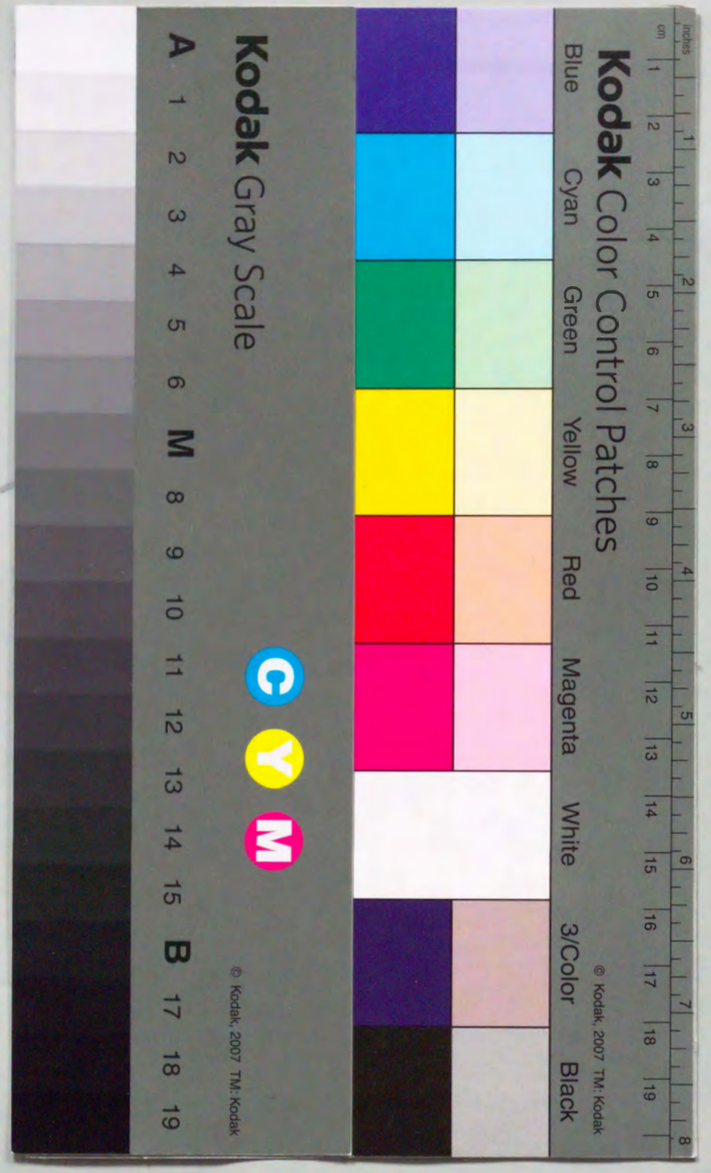
スーパースカラ・プロセッサの構成方式に関する研究

久我, 守弘
九州大学総合理工学研究科情報システム学専攻

https://doi.org/10.11501/3060381

出版情報：九州大学, 1991, 博士（工学）, 課程博士
バージョン：
権利関係：



スーパースカラ・プロセッサの構成方式に関する研究

平成3年12月

久我 守弘

スーパースカラ・プロセッサの
構成方式に関する研究

平成3年12月

久我 守弘

目 次

図目次

表目次

第 1 章

序論

本論文は，単一プロセッサの速度向上を図るプロセッサ・アーキテクチャとしてスーパースカラ・アーキテクチャに着目し，その構成方式について行った研究に関してまとめたものである．

1.1 研究の背景

世界最初のコンピュータが登場して以来，この 40 数年の間にコンピュータの処理能力は素子技術の発展と構成方式の研究により著しく進歩してきた．今日，パーソナル・コンピュータからスーパ・コンピュータにいたるまで幅広い分野であらゆる目的に使用されている．しかしながら，コンピュータを使用している全ての分野において，現存のコンピュータがユーザに対して十分な処理能力を提供しているとは言えず，より高速な処理能力を持つコンピュータを望む声が大きい．そのため，コンピュータの高速化を目指した素子技術および構成方式について，多くの研究が行われている．

素子技術の発展は現在もその途上であり，シリコン素子の高速化および高集積化は現在の高速コンピュータ開発の根本をなしている．一方，コンピュータの構成方式の改善に注目すると，プログラムに内在する並列性を引き出し，並列処理を行うことで高速化を狙う．並列処理を行う場合，並列処理をプログラム階層構造のどのレベルに適用するかで，少なくとも次の 3 レベルが存在する．

(a) タスク/スレッド・レベル：本来独立なプログラムやサブルーチン単位などでの並列性を利用するレベル．

(b) ステートメント/ループ・レベル：高級言語におけるステートメントやループ構造

に内在する並列性を利用するレベル.

(c) 命令レベル : プロセッサの機械語命令間に内在する並列性を利用するレベル.

今日，上記 (a) タスク/スレッド・レベルの並列処理を行うマルチプロセッサの研究開発が盛んであり，(c) 命令レベル並列処理の有用性を疑問視する向きがある．確かに, 命令レベルの実用的な並列度 (厳密には空間並列度) は 2〜10 程度と大きくはない．100〜1000 台規模のマルチプロセッサに比べると，2〜3 桁低い．

しかし，いま，命令レベルの並列処理 (特に空間並列処理) を積極的に利用してはいないプロセッサを要素プロセッサとして使用した 1000 台のプロセッサで構成されるマルチプロセッサが存在したとして，その性能を 2 倍に引き上げる方法を考えてみる．

- あと，1000 台の同一プロセッサを追加する．しかし，プロセッサ数を 2 倍の 2000 台にしてもシステム性能は単純に比例して 2 倍にならないことは，よく知られている [**HwangBriggs87**]．したがって，さらに数十台か数百台の追加は必要であろう．
- 命令レベルの空間並列度を積極的に利用し 2 倍の性能を保証するプロセッサ 1000 台で, 元のプロセッサ 1000 台をすべて置き換える．この場合，システム性能は確実に 2 倍になる.

したがって，命令レベル並列性を利用しプロセッサの性能向上を図ることは，たとえその並列度が小さくても，軽視してはならない.

1.1.1 スーパースカラ方式の位置付け

さて，命令レベル並列性には，時間並列性 *(temporal parallelism)* と空間並列性 *(spatial parallelism)* の 2 つがある [**HwangBriggs87**]．従来提案されたプロセッサ・アーキテクチャは，これらの一方または双方の並列性を利用している．

(a) 時間並列性のみを利用

パイプライン方式は時間並列性を利用したアーキテクチャである．機能的に分割して行える作業を時間的に多重化して並列実行する (Figure 1.1 参照)．Figure 1.1(a) は基本的なパイプライン方式である．また，この方式を発展させた方式として，さらに細かく機能分割を行い動作周波数をあげることで高速処理を狙う**スーパーパイプライン** *(superpipeline)* **方式 [Jouppi89a] (Figure 1.1(b))** も提案されている.

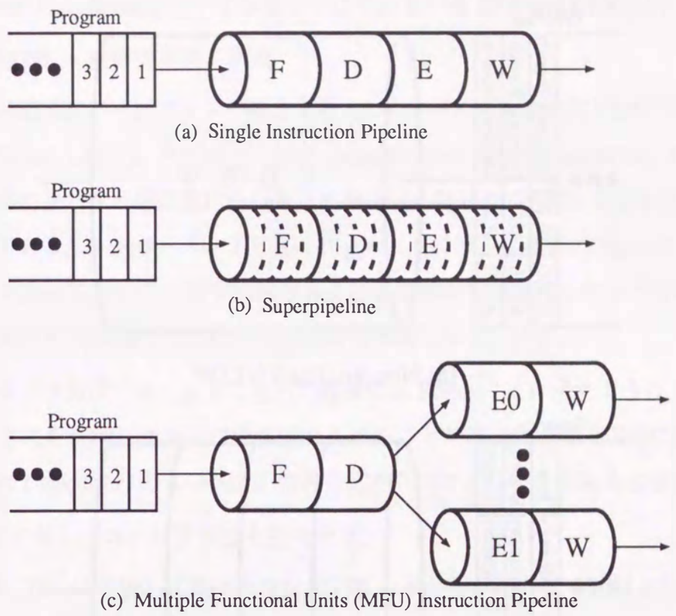
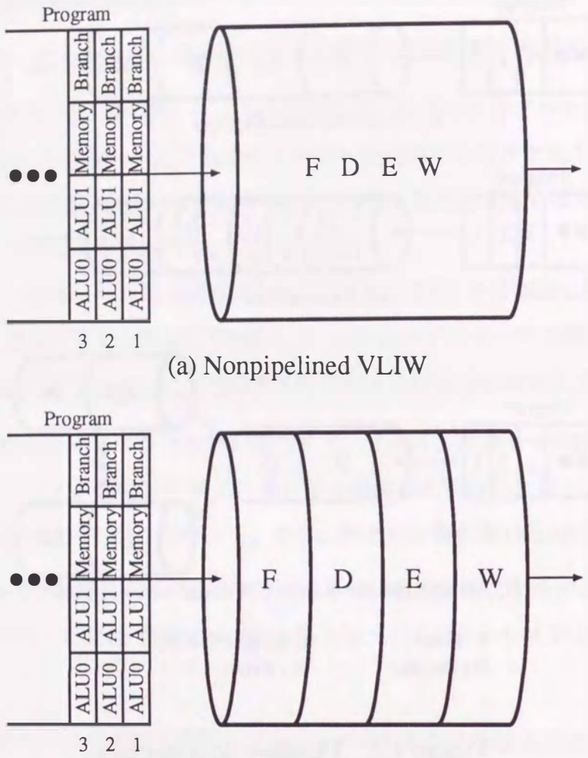


Figure 1.1 Pipeline Architecture.

(b) 空間並列性のみを利用

Very Long Instruction Word (VLIW) 方式 [Fisher83] は空間並列性を利用する．すなわち，並列に実行可能な命令を，並列動作可能な演算器 (機能ユニット) で並列に処理を行う．並列動作可能な機能ユニットが **n** 個あり，各機能ユニットを制御するのに 32 ビット長の命令フィールドが必要だとすると，VLIW では各機能ユニット対応に命令フィールドを設け，**n** フィールドから成る 32×**n** ビット長の超長形式機械命令 (VLIW) を用いて処理を行う．各機能ユニットを制御する命令 (オペレーションと呼ぶ) は，予め定められた命令フィールドに置かれ，**n** 個のオペレーションと **n** 個の機能ユニットとは静的に 1 対 1 に対応付けられる．並列に実行可能な命令の抽出はコンパイラによって行い命令を生成する．Figure 1.2(a) は，2 つ演算器，メモリアクセスおよび分岐制御の 4 つの機能ユニットを備えた VLIW の例である.



(a) Nonpipelined VLIW

(b) Pipelined VLIW

F : Fetch　　E : Execute　　ALU : Arithmetic Logic Unit
D : Decode　　W : Write

Figure 1.2　VLIW Architecture.

(c) 時間並列性 + 部分空間並列性を利用

パイプライン方式に一部空間並列性を利用することが可能である．通常，整数演算と浮動小数点数演算とを比較した場合，浮動小数点数演算の方が時間がかかる．整数および浮動小数点数の演算が可能であるが，同時に処理できない演算器を持つパイプライン・プロセッサでは，整数および浮動小数点数演算が混在するプログラムを実行する場合，パイプラインに乱れが生じ処理の妨げになる．そこで，Figure 1.1(c) のように複数の機能ユニットを持たせ，命令を同時に実行できるようにすることでパイプラインの乱れを抑えるプロセッサが考えられている．これを**複数機能ユニット** *(MFU*

: Multiple Functional Unit) プロセッサと呼んでいる [**HwangBriggs87**].

(d) 時間並列性 + 空間並列性を利用

VLIW 方式にパイプラインの概念を導入した，パイプライン化VLIW が考えられている (Figure 1.2(b))．**スーパースカラ** *(superscalar)* **方式** [**Jouppi89a, Johnson90**] も，時間的並列性と空間並列性の両方を積極的に利用して速度の向上を図るアーキテクチャである．スーパースカラ方式は Figure 1.3 に示すように，Figure 1.1(a) のパイプライン方式において，命令供給系および演算器系を多重化したような構成を採り，同時に実行可能な命令を並列に処理可能な構成となっている．

スーパースカラ方式が誕生するに至った経緯には Figure 1.4 に示すように，VLIW，パイプライン方式を洗練しクロック周波数をあげることで高速化を図る RISC*(Reduced Instruction Set Computer)* および MFU の各方式からのアプローチがあると考えられる．

(a) VLIW に対してコード互換性を持たせる．

VLIW 方式は超長形式機械命令を用いることから，現在良く利用されているパイプライン・プロセッサの命令セット・アーキテクチャとは異なる命令体系を持つ．このことは計算機システムのリプレースの際に多大な影響を与える．スーパースカラ方式では，命令セット・アーキテクチャ・レベルでの互換性を保つことができ，さらにオブジェクト・コード・レベルでの互換性も維持することが可能である特長がある．

(b) パイプライン方式 (特に RISC) に対して，多重命令発行能力を持たせる．

パイプライン方式では，基本的に命令をひとつずつ処理していくことから，1 サイクルあたりに処理できる命令数は 1 を越えることがない．空間並列性を利用するために，

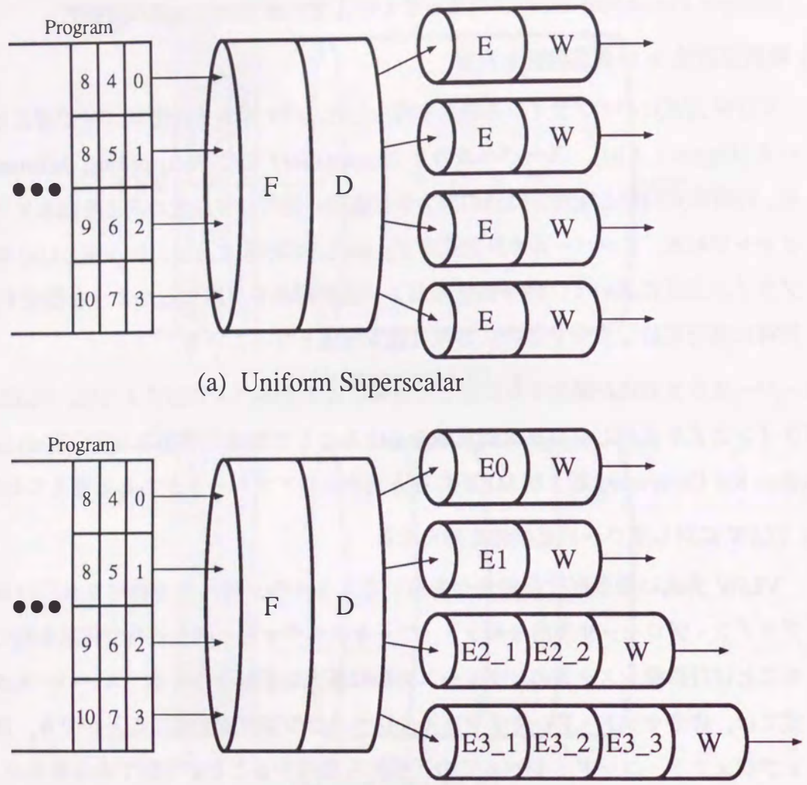
(i) 命令パイプライン自体を多重化する (Figure 1.3(a)).

(ii) 命令発行多重度に応じて，必要なハードウェア機構のみを多重化する (Figure 1.3(b)).

