修 士 論 文

関数型プログラムの実行に適した
マルチスレッド型プロセッサ・アーキテクチャに関する研究

指導教官　日比野靖 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

伊藤英治

1997 年 2 月 14 日

要 旨

本研究では、マルチスレッド型プロセッサ・アーキテクチャと関数型プログラムの特徴を組み合わせることにより、高性能化を実現するプロセッサ・アーキテクチャを提案する。本プロセッサでは、プロセッサの各パイプライン・ステージをすべて異なるスレッドからの命令で埋める機構、各ハードウェア資源の多重化によって、データハザード、制御ハザード、および構造ハザードの発生を回避する。さらに、キャッシュミスが生じた場合でもパイプラインをストールさせない機構を加えることによって、高いスループットを実現する。

目 次

第 1 章

はじめに

MOS デバイスは、その物理寸法を縮小することによって、動作速度が高速になる性質を持つ。この性質と LSI 製造技術の進歩による MOS デバイスの微細化によって、近年のデバイスの動作速度は高速になっている。そして、デバイスの高速動作 [11][22] が、近年のプロセッサが高速なクロックで動作することを可能にしている。ただし、物理寸法を縮小しても配線遅延は減少しない [10][13]。この結果、プロセッサのパイプラインの各ステージの動作時間[1]が素子のスイッチング時間よりも配線遅延によって支配されることになり、「プロセッサの高性能化を実現するために動作クロックを高速化する」ということが困難な状態になる [7]。

この点を解消する手段として、プロセッサのパイプライン・ステージを細分化することによって配線遅延を減少させ、動作クロックの高速化を図るスーパーパイプライン方式が考えられる [15]。しかし、パイプライン・ステージを細分化すると、単一ストリームから命令を発行する限り、命令間の依存関係によってパイプラインの持つ性能を引き出すことができないという問題が生じる。

この問題を解決する方法の 1 つとして、マルチスレッド型プロセッサがある [5] [4] [14] [20] [18]。マルチスレッド型プロセッサは、単一のプロセッサで複数の命令ストリーム（スレッド）を実行するプロセッサである。その特徴として、複数のスレッドを独立に制御するために、複数のプログラムカウンタと各種状態レジスタを持ち、複数のスレッドが機能ユニットを共有して命令を実行する形態をとる。これによって、命令発行の抑止されたスレッドに代わり、他のスレッドから命令を発行することによってパイプラインのスルー

[1] 前のラッチの出力から次のラッチの入力に信号の変化分が表れるまでの時間

プットを向上させることができる。また、マルチスレッド型プロセッサでは、レジスタ類（プログラムカウンタやレジスタファイルなど）をスレッド数分必要とするが、集積度の向上によってチップ上の素子数が増大しているので設計上問題はない [28]。

これらの点から、今後LSI 製造技術の進歩を生かすためのプロセッサ・アーキテクチャは、マルチスレッド型プロセッサ・アーキテクチャであると考えられる。

本論文では、マルチスレッド型プロセッサ・アーキテクチャを、並列処理構造を持つ関数型プログラム [2] の実行に適したアーキテクチャにすることによって、簡単なハードウェアで実現することが可能な並列処理のための高性能なプロセッサ・アーキテクチャを提案する。

2 章ではマルチスレッド型プロセッサと関数型プログラムに関して説明を行ない、3 章では今回提案するマルチスレッド型ウルトラパイプライン・プロセッサ・アーキテクチャを説明する。4、5 章では性能見積りを行なうための具体的な MUP の設計例と設計した MUP の性能見積りを行なう。最後に 6 章でまとめる。

第 2 章

マルチスレッド型プロセッサ・アーキテクチャと関数型プログラム

本章では、マルチスレッド型プロセッサ・アーキテクチャの概要と、関数型プログラムの特徴について述べた後、関数型プログラムの特徴がどのようにマルチスレッド型プロセッサ・アーキテクチャに生かされるのかを説明する。

2.1 マルチスレッド型プロセッサ・アーキテクチャ

2.1.1 マルチスレッド処理の概要

近年の高性能プロセッサは、動作クロックを非常に高速化すると共に、命令レベルでの並列処理を行なうことによって性能向上を図っている。これらは、パイプラインを細分化して動作クロックを高速化するスーパーパイプライン方式、命令を並列実行するためのスーパースカラ方式および VLIW 方式をプロセッサに導入することによって実現されている [6]。

しかし、単一ストリームしか扱うことができない従来のプロセッサでは、命令間の依存関係やメモリシステムへのアクセスによって命令の並列実行が阻害され、プロセッサが持つ本来の性能を引き出すことができない。

この例として、パイプライン・プロセッサの場合について説明する。図 2.1にパイプラインを示す。この図に示すように、命令 1 と命令 2 の間に依存関係が存在すると（図中で

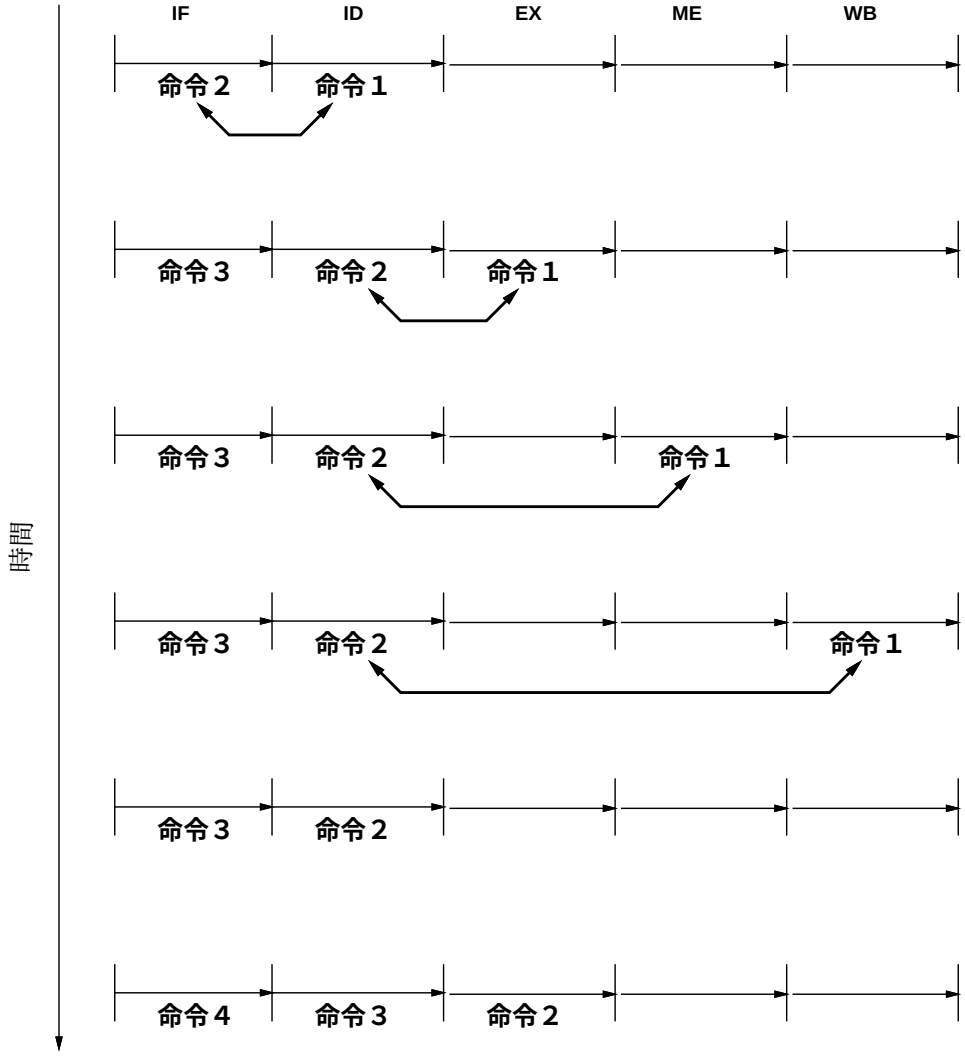

図 2.1: パイプライン・バブル

は依存関係を矢印で表している)、この依存関係が解消されるまでは命令 2 はパイプライン中でストールすることになる。このことによって、命令 1 と命令 2 の間にパイプライン・バブルが発生することになり、プロセッサが持つ本来の並列度を引き出すことができなくなるという問題が生じる。

この問題を解消する手段としてマルチスレッド処理方式がある。この方式は、単一プロセッサにおいて複数のストリーム（スレッド）を並列実行することにより、パイプライン・ハザードを回避し、パイプラインのスループットを向上する方法である。

この例を、前と同様にパイプライン・プロセッサのパイプラインを使って説明する。図

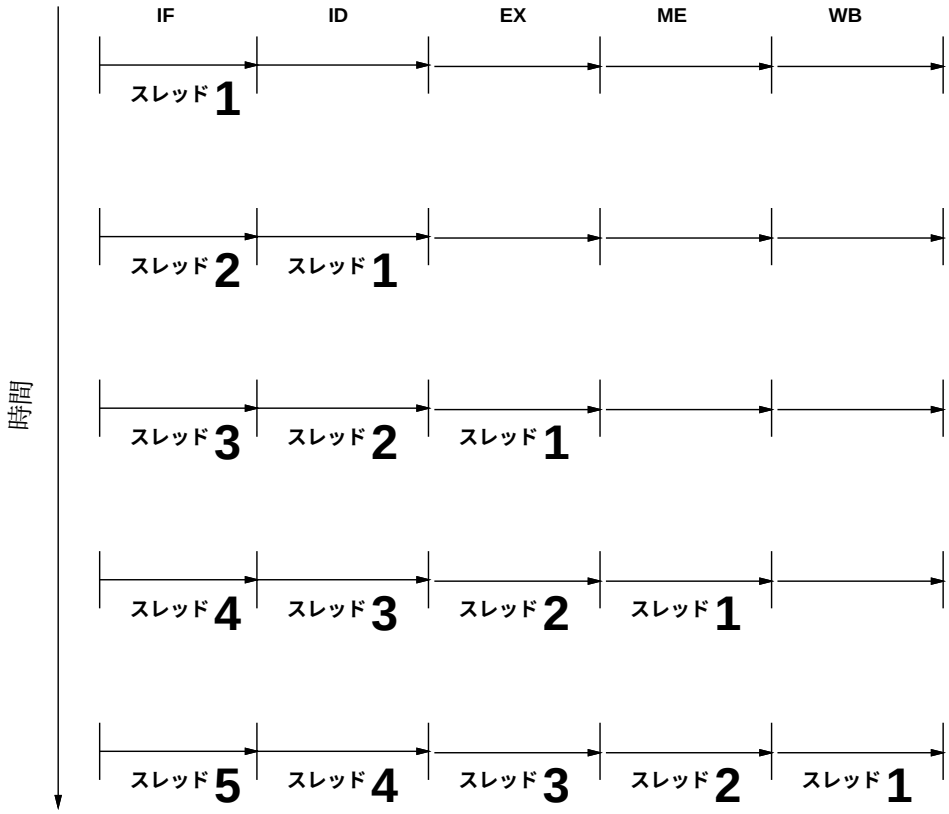

図 2.2: パイプラインのスループット向上

2.2にマルチスレッド処理方式を行なうパイプライン・プロセッサのパイプラインを示す。この図に示すように、毎サイクル、異なるスレッドから命令を発行すると、異なるスレッドから発行された命令間には依存関係が存在しないので、パイプラインの各ステージをすべて有効な命令で埋めることができる。これによって、パイプラインの並列性を生かすことが可能となる。

近年、このマルチスレッド処理を導入したプロセッサ・アーキテクチャの研究が行なわれている [14] [18] [20] [26]。これらマルチスレッド型プロセッサのアーキテクチャは、スレッド毎に独立した命令制御を可能にするために、各スレッドに対応したプログラム・カウンタと各種状態レジスタを持ち、機能ユニットを複数のスレッドにより共有する形態をとる。これによって、命令発行の抑止されたスレッドに代わって、別のスレッドから命令発行を行なうことにより、パイプラインのスループット向上を図ることが可能となる。

マルチスレッド処理では、粗粒度並列を活用したスレッド単位での並列処理を行なう。

そこで、命令レベルの並列処理とマルチスレッド処理を組み合わせることにより、従来の命令レベルの並列処理だけでは得られなかった大きな並列度を引き出すことが可能になる。

命令レベルの並列処理とマルチスレッド処理とを組み合わせたプロセッサには次の3つの方式がある。

- マルチスレッド型スーパースカラプロセッサ・アーキテクチャ[18][26]
- マルチスレッド型VLIWプロセッサ・アーキテクチャ [20]
- マルチスレッド型スーパーパイプラインプロセッサ・アーキテクチャ[14]

次節にそれぞれの方式についてまとめる。

2.1.2 マルチスレッド処理と命令レベル並列処理を組み合わせたプロセッサ・アーキテクチャ

マルチスレッド型スーパースカラ・プロセッサ・アーキテクチャ

スーパースカラ・プロセッサ・アーキテクチャは、多数の独立した機能ユニットを搭載し、1サイクル当たりに複数の機能ユニットへ命令を発行することにより並列実行を行なう。複数の機能ユニットは、常に100%の稼働状態ではないため、マルチスレッド処理を導入することにより、機能ユニットの稼働率を高めることが可能である。

しかし、スーパースカラ・アーキテクチャは、実行時に命令の並列性を抽出するために、命令間の依存関係をすべて検出する必要がある。さらに、マルチスレッド処理を行なう場合には、各スレッド毎にこれら依存関係の検出を行なった上で、複数のスレッドに対して機能ユニットの調停を行なう必要がある。

このように、スーパースカラ・アーキテクチャでは、マルチスレッド処理に必要なハードウェアが増大するため、クロック速度の向上を妨げる可能性がある。

マルチスレッド型VLIWプロセッサ・アーキテクチャ

VLIWプロセッサ・アーキテクチャでは、並列実行可能なオペレーションの集まりである長形式の命令（VLIW命令）を対応する機能ユニットにおいて並列実行する。

VLIW プロセッサでマルチスレッド処理を行なう場合、あるスレッドの VLIW 命令中のオペレーションから使用しない（NOP 指示の）機能ユニットを他のスレッドの VLIW 命令のオペレーション実行に割り当てる。これにより、スーパースカラ・アーキテクチャと同様に、機能ユニットの稼働率を高めることが可能になる。

VLIW アーキテクチャにおいてマルチスレッド処理を行なうためには、複数のスレッドに対して機能ユニットの調停を行なう必要がある。さらに、VLIW 命令をオペレーション毎に分離して命令発行・命令保持を行なう機構が必要となる。これらのことから、スーパースカラ・アーキテクチャと同様にハードウェアが複雑になり、クロック速度の向上することが難しくなる可能性がある。

マルチスレッド型スーパーパイプライン・プロセッサ・アーキテクチャ

スーパーパイプライン・プロセッサ・アーキテクチャは、パイプラインの処理単位を細分化し、動作クロックを高める方式である。パイプラインを多段化するため、従来のパイプラインよりも命令間の依存関係によるペナルティが大きくなるが、マルチスレッド処理を組み合わせることにより、パイプラインのスループットを大幅に向上することが可能である。さらに、スーパースカラ・アーキテクチャや VLIW アーキテクチャとは異なり、命令間の依存関係を調べる複雑なハードウェアを必要としないので、クロック速度の向上を妨げられない。しかし、パイプラインの処理段数の増加に伴って、パイプライン処理遅延を隠すために多くのスレッドを必要とする。

ただし、マルチスレッド型パイプラインプロセッサ・アーキテクチャのこの欠点は、2.2.1 節で述べる関数型プログラムの特徴により解決することが可能である。

2.2 関数型プログラム

2.2.1 関数型プログラムの特徴

関数型プログラムは並列処理に適した 2 つの特徴を持つ [2]。この 2 つの特徴を以下に示す。

1. “並列実行” と “パイプライン実行” という 2 つの並列処理構造

- 並列実行

関数間にデータ依存関係がなければその関数同士を並列に実行することがが可能である。

- パイプライン実行

  先に求まった値を被適用側の関数に送ってその実行を先行させ、また被適用側の関数で求められた値も直ちに適用側の関数に返し、適用側の実行を続行させることによって、関数適用側の実行と関数被適用側の実行を並行させることができる。

2. 関数活性体の増大

関数型プログラムを実行すると、1 つの関数が複数箇所で起動[1] されることになる。この結果、関数活性体同士は並列実行可能であるから、並列実行可能なストリームを多数供給することができることになる。

2.2.2 関数型プログラムの特徴を生かすための研究

関数型プログラムの特徴を生かすためにいくつかの研究が行なわれている。

コンパイラ技術の研究として、1 つのプログラムから多数のスレッドを生成するためのコンパイラ [25] [21] が提案されている。

また、先に述べた関数型プログラムの特徴を生かして並列処理を行なうコンピューターシステムの研究として、D-RISC[3]、GRIP[8] などが提案されている。しかし、関数型プログラムの特徴を生かして並列処理を行なう従来のコンピューターシステムは、マルチプロセッサで構成されており、プログラムの並列性をすべて生かすために複雑なシステムになっている。

