卒業論文

# ツインテール・アーキテクチャ

平成 18 年 2 月 17 日提出

指導教員

坂井修一 教授

五島正裕 助教授

電子工学科

40453 平井 遥

# 概要

スーパスカラ・プロセッサの命令パイプラインにおいて、命令ウインドウより上流をフロントエンドと呼び、命令ウインドウおよびその下流をバックエンドと呼ぶ。従来のスーパスカラ・プロセッサでは演算器はバックエンドに配置され、命令の実行はバックエンドのみで行われる。これに対して我々はフロントエンド実行という手法を提案している。フロントエンド実行とはバックエンドに加えてフロントエンドにも演算器を配置し、実行可能な命令をフロントエンドでも実行することである。フロントエンド実行には従来のプロセッサに比べクリティカル・パス上の命令の実行間隔を狭める効果がある。本稿ではフロントエンド実行の考え方を押し進め、改良手法としてツインテール・アーキテクチャと呼ぶ手法を提案する。ツインテール・アーキテクチャはフロントエンド実行ステージを通常のパイプラインから独立させたものであり、これによってフロントエンド実行ステージによるパイプライン段数の増加はなくなる。この手法はフロントエンド実行において難点であった部分を改善してさらなる性能向上を図ることを目的とした手法である。

# 目 次

# 第1章　はじめに

## 1.1　背景

スーパスカラ・プロセッサの命令パイプラインにおいて、命令ウインドウより上流の部分を**フロントエンド**と呼び、命令ウインドウおよびその下流にあたる部分を**バックエンド**と呼ぶ。従来のスーパスカラ・プロセッサでは演算器はバックエンドに配置され、命令の実行はバックエンドのみで行われる。これに対して我々は**フロントエンド実行**(frontend execution) という手法を提案している。フロントエンド実行とはバックエンドに加えてフロントエンドにも演算器を配置し、フロントエンドの時点で実行可能な命令に関してはフロントエンドでも実行を行う手法である。

**図 1.1** の A および B に従来のプロセッサのパイプラインとフロントエンド実行を行う場合のパイプラインを示す。図のようにフロントエンド実行を行うステージは従来のパイプラインにおけるレジスタ読み出しステージとディスパッチ・ステージの間のステージである。図はフロントエンド実行ステージとして 1 ステージを割り当てた場合のパイプラインであるが、これに加えて演算器を多段に配置し、フロントエンド実行ステージとして 2 ステージ以上を割り当てることも提案されている。

フロントエンド実行は従来のプロセッサに比べ、クリティカル・パス上にある命令の実行間隔を狭めることができる効果がある。クリティカル・パスとはプログラム全体の実行時間を決定付けるパスのことであり、クリティカル・パス上の命令を最適に処理することができれば、プログラムの実行時間は最も短くなる。しかし、従来のプロセッサでは演算器の不足やスケジューリング能力の不足といった理由からこれを最適に実行することができていない。フロントエンド実行はクリティカル・パス上の命令の実行間隔を狭めることで性能を向上させることができる。先行研究によりフロントエンド実行によって従来のプロセッサに比べ 5〜15%の性能向上が得られることが示されている [1]。この仕組みについては 2.2.1 でより詳細に述べる。

しかし、フロントエンド実行を行う場合には図 1.1 に示したようにその実行ステージによってパイプラインの段数は増加してしまう。このことによる分岐予測ミス・ペナルティの増加は分岐予測の高いヒット率や予測ミス直後の命令がフロントエンド実行可能な場合が多いことなどから必ずしも大きな問題とはならない。だが、パイプライン段数の増加がなければより高い効果をあげることが可能であり、フロントエンド実行ステージを多段化することに対する余裕も増えるであろうと思われる。

この点をふまえた上で本稿では新たに**ツインテール・アーキテクチャ**(Twintail Architecture) という手法を提案する。この手法ではフロントエンドに実行ステージを設けるということは行わず、それに代わってフロントエンドでの処理を終えた命令を複製し、それぞれの以降の処理を In-order tail と Out-of-order tail と呼ばれる 2 つの異なる部位で行うのである。

ツインテール・アーキテクチャは新たに加える実行ステージを従来のパイプラインから切り離すことにより、実行ステージによるパイプライン段数の増加という難点を改善している。これによってフロントエンド実行以上の性能向上を狙う手法である。

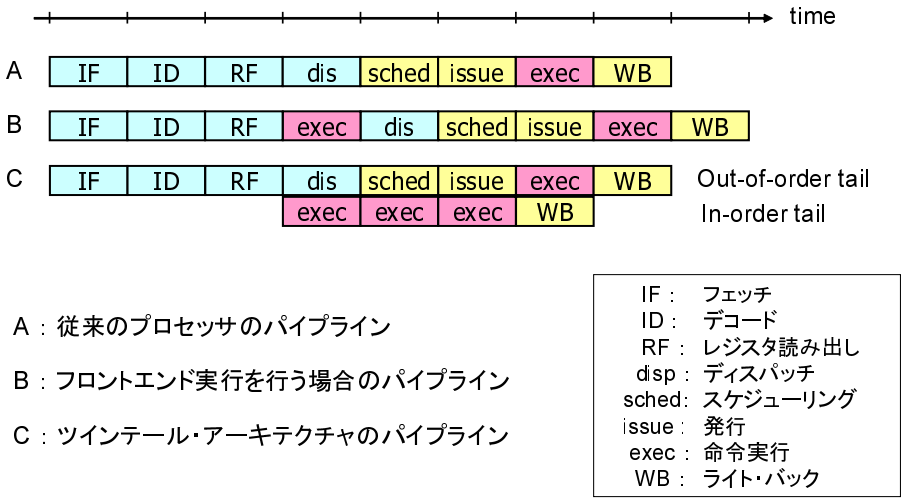


図 1.1: フロントエンド実行およびツインテール・アーキテクチャのパイプライン

図 1.1 の C としてツインテール・アーキテクチャのパイプラインの概要を示す。図のように In-order tail では命令はディスパッチ・スケジューリング・発行といった処理を行わずに即実行を行う。これはフロントエンド実行におけるフロントエンド実行ステージと同様の効果を持つ。また Out-of-order tail では従来のスーパスカラ・プロセッサのバックエンドと同様の処理を行う。この場合、図のように In-order tail と Out-of-order tail は別々のものなので In-order tail での実行ステージによって Out-of-order tail のパイプラインが深くなるということはないのである。このようにして、ツインテール・アーキテクチャはフロンドエンド実行の難点を改善しているのである。