といった，ハードウェア機構を導入することで処理能力の向上を狙う．

(c) MFU において，機能ユニットの多重度に見合うよう，命令発行機構などのハードウェア機構を多重化する．

このように，スーパースカラ方式は従来考案されたアーキテクチャの欠点を補い，かつ，性能を得るために最終的にたどり着く方式と見なすことができる極めて重要なアーキ



(a) Uniform Superscalar

(b) Nonuniform Superscalar

F : Fetch　　E, E0, E1 : Execute　　E_1, E_2, E_3 : pipelined arithmetic logic unit
D : Decode　　W : Write

Figure 1.3　Superscalar Processor.

テクチャであるといえる．

1.1.2　スーパースカラ実現上の課題

スーパースカラ方式を実現する場合，命令の供給および実行を多重化することにより，単一命令パイプラインの時に比べて以下に挙げる新たな技術的課題が生じる．これらの課題について，何らかの対処を施さなければならない．

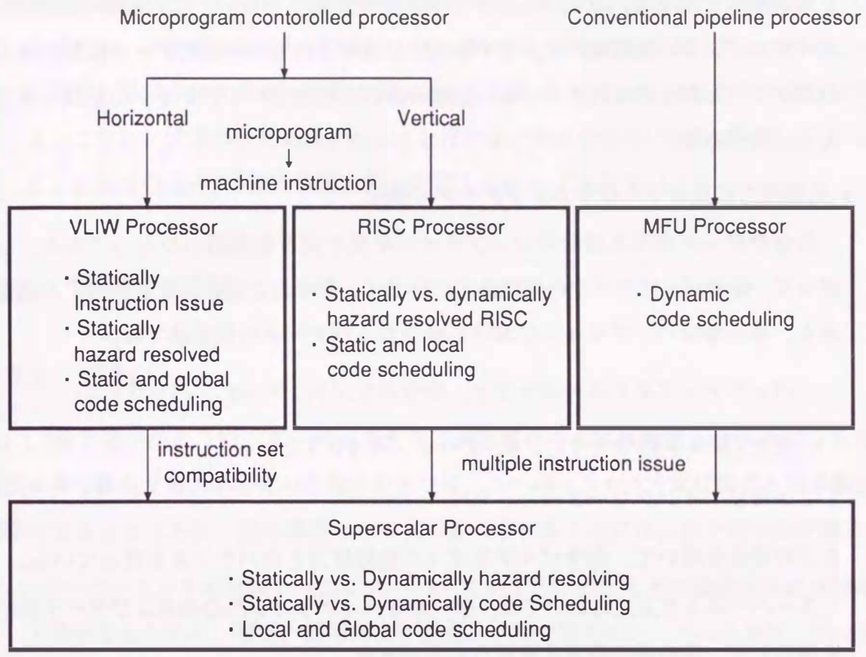


Figure 1.4　The Situation of the Superscalar Processor.

- 命令間のデータ依存関係 *(data dependency)* に起因するハザードの回避：データ依存関係には，入力依存 *(RAR : Read-After-Read)*，フロー依存 *(RAW : Read-After-Write)*，逆依存 *(WAR : Write-After-Read)*，および，出力依存 *(WAW : Write-After-Write)* の 4 種類がある．命令レベルの並列処理を行う場合，入力依存関係を除く 3 種類の依存関係にある命令の実行順序を逐次的 (in-order [1]) に制限する．つまり，依存関係を考慮しない場合，得られる結果は保証されない[2]．これらの依存関係は，命令間の実行を in-order に制限し，プログラムの実行を並列に実行できる機会を妨げるため，何らかの対策を施す必要がある．
- 分岐命令に起因する制御依存関係 *(control dependency)* への対処：分岐命令が存在する場合，分岐するか否か (Taken/Not-Taken：条件分岐の場合)，および分岐先アド

1 逐次的という意味．本来決められている順番を守ること．

2 正しい結果が得られない状況をハザード *(hazard)* という．

レスが確定するまで，次にフェッチすべき命令が決まらない．この制御依存関係は，パイプライン中に無効なサイクルを多数生じさせるため，一般にデータ依存関係よりパイプラインの乱れを大きくする．分岐命令による命令パイプラインの乱れを極力押える必要がある．

- 処理能力に見合った命令およびデータの供給

  命令やデータの不在は命令パイプラインを乱す大きな原因のひとつである．したがって，命令パイプラインの処理能力に応じて，命令およびデータを供給する必要がある．単一命令パイプラインにおける命令およびデータの供給系では，

  – バッファ：アドレス・バッファ，命令バッファ，データ・バッファ

  – キャッシュ：命令キャッシュ，データ・キャッシュ

  – メモリ・インタリーブ

  などの手法を用いて，命令パイプラインの処理能力とのバランスを取っている．

  スーパースカラ方式においては，単一命令パイプラインの命令およびデータ供給系を基にして，供給系の多重化を行う必要がある．

- ISP(Instruction Set Processor：命令セット・プロセッサ) アーキテクチャ

  スーパースカラ方式では，スーパースカラ度[3]を **n** とするとき，**n** 個の命令を同時にフェッチ *(fetch)*，デコード *(decode)* を行うことから，ある程度，命令の構造 (命令長やデコードの容易さなど) に制限が課せられる．また浮動小数点数演算のように，結果を得るまでに数サイクル必要とする演算が存在する場合，命令パイプラインを乱す原因となる．したがって，これらの命令を定義する場合には，パイプラインの乱れによる性能低下を押える方法を考慮する必要がある．

- 正確な割込み/分岐の保証

  スーパースカラに限らず，命令レベルの並列処理を行うプロセッサでは，内部割込みや分岐命令により，不正確なマシン状態 *(imprecise machine state)* が発生する可能性があり問題となる．この問題は，内部割込みの原因となった命令の実行時点，ないし，分岐命令の実行時点のプロセッサ状態（レジスタやメモリの内容）が，それ以降の命令の実行により既に変更されていて，正しい状態が保証されないため生じる．

3 命令供給系の多重度 (1 サイクルで何命令ずつフェッチするか) のことをスーパースカラ度と定義する．

MFU プロセッサのように演算命令間の追い越し実行 (out-of-order [4]実行という) がある場合は，不正確な割込みが問題となる．さらに，分岐命令を越えて out-of-order 実行制御を行おうとすると，問題は分岐命令にも及ぶ．これは，分岐命令の分岐方向によってマシン状態の復元処理 *(repair)* を行わなければならないからである．このように，正確な割込み/分岐の保証方法を考慮しなければならない．

1.2 研究の概要

1.2.1 目的

1.1.2項で述べたように，スーパースカラ方式実現のためには考慮しなければならない事項が多く存在する．本研究の目的のひとつは，スーパースカラ方式に関する基本事項を明確にすることにある．基本事項として明らかにすべきことには，以下の 2 つがある．

(a) スーパースカラの分類：一口にスーパースカラといってもその構成方法には種々の方法が考えられる．事実，商用および研究レベルで種々のスーパースカラ・プロセッサが製造・提案されている．これらのスーパースカラ・プロセッサを，最も基本的となる特長・選択肢により区分できるような分類方法を考察する必要がある．

(b) 問題点への対処方法と実現上の選択肢：上記 1.1.2項で述べた実現上の課題に対して，従来から種々の対処法が考案されている．それらは，スーパースカラ・プロセッサの性能およびハードウェア・コストに種々の影響を与えるため，プロセッサを設計する上で重要である．これら選択肢を，それぞれ “実現上の問題” 別に整理すると共に，スーパースカラ・プロセッサの処理能力を最大に引き出す新しい選択肢を考案することは重要な課題である．

さらに，スーパースカラ方式を採用したプロセッサが，現状の半導体技術で実現可能であり，性能およびハードウェア・コストが適度なプロセッサ構成をとる必要がある．そこで以下のように，実際にプロセッサの開発および評価を行い，スーパースカラ・プロセッサの設計が妥当であることを検証する．

(c) 試作プロセッサの設計

4 非逐次的という意味．本来決められている順番を守らないこと．

プロセッサの開発方針を明らかにするとともに，具体的な設計方針を明確にする．そして，プロセッサ実現上の課題に対していかなる対処を施すか，また性能向上を狙うために導入するハードウェア機構などを考慮して設計を行い，ハードウェア的に実現可能か否か実際に設計を行う．

(d) ソフトウェア・シミュレータによる評価

ハードウェア設計と並行して，設計したプロセッサの動作をシミュレートするソフトウェア・シミュレータを作成し，実際に期待する性能を得ることができるか評価を行う．

スーパースカラ・プロセッサの設計では，2種類の異なる特長を持つプロセッサを設計し，それぞれ性能評価を行う．

1.2.2 論文の構成

本論文は9章から成る．第1章では，スーパースカラ方式が生まれるに至った背景について述べ，スーパースカラ方式と従来研究されてきたプロセッサ・アーキテクチャとの関係を明確にした．そして，本研究の目的および概要について述べた．

続く第2〜4章は，スーパースカラ方式に関する基本事項をまとめたものである．まず第2章では，スーパースカラ方式の分類を行う．ここでは，スーパースカラ方式を実現する上で，その設計思想の違いによって3種類に分類を行う．第3章ではスーパースカラ方式の性能を最大限に引き出すために必要不可欠な，ハードウェアおよびソフトウェアによるコード・スケジューリング技術について述べる．第4章では，スーパースカラ方式を実現する際に決定しなければならない種々の選択肢についてまとめる．

第5〜8章は，スーパースカラ・プロセッサの実現例に関する研究成果について述べる．ここでは，3つに分類したスーパースカラ方式のうち，2種類について具体的な設計および評価を行う．第5, 6章は3種類の方式の1つであるDD型について，また，第7, 8章はもうひとつの方式であるDS型スーパースカラ方式の具体例について述べる．

それぞれのスーパースカラ方式について，その開発・設計方針，実現上の課題と対処法および試作プロセッサの特長を第5, 7章で述べる．さらに，各プロセッサの構成，ソフトウェア・シミュレーションによる性能評価およびその結果を各々第6, 8章で述べる．

最後に第9章において，本研究の成果をまとめると共に，今後の課題を明らかにする．

第 2 章

スーパースカラ・アーキテクチャ

本章では，まずスーパースカラ方式とVLIW方式との違いを明らかにすることでスーパースカラ方式の定義を明確にする．さらに，スーパースカラ方式を構成上基本となる3つの種類に分類する．

2.1 スーパースカラ方式の定義

1.1.1項で述べたように，スーパースカラ方式は，従来から提案されている単一プロセッサ・アーキテクチャである，VLIW，RISCおよびMFUからの延長線上にあると考えられる．ところで，スーパースカラ方式およびVLIW方式は，共に時間および空間並列性を利用したアーキテクチャであり，命令(VLIWではオペレーション)の供給系および機能ユニットが多重化されたハードウェアを備え，複数の命令を同時に処理することが可能である．両方式において命令の供給系および機能ユニットが同等に多重化されている場合，命令の実行においてパイプラインの乱れが発生しなければ，両方式が持つ潜在的な性能は等価であると考えられる．そこで性能以外に関して，スーパースカラ方式とVLIW方式を明確に区別すると共に，スーパースカラ方式の定義を明らかにする必要がある．

スーパースカラ方式とVLIW方式との違いには，少なくとも以下の5つがある[村上90]．

(a) ハザードの解消時点：命令間の依存関係に起因するハザードをは，VLIWでは静的に，すなわちコンパイルの段階で回避する必要がある．これは，命令の各オペレーションが1対1に機能ユニットを制御し，オペレーション間の相互干渉がないからである．一方，スーパースカラでは，ハーザードの回避を静的あるいは動的のいずれで行ってもよい．つまり，2.2節で述べるように，静的ハザード解消スーパースカ

ラ *(statically hazard resolved superscalar)* および動的ハザード解消スーパースカラ *(dynamically hazard resolved superscalar)* の 2 つの選択肢が存在する．VLIW および静的ハザード解消スーパースカラにおいては，資源競合および命令間依存関係が実行時に生じないよう，コンパイラにより静的にオペレーションないし命令の並べ替えを行う．動的ハザード解消スーパースカラにおいては，資源競合および命令間依存関係が実行時に生じ得るので，これらをハードウェアで検出し解消 (機能ユニットなどの資源競合および命令間依存関係の解消を待ってから命令を発行) する必要がある．

(b) 命令の機能ユニットへの割り当て時点：VLIW ではコンパイル時に並列に実行可能な命令の抽出を行い超長形式機械命令の各フィールドに静的に割り付ける．各々のフィールドがそれに対応する機能ユニットを制御することから，命令の機能ユニットへの割り当ては静的に行っているといえる．一方，スーパースカラ度 **n** のスーパースカラは同時に **n** 命令をフェッチするが，各命令の **n** 命令内相対位置のみでは当該命令の発行対象機能ユニットが一意に定まらない．つまり，デコード後決まるもので，機能ユニットに対して命令の割り当てを動的に行う．

(c) コード最適化時点：VLIW ではコンパイル時に並列に実行可能な命令の抽出を行うことから静的な最適化のみを行う，したがって，最適化コンパイラの性能によってプロセッサの性能が左右される．一方，スーパースカラでは静的な最適化に加えて，命令の実行段階で動的なコード最適化も行うことが可能である．

(d) オブジェクト・コード・サイズ：一般に，VLIW の方がスーパースカラよりコード・サイズが大きくなる．これは，前述の (a) ハザードの解消時点に起因するもので，命令内の各命令フィールドは，対応する機能ユニットの busy/idle に関わらず，常にオブジェクト・コード中に存在する必要がある．つまり，たとえ機能ユニットが idle であっても，対応する命令フィールドには必ず NOP*(No OPeration)* が出現しなければならない．スーパースカラにおいては，静的ハザード解消スーパースカラでは VLIW と同様にこのような NOP は必要であるが，動的ハザード解消スーパースカラでは不要である．VLIW において，この問題を解決するために，動的に NOP を生成する VLIW プロセッサもある [**Colwell88**]．

(e) 互換性：これには，命令セットの互換性とオブジェクト・コードの互換性の 2 面がある．まず，VLIW の命令フォーマットは 1.1.1項で述べた通り，機能ユニット数を直接反映しているため機能ユニット数が異なる VLIW プロセッサ間では命令セット自

身の互換性がない．よって，オブジェクト・コードの互換性を保つのは難しい．一方，スーパースカラでは，命令セットには互換性がある．しかし，オブジェクト・コードの互換性については，静的および動的ハザード解消スーパースカラでは事情が異なる．つまり，動的ハザード解消スーパースカラではオブジェクト・コードの互換性があるが，静的ハザード解消スーパースカラは VLIW 同様オブジェクト・コードの互換性を保つのは難しい．なぜなら，スーパースカラ度 **n** の静的ハザード解消スーパースカラ用にスケジュールされたオブジェクト・コードは，スーパースカラ度 **n'**(≠**n**) の静的ハザード解消スーパースカラ上では正常に動作しないからである．

以上から，スーパースカラ方式の VLIW 方式に対する存在意義は，

(a) オブジェクト・コード・サイズが小さい．

(b) 命令セットおよびオブジェクト・コードの互換性がある．

(c) 動的なコード最適化を利用可能．

の 3 点となる．ただし，上記 (a)～(c) のすべてを満たすのは，2.2節で後述するように，動的ハザード解消および動的コード最適化を行うスーパスカラのみである．

上記スーパースカラ方式の VLIW 方式に対する存在意義および 1.1.1項で述べたスーパースカラ方式誕生の経緯を踏まえて，スーパースカラ・プロセッサの定義を明確にすると，以下のようになる．