2.3 マルチスレッド型プロセッサ・アーキテクチャと関数型プログラムとの関係

2.2.1節で述べたように、関数プログラムの特徴は、“多数の並列実行可能なストリームを供給することが可能である” ということである。一方、マルチスレッド型プロセッサは、

[1] 起動された関数を関数活性体と呼ぶ

プロセッサ自身が持つ性能（パイプラインの高いスループット）を引き出すために、複数のスレッドを必要とする。

そこで、マルチスレッド型プロセッサに対して多数のスレッドを供給することによって、マルチスレッド型プロセッサの “パイプライン制御の簡単化” および “動作クロックの高速化による高スループット” を実現することが可能になる。

第 3 章

マルチスレッド型ウルトラパイプライン・プロセッサ・アーキテクチャ (Multithreaded Ultrapipeline Processor architecture: MUP)

3.1 マルチスレッド型パイプラインプロセッサ・アーキテクチャの関数プログラムへの適用

マルチスレッド型パイプライン・プロセッサ・アーキテクチャに対して、関数型プログラムの特徴である “多数のスレッドを供給することができる” という点を生かすと以下に示す 2 つのことが可能になる。

1. パイプラインを長くすることが可能

長いパイプラインに対して、命令を供給することができるスレッドの数が少ない場合、同一スレッドから発行する命令と命令の間に挟む他のスレッドの命令が少なくなる。つまり、先行命令との依存関係が解消されないうちに、後続命令を発行することになる。この結果、同一スレッドの命令間の依存関係を解消するためにパイプラインをストールし、パイプライン中にバブルが生じることになる。

このように、長いパイプラインに対してスレッド数が少ないと、パイプラインの持つ性能を引き出すことができないことになる。

これに対して、関数型プログラムのように十分な数のスレッドを長いパイプラインに対して供給することができる場合、同一スレッドから発行する命令と命令の間に挟む他のスレッドの命令が多くなる。これは、先行命令との依存関係が解消されてから後続命令を発行することになる。この結果、パイプラインの各ステージをすべて有効な命令で満たすことができ、パイプラインの性能を引き出すことが可能になる。

このように、多数のスレッドを供給することができるならばパイプラインを長くすることが可能であるため、パイプラインの各ステージを非常に細分化し、プロセッサの動作クロックの高速化によりパイプラインのスループットを向上させることができる。

2. 各パイプライン・ステージをすべて異なるスレッドからの命令で埋めることが可能

関数型プログラムが、プロセッサのパイプライン・ステージ数以上のスレッドを提供するならば、各パイプライン・ステージをすべて異なるスレッドから発行された命令で埋めることができる。

このことによって、プロセッサは1サイクル1命令の速度で命令を実行することが可能になる。

また、パイプライン中のすべての命令が異なるスレッドから発行されたものであれば、互いに依存関係を持たないのでデータハザードや制御ハザードが発生することがない。このため、パイプラインの制御を非常に簡単化することができる。

3.2 マルチスレッド型ウルトラパイプラインプロセッサ・アーキテクチャの概要

3.1節を考慮したマルチスレッド型プロセッサ・アーキテクチャが、本論文で提案するマルチスレッド型ウルトラ[1]パイプライン・プロセッサ・アーキテクチャ（Multithreaded Ultrapipeline Processor architecture: MUP）である。MUPのハードウェア構成を図3.1に示す。

[1] ウルトラというのは、従来のパイプラインやスーパーパイプライン・プロセッサよりもパイプラインを細分化しているという意味で使用している

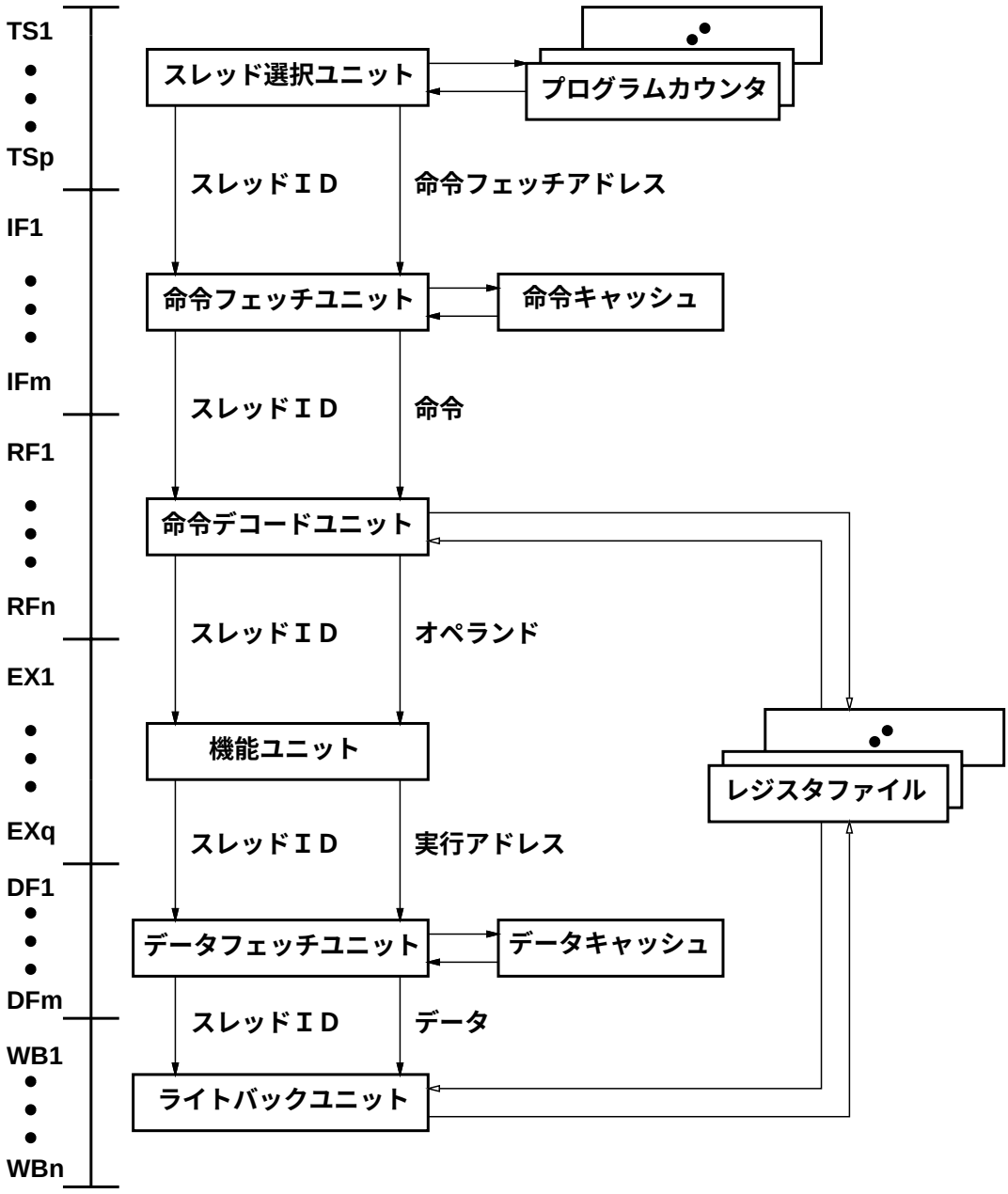

図 3.1: MUP のハードウェア構成

MUP は、プログラムカウンタ、各種状態レジスタ、およびレジスタファイルをスレッド毎に備え、命令およびデータフェッチユニット、命令デコードユニット、命令キャッシュ、データキャッシュ、ライトバックユニット、および機能ユニットを複数のスレッドによって共有する形態をとる。

3.2.1 MUP の特徴

MUP は、次に示す特徴によって、動作クロックの高速化による高いスループットを実現し高性能化を図る。

パイプライン・ステージ数と同数のレジスタセット[2]

MUP では、各パイプラインステージをすべて異なるスレッドから発行された命令で埋めることによって、1 サイクル 1 命令の命令実行速度およびパイプライン制御の簡単化を実現する。このことを実現するためには、プロセッサがパイプライン・ステージ数と同数のスレッドを同時に扱うことができなければならない。そこで、MUP ではパイプライン・ステージ数と同数のレジスタセットを用意している。さらに、用意しているレジスタ類は、1 つのレジスタファイルを使って複数のスレッドで共有する形ではなく、各スレッド毎に独立した形をとる。これによってパイプラインの構造ハザードを回避する。

スレッド ID 用パイプライン

MUP は、複数スレッドからの命令を 1 つのパイプラインを共有して実行する。このような形態で命令を実行する場合、どのスレッドから発行された命令なのかを識別する必要が出てくる（例えば、演算結果をレジスタファイルへ書き戻す場合やプログラム・カウンタの値を更新する場合には、どのスレッドに対応するレジスタファイルまたはプログラム・カウンタに書き込むかを識別しなければならない）。この “識別が必要” ということに対して、MUP では命令およびオペランドが流れるパイプラインの他に、スレッドを識別するための情報[3]が流れるパイプラインを持つことによって対処する。

[2] 1 つのスレッドを制御するのに必要な、プログラムカウンタ、各種状態レジスタ、レジスタファイルの組を表す表現

[3] これを “スレッド ID” と呼ぶ

ラウンドロビン選択方式による実行スレッドの切り替え

先にも述べたように、MUP では、データハザードや制御ハザードを回避しパイプライン制御を簡単化するなどの目的のために、パイプラインの各ステージをすべて異なるスレッドからの命令で埋める（同一スレッドからの命令発行を、先行命令が完了してから後続命令を発行する逐次実行の形態にする）。このためには、スレッド選択ユニットが、パイプラインの各ステージをすべて異なるスレッドからの命令で埋めることができるスレッド選択方式に従って、命令発行を行なうスレッドを毎サイクル切り替えなければならない。MUP は、このスレッド選択方式としてラウンドロビン選択方式を用いている。このラウンドロビン方式によるスレッド選択を、図 3.2に示す。

図 3.2では、プロセッサのパイプラインをスレッド選択ユニットを含めて N ステージで構成している。そして、N 段あるパイプライン・ステージをすべて異なるスレッドからの命令で埋めるために N のレジスタセットを持つ。

プロセッサが動作を始めると、1 クロックサイクル目には 0 番のレジスタセットに対応するスレッドから命令発行を行ない、2 クロックサイクル目には 1 番のレジスタセットに対応するスレッドから命令発行を行なう。そして、N クロックサイクル目に N-1 番のレジスタセットに対応するスレッドから命令を発行すると、N+1 クロックサイクル目には再び 0 番のレジスタセットに対応するスレッドから命令発行を行なう。

命令キャッシュとデータキャッシュの分離

後でも述べるが、MUP はスレッド毎に独立したキャッシュメモリを持つということをせず、すべてのスレッドで 1 つのキャッシュメモリを共有する形態をとる。このため、命令とデータを統合した 1 つのキャッシュメモリでは、あるスレッドの命令フェッチと他のスレッドのデータアクセスが衝突するという構造ハザードが発生する。この構造ハザードを回避するために、命令キャッシュとデータキャッシュを分離する [19]。

キャッシュメモリとレジスタファイルのパイプライン化

MUP では、動作クロックの高速化を目的として、パイプライン・ステージの細分化を行なう。このため、レジスタファイルやキャッシュメモリに対しても高いスループットが求められる。この要求を満たすために、レジスタファイルとキャッシュメモリのパイプライン化を行なっている。

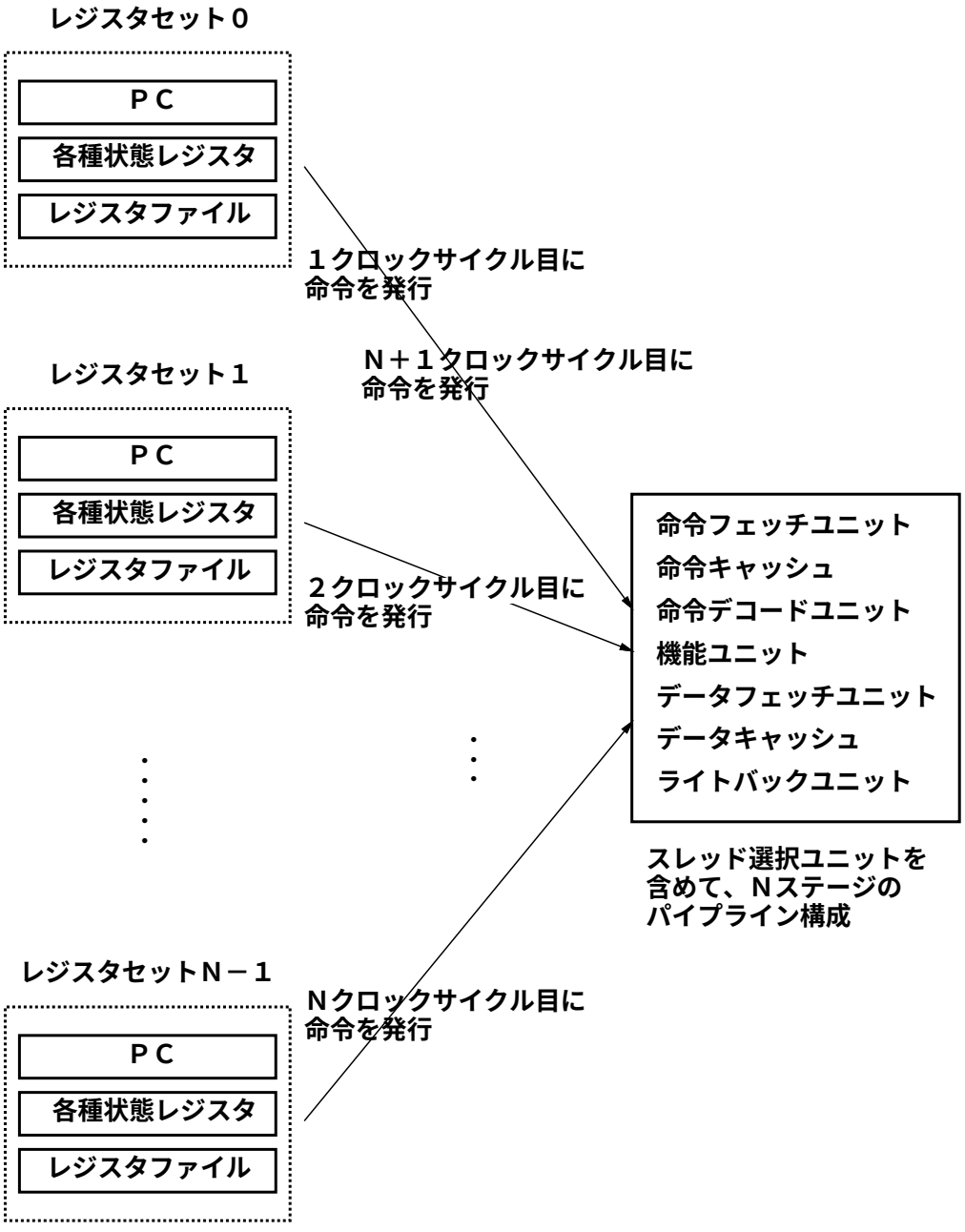

図 3.2: 実行スレッドの切り替え方式

レジスタファイルへのアクセスは、デコードフェーズとレジスタアクセスフェーズの2つのフェーズによって行なわれる。この2つのフェーズを分割することにより、レジスタファイルのパイプライン化を行なう。

同様にキャッシュメモリへのアクセスは、大まかには次の4つのフェーズによって行なわれる。

1. アドレスデコード
2. メモリセル・アレイの読み出し
3. 判定
4. データ排出

したがって、この4つのフェーズを分割することによって、図. 3.3に示す4ステージで構成するパイプライン化キャッシュを実現する。

複数スレッド間でのキャッシュメモリの共有

関数型プログラムの実行時には、同一関数の起動に対して時間的局所性が存在する可能性がある。そのため、スレッド毎に独立したキャッシュメモリを用意した場合では、あるスレッドの実行によって関数がキャッシュメモリに格納されていても、他のスレッドからその関数を参照することができない。つまり、キャッシュメモリのミス率が上がることになる。

また、キャッシュの容量性ミスの発生について、スレッド毎にキャッシュメモリを用意する場合と大容量のキャッシュメモリを複数のスレッドで共有する場合とを比較すると、共有する場合の方がミス率は下がると考えられる [1]。

以上により、MUP ではキャッシュメモリのミス率を下げるために、複数のスレッドで1つのキャッシュメモリを共有する形態をとる。

大容量キャッシュの搭載

MUP では、実行スレッドをラウンドロビン選択方式によって選択するため、キャッシュにアクセスするスレッドが毎サイクル切り替わる。そのためメモリアクセスの局所性が無くなる可能性がある。そのため MUP では、扱うスレッド数に応じた大容量のキャッシュメモリを持つことによって、キャッシュメモリのミス率の上昇を抑える。