## 1.2 構成

次に本稿の構成について述べる。本稿ではまず、第 2 章でフロントエンド実行の詳細やその効果について述べる。次に第 3 章でツインテール・アーキテクチャを提案する。そして、第 4 章で関連研究について述べた後、第 5 章でまとめと今後の課題について述べる。

# 第2章　フロントエンド実行

本章では先行研究であるフロントエンド実行についてより詳しい説明を行う。まずは、フロントエンド実行の仕組みについてその詳細を述べ、次にフロントエンド実行の効果について述べる。

## 2.1　フロントエンド実行の詳細

### 2.1.1　取り扱う命令とサイクル数

先行研究においてはフロントエンド実行にはそのステージとして 1〜2 サイクルを割り当てることが提案されている。

フロントエンド実行に 1 サイクルを割り当てる際にはシングル・サイクルで実行を完了することができる ALU 演算やシフト演算のみを行う。

また、フロントエンド実行に 2 サイクルを割り当てる際には依存関係にある 2 つのシングル・サイクル演算を行うことに加えて、ロード命令を行うことも提案されている。この場合には 1 ステージ目でアドレスの計算を行い、2 ステージ目にキャッシュへのアクセスを行う。アクセスするキャッシュとしては 1 ステージでアクセスすることが可能な 1KB 程度の 0 次キャッシュをフロントエンドにも用意することが考えられている [3]。

ロード命令を行う際には、0 次キャッシュにヒットしなかった場合にそのロード命令を引き続き行うかどうかにより二通りの方法が考えられる。ロード命令を引き続き行う方法については 2.2.2 においてその効果について述べる。

### 2.1.2　スーパスカラ・プロセッサの構成方式

現在の Out-of-order スーパスカラ・プロセッサを構成する方式には、リザベーション・ステーション（RS）とリオーダ・バッファ（RB）を併用する方式と、物理レジスタに対するリネーミングを行う方式が存在する。以降では前者を **RS+RB 方式**、後者を**物理レジスタ方式**と呼ぶことにする。以下ではまず両者が命令パイプライン上のどこでレジスタ読み出しを行うかに注目して両者の説明を行い、フロントエンド実行にとってどちらの方式が望ましいかということについて述べる。

**物理レジスタ方式**　物理レジスタ方式は物理レジスタの読み出しをバックエンドで行う。従って、物理レジスタ方式はフロントエンドとバックエンドを命令ウインドウにおいてほぼ完全に分離することができ、チップ上のレイアウトの点で有利である。しかし、物理レジスタ方式はバックエンドでレジスタ読み出しを行うため、命令が発行されてから実際に実行されるまでのレイテンシである発行レイテンシが長くなる。

**RS+RB 方式**　RS+RB 方式はレジスタの読み出しをフロントエンドで行う。これにより、レジスタ・ファイルはフロントエンドで読み出され、バックエンドで書き込まれることになるためフロントエンドとバックエンドを分離することは困難となる。さらに実行ステージからリザベーション・ステーションへの実行結果のフォワーディング・パスを用意することも必要である。従って、RS+RB 方式はチップ上のレイアウトに対する制約はかなり厳しいものとなる。しかし、レジスタ読み出しをフロントエンドで行うため、発行レイテンシは物理レジスタ方式の半分程度で済ませることが可能である。

フロントエンド実行を行う際にはフロントエンドの時点でレジスタの読み出しを行うことが必要である。レジスタの読み出しをフロントエンドで行う場合、その直下に演算器を配置すれば自然にフロントエンド実行を行うことができる。したがってフロントエンド実行を行う際には RS+RB 方式を採用することが提案されている。

### 2.1.3　導入が必要なハードウェア

フロントエンド実行では整数演算のみを行うことを基本とする。そのため、フロントエンド実行のために新たに導入が必要となるハードウェアは基本的にはフェッチ幅と同数の演算器のみである。上記の構成方式を採用すればフロントエンドにおいてレジスタ読み出しが行われるため、その直下に演算器を配置すれば、自然にフロントエンド実行を行うことができる。

また、すでに述べたようにロード命令を行う際にはフロントエンドに小容量の 0 次キャッシュを用意する必要がある。

しかし、これらはいずれも現存のスーパスカラ・プロセッサにとって大きなハードウェアではない。したがって、フロントエンド実行のために必要となるハードウェア・コストはそれほど過大なものではないと言ってよい。

### 2.1.4　フロントエンド実行機構

以上のことを踏まえた上で**図 2.1**、**図 2.2** に具体的なフロントエンド実行機構を示す。図 2.1 は 2 ウェイのスーパスカラ・プロセッサにおいて 1 段のフロントエンド実行を行う場合の実行機構であり、図 2.2 は 2 段のフロントエンド実行を行う場合の実行機構である。

**フロントエンド実行結果の書き戻し**　フロントエンド実行で得られた結果を後続の命令が利用するためにはバックエンドの書き戻し処理と同様の処理が必要となる。フロントエンド実行では書き戻しはディスパッチと並行して行われる。

フロントエンド実行ではプロセッサの構成方式として 2.1.2 で述べたようにリザベーション・ステーションとリオーダ・バッファを併用する方式を採る。そのため、仮にフロントエンド実行された命令 Ip と同じフェッチ・グループに属する命令 Ic がフロントエンド実行の結果を利用したい場合、フロントエンド実行の結果の書き込みは Ic のレジスタ読み出しには間に合わない。この場合、Ip のフロントエンド実行結果をリザベーション・ステーションの Ic に対応するエントリに書き込む必要がある。これには通常のフォワーディングと同様の処理が必要となる。

ハザードの回避のためには、以上で十分であり、フロントエンドの演算器からバックエンドの演算器へバイパスを設ける必要はない。Ip の結果は Ic のバックエンドの実行ステージの 1 サイクル以上前に得られるため、バイパスを設ける必要はないのである。