- 同時に複数の命令の供給を行う．(単一パイプライン，MFU との差異)
- 複数の機能ユニットを備える．
- 命令の機能ユニットへの割当が，命令実行時に行われる．(VLIW との差異)
- 同時に複数の命令を機能ユニットに対してディスパッチ可能である．
- 従来のパイプライン・プロセッサにおける命令セットと互換性を保つことが可能である．

2.2 スーパースカラ方式の分類

本節では，以下にあげる基本選択肢によってスーパースカラ・プロセッサを 3 つの範疇に分類する [**村上 90**]．

2.2.1 基本選択肢

スーパースカラ・プロセッサには，様々な構成のものが存在するが，以下の 3 つの基本選択肢を用いてスーパースカラ・プロセッサを分類する．

(1) ハザードの解消時点： 静的 vs. 動的ハザード解消 [SmithM90]

命令間の依存関係 (データ依存，制御依存) の存在やプロセッサの資源 (レジスタのポート，機能ユニットなど) への競合の発生が，命令の処理に影響を与える．つまり，命令間依存関係の遵守および資源競合の回避を行わなければ，ハザードの発生により正しい結果を得ることが出来ない．したがって，このようなハザードを回避するために，静的 (コンパイル時) あるいは動的 (実行時) に命令間依存関係および資源競合を検出・解消しなければならない．

(a) 静的ハザード解消スーパースカラ *(statically hazard resolved superscalar)*

基本的には，コンパイラにより資源競合および命令間依存関係を完全に解消する．よって，実行時にハードウェアでこれらを検出・解消する必要がないので，プロセッサの簡素化・高速化が図れる．ただし，例外として，制御依存およびメモリ参照に関するデータ依存は静的に解消しないで，実行時に解消させる技法もある．

(b) 動的ハザード解消スーパースカラ *(dynamically hazard resolved superscalar)*

資源競合および命令間依存関係が実行時に生じる可能性がある．よって，これらを検出し，パイプライン・インターロックなどの手法でハザードを解消するハードウェア機構が必要となる．しかし，スーパースカラ度の異なるプロセッサに対しても，オブジェクト・コードの互換性を保つことができる．

(2) 命令最適化時点： 静的 vs. 動的コード・スケジューリング

資源の競合や命令間の依存関係が存在する場合，同時に実行可能な命令数が減少するため命令レベル並列性が低下する．通常，最適化されていないオブジェクト・コードの命令レベル並列性は 2〜3 程度であり，そのまま実行してもスーパースカラ方式による性能向上は達成できない．したがって，プログラム内の命令間依存関係などを検出して並列性を最大限に発揮するように命令順序の並べ替え，すなわち，コード・スケジューリング

(code scheduling) を行う必要がある．このコード・スケジューリングに関しても静的に行うか動的に行うかの選択肢がある．

(a) 静的コード・スケジューリング *(static code scheduling)*

コンパイラが並列性の抽出およびプロセッサの構成を考慮した命令コードの最適化を行う．したがって，ハードウェアの制御を容易にすることが可能である．また，最適化を行う範囲をプログラムの全体を対象として行うことができる利点がある．

(b) 動的コード・スケジューリング *(dynamic code scheduling)*

実行時にハードウェアが，主にデータ依存関係を解析してハザードが生じない範囲で命令の実行順序の並べ替えを行う．つまり，オブジェクト・コード上では後の命令でも前の命令と同時あるいは先に実行可能である場合には，命令の実行順序を入れ替える out-of-order 実行を行う．

動的コード・スケジューリングは，静的コード・スケジューリングでは対処できないメモリ・アクセスによるデータ依存などの実行時でなければ判明しない依存関係についても対処可能である．しかしながら当然のこととして，動的コード・スケジューリングを行うための複雑なハードウェア機構が必要であり，それに従って制御も複雑化する恐れがある．また，プロセッサ内部に取り込まれた命令のみが動的コード・スケジューリングされる対象となるため，静的コード・スケジューリングのようにプログラムの広い範囲をスケジュールすることはできない．

以上述べたように，静的および動的コード・スケジューリングには，各々一長一短があり互いに排他し合うものではない．つまり，動的コード・スケジューリングを行っているスーパースカラにおいても静的コード・スケジューリングによる性能向上は期待できる．

(3) 機能ユニットの均質性： 均質型 vs. 非均質型機能ユニット

スーパースカラ・プロセッサは複数命令の同時実行を行うため，命令フェッチ/デコード・ユニット，レジスタ・ポート，バス，命令の実行を行う機能ユニットなどが複数個存在する．スーパースカラ度を n とした場合に，これらのハードウェアの均質性に関して，以下の 2 方式が存在する [久我 90, Jouppi89b]．

(a) 均質型 (uniform) 機能ユニット

命令フェッチ/デコード・ユニット，レジスタ・ポート，バスおよび機能ユニットがスーパースカラ度に対応して n 多重化されている．命令発行の対象である各機能ユニットは，多機能ユニット，もしくは，単機能ユニットの集合体で，全ての種類の命令を実行することができる汎用機能ユニットである．発行可能な命令各々に対して汎用機能ユニットが用意してあるため，資源競合は生じ得ないが，使用頻度の低い機能ユニットまで多重化されているので，ハードウェア・コストは高くなる．Figure 1.3(a) はスーパースカラ度 4 の均質型スーパースカラ・プロセッサの例である．

(b) 非均質型 (nonuniform) 機能ユニット

命令フェッチ/デコード・ユニット，レジスタ・ポート，バスは各々n 多重化されているが，機能ユニットについては，必要なものだけを多重化している．したがって，各々の機能ユニットの機能が均質でなく，命令の種類により発行先の機能ユニットが決まる．Figure 1.3(b) は 4 つの異なる機能ユニットを持つ例である．これらは同時に動作可能であるが，同一の機能ユニットを使用する命令が複数存在する場合，資源競合が発生し得る．

2.2.2 スーパースカラ方式の分類

前項で挙げた基本選択肢の中で，静的 (S)vs. 動的 (D) ハザード解消と静的 (S)vs. 動的 (D) コード・スケジューリングの組合せについては次のことがいえる．

動的ハザード解消スーパースカラは，静的あるいは動的コード・スケジューリングのどちらも行うことができる．これに対して，静的ハザード解消スーパースカラではコンパイラでハザードを解消するため，必然的に静的コード・スケジューリングを行うことになる．したがって，スーパースカラ・アーキテクチャは基本的に以下の 3 つに分類できる．

- **SS 型**：静的ハザード解消スーパースカラ + 静的コード・スケジューリング
- **DS 型**：動的ハザード解消スーパースカラ + 静的コード・スケジューリング
- **DD 型**：動的ハザード解消スーパースカラ + 動的コード・スケジューリング (+ 静的コード・スケジューリング)

なお DD 型については，2.2.1項で述べたように，静的コード・スケジューリングは排他される技術ではなく，動的コード・スケジューリングの欠点を補うものである点に注意されたい．

Table 2.1 The category of a superscalar processor.

When Harzards Resolved?	Static	Dynamic	
When Code Scheduled?	Static		Dynamic
Category	SS	DS	DD
Object Code Portability	× (need to re-compile)	○ (not optimum)	○
NOP Instruction in Object Code	Necessary	Unnecessary (need to optimize)	Unnecessary
Execution Order	In-order	In-order (Out-of-order†)	Out-of-order
Completion Order	Out-of-order	Out-of-order	Out-of-order
Examples of a hardware for resolving hazards	Unnecessary	Interlock	Scoreboard Register Renaming Reservation Station
Hardware Scale (complexity)	Small	Middle	Large
Cooperation between Compiler and Hardware	Burden on Compiler	Moderately	Burden on Hardware
Machines	Stanford TORCH	Kyushu DSNS Intel 80960CA	IBM RS/6000 Stanford MATCH Kyushu DDUS

†Instructions in an instruction block can be executed out-of-order if no dependency exists.

SS, DS および DD 型スーパースカラの各々の特長をまとめると，Table 2.1 のようになる．

- SS 型：動的コード・スケジューリングを行わないためハードウェアが簡単である．しかし，命令レベル並列性を確保できない場合，ハザード解消のための NOP が多数必要である．また，オブジェクト・コードの互換性を保つことができない．
- DS 型：動的ハザード解消を行うためオブジェクト・コードの互換性を保つことができ，最適化のために必要な NOP も SS 型ほど多数必要ではない．また，DD 型と比較してハードウェアが簡単である．
- DD 型：動的コード・スケジューリング機構のために複雑なハードウェアを必要とするが，静的および動的コード・スケジューリングの長所を最大限に利用できる．

これらの基本的な分類にスーパースカラ・プロセッサ構成上の均質型 (U) vs. 非均質型 (N) の選択肢を加えて，スーパースカラ・アーキテクチャは最終的に **SSU, SSN, DSU, DSN, DDU, DDN** の 6 つの型に分類できる．

Figure 2.1に上記のプロセッサ構成例を用いて，そのハードウェア複雑度に関するスペ

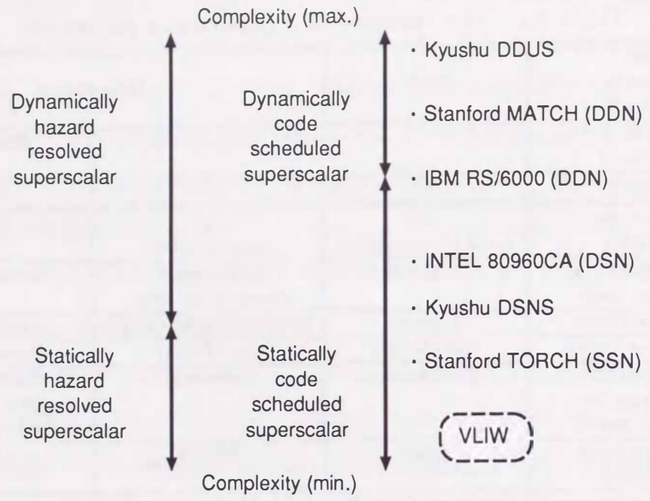


Figure 2.1　The Spectrum of the hardware complexity.

クトラムを示す．ここに挙げている 6 種のプロセッサは，研究および商用化された主なスーパースカラ・プロセッサであり，このうち我々が研究・開発を行ったプロセッサは DDUS(Dynamically-hazard-resolved, Dynamically-code-scheduled, Uniform Superscalar) および DSNS(Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscalar) プロセッサである．上記分類法に従うと，DDUS および DSNS は文字どおり各々DDU 型と DSN 型に分類できる．また，スタンフォード大学の TORCH は SSN 型，MATCH は DDN 型となる [**SmithM90**]．商用スーパースカラの Intel 80960CA[**Intel89**] は DSN 型，IBM RS/6000[**IBM90**] は DDN 型スーパースカラに分類できる．さらに，VLIW プロセッサは SSN 型スーパースカラにおいて命令の発行機構が単純化されたものととらえることが可能であり，SS 型スーパースカラの 1 形態とみなすことができる．

その他のスーパースカラ・プロセッサについては，付録 A にその分類を示す．

第 3 章

スーパースカラのためのコード最適化技術

本章では，スーパースカラ方式の性能を最大限に引き出すために必要な，コード最適化技術について述べる．コード最適化技術には，命令の実行段階で行う動的コード・スケジューリングとコンパイルの段階で行う静的コード・スケジューリングの 2 種類がある．

3.1　動的コード・スケジューリング

データ依存関係には，1.1.2項で述べたように，フロー依存，逆依存および出力依存の 3 種類がありハザード原因となる．したがって，これらの依存関係にある命令の実行順序を in-order に制限する．データ依存関係を命令の実行時に検出し，依存関係のない命令から実行を開始できるように制御することが動的コード・スケジューリングである．動的コード・スケジューリングには，データ依存によるハザードの回避法およびコード・スケジューリングの導入方法に関して，少なくとも以下の 3 方式がある [**Weiss84**]．

(a) インターロック *(interlock)* 制御：発行対象命令が先行命令に対していずれかのデータ依存関係にある場合，当該命令の発行をブロックする．また，ブロックされた命令の後続命令も同様にブロックする．つまり，データ依存関係に起因するハザードの回避のためには，最低限必要なハードウェア機構である．この場合，各依存関係の検出を行わなければならない．フロー依存および出力依存の検出については，レジスタに 1 ビットのスコアボード *(scoreboard)* を設け，レジスタに書き込みを行う命令が当該レジスタのスコアボードをセットすることで，後続命令が依存関係の検出をできるようにする方法が用いられる．一般にスコアボードという場合，上記制御方法のことを指し，後述する Thornton のスコアボードとは区別する．なお逆依存については，パ

イプライン・ステージの最終段のみで書き込みを行うプロセッサにおいては起こり得ない．なぜなら，後続の書き込み命令が発行するためには，先行命令が発行済みである必要があるが，発行済みであるということは既にオペランドの読み出しを完了していることを意味するからである．

この方式では，後続命令はインターロックされて先のステージに進めなくなるため，命令の実行開始順序は in-order となり，厳密には動的コード・スケジューリングを行っているとはいえない．なお，インターロック制御の拡張として，後続命令のうち全くデータ依存関係のない命令に関しては発行を許す方法が考えられる．この場合，命令の実行開始順序は out-of-order となり動的コード・スケジューリングを行っているといえる．

(b) Thornton のスコアボード・アルゴリズム [**Thornton64**] : 発行対象命令が先行命令に対して出力依存および逆依存のいずれかの依存関係にある場合，当該命令の発行をブロックする．このとき，後続命令もインターロックされて先のステージに進めなくなる．一方，発行対象命令が先行命令に対してフロー依存関係にある場合は，当該命令はフロー依存が解消されるまで待つことになる．この間，後続命令はインターロックされることなく，当該命令をバイパスしていく．よって，命令の実行開始順序は out-of-order となる． フロー依存および出力依存の検出は，上記 (a) インターロック制御と同じである．逆依存の検出に関しては，

- レジスタ読み出しに関するスコアボードを設けることで検出．
- デスティネーション・レジスタ番号と先行命令のソース・レジスタ番号を比較することで検出．

の，2 つの方法が可能である．

また，Thornton のスコアボード・アルゴリズムの拡張として，後続命令のうち逆依存関係を起こす命令の発行を許すことも可能である．その方法は，先行する命令がオペランドとして必要とするレジスタ内容のコピーを命令発行時の情報の一部として持たせることで実現できる (レジスタの名前替え．下記 (c)Tomasulo のアルゴリズム参照)．

(c) Tomasulo のアルゴリズム [**Tomasulo67**] : 発行対象命令が先行命令に対して，いずれのデータ依存関係にあっても，当該命令の発行をブロックする必要はない．まず，

フロー依存関係の検出はスコアボードによって可能である．命令を発行する際に，レジスタに対して書き込みを行う命令はスコアボードをセットするが，それと共に，どの命令が書き込みを行うかが識別できるようにタグ付けを行う．出力依存および逆依存に対しては，先行命令が記したタグを利用することで，データ依存関係に起因するハザードを回避できる．命令が持つレジスタ番号を用いる替わりにタグを用いてデータの享受を行うことから，この方法をレジスタの名前替え *(register renaming)* と呼ぶ．この方法により命令の実行開始順序は out-of-order となる．

Tomasulo のアルゴリズムでは register renaming を行う場合，直接レジスタを renaming するため，分岐命令を越えた out-of-order 実行に対処できない．Tomasulo のアルゴリズムを拡張し，間接的に register renaming を行うことで制御依存関係にも対処可能なアルゴリズムも考案されている [**Hwu87**, **Sohi87**, **久我 89a**].

out-of-order 実行の効果は，a→b→ cの順に大きくなるが，この順序に従って制御およびハードウェア機構も複雑になる．したがって，動的コード・スケジューリングを導入する場合，導入による性能向上とハードウェア・コストとのトレードオフが重要となる．

