卒 業 論 文

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

氏　　名： 古川　達久
学籍番号： 2210990196-6
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

内容梗概

本論文ではハードウェア記述言語 VerilogHDL を用いて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の設計を行った。VerilogHDL と CAD ツールの習得、マイクロプロセッサ基本的なアーキテクチャを理解することが目的である。マイクロプロセッサの命令セットには、慶應義塾大と東京工科大で共同開発された実験教育用の 16bit RISC プロセッサ PICO16 の命令セットを用いた。

マルチサイクルマイクロプロセッサは、性能や実装のことよりも、後々プログラムを見た際にその動作フローを見やすく理解しやすくすることに重点を置いたプログラミングを行った。マイクロプロセッサに不可欠なメモリの記述に関しては、テストベンチ内に記述することで、メモリ自体を論理合成させない工夫をしてある。また、シミュレーションのためのテストパタンを作成し、ゲートレベルシミュレーションまでの動作検証を行った。CAD ツールから得られた遅延情報から、ALU に桁上げ先見方式の加算器を導入することで性能向上が得られると考えられる。

パイプラインマイクロプロセッサは現在設計途中であるが、こちらはマルチサイクルマイクロプロセッサの設計で得た経験を基にして、今度は性能や実装の事も考えた設計を行っている。

目次

図目次

表目次

１　はじめに

1970 年代、集積回路が初めて一般の人々に普及した時代に集積回路を使った製品は電卓、時計および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、これ以降の半導体の設計規模が急激に増大し始めた。1995 年代での設計規模は 100k～500k ゲート程度の規模であったが、現在ではその 10 倍を超える設計規模の拡大となっている。これは、それ以前までは単体の機能だけを考慮して設計していたものが、現在ではシステム自身を１Chip 化する設計(システムオンチップ)に変わってきており、設計規模が非常に大きくなってきているためである。

LSI 設計におけるシミュレーション技術も LSI 技術と同様に発展してきた。1970 年代にはレイアウト･マスク設計に関する自動化が始まり、1980 年代にはゲートレベルでのシミュレーションが自動化され、実際の基板上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となった。しかし、設計規模の拡大により、従来から行われている仕様から論理ゲートやフリップフロップなどの記憶素子を接続する設計手法では、ゲートレベルシミュレータを用いたとしても、それを短時間で仕様通り正確に設計することは困難になってきた。そして近年、コスト削減につながる設計時間の短縮の必要性と、EDA ツールの出現により従来の論理ゲートから行うボトムアップ設計方式から、HDL(Hardware Description Language)記述によるトップダウン設計方式へと変移している。HDL 記述によるトップダウン設計は、回路設計を上位レベルの機能検証からアプローチすることを可能とし、機能に関する致命的なバグを早期に発見することが可能となり、開発期間の短縮とコストの削減に繋がる。また、FPGA(Field Programmable Gate Array)の登場により、設計した回路を何度でもその場ですぐに LSI として利用できるようになった。これによって、開発段階でのエミュレーションや、プロトタイプの作成として利用され、さらなる開発期間の短縮と繋がっている。

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 VerilogHDL によるマルチサイクル･パイプライン方式マイクロプロセッサの設計を行う。実際にハードウェア記述言語を用いて設計を行うことで、トップダウン設計の流れの理解、ハードウェア記述言語の習得、各プロセッサアーキテクチャの理解、CAD ツールの使用法の習得を目的とする。

研究で用いた PICO16 は慶應義塾大と東京工科大で共同開発された実験教育用の 16bit RISC プロセッサであり、ハードウェア記述言語 SFL で記述されている。今回はこの PICO16 の命令セットを基に VerilogHDL で RTL 記述を行い、CAD ツール Xilinx 社 Foundation ISE とシミュレーションツール Model Technology 社 ModelSim 用いて、ビヘイビアレベルシミュレーション、論理合成、ゲートレベルシミュレーションを実行し検証する。両シミュレーションで用いたテストパタンは、加算、乗算、最大値、三乗、平均値を求めるプログラムである。

第２章でハードウェア記述言語による設計の概要を述べ、第３章で PICO16 の命令セットアーキテクチャについて述べる。第４章ではマルチサイクルマイクロプロセッサの設計と検証、第５章でパイプラインマイクロプロセッサの設計について述べる。

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

ディジタル回路の設計は、長い間、論理ゲートや Flip Flop などの記憶素子の接続を示すスケマティックを用いた設計が行われてきた。現在もこの手法による設計は行われているが、最近は計算機のプログラミングをするように言語で記述し、その記述から回路を自動生成する方法が一般的になっている。この言語のことをハードウェア記述言語（HDL:Hardware Description Language）と呼ぶ。

HDL による設計は次のような特徴をもつ。

- 上位レベルで記述するため、バグを発見し修正しやすく、他人が見てもわかりやすい
- 機能レベルでのシミュレーションで動作を検証できるため、早期にバグを発見することができる
- 時間の概念を持ち、遅延などの記述方法をもつ
- 論理の簡単化等、細かいレベルの設計は自動的に行われるために、時間の短縮とバグの発生率も減る。

HDL の設計記述レベルには大きく、動作レベル、RTL、ゲートレベルの 3 つがある。以下にそれぞれの特徴を示す。

- 動作レベル
  仕様要求から動作をイメージし、言語の構文要素に設計アルゴリズムを適用する RTL の上位モデル。論理合成は不可。
- RTL
  レジスタを明確に定義し、レジスタ間のデータの流れや、そのデータの設計による処理方法を記述する論理合成可能なモデル。
- ゲートレベル
  RTL から導かれた論理ゲート間の接続を保持した回路表現の最下位モデル

HDL には VHDL、VerilogHDL、SFL 等が代表的である。各々の言語の特徴を以下に示す。

(1)　VHDL

VHDL (VHSIC Hardware Description Language) はアメリカ合衆国国防総省のプロジェクトが中心となって開発が行われた言語である。主にハードウェアドキュメンテーション用言語に合成機能が付いたもので、多種多様な構文を持つ一方、論理合成できる範囲が限られている。また、単純な回路に対しても記述が複雑になる傾向がある。現在、IEEE によって IEEE 1076-1987 および IEEE 1164-1993 規格として正式に承認された。

(2)　VerilogHDL

VerilogHDL は論理シミュレーション用言語として開発され、その後発達したハードウェア記述言語である。抽象度の異なる記述レベルを混在させて回路を表現することができ、設計、テスト、検証のあらゆる面において同一の言語で表現が可能である。1993 年に OVI（Open Verilog International)、1995 年に IEEE-1364 に認定され標準化されている。

(3)　SFL

SFL（Structured Function description Language）は NTT によって開発された国産のハードウェア記述言語である。効率的な設計手法を提案することをねらいとして設計され、複雑な制御回路を簡単に記述できる点が優れている。記述環境 SFL と設計環境 PARTHENON は PARTHENON ホームページから無料で自由に入手でき、手軽にその開発環境を手に入れる事ができる。日本の大学を中心に教育用に広く用いられている。

次に半加算器の VHDL、VerilogHDL、SFL での各々の記述例を示す。

```
entity halfadder is
  port(
    a, b : std_logic;          } 入出力ポート宣言部
    s, c: std_logic);
end halfadder;

architecture halfadder_body of half_adder is
begin
  c <= a and b;                } 機能記述部
  s <= a xor b;
end halfadder_body;
```

図 1：半加算器の VHDL による記述例

```
module halfadder (s, c, a, b);
  input a, b;                    } 入出力ポート宣言部
  output s, c;

  assign c = a & b;              }
  assign s = (a & ~b)|(~a & b);  } 機能記述部
endmodule
```

図 2：半加算器の VerilogHDL による記述例

```
module halfadder {
  input a, b;                    } 入出力ポート宣言部
  output s, c;
  instrin enable;          ···· 制御入力端子宣言

  instruct enable par{
    c = a & b;                   }
    s = (a & ^b)|(^a & b);       } 機能記述部
  }
}
```

図 3：半加算器の SFL による記述例

２．２　システム設計

ハードウェアの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。ボトムアップ設計は、以前から長い間、用いられている設計手法である。仕様から論理ゲートやフリップフロップなどの記憶素子を接続する様子を示す回路図（スケマティック）を作成し、ゲートレベルシミュレータを用いて設計行う。トップダウン設計は現在行われつつある設計手法である。ディジタル回路を計算機のプログラミングをするように言語で記述し、その記述から論理合成ツールによって回路を自動生成する方法である。

以下にボトムアップ設計とトップダウン設計の各々の特徴を示す。

■ ボトムアップ設計

- テクノロジ、プロセスを設計の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、全体の検証が始められない
- デバッグが困難
- 設計が進むにつれ、修正が困難
- 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

- テクノロジは設計の最終段階まで決める必要はないため、最新の物を使用しやすい
- 設計初期にシミュレーションができるため、致命的な欠陥を早期に発見できる
- 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が節約でき、設計者は仕様の検討に専念できる
- テストパタンを設計初期段階で作成でき、利用可能
- 設計データを簡単に再利用可能

図 4 にトップダウン設計の HDL 設計フローを示す。

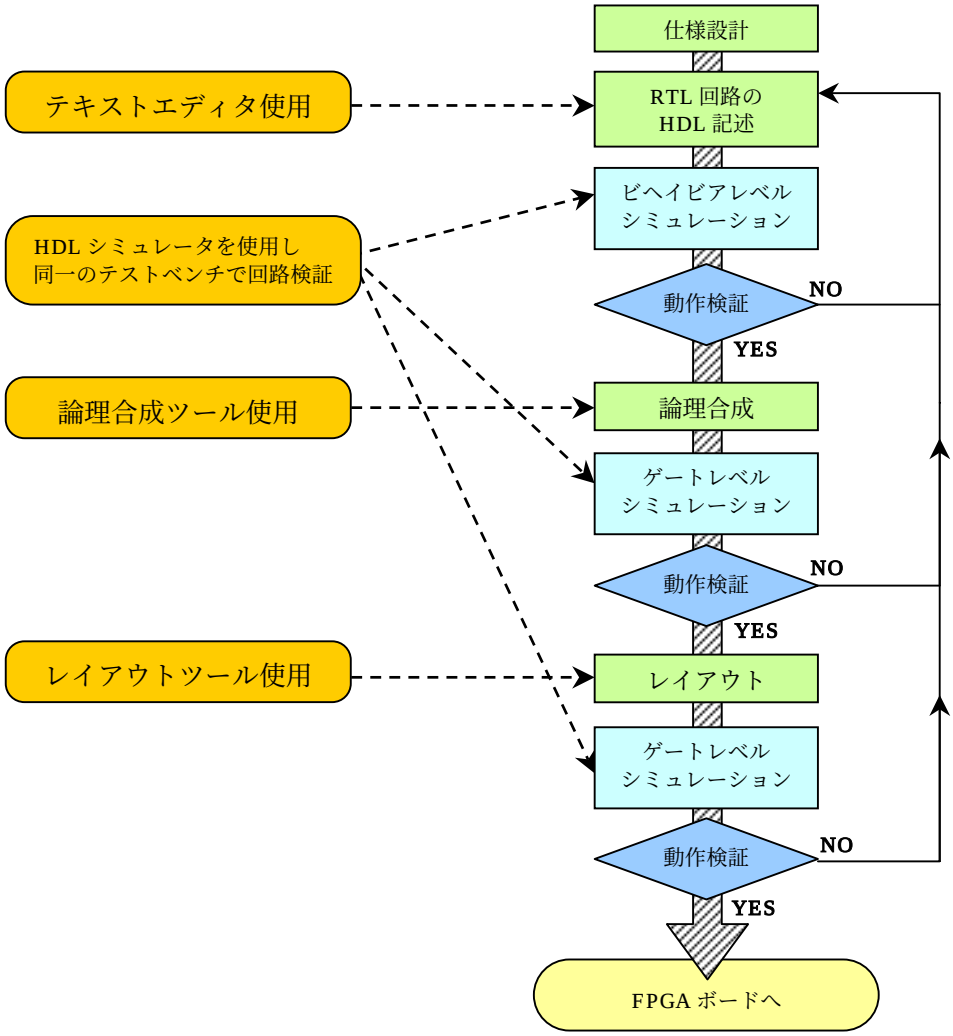

図 4：トップダウン HDL 設計フロー

図 4 にあるようにトップダウン設計は、まず、仕様設計から汎用のテキストエディタを使用し、仕様に基づいて実際に動作レベルの HDL 記述を行う。そして、HDL シミュレータを用いて動作が仕様を満たしているのかを確認する。ここで確認するのは動作のみの検証であり、遅延などの実際のタイミングは考慮しない。シミュレーション結果が正しければ、次に CAD ツール、論理合成ツールを用いて論理合成を行い、ネットリストを出力する。そして、出力されたネットリストからゲートレベルシミュレーションを行い、実装を想定した遅延などのタイミング検証を行う。また、ゲートレベルシミュレーションで用いるテストベンチは動作レベルシミュレーションで用いたものと同一のテストベンチを用いる必要がある。

