなければならない。制御ユニットからでる制御線は ID ステージで生成しておかなければならない。なぜなら、各命令の制御は EX ステージから始まるからである。ID ステージで生成された制御線は命令とともにパイプライン中を進み、該当するパイプラインステージで使用される[1]。

上記までのパイプラインマイクロプロセッサのデータパスでは次のクロックサイクルで次の命令を実行できない事態が起こりうる。そのような現象をハザードと呼ぶ[1]。以降はパイプラインのデータハザード、制御ハザードについて説明する。

4.1.2 データハザードとフォワーディング

データハザードとは、まだパイプライン中にある命令の出す結果を、他の命令が使おうとする場合におきる。このように、各命令に依存関係がある場合に、データハザードは生じる。

以下に、依存関係のある場合の命令列を示す。

```
sub $2, $1, $3        // $1 と$3 の減算の結果を$2 に収める
and $4, $2, $5        // 第 1 ソースオペランド($2)が sub 命令に依存
or $5, $2, $5         // 第 1 ソースオペランド($2)が sub 命令に依存
add $6, $2, $2        // 第 1、第 2 ソースオペランド($2)が sub 命令に依存
sw $7, 1($2)          // アドレス($2)が sub 命令に依存
```

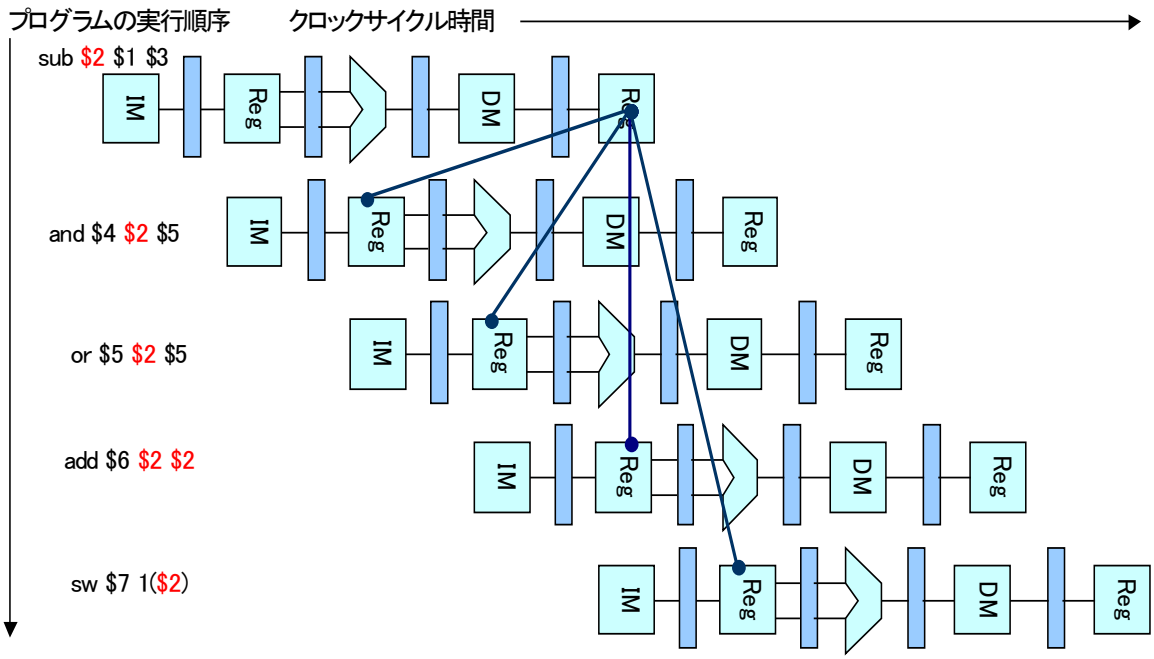

図 9 データの依存関係

図 9 は各クロックサイクルで使用されているの物理資源を表したものである。縦軸はプログラムの実行順序を命令単位で示しており、横軸は時間をクロックサイクル単位で示している。図中の線はデータの依存関係の様子を示すものである[1]。

最初の命令は$2 に書き込む。その後の命令は全て$2 を読み出すが、sub 命令では減算の結果の書き込みは WB ステージ(ステージ 5)で行われるためそれ以前に、減算の結果を読み出そうとしても正しい値を読み出すことは出来ない。

すなわち、クロックサイクル 5 以降でないと正しい減算の結果を読み出すことは出来ない。この場合は、and 命令、or 命令で正しい減算の結果を使用できないということになる。線が時間をさかのぼる方向をさしている場合データハザードが生じているという。

この問題を解決する方法はフォワーディングと呼ばれる、データの先送りである。sub 命令が EX ステージで行う減算の結果が EX/MA パイプライン及び、MA/WB パイプラインから得ることが出来れば、後続の命令は正しい計算結果を使用することが出来る。フォワーディング後のデータの依存関係を図 10 に示す。

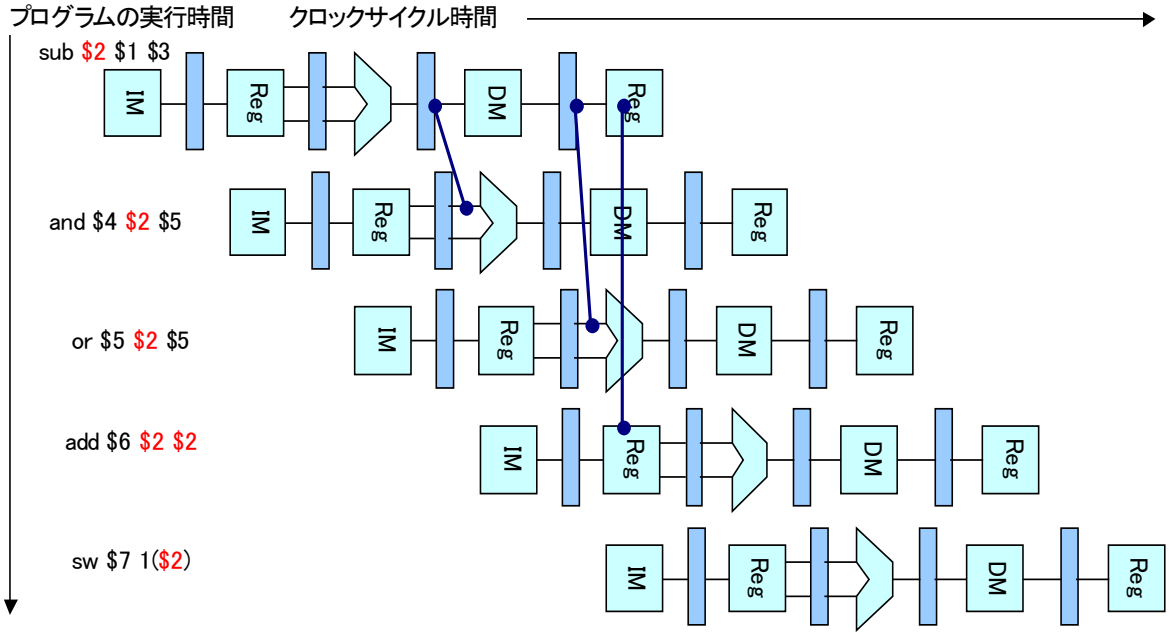

図 10 フォワーディング後のデータの依存関係

このようにパイプラインレジスタ内にある値を後続の命令が必要とする ALU 入力にフォワーディングすればパイプラインをストールせずに済む。

4.1.3　データハザードの制御：ストール

フォワーディングではデータハザードを救済できない場合がある。それはロード命令が書き込むのと同じレジスタを直後の命令が読み出そうとするときである。

このような命令の例を以下に示す。

```
lw $2, 1($0)          // $2 にメモリの 1 番地の値をロード
and $3, $2, $5        // 第 1 ソースオペランド($2)が lw 命令に依存
or$7, $2, $6          //第 1 ソースオペランド($2)が lw 命令に依存
sub $5, $5, $2        // 第 2 ソースオペランド($2)が lw 命令に依存
```

