卒 業 研 究 報 告

題　　目

32 ビットRISC プロセッサの設計

指 導 教 員

矢野 政顯　教授

報 告 者

松見 隆之

平成　14 年　2 月　7 日

高知工科大学 電子・光システム工学科

目次

第１章　はじめに

今日、コンピュータだけではなくほとんどの情報端末にプロセッサが搭載されている。プロセッサの中でも RISC(Reduced Instruction Set Computer)アーキテクチャを採用したものは、ワークステーション、ゲーム機等々さまざまな製品に広く使用されている。また、ほとんどのプロセッサがこの RISC 方式を採用し、RISC はプロセッサの標準ともいえる存在になっている。RISC は従来の CISC(Complex Instruction Set Computer)と異なり、回路を簡単にし、コンパイラを通して高級プログラム言語との親和性が高まるように設計され、またパイプライン方式およびロード＆ストア方式を採用することで処理効率を高めている。大学院で研究に必要な中央処理装置（CPU(Central Processing Unit)）技術習得のため、卒業研究として、RISC アーキテクチャに基づいたプロセッサの設計を行った。本論文の第２章では RISC アーキテクチャの概要について、第３章には卒業研究で取り上げたプロセッサの設計について、第４章では設計したプロセッサの VHDL(Very High Speed Integrated Circuit)記述およびテストについて述べるとともに、第５章では本研究によって得られた成果を要約し、今後に残る課題に言及する。

第２章　RISC アーキテクチャ

2.1 RISC アーキテクチャとは

RISC 登場までコンピュータアーキテクチャは CISC という複雑なアーキテクチャによって占められていた。この CISC アーキテクチャは細かいレベルでの演算が可能であり、高級言語および OS(Operating System)を強力に支援する。そして回路規模が上昇し、相次ぐ機能追加でますます複雑になっていた。しかしながら CISC はその構造上常にプロセッサ内部速度から比べると遅いメインメモリとアクセスするだけではなく回路が大変複雑で回路規模が大きいため高速化に向かなかった。それと比較して RISC は統計的な分析を元にソフトウェア、ハードウェアの機能分担が明確になっている。RISC におけるハードウェアの制御は、CISC がマイクロプログラムを利用しているのと違い、すべてコンパイラから行う。理由としては近年コンピュータ用の言語が複雑になっている。それだけではなく半導体メモリの技術が進歩し、マイクロプログラム用 ROM(Read Onry Memory)ではなく、高速の RAM(Randam Access Memory)の実装が可能になったことからソフトウェアからの直接制御が可能になったためである。

RISC(Reduced Instruction Set Computer)はその名のとおり簡易命令セットコンピュータである。上にも述べたがコンパイラから吐き出されるコードを統計的に分析し、それをもとに必要のない命令を削ることによって回路を単純にしている。また、RISC は１クロック１命令を目標に設計され、そのためにパイプライン等のさまざまな方式を導入している。

2.2 パイプライン方式

命令の実行に際してある一定の時間が必要である。これを減らす手法としてパイプラインが上げられる。パイプラインとは命令をいくつかの部分に分割して、順番に多数の命令が実行される場合に、部分ごとに複数の命令を同時に実行する方式である。　例として図 2.1 に示すような

５ステージのパイプラインがあげられる。この場合、命令実行に必要なサイクル数は５サイクルである。

図2.1 命令パイプライン

第１サイクル（IF(Instruction Fetch)）は命令のフェッチ部である。プログラムカウンタで指示されたアドレスに格納されている命令を読み取る。第２サイクル（RR(Register Read)）はレジスタリード部である。フェッチした命令の中に含まれるアドレスから対象のデータを読み出す。第３サイクル（EX(Execute)）は演算部である。ここでは読み出されたデータと命令を元にALU(Arithmetic Logic Unit)で演算を行う。第４サイクル(MA(Memory Access)はメモリアクセス部である。ここではメモリへの書込み、読み出しを行う。最後のサイクル（RW(Register Write)）では演算結果や、メモリより読み出したデータをレジスタに格納する。

このように処理を５分割にすることで、そのまま実行すれば５サイクル必要な命令を、１クロックあたり１命令の割合で実行できることになる。ベルトコンベアーのように各部分で処理を分担し専念させることで、総合的に見て同時に５個の命令を実行できることになる。パイプラインを採用していない場合、１命令に５サイクル必要な命令では処理が非常に低速である。

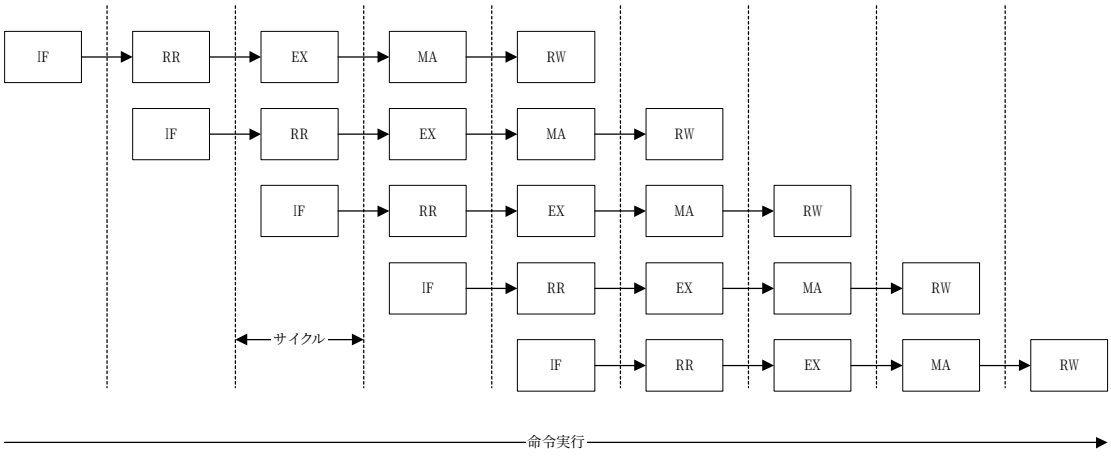


図2.2 命令パイプライン

図 2.2 に示すように５段の深さのパイプラインを使用することによって命令サイクルを減らすことができる。しかし、直前の命令の演算結果に応じた条件分岐が必要な場合、パイプラインの命令実行を遅らせなければならない。例えばに図 2.3 のような４段パイプラインの場合、以下のようになる。

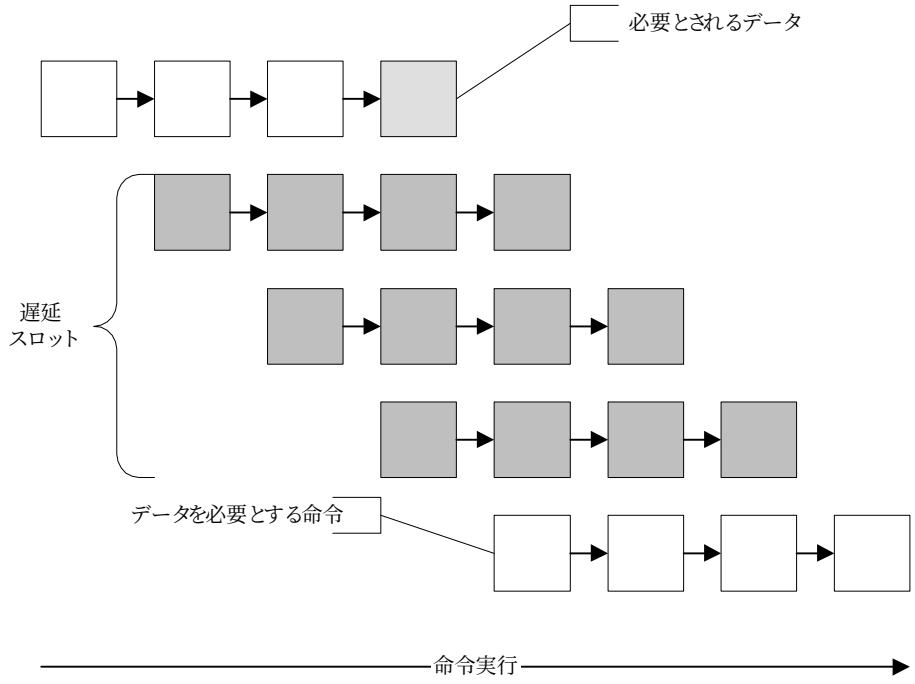


図2.3 遅延スロット

この例では、最初の命令の第 4 ステージで、次に実行される命令のデータを格納するが、次の命令の開始時点では最初の命令の結果が格納されていないので実行できない。このため要求されるデータが使用可能になるまで遅延スロット（NOP 命令）を次の命令との間に入れなければならない。したがってパイプラインによってサイクル数を減らしても、このような場合にはその利点が生かされない。このためパイプライン管理ではパイプラインの流れを妨害する可能性のある命令（Branch 命令、Jump 命令）をコンパイラによって正しく効率よく管理する必要である。上記の例では、命令長および処理時間を一定としていたが、仮にそれらが可変であった場合には、パイプラインにおける効率のよいスケジュール管理はほぼ不可能である。このため、RISC において１サイクル１命令を実現するためには必ず命令および部分ごとの命令実行時間は一定である必要がある。

2.3 ロード&ストア方式

CISC において処理速度を高速化できなかった大きな理由は 2.1 節で述べたとおりメモリアクセス速度である。メインメモリへのアクセスは内部メモリと比べて速度が遅く余分なサイクルが使われるため RISC ではロード&ストアアーキテクチャを採用し、命令実行の結果は必ずレジスタに格納ように設計されている。メインメモリへのアクセスはメモリアクセス命令（Load 命令、Store 命令）によってのみ行われる。それにより、1 命令ごとにメインメモリにアクセスする必要性がなくなるため、パフォーマンスが向上する。

表 2.1　RISC と CISC の違い

	RISC	CISC
メモリアクセス	専用命令のみ	大半の命令
レジスタ	多数の汎用レジスタ	用途固定のレジスタ
演算結果格納	レジスタ	メモリ

2.4 RISC の命令形式

RISC と CISC の大きな違いは命令長の違いであることは上で述べたとおりである。RISC では固定幅の命令長を実現するために多数ある命令形式のうちから 3 種類の命令形式だけを採用している。第 1 は主に演算等を高速に行うための R 形式（Register 形式）、第 2 は直接データ入力の可能な演算形式であり、且つメモリの制御が可能な I 形式（Immediate 形式）、第 3 は命令のジャンプのために使われる J 形式（Jump 形式）である。

2.4.1 R 形式

R 形式の命令は 3 つのオペランドを持っており、これらのオペランド間で演算を行う。ただし、3 つのオペランド全てが直接演算可能な値ではないのでオペランドを数値に変換する必要がある。変換方法については

後に述する。それぞれのオペランドは各々名前をもっており命令コードの 25 ビット～21 ビット目までをソースレジスタ（RS(Source Register)）、20 ビット～16 ビット目までをターゲットレジスタ（RT(Target Register)）、15 ビット～11 ビット目までをデスティネーションレジスタ（RD(Destination Register)）という。機能としてはソースレジスタおよびターゲットレジスタがそれぞれ演算する値、演算される値の格納場所を指示し、デスティネーションレジスタは演算された結果を格納する場所を示している。R 形式命令のオペランド以外のビットは 31 ビット～26 ビット目がオペレーションコード(Operation Code)、10 ビット～６ビット目がシフト数(Shamt)、５ビット～０ビット目が ALU 機能コード(ALU Function Code)を表している。通常 R 形式の命令のオペレーションコードおよびシフト数はすべて０であり、どのように演算するかの判別は ALU 機能コードをもとに行われる。

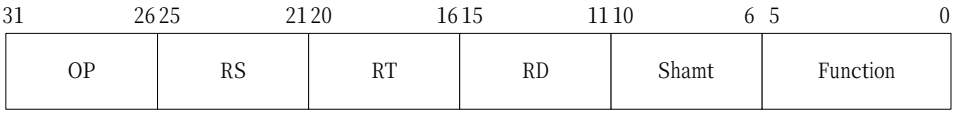


図2.4　　R形式ビット配列

2.4.2 Ｉ形式

Ｉ形式の命令は２つのオペランドおよび１つの Immediate と呼ばれる数を持っている。２つのオペランドは R 形式と同じく直接演算可能な数ではないので、データー格納場所をアドレスに変換する必要がある。また、Immediate は直接演算可能な数であるが、RISC において演算およびデータは常に 32 ビットに固定されているため、16 ビットの Immediate も 32 ビットに変換する必要がある。
命令の 25 ビット～21 ビット目をソースレジスタ（RS）、20 ビット～16 ビット目をターゲットレジスタ（RT）、15 ビット～０ビット目を Immediate といい、RS は演算される数として、Immediate は演算する数として主に機能する。また、ターゲットレジスタは R 形式のデスティネーションと同じく演算結果を格納する場所である。オペランド使用法の例外として Branch 命令、Store 命令がある。Operation Code の位置は R 命令と同じで、どのような演算を行うかの判別は R 形式と異な

り Operation Code を元に行う。

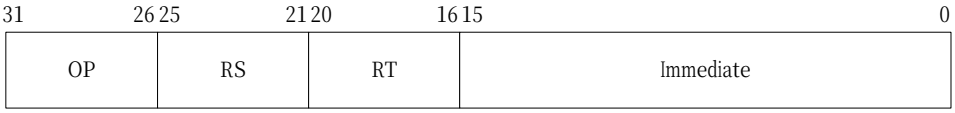


図2.5　　I形式ビット配列

2.4.3 J 形式

J 形式は他の形式と大きく異なり演算を実行しない。そのためオペランドは無く、26 ビットの Address と Operation Code のみである。RISC におけるメモリアドレスは 32 ビットのため、この Address はこのままは使用できない。このため 32 ビットに拡張される。拡張後のアドレスをジャンプ先アドレスとしてジャンプするものである。

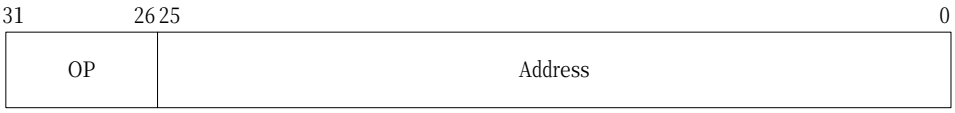


図2.6　　J形式ビット配列

2.5 アドレッシングモード

RISC の命令長は 32 ビットに固定されているため、32 ビットのオペランドを直接扱うことはできない。そのため RISC では以下の４通りのアクセス方式を採用している。

2.5.1 小さな定数のオペランド

実際のプログラムでは定数を使用した演算は比較的多く使用されている。C 言語での FOR 文や While 文、IF 文では小さな定数を扱う場合が

ほとんどである。実際にプログラムを組む場合このような命令を多用するので、ますます小さな定数を扱う機会は増える。そのため RISC では Immediate という方法で小さな定数を直接扱えるようにしている。もちろん RISC での命令長や数値幅は 32 ビット固定なので 16 ビットの Immediate を 32 ビットに拡張して使用する。

例えばに 10 進数で-550 という値を RISC で扱う場合には次のようになる。

$-550[10]=1111110111011010_2$

なので Immediate は

$Immediate=1111110111011010$

となる。これに 16 ビット分の０を付加し、

$DATA=00000000000000001111110111011010$

とすると、この 32 ビットの値は、10 進数に直すと

$00000000000000001111110111011010_2=64986_{10}$

となり、正しくない。これは負の値に２の補数表現を用いているためである。２の補数表現に対応させるために単なる拡張ではなく符号拡張を使用しなければならない。符号拡張は次のプロセスで行う。

① Immediate の先頭１ビットを見る
② 先頭１ビットを 16 ビット分結合

上記の例の場合-550_{10}なので符号拡張を行うと

$DATA=11111111111111111111110111011010$

となる。

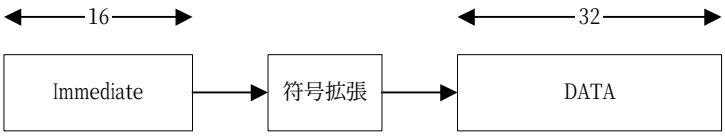


図2.7 符号拡張

2.5.2 分岐命令用のオペランド

プログラム中での分岐命令はその分岐先が比較的近くにかかれている場合が多い。そのため、分岐命令でのアドレッシングでは PC 相対と呼ばれるものを使用し、PC の値と Immediate との和をとることで、効率よ

く分岐できるようにしている。

① Immediate の値を符号拡張する
② PC の値と拡張後の Immediate の値との和をとる
③ 結果をアドレスとする

仮に現在プログラムカウンタと Immediate の値が
$PC=00000000000000000000000111110100_2=500_{10}$
$Immediate=0000001111101000_2=1000_{10}$
の場合、必要なアドレスは
$Address=00000000000000000000010111011100_2$
となる。

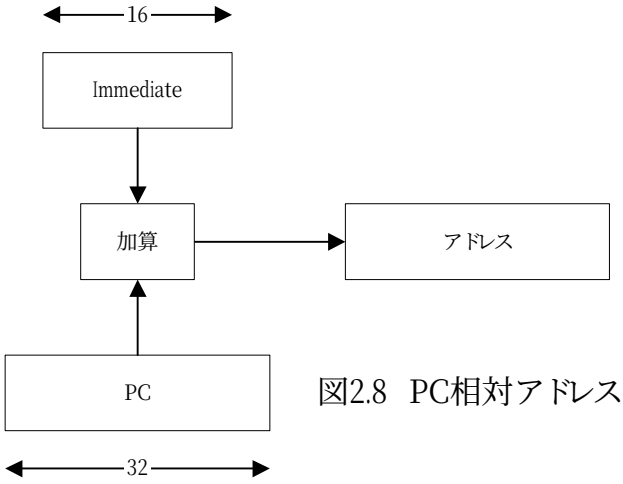


図2.8 PC相対アドレス

命令におけるこのようなアドレッシングは PC 相対アドレッシングとよばれる。

2.5.3 ジャンプ命令用のオペランド

ジャンプ命令においてもジャンプ先はある程度近くに置かれている場合が多い。分岐命令では分岐条件を指定するための Immediate が 16 ビットに制限されていたが、ジャンプ命令では、特に条件を指定する必要が無いため、入力できる DATA は 26 ビットである。
例を 2.5.2 節と同じとした場合ジャンプ先アドレスは

$Address=00000000000000000000001111101000_2$

となる。

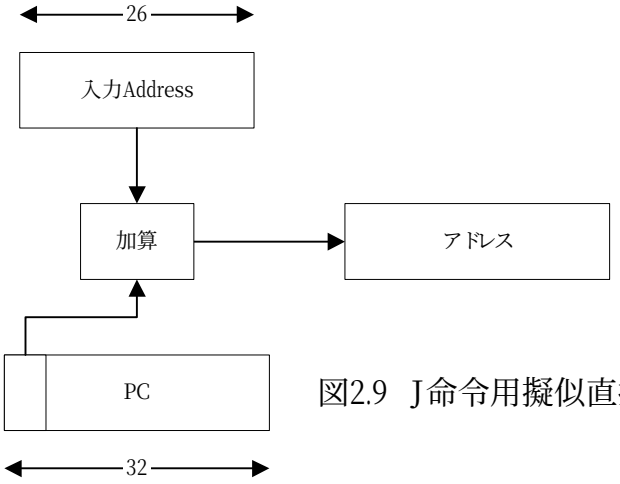


図2.9　J命令用擬似直接アドレス

2.5.4 R命令のオペランド

R形式命令のデータ方式はレジスタ・アドレッシングといわれる。命令の中の５ビットで指定されたレジスタの中にあるデータを読み出し、それを演算に使う。

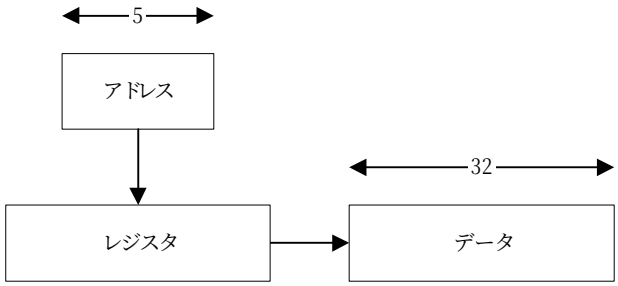


図2.10　レジスタアドレッシング

2.6 データ形式

RISCにおいて、取り扱い可能なデータ形式は32ビットワード、16ビットハーフワード、8ビットバイトである。全てのメモリは32ビットで扱われるために、バイトの取り扱いには一定の決まりがある。１つはビッグエンディアン、もう一つがリトルエンディアンである。

●ビッグエンディアンを使用する場合。

システムを全てビッグエンディアンで構成する場合には、バイト番号０は必ず 32 ビットの最上位である。32 ビット目から８ビット区切りにバイト番号０～３と割り振る。この方式をとる代表的な CPU にはモトローラ社の 68000 が挙げられる。

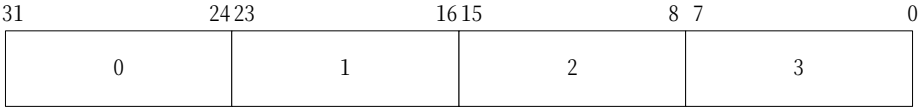


図2.11　ビッグエンディアン構成

●リトルエンディアンを使用する場合。

システムをリトルエンディアンで構成する場合、バイト番号０はビッグエンディアンと異なり最下位バイトである。32 ビット目から８ビット区切りにバイト番号３～０が割り振られる。この方式を採用している CPU としては X86 や DEC 社製品が挙げられる。

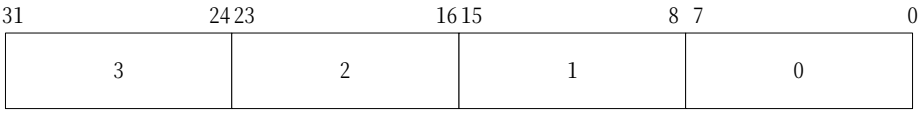


図2.12　リトルエンディアン構成

2.7 高度メモリ管理システム

RISC には TLB(Translation Lookaside Buffer)）（アドレス変換バッファ）を採用したメモリ管理ユニットを内蔵し、物理アドレスでは実現不可能な大容量の仮想アドレスとの変換を高速に行っている。通常、メモリにアクセスするためには、一度メモリにアクセスしてその情報を取得し、それをもとにデータアクセスを行う。このままではレジスタに比べて低速なメモリアクセス時間が２倍になってしまい非常に非効率的である。そのため、メモリアドレス変換を記録しておく専用のメモリ（TLB）が搭載されている。

32 ビットで構成された仮想アドレス空間形式では、上位 20 ビットは仮想ページ番号として動作し、下位 12 ビットは TLB に渡されて物理アドレスとの変換に使用される。仮想ページ番号のうちの上位３ビットをメモリのモード指定に使用する。

2.7.1 ユーザーモードでのアドレス指定

RISC には特権状態としてカーネルモードとユーザーモードがある。通常はユーザーモードで処理されているが、例外が発生した場合のプロセッサはカーネルモードで動作する。

ユーザーモードで動作している場合には、取り扱い可能な仮想メモリ空間として、2G バイトの空間が用意されている。アドレス指定は先頭ビットが常に０であるほかは通常のアドレス指定と同じである。このため、扱えるアドレスは 0000000~7FFFFFF である。TLB を使ってこの空間をどちらのモードからでも同じようにマッピングできる。この空間はデータ格納のみに使われる。

2.7.2 カーネルモードでのアドレス指定

プロセッサが例外を起こしカーネルモードで動作している場合、プロセッサはユーザーモードのメモリ空間以外に３種類のメモリ空間を使用できる。

KSEG0：このアドレス空間は 512M で TLB を使用しない。またこの空間へのアクセスにはキャッシュが用いられる。命令実行コードの一部とそのデータを格納するのに主に使用される。

KSEG1：このアドレス空間は 512M で TLB を使用しない。KSEG0 と違う点はキャッシュを使用しないことである。用途としてはディスクバッファや ROM として使用される。

KSEG2；このアドレス空間は 1024M である。他の空間と異なりこの空間では TLB を使用し、仮想メモリを任意の物理アドレスにマッピングできる。主な用途は、メモリマップやスタックである。