２．３　シミュレーション

２．３．１　テストベンチの記述

HDL 設計では RTL 回路を HDL で記述するだけでなく、RTL 回路が仕様要求通りの動作をするかどうかを HDL シミュレータで検証する必要がある。このときテスト対象となる回路以外に入力信号系列を与え、出力を観測するテストベンチが必要となる[8]。

今回研究で用いた VerilogHDL のテストベンチ記述例として、図 2 の半加算器に対するテストベンチ記述例を 図 5 に示す。

```
1:  module test_halfadder;
2:  reg a, b;        ···· テスト対象ポートに接続するレジスタ宣言
3:  wire s, c;       ···· テスト対象ポートに接続するワイヤ宣言
4:
5:  halfadder ha (.s(s), .c(c), .a(a), .b(b));
6:                                ···· テスト対象回路のインスタンス
7:  initial
8:     begin
9:        a = 0; b = 0;
10:       #10  a = 1;      } テストパタン記述
11:       #10  b = 1;
12:    end
13: endmodule
```

図 5：VerilogHDL によるテストベンチ記述例

図 5 においては、2,3行目でテスト対象に接続するポートのレジスタ宣言とワイヤ宣言を行い、5行目でテスト対象モジュールをインスタンスしている。そして、7行目からの initial 構文にて、値の挿入を行っている。また「#」は遅延を表し、「＃10」ならば時間 10 の遅延を行うことを示している。

２．３．２　シミュレーションの各レベル

トップダウン設計におけるシミュレーションには 図 4 にあるように、論理合成前に行われる動作レベルシミュレーション､論理合成後に行われるゲートレベルシミュレーションと大きく２つに分けられる。

動作レベルシミュレーションはあくまでも動作(機能)のみのシミュレーションである。設計した HDL 記述の動作が与えられた仕様を満たすものかの検証を行う。高速にシミュレートできるため、致命的な欠陥を早期に発見、デバッグが可能である。

ゲートレベルシミュレーションは配線状況や回路資源(CLB)に関わる遅延などを考慮した検証を行う。また、FPGA 上に実際に配置配線を行った後のシミュレーションが可能であり、FPGA の総資源に対してどの程度の資源を必要とするか確認が可能である。

２．４　FPGA ボード

FPGA（Field Programmable Gate Array）はプログラム書き換え可能な LSI である。論理ゲートをアレイ状に敷き詰めた集積回路で､回路データをダウンロードすることで回路構成を変更することが可能である。FPGA 上に構成された回路は電源を断つか、新たに回路構成データをダウンロードすることで、何度も書き換えることが可能である。従来、設計した回路は実際に LSI に焼き付けるか、シミュレーションによって動作検証を行っていた。しかし、LSI に焼き付けるには時間とコストがかかり、大規模な回路のシミュレーションには膨大な時間が必要であった。FPGA を用いることで設計した LSI をその場で実現し、実機での動作検証を行うことができ、時間とコストの削減が可能となる。

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

本研究において用いた PICO シリーズは、慶應義塾大と東京工科大で共同開発された実験教育用の CPU のシリーズで FPGA のサイズに応じて様々なサイズの命令セットを構築することができる[2]。今回の設計するマイクロプロセッサの命令セットは 16bit 幅の RISC 型 PICO16 を用いる。

PICO16 の特徴として単一命令長であることがあげられる。これには制御や後のパイプライン化が容易になるという利点がある。また、計算を行うには必ずメモリからレジスタにデータを Load してからレジスタ間で計算を行う Load/Store マシンである。

図 6 にこの PICO16 の命令フォーマットを示す。

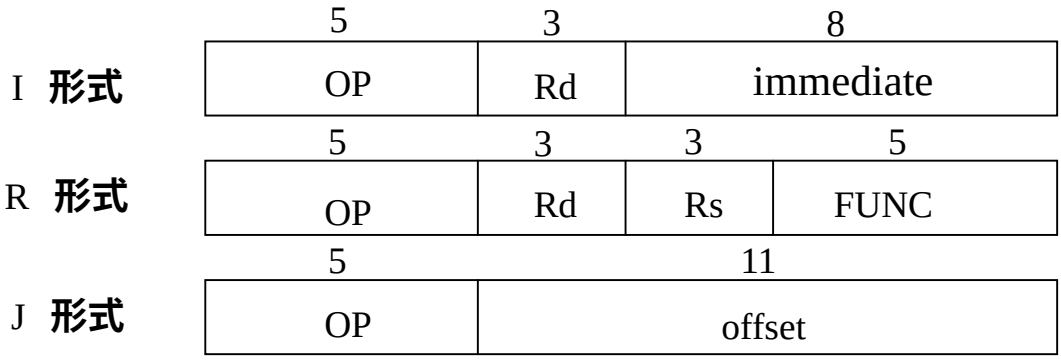

図 6：PICO16 の命令フォーマット

■　命令フィールド

- OP　　：オペレーションコード
- FUNC　：ファンクションコード
- Rd　　：デスティネーションレジスタ
- Rs　　：ソースコードレジスタ
- immediate：データ格納する即値
- offset　：アドレスを示すオフセット値

すべての命令は 16bit 固定長であり、上位 5bit のオペレーションコードと、下位 5bit のファンクションコードで命令の判別を行う。命令形式には、即値命令や条件分岐命令を行うＩ形式、算術論理演算を行うＲ形式、ジャンプ命令を行うＪ形式の三種類がある。

３．２　レジスタ間演算命令

PICO はレジスタを８つ持つ汎用レジスタマシンであり、ADD,SUB 命令などの演算処理を行う場合、演算対象であるレジスタをオペラントで直接指定しての演算が可能である。このような指定方式をレジスタ直接アドレッシングという。

図 7 に ADD 命令によるレジスタ間演算命令の様子を示す。

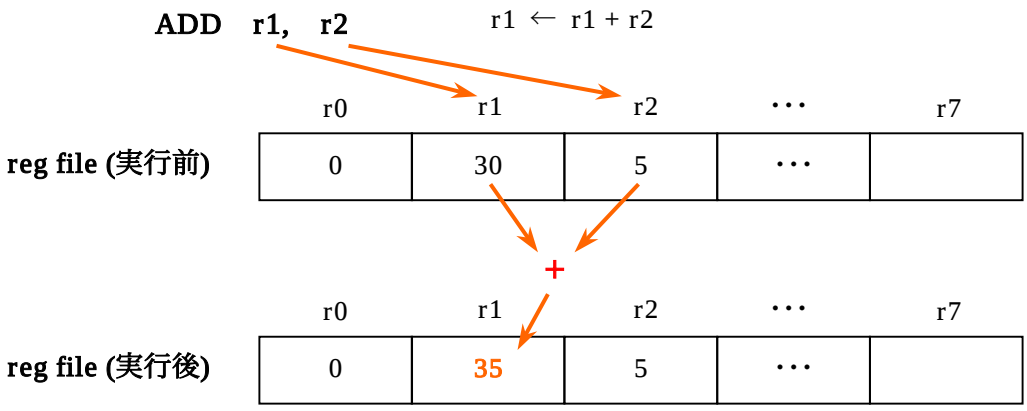

図 7：ADD 命令によるレジスタ直接アドレッシング

図 7 のような ADD 命令の場合、レジスタファイルのレジスタ番号 r1 からデータ(30)を、レジスタ番号 r2 からはデータ(5)を読み出し、両方を加算した後に、第一オペラントである r1 に結果(35)が格納される。

３．３　メモリアクセス命令

PICO は Load/Store マシンであり、メモリとレジスタ間でデータを移動する命令 LD、ST 命令はレジスタ間接アドレッシングを用いる。

図 8 に LD 命令のレジスタ間接アドレッシングの様子を示す。

図 8 のような LD 命令の場合、レジスタファイルのレジスタ番号 r2 に格納されているデータ(102 番地)を参照し、そのデータ(102 番地)が示すメモリの番地のデータ(20)が、レジスタ番号 r1 に格納される。

このようにレジスタ間接アドレッシングは、メモリにアクセスするために、まず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。この方式は、実際は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で、配列やスタックを実現したりポインタを作ったりすることができるため便利な方式で、ほとんど全てのプロセッサがこの方式を利用している。

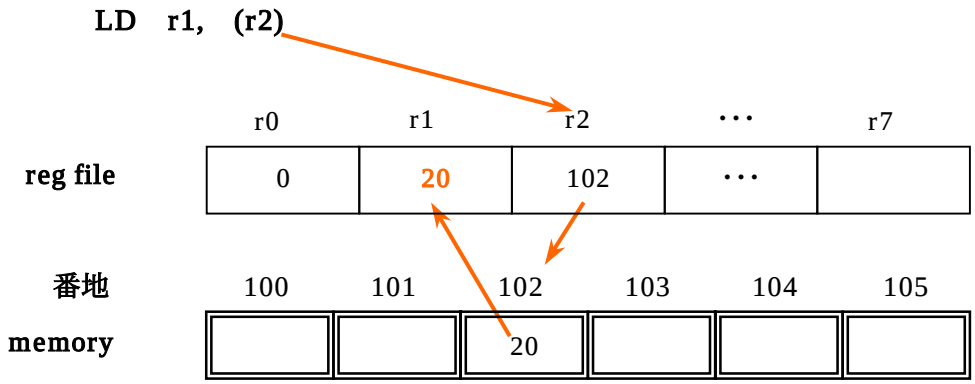

図 8：LD 命令によるレジスタ間接アドレッシング

３．４　即値命令

レジスタに即値を指定する即値命令の場合、即値アドレッシングを用いる。RISC 型である PICO16 は命令長がデータ長と同じでかつ固定長であることから、即値命令を全部の命令の中に収めることは出来ない。そこで、即値命令 ADDI,SUBI 命令などを処理する場合は、命令中の 8bit データの即値を符号拡張して、16bit 化してから演算を行う。

図 9 に ADDI 命令即値アドレッシングの様子を示す。

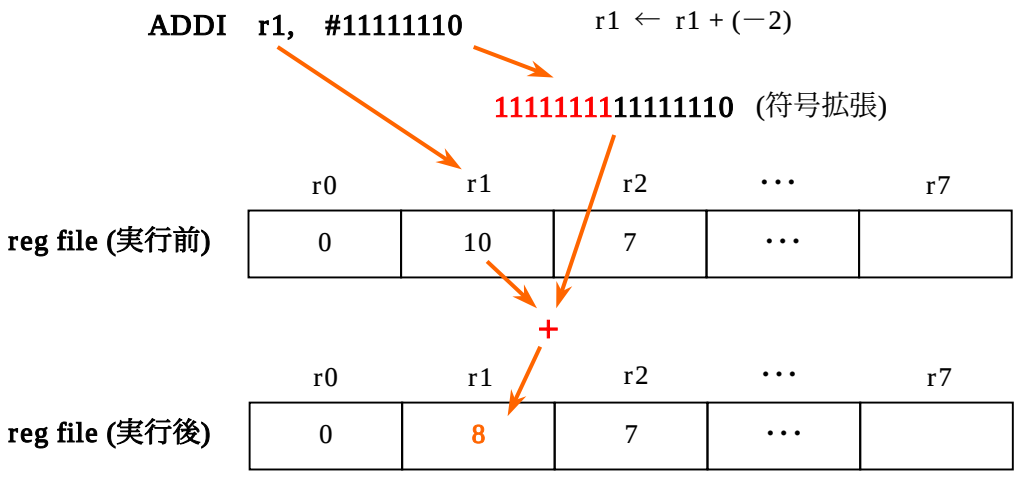

図 9：ADDI 命令の即値アドレッシング

図 9 のように、第二オペラント(11111110)が 16bit に符号拡張された後、r1 に格納されているデータ(10)と演算処理が行われ、第一オペラントである r1 に結果(8)が格納される。