ロード命令はレジスタに書き込む値が得られるのは MA ステージ(ステージ 4)の後になってからである。この場合、データをフォワーディングをしても次の and 命令の ALU 入力には間に合わない。

このときのデータの依存関係を図 11 に示す。

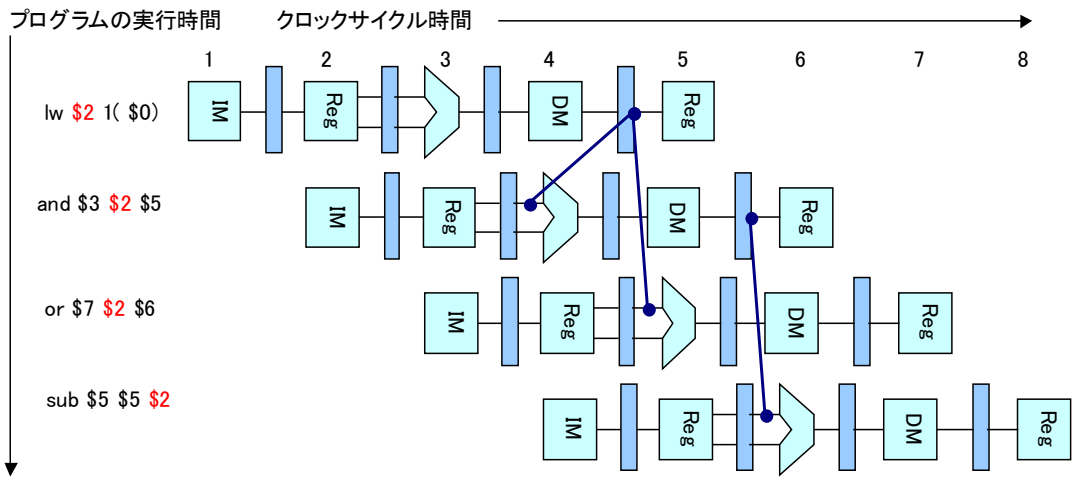

図 11 データの依存関係

クロックサイクル 4 にいて lw 命令の次の and 命令は ALU での演算を行っている。その時、lw 命令はデータメモリからデータを読み出しているところである。従って and 命令は例えフォワーディングを行ってもデータが間に合わないことになる。この場合、パイプラインをストールさせなければならない。

パイプラインをストールさせたときの様子を図 12 に示す。

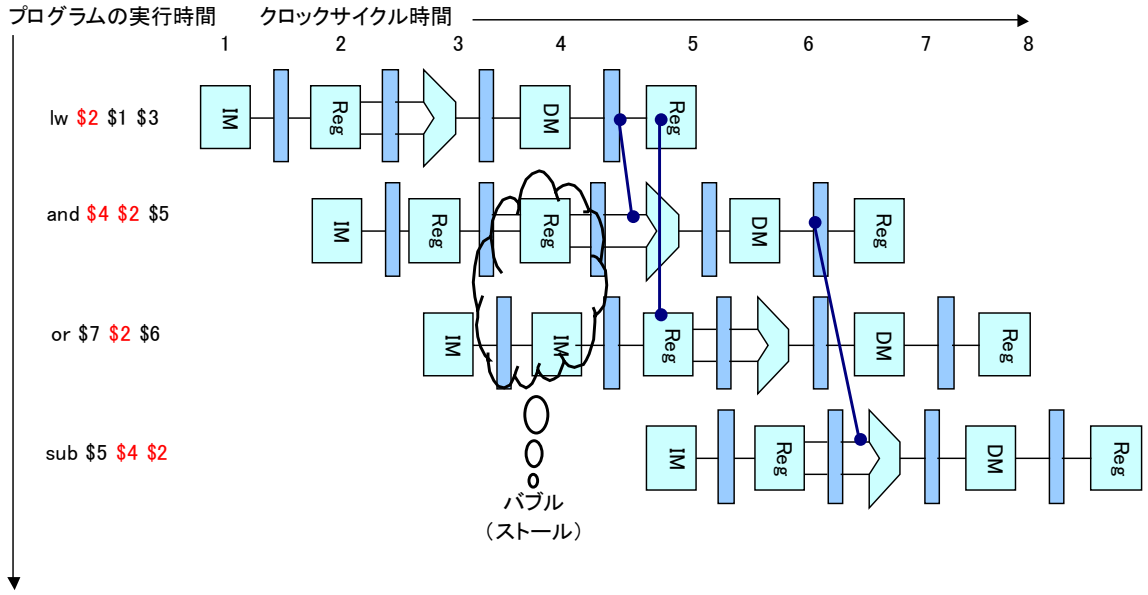

図 12 ストール挿入後のデータの依存関係

パイプラインをストールさせるとは、いったんパイプラインを停止させた後に再びパイプライン処理を行うことである。すなわち、lw 命令の後の and 命令の ID ステージ及び、or 命例の IF ステージを 2 回繰り返すことでパイプラインをストールさせることができる。この処理により、MA ステージ以降で正しい値が得られるロード命令のデータを and 命令、or 命令で使用することができる。

4.1.4 分岐ハザード

分岐ハザードは制御ハザードとも呼ばれる。ある命令の実行に対する判断を、まだ実行中の他の命令の結果に基いて下さなければならない場合に生じる。分岐命令に続く命令を正しく実行しようとすれば、分岐判定が下されるまでパイプラインをストールさせ続けなければならない。しかし、それではパイプラインの性能が極端に落ちてしまう。そのために、分岐命令の場合でも分岐しないと仮定して後続の命令をフェッチし続け、分岐が成立した場合にのみパイプラインをストールさせる方法を取る。

例えば以下のようなプログラムがあった場合のパイプラインの実行の様子を示す。

```
1 ：beq $1, $2, 5          // $1 と$2 が等しければ命令 5 へジャンプ
2 ：and $4, $2, $5
3 ：or $5, $2, $5
4 ：sub $5, $6, $2
5 ：sw $7, 1($2)           // ジャンプ先命令
```