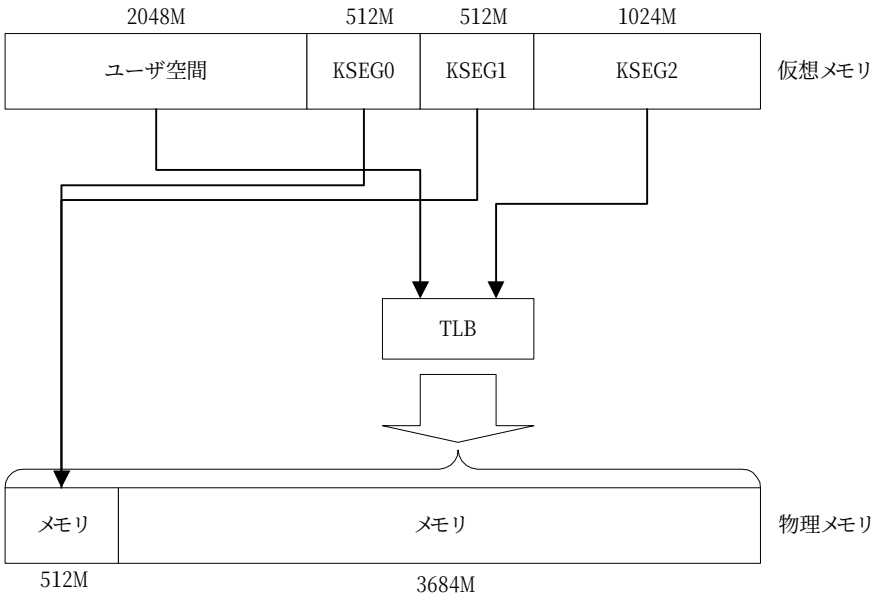


図2.13　　メモリマップ

2.8 スーパースカラ方式

コンピュータの性能を上げていく手法として、パイプライン方式は既に述べたとおりである。この他の方法としてコンピュータ内部の構成を多重化して、一度に処理できる命令数を増やす手法がある。この方式はスーパースカラ方式と呼ばれている。RISC ではこの方法を実現し、CPU によってさまざまな多重化を施している。仮に、2 重スーパースカラを考えるとする。RISC 命令の中でもっとも頻発し、かつもっとも時間のかかる命令は Load 命令、Store 命令である。この命令をもう一つのパイプラインで実行することで圧倒的に効率を上げられる。

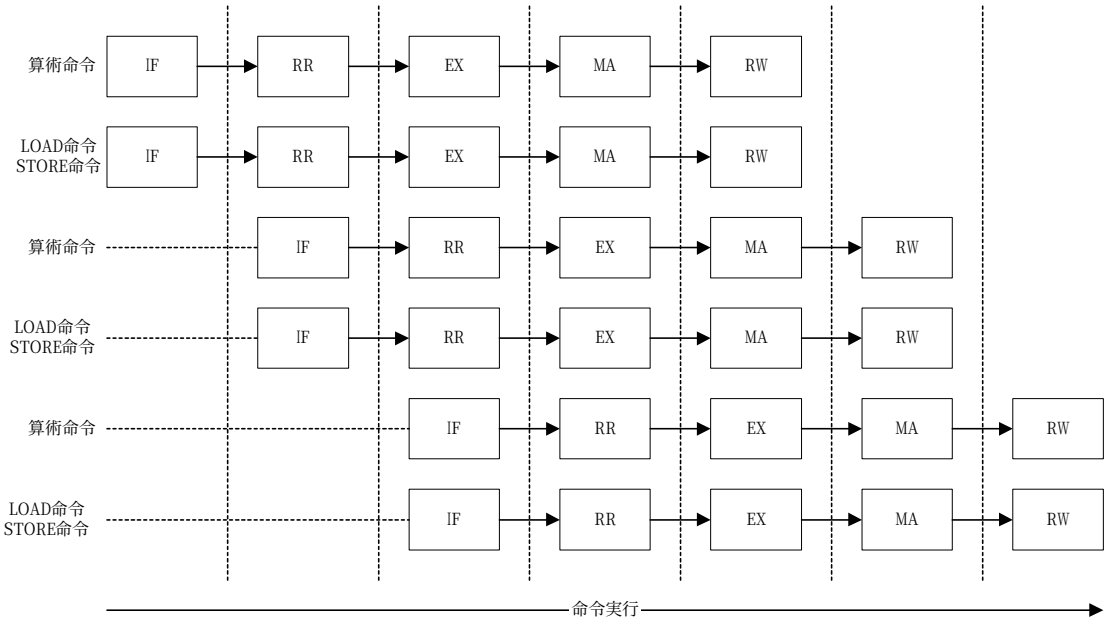


図2.14　　スーパースカラのパイプライン

2.9 動的パイプラインスケジューリング

動的パイプラインスケジューリングでは、ストールする場合、ストール解消までの処理中にその先の命令より実行可能な命令を選び出し、実行する。そのためにはパイプラインを機能ごとに分割しなければならない。各ユニットにはリザベーションバッファを持ち、ここで、命令操作とオペランドを保持する。このバッファに全てのデータが整い、実行可能となれば命令が実行され、確定ユニットを通して結果をストアする。普通はパイプライン通して命令を順序どおりに実行しメモリに書き込む。この方式をイン・オーダー完了という。一方、機能ユニットは命令処理を好きなときに開始、終了させて良いのでこれを許すやり方としてアウト・オブ・オーダー完了がある。ただし、この方法の場合には割り込みが不正確になってしまうという欠点がある。またこの動的パイプライン処理というのは従来の静的パイプラインと比べて圧倒的に設計が難しい。これは、動的なスケジューリングは分岐予測と併用して使用される場合が多いためである。もちろん分岐予測なので、予測が外れた場合には全ての実行ユニットで分岐命令以降に実行されていた命令の結果を破棄しなければならない。この方法はスーパースカラを伴う場合が多いので場合によっては結果が確定してしまっている可能性があるからである。こ

のように併用して演算を行う方法を投機的実行という。

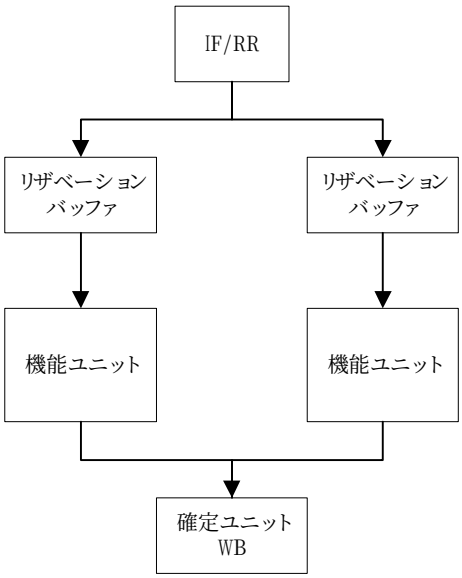


図2.15 動的スケジューリングにおける機能ブロック

2.10 例外

プロセッサはさまざまな命令を処理するため予期しないところで予想外の動作を起こす場合がある。それを例外としてプロセッサでは処理を行う。まず、正常な命令の流れに割り込みをかけ、例外を起こした命令を異常終了させる。それと同時にパイプラインで処理されていた命令もすべて異常終了させ、プロセッサは例外処理プログラムへジャンプする。

●アドレスエラー例外

この例外は、データアドレスのワード境界がずれた状態で Load、Store または命令をフェッチで発生する。メモリアドレスのワードはすべてアドレスの最下位２ビットが 00 でなければならない。この例外は致命的例外である。

●ブレークポイント例外

この例外はプロセッサがブレーク命令を実行した場合に発生する。この例外が発生した場合には該当するシステムルーチンに制御を渡す。

●バスエラー例外

この例外は CPU に対して外部からのエラー入力があった場合に発生する。例としてバスがタイムアウトした場合、パリティエラーがあった場

合があげられる。

●コプロセッサ使用不可能例外

プロセッサがコプロセッサを利用をしようとした際、コプロセッサが利用可能でない場合に発生する。

●割り込み例外

この例外はハードウェア若しくはソフトウェアからの割り込みがあった場合に発生する。

●オーバーフロー例外

この例外はもっとも基本的な例外で、算術命令を実行した際、実行中にオーバーフローした場合に発生する。例外処理ルーチンで適切に処理された後、カーネルは実行プロセスに対してオーバーフローエラーを通知する。

●予約命令例外

この例外は命令が存在しない命令だった場合、すなわち、Operation Code および ALU Function Code に該当するコードが無かった場合に発生する。例外処理ルーチンは実行プロセスに対して違法命令を通知する。

●リセット例外

この例外はプロセッサの入力信号に RESET が入った場合に発生する。この例外の操作としてはすべてのレジスタ、コプロセッサ、キャッシュ、およびメモリを初期化し、その後診断処理等が行われオペレーティングシステムに移る。

●システム呼び出し例外

この例外は SYSCALL 命令特有のもので、適切な例外処理の後、オペレーティングシステムは該当するプロセスに制御を渡す。

●TLB に関する例外

この例外は仮想ページ番号が TLB のどれにもヒットしなかった場合や、ヒットしても Valid,Dirty 等の信号が出ている場合に発生する。

2.11 R3000 の命令コード

RISC アーキテクチャをとっている CPU である R3000 の命令には次の命令が存在する。[1]

- ADD 命令　　加算命令
- ADDI 命令　　Immediate 型加算命令

- ADDIU 命令　　Immediate 型無符号加算命令
- ADDU 命令　　無符号加算命令
- AND 命令　　論理積命令
- ANDI 命令　　Immediate 型論理積命令
- BCZF 命令　　コプロセッサ Z が False 分岐
- BCZT 命令　　コプロセッサ Z が True 分岐
- BEQ 命令　　等号成立分岐命令
- BGEZ 命令　　ゼロ以上分岐
- BGEZAL 命令　　状態保存ゼロ以上分岐
- BGTZ 命令　　正の数分岐
- BLEZ 命令　　ゼロ以下分岐
- BLTZ 命令　　負の数分岐
- BLTZAL 命令　　状態保存負の数分岐
- BNE 命令　　不等号分岐
- BREAK 命令　　ブレーク
- CFCZ 命令　　コプロセッサからの制御移動
- COPZ 命令　　コプロセッサ命令
- CTCZ 命令　　コプロセッサに制御移動
- DIV 命令　　除算
- DIVU 命令　　無符号除算命令
- J 命令　　ジャンプ
- JAL 命令　　状態保存ジャンプ
- JALR 命令　　状態保存レジスタジャンプ
- JR 命令　　レジスタジャンプ
- LB 命令　　バイトロード
- LBU 命令　　無符号バイトロード
- LH 命令　　ハーフワードロード
- LHU 命令　　無符号ハーフワードロード
- LUI 命令　　上位に Immediate ロード
- LW 命令　　ワードロード
- LWCZ 命令　　コプロセッサワードロード
- LWL 命令　　ワードの左ロード
- LWR 命令　　ワードの右ロード
- MFC0 命令　　システム制御コプロセッサからの転送
- MFCZ 命令　　コプロセッサからの転送
- MFHI 命令　　HI レジスタからの転送

- MFLO 命令　　LO レジスタからの転送
- MTC0 命令　　システム制御コプロセッサへの転送
- MTCZ 命令　　コプロセッサへの転送
- MTHI 命令　　HI レジスタへの転送
- MTLO 命令　　LO レジスタへの転送
- MULT 命令　　乗算
- MULTU 命令　　無符号乗算
- NOR 命令　　否定論理和
- OR 命令　　論理和
- ORI 命令　　Immediate 論理和
- RFE 命令　　例外からの復元
- SB 命令　　バイト格納
- SH 命令　　ハーフワード格納
- SLL 命令　　論理左シフト
- SLLV 命令　　論理変数左シフト
- SLT 命令　　未満セット
- SLTI 命令　　Immediate 未満セット
- SLTIU 命令　　Immediate 無符号未満セット
- SLTU 命令　　無符号未満セット
- SRA 命令　　算術右シフト
- SRAV 命令　　算術変数右シフト
- SRL 命令　　論理右シフト
- SRLV 命令　　論理変数右シフト
- SUB 命令　　減算
- SUBU 命令　　無符号減算
- SW 命令　　ワード格納
- SWCZ 命令　　コプロセッサワード格納
- SWL 命令　　ワード左方向格納
- SWR 命令　　ワード右格納
- SYSCALL 命令　　システムコール
- TLBP 命令　　TLB エントリ一致
- TLBR 命令　　TLB 読み取り
- TLBWI 命令　　TLB 書込み
- TLBWR 命令　　ランダム TLB 読み込み
- XOR 命令　　排他的論理和
- XORI 命令　　Immediate 排他的論理和

第３章　プロセッサの設計

3.1　命令の選別

3.1.1 プロセッサ要求仕様の決定

本研究で設計したプロセッサの仕様は次のとおりである。

方式	MIPS-RISC 互換 Load&Store 方式
命令長	固定 32Bit
データ長	固定 32Bit
命令形式	R 形式（3 オペランド方式）
	I 形式（2 オペランド Immediate 方式）
	J 形式（Jump 命令専用方式）
レジスタ数	32Bit 汎用レジスタ 32 個
パイプライン	５段パイプライン
	フェッチステージ　：命令の読み込みを行う
	命令デコードステージ　：レジスタアクセスを行う
	命令実行ステージ　：演算を行う
	メモリステージ　：メモリへのアクセスを行う
	格納ステージ　：レジスタへの格納を行う
演算能力	基本算術演算（加算、減算）
	基本論理演算（AND、OR、XOR、NOR）
	大小比較演算
命令処理能力	１クロック当り１命令
演算方式	単一クロックサイクル方式

まず、RISC 最大の特徴であるパイプライン処理および１クロック当り１命令の演算を実現することにした。回路が大規模になり、限られた時間の内で実現不可能と思われた TLB や浮動小数点演算ユニットの採用は断念した。また浮動小数点演算ユニットを内蔵しているシステム制御コプロセッサも搭載しないこととした。システム制御コプロセッサの中には乗算、除算制御のためのユニットや、さまざまな例外処理レジスタが

含まれているが、それらも搭載しないこととした。さらに、RISC を採用するさまざまなプロセッサで使われているスーパースカラや投機的実行、およびマルチサイクル方式も採用しないものとした。

3.1.2 命令の選別

第 2 章に記述した命令のうち、3.1.1 節で不採用としたユニットにかかわる命令、および乗算、除算にかかわる命令は除外した。その上でプログラムの実行に最低限必要な命令のみを使用するものとした。具体的には gcc や Spice が吐き出す機械語の統計より必要な命令を選別した。また、この統計で一度除外された命令のうち、その後の検討で無いと不便と考えられる命令を若干追加した。GCC や Spice が吐き出す機械語の統計は表 3.1 のとおりである。[2]

表 3.1　命令出現確率（%）

命令名	GCC	SPICE
add	0	0
addi	0	0
addu	9	10
addiu	17	1
subu	0	1
and	1	0
andi	2	1
sll	5	5
srl	0	1
lui	2	6
lw	21	7
sw	12	2
lb	1	0
sb	1	0
beq	9	3
bne	8	2
jal	1	1
jr	1	1
slt	2	0
slti	1	0
sltu	1	0
sltiu	1	0

この表には命令出現確率が 0.5%を下回るものは載せてない。この表を元に命令の選別を行ったが、LB,SB(Load Byte, Store Byte)命令に関しては、LW,SW(Load Word, Store Word)で代用のできるものであるから除外した。また、無いと困ると思われる命令、NOR,XOR,SUB 命令等を付け加えて、必要な命令を決定した。採用した命令は以下のとおりである。

●加算命令	ADD 命令、ADDI 命令、ADDIU 命令、ADDU 命令
●減算命令	SUB 命令、SUBU 命令
●論理演算命令	AND 命令、ANDI 命令、OR 命令、ORI 命令
	XOR 命令、XORI 命令、NOR 命令
●大小比較命令	SLT 命令、SLTI 命令、SLTU 命令、SLTUI 命令
●ビット列操作	SLL 命令、SRL 命令、LUI 命令
●メモリ命令	LW 命令、SW 命令
●ジャンプ命令	J 命令、JAL 命令、JR 命令
●条件分岐命令	BEQ 命令、BNE 命令
●その他	NOP 命令

3.1.3 全体の概要

以上の命令を処理する本設計での全体のブロック概要図は図 3.1 のとおりである。

プログラムカウンタ：プロセッサに順番に命令をフェッチする装置
命令メモリ：命令を格納しているメモリ
レジスタ：演算するデータを格納、読み出しする装置
ALU：データを演算、ビット操作する装置
データメモリ：データを蓄積するメモリアレイ

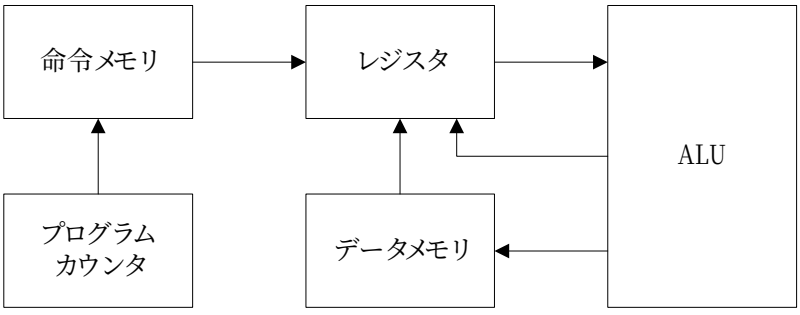


図3.1 全体ブロック図

3.2 プログラムカウンタ設計

3.2.1 基本仕様設計

プロセッサに順番に命令をフェッチする装置としてプログラムカウンタがある。プログラムカウンタは命令のアドレス、すなわち現在実行中の命令のアドレスを記憶しておく装置である。プログラムカウンタは現在保持している値を命令メモリに出力し、命令を読み出す。その後、次の命令のアドレスに自動的に更新し、次の命令を読み出す準備を整える。その他 Jump 命令等で命令のアドレスを変更する時には､変更するアドレスをプログラムカウンタの値としてセットする。そのためプログラムカウンタの入力線にはカウンタ更新アドレス、およびその制御線があり、クロックで同期して動作する。

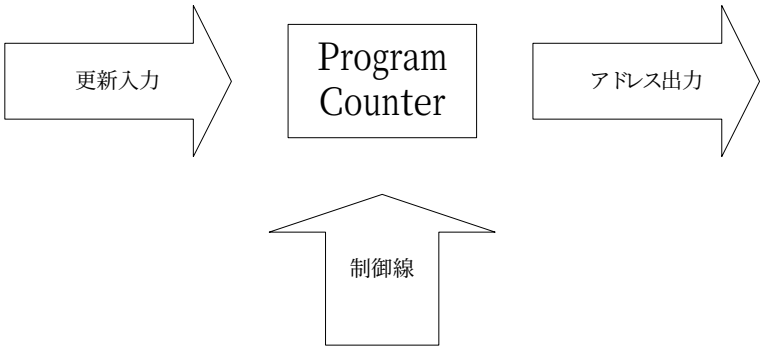


図3.2　プログラムカウンタ概要

3.2.2 入出力線

プログラムカウンタの入出力線の概要は図 3.2 のとおりで、各機能およびその詳細は以下のとおりである。

入力線　Program Counter Input　：Program Counter 更新入力
　　ALU2 よりのアドレス入力
　　Control Signal　：Program Counter 制御信号
　　Operation Code

ALU Function Code
All Zero 信号
出力線　Program Counter Output　：Instruction Memory へのアドレス出力。

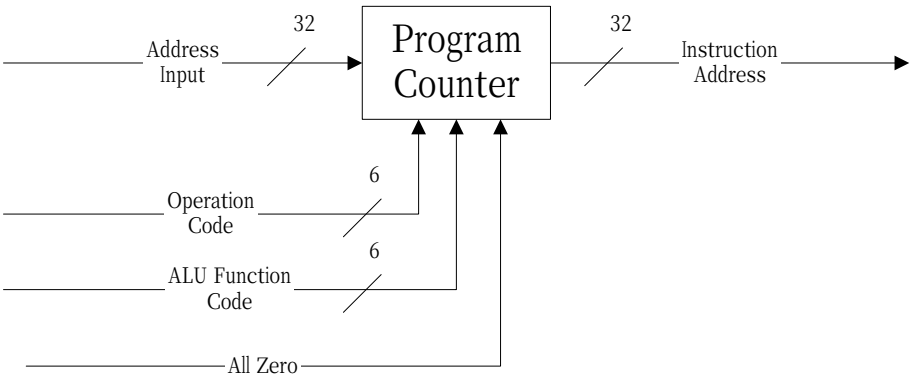


図3.3　プログラムカウンタ概要

3.2.3 動作設計

本研究の Program Counter は、通常の１ずつ増加していく Program Counter と異なり 4 ずつ増加するように設計した。理由はバイトという概念を使用できるようにしたので 32 ビットの命令長は 4 バイトを有することになったためである。そのため Instruction Memory のアドレスは下位２ビットが 00 に固定されているので 4 ずつ加算する必要がある。また、プログラムカウンタへの更新入力は、Jump 命令、Branch 命令、All Zero 制御入力より判定して更新するように設計した。

表 3.2　PC 更新を行う命令一覧

命令	形式	Operation Code						Fuction Code					
BEQ	I	0	0	0	1	0	0						
BNE	I	0	0	0	1	0	1						
J	J	0	0	0	0	1	0						
JAL	J	0	0	0	0	1	1						
JR	R	0	0	0	0	0	0	0	0	1	0	0	0

図 3.2 の命令では、プログラムカウンタの外部入力による更新を行うが、他の命令の時には通常更新をおこなう。どのような命令の場合も、パイプライン中の 4 段目すなわちメモリアクセス時に更新を行うように

する。
Branch 命令の条件判定方法は以下のとおりである。
①Branch 命令の種類を判定する。
②BEQ 命令の時には All Zero 信号が１の場合にのみプログラムカウンタを更新する。
③BNE 命令の時には ALL Zero 信号が０のときにのみプログラムカウンタを更新する。

3.2.4 ソフトウェア作成上の注意点

本研究の Program Counter の設計では Jump 命令や Branch 命令での遅延スロットの制御を一切行わないため、これらの命令の直後に Jump 命令等の Program Counter を更新する命令が入る場合、結果が意図しないものになることがある。また、Branch 命令の直後も同様で、その直後の命令には、Branch 条件とはまったく無関係の命令若しくは NOP 命令を重ねてパイプラインをストールさせなければならない。

3.3 ALU 基本仕様設計

本研究の ALU はデータ３入力、制御線入力、AllZero 出力、そしてデータ出力を備えている。また基本動作として論理演算（論理積、論理和、排他的論理和、否定論理和）、算術演算（加算、減算）、ビット列操作（シフト）が行えるように設計する。すべての命令はこの動作の組み合わせで動作するようにする。ALU に対するクロックタイミング制御は、すべてパイプラインレジスタ側の方で行う。本設計では、CPU レジスタと同じく ALU 自身に命令解釈部を含み、命令を直接読み込んで動作するものとした。

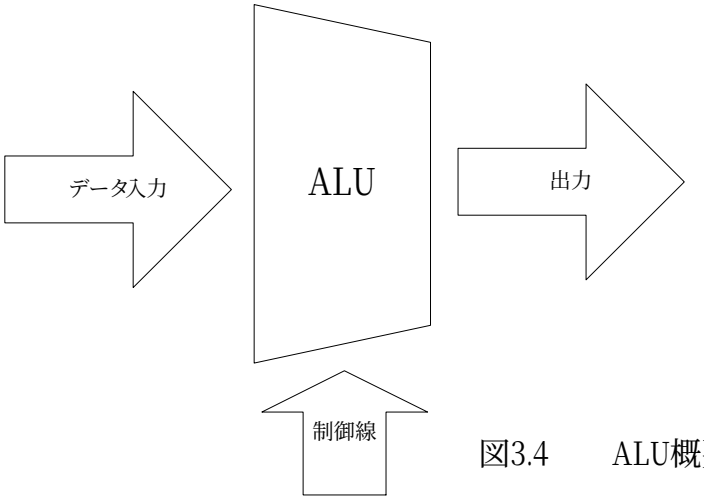


図3.4　　ALU概要

3.3.1 入出力線

ALU に対する入出力線の詳細は次のとおりである。

入力線　ALU Data Input,1 :レジスタよりのデータ入力
　　Source Register Data
　　ALU Data Input,2 :レジスタよりのデータ入力
　　Target Register Data
　　ALU Data Input,3 :符号拡張ユニットよりのデータ入力
　　Sign Extended Data
　　ALU Control Signal,1　:ALU 制御用命令入力
　　Operation Code[31~26]
　　ALU Control Signal,2　:ALU 制御用命令入力
　　ALU Function Code[5~0]
　　ALU Control Signal,3　:シフト量入力(Shamt)
　　Instruction Code[10~6]
出力線　ALU Output Data　:ALU の演算結果
　　Execution Result
　　ALU Output Signal :ALU 演算結果のゼロ判定
　　All Zero