４　マルチサイクル方式によるＰＩＣＯ１６の設計と検証

４．１　マルチサイクルデータパス

図 10 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。

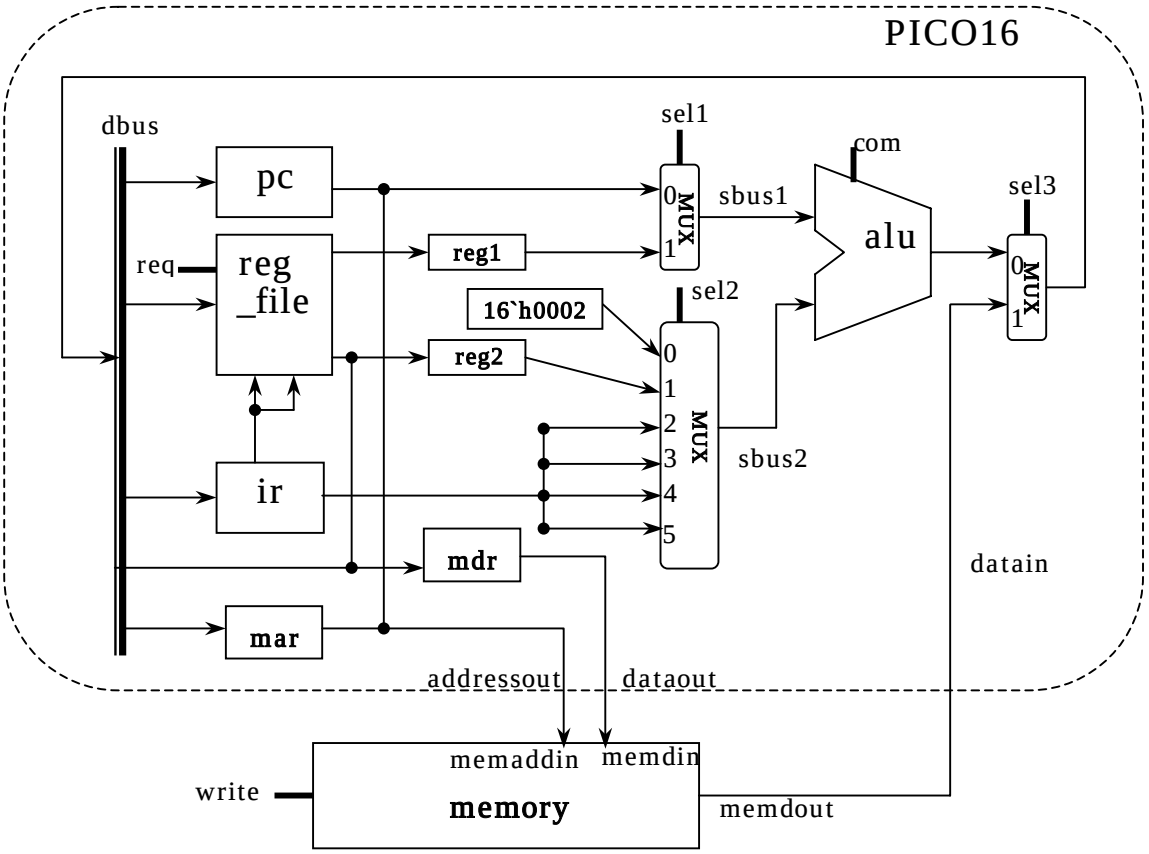

図 10：マルチサイクルマイクロプロセッサのデータパス

マルチサイクルマイクロプロセッサのデータパスは以下の要素から構成される。

- pc：プログラムカウンタ。次に実行する命令の番地を保持するレジスタ。
- ir：命令レジスタ。メモリからフェッチしてきた命令を格納するレジスタ。
- mar：メモリアドレスレジスタ。データを読み書きするためにメモリをアクセスする際、アドレスを格納するレジスタ。
- mdr：メモリデータレジスタ。データをメモリに書き込む際にデータを格納するレジスタ。
- alu：算術演算ユニット。
- reg_file：演算結果などの途中データを書き込む８つのレジスタ。Dual Port メモリ型。
- mux：マルチプレクサ。

図 10 のデータパスでは、あるステップにおいて全てのレジスタ内のデータは sbus1 と sbus2 を通り、alu に送られる。そして、alu は指定された算術演算を行い、dbus を通って全てのレジスタに再び格納され次のステップに移る。このように、各ステップにおけるデータは alu を通ることになるが、計算を行う必要のない場合でも、alu をスルーしてデータを通す必要がある。

４．２　マルチサイクル命令実行と制御

マルチサイクルマイクロプロセッサは 1 命令を複数のステップに分け、各ステップを 1 クロックサイクルで実現し、命令フェッチ(IF)、レジスタフェッチ(RF)、命令実行(EX,EX2)というステップを繰り返しながら処理を行う。

図 11 にマルチサイクル方式の状態遷移の様子を示す。

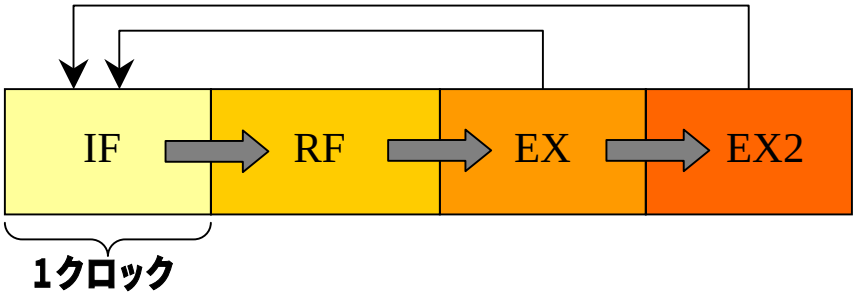

図 11：マルチサイクル方式の状態遷移

1. IF ：命令フェッチ。pc で指定されるアドレスの命令を ir にセットする。
2. RF：レジスタフェッチ。ir のレジスタフィールドで設定される番号のレジスタをレジスタファイルから読み出し、reg1,reg2 に格納する。また、alu を用いて pc をインクリメントする。
3. EX：実行その１。reg1,reg2 のデータを alu で演算し、dbus を通して reg_file に書き込む。即値命令は、reg2 の代わりに ir の下位に入っているデータを用いる。LD 命令では、読み出したレジスタの値を mar に格納する。
4. EX2：実行その２。メモリアクセス命令の場合、メモリアクセスを行う。

IF と RF では全命令に共通のステップである。

まず IF ステージでは pc に格納されているアドレスを参照し、メモリからそのアドレスにあるデータ(命令)を読み出す。そして、読み出した命令を dbus 経由で命令レジスタ ir にセットする。

次に RF ステージでは ir にある命令中のオペラントフィールド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データを読み出し、それぞれ reg1,reg2 に格納する。こ

の間に alu を用いて pc をインクリメントし、次のサイクルの命令フェッチに備える。また、ST 命令の場合は読み出したレジスタの値を mdr に格納しておく。

EX ステージでは命令レジスタ ir のオペコードフィールドそれぞれの命令を判断し、実行する。ここで、LD、ST 命令の場合はそれぞれアクセスするアドレスを保持するレジスタの中身を mar に格納する。

EX2 ステージでは LD 命令と ST 命令のメモリアクセスを行う。LD 命令は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 命令ではメモリの書き込み制御入力端子 write をアクティブにし、データ書き込みを行う。

４．３　ＨＤＬによるマルチサイクルマイクロプロセッサの設計

４．３．１　モジュール構成

汎用テキストエディタを用いて VerilogHDL 記述によるマルチサイクルマイクロプロセッサの設計を行った。マルチサイクルマイクロプロセッサの回路を記述する基本構成である構成モジュールは ALU、加減算器、全加算器、マルチプレクサ、レジスタファイルからなっており、図 12 に示すようなモジュールの構成になっている。

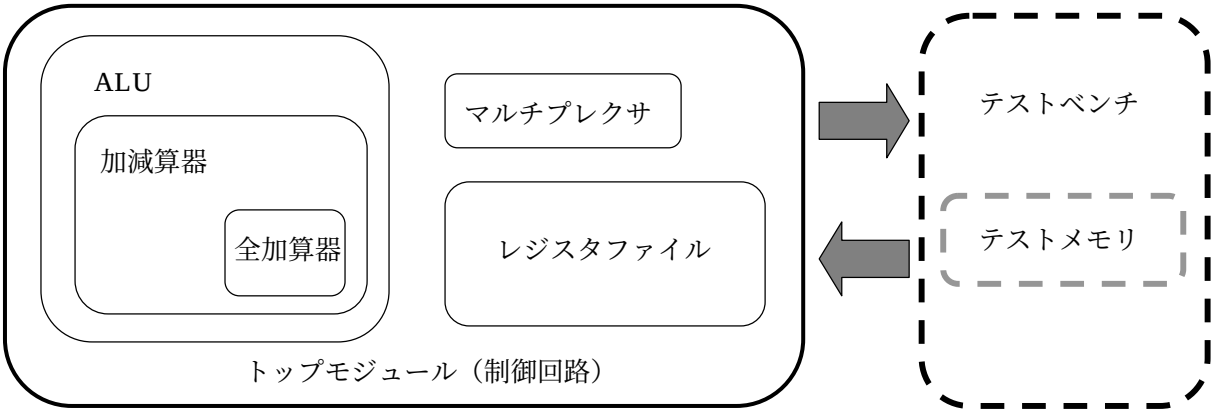

図 12：マルチサイクルマイクロプロセッサのモジュール構成

図 12 にあるように、全加算器は加減算器の下位モジュールであり、さらに加減算器は ALU の下位モジュールとなっている。そして ALU、マルチプレクサ、レジスタファイルはトップモジュールによって結線され、全体の制御もトップモジュールで行うように設計を行っている。

４．３．２　加減算器の設計

加算器と減算器は非常に共通点の多いハードウェアであるため、両方の機をまとめて加算と減算を切り替えて使うことができる加減算器(addsub16)を設計する。

図 13 に加減算器の HDL 記述の一部を示す。

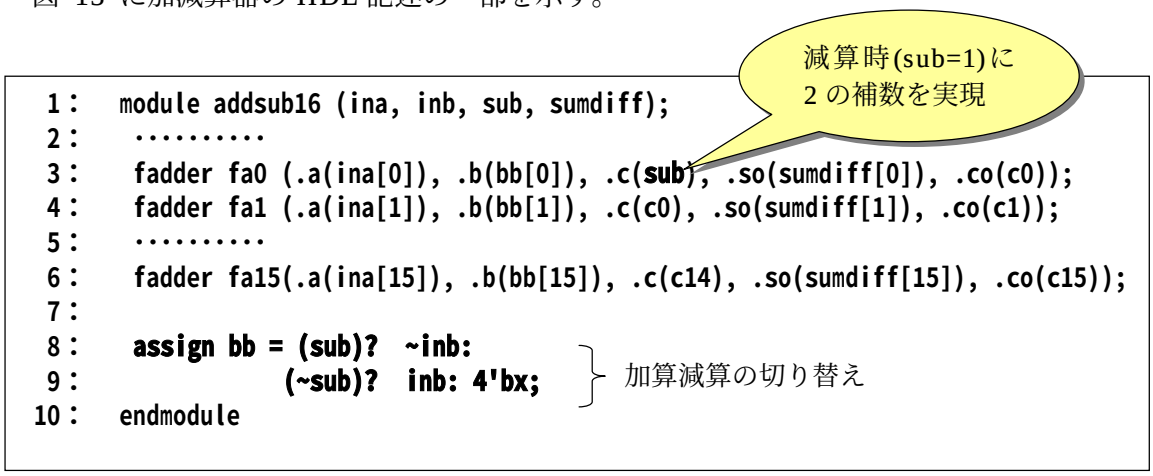

図 13：16bit 加減算器の HDL 記述例

3～6 行目では 16 個の全加算器をインスタンスし、それらを連結することで 16 ビット加算器を実現している。そして、8,9 行目において加算と減算の切り替えを行っている。sub が 1 の場合は入力 inb を反転し、なおかつ、3 行目において sub(ここでは１)を引数といて与えておくことで 2 の補数を加算する、つまり減算を実現している。sub が 0 の場合は入力 inb をそのまま流して加算器としている。

４．３．３　ＡＬＵの設計

加減算器にさまざまな機能を加えて ALU を設計する。

図 14 に ALU の HDL 記述の一部を示す。

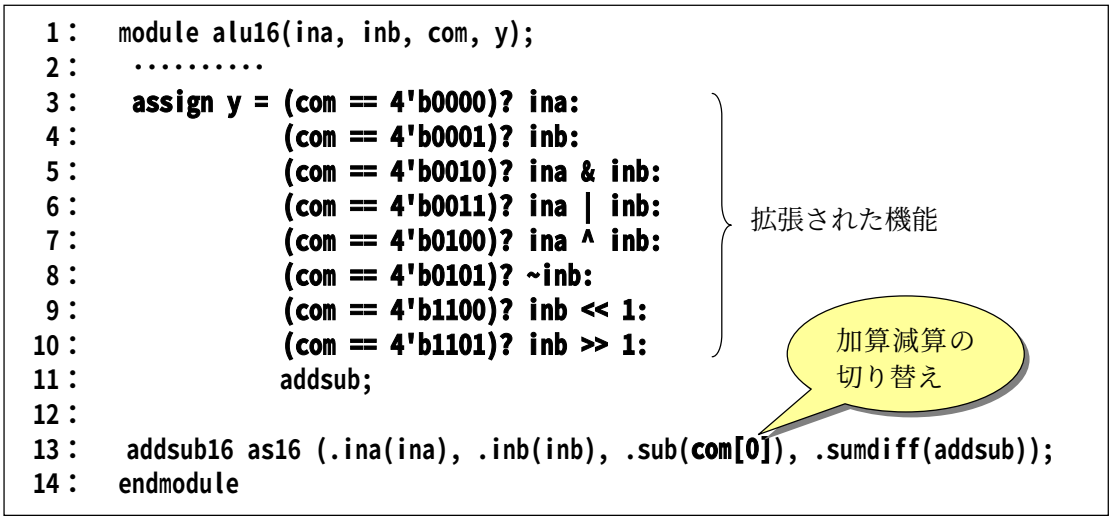

図 14：16bitALU の HDL 記述例

３～10 行目が拡張された機能である。入力 com の値によってそれぞれの演算結果が出力される。それ以外の場合(11 行目)は com の最下位ビット com[0]によって加算器または減算器としてインスタンスされた加減算器(addsub16)が動作し(13 行目)、その結果が出力される。