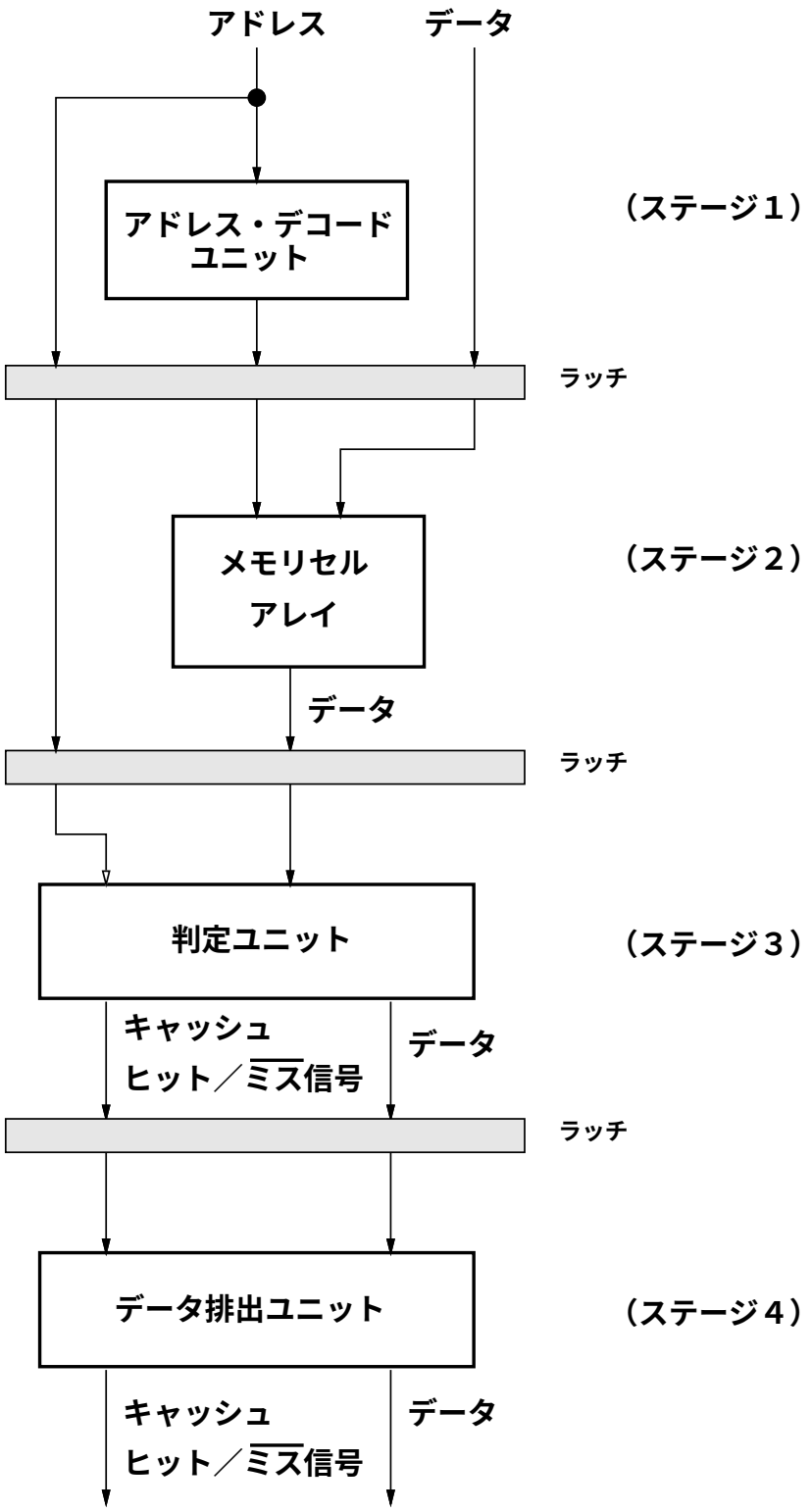

図 3.3: 4 ステージで構成するパイプライン化キャッシュメモリ

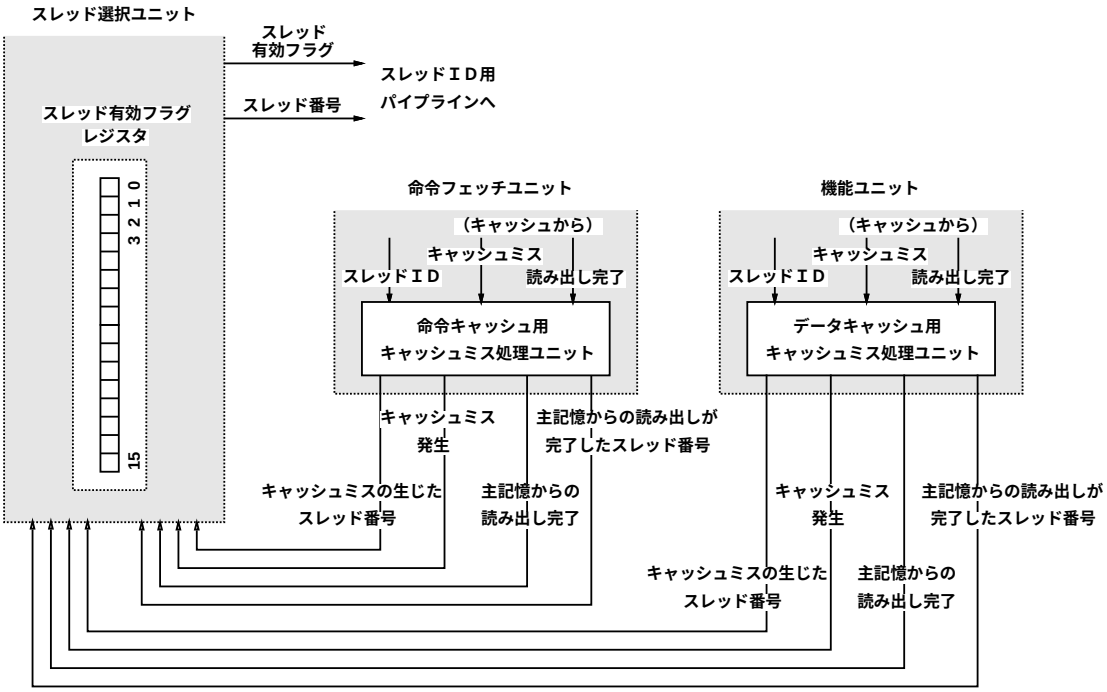

図 3.4: MUP のキャッシュミス取り扱い機構

キャッシュミスの取り扱い

MUP では、高いスループットを保つために、あるスレッドの命令がキャッシュミスを起こした場合、キャッシュミスを起こしたスレッドの実行を停止（キャッシュ更新待ち状態になる）する。これによって、パイプラインをストールさせることなく他のスレッドから発行された命令の実行を続ける。また、キャッシュミスが発生してキャッシュ更新待ち状態になったスレッドを、ソフトウェアまたはハードウェアによって他の実行可能なスレッドと入れ換えることは行なわない。これは、ソフトウェアでスレッドの入れ換えを行なう場合には、ソフトウェアによるコンテキストスイッチ時間が、キャッシュミスによって生じる主記憶からの読み出しに必要な時間に対して長いからである。ハードウェアによってコンテキストスイッチを行なう場合には、ハードウェア自身が複雑になる。

MUP のキャッシュミス取り扱いに関して説明する。MUP は、あるスレッドの命令を実行中にキャッシュミスが生じた場合でも、パイプラインをストールさせ主記憶からの読み出しを待つことを行なわず、キャッシュミスを起こしていない他のスレッドの命令の実行を続けることによって高いスループットを保つ。この動作を行なうための機構を図 3.4

（スレッド数が 16 個の場合）に示す。この機構は、スレッド有効フラグ・レジスタと 2 つのキャッシュミス処理ユニット（命令キャッシュ用、データキャッシュ用）から構成される。

各スレッドは、“valid（キャッシュミスが生じていないため命令実行可能）” と “invalid（キャッシュミスによる主記憶からの読み出し待ちのため命令実行不可能）” の 2 つの状態に分類される。スレッド有効フラグ・レジスタの各ビットが、各スレッドの状態に対応し、すべてのスレッドの現在の状態を保持している。キャッシュミス処理ユニットは、スレッド番号を格納するキューを持つ。キャッシュミスが発生すると、ミスが生じたスレッドのスレッド番号がこのキューに格納され、主記憶からの読み出しが完了すると、このキューからスレッド番号が取り出される。またキャッシュミス処理ユニットは、スレッド有効フラグ・レジスタに対して、キャッシュミス情報（“キャッシュミス発生” と “キャッシュミスが発生したスレッド番号” の対から構成される）、または読み出し完了情報（“主記憶から読み出し完了” と “読み出しが完了したスレッド番号” の対から構成される）を伝える動作を行なう。

命令またはデータキャッシュへのアクセスでキャッシュミスが生じると、キャッシュミス処理ユニットは、スレッド有効フラグ・レジスタに対して、キャッシュミス情報を伝える（これによって、キャッシュミスを引き起こした命令を発行したスレッドを invalid 状態とする）。同時に、パイプライン中に存在するキャッシュミスが生じたスレッドのスレッド有効フラグを invalid に変更する。

invalid 状態のスレッドがスレッド選択ユニットによって選択され、パイプラインに投入されると（同時に、スレッドに対応するスレッド有効フラグもスレッド ID 用パイプラインに投入される）、各パイプライン・ステージは、invalid 状態を検知し、いかなる動作も行なわない。すなわち、パイプライン中には、invalid 状態のスレッドが発行した命令がバブルとなって存在する形になる。

キャッシュミスによって生じた主記憶からの読み出しが完了すると、キャッシュミス処理ユニットは、スレッド有効フラグ・レジスタに対して、読み出し完了情報を伝える。これによって、invalid 状態のスレッドが valid 状態に変更され、命令の実行を停止させられていたスレッドが、再び命令を実行することができる。

この機構によって、パイプラインをストールすることなく、命令を実行することが可能となる。

3.2.2 パイプラインの各フェーズの役割

命令パイプラインは、図3.1の左側に示すように、おおまかに6つのフェーズから構成されている。6つの各フェーズは、それぞれ数段に細分化されたパイプライン構造になっている。以下に各パイプライン・フェーズの役割の詳細を述べる。

TS1〜p (Thread Select)

スレッド選択ユニットが、すべてのスレッドの中からラウンドロビン選択方式に従って、次に命令発行を行なうスレッドを選択する。さらに、この選択されたスレッドに対応するスレッドIDとプログラムカウンタを命令フェッチユニットに渡す。

IF1〜m (Instruction Fetch)

命令キャッシュへのアクセスを要求するスレッドが“valid状態”ならば、命令キャッシュを通じて、プログラムカウンタの指すアドレスから命令を読み出す。

命令キャッシュへのアクセスを要求するスレッドが“invalid状態”ならば、命令キャッシュへのアクセスは行なわない。

また、命令キャッシュからのキャッシュヒット
ミス信号および読み込み完了信号の値に応じた処理が先に述べたキャッシュミス処理ユニットによって行なわれる。

RF1〜n (Register Fetch)

命令をデコードし、EXフェーズで使用するソースオペランドを決定する。レジスタファイルからオペランドを読み出す場合には、スレッドIDに対応するレジスタファイルから読み出す。

EX1〜q (Execution)

与えられたソースオペランドを使用して指定された演算を行なう。ロードまたはストア命令の場合は実行アドレスを計算する。分岐命令の場合は分岐条件が真か偽かの計算を行なう。

DF1〜m (Data Fetch)

データキャッシュへのアクセスを要求するスレッドが “valid 状態” ならば、データキャッシュを通じて、EX ステージで得られた実行アドレスに対してロードまたはストアを行なう。

データキャッシュへのアクセスを要求するスレッドが “invalid 状態” ならば、データキャッシュへのアクセスは行なわない。

また、IF フェーズと同様に、データキャッシュからのキャッシュヒットミス信号および読み込み完了信号の値に応じた処理がキャッシュミス処理ユニットによって行なわれる。

WB1〜n (Write Back)

スレッドが valid 状態ならば、EX フェーズで得られた演算結果または DF フェーズでメモリからロードされたデータを、スレッド ID に対応するレジスタファイルへ書き戻す。さらに、対応するプログラムカウンタの値を更新する。

スレッドが invalid 状態ならば、このフェーズでは何も行なわれない。

3.2.3 例外処理機能

例外の発生源には、プロセッサ内部（未定義命令実行、算術オーバーフローなど）とプロセッサ外部（割り込み）の 2 種類がある。MUP では例外を発生源別に次のように処理する。

プロセッサ内部で発生した例外の処理

例外の発生したスレッドの実行だけを中断し例外処理を行なう。

プロセッサ外部で発生した例外の処理

実行中のスレッドの内、どれか 1 つのスレッドが命令の実行を中断し例外処理を行なう。どのスレッドが命令の実行を中断して例外処理を行なうかというのは割り込みの発生するタイミングに依存する。また、割り込みに関する制御（割り込みマスク）はすべてのスレッド間で共有する。したがって、あるスレッドが 1 つの割り込み原因からの割り込み

の受け付けをマスクすると、他のすべてのスレッドもその割り込み原因からの割り込みを受け付けない。

第 4 章

設計と評価

4.1 評価方法

3章で提案した MUP の基本的な性能を評価するために、“動作クロック” および “MUP を構成するハードウェア量” の見積りを行なう。

この見積りのために、MUP を具体的に設計し、さらに、この設計した MUP をハードウェア記述言語 SFL[17] を用いて記述し、動作記述論理設計支援ツール PARTHENON[17] 上でゲートレベルの論理合成を行なう。この合成結果から “クリティカルパスの評価による動作クロック”、“MUP を構成するハードウェア量” を評価する。

4.2 設計範囲

MUP の “動作クロック” および “MUP を構成するハードウェア量” を評価するためには、MUP の設計範囲を決める必要がある。

“動作クロック” および “MUP を構成するハードウェア量” を評価するために必要な設計範囲を以下に示す。

動作クロックを評価するために必要な設計範囲 データパス部、および制御部

ゲート量を評価するために必要な設計範囲 データパス部、制御部、および例外処理部

このことから、“動作クロック” および “MUP を構成するゲート量” の評価を行なうために、本章で設計する MUP はデータパス部、制御部、および例外処理部を含めて設計を

行なう。

4.3 具体的設計

本節では、MUP の動作クロックおよびゲート量を見積もるための具体的な設計（命令セット、ハードウェア構成、例外処理構成、およびパイプライン構成）について述べる。

4.3.1 命令セット

表 4.1: MUP の持つ命令セット

ロード	LB, LBU, LH, LHU, LW
ストア	SB, SH, SW
論理演算	AND, ANDI, OR, ORI, XOR, XORI
算術演算	ADD, ADDI, ADDIU, ADDU, SUB, SUBU
分岐	J, JAL, BEQ, BGEZ, BGTZ, BLEZ, BLTZ, BNE
その他	SYSCALL, RFE, LDSTW, MFPC, MFSR MFCR, MFEPC, MTPC, MTSR, MTEPC

本論文において設計する MUP は、命令セットとして表 4.1に示す命令を持つ。

この表 4.1に示す通り、性能見積りのために設計した MUP は、一般的な命令はほとんど含んでいる。しかし、整数除算・乗算命令、浮動小数点に関する命令は持っていない。また、すべての命令は 32 ビット固定長で、一部の命令を除いて命令フォーマットは MIPS 社の R2000[9] と同じである。

4.3.2 データパス構成

本論文で設計した MUP のハードウェア構成とデータパスを図 4.1に示す。実線が “命令およびオペランドが流れるパイプライン”、破線が “スレッド ID が流れるパイプライン” である。

