卒 業 論 文

ハードウェア記述言語による
単一サイクル/パイプラインマイクロプロセッサの設計

氏　　名　　：池田修久
学 籍 番 号　　：2210980014-0
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

本論文では、LSI 設計の主流となっているハードウェア記述言語を用いたトップダウン設計をマイクロプロセッサを設計する事で実装する。ハードウェア記述言語としては Verilog-HDL を用いる。Verilog-HDL は C 言語を元に開発され、ASIC 設計の記述性を重視した言語であり、シミュレーションの為の機能も充実している。設計するマイクロプロセッサとして最初に単一のクロック･サイクルで 1 命令を実行する単一サイクル方式マイクロプロセッサを設計する。続いて単一サイクル･データパスの 5 段パイプライン化を行う過程を述べ、実際にパイプライン方式マイクロプロセッサを設計する。設計したプロセッサは MONI 命令セットと名づけた独自命令セットによる動作を行う。MONI 命令セットは全 11 命令の命令セットである。MONI 命令セットを用いてテストプログラムを作成し、FPGA 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。テストプログラムとして作成した問題は和、最大値、最大公約数を算出するものである。また、本研究室の大八木氏により設計されたマルチサイクル･マイクロプロセッサを用いて、3 通りの命令実行方式の違いによるプロセッサの性能比較を行った。シミュレーションにより得られた結果はパイプライン処理を行う事で単一サイクルに比べ最大 3.4 倍の速度向上を得る事が出来た。また、マルチサイクルは単一サイクルに比べ 0.6 倍の速度向上という結果が得られた。本論文の最後にシミュレーションにより得られた結果を考察する。

目次

付録

図目次

表目次

1　はじめに

1970 年代、集積回路 LSI を使った製品は電卓、時計、ゲーム機といったパーソナル機器であった。80 年代に入り、DRAM の普及とともに設計規模がリニアに増大していった。90 年代半ばに 100k～500k ゲートの規模であったものが現在では 10 倍を超える数百万ゲートに達した。集積度が高まるにつれ、今までは単体の機能だけを考慮して設計していたものが、現在では携帯電話や電子手帳、テレビなどに見られるようにシステム自身を 1Chip 化するといった設計（SOC : System on Chip）がなされるようになり、設計規模が非常に大きくなっている[1]。

LSI 設計は、古くからコンピュータの導入による設計自動化が推進されていた分野である。それは 1970 年代の LSI を製造するためのレイアウト・マスク設計に関する自動化から始まった。80 年代になると、論理ゲートに基づいた回路図入力とゲートレベル・シミュレーションによる論理ゲートレベルの設計自動化が進んだ。しかし、ゲートレベル設計においても、設計する回路規模が数万ゲート以上になるとシミュレーションの実行時間が長くなり、設計ミスの発見と修正、再確認を行う設計サイクルでは、要求された開発期間を満足できなくなり、設計環境および設計フローの変化が必要であった[2]。

1990 年代になると、論理合成ツールの開発、そして実用化により HDL（Hardware Description Language）と呼ばれるハードウェア記述言語を用い、論理ゲートによる設計より上位レベルでの設計が可能になった。また、論理機能を焼き付けることで、その場ですぐに LSI として利用することが出来るプログラムの書き換え可能なゲートアレイ（FPGA : Field Programmable Gate Array）の誕生により、特定用途向け LSI(ASIC : Application Specific IC)等の開発段階においてのエミュレーションやプロトタイプの作成に利用され、開発コストの削減と開発期間の短縮に繋がった[1][2]。

LSI の発展とともに CPU も発展してきた。世界で最初の CPU は Intel 社の「4004」である。内部構造は現在の CPU と比べると驚くほど単純なもので、2,250 個のトランジスタを集積し、0.75MHｚのクロック周波数で動作する仕組みになっていた。データバス幅は 4 ビットで構成され、当時としては高速な 0.06MIPS の演算性能を持っているものであった。そして 2000 年には Intel 社より「PentiumⅣ」が発表され、約 4,200 万個のトランジスタを集積し 2GHｚのクロック周波数での動作、20 段のパイプライン、スーパースケーラなどの原理により CPU の高速化が実現されている[3][4]。

以上の様な背景を踏まえ、本研究ではハードウェア記述言語による単一サイクル/パイプライン命令実行方式によるマイクロプロセッサの設計を行う。ハードウェア記述言語を用いて設計を行うことでハードウェア記述言語によるトップダウン設計を理解し、単一サイクル/パイプラインプロセッサを設計することで CPU の命令実行の高速化の原理を理解する事を目的とする。

今回、設計した単一サイクル/パイプラインプロセッサは RISC プロセッサの代表である MIPS のサブセットに位置する。命令語長を 16 ビットとし、3 形式に分類される全 11

命令を実現する 16 ビットマイクロプロセッサである。本研究では仕様作成、動作レベル HDL 作成、動作レベルシミュレーション、RTL レベル HDL 作成、論理合成、最適化、テクノロジマッピング、ゲートレベル･シミュレーションの過程を経て設計を完了とした。ゲートレベル･シミュレーションを行う際、ある FPGA チップを想定し、シミュレーションを行った。本論文では、FPGA 上での実機検証は行っていない。FPGA 上での動作検証については第 5 章で述べる。

初めに、1 つの命令を 1 クロック･サイクルで実行完了する単一サイクル方式マイクロプロセッサの設計を行う。ゲートレベル･シミュレーション完了後、単一サイクル方式マイクロプロセッサを命令フェッチ、デコード、演算実行、メモリ参照、結果出力のパイプライン･ステージに分類し、5 段のパイプライン処理が可能なプロセッサへと高速化を実現する。また、今回設計した CPU は命令メモリ、データメモリをレジスタとして定義した。よって外部メモリとのアクセスは考えない事とする。

第 2 章でハードウェア記述言語によるシステム設計について述べる。第 3 章で単一サイクルマイクロプロセッサの設計について述べ、第 4 章においてパイプラインによる CPU の高速化の原理と、5 段パイプラインプロセッサの設計について述べる。第 5 章において生成されたプロセッサの評価と考察について述べる。

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によるトップダウン設計

ハードウェア記述言語(Hardware Description Language、以下 HDL)はハードウェア動作を論理ゲートレベルより抽象度の高いレベルで記述できる言語である。C 言語などのプログラミング言語の構文に似ているが、大きな違いは、「HDL は入力信号の変化に従って並行的に動作するような並列処理やタイミングなどが記述可能」であり、回路設計のための言語である。HDL を用いれば、論理ゲートによる設計より上位レベルでの設計が可能であり、仕様に即した形での動作の記述と検証を可能にする。HDL の適用範囲は広く、システムの抽象的なレベルから論理式レベル、ネットリスト･レベルまで、幅広い範囲での設計仕様の検証や記述を行うことが出来る[1][2]。

代表的な HDL として VHDL と Verilog-HDL の 2 種類が挙げられる。この 2 種類の HDL は、基本的な概念は同じである。両者の特徴を以下に記す[1][2]。

- VHDL
  - 設計仕様のドキュメント言語として開発
  - 読解性に優れ、システム仕様の曖昧さを排除した厳格な言語
- Verilog-HDL
  - C 言語を基にし、ASIC 設計の記述性を重視した言語
  - ゲートレベル･シミュレーションの為の機能も充実

VHDL と Verilog-HDL は、どちらかの HDL が今後支配的になるわけではなく、両者それぞれの特徴を生かした分野での使い分けが進行していくと予想される[1][2]。

HDL と優れた論理合成ツールの登場により LSI 設計フローは回路図入力･ゲートレベルシミュレーションの時代の設計フローであるボトムアップ設計からトップダウン設計へと推移していった。以下に HDL によるトップダウン設計のフローチャートを図 1 として示す[1]。

まず最初に仕様により得られた要求に即した動作を表現する HDL を動作レベル HDL として作成する。テストパターンにより動作レベル･シミュレーションを行い、仕様を満たす機能が得られるように HDL を修正する。動作レベル･シミュレーションが完了した所で動作レベル HDL から RTL（ RTL ： レジスタ間のデータの流れと、そのデータの設計による処理方法を記述するモデル）HDL へと変換する。RTL HDL を論理合成ツールに通し、ネットリスト(ゲートレベル HDL)を出力する。得られたネットリストを用いて配置配線を行い遅延情報ファイルを出力する。この時点でターゲットとするテクノロジは決定していなければならない。次に、論理合成により得られたネットリストと配置配線により得られた遅延情報を用い、ゲートレベル･シミュレーションを行う。ゲートレベル･シミュレーションの結果が仕様を満たすものであれば FPGA 等のテクノロジにデータをダウンロードし、実機テストを行う。これが HDL によるトップダウン設計である[1][2]。

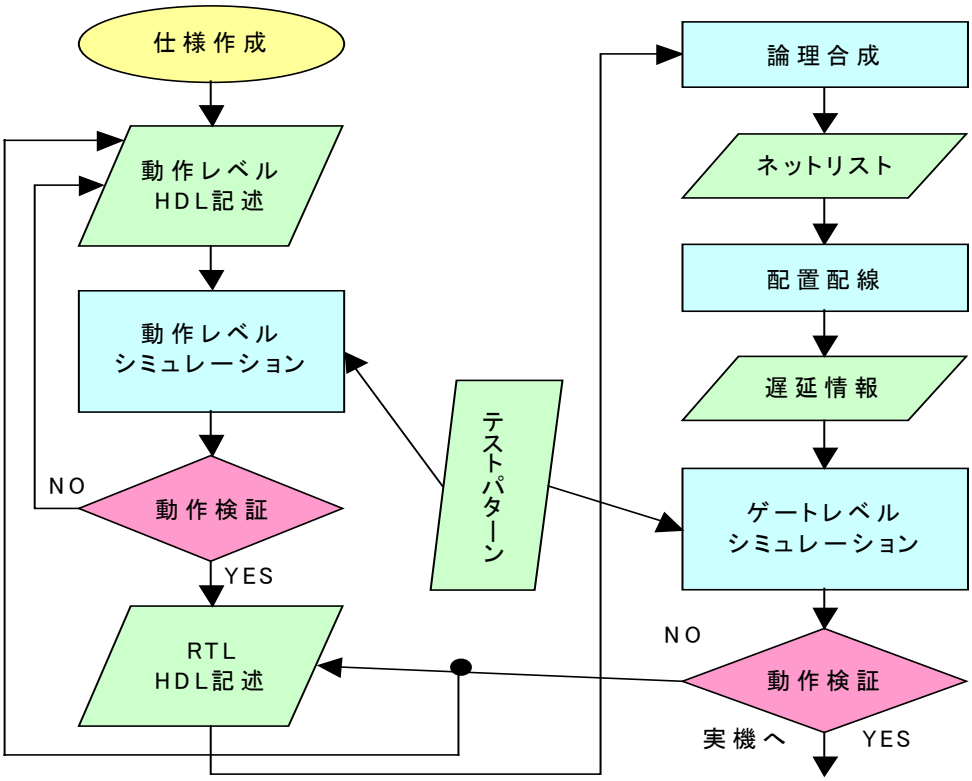

図 1：トップダウン設計のフローチャート

2.2 FPGA

FPGA（Field Programmable Gate Array）はプログラム書き換え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的ではなかった。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ている。