４．３．４　レジスタファイルの設計

汎用レジスタマシンはレジスタの数が多いため、レジスタファイルという一種のメモリを作成する。性能を維持するために同時に２つのデータを読むことのできる Dual Port メモリ型のレジスタファイルを設計する。

図 15 にレジスタファイルの HDL 記述の一部を示す。

３～８行目は、制御入力端子 req が 0 の時、addr1 で指定された番号のレジスタ内容が dout1 から読み出され、addr2 で指定された番号のレジスタ内容が dout2 から読み出される機能である。19～20 行目では、制御入力端子 req が 1 の時に addr1 で指定された番号のレジスタに din で入力されたデータが書き込まれる機能を実現している。

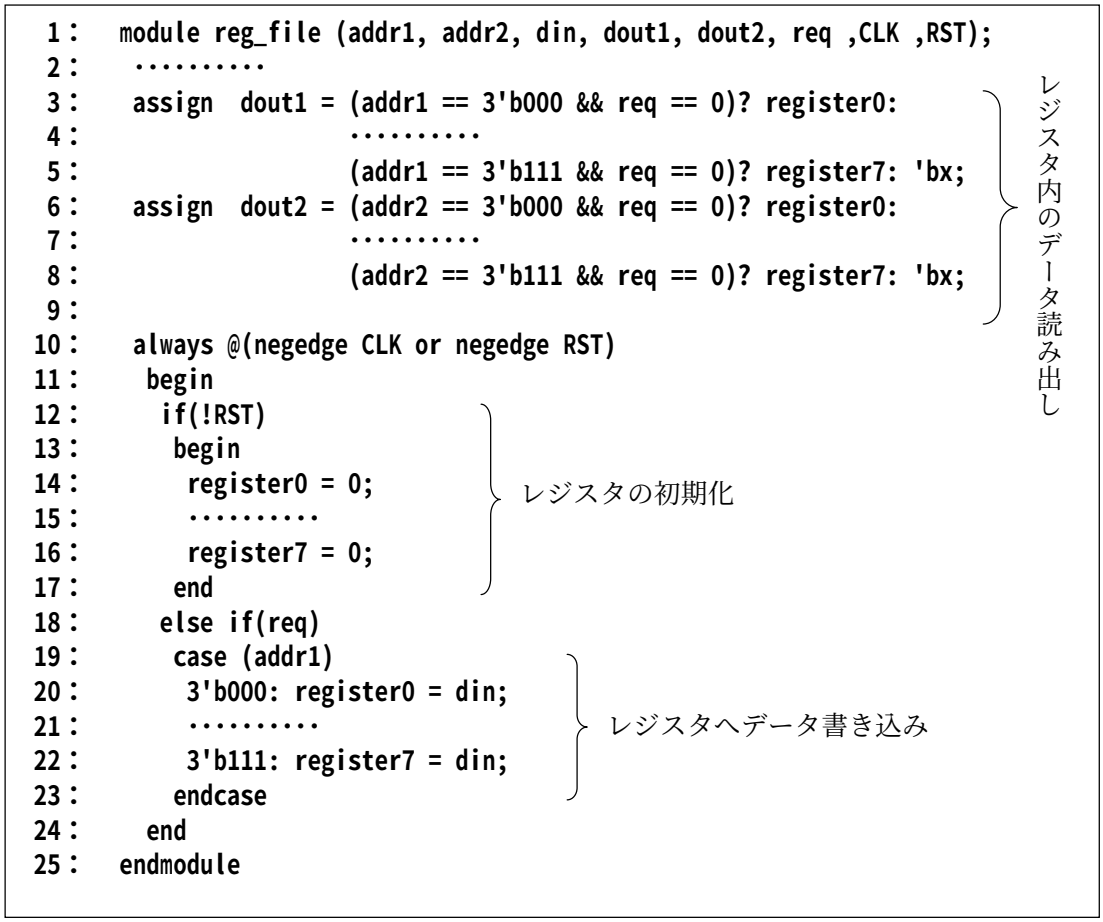

図 15：Dual Port メモリ型レジスタファイルの HDL 記述例

４．３．５　メモリの記述

今回、CAD ツールとして Xilinx 社 Foundation ISE を用いたが、この CAD ツールに対して、マイクロプロセッサに不可欠なメモリの VerilogHDL 記述方法には次の 3 通りがある。

(1) BlockRAM

Foundation ISE に付属の Core Generator によって生成されるソフト IP(Intellectual Property)である。パラメータを設定することで遅延など柔軟なデザインが可能であり、配置配線時に FPGA 上にインプリメントされる。

(2) レジスタメモリ

VerilogHDL でトップモジュールの下位モジュールとしてメモリを記述する。つまり、実質的にはメモリ機能を持ったレジスタ群をメモリとして扱う。そのためメモリまでを論理合成することになり、配置配線時にメモリの容量によっては膨大な領域を占有することになる。

(3) テストメモリ

メモリの記述をテストベンチ内に記述する。つまり、メモリ機能を持ったテストパタンを仮想的なメモリとして扱う。テストベンチ内に記述されているため、論理合成は行われず、配置配線されることもない。

メモリ素子は、それ自体を論理合成して作ることはまずなく、既にメモリとして製造されたデバイスを基盤に組み込んで製品化される[2]。そのことを考慮した上で、今回の研究では 図 12 の右側ようにテストベンチ内に仮想的なメモリを記述することで、各シミュレーション時にだけ用いるテストメモリとして設計を行った。

図 16 にテストベンチ内におけるテストメモリ記述の構造を示す。

```
 1：  module bpicosim;
 2：   ··········
 3：   assign  memdout = (!write) ? {mem[memaddin & 8'b11111110],
 4：                                mem[memaddin | 8'b00000001]}:16'bx;
 5：   always @(negedge CLK)
 6：    if(write) begin
 7：     mem[memaddin & 8'b11111110] <= memdin[15:8];
 8：     mem[memaddin | 8'b00000001] <= memdin[7:0];
 9：    end
10：
11：   bpico pico(.datain(memdout),.addressout(memaddin),
12：              .dataout(memdin),.write(write),.CLK(CLK),.RST(RST))
13：   initial
14：    begin
15：     $readmemb("memdata.txt",mem);
16：     ··········
17：     #40000 $stop;
18：    end
19：  endmodule
```

テストメモリ記述部

システムタスクを用いてtxtファイルからテストパタンを読み込む

図 16：テストベンチ内のテストメモリの HDL 記述例

図 16 の破線内がテストメモリの記述に当たる。３～９行目は、制御入力端子 write が 0 の時は memaddin で指定されたアドレスの内容を memdout に出力し、write が 1 の時には memaddin で指定されたアドレスに memdin の内容を書き込む動作を実現している。また、テストパタンはシステムタスク$readmemb **を用いる事で外部 txt ファイル等から読み込む事が可能である(15 行目)。**

４．４　シミュレーションによる検証

マルチサイクルマイクロプロセッサの HDL 記述後、Model Technology 社 ModelSim を用いてビヘイビアシミュレーションを行い、動作検証を行う。私用に沿った正しい動作が得られれば、Xilinx 社 Foundation ISE を用いて論理合成を行い、ネットリストを生成する。次に再度 ModelSim にて、ゲートレベルシミュレーションを行い動作検証する。

今回テストパタンとして加算、乗算、8 個中の最大値、ある数の３乗、3 つの値の平均を求めるテストプログラムを作成し、シミュレーションのテストパタンとした。

表 1 にシミュレーションで用いたテストパタンの各プログラムの行数を示す。

表 1：テストパタンのプログラム行数

	加算	乗算	最大値	３乗	平均値
プログラム行数	7	9	11	12	19

(単位:行)

図 17 にゲートレベルシミュレーションの様子を示す。

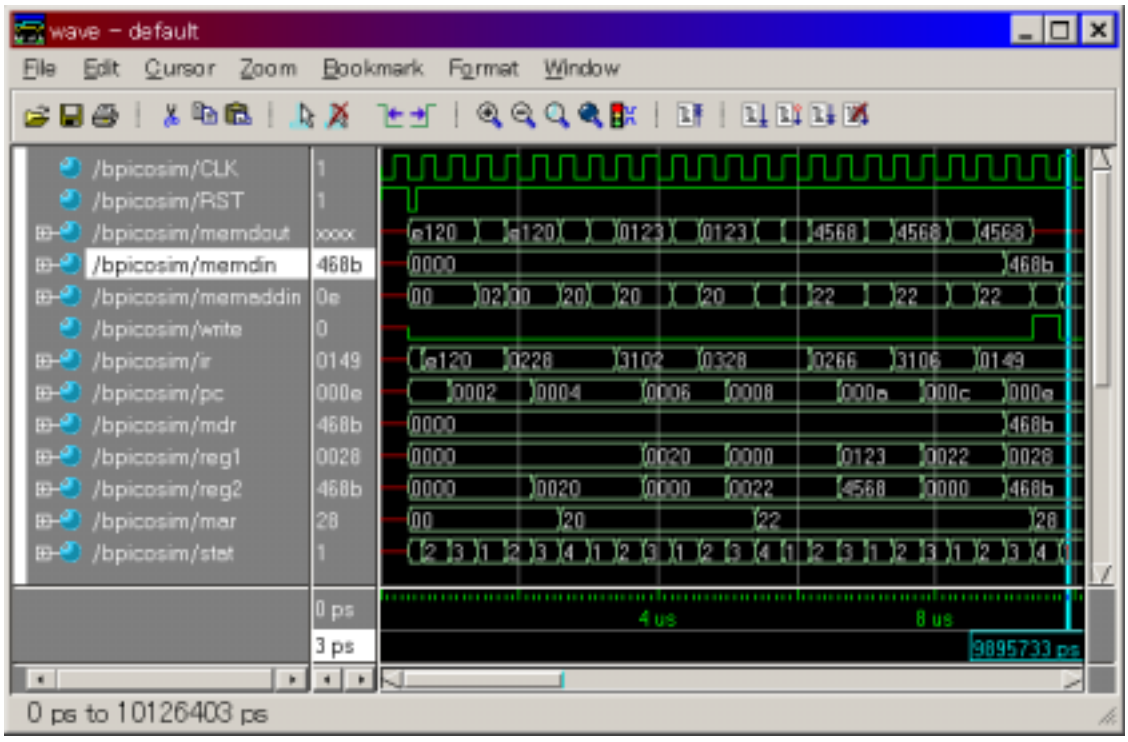

図 17：ModelSim によるタイミングチャート

図 17 はメモリ上にある 16'b0123 と 16'b4568 をロードし、レジスタ間演算を用いての加算を実行した様子である。確かに、加算結果 16'b468b が memdin に出力され、メモリにストアされているのが分かる。

表 2 に各テストパタンによるシミュレーション結果を示す。

表 2：テストパタンによるシミュレーション結果

	加算	乗算	最大値	3乗	平均値
実行命令回数	7	12	56	28	72
クロックサイクル数	24	37	175	86	220
CPI	3.43	3.08	3.13	3.07	3.06

(実行命令回数,クロックサイクル数:回)

表 2 において、実行命令回数はループなどで実際に命令が実行された回数を示し、クロックサイクル数はそれらの命令実行にかかったクロック数である。

設計したマルチサイクルマイクロプロセッサはメモリアクセスの LD,ST 命令と、JAL 命令の場合は4クロックサイクルで行い、その他の命令は3クロックサイクルで処理を行う。そのため、CPI が3以上4以下の値に収まっていることが 表 2 から確認できる。加算の CPI が最も高いのは、テストプログラムが短いためであり、全体に占める LD,ST 命令の割合が必然的に高くなっていることが原因である。

表 3 に Foundation の Place&Route レポートから得た性能を示す。

表 3：マルチサイクルマイクロプロセッサの性能

クリティカルパス	最大動作周波数
64.97 ns	15.39 MHz

表 3 のようにクリティカルパス(最大遅延)は 64.97 ns であった。最大動作周波数はクリティカルパスの逆数を取ることで求めている。

この性能を上げるためには、現在の ALU 内の加算器は順次桁上げ方式であるので、これを桁上げ先見方式に入れ替えることが上げられる。資源の消費面積と消費電力の増大を伴うことにはなるが、最大遅延を短くでき、最大動作周波数を上げることができると考えられる。