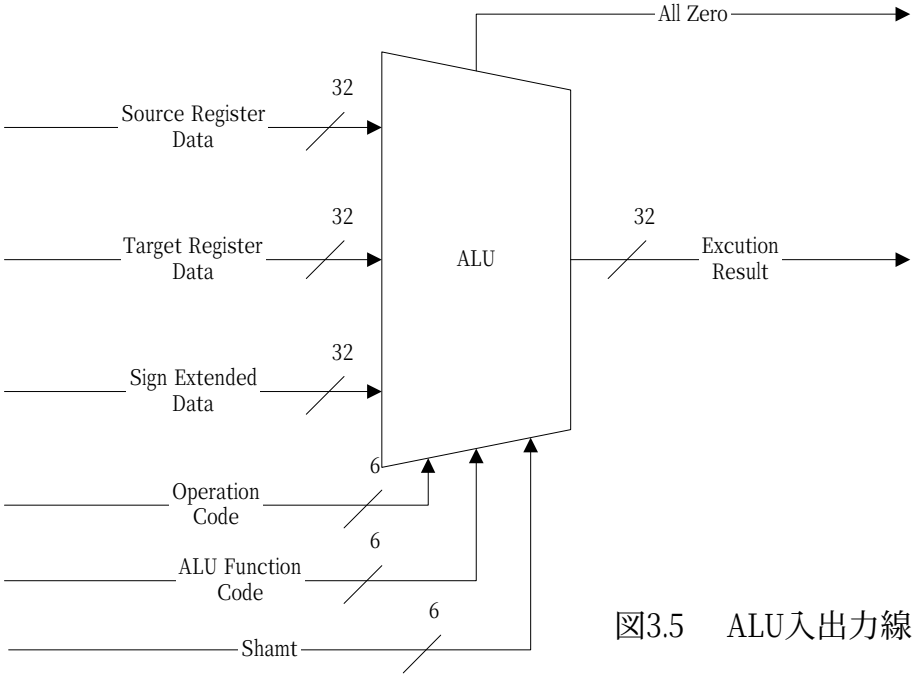


図3.5　ALU入出力線図

3.3.2 動作設計

●加算機能

ALU における基本的な動作の一つが算術加算である。２つの入力データをお互い加算しその結果を出力する。万が一、演算結果が０となる場合には命令に応じて All Zero を出力する。また RISC の加算命令では桁上げ信号の処理を行わないので 32 ビットを超えて桁上げがある場合には無視される。

Operand1 + Operand2 → Execution Result

●減算機能

加算機能のほかに算術命令で重要な働きをするのが減算機能である。本来、２の補数を使う計算の場合、減算は必要ないが、RISC では減算は主に大小判定、等号判定用として必要である。加算と同じく減算においても 32 ビットを超えて桁下がりがある場合は無視される。

Operand1 - Operand2 → Execution Result

●論理積機能
　論理演算のなかでもっとも一般的なものが論理積である。オペランドを数字としてではなくデータとして扱い、各ビットにおいて論理積をとったものを結果として出力する。

Operand1 AND Operand2 → Execution Result

●論理和機能
　論理積と同じく多用される機能として論理和がある。動作は論理積と同様である。

Operand1 OR Opernad2 → Execution Result

●否定論理和機能,排他的論理和機能
　論理演算のなかで特に特殊な動きをするのがこれらの命令である。

Operand1 NOR Operand2 → Execution Result
Operand1 XOR Operand2 → Execution Result

●ビット列操作機能
　シフト命令で使用する。シフト量は ALU シフト数入力で決まる。動作についての詳細はシフト命令の項目で記述する。

　以上の機能は命令入力に応じてオペランドを切り替え実行される。命令コードとの対応は表 3.3 のとおりである。

　表 3.3 では記述の関係上、各機能および信号を以下のように略している。
算術加算機能　→加算　　算術減算機能　→減算
論理積機能　　→AND　　論理和機能　　→OR
否定論理和機能 →NOR　　排他的論理和　→XOR
ビット列操作　→Shift　　Source Register Data　→RS
Target Register Data →RT
Sign Extended Data → SD

表 3.3　ALU のデコード

命令	形式	Operation Code						Fuction Code						Operand1	Operand2	ALU機能
ADD	R	0	0	0	0	0	0	1	0	0	0	0	0	RS	RT	加算
ADDI	I	0	0	1	0	0	0							RS	SD	加算
ADDU	R	0	0	0	0	0	0	1	0	0	0	0	1	RS	RT	加算
ADDIU	I	0	0	1	0	0	1							RS	SD	加算
SUB	R	0	0	0	0	0	0	1	0	0	0	1	0	RS	RT	減算
SUBU	R	0	0	0	0	0	0	1	0	0	0	1	1	RS	RT	減算
AND	R	0	0	0	0	0	0	1	0	0	1	0	0	RS	RT	AND
ANDI	I	0	0	1	1	0	0							RS	SD	AND
OR	R	0	0	0	0	0	0	1	0	0	1	0	1	RS	RT	OR
ORI	I	0	0	1	1	0	1							RS	SD	OR
SLL	R	0	0	0	0	0	0	0	0	0	0	0	0	RS	RT	Shift
SRL	R	0	0	0	0	0	0	0	0	0	0	1	0	RS	RT	Shift
NOR	R	0	0	0	0	0	0	1	0	0	1	1	1	RS	RT	NOR
XOR	R	0	0	0	0	0	0	1	0	0	1	1	0	RS	RT	XOR
XORI	I	0	0	1	1	1	0							RS	SD	XOR
NOP	R	0	0	0	0	0	0	0	0	0	0	0	0	RS	RT	Shift
LUI	I	0	0	1	1	1	1							RS	SD	Shift
LW	I	1	0	0	0	1	1							RS	SD	加算
SW	I	1	0	1	0	1	1							RS	SD	加算
BEQ	I	0	0	0	1	0	0							RS	RT	減算
BNE	I	0	0	0	1	0	1							RS	RT	減算
J	J	0	0	0	0	1	0							-----	-----	-----
JAL	J	0	0	0	0	1	1							-----	-----	-----
JR	R	0	0	0	0	0	0	0	0	1	0	0	0	RS	RT	Shift
SLT	R	0	0	0	0	0	0	1	0	1	0	1	0	RS	RT	減算
SLTI	I	0	0	1	0	1	0							RS	SD	減算
SLTU	R	0	0	0	0	0	0	1	0	1	0	1	1	RS	RT	減算
SLTIU	I	0	0	1	0	1	1							RS	SD	減算

デコードの表 3.3 および各命令動作より次のことがいえる。

●Operation Code が全て 0 である場合、命令は R 形式である。

●メモリに関する命令は Operation Code の最上位ビットが 1 である。

●Operation Code の 3 ビット目が 1 である場合には Jump 命令、分岐命令以外の標準の I 形式の命令である。

デコードの手順

①　もし、Operation Code=All Zero ならば R 形式命令なので

Operand1=RS, Operand2=RT とする。

② ①の条件を満たす場合にはその演算形式を判定する。

ALU Function Code[5][2][1]=100 ならば ALU は加算を行う。
ただし ALU Function Code[0]=1 ならば AllZero 出力をしない。
ALU Function Code[5][2][1]=101 ならば ALU は減算を行う。
ただし ALU Function Code[3]=1 ならば大小比較命令を行う。
また、ALU Function Code[0]=1 ならば AllZero 出力をしない。
ALU Function Code[5][2][1][0]=1100 なら ALU は AND を行う。
ALU Function Code[5][2][1][0]=1101 なら ALU は OR を行う。
ALU Function Code[5][2][1][0]=1110 なら ALU は NOR を行う。
ALU Function Code[5][2][1][0]=1111 なら ALU は XOR を行う。
ALU Function Code[5][2][1]=000 ならば ALU は Shift を行う。
ただし ALU Function Code[3]=0 の時には右シフトを、１のときには左シフトを行う。

③ ①を満たさない場合には Operation Code[5]=1 ならばメモリ命令として、ALU は加算を行う。

④ ③を満たさない場合 Operation Code[3]=1 ならば I 形式命令なので、Operand1=RS, Operand2=SD とする。

⑤ ④の条件を満たす場合その演算形式を判定する。

Operation Code[2][1]=00 ならば ALU は加算を行う。
ただし、Operation Code[0]=1 ならば AllZero を出力しない。
Operation Code[2~0]=100 ならば ALU は AND を行う。
Operation Code[2~0]=101 ならば ALU は OR を行う。
Operation Code[2~0]=110 ならば ALU は XOR を行う。
Operation Code[2~0]=111 ならば ALU は LUI 命令を行う。
Operation Code[2][1]=01 ならば ALU は減算を行う。
ただし Operation Code[0]=1 の時には AllZero を出力しない。

⑥ ④を満たさない場合

Operation Code[2~0]=100 ならばALUはBEQ命令として減算を行う。
Operation Code[2~0]=101 ならばALUは減算を行い、Allzeroには通常の出力を反転させたものを出力する。
Operation Code[2][1]=01 ならばJump命令なのでALUは動作しない。

3.4 CPUレジスタ仕様設計

3.4.1 仕様

本研究で設計したプロセッサには、32個各32ビットの汎用レジスタを内蔵している。レジスタ番号0および31は特殊レジスタとして予約され、レジスタ0はNOP命令および0を必要とする演算のためにハードウェア的に0に固定されている。レジスタ31はJAL（Jump And Link）命令用のProgram Counter値格納用である。レジスタ1～30まではユーザがプログラムで自由に使用可能なレジスタである。また、レジスタは複数入出力線を持つマルチポートメモリである。

本設計ではCPUレジスタの部分に命令解釈部を設け、命令コードを直接読み込んでそれを解読し、動作するものとした。

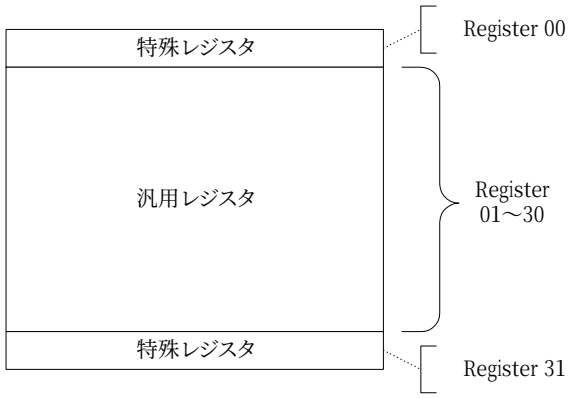


図3.6　CPUレジスタ

3.4.2 入出力線

すべての命令に対応するためにCPUレジスタの入出力線は次のように設定した。

入力線　R形式用　Source Register Address線　5ビット
　　Target Register Address 線　5ビット
　　Destination Register Address 線　5ビット
　　Execution Result 線　32ビット
　　Program Counter 線　32ビット

　　I形式用　周回Target Register Address 線　5ビット
　　Load Data 線 32ビット
　　制御線　　Operation Code 線 6ビット
出力線　　Source Register Data 線　32ビット
　　Target Register Data 線　32ビット

Source Register Address, Target Register Address, Destination Address線は命令コードの一部分である。
Source Register Address=命令コードの25ビット目～21ビット目。
Target Register Address=命令コードの20ビット目～16ビット目。
Destination Register Address=命令コードの15～10ビット目。

周回Target Register AddressはI形式命令においてDestination Address となるものであり、パイプラインを周回させてきた Target Register Addressである。データ入力線のLoad Data, Execution Result, Program Counter線はそれぞれデーターメモリからの出力データ、ALU演算結果、プログラムカウンタの値となる。

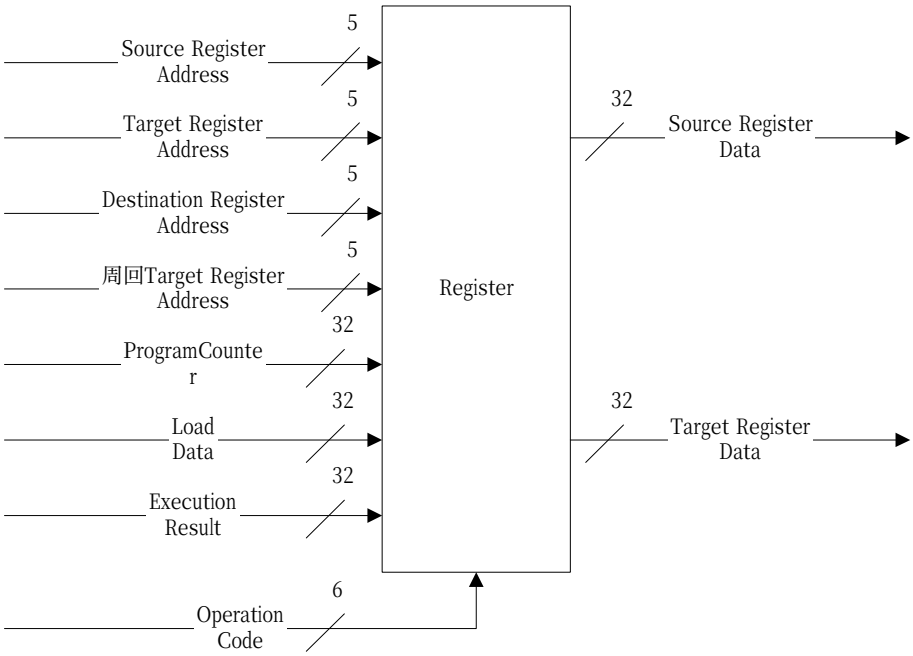


図3.7　レジスタ概要図

3.4.3 動作

まず、Source Register Address および Target Register Address で指示されたレジスタ番号のデータを読み出し、それを各々 Source Register Data, Target Register Data とする。半クロック遅れで命令をデコードし、それをもとに書込み作業を行う。

デコードの対応表は次のとおりである。なお、周回 Target Register Data は周回 TR、Destination Register は RD、Execution Result は ExecResult とする。

表 3.4　レジスタデコード

命令	形式	OperationCode						保存先アドレス	保存データ
ADD	R	0	0	0	0	0	0	RD	ExecResult
ADDI	I	0	0	1	0	0	0	周回RT	ExecResult
ADDU	R	0	0	0	0	0	0	RD	ExecResult
ADDIU	I	0	0	1	0	0	1	周回RT	ExecResult
SUB	R	0	0	0	0	0	0	RD	ExecResult
SUBU	R	0	0	0	0	0	0	RD	ExecResult
AND	R	0	0	0	0	0	0	RD	ExecResult
ANDI	I	0	0	1	1	0	0	周回RT	ExecResult
OR	R	0	0	0	0	0	0	RD	ExecResult
ORI	I	0	0	1	1	0	1	周回RT	ExecResult
NOR	R	0	0	0	0	0	0	RD	ExecResult
XOR	R	0	0	0	0	0	0	RD	ExecResult
XORI	I	0	0	1	1	1	0	周回RT	ExecResult
NOP	R	0	0	0	0	0	0	RD	ExecResult
SLL	R	0	0	0	0	0	0	RD	ExecResult
SRL	R	0	0	0	0	0	0	RD	ExecResult
LUI	I	0	0	1	1	1	1	周回RT	ExecResult
LW	I	1	0	0	0	1	1	周回RT	LoadData
JAL	J	0	0	0	0	1	1	レジスタ31	PC
SLT	R	0	0	0	0	0	0	RD	ExecResult
SLTI	I	0	0	1	0	1	0	周回RT	ExecResult
SLTU	R	0	0	0	0	0	0	RD	ExecResult
SLTIU	I	0	0	1	0	1	1	周回RT	ExecResult

表 3.4 より、デコードは次の手順で行う。

①Operation Code=AllZero すなわち
(Operation Code)XOR = 1 ならば
Input Address = RD , Input Data = Exec Result

②Operation Code[3]= 1 ならば
Input Address= 周回 RT , Input Data= Exec Result

③Operation Code[5]= 1 ならば
Input Address= 周回 RT , Input Data= Load Data

④すべてに該当しないなら、
Input Address= Register[31] , Input Data = PC

3.5 その他機能部の設計

3.5.1 命令メモリ（Instruction Memory）の設計

本研究ではプロセッサの命令メモリを 32 ビット幅 64 行のメモリとした。ハードウェア量としては非常に少ないが、シミュレーションおよび動作検証を行う上では十分だと判断したからである。本来はメモリとして外部から書き換えできるように設計するべきだが、本設計ではあらかじめ命令テーブルを作っておく方式とした。入力線としては 32 ビットの命令アドレス線、出力線としては 32 ビットの命令データ線のみで、クロックによる制御は行わない。

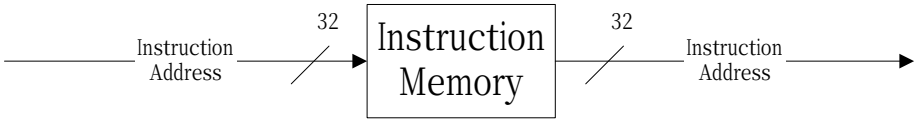


図3.8　　命令メモリ入出力線

動作はきわめて簡単で、入力アドレスを元に対象となるデータを出力するだけである。

3.5.2 データーメモリ（Data Memory）の設計

データーメモリは命令メモリと違って命令デコーダを内蔵し、データーメモリを必要とする命令の場合のみに書込み動作を行う。データーメモリを必要とする命令は Load 命令、および Store 命令のみである。書込みを行う命令は Operation Code を解読して書込み動作を行う。入力線ではデーターメモリ読み出し用アドレス、データーメモリ書込み用データ、出力線は読み出しデータである。なお、メモリアクセスは 1 クロックに付き 1 度しか動作しないので Register のようにマルチポートメモリとする必要性はない。さらに命令読み出しは、書込み命令以外の全ての命令で読み出しを行ったとしても、読み出したデータの行き先は Register となり、Register で適切に処理されるため問題はないが、

32 ビット分のアドレスのメモリを積み込むわけではないので誤動作を避けるために Load 命令のときだけメモリ読み出しを行うようする。Load 命令の Operation Code= (100011),Store 命令の Operation Code=(101011)であることからメモリアクセス動作は Operation Code を解読して行う。

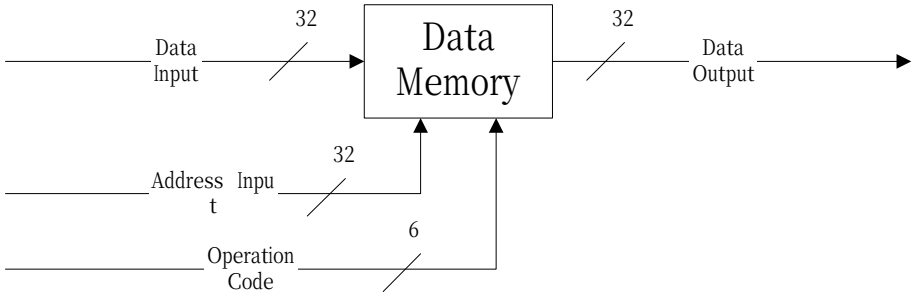


図3.9　　データメモリ入出力線

3.5.3 補助 ALU（ALU2）の設計

Branch 命令、JAL 命令では ALU がすでに使用されているので同時にアドレス計算をできない。そのためアドレス計算専用の ALU2 を用意している。ALU2 では４つのデータ入力線、２つの制御線、１つの出力線を備え、各命令に応じて入力線を選びアドレスを計算する。各入力線は以下のとおりである。

入力線　ALU2 Input Data,1：Pipeline Register を通過してきた Program Counter のデータ
ALU2 Input Data,2：Source Register Data
ALU2 Input Data,3：Sign Extended Data
ALU2 Input Data,4：Instruction Code[25~0]
出力線 ALU2 Output Data：Program Counter Address Input

ALU2 を使用する命令は Branch 命令、JAL 命令のほかに、JR 命令、Jump 命令があげられる。各命令の Operation Code および ALU Function Code は表 3.5 のとおりである。なお各端子の記述は省略形にしている。

表 3.5　　ALU2 デコード表

命令	形式	Operation Code						Function Code						Operand1	Operand2
BEQ	I	0	0	0	1	0	0	-----						PC	SD
BNE	I	0	0	0	1	0	1	-----						PC	SD
J	J	0	0	0	0	1	0	-----						PC	Inst[25-0]
JAL	J	0	0	0	0	1	1	-----						PC	Inst[25-0]
JR	R	0	0	0	0	0	0	0	0	1	0	0	0	RS	-----

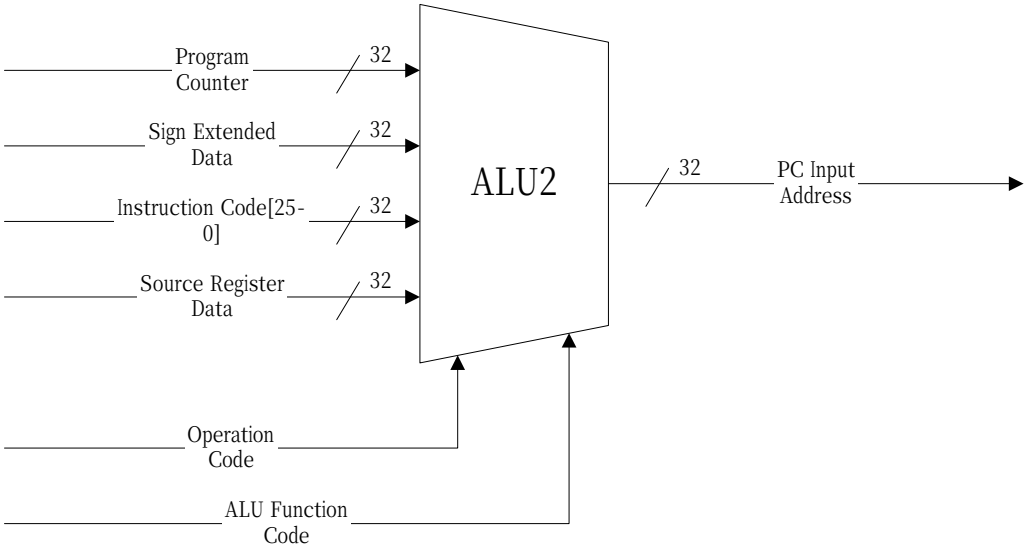


図3.10　　補助ALU入出力線

動作としては、まず Operation　Code および ALU Function Code が一致することを確認する。次にその命令動作に移る。命令自身の詳細は後述する。表 3.5 の Operand2 に関しては、入力が無いものとして扱うため内部では 0 として処理するものとする。

●BEQ 命令の場合

SD の入力値の下位 2 ビットに 0 を結合する。結合後、Program Counter の値に 4 を加えたものを加算する。加算結果が 32 ビットを超えてしまうので、下位から 32 ビット分残し上位ビットを切り捨てる。

●BNE 命令の場合

BEQ 命令と同じである。

●J 命令の場合

Program Counter の 31 ビット目～28 ビット目と命令コードの 25 ビット目～0 ビット目、そして 0 を 2 ビット分全て結合する。結合した結果が出力である。

●JAL 命令の場合

J 命令と同じである。

●JR 命令の場合

RS で読み出されたデータをそのまま出力とする。

3.5.4 符号拡張ユニット（Sign Extend Unit）の設計

Immediate 形式の命令の場合、ALU が 32 ビットなのであらかじめ符号拡張が必要である。符号拡張についての詳細は第 2 章で述べたとおりである。この符号拡張ユニットでは命令ごとにその拡張方法を判断し、ゼロ拡張か符号拡張かのいずれかを行う。符号拡張ユニットを使用する命令は表 3.6 のとおりである。

表 3.6　　拡張形式命令対応表

命令	形式	Operation Code						Operand1
ADDI	I	0	0	1	0	0	0	符号拡張
ADDIU	I	0	0	1	0	0	1	ゼロ拡張
ANDI	I	0	0	1	1	0	0	ゼロ拡張
ORI	I	0	0	1	1	0	1	ゼロ拡張
XORI	I	0	0	1	1	1	0	ゼロ拡張
LUI	I	0	0	1	1	1	1	ゼロ拡張
LW	I	1	0	0	0	1	1	符号拡張
SW	I	1	0	1	0	1	1	符号拡張
BEQ	I	0	0	0	1	0	0	符号拡張
BNE	I	0	0	0	1	0	1	符号拡張
SLTI	I	0	0	1	0	1	0	符号拡張
SLTIU	I	0	0	1	0	1	1	符号拡張

ゼロ拡張を必要とする演算は符号なし（ADDIU のみ）の通常演算と論

理演算である。その他、ビット操作命令の一つである LUI 命令も同様である。符号拡張ユニットの入力線は 16 ビットの Immediate および 6 ビットの制御線、出力は拡張された Immediate の 32 ビットである。

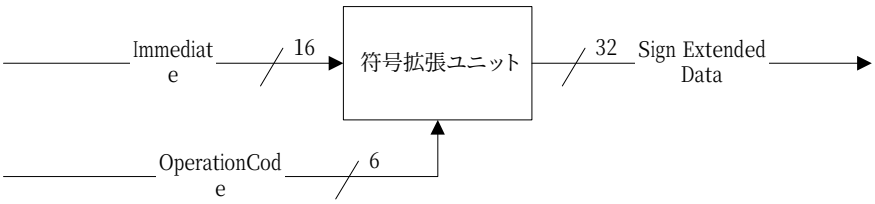


図3.11　　符号拡張ユニット入出力線図

3.5.5 パイプラインレジスタ（PR0,PR1,PR2）の設計

本設計のプロセッサの各ユニットにはクロックによって制御するものも若干あるが、大半のユニットではクロックによる制御を行わない。また、パイプライン方式を採用することで、回路上にまったく別の命令を次々にフェッチすることになるので、パイプラインを構成する各ユニットのタイミング制御を適切に行わなければ命令が正しく動作しない。そのため、パイプラインレジスタというレジスタをユニット毎（正確には１パイプラインステージ毎）に設置する必要性がある。パイプラインレジスタは特にこれといった動作はしない。ただ単に入力したデータを一時的に格納し、次のクロックで出力するものである。パイプラインレジスタは基本的に D-FF で構成される。また、回路とパイプラインステージの都合上パイプラインレジスタを３つに分割する。

