卒業論文

ハードウェア記述言語による
教育用マイクロプロセッサの設計（II）

氏名：藤原淳平
学籍番号：2210990195－8
指導教員：山崎勝弘教授
提出日：2003 年 2 月 21 日

立命館大学理工学部情報学科

内容概要

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 Verilog-HDL を用いて、マルチサイクル・パイプラインプロセッサを設計する事でアーキテクチャを理解する。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、もともと Verilog-XL というシミュレータ用に開発されたので、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マルチサイクルプロセッサを設計する。続いてそのマルチサイクルプロセッサの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慶應義塾大と東京工科大が共同開発した実験教育用ＣＰＵである 16bit RISC PICO の命令セットを基本としている。命令語長を 16 ビットとし、3 形式に分類される全 27 命令を実現する 16 ビットマイクロプロセッサである。この命令セットを用いてテストプログラムを作成し、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最大値、最大公約数、バブルソートを算出するものである。また、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。ハードウェア規模はマルチサイクル方式よりパイプライン方式の方が大規模で、マルチサイクル方式よりパイプライン方式が最大で 2.85 倍の速度向上が得られた。

目次

図目次

表目次

1　はじめに

1970 年代、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、設計規模が増大していった。1990 年代半ばには、10 万ゲートから 50 万ゲートの規模だったものが、現在では 10 倍を超える数百万ゲートに達した。このように、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。また、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、設計規模がさらに増大している。

LSI 設計は、1980 年代に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、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では困難になり、設計環境および設計フローの変化が必要であった。1990 年代には、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）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。HDL 設計ではＲＴＬ設計の段階で回路の記述と検証を行い、ゲートレベルへの展開を自動的に行う。これにり、回路記述の簡単化・シミュレーションの早期開始可能・ゲートレベル設計の自動化というメリットが生まれた。また、以前までのボトムアップ的なゲートレベル設計ではシミュレーションはゲートレベルに展開してから行っていた。このためシミュレーションでバグが見つかっても、それがどのゲートの接続の誤りによって発生したのか、上位レベルの設計の誤りにより発生したのか、すぐに判断することが困難だった。そのため、開発期間に長い時間を要した。しかし、トップダウン的なＨＤＬ設計では、早い段階からシミュレーションを行うことができるので、バグの発見、ＨＤＬの修正、シミュレーションの繰り返しをすばやく行うことができるようになり、開発期間の短縮や早い段階での高い設計品質を確保に繋がった。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。

以上のような背景を踏まえ、本研究ではハードウェア記述言語 Verilog-HDL によるマルチサイクル/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、マルチサイクル/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、設計したマルチサイクル/パイプラインプロセッサは慶應義塾大と東京工科大が共同開発した実験教育用ＣＰＵである 16bit RISC PICO の命令セットを基本としている。命令語長を 16 ビットとし、3 形式に分類される全 27 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 HDL による設計、機能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 章でハードウェア記述言語によるシステム設計について述べる。第 3 章でマルチサイクルプロセッサの設計、及びシミュレーションによる動作検証について述べ、第 4 章ではパイプラインの設計について述べる。マルチサイクルプロセッサのパイプライン化から、データハザードやフォワーディング、制御ハザードの詳細、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５ 章ではこれまで作成したマルチサイクルプロセッサとパイプラインプロセッサの性能比較について述べ、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(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、時間の概念があることが最大の違いであり、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の設計記述レベルにはゲートレベル、ＲＴＬレベル、動作レベルの 3 つがある。

また、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。その中で、VHDL、Verilog-HDL、SFLについて述べる。

・ VHDL

1970年代にスタートした米国防省のVHSIC(Very High Speed IC)プロジェクトで開発されたHDLであり、1987年にIEEE Std-1076として標準化された。VHDLはシステムからスイッチに至るハードウェア全般の記述を対象としており、ハードウェアの構造と動作の両方を記述できる。VHDLでは回路を階層的にとらえ、下位の階層で定義された回路を組み合わせて接続することにより、上位の回路を記述する。すなわち最下層の回路の入出力関係をソフトウェア・プログラムのような形で記述し、それより上位の回路を構造的に記述する。VHDLはユーザ定義のタイプ、抽象度の高いデータタイプ、回路構成をコントロールするためのコンフィギュレーションなどに特徴がある。

・ Verilog-HDL

米国Geteway社(現Cadence社)でVerilog-XLというシミュレータ用に開発されたHDLであり、世界的に普及している。しかし、標準化されたのは遅く、1995年にIEEE Std-1364となった。Verilog-HDLはシステムからスイッチに至るハードウェア全般の記述を対象としており、ハードウェアの構造と動作の両面から記述できる。VHDLと同じようにVerilog-HDLでも回路を階層的に記述する。すなわち最下層の回路は動作的に記述し、上位の回路は構造的に記述する。特徴としてはRTLに的を絞った簡潔な記述、テストパタン記述言語としての高い効率性などが上げられる。

・ SFL

日本で開発されたHDLであり、制御回路の記述に特徴がある。SFLでは回路をデータパスと制御回路を明示的に分けて記述し、制御回路の記述中からデータパスを構成する部品を手続き的に呼び出すことにより回路の動作を記述する。データパスを構成す

る部品間の接続やデータパスと制御回路の間の接続は合成によって行われる。制御回路の記述は単一クロックの同期式ステートマシンを基本としている。SFLでは階層的なステートマシンの表現が可能である。

2.2 HDL設計の特徴

HDL 設計の手法には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
- 回路図作成してからゲートレベルシミュレータで設計する
- テクノロジの設計を最初に決めなければならない
- デバックが困難
- 設計の最終段階まで製品の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
- シミュレーションが早期開始可能
- 初期段階からバグを発見できる
- 回路記述の簡単化
- 論理合成ツールによる時間短縮、設計品質向上
- 設計資産の再利用性が高い

ボトムアップ設計とトップダウン設計の比較を図１に示す。

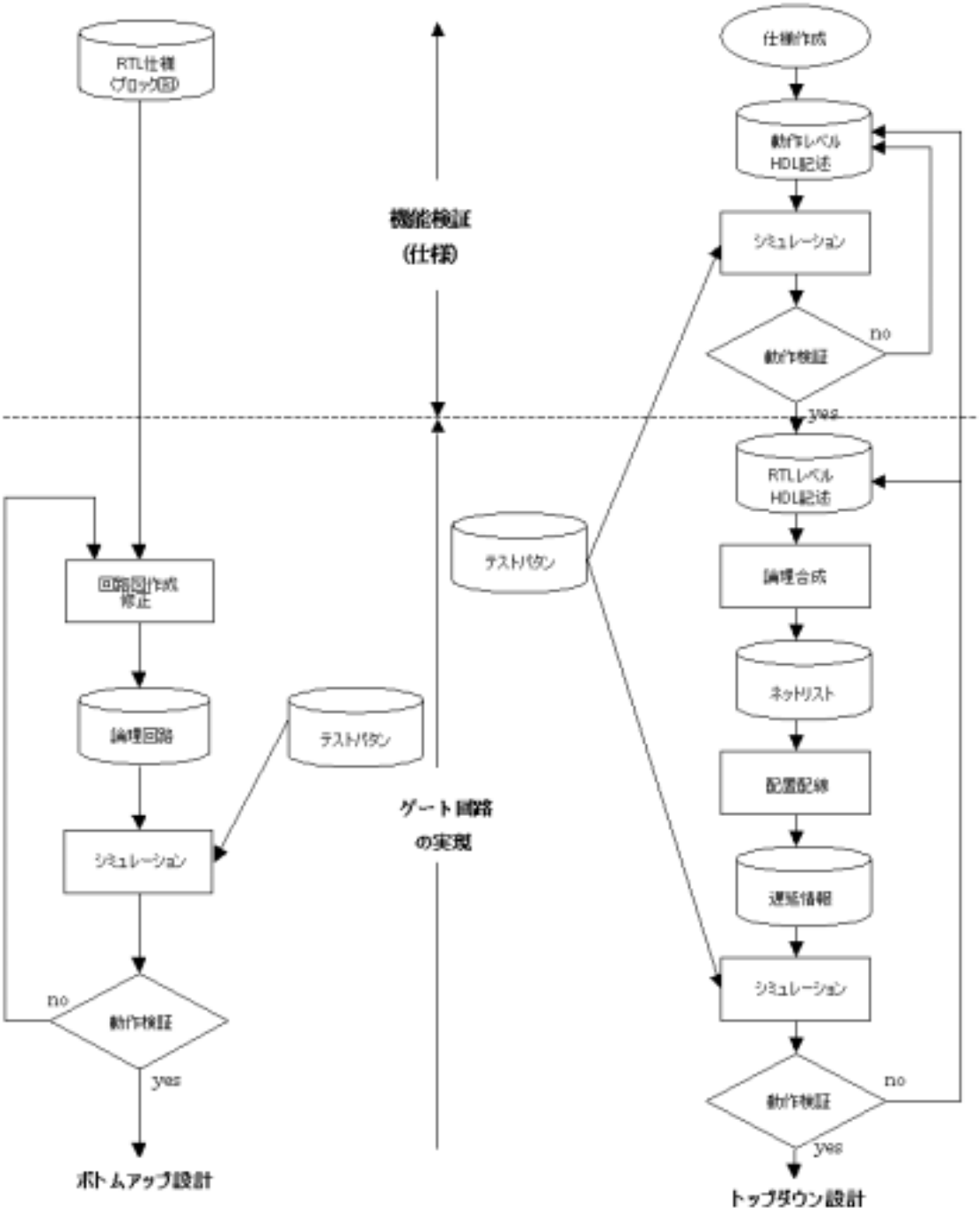

図１ ： ボトムアップ設計とトップダウン設計の比較

2.3 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。

2.4 プロセッサアーキテクチャの分類

マイクロプロセッサには、命令実行方式の違いにより、いくつかのプロセッサアーキテクチャに分類することができる。以下に代表的なプロセッサアーキテクチャの特徴を示す。

(1)単一サイクル方式

単一サイクル方式は、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。そのため、どのデータパス資源も 1 命令当たり 2 回以上使用できない。また、処理が最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い。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