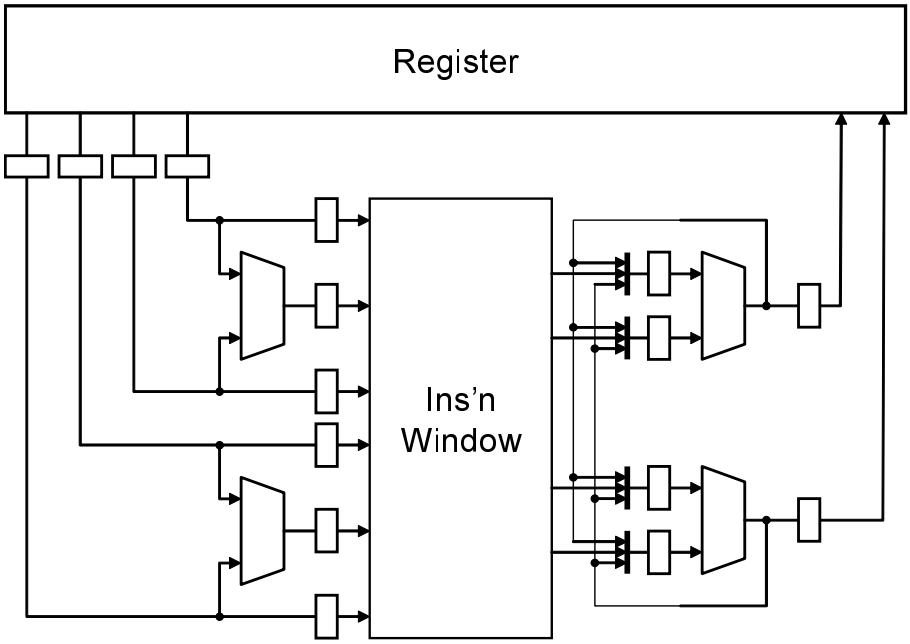


図 2.1: フロントエンド実行機構

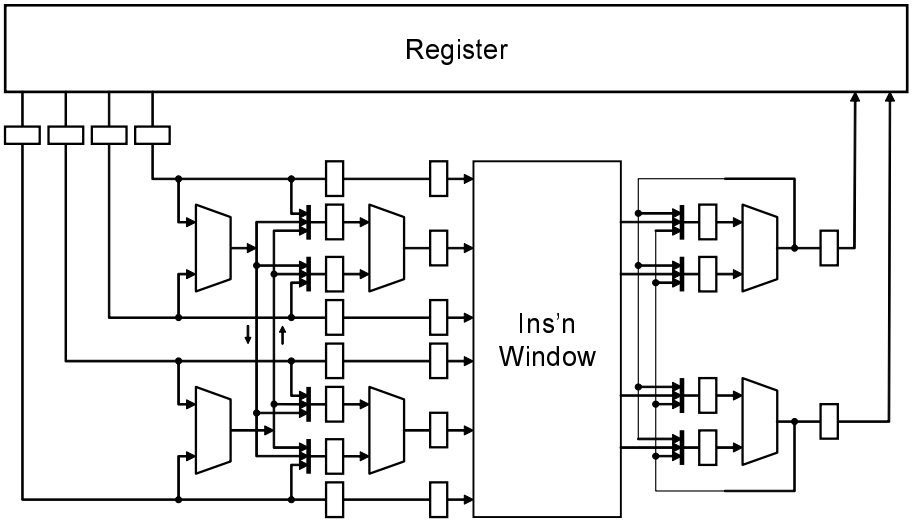


図 2.2: フロントエンド実行機構 (カスケード接続)

**カスケード接続の必要性**　先行研究によってフロントエンド実行ステージを多段にすることで依存関係にある複数の命令をフロントエンド実行することが提案されている。図2.2のようにフロントエンド実行を多段にする場合には、基本的には全対全のカスケード接続が必要となる。

図2.2のフロントエンド実行ユニットでは自分自身やフロントエンド実行の2段目から1段目へのバイパスを設けることは行っていないが、これらのバイパスを設けることも考えられる。この場合、フロントエンド実行できる命令数は増加すると考えられるがフロントエンド実行のメリットであるシンプルさは失われることになる。

## 2.2　フロントエンド実行の効果

次にフロントエンド実行の効果について述べる。以下ではまずフロントエンド実行によってクリティカル・パス上の命令の実行間隔を短縮することができることについて述べる。その後フロントエンド実行でロード命令を開始することでキャッシュ・アクセス・レイテンシを秘匿する仕組みについて簡単に述べる。

### 2.2.1　クリティカル・パス上の命令の実行間隔の短縮

フロントエンド実行には従来のプロセッサに比べ、クリティカル・パス上の命令の実行間隔を狭めることができる効果がある。

以下ではこれを**図 2.3** を用いて説明する。図 2.3 は縦軸を時間軸として命令がいつどの実行ユニットで実行されたかを表している。図 2.3 では数字やアルファベットが命令自身を表し、矢印は命令同士の依存関係を表したデータフロー・グラフである。四角は実行ユニットを表し、その中に数字やアルファベットが書かれている場合、実行ユニットがその命令を実行したことを表している。

今、図 2.3 のように 1 → 2 → 3 → 4 → 5、1 → A → B、1 → C → D → E の 3 つのパスを処理しなければならない場合について考える。この場合、このパスの処理時間を決めるクリティカル・パスは 1 → 2 → 3 → 4 → 5 である。これらのパスが従来のプロセッサによって処理された場合の例を図の左側に、フロントエンド実行を行うプロセッサによって処理された場合の例を図の右側に示す。

この場合、従来のプロセッサはクリティカル・パス上の命令を最適に実行することが出来ていない。この原因は演算器の不足およびスケジューリング能力の不足によるものである。図 2.3 に示した例では、プログラムを最速で処理するためには 1 → 2 → 3 → 4 → 5 のパスを優先して処理しなければならない。しかし、演算器数の制限とスケジューリング能力の不足からこの場合にはクリティカル・パス上にない命令の実行によってクリティカル・パス上の命令の実行が遅れてしまっていることが分かる。