まず第１パイプラインレジスタ（PR0）ではパイプラインステージのフェッチ、デコード（レジスタリード）間でのデータの受け渡し、そのあとのメモリアクセス、ライトバック（レジスタライト）間での動作を同時に行う。命令コードを入力されると、これをソースレジスタアドレス、ターゲットレジスタアドレス、デスティネーションレジスタアドレス、Immediate データ、Operation Code、ALU Function Code、Jump Address の各々に分割する。このレジスタに対しての入出力線は次のとおりである。

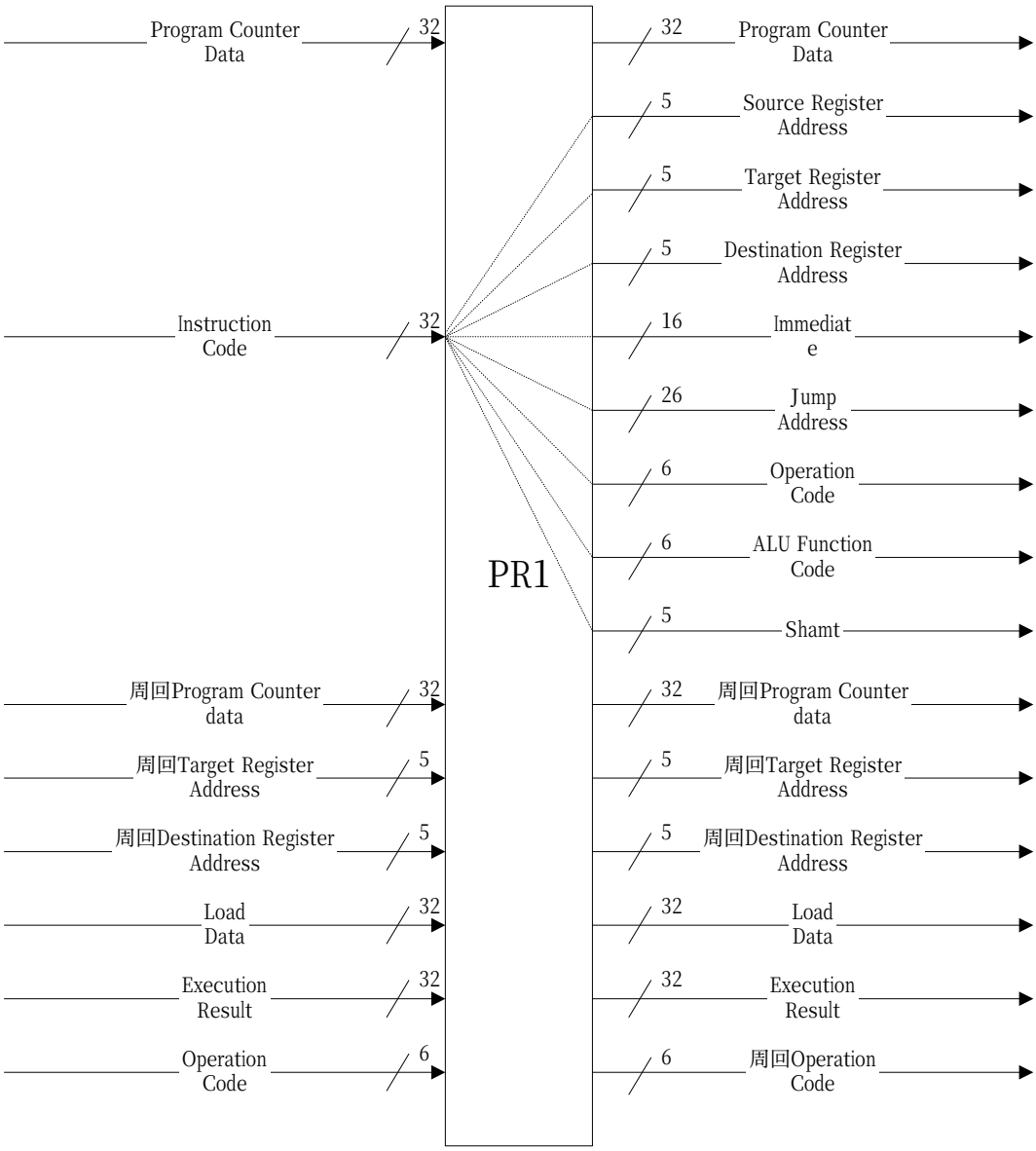


図3.12　　第1パイプラインレジスタ入出力線図

```
Source Register Address”5=Instruction Code[25~21]
Target Register Address”5=Instruction Code[20~16]
Destination Register Address”5=Instruction Code[15~11]
Immediate”16=Instruction Code[15~0]
Jump Address”26=Instruction Code[25~0]
Operation Code”6=Instruction Code[31~26]
ALU Function Code”6=Instruction Code[5~0]
```

Shamt”5=Instruction Code[10~6]

次に第 2 パイプラインレジスタは、第1パイプラインレジスタと異なり単純に入力されたデータを次のクロックで出力するだけでり、第 1 パイプラインレジスタよりも信号が少ない。ただし、動作的にはまったく変わり無いがここで Destination Register Address を周回 Destination Register Address に、Target Register Address を周回 Target Register Address に変更する。入出力信号線の概要は図 3.13 のとおりである。

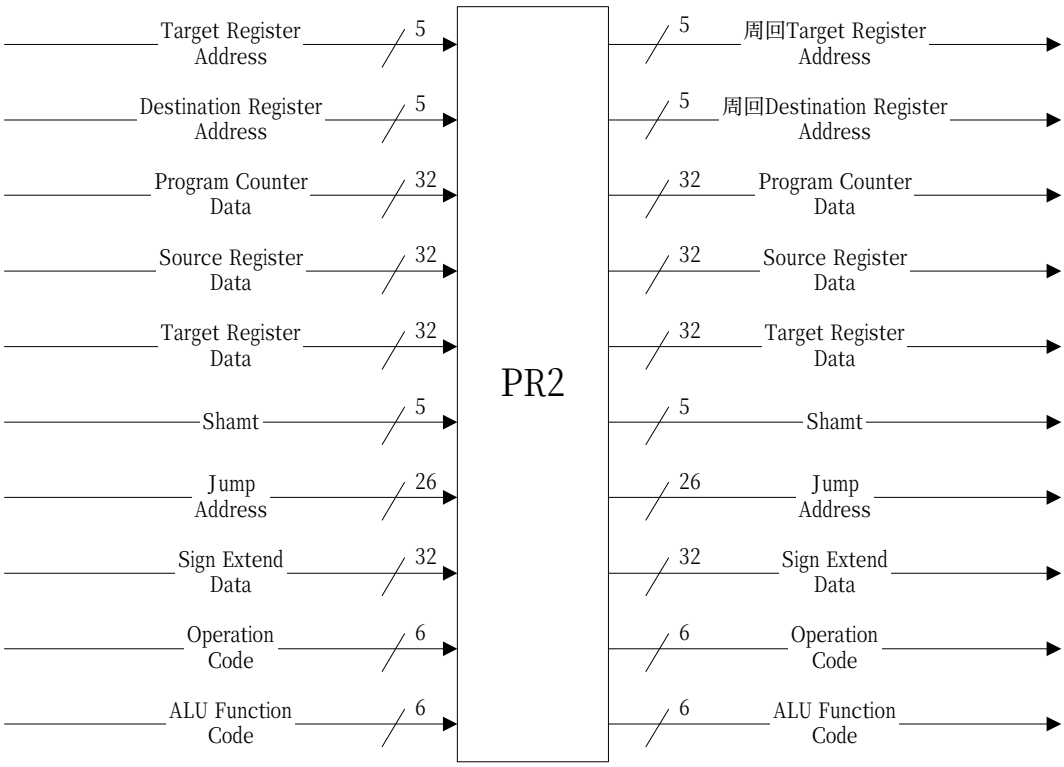


図3.13　第2パイプラインレジスタ入出力線図

第 3 パイプラインレジスタは第 2 パイプラインレジスタと同様の動作をする。ただし、Program Counter　Data が周回 Program Counter Data となる。第 3 パイプラインレジスタの入出力線は図 3.14 のとおりである。

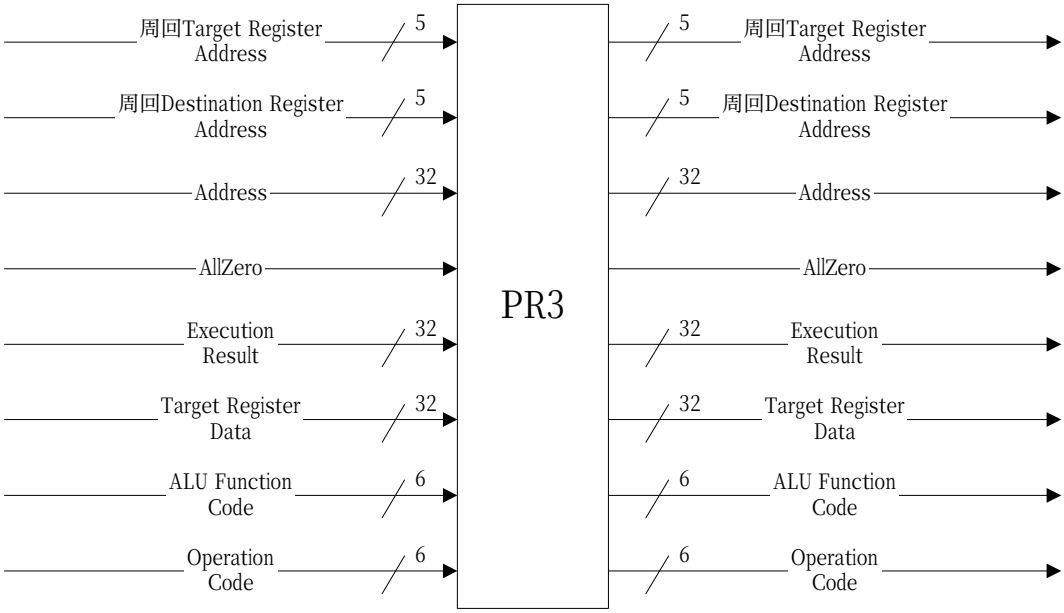


図3.14　第3パイプラインレジスタ入出力線図

3.6 データーパス設計

3.6.1 R 形式命令用データーパス設計

R 形式命令の処理手順は命令フェッチ→デコード→演算→レジスタ格納である。この順番で図 3.15 のようにデータパスをつなげば命令処理できる。なお各ブロック間にパイプラインレジスタを挟み、タイミングを取っている。またパイプラインは５段なので R 形式命令の場合にはメモリアクセスフェーズでは動作しないようにした。

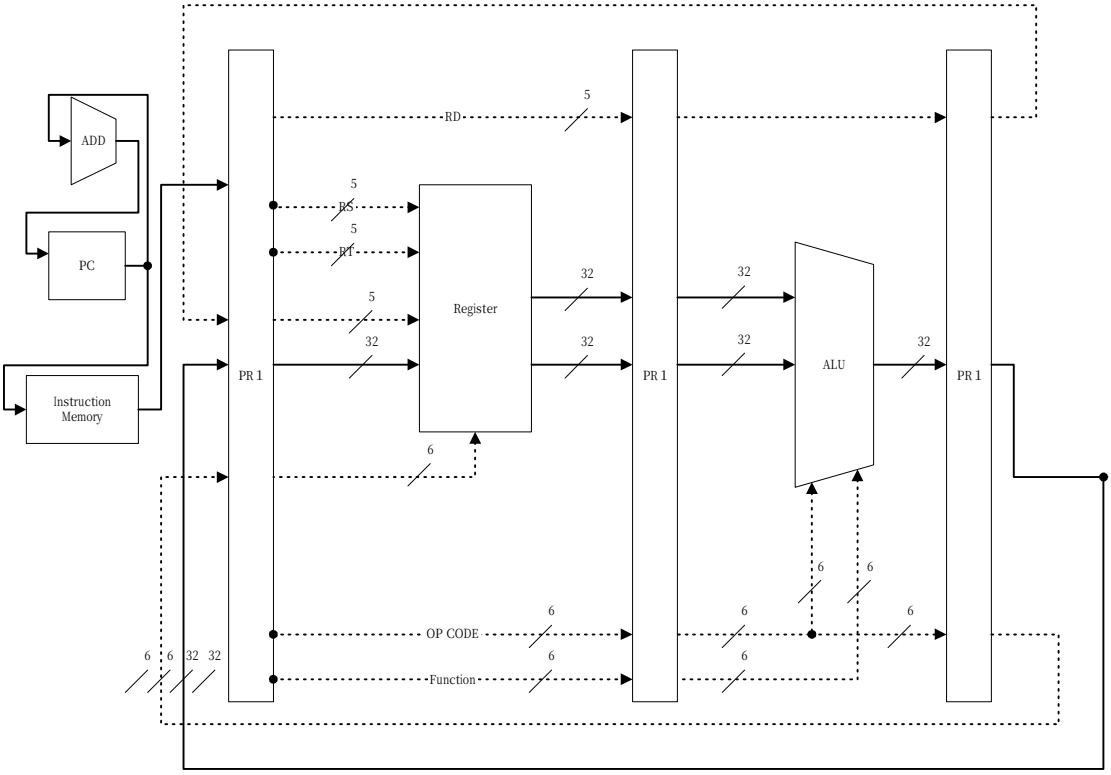


図 3.15　R 形式データパス

3.6.2 I 形式命令用データーパス設計

I 形式命令には分岐命令やメモリアクセス命令が含まれる。そのため単純な演算のみのデーターパスだけではなくこれら命令を処理するためのデーターパスが必要になる。このためデーターパスは、上記の R 形式データーパスに分岐命令、メモリアクセス命令の処理系を付け加えたものとなる。また、メモリアクセス命令および、Immediate を扱うので、伴う機能部（データメモリ、符号拡張ユニット、補助 ALU）が必要になる。なお、I 形式ではメモリアクセス命令においてパイプラインのメモリアクセスフェーズにメモリアクセスを行うが、他の命令ではメモリアクセスを行わないので、R 形式と同じようにここでは動作しないように設計した。

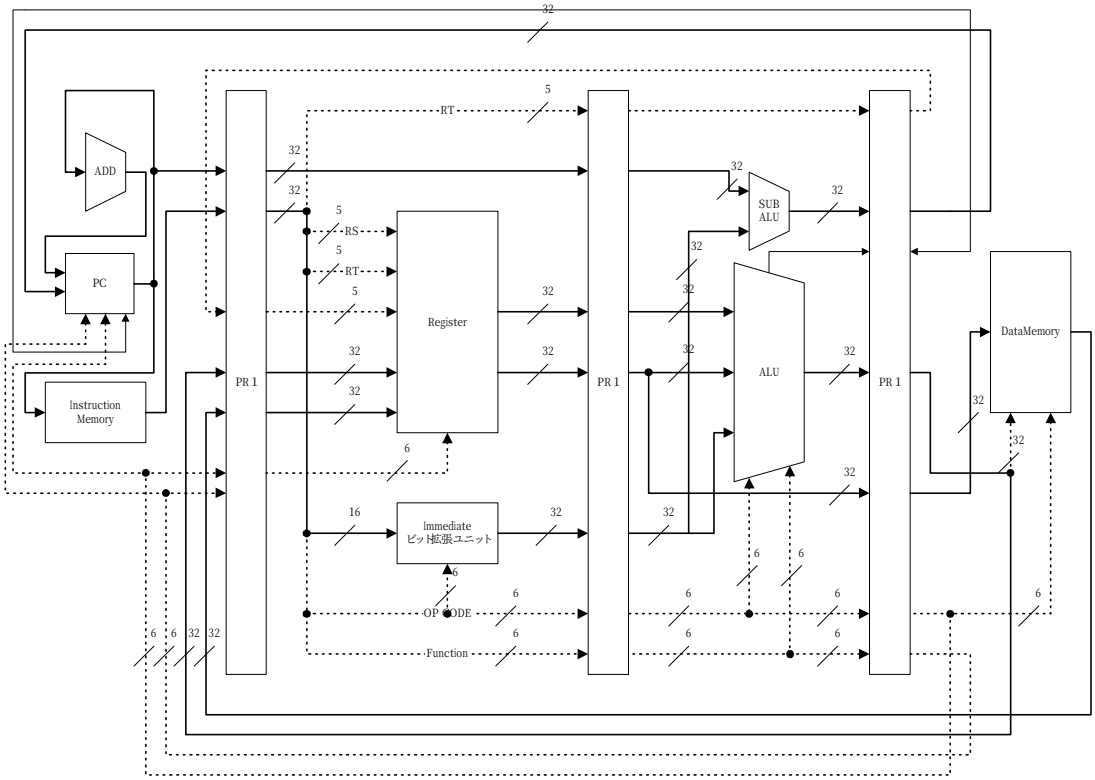


図 3.16　　I 形式データーパス

3.6.3 J 形式命令用データーパス設計

本設計での J 形式命令は Jump 命令、Jump And Link(JAL)命令のみで、これらを処理するためのデーターパスを設計した。J 形式命令ではオペランドは 26 ビットのアドレスのみなので、それを I 形式の Branch 命令と同じように付録の命令処理に従って、補助 ALU で処理する。また、ALU やデータメモリ、および符号拡張ユニットは使用しない。他の形式とは異なり、JAL 命令では保存先アドレスが命令コードの中に含まれていないので、専用のアドレスレジスタ（レジスタ 31）を使用する。設計したデーターパスは図 3.17 の通りである。

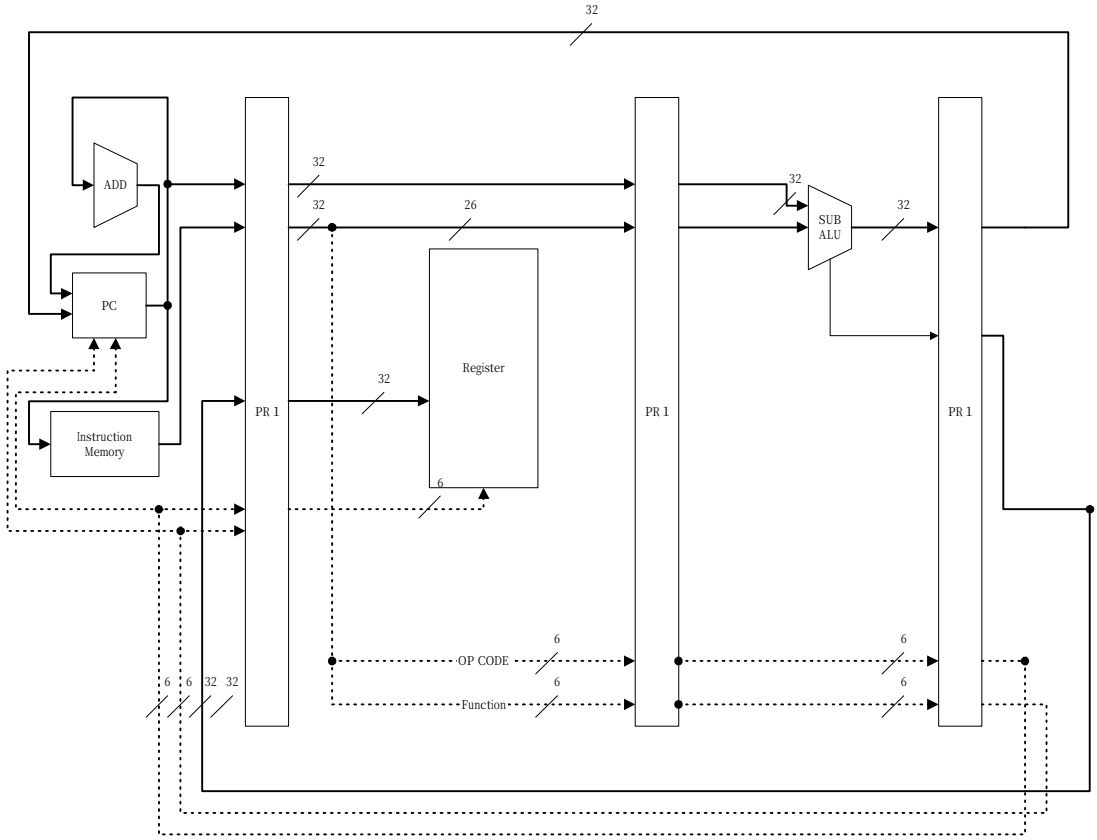


図 3.17　　J 形式データーパス

3.6.4 全体のデータパス

全体のデーターパスはすべての命令形式（R 形式、I 形式、J 形式）のデーターパスを結合したものである。結合し、完成したデータパスは付録に添付するとおりである。

第４章 VHDL 記述

第 3 章での設計を元に VHDL による記述およびシミュレーションを行った。VHDL の各モジュールは図 4.1 のように階層構成とした。各モジュールは第 3章で述べた各機能ブロックに対応する。VHDL エディタは Xilinx 社 WebPackISE、シミュレータは ModelSim XE を使用した。[3][4][5]

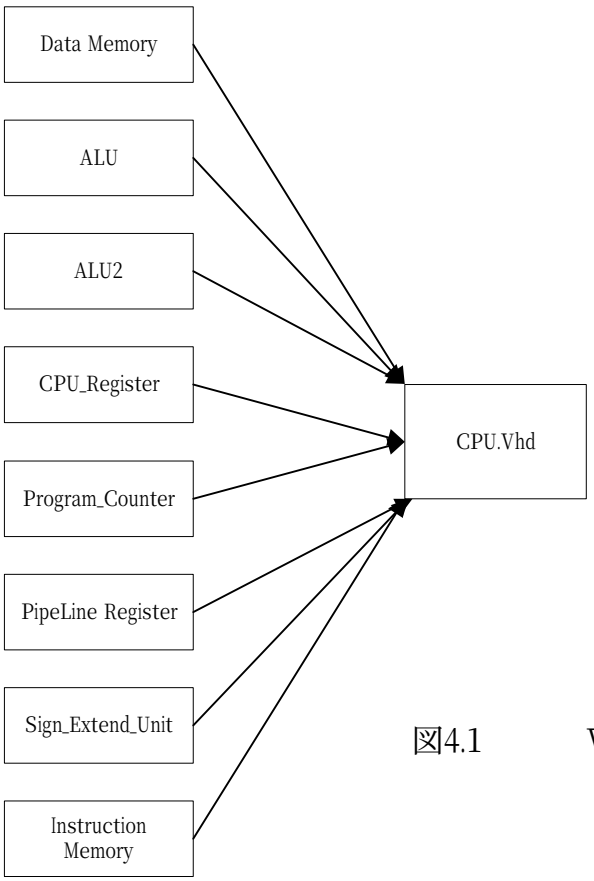


図4.1　　VHDL階層

4.1 各機能ブロックの記述

4.1.1　ALU の記述

library IEEE;

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ALU is
      Port ( Source_Register_Data : in std_logic_vector(31 downto 0);
                Target_Register_Data : in std_logic_vector(31 downto 0);
                Sign_Extended_Data : in std_logic_vector(31 downto 0);
                Operation_Code : in std_logic_vector(5 downto 0);
                ALU_Function_Code : in std_logic_vector(5 downto 0);
                Shamt : in std_logic_vector(4 downto 0);
                AllZero : out std_logic;
                Execution_Result : out std_logic_vector(31 downto 0));
end ALU;

architecture TIKUWA of ALU is
begin

process(Operation_Code,ALU_Function_Code,Source_Register_Data,Target_Reg
ister_Data,Sign_Extended_Data,Shamt)
   variable temp: std_logic_vector(32 downto 0);
   variable zero: std_logic_vector(32 downto 0);
   begin
      zero := "000000000000000000000000000000000";
-- R Type Instruction Process
--------------------------------------------------------------------------
case Operation_Code is
   when "000000" =>
   case ALU_Function_Code is
-- ADD Arithmetical Instruction Process
------------------------------------------------------------------------
   when "100000" =>
   temp := (Source_Register_Data(31) & Source_Register_Data) +
                  (Target_Register_Data(31) & Target_Register_Data);
   if (temp(32)=temp(31)) then
   else
```

```
   temp := zero;
   end if;
-- ADDU Arithmetical Instruction Process
-------------------------------------------------------------------------
   when "100001" =>
      temp := ("0" & Source_Register_Data) + ("0" & Target_Register_Data);
-- SUB Arithmetical Instruction Process
-------------------------------------------------------------------------
   when "100010" =>
      temp := (Source_Register_Data(31) & Source_Register_Data) -
                    (Target_Register_Data(31) & Target_Register_Data);
   if (temp(32)=temp(31)) then
   else
   temp := zero;
   end if;
-- SUBU Arithmetical Instruction Process
-------------------------------------------------------------------------
   when "100011" =>
   temp := ("0" & Source_Register_Data) - ("0" & Target_Register_Data);
-- AND Logical Instruction Process
-------------------------------------------------------------------------
   when "100100" =>
      temp := ("0"&Source_Register_Data) AND ("0"&Target_Register_Data);
-- OR Logical Instruction Process
-------------------------------------------------------------------------
   when "100101" =>
      temp := ("0"&Source_Register_Data) OR ("0"&Target_Register_Data);
-- NOR Logical Instruction Process
-------------------------------------------------------------------------
   when "100111" =>
      temp := ("0"&Source_Register_Data) NOR ("0"&Target_Register_Data);
-- XOR Logical Instruction Process
-------------------------------------------------------------------------
   when "100110" =>
      temp := ("0"&Source_Register_Data) XOR ("0"&Target_Register_Data);
-- SLL,NOP,JR Logical Instruction process
```