2.3 プロセッサアーキテクチャの分類と高速化の原理

コンピュータの中枢を成すマイクロプロセッサである CPU には命令実行方式の違いによりいくつかに分類することが出来る。ここではアーキテクチャ別の特徴と、CPU 高速化の原理について述べる[5][6][7]。

(1) 単一サイクル方式命令実行

単一サイクル方式は、1 つの命令を 1 クロック･サイクルで実行完了する。そのため、どのデータパス資源も 1 命令当たり 2 回以上の使用は出来ない。クロック･サイクルは命令の最長パスによって決まる(ロード命令)。効率が悪いので現代のコンピュータで実際に採用

されることはまず無い。当然の事ながら CPI は 1 である。本論文の第 3 章にて詳しく述べる。

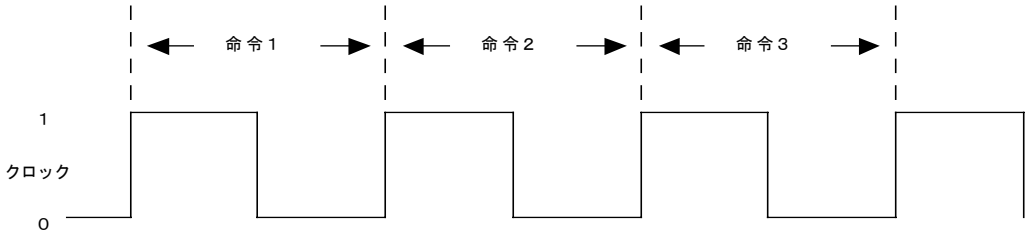

図 2：単一サイクル方式命令実行

(2) マルチサイクル方式命令実行

マルチサイクル方式は、命令の各実行ステップ(命令フェッチ、命令デコード、演算実行、メモリ･アクセス、メモリ読み出し)が 1 クロック･サイクルを占める。単一サイクル方式からの変更点は 1 つの命令の実行中に機能ユニットを共有出来る。これによりハードウェア量の削減に繋がる。命令のタイプ別で使用するクロック･サイクルが異なる。ジャンプ命令、分岐命令は 3 サイクル、算術論理演算命令、ストア命令は 4 ステップ、ロード命令は 5 ステップで命令完了となる。単一サイクルに比べ CPI は大きくなるが、クロック周波数を上げる事が出来、高速化に繋がる。単一サイクル方式とマルチサイクル方式の比較は本論文の第 5 章にて詳しく述べる。

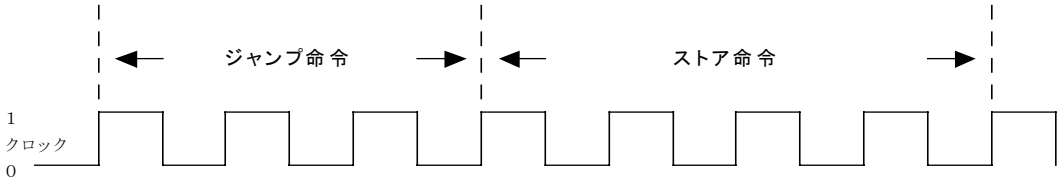

図 3：マルチサイクル方式命令実行

(3) パイプライン方式命令実行

パイプライン方式は命令の各実行ステップを「ステージ」と呼ばれる段階に分割し、連続した命令の各ステージをオーバーラップさせ同時に処理するものである。一般的なパイプラインでは命令フェッチ(IF)、命令解読(ID)、命令実行(EX)、メモリアクセス(MEM)、結果のレジスタへのライトバック(WB)で成り立っている。マルチサイクル方式では、1 つの命令が終わり次第、次の命令を発行するがパイプライン方式ではオーバーラップさせることにより CPI が小さくなり、全体的な処理時間が短縮される。パイプライン･ステージを増やすことで論理的には高速動作をさせる事が可能であるが、データハザードや分岐ハザードと呼ばれる障害を克服しなければならず、制御が複雑になる。パイプライン方式につ

いては本論文の第 4 章にて詳しく述べる。

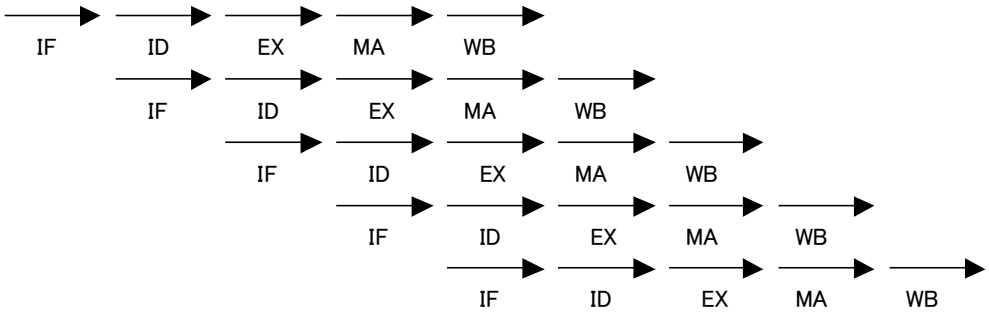

図 4 : パイプライン方式命令実行

(4) スーパースケーラ方式命令実行

スーパースケーラ方式は、逐次命令を命令ユニットで並列に置き換えて、複数の命令を複数の実行ユニットに発行して、並列に複数の命令を処理するものである。発行された命令を一時的に貯えておくリザベーションステーションと、何らかの処理を行うときに順番を無視して処理を行うアウトオブオーダー(out of order)制御と、レジスタ数の制限を取り除くためにプログラミングモデルのレジスタをマッピングするリネームレジスタとその割り当てを行うレジスタリネームイングと、分岐予測に基づいた命令の投機実行制御などがある。

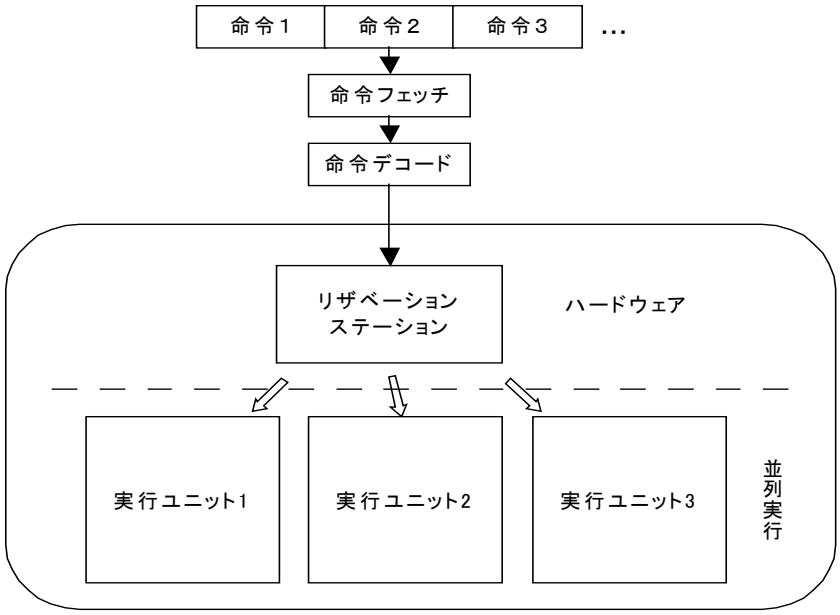

図 5 : スーパースケーラ方式命令実行

(5) VLIW 命令実行方式

VLIW（Very Long Instruction Word Set）方式は、C 言語等の高級言語で書かれたプログラムを、VLIW 命令アーキテクチャに対応するようコンパイルし、VLIW 命令として 5 つの命令スロットに供給され、最大 5 つの命令が並列に実行される。各実行ユニットとして、整数、浮動小数点、ロードストア、分岐、マルチメディアなどのユニットがある。命令語長は 256～1024 ビットとなる。

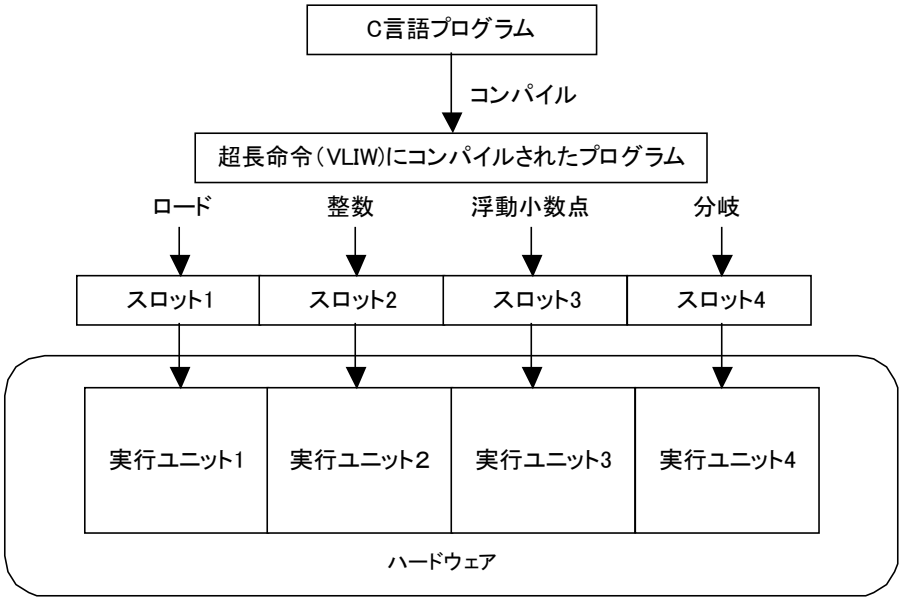

図 6 : VLIW 方式命令実行

3　単一サイクル方式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

本研究において、設計した単一サイクル方式、パイプライン方式のマイクロプロセッサは16 ビットの命令語長を持つプロセッサである。このプロセッサを動作させる命令セットとして 16 ビットの独自命令セット（以下 MONI 命令セットとする）を想定する。この命令セットは MIPS 32 ビット命令セットのサブセットにあたるものである。表 7 に命令のフォーマットを示す。

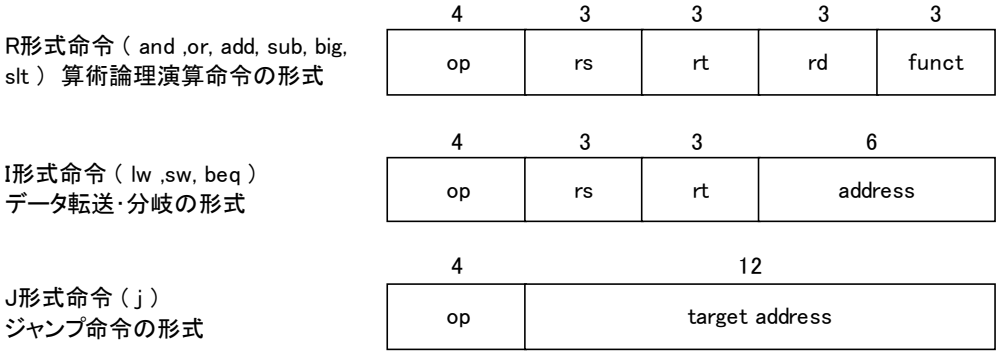

図 7 : MONI 命令フォーマット

(1)　命令フィールド

op　　: opcode。命令操作コード。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。opcode 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