次にフロントエンド実行を行う場合について考える。図 2.3 では命令 1 の実行後に十分時間がたってから命令 A や命令 2 がフロントエンド実行ステージに到達し、これらがフロントエンドで実行できた場合を示している。この場合、命令 A や命令 2 がフロントエンド実行されたことによって 1 → 2 → 3 → 4 → 5 や 1 → A → B のパス上の命令が無駄なく実行されているため、結果としてプログラム全体の実行時間を短縮することが出来ていることが分かる。

このように、フロントエンド実行は従来のプロセッサがクリティカル・パス上の命令を最適に処理できず、その実行間隔を必要以上に開けてしまっている場面でその効果を発揮する。クリティカル・パス上の命令をフロントエンド実行することでその実行間隔を縮めることができるため、結果としてプログラム全体の実行時間を短縮することができるのである。

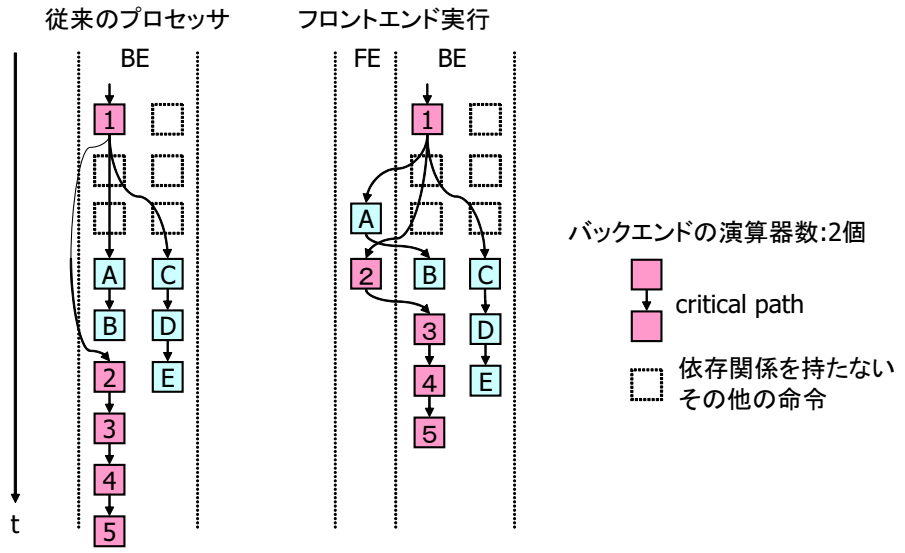


図 2.3: フロントエンド実行の効果

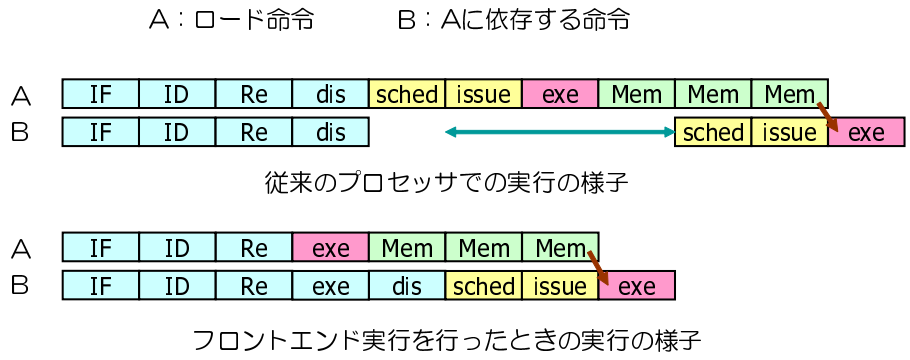


図 2.4: プリロードの効果

### 2.2.2 プリロードによるキャッシュ・アクセス・レイテンシの秘匿

フロントエンド実行でロード命令を行う場合にはそのロード命令がフロントエンドに用意した 0 次キャッシュにヒットしなかった場合、命令はキャンセルされ、バックエンドで再び実行される。これに対してフロントエンドのキャッシュにヒットしなかった場合にも引き続きバックエンドに存在する高次のキャッシュにアクセスを行う手法も提案されている。

この手法では、ロード命令に依存する後続の命令を投機的に発行することで性能向上を図っている。この様子を**図 2.4** に示す。

図のようにこの手法ではキャッシュのアクセスが後続の命令のスケジューリング処理と同じタイミングで行われていることが分かる。そのため投機が成功すれば、後続の命令からはこのロード命令のキャッシュ・アクセス・レイテンシが 0 サイクルになったかのように見えるのである。この手法を取り入れることによって平均約 11%の性能向上が見込めることが示されている [2]。

# 第3章　ツインテール・アーキテクチャ

前章まででフロントエンド実行の仕組みやその効果について説明した。しかし、フロントエンド実行を行う場合には第 1 章で述べたように全体のパイプライン段数は増加してしまうことになる。本稿ではこの点を改善し、さらなる性能向上を図る手法としてツインテール・アーキテクチャを提案する。

## 3.1　ツインテール・アーキテクチャの提案

フロントエンド実行を行う場合にはその実行ステージの分だけ全体のパイプライン段数は増加してしまう。これは、図 1.1 で示したようにフロントエンド実行ステージが従来のプロセッサにおけるパイプラインのレジスタ読み出しとディスパッチの間に新たなステージとして組み込まれているからである。その結果、フロントエンド実行できなかった命令のバックエンド実行はフロントエンド実行ステージの分だけ遅れてしまうことになる。

本稿ではこれを改善する手法としてツインテール・アーキテクチャと呼ばれる手法を提案する。この手法では**図 3.1** に示したようなパイプライン処理を行う。

まず、命令は従来のプロセッサと同様にフェッチされ、デコード、レジスタ読み出しの処理が行われる。その後、命令を複製し、これに対して In-order tail と Out-of-order tail で異なる処理を行う。

In-order tail ではフロントエンド実行ステージと同様の処理が行われる。In-order tail には演算器が多段に配置されており、そのステージで実行可能な命令は実行する。実行可能でない命令に関してはそのまま次のステージへと送られる。本稿では In-order tail として 3 段程度の実行ステージを考えている。