3.2 静的コード・スケジューリング

2.2節で述べたように，プログラムに内在する命令レベル並列性を，プログラムのコンパイル時点で抽出することも可能である．最適化コンパイラによって，命令レベル並列性を抽出し，命令の並べ替えを行うことによってスーパースカラの性能を引き出すことができる．一般に最適化コンパイラにおける最適化には，大別して局所コード・スケジューリング *(local code scheduling)* および広域コード・スケジューリング *(global code scheduling)* の 2 種類がある [**中谷 90**]．以下，まずスーパースカラのための最適化コンパイラのについて述べ，局所および広域コード・スケジューリングに関し特にスーパースカラ方式にとって重要な従来の研究をまとめる．なお，静的コード・スケジューリングを支援する種々のハードウェア機構については 4章で述べる．

3.2.1 最適化コンパイラ

一般に，最適化コンパイラでは，

(a) タスク/スレッド・レベル

(b) ステートメント/ループ・レベル

(c) 命令レベル

の各レベルにおいて並列性を抽出する．特にスーパースカラ用最適化コンパイラにおいては，(b) ステートメント/ループ・レベルおよび (c) 命令レベルを重視しなければならない．これらはそれぞれ，広域コード・スケジューリングと局所コード・スケジューリングの 2 種類の静的コード・スケジューリング技術に対応する．

広域コード・スケジューリングは，基本ブロック *(basic block)* [1]内の命令レベルの並列度を増加させるように，基本ブロックを越えてコード・スケジューリングする方法である [**Fisher81**]．局所コード・スケジューリングでは，NOP ないしクリティカル・パスが最少となるように，基本ブロック内でコード・スケジューリングを行う．

広域コード・スケジューリングは主に VLIW 用コンパイラ [**Ellis86**] から，また，局所コード・スケジューリングは RISC 用コンパイラ [**久野 90**] からそれぞれ継承した技術である (Figure 3.1参照)．実用化の現状としては，まだ RISC 用コンパイラの局所コード・スケジューリングを強化した程度 [**Warren90, Golumbic90**] であるが，今後は広域コード・スケジューリングの導入が進むと考えられる．

コード・スケジューリングする時期は，レジスタ割付け *(register allocation)* との前後関係によってプリパスおよびポストパス・スケジューリングの 2 方式が存在する．組合せとしては，

(a) GCS→LCS→RA

(b) GCS→RA→LCS

(c) RA→GCS→LCS

の 3 方法が考えられる．ここで，GCS : 広域コード・スケジューリング，LCS : 局所コード・スケジューリング，RA : レジスタ割り付け，とする．それぞれ一長一短があり，どちらか一方に決めるのは難しい．しかし，以下のことはいえる．

- (a)GCS→LCS→RA の場合 : レジスタの再使用といった問題が生じないため，広域コード・スケジューリングの自由度は確保できるが，コード移動によって各変数の生

1 複数の命令から構成される命令列であるが，この命令列への入口は先頭命令からのみ可能であり，出口も最後の命令からしか許されていない．

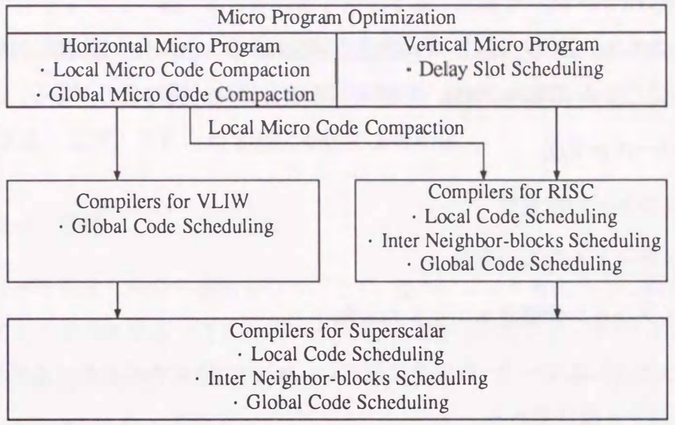


Figure 3.1 The Situation of the Optimizing Compiler for the Superscalar.

存期間 *(life time)* [2]の重なりが多くなり，余分なスピル・コード *(spill code)* [3]が必要になる [**ChowF84**]．また，スピル・コードのスケジュールができない．

- (c)RA→GCS→LCS の場合 : 広域コード・スケジューリングによるスピル・コードの増加は生じないが，レジスタの再使用によって本質的でないデータ依存関係が発生し，コードの移動に制約を与える．

したがって，(b)GCS→RA→LCS のようにレジスタ割付けの前後に 2 回のスケジューリングするコンパイラもある [**Warren90**]．

3.2.2 局所コード・スケジューリング

局所コード・スケジューリングでは，ひとつの基本ブロックが与えられたとき，プログラムの意味を変えないでその実行時間が最小となるように基本ブロック内の命令を並び替える．このスケジューリング問題は，ほとんどが NP 完全問題となる [**Hennessy83**] ため，現実的な対処法として多項式時間で良い近似解を与えるヒューリスティック・アルゴリズムを用いる．このアルゴリズムとしては，リスト・スケジューリング *(list scheduling)* [**Coffman76**]

2 その変数が定義されてから最後に使用されるまでの期間

3 レジスタを使い回すために挿入するロード/ストア命令

が広く用いられている．

ヒューリスティック・アルゴリズムを使う場合は，プロセッサの各種制約をどう反映させるかが重要となる [ChowP89]．考慮すべき項目には，次のものがある．

(a) スーパースカラ度

(b) 機能ユニットの均質性

(c) マルチサイクル命令の有無

(d) 非パイプライン化機能ユニットの有無

ここで，(a) と (b) はスーパースカラに固有の，(c) と (d) はすべてのパイプライン・プロセッサに共通する項目である．

すべての命令が単一サイクル命令というプロセッサは一般に存在せず，命令によって所要サイクル数が異なるのが普通である．このような，マルチ・サイクル命令よって生じる遅延を隠蔽するスケジュールを作り上げる必要がある．マルチ・サイクル命令が存在する場合のヒューリスティクスとしては，クリティカル・パス *(critical path)* 法 [**Coffman76**] が有効である．

機能ユニットが均質で非パイプライン化機能ユニットが存在しない場合，命令の発行において機能ユニットの競合が起きないので問題は簡単である．しかし，そうでない場合は，競合の調停と同時に競合の影響を出来るだけ抑えるようにスケジュールする必要がある．これはクリティカル・パス法では対処できない．

個々の基本ブロックを最適にスケジュールできたとしても，これらを接続すると必ずしも最適ではなくなる場合がある．これは，データ依存，分岐，資源競合などによって生じた遅延が基本ブロックを越えて伝搬するためである．このような遅延に対処するには，隣接する基本ブロック間でスケジューリングを行う必要がある．たとえば，遅延分岐 *(delayed branch)* 方式のプロセッサに対しては，ディレイ・スロットを NOP 以外の命令で埋めるディレイ・スロット・スケジューリングを行う [**Gross82**]．

3.2.3 広域コード・スケジューリング

一般に分岐命令の実行頻度は，全プログラム中の 20%程度であり [Lee84]，基本ブロックの平均サイズは 4〜5 命令程度と小さく [Jouppi89b] 命令レベルの並列度は低い．広域コード・スケジューリングでは，基本ブロックを越えたコード・スケジューリングを行う

ことで，プログラムの広い範囲から並列性を抽出する．広域コード・スケジューリングには，基本ブロックを越えた命令の移動を行う広域コード移動 *(global code motion)* とプログラムに含まれるループに着目しループの最適化を行うループ変換 *(loop transformation)* の 2 種類がある．以下，これらの手法についてまとめる．

(1) 広域コード移動

基本ブロックを越えた命令の移動を行うことで，基本ブロックを大きくし命令レベル並列性を高めることを目的とする．VLIW プロセッサ用に考案されたトレース･スケジューリング *(trace scheduling)*[**Fisher81**, **Ellis86**] やパーコレーション・スケジューリング *(percolation scheduling)* [**Aiken88**] などがある．

(a) トレース・スケジューリング：トレース・スケジューリングは，分岐頻度に偏りがあることを利用する．実行頻度の高い複数の基本ブロックを 1 つの大きな基本ブロック (トレース *(trace)* と呼ぶ) とみなし，局所コード・スケジューリングを行う．スケジューリングの手順は以下の 3 ステップからなる (Figure 3.2 参照)．

(i) 実行頻度の高い基本ブロックを選択してトレースを生成する (Figure 3.2(a) 網がけ部分)．

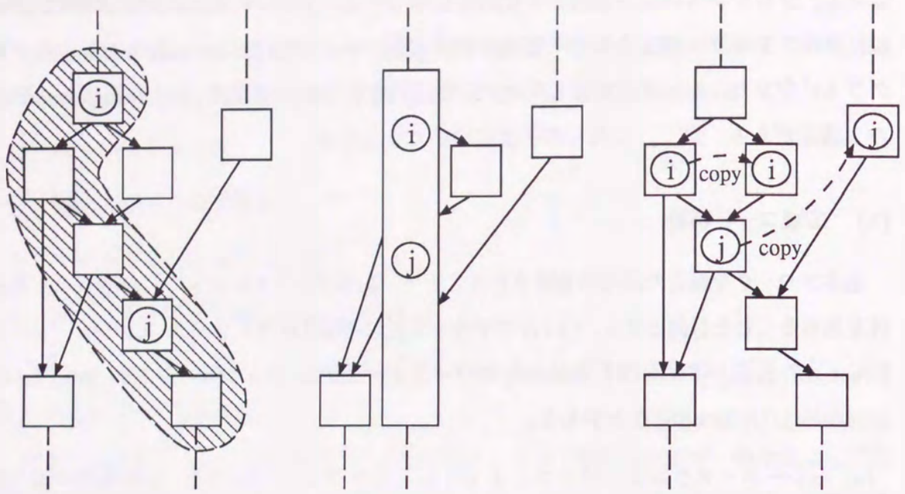
(ii) トレースを 1 つの大きな基本ブロックとみなし，局所コード・スケジューリングを行う (Figure 3.2(b))．

(iii) プログラムの実行結果が変わらないように，補正コードを挿入する (ブック・キーピング *(book keeping)* という)．Figure 3.2 の場合，スケジュールの結果，命令 **i** および命令 **j** が元の基本ブロックから別の基本ブロックへ移動しているため，補正を行う必要がある．具体的には，命令 **i** および命令 **j** が各々破線矢印先の基本ブロックにコピーされる (Figure 3.2(c))．

コード移動により高い並列度を得られるが，以下のような問題点が残されている．

問題点 1： 分岐頻度に偏りがあるようなプログラムには向いているが，そうでない場合はトレースの決定が難しく，トレースの予測が外れた場合はトレース・スケジューリングの効果が低下する．

問題点 2： ブック・キーピング処理が複雑である．



(a) pick trace　　(b) schedule　　(c) bookkeeping

Figure 3.2　Trace Scheduling.

問題点 3 : ブック・キーピング処理で補正コードを追加することによりコード・サイズが増加する．この増加は指数的に大きくなる可能性がある [**Ellis86**].

(b) パーコレーション・スケジューリング : パーコレーション・スケジューリングは，前述したトレース・スケジューリングの問題点のうち 1 および 2 を解決した方式である．

問題点 1 の解決法 : 1 つの基本ブロックに着目し，この基本ブロックに移動可能な命令を後続基本ブロックのブロック全体から得る．したがって，移動範囲が広く，かつ，分岐に偏りがない場合にも対処できる．

問題点 2 の解決法 : 命令の移動と同時に補正処理を行うため，トレース・スケジューリングに比べ補正の複雑さが緩和される．

しかし問題点 3 については，トレース・スケジューリングでは行わない同一コードの単一化 *(unification)* を行うためコードの爆発を緩和できることが期待できるが，根本的な解決にはなっていない．

さらに，トレース・スケジューリングおよびパーコレーション・スケジューリングでは共にループの取り扱いに注意する必要がある．ループ内に十分な命令レベル並列性が存在

しない場合スケジューリングの効果が生かせないからである，この場合，後述するループ変換技術，特にループ・アンローリングを十分に施し並列性を引き出す必要がある．

(2)　ループ変換

一般にプログラム実行時間の多くはループに費やされるため，ループの最適化は重要な課題である．ループ変換において重要な手法としては，ループ・アンローリング *(loop unrolling)* およびソフトウェア・パイプライニング *(software pipelining)* がある．

(a) ループ・アンローリング

ループ・アンローリングはループ・ボディをループ内に複写することで，1 回の繰り返しにおける命令数を増加させると共に，ループの分岐命令の実行回数を減らすことができる．例えば，ループを (n-1) 回展開すると，ループ内には n 個のループ・ボディが存在し，新しいループの繰り返し回数は元の 1/n となる．しかし，十分な並列性を得るためには，かなりの回数アンローリングを繰り返す必要があり，コード・サイズの増加をもたらすといった問題がある．さらに，繰り返しの終了部分で並列性が落ちるといった問題もある．

(b) ソフトウェア・パイプライニング

ループ・アンローリングした結果得られる定常状態を用いて，それでループ本体を構成するようにループ変換を行う手法である．ループ・アンローリングにおける 2 つの問題，i) コードサイズの増加，および，ii) 繰り返しの終了部分での並列性の低下，を解消できる．

ソフトウェア・パイプライニングの例を Figure 3.3 に示す．オリジナル・コードの 3 つの命令は $A_i \rightarrow B_i \rightarrow C_i$ の実行順序を守る必要があり，また，第 i 番目のループと第 $(i+1)$ 番目のループ間には $A_i \rightarrow A_{i+1}, B_i \rightarrow B_{i+1}$ の実行順序を守る必要があるとする．このとき，N 回ループ・アンローリングを施すと，網がけを施した横に並ぶ命令は互いに依存関係がない命令であり並列に実行可能である．ここで，命令 C_i, B_{i+1}, A_{i+2} をループ・ボディとしてループ変換を行うと，ソフトウェア・パイプライン化されたコードを得ることができる．元のループの各イタレーションは，あたかもパイプライン処理されているかのように実行される．

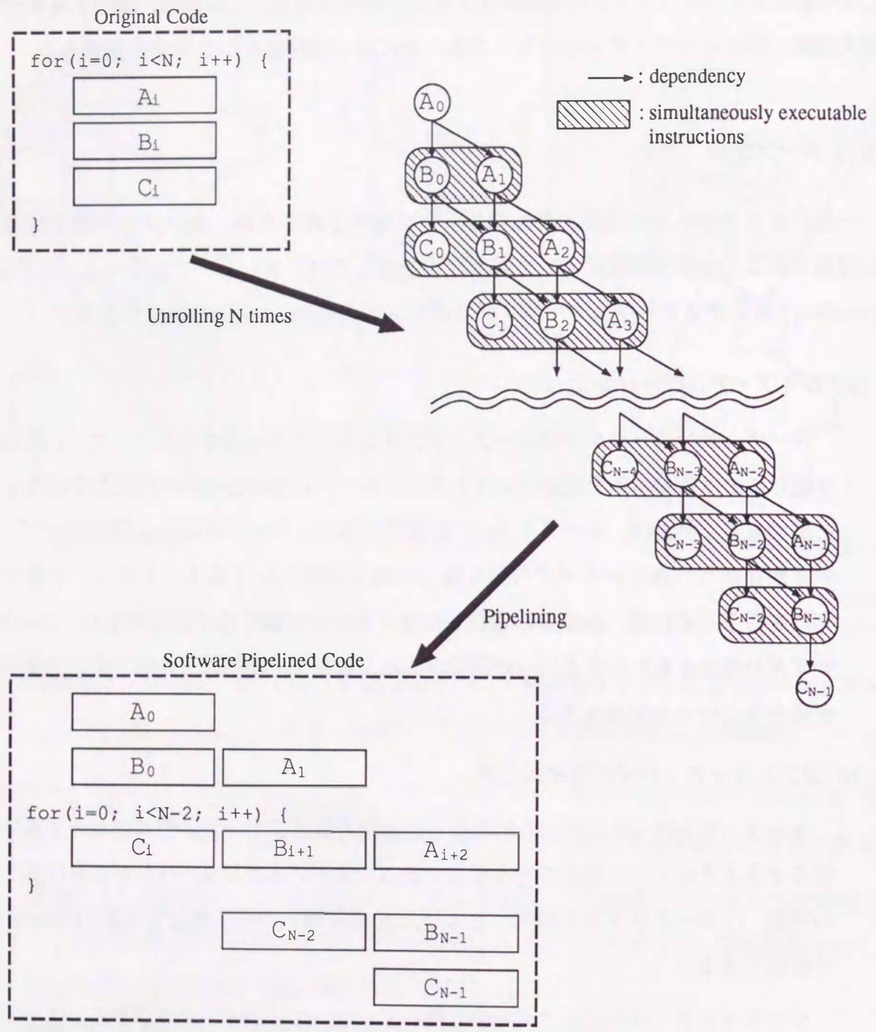


Figure 3.3 Software Pipelining.