MONI 命令セットはロード、ストア、ジャンプ、算術論理演算、停止などからなる基本的な命令により構成された全 11 命令の命令セットである。今後、命令操作コード「op」と命令機能コード「funct」フィールドを追加する事により、命令の拡張が可能である。以下に命令操作コード、機能コードの対応表を示す。

本研究において、この MONI 命令セットによるテストパターンを作成し、設計した単一サイクル方式、パイプライン方式マイクロプロセッサのシミュレーションを行った。作成したテストパターンは「1 から N までの和」、「N 個の中の最大値の算出」、「ユーグリット互除法による 2 数の最大公約数の算出」の 3 通りである。テストパターンは本論文の付録と

して載せる。現段階で MONI 命令セットは全 11 命令ではあるが、単一サイクル方式、パイプライン方式による命令実行の動作原理を理解するには十分な命令数である。

表 1 : MONI 命令の命令操作コードと機能コードの対応

命令	Op3	Op2	Op1	Op0
R形式	0	0	0	0
lw	0	0	0	1
sw	0	0	1	0
beq	0	1	0	0
jump	1	0	0	0
halt	1	1	1	1

R形式	Fn2	Fn1	Fn0
add	0	0	0
sub	0	0	1
and	0	1	0
slt	0	1	1
or	1	0	0
big	1	1	1

(2) 命令操作コードと機能コード

操作コードにより命令は R 形式、lw(ロード)、sw(ストア)、beq(条件分岐)、jump(無条件分岐)、halt(終了命令)に大別できる。R 形式は、機能コードとの組み合わせにより、add(加算)、sub(減算)、and(論理積)、or(論理和)、big(2 つのオペランドを比較し、大きい方を格納)、slt(大小比較)の 6 命令を表現出来る。

3.2 単一サイクル方式マイクロプロセッサのアーキテクチャ

(1) 単一サイクル方式データパス

単一サイクル方式命令実行とは、1 つの命令を 1 つのクロック･サイクルで実行完了しようとするものである。これは、どのデータパス資源も 1 クロック･サイクル中に 2 回以上の使用は出来ない事を意味する。つまり、2 回以上必要となる要素は必要個数分だけ別個に設けなければならない。図 8 に単一サイクル方式実行のデータパスを示す[6]。

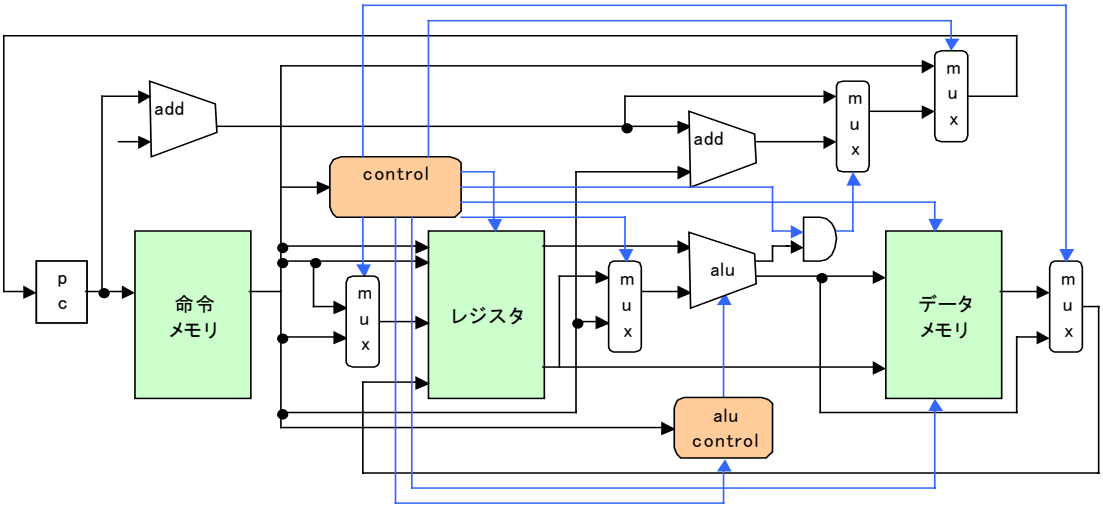

図 8：単一サイクル方式データパス

(2)　単一サイクル方式マイクロプロセッサの構成要素

設計した単一サイクル方式マイクロプロセッサは 3 つの ALU、5 つのマルチプレクサ、3 つの状態記憶素子(命令メモリ、レジスタ、データメモリ)、プログラムカウンタ、2 つの制御ユニット(control、alu control)のモジュールによって構成されている。

- ALU

プログラムカウンタの算出用の ALU2 つと、レジスタから読み出された値の算術論理演算を行う ALU である。

- マルチプレクサ

制御信号によって 1 つの入力を選択するデータ･セレクタである。

- 状態記憶素子

命令メモリ、データメモリ、レジスタである。一般的には CPU 設計において命令メモリ、データメモリは外部メモリとして使用するものであるが、本研究ではレジスタ同様、内部メモリ(レジスタ)として定義した。命令メモリ、データメモリは 32 個、レジスタは 8 個の 16 ビットデータを格納出来る。エッジ･トリガ方式で制御されるので、1 つのクロックサイクル内で読み出しと書き込みを行うことができる。

- 制御ユニット

control は単一サイクルプロセッサの主制御ユニットであり、データ資源に制御を投げかける。alu control は control からの制御を受け、ALU の算術論理演算項目を決定する。制御を複数レベルに分ける事で、主制御ユニットを小さくすることが出来、速度向上にも繋がる。

3.3　命令の実行と制御

単一サイクル方式による命令の実行は、クロックの立上がりでプログラムカウンタ(以下PC)が更新され、命令の実行が開始する。PC が命令メモリの命令番地を指し、格納されている命令が読み出される。読み出された命令が主制御ユニットに入り、各モジュールの制御が決定する。制御に従い各モジュールは動作を行い、最終的な結果がクロックの立下りでレジスタへの書き込みを開始する(レジスタへの書き込みを行う命令:ロード命令、算術論理演算命令等)。クロックの立下りエッジまでに書き込まれる値は安定していなければならない。単一サイクルによる命令実行方式は効率が悪いので現代のコンピュータにおいて実際に採用される事はまず無い。それは、以下の様な理由によるものである。

- 単一サイクル方式では各命令のクロックサイクルの長さは同一でなければならない。つまり、CPI は 1 でなければならない。
- クロックサイクルはマシン中の最長パスによって決まり、そのパスはロード命令である。
- もっと短いクロックサイクルで処理可能な命令(ストア命令、ジャンプ命令等)もロード命令で決定される最長パスに合わせなければならない。

上記の様な理由から、単一サイクル方式は CPI は 1 であるが、全体的な性能はあまり高くないと言える。

3.4 シミュレーションによる動作検証

Verilog-HDL により設計した、単一サイクル方式マイクロプロセッサのデータを論理合成ツール(Synopsys 社、FPGA Express Xilinx Edition 3.6.1)に通し、MTI 社の ModelSim XE5.5b にて機能レベルシミュレーションからゲート・レベルシミュレーション(配置配線後)までを行った。この際、FPGA チップとして Xilinx 社の「Virtex2 xc2v8000‐5bf957」を想定した。

機能レベルから配置配線後のゲート・レベルまで 4 ステップのシミュレーションが存在する。以下の様なステップとなる。

1. Behavioral
   - 機能レベルのシミュレーション。論理合成を行う前の段階であり、HDL の動作が与えられた仕様を満たすものかを確認する。
2. Post - Translate
   - 論理合成後のシミュレーション。階層化された回路の配線状況を確認する。未結線であった場合、エラーを修正しもう一度論理合成を行う。
3. Post - Map
   - 実装する回路を FPGA 上の回路資源(CLB)に対応付ける処理を行う。遅延を考慮したシミュレーションとなり、回路の複雑さを確認する事が出来る。
4. Post - Place & Route
   - FPGA に実際に配置配線を行った後のシミュレーションである。想定している

FPGA の最大実装規模に対してギリギリの論理資源を使用している場合など、配置配線処理が不可能となりこのシミュレーションを行う事が出来ない。

以上の様な工程を経て、単一サイクル方式マイクロプロセッサは Post-Place & Route シミュレーションまでを完了した。

シミュレーションに用いたテストプログラムは MONI 命令セットによって表現した「1 から N までの和」、「N 個の中の最大値の算出」、「ユーグリット互除法による最大公約数の算出」の 3 通りのプログラムである。これらのテストプログラムを与え、Post-Map シミュレーションと Post-Place & Route シミュレーションにより得られた結果を表 2、3、4、5 として示す。

表 2：ロード命令による最小クロック･サイクルの算出(Post-Map)

書き込みデータの安定	18.225 ns
動作確実な最小クロック･サイクル	38 ns
レジスタへの書き込み遅延	7.05 ns

表 3：単一サイクル方式シミュレーション結果(Post-Map)

	1 から N までの和 （N = 100）	N 個の中の最大値 （N = 5）	ユーグリット互除法による 最大公約数（152 と 36）
実行時間	15466 ns	1444 ns	2660 ns
クロック･サイクル数	407	38	70
命令数	407	38	70
CPI	1	1	1

表 4：ロード命令による最小クロック･サイクルの算出(Post-Place & Route)

書き込みデータの安定	63.797 ns
動作確実な最小クロック･サイクル	130 ns
レジスタへの書き込み遅延	15.301 ns

表 5：単一サイクル方式シミュレーション結果(Post-Place & Route)

	1 から N までの和 （N = 100）	N 個の中の最大値 （N = 5）	ユーグリット互除法による 最大公約数（152 と 36）
実行時間	52910 ns	4940 ns	9100 ns
クロック･サイクル数	407	38	70

命令数	407	38	70
CPI	1	1	1

単一サイクル方式ではレジスタへの書き込みはクロックの立下り時の入力データを書き込む。そのため、書き込まれるデータはクロックの立下りまでに安定していないといけない。安定した値を書き込む事の出来るクロック・サイクルは命令の最長パスとなるロード命令によって決定すると言うことは前述した。表 2、4 にはそれぞれ Post-Map シミュレーションと Post-Place & Route シミュレーションにより得られた最長データパスとなるロード命令における書き込みデータ安定までの時間と、それにより得られた単一サイクル方式の動作確実な最小クロック・サイクルを示す。表 3、5 においての実行時間は表 2、4 により得られた最小クロック・サイクルを用いた時の実行時間である。単一サイクルは 1 命令を 1 クロックで完了するという特徴より、クロック・サイクル数がそのまま実行した命令数となる(CPI＝１)。Post-Map シミュレーションと Post-Place & Route シミュレーションによる結果を比較して分かる通り、実際に配置配線を考慮する事で最小クロック・サイクル、実行時間が大幅にアップしている事が確認出来る。マルチサイクル方式、パイプライン方式との比較は第 5 章にて述べる。