```
-------------------------------------------------------------------------
   when "000000" | "001000"=>
      temp := "0" & SHL(Source_Register_Data,shamt);
-- SRL Logical Instruction Process
-------------------------------------------------------------------------
   when "000010" =>
      temp := "0" & SHR(Source_Register_Data,shamt);
-- SLT Logical Instruction Process
-------------------------------------------------------------------------
   when "101010" =>
      if (Source_Register_Data < Target_Register_Data) then
         temp := zero(31 downto 0) & "1";
            else
         temp := zero;
end if;
-- SLTU Logical Instruction Process
-------------------------------------------------------------------------
   when "101011" =>
      if ("0" & Source_Register_Data < "0" & Target_Register_Data) then
         temp := zero(31 downto 0) & "1";
         else
         temp := zero;
         end if;
when others =>
temp := zero;
end case;
-- I type Instruction
-- ADDI Arithmetical Instruction Process
-------------------------------------------------------------------------
   when "001000" =>
      temp := (Source_Register_Data(31) & Source_Register_Data) +
                  (Sign_Extended_Data(31) & Sign_Extended_Data);
      if (temp(32)=temp(31)) then
      else
      temp := zero;
      end if;
```

```
-- ADDIU Arithmetic Instruction Process
-------------------------------------------------------------------------
   when "001001" =>
      temp := ("0" & Source_Register_Data) + ("0" & Sign_Extended_Data);
-- AND Logical Instruction Process
-------------------------------------------------------------------------
   when "001100" =>
      temp := ("0"&Source_Register_Data) AND ("0"&Sign_Extended_Data);
-- OR Logical Instruction Process
-------------------------------------------------------------------------
   when "001101" =>
      temp := ("0"&Source_Register_Data) OR ("0"&Sign_Extended_Data);

-- XORI Logical Instruction Process
-------------------------------------------------------------------------
   when "001110" =>
      temp := ("0"&Source_Register_Data) XOR ("0"&Sign_Extended_Data);
-- LUI Logical Instruction Process
-------------------------------------------------------------------------
   when "001111" =>
      temp := "0" & SHL(Sign_Extended_Data,"10000");
-- LW,SW Logical Instruction Process
-------------------------------------------------------------------------
   when "100011" | "101011" =>
      temp := "0" & (Source_Register_Data + Sign_Extended_Data);
      temp(1 downto 0) := "00";
-- BEQ,BNE Logical Instruction Process
-------------------------------------------------------------------------
   when "000100" | "000101" =>
      temp := "0" & (Source_Register_Data - Target_Register_Data);
-- SLTI Logical Instruction Process
-------------------------------------------------------------------------
   when "001010" =>
      if (Source_Register_Data < Sign_Extended_Data) then
      temp := zero(31 downto 0) & "1";
      else
```

```
      temp := zero;
      end if;
-- SLTIU Logical Instruction Process
-------------------------------------------------------------------------
   when "001011" =>
      if ("0" & Source_Register_Data < "0" & Sign_Extended_Data) then
      temp := zero(31 downto 0) & "1";
      else
      temp := zero;
      end if;
when others =>
   temp := zero;
end case;
-- ALL ZERO Detection
-------------------------------------------------------------------------
if(temp="0") then
   AllZero <='1';
else
   AllZero <='0';
end if;
-- Result Output
-------------------------------------------------------------------------
   Execution_Result <= temp (31 downto 0);
end Process;
end TIKUWA;
```

4.1.2 CPU レジスタの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cpu_register is
   Port ( Source_Register_Address : in std_logic_vector(4 downto 0);
```

```
        Target_Register_Address : in std_logic_vector(4 downto 0);
        Destination_Register_Address : in std_logic_vector(4 downto 0);
        Cyclic_Target_Register_Address : in std_logic_vector(4 downto 0);
        Program_Counter : in std_logic_vector(31 downto 0);
        Load_Data : in std_logic_vector(31 downto 0);
        Execution_Result : in std_logic_vector(31 downto 0);
        Operation_Code : in std_logic_vector(5 downto 0);
        Source_Register_Data : out std_logic_vector(31 downto 0);
        Target_Register_Data : out std_logic_vector(31 downto 0);
        Clock : in std_logic);
end cpu_register;

architecture TIKUWA of cpu_register is

type Register_Array is array (0 to 31) of std_logic_vector(31 downto 0);
  signal Reg: Register_Array;

begin
t1:process
(Operation_Code,Source_Register_Address,Target_Register_Address,Clock)
  begin
  Reg(0) <= "00000000000000000000000000000000";
  Source_Register_Data <= Reg(CONV_INTEGER(Source_Register_Address));
  Target_Register_Data <= Reg(CONV_INTEGER(Target_Register_Address));
end process;
t2: process (Clock)
  begin
  Reg(0) <= "00000000000000000000000000000000";
  if (Clock'event and Clock='0') then
    case Operation_Code is
      when "000000" =>
        if (Destination_Register_Address="00000") then
        Reg(0) <= "00000000000000000000000000000000";
        else
        Reg(CONV_INTEGER(Destination_Register_Address))                <=
Execution_Result;
```

```
            end if;
          when "001000" | "001001" | "001100" | "001101" | "001110" | "001111" |
               "001010" | "001011" =>
            if (Cyclic_Target_Register_Address="00000") then
            Reg(0) <= "00000000000000000000000000000000";
            else
            Reg(CONV_INTEGER(Cyclic_Target_Register_Address))                <=
Execution_Result;
            end if;
          when "100011" =>
            if (Cyclic_Target_Register_Address="00000") then
            Reg(0) <= "00000000000000000000000000000000";
            else
            Reg(CONV_INTEGER(Cyclic_Target_Register_Address)) <= Load_Data;
            end if;
          when "000011" =>
            Reg(31) <= Program_Counter;
          when others =>
            Reg(0) <= "00000000000000000000000000000000";
          end case;
       end if;
end process;
end TIKUWA;
```

4.1.3 補助 ALU の記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ALU2 is
      Port ( Program_Counter : in std_logic_vector(31 downto 0);
          Sign_Extended_Data : in std_logic_vector(31 downto 0);
          Address_Input : in std_logic_vector(25 downto 0);
```

```
        Source_Register_Data : in std_logic_vector(31 downto 0);
        Operation_Code : in std_logic_vector(5 downto 0);
        ALU_Function_Code : in std_logic_vector(5 downto 0);
        PC_Output : out std_logic_vector(31 downto 0);
        Link_Register_Data : out std_logic_vector(31 downto 0));
end ALU2;
architecture TIKUWA of ALU2 is
begin

process (Operation_Code,Alu_Function_Code,Program_Counter)
variable temp1,temp2: std_logic_vector(31 downto 0);
begin
  case Operation_Code is
----------Branch Instruction ----------------------------------------
  when "000100" | "000101" =>
        temp1 := SHL(Sign_Extended_Data,"10");
        temp2 := Program_Counter + "100";
        PC_Output <= temp1 + temp2;
----------Jump Instruction ----------------------------------------
  when "000010" =>
        PC_Output <= Program_Counter(31 downto 28) & Address_Input &
"00";
----------Jump And Link Instruction ---------------------------------
  when "000011" =>
        PC_Output <= Program_Counter(31 downto 28) & Address_Input &
"00";
        Link_Register_Data <= Program_Counter + "1000";
----------Jump Register Operation ---------------------------------
  when "000000" =>
        if (ALU_Function_Code="001000") then
        PC_Output <= Source_Register_Data ;
        end if;
  when others =>
        PC_Output <= "00000000000000000000000000000000";
  end case;
end process;
```

```
end TIKUWA;
```

4.1.4 プログラムカウンタの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity PC is
      Port ( Program_Counter_Input : in std_logic_vector(31 downto 0);
          Operation_Code : in std_logic_vector(5 downto 0);
          ALU_Function_Code : in std_logic_vector(5 downto 0);
          AllZero : in std_logic;
          Program_Counter_Output : out std_logic_vector(31 downto 0);
          Clock : in std_logic;
          Reset : in std_logic);
end PC;

architecture TIKUWA of PC is
signal count: std_logic_vector(31 downto 0);
begin
   Program_Counter_Output <= count;
process (Clock,Reset) begin
   if(Reset='1') then
   count <= "00000000000000000000000000000000";
   else if(Clock'event and Clock='1') then
      case Operation_Code is
         when "000100" =>
            if (AllZero='1') then
               count <= Program_Counter_Input;
            else
               count <= count + "100";
            end if;
```

```
        when "000101" =>
          if (AllZero='0') then
            count <= Program_Counter_Input;
          else
            count <= count + "100";
          end if;
        when "000010" | "000011"=>
          count <= Program_Counter_Input;
        when "000000" =>
          if (ALU_Function_Code="001000") then
            count <= Program_Counter_Input;
          else
            count <= count + "100";
          end if;
        when others =>
            count <= count + "100";
        end case;
      end if;
   end if;
end process;
end TIKUWA;
```

4.1.5 データメモリの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity DataMemory is
      Port ( Target_Register_Data : in std_logic_vector(31 downto 0);
          Execution_Result : in std_logic_vector(31 downto 0);
          Operation_Code : in std_logic_vector(5 downto 0);
          Load_Data : out std_logic_vector(31 downto 0));
end DataMemory;
```

```
architecture TIKUWA of DataMemory is
  type DATA_Memory_Array is array (0 to 31) of std_logic_vector(31 downto 0);
  signal Mem: Data_Memory_Array;
begin
process (Operation_Code,Target_Register_Data,Execution_Result)
  begin
  case Operation_Code is
    when "100011" =>
   Load_Data <= Mem(CONV_INTEGER('0' & Execution_Result(6 downto 2)));
    when "101011" =>
      Mem(CONV_INTEGER('0' & Execution_Result(6 downto 2)))
       <= Target_Register_Data;
    when others =>
  end case;
end process;
end TIKUWA;
```

4.1.6 符号拡張ユニットの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Sign_Extend_Unit is
    Port ( Immediate : in std_logic_vector(15 downto 0);
        Operation_Code : in std_logic_vector(5 downto 0);
        Sign_Extended_Data : out std_logic_vector(31 downto 0));
end Sign_Extend_Unit;

architecture TIKUWA of Sign_Extend_Unit is
begin
process (Immediate,Operation_Code)
  begin
```

```
      case Operation_Code is
        when "001000" | "100011" | "101011" | "000100" | "000101" | "001010" |
"001011" =>
          Sign_Extended_Data <= Immediate(15) & Immediate(15) &
              Immediate(15) & Immediate(15) & Immediate(15) &
              Immediate(15) & Immediate(15) & Immediate(15) &
              Immediate(15) & Immediate(15) & Immediate(15) &
              Immediate(15) & Immediate(15) & Immediate(15) &
              Immediate(15) & Immediate(15) & Immediate;
        when "001001" | "001100" | "001101" | "001110" | "001111" =>
          Sign_Extended_Data <= "0000000000000000" & Immediate;
        when others =>
          Sign_Extended_Data <= "00000000000000000000000000000000";
      end case;
end process;
end TIKUWA;
```

4.1.7 第１パイプラインレジスタの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity PR0 is
Port ( Program_Counter_Data_in : in std_logic_vector(31 downto 0);
   Instruction_Code_in : in std_logic_vector(31 downto 0);
   Cyclic_Program_Counter_Data_in : in std_logic_vector(31 downto 0);
   Cyclic_Target_Register_Address_in : in std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_in : in std_logic_vector(4 downto 0);
   Load_Data_in : in std_logic_vector(31 downto 0);
   Execution_Result_in : in std_logic_vector(31 downto 0);
   Operation_Code_in : in std_logic_vector(5 downto 0);
   Program_Counter_Data_out : out std_logic_vector(31 downto 0);
   Sourcd_Register_Address_out : out std_logic_vector(4 downto 0);
```

```
   Target_Register_Address_out : out std_logic_vector(4 downto 0);
   Destination_Register_Address_out : out std_logic_vector(4 downto 0);
   Immediate_out : out std_logic_vector(15 downto 0);
   Jump_Address_out : out std_logic_vector(25 downto 0);
   Operation_Code_out : out std_logic_vector(5 downto 0);
   ALU_Function_Code : out std_logic_vector(5 downto 0);
   Shamt_out : out std_logic_vector(4 downto 0);
   Cyclic_Program_Counter_Data_out : out std_logic_vector(31 downto 0);
   Cyclic_Target_Register_Address_out : out std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_out : out std_logic_vector(4 downto 0);
   Load_Data_out : out std_logic_vector(31 downto 0);
   Execution_Result_out : out std_logic_vector(31 downto 0);
   Cyclic_Operation_Code : out std_logic_vector(5 downto 0);
   Clock : in std_logic);
end PR0;
architecture TIKUWA of PR0 is
begin
process(Clock)
   begin
      if(Clock'event and Clock = '1') then
   Program_Counter_Data_out <= Program_Counter_Data_in after 100 ps;
Sourcd_Register_Address_out <= Instruction_Code_in(25 downto 21) after 100
ps;
   Target_Register_Address_out <= Instruction_Code_in(20 downto 16) after
100 ps;
   Destination_Register_Address_out <= Instruction_Code_in(15 downto 11)
after 100 ps;
   Immediate_out <= Instruction_Code_in(15 downto 0) after 100 ps;
   Jump_Address_out <= Instruction_Code_in(25 downto 0) after 100 ps;
   Operation_Code_out <= Instruction_Code_in(31 downto 26) after 100 ps;
   ALU_Function_Code <= Instruction_Code_in(5 downto 0) after 100 ps;
   Shamt_out <= Instruction_Code_in(10 downto 6) after 100 ps;
   Cyclic_Program_Counter_Data_out <= Cyclic_Program_Counter_Data_in after
100 ps;
   Cyclic_Target_Register_Address_out <= Cyclic_Target_Register_Address_in
after 100 ps;
```

```
   Cyclic_Destination_Register_Address_out<=Cyclic_Destination_Register_Addr
ess_in after 100 ps;
   Load_Data_out <= Load_Data_in after 100 ps;
   Execution_Result_out <= Execution_Result_in after 100 ps;
   Cyclic_Operation_Code <= Operation_Code_in after 100 ps;
   end if;
end process;
end TIKUWA;
```

４.１.８　第２パイプラインレジスタの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity PR2 is
Port ( Target_Register_Address_in : in std_logic_vector(4 downto 0);
   Destination_Register_Address_in : in std_logic_vector(4 downto 0);
   Program_Counter_Data_in : in std_logic_vector(31 downto 0);
   Source_Register_Data_in : in std_logic_vector(31 downto 0);
   Target_Register_Data_in : in std_logic_vector(31 downto 0);
   Shamt_in : in std_logic_vector(4 downto 0);
   Jump_Address_in : in std_logic_vector(25 downto 0);
   Sign_Extended_Data_in : in std_logic_vector(31 downto 0);
   Operation_Code_in : in std_logic_vector(5 downto 0);
   ALU_Function_Code_in : in std_logic_vector(5 downto 0);
   Cyclic_Target_Register_Address_out : out std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_out : out std_logic_vector(4 downto 0);
   Program_Couter_Data_out : out std_logic_vector(31 downto 0);
   Source_Register_Data_out : out std_logic_vector(31 downto 0);
   Target_Register_Data_out : out std_logic_vector(31 downto 0);
   Shamt_out : out std_logic_vector(4 downto 0);
   Jump_Address_out : out std_logic_vector(25 downto 0);
   Sign_Extended_Data_out : out std_logic_vector(31 downto 0);
   Operation_Code_out : out std_logic_vector(5 downto 0);
```

```
   ALU_Function_Code_out : out std_logic_vector(5 downto 0);
   Clock : in std_logic);
end PR2;
architecture TIKUWA of PR2 is
begin
process(Clock)
begin
if (Clock'event and Clock='1') then
Cyclic_Target_Register_Address_out <= Target_Register_Address_in after 100
ps;
   Cyclic_Destination_Register_Address_out<= Destination_Register_Address_in
after 100 ps;
        Program_Couter_Data_out <= Program_Counter_Data_in after 100 ps;
        Source_Register_Data_out <= Source_Register_Data_in after 100 ps;
        Target_Register_Data_out <= Target_Register_Data_in after 100 ps;
        Shamt_out <= Shamt_in after 100 ps;
        Jump_Address_out <= Jump_Address_in after 100 ps;
        Sign_Extended_Data_out <= Sign_Extended_Data_in after 100 ps;
        Operation_Code_out <= Operation_Code_in after 100 ps;
        ALU_Function_Code_out <= ALU_Function_Code_in after 100 ps;
      end if;
   end process;
end TIKUWA;
```

4.1.9 第 3 パイプラインレジスタの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity PR3 is
      Port ( Cyclic_Target_Register_Address_in : in std_logic_vector(4 downto 0);
          Cyclic_Destination_Register_Address_in : in std_logic_vector(4 downto
0);
          Address_in : in std_logic_vector(31 downto 0);
```

```
        AllZero_in : in std_logic;
        Execution_Result_in : in std_logic_vector(31 downto 0);
        Target_Register_Data_in : in std_logic_vector(31 downto 0);
        ALU_Function_Code_in : in std_logic_vector(5 downto 0);
        Operation_Code_in : in std_logic_vector(5 downto 0);
        Cyclic_Target_Register_Address_out : out std_logic_vector(4 downto 0);
        Cyclic_Destination_Register_Address_out : out std_logic_vector(4
downto 0);
        Address_out : out std_logic_vector(31 downto 0);
        AllZero_out : out std_logic;
        Execution_Result_out : out std_logic_vector(31 downto 0);
        Target_Register_Data_out : out std_logic_vector(31 downto 0);
        ALU_Function_Code_out : out std_logic_vector(5 downto 0);
        Operation_Code_out : out std_logic_vector(5 downto 0);
        Link_Register_Data_in : in std_logic_vector(31 downto 0);
        Link_Register_Data_out : out std_logic_vector(31 downto 0);
        Clock : in std_logic);
end PR3;
architecture TIKUWA of PR3 is
begin
process(Clock)
begin
  if (Clock'event and Clock= '1') then
  Cyclic_Target_Register_Address_out <= Cyclic_Target_Register_Address_in
after 100 ps;
  Cyclic_Destination_Register_Address_out <=
Cyclic_Destination_Register_Address_in after 100 ps;
  Address_out <= Address_in after 100 ps;
  AllZero_out <= AllZero_in after 100 ps;
  Execution_Result_out <= Execution_Result_in after 100 ps;
  Target_Register_Data_out <= Target_Register_Data_in after 100 ps;
  ALU_Function_Code_out <= ALU_Function_Code_in after 100 ps;
  Operation_Code_out <= Operation_Code_in after 100 ps;
  Link_Register_Data_out <= Link_Register_Data_in after 100 ps;
end if;
end process;
```

```
end TIKUWA;
```

4.1.10 命令メモリの記述

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IM is
      Port ( Instruction_Address : in std_logic_vector(31 downto 0);
          Instruction_Code : out std_logic_vector(31 downto 0));
end IM;
architecture TIKUWA of IM is
 type Instruction_Memory is array (0 to 63) of std_logic_vector (31 downto 0);
 signal ROM : Instruction_Memory :=(
 "00000000000000000000000000000000",
 "00000000000000000000000000000000",
              ↓ 省略) ↓
 "00000000000000000000000000000000",
 "00000000000000000000000000000000");
begin
process (Instruction_Address)
begin
   Instruction_Code <= ROM(conv_integer("0" & Instruction_Address(7 downto
2)));
end process;
end TIKUWA;
```

4.1.11 データパス記述

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
use IEEE.std_logic_unsigned.all;

entity CPU is
port (     Reset : in std_logic;    Clock : in std_logic;      Result_Exe :
           out std_logic_vector(31 downto 0);
   Result_Mem : out std_logic_vector(31 downto 0);
   PC_PC : out std_logic_vector(31 downto 0);SRD : out std_logic_vector(31
   downto 0);
   TRD : out std_logic_vector(31 downto 0);SED : out std_logic_vector(31
   downto 0));
end CPU;
architecture RTL of CPU is
component ALU
Port ( Source_Register_Data : in std_logic_vector(31 downto 0);
   Target_Register_Data : in std_logic_vector(31 downto 0);
   Sign_Extended_Data : in std_logic_vector(31 downto 0);
   Operation_Code : in std_logic_vector(5 downto 0);
   ALU_Function_Code : in std_logic_vector(5 downto 0);Shamt : in
      std_logic_vector(4 downto 0);
   AllZero : out std_logic;Execution_Result : out std_logic_vector(31 downto 0));
end component ;

component ALU2
Port ( Program_Counter : in std_logic_vector(31 downto 0);
   Sign_Extended_Data : in std_logic_vector(31 downto 0);
   Address_Input : in std_logic_vector(25 downto 0);
   Source_Register_Data : in std_logic_vector(31 downto 0);
   Operation_Code : in std_logic_vector(5 downto 0);
   ALU_Function_Code : in std_logic_vector(5 downto 0);
   PC_Output : out std_logic_vector(31 downto 0);
   Link_Register_Data : out std_logic_vector(31 downto 0));
end component ;

component IM
Port ( Instruction_Address : in std_logic_vector(31 downto 0);
   Instruction_Code : out std_logic_vector(31 downto 0));
```

```
end component ;

component PC
Port ( Program_Counter_Input : in std_logic_vector(31 downto 0);
   Operation_Code : in std_logic_vector(5 downto 0);
   ALU_Function_Code : in std_logic_vector(5 downto 0);AllZero : in std_logic;
   Program_Counter_Output : out std_logic_vector(31 downto 0);
   Clock : in std_logic;Reset : in std_logic);
end component ;

component PR0
Port ( Program_Counter_Data_in : in std_logic_vector(31 downto 0);
   Instruction_Code_in : in std_logic_vector(31 downto 0);
   Cyclic_Program_Counter_Data_in : in std_logic_vector(31 downto 0);
   Cyclic_Target_Register_Address_in : in std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_in : in std_logic_vector(4 downto 0);
   Load_Data_in : in std_logic_vector(31 downto 0);
   Execution_Result_in : in std_logic_vector(31 downto 0);
   Operation_Code_in : in std_logic_vector(5 downto 0);
   Program_Counter_Data_out : out std_logic_vector(31 downto 0);
   Sourcd_Register_Address_out : out std_logic_vector(4 downto 0);
   Target_Register_Address_out : out std_logic_vector(4 downto 0);
   Destination_Register_Address_out : out std_logic_vector(4 downto 0);
   Immediate_out : out std_logic_vector(15 downto 0);
   Jump_Address_out : out std_logic_vector(25 downto 0);
   Operation_Code_out : out std_logic_vector(5 downto 0);
   ALU_Function_Code : out std_logic_vector(5 downto 0);
   Shamt_out : out std_logic_vector(4 downto 0);
   Cyclic_Program_Counter_Data_out : out std_logic_vector(31 downto 0);
   Cyclic_Target_Register_Address_out : out std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_out : out std_logic_vector(4 downto 0);
   Load_Data_out : out std_logic_vector(31 downto 0);
   Execution_Result_out : out std_logic_vector(31 downto 0);
   Cyclic_Operation_Code : out std_logic_vector(5 downto 0);
   Clock : in std_logic );
end component ;
```

```
component PR2
Port ( Target_Register_Address_in : in std_logic_vector(4 downto 0);
   Destination_Register_Address_in : in std_logic_vector(4 downto 0);
   Program_Counter_Data_in : in std_logic_vector(31 downto 0);
   Source_Register_Data_in : in std_logic_vector(31 downto 0);
   Target_Register_Data_in : in std_logic_vector(31 downto 0);
   Shamt_in : in std_logic_vector(4 downto 0);
   Jump_Address_in : in std_logic_vector(25 downto 0);
   Sign_Extended_Data_in : in std_logic_vector(31 downto 0);
   Operation_Code_in : in std_logic_vector(5 downto 0);
   ALU_Function_Code_in : in std_logic_vector(5 downto 0);
   Cyclic_Target_Register_Address_out : out std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_out : out std_logic_vector(4 downto 0);
   Program_Couter_Data_out : out std_logic_vector(31 downto 0);
   Source_Register_Data_out : out std_logic_vector(31 downto 0);
   Target_Register_Data_out : out std_logic_vector(31 downto 0);
   Shamt_out : out std_logic_vector(4 downto 0);
   Jump_Address_out : out std_logic_vector(25 downto 0);
   Sign_Extended_Data_out : out std_logic_vector(31 downto 0);
   Operation_Code_out : out std_logic_vector(5 downto 0);
   ALU_Function_Code_out : out std_logic_vector(5 downto 0);
   Clock : in std_logic);
end component ;

component PR3
Port ( Cyclic_Target_Register_Address_in : in std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_in : in std_logic_vector(4 downto 0);
   Address_in : in std_logic_vector(31 downto 0);
   AllZero_in : in std_logic;
   Execution_Result_in : in std_logic_vector(31 downto 0);
   Target_Register_Data_in : in std_logic_vector(31 downto 0);
   ALU_Function_Code_in : in std_logic_vector(5 downto 0);
   Operation_Code_in : in std_logic_vector(5 downto 0);
   Cyclic_Target_Register_Address_out : out std_logic_vector(4 downto 0);
   Cyclic_Destination_Register_Address_out : out std_logic_vector(4 downto 0);
```