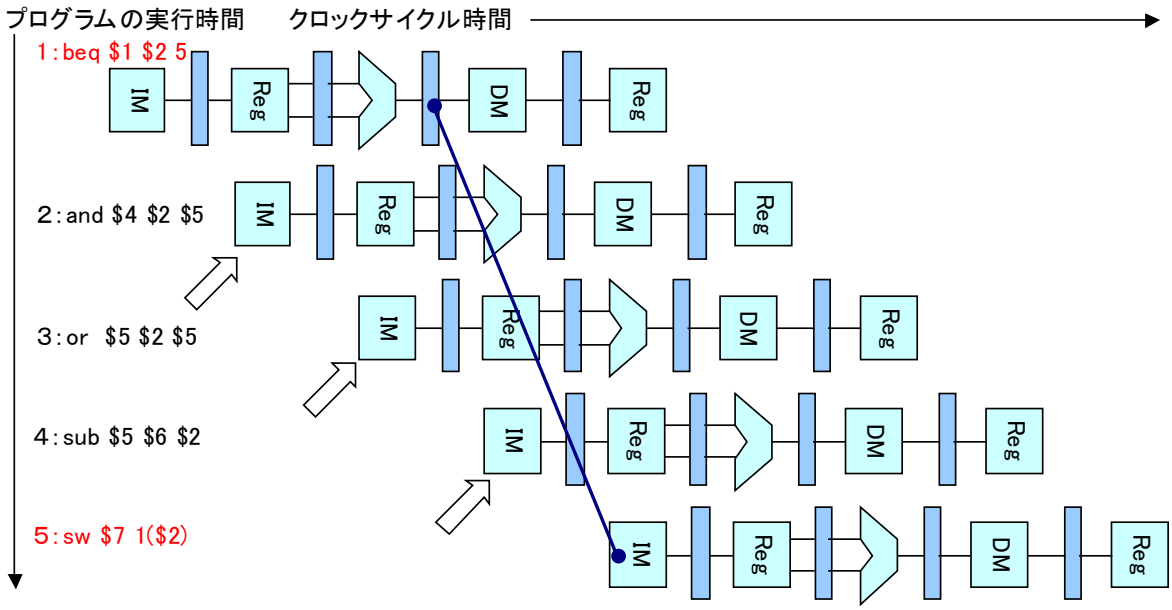

図 13 パイプラインに対する分岐命令の影響

命令１が分岐命例のとき、分岐するかどうかの判断は EX ステージ(ステージ 3)が終わるまでわからない。そのため、分岐が成立するとわかったときには既に後続の 3 つの命令がフェッチ済みであり、それらの命令が無駄になり余計なクロックサイクルを費やしてしまう。

そのような分岐によるパイプラインの無駄を少なくするために分岐判定を EX ステージ(ステージ 3)から ID ステージ（ステージ 2）に移行すれば、分岐するとわかったときにはフェッチされている命令は 1 つだけであり、無駄になる命令が削減できる。

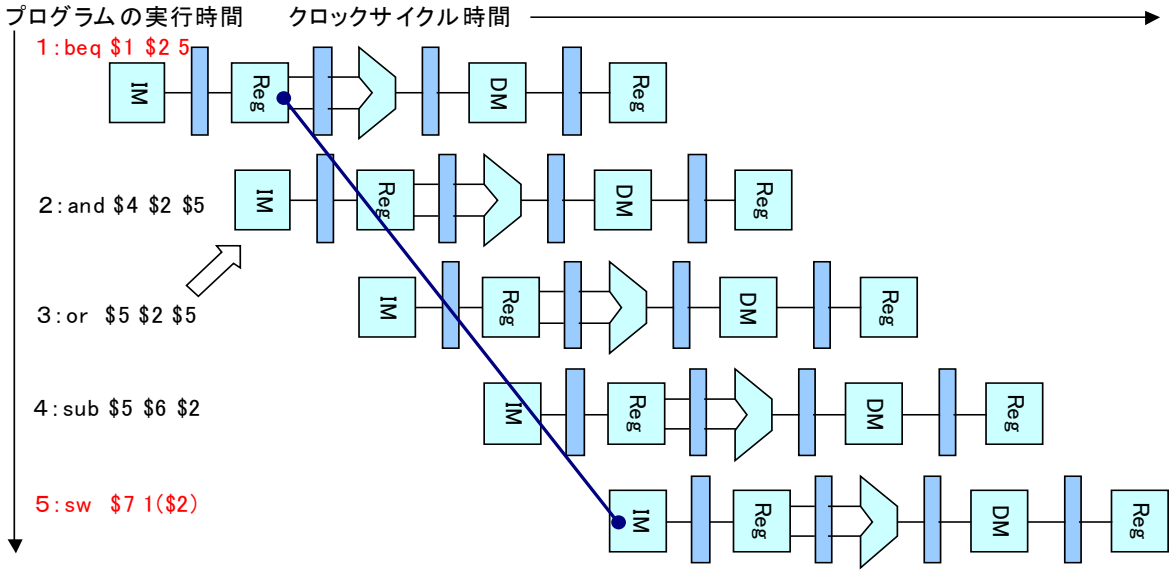

図 14 分岐判定移動後のパイプラインの動き

4.1.5　パイプラインマイクロプロセッサのデータパス

以上のようなパイプラインハザードを解消する機能を付け加えた最終的なパイプラインマイクロプロセッサのデータパスを図 15 に示す。

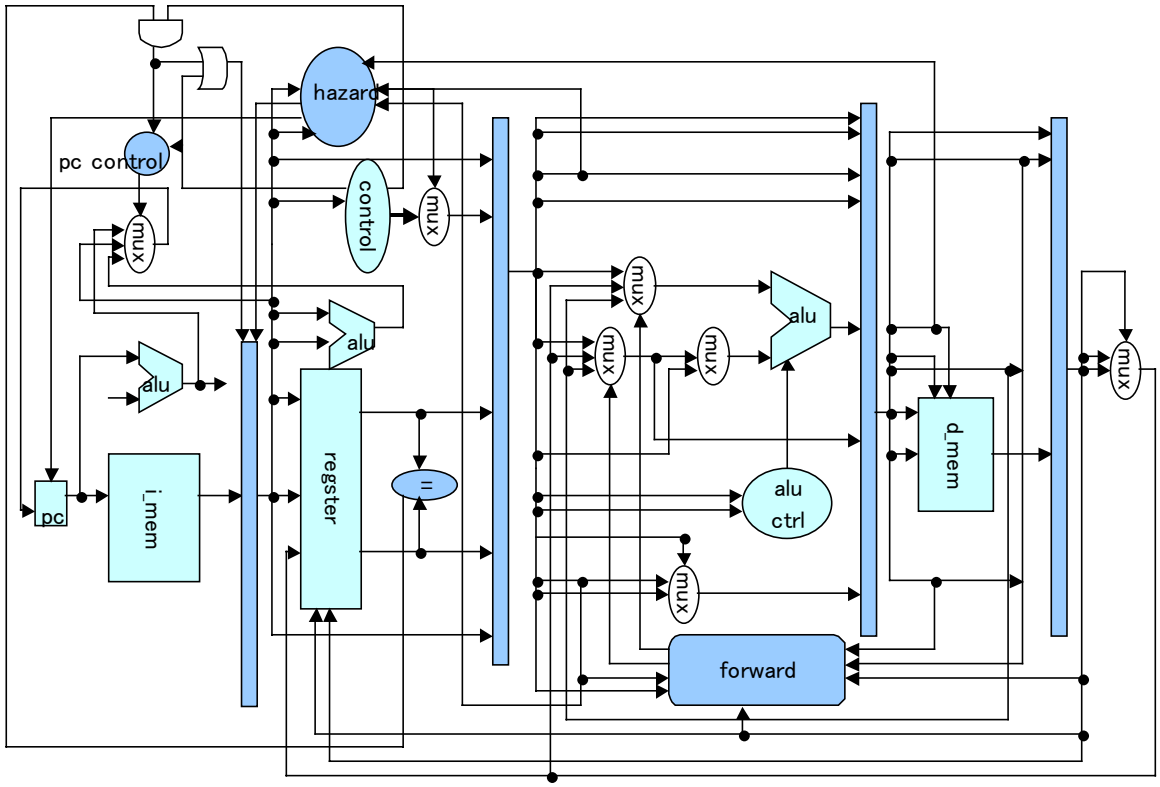

図 15 パイプラインマイクロプロセッサのデータパス

付け加えたユニットは以下の通りである。

- フォワーディングユニット(forward)
  ・・・フォワーディングをするかどうかの判定を行い、ALU の入力を制御する。
- ハザード検出ユニット(hazard)
  ・・・ハザードを検出し、パイプラインをストールさせる
- pc 制御(pc_control)
  ・・・プログラムカウンタの制御
- 分岐判定ユニット(=)
  ・・・EX ステージ(ステージ 3)から移動させた分岐判定専用ユニット