4　パイプライン方式マイクロプロセッサの設計

4.1　パイプライン方式マイクロプロセッサのアーキテクチャ

パイプライン方式は、命令の各実行ステップを「ステージ」と呼ばれる段階に分け、連続した命令の各ステージをオーバーラップさせ、同時に処理するものである。一般的なパイプラインでは命令フェッチ(IF)、命令解読(ID)、命令実行(EX)、メモリアクセス(MEM)、結果のレジスタへのライトバック(WB)で成り立っている。命令をオーバーラップさせる事で命令のスループットを大きくする(CPI を小さくする事に等しい)事で全体的な処理時間が短縮される。パイプライン・ステージを増やす事で論理的には高速動作をさせる事が可能であるが、データハザードや分岐ハザードと呼ばれる障害を克服しなければならず、制御が複雑になる。Intel 社の PentiumⅣはパイプラインの段数を増やし各ステージの論理段数を少なくし 20 段パイプラインによる高速化を実現している[4]。

本研究では、上記に示した 5 段パイプライン方式によるプロセッサの設計を行った。これは、本論文の第 3 章で示した、単一サイクル方式プロセッサにパイプライン・レジスタ、フォワーディング・ユニット、ハザード検出ユニット、分岐ハザード検出ユニット等を付加する事で単一サイクル方式を基に 5 段パイプライン化したものである。次節にてパイプライン化の過程を述べる。

4.2　単一サイクル方式命令実行からパイプライン方式へ

本節では、単一サイクル方式の 5 段パイプライン化の過程を説明する。ここでは過程を説明するにあたって使用する図の説明を行う。図 9 に例を示す[6]。

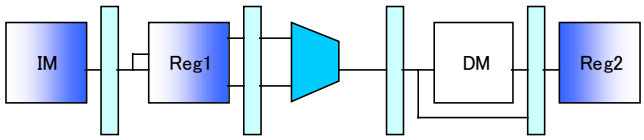

図 9：パイプラインの模式図表現

図 9 において、縦の棒 4 本と IM、Reg1、DM、Reg2 と書かれた正方形が 4 個、台形が 1 個を結線している。正方形の右半分が色付いているものはデータの読み出しを意味し(IM、Reg1)、左半分が色付いているものはデータの書き込みを意味している(Reg2)。色付けがされていない(DM)はアクセスがされていない状態を示している。同様に台形が色付いている時はアクセスがされているという状態を示している。

4.2.1　パイプライン･レジスタの付加によるパイプライン化

単一サイクル方式は 1 クロック・サイクル間に命令が 1 つ実行するものであった。5 段パイプラインでは 1 クロック・サイクル間に命令が最大で 5 つ発行される。単一サイクルのデ

ータパスを 5 つに分割し、それぞれに命令を実行する各ステージに対応した下記の様な名前を付ける。

1．IF　　：　　命令フェッチ
2．ID　　：　　命令解読とレジスタ･フェッチ
3．EX　　：　　命令実行/アドレス生成
4．MEM　　：　　データ･メモリ･アクセス
5．WB　　：　　書き込み

これらの各ステージを 1 クロック･サイクルで実行する。命令を各ステージに分けパイプライン方式で命令を実行する場合、命令メモリは 1 命令の 5 ステージ中の 1 つだけで使用される。従って、残りの 4 つのステージの間に、命令メモリを他の命令と共有することが可能となる。

第 1 ステージ IF でフェッチした命令をそれ以降の残り 4 ステージで使用できるよう保持しておくためには、その値をレジスタ中に保持しておく必要がある。それらの事は同様に、パイプラインの各ステージに当てはまる。そこで、各パイプラインステージの間に、パイプライン･レジスタと呼ばれる記憶素子を配置する。これにより全ての命令は 1 クロックサイクル間に 1 つのパイプライン･レジスタから次のパイプライン･レジスタまで進む事になる。以下に単一サイクル方式にパイプライン･レジスタを付加したデータパスを示す[6]。

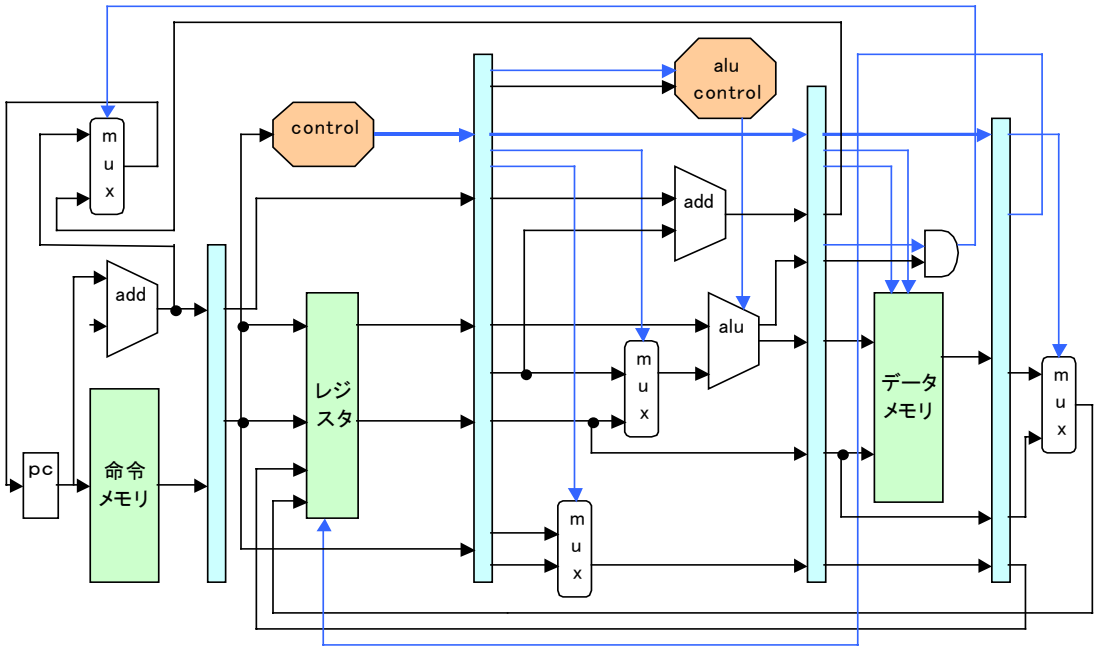

図 10：単一サイクル方式にパイプライン･レジスタを付加したデータパス

図 10 の縦に細長い 4 本の棒がパイプライン･レジスタである。左から FD、DE、EM、MW とする。これで単一サイクル方式をパイプライン化する事ができた。次に、パイプライン

方式で生じるデータ･ハザードの回避方法を説明する。

4.2.2 データ･ハザードとフォワーディング

各実行ステージをオーバーラップさせ実行するパイプライン方式では、命令に使用するデータに依存関係が生じる事がある。図 11 のがその例である[6]。

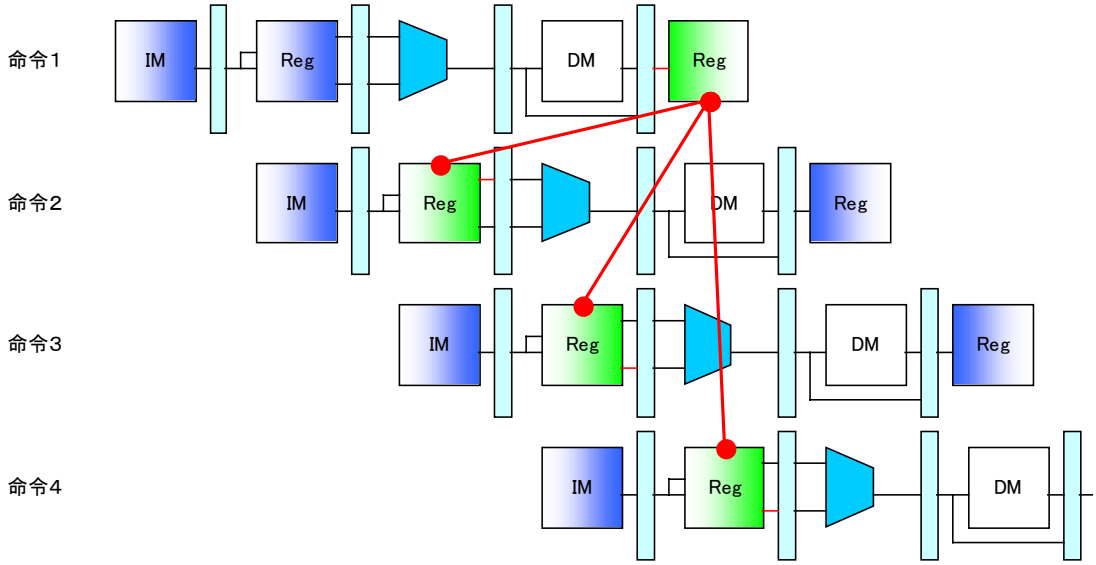

図 11：データ･ハザードが生じるデータ依存関係(フォワーディング)

命令 1 は第 3 ステップで演算された結果を第 5 ステップでレジスタに書き込む命令である。この時、命令 2、命令 3、命令 4 のそれぞれ第 2 ステージで命令 1 でレジスタに書き込まれる値を読み出して第 3 ステージで演算を行いたいとする。命令 4 においては正常に読み出す事が出来るが、命令 2、命令 3 では未だ書き込みが完了していない事が図 11 により分かる。このような命令間のデータ依存関係の事を、「パイプライン･データ･ハザードが生じている」と言う。

このデータ･ハザードに対する解決策として次の様な事が考えられる。命令 1 においてレジスタで書き込まれる値は、実際には第 3 ステップで決定すると言うことである。第 3 ステージで決定する値を第 5 ステージでレジスタに書き込む前に命令 2、命令 3 に渡してやる(先送り)ことによってこの問題を解決出来る。この様に、命令間のデータの先送りをフォワーディングと言う。図 11 のデータ依存関係はフォワーディング処理を行うと図 12 の様に解決出来る。

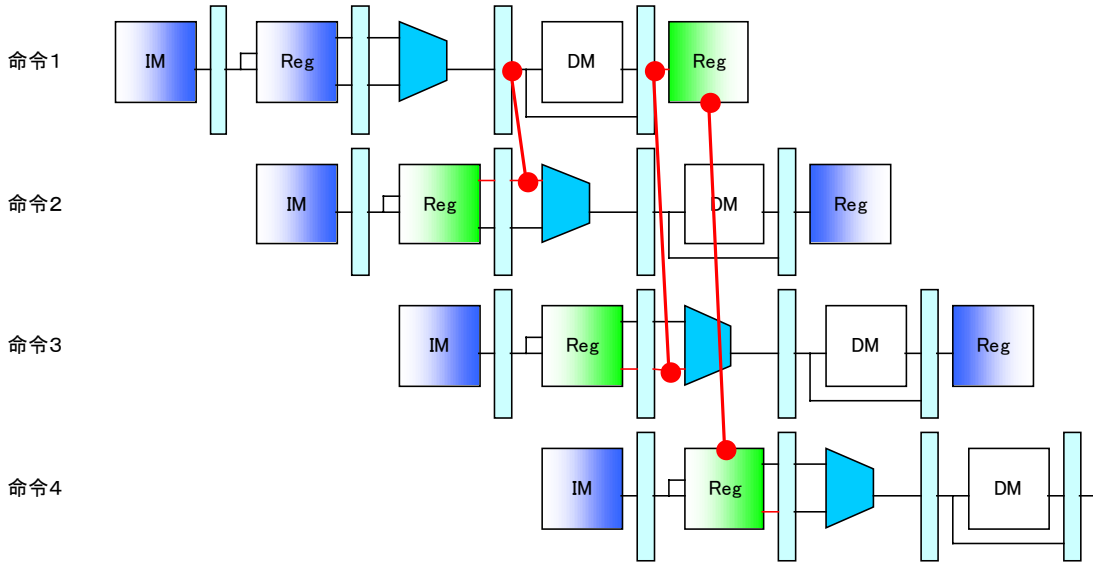

図 12：フォワーディング処理後のデータ依存関係

図 12 より、命令 2、命令 3 においてフォワーディングを行う事で第 3 ステージにおいて適切な演算を行う事が出来ることを確認できる。次に、フォワーディングでは対処出来ないデータ・ハザードが生じた場合の制御方法を説明する。