```
   Address_out : out std_logic_vector(31 downto 0);
   AllZero_out : out std_logic;
   Execution_Result_out : out std_logic_vector(31 downto 0);
   Target_Register_Data_out : out std_logic_vector(31 downto 0);
   ALU_Function_Code_out : out std_logic_vector(5 downto 0);
   Operation_Code_out : out std_logic_vector(5 downto 0);
   Link_Register_Data_in : in std_logic_vector(31 downto 0);
   Link_Register_Data_out : out std_logic_vector(31 downto 0);
   Clock : in std_logic);
end component ;

component cpu_register
Port ( Source_Register_Address : in std_logic_vector(4 downto 0);
   Target_Register_Address : in std_logic_vector(4 downto 0);
   Destination_Register_Address : in std_logic_vector(4 downto 0);
   Cyclic_Target_Register_Address : in std_logic_vector(4 downto 0);
   Program_Counter : in std_logic_vector(31 downto 0);
   Load_Data : in std_logic_vector(31 downto 0);
   Execution_Result : in std_logic_vector(31 downto 0);
   Operation_Code : in std_logic_vector(5 downto 0);
   Source_Register_Data : out std_logic_vector(31 downto 0);
   Target_Register_Data : out std_logic_vector(31 downto 0);
   Clock : in std_logic);
end component ;

component Sign_Extend_Unit
Port ( Immediate : in std_logic_vector(15 downto 0);
   Operation_Code : in std_logic_vector(5 downto 0);
   Sign_Extended_Data : out std_logic_vector(31 downto 0));
end component ;

component DataMemory;
Port ( Target_Register_Data : in std_logic_vector(31 downto 0);
   Execution_Result : in std_logic_vector(31 downto 0);
   Operation_Code : in std_logic_vector(5 downto 0);
   Load_Data : out std_logic_vector(31 downto 0)        );
```

```
end component ;
signal fetch : std_logic_vector(255 downto 0);
signal decode : std_logic_vector(360 downto 0);
signal execution : std_logic_vector(277 downto 0);
signal memory : std_logic_vector(277 downto 0);
signal write : std_logic_vector(255 downto 0);
begin
U_ALU : ALU port map
(  Source_Register_Data => execution(73 downto 42),
   Target_Register_Data => execution(105 downto 74),
   Sign_Extended_Data => execution(168 downto 137),
   Operation_Code => execution(174 downto 169),
   ALU_Function_Code => execution(180 downto 175),
   Shamt => execution(110 downto 106),
   AllZero => execution(213),
   Execution_Result => execution(245 downto 214));
U_ALU2 : ALU2 port map
(  Program_Counter => execution(41 downto 10),
   Sign_Extended_Data => execution(168 downto 137),
   Address_Input => execution(136 downto 111),
   Source_Register_Data => execution(73 downto 42),
   Operation_Code => execution(174 downto 169),
   ALU_Function_Code => execution(180 downto 175),
   PC_Output => execution(212 downto 181),
   Link_Register_Data => execution(277 downto 246));
U_IM : IM port map
(  Instruction_Address => fetch(31 downto 0),
   Instruction_Code => fetch(63 downto 32));
U_PC : PC port map
(  Program_Counter_Input => memory(41 downto 10),
   Operation_Code => memory(118 downto 113),
   ALU_Function_Code => memory(112 downto 107),
   AllZero => memory(42),
   Program_Counter_Output => fetch(31 downto 0),
   Clock => Clock,
   Reset => Reset);
```

```
U_PR0 : PR0 port map
(  Program_Counter_Data_in => fetch(31 downto 0),
   Instruction_Code_in => fetch(63 downto 32),
   Cyclic_Program_Counter_Data_in => memory(182 downto 151),
   Cyclic_Target_Register_Address_in => memory(4 downto 0),
   Cyclic_Destination_Register_Address_in => memory(9 downto 5),
   Load_Data_in => memory(150 downto 119),
   Execution_Result_in => memory(74 downto 43),
   Operation_Code_in => memory(118 downto 113),
   Program_Counter_Data_out => decode(31 downto 0),
   Sourcd_Register_Address_out => decode(36 downto 32),
   Target_Register_Address_out => decode(41 downto 37),
   Destination_Register_Address_out => decode(46 downto 42),
   Immediate_out => decode(62 downto 47),
   Jump_Address_out => decode(88 downto 63),
   Operation_Code_out => decode(94 downto 89),
   ALU_Function_Code => decode(100 downto 95),
   Shamt_out => decode(105 downto 101),
   Cyclic_Program_Counter_Data_out => decode(137 downto 106),
   Cyclic_Target_Register_Address_out => decode(142 downto 138),
   Cyclic_Destination_Register_Address_out => decode(147 downto 143),
   Load_Data_out => decode(179 downto 148),
   Execution_Result_out => decode(211 downto 180),
   Cyclic_Operation_Code => decode(217 downto 212),
   Clock => Clock);
U_PR2 : PR2 port map
(  target_Register_Address_in => decode(41 downto 37),
   Destination_Register_Address_in => decode(46 downto 42),
   Program_Counter_Data_in => decode(31 downto 0),
   Source_Register_Data_in => decode(249 downto 218),
   Target_Register_Data_in => decode(281 downto 250),
   Shamt_in => decode(105 downto 101),
   Jump_Address_in => decode(88 downto 63),
   Sign_Extended_Data_in => decode(313 downto 282),
   Operation_Code_in => decode(94 downto 89),
   ALU_Function_Code_in => decode(100 downto 95),
```

```
   Cyclic_Target_Register_Address_out => execution(4 downto 0),
   Cyclic_Destination_Register_Address_out => execution(9 downto 5),
   Program_Couter_Data_out => execution(41 downto 10),
   Source_Register_Data_out => execution(73 downto 42),
   Target_Register_Data_out => execution(105 downto 74),
   Shamt_out => execution(110 downto 106),
   Jump_Address_out => execution(136 downto 111),
   Sign_Extended_Data_out => execution(168 downto 137),
   Operation_Code_out => execution(174 downto 169),
   ALU_Function_Code_out => execution(180 downto 175),
   Clock => Clock);
U_PR3 : PR3 port map
(  Cyclic_Target_Register_Address_in => execution(4 downto 0),
   Cyclic_Destination_Register_Address_in => execution(9 downto 5),
   Address_in => execution(212 downto 181),
   AllZero_in => execution(213),
   Execution_Result_in => execution(245 downto 214),
   Target_Register_Data_in => execution(105 downto 74),
   ALU_Function_Code_in => execution(180 downto 175),
   Operation_Code_in => execution(174 downto 169),
   Cyclic_Target_Register_Address_out => memory(4 downto 0),
   Cyclic_Destination_Register_Address_out => memory(9 downto 5),
   Address_out => memory(41 downto 10),
   AllZero_out => memory(42),
   Execution_Result_out => memory(74 downto 43),
   Target_Register_Data_out => memory(106 downto 75),
   ALU_Function_Code_out => memory(112 downto 107),
   Operation_Code_out => memory(118 downto 113),
   Link_Register_Data_in => execution(277 downto 246),
   Link_Register_Data_out => memory(182 downto 151),
   Clock => Clock);
U_cpu_re : cpu_register port map
(  Source_Register_Address => decode(36 downto 32),
   Target_Register_Address => decode(41 downto 37),
   Destination_Register_Address => decode(147 downto 143),
   Cyclic_Target_Register_Address => decode(142 downto 138),
```

```
   Program_Counter => decode(137 downto 106),
   Load_Data => decode(179 downto 148),
   Execution_Result => decode(211 downto 180),
   Operation_Code => decode(217 downto 212),
   Source_Register_Data => decode(249 downto 218),
   Target_Register_Data => decode(281 downto 250),
   Clock => Clock);
U_Sign_Extend_Unit : Sign_Extend_Unit port map
(  Immediate => decode(62 downto 47),
   Operation_Code => decode(94 downto 89),
   Sign_Extended_Data => decode(313 downto 282));
U_DataMemory : DataMemory port map
(  Target_Register_Data => memory(106 downto 75),
   Execution_Result => memory(74 downto 43),
   Operation_Code => memory(118 downto 113),
   Load_Data => memory(150 downto 119));
process (Clock)
begin
        SRD <= execution(73 downto 42);
        TRD <= execution(105 downto 74);
        SED <= decode(313 downto 282);
        PC_PC <= fetch(31 downto 0);
        Result_Exe <= execution(245 downto 214);
        Result_Mem <= decode(211 downto 180);
end process;
end RTL;
```

4.2 シミュレーションおよびその考察

4.1 節での記述シミュレーションを行った。シミュレーションは Instruction Memory にアセンブラのプログラムを記述し、結果を出力し、その出力結果を Data Memory に格納してシミュレータのデータと照合するものとした。シミュレーションに用いたプログラムおよびその出力は次の通りである。各機能ブロック記述の段階で各々シミュレーションを行ったが、データパス記述において各パイプラインレジスタでのデータがクロックと同時に通り抜けてしまうエラーが発生した。

各パイプラインレジスタの記述に伝搬遅延を設定することによって対処し、その結果、第 3章で述べた設計条件をすべて満たしていることと、各ブロック毎で正常に動作していることを確認した。

4.2.1　オーバーフロー動作確認用アセンブラ

```
        命令コード                                  アセンブラコード
  "00000000000000000000000000000000",--NOP
  "00100000000000010000000000000001",--ADDI GPR0,GPR1,HEX[0001]
  "00111100000000101000000000000000",--LUI GPR0,GPR2,HEX[8000]
  "00100000000000011111111111111111",--ADDI GPR0,GPR3,HEX[FFFF]
  "00000000000000000000000000000000",--NOP
  "00000000000000000000000000000000",--NOP
  "00000000000000000000000000000000",--NOP
  "00000000100001101000000000100000",--ADD   GPR2,GPR3,GPR8
  "00000000100001101001000000100001",--ADDU GPR2,GPR3,GPR9
  "00000000100000101010000000100010",--SUB   GPR2,GPR1,GPR10
  "00000000100000101011000000100011",--SUBU GPR2,GPR1,GPR11
  "10101100000000010000000000000100",--SW    GPR1,1
  "10101100000000010000000000001000",--SW    GPR2,2
  "10101100000000011000000000001100",--SW    GPR3,3
  "10101100000010000000000000010000",--SW    GPR8,4
  "10101100000010010000000000010100",--SW    GPR9,5
  "10101100000010100000000000011000",--SW    GPR10,6
  "10101100000010110000000000011100",--SW    GPR11,7
```

このプログラムは加算命令と減算命令を行い、オーバーフローが正常に検出されることを確認するためのものである。DataMemory１にはオペランド１を、２にはオペランド２を、３にはオペランド３が入る。４、５にはオペランド２＋オペランド３の結果を、６、７にはオペランド２－オペランド１の結果が入る。なお５，７は符号無視の命令である。期待すべき結果と演算結果は以下の通りである。

期待すべき結果
ADD 命令, ADDU 命令

$$
\begin{array}{l}
\quad 11111111111111111111111111111111_2 \\
+\underline{10000000000000000000000000000000_2} \\
\quad 101111111111111111111111111111111_2 \ \text{Over Flow}
\end{array}
$$

SUB 命令, SUBU 命令

$$
\begin{array}{l}
\quad 10000000000000000000000000000000_2 \\
-\underline{00000000000000000000000000000001_2} \\
\quad 101111111111111111111111111111111_2 \ \text{Over Flow}
\end{array}
$$

このプログラムの出力結果は以下のとおりである。

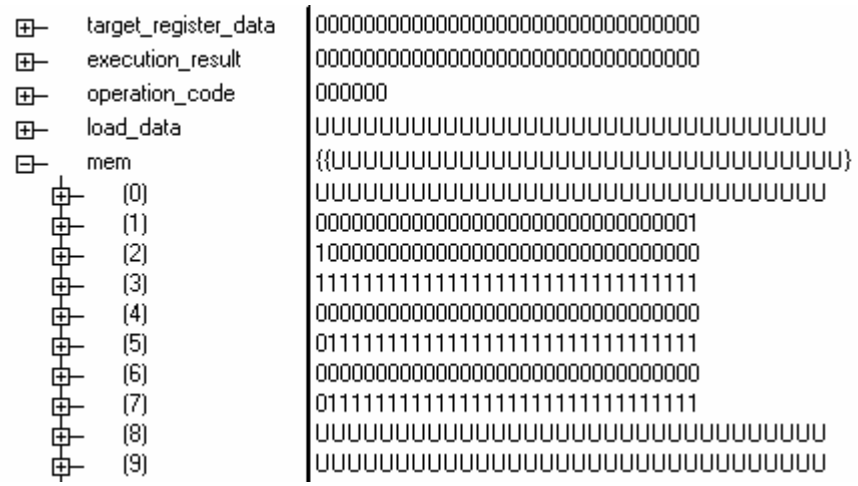


図 4.2　オーバーフロー検証プログラム結果

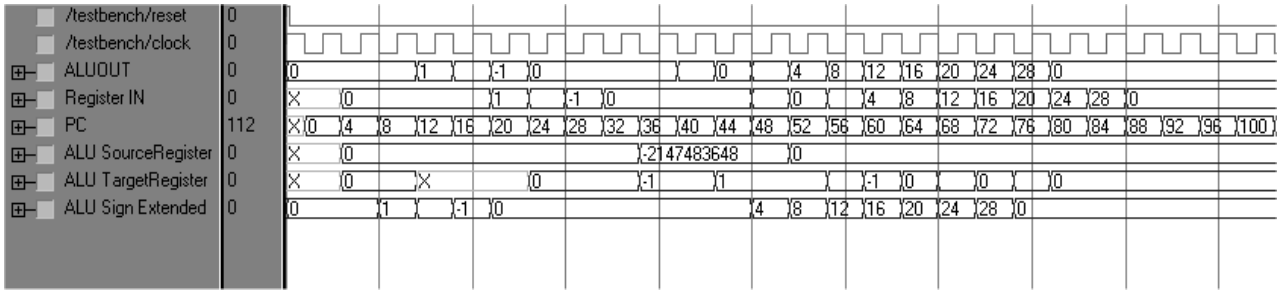


図 4.3　命令処理波形

このプログラムによってオーバーフローが正常に検出され出力が 0 になっていることが確認できた。また、符号無視型の演算ではオーバーフローが無視されて結果がそのまま出力され、正常に動作していることが確認でき

た。

4.2.2 ロード命令ストア命令動作確認用アセンブラ

このプログラムは、まずADDI命令でデーターを一度レジスタに格納し、それを SW 命令でメモリに出力し、再び出力したデータを別のレジスタに格納することで LW 命令、SW 命令の動作確認を行うものである。

```
"00000000000000000000000000000000",--NOP
"00100000000000010000000000000001",--ADDI GPR0,GPR1,1
"00100000000000010000000000000100",--ADDI GPR0,GPR2,100
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"10101100000000010000000000000100",--SW     GPR1,1
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"10001100000000011000000000000100",--Lw     GPR3,1
```

出力結果

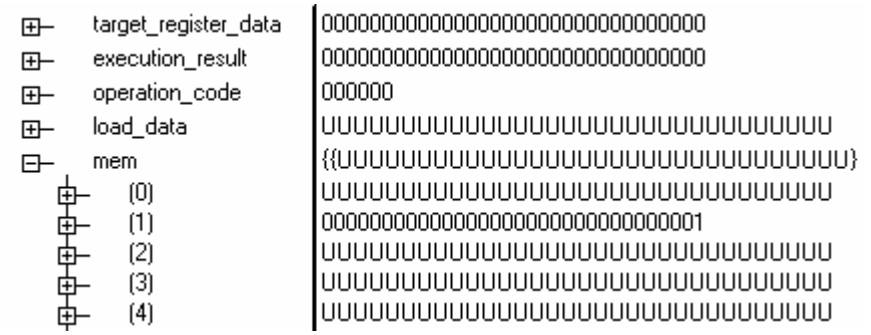


図 4.4　データメモリの状態

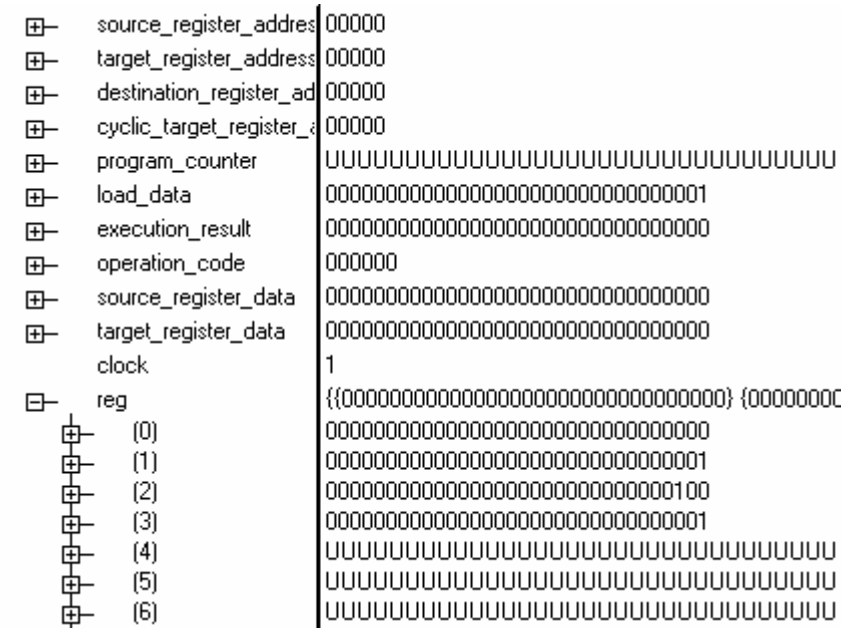


図 4.5　　レジスタの状態

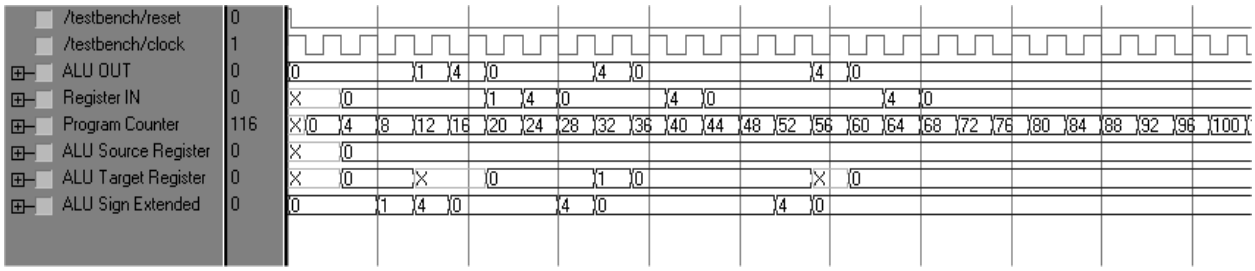


図 4.6　　命令処理波形

以上の図 4.5、図 4.6 よりレジスタ番号１に入力データ１が入力され、それがデータメモリ１番地に正常に格納され、レジスタ番号３が１になっていることから、LW 命令、SW 命令が正常に動作していることを確認した。

4.2.3　論理演算命令テスト用アセンブラ

命令コード　　アセンブラコード

"00000000000000000000000000000000",--NOP

```
"00100100000000011010101010101010",--ADDIU GPR0,GPR1,HEX[AAAA]
"00111100000000101010101010101010",--LUI GPR0,GPR2,HEX[AAAA]
"00100000000000110101010101010101",--ADDI GPR0,GPR3,HEX[5555]
"00100000000001001111111111111111",--ADDI GPR0,GPR4,HEX[FFFF]
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00000000010001001000000000100100",--AND  GPR1,GPR2,GPR8
"00110000010100110101010101010101",--ANDI GPR1,GPR9,HEX[AAAA]
"00000000010001101010000000100101",--OR   GPR1,GPR3,GPR10
"00110100001010110101010101010101",--ORI GPR1,GPR11,HEX[5555]
"00000000010000001100001000000000",--SLL GPR1,GPR12,HEX[4]
"00000000010000001101001000000010",--SRL GPR1,GPR13,HEX[4]
"00000000010001101110000000100111",--NOR GPR1,GPR3,GPR14
"00000000010010001111000000100110",--XOR GPR1,GPR4,GPR15
"00111000011000000000000011111111",--XORI GPR1,GPR16,HEX[00FF]
"00000000010001010001000000101010",--SLT GPR1,GPR2,GPR17
"00000000010001010010000000101011",--SLTU GPR1,GPR2,GPR18
"00101000010100110101010101010110",--SLTI GPR2,GPR19,HEX[5556]
"00101100010101000101010101010110",--SLTIU GPR2,GPR20,HEX[5556]
"00001100000000000000000000011010",--JAL PC[26]
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00000000000000000000000000000000",--NOP
"00001000000000000000000000000000",--J PC[0]
"10101100000000010000000000000100",--SW   GPR1,1
"10101100000000010000000000001000",--SW   GPR2,2
"10101100000000011000000000001100",--SW   GPR3,3
"10101100000001000000000000010000",--SW   GPR4,4
"10101100000001000000000000010100",--SW   GPR8,5
"10101100000001001000000000011000",--SW   GPR9,6
"10101100000001010000000000011100",--SW   GPR10,7
"10101100000001011000000000100000",--SW   GPR11,8
"10101100000001100000000000100100",--SW   GPR12,9
"10101100000001101000000000101000",--SW   GPR13,10
"10101100000001110000000000101100",--SW   GPR14,11
```