4.2　シミュレーションによる動作検証と考察

パイプラインマイクロプロセッサのゲートレベルシミュレーションにおける各種遅延と最短クロックサイクルを表 7 に示す。

表 7　パイプラインマイクロプロセッサの遅延と最短クロックサイクル

	パイプラインレジスタへの書き込み、及び読み出し遅延	最短クロックサイクル
Post-Map	6	12
Post-Place&Route	15	30

（単位： ns）

パイプラインマイクロプロセッサは各クロックの立下りでパイプラインレジスタに値を書き込み、次のクロックの立ち上がりでその値を使用する。クロックの立下りでパイプラインレジスタに書き込まれた値が、次のクロックの立ち上がりまでに安定していなければならない。それが安定していないと正しい値が各ステージで計算されずに、次のパイプラインレジスタへと書き込まれてしまう。よってパイプラインマイクロプロセッサは、各パイプラインレジスタへの書き込み、及び読み出し遅延が、クロックサイクルの長さを決める大きな要因であると思われる。

ゲートレベルシミュレーションによって得られた結果を以下の表 8 に示す。

表 8　パイプラインマイクロプロセッサのシミュレーション結果

	実行時間		クロック数	CPI	ストール回数
	Map	Place&Route			
1 から 100 の和	6144	15360	512	1.26	1
5 つの数の最大値	708	1770	59	1.55	11
2 数の最大公約数	1620	4050	135	1.90	46

（実行時間：ns、クロック数：回、ストール回数：回）

各テストパターンを比較すると、ストールの回数が 1 と少ない「1 から 100 までの和」では CPI が 1 に近く、理想的な結果になっている。それに対し、ストールの数が 46 と多い「2 数の最大公約数」のテストパターンでは他の 2 つのテストパターンと比べ CPI が大きくなってしまっている。

よって、パイプラインは命令の処理を非常に高速に行うことができるが、命令に依存関係があり、ストールを数多く行わなければならない場合、期待通りの高速化が期待できないことが確認できた。

また、実行時間の考察は 5 章の「プロセッサの性能比較と考察」で行う。

5 生成されたプロセッサの性能

5.1 プロセッサの性能比較と考察

本研究で作成したマルチサイクルマイクロプロセッサ、パイプラインマイクロプロセッサ、及び池田 修久氏により作成された単一サイクルマイクロプロセッサのハードウェア規模や、性能比較を行った。性能比較は Post-Map レベル、及び Post-Place&Route レベルでのゲートレベルシミュレーションにより行った。また、シミュレーションの際のテストパターンとしてマルチサイクル、パイプラインと同様に「1 から 100 までの和」、「5 つの数の最大値」、及び「ユークリッド互除法による 2 数の最大公約数」を求めるプログラムを使用した。

以下の表 9 に設計した単一サイクル、マルチサイクル、パイプラインの各マイクロプロセッサの 2 種類のゲートレベルシミュレーションにおける最短クロックサイクルを示す。

表 9　各プロセッサの最短クロックサイクル

	単一サイクル	マルチサイクル	パイプライン
Map	38	24	12
Place&Route	130	54	30

（単位：ns）

Map と Place&Route シミュレーションとも、共通してパイプラインのクロックサイクルが最短であった。マルチサイクルの最短クロックサイクルは理想的にはパイプラインと同じ長さであるが、パイプラインと比べ 2 倍ほど長くなっている。これは、3 章でも述べたように、マルチサイクルの命令実行の性質より、クロックの立下りで書き込まれた値を、次のクロックの立ち上がりで使用しなければならない場合があるため、これ以上クロックサイクルを短くすると正しい値が入らなくなるからであると考えられる。

また、Map シミュレーションと Place&Route シミュレーションを比較すると、実際の CLB 間の遅延を含まない Map シミュレーションの方がクロックサイクルが短くなった。この結果から、FPGA 上の CNB 間の遅延や、スイッチングにかかる遅延が大きいことがわかった。

以下の表 10 に各マイクロプロセッサのハードウェア規模を示す

表 10　各プロセッサのハードウェア規模

	フリップフロップ数	LUT	HDL 記述
単一サイクル	1157	3228	402
マルチサイクル	1223	2393	562
パイプライン	1337	3549	892

（FF 数、LUT 数：個、HDL 記述：行）

ハードウェア規模を比較するために、設計したマイクロプロセッサ中のフリップフロップとLUTの数を比較した。フリップフロップの数は単一サイクルと比べ、マルチサイクルの方が多くなっている。これは、マルチサイクルは命令を正しく実行するために、一時レジスタを使用しているからだと思われる。それに対し、ゲートの数を示すLUTの数はマルチサイクルの方が少なくなっている。これは、マルチサイクル方式は命令を複数のステップに分けて実行するため、機能ユニットを同じ命令中に複数回利用できるため、メモリやALUなどを半減させることが出来たためと思われる。

また、パイプラインマイクロプロセッサはパイプラインレジスタなど制御が複雑なためフリップフロップ、LUT共に最大となった。

次に、以下の表11～表13に3つのテストパターンでの各マイクロプロセッサの実行結果を示す。

表 11　1から１００での和：命令数 407

	実行時間		クロック数	CPI	ストール回数
	Map	Place&Route			
単一サイクル	15460	52910	407	1.00	—
マルチサイクル	32428	77058	1427	3.51	—
パイプライン	6114	15360	512	1.26	1

（実行時間：ns、クロック数：回、ストール回数：回）

表 12　5つの数の最大値：命令数 38

	実行時間		クロック数	CPI	ストール回数
	Map	Place&Route			
単一サイクル	1444	4940	38	1.00	—
マルチサイクル	3552	7992	148	3.90	—
パイプライン	708	1770	59	1.55	11

（実行時間：ns、クロック数：回、ストール回数：回）

表 13　ユークリッド互除法による2数の最大公約数：命令数 70

	実行時間		クロック数	CPI	ストール回数
	Map	Place&Route			
単一サイクル	2660	9100	70	1.00	—
マルチサイクル	5880	13230	135	3.50	—
パイプライン	1620	4050	135	1.90	46

（実行時間：ns、クロック数：回、ストール回数：回）

また、表 14 に、各マイクロプロセッサの最短クロックサイクルの比率と、3 つのテストパターンでの実行時間における平均比率を示す。

表 14　各プロセッサの性能比

	単一サイクル	マルチサイクル	パイプライン
最短クロックサイクル	3	2	1
命令実行時間	3	5	1

命令実行時間は 3 つのテストパターンにおいてマルチサイクルが一番長くなっている。本来ならば、マルチサイクルの命令実行時間は、単一よりも短く、パイプラインよりも長くなるのが理想であるが、上記のような結果になったのは以下のような理由が考えられる。

先ず初めに、マルチサイクルの最短クロックサイクルが長いことである。本来、マルチサイクルのクロックサイクルは単一サイクルの 5 分の 1 程度が理想である。しかし、マルチサイクルは各クロックごとに制御ユニットを通り、その出力遅延とメモリの読み立ちし遅延がある。また、命令フェッチ（ステップ１）ではクロックの立ち上がりで値を出力し、同じクロックの立下りでその値を使用するので、その二つの遅延を加えたクロックサイクル以下では動作しないと考えられる。そのため、クロックサイクルが単一サイクルの 3 分の 2 程度にしか短く出来ずに、命令実行時間が長くなったものと思われる。