MUP を構成するユニットを以下に示す。

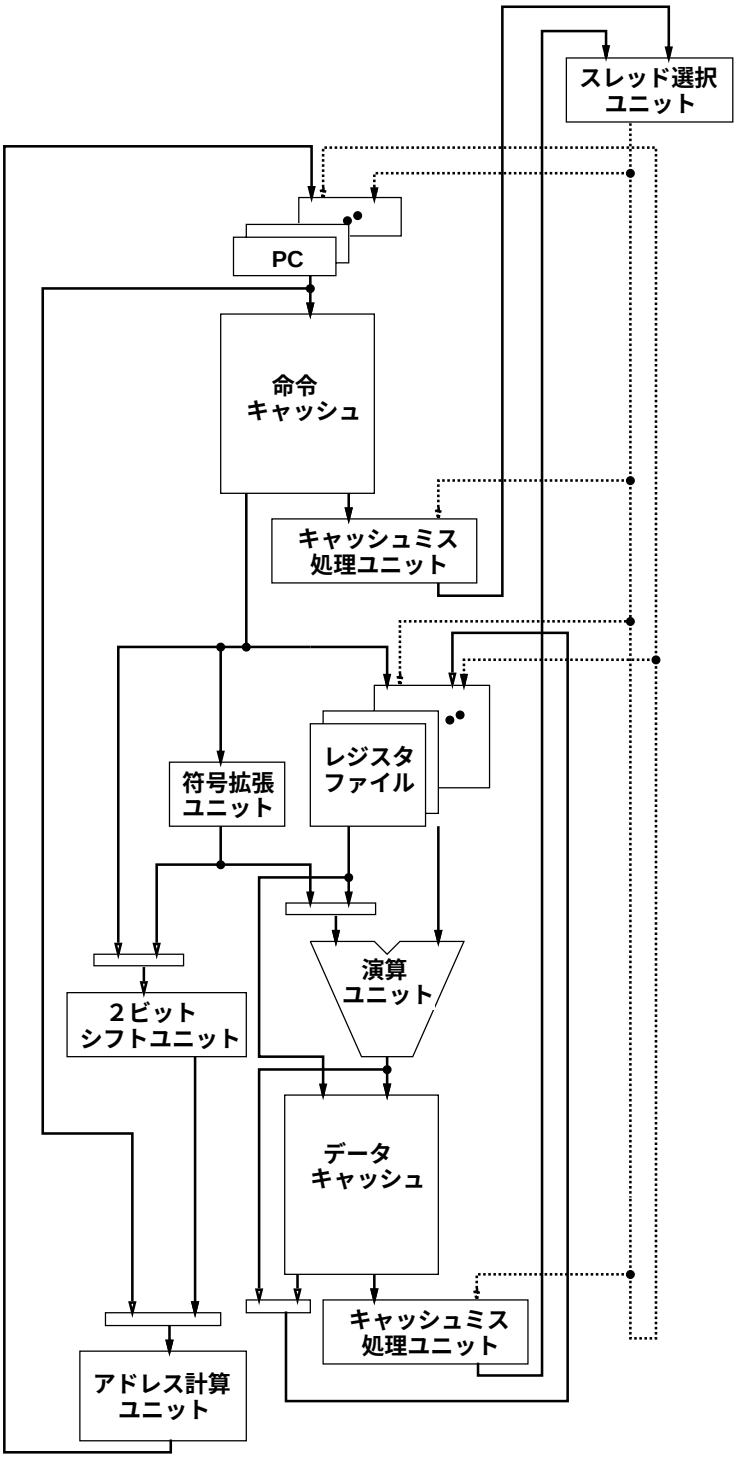

図 4.1: MUP のデータパス構成

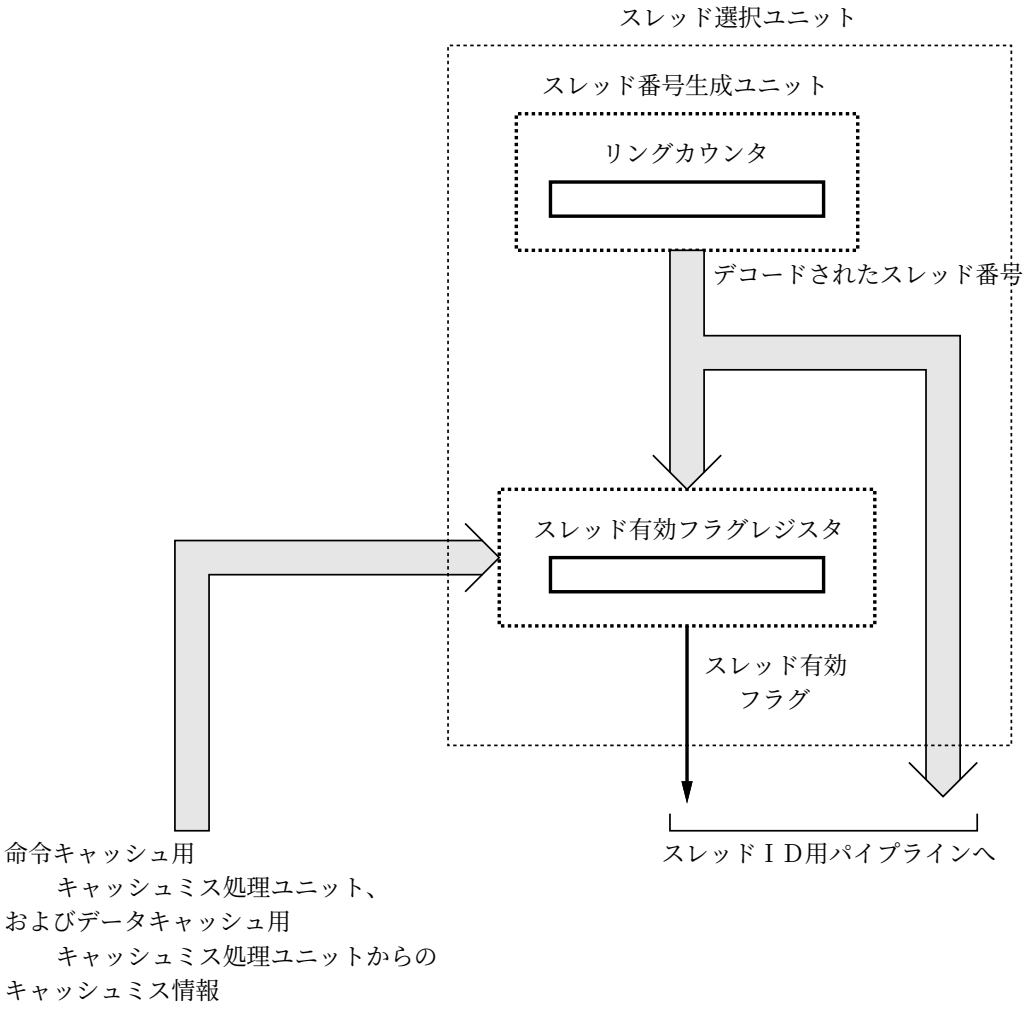

図 4.2: スレッド選択ユニットのハードウェア構成

スレッド選択ユニット

ラウンドロビン方式に従って命令発行を行なうスレッドを選択し、選択されたスレッドのスレッド番号およびスレッド有効フラグをスレッド ID 用パイプラインに投入するユニットである。

スレッド選択ユニットの詳細を図 4.2に示す。スレッド選択ユニットは、“スレッド番号生成ユニット” と “スレッド有効レジスタ” で構成する。

スレッド番号生成ユニットは、扱うスレッド数と同じ長さのリングカウンタ（サーキュラシフトレジスタ）を持つ（扱うスレッド数が 8 ならば、8 進リングカウンタを持つ)。図

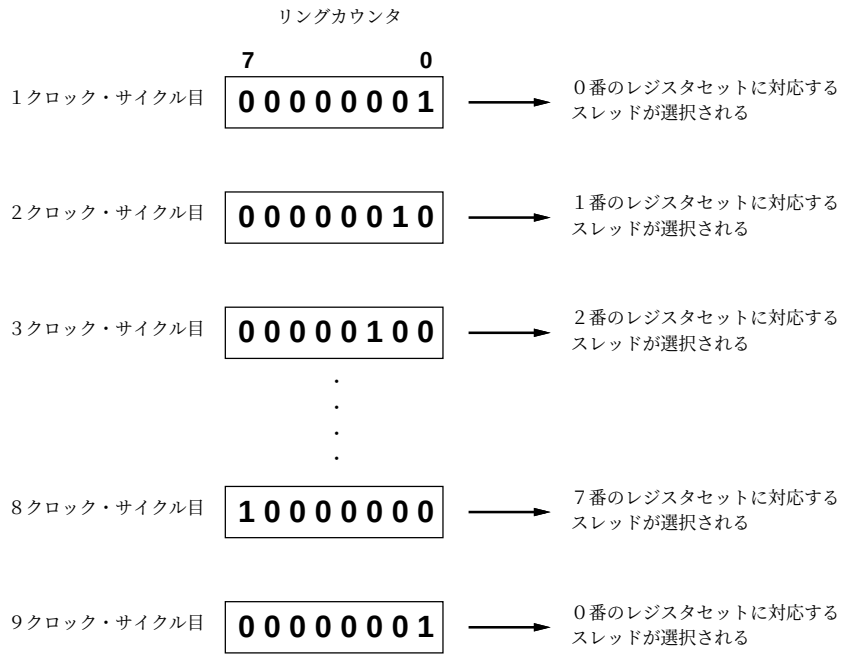

図 4.3: リングカウンタのカウント・アップ

4.3に示すように、毎クロック、リングカウンタのカウンタ値をアップさせることにより、ラウンドロビン方式によるスレッド番号発行を行ない、発行したスレッド番号をスレッド ID 用パイプラインとスレッド有効フラグレジスタに伝える。

また、スレッド有効フラグレジスタは、3.2節で述べたように、各スレッドの現在の状態を保持し、スレッド番号生成ユニットから伝えられたスレッド番号に対応するスレッドのスレッド有効フラグをスレッド ID 用パイプラインに投入する。

PC 群（プログラムカウンタ群）

32 ビット幅のプログラムカウンタを、扱うスレッド数分用意したプログラムカウンタの集合である。

プログラムカウンタ群では、指定されたスレッド番号に対応するプログラムカウンタの読み出し、および書き込みを行なう。

また、プログラムカウンタ群へのアクセスは、TS（スレッド選択）フェーズ、RF（レジスタフェッチ）フェーズ、および WB（ライトバック）フェーズから同時に発生する。そのため、図 4.4に示すように TS フェーズ、RF フェーズ、および WB フェーズからのアク

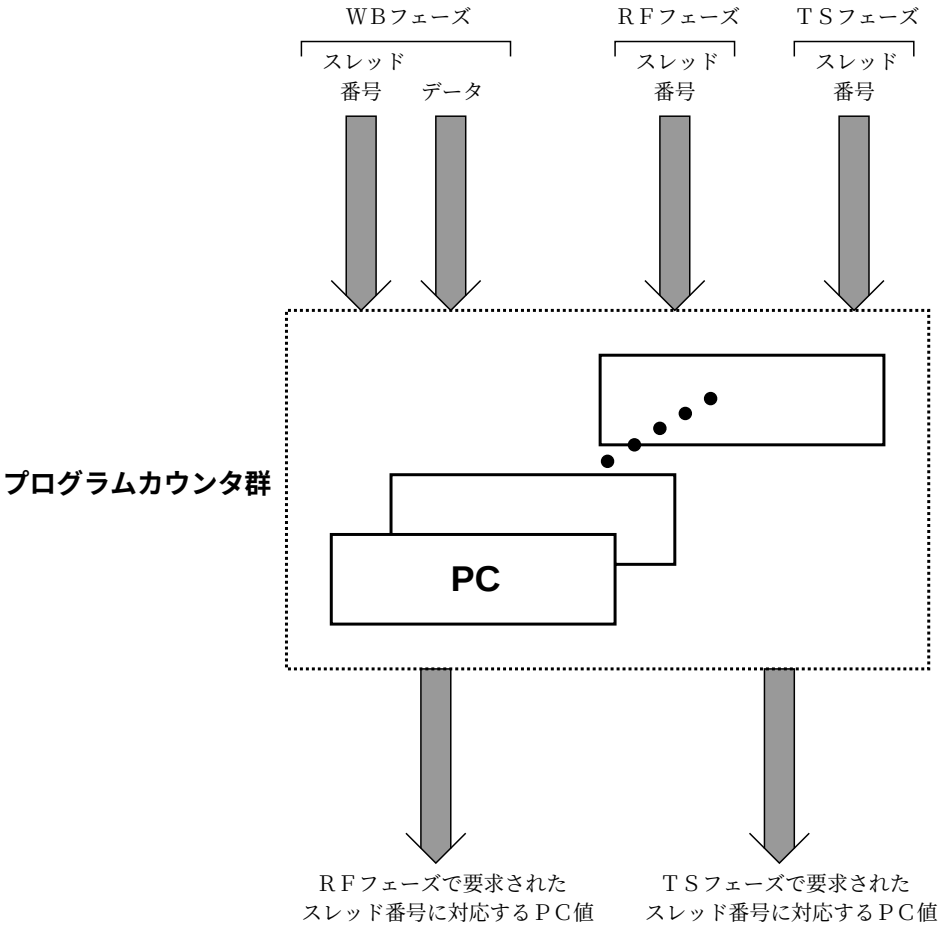

図 4.4: PC 群へのアクセス

セスを同時に受け付けることが可能にしてある。すなわち、TS、RF、および WB フェーズにはすべて異なるスレッドの命令が入っているので、同時に発生するアクセス要求はすべて異なるスレッド用のプログラムカウンタに対して起こり、プログラムカウンタ群中の各スレッド用の同一プログラムカウンタに対して同時にアクセスが発生することはない。以上によって、プログラムカウンタ群での構造ハザードを回避する。

命令キャッシュとデータキャッシュ

3章で述べた通り、命令キャッシュとデータキャッシュをそれぞれ別々に用意する。

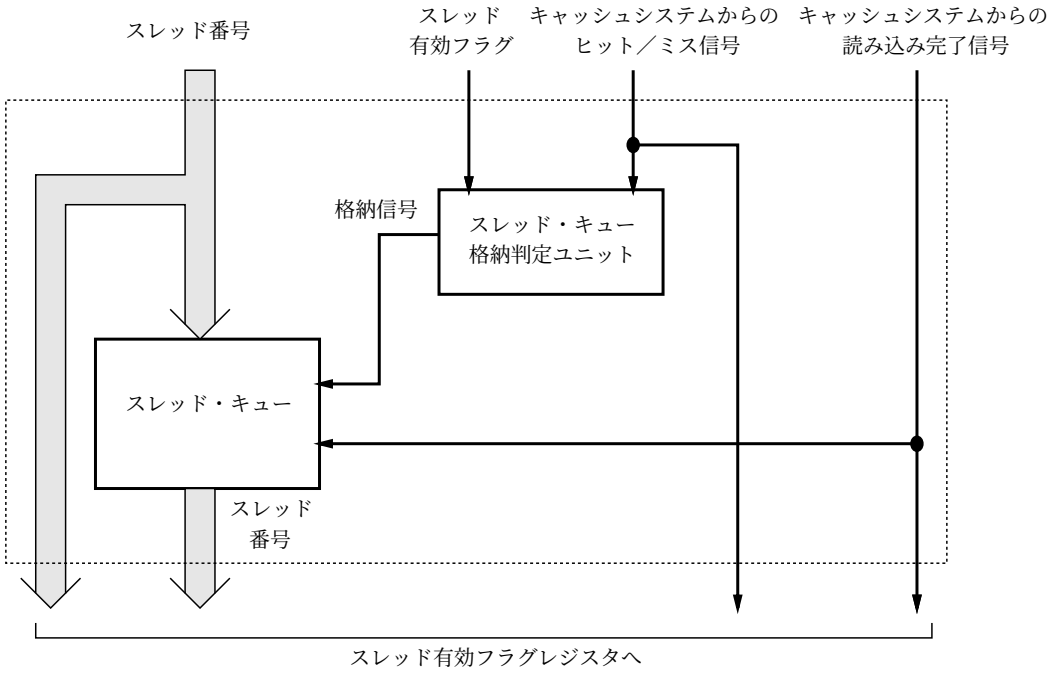

図 4.5: キャッシュミス処理ユニットの詳細

キャッシュミス処理ユニット

キャッシュミス処理ユニットの詳細を図 4.5 に示す。

キャッシュミス処理ユニットは、“スレッド・キュー格納判定ユニット” と “スレッド・キュー” で構成される。

スレッド・キュー格納判定ユニットは、“スレッド有効フラグ” と “キャッシュシステムからのヒット / ミス信号” を受け取る。スレッド有効フラグが valid 状態かつキャッシュシステムからのヒット / ミス信号がミス状態のときのみ、スレッド・キューに対して、スレッド番号を格納を要求するための “ 格納信号” を伝える。

スレッド・キューは、格納信号が伝えられると、スレッド番号をキューに格納する。また、“キャッシュシステムからの読み込み完了信号” が伝えられると、スレッド番号をキューから取り出し、スレッド有効フラグレジスタへ送る。

レジスタファイル群

1 つのスレッドに対応するレジスタファイルは、32 ビット幅レジスタを 32 個用意している。レジスタファイル群は、このレジスタファイルをパイプライン・ステージ数分用意したレジスタファイルの集合である。

レジスタファイル群では、指定されたスレッド番号に対応するレジスタファイルの指定されたレジスタ番号の値の読み出しおよび書き込みが行なわれる。

また、レジスタファイル群へのアクセスは、RF（レジスタフェッチ）フェーズからの読み出しと、WB（ライトバック）フェーズからの書き込みが同時に発生する。そのため、図 4.6に示すように、RF フェーズ、および WB フェーズからのアクセスを同時に受け付けることが可能にしてある。すなわち、RF および WB フェーズにはそれぞれ異なるスレッドの命令が入っているので、同時に発生するアクセスはすべて異なるスレッド用レジスタファイルに対して起こり、レジスタファイル群中の各スレッド用の同一のプログラムカウンタに対して同時にアクセスが発生することはない。
以上により、レジスタファイル群での構造ハザードを回避する。