マルチサイクル方式は、ひとつの命令を複数のステップ（命令フェッチ、レジスタフェッチ、演算実行、メモリアクセス）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。命令によってクロックサイクルが異なる。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

パイプライン方式は、ひとつの命令を各実行ステップ（ステージ）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て同時並行的に実行するものである。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が多いほど高速化が期待できるというわけではない。実際にはデ

ータハザードや制御ハザードなどの障害によって、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が起こるためである。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。パイプライン方式の詳細は4章で述べる。

(4)スーパースカラ方式

スーパースカラ方式は、プロセッサ内のユニットを複数発行して、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もの。発行された命令を一時的に蓄えておくリザベーションステーション、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、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、その割り当てを行うレジスタリネームイング及び、分岐予測に基づいた命令の投機実行制御などがある。命令の実行順序はプログラムの順序とは異なったものになる。

(5)VILW 方式

VLIW（Very Long Instruction Word Set）方式は、多数のフィールドに分けられた非常に長い命令語長（256～1024 ビット）を持ち、その各々のフィールドで対応する演算器などの制御 を行う。命令レベルの並列性の抽出やスケジューリングをコンパイル時に行い、 並列実行するものである。このため、並列性の抽出を大域的に行うことができ、 実行時にはプロセッサは静的にスケジュールされた命令語を実行するだけで良 いので構成が簡単になり、比較的高速で高並列度のマシンをつくりやすい。し かし、コードスケジューリングが、コンパイル時にそれぞれのマシンフォーマット に合わせて静的に行われるため、バイナリレベルでの互換性が乏しい。

3　教育用マイクロプロセッサ PICO１６のアーキテクチャ

3.1　命令セットアーキテクチャ

本研究では１６bit RISC PICO16 の命令セットを使用した。その命令セットのフォーマットを図 2 に示す。

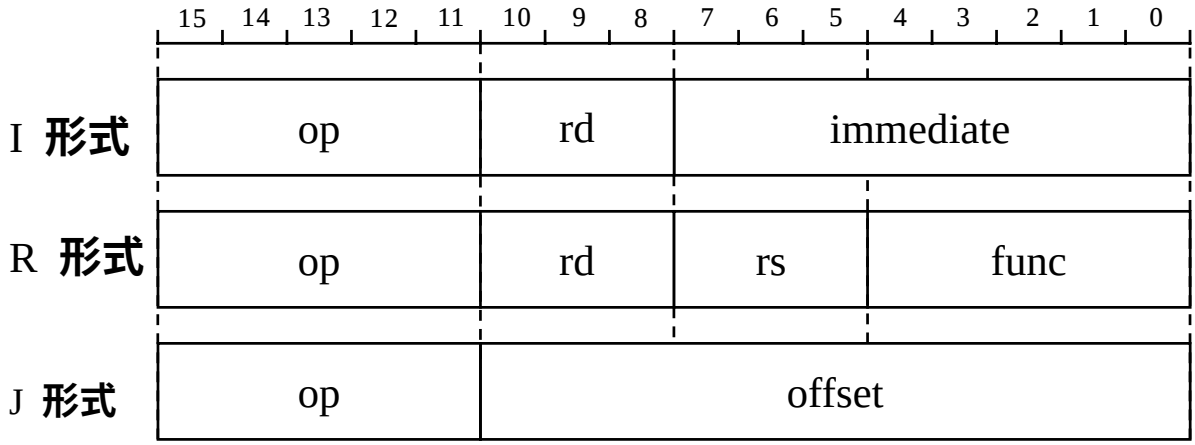

図２ ： 命令フォーマット

以下に各フィールドの詳細を示す。

op	:	opcode。命令ソースコード。命令形式の判定。5 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
rs	:	ソースレジスタ。3 ビット
offset	:	メモリ番地を示すアドレス
immediate	:	rd に対する offset 値
func	:	第 2opcode。機能コード。R 形式命令で実際の ALU の機能を明確にする。5 ビット

表１に PICO16 の命令セットの概要を示す。

表１：PICO１６の命令セット

命令形式	命令内容	オペランド	命令パターン	概要
I 形式命令	イミディエイト命令	ADDI	00110dddxxxxxxxx	d＋X をdに格納
		SUBI	00111dddxxxxxxxx	d－X をdに格納
		ANDI	00010dddxxxxxxxx	dと X の論理積をdに格納
		ORI	00011dddxxxxxxxx	dと X の論理和をdに格納
		XORI	00100dddxxxxxxxx	dとXの排他的論理和をdに格納
		LDLI	11100dddxxxxxxxx	Xをdの下 8 ビットに格納
		LDHI	11101dddxxxxxxxx	Xをdの上 8 ビットに格納
	条件分岐命令	BNEZ	01001dddxxxxxxxx	d≠0ならば相対分岐
		BEQZ	01010dddxxxxxxxx	d＝0ならば相対分岐
		BMI	01011dddxxxxxxxx	d＜0ならば相対分岐
		BPL	01100dddxxxxxxxx	d≧0ならば相対分岐
	無条件分岐命令	JR	01110dddduuuuuuuuu	dの内容に絶対分岐
		JALR	01000dddduuuuuuuuu	R 7に戻り番地を格納して dの内容に絶対分岐
R 形式命令		NOP	00000dddsss00000	有効な操作は行われない
		MV	00000dddsss00001	レジスタsの値をdにコピーする
		AND	00000dddsss00010	dとsの論理積をdに格納
		OR	00000dddsss00011	dとsの論理和をdに格納
		XOR	00000dddsss00100	dとsの排他的論理和をdに格納
		NOT	00000dddsss00101	sのＮＯＴをdに格納
		ADD	00000dddsss00110	d＋sをdに格納
		SUB	00000dddsss00111	d－sをdに格納
		LD	00000dddsss01000	sで示す番地の中身をdに格納
		ST	00000dddsss01001	dで示す番地にsを格納
		SL	00000dddsss01100	sの１ビット左シフトをdに格納
		SR	00000dddsss01101	sの１ビット右シフトをdに格納
J 形式命令	ジャンプ命令	JAL	01101xxxxxxxxxxxx	R 7に戻り番地を格納して相対分岐
		JMP	01111xxxxxxxxxxxx	無条件相対分岐

また、16bit RISC PICO は汎用レジスタマシンであり、以下の特徴を持つ。

・Load/Store マシン(register-register マシン)である。

１６ビットの汎用レジスタを８つ持ち、計算はレジスタ間でのみ許される。すなわち、計算を行うためには、必ずメモリからレジスタにデータを Load する必要がある。メモリアドレスの指定は全くできない。

・単一命令長である。

命令長を１６ビットに固定することによって、命令の作り方は難しい部分があるが、制御は容易になる。しかし、イミディエイト命令の直値を全部の命令の中に納めることができない。そこで、命令中の 8 ビットデータの直値を符号拡張して 16 ビットにする。符号拡張とは 2 の補数表現の符号を保持したまま、ビット幅を拡張する操作のことである。

3.2　アドレス指定方式

PICO16 の LD・ST 命令はレジスタ間接アドレッシングを用いている。すなわち
LD　r1、(r2)
を実行すると、r2 の内容が指す番地のデータが r1 にロードされ、
ST　(r1)、r2
を実行すると、r1 の内容が指す番地に、r2 を格納する方式をとる。この方式でメモリアクセスをするためには、最初にアクセスする番地をレジスタ上にセットする必要があるが、アドレスを格納したレジスタを加算、減算したりすることによって、配列やスタックを実現したり、ポインタを作ったりできるという利点がある。

メモリのアドレスはバイトアドレッシングといい、8bit 単位にアドレスが振られている。メモリ中のデータの順序付けの方法には、以下の二つの方法がある。

・Big Endian　：２進数として見た場合の最上位ビットを 0 桁目と考え、アドレも上位の 8 ビットを低いほうの番地とする方法

・Little Endian　：２進数として見た場合最下位ビットを 0 桁目と考え、アドレスも下位の 8 ビットを低いほうの番地とする方法

PICO１６では、データのビット順に関しては、Little Endian、アドレスのつけ方は Big Endian を用いている。この方法で 16 ビットデータを扱う場合は{偶数番地８ビット、奇数番地 8 ビット}の組を一つの 16 ビットデータとして偶数番地で代表させる。
以下の図２でメモリの番地付けを示す。

LSB＼MSB	15 14 ‥‥‥‥ 9	8 7 6 ‥‥‥‥ 1 0
0	0	1
2	2	3
4	4	5
6	6	7
	・ ・ ・	・ ・ ・
n-2	n-2	n-1
n	n	n+1