また、Out-of-order tail では従来のスーパスカラ・プロセッサと同様の処理を行う。Out-of-order tail は命令ウィンドウや演算器など従来のプロセッサと同様な構成をとる。命令はスケジューリング処理の後、実行ステージで実行される。

ただし、ツインテール・アーキテクチャでは Out-of-order tail は In-order tail のことを考慮に入れてスケジューリングを行わなければならない。したがって、従来のプロセッサと全く同じスケジューリング処理を行うというわけにはいかない。これについては 3.3 でその詳細について述べる。

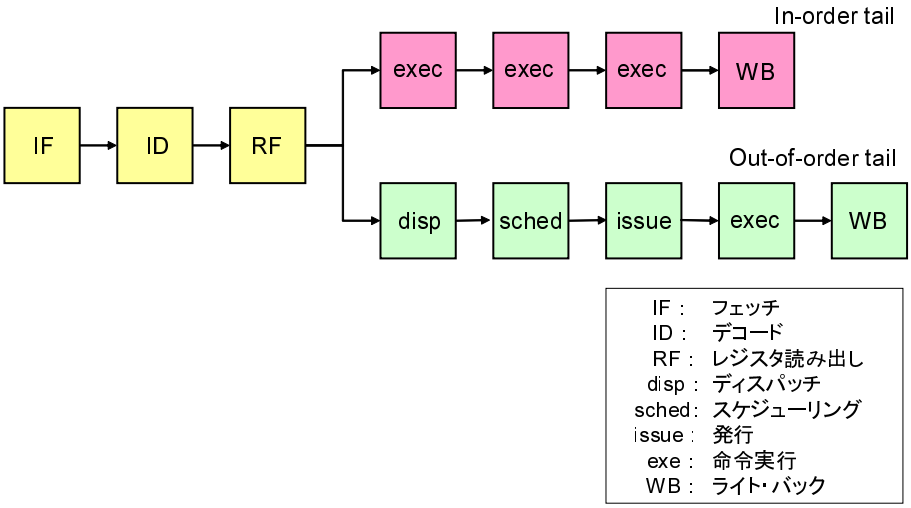


図 3.1: ツインテール・アーキテクチャ

## 3.2 提案手法の利点

ツインテール・アーキテクチャはフロントエンド実行ステージによってパイプライン段数が増加してしまうことを改善した手法である。

フロントエンド実行では従来のパイプラインにフロントエンド実行ステージを追加していたために全体のパイプライン段数が増加してしまっていた。それに対して、ツインテール・アーキテクチャでは新たに加える実行ステージを従来のパイプラインである Out-of-order tail から切り離し、全く別のパイプラインである In-order tail として独立させている。

したがって、In-order tail のパイプライン段数をいくら増加させたとしても Out-of-order tail のパイプライン段数が増加することはない。ツインテール・アーキテクチャではパイプライン段数の増加を気にすることなく、In-order tail の実行ステージを多段化することが出来るのである。

## 3.3 提案手法詳細

### 3.3.1 特殊な wakeup 論理とバイパス・ネットワークの必要性

すでに述べたとおり、ツインテール・アーキテクチャではレジスタ読み出しの後、命令は複製され、そのそれぞれが In-order tail と Out-of-order tail で別々の処理を行われることになる。そのため、何の対策も講じなければ命令は In-order tail と Out-of-order tail で冗長に実行されてしまう。これを防ぐためには In-order tail で実行された命令を Out-of-order tail の命令ウィンドウから取り除いてやる必要がある。そのためには Out-of-order tail に存在する命令ウィンドウが Out-of-order tail の実行ステージの情報に加えて In-order tail の実行ステージの情報を知っている必要がある。

さらに依存関係にある命令を 2 つの tail 間を通して back-to-back に処理するためには、In-order tail で依存元の命令の実行結果が得られるタイミングに合わせて、Out-of-order tail で依存先の命令の実行を開始できるようにしてやる必要がある。

これらの条件を満たすためにツインテール・アーキテクチャには通常とは異なる特殊な wakeup 論理が必要となる。

さらに、この場合には In-order tail での実行結果は Out-of-order tail での実行ステージの直前に得られることになる。したがって、フロントエンド実行の時とは異なり、ツインテール・アーキテクチャでは In-order tail の演算器から Out-of-order tail の演算器へのバイパス・ネットワークが必要となる。

以下ではこれらについてより詳細に説明する。

### 3.3.2 wakeup 論理

3.3.1 で述べたようにツインテール・アーキテクチャが効率的な動作を行うためには、Out-of-order tail の命令ウィンドウの wakeup 論理には従来のプロセッサの wakeup 論理に加えて工夫が必要となる。

**冗長実行の除去** In-order tail と Out-of-order tail で冗長に命令を実行してしまわないようにするには、最低でも In-order tail で命令が実行されたことを Out-of-order tail に知らせる必要がある。このためには wakeup 論理で実行された命令の情報を得る際に、Out-of-order tail で実行された命令に加えて In-order tail で実行された命令の情報も受け取れるようにすればよい。In-order tail で実行された命令を Out-of-order tail の命令ウィンドウから取り除いてやることで冗長な実行を防ぐことができると考えられる。

たとえば、ディスティネーション・オペランド・タグを放送することで命令ウィンドウに実行された命令の情報が伝えられる場合には、命令ウィンドウが Out-of-order tail からの放送に加えて In-order tail からの放送を受け取るようにすればよい。これを実現するには In-order tail では命令が実行された後に命令ウィンドウに対してディスティネーション・オペランド・タグを放送するようにすればよい。この場合、wakeup は Out-of-order tail の命令に対しては従来のプロセッサと同様にその select ロジックに依存することになるが、In-order tail の命令に対してはその実行に依存することになる。このようにすることで、Out-of-order tail は In-order tail での実行状況をある程度知ることができる。

しかし、ただ単に In-order tail で命令が実行されたという情報を受け取っているだけでは命令ウィンドウが In-order tail に関して持つ情報は十分なものとはならない。この情報だけでは、In-order tail と Out-of-order tail での冗長な実行を完全に排することは不可能である。このことについて図を用いて説明する。