また、CPI を減らすことで、性能向上を図ることが考えられる。この方法については5章の「パイプライン方式による PICO16 の設計」で説明する。

5　パイプライン方式によるＰＩＣＯ１６の設計

5．1　パイプラインデータパス

図 18 にパイプラインマイクロプロセッサのデータパスを示す。

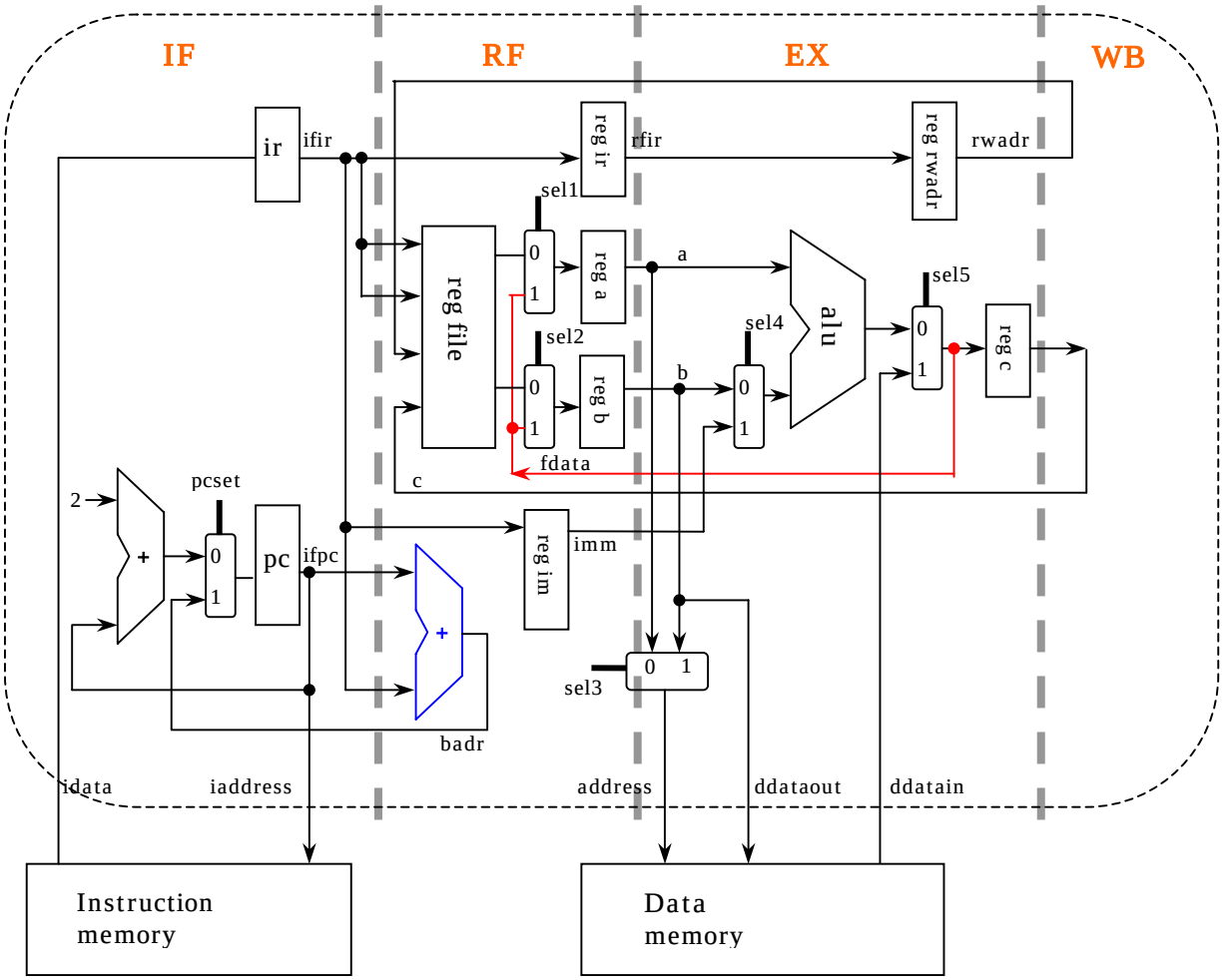

図 18：パイプラインマイクロプロセッサのデータパス

パイプラインマイクロプロセッサのデータパスは以下の要素から構成される。

- Instruction memory：命令メモリ。命令が格納された読み出し専用のメモリ。
- Data memory：データメモリ。データの読み書きが可能なメモリ。
- pc：プログラムカウンタ。次に実行する命令メモリの番地を保持するレジスタ。
- ir：命令レジスタ。命令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_im：即値レジスタ。ir[7:0]を符号拡張した値を格納するレジスタ。
- alu：算術演算ユニット。
- reg_file：演算結果などの途中データを書き込む 8 つのレジスタ。3 ポートメモリ型。
- mux：マルチプレクサ

５．２　パイプライン命令実行と制御

５．２．１　パイプライン命令実行

パイプラインマイクロプロセッサは、処理の効率を上げるために１つの命令を、IF(命令フェッチ)、RF(レジスタフェッチ)、EX(実行)、WB(書き込み)などの複数のステージに分割し、それぞれのステージを独立して動作させることにより、流れ作業的に命令の処理を行う。

図 19 にそのパイプライン処理の様子を示す。

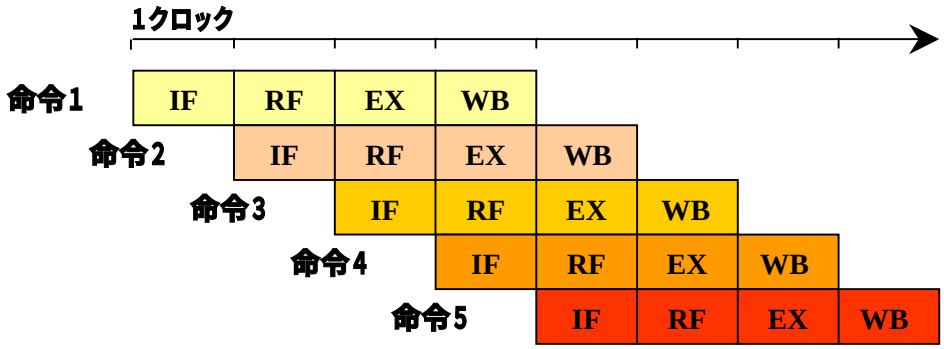

図 19：パイプライン方式の処理

図 19 のパイプライン処理では１つの命令を IF,RF,EX,WB の４つのステージに分割し、各ステージを 1 クロックサイクルで処理を行っている。命令１の IF ステージが終わった直後に命令２の IF ステージの実行が開始され、それ以降同様に 1 クロックサイクルごとに次々と命令が実行開始される。このように、パイプライン処理では 1 クロックサイクル当たり複数命令のステージ実行が並列的に行われることにより、CPI を小さくすることが可能となり、処理の効率化へと繋がる。

５．２．１　データハザード

データハザードとは、パイプライン処理中に命令前後のデータの依存性によって、命令の計算結果が出る前に、次の命令によってそのデータが読み出されてしまうことである。

図 20 にデータハザードの様子を示す。

赤線で囲まれて部分では、最初の命令の WB ステージにおいて、初めて r1 にデータ 0 が格納されるが、それ以前の時刻である次の命令の RF ステージにおいて r1 が読み出されようとするために、ここでは正しい r1 のデータを読み出すことができない。これがデータハザードである。青線で囲まれた部分についても同様にデータハザードが生じている。

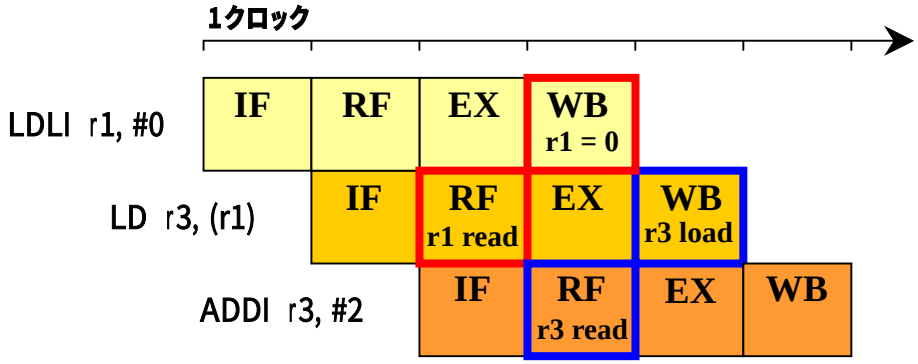

図 20：データハザードの様子

５．２．２　ストール

データハザードを回避する１つの方法としてストールを用いる方法がある。ストールとは、何もしない命令である NOP 命令をパイプライン中に流すことによって、次の命令へのタイミングを合わせる方法である。この NOP 命令はバブルと呼ばれている。

図 21 にストールによるデータハザードの回避の様子を示す。

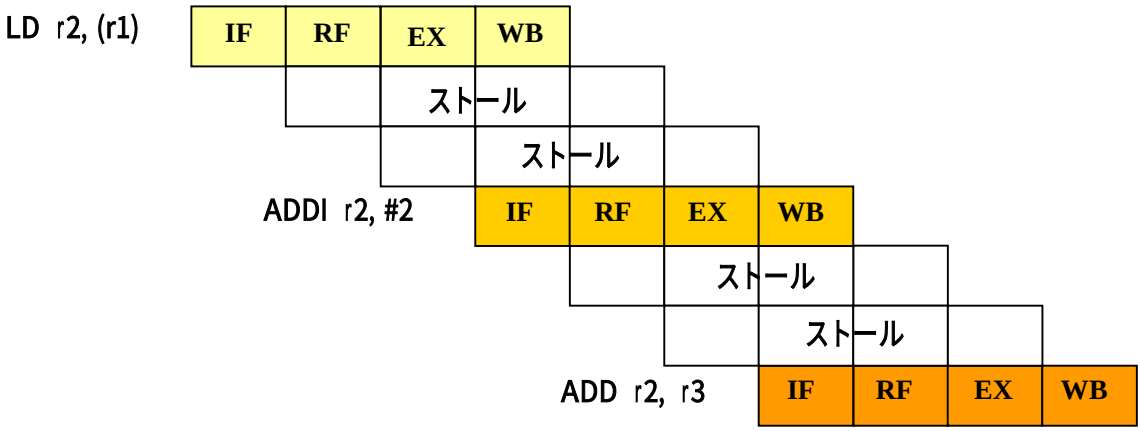

図 21：ストールによるデータハザードの回避

図 21 のようにストールによってデータの授受のタイミング合わせることで、データハザードを回避することができる。しかし、実際のプログラミング上では、ある命令の実行した結果を次の命令で使うことは、かなりの確率で起こり得ることであり[2]、必然的にこのストールの発生率も高くなる。そして、ストールの発生率が高くなると CPI が増大してしまい、CPI を小さくできるというパイプライン処理の恩恵が無駄になる。

５．２．３　フォワーディング

CPI を増やさずにデータハザードを回避するためにフォワーディングという方法を用いる。フォワーディングとはデータの先送りのことで、レジスタファイルに書き込むと同時、または ALU による演算直後のデータを横流し(先送り)する機構を設け、次の命令で読み出したデータの代わりに用いる方法である。

図 22 にフォワーディングによるデータハザードの回避の様子を示す

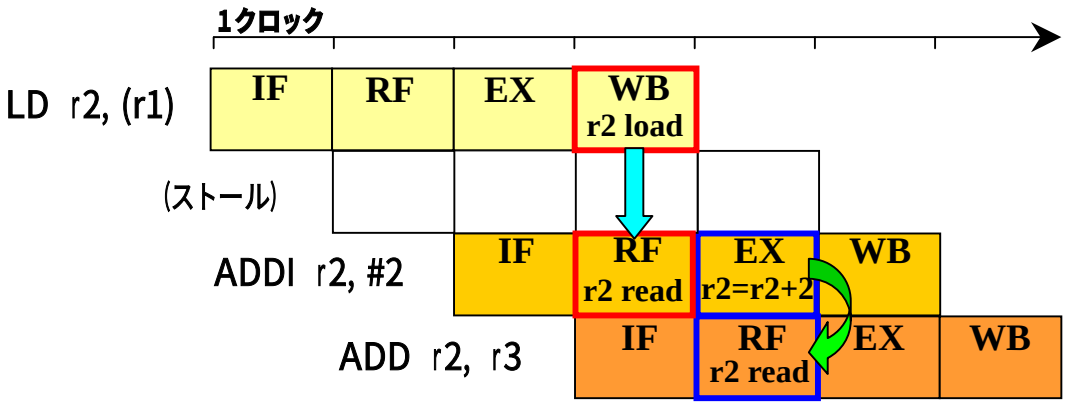

図 22：フォワーディングによるデータハザードの回避

今回の PICO のパイプラインにおいては２箇所にフォワーディングを設けている。

まず、図 22 における水色の矢印は、初めの LD 命令の WB ステージにてレジスタファイルに書き込むと同時にデータを横流しし、次の ADDI 命令の RF ステージで読み出すデータとして用いている様子を示している。この機能は実際のプログラミング上ではレジスタファイルの内部で実現している。

次に緑色の矢印は、ADDI 命令の EX ステージの ALU で計算した直後のデータを、次の ADD 命令の RF ステージに横流しすることでフォワーディングしている様子を示す。図 18 の赤線のパスがこの機能に当たる。

このようにフォワーディングを行う機構を設けることで、ストールの発生率を抑えることができ、CPI の小さい、効率的なパイプライン処理を実現することができる。

５．２．４　制御ハザード

パイプライン処理は、前に述べたように 1 クロックサイクル毎にどんどんと命令をフェッチして実行する形式である。通常の命令の場合は問題ないが、分岐命令の場合は、次の命令のフェッチするアドレス変更する処理を行うため、パイプラインが正常に動作することができない。このことを制御ハザードという。