ソフトウェア・パイプライニングのアルゴリズムには，比較的実現が容易なものとしてモジュロ・スケジューリング *(modulo scheduling)* [**Touzeau84**] がある．しかし，モジュロ・スケジューリングはループ本体が単一の基本ブロックから成る場合にのみ有効であり，IF-THEN-ELSE を内部に含むようなループには適用できない．これに

対処するために，ハイアラーキカル・リダクション *(hierarchical reduction)* 法と呼ばれるアルゴリズムが提案されている [Lam88]．また，ソフトウェア・パイプライニングとトレース・スケジューリングあるいはパーコレーション・スケジューリングとを組み合わせたアルゴリズムも提案されている．

3.3 まとめ

本章では，動的および静的コード最適化技術について述べた．ここで注意すべきことは，動的および静的コード・スケジューリングには，以下のように一長一短があることである．

- 動的コード・スケジューリング
  - 長所：静的には確定しない分岐先アドレスやメモリ・アドレスに起因する依存関係について対処可能である．
  - 短所：スケジューリングを行える範囲が，プロセッサ内部に取り込んである命令 (最大 20 数命令程度) に限定される．
- 静的コード・スケジューリング
  - 長所：プログラムの広範囲に渡ってスケジューリングが可能．
  - 短所：実行時にしかわからない依存関係には対処不可能．

このように，動的コード・スケジューリングの欠点は，逆に静的コード・スケジューリングの長所であり，その逆もいえる．すなわち，動的および静的コード・スケジューリング技術は相反するものではない．特にプログラム全体から並列性を抽出可能な静的コード・スケジューリング技術は動的のそれに比べ重要である．

第 4 章

スーパースカラ構成上の選択肢

スーパースカラ・プロセッサを構成するに当たっては，2.2節で述べた 3 つの基本選択肢に加えてその他の様々な選択肢が存在する．また，これらの選択肢は 1.1.2項で述べた実現上の課題への対処方法でもある．本章では，これらの選択肢についてまとめる．

4.1 多重命令供給

スーパースカラ度 n のスーパースカラにおいては，毎サイクル少なくとも n 個の命令をフェッチ・ブロックとしてフェッチする必要がある．このとき，以下の 2 点に関して考慮する必要がある．

(1) 命令フェッチ・バウンダリ

命令キャッシュは通常 n バウンダリで命令が配置されている．したがって，n 個の命令をフェッチする際の命令バウンダリに関して，次の 2 つの選択肢が存在する [村上 88, 原 90a]．

(a) 固定フェッチ・バウンダリ：必ず n 命令バウンダリから始まる n 個の命令をフェッチする．分岐命令が実行され分岐先アドレスが命令フェッチ・バウンダリに一致していない場合，不要な命令がフェッチ・ブロック中に含まれるといった，命令ミス・アラインメントが生じる．この不要な命令は NOP として処理されるが，それが性能に与える影響は大きい [SmithM89, Johnson89, 原 90a]．

(b) 可変フェッチ・バウンダリ：フェッチ・バウンダリが可変で，任意の命令フェッチ・アドレスから連続する n 個の命令をフェッチする．n 命令バウンダリに揃っていない場合，最も若いアドレスの命令が n 個の命令の先頭となるよう並び替えを行う．よって，命令ミスアラインメントに起因する問題は生じない．しかし，フェッチ・バウンダリに一致していない n 個の命令を読み出すためには，命令キャッシュに関して，i)n 個のバンクに分割し各々を個別にアクセス可能にする [Intel89, 原 90a, IBM90] か，ii)1 サイクルに 2 回のアクセスを可能にする [Johnson89] などの対策が必要であり，さらに，フェッチした命令の並べ替えを行う命令アラインメント機構が必要となる．

(2) キャッシュ・ライン・クロス

可変フェッチ・バウンダリの場合，フェッチ・ブロックがキャッシュ・ラインをまたぐ可能性が生じる．したがって，命令キャッシュ・ラインに関して，次の 2 つの選択肢が存在する [原 90a]．

(a) ライン・クロス不可：フェッチ・ブロック内の後尾の方の命令が次キャッシュ・ラインに存在する場合，これらの命令は当該サイクルではフェッチしない．つまり，2 つのラインにまたがった命令フェッチを行わない．これの性能に与える影響が問題となる．

(b) ライン・クロス可：2 つのラインにまたがった命令フェッチを行う．したがって，1 サイクルで 2 つのキャッシュ・ラインへのアクセスが要求される．そこで，i) 偶数番号と奇数番号のキャッシュ・ラインを格納するディレクトリを別々に設けたキャッシュ構成としたり [IBM90]，ii)1 サイクルで 2 回のキャッシュ・アクセスが可能となるようにアクセス速度を上げる，などの対策が必要である．

4.2 データ依存への対処

動的ハザード解消スーパースカラにおいては，さらに資源競合および命令間依存関係への対処法として種々の選択肢が可能である．命令間のデータ依存に関しては 3.1節で述べたように，インターロック制御，Thornton のスコアボードおよび Tomasulo のアルゴリズム [Weiss84] などがある．また，Tomasulo のアルゴリズムを拡張し，制御依存にも対処可能な動的コード・スケジューリングを利用するアーキテクチャも考案されている [Hwu87, Sohi87, 久我 89a]．

4.3 分岐命令への対処

命令パイプライン・プロセッサにおいては，分岐命令およびそれに起因する制御依存の存在によって性能が低下する．スーパースカラ・プロセッサにおいては，これら分岐ペナルティの影響がさらに著しい．分岐ペナルティの主な原因には，4.1節 (1)命令フェッチ・バウンダリで述べた命令ミス・アラインメントの他に以下のものがある．

- 命令フェッチの阻害：分岐するか否か (条件分岐の場合)，および，分岐先アドレスが確定するまで次にフェッチすべき後続命令が決まらない．
- 制御依存による後続命令実行の阻害：もし次にフェッチすべき後続命令を予測してフェッチしても，制御依存が解消するまでこれら後続命令の実行は開始あるいは終了できない．
- 分岐遅延によるパイプラインの乱れ：しかも，分岐予測が外れた場合は，パイプラインをフラッシュして正しい命令を再フェッチしなければならない．よって，分岐命令実行の遅延時間が長くなると，それだけパイプラインの乱れは大きくなる．

これらの分岐ペナルティの影響を軽減する方法として，以下の多くの手法が提案されている．なお，下記 (1)〜(4) の方法はほぼ直交関係にあり，実現に際しては様々な組合せが可能である．

(1) 分岐方式

現状のプロセッサで良く用いられる分岐方式としてはcompare-and-branch 方式やbranch-on-condition 方式がある．その他に以下の手法がある．

(a) 先行条件決定 *(advanced conditioning)* 方式 [**Rosocha79**, **久我 89b**]：条件決定 (分岐するか否か) の 1 ビットの情報を分岐に先行して生成し，分岐命令実行に伴う分岐遅延の低減を図る．また，分岐決定の情報は 1 ビットで済むため，分岐決定情報を複数保存するレジスタを容易に備えることができる．このことは，条件決定命令と分岐命令間に一般の演算命令などを自由に配置可能なことを意味し，静的コード・スケジューリングにおけるコード移動の自由度を大きくする特長がある．

(b) 分岐予告方式 [**Lilja88**]：分岐予告命令 *(prepare-to-branch instruction)* により分岐先命令を予めフェッチしてバッファリングする．これにより，分岐命令が分岐する場合にパイプラインの乱れを抑える．

(2) 分岐命令

分岐命令の機能として，以下の手法を取り入れることができる．

(a) 遅延分岐 *(delayed branch)* [**Pleszkun87**]：分岐命令の遅延スロット *(delay slot)* を定義し，遅延スロット内の命令は制御依存を受けないようにする．つまり，分岐結果 (TAKEN/NOT-TAKEN [1]) に関係なく必ず実行されるスロットを設ける．遅延スロットをすべて埋めることができた場合，分岐によるペナルティは生じない．しかし，遅延スロットに入るべき命令は制御依存を受けない命令である必要があり，プログラム内から抽出するのは難しい．そこで，分岐結果によっては遅延スロット内の命令を無効化するような無効化付き遅延分岐 *(delayed branch with squashing)* も提案されている [**McFarling86**].

(b) 静的分岐予測 [**McFarling86**]：コンパイラが意味解析結果および仮実行によるプロファイル情報に基づいて，個々の条件分岐命令について分岐予測を静的に行う．予測結果を分岐命令に反映するために，通常 2 種類の分岐命令を用意する．すなわち，i) 分岐する確率が高い分岐命令，および，ii) 分岐しない確率が高い分岐命令，である．これにより命令フェッチが阻害されることを防ぐ．

(3) 命令フェッチ

命令フェッチを中断することなく連続して行う方法として，以下の手法がある．

(a) 動的分岐予測 [**SmithJ81**, **Lee84**]：静的分岐予測とは異なり，ハードウェアが実行時に分岐予測を行う．これにはさらに，次の手法がある．

(i) not-taken 予測：not-taken と予測して，常に非分岐先の命令流をフェッチし続ける．

(ii) taken 予測：taken と予測して，常に分岐先の命令流をフェッチする．

---

1 分岐予測に関して，以下のルールを用いて記述を行う．
i)taken/not-taken：分岐予測において “分岐する/分岐しない” と予測する
ii)TAKEN/NOT-TAKEN：分岐命令を実行した結果 “分岐する/分岐しない”

(iii) 分岐予測バッファ*(BPB : Branch Prediction Buffer)*：個々の分岐命令の実行時の履歴をバッファリングしておき，それに基づいて分岐予測する．

(iv) 分岐先バッファ*(BTB : Branch Target Buffer)*[**SmithJ81**]：BPB にさらに分岐先アドレスまたは分岐先命令をバッファリングするようにする．これにより，taken 予測の場合，アドレス計算を行わずに直ちに分岐先命令の供給が可能となる．

(b) **複数命令流 (multiple instruction stream)** フェッチ [**Lee84**, **Lilja88**]：非分岐先と分岐先の両方の命令流をフェッチする．

(4) 分岐命令実行

分岐命令の実行に当り，次のような配慮を行うことが可能である．

(a) 早期分岐解消 *(early branch resolution)* [**Lilja88**]：命令パイプラインの早いステージで分岐命令を実行する．これにより，制御依存の存在時間，および，分岐命令の実行遅延時間の低減を図る．

(b) 分岐命令畳込み *(branch folding)* [**Ditzel87**]：通常の命令パイプラインの前に命令プリフェッチおよびプリデコード・ステージを設け，無条件分岐命令の分岐先アドレス生成を次サイクルの命令フェッチに間に合わせる．よって，命令パイプラインに無条件分岐命令が入ることはない．しかし，条件分岐命令は命令パイプラインで実行される．無条件分岐命令に関しては，分岐ペナルティは生じない．

4.4 制御依存への対処

分岐命令の後続命令は一般的に，当該分岐命令に対して制御依存関係にある．すなわち，分岐命令の後続命令をフェッチできたとしても，後続の命令のうちどちらのパス上にある命令を実行すれば良いのかは，制御依存が解消するまで定まらない．これは，制御依存関係にある命令がレジスタ内容などを更新してしまうと正確なマシン状態 (*precise machine state*) が保証できないことによる．したがって，このような制御依存関係にある命令の命令パイプライン内での処置については，特別な配慮が必要である．これに関しては，少なくとも以下の 2 つの方法がある．

(a) 非投機的実行：制御依存が解消するまで，当該制御依存関係にある命令の実行を開始しない．実現は容易だが，分岐命令を越えた命令の実行ができない．つまり，最大同時に実行可能な命令が，1 個の基本ブロックに限定される．よって，基本ブロック内の命令数が少ない場合，命令レベル並列性が低下する．

(b) 投機的実行 *(speculative execution)*[**Johnson90**]：制御依存が解消しなくても，当該制御依存関係にある命令の実行を開始する．ただし，制御依存が解消しないうちは，これらの命令の実行結果がレジスタ内容などの変更を行うことを禁止する．もし制御依存が解消した結果，命令の実行が不要と判明した場合は，命令自身および実行結果を無効化しなければならない．つまり，正確な分岐を保証するハードウェア機構が必要であり，この方式の実現はやや複雑である．しかし，分岐命令を越えた命令の実行が可能となり，命令レベル並列性の向上が期待できる．

さて，投機的実行の具体的な実現方式としては，次の 2 つがある．なお，正確な分岐を保証するハードウェア機構については，4.6節で後述する．

(i) 条件付実行モード *(conditional mode)*：分岐予測によりフェッチされた命令は，対応する分岐命令に起因する制御依存が解消するまで条件付実行モード下に置かれる．条件付実行モード下にある命令は実行を行うことは可能だが，実行結果でレジスタ内容などを更新することはできない．

(ii) ブースティング *(boosting)* [**SmithM90**]：個々の命令に対して，条件付実行モード下に置くか否かをコンパイラが決定する．条件付実行モード下に置かれた命令をブースト *(boost)* 命令と呼び，その指定は命令コードにより行う．ブースト命令はオブジェクト・コード上，対応する分岐命令より前に位置する．すなわち，条件付実行モードとは逆に，条件付実行モード下の命令が対応する分岐命令より先にフェッチされ実行を開始する．対応する分岐命令の分岐予測が正しい場合，ブースト命令の演算結果は有効となり，分岐予測が外れた場合，その結果は無効とする．ブースティングは静的コード・スケジューリングの自由度を大きくすると共に，スケジューリング次第では，条件付実行モードよりも投機的実行の効果が期待できる．

4.5 パイプライン復元処理

分岐予測が外れた際，誤ってフェッチした命令を無効化して，パイプラインを復元する必要がある．この復元処理 *(repair)* の方法には，以下の 2 方式が存在する．