図 3：メモリの番地付け

４　マルチサイクル方式ＰＩＣＯ１６の設計

4.1　マルチサイクル方式ＰＩＣＯ１６のアーキテクチャ

図 3 にマルチサイクル方式ＰＩＣＯ１６のデータパスを示す。

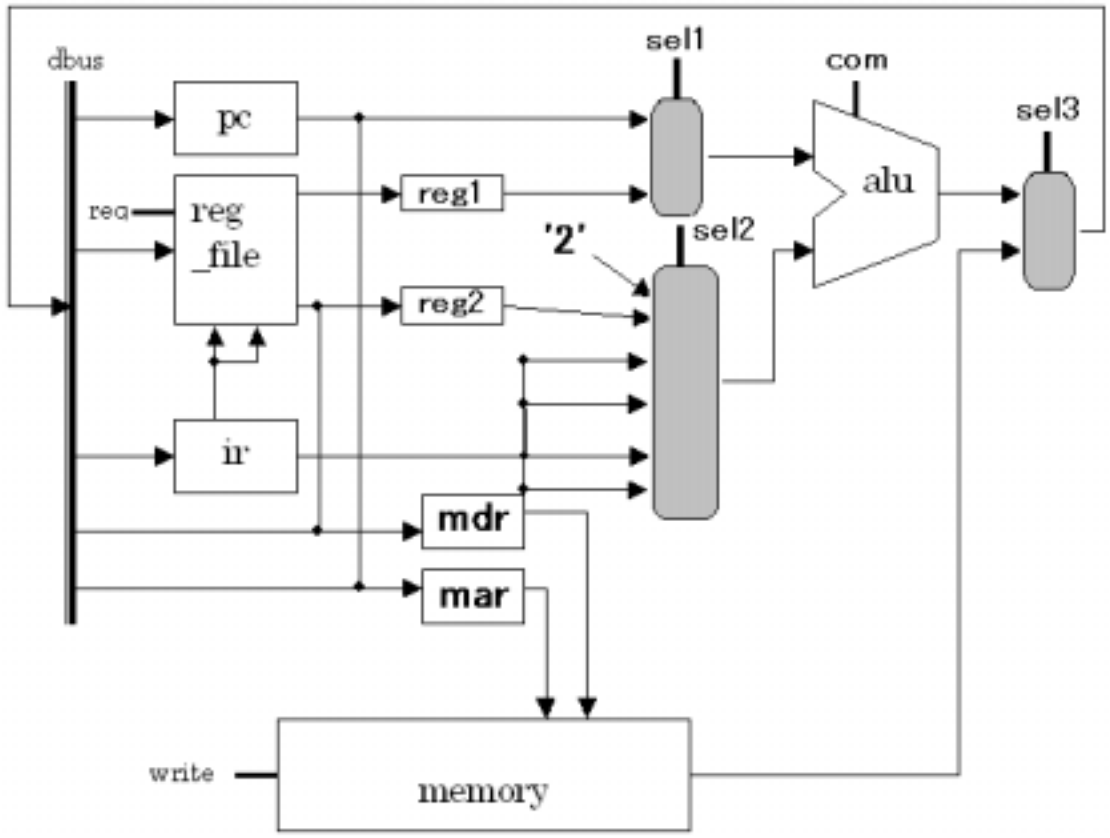

図 4：マルチサイクル方式ＰＩＣＯ１６のデータパス

マルチサイクル方式ＰＩＣＯ１６は以下の機能ユニットで構成されている。それぞれの機能を以下に示す。

- プログラムカウンタ(pc) ・・・ **これから実行する命令の番地を保持する。**
- 命令レジスタ(ir) ・・・ **メモリからフェッチしてきた命令を格納する。**
- レジスタ(reg_file) ・・・ **データなどを格納する。8 つの汎用レジスタを格納。**
- メモリアドレスレジスタ(mar) ・・・ **メモリアクセスする際、アドレスを格納する。**
- メモリデータレジスタ(mdr) ・・・ **データをメモリに書き込む際、データを格納する。**
- データ格納レジスタ(reg1) ・・・ **レジスタから読み出してきたデータを格納する。**
- データ格納レジスタ(reg2) ・・・ **レジスタから読み出してきたデータを格納する。**
- メモリ(memory) ・・・ **命令とデータを格納する。**
- ALU(alu) ・・・ **算術演算を行う。**

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(ir)に保存しておく必要がある。命令レジスタは次の命令がメモリからフェッチしてきたときに書き換えられる。

データ格納レジスタ(reg1,reg2)は、レジスタ(reg_file)が読み出しを行う時に時間を要するため、読み出したデータを保持できるようにしている。

4.2　命令の実行と制御

図４にマルチサイクル方式ＰＩＣＯ１６の状態遷移図を示す。

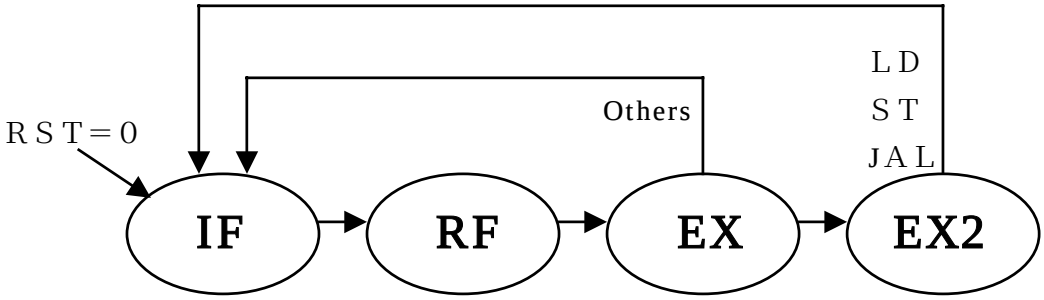

図５：状態遷移図

マルチサイクル方式ＰＩＣＯ１６は、

１．命令フェッチ（ＩＦ）

２．レジスタフェッチ（ＲＦ）

３．実行１（ＥＸ）

４．実行２（ＥＸ２）

の各ステップを１クロックサイクルで行う方式である。

以下の表 3 に各ステップについて述べる。

表２：マルチサイクル方式ＰＩＣＯの実行ステップ

ステップ/命令	I 形式	R 形式	ロード命令	ストア命令	JMP 命令	JAL 命令
1	命令フェッチ					
2	命令のデコードとレジスタフェッチ					
3	演算実行		メモリアドレスの計算		pc を分岐先アドレスへ	pc の保存
4			メモリの読み出し	メモリの書き込み		pc を分岐先アドレスへ

すべての命令のおいてステップ 1 の命令フェッチと、ステップ 2 の命令のでコードとレジスタフェッチは共通している。

ステップ 1 では、アドレスバスに PC の値、つまり取ってくる命令のアドレスを載せ、

メモリを読み出す。そして、読み出してきた命令を命令レジスタ ir に格納する。

ステップ 2 では、取ってきた命令中のオペランドフィールド(10～8bit と 7～5bit)に入っているレジスタ番号に従ってレジスタファイルから読み出しを行い、それぞれをレジスタ reg1,reg2 に格納する。また、イミディエイト命令などでは、7～5bit はイミディエイトの数値の一部であるため、レジスタ reg2 のデータは利用されないが、読み出しても害はないため一律に読み出す。この間 ALU が空いているので、この間に PC を＋２させる。ここで、ストア命令であった場合は、読み出したレジスタの値をレジスタ mdr にも格納しておく。

ステップ３では命令によって処理の仕方が変わってくる。R 形式では、第二オペコードである 4～0bit によりそれぞれの命令を判別する。また、この 4～0bit は ALU のコマンドに対応しているため、そのコマンドを用いて演算を実行し結果をレジスタに格納する。ここで R 形式のロード命令とストア命令では、それぞれアクセスするアドレスの保持するレジスタの中身をレジスタ mar に転送しておく。I 形式では、イミディエイトデータを 16 ビットに符号拡張し、レジスタ reg1 に格納されているデータと演算を行って、結果を命令レジスタのディスティネーションオペランドの指すレジスタに格納する。JMP 命令では、ｐｃに分岐先アドレスを格納し、JAL 命令では、ｐｃを格納するレジスタをあらかじめ決めておき、そこにｐｃを格納する。

ステップ 4 では、ロード命令とストア命令と JAL 命令の作業だけである。ロード命令とストア命令では、メモリアクセスを行う。LD 命令の場合、読み出したデータを直接レジスタファイルに格納する。ST 命令ではメモリへの書き込みを行っている。また JAL 命令では、ｐｃに分岐先アドレスを格納する。

4.3　HDL によるマルチサイクル PICO16 の設計

図３のデータパスにより設計した PICO にはモジュールが、bpico、reg_fire、alu16、addsub16、mux2_1(s1)、sel6_1(s2)、mux2_1(s3)の７つがある。以下に各モジュールの入出力と動作を説明する。

・モジュール bpico

最上位モジュールであり、制御信号の判断、ステートの移動、命令の判別、各種レジスタへの格納などを行う。以下の表 3 に bpico の入出力信号を示す。

表 3：bpico の入出力信号

module 名	bpico	
入力ピン	datain[15:0]	メモリからのデータ入力

	sel2_0[15:0]	PC インクリメント用データ (16 進数 0002)
	CLK	クロック
	RST	リセット
出力ピン	dataout[15:0]	メモリへのデータ出力
	addressout[7:0]	メモリへのアドレス出力
	write	メモリに書き込むとき1、メモリから読み出すとき0

・モジュール reg_file

モジュール bpico の下位モジュールであり、命令を実行するためのレジスタのアドレスを受け取り各種演算命令の結果を出力する。また、その結果を再び格納する。以下の表 4 に reg_file の入出力信号を示す。