図 23 に制御ハザードによる分岐の遅れの様子を示す。

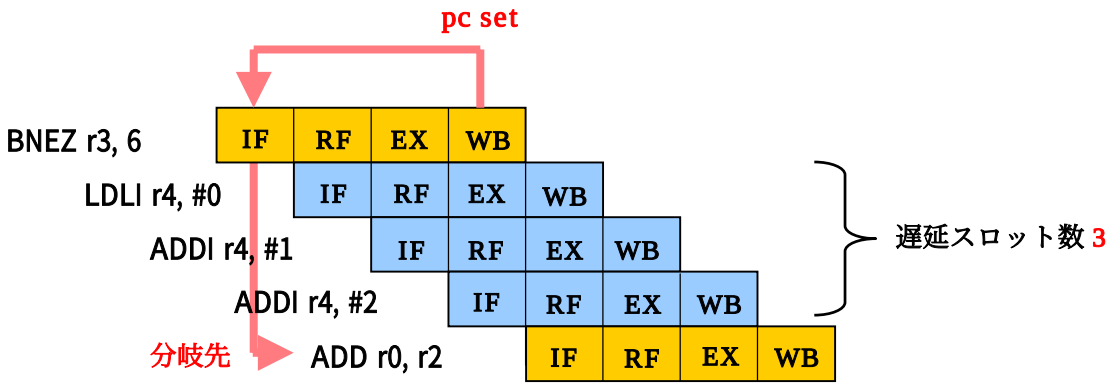

図 23：制御ハザードによる分岐の遅れ

図 23 の例のように、BNEZ 命令が実行開始して、飛び先番地が実際に pc にセットされるのは WB ステージであり、ここで初めて分岐が成立する。しかし、IF ステージから WB ステージまで 3 クロックサイクルかかるために、本来フェッチする必要のない次の 3 つの命令が、次々とフェッチされてしまい、パイプラインは正常に動作しない。ここでは、分岐が 3 命令遅れているのでこのことを遅延スロット数は 3 である。

これを回避する最も簡単な方法は、分岐命令後の 3 命令は NOP 命令以外許さない機構を設けることである。しかし、一般に分岐命令の生起確率は 20%～30%と言われているため、これを考慮すると性能が約半分になり、パイプラインの恩恵があまり感じられなくなってしまう。

５．２．５　制御ハザードの処置

制御ハザードの影響を軽減するためには、分岐命令か否かを早いステージで検出し、分岐命令であれば、早めに分岐する必要がある。命令は IF ステージでフェッチするため、RF ステージに分岐命令かどうかを判断する機構と、新たに飛び先番地を計算するための加算器を ALU とは別に設ける。

このような拡張によって、制御ハザードを軽減した様子を 図 24 に示す。

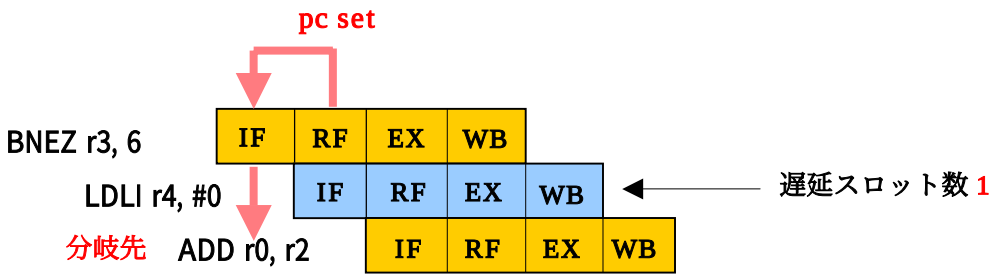

図 24：制御ハザードの軽減

RF ステージに分岐判定の機構と、飛び先番地を計算する加算器(図 18 における青色の加算器)を新たに設けることで、図 23 の遅延スロット数が３であったハードウェアから、図 24 のように遅延スロット数を１にまで軽減することができる。

５．３　ＨＤＬによるパイプラインマイクロプロセッサの設計

５．３．１　モジュール構成

VerilogHDL 記述によるパイプラインマイクロプロセッサの設計を行った。パイプラインマイクロプロセッサの回路を記述する基本構成である構成モジュールは、ALU、加減算器、全加算器、加算器、マルチプレクサ(MUX)、レジスタファイルからなっており、に示すようなモジュール構成になっている。

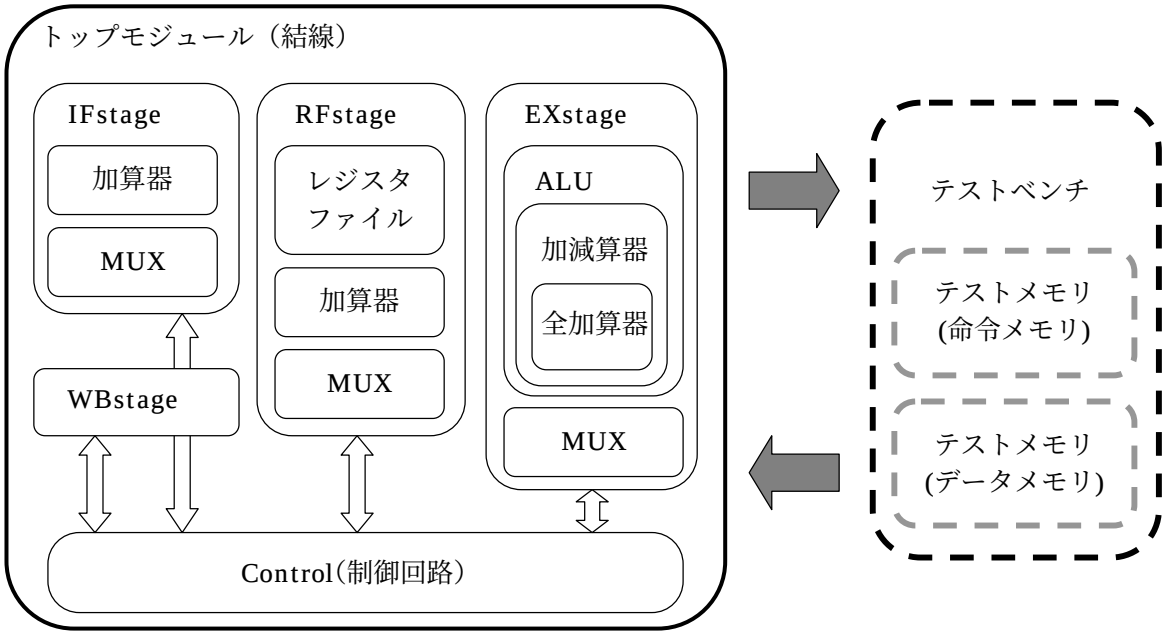

図 25：パイプラインマイクロプロセッサのモジュール構成

図 25 のように、トップモジュールには大きく、IFstage モジュール,RFstage モジュール,EXstage モジュール,WBstage モジュール,Control モジュールが結線されており、さらにそれぞれに下位モジュールがインスタンスされている。制御に関する機能はすべて Control モジュールで行うように設計を行っている。

５．３．２　３ポートメモリ型レジスタファイルの設計

マルチサイクルマイクロプロセッサのレジスタファイルは２つの読み出しを同時に行うことができる Dual Port メモリ型であったが、パイプラインマイクロプロセッサでは、RF ステージでの読み出しと、WB ステージの書き込みを同時に行う必要があるため、２つの読み出しと１つの書き込みを同時に行うことのできる３ポートメモリ型に拡張する。

また、レジスタファイル内には WB ステージでの書き込みの際のフォワーディングの機構も実装する。

図 26 に拡張したレジスタファイルの HDL 記述の一部を示す。

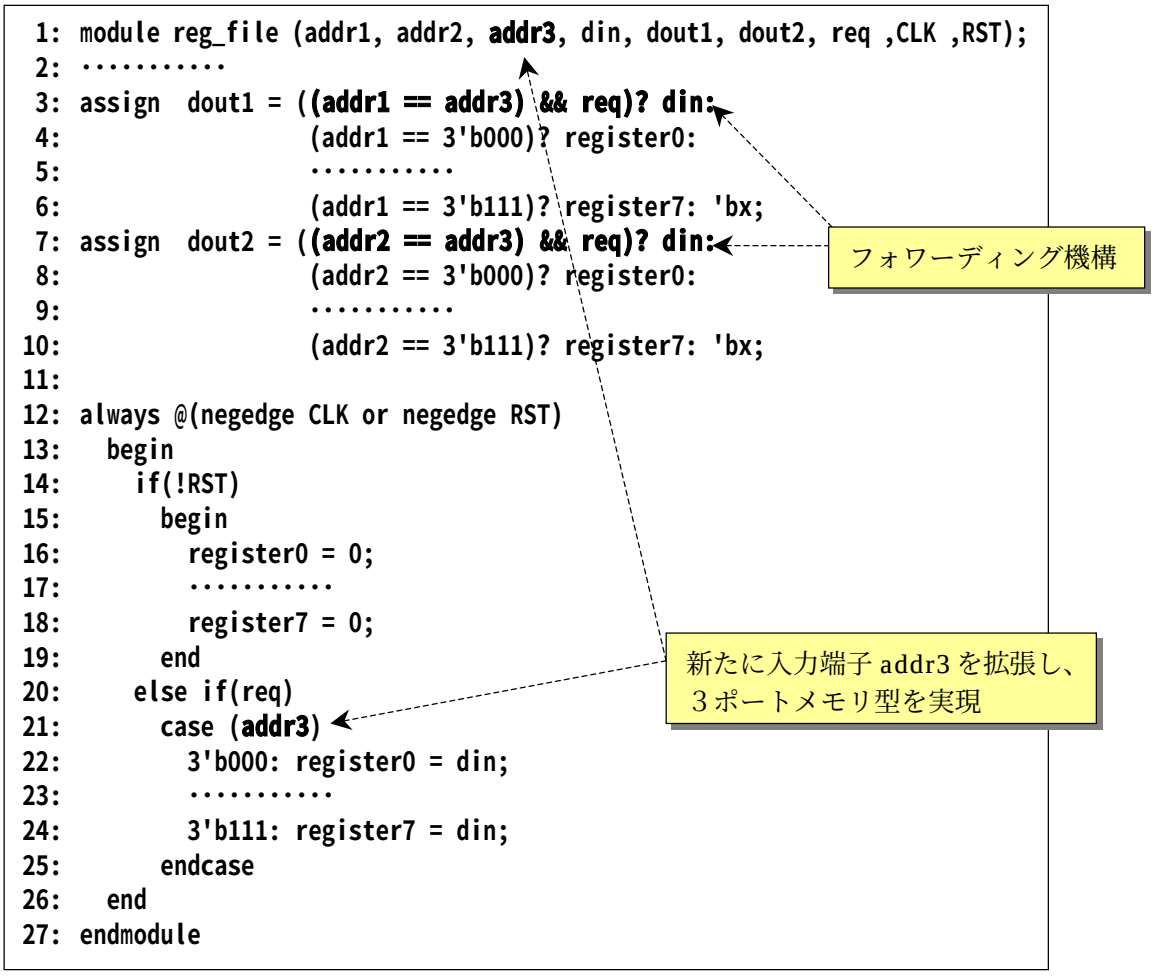

図 26：拡張した３ポートメモリ型レジスタファイルの記述例

図 26 の 1 行目と 21 行目にあるように、addr3 を入力端子として追加することで、３ポートメモリ型への拡張を実現している。

また、3 行目と７行目がフォワーディング機能に当たる。ここでは addr1 と addr3、または、addr2 と addr3 が同じとき、din を dout1 または dout2 に横流ししている。つまり、RF ステージでの読み出しと WB ステージの書き込みが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の横流しをすることでフォワーディングを実現している。図 22 における水色の矢印の機能がこの記述に相当する。

５．３．３　ＥＸステージからのフォワーディング機能の設計

ストールによる性能低下を軽減するために、EX ステージの結果を格納するレジスタ(reg c)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(reg a, reg b)の入力へ横流しを行うフォワーディング機構を実装する。

図 27 に Control モジュール内のフォワーディング機能の HDL 記述の一部を示す。

```
1:  module control (sel1, sel2, sel3, sel4, sel5, pcset, com, rwen, write,
2:                                              b_ifir, rfir, ifir, f_reg_a);
3:  ..........
4:  assign sel1 = (rwen && (rfir[10:8] == ifir[10:8]))? 1: 0;
5:  assign sel2 = (rwen && (rfir[10:8] == ifir[7:5]))? 1: 0;
6:  ..........
7:
8:  endmodule
```

図 27：Control モジュール内のフォワーディング機能の記述例

図 27 の４,５行目が EX ステージからのフォワーディング機構に当たる。先行する命令の書き込みレジスタ番号(rfir[10:8])と読み出しを行うレジスタ番号(ifir[10:8])が一致する場合、reg a にはレジスタファイルから読み出した値ではなく、フォワーディングした値(fdata)を格納するように、マルチプレクサ sel1 を切り替える。同様に、先行する命令の書き込むレジスタ番号(rfir[10:8])と読み出しを行うレジスタ番号(ifir[7:5])が一致する場合には、マルチプレクサ sel2 を切り替え、reg b にフォワーディングした値(fdata)を格納する。これらは図 22 における緑色の矢印の機能に相当する記述である。