符号拡張ユニットと 2 ビットシフトユニット

本 MUP の命令フォーマットは、MIPS 社の R2000 の命令と同じフォーマットである。そのため、ジャンプ命令と分岐命令中に含まれるターゲットアドレス部分に対して、符号拡張または 2 ビットシフトを行なう必要がある。この操作を行なうために、本 MUP では符号拡張ユニットと 2 ビットシフトユニットを用意している。

演算ユニット

演算ユニットの詳細を図 4.7に示す。演算ユニットは、“桁上げ生成・伝搬ユニット”、“桁上げ先見ユニット”、および “32 ビット ALU” で構成する。

加算演算を行なう場合、32 ビット ALU は、桁上げ生成・伝搬ユニットで作られる “部分和（図 4.7中の P）” と桁上げ先見ユニットで作られる “全桁の桁上げ（図 4.7中の C）” を用いて加算演算を行なう。

and、or、および exor 演算を行なう場合、32 ビット ALU は、ALU 入力 “1” と “2” を使用して、指定された演算を行なう。

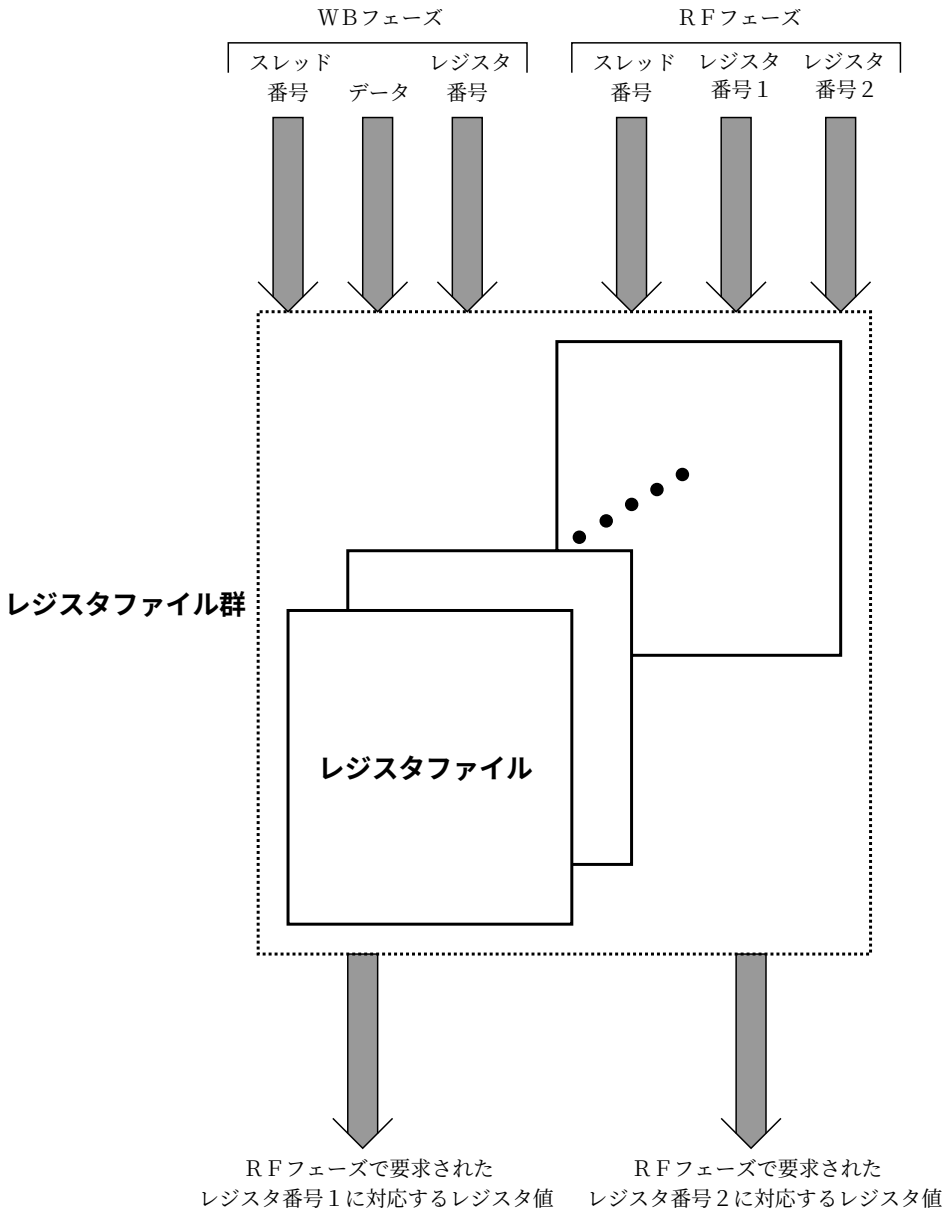

図 4.6: レジスタファイル群へのアクセス

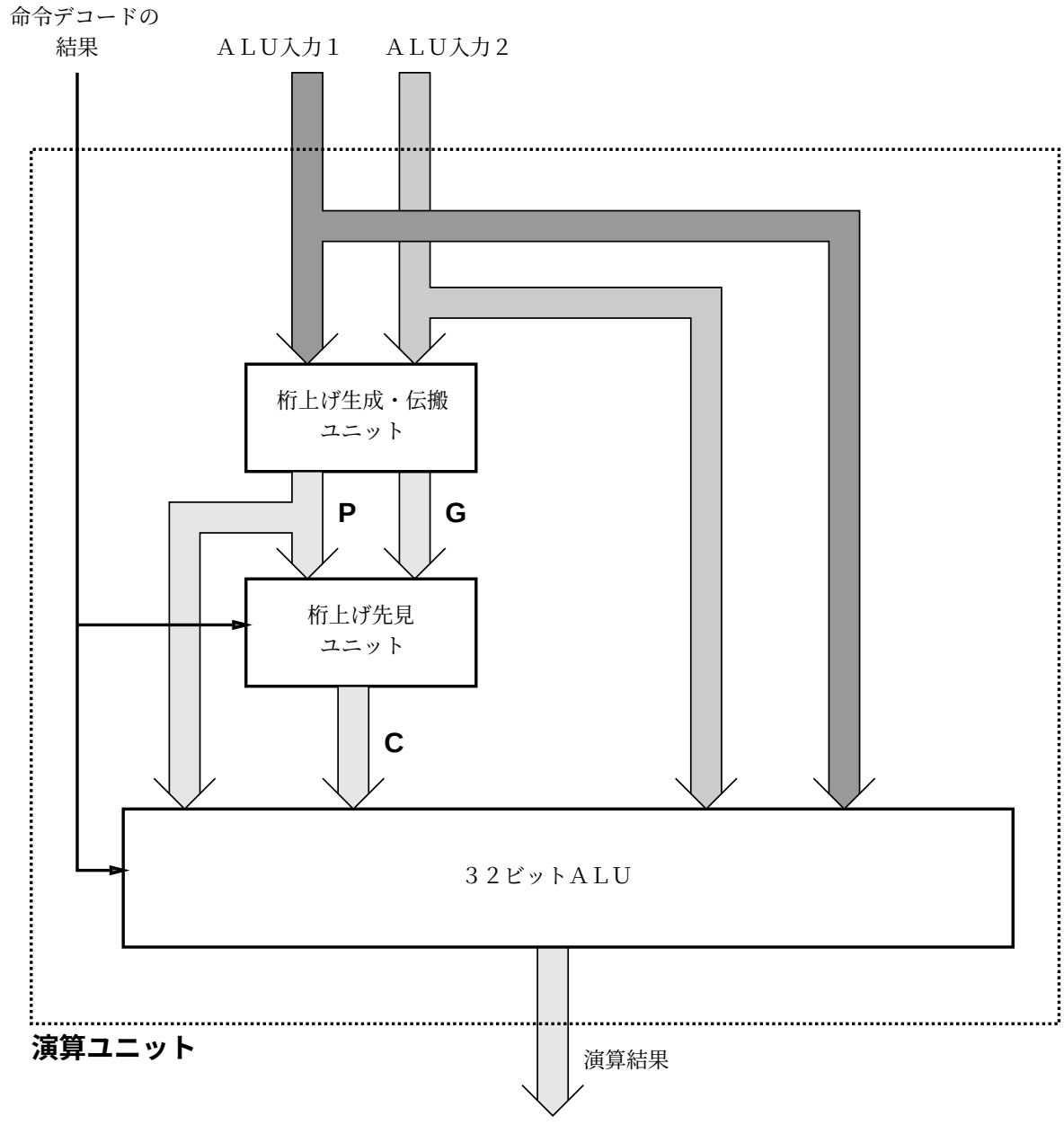

図 4.7: 演算ユニットの詳細

アドレス計算ユニット

分岐命令の条件の真偽と例外処理条件の成立から次のアドレスを計算するユニットである。

4.3.3 例外処理機能

例外処理

MUP で発生する例外要因とその例外のパイプライン上での発生場所を以下に示す。

- 外部割り込み（DF2 ステージ）
- 未定義命令例外（RF1 ステージ）
- 特権命令例外（RF1 ステージ）
- アドレスエラー（IF1 ステージ、DF1 ステージ）
- 算術オーバーフロー（EX3 ステージ）
- システムコール（RF1 ステージ）

外部割り込みは複数あり、後で述べる外部割り込みレジスタのマスク・フィールドを “0” にすることにより、各々を個別にマスクすることが可能である。

3.2.3節で述べた通り、プロセッサ内部で生じる例外（未定義命令例外、特権命令例外、アドレスエラー、算術オーバーフロー、システムコール）が発生した場合には、例外の発生したストリームの実行だけを中断し、このストリームに対応する PC 等を書き換えて例外処理を行なう。外部割り込み処理は、実行中のスレッドが DF2 ステージまで例外を発生せず、かつ割り込みマスクがクリアされていれば、そのスレッドの実行だけを中断して外部割り込みを受け付ける。

例外処理をサポートするレジスタ

MUP では、例外処理を行なう際にこの処理をサポートするレジスタを用意している。用意しているレジスタを以下に示す。

- 外部割り込みレジスタ

  外部割り込みレジスタの役割は次の通りである。

  複数の外部割り込みを個別にマスクする

  複数の外部割り込みの内、どの外部割り込みが生じたかを示す

また、本レジスタは 1 つだけ用意され、すべてのスレッドで共有する。

- ステータスレジスタ

  ステータスレジスタはプロセッサの状態保存を行なう。本レジスタは、各スレッド毎に提供される。

- 例外原因レジスタ

  例外原因レジスタの役割は、最後に発生した例外を示すことである。本レジスタは各スレッド毎に提供される。

- 例外 PC(Exception PC)

  例外 PC には、例外が処理された後に処理が再開されるアドレスが入る。ハードウェアは、このレジスタに例外を引き起こした命令のアドレスを入れる。また、本レジスタは各スレッド毎に提供される。

例外および割り込みが発生した場合の例外処理手順

例外処理に関するデータパスを図 4.8に示す。例外が発生した場合、“例外検出ユニット”、および “例外原因レジスタ・ステータスレジスタ書き込みユニット” により例外処理を行なう。また外部割り込みが生じた場合は、“例外検出ユニット”、“外部割り込み検出ユニット” および “例外原因レジスタ・ステータスレジスタ書き込みユニット” によって例外処理を行なう。

“未定義命令例外”、“特権命令例外”、“アドレスエラー”、“算術オーバーフロー”、および “システムコール” が発生すると、例外検出ユニットは、各ユニットからの入力から例外を検出し、例外原因レジスタ・ステータスレジスタ書き込みユニット、およびプログラムカウンタ計算ユニットに対して、“例外発生” を伝える。

外部割り込み検出ユニットは、例外検出ユニットから例外発生が伝えられた場合、外部割り込み検出を一切行なわない。例外検出ユニットから例外発生が伝えられなかった場合は、外部割り込みの検出を行ない、割り込みが発生していれば、例外原因レジスタ・ステータスレジスタ書き込みユニットおよびプログラムカウンタ計算ユニットに対して、“外部割り込み発生” を伝える。

例外原因レジスタ・ステータスレジスタ書き込みユニットには、“例外発生” と “外部割り込み発生” のいずれかの信号が伝えられる。例外原因レジスタ・ステータスレジスタ

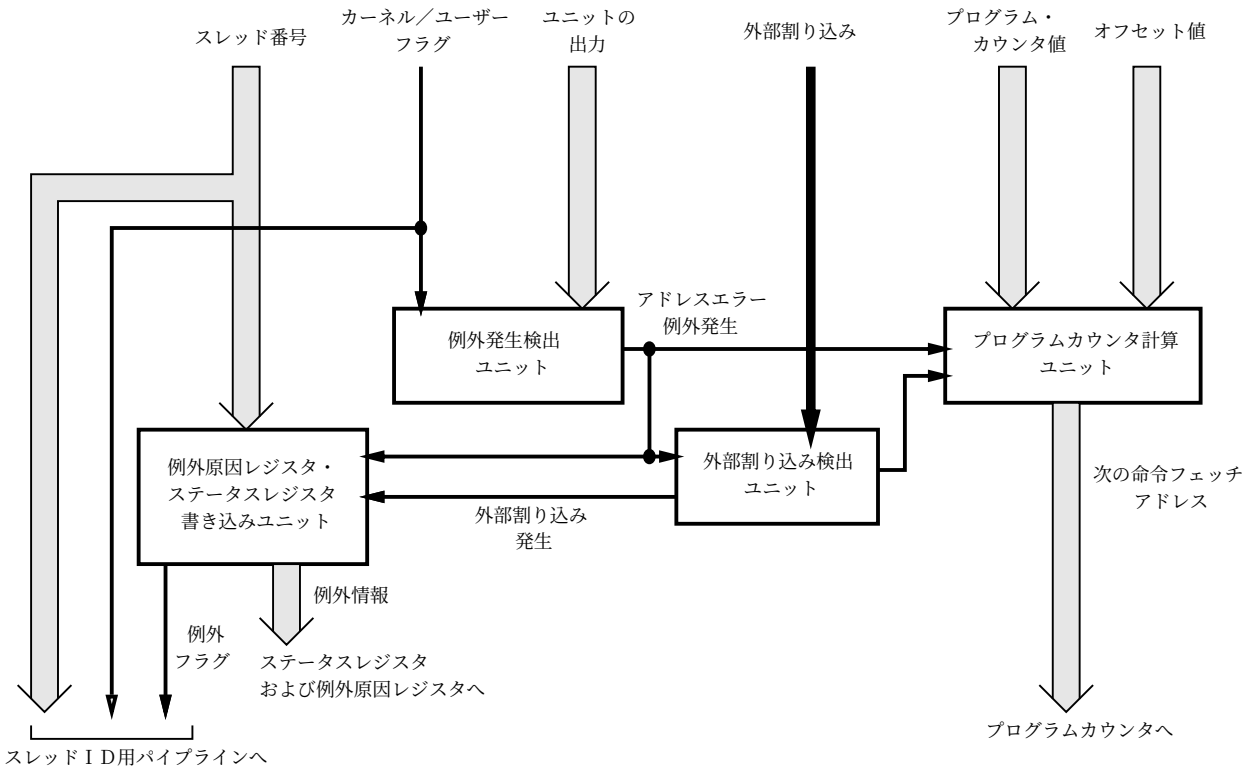

図 4.8: 例外処理のデータパス

書き込みユニットは、伝えられた例外発生に応じて、例外原因レジスタとステータスレジスタの値を変更する。さらに例外原因レジスタ・ステータスレジスタ書き込みユニットは、スレッド ID 用パイプラインに例外が発生したか否かを示す “例外発生フラグ” を投入する。

プログラムカウンタ計算ユニットには、“例外発生”、および “外部割り込み発生” が伝えられる。プログラムカウンタ計算ユニットは、この 2 つの信号のどれも伝えられなかった場合には、通常通り次のプログラムカウンタ値の計算を行なう。2 つの信号のいずれかが伝えられた場合、プログラムカウンタ計算ユニットは、プログラムカウンタに格納される値を、例外処理開始アドレス値[1]に変更し、同時に EPC レジスタに変更前のプログラムカウンタ値を格納する。

ここで説明した例外処理機構の詳細については、本論文巻末の付録に示してある。

4.3.4 パイプライン構成

論理回路 1 段当たりの遅延時間とメモリセル・アレイの読み出し時間を比較すると、より多くの時間を必要とするのはメモリセル・アレイの読み出し時間である。従って、パイプラインの各ステージの大きさは、このメモリセル・アレイの読み出し時間を考慮に入れて決める必要がある。

本節では、“論理回路 1 段当たりの遅延時間見積り” と “メモリセル・アレイの読み出し時間と論理回路 1 段当たりの遅延時間を考慮し分割した MUP のパイプライン構成” について述べる。

論理回路 1 段当たりの遅延時間見積り

プロセッサの動作クロックを見積もるためには、パイプライン 1 ステージ当たりの実行時間を見積もる必要がある。この実行時間の見積りためには、論理回路 1 段当たりの遅延時間を見積もる必要がある。