次に、MONI 命令セットの簡易性が上げられる。MIPS のサブセットとして独自で作成した命令セットは、命令語長が 16 ビットで命令数が 11 と限られている。また、最も複雑な命令が lw 命令であり、この命令は 5 ステップで完了する。そのため、単一サイクルのクロックサイクルの長さを短くすることが可能であり、マルチサイクルよりも実行時間の短縮に繋がったものと思われる。命令が今よりも複雑になり、より長いステップ数のかかる命令が多数含まれる命令セットでならば、マルチサイクルの方が命令の実行時間が短くなる可能性がある。

単一サイクルとパイプラインの実行時間などを比べると、パイプラインは理想的な高速化が得られた。以下の表 15 に、3 つのテストパターンの命令数と、各テストパターンを実行した場合のパイプラインのストール回数、及び単一サイクルに対するパイプラインマイクロプロセッサの速度向上比を示す。

表 15　単一サイクルとパイプラインの速度向上比

	命令数	ストール回数	速度向上比
1 から 100 までの和	407	1	3.44
5 つの数の最大値	38	11	2.79
2 数の最大公約数	70	46	2.25

（命令数：個、ストール回数：回）

本研究で設計したパイプラインマイクロプロセッサは 5 段でパイプライン化を行っているため、理想的には単一サイクルの 5 倍の速度向上が得られるはずである。しかし、実際には理想通りにはいかない。なぜなら、ステージ間のバランスが完全に取れていない場合や、パイプライン処理のオーバーヘッドなどがあるからである。それに、パイプラインに大きな影響を及ぼすのはストールの発生回数である。4 章で述べたようにストールはパイプラインを一時的に停止させることである。ストールが大量に発生するとパイプラインの速度向上比は極端に下がってしまう。

また、パイプラインに投入される命令の数が少ないとパイプラインは十分な性能を発揮することが出来ない。

以上のことを踏まえると、「1 から 100 までの和」のテストパターンでは命令数が多く、ストールの数が 1 回と少ないことで、より理想に近い速度向上が得られているといえる。逆に、「2 数の最大公約数」のテストパターンでは命令数が少ない割に、ストール回数が 46 と多いので、単一サイクルに比べると性能は 2.25 倍に留まっている。

このことから、パイプラインの性能を十分に発揮させるためには命令の数が多いプログラムでストール回数を如何にに少なくするかが重要になってくると思われる。

5.2 FPGA へのロードと検証

今回作成したマイクロプロセッサは全て内部でメモリを定義しており、FPGA に載せるデータの容量が大きくなってしまっている。そのために「Virtex2」と呼ばれる大きなチップを想定したゲートレベルシミュレーションを行ったため、FPGA 内の CLB 間の遅延が大きくなり、クロックサイクル数を短くするのに限界があった。今後、メモリを外部で定義し FPGA のロードデータを小さくすることで、より小さい規模の FPGA へのロードが可能になり、CLB 間の遅延が小さくなり、クロックサイクルを短くできるのではないかと思われる。

6　おわりに

本論文では、ハードウェア記述言語によるマルチサイクル、及びパイプラインマイクロプロセッサの設計について述べた。3 つのテストパターンを作成し、それぞれのゲートレベルシミュレーションによって動作検証を行った。また、その 3 つのテストパターンにより、単一サイクル、マルチサイクル、パイプラインの各マイクロプロセッサのハードウェア規模や最短クロックサイクル、命令実行時間、CPI などの観点から性能比較を行った。

その結果、ハードウェア規模に関してはゲート数を示す LUT においてはマルチサイクルマイクロプロセッサが最小になった。また、パイプラインマイクロプロセッサは制御が複雑なためデータパスも複雑化し、ハードウェア規模は最大になった。また、各マイクロプロセッサの性能比較では単一サイクル、マルチサイクル、パイプラインの実行時間の比率が、３：５：１となった。これは、マルチサイクルの最短クロック数が単一サイクルの、3 分の 2 と理想である 5 分の 1 程度よりも長かったためと考えられる。単一サイクルとパイプラインマイクロプロセッサを比較すると、理想的な結果を得ることが出来た。ハザードの回数が少なければ、3.44 倍の速度向上が得られた。これにより、単一サイクルのデータパスをパイプライン化することで、高速化が達成されたと思われる。

今後の課題としては、先ず、マルチサイクルマイクロプロセッサの命令実行速度を向上させる検討が必要である。今回作成したマルチサイクルマイクロプロセッサは、制御ユニット出力遅延とメモリ読み出し遅延がボトルネックとなりクロックサイクルを短くすることが出来なかった。マルチサイクルマイクロプロセッサの性能を出すためには複雑な命令セットを作成し、実行することが考えられる。もうひとつの課題としては、本研究で作成した各マイクロプロセッサを FPGA にダウンロードして動作検証を行うことである。そのためには、メモリを内部で定義するのではなく、外部で定義し、メモリアクセスユニットなどの設計を行い設計データをより小さくすることが必要になってくる。そうすることで、ロードする FPGA のチップが小さくなり、各 CLB 間の遅延も小さくなると考えられる。さらに、パイプラインよりも複雑な制御を行い高速化を図るマイクロプロセッサの設計も課題としてあげられる。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導をいただきました山崎勝弘教授、西村俊和教授に深く感謝いたします。また、本研究に関して貴重なご意見をいただきました、Tran So Cong 氏、松井 誠二氏、同じハードウェアグループの池田 修久氏、牧岡 幸一氏、及び色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]JohnL.Hennessy,David A.Patterson 著,成田光章訳: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 VerilogHDL 設計入門,CQ 出版社,2001.
[3]立命館大学 VLSI センター:社会人向け VLSI 設計セミナー レクチャー用マニュアル,2001.
[4]桜井至:HDL によるデジタル設計の基礎,テクノプレス,1998.
[5]並木俊明,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.
[6]長谷川裕恭:VHDL によるハードウェア記述入門,CQ 出版,1999.
[7]嶋正利: プロセッサの 25 年,電子情報通信学会誌,Vol.82,No.10,pp.997-1017,1999.
[8]曽和将容:コンピュータアーキテクチャの原理,コロナ社,1994.
[9]アスキーデジタル用語辞典, http://yougo.ascii24.com/
[10]田中義久:ハードウェア記述言語による FPGA 上への教育用マイクロプロセッサ,立命館大学理工学部卒業論文,1998.
[11]田中義久:C 言語からのハードウェア自動生成システムの構築,立命館大学理工学研究科修士論文,2000.
[12]松原哲雄: VHDL 記述によるハードウェア設計,立命館大学理工学部卒業論文,2001.