(a) パイプライン・フラッシュ：当該分岐命令以降のすべての命令を無効化する．そして，分岐結果に従って，正しい命令を再フェッチする．

(b) 選択的命令無効化 [久我 89b]：無効化すべき命令のみを選択し，それらのみを無効化する．有効な命令は無効化されないで残る．もし，有効な命令が残らなかった場合にのみ，命令の再フェッチを行う．スーパースカラ方式では，単一命令パイプライン・プロセッサと比較して，パイプライン・ステージ数が多いため，分岐先の命令がすでにパイプライン中にフェッチされることは十分あり得る．したがって，この方式はスーパースカラ度が大きくなるほど効果が大きくなる．

4.6 正確な割込み/分岐の保証

命令実行終了順序が out-of-order となる場合，後続命令が先行命令を追い越してレジスタおよびメモリ内容を更新する可能性がある．このとき，内部割込みが発生し，当該割込みを引き起こした命令の後続命令が既にレジスタ内容などを更新していると，適切な割込み処理，および，割込み処理後のプログラム再開が出来ない場合がある．このような割込みを “不正確な割込み (imprecise interrupt)” と呼ぶ [**SmithJ88**]．仮想記憶や IEEE 浮動小数点フォーマットをサポートするマシンであれば，このような不正確な割込みによって再開が不可能になってはならない．

また，制御依存の影響を軽減するため投機的実行を行う場合，条件付実行モード下の命令 (ブースト命令も含む) が制御依存の解消を待たずに，レジスタ内容などを更新してしまう危険性がある．

このような不正確なマシン状態を防ぐ方法には種々あるが，以下の方式が主なものである．

(a) バッファ方式 [**SmithJ88**]：レジスタ・ファイルにバッファを設ける．このバッファの使用方法には，次の 2 種類がある．

(i) reorder buffer[**Sohi87**, **SmithJ88**]：条件付実行モード下にある命令の実行結果をバッファリングする．そして，制御依存が解消した時点で分岐予測が当たっていれば，その内容をレジスタに反映する．ただし，本バッファからもオペランド・フェッチが出来るよう，バイパス機構が必要となる．

(ii) history buffer[**SmithJ88**]：条件付実行モード下にある命令の実行結果でレジスタ内容を変更するのを許す．ただし，そのとき元のレジスタ内容を本バッファにバッファリングする．そして，制御依存が解消した時点で分岐予測が外れていれば，その内容でレジスタ内容を復元する．

(b) future file[**SmithJ88**]：同一レジスタ・ファイルを 2 組設け，一方をアーキテクチャ上定義されたレジスタ・ファイル *(architectural file)*，他方を条件付実行モード下にある命令の実行結果書込み用ファイル *(future file)* とする．そして，制御依存が解消した時点で分岐予測が当たっていれば，future file の内容を architectural file に反映する．この反映方法には，ファイル間転送による方法 [**SmithJ88**, **SmithM90**]，および，7.2節 (5) で述べるレジスタ・アクセスパスの切り換えによる方法 [村上 90] がある．

(c) checkpoint repair[**Hwu87**]：制御依存が発生した時点 (チェックポイント) におけるレジスタ内容を保存しておく．そして，当該制御依存が解消した時点で分岐予測が外れていれば，その内容でレジスタ内容を復元する．

(d) guarantee-not-interrupt[**HePa90**]：先行して発行された全ての命令が割込みを起こさないと保証できる場合に，命令の発行を行う．割込みの判定が遅れるとそれだけ後続命令の発行がブロックされるので，浮動小数点演算器のように演算遅延 (latency) の大きい機能ユニットは割込みを起こすか否かをできるだけ早く判定する必要がある．

(e) weakly-precise-interrupt[**HePa90**]：アーキテクチャ・レベルでは不正確な割込みとするが，その代わりにハードウェアの割込みハンドラと割込み情報を用意して，プログラムの正常な再開を可能にする方式．

上記のうち，バッファ方式，future file，および，checkpoint repair は正確な割込み，および，正確な分岐の双方を保証できる．

4.7 ISP(命令セット・プロセッサ) アーキテクチャ

スーパースカラ・プロセッサはプロセッサの構成上，ISP アーキテクチャに対して以下に挙げる制約を課す．

(a) 命令長への制約：命令長が固定かつ同一であることが望ましい．命令長が可変長あるいは固定長だが命令毎にサイズが異なる場合では，命令のデコードが終了しなければ命令長が定まらない．スーパースカラでは複数命令の同時フェッチ，デコードをすることから，命令が可変長の場合スーパースカラを実現することは不可能に近い．命令長が固定であるが命令毎にサイズがことなる場合は，命令をフェッチした際に実際にフェッチが完了した命令のみを発行するような制御を行えば良い．いずれにせよ，単一命令長の場合が最も制御が容易だからである．

(b) 命令間依存関係解析からの制約：LOAD/STORE アーキテクチャ(レジスタ- レジスタ演算であること) が望ましい．レジスタ-レジスタ演算は，命令間のデータ依存関係の検出を容易にする．すなわち，命令をデコードした段階でデータの依存関係の有無を判定可能である．これに対し，レジスタ-メモリ演算やメモリ-メモリ演算では，メモリ・アドレスが生成されるまでデータ依存関係の検出ができない．また，(a) 命令長への制約で述べた単一命令長を実現する場合，命令フィールドの割り当て上 LOAD/STORE アーキテクチャが向いている．また，レジスタ-メモリ演算ではパイプライン・ステージとして，オペランド・フェッチのために必要なアドレス生成およびオペランド・フェッチ・ステージが必要となり，パイプライン・ステージ構成が複雑となることも理由としてあげられる．

(c) コード・スケジューリングからの制約：固定サイクル演算，すなわち命令毎に決まった固定サイクルで演算が終了することが望ましい．演算パイプラインにおいてコード・スケジューリングを行う場合，命令毎に演算サイクルが固定されていることが命令スケジューリングのための必要条件である．これは，演算パイプライン化されていることにより，命令の演算開始が毎サイクル可能となるからである．一方，可変サイクル命令は，その演算サイクル数が命令自身ではなくオペランドの長さや内容で決まり，また，演算器を繰り返し使用し，しかも実行してみなければわからないため，後続命令の実行開始時点を決定するのが困難だからである．

命令セットへ以上のような制約が課せられるが，これらを満たす命令セットは，必然的に RISC プロセッサにおける命令セットに近いものとなることがわかる．

4.8 まとめ

以上，スーパースカラ方式を実現するにあたって，実現上の課題に対処するための選択肢について述べた．これら選択肢の決定によって，種々のスーパースカラ・プロセッサを構築することが可能である．したがって，これら選択肢の決定は慎重に行わねばならない．参考として，付録 Aに，現在までに発表されたスーパースカラ・プロセッサがどのような選択肢の決定を行ったかをプロセッサの諸元として示す．

スーパースカラの命令セットについては，現状では RISC 風の命令セット，すなわち命令長が固定かつ同一である命令体系のプロセッサがほとんどである．しかしながら，CISC 風の命令セットにおいても，レジスタ-レジスタ間演算命令などに限れば，通常固定かつ同一である．このことは，CISC でも命令を限定すれば十分にスーパースカラとなりうることを意味する．[Johnson90] では，Intel i386 のスーパースカラ化について考察を行っている．

第 5 章

DD 型スーパースカラ・プロセッサの設計

本章では，DD 型スーパースカラ方式を採用した試作プロセッサの開発および設計方針について述べる．また，DD 型試作スーパースカラ・プロセッサの特長のひとつである動的コード・スケジューリング・アルゴリズムの詳細および命令セットの中で特長ある先行条件決定方式についても述べる．

5.1 DD 型スーパースカラの開発方針

DD 型スーパースカラ方式に基づくプロセッサを実現し，その有効性を検証することを第一の目的とする．DD 型スーパースカラ・プロセッサの開発においては，以下の 3 点をその開発方針とした [村上 88]．

【開発方針 1】オブジェクト・コードの互換性：命令発行多重度の異なるプロセッサ間でもオブジェクト・コードが正常に動作するよう，コードの互換性を保証する．

【開発方針 2】ハードウェアによる高速化：最適化コンパイラによって極度に最適化されていないようなオブジェクト・コードに対しても，高速化ハードウェア機構により最適化コードに匹敵する性能を引き出す．

【開発方針 3】スーパースカラ度のスケーラビリティ：要求される性能に応じて，スーパースカラ度を決定する．このためには，要求性能の変化に応じてスーパースカラ度を容易に増減できるプロセッサ構成法を採る．このような構成法を採る理由の 1 つとして，現状 (1987 年) においては，シングルチップ・スーパースカラを実現できるほど，VLSI 集積度が高くないことが挙げられる．

したがって，2.2節で述べたスーパースカラの基本選択肢に当てはめると，以下のように DDU 型に分類できる．

- 開発方針 1 に対しては，動的ハザード解消スーパースカラとする．
- 開発方針 2 に対しては，高度な動的コード・スケジューリング・アルゴリズムを実装する．
- 開発方針 3 に対しては，同一構成の命令パイプラインを複数本備えた形態を採り，機能ユニットは均質型とする．

以後，DD 型試作スーパースカラ・プロセッサを DDUS*(Dynamically-hazard-resolved, Dynamically-code-scheduled, Uniform Superscalar)* プロセッサと呼ぶことにする[1]．

5.2 DDUS プロセッサの設計方針

開発方針に基づき以下のように設計方針を定める．また，1.1.2項で述べたスーパースカラ実現上の課題はプロセッサの性能を左右する重要な問題である．DDUS ではこれらの課題に対して，4章で述べた選択肢から以下の選択を行い対処している．

(1) 命令間依存関係への対処

単一命令パイプラインではインターロック制御でデータ依存関係に対処している．しかし，単純なインターロック制御ではデータ依存関係がある度にパイプラインに乱れが生じ性能低下を招く．さらに，スーパースカラ方式においては，全体のパイプライン・ステージ数が多いため性能低下はより深刻である．そこで，**動的コード・スケジューリング**を採用する．動的コード・スケジューリングでは，最適化コンパイラによる静的コード・スケジューリングで対処できないメモリ・アドレスや分岐先アドレスなど実行時にならないと分からない依存関係についても対処可能である．

DDUS では，静的コード・スケジューリングおよび動的コード・スケジューリングの両者を活用する．また，動的コード・スケジューリング・アルゴリズムとしては，Tomasulo

1 DDUS プロセッサは，同一構造の命令パイプラインを複数本備えた形態を採るため，単一命令流/多重命令パイプライン (SIMP : Single Instruction stream/Multiple instruction Pipelining) アーキテクチャと名付けた [Murakami89]．この方式に基づく試作プロセッサを『新風』と呼んでいる [五島 88]．本論文ではスーパースカラの分類に合わせ，DDUS プロセッサの呼び名で統一する．

のアルゴリズムを拡張することで，データ依存関係だけでなく制御依存関係に対しても有効なアルゴリズムを採用する [久我 89a]．すなわち，分岐命令を越えた命令実行である投機的実行を行い，その方法としては条件つき実行モードで実現する．詳細については，5.3節で述べる．

(2) 機能ユニットの均質性

プロセッサの規模を要求される処理能力に合わせて手軽に可変できるよう，スーパースカラ度に関してスケーラビリティを持たせる．つまり，プロセッサの構成を従来の単一命令パイプラインを複数本空間的に並べた構成を採る．このようなプロセッサ構成法を単一命令流/多重命令パイプライン (SIMP : Single Instruction stream/Multiple instruction Pipelining) 方式と名付ける [**Murakami89**]．

(3) 命令およびデータの供給

命令供給系はマルチバンク化，データ供給系はマルチポート化することにより，十分な供給能力を確保する．命令供給においては，分岐先バッファ(BTB : Branch Target Buffer) を採用した動的分岐予測を行うことで分岐命令によるパイプラインの乱れを抑える．なお命令供給は複数の命令を同時にフェッチすることから，分岐予測は命令ブロック (**Instruction Block**) [2]単位で行う．さらに，命令の供給時に問題となるフェッチ・バウンダリについては，キャッシュのライン・クロスは行わない可変フェッチ・バウンダリを採用する．

(4) 分岐方式

分岐方式は先行条件決定方式を採用する．詳細は 5.4.3項で述べる．なお，分岐予測が外れた際のパイプライン復元処理は，パイプラインステージの有効利用を図ため真に無効にすべき命令のみを選択して無効にする選択的命令無効化を採用する．

(5) 正確な割込み/分岐機構

DDUS では命令実行終了順序が out-of-order となるため，不正確なマシン状態が発生する可能性がある．DDUS では，reorder buffer を用いレジスタなどの更新を in-order にすることで正確な割込みおよび分岐を保証する．

2 スーパースカラでは複数の命令を同時にフェッチするが，その同時にフェッチする命令の集まりのことを指す．命令ブロックを構成する命令数はスーパースカラ度と同じである．

5.3 動的コード・スケジューリング・アルゴリズム

本節では，DDUS プロセッサにおける動的コード・スケジューリング・アルゴリズムについて述べる．

5.3.1 特長

3.1節において，動的コード・スケジューリング・アルゴリズムには，インターロック制御，Thornton のスコアボード，および Tomasulo のアルゴリズムなどがあることは述べた．データ依存関係であるフロー依存，出力依存および逆依存のすべてに対処可能なアルゴリズムは Tomasulo のアルゴリズムである．しかしながら，Tomasulo のアルゴリズムでは，register renaming を直接レジスタに対して施すため分岐命令を越える投機的実行には対処できない．投機的実行を行えるように，Tomasulo のアルゴリズムを拡張した方法として間接的な renaming を行うアルゴリズムがある [**Sohi87**]．

DDUS では，[**Sohi87**] における単一依存関係表現を拡張することで，複数の制御依存関係にも対処可能な多重依存関係表現法を採用している [久我 89a]．多重依存関係表現を用いることで，分岐予測が外れた際のパイプライン復元処理として選択的命令無効化を実現することができる．

以後，[**Sohi87**] における拡張 Tomasulo アルゴリズムによる投機的実行と，多重依存関係表現を用いた拡張 Tomasulo アルゴリズムによる投機的実行を区別するために，前者を *eager execution*，後者を *greedy execution* と呼ぶことにする．また，これに対して，非投機的実行を *lazy execution* と呼ぶことにする．

DDUS で採用する動的コード・スケジューリング・アルゴリズムには以下の特長がある．なお，このアルゴリズムは DDUS プロセッサの構成と密接に関係があるため，プロセッサの構成に関しては 6.1節，特に Figure 6.1 を参照して頂きたい．

- out-of-order 実行を 4 命令ブロック 16 命令間に対して行う．これは out-of-order 実行を行うために備える供給格納バッファ(Waiting and Reorder Buffer) の段数に起因している．

- 命令間依存関係の検出を逐次的でなく，1命令ブロック中の4命令に対して同時に行う．
- 1先行命令との間の単一のフロー依存関係だけでなく，複数(12〜15)先行命令との間の複数のフロー依存関係が，実行制御情報(局所データフローグラフ)として表現される．これを多重依存関係表現(multiple-dependency representation)と呼ぶ．
- 分岐命令が16命令中に含まれていても，分岐命令の実行完了を待たずに，その分岐命令をまたいでout-of-order実行を行う．つまり，投機的実行を行う．
- 命令ブロック単位の分岐予測を行う．分岐予測がはずれた場合でも，パイプライン全体を単にフラッシュするのではなく，真に無効化すべき命令のみを選択して無効化する．これにより，多重条件付き実行モード(multiple conditional mode)を提供する．

多重依存関係表現および多重条件付き実行モードを活用して，先行する命令に起因するフロー依存関係が解消され次第，命令実行を開始可能とする投機的実行を行う．もし，投機的実行後，制御依存関係の解消により新たなフロー依存関係が生じた場合は，命令の再実行(backtrack)を行う．

- バッファ方式の1方式であるreorder buffer[**Sohi87**, **SmithJ88**]を用いることにより，正確な割込みおよび分岐の双方を保証する．

5.3.2 多重依存関係表現法

DDUSでは，データ依存関係のうち，逆依存および出力依存に関しては容易に対処できる．すなわち，レジスタ・アクセスに関しては命令発行(I)ステージ(読出し)およびリタイア(R)ステージ(書込み)のみで行い，また，キャッシュ・アクセスに関しては実行(E)ステージ(ロード)およびRステージ(ストア)のみで行うため，もともとin-orderが保証されている．一方，フロー依存によるパイプラインの乱れを小さくするために，Eステージで先行命令に対してフロー依存関係にない命令から先に(out-of-orderに)実行を開始する．

フロー依存関係と制御依存関係との相互関係に着目して，フロー依存関係をさらに以下の2種類に分ける．

(a) 最尤フロー依存関係(PFD : Probable Flow Dependency) : Figure 5.1において，命令Eが命令Aに対してフロー依存関係にあり，かつ，命令Aに制御依存関係が存在

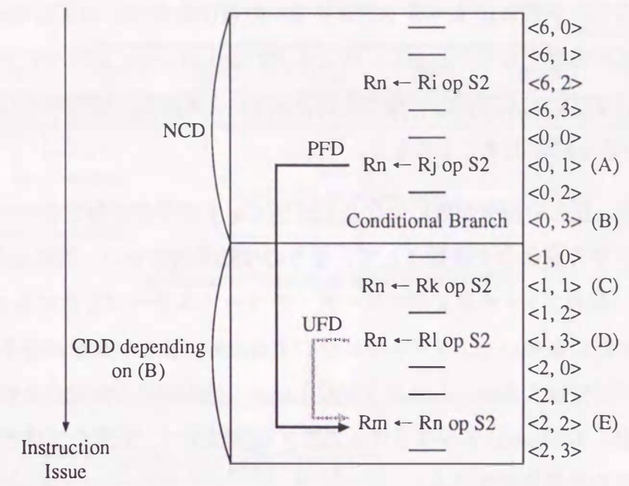


Figure 5.1 Probable flow dependency(PFD) and uncertain flow dependency(UFD).