５．３．４　制御ハザードの回避機能の設計

制御ハザードによる性能低下を回避するために、RF ステージに分岐制御を行う機構と、番地の飛び先を計算する加算器を実装する。

図 28 に RF ステージにおける制御ハザード回避機能の HDL 記述の一部を示す。

```
1:  module rf_stage (rfir, a, b, rfim, badr, f_reg_a, ifir, ifpc, c, rwadr, fdata,
2:                              rwe, sel1, sel2, jal_en, jr_en, jmp_jal_en, CLK, RST);
3:  ..........
4:  assign  badr_out = ifpc + in_badr;
5:  assign  in_badr = (jmp_jal_en)?
6:      {ifir[7],ifir[7],ifir[7],ifir[7],ifir[7],ifir[7],ifir[10:0]}:
7:      {ifir[7],ifir[7],ifir[7],ifir[7],ifir[7],ifir[7],ifir[7],ifir[7],ifir[7:0]};
8:
9:  assign  in_reg_ir = (jal_en)? 16'h0701: ifir;
0:  assign  badr = (jr_en)? f_reg_a: badr_out;
1:  ..........
2:
3:  endmodule
```

図 28：RF ステージにおける制御ハザード回避機能の記述例

図 28 の４行目の記述が、飛び先を計算する専用の加算器である。そして、４～６行目の記述では JMP,JAL 命令の時、相対分岐するアドレスが格納された ifir[10:0]を 16 ビット符号拡張した値と、現在の番地を加算器で加算し、飛び先番地を計算している。また、9 行目は JAL,JALR 命令の時に in_reg_ir(reg ir)に 16'h0701(16'b0000_0111_0000_0001)を格納することで、現在の番地が alu をスルー(com=4'b0001)してレジスタファイルの 7 番地(rwadr=3'b111)に退避する仕掛けになっている。10 行目では JALR,JR 命令の時に、絶対分岐させるため、加算器を介さずに直接レジスタファイルの出力を飛び先番地としている。

6　おわりに

本論文ではマルチサイクル･パイプライン方式のマイクロプロセッサのアーキテクチャを述べると共に、ハードウェア記述言語 VerilogHDL を用いての各マイクロプロセッサの設計を行った。命令セットは実験教育用マイクロプロセッサ PICO16 の用いての設計を行った。設計したマルチサイクルマイクロプロセッサはビヘイビアシミュレーションの後、論理合成を行い、ゲートレベルシミュレーションまでの動作検証を行った。これらの経緯から、ハードウェア記述言語 VerilogHDL, SFL の記述法、シミュレータ、論理合成ツールの使用法を習得することができ、大学院での研究へと繋がる基礎的な技術を習得できたと思う。

今後の課題としては、まず、パイプライン方式マイクロプロセッサの完成である。そして、ゲートレベルシミュレーションを行い、マルチサイクル方式マイクロプロセッサとの性能比較を行いたいと考えている。また、さらなる高速化技術である浮動小数点演算装置の付加を検討すると共に、プログラムの最適化を行う。そして、最終的には配置配線処理を行った上で FPGA 上へマイクロプロセッサを実装し、実際に LSI としてプログラムを動作させたいと考えている。

また、現在注目を集めている HDL と比べてより C 言語に近い C 言語ベースのシステム設計言語(SystemC,SpecC,HandelC 等)によるハードウェアの設計手法についても学びたいと考えている。

謝辞

この度、私が研究するにあたりまして、貴重な助言、御指導をいただきました山崎 勝弘教授、小柳 滋教授に深く感謝致します。また、本研究に関して貴重な御意見をいただきました、池田 修久氏、大八木 睦氏及び同研究室内の方々、様々な助言や励ましを下さったことを深く感謝致します。

参考文献

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

[2] 天野英晴, 西村克信 共著, 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.

[3] 立命館大学 VLSI センター作成:社会人向け VLSI 設計セミナー, レクチャー用マニュアル,2002.

[4] VDEC 監修, 浅田邦博 編:ディジタル集積回路の設計と試作,培風館,2000.

[5] 電子情報通信学会 VLSI 設計技術研究専門委員会主催 講習会 ディジタル集積回路設計 レクチャー用マニュアル.

[6] 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.

[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] 深山正幸, 北川章夫, 秋田純一, 鈴木正國 著:HDL による VLSI 設計 第 2 版, 2002.