表 4：reg_file の入出力信号

module 名	reg_file	
入力ピン	addr1[2:0]	演算などに用いるレジスタファイルのアドレスである 命令が格納されているレジスタ ir の 10～8bit を入力
	addr2[2:0]	レジスタファイルのアドレスである命令が格納されて いるレジスタ ir の7～5bit を入力
	din[15:0]	演算結果などのデータをレジスタに入力
	req	レジスタに書き込むとき1、レジスタから読み出すとき0
	CLK	クロック
	RST	リセット
出力ピン	dout1[15:0]	addr1 の示すアドレスにあるデータを出力
	dout2[15:0]	addr2 の示すアドレスにあるデータを出力

・モジュール alu16

モジュール bpico の下位モジュールであり、mux2_1(s1)と sel6_1(s2)のそれぞれから出力データを受け取り、命令レジスタ ir から受け取った opcode によって決められた演算を実行する。また、ステート RF では pc のインクリメントに使われ、ステート EX では算術演算を実行する。以下の表 5 に alu16 の入出力信号を示す。

表 5：alu16 の入出力信号

module 名	alu16	
入力ピン	alu_ina[15:0]	s1(mux2_1)の出力から入力

	alu_inb[15:0]	s2(sel6_1)の出力から入力
	com[3:0]	第 2 オペコードの[3:0]によって演算を決定
出力ピン	alu_out[15:0]	演算結果を出力

・モジュール addsub16

モジュール alu16 の下位モジュールであり、算術演算である加算と減算を行い、結果を alu16 に渡す。以下の表 6 に addsub16 の入出力信号を示す。

表 6：addsub16 の入出力信号

module 名	addsub16	
入力ピン	addsub_ina[15:0]	alu 演算が加算と減算の場合に入力
	addsub_inb[15:0]	alu 演算が加算と減算の場合に入力
	sub	1なら減算、0なら加算
出力ピン	addsub_out[15:0]	演算結果を出力

・モジュール mux2_1(s1)

モジュール bpico の下位モジュールであり、レジスタ reg1 からのデータとレジスタ pc からのデータが入力され、どちらかを選択して出力する。以下の表 7 に mux2_1(s1)の入出力信号を示す。

表 7：mux2_1(s1)の入出力信号

module 名	mux2_1(s1)	
入力ピン	a[15:0]	レジスタ reg1 から出力されたデータを入力
	b[15:0]	レジスタ pc から出力されたデータを入力
	sel1	0なら a の値を出力し、1 なら b の値を出力
出力ピン	out	選択された値を出力

・モジュール sel6_1(s2)

モジュール bpico の下位モジュールであり、pc のインクリメント用のデータ、レジスタ reg2 からのデータ、LDLI,ADDI,SUBI,BNEZ,BEQZ,BMI,BPL 命令用に符号拡張したデータ、LDHI 命令用に符号拡張したデータ、ANDI,ORI,XORI 命令用に符号拡張したデータ、JAL,JMP 命令用に符号拡張したデータがそれぞれ入力され、どれかを選択して出力する。以下の表 8 に sel6_1(s2)の入出力信号を示す。

表 8：sel6_1(s2)の入出力信号

module 名	sel6_1(s2)	
入力ピン	a[15:0]	PC インクリメント用データ 16 進数 0002 を入力
	b[15:0]	レジスタ reg2 から出力されたデータを入力
	c[15:0]	ir[7]を上位 8 ビットに拡張、下位 8 ビットに ir[7:0]を入力
	d[15:0]	上位 8 ビットは 0、下位 8 ビットに ir[7:0]を入力
	e[15:0]	上位 8 ビットに ir[7:0]、下位 8 ビットは 0 を入力
	f[15:0]	上位 5 ビットに ir[10]を拡張、下位 10 ビットに ir[10:0]を入力
	sel2[2:0]	000 なら a の値を出力、001 なら b の値を出力、010 なら c の値を出力、011 なら d の値を出力、100 なら e の値を出力、101 なら f の値を出力
出力ピン	out	選択された値を出力

・モジュール mux2_1(s3)

モジュール bpico の下位モジュールであり、ALU からのデータとメモリからのデータが入力され、どちらかを選択して出力する。以下の表 9 に mux2_1(s3)の入出力信号を示す。

表 9：mux2_1(s3)の入出力信号

module 名	mux2_1(s3)	
入力ピン	a[15:0]	ALU から出力されたデータを入力
	b[15:0]	メモリから出力されたデータを入力
	Sel3	0 なら a の値を出力し、1 なら b の値を出力
出力ピン	out	選択された値を出力

4.4 シミュレーションによる動作検証

本研究では XILINX 社 FoundationISE5.1 を用いて HDL 設計を行い、MTI 社の ModelesimXE 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「Spartan2E」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 種類がある。

・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 を配置する前のシミュレーションである。

・Post-Place&Route シミュレーション
配置敗戦後のゲートレベルシミュレーション、FPGA にダウンロードした場合にもこのシミュレーション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シミュレーションである。

しかし、本研究では、メモリはテストフィクチャ中で呼び出されるため、メモリの論理合成を行っていない。よってメモリの呼び出し遅延がない。そのためここでは Post-Map シミュレーションの結果を示す。

シミュレーションを行う際に使用するテストパタンとして「8 つの数の最大値」、「ユーグリッド互除法による最大公約数」、及び「バブルソート」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評価を行った。以下の表 10 にマルチサイクル方式 PICO１６のシミュレーション結果を示す。

表１０：マルチサイクル方式 PICO１６のシミュレーション結果

	命令数	実行時間(ns)	クロック数	CPI
8 個の数の最大値	57	36100	180	3.16
2 数の最大公約数	27	17500	84	3.11
バブルソート	248	161900	809	3.07

(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3.0～3.9 の間の数値である。８ つの数の最大値のCPI が3.16 と3 つのテストパタン中最大なのは、命令列中にロード・ストア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、ロード命令、ストア命令、JAL命令、JALR命令は４クロック、それ以外の命令は3クロックを要する。すなわち、ロード命令、ストア命令、JAL命令、JALR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６章の「プロセッサの性能比較と考察」で行う。

５　パイプライン方式ＰＩＣＯ１６の設計

5.1　パイプライン方式ＰＩＣＯ１６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。今回作成したパイプラインでは命令フェッチ(IF)、レジスタフェッチ(RF)、命令実行(EX)、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ことで命令のスループットを大きく(CPI を小さくする事に等しい)し、全体的な処理時間も短縮される。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。以下の図7でパイプライン方式の動作を示す。

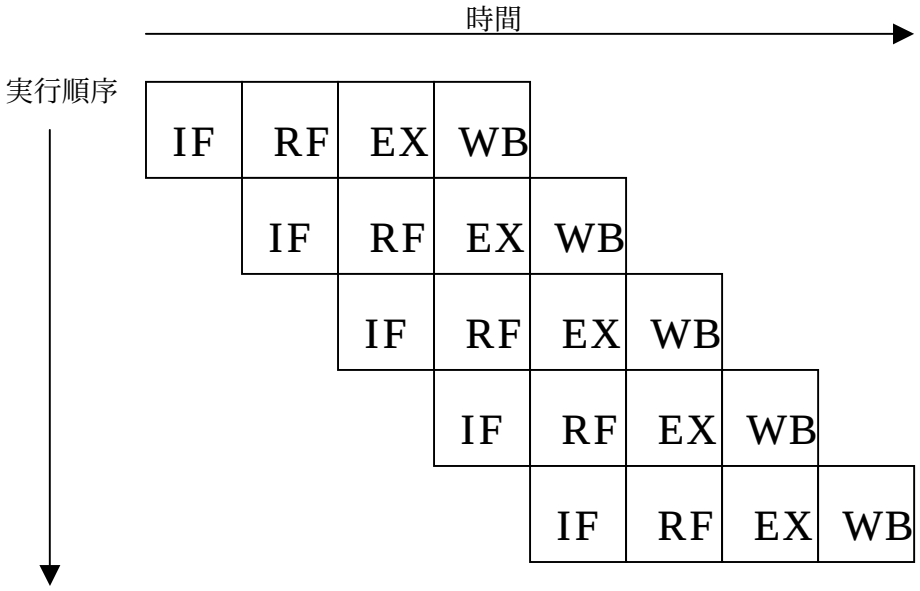

図6：パイプラインの動作

本研究では、上記に示した４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４章で示した、マルチサイクル方式PICO１６を４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。

5.2　マルチサイクル方式ＰＩＣＯ１６のパイプライン化

CPU のパイプライン化は、パイプライン化されていない CPU の制御における状態遷移を基本に考える場合が多い。よって以下のように４段パイプライン化を行う。

(1) IF ： 命令フェッチ、メモリから命令をとってくると同時に
ＰＣを2インクリメントする。
(2) RF ： レジスタフェッチ、レジスタからデータを読み出す。
ロード･ストア命令では読み書きの準備をする。
(3) EX ： 命令を実行する。
(4) WB ： 結果をレジスタファイルに書き込む。

マルチサイクル方式と異なり、各ステージでは次々と呼んでくる異なった命令が実行される。このため、以下の資源が競合する。

- ALU ： **IF 状態で命令をフェッチすると同時に、ALU を用いて、PC をインクリメントするが、ALU は EX ステージで毎クロック用いられている。**

- メモリアクセス ： **IF ステージでは毎回命令を読み出すためにメモリアクセスが必要であるが、EX ステージでロード･ストア命令を実行するにはデータの読み書きのためにメモリアクセスを行う必要がある。**