**図 3.2** のように In-order tail と Out-of-order tail で同一の命令を処理している場合を考える。この場合、先に述べた方法により、In-order tail の実行ステージのうち 1 段目で実行された命令に関してはその次のサイクルの wakeup 論理によって、Out-of-order tail はその命令が実行されたことを知ることができる。しかし、図 3.2 に示したように In-order tail の 2 段目と 3 段目で実行された命令に関しては、命令の実行が伝えられるのが Out-of-order tail の命令ウィンドウでその命令が発行された後になってしまう場合がある。In-order tail での命令実行と同時かそれ以前のサイクルに命令ウィンドウでこの命令が wakeup され、次のサイクルで発行されてしまうのである。この場合にはその命令は In-order tail で実行した後で、Out-of-order tail で再度実行されてしまうことになる。これは In-order tail の実行ステージのいくつかが、Out-of-order tail のディスパッチ以降のステージと並行して存在するからである。したがって、冗長な実行を完全に排するためには wakeup 論理にさらなる工夫が必要である。

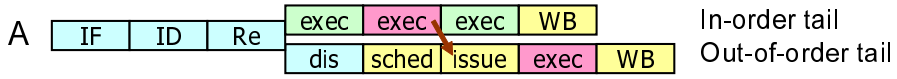


図 3.2: wakeup が間に合わない場合

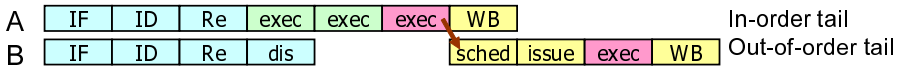


図 3.3: wakeup の遅れ

**理想的な動作**　ツインテール・アーキテクチャは依存関係にある命令のうち依存元の命令を In-order tail で実行してしまうことで、依存先の命令が Out-of-order tail で実行結果を待ち合わせる時間を短縮することができる。この場合、これらの命令を In-order tail と Out-of-order tail の間で back-to-back に処理することができれば理想的であると考えられるが、このためにも wakeup 論理をさらに工夫する必要がある。

図 3.1 から分かるように Out-of-order tail では sched ステージで wakeup 論理によって実行可能な命令を見つけだしてから実際にその命令を実行するまでには最低でも 2 サイクルの時間が必要である。そのため、たとえば**図 3.3** に示すように、In-order tail の 3 段目で実行される命令の実行が完了してから、この命令に依存する命令を Out-of-order tail の命令ウィンドウから発行した場合、その命令が実行されるのは最低でも 2 サイクル後ということになる。したがって、この命令が In-order tail の実行結果を用いて Out-of-order tail で back-to-back に実行されるためには、In-order tail の 3 段目で実行される命令を実際に実行される 2 サイクル前に知ることができなければならないのである。

以上より、ツインテール・アーキテクチャの理想的な動作のためには、命令ウィンドウが In-order tail の 1 段目での実行結果を受け取った際に 2 段目と 3 段目で実行されるであろう命令を把握して、それらの命令に依存する命令を wakeup することができればよいことになる。

本稿ではこのことを示すだけにとどめ、具体的な実装方法については言及しないものとする。具体的な実装についてはシミュレーションによる評価を行い、効果があると判断できた場合の今後の課題とする。

### 3.3.3　バイパス・ネットワーク

ツインテール・アーキテクチャでは In-order tail と Out-of-order tail の 2 つが並列にならぶ構成となる。また、In-order tail にはカスケード接続された演算器が多段に配置され、Out-of-order tail には従来のプロセッサと同様に命令ウィンドウやバックエンド演算器が配置される。したがって、これはちょうど図 2.1、図 2.2 におけるフロントエンド演算器を命令ウィンドウと直列ではなく並列に並べた構成となる。

しかし、ツインテール・アーキテクチャではフロントエンド実行の場合と違い、In-order tail の最後の実行ステージでの実行結果は次のサイクルに Out-of-order tail で使用されることがある。したがって、In-order tail の最後の段の演算器から Out-of-order tail の演算器へとバイパスを設ける必要がある。

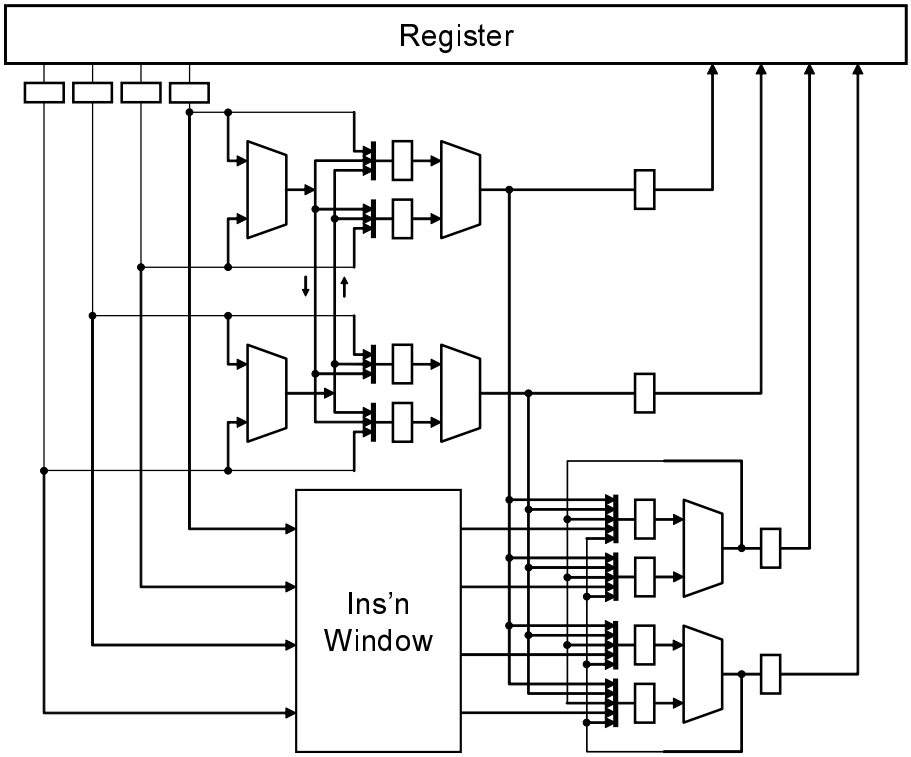


図 3.4: ツインテール・アーキテクチャの実行機構

以上より、ツインテール・アーキテクチャの実行機構は**図 3.4** のようなものとなる。図 3.4 は図 2.2 と対比するために In-order tail として 2 段の実行ステージを設けた場合の実行機構である。図において上半分が In-order tail、下半分が Out-of-order tail を表している。図 2.2 に示したフロントエンド実行機構と比べると分かるように、ツインテール・アーキテクチャではフロントエンドにあった演算器が In-order tail に移動した配置となっていることが分かる。