しない場合，命令Eは命令Aに対して最尤依存関係にあると言う．

(b) 未確定フロー依存関係(UFD : Uncertain Flow Dependency) : Figure 5.1において，命令(E)が命令(D)に対してフロー依存関係にあり，かつ，命令(D)に制御依存関係(命令(B)に起因)が存在する場合，命令(E)は命令(D)に対して未確定フロー依存関係にあると言う．

したがって，ある1命令に関しては，高々1個の最尤フロー依存関係と0個以上の未確定フロー依存が存在し得る．

Figure 5.1の同じ例に対して，Tomasuloアルゴリズムを適用すると，唯一命令(D)のみが命令(E)に対してフロー依存関係を及ぼしていることしか表現できない．しかし，命令(D)が実行されるか否かは，分岐命令(B)に依存している．したがって，もし分岐命令(B)がTakenで，かつ，命令(D)と命令(E)との間に分岐したならば，命令(D)はもはや実行されない命令であり，命令(D)を識別していたタグも何ら意味を持たなくなる．

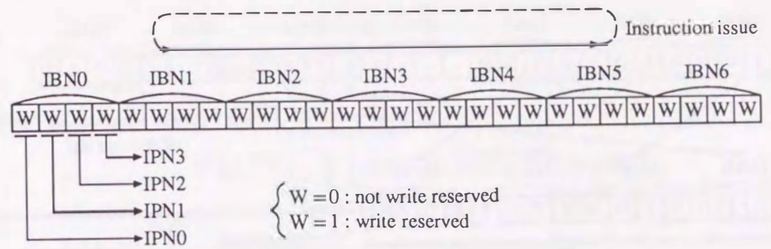
このとき，命令 (E) が読み出すべきレジスタ Rn の値は命令 (A) の実行結果であるため，依存情報の修正が必要となる．しかし，ほとんどの out-of-order 実行モデルでは，命令パイプラインをフラッシュして正しい命令を再フェッチするので，結局命令 (E) はレジスタ Rn から正しい値を読み出すことになる．

しかしながら，正しい命令が既に命令パイプライン中に存在するしないに関わらず，命令パイプライン中の全命令を無効にしてしまうのは効果的でない．特に DDUS プロセッサの場合には，命令フェッチおよびデコード・デコードステージも含めると，最大 23 個もの後続命令が既に命令パイプライン中に存在するため，正しい命令が命令パイプライン中に存在する可能性は大きい．このような理由から，選択的命令無効化を行う．選択的無効化を行う場合，Tomasulo アルゴリズムのタグ表現法では，1 個のフロー依存関係しか表現できないため不都合が生じる．

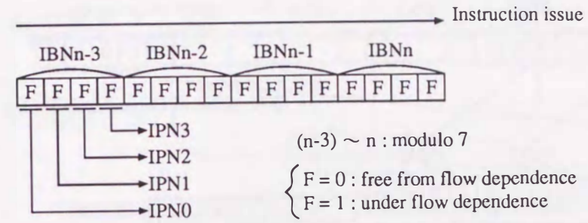
DDUS では，16 個の命令間で out-of-order 実行制御を行うことから，最大 15 個の先行命令を識別する．Figure 5.2(b) に示すソース供給リスト (SSL : Source Suplly List) と呼ぶ 16 ビットのビットマップを用いて，複数のフロー依存関係を表現できるようにした．SSL は命令発行 (I) ステージにおいて，各レジスタ対応の書き込み予約テーブル (WRT : Write Reservation Table)(Figure 5.2(a) 参照) から作成される．WRT は，対応するレジスタをデスティネーション・レジスタとする命令を全て登録する．

また，PFD と UFD を区別するために，Figure 5.2(d) に示すような制御依存リスト (CDL : Control Dependency List) と呼ぶ 16 ビットのビット・マップを導入する．CDL は SSL 同様，I ステージにおいて，Figure 5.2(c) に示す制御依存テーブル (CDT : Control Dependency Table) から作成する．CDT は，命令パイプライン中に存在する全ての分岐命令を登録する．CDL により，各命令は自分自身に対して制御依存を及ぼしている分岐命令を全て知ることができる．

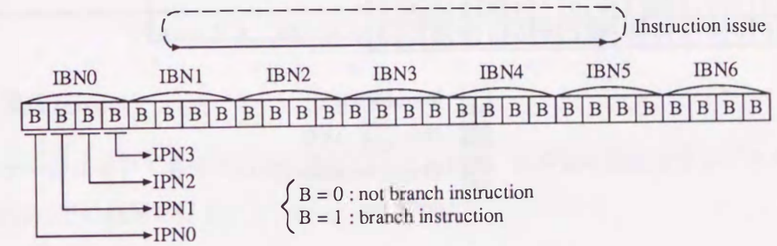
SSL と CDL とから PFD を識別するために，CDL を非制御依存領域 (NCD : No Control-dependent Domain) と制御依存領域 (CDD : Control Dependent Domain) とに分割する．NCD は CDL の最左ビット (最も先行する命令に対応) から最初に分岐命令が現れたビット位置までの領域であり，CDD はその残りの領域である．PFD を及ぼす命令は，SSL において NCD に対応する領域の中で最も右側に位置して (最も遅く投入された) フロー依存関係を起こしている命令である (Figure 5.3 参照)．



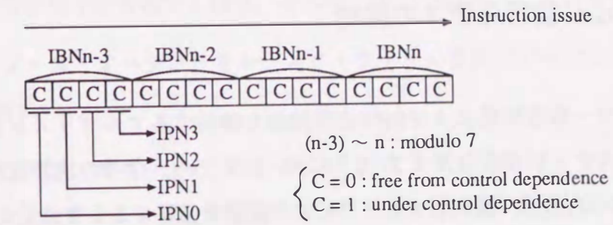
(a) Format of Write Reservation Table (WRT)



(b) Format of Source Supply List (SSL)



(c) Format of Control Dependency Table (CDT)



(d) Format of Control dependency List (CDL)

Figure 5.2　Formats of WRT, SSL, CDT and CDL.

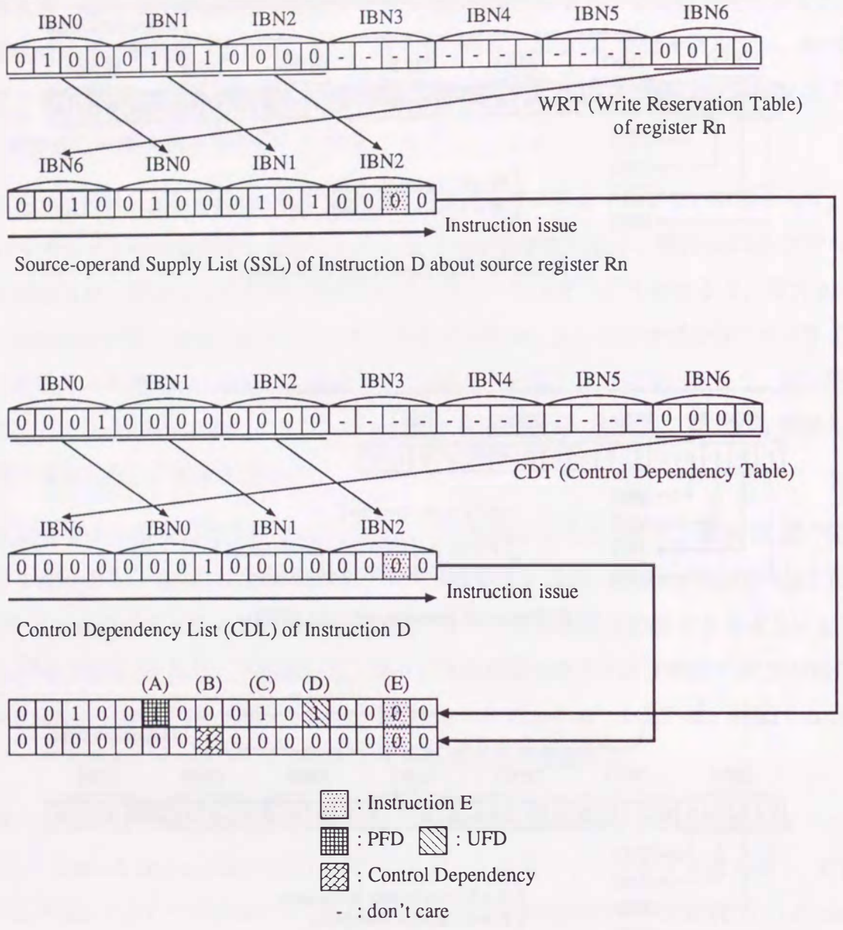


Figure 5.3 Mechanism of identifying PFD.

5.3.3 Out-of-order 実行モデルの詳細

レジスタに関するフロー依存関係および制御依存関係を解消するアルゴリズムを以下に示す．デコード (D) ステージからリタイア (R) ステージまでの，命令の状態遷移図を Figure 5.4 に示す．各々の命令は，図に示すトリガにより状態を遷移する．なお，ロード/ストア命令における依存関係を解決するアルゴリズムについては 5.3.4項において述べる．

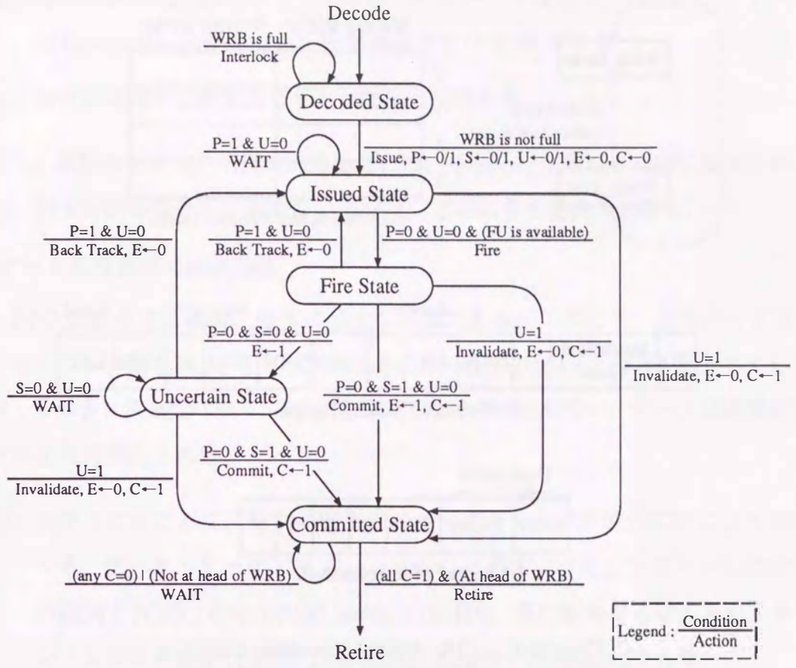


Figure 5.4 State Diagram of an Instruction.

(1) 発行 (issue)

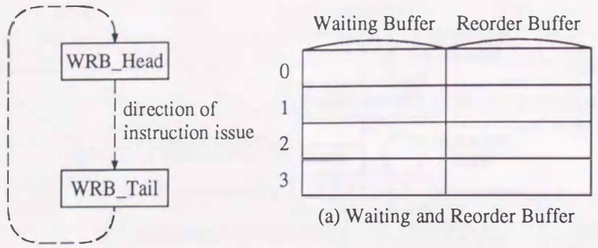
I ステージにおいて各々の命令を発行するにあたり，依存解析機能付きレジスタ・ファイルでは以下の処理を行う

(a) ディスティネーション・レジスタに対応する WRT エントリに，書込みを予約する．ソース・レジスタに対応する WRT エントリを基に SSL を作成する．

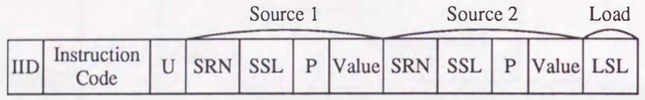
(b) 分岐命令が存在する場合，それを CDT に登録する．CDT を基に CDL を作成する．

(c) ソース・オペランドをレジスタ・ファイルまたはバイパスバッファ(BB) から読み出す．

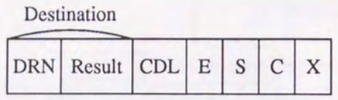
(d) ソース・オペランド，SSL および CDL を供給格納バッファ(WRB) の最尾エントリに登録することで命令を発行する．WRB が一杯の場合インターロックする．WRB の構成を Figure 5.5 に示す．



(a) Waiting and Reorder Buffer



(b) Format of Waiting Buffer



(c) Format of Reorder Buffer

C : Commited	CDL : Control Dependency List
E : Executed	DRN : Destination Register Number
P : PFD	IID : Instruction ID
S : Safe	SRN : Source Register Number
U : Undone	SSL : Source Supply List
X : eXception	LSL : Load-operand Supply List

Figure 5.5 Configuration of Waiting and Reorder Buffer (WRB).