付録A ソースプログラム

A.1. マルチサイクルマイクロプロセッサの HDL 記述

```
//-----------------------------------------top module ----------------------------------------
`timescale 1ns/1ns
module moni_pro(result, rst, clk, set_data, set_addr);

        input rst,clk;
        input [15:0] set_data;
        input [4:0] set_addr;
        output [15:0] result;

memun2 (
        .mem_data(mem_data),
        .mem_addr(mem_addr),
        .read_data2(read_data2),
        .mem_write(mem_write),
        .clk(clk),
        .rst(rst),
        .set_addr(set_addr),
        .set_data(set_data),
        .result(result)
        );

        control un3 (
                .pc_write(pc_write),
                .pc_writecond(pc_writecond),
                .iord(iord),
                .mem_write(mem_write),
                .ir_write(ir_write),
                .memto_reg(memto_reg),
                .pc_source(pc_source),
                .alu_op(alu_op),
                .alu_srcb(alu_srcb),
                .alu_srca(alu_srca),
                .reg_write(reg_write),
                .reg_dst(reg_dst),
```

```
          .opcode(opcode),
          .clk(clk),
          .rst(rst));
          i_reg un4 (.ir(ir),
          .mem_data(mem_data),
          .ir_write(ir_write),
          .clk(clk),
          .rst(rst)
          );

alu un5 (
          .alu_out(alu_out),
           .zero(zero),
           .alu_code(alu_code),
           .alu_in1(alu_in1),
           .alu_in2(alu_in2)
          );

alu_control un6 (
                    .alu_code(alu_code),
                    .funct(funct),
                    .alu_op(alu_op));

mux_1 un7 (
          .mem_addr(mem_addr),
          .pc(pc),
          .aluout_cut(aluout_cut),
          .iord(iord)
          );

mux_2 un8 (
          .rt_i(rt_i),
          .rd_r(rd_r),
          .reg_dst(reg_dst),
          .write_reg(write_reg)
          );
```

```
mux_3 um9 (
          .write_data(write_data),
          .alu_tmp(alu_tmp),
          .memto_reg(memto_reg),
          .mem_data(mem_data)
          );

mux_4 un10 (
          .alu_in1(alu_in1),
          .pc(pc),
          .read_data1(read_data1),
          .alu_srca(alu_srca)
          );

mux_5 un11 (
          .alu_in2(alu_in2),
          .read_data2(read_data2),
          .addr(addr),
          .alu_srcb(alu_srcb)
          );

mux_6 un12 (
          .next_16(next_16),
          .pc_source(pc_source),
          .aluout_reg(aluout_reg),
          .tar_addr(tar_addr),
          .alu_out(alu_out),
          .zero(zero),
          .ir(ir)
          );
pc un13 (
          .pc(pc),
          .next_pc(next_pc),
          .pc_control(pc_control),
          .rst(rst),
          .clk(clk)
          );
```

```
        register un14 (
                .read_data1(read_data1),
                .read_data2(read_data2),
                .read_reg1(read_reg1),
                .read_reg2(read_reg2),
                .write_reg(write_reg),
                .write_data(write_data),
                .reg_write(reg_write),
                .clk(clk),
                .rst(rst)
                );

        aluout_reg un15(
                .alu_out(alu_out),
                .aluout_reg(aluout_reg),
                .clk(clk)
                );

        alu_tmp un16 (
                .alu_out(alu_out),
                .alu_tmp(alu_tmp),
                .clk(clk));

assign aluout_cut = alu_tmp[4:0];
        assign next_pc = next_16[4:0];
        assign pc_control = ((zero && pc_writecond)||pc_write);
        assign funct = ir[2:0];
        assign read_reg1 = ir[11:9];
        assign read_reg2 = ir[8:6];
        assign rt_i = ir[8:6];
        assign rd_r = ir[5:3];
        assign tar_addr = ir[11:0];
        assign addr = ir[5:0];
        assign opcode = ir[15:12];
endmodule

//------------------------------------------- alu -------------------------------------------
```

```
`timescale 1ns/1ns
module alu (alu_out, zero, alu_code, alu_in1, alu_in2);
        input    [15:0]   alu_in1, alu_in2;
        input    [2:0]            alu_code;
        output [15:0]     alu_out;
        output                    zero;
        parameter [2:0]   ADD = 3'b010,
                          SUB = 3'b100,
                          AND = 3'b000,
                          OR  = 3'b001,
                          BIG = 3'b111,
                          SLT = 3'b011;

        wire [15:0]               alu_out;
        wire                      zero;

        assign   alu_out =
                 (alu_code==AND)? (alu_in1 & alu_in2):
                 (alu_code==OR )? (alu_in1 | alu_in2):
                 (alu_code==ADD)? (alu_in1 + alu_in2):
                 (alu_code==SUB)? (alu_in1 - alu_in2):
                 ((alu_code==BIG)&&(alu_in1 <= alu_in2))? alu_in2:
                 ((alu_code==BIG)&&(alu_in1 > alu_in2))? alu_in1:
                 ((alu_code==SLT)&&({alu_in1[15],alu_in2[15]}==2'b00)&&
                 (alu_in1 <= alu_in2))?     1:((alu_code==SLT)&&
                 ({alu_in1[15],alu_in2[15]}==2'b00)&&
                 (alu_in1 > alu_in2))?0:((alu_code==SLT)&&
                 ({alu_in1[15],alu_in2[15]}==2'b11)&&
                 (alu_in1 <= alu_in2))?     0:
                 ((alu_code==SLT)&&
                 ({alu_in1[15],alu_in2[15]}==2'b11)&&
                  (alu_in1 <= alu_in2))?    1:
                 ((alu_code==SLT)&&({alu_in1[15],alu_in2[15]}==2'b01))?0:
                 ((alu_code==SLT)&&({alu_in1[15],alu_in2[15]}==2'b10))? 1:       16'bx;
        assign   zero     =        (alu_in1==alu_in2)? 1 : 0;
endmodule
```

```
//--------------------------------------- alu control ----------------------------------------
`timescale 1ns/1ns
module alu_control (alu_code, funct, alu_op);
        input    [2:0]            funct;
        input    [1:0]            alu_op;
        output [2:0]              alu_code;

        reg      [2:0]            alu_code;
        wire [2:0]                funct;
        wire     [1:0]            alu_op;

        always @ (funct or alu_op)
                begin
                if(alu_op == 2'b00)
                        alu_code <= 3'b010;

                else if(alu_op == 2'b01)
                        alu_code <= 3'b100;

                else if(alu_op == 2'b10)
                        case ( funct )
                                3'b000 : alu_code <= 3'b010; //add
                                3'b001 : alu_code <= 3'b100; //sub
                                3'b010 : alu_code <= 3'b000; //and
                                3'b100 : alu_code <= 3'b001; //or
                                3'b111 : alu_code <= 3'b111; //big
                                3'b011 : alu_code <= 3'b011; //slt
                                default : alu_code <= 3'bzzz;
                        endcase
                 else
                        alu_code <= 3'bzzz;
                end
endmodule

//--------------------------------------- alu_tmp ----------------------------------------
module alu_tmp( alu_out, alu_tmp, clk);
```

```
        input [15:0] alu_out;
        input clk;
        output [15:0] alu_tmp;

        reg [15:0] alu_tmp;

        always @(negedge clk)
                alu_tmp <= alu_out;
endmodule

//-------------------------------------- alu_reg ----------------------------------------
module aluout_reg (alu_out, aluout_reg, clk);
        input [15:0] alu_out;
        input clk;
        output [15:0] aluout_reg;

        reg [15:0] aluout_reg;

        always @(negedge clk)
                aluout_reg <= alu_out;
endmodule

//-------------------------------------- control ----------------------------------------
`timescale 1ns/1ns
module control(pc_write, pc_writecond, iord, mem_write, ir_write,
memto_reg, pc_source, alu_op, alu_srcb, alu_srca, reg_write,reg_dst, opcode, clk, rst);

        input                   clk, rst;
        input    [3:0]          opcode;
        output                  pc_write, pc_writecond, iord, mem_write,
                                ir_write,memto_reg, alu_srca, reg_write, reg_dst, pc_source;
        output [1:0]      alu_op, alu_srcb;

        parameter [3:0] R = 4'b0000,
                        lw = 4'b0001,
                        sw = 4'b0010,
                        J = 4'b1000,
```

```
                B = 4'b0100;

        wire [3:0]      opcode;

        reg             pc_write, pc_writecond, iord, mem_write,
                        ir_write,memto_reg,alu_srca,reg_write,reg_dst, pc_source;
        reg [1:0]alu_op, alu_srcb;
        reg [3:0]       state;

always @(posedge clk)
        if (rst) begin
                pc_write<=1'b0; pc_writecond<=1'b0; iord<=1'b0;  mem_write<=1'b0;
                ir_write<=1'b0; memto_reg<=1'b0; pc_source<=1'b0; alu_op<=2'b00;
                alu_srcb<=2'b00; alu_srca<=1'b0; reg_write<=1'b0; reg_dst<=1'b0;state<=4'b0000;
        end

        else
        case(state)
        //---  fetch ---
        4'b0000 : begin
                pc_write<=1; pc_writecond<=0; iord<=0; mem_write<=0; ir_write<=1; memto_reg<=0;
                pc_source<=0; alu_op<=2'b00; alu_srcb<=2'b01;alu_srca<=0; reg_write<=0; reg_dst<=0;
                state <= 4'b0001;
                end

//--- decode ---
        4'b0001 : begin
                pc_write<=0;  ir_write<=0; alu_srcb<=2'b10;
                        case(opcode)
                                R : state <= 4'b0110; //R
                                B : state <= 4'b1000; //B
                                J : state <= 4'b1001; //J
                                lw : state <= 4'b0010; //lw
                                sw : state <= 4'b0010; //sw
                        endcase
                end
```

```
// --- operation ---
4'b0110 : begin // R3
          alu_op<=2'b10; alu_srcb<=2'b00; alu_srca<=1;
          state<=4'b0111;
          end

4'b1000 : begin //B3
          pc_writecond<=1; pc_source<=0; alu_op<=2'b00;
          alu_srcb<=2'b00; alu_srca<=1;
          state<=4'b0000;
          end

4'b1001 : begin // J3
          pc_write<=1; pc_source<=1; alu_srcb<=2'b00;
          state<=4'b0000;
          end

4'b0010 : begin // sw, lw3
          alu_srca<=1;
                    case(opcode)
                              lw : state<=4'b0011; //lw
                              sw : state<=4'b0101; // sw
                    endcase
          end

// --- memory access ---
4'b0111 : begin //R4
          reg_write<=1;
          reg_dst<=1; state<=4'b0000;
          end

4'b0101 : begin //sw4
          iord<=1; mem_write<=1; alu_srca<=1; alu_srcb<=2'b10;
          state<=4'b0000;
          end

4'b0011 : begin //lw4
```

```
                iord<=1;
                state<=4'b0100;
                end

        // --- memory read finish ---
        4'b0100 : begin //lw5
                memto_reg<=1; reg_write<=1;
                state<=4'b0000;
                end
        endcase
endmodule

//---------------------------------------- i_reg ----------------------------------------
`timescale 1ns/1ns
module i_reg ( mem_data, ir_write, ir,rst,clk);
        input [15:0] mem_data;
        input ir_write,rst,clk;
        output [15:0] ir;

        reg [15:0] ir;

        always @(negedge clk)
                if(rst)
                        ir <= 16'h00;
                else if(ir_write)
                        ir <= mem_data;

endmodule

//---------------------------------------- mem ----------------------------------------
`timescale 1ns/1ns
module mem ( mem_data, mem_addr, read_data2, mem_write,clk, rst, set_addr,set_data,result);

        input [15:0] read_data2, set_data;
        input [4:0] mem_addr, set_addr;
        input mem_write,clk, rst;
        output [15:0] mem_data, result;
```

```
        reg [15:0] memory [0:31];

        always @(posedge clk) begin
                if(rst)
                        memory[set_addr] <= set_data;
                else if(mem_write)
                        memory[mem_addr] <= read_data2;
                end
        assign mem_data = memory[mem_addr];
        assign result = memory[set_addr];
endmodule

//---------------------------------------- mux_1 ----------------------------------------
`timescale 1ns/1ns
module mux_1 ( mem_addr, pc, aluout_cut, iord);
        output [4:0] mem_addr;
        input [4:0] pc;
        input [4:0] aluout_cut;
        input iord;

        assign mem_addr = iord ? aluout_cut : pc;

endmodule

//---------------------------------------- mux_2 ----------------------------------------
`timescale 1ns/1ns
module mux_2 ( rt_i, rd_r, reg_dst, write_reg);
        input [2:0] rt_i;
        input [2:0] rd_r;
        input reg_dst;
        output [2:0] write_reg;

         assign write_reg = (reg_dst) ? rd_r : rt_i;

endmodule
//---------------------------------------- mux_3 ----------------------------------------
`timescale 1ns/1ns
```

```
module mux_3 ( alu_tmp, mem_data, memto_reg, write_data);
          input [15:0] alu_tmp;
          input [15:0] mem_data;
          input memto_reg;
          output [15:0] write_data;

          wire [15:0] alu_tmp;
          wire [15:0] mem_data;
          wire [15:0] write_data;

           assign write_data = memto_reg ? mem_data : alu_tmp;

endmodule
//--------------------------------------- mux_4 ----------------------------------------
`timescale 1ns/1ns
module mux_4 (alu_in1, pc, read_data1, alu_srca );
          input [4:0] pc;
          input [15:0] read_data1;
          input alu_srca;
          output [15:0] alu_in1;

          wire [4:0] pc;
          wire [15:0] read_data1;
          wire alu_srca;
          wire [15:0] alu_in1;

          assign alu_in1 = alu_srca ? read_data1 : pc;
endmodule
//--------------------------------------- mux_5 ----------------------------------------
`timescale 1ns/1ns
module mux_5 ( alu_in2, read_data2, addr, alu_srcb);
          input [15:0] read_data2;
          input [4:0] addr;
          input [1:0] alu_srcb;
          output [15:0] alu_in2;

          wire [15:0] read_data2;
```

```
        wire [4:0] addr;
        wire [1:0] alu_srcb;
        wire [15:0] alu_in2;

        assign alu_in2 = (alu_srcb == 2'b00) ? read_data2 :
                        (alu_srcb == 2'b01) ? 16'h01 : (alu_srcb == 2'b10) ? addr : 16'hzz;
endmodule

//--------------------------------------- mux_6 ----------------------------------------
`timescale 1ns/1ns
module mux_6 ( next_16, aluout_reg, tar_addr, pc_source, alu_out, zero, ir );
        input [15:0] aluout_reg, alu_out, ir;
        input [11:0] tar_addr;
        input pc_source, zero;
        output [15:0] next_16;

        //wire [15:0] alu_out;
        //wire [11:0] tar_addr;
        //wire pc_source;
        //wire [15:0] next_16;

        assign next_16 = pc_source ? tar_addr :
                (zero && (ir[15:12] == 4'b0100))     ? aluout_reg : alu_out;
endmodule

//---------------------------------------- pc ----------------------------------------
`timescale 1ns/1ns
module pc ( next_pc, pc_control, pc, rst,clk);
        input [4:0] next_pc;
        input pc_control, rst,clk;
        output [4:0] pc;

         reg [4:0] pc;

        always @(negedge clk) begin
                if(rst)
                        pc <= 5'b0;
```

```
                else if(pc_control)
                        pc <= next_pc;

end
endmodule

//-------------------------------------- register ----------------------------------------
`timescale 1ns/1ns
module register (read_data1, read_data2, read_reg1, read_reg2,
                write_reg, write_data, reg_write, clk, rst);

        input [2:0] read_reg1, read_reg2, write_reg;
        input [15:0] write_data;
        input reg_write, clk, rst;
        output [15:0] read_data1, read_data2;

        reg [15:0] reg_mem [0:7];

        always @(negedge clk)
                begin
                if(rst) begin
                        reg_mem[7] <= 16;//16'h0010;
                        reg_mem[0] <= 0;
                        end
                else if(reg_write)
                        reg_mem[write_reg] <= write_data;
                 end
        assign  read_data1 = reg_mem[read_reg1];
        assign  read_data2 = reg_mem[read_reg2];

endmodule
```