4.2.3 データ・ハザードの制御とストール

全ての命令がフォワーディングを行う事で、適切な値を用いて演算を行えるという訳ではない。それは、ロード命令がレジスタに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時である。図 13 にそのような場合のデータ依存関係を示す[6]。

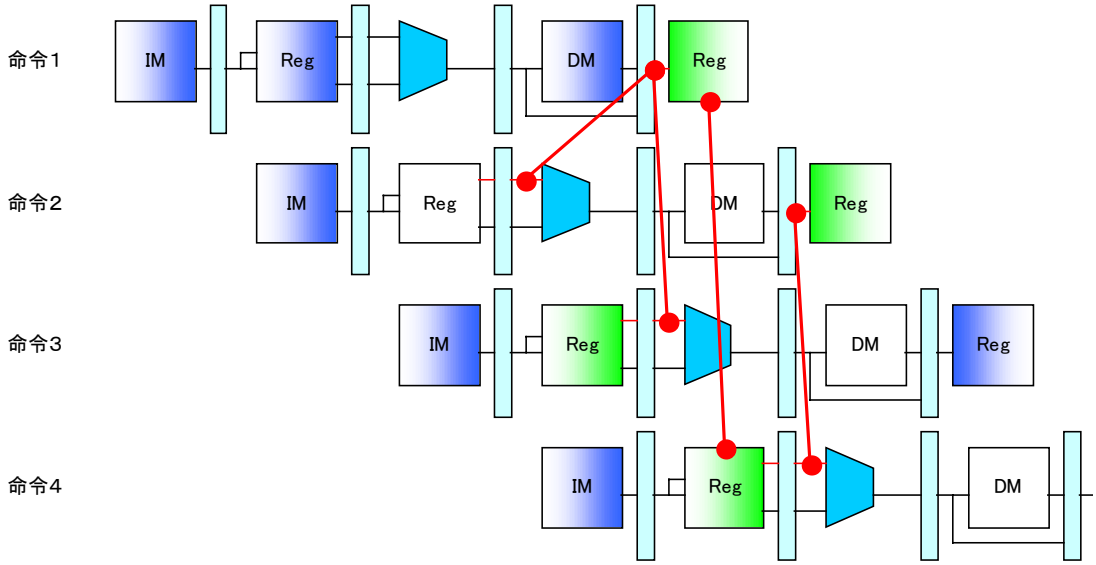

図 13：データ・ハザードが生じるデータ依存関係(ストール)

命令 1 の第 4 ステージにおいて、命令 2 は ALU での演算を行っている。その間、命令 1 は未だデータ・メモリからデータを読み出している最中である。このようなデータ依存関係にある場合、フォワーディングだけでは解決出来ない。ロード命令とその結果を読み出す次命令との組み合わせに対しては、パイプラインをストールさせる事で解決出来る。ストールを行った場合のデータ依存関係を図 14 に示す。

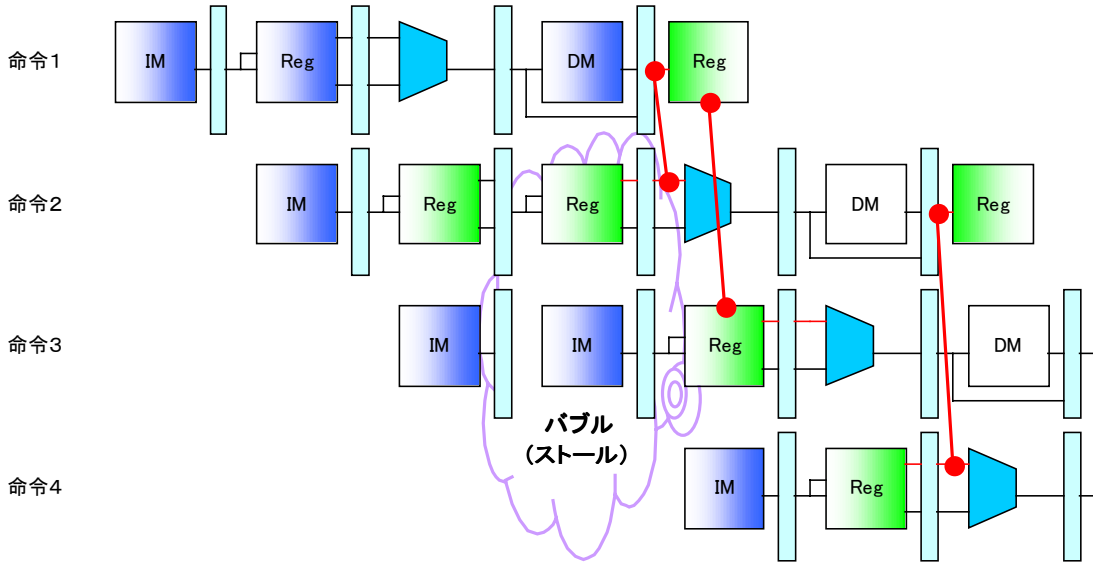

図 14：パイプラインにストールを挿入した場合のデータ依存関係

図 14 より、命令 2 は第 2 ステージを 2 回繰り返す事で第 3 ステージにおいて適切な値を用いて演算を行う事が出来る。命令 3 も同様である。ストールとは同じステージを繰り返し実行する事で、命令の実行を遅らせる事である。フォワーディングとストールを組み合わせる事で、パイプライン処理における命令間のデータ依存関係による障害(データ・ハザード)を克服できる。次に、分岐命令が生じた時のパイプライン処理に及ぼす影響とその回避方法を説明する。

4.2.4 制御ハザードの回避

パイプライン方式命令実行において、分岐に起因する制御ハザードと呼ばれるハザードがある。制御ハザードとはフェッチした命令が条件分岐命令だった場合の分岐判断の決定が遅れる事を指す。そのような場合の命令依存関係を図 15 に示す[6]。

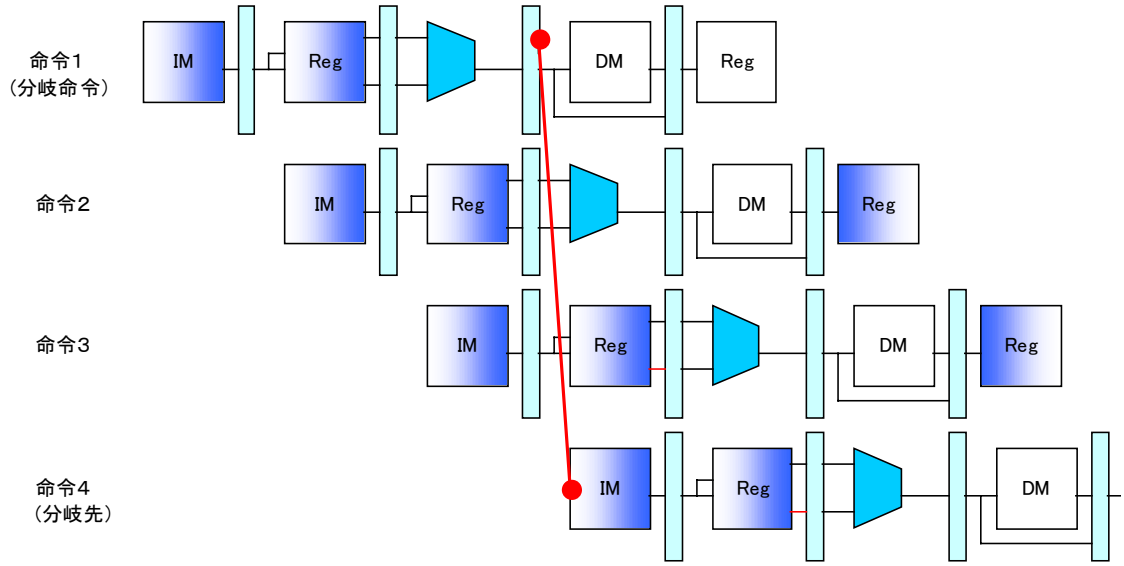

図 15：パイプラインに対する分岐命令の影響

図 15 において命令 1 は分岐命令であるとする。分岐の判定は第 4 ステージになされている。分岐が成立した場合、命令 2、3 は全く意味の無い命令になってしまう。この制御ハザードの解決策として、命令 1 の第 4 ステージに達するまでそれ以降の命令をストールさせる事が考えられる。しかし、それでは全体的な速度に影響が出てしまい解決策と呼べない。本研究ではこの制御ハザードの解決策として、分岐判定を第 4 ステージではなく第 2 ステージにおいて行う様に改良する。図 16 に分岐判定改良後の命令依存関係を示す。

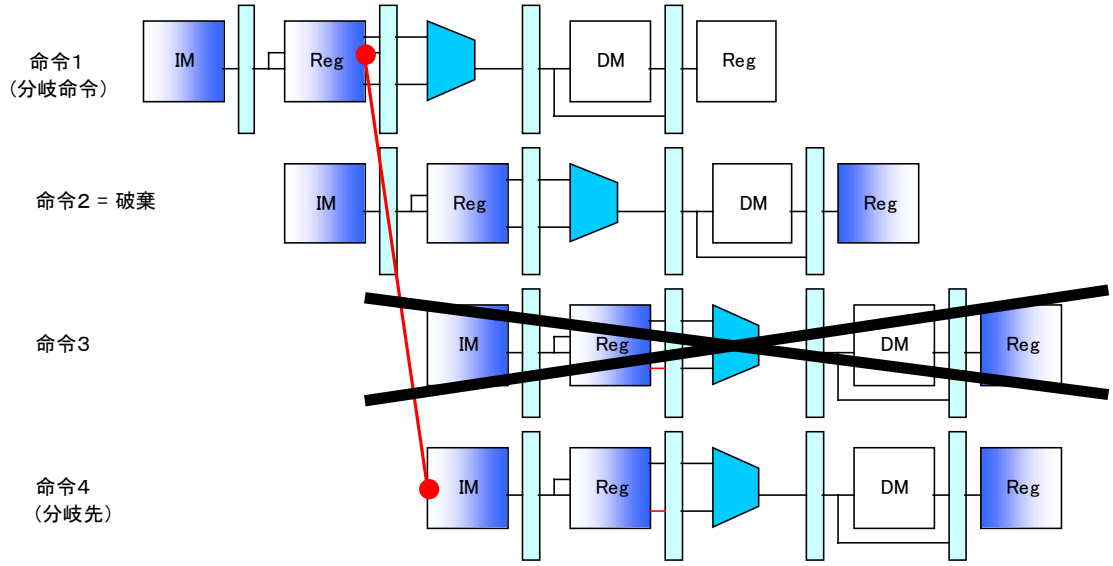

図 16：分岐判定改良後の命令依存関係

分岐判定を第 2 ステージで行う事で、破棄しなければならない実行中の命令は命令 2 のみとなる。本研究の MONI 命令セットには条件分岐、無条件分岐の 2 種類の分岐命令が存在する。条件分岐判定はレジスタから読み出した 2 つの値を比較し、その時点で等しいか等しくないかを判断し、分岐するか否かを決定する。無条件分岐命令の場合はレジスタの値に関わらず、第 2 ステージにて分岐を決定する。