- レジスタファイル：**RF ステージではレジスタの読み出しを行い、WB ステージではレジスタへの書き込みを行う。これが同時に起こると競合が起きる。**

まず、ALU の競合は、PC のインクリメントするため専用の加算器を用意する。
次に、メモリアクセスの競合は、命令メモリとデータメモリに分離させ、同時にアクセス可能にする。
最後に、レジスタファイルの競合は、レジスタファイルのポート数を増やすことにより実現する。パイプライン方式で用いたレジスタファイルは、2 つの読み出しを同時に行うことのできるデュアルポートメモリであったが、これを 2 つの読み出し、1 つの書き込みを同時に行うことのできる 3 ポートメモリに拡張する。
以上の改造を行うことにより、今回設計したパイプライン方式 PICO１６は、資源の競合を完全になくすことができる。よって、資源の競合によってパイプラインの動作に問題が生じる構造ハザードは起きず、構造ハザードによるストールは生じない。
上記のことを踏まえて、図８にパイプラインの基本的データパスを示す。

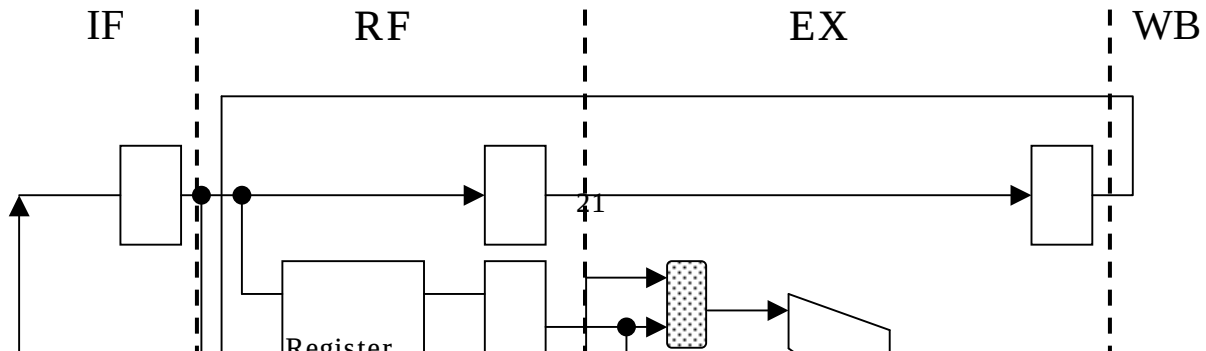

図 7：パイプラインの基本的データパス

5.2.1　データハザード

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ことがある。以下の図 9 でその例を示す。

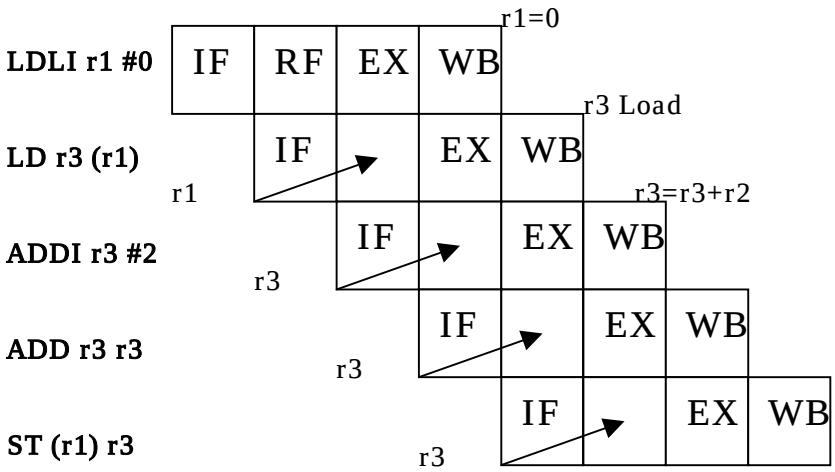

図 8：データハザード

図 9 のように命令が 1 クロックずつフェッチされて実行するため、WB ステージで計算の結果がレジスタファイルに書き込まれる前に、次の命令と、さらに次の命令が RF ステージでレジスタを呼んでしまうことによって生じる。このように、データの依

存関係によって、パイプラインが正常に動作しないことをデータハザードと呼ぶ。一般的にデータハザードには以下の 3 種類がある。

- **RAW ハザード　：データを書く前に、値を読み出されてしまう問題。**
- **WAR ハザード　：値を読み出す前に、それが書き潰されてしまう問題。**
- **WAW ハザード　：値を書きこむ前に別の値を書きこんでしまう問題。**

今回作成したパイプライン方式 PICO16 では、レジスタファイルへの書き込みは最終ステージの WB ステージで行われるため、RAW ハザードしか起こらない。

したがって、正常にどうさせるには以下の図 10 に示すように NOP 命令を入れる。その様子を図 10 に示す。

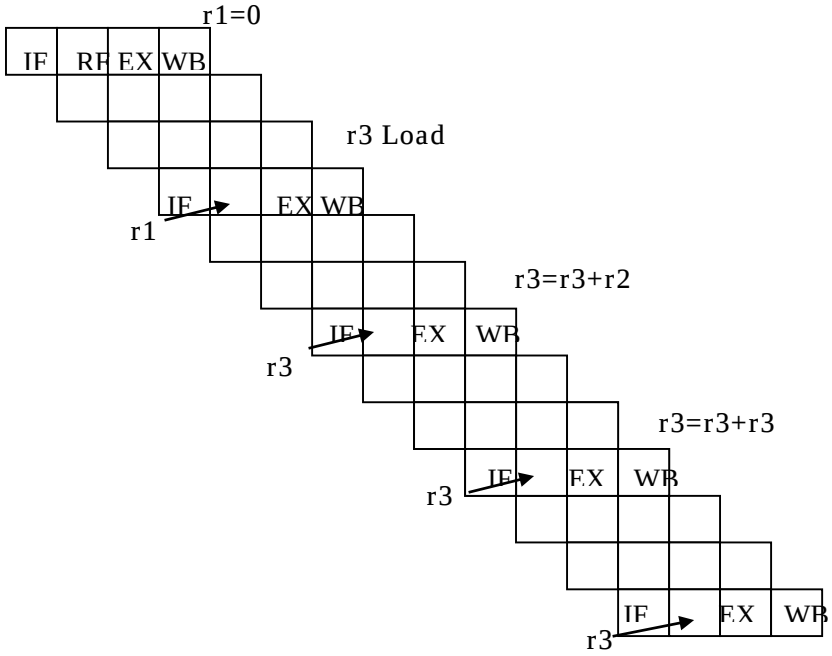

図 9：ＮＯＰによるデータハザードの回避（ストール）

この場合の NOP 命令は、水流の中の泡にたとえられ、バブルと呼ばれる。このようにデータハザードを回避するために、パイプライン中に NOP 命令を流すのも、一種のパイプラインストールである。

5.2.2　フォワーディング

データハザードを回避するためには、WB ステージでレジスタファイルに書き込む前のデータを横流しにし、読み出したデータの代わりに使う機構を設ける。このように、

命令間のデータの先送りをフォワーディングという。今回設計したパイプラインは 4 段構成なので、フォワーディングは、レジスタファイルと EX ステージの 2 箇所に設ける。

まず、レジスタファイルの書き込みのフォワーディングを設ける。これは、RF ステージの読み出しと WB ステージでの書き込みが同時に同じレジスタに対して起きた場合、書き込むと同時に、データをそのまま読み出し側へ横流しするような回路をレジスタファイルに設ける。このレジスタファイルにフォワーディングを設けることによって、以下の図 11 のように NOP 命令を流すのが 2 クロックから 1 クロックへ減少する。この様子を以下の図 11 で示す。

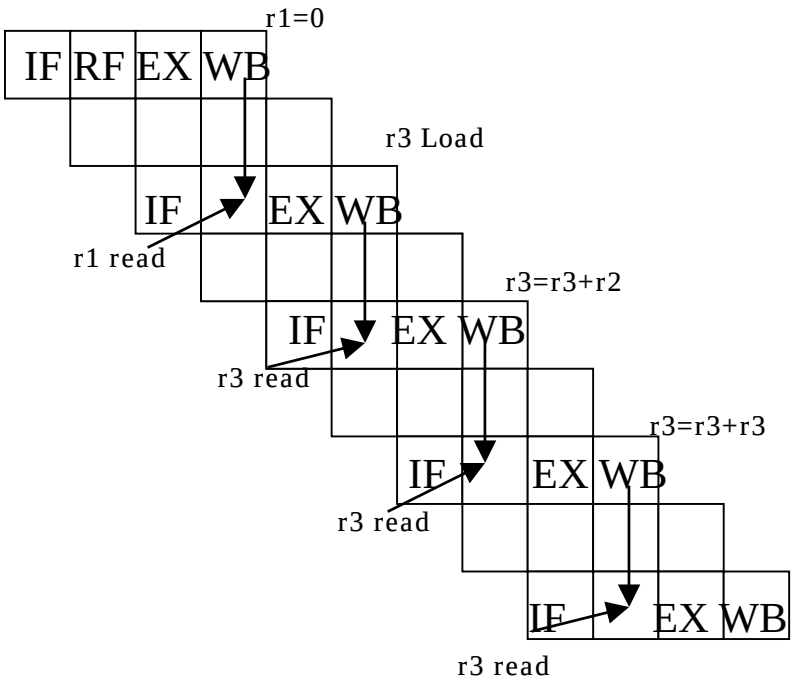

図 10：フォワーディングによるデータハザードの回避

次に、EX ステージにもフォワーディングを設ける。これは NOP 命令のバブルを完全に駆逐するために設け、EX ステージからの計算結果をそのまま横流しすることによって実現する。方法としては、EX ステージの結果を格納するレジスタの直前から RF ステージのレジスタ読み出し結果を格納するレジスタへの入力へ横流しをする。その時、レジスタの前にデータの流れを切り替えるマルチプレクサを拡張する。この EX ステージでのフォワーディングは、EX ステージで実行されている命令がレジスタに対する書き込みを伴うものであり、その書き込みレジスタの番号が、ALU で計算される入力レジスタ番号と一致している場合に用いる。