本節では論理回路 1 段当たりの遅延時間の見積りについて述べる。

本章で設計する MUP をウェーハ上に実現する技術として $0.1\mu m$ のプロセスルール、アルミ配線で設計すると仮定する。$0.1\mu m$ テクノロジの回路パラメータを表 4.2に示す [11][27][24]。

[1]固定アドレス値である

表 4.2: $0.1\mu m$ テクノロジの回路パラメータ

ゲート長	$0.1\mu m$
素子の遅延時間	$12psec$
配線幅	$0.15\mu m$
アルミ配線の膜厚	$0.5\mu m$
アルミ配線のシート抵抗	$0.6\Omega/sq$
SiO_2の膜厚	$0.5\mu m$
アルミ配線の単位面積当たりの容量	$88.8fF/\mu m^2$

論理回路 1 段当たりの遅延時間は次式で表すことができる。

$$\text{1 段当たりの遅延時間} = \text{素子の遅延時間} + \text{素子の負荷容量に比例する遅延時間} \quad (4.1)$$

式 4.1中の “素子の負荷容量に比例する遅延時間” は、

$$\text{素子の負荷容量に比例する遅延時間} = \text{配線抵抗} \times \text{配線容量} \quad (4.2)$$

で表すことができる。

式 4.2中の “配線抵抗” は次式で表すことができるので、

$$\text{配線抵抗} = \text{シート抵抗} \times \text{配線長/配線幅} \quad (4.3)$$

式 4.3の配線長は、各ユニット内部の素子間の配線長とユニット間の配線長とを算術平均することにより求めた値である。

$$\begin{aligned} \text{配線抵抗} &= 0.6\Omega/sq \times 43.8\mu m/0.15\mu m \\ &= 175\Omega \end{aligned}$$

また、式 4.2中の “配線容量” は次のように表すことができるので、

$$\text{配線容量} = \text{配線の単位面積当たりの容量} \times \text{配線長} \times \text{配線幅} \quad (4.4)$$

表 4.2で示したパラメータをそれぞれ代入することにより次に示す値になる。

$$\begin{aligned} \text{配線容量} &= 88.8fF/\mu m^2 \times 43.8\mu m \times 0.15\mu m \\ &= 583fF \end{aligned}$$

上で求めた配線抵抗、配線容量、および表 4.2の値を式??、4.2に代入することにより、論理回路 1 段当たりの遅延時間を $114psec$ とする。

パイプライン分割

キャッシュメモリのメモリセル・アレイの読み出し時間が、論理回路 1 段当たりの遅延時間よりも遅いので、パイプラインの分割はメモリセル・アレイの読み出し時間を考慮して行なわなければならない。

そのためにメモリセル・アレイの読み出し時間を仮定する。
現在、メモリセル･アレイの読み出し時間として、$0.5\mu m$ のプロセスルールの場合、$1nsec$[16] というのが報告されている。このため、$0.1\mu m$ のプロセスルールで実現した場合、読み出し時間はさらに高速になる。また、1 つのメモリセル・アレイを複数のサブアレイに分割することによって、読み出し時間を高速にする方法もある [1]。
これらを考慮して、ここではメモリセル・アレイの読み出し時間をパイプライン・ラッチ分を含めて $1nsec$ と仮定する。

上記のようにメモリセル・アレイの読み出し時間をパイプライン・ラッチ分を含めて $1nsec$ と仮定することにより、パイプラインを、1 ステージ当たりの論理回路段数がパイプライン・ラッチ分を除いて 6 段（遅延時間が $684psec$ になる）以下となるように分割した。

パイプライン 1 ステージ当たりの論理段数を 6 段以下とした 17 ステージから構成する MUP を図 4.9に示す。また、17 ステージに分割したパイプラインの各ステージの動作を以下に示す。

1.TS1

命令発行を行なうスレッドを、スレッド選択ユニットがラウンドロビン方式によって選択し、スレッド番号をスレッド ID 用パイプラインに投入する。

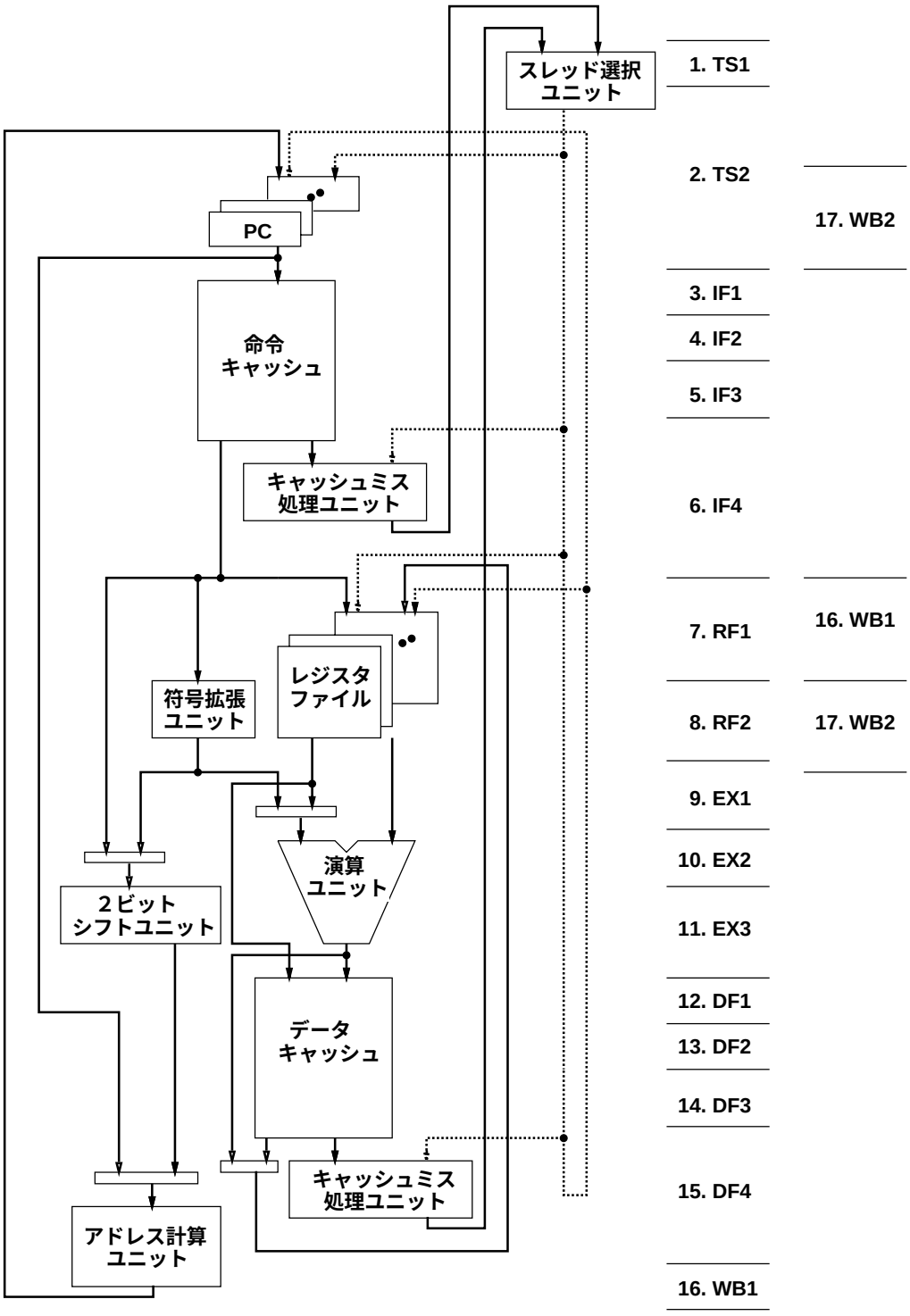

図 4.9: 17 ステージで構成する MUP のパイプライン構成

2.TS2

TS1 ステージで選択されたスレッド番号に対応するプログラムカウンタの値とスレッド有効フラグを、それぞれプログラムカウンタ群とスレッド有効フラグ・レジスタから読み出す。プログラムカウンタの値は、命令およびオペランドが流れるパイプラインへ投入し、スレッド有効フラグはスレッド番号と共にスレッド ID 用パイプラインに投入する。

3.IF1

命令キャッシュへのアクセスを開始する（プログラムカウンタの第 12〜第 2 ビットまでをデコードを行なう）。

4.IF2

命令キャッシュへのアクセス中（IF1 ステージでデコードした結果を使用して、キャッシュメモリのタグ部分とデータ部分を読み出す）。

5.IF3

命令キャッシュへのアクセス中（IF2 ステージで読み出したタグ部分と、プログラムカウンタの第 31〜第 13 ビットを比較することによって、キャッシュ・ミスかキャッシュ・ヒットかの判定を行なう）。

6.IF4

命令キャッシュへのアクセス終了（キャッシュ・ヒット / ミスの判定に応じてキャッシュミス処理ユニットが処理を行なう）。

7.RF1

命令のデコードを行なう。また、オペランドを得るためにレジスタファイルへのアクセスを開始する。

8.RF2

レジスタファイルからのオペランドのフェッチが完了する。また、符号拡張ユニットが即値命令および分岐命令で使用されるオペランドを生成する。

9.EX1

演算ユニットが 2 つのソースオペランドを使用して指定された演算を開始する。

指定された演算が加算の場合、このステージでは桁上げ生成・伝搬ユニットにより桁上げ生成・伝搬信号が生成され、また、桁上げ先見ユニットが桁上げ信号の生成を開始する。指定された演算が加算以外の場合、このステージでは何も行なわれない。

10.EX2

演算ユニットにおいて演算中。

指定する演算が加算の場合、このステージで桁上げ先見ユニットが桁上げ信号の生成を完了する。指定された演算が加算以外の場合、このステージでは何も行なわれない。

11.EX3

演算ユニットにおいて演算が完了する。

指定された演算が加算の場合、EX1 および EX2 ステージで生成された部分和と桁上げ先見信号使用した演算が完了する。指定された演算が加算以外の場合、このステージで演算が完了する。

また、2 ビットシフトユニットによって、分岐命令およびジャンプ命令のターゲットアドレスを決定する際に使用されるオペランドが生成される。

12.DF1

メモリ命令の場合、データキャッシュへのアクセスを開始する（ターゲットアドレスの第 12〜第 2 ビットまでをデコードを行なう）。

メモリ命令以外の命令の場合、このステージでは何も行なわれない。

13.DF2

メモリ命令の場合、データキャッシュへアクセス中（DF1 ステージでデコードした結果を使用して、ロード命令ならばキャッシュメモリのタグ部分とデータ部分を読み出し、ストア命令ならばキャッシュメモリへデータを書き込む)。

メモリ命令以外の命令の場合、このステージでは何も行なわれない。

14.DF3

メモリ命令の場合、データキャッシュへのアクセス中（DF2 ステージで読み出したタグ部分と、ターゲットアドレスの第 31〜第 13 ビットを比較することによって、キャッシュ・ミスかキャッシュ・ヒットかの判定を行なう)。

メモリ命令以外の命令の場合、このステージでは何も行なわれない。

また、アドレス計算ユニットが、次に命令フェッチを行なうアドレスの計算を開始する。

15.DF4

メモリ命令の場合、データキャッシュへのアクセス終了（キャッシュ・ヒット/ミスの判定に応じてキャッシュミス処理ユニットが処理を行なう)。

メモリ命令以外の命令の場合、このステージでは何も行なわれない。

アドレス計算ユニットは、次に命令フェッチを行なうアドレスの計算中。

16.WB1

演算結果やメモリからロードされたデータを、レジスタファイルへの書き戻す動作を開始する。

アドレス計算ユニットは、次に命令フェッチを行なうアドレスの計算を完了。

17.WB2

演算結果とロードされたデータをレジスタファイルへ書き戻す動作が完了する。

次に命令フェッチを行なうアドレスが、プログラムカウンタに格納される。

表 4.3: MUP のハードウェア量

ユニット名	ゲート数	レジスタ（ビット数）
スレッド選択・ユニット	2,108	578
命令フェッチ・ユニット	2,744	289
命令デコード・ユニット	609	-
機能ユニット	1,928	-
データフェッチ・ユニット	2,827	289
ライトバック・ユニット	101	-
レジスタアクセス・ユニット	1,831	17,408
アドレス計算・ユニット	1,598	-
例外処理・ユニット	4,623	1,664

4.4 合成結果

本節では、具体的に設計したMUP をゲートレベルで論理合成した結果について述べる。

論理設計の際には動作記述論理設計支援ツール PARTHENON[17] を使用した。また、論理合成は “ゲートのファンアウトを 1 に制限”、“ゲートの入力数は最大 4”、および “ゲートのドライブ能力が不足する場合はバッファを挿入” という条件の下で行なった。

各部のハードウェア量を表 4.3に示す。

プロセッサ全体のハードウェアは、表 4.3のハードウェア量に、さらにフリップフロップが “5,059 個” 加わることにより、以下に示すゲート数から構成される。

- ゲート数：48,705（フリップフロップ 5,059 個を含む）
- レジスタファイル等のビット数：20,228

また、表 4.4に各ユニットが複数スレッドを扱う場合のハードウェア量と単一スレッドを扱う場合のハードウェア量の比較を示す。この表から、複数スレッドを扱うためにハードウェア量が増加するのは、新たに追加した “スレッド選択・ユニット”、複数の資源（レジスタファイルなど）を扱う “レジスタアクセス・ユニット”、“例外処理・ユニット”、そしてキャッシュミス処理ユニットを含んでいる “命令フェッチ・ユニットおよびデータフェッ

表 4.4: 単一スレッドを扱う場合と複数スレッドを扱う場合のハードウェア量の比較

	単一スレッドを扱う場合		複数スレッドを扱う場合	
ユニット名	ゲート数	レジスタ（ビット数）	ゲート数	レジスタ（ビット数）
スレッド選択・ユニット	-	-	2,108	578
命令フェッチ・ユニット	1,571	-	2,744	289
命令フェッチ・ユニット増加分			1,173	289
命令デコード・ユニット	609	-	609	-
機能ユニット	1,928	-	1,928	-
データフェッチ・ユニット	1,654	-	2,827	289
データフェッチ・ユニット増加分			1,173	289
ライトバック・ユニット	101	-	101	-
レジスタアクセス・ユニット	101	1024	1,831	17,408
レジスタアクセス・ユニット増加分			1,730	1,6384
アドレス計算・ユニット	1,598	-	1,598	-
例外処理・ユニット	271	128	4,623	1,664
レジスタアクセス・ユニット増加分			4,352	1,536
スレッド ID 用ラッチ	-		340 ビット	

表 4.5: 各ステージの論理段数

ステージ	段数	ステージ	段数
TS1	3	EX1	4
TS2	4	EX2	5
IF1	5	EX3	3
IF2	-	DF1	5
IF3	4	DF2	-
IF4	5	DF3	4
RF1	4	DF4	5
RF2	5	WB1	4
-	-	WB2	5

チ・ユニット” であることがわかる。さらに、スレッド ID 用パイプラインのパイプライン・ラッチが必要となる。
その他のユニットは、ハードウェア量は増加しない。

次に、各パイプライン・ステージの論理段数を表 4.5に示す。各パイプライン・ステージの論理段数はすべて、メモリセル・アレイの読み出し時間 $1nsec$ を考慮し、6 段以下としている。なお、IF2 と DF2 ステージの論理段数が無記入であるのは、これらのステージではメモリセル・アレイへのアクセスが行なわれているだけなので、論理段数を明示的に示すことができないためである。

また表 4.5から、ここで設計した MUP のクリティカルパスはメモリセル・アレイの読み出し部分であることがわかる。このメモリセル・アレイの読み出し時間は、パイプライン・ラッチ分（論理段数 2 段に相当）を含めて $1nsec$ であるので、本 MUP は $1GHz$ で動作可能である。

4.5 考察

合成結果から、パイプラインを 17 ステージで構成する MUP を、ゲート数 “48K”、レジスタファイル等のビット数 “20K” で構成できることを示した。また、パイプラインのクリティカルパスを評価することによって、本 MUP は 1GHz のクロックで動作できることを示した。

本 MUP を構成するハードウェアのほとんどは、レジスタセット（19K ビット）とパイプライン・ラッチのためのフリップフロップ（5K 個）である。逆に、制御用のハードウェア（ゲート数：3K）はわずかである。

これは、“パイプライン・ステージ数分のスレッドを同時に扱うことによって命令間の依存関係を無くしパイプライン制御を簡単化する” という MUP の特徴を表している。また、パイプライン・ステージを現在の 17 ステージから増加させた場合、主に増加するハードウェアは‘レジスタセット” である。このことは、LSI 化にとって有利である。

ここで設計した MUP の問題点として、スレッド番号がデコードされた形でスレッド ID 用パイプラインを流れるために、スレッド ID 用パイプラインのフリップフロップが多数必要となるという点である。

この問題を解決する方法として、スレッド番号をエンコードされた形でスレッド ID 用パイプラインに流し、スレッド番号が必要となるパイプライン・ステージの直前のステージでスレッド番号をデコードし、次のステージで使用するという方法が考えられる。ただし、この方法でもデコードされたスレッド番号を保持するラッチが必要となるが、スレッド番号を必要とするパイプライン・ステージは全パイプライン・ステージ中のわずかであるので、パイプラインが長くなるほど、デコードしたスレッド番号を流す場合と比較してラッチの数を少なくすることができる。