```
"10101100000011110000000000110000",--SW    GPR15,12
"10101100000100000000000000110100",--SW    GPR16,13
"10101100000100010000000000111000",--SW    GPR17,14
"10101100000100100000000000111100",--SW    GPR18,15
"10101100000100110000000001000000",--SW    GPR19,16
"10101100000101000000000001000100",--SW    GPR20,17
"10101100000111110000000001001000",--SW    GPR31,18
```

期待すべき結果

AND **命令**

0000AAAAh AND AAAA0000h = 00000000h

ANDI **命令**

0000AAAAh AND 0000AAAAh = 0000AAAAh

OR **命令**

0000AAAAh OR 00005555h = 0000FFFFh

ORI **命令**

0000AAAAh OR 00005555h = 0000FFFFh

SLL **命令**

0000AAAAh 4 **ビット左シフト** = 000AAAA0h

SRL **命令**

0000AAAAh 4 **ビット右シフト** = 00000AAAh

NOR **命令**

0000AAAAh NOR 00005555h = FFFF0000h

XOR **命令**

0000AAAAh XOR FFFFFFFFh = FFFF5555h

XORI **命令**

0000AAAAh XOR 000000FFh = 0000AA55h

SLT **命令**,SLTU **命令**

0000AAAAh < AAAA0000h = 00000001h

SLTI **命令**,SLTIU **命令**

0000AAAAh < 00005556h = 00000000h

シミュレーション結果は図 4.7**、図** 4.8 **のとおりである。**

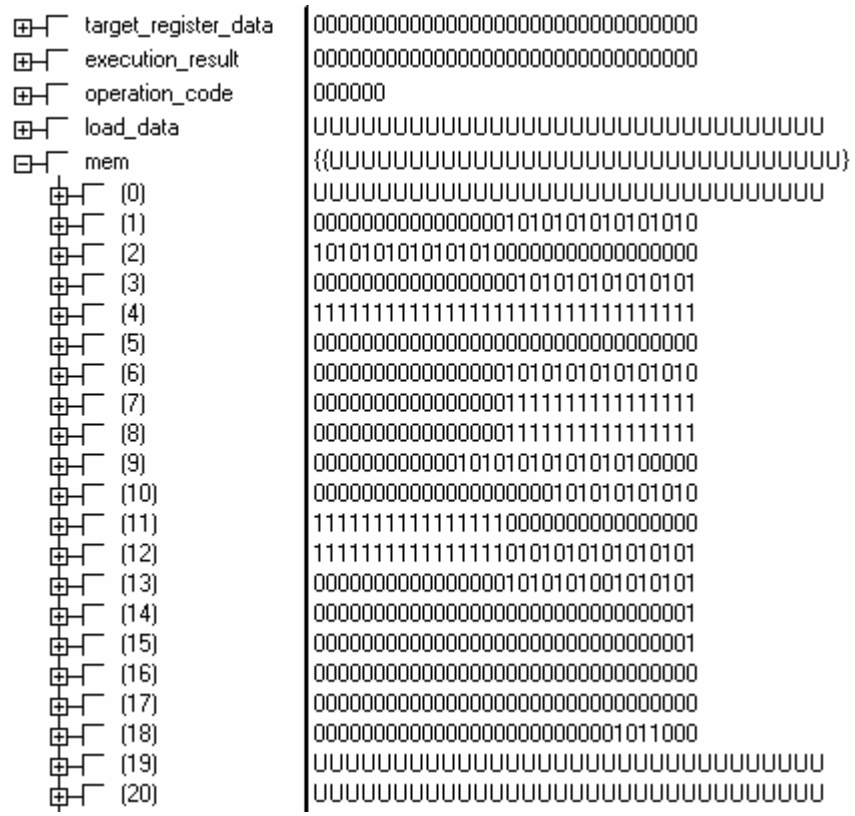


図 4.7　　論理演算シミュレーション結果

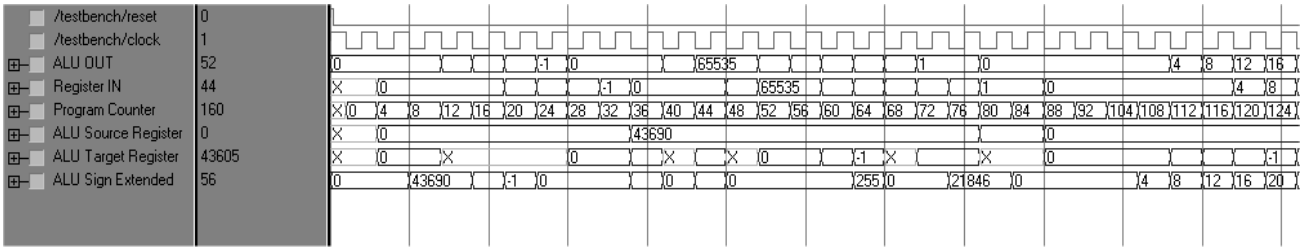


図 4.8　　命令処理波形

図 4.7 はデータメモリの状態を表したものであり、プログラムの SW 命令により各々の演算結果がデータメモリ出力されたものである。この結果からこのプログラムにおいてすべての命令は正常に動作していることが確認された。

また、図 4.8 の命令処理波形の Program Counter の値 92 で、次の値が 104 となっていることから Jump 命令も正常に動作していることが確認された。

第５章　まとめ

本研究での RISC プロセッサの設計では、今までの講義で得られた知識では不十分であったため、文献やゼミを通して必要な知識を取得するのに大変時間がかかってしまった。今回の設計は、今までの実験で経験したものよりはるかに回路や配線が複雑だったため、各ブロック間での動作確認を行っていても、全体として動作させるとさまざまなところにエラーが生じてしまった。VHDL 記述だけではなく、設計段階でミスがあると規模が大きいほど気付きにくい。かつそれが致命傷になりかねない。基礎段階での十分な設計とテストが必要であることを今回の研究を通して学んだ。また、VHDL による回路設計を習得できたことも大きな成果だと思う。レジスタ等のメモリアレイが大規模になってしまい、FPGA への実装は出来なかった。この点をふまえ、今後の課題は、回路設計においてその規模を十分に意識しながら設計を進めていくことである。

謝辞

全過程を通じてご指導に当たってくださった矢野政顕教授、適宜ご指導いただいた原央教授、橘昌良助教授に厚くお礼申し上げます。また、様々な点で協力してくださった矢野研究室、原研究室、橘研究室の諸氏に感謝します。

参考文献

[1],Gerry Kane、前川　守:"mips RISC アーキテクチャ",共立出版株式会社
[2],John L. Hennessy ， David A. Patterson ："コンピュータの構成と設計 第 2 版",日経 BP 社
[3], K.C.Chang :"Digital Systems Design with CHDL and Synthesis An Integrated Approach", IEEE Computer Society
芝山 潔："コンピュータアーキテクチャの基礎",近代科学社
[4],長谷川　裕恭:"VHDL によるハードウェア設計入門", CQ 出版社
[5],深山　正幸、北川　章夫、秋田 純一、鈴木　正國:"HDL による VLSI 設計", 共立出版株式会社
[6],田丸　啓吉、安浦　寛人:"マイクロコンピュータ",共立出版株式会社
[7],MIPS Technologies Inc. : "MIPS32 4K Processor Core Family Software User's Manual" http://www.mips.com/
[8],MIPS Technologies Inc. : "MIPS32 4K Processor Core Family Integrator's Guide", http://www.mips.com/

ADD命令　　符号あり算術加算命令

RS:オペランド1

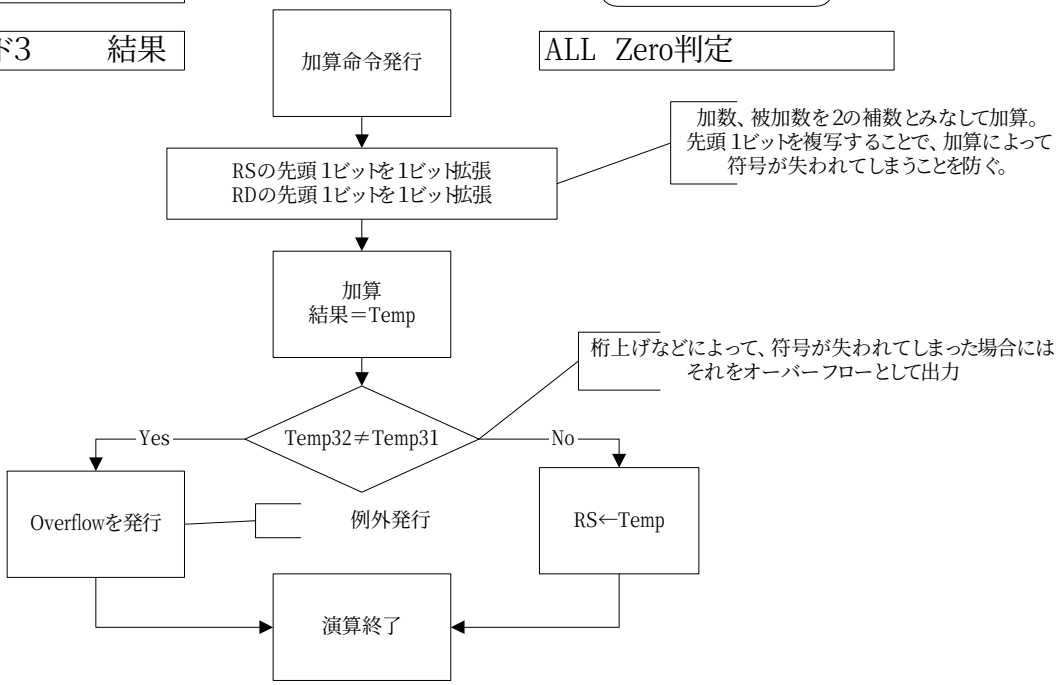
RT:オペランド2

RD:オペランド3　　結果

例外

オーバーフロー

ALL　Zero判定



ADDU命令　　符号なし算術加算命令

RS:オペランド1

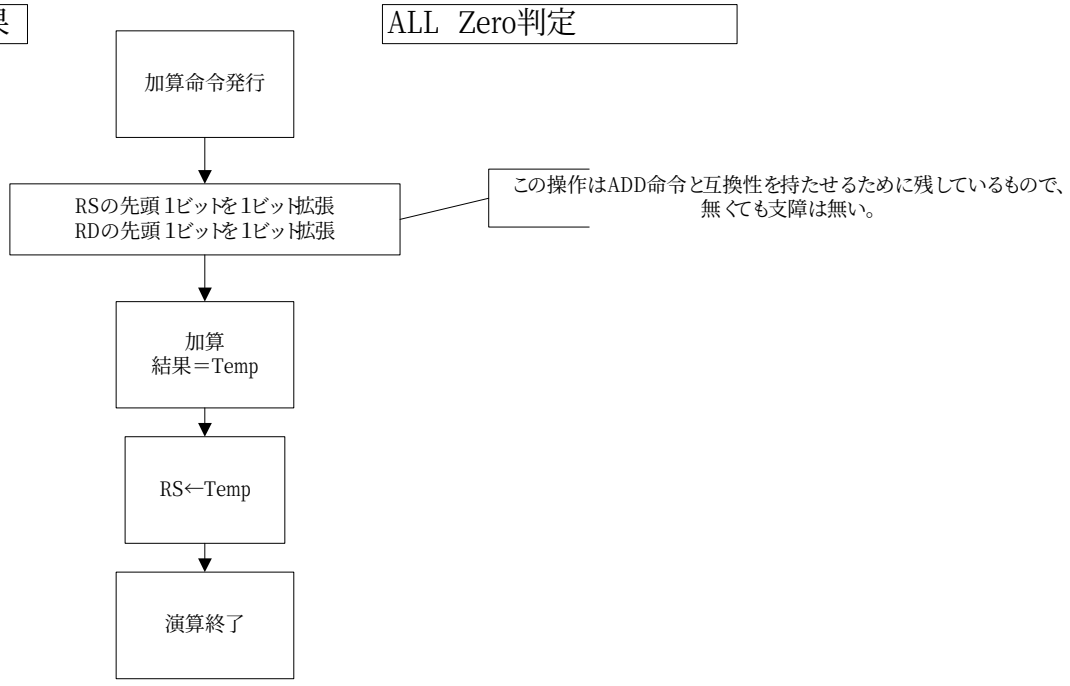
RT:オペランド2

RD:オペランド3　　結果

例外

なし

ALL　Zero判定



ADDI命令　　Immediate型符号あり算術加算命令

GPR[rs]: 加算される数