4.3　パイプライン方式データパス

データ・ハザード、制御ハザードを考慮した本研究における最終的なパイプライン方式命令実行のデータパスを以下に示す。

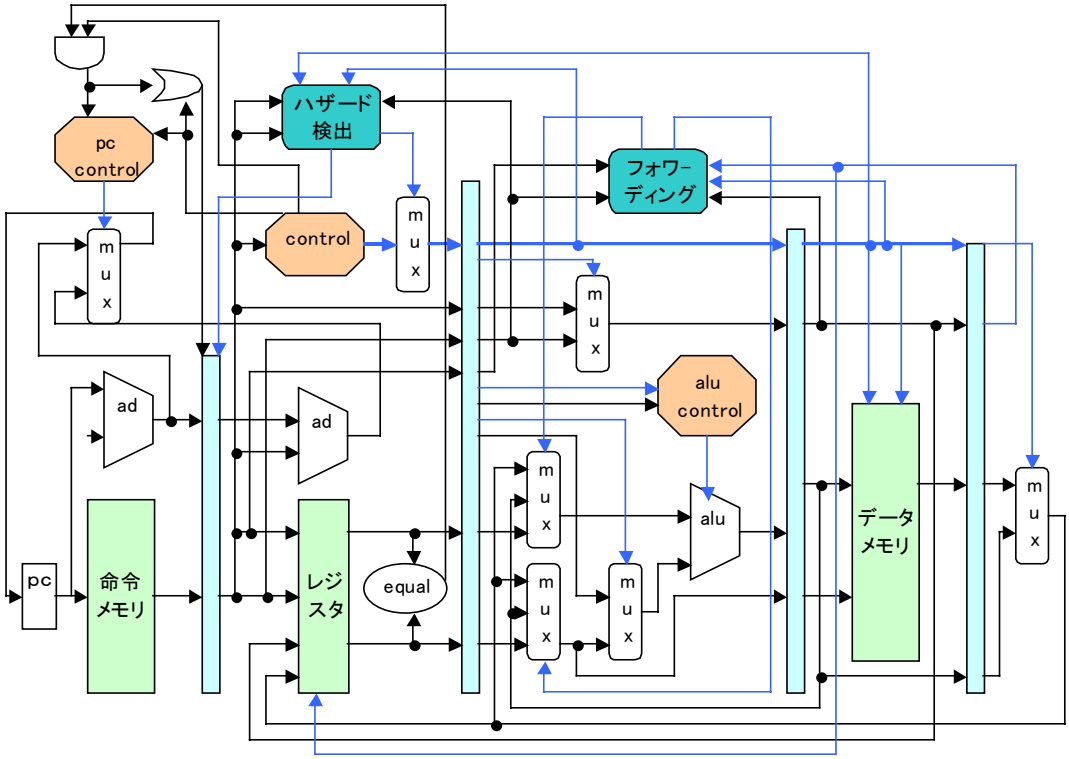

図 17：パイプライン方式データパス

● パイプライン方式マイクロプロセッサの構成要素

設計したパイプライン方式マイクロプロセッサは 3 つの ALU、7 つのマルチプレクサ、3 つの状態記憶素子(命令メモリ、レジスタ、データメモリ)、プログラムカウンタ、3 つの制御ユニット(control、alu contorol、pc control)、3 つのハザード制御ユニット(フォワーディング、ハザード検出、equal)、4 つのパイプライン・レジスタで構成されている。基本的に、単一サイクル方式をパイプライン化したものであるので、構成要素に変わりは無く、パイプライン・レジスタ、ハザード制御ユニット、マルチプレクサ等が付加され、5 段パイプライン方式命令実行を実現した。

4.4　命令の実行と制御

パイプライン処理は命令の平均実行時間を短縮する。これは、単一クロック･サイクル方式のデータパスではクロック･サイクルの短縮を意味する。言い換えれば、パイプラインは命令のスループットを向上させるが、各々の命令の実行時間を短縮するわけではない。パイプライン方式ではステージごとに専用のハードウェアを使用し、各クロック･サイクルごとに命令を開始する。プログラム中の命令間にデータ依存関係があると、処理時間が長くなるという問題が生ずる。これは、4.1 から 4.2 節で説明したデータ･ハザードが生じるためであり、フォワーディング、ストールを行う事でハザードによるコストを低減する事が出来る。4.3 節で説明した分岐判定の改良により、分岐成立によって無駄になる命令数を減らす事が出来、命令全体の実行時間を短縮する事が出来る。

4.5　シミュレーションによる動作検証

パイプライン方式マイクロプロセッサは単一サイクル方式と同じ環境にて設計し、同じテストプログラムを与えてシミュレーションを行った。Post-Map シミュレーションと Post-Place & Route シミュレーションを行った結果を表 6、7、8、９に示す。

表 6：パイプライン方式最小クロック･サイクル(Post-Map)

動作確実な最小クロック･サイクル	12 ns

表 7：パイプライン方式シミュレーション結果(Post-Map)

	1 から N までの和 （N = 100）	N 個の中の最大値 （N = 5）	ユーグリット互除法による 最大公約数（152 と 36）
実行時間	6144 ns	708 ns	1620 ns
クロック･サイクル数	512	59	135
命令数	407	38	70
CPI	1.3	1.6	1.9
ストール発生回数	1	11	46

表 8：パイプライン方式最小クロック･サイクル(Post-Place & Route)

動作確実な最小クロック･サイクル	30 ns

表 9：パイプライン方式シミュレーション結果(Post-Place & Route)

	1 から N までの和	N 個の中の最大値	ユーグリット互除法による

	（N = 100）	（N = 5）	最大公約数（152 と 36）
実行時間	15360 ns	1770 ns	4050 ns
クロック･サイクル数	512	59	135
命令数	407	38	70
CPI	1.3	1.6	1.9
ストール発生回数	1	11	46

表 6、8 で示した通りそれぞれのレベルでのシミュレーションから得られる最小クロック･サイクルは 12 ns と 30 ns である。これ以上クロック･サイクル数を小さくすると、パイプライン･レジスタへの書き込みが読み出しを開始するまでに完了しないため、適切な結果を得る事が出来なくなる。それぞれのテストプログラムの命令数は単一サイクル方式のクロック･サイクル数に等しい。最大公約数のストール発生回数が 46 と 3 つの中で一番多く、その為 CPI も 3 つの中で最大となっている。これは、プログラム中のデータ依存関係が密であった事を示している。単一サイクル方式と同様に、Post-Map シミュレーションと Post-Place & Route シミュレーションにより得られた結果を比較すると、配置配線を行う事で大幅に最小クロック･サイクルと命令の実行時間がアップしている事が確認できる。単一サイクル方式、マルチサイクル方式、パイプライン方式によるシミュレーション結果の比較と考察は第 5 章にて詳しく述べる。

5　生成されたプロセッサの評価

5.1　単一サイクル方式とマルチサイクル方式とパイプライン方式の背景

単一サイクル方式、マルチサイクル方式、パイプライン方式という CPU アーキテクチャの中で、歴史的背景から単一クロック･サイクルで 1 命令を実行完了する単一サイクル方式が最初の CPU アーキテクチャであると言える。単一サイクル方式の短所として、命令セット中の最長パスであるロード命令のクロック･サイクル時間に全ての命令が合わせなければならないという事があげられる。また、1 クロック･サイクルで命令を実行完了する為、ALU 等の機能ユニットを共有できず、必要な分だけ用意しなければならないという欠点もある。これら単一サイクル方式の短所を改善すべく、1 命令を複数ステップに分けて実行する事で、命令種類別に実行完了にかかるクロック･サイクル数が異なるマルチサイクル命令実行方式が誕生した。また、1 ステップに 1 クロック･サイクルを割り当てる事で、クロック･サイクルを小さくする事ができ、全体として単一サイクル方式に比べ、実行時間の短縮が期待できる。クロック･サイクル毎での機能ユニットの共有が可能となり、ハードウェア量の削減に繋がった。パイプライン方式は、マルチサイクル方式を多重化する事に等しい。命令をオーバーラップして実行させる事により CPI を小さくする事ができ、全体的な命令処理時間を短縮させる事が出来る。以上の様な歴史的背景を踏まえた上で、単一サイクル方式、マルチサイクル方式、パイプライン方マイクロプロセッサを設計し Post-Place & Route シミュレーションを行った結果を次節にて示す。

5.2　単一サイクル方式とマルチサイクル方式とパイプライン方式の比較と評価

本節では設計した単一サイクル方式とマルチサイクル方式、パイプライン方式マイクロプロセッサについて 3 通りのテストプログラムを与え、Post-Place & Route シミュレーションによって得られた結果を比較する。ここでマルチサイクル方式マイクロプロセッサは本研究室の大八木 睦氏によって設計されたものである。

表 10：各命令実行方式の最小クロック･サイクル

単一サイクル方式	マルチサイクル方式	パイプライン方式
130	54	30

単位 ：ns

表 10 は各命令方式別の最小クロック･サイクルである。上記のクロック･サイクルより小さい値でシミュレーションを行うと、データ遅延によりレジスタへの書き込みが完了するまでに読み出しが開始され、適切なデータを読み出す事が出来ない。設計段階において、マルチサイクル方式は単一サイクル方式の 5 分の 1 程度の最小クロック･サイクルが目標であった。それは、マルチサイクル方式は 1 つの命令完了に 3～5 クロック･サイクル必要であるからである。今回のシミュレーションにより得られた最小クロック･サイクルに関する考

察は次節５．２において行う。

表 11 : 3 通りのテストプログラムの実行時間(Post-Place & Route)

	和 1～100	最大値 要素数 = 5	公約数 152 と 36
単一サイクル方式	52910	4940	9100
マルチサイクル方式	77058	7992	13230
パイプライン方式	15360	1770	4050

単位 ：ns

表 12 : 各命令実行方式の命令実行完了までのクロック･サイクル数と CPI

	和 1～100	最大値 要素数 = 5	公約数 152 と 36
単一サイクル方式	407(1)	38(1)	70(1)
マルチサイクル方式	1427(3.5)	148(3.5)	245(3.5)
パイプライン方式	512(1.3)	59(1.6)	135(1.9)

書式 ： クロック･サイクル数(CPI)