第 5 章

結論

本論文では、まず第 2 章でマルチスレッド型プロセッサ・アーキテクチャと関数型プログラムとの関係について述べた。関数型プログラムは多数のスレッドを展開することが可能であるので、この多数のスレッドをマルチスレッド型プロセッサに供給することによって、プロセッサの “パイプライン制御の簡単化” および “動作クロックの高速化による高スループット” を実現することを論じた。

第 3 章では、本論文で提案するマルチスレッド型ウルトラパイプライン・プロセッサ・アーキテクチャを提案した。本アーキテクチャの持つ特徴は次の通りである。

- パイプラインステージ数と同数のレジスタセット
- スレッド ID 用パイプライン
- ラウンドロビン選択方式による実行スレッドの切り替え
- 命令キャッシュとデータキャッシュの分離
- キャッシュメモリとレジスタファイルのパイプライン化
- 複数スレッド間でのキャッシュメモリの共有
- 大容量キャッシュの搭載
- パイプラインをストールしないキャッシュミスの取り扱い

これらの特徴により動作クロックの高速化による高いスループットを得ることが可能であることを論じた。第 4 章では、マルチスレッド型ウルトラパイプライン・プロセッサ・アーキテクチャの性能を見積もるために、具体的な設計とゲートレベルでの論理合成を行なった。この合成結果から、ゲート数 “48k”、レジスタファイル等のビット数 “20k” でプロセッサを構成することができること、さらにプロセスルールとして $0.1mum$ の技術を仮定すると $1GHz$ で動作可能であることついて述べた。

以上、本論文ではマルチスレッド型プロセッサ・アーキテクチャと関数型プログラムの特徴を組み合わせることによって、並列処理のための高性能なマルチスレッド型プロセッサ・アーキテクチャの提案し、ゲートレベルでの論理合成を行なうことによる性能見積りについて述べた。

マルチスレッド型ウルトラパイプライン・プロセッサは、“動作クロックの高速化” と “LSI 化に適したハードウェア構成” という点から、MOS デバイスの微細化を有効に使うことができるプロセッサ・アーキテクチャであることが主張できる。

最後に、今後の課題について述べる。17 ステージで構成する MUP のパイプライン 1 ステージの大きさは、キャッシュメモリのメモリセル・ アレイの読み出し時間によって決定された。しかし、メモリセル・アレイを複数に分割することによって高速に動作することが可能なパイプライン・キャッシュの研究が行なわれている [1]。そこで、プロセッサのパイプラインをどこまで細分化できるかという検討が必要となる。

また、プロセッサの総合的評価を行なう環境（コンパイラやテストプログラムなどのソフトウェアに関する環境）を整え、実際に関数型プログラムの並列実行評価が可能なようにし、アプリケーションプログラムにより MUP の有効性を実証する必要がある。

謝辞

PARTHENON の使用を快諾して下さいました日本電信電話株式会社に感謝致します。また、PARTHENON に関しお教え頂いた京都大学の中村行宏教授、NTT 研究所の小栗清氏に感謝致します。

また、日比野研、横田研の皆様には、度々、有益なアドバイスをいただきありがとうございました。特に、日比野研の相原孝一氏には、パイプライン化キャッシュメモリの構成について助言を頂き本当にありがとうございました。

さらに、適切な助言をして頂きました堀口進教授、横田治夫助教授、丹康雄助手に深く感謝致します。

最後に、本研究を進めるに当たり日頃熱心に御指導下さいました、日比野靖教授に心より感謝致します。

参考文献

[1] 相原 孝一, マルチスレッド型プロセッサ向きのキャッシュ機構のパイプライン化に関する研究, 北陸先端科学技術大学院大学修士論文, 1997

[2] 雨宮 真人, 田中 護, コンピューターアーキテクチャ, オーム社, 1988

[3] T.J.W.Clarke, The D-RISC: An architecture for use in multiprocessor, Proc. of the 3rd Conference on Functional Programing Language and Computer Architecture, pp.16-33, 1987.

[4] J.S.Emer, and E.S.Davidson, Control Store Organization for Multiple Stream Pipelined Processor, Proc. of International Conference on Parallel Processing, pp.43-48, 1978.

[5] M.J.Flynn, Some Computer Organization and Their Effectiveness, IEEE Transactions on Computer, pp.948-960, Sept, 1972.

[6] John L. Hennecy and David A. Patterson, Computer Architecture: A Quantitative Approach 2nd Edition, Morgan Kaufmann Publishers Inc, 1996.

[7] 井口 秀之, 次世代集積技術による RISC アーキテクチャプロセッサの設計と評価, 北陸先端科学技術大学院大学修士論文, 1995

[8] S.L.Peyton Jones, C.Clack, J.Salkild, and M.Hardie, GRIP - A high-performance architecture for parallel graph reduction, Proc. of the 3rd Conference on Functional Programing Language and Computer Architecture, pp.98-112, 1987.

[9] Gerry Kane, 監訳 前川 守, mips RISC アーキテクチャ -R2000/R3000-, 共立出版株式会社, 1994.

[10] 菅野 卓雄 監修, 飯塚 哲哉 編, CMOS 超LSI の設計, 培風館, 1989.

[11] K.F.Lee, et al., Room Temperature 0.1μm CMOS Technology with 11.8ps Gate Delay, IEDM TECHNICAL DIGEST, pp.131-134, 1993.

[12] Kazuya Masu, et al., Multilevel Metallization Based on Al CVD, Symposium on VLSI Technology Digest of Technical Papers, pp.44-45, 1996.

[13] 松山 泰男, 富沢 孝, VLSI の設計入門, 共立出版株式会社, 1983.

[14] Daniel C. McCrackin, Eliminating Interlocks in deeply Pipelined Processor by DelayEnforced Multistreaming, IEEE Trans. on Computer, Vol.40, No.10, pp.1125-1132, Oct 1991.

[15] MIPS Technology Inc., R4400 MICROPROCESSOR PRODUCT INFORMATION, 1996.

[16] 日経BP 社, 日経エロクトロニクス, No.630, 1995.

[17] 小栗 清, 名古屋 彰, 野村 亮, 雪下 充輝, はじめてのPARTHENON, CQ 出版社, 1994.

[18] Won Woo Park, Donald S. Fusell and Roy M. Jenevein, Performance Advantages of Multithreaded Processors, Proc. of the Third Int'l Conf. on Parallel Processing, Vol.1, pp.97-101, 1991.

[19] David A. Patterson and John L. Hennecy, COMPUTER ORGANIZATION & DESIGN THE HARDWARE/SOFTWARE INTERFACE, Morgan Kaufmann Publishers Inc, 1994.

[20] R. Guru Prasadh and Chuan-lin Wu, A Benchmark Evaluation of a Multi-threaded RISC Processor Architecture, Proc. of the 20th Int'l Conf. on Parallel Processing, Vol.1 pp.84-91, 1991.

[21] Klaus Erik Schauser, David E. Culler and Thorsten von Eicken, Compiler-Controlled Multithreading for Lenient Parallel Languages, Proc. of the 5th ACM Conference on Functional Programing Language and Computer Architecture, pp.50-72, 1991.

[22] L.Su, et al., A High-Performance 0.08μm CMOS, VLSI Technology Digest of Technical Papers, pp.12-13, 1996.

[23] Daniel Tabak, 訳 大森 健児, RISC システム -新しいプロセッサのアーキテクチャ-, 海文堂出版株式会社, 1991.

[24] Y.Taru, et al., High Performance 0.1μm CMOS Devices with 1.5V Power Supply, IEDM TECHNICAL DIGEST, pp.127-130, 1993.

[25] Kenneth R.Traub, Multithread Code Generation for Dataflow Architecture from Non-Strict Programs, Proc. of the 5th ACM Conference on Functional Programing Language and Computer Architecture, pp.73-101, 1991.

[26] D.M.Tullsen, S.J.Eggers, and H.M.Levy, Simultaneous multithreading: Maxmizing on-chip parallelism, Proc. of the 22nd Annual International Symposium on Computer Architecture, pp.392-403, 1995.

[27] J.D.Ullman, 訳 都倉信樹, 萩原兼一, 和田幸一, 平山正治, 瀬尾和男, VLSI 計算の諸側面 -VLSI 設計のための理論とアルゴリズム-, 近代科学社, 1990.

[28] B.Willkinson, 訳 高橋義造, 渡辺尚, 小林真也, 計算機設計技法 -マルチプロセッサシステム論-, 株式会社トッパン, 1994.

第 A 章

例外処理機構の詳細

ここでは、各例外が発生した場合にその例外をどのように処理するかについて詳細に説明する。

例外処理

MUP で発生する例外要因とその例外のパイプライン上での発生場所を以下に示す。

- 外部割り込み（DF2 ステージ）
- 未定義命令例外（RF1 ステージ）
- 特権命令例外（RF1 ステージ）
- アドレスエラー（IF1 ステージ、DF1 ステージ）
- 算術オーバーフロー（EX3 ステージ）
- システムコール（RF1 ステージ）

外部割り込みは 6 種類あり、外部割り込みレジスタの IntMask フィールドを “0” にすることにより 6 種類を個別にマスクすることが可能である。

3.2.3節で述べた通り、プロセッサ内部で生じる例外（未定義命令例外、特権命令例外、アドレスエラー、算術オーバーフロー、システムコール）が発生した場合には、例外の発生したストリームの実行だけを中断し、このストリームに対応する PC 等を書き換えて例

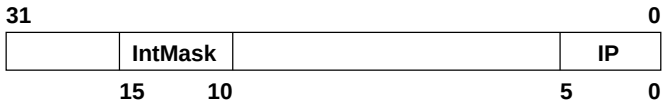

図 A.1: 外部割り込みレジスタ

外処理を行なう。外部割り込み処理は、実行中のスレッドが DF2 ステージまで例外を発生せず、かつ割り込みマスクがクリアされていれば、そのスレッドの実行だけを中断して外部割り込みを受け付ける。

例外処理をサポートするレジスタ

外部割り込みレジスタ

外部割り込みレジスタを図 A.1に示す。外部割り込みレジスタは、“IntMask フィールド（外部割り込みをマスクするフィールド）” と “IP フィールド（外部割り込み発生原因を示す)” の 2 つのフィールドで構成する。本レジスタはすべてのストリームで共有する。

- IntMask(Interrupt Mask)：6 種類の外部割り込みを個別にマスクする
- IP（Interrupt）
  6 種類の外部割り込みの内、どの外部割り込みが生じたかを示している。
  0：割り込みが生じていない
  1：割り込みが生じている

ステータスレジスタ

ステータスレジスタを図 A.2に示す。ステータスレジスタは、プロセッサの状態保存を行なう。本レジスタは各スレッド毎に提供される。

- KUc(Kernel/User mode : Current)：現在のモードを示す（1：ユーザーモード、0：カーネルモード）

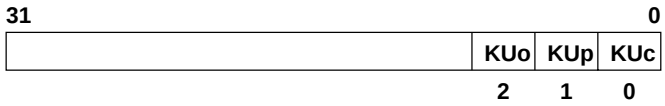

図 A.2: ステータスレジスタ

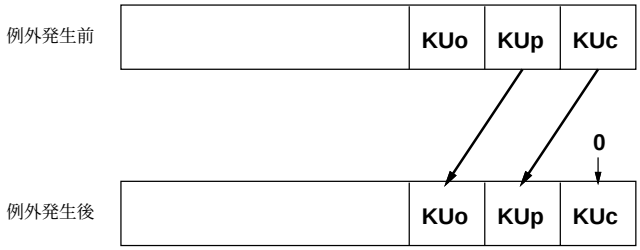

図 A.3: ステータスレジスタと例外識別

- KUp(Kernel/User mode : Previous)：例外発生前のモードを示す。
- KUo(Kernel/User mode : Old)：2 レベルの例外発生前のモードを示す。

MUP が例外に応答するとき、MUP は例外処理を行なうために中断するスレッドに対応するステータスレジスタの現在のカーネル/ユーザーモード (KUc) ビットを、前モードビット (KUp) に保存する。前モードビット (KUp) は、旧モードビット (KUo) に保存される。現モードビット (KUc) はクリアされ、プロセッサはカーネルモードになる。この 3 レベルのモードビットが備わっているため、ソフトウェアがステータスレジスタの内容を保存する前に、MUP は 2 レベルの例外に応答することができる [9]。例外処理による KU ビットの動作を図 A.3に示す。

例外処理ルーチンが完了すると、MUP は可能ならば例外前のシステム状況に復帰しなければならない。これは、rfe(Restore From Exception) 命令によって行なわれる。

rfe 命令が実行されると、ステータスレジスタの前カーネル/ユーザーモード (KUp) ビットを、現モードビット (KUc) に復元する。また、旧モードビット (KUo) を前モードビットに復元する。旧モードビットはそのままである。rfe 命令が実行されたときのステータスレジスタの動作を図 A.4に示す。

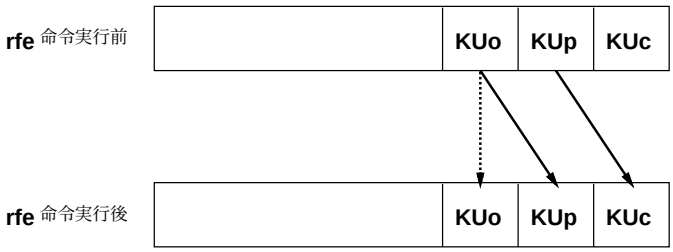

図 A.4: 例外処理からの復元

図 A.5: 例外原因レジスタ

例外原因レジスタ

例外原因レジスタを図 A.5に示す。例外原因レジスタは最後に発生した例外を示す。本レジスタは各スレッド毎に提供される。

- ExcCode ビット
  例外発生原因を示している。
  0：外部割り込み
  1：アドレスエラー例外
  2：システムコール例外
  3：特権命令例外
  4：未定義命令例外
  5：算術オーバーフロー例外
  6〜15：予約

例外 PC(Exception PC)

例外 PC を図 A.6に示す。例外 PC には、例外が処理された後に処理が再開されるアドレスが入る。ハードウェアは、このレジスタに例外を引き起こした命令のアドレスを入れ

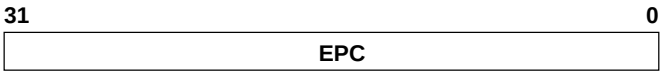

図 A.6: 例外 PC

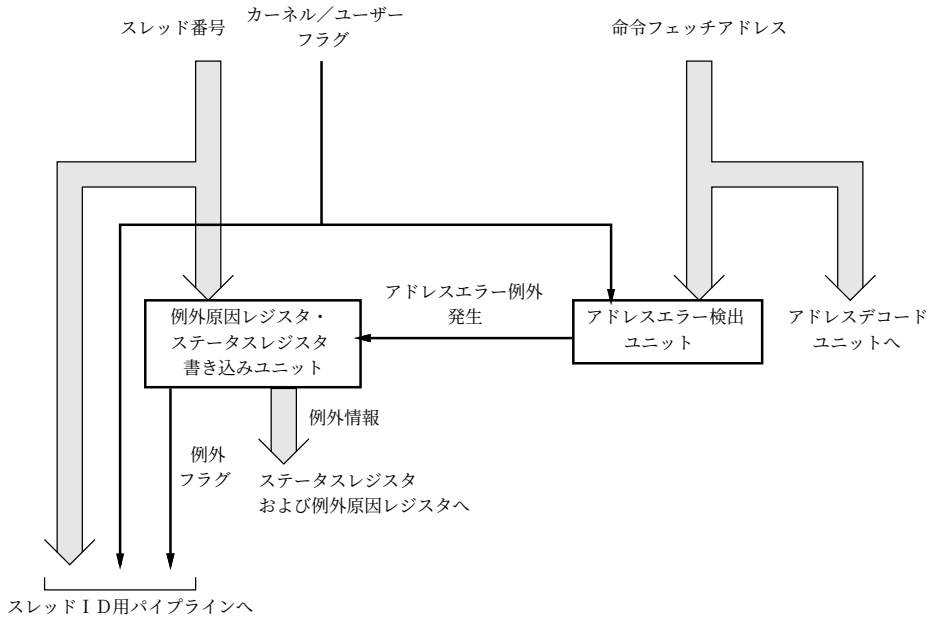

図 A.7: 命令フェッチ時にアドレスエラーが発生した場合の例外処理

る。本レジスタは各スレッド毎に提供される。

各例外が発生した場合の例外処理手順

各々の例外処理を行なうために必要となるユニットとその処理手順を示す。

命令フェッチ時にアドレスエラー例外が発生した場合

例外処理に関する IF フェーズのデータパスを図??に示す。命令フェッチを行なう際にアドレスエラーが発生すると、“アドレスエラー検出ユニット” と “例外原因レジスタ・ステータスレジスタ書き込みユニット” によって例外処理を行なう。