このように、2 箇所にフォワーディングを設けることによって、最初の NOP 命令と

いうバブルを使用しないで、命令を実行できる。今回設計したのは、4 ステージ構成で、メモリアクセスと演算が同一ステージで行われるために問題は発生しないが、一般の 32bit RISC は、演算とメモリアクセスに分けて５ステージ構成をとることが多い。この場合、ロード命令のメモリアクセスは演算ステージの後で実行されるため、完全にデータハザードを除去することができない場合もある。

5.2.3　制御ハザード

パイプライン方式の命令実行において、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ことを指し、その場合の命令依存関係を図１２で示す。

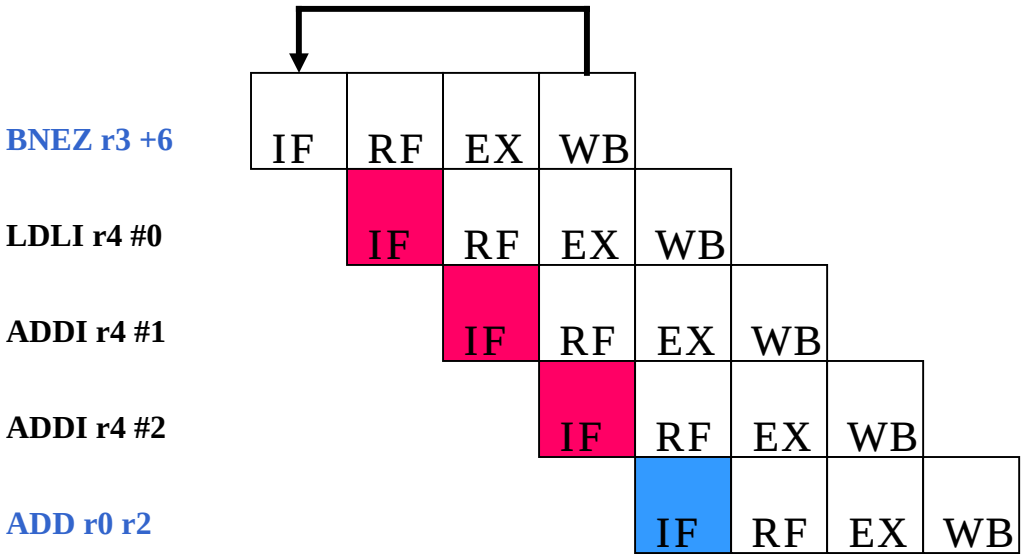

図 11 :パイプラインに対する分岐命令の影響

図１２のように BNEZ 命令がパイプラインに投入され分岐が成立しても、プログラムカウンタはすぐに分岐先の番地に戻らず、LDLI 命令、ADDI 命令、ADDI 命令の 3 つの不要な命令が実行されてから分岐後の処理が実行されている。この制御ハザードの解決策として、分岐命令の第 4 ステージに達するまでに投入できる命令を NOP 命令だけにすることであるが、これでは全体的な速度に影響が出てしまい抜本的な解決策にはならない。そこで本研究ではこの制御ハザードの解決策として、分岐命令を第 2 ステージである RF ステージで検出するように RF ステージに以下のような機構を設ける。

・　分岐命令時に、レジスタを読んで 0 かどうか判断する。図 14 の reg_a

・　飛び先の番地を計算する。図１４の adder jpc

といった二つの機構を設ける。以下の図 13 で分岐判定改良後の命令依存関係を示す。

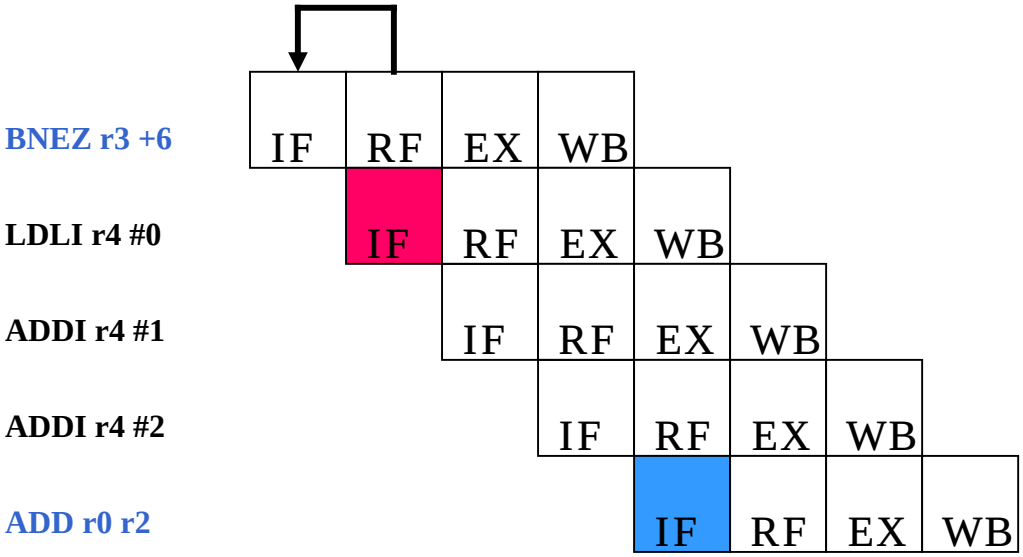

図 12：分岐判定改良後の命令依存関係

5.2.4　パイプライン方式ＰＩＣＯ１６のデータパス

　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 PICO１６のデータパスを以下の図 14 で示す。

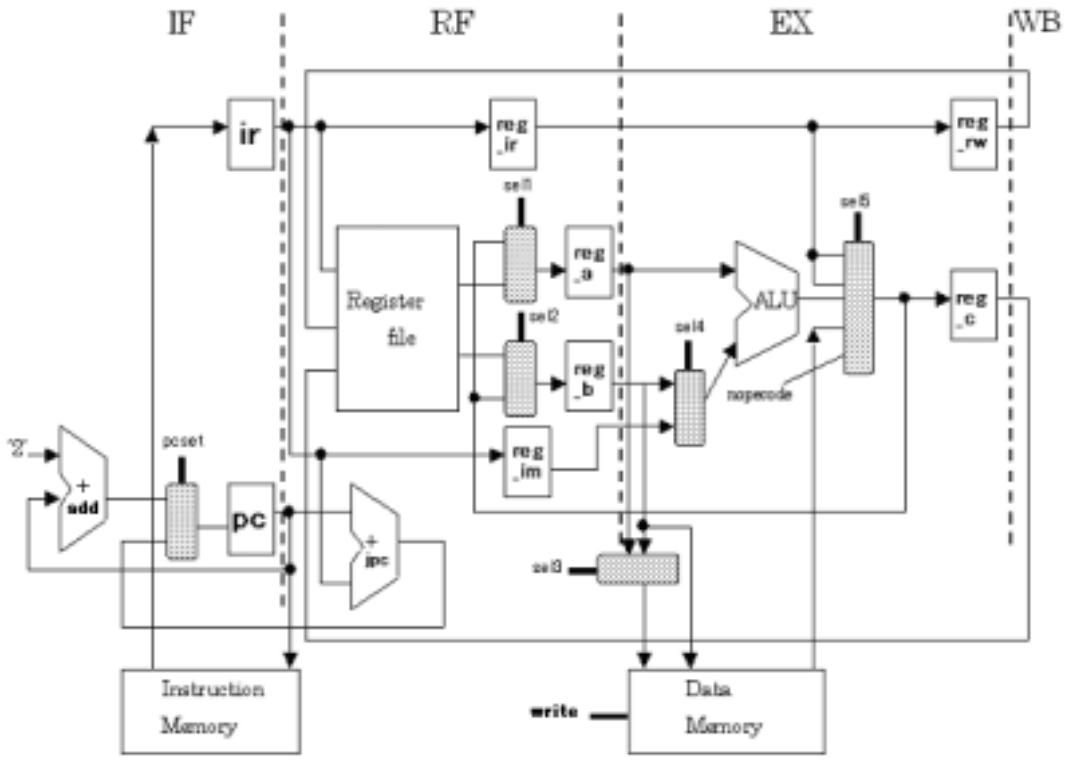

図 13：パイプライン方式ＰＩＣＯ１６のデータパス

- pc ・・・ **これから実行する命令の番地を保持する。**
- ir ・・・ **メモリからフェッチしてきた命令を格納する。**
- Register file ・・・ **データなどを格納する。8 つの汎用レジスタを格納。**
- reg_ir ・・・ IF ステージから送られてきた命令を格納する。
- reg_a ・・・ mux s1 に出力されたデータを格納
- reg_b ・・・ mux s2 に出力されたデータを格納
- reg_im ・・・ 符号拡張されたデータを格納。
- reg_rw ・・・ RF ステージから命令の第一オペコードを格納。
- reg_c ・・・ ALU 演算結果とデータメモリからのデータを格納。
- Instruction memory ・・・ **命令を格納する。**
- data memory ・・・ **データを格納する。**
- add ・・・ **PC のインクリメントを行う。**
- jpc ・・・ **分岐命令の飛び先番地の計算を行う。**
- ALU ・・・ **算術演算を行う。**

・ IF ステージ：pc の内容をアドレスに出力し、外部の命令メモリをアクセスして命令フェッチを毎クロックずつ行い、呼び出してきた命令を命令レジスタ ir に格納する。プログラムカウンタは加算機で毎回 2 ずつインクリメントされ、各クロックで次々と命令がアクセスされる。また、分岐命令が実行されると制御信号である pcset がアクティブになり、飛び先の番地が Rf ステージから送られき、レジスタ pc に格納される。