表 11 は表 10 より得られた最小クロック･サイクルを用いて通りのテストプログラムを与えた場合の実行時間である。実行時間を比較して分かる通り、パイプライン方式が一番早く実行完了しており､単一サイクル方式をパイプライン化する事での高速化が確認出来る。表 10 の最小クロック･サイクルと、表 12 における命令実行完了までのクロック･サイクル数と CPI より、マルチサイクル方式の実行時間がどのテストプログラムにおいても最大であることが確認できる。パイプライン方式における CPI は表 12 において 1.3、1.6、1.9 とばらつきが生じている｡これはパイプライン･ハザードであるストールの発生回数に依存している。表 13 に 3 通りのテストプログラムにおける実行完了までに生じるストールの発生回数を示す。

表 13 : パイプラン･ハザードの発生回数(ストール)

和 1～100	最大値 要素数 = 5	公約数 152 と 36
1	11	46

単位 ： 回

表 13 のストール発生回数により、テストプログラムにおける命令間のデータ依存関係の度合いが確認出来る。3 つのテストプログラムにおいて和のプログラムはストール回数が 1 回であり、表 12 におけるパイプライン方式の CPI も 1.3 と最小となっている。一方、公約数のストール発生回数は 46 回であり、CPI も 1.9 となっておりパイプライン方式に

おいて最大である。本論文の末尾に付録として 3 つのテストプログラムを載せる。それにより、命令間のデータ依存関係が確認できる。

表 14 : ハードウェア規模(フリップ･フロップ、LUT、Verilog-HDL 記述)

	フリップ･フロップ	LUT	Verilog-HDL 記述
単一サイクル方式	1157	3228	402
マルチサイクル方式	1223	2393	562
パイプライン方式	1337	3549	892

単位 : フリップ･フロップ、LUT は個、Verilog-HDL 記述は行

表 14 により命令実行方式別のハードウェア規模を比較できる。フリップ･フロップ数、LUT、Verilog-HDL 記述においてパイプライン方式が最大である。フリップ･フロップ数が他の 2 つに比べて多いのはパイプライン･レジスタが影響している。単一サイクル方式とマルチサイクル方式を比較すると、フリップ･フロップでは単一サイクル方式が多く、LUT ではマルチサイクル方式が多くなっている。LUT においてマルチサイクル方式が単一サイクル方式より少なくなっている理由として、ALU 等の機能ユニットを共有しているからである。フリップ･フロップ数が多くなっている理由としては、複数クロックで 1 命令を実行完了する為に、クロック毎に状態を保持しておく記憶素子が必要な為である。

5.3 シミュレーション結果の考察

本研究では単一サイクル方式、マルチサイクル方式、パイプライン方式という命令実行の違いによるマイクロプロセッサを設計した訳であるが、ゲートレベル･シミュレーションから得られた結果は単一サイクル方式よりも、マルチサイクル方式の実行時間が大きくなってしまうという結果であった。この原因として以下の事が考えられる。

- 動作する命令セット（MONI）の命令数が少なく、制御が簡単
- マルチサイクル方式は、クロック間のデータ依存関係が強い

命令セットの命令数が現段階では全 11 命令である。その為、単一サイクル方式において制御が簡単に行える。パイプライン方式においても、パイプライン･ハザードに対する制御以外の主制御は基本的に単一サイクル方式と同じである。その為、単一サイクル方式、パイプライン方式は命令数が多くなったと仮定すると、新たに機能ユニットを付加する必要があり、各機能ユニットへの制御が複雑になる。その結果、最小クロック･サイクルが大きくなると予想される。一方、マルチサイクル方式の制御はクロック･サイクル毎に主制御ユニットから出力される。命令数が多くなったと仮定しても、各機能ユニットに対する制御量は変わらない。またマルチサイクル方式はクロック間のデータ依存関係が強く、確実にデータが安定するクロック･サイクル時間が必要である。本研究において、単一サイクル方式に対するマルチサイクル方式による高速化が確認できなかった理由として、以上に述べた理由が関係していると考察する。

5.4 FPGA へのロードと検証

今回の論文では FPGA へのロードは行っていない。本研究室には KITE マイクロプロセッサボード+と呼ばれる FPGA 評価ボードがある。この評価ボードには Xilinx 社の FPGA「XC4010-PG191」が搭載されている。またこのボードには 32K ワードの RAM がある。そしてこの RAM は KITE マイクロプロセッサの命令セットを用いる事で使用できる。今回作成したマイクロプロセッサは MONI 命令セットと名づけた独自命令セットによる設計であった。開発の初期段階から KITE マイクロプロセッサボード+上での検証を考慮していなかった為、FPGA にダウンロードし検証しようとしても、MONI 命令セットでは RAM を使用する事が出来ない。そこで、KITE マイクロプロセッサボード+上での FPGA 実機検証を諦め、大きな FPGA チップを仮定する事で、命令メモリ、データメモリを内部メモリ(レジスタ)として定義し、MONI 命令セットにより動作するマイクロプロセッサを設計した。その際仮定した FPGA チップは Xilinx 社の「Virtex2」である。大容量 FPGA チップでのゲートレベル･シミュレーションを行った為、FPGA 内の CLB 間の遅延が大きくなり、クロック･サイクル時間を短くするのに限界があった。今後、設計した単一サイクル方式/パイプライン方式マイクロプロセッサを FPGA 上へ実際にダウンロードし検証を行うには、Virtex2 の搭載可能な FPGA 評価ボードを用いるか、MONI 命令セットを KITE マイクロプロセッサ命令セットを用いて再び設計を行う事で FPGA 上での実機検証が行えると思われる。

6　おわりに

本論文では、ハードウェア記述言語による単一サイクルマイクロプロセッサを設計し、パイプライン化を行い、5 段パイプライン処理を行うパイプラインプロセッサを設計した。独自命令セットを用いて 3 通りのテストパターンを作成し、FPGA チップを想定してのゲートレベルシミュレーションを行った。得られたシミュレーション結果を元に命令実行方式の違いによるハードウェア規模、命令実行時間等の性能比較を行った。

その結果、単一サイクルをパイプライン化して命令実行させる事で最大 3.44 倍の速度向上が得られ、パイプライン処理による命令実行の高速化を確認できた。また、マルチサイクル方式と単一サイクル方式を比較した場合、マルチサイクル方式の実行時間が大きくなってしまう結果が得られた。

今後の課題として、命令セットが複雑になった時の性能比較があげられる。命令セットが複雑になれば、本研究によって得られた結果は異なってくるであろうと予想される。

また、実際に FPGA 上での実機検証を行いたいと考えている。その為に必要な事としては本論文の第 5 章に述べた通りである。

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大八木 睦氏、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心より深く感謝いたします。

参考文献

[1]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.

[2]桜井至:HDL によるデジタル設計の基礎,テクノプレス,1998.

[3]渡邉郁郎:AT 互換機のチップセット,エーアイ出版,1999.

[4] インテル:
http://www.intel.com/jp/ ,2002.

[5]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.

[6]John L.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.

[7] 上平祥嗣: 並列アルゴリズムクラスに基づくハードウェア自動生成, 立命館大学大学院理工学研究科修士論文, 2000.

[8] 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, 立命館大学大学院理工学研究科修士論文, 2000.

[9] 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.

[10] 上平祥嗣: KITE マイクロプロセッサを用いたハードウェア/ソフトウェア・コデザイン,立命館大学理工学部卒業論文,1998.

[11] 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の実装,立命館大学理工学部卒業論文,1998.

[12] 小林優:入門 VerilogHDL 設計入門,CQ 出版社,2001.

[13] 曽和将容:コンピュータアーキテクチャ原理,コロナ社,1993.

[14] 柴山潔:コンピュータアーキテクチャの基礎,近代科学社,1993.

付録

1. 単一サイクル方式命令実行マイクロプロセッサ Verilog-HDL 記述

- single_cycle

```
`timescale 1ns/1ns
module single_cycle ( clk, rst, set, ins_data, ins_address, mem_addr, mem_data,
                      result);

    input          clk, rst, set;
    input [15:0]   ins_data, mem_data;
    input [4:0]    mem_addr, ins_address;
    output [15:0]  result;

    // Instance function unit
    pc u1(
    .pc(pc),
    .next_pc(next_pc),
    .clk(clk),
    .rst(rst),
    .set(set)
    );

    assign  ins_op         =      ins[15:12];
    assign  ins_rs         =      ins[11:9];
    assign  ins_rt         =      ins[8:6];
    assign  ins_rd         =      ins[5:3];
    assign  ins_addr       =      ins[5:0];
    assign  ins_funct      =      ins[2:0];
    assign  ins_target     =      ins[11:0];

    //pc_add
    assign  add1_pc        =      pc + 1;
    //pc_beq
    assign  beq_pc         =      add1_pc + beq_addr;
    //sign_cut1
    assign  jump_pc        =      ins_target;
    //sign_cut2
```

```
    assign  beq_addr       =       offset;
    //sign_cut3
    assign  data_addr      =       alu_out;
    //sign_extend
    assign offset          =       ins_addr;
endmodule
```

- alu

```
`timescale 1ns/1ns
module alu ( alu_out, zero, read_data1, read_data3, alu_cont );

    input [15:0]   read_data1, read_data3;
    input [2:0]    alu_cont;
    output [15:0]  alu_out;
    output         zero;

    parameter [2:0]ADD = 3'b000,
                   SUB = 3'b001,
                   AND = 3'b010,
                   OR  = 3'b100,
                   BIG = 3'b111,
                   SLT = 3'b011;

    wire [15:0]    alu_out;
    wire           zero;

    assign  alu_out =  (alu_cont==AND)? (read_data1 & read_data3):
            (alu_cont==OR )? (read_data1 | read_data3):
            (alu_cont==ADD)? (read_data1 + read_data3):
            (alu_cont==SUB)? (read_data1 - read_data3):
            ((alu_cont==BIG)&&(read_data1 <= read_data3))? read_data3:
            ((alu_cont==BIG)&&(read_data1 > read_data3))? read_data1:
            ((alu_cont==SLT)&&({read_data1[15],read_data3[15]}==2'b00)&&
            (read_data1 <= read_data3))?  1:
            ((alu_cont==SLT)&&({read_data1[15],read_data3[15]}==2'b00)&&
            (read_data1 > read_data3))?   0:
```

```
            ((alu_cont==SLT)&&({read_data1[15],read_data3[15]}==2'b11)&&
            (read_data1 <= read_data3))?  0:
            ((alu_cont==SLT)&&({read_data1[15],read_data3[15]}==2'b11)&&
            (read_data1 <= read_data3))?  1:
            ((alu_cont==SLT)&&({read_data1[15],read_data3[15]}==2'b01))?
            0:
            ((alu_cont==SLT)&&({read_data1[15],read_data3[15]}==2'b10))?
            1:      16'bx;

    assign  zero    =       (read_data1==read_data3)? 1 : 0;
endmodule
```

- alu_control

```
`timescale 1ns/1ns
module alu_control ( alu_cont, ins_funct, alu_op );

    input [2:0]    ins_funct;
    input [1:0]    alu_op;
    output [2:0]   alu_cont;

    parameter [2:0]ADD     = 3'b000,
                   SUB     = 3'b001,
                   AND     = 3'b010,
                   SLT     = 3'b011,
                   OR      = 3'b100,
                   BIG     = 3'b111;

    assign  alu_cont =     (alu_op==2'b00)?                    ADD:
                   (alu_op==2'b01)?                            SUB:
                   ((alu_op==2'b10)&&(ins_funct==3'b000))?     ADD:
                   ((alu_op==2'b10)&&(ins_funct==3'b001))?     SUB:
                   ((alu_op==2'b10)&&(ins_funct==3'b010))?     AND:
                   ((alu_op==2'b10)&&(ins_funct==3'b011))?     SLT:
                   ((alu_op==2'b10)&&(ins_funct==3'b100))?     OR:
                   ((alu_op==2'b10)&&(ins_funct==3'b111))?     BIG:
                   3'bzzz;
```

```
endmodule
```