以下に例外が発生した場合の、各ユニットの動作について述べる。

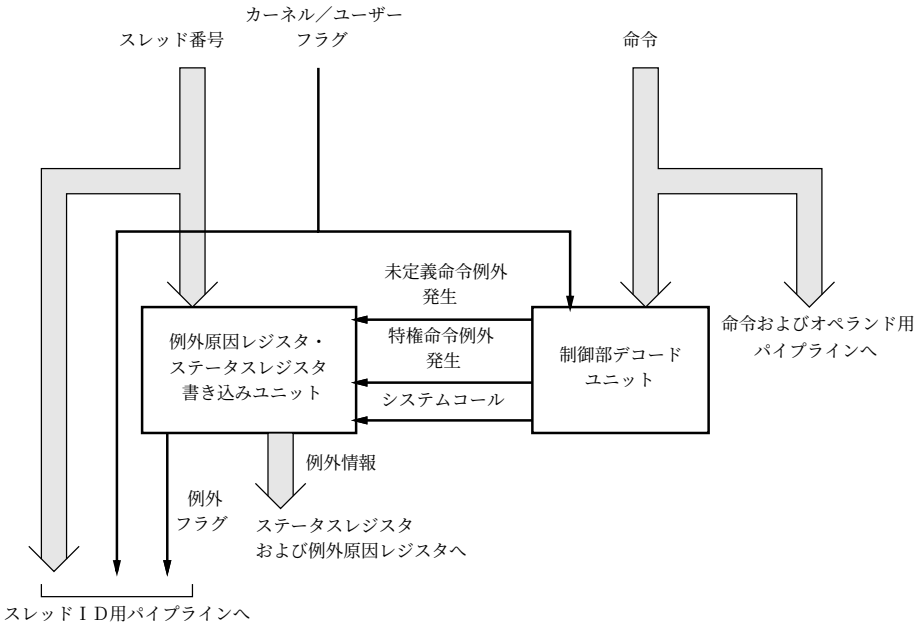

図 A.8: 未定義命令・特権命令・システムコールが生じた場合の例外処理

アドレスエラー検出ユニットは、アドレスの下位 2 ビットが “00” 以外[1]、またはカーネル/ユーザービットが “0” すなわちユーザーモード時にアドレスがカーネル空間を指し示しているならば、例外原因レジスタ・ステータスレジスタ書き込みユニットに対して、“例外発生” を伝える。

例外原因レジスタ・ステータスレジスタ書き込みユニットは、例外発生を伝えられると、例外が発生したスレッドに対応する例外原因レジスタとステータスレジスタに対して、それぞれ例外原因を表す ExcCode と新しいカーネル/ ユーザービットを書き込む。さらに例外原因レジスタ・ステータスレジスタ書き込みユニットは、スレッド ID 用パイプラインに例外が発生したか否かを示す “例外発生フラグ” を投入する。

未定義命令例外・特権命令例外・システムコールが発生した場合

例外処理に関する RF フェーズのデータパスを図??に示す。命令デコードを行なう際に未定義命令例外、特権命令例外、およびシステムコールのいずれかが生じると、“制御部デコードユニット” と “例外原因レジスタ・ステータスレジスタ書き込みユニット” によっ

[1] 1 命令は 32 ビット長なので、アドレスは 4 で割り切れなければならない

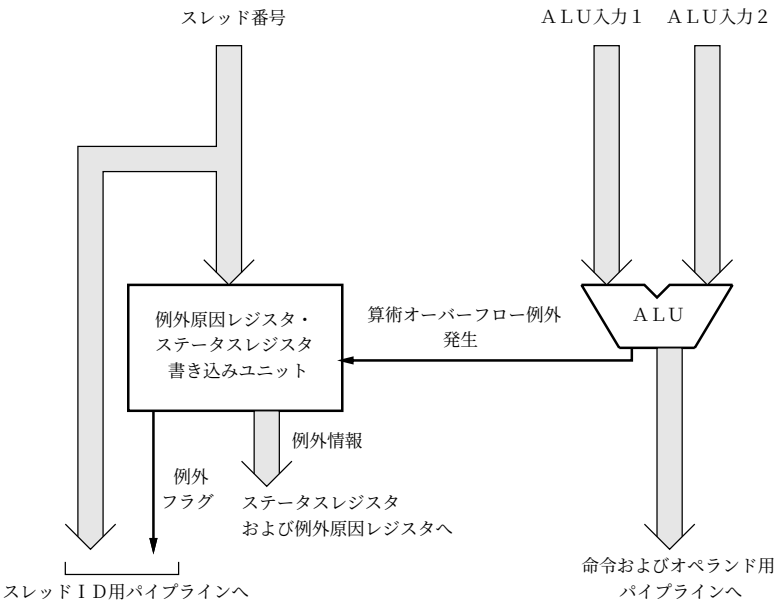

図 A.9: 算術オーバーフローが生じた場合の例外処理

て例外処理を行なう。

制御部デコードユニットは、未定義命令をデコード、ユーザーモード時に特権命令をデコード、およびシステムコール命令のいずれかをデコードすると、例外原因レジスタ・ステータスレジスタ書き込みユニットに対して、それぞれ “未定義命令例外発生”、“特権命令例外発生”、“システムコール” を伝える。

例外原因レジスタ・ステータスレジスタ書き込みユニットは、例外発生を伝えられると命令フェッチ時にアドレスエラー例外が生じた場合と同様に、例外が発生したスレッドに対応する例外原因レジスタとステータスレジスタに対して、それぞれ例外原因を表す ExcCode と新しいカーネル/ユーザービットを書き込む。さらに例外原因レジスタ・ステータスレジスタ書き込みユニットは、スレッド ID 用パイプラインに例外が発生したか否かを示す “例外発生フラグ” を投入する。

算術オーバーフロー例外が発生した場合

例外処理に関する EX フェーズのデータパスを図??に示す。算術計算を行なった結果によって算術オーバーフロー例外が生じると、“ALU” と “例外原因レジスタ・ステータス

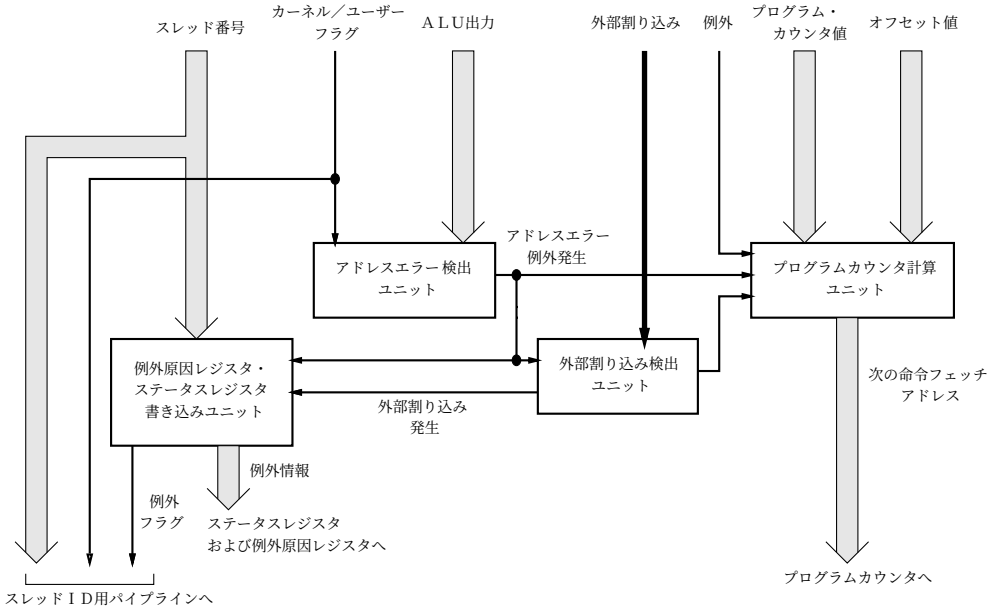

図 A.10: データフェッチ時にアドレスエラーが発生、および外部割り込みが生じた場合の例外処理

レジスタ書き込みユニット” によって例外処理を行なう。

ALU は、算術計算によって算術オーバーフローが生じると、例外原因レジスタ・ステータスレジスタ書き込みユニットに対して、“算術オーバーフロー例外発生” を伝える。

例外原因レジスタ・ステータスレジスタ書き込みユニットは、例外発生を伝えられると、例外が発生したスレッドに対応する例外原因レジスタとステータスレジスタに対して、それぞれ例外原因を表す ExcCode と新しいカーネル/ ユーザービットを書き込む。さらに例外原因レジスタ・ステータスレジスタ書き込みユニットは、スレッド ID 用パイプラインに例外が発生したか否かを示す “例外発生フラグ” を投入する。

データフェッチ時にアドレスエラーが発生、または外部割り込みが生じた場合

例外処理に関する DF フェーズのデータパスを図??に示す。データフェッチ時にアドレスエラーが発生、または外部割り込みが生じると、“アドレスエラー検出ユニット”、“外部割り込み検出ユニット” および “ 例外原因レジスタ・ステータスレジスタ書き込みユ

ニット” によって例外処理を行なう。

アドレスエラー検出ユニットは、位置合わせされていないワード[2]、に対するロードまたはストアを行なおうとした時、またはカーネル/ユーザービットが “0” すなわちユーザーモード時にアドレスがカーネル空間を指し示しているならば、外部割り込みユニット、例外原因レジスタ・ステータスレジスタ書き込みユニット、およびプログラムカウンタ計算ユニットに対して、“アドレスエラー例外発生” を伝える。

外部割り込み検出ユニットは、アドレスエラー検出ユニットからアドレスエラー例外発生が伝えられた場合、外部割り込み検出を一切行なわない。アドレスエラー検出ユニットからアドレスエラー例外発生が伝えられなかった場合は、外部割り込みの検出を行ない、割り込みが発生していれば、例外原因レジスタ・ステータスレジスタ書き込みユニットおよびプログラムカウンタ計算ユニットに対して、“外部割り込み発生” を伝える。

例外原因レジスタ・ステータスレジスタ書き込みユニットには、“アドレスエラー例外発生” と “外部割り込み発生” のいずれかの信号が伝えられる。例外原因レジスタ・ステータスレジスタ書き込みユニットは、伝えられた例外発生に応じて、例外原因レジスタとステータスレジスタに対して、それぞれ例外原因を表す ExcCode と新しいカーネル/ユーザービットを書き込む。さらに例外原因レジスタ・ステータスレジスタ書き込みユニットは、スレッド ID 用パイプラインに例外が発生したか否かを示す “例外発生フラグ” を投入する。

プログラムカウンタ計算ユニットには、“前のフェーズで例外発生”、“アドレスエラー例外” および “外部割り込み発生” が伝えられる。プログラムカウンタ計算ユニットは、この 3 つの信号のどれも伝えられなかった場合には、通常の次のプログラムカウンタ値の計算を行なう。3 つの信号のいずれかが伝えられた場合、プログラムカウンタ計算ユニットは、プログラムカウンタに格納される値を、例外処理開始アドレス値[3]に変更し、同時に EPC レジスタに変更前のプログラムカウンタ値を格納する。

[2] 4 または 2 で割り切れないアドレスのワードまたはハーフワード

[3] 固定アドレス値である

第 B 章

配線長の見積り方法

配線長の見積り方法について述べる。

配線長は、各々のユニット内部の素子間の配線長と、各ユニット間の配線長を算術平均したものである。以下に各部の配線長の算出法について述べる。

B.1 ユニット内部の配線長の算出方法

B.1.1 素子の配置

素子間の配線長を求めるためには、各素子をどこに配置するかを決める必要がある。そこで、素子はなるべく正方形になるように配置する。しかし、素子の種類が異なると、表B.1に示すように素子の大きさも異なる。このような場合、図 B.1に示すように、なるべく正方形となるように配置する。

B.1.2 素子間の配線長

B.1.1節で述べたように素子の配置を決定したら、素子間の配線長を求めることができる。素子間の配線長の求め方を図 B.2を例に説明する。

- 素子間の位置関係が素子 1 と素子 3 のような場合

  素子 1 と素子 3 間の配線長は、(1) の矢印の長さである。

- 素子間の位置関係が素子 1 と素子 4 のような場合

表 B.1: 素子の大きさの例

2 入力 NOR	$28\lambda \times 20\lambda$
2 入力 NAND	$28\lambda \times 20\lambda$
3 入力 NOR	$28\lambda \times 28\lambda$
3 入力 NAND	$28\lambda \times 28\lambda$
4 入力 NOR	$36\lambda \times 28\lambda$
3 入力 NAND	$36\lambda \times 28\lambda$

2λ: 素子のゲート長

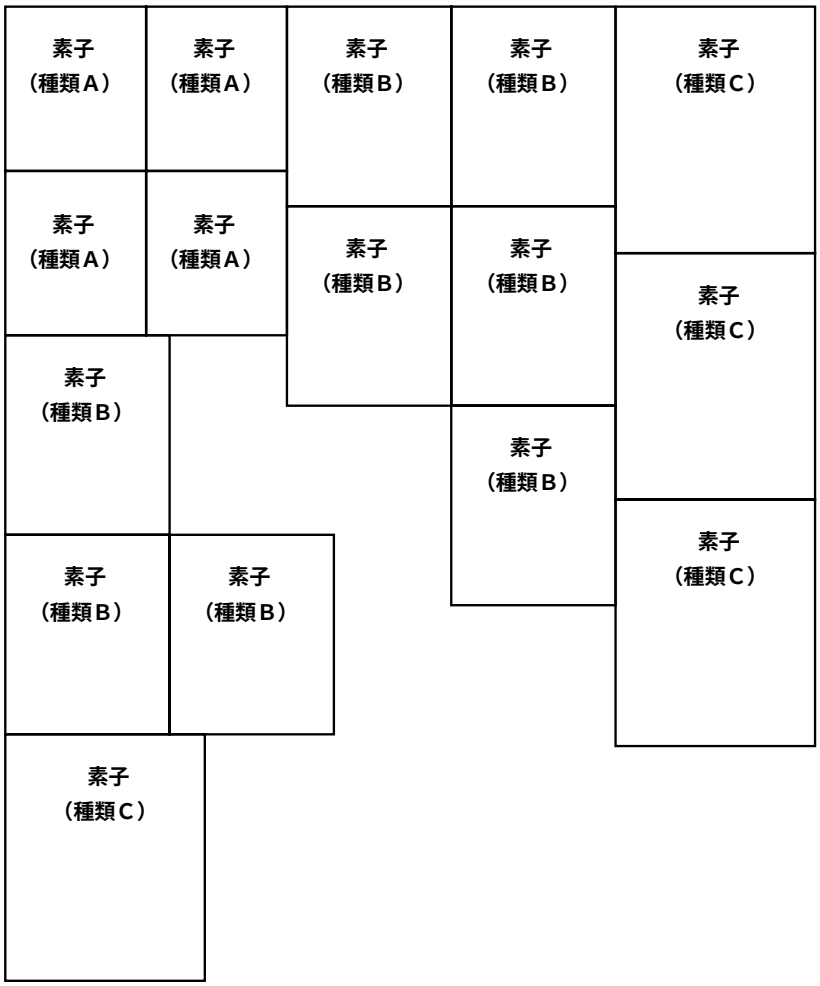

図 B.1: 素子の配置

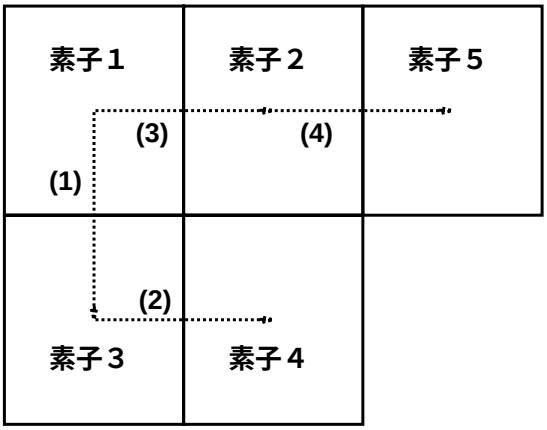

図 B.2: 素子間の配線長

素子 1 と素子 4 間の配線長は、(1) と (2) の矢印の長さの和である。

- 素子間の位置関係が素子 1 と素子 5 のような場合

  素子 1 と素子 5 間の配線長は、(3) と (4) の矢印の長さの和である。

このようにして、ユニット内部の素子間の配線長を求める。

B.2 ユニット間の配線長の算出方法

ユニット間の配線長も、先に述べた素子間の配線長を求めた場合と同様にして求める。

先に定めたユニット内の素子の配置により、各々のユニットの形と大きさが決まる。このユニットを、素子の配置の場合と同様に、なるべく正方形となるように配置し、ユニット間の配線長を求める。

第 C 章

MUP の SFL 記述

ハードウェア記述言語 SFL の記述例を以下に示す。ここで記述しているハードウェアは、17 ステージで構成されるマルチスレッド型ウルトラパイプライン・プロセッサ・アーキテクチャである。