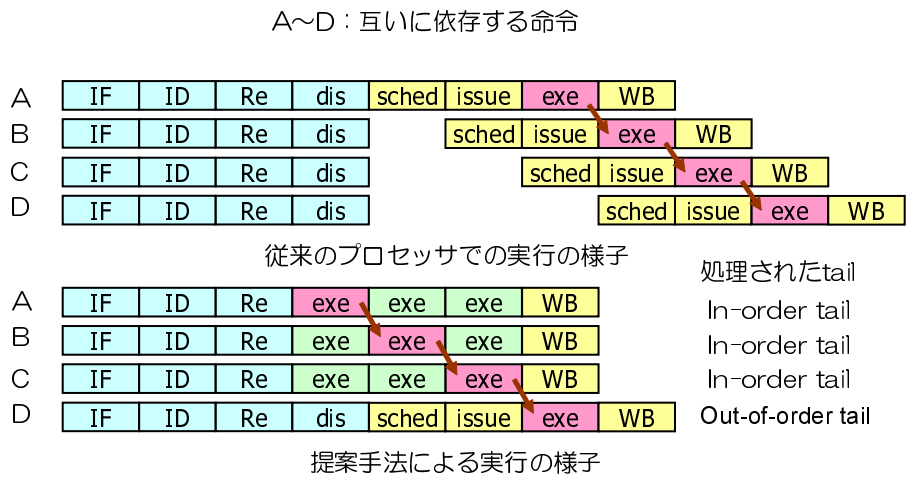


図 3.5: ツインテール・アーキテクチャの効果

## 3.4 ツインテール・アーキテクチャの効果

ツインテール・アーキテクチャはフロントエンド実行と同様、クリティカル・パス上の命令の実行間隔を狭めることができる効果がある。In-order tail と Out-of-order tail で異なったタイミングで命令の実行を行うため、命令をより適切なタイミングで実行することができるのである。ツインテール・アーキテクチャによる理想的な性能向上の一例を**図 3.5** に示す。

図 3.5 は同時にフェッチされた 4 つの命令 A～D を従来のプロセッサとツインテール・アーキテクチャで実行した場合の様子を表している。このとき B は A に、C は B に、D は C にそれぞれ依存しているものとする。

図に示したように、従来のプロセッサでは A～C の実行レイテンシの分だけ D は命令ウィンドウで待ち合わせを行わなければならない。

それに対し、ツインテール・アーキテクチャでは D は命令ウィンドウで待ち合わせることなくすぐに発行することが可能である。D のスケジューリング処理の間に A～C が In-order tail で実行されるため、Out-of-order tail で実行される D からは A～C までの実行レイテンシが 0 になったようにみえるのである。

この例はツインテール・アーキテクチャの効果が理想的に表れて従来のプロセッサに比べて 3 サイクルの短縮を行うことができている。実際にはプロセッサがプログラムを実行する中でこのように理想的な状況になることは少ないかもしれないが、仮に命令 A に依存する命令 B が Out-ot-order tail で実行されるだけでも 1 サイクル短縮の効果があり、このような状況は頻繁に起こるであろうと思われる。

また、フロントエンド実行と同様に In-order tail で命令が実行されることによって Out-of-order tail で実行しなければならない命令数が減少するという効果も考えられる。これによって Out-of-order tail での演算器の競合が緩和されるため、演算器の競合によるストールが減少することが期待できる。

# 第4章　関連研究

## 4.1　RENO: A Rename-Based Instruction Optimizer

Vlad ら [5] はリネーミングを工夫することで一部の命令をリネーミング・ステージで取り除く手法として RENO（RENaming Optimizer）を提案している。RENO はマッピング・テーブルの上手な利用と物理レジスタの共有構成を用いて命令を取り除き、依存関係を構成し直す手法であり、これまでにいくつかの方法が示されている。RENO の具体的な方法としては move 命令を取り除く RENOme、冗長な命令を取り除く RENOcse、投機的メモリ・バイパスによりロード命令を取り除く RENOra、即値演算を他の演算とまとめて行うことにより取り除く RENOcf などが提案されている。RENO は（１）取り除かれた命令をデータフロー・グラフから取り除ける（２）取り除かれた命令は以降でハードウェア資源を使用しない、といった利点を持っており、これによって 8〜13%の性能向上が見込めることが示されている。

RENO はフロントエンドのリネーミング・ステージにおいて命令の実行に変わる処理を行うことで、命令を取り除く手法であるといえる。したがってこの手法は従来のプロセッサの実行ステージ以外の場所で命令の実行処理を行うという意味でフロントエンド実行やツインテール・アーキテクチャと似た手法であるといえる。フロントエンド実行やツインテール・アーキテクチャでも従来以外の実行ステージで実行された命令は従来の実行ステージからは取り除かれたように見えるため、性能が向上するのである。

## 4.2　スーパスカラ・プロセッサのための物理レジスタ 2 段階解放

山本らは物理レジスタをリネーム・ステージと命令ウインドウとの 2 段階で解放することによって、物理レジスタの不足によるリネーム・ステージでのストールを解消することを提案している [7]。さらに、命令ウインドウで 2 段階目の解放を待ち受けている命令が実行可能であるならば先行実行を行い、そのことによってロード・データのキャッシュへのプリフェッチを実現している。先行実行された命令の結果をその命令の結果に依存する命令にフォワーディングすることによって、先行実行は連鎖して行うことができるため、先行実行された命令の中にロード命令が存在する場合には、そのロード命令によるプリフェッチが行われるのである。山本らは SPECfp2000 ベンチマークを用いて測量を行い、この手法によって物理レジスタ数が 64 個の場合に平均で 32 %もの大きな性能向上を達成できることを示している。この手法は先行実行を行い、ロード命令の一部を事前に行うことで後々の本実行となるロード命令のレイテンシを短くするという点でフロントエンド実行やツインテール・アーキテクチャと類似しているため、関連研究として本章で取り上げた。