- control

```
`timescale 1ns/1ns
module control ( reg_dst, alu_src, memto_reg, reg_write, mem_write, mem_read,
            branch, alu_op, jump, ins_op);

    input [3:0]    ins_op;
    output [1:0]   alu_op;
    output         reg_dst,  alu_src,  memto_reg,  reg_write,  mem_write,
                   mem_read, branch, jump;

    parameter [3:0]r_form  = 4'b0000,
                   lw      = 4'b0001,
                   sw      = 4'b0010,
                   beq     = 4'b0100,
                   j       = 4'b1000,
                   halt    = 4'b1111;

    assign  reg_dst = (ins_op==r_form)?    1:      0;
    assign  alu_src = ((ins_op==lw)||(ins_op==sw))?       1:      0;
    assign  memto_reg = (ins_op==lw)?      1:      0;
    assign  reg_write = ((ins_op==lw)||(ins_op==r_form))? 1:      0;
    assign  mem_read = (ins_op==lw)?       1:      0;
    assign  mem_write = (ins_op==sw)?      1:      0;
    assign  branch = (ins_op==beq)?        1:      0;
    assign  alu_op[1] = (ins_op==r_form)?  1:      0;
    assign  alu_op[0] = (ins_op==beq)?     1:      0;
    assign  jump =  (ins_op==j)?           1:      0;
endmodule
```

- data_mem

```
`timescale 1ns/1ns
module data_mem ( read_data, data_addr, read_data2, mem_write,
                  mem_read, clk, mem_addr, mem_data, rst, result );
```

```
    parameter       size = 32;

    input [15:0]    read_data2, mem_data;
    input [4:0]     data_addr, mem_addr;
    input           mem_write, clk, mem_read, rst;
    output [15:0]   read_data, result;

    reg [15:0]      data_memory[0:size-1];

    always @( negedge clk )
            if( rst )               // when rst = 1, do syoki settei //
                    data_memory[mem_addr] <= mem_data;
            else if( mem_write )    // tsuujou shori //
                    data_memory[data_addr] <= read_data2;

    //      do after reset //
    assign  read_data = (mem_read)?data_memory[data_addr] :       16'bz;
    // do after all work finished watch results //
    assign  result    =    (~rst)? data_memory[mem_addr]  :       16'bz;
endmodule
```

- ins_mem

```
`timescale 1ns/1ns
module ins_mem ( ins, pc, ins_data, ins_address, rst, clk );

    parameter               size = 32;

    input [4:0]     pc, ins_address;
    input [15:0]    ins_data;
    input           clk, rst;
    output [15:0]   ins;

    reg [15:0]      ins_memory [0:size-1];

    always @( negedge clk )
            if( rst )       // when rst = 1, do shoki settei
```

```
                ins_memory[ins_address]<=     ins_data;

    assign  ins     =       ins_memory[pc];// start program
endmodule
```

- mux_1

```
`timescale 1ns/1ns
module mux_1 ( write_addr, ins_rt, ins_rd, reg_dst );

    input [2:0]    ins_rt, ins_rd;
    input          reg_dst;
    output [2:0]   write_addr;

    assign write_addr = ( reg_dst )? ins_rd : ins_rt;
endmodule
```

- mux_2

```
`timescale 1ns/1ns
module mux_2 ( read_data3, read_data2, offset, alu_src );

    input [15:0]   read_data2, offset;
    input          alu_src;
    output [15:0]  read_data3;

    assign read_data3 = ( alu_src ) ? offset : read_data2;
endmodule
```

- mux_3

```
`timescale 1ns/1ns
module mux_3 ( write_data, read_data, alu_out, memto_reg );

    input [15:0]   read_data, alu_out;
    input          memto_reg;
    output [15:0]  write_data;

    assign  write_data = (memto_reg)? read_data : alu_out;
```

```
endmodule
```

- mux_4

```
`timescale 1ns/1ns
module mux_4 ( unjump_pc, add1_pc, beq_pc, zero, branch );

    input [4:0]    add1_pc, beq_pc;
    input          zero, branch;
    output [4:0]   unjump_pc;

    assign  unjump_pc = ( zero & branch ) ? beq_pc : add1_pc;
endmodule
```

- mux_5

```
`timescale 1ns/1ns
module mux_5 ( next_pc, jump_pc, unjump_pc, jump );

    input [4:0]    jump_pc, unjump_pc;
    input          jump;
    output [4:0]   next_pc;

    assign         next_pc = ( jump )?    jump_pc : unjump_pc;
endmodule
```

- pc

```
`timescale 1ns/1ns
module pc ( pc, next_pc, clk, rst, set );

    input [4:0]    next_pc;
    input          lk, rst, set;
    output [4:0]   pc;

    reg [4:0]      pc;

    always @( posedge clk )
            if( ~rst && set)
```

```
                pc <= 5'b0;
        else
                pc <= next_pc;
endmodule
```

- register

```
`timescale 1ns/1ns
module register ( read_data1, read_data2, ins_rs, ins_rt, write_addr,
                write_data, reg_write, clk, rst );

    parameter       size = 8;

    input [2:0]     ins_rs, ins_rt, write_addr;
    input [15:0]    write_data;
    input           clk, reg_write, rst;
    output [15:0]   read_data1, read_data2;

    reg [15:0]      reg_mem [0:size-1];

    always @( negedge clk )
        if( rst )       // base regi(rs)
                reg_mem[0]      <=      16'b0;
        else if( reg_write )
                reg_mem[write_addr] <= write_data;

    assign read_data1 = reg_mem[ins_rs];
    assign read_data2 = reg_mem[ins_rt];
endmodule
```

2. テストプログラム（1から100までの和）

前提条件として、データメモリの 0 番地に 0、1 番地に 1、2 番地に N＋1(この場合 101)が格納されており、合計結果を命令メモリの 8 番地へストアするものとする。

```
0 : 0001_000_001_000001;        //lw, $1, mem[1];     ループカウンタのロード
1 : 0001_000_100_000001;        //lw, $4, mem[1];     '1'のロード
2 : 0001_000_010_000010;        //lw, $2, mem[2];     N のロード
3 : 0001_000_011_000000;        //lw, $3, mem[0];     '0'のロード
```

```
4 : 0100_001_010_000011;      //beq, $1, $2, 8;     N までの和が求められたならば 8 へ分岐
5 : 0000_011_001_011_000;     //add, $3, $3, $1;    足す数の更新
6 : 0000_001_100_001_000;     //add, $1, $1, $4;    ループカウンタの更新
7 : 1000_0000_0000_0100;      //jump, 4;            第 4 命令へジャンプ
8 : 0010_000_011_001000;      //sw, $3, mem[8];     最終的な結果を命令メモリの 8 番地へストア
9 : 1111_000_000_000000;      // halt;              終了
```

3. テストプログラム（5つの中の最大値）

前提条件として、データメモリの 0 番地に 1、1 番地に 5(条件判定)、2 番地に 4(ループカウンタ初期値)、3 番地から 7 番地に任意の整数（比較対象となる数）が格納されており、最大値を命令メモリの 31 番地へストアするものとする。

```
0 : 0001_000_011_000001;     //lw, $3, mem[1];    2 回目の比較に必要な値が格納先アドレスをロード
1 : 0001_000_100_000010;     //lw, $4, mem[2];    ループカウンタのロード（この場合4）
2 : 0001_000_101_000000;     //lw, $5, mem[0];    ループカウンタ更新の際に必要な'1'のロード
3 : 0001_000_001_000011;     //lw, $1, mem[3];    最初に比較する第 1 オペランドのロード
4 : 0001_000_010_000100;     //lw, $2, mem[4];    最初に比較する第 2 オペランドのロード
5 : 0000_001_010_110_111;    //sl, $6, $1, $2;    レジスタ＄1 と＄2 を比較し、大きい方を＄6 へ格納
6 : 0100_110_010_000010;     //beq, $6, $2, 9;    レジスタ＄2 と＄6 を比較し、等しければ 9 へ分岐
7 : 0001_011_010_000000;     //lw, $2, mem[$3];   小さかった方の値を新たな値へ置き換える
8 : 1000_000000_001010;      //jump, 10;          第 10 命令へジャンプ
9 : 0001_011_001_000000;     //lw, $1, mem[$3];   小さかった方の値を新たな値へ置き換える
10 : 0000_011_101_011_000;   //add, $3, $3, $5;   ロード先アドレスの更新
11 : 0000_100_101_100_001;   //sub, $4, $4, $5;   ループカウンタの更新
12 : 0010_000_110_011111;    //sw, $6, mem[31];   比較し大きかった方の値をストアする
13 : 0100_100_000_000001;    //beq, $4, $0, 15;   ループカウンタを比較し終了条件を判定する
14 : 1000_000000_000101;     //jump, 5;           第 5 命令へジャンプ
15 : 1111_000000_000000;     //halt;              終了
```

4. テストプログラム（ユーグリット互除法による最大公約数）

前提条件として、データメモリの 0 番地に 1、1 番地、2 番地に任意の整数(比較対象となる数)が格納されており、得られた最大公約数を命令メモリの 31 番地へストアするものとする。

```
0 : 0001_000_001_000000;     //lw, $1, mem[0];    条件判定に使用する' 1' をレジスタにロード
1 : 0001_000_010_000001;     //lw, $2, mem[1];    比較する対象となる値をロード
2 : 0001_000_011_000010;     //lw, $3, mem[2];    比較する対象となる値をロード
```

```
3 : 0000_010_011_010_001;      //sub, $2, $2, $3;     最大公約数の計算開始
4 : 0100_010_000_001000;       //beq, $2, $0, 13;     終了判定
5 : 0000_010_000_100_011;      //slt, $4, $2, $0;     レジスタ$2 と 0 の大小比較
6 : 0100_100_001_000001;       //beq, $4, $1, 7;      大小比較の結果レジスタ$2 が負ならば
7 : 1000_0000_0000_0011;       //jump, 3;             第 3 命令へジャンプ
8 : 0000_010_011_010_000;      //add, $2, $2, $3;     加算命令
9 : 0000_010_000_101_000;      //add, $5, $2, $0;     レジスタ$2 を$5 へ退避(値の交換)
10 : 0000_011_000_010_000;     //add, $2, $3, $0;     レジスタ$3 を$2 へ上書き(値の交換)
11 : 0000_101_000_011_000;     //add, $3, $5, $0;     レジスタ$5 を$3 へ上書き(値の交換)
12 : 1000_0000_0000_0011;      //jump, 3;             第 3 命令へジャンプ
13 : 0010_000_011_011111;      //sw, $3, mem[31];     得られた最大公約数をメモリの 31 番地へストア
14 : 1111_000000_000000;       //halt;                終了
```