・ RF ステージ：IF ステージで読み込まれた命令と pc を受け取り、レジスタフィールドの値によってレジスタファイルから読み出しを行う。第一オペランドは mux s1 を介して reg_a に格納し、第二オペランドは mux s2 を介して reg_b に格納する。イミディエイト命令は、符号拡張を行い、reg_im に格納する。また、5.2.3 で述べた加算機 jpc で飛び先番地を計算し、IF ステージに送る。レジスタファイルは第一引数と第二引数で示した番号のレジスタを読み出す。WB ステージから送られる制御信号がアクティブの時は、第三引数で示す番号のレジスタに第四引数のデータを書き込む。さらに 5.2.2 で述べたフォワーディング機能も実装している。

・ EX ステージ：RF ステージから送られてきた命令のオペコードに従って、それぞれの命令を行う。R 形式の場合の演算対象は reg_a と reg_b の内容であるが、I 形式では reg_a と reg_im の内容である。また、LD・ST 命令では、読み出されたレジスタによって、データメモリをアクセスする。ALU 演算結果とデータメモリからのデータは、reg_c に格納される。格納するかどうかを制御信号で制御し、その制御信号は WB ステージを経由して、次のクロックで RF ステージ中のレジスタファイルに格納するのに用いられる。また、5.2.2 で述べたフォワーディング機能も実装している。

・ WB ステージ：EX ステージから送られてきたデータをレジスタに格納するかどうかという制御信号をレジスタに格納して 1 クロック遅らせている。

5.3　HDL によるパイプライン PICO１６の設計

図 14 のデータパスにより設計した PICO にはモジュールが、ppico、if、add16(add)、mux2_1(s0)、rf、reg_file、add16(jpc)、mux2_1(s1)、mux2_1(s2)、ex、alu16、addsub16、mux2_1(s3)、mux2_1(s4)、sel5_1(s5)、wb の 16 個ある。以下に各モジュールの入出力と動作を説明する。

・モジュール ppico

最上位モジュールであり、主にモジュール if、モジュール rf、モジュール ex、モジュール wb の 4 つのモジュールの橋渡し役をしている。以下の表 11 に ppico の入出力信号を示す。

表 11：ppico の入出力信号

module 名	ppico	
入力ピン	idata[15:0]	Instruction Memory からのデータ入力
	ddatain[15:0]	Data Memory からのデータ入力
	in[15:0]	ｐｃインクリメント用 16 進数 0002
	nopecode[15:0]	NOP 命令　16 進数 0000
	CLK	クロック
	RST	リセット
出力ピン	iaddr[15:0]	Instruction Memory へのアドレス出力
	daddr[15:0]	Data Memory へのアドレス出力
	ddataout[15:0]	Data Memory へのデータ出力
	write	Data Memory に書き込むとき 1、メモリから読み出すとき 0

・モジュール if

モジュール ppico の下位モジュールであり、命令フェッチ、ｐｃのインクリメントを行う。以下の表 12 に if の入出力信号を示す。

表 12：if の入出力信号

module 名	if	
入力ピン	idata[15:0]	Instruction Memory からのデータ入力
	badr[15:0]	RF ステージからのｐｃ飛び先番地
	in[15:0]	ｐｃインクリメント用 16 進数 0002
	pcset	分岐命令かどうか判別する制御信号
	CLK	クロック
	RST	リセット
出力ピン	ifpc[15:0]	Instruction Memory へのアドレス出力
	ifjpc[15:0]	adder jpc へのアドレス出力
	ifir[15:0]	RF ステージへの命令データ出力

・モジュール add16(add)

モジュール if の下位モジュールであり、ｐｃのインクリメントを行う。以下の表 13 に add16(add)の入出力信号を示す。

表 13：add16(add)の入出力信号

module 名	add16(add)	
入力ピン	add_ina[15:0]	pcインクリメント用 16 進数 0002
	add_inb[15:0]	現在の命令アドレス
出力ピン	add_out[15:0]	演算結果を出力

・モジュール mux2_1(s0)

モジュール if の下位モジュールであり、分岐命令を制御する信号である pcset によって出力値が変わる。以下の表 14 に mux2_1(s0)の入出力信号を示す。

表 14：mux2_1(s0)の入出力信号

module 名	mux2_1(s0)	
入力ピン	a[15:0]	add16(add)から出力されたデータを入力
	b[15:0]	RF ステージから出力されたをpc飛び先番地を入力
	pcset	0なら a の値を出力し、1 なら b の値を出力
出力ピン	out	選択された値を出力

・モジュール rf

モジュール ppico の下位モジュールであり、レジスタファイルの読み出し・書き込み、分岐命令の判別を行う。以下の表 15 に rf の入出力信号を示す。

表 15：rf の入出力信号

module 名	rf	
入力ピン	ifjpc[15:0]	adder jpc へのアドレス入力
	ifir[15:0]	レジスタ reg_ir への命令データの入力 レジスタファイルへのアドレス入力 adder jpc へ符号拡張したイミディエイトデータを入力
	ex_c[15:0]	レジスタファイルへのデータ入力
	fdata[15:0]	sel5_1(s5)の出力データを mux2_1(s1,s2)に入力
	rwadr[2:0]	レジスタファイルへのアドレス入力
	rwe	レジスタファイルに格納するか判別する制御信号
	rwen	レジスタファイルに格納するか判別する制御信号
	CLK	クロック
	RST	リセット
出力ピン	rfir[15:0]	EX ステージへの命令データの出力
	rf_a[15:0]	ALU へのデータ出力、mux2_1(s3)へのデータ出力
	rf_b[15:0]	mux2_1(s3,s4)へのデータ出力
	imm[15:0]	mux2_1(s4)へのデータ出力
	badr[15:0]	mux2_1(s0)への pc とび先番地出力
	pcset	分岐命令かどうか判別する制御信号

・モジュール add16(jpc)

モジュール rf の下位モジュールであり、分岐命令の飛び先アドレスを計算するのに使用される。以下の表 16 に add16(jpc)の入出力信号を示す。

表 16：add16(jpc)の入出力信号

module 名	add16(jpc)	
入力ピン	add_ina[15:0]	レジスタ pc からのアドレス入力
	add_inb[15:0]	レジスタ ir からの符号拡張したイミディエイトデータを入力
出力ピン	add_out[15:0]	演算結果を出力

・モジュール reg_file

モジュール rf の下位モジュールであり、命令を実行するためのレジスタのアドレスを受け取り各種演算命令の結果を出力する。また、その結果を再び格納する。また、フォワーディング機能も装備している。以下の表 17 に reg_file の入出力信号を示す。

表 17：reg_file の入出力信号

module 名	reg_file	
入力ピン	addr1[2:0]	演算などに用いるレジスタファイルのアドレスである 命令が格納されているレジスタ ir の 10～8bit を入力
	addr2[2:0]	レジスタファイルのアドレスである命令が格納されて いるレジスタ ir の 7～5 bit を入力
	addr3[2:0]	レジスタ reg_rw からのアドレスを入力
	din[15:0]	演算結果などのデータをレジスタに入力
	req	レジスタに書き込むとき 1、レジスタから読み出すとき 0
	CLK	クロック
	RST	リセット
出力ピン	dout1[15:0]	addr1 の示すアドレスにあるデータを出力
	dout2[15:0]	addr2 の示すアドレスにあるデータを出力

・モジュール mux2_1(s1)

モジュール rf の下位モジュールであり、フォワーディングを実現するために、EX ステージの結果レジスタの直前から RF ステージのレジスタ読み出し結果のレジスタへ横流しを行う役割を持つ。以下の表 18 に mux2_1(s1)の入出力信号を示す。

表 18：mux2_1(s1)の入出力信号

module 名	mux2_1(s1)	
入力ピン	a[15:0]	sel5_1(s5)の出力データを入力
	b[15:0]	レジスタファイルの dout1 のデータを入力
	sel1	0 なら a の値を出力し、1 なら b の値を出力
出力ピン	out	選択された値を出力

・モジュール mux2_1(s2)

モジュール rf の下位モジュールであり、フォワーディングを実現するために、EX ステージの結果レジスタの直前から RF ステージのレジスタ読み出し結果のレジスタへ横流しを行う役割を持つ。以下の表 19 に mux2_1(s2)の入出力信号を示す。

表 19：mux2_1(s2)の入出力信号

module 名	mux2_1(s2)	
入力ピン	a[15:0]	レジスタファイルの dout2 のデータを入力
	b[15:0]	sel5_1(s5)の出力データを入力

	sel2	0 なら a の値を出力し、1 なら b の値を出力
出力ピン	out	選択された値を出力

・モジュール ex

モジュール ppico の下位モジュールであり、それぞれの命令の演算実行、メモリへの書き込み、読み込みを行う。演算結果とメモリからのデータをレジスタファイルに格納するかは制御信号 rwen で制御する。以下の表 20 に ex の入出力信号を示す。