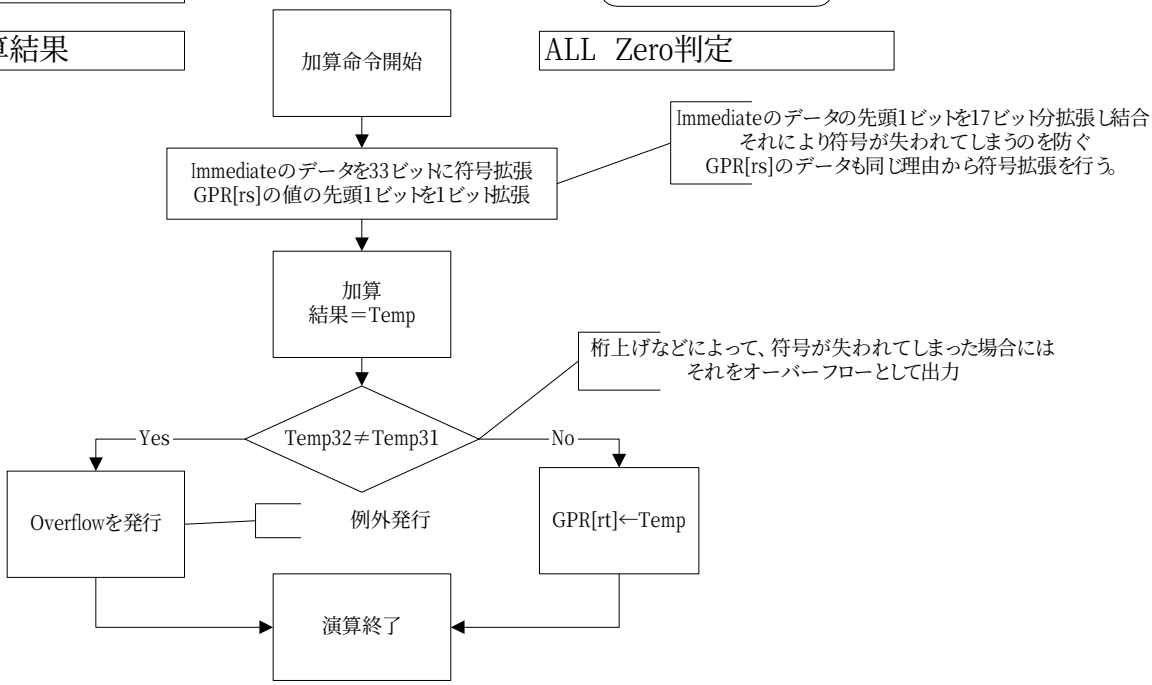
Immediate: 加算する数

GPR[rt]: 加算結果

例外

オーバーフロー

ALL Zero判定



ADDU命令　　Immediate型符号なし算術加算命令

GPR[rs]: 加算される数

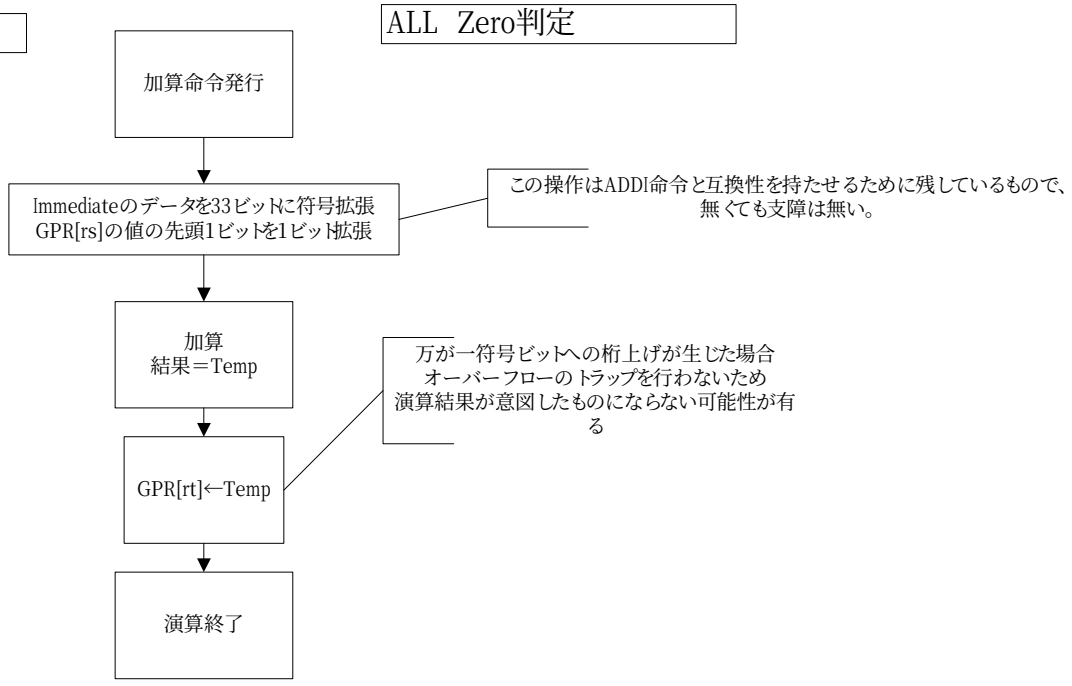
Immediate: 加算する数

GPR[rt]: 加算結果

例外

なし

ALL Zero判定



SUB命令　　符号あり算術減算命令

RS:オペランド1

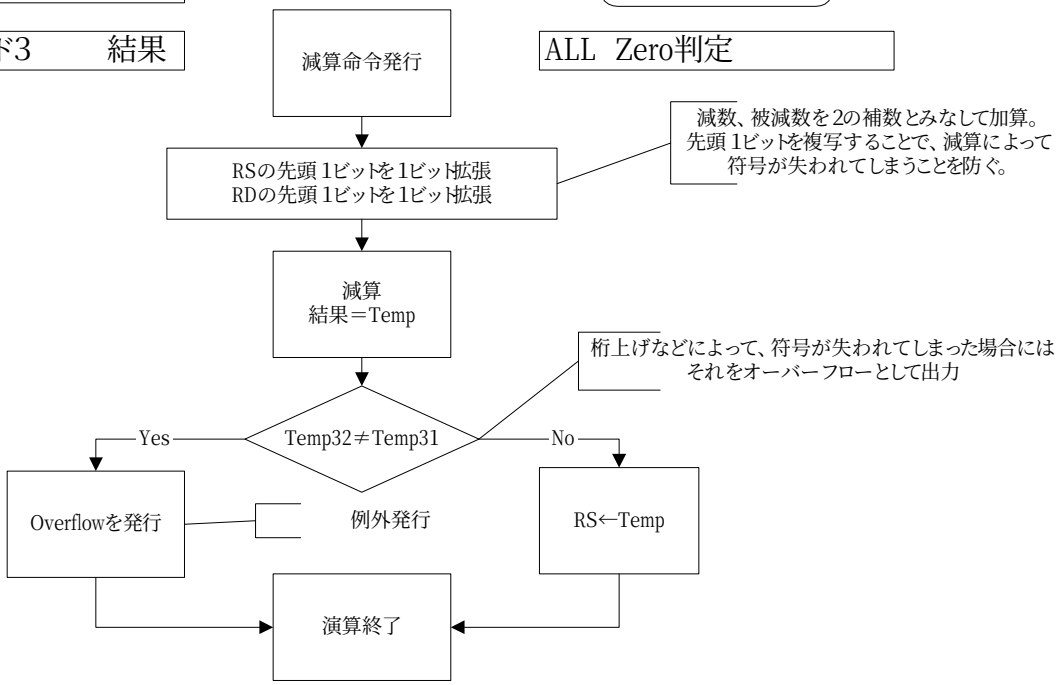
RT:オペランド2

RD:オペランド3　　結果

例外

オーバーフロー

ALL　Zero判定



SUBU命令　　符号なし算術減算命令

RS:オペランド1

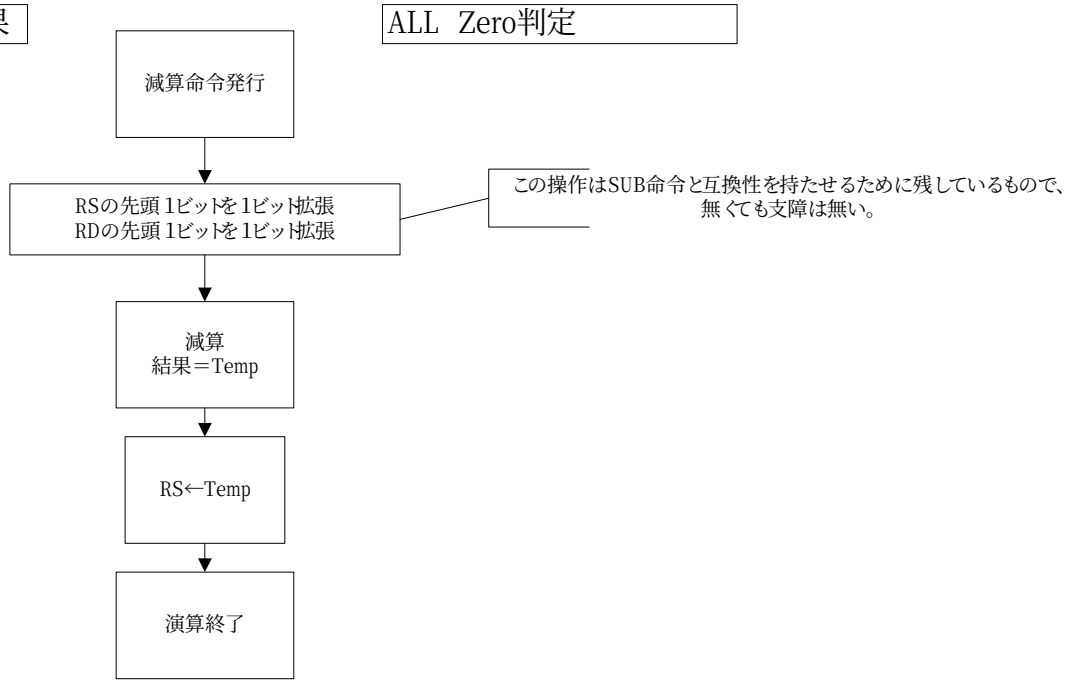
RT:オペランド2

RD:オペランド3　　結果

例外

なし

ALL　Zero判定



AND命令　　論理積命令

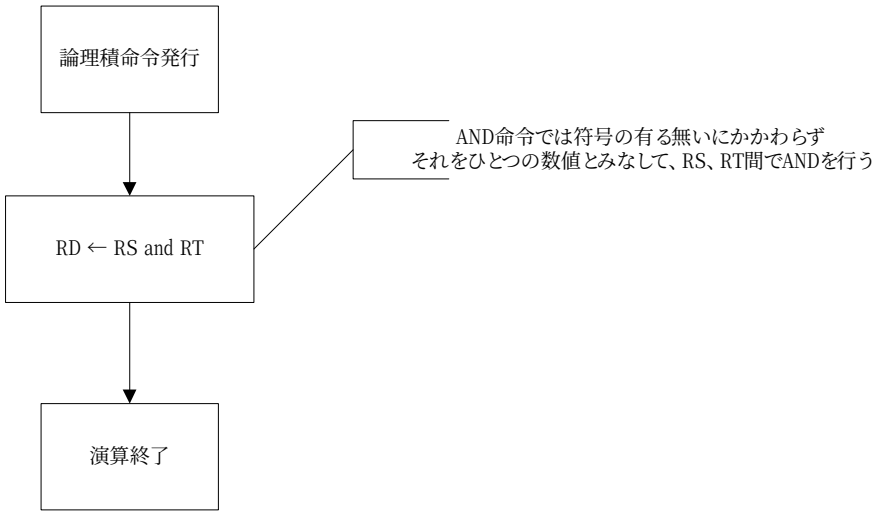
RS:オペランド1

RT:オペランド2

RD:オペランド3　　結果

フラグレジスタ変化

ALL ZERO



OR命令　　論理和命令

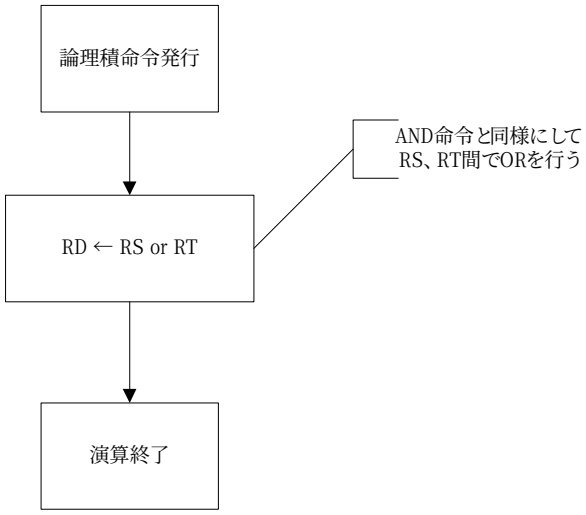
RS:オペランド1

RT:オペランド2

RD:オペランド3　　結果

フラグレジスタ変化

ALL ZERO



ANDI命令　　Immediate論理積命令

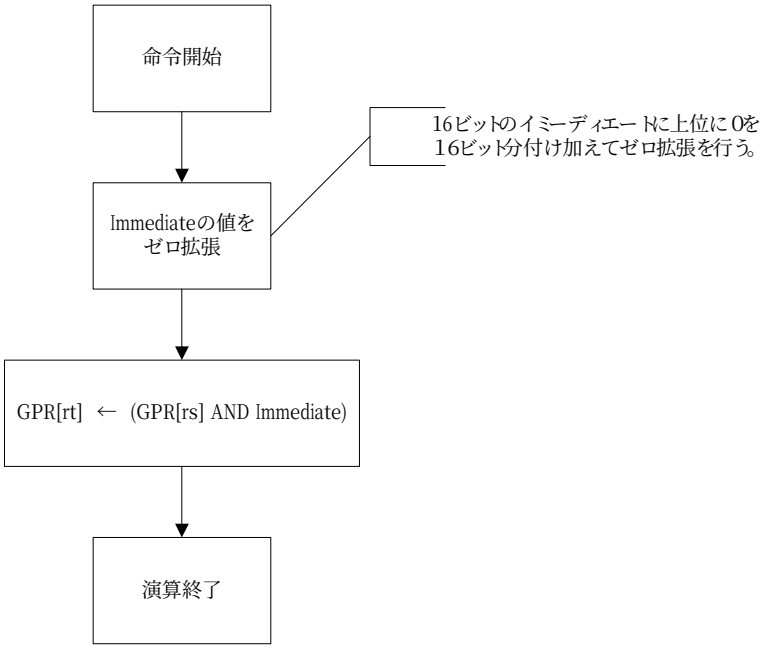
GPR[rs]:　オペランド1

Immediate:オペランド2

GPR[rt]:　結果

例外

なし



ORI命令　　Immediate論理和命令

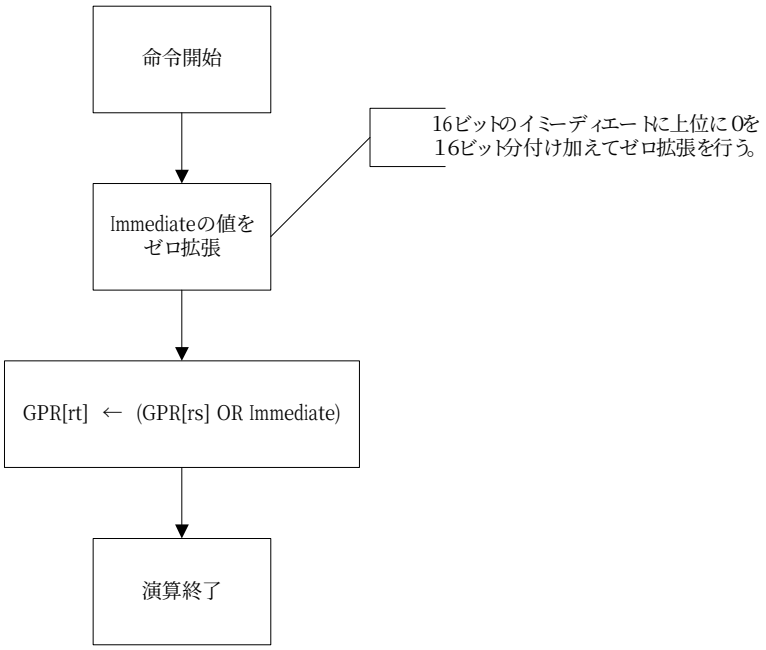
GPR[rs]:　オペランド1

Immediate:オペランド2

GPR[rt]:　結果

例外

なし



BEQ命令(Branch on Equal)　　等号条件分岐

GPR[rs]: 比較対象

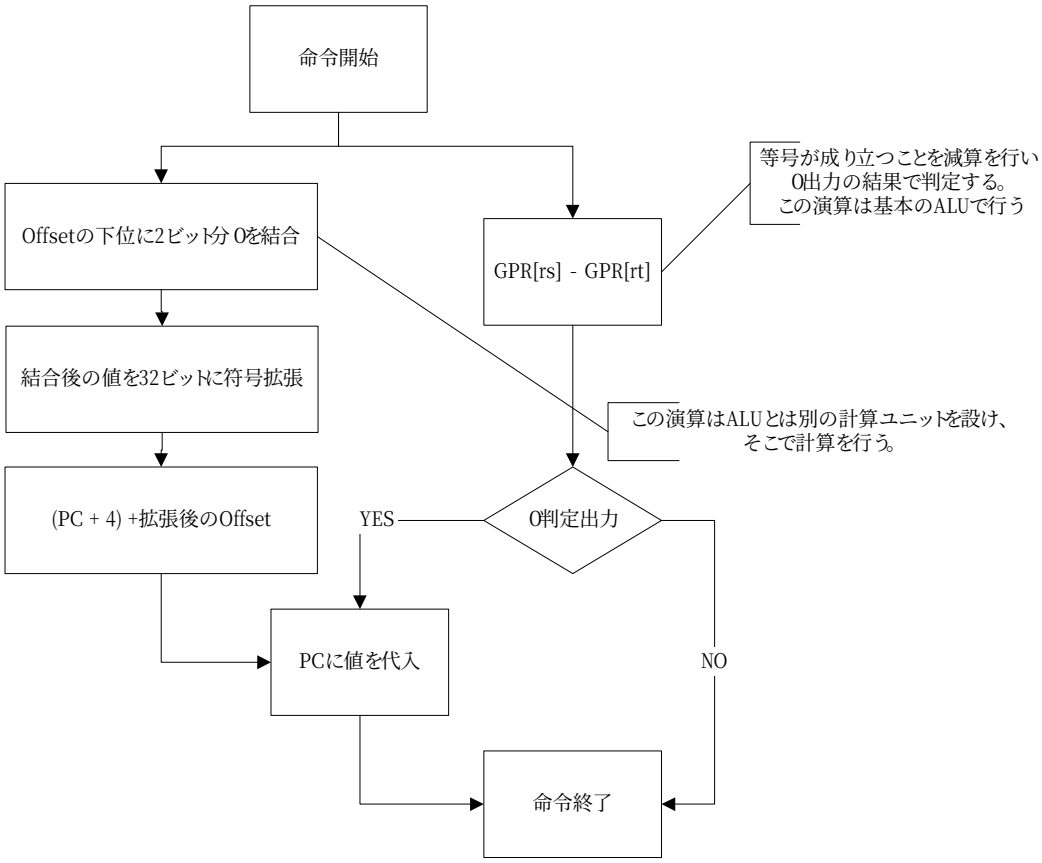
GPR[rt]: 比較対象

Offset: 分岐先

例外

なし

ALL Zero判定



BNE命令(Branch on Not Equal)　　不等号条件分岐

GPR[rs]: 比較対象

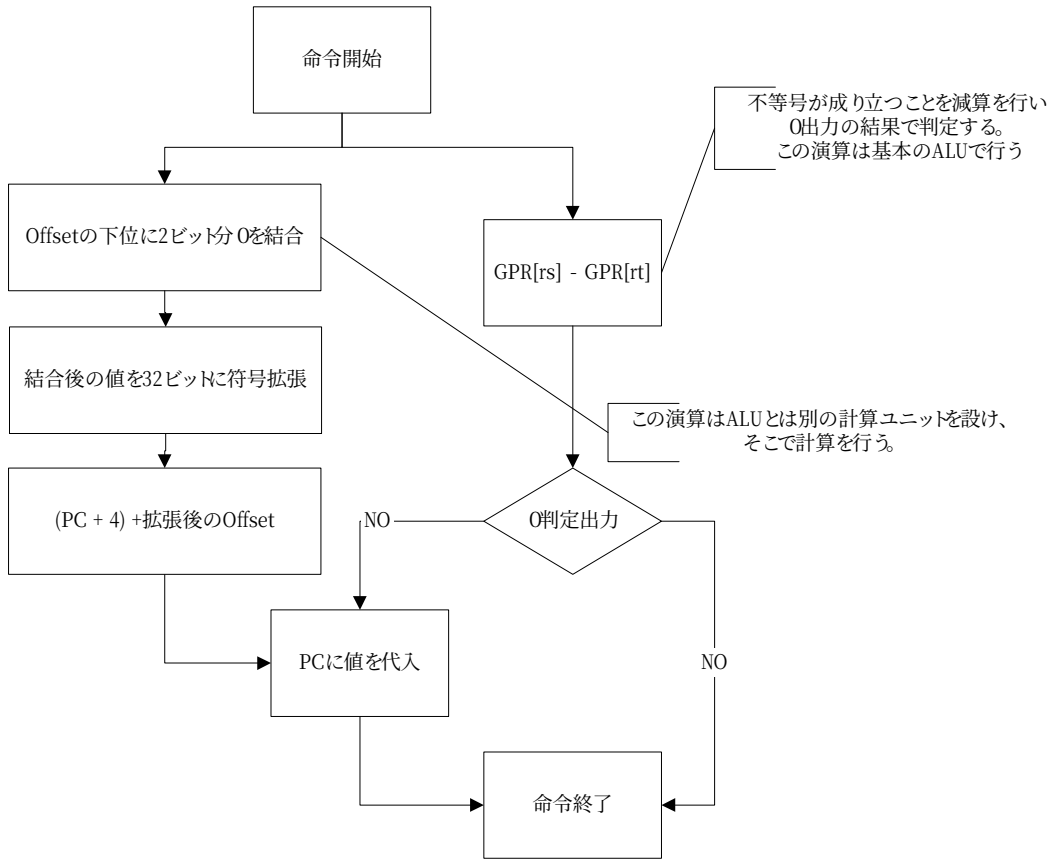
GPR[rt]: 比較対象

Offset: 分岐先

例外

なし

ALL Zero判定



SLT命令　　大小比較命令

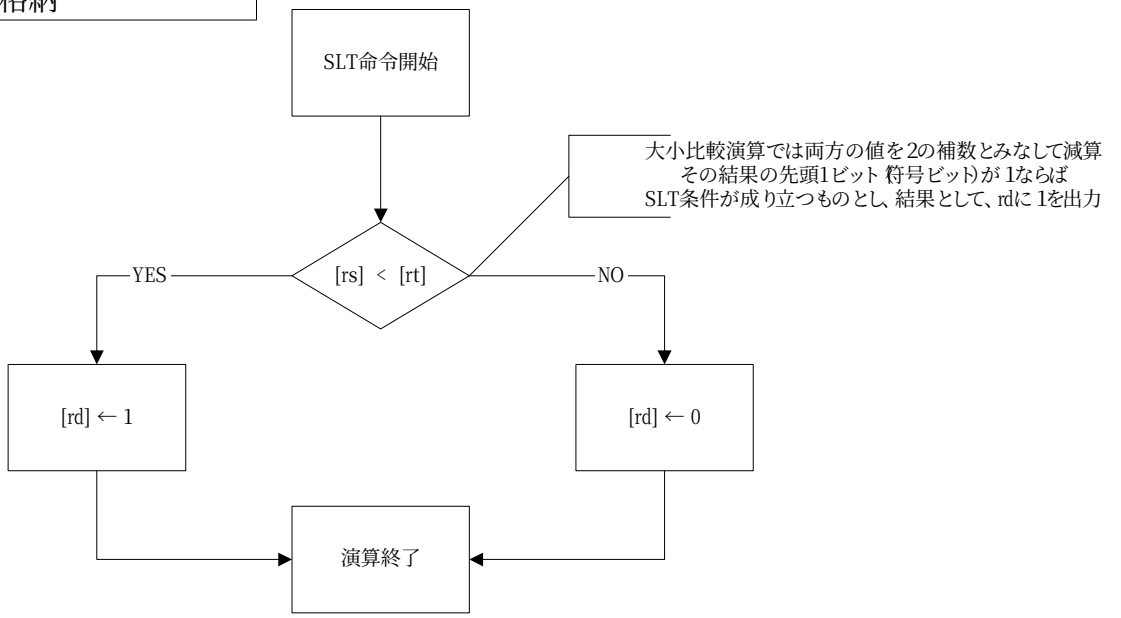
rs:比較される数

rt:比較する数

rd:結果格納

例外

なし



SLTU命令　　符号なし大小比較命令

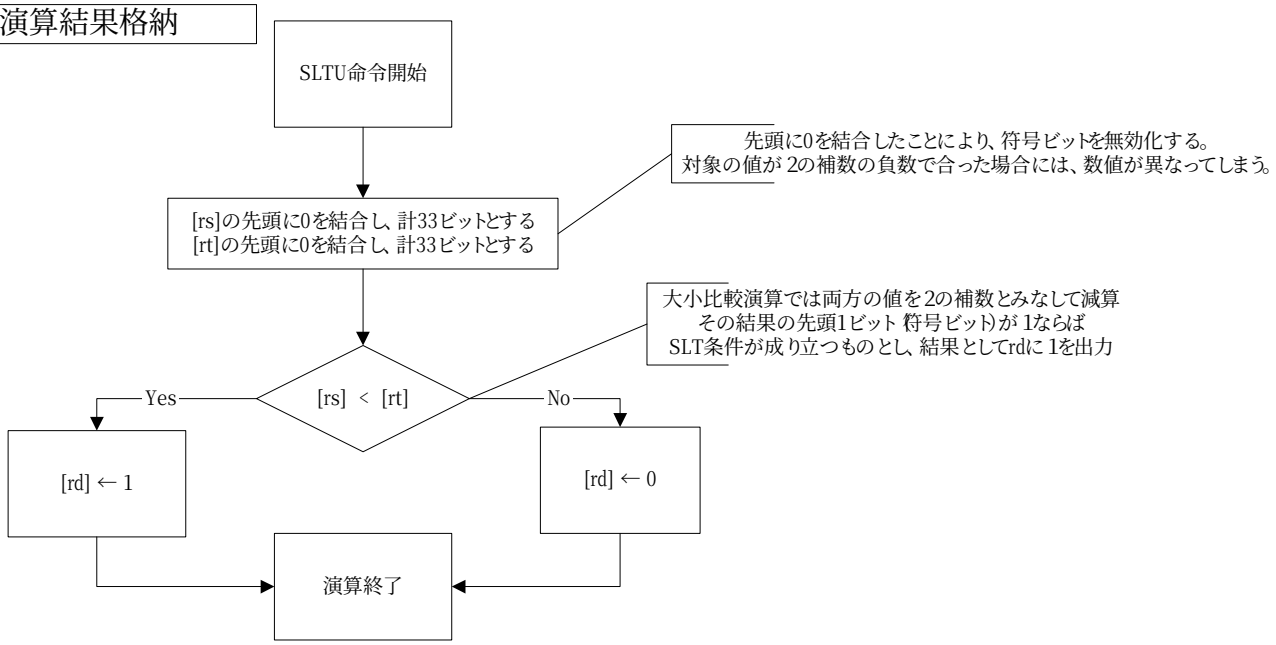
rs:比較される数

rt:比較する数

rd:比較演算結果格納

例外

なし

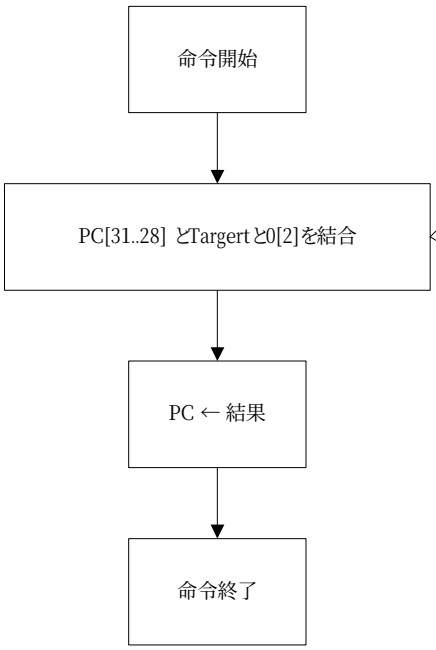


J命令（Jump）　　ジャンプ命令

Targert:　ジャンプアドレス

例外

なし



temp← PC[31..28] ¦¦ target ¦¦ 00

下位 2ビットに 0結合することで
アドレス例外発生を防ぐ

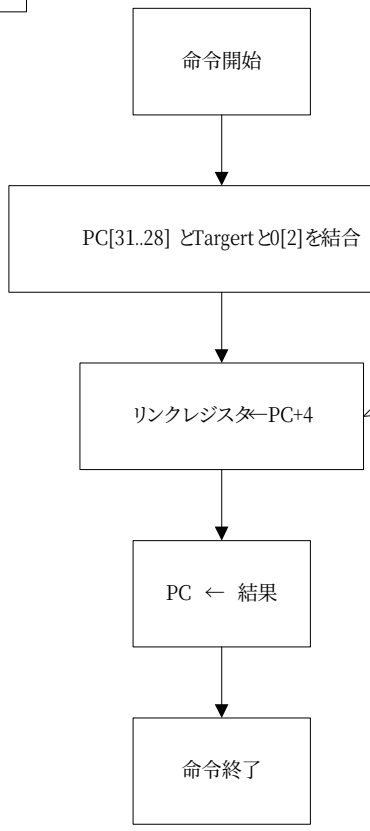
注意点
この命令の直後にJUMPもしくはBranch命令が有る場合
予期せぬ動作をする恐れがある。この場合、NOP命令にて
パイプラインをストールさせる必要がある。

JAL命令（Jump And Link）　　アドレス格納ジャンプ命令

Targert:　ジャンプアドレス

例外

なし



PCに4を足した値を代入することによって、
JR命令にて戻ってきた場合にもう一度この命令を
実行してしまうのを防ぐ。

リンクレジスタには汎用レジスタ31を使用する

注意点
この命令の直後にJUMPもしくはBranch命令が有る場合
予期せぬ動作をする恐れがある。この場合、NOP命令にて
パイプラインをストールさせる必要がある。

LW命令　　32ビットロード命令

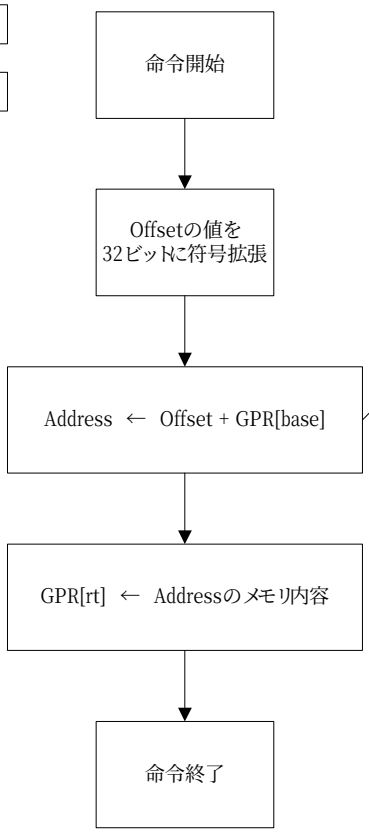
GPR[base]:OffsetBase

Offset:　　Offset

GPR[rt]:　　ロード結果

例外

アドレスエラー



下位2ビットが0で無い場合にはアドレス例外が発生するので、
下位2ビット分は0で固定し、アドレス例外が発生しないように設計する。

SW命令　　Immediate論理和命令

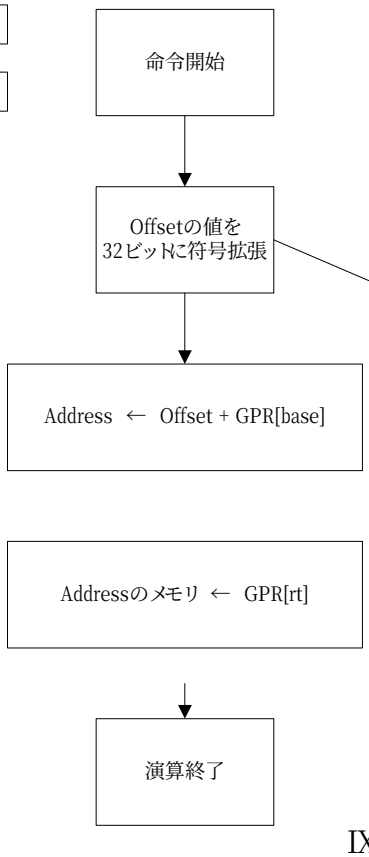
GPR[rs]:　オペランド1

Immediate:オペランド2

GPR[rt]:　　結果

例外

アドレスエラー



下位2ビットが0で無い場合にはアドレス例外が発生するので、
下位2ビット分は0で固定し、アドレス例外が発生しないように設計する。

SLL命令　　論理左シフト命令

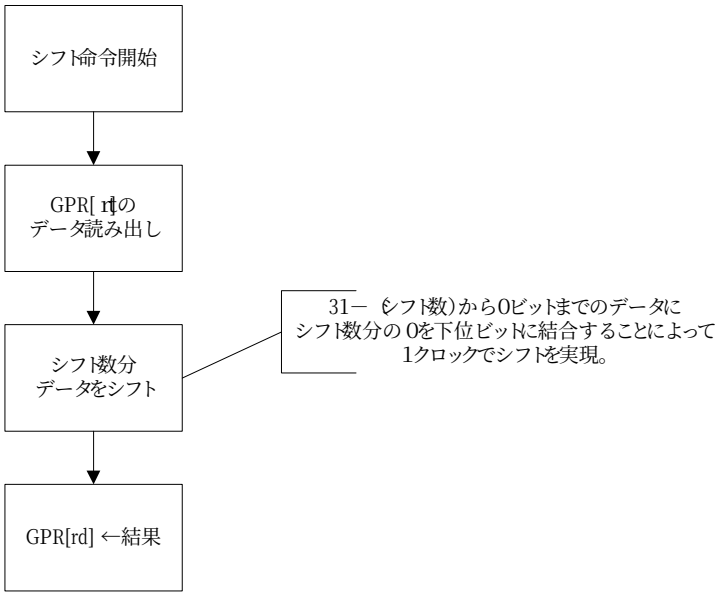
GPR[rt]:シフト対象データ

shamt:　シフト量

GPR[rd]:　シフト結果

例外

なし



SRL命令　　論理右シフト命令

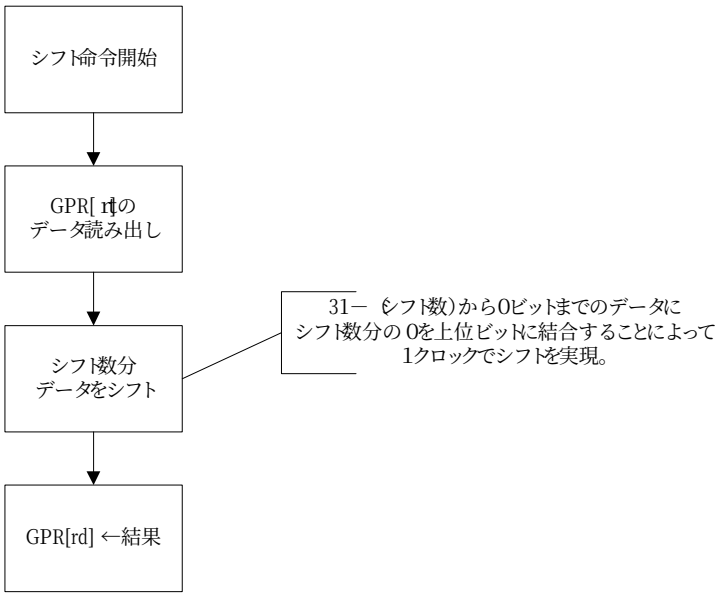
GPR[rt]:シフト対象データ

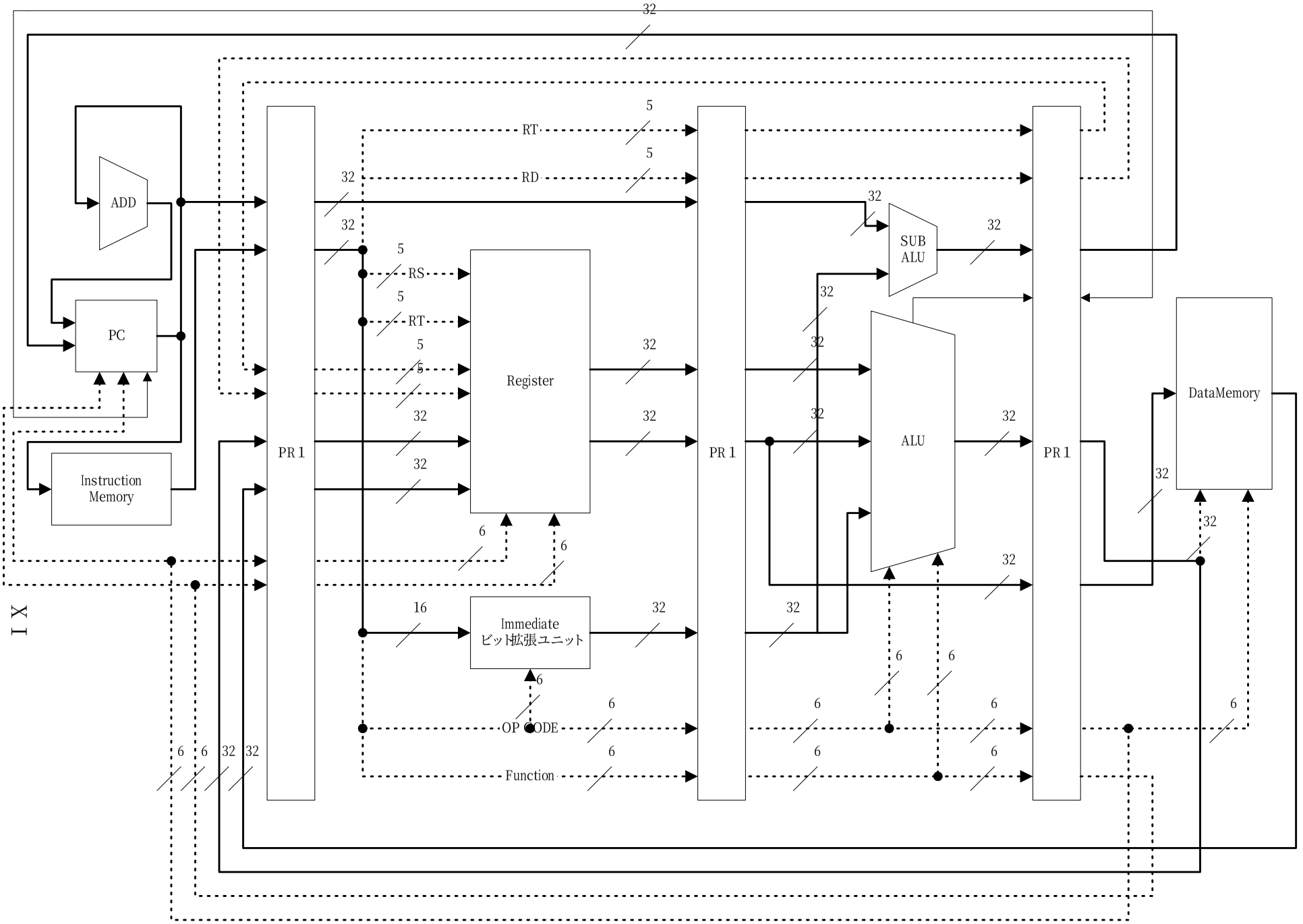
shamt:　シフト量

GPR[rd]:　シフト結果

例外

なし




32
ADD
PC
Instruction Memory
PR 1
RT
RD
RS
5
32
Register
Immediate
ビット拡張ユニット
16
6
OP CODE
Function
SUB ALU
ALU
PR 1
PR 1
DataMemory
IX