A.2. １から N の和を求めるテストパターンの HDL 記述

```
`timescale 1ns/1ns
module test_55;

        reg clk, rst;
```

```
        reg [4:0]   set_addr;
        reg [15:0]  set_data;
        wire [15:0] result;

        moni_pro un (
                .result(result),
                 .rst(rst),
                .clk(clk),
                .set_data(set_data),
                .set_addr(set_addr)
                )

        always #27 clk = ~clk;

        initial
                begin
                clk = 0; rst = 1; //mem_read = 1'b0; mem_write = 1'b1;

                #54 set_addr = 16; set_data = 0;
                #54 set_addr = 17; set_data = 1;
                #54 set_addr = 18; set_data = 101;
                #54 set_addr = 0; set_data= 16'b0001_111_001_000001;   //lw(i)
                #54 set_addr = 1; set_data= 16'b0001_111_100_000001;   //lw(1)
                #54 set_addr = 2; set_data= 16'b0001_111_010_000010;   //lw(n)
                #54 set_addr = 3; set_data= 16'b0001_111_011_000000;   //lw(0)
                #54 set_addr = 4; set_data= 16'b0100_001_010_000011;   //beq
                #54 set_addr = 5; set_data= 16'b0000_011_001_011_000;  //add
                #54 set_addr = 6; set_data= 16'b0000_001_100_001_000;  //add
                #54 set_addr = 7; set_data= 16'b1000_000000000100;     //jp
                #54 set_addr = 8; set_data= 16'b0010_111_011_001111;   //sw
                #54 rst = 1'b0;
                #54 set_addr = 31;
                        end
endmodule
```