表 20：ex の入出力信号

module 名	ex	
入力ピン	rfir[15:0]	レジスタ reg_rw への命令データ入力 sel5_1(s5)への符号拡張したイミディエイトデータを入力
	rf_a[15:0]	ALU へのデータ入力
	rf_b[15:0]	mux2_1(s4)へのデータ入力
	imm[15:0]	mux2_1(s4)へのデータ入力
	ddatain[15:0]	Data Memory から sel5_1(s5)へのデータ入力
	nopcode[15:0]	NOP 命令（16 進数 0000）を sel5_1(s5)へ入力
	CLK	クロック
	RST	リセット
出力ピン	ex_c[15:0]	レジスタファイルへデータを出力
	address[15:0]	Data Memory へアドレスを出力
	ddataout[15:0]	Data Memory へデータを出力
	fdata[15:0]	mux2_1(s1,s2)へデータを出力
	rwadr[2:0]	レジスタファイルへ rfir[10:8]を出力
	rwen	レジスタファイルに格納するか判別する制御信号
	write	Data Memory に書き込むとき 1、メモリから読み出すとき 0

・モジュール alu16

モジュール ex の下位モジュールであり、mux2_1(s4)とレジスタ reg_a のそれぞれから出力データを受け取り、命令レジスタ rfir から受け取った opcode によって決められた演算を実行する。以下の表 21 に alu16 の入出力信号を示す。

表 21：alu16 の入出力信号

module 名	alu16	
入力ピン	alu_ina[15:0]	レジスタ reg_a の出力から入力
	alu_inb[15:0]	mux2_1(s4)の出力から入力

	com[3:0]	第 2 オペコードの[3:0]によって演算を決定
出力ピン	alu_out[15:0]	演算結果を出力

・モジュール addsub16

モジュール alu16 の下位モジュールであり、算術演算である加算と減算を行い、結果を alu16 に渡す。以下の表 22 に addsub16 の入出力信号を示す。

表 22：addsub16 の入出力信号

module 名	addsub16	
入力ピン	addsub_ina[15:0]	alu 演算が加算と減算の場合に入力
	addsub_inb[15:0]	alu 演算が加算と減算の場合に入力
	sub	1なら減算、0なら加算
出力ピン	addsub_out[15:0]	演算結果を出力

・モジュール mux2_1(s3)

モジュール ex の下位モジュールであり、Data Memory へのアドレスを制御している。ST 命令の場合は０で、それ以外の場合は１である。以下の表 23 に mux2_1(s3)の入出力信号を示す。

表 23：mux2_1(s3)の入出力信号

module 名	mux2_1(s3)	
入力ピン	a[15:0]	レジスタ reg_a の出力から入力
	b[15:0]	レジスタ reg_b の出力から入力
	sel3	0なら a の値を出力し、1 なら b の値を出力
出力ピン	out	選択された値を出力

・モジュール mux2_1(s4)

モジュール ex の下位モジュールであり、命令が R 形式の場合は０、I 形式の場合は１である。以下の表 24 に mux2_1(s4)の入出力信号を示す。

表 24：mux2_1(s4)の入出力信号

module 名	mux2_1(s4)	
入力ピン	a[15:0]	レジスタ reg_b の出力から入力
	b[15:0]	レジスタ imm の出力から入力

	sel4	0 なら a の値を出力し、1 なら b の値を出力
出力ピン	out	選択された値を出力

・モジュール sel5_1(s5)

モジュール ex の下位モジュールであり、LDLI 命令の場合 000、LDHI 命令の場合 001、LD・ST 命令以外の R 形式と LDLI・LDHI 命令以外の I 形式の場合 010、LD 命令の場合 011、それ以外の場合 100 である。以下の表 25 に sel5_1(s5)の入出力信号を示す。

表 25：sel5_1(s5)の入出力信号

module 名	sel5_1(s5)	
入力ピン	a[15:0]	ir[7]を上位 8 ビットに拡張、下位 8 ビットに ir[7:0]を入力
	b[15:0]	上位 8 ビットは 0、下位 8 ビットに ir[7:0]入力
	c[15:0]	alu の演算結果を入力
	d[15:0]	Data Memory からのデータを入力
	e[15:0]	NOP 命令 (16 進数 0000) を入力
	sel5[2:0]	000 なら a の値を出力、001 なら b の値を出力、010 なら c の値を出力、011 なら d の値を出力、100 なら e の値を出力
出力ピン	out	選択された値を出力

・モジュール wb

EX ステージから送られてきたデータをレジスタファイルに格納するかどうかという制御信号をレジスタに格納して 1 クロック遅らせている。以下の表 26 に wb の入出力信号を示す。

表 26：wb の入出力信号

module 名	wb	
入力ピン	rwen	レジスタファイルに格納するか判別する制御信号
	CLK	クロック
	RST	リセット
出力ピン	rwe	レジスタファイルに格納するか判別する制御信号

5.4　シミュレーションによる動作検証

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 27 に示す。

表 27：パイプライン方式 PICO１６のシミュレーション結果

	命令数	実行時間(ns)	クロック数	CPI	ストール
8 個の数の最大値	57	15100	75	1.32	16
2 数の最大公約数	27	7500	38	1.41	8
バブルソート	248	57100	284	1.15	64

(実行時間：ns、クロック・ストール数：回)

各テストパタンを比較すると、命令数に対するストールの回数が少ない「バブルソート」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、命令数に対するストールの数が多い「2 数の最大公約数」のテストパタンでは他の2 つのテストパタンと比べ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６章の「プロセッサの性能比較と考察」で行う。

6　生成されたプロセッサの評価

6.1　マルチサイクル方式とパイプライン方式の比較と考察

本研究で作成したマルチサイクル方式 PIC016 とパイプライン方式 PIC016 のハードウェア規模、性能比較を行った。性能を比較するためのゲートレベルシミュレーションで

は、マルチサイクル、パイプラインと同様の「8 つの数の最大値」、ユーグリッド互助法による 2 数の最大公約数」、及び「バブルソート」を求めるプログラムを作成した。

以下の表 28 に各プロセッサのハードウェア規模を示す。

表 28：各プロセッサのハードウェア規模

	FF 数	LUT 数	HDL 記述
マルチサイクル	204	543	535
パイプライン	213	627	655

(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。パイプライン方式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 共にマルチサイクルを上回った。

次に、以下の表29～表31に3 つのテストパタンでの各マイクロプロセッサの実行結果を示す。

表 29：8 個の数の最大値

	命令数	実行時間(ns)	クロック数	CPI	ストール
マルチサイクル	57	36100	180	3.16	—
パイプライン	57	15100	75	1.32	16

(実行時間：ns、クロック数・ストール数：回)

表 30：ユーグリッド互除法による 2 数の最大公約数

	命令数	実行時間(ns)	クロック数	CPI	ストール
マルチサイクル	27	17500	84	3.11	—
パイプライン	27	7500	38	1.41	8

(実行時間：ns、クロック数・ストール数：回)

表 31：バブルソート

	命令数	実行時間(ns)	クロック数	CPI	ストール
マルチサイクル	248	161900	809	3.07	—

パイプライン	248	57100	284	1.15	64

(実行時間：ns、クロック数・ストール数：回)

以下の表 32 に、3 つのテストパタンの命令数と、各テストパタンを実行した場合のパイプラインのストール回数、及びマルチサイクル方式に対するパイプライン方式の速度向上比を示す。

表 32：マルチサイクル方式に対するパイプライン方式の速度向上比

	命令数	ストール	速度向上比
8 個の数の最大値	57	16	2.40
2 数の最大公約数	27	8	2.21
バブルソート	248	64	2.85

(ストール数：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４ 段でパイプライン化を行った。しかし、理想的には速度向上比は実際には得られなかった。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。さらに、パイプラインに大きな影響を及ぼすストールが発生するからである。5 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。命令数に対してストールの回数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、「バブルソート」のテストパタンでは命令数が多く、命令数に対するストールの数が少ないことで、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」のテストパタンでは命令数 27 と少ない割に、ストール回数が８ と多いので、性能は 2.21 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の発生回数をどのように減らすかが重要になってくると思われる。

7　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方式PIC016、及びパイプライン方式PIC016 の設計について述べた。3 つのテストパタンを作成し、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 つのテストパタンにより、マルチサイクル方式、パイプライン方式の各マイクロプロセッサのハードウェア規模や命令実行時間、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、パイプライン方式が制御など複雑なためデータパスも複雑化し、ハードウェア規模はマルチサイクル方式と比べて大きかった。また、各マイクロプロセッサの性能比較ではマルチサイクル、パイプラインの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、2.85 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、本研究で作成した4段パイプラインを一般的なRISCが取っている5段パイプライン化を行い、本研究では発生しなかったハザードなどについての対処法を考える。また、マルチサイクル方式、パイプライン方式ともにメモリの論理合成を行っていなかったため、詳細な実行結果が得られなかったので、メモリの論理合成を行い、FPGAにダウンロードした時に起こる、メモリなどの各種遅延についても調べたい。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。

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、小柳滋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、池田修久氏、大八木睦氏、同じハードウェアグループの中村央志氏、古川達久氏、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]池田誠、小林和淑著、浅田邦博、越智裕之編、VDEC 監修：ディジタル集積回路の設計と試作、培風館、2000.

[2]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.
[3]天野英晴、西村克信著、小栗清監修：作りながら学ぶコンピュータアーキテクチャ、培風館、2001.
[4]木村真也著：自習 Verilog-HDL 論理回路設計、CQ 出版、2001.
[5]深山正幸、北川章夫、秋田純一、鈴木正國著：HDL による VLSI 設計、共立出版、2002.
[6]小林優：入門 VerilogHDL 設計入門、CQ 出版、2001.
[7]池田修久：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、立命館大学理工学部卒業論文、2002.
[8]大八木睦：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、立命館大学理工学部卒業論文、2002.
[9]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情報学科卒業論文、1998.
[10] 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．
[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] 中村央志：ハードウェア記述言語による教育用マイクロプロセッサの設計(Ⅰ)，立命館大学理工学部卒業論文，2003
[13] 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