## 4.3 値予測

値予測はデータ依存を解消する手法として盛んに研究が行われている。しかし、値予測を行うためのハードウェア・コストは非常に大きく、現在では予測能力を若干下げる代わりにハードウェア・コストを小さくする研究などが行われている [6]。値予測は命令の実行結果を予測してその命令に依存する命令の命令ウィンドウでの待ち合わせサイクル数を短縮することで性能向上を図っている。しかし、値予測は実行結果を予測した命令も確認のために実行を行わなければならないため、バックエンドで実行しなければならない命令の総数は減るわけではない。また、値予測は投機であるため、予測に失敗した場合には当然ミス・ペナルティが生じることになる。

フロントエンド実行やツインテール・アーキテクチャは従来の実行ステージ以外で処理した命令は従来の実行ステージで再度実行する必要がない。また、投機ではないため予測ミス・ペナルティも当然存在しないという利点がある。

# 第5章　まとめと今後の課題

本稿では我々が提案しているフロントエンド実行という手法についてその説明を行い、さらにこれを改善する手法としてツインテール・アーキテクチャと呼ぶ手法を提案した。これはフロントエンド実行ステージによって全体のパイプライン段数が増加してしまうという点を改善することを目的とした手法である。

さらにツインテール・アーキテクチャを効率的に動作させるためには従来の命令ウィンドウで行われる wakeup 論理よりも複雑な wakeup 論理が必要となることを示した。しかし、この wakeup 論理の具体的な実現方法については述べることができず、シミュレータによる評価を行うこともできなかった。

したがって、今後の課題としてまずはツインテール・アーキテクチャの評価を行うことがあげられる。本稿では In-order tail の実行ステージを 3 段としているが、評価を行うことによってこの段数を変更することなども考えられる。

また、第二の課題として具体的な wakeup 論理の実現方法を示すことがあげられる。IPC を低下させないためには wakeup 論理には select 論理と合わせて 1 サイクル内に収めなければならないという制約がある [3]。したがって、複雑な wakeup 論理を実現する際にはこのことに十分な注意を払う必要がある。具体的にはより高速な wakeup を可能とする手法を基にした wakeup を行うことなどが考えられる [4]。

# 第6章　おわりに

本提案手法はフロントエンド実行がスケジューリング処理を必要としない命令に対してスケジューリングを省略して実行を行ってしまっているということに着目した手法でもある。これは、レジスタ読み出しを終えた段階で即実行が可能な命令の実行を後続命令のスケジューリング処理と並行して行うことで性能向上を図る手法である。この手法は命令の実行にかかるレイテンシを隠蔽するという意味で先行研究である「フロントエンド実行によるプリロードの提案」[2] と同様な考え方に基づいた手法である。

現存の Out-of-order スーパスカラ・プロセッサにおいてスケジューリングとは他の命令へのデータ依存やバックエンドでの演算器の不足によって、実行を即座に開始することができない命令に対して、これを正しいタイミングで命令ウインドウから発行できるようにするための処理である。したがって、これは命令を実行することの本質ではないと考えることができる。実際、演算器が不足していない状態で他の命令へのデータ依存がない命令に対してスケジューリング処理を行うことは無駄であると言える。

提案手法の In-order tail では命令に対するスケジューリング処理を省略して即実行を行い、その一方で Out-of-order tail では従来通りのスケジューリング処理を行った後に実行を行う仕組みとなっている。この仕組みによって従来のプロセッサにおいて結果的にスケジューリングを必要としなかった命令は In-order tail で実行されることになる。一方で結果的にスケジューリングが必要となる命令については Out-of-order tail で処理されることになるのである。本提案手法はこの仕組みによって従来のプロセッサに比べて命令をより最適なタイミングで実行することができるといえる。

# 参考文献


[1] 小西将人、五島正裕、中島康彦、森眞一郎、富田眞治：フロントエンド実行．情報処理学会研究報告，2004-ARC-158, pp13-17, March

[2] 小西将人、福田匡則、五島正裕、中島康彦、森眞一郎、富田眞治：フロントエンド実行によるプリロードの提案．情報処理学会研究報告，2004-ARC-159, pp31-36, July

[3] 五島正裕：Out-of-order ILP プロセッサにおける命令スケジューリングの高速化の研究. 京都大学、Mar(2004)

[4] 五島正裕：スーパースケーラのための高速な動的命令スケジューリング方式．2001-ARC-142(HOKKE2001), pp121-126

[5] Vlad Petric, Tingting Sha, Amir Roth：RENO: A Rename-Based Instruction Optimizer. Department of Computer and Information Science, University of Pennsylvania

[6] 佐藤寿倫、有田五次郎：頻繁な値の局所性を考慮したデータ値予測機構のハードウエア量削減．情報処理学会研究報告, 2002-ARC-146-12

[7] 山本哲弘、安藤秀樹、島田俊夫：スーパスカラ・プロセッサのための物理レジスタ 2 段階解放. 情報処理学会研究報告, 2005-ARC-164, pp7-12

# 謝辞

本研究を進めるに当たって様々な方にお世話になりました。

坂井修一教授にはご多忙であるにも関わらず、卒論相談会等を通じて研究内容に対する御指導を頂きました。

五島正裕助教授には、初めの研究テーマの内容から論文タイトルの考案、論文の添削まで、本研究におけるほぼ全ての面において相談に乗って頂き、御指導頂きました。

入江英嗣氏には、研究内容に対する助言や論文タイトルの考案、論文の添削などで御指導頂きました。

清水修助手、事務補佐員の八木原晴水さん、月村美和さん、北原杏さんには研究を行う上での事務などでお世話になりました。ありがとうございます。

また、坂井・五島研究室の先輩にあたるルォン ディン フォンさん、豊島隆志さん、初田直也さん、清水一人さん、栗田弘之さん他の皆様には研究や論文執筆、研究室での生活面などでご協力いただき、大変お世話になりました。本当にありがとうございました。