A.3. N 個の数字の中の最大値を求めテストパターンの HDL 記述

```
`timescale 1ns/1ns
```

```
module test_saidai;

        reg             clk, rst;
        reg [4:0]       set_addr;
        reg [15:0]      set_data;
        wire [15:0]     result;

        wire [15:0]     ir, alu_out,aluout_reg, alu_in1, alu_in2, next_16, mem_data,
                        read_data1, read_data2, write_data, alu_tmp;

        moni_pro un (
                .result(result),
                 .rst(rst),
                .clk(clk),
                .set_data(set_data),
                .set_addr(set_addr)
                )

        always #12 clk = ~clk;

        initial begin
        clk = 0; rst = 1;//mem_read = 1'b0; mem_write = 1'b1;

        // integer
        #24 set_addr = 16; set_data = 1;
        #24 set_addr = 17; set_data = 21;
        #24 set_addr = 18; set_data = 4;

        // data
        #24 set_addr = 19; set_data = 1;
        #24 set_addr = 20; set_data = 2;
        #24 set_addr = 21; set_data = 3;
        #24 set_addr = 22; set_data = 4;
        #24 set_addr = 23; set_data = 5;


        // program
```

```
        #24 set_addr = 0;  set_data = 16'b0001_111_011_000001;  // lw: $3 = mem[33] address
        #24 set_addr = 1;  set_data = 16'b0001_111_100_000010;  // lw: $4 = mem[34] loop
        #24 set_addr = 2;  set_data = 16'b0001_111_101_000000;  // lw: $5 = mem[32] +-1
        #24 set_addr = 3;  set_data = 16'b0001_111_001_000011;  // lw: $1 = mem[35]
        #24 set_addr = 4;  set_data = 16'b0001_111_010_000100;  // lw: $2 = mem[36]
        #24 set_addr = 5;  set_data = 16'b0000_001_010_110_111; // sl: ($1>$2)?$6=$1:$6=$2
        #24 set_addr = 6   set_data = 16'b0100_110_010_000010;  // beq: if($6=&2) jumpto 10
        #24 set_addr = 7;  set_data = 16'b0001_011_010_000000;  // lw: &2 mem[$3]
        #24 set_addr = 8;  set_data = 16'b1000_000000_001010;   // j: jump to 10
        #24 set_addr = 9;  set_data = 16'b0001_011_001_000000;  // lw: $1 mem[$3]
        #24 set_addr = 10; set_data = 16'b0000_011_101_011_000; // add: $3 = $3 + 1
        #24 set_addr = 11; set_data = 16'b0000_100_101_100_001; // sub: $4 = $4 - 1
        #24 set_addr = 12; set_data = 16'b0010_111_110_001111;  // sw: mem[63] = $6
        #24 set_addr = 13; set_data = 16'b0100_100_000_000001;  // beq: if($4=$0) jumpto 16
        #24 set_addr = 14; set_data = 16'b1000_000000_000101;   // j: jumpto 5
        #24 set_addr = 15; set_data = 16'b1111_000000000000;    // halt
        #24 set_addr = 31;
        #24 rst = 1'b0;
end
endmodule
```

A.4. 最大公約数を求めるテストパターンの HDL 記述

```
`timescale 1ns/1ns
module test_kouyaku;

        reg clk, rst;
        reg [4:0]   set_addr;
        reg [15:0]  set_data;
        wire [15:0] result;

        moni_pro un (     .result(result),
                          .rst(rst),
                          .clk(clk),
                          .set_data(set_data),
                          .set_addr(set_addr))

        always #27 clk = ~clk;
```

```
        initial
        begin
        clk = 0; rst = 1; //mem_read = 1'b0; mem_write = 1'b1;

        // integer
        #54 set_addr = 16; set_data = 1;

        // data
        #54 set_addr = 17; set_data = 152;
        #54 set_addr = 18; set_data = 36;

        // program
        #54 set_addr = 0;  set_data = 16'b0001_111_001_000000;  //lw $1 = 1
        #54 set_addr = 1;  set_data = 16'b0001_111_010_000001;  //lw $2 = mem[33]
        #54 set_addr = 2;  set_data = 16'b0001_111_011_000010;  //lw &3 = mem[34]
        #54 set_addr = 3;  set_data = 16'b0000_010_011_010_001; //sub $2 = $2-$3
        #54 set_addr = 4;  set_data = 16'b0100_000_010_001000;  //beq if ($2==0) jump to 13
        #54 set_addr=5;set_data=16'b0000_010_000_100_011;       //sltif($2<0) then $4=1 else $4=0
        #54 set_addr = 6;  set_data = 16'b0100_100_001_000001;  //beq if($4==$1) then jump to9
        #54 set_addr = 7;  set_data = 16'b1000_000_000_000_011; //j jump to 3
        #54 set_addr = 8;  set_data = 16'b0000_010_011_010_000; //add $2 = $2+$3
        #54 set_addr = 9;  set_data = 16'b0000_010_000_101_000; //add $5 = $2+$0
        #54 set_addr = 10; set_data = 16'b0000_011_000_010_000; //add $2 = $3+$0
        #54 set_addr = 11; set_data = 16'b0000_101_000_011_000; //add $3 = $5+$0
        #54 set_addr = 12; set_data = 16'b1000_000_000_000_011; //j jump to 3
        #54 set_addr = 13; set_data = 16'b0010_111_011_001111;  //sw mem[63] = $3
        #54 rst = 1'b0;
        #54 set_addr = 31;
        end
endmodule
```