付録 ソースプログラム

付録 マルチサイクルマイクロプロセッサ

```
/*----------------------------------------------------top module----------------------------------------------------*/
`timescale  1 ns / 1 ps

`define IF 1
`define RF 2
`define EX 3
`define EX2 4

module bpico(dataout,addressout,write,datain,CLK,RST,
//test pin
pc,mar,mdr,ir,reg1,reg2,sel1,sel2,sel3,com,req,stat,dbus, sbus1, sbus2, dout1, dout2,
aluout
//test pin end
);

//test define
  output [15:0]    ir, pc, mdr, reg1 ,reg2;
  output [7:0]     mar;
  output [2:0]     stat;
  output          sel1, sel3;
  output [2:0]    sel2;
  output          req;
  output [3:0]    com;
  output [15:0]   dbus, sbus1, sbus2, dout1, dout2, aluout;
//test define end

  output [15:0] dataout;
  output [7:0]  addressout;
  output        write;
  input [15:0]  datain;
  input         CLK, RST;

  reg [15:0]    ir, pc, mdr, reg1 ,reg2;
  reg [7:0]     mar;
  reg [2:0]     stat;

  wire [15:0]   dbus, sbus1, sbus2, dout1, dout2, aluout;
  wire          sel1, sel3;
  wire [2:0]    sel2;
  wire          req, write;
  wire [3:0]    com;
  wire [2:0]    addr1;   /* JAL */
```

```
  parameter [4:0] ROP = 5'b00000,
                  LDLI= 5'b11100,
                  LDHI= 5'b11101,
                  BNEZ= 5'b01001,
                  BEQZ= 5'b01010,
                  BMI = 5'b01011,
                  BPL = 5'b01100,
                  JR  = 5'b01110,
                  JMP = 5'b01111,
                  JAL = 5'b01101,
                  LD  = 5'b01000,
                  ST  = 5'b01001,
                  SL  = 5'b01100,
                  SR  = 5'b01101;


/********************** not sync CLK ***********************/
  assign dataout = mdr;
  assign addressout = (stat == `IF) ? pc: mar;
  assign addr1 = (stat == `EX && ir[15:11] == JAL) ? 3'b111 : ir[10:8];   /* JAL */

  assign sel1 = ( (stat == `IF)
                ||(stat == `EX2 && ir[4:0] == LD)
                ||(stat == `EX2 && ir[4:0] == ST) )? 1'bx:

                ( (stat == `RF)
                ||(stat == `EX && ir[15:11] == BNEZ)
                ||(stat == `EX && ir[15:11] == BEQZ)
                ||(stat == `EX && ir[15:11] == BMI)
                ||(stat == `EX && ir[15:11] == BPL)
                ||(stat == `EX && ir[15:11] == JMP)
                ||(stat == `EX && ir[15:11] == JAL)
                ||(stat == `EX2 && ir[15:11] == JAL) )? 1'b0:

                ( (stat == `EX && ir[15:11] == ROP && ir[4:0] == LD)
                ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == ST)
                ||(stat == `EX && ir[15:11] == ROP && ir[4:3] == 2'b00)
                ||(stat == `EX && ir[15:11] == ROP && (ir[4:0] == SL || ir[4:0] == SR ))
                ||(stat == `EX && ir[15:11] == LDLI)
                ||(stat == `EX && ir[15:11] == LDHI)
                ||(stat == `EX && ir[15:14] == 2'b00)
                ||(stat == `EX && ir[15:11] == JR) )? 1'b1:

                'bx;
```

```
assign sel2 = ( (stat == `IF)
              ||(stat == `EX2 && ir[4:0] == LD)
              ||(stat == `EX2 && ir[4:0] == ST) )? 'bx:

              (stat == `RF)? 0:

              ( (stat == `EX && ir[15:11] == ROP && ir[4:0] == LD)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == ST)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:3] == 2'b00)
              ||(stat == `EX && ir[15:11] == ROP && (ir[4:0] == SL || ir[4:0] == SR ))
              ||(stat == `EX && ir[15:11] == JR)
              ||(stat == `EX && ir[15:11] == JAL) )? 1:

              ( (stat == `EX && ir[15:11] == LDLI)
              ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] == 2'b11)
              ||(stat == `EX && ir[15:11] == BNEZ)
              ||(stat == `EX && ir[15:11] == BEQZ)
              ||(stat == `EX && ir[15:11] == BMI)
              ||(stat == `EX && ir[15:11] == BPL) )? 2:

              (stat == `EX && ir[15:14] == 2'b00 && ir[13:12] != 2'b11)? 3:

              (stat == `EX && ir[15:11] == LDHI)? 4:

              ( (stat == `EX && ir[15:11] == JMP)
              ||(stat == `EX2 && ir[15:11] == JAL) )? 5:

              'bx;




assign sel3 = ( (stat == `RF)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == LD)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == ST)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:3] == 2'b00)
              ||(stat == `EX && ir[15:11] == ROP && (ir[4:0] == SL || ir[4:0] == SR ))
              ||(stat == `EX && ir[15:11] == LDLI)
              ||(stat == `EX && ir[15:11] == LDHI)
              ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] == 2'b11)
              ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] != 2'b11)
              ||(stat == `EX && ir[15:11] == BNEZ)
              ||(stat == `EX && ir[15:11] == BEQZ)
              ||(stat == `EX && ir[15:11] == BMI)
              ||(stat == `EX && ir[15:11] == BPL)
              ||(stat == `EX && ir[15:11] == JMP)
              ||(stat == `EX && ir[15:11] == JR)
              ||(stat == `EX && ir[15:11] == JAL)
              ||(stat == `EX2 && ir[15:11] == JAL) )? 1'b0:
```

```
                ( (stat == `IF)
                ||(stat == `EX2 && ir[4:0] == LD) )? 1'b1:

                (stat == `EX2 && ir[4:0] == ST)? 1'bx:

                'bx;


  assign  com = ( (stat == `IF)
                ||(stat == `EX2 && ir[4:0] == LD)
                ||(stat == `EX2 && ir[4:0] == ST) )? 4'bx:

                ( (stat == `EX && ir[15:11] == ROP && ir[4:0] == ST)
                ||(stat == `EX && ir[15:11] == JR)
                ||(stat == `EX && ir[15:11] == JAL) )? 4'b0000:

                (stat == `EX && ir[15:11] == ROP && ir[4:3] == 2'b00)? ir[3:0]:
                (stat == `EX && ir[15:11] == ROP && (ir[4:0] == SL || ir[4:0] == SR))?
{1'b1,ir[2:0]}:

                ( (stat == `EX && ir[15:11] == ROP && ir[4:0] == LD)
                ||(stat == `EX && ir[15:11] == LDLI)
                ||(stat == `EX && ir[15:11] == LDHI) )? 4'b0001:

                ( (stat == `EX && ir[15:14] == 2'b00 && ir[13:12] == 2'b11)
                ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] != 2'b11) )? ir[14:11]:

                ( (stat == `RF)
                ||(stat == `EX && ir[15:11] == BNEZ)
                ||(stat == `EX && ir[15:11] == BEQZ)
                ||(stat == `EX && ir[15:11] == BMI)
                ||(stat == `EX && ir[15:11] == BPL)
                ||(stat == `EX && ir[15:11] == JMP)
                ||(stat == `EX2 && ir[15:11] == JAL) )? 4'b0110:

                'bx;


  assign  req = ( (stat == `IF)
                ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == LD)
                ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == ST)
                ||(stat == `EX && ir[15:11] == BNEZ)
                ||(stat == `EX && ir[15:11] == BEQZ)
                ||(stat == `EX && ir[15:11] == BMI)
                ||(stat == `EX && ir[15:11] == BPL)
                ||(stat == `EX && ir[15:11] == JMP)
                ||(stat == `EX && ir[15:11] == JR)
                ||(stat == `EX2 && ir[4:0] == ST)
                ||(stat == `EX2 && ir[15:11] == JAL) )? 1'bx:

                (stat == `RF)? 1'b0:
```

```
              ( (stat == `EX && ir[15:11] == ROP && ir[4:3] == 2'b00)
              ||(stat == `EX && ir[15:11] == ROP && (ir[4:0] == SL || ir[4:0] == SR))
              ||(stat == `EX && ir[15:11] == LDLI)
              ||(stat == `EX && ir[15:11] == LDHI)
              ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] == 2'b11)
              ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] != 2'b11)
              ||(stat == `EX && ir[15:11] == JAL)
              ||(stat == `EX2 && ir[4:0] == LD) )? 1'b1:

              'bx;

assign write =( (stat == `RF)
              ||(stat == `EX && ir[15:11] == LDLI)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == LD)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:0] == ST)
              ||(stat == `EX && ir[15:11] == ROP && ir[4:3] == 2'b00)
              ||(stat == `EX && ir[15:11] == ROP && (ir[4:0] == SL || ir[4:0] == SR))
              ||(stat == `EX && ir[15:11] == LDHI)
              ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] == 2'b11)
              ||(stat == `EX && ir[15:14] == 2'b00 && ir[13:12] != 2'b11)
              ||(stat == `EX && ir[15:11] == BNEZ)
              ||(stat == `EX && ir[15:11] == BEQZ)
              ||(stat == `EX && ir[15:11] == BMI)
              ||(stat == `EX && ir[15:11] == BPL)
              ||(stat == `EX && ir[15:11] == JMP)
              ||(stat == `EX && ir[15:11] == JR)
              ||(stat == `EX && ir[15:11] == JAL)
              ||(stat == `EX2 && ir[15:11] == JAL) )? 1'bx:

              ( (stat == `IF)
              ||(stat == `EX2 && ir[4:0] == LD) )? 1'b0:

              (stat == `EX2 && ir[4:0] == ST)? 1'b1:

              'bx;


/********************** sync CLK ************************/
always @(posedge CLK or negedge RST)
  begin
    if(!RST)              /* Initialize */
      begin
        pc <= 0;
        mar <= 0;
        mdr <= 0;
        ir <=0;
        reg1 <= 0;
        reg2 <= 0;
        stat <= `IF;
      end
```

```
else
  begin
    case(stat)
      `IF:                          /**** IF ****/
        begin
          ir <= dbus;
          stat <= `RF;
        end

      `RF:                          /**** RF ****/
        begin
          pc <= dbus;
          reg1 <= dout1;
          reg2 <= dout2;

          if((ir[15:11] == ROP ) && (ir[4:0] == ST))  /* ST */
            mdr <= dout2;

          stat <= `EX;
        end

      `EX:                          /**** EX ****/
        begin
          if(ir[15:11] == ROP )       /** R type functions **/
            begin
              if(ir[4:0] == LD || ir[4:0] == ST )    /* LD/ST*/
                begin
                  mar <= dbus;
                  stat <= `EX2;
                end
              else if(   ir[4:3] == 2'b00          /* ALU instructions */
                      || ir[4:0] == SL             /* SL */
                      || ir[4:0] == SR   )         /* SR */
                begin
                  stat <= `IF;
                end
            end // if (ir[15:11] == ROP )
          else if(   ir[15:11] == LDLI             /* LDLI*/
                  || ir[15:11] == LDHI             /*LDHI*/
                  || ir[15:14] == 2'b00  )         /* ADDI/SUBI/ANDI/ORI/XORI */
            begin
              stat <= `IF;
            end

          else if(ir[15:11] == BNEZ)   /* BNEZ */
            begin
              if(reg1 != 0)
                pc <= dbus;
              stat <= `IF;
            end
          else if(ir[15:11] == BEQZ)  /* BEQZ */
```

```
            begin
              if(reg1 == 0)
                pc <= dbus;
              stat <= `IF;
            end


          else if(ir[15:11] == BMI)   /* BMI */
            begin
              if(reg1[15] == 1'b1)
                pc <= dbus;
              stat <= `IF;
            end
          else if(ir[15:11] == BPL)   /* BPL */
            begin
              if(reg1[15] != 1'b1)
                pc <= dbus;
              stat <= `IF;
            end

          else if(   ir[15:11] == JMP
                  || ir[15:11] == JR  )   /* JUMP/JR */
            begin
              pc <= dbus;
              stat <= `IF;
            end

          else if(ir[15:11] == JAL)   /* JAL */
            begin
              stat <= `EX2;
            end


        end // case: `EX

       `EX2:                    /**** EX2 ****/
        begin
          if(ir[4:0] == LD || ir[4:0] == ST)       /* LD/ST*/
            begin
              stat <= `IF;
            end
          else if(ir[15:11] == JAL)                /* JAL */
            begin
              pc <= dbus;
              stat <= `IF;
            end
        end
      endcase // case(stat)

    end // else: !if(!RST)
end // always @ (posedge CLK or negedge RST)
```

```
  mux2_1 s1 (.out(sbus1),
             .a(pc),
             .b(reg1),
             .sel(sel1));

  mux5_1 s2 (.out(sbus2),
             .a(16'h0002),
             .b(reg2),
             .c({ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7],ir[7:0]}),
             .d({8'b0,ir[7:0]}),
             .e({ir[7:0],8'b0}),
             .f({ir[10],ir[10],ir[10],ir[10],ir[10],ir[10:0]}),
             .sel(sel2));

  mux2_1 d  (.out(dbus),
             .a(aluout),
             .b(datain),
             .sel(sel3));

  alu16 alu (.ina(sbus1),
             .inb(sbus2),
             .com(com),
             .y(aluout));

  reg_file rf (.addr1(addr1),
               .addr2(ir[7:5]),
               .din(dbus),
               .dout1(dout1),
               .dout2(dout2),
               .req(req),
               .CLK(CLK),
               .RST(RST));

endmodule // bpico
```

```
/*-----------------------------------------------mux2_1-----------------------------------------------*/
module mux2_1(out,a,b,sel);

  input [15:0] a,b;
  input sel;
  output [15:0] out;

assign out = (sel == 0)? a:
         (sel == 1)? b:
         'bx;

endmodule




/*-----------------------------------------------mux5_1-----------------------------------------------*/
module mux5_1(out,a,b,c,d,e,f,sel);

  input [15:0] a,b,c,d,e,f;
  input [2:0] sel;
  output [15:0] out;

  assign out = (sel == 0)? a:
               (sel == 1)? b:
               (sel == 2)? c:
               (sel == 3)? d:
               (sel == 4)? e:
               (sel == 5)? f:
               'bx;

endmodule




/*-----------------------------------------------regster file-----------------------------------------------*/

module reg_file (addr1, addr2, din, dout1, dout2, req ,CLK ,RST);

  input[2:0] addr1, addr2;
  input [15:0] din;
  input        req, CLK, RST;
  output [15:0] dout1, dout2;

  reg [15:0]     register0, register1, register2, register3, register4, register5,
register6, register7;
```

```
  // read req=0
  assign        dout1 = (addr1 == 3'b000 && req == 0)? register0:
                        (addr1 == 3'b001 && req == 0)? register1:
                        (addr1 == 3'b010 && req == 0)? register2:
                        (addr1 == 3'b011 && req == 0)? register3:
                        (addr1 == 3'b100 && req == 0)? register4:
                        (addr1 == 3'b101 && req == 0)? register5:
                        (addr1 == 3'b110 && req == 0)? register6:
                        (addr1 == 3'b111 && req == 0)? register7: 'bx;

  assign        dout2 = (addr2 == 3'b000 && req == 0)? register0:
                        (addr2 == 3'b001 && req == 0)? register1:
                        (addr2 == 3'b010 && req == 0)? register2:
                        (addr2 == 3'b011 && req == 0)? register3:
                        (addr2 == 3'b100 && req == 0)? register4:
                        (addr2 == 3'b101 && req == 0)? register5:
                        (addr2 == 3'b110 && req == 0)? register6:
                        (addr2 == 3'b111 && req == 0)? register7: 'bx;


  always @(negedge CLK or negedge RST)
    begin
     if(!RST)   // reset
        begin
          register0 = 0;
          register1 = 0;
          register2 = 0;
          register3 = 0;
          register4 = 0;
          register5 = 0;
          register6 = 0;
          register7 = 0;
        end
      else if(req)         // write req=1
        case (addr1)
          3'b000: register0 = din;
          3'b001: register1 = din;
          3'b010: register2 = din;
          3'b011: register3 = din;
          3'b100: register4 = din;
          3'b101: register5 = din;
          3'b110: register6 = din;
          3'b111: register7 = din;
        endcase

    end


endmodule
```

```
/*-------------------------------------------------ALU-------------------------------------------------*/
module alu16(ina, inb, com, y);

input[15:0] ina, inb;
input[3:0] com;
output[15:0] y;

wire[15:0] addsub;

assign y = (com == 4'b0000)? ina:
           (com == 4'b0001)? inb:
           (com == 4'b0010)? ina & inb:
           (com == 4'b0011)? ina | inb:
           (com == 4'b0100)? ina ^ inb:
           (com == 4'b0101)? ~inb:
           (com == 4'b1100)? inb << 1:
           (com == 4'b1101)? inb >> 1:
           addsub;

addsub16 as16 (.ina(ina), .inb(inb), .sub(com[0]), .sumdiff(addsub));

endmodule


/*-------------------------------------------------addsub16-------------------------------------------------*/
module addsub16 (ina, inb, sub, sumdiff);

input[15:0] ina, inb;
input sub;
output[15:0] sumdiff;

wire c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15;
wire[15:0] bb;

fadder fa0 (.a(ina[0]), .b(bb[0]), .c(sub), .so(sumdiff[0]), .co(c0));
fadder fa1 (.a(ina[1]), .b(bb[1]), .c(c0), .so(sumdiff[1]), .co(c1));
fadder fa2 (.a(ina[2]), .b(bb[2]), .c(c1), .so(sumdiff[2]), .co(c2));
fadder fa3 (.a(ina[3]), .b(bb[3]), .c(c2), .so(sumdiff[3]), .co(c3));
fadder fa4 (.a(ina[4]), .b(bb[4]), .c(c3), .so(sumdiff[4]), .co(c4));
fadder fa5 (.a(ina[5]), .b(bb[5]), .c(c4), .so(sumdiff[5]), .co(c5));
fadder fa6 (.a(ina[6]), .b(bb[6]), .c(c5), .so(sumdiff[6]), .co(c6));
fadder fa7 (.a(ina[7]), .b(bb[7]), .c(c6), .so(sumdiff[7]), .co(c7));
fadder fa8 (.a(ina[8]), .b(bb[8]), .c(c7), .so(sumdiff[8]), .co(c8));
fadder fa9 (.a(ina[9]), .b(bb[9]), .c(c8), .so(sumdiff[9]), .co(c9));
fadder fa10(.a(ina[10]), .b(bb[10]), .c(c9), .so(sumdiff[10]), .co(c10));
fadder fa11(.a(ina[11]), .b(bb[11]), .c(c10), .so(sumdiff[11]), .co(c11));
fadder fa12(.a(ina[12]), .b(bb[12]), .c(c11), .so(sumdiff[12]), .co(c12));
fadder fa13(.a(ina[13]), .b(bb[13]), .c(c12), .so(sumdiff[13]), .co(c13));
fadder fa14(.a(ina[14]), .b(bb[14]), .c(c13), .so(sumdiff[14]), .co(c14));
fadder fa15(.a(ina[15]), .b(bb[15]), .c(c14), .so(sumdiff[15]), .co(c15));
```

```
assign bb = (sub)?  ~inb:
            (~sub)?  inb: 4'bz;

endmodule


/*-------------------------------------------------full adder-----------------------------------------------*/

module fadder (a,b,c,so,co);
input a,b,c;
output so,co;

assign co = (a&b)|(b&c)|(a&c);
assign so = (a&~b&~c)|(~a&b&~c)|(~a&~b&c)|(a&b&c);

endmodule


/*-------------------------------------------------test bench file-----------------------------------------------*/
module bpicosim;

  reg CLK, RST;

//memory
  wire [15:0] memdout, memdin;
  wire [7:0]  memaddin;
  wire        write;
  reg [7:0]   mem [0:200];

  assign       memdout = (!write) ? {mem[memaddin & 8'b11111110], mem[memaddin |
8'b00000001]}:16'bx;

  always @(negedge CLK)
    begin
      if(write)
        begin
          mem[memaddin & 8'b11111110] <= memdin[15:8];
          mem[memaddin | 8'b00000001] <= memdin[7:0];
        end
    end

//test define
wire [15:0]    ir, pc, mdr, reg1 ,reg2;
wire [7:0]     mar;
wire [2:0]     stat;
wire           sel1, sel3;
wire [2:0]     sel2;
wire           req;
wire [3:0]     com;
wire [15:0]    dbus, sbus1, sbus2, dout1, dout2, aluout;
```

```
//test define end
  bpico pico(.datain(memdout),
             .addressout(memaddin),
             .dataout(memdin),
             .write(write),
             .CLK(CLK),
             .RST(RST),
//test pin
.pc(pc),.mar(mar),.mdr(mdr),.ir(ir),.reg1(reg1),.reg2(reg2),.sel1(sel1),.sel2(sel2),.
sel3(sel3),.com(com),.req(req),.stat(stat),.dbus(dbus),.sbus1(sbus1),.sbus2(sbus2),.d
out1(dout1),.dout2(dout2),.aluout(aluout)
//test pin end
);

  always #200 CLK =~CLK;

  initial
    begin

      $readmemb("memdata.txt",mem);

      CLK <= 0;

       RST <= 1;
      #410 RST <= 0;
      #100 RST <= 1;

      #40000 $stop;
    end

endmodule
```