発行された命令は，発行済み (issued) 状態に入る．

(2) 発火 (fire)

(a) 発火ルール

WRB 中の発行済み状態の各命令について，各命令パイプライン・ユニット (IPU : Instruction Pipeline Units)(Finger 6.1 参照) は，命令を発火できるかどうかチェックする．以下の発火ルールを満たす命令が発火される；

(i) すべてのソース・オペランドが有効である．すなわち，WRB の P(PFD) ビットが '0' である (PFD が存在しない)．

(ii) 命令が無効化/実行/コミットされていない．すなわち，WRB の U (Undone)，E(Executed)，C(Committed) ビットがすべて '0' である．

(iii) 命令が使用する演算ユニットが使用可能である．

もし，複数命令が発火可能である場合には，命令発行が早かった順に発火される．また，発火可能な命令が 1 個もない場合は，どの命令も発火されない．

(b) データ依存関係の解決方法

PFD が存在する場合，命令は発火不可能である．このとき，命令パイプライン・チェイニング網 (IPCN : IPU Chaining Network)(Finger 6.1 参照) をモニタしてソース・オペランドを受け取り，PFD を解決しなければならない．データ依存関係は，以下のように解決される．

(i) 各サイクルごとに，最大 4 個までのデータ・トークンが IPCN により送られてくる．データ・トークンは Figure 5.6(a) に示すように，コミットした命令の命令識別子 (CID : Committed Instruction ID)，実行結果およびデスティネーション・レジスタ番号 (DRN : Destination Register Number) を含む．

(ii) すべてのデータ・トークン中の DRN(最大 4 個) と WRB 中のソース・レジスタ番号 (最大 8 個) とを総当たりで比較する．

(iii) 一致した個々の WRB エントリについて，CID でインデクスされる SSL ビットをリセットする．

(iv) もし，リセットされた SSL ビットが PFD を示していて，かつ，P ビットが '1' であれば，待っていたソース・オペランドが到着したことを意味する．よって，データ・トークン中の実行結果を WRB のソース・オペランド・フィールドに格納すると同時に P ビットをリセットする．

(c) 実行完了後の状態遷移

発火した命令の実行が完了したら，その結果は WRB 中のデスティネーション・フィールドに格納される．この時点で，以下のチェックがなされる．

(i) もし，P ビットが '1' であれば，新たな PFD が出現したことを意味する．このとき，その命令は発行済み状態にバックトラックされる．

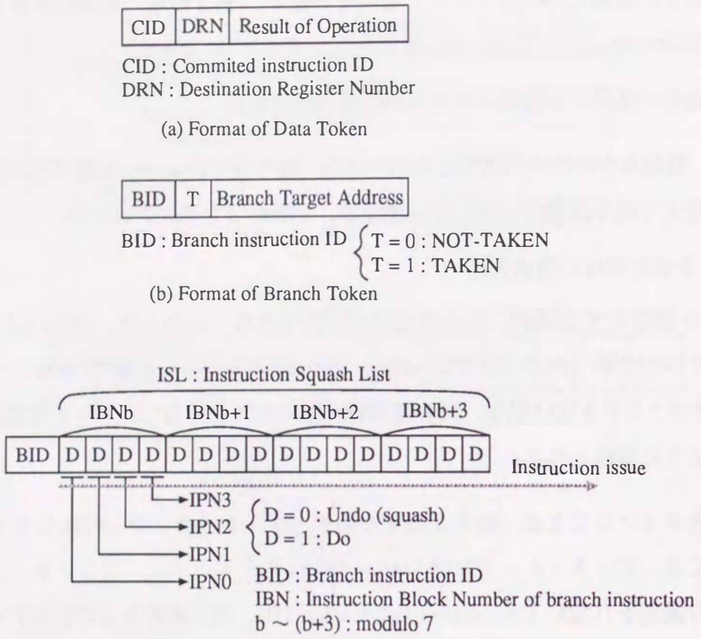


Figure 5.6　Formats of tokens.

(ii) 逆に，P ビットが '0' であれば E ビットをセットする．さらに，S(Safe) ビットが '1' であれば，制御依存関係がすべて解決されたことを意味するので，その命令は直ちにコミットされ，コミット済み (committed) 状態に入る．

(iii) 一方，S ビットが '0' で制御依存関係が解決していない場合は，解決するまで命令は未確定 (unsafe) 状態に入る．

(d) 制御依存関係の解決方法

制御依存関係は，以下のように解決される．

(i) 分岐命令がコミットされると，Figure 5.6(b) に示す分岐トークンが命令ブロック供給ユニット (IBSU : Instruction-Block Supply UnitInstruction-Block Supply Unit)(Finger 6.1 参照) に送られる．IBSU は命令ブロックのフェッチを行うと共に，選択的命令無効化を行えるようにプロセッサ内のすべての命令を管理している．

(ii) IBSU は，分岐トークンにより無効すべき命令を決定する．そして，Figure 5.6(c) 命令無効化リスト (ISL : Instruction Squash List) を作成し，無効化すべき命令を全 IPU に知らせる．

(iii) 各 IPU は制御トークンを受け取ると，無効化すべき命令に対応する WRB の U ビットをセットして，その命令が無効化されたことを示す．

(iv) また，SSL および CDL を ISL でマスクする．これにより，無効化すべき命令に対応するビットがリセットされる．

(v) さらに，制御トークン中の分岐命令識別子 (BID : Branch instruction ID) でインデクスされる CDL ビットをリセットする．

(vi) もし，CDL ビットがすべて '0' となれば，制御依存関係が解決されたことを意味し，S ビットをセットする．このとき，新たな PFD が現われた場合には，P ビットをセットし，バックトラックを起こす．

(3)　バックトラック (backtrack)

先行する分岐命令の実行完了／未完了に関わらず "投機的実行" を行える．一旦発火された命令に対しても，制御依存関係の解決に伴って新たな PFD が現われる可能性がある．このとき，その命令は発行済み状態にバックトラックされ，新しい PFD の発生により再発火可能となるまで待つ．

(4)　無効化 (undo)

無効化された命令は WRB 中の U ビットがセットされる．そして，E ビットをリセット，C ビットをセットして，直ちにコミットされてコミット済み状態に入る．

(5)　コミット (commit)

命令をコミットするためのルールは以下の通りである．

(a) 命令の実行が完了し E ビットがセットされている．

(b) PFD が残っていない．すなわち，正しいオペランドを使用して演算を行った．

(c) 制御依存が全て解決している．すなわち，S ビットがセットされている．

以上の条件を満たしている場合，WRB 中の C ビットをセットし，その命令をコミットする．コミットした命令の実行結果は，データ・トークンにより IPCN を経由して各 IPU，依存解析機能付きレジスタファイル (DHRF : Dependency-Handling Register File) 内の BB，および，データ供給機構 (DSU : Data Supply Unit) 内の SB に送られる．また，コミットした命令が分岐命令の場合，分岐トークンを IPCN 経由で IBSU に送る．

(6) out-of-order 命令実行の例

Figure 5.1の命令流において，命令 (A)，(B) の順に実行終了し，命令 (B) が命令 (C) と命令 (D) との間に分岐した場合の実行過程を Figure 5.7 に示す．

(a) 命令 (A)，(B) がまだコミットしていないとき，命令 (E) は Figure 5.3 に示す SSL および CDL を持つ．

(b) PFD である命令 (A) がコミットしたとき，その結果を用いて命令 (E) は発火する．命令 (E) の実行終了後は，まだ制御依存が解消されていないため未確定ステート (uncertain state) に入る (Figure 5.7(a) 参照)．

(c) 命令 (B) がコミットし命令 (C) と (D) との間に分岐すると，命令 (C) は選択的に無効化され命令 (D) が新しい PFD となる．したがって，命令 (E) はバックトラックを起こし再び発行済み状態 (issued state) に入る (Figure 5.7(b) 参照)．

(d) 命令 (D) がコミットすると，命令 (E) はその結果を用いて再び発火する．命令 (E) の実行終了後は，SSL および CDL はすべて '0' でありコミットルール満たすためコミット済みステート (Commited State) に入る (Figure 5.7(c) 参照)．

5.3.4 LOAD-After-STORE の依存解析

ロード–ストア命令間の依存関係には，i)LAS*(LOAD-After-STORE)*，ii)SAL*(STORE-After-LOAD)*，iii)SAS*(STOR-After-STORE)*，iv)LAL*(LOAD-After-LOAD)* の依存関係がある．このうち LAL は入力依存関係であり本質的なものではない．

SAL 依存関係については，SB において命令識別子により命令の順序を管理，すなわち renaming することで解消する．SAS 依存関係については，リタイア時に SB から MPDC

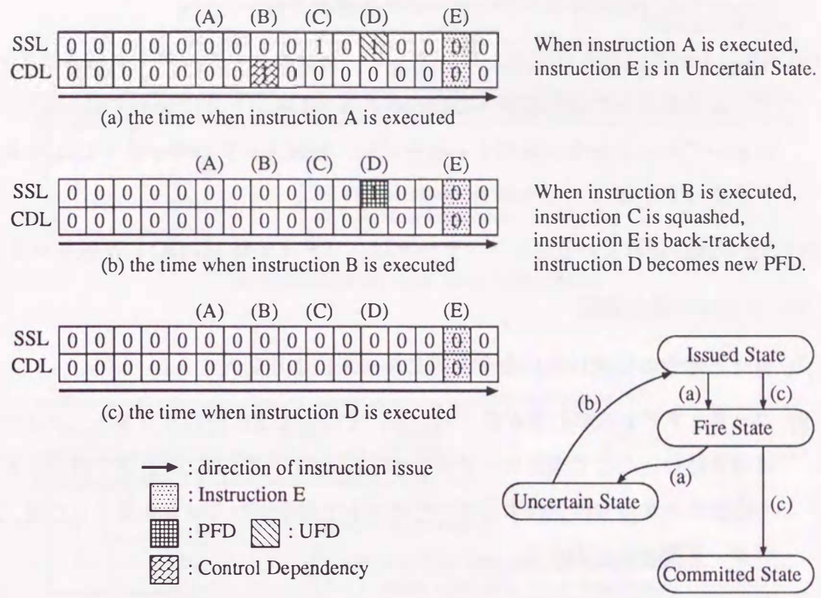


Figure 5.7 Example of out-of-order instruction execution.

への書き込みを in-order に行うことでハザードを回避する．残る LAS については，その解消を行うためにロード命令のメモリ・アドレスの計算が必要である．つまり，ロード命令のオペランドを示すソース供給リスト (SSL) だけでは依存関係の解析ができない．そこで制御依存関係を表現するように，Figure 5.8(a) に示すストア命令予約テーブル (SRT : Store Reservation Table) を用意し，命令パイプライン内のストア命令を管理する．また，ロード命令には SSL と同じように 16 ビットのビットマップである，ロード・オペランド供給リスト (LSL : Load-operand Supply List)(Figure 5.8(b)) が付けられ，先行するストア命令から受けるフロー依存関係を表現する．

LSL を使用し，ロードおよびストア命令を以下の手順により，実行することで LAS 依存関係を保証する．

(a) ストア命令の実行過程

(i) ストア命令は前述のアルゴリズムに従って発火される．ストア・アドレスが生成され，コミット・ルールを満たすまでは RB 内のリザルト・フィールドに保存される，コミット・ルールを満たしたとき，ストア命令は直ちにコミットし

Figure 5.8(c) に示すストア・トークンを送出する.

(ii) SB において，送られてきたストア・トークンのストア・データおよびストア・アドレスをストア命令識別子で識別される SB エントリに保存する．また，WB においてロード命令が存在する場合には，当該ストア命令を示す LSL 中のビットをクリアする.

(iii) SB に保存されているストア・データは，リタイア時 MPDC に反映される.

(b) ロード命令の実行過程

(i) ロード命令の発火は他の命令と同じルールにより発火する.

(ii) ロード・アドレスの計算が終了したら，その値を RB 中のリザルト・フィールドに書き込む．ここで当該ロード命令の LSL がすべて '0' になるまで待つ．すなわち当該ロード命令に先行するすべてのストア命令がコミットを終了した後，DSU へロード要求を出力する.

(iii) 要求されたメモリ・アドレスのデータを，マルチポート・データキュッシュ(MPDC : Multiple-Port Data Cache) およびストア・バッファ(SB : Store Buffer) から読み出す.

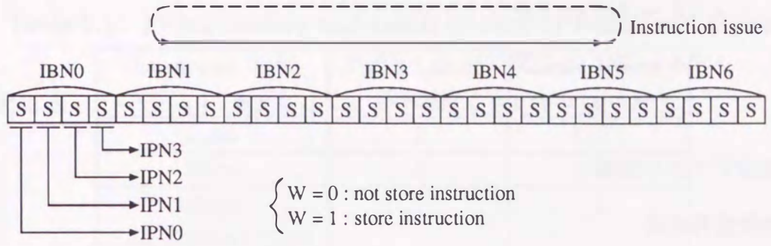
(iv) SB 内では当該ロード命令のロード・アドレスと SB 内のストア命令のストア・アドレスを比較し，アドレスが一致した場合そのストア命令のストア・データをロード・データとする．アドレスが一致しなかった場合は，最新値が MPDC 内にあることを意味するため，MPDC からフェッチしたデータをロードデータとする.

(v) 上記ロード・データは直ちにコミットされ，データ・トークンにより全ての IPU と BB，および SB に対し IPCN を経て伝えられる.

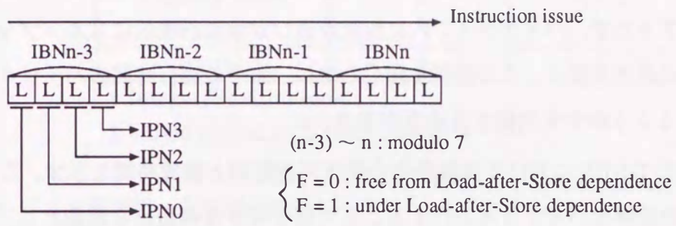
5.4 DDUS の ISP(命令セット・プロセッサ) アーキテクチャ

5.4.1 特長

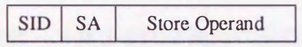
DDUS プロセッサの命令セットは，4.7節で述べた点を考慮て設計しており，以下のような特長を持つ.



(a) Format of Store Reservation Table (SRT)



(b) Format of Load-operand Supply List (LSL)



SID : Store instruction ID
SA : Store Address

(c) Format of Store Token

Figure 5.8 Formats of SRL, LSL and store token.

- 32 ビットの単一命令長
- ロード/ストア・アーキテクチャ
  - 3 種類のアドレッシング・モード
    - 実効アドレス ← ベースレジスタ
    - 実効アドレス ← ベースレジスタ + 変位
    - 実効アドレス ← プログラム・カウンタ + 変位
  - レジスタ指向の 3 オペランド命令

* Rd ← Rs1 op Rs2
* Rd ← Rs1 op 定数

ここで，Rd：デスティネーション・レジスタ，Rs：ソース・レジスタ

- 固定サイクル演算
- 均衡演算時間

DDUSプロセッサでは正確な割込みおよび分岐を保証するため，reorder bufferを導入している．reorder bufferにより，レジスタなどの更新をin-orderに行う．命令の演算時間に大きなバラツキがある場合には，演算時間が長い命令がreorder bufferからリタイアされず，パイプライン内に無駄な遊びが生じ結果的にスループットおよび応答速度の低下を招く．この弊害を防ぐために，命令の実行時間のバラツキをある範囲に収まるよう命令を制限する必要がある．

Table 5.1にDDUSにおける演算命令の発火可能周期と演算時間を示す．このように，すべての演算をパイプライン化することで毎サイクルの発火を可能とし，演算時間のバラツキも2〜6マシン・サイクル内に収めている．

- 先行条件決定に基づく条件分岐命令

先行条件決定により分岐遅延の低減を図ると共に，静的コード・スケジューリングの自由度を増加させる(詳細は5.4.3項で述べる)．

5.4.2 命令一覧

命令を分類すると，以下の8種類に分類できる．また，演算命令は演算器として使用しているビルディング・ブロックICの演算機能に依存する．DDUSプロセッサの命令フォーマットおよび命令の一覧表を付録B.3DDUSプロセッサの命令一覧にまとめる．

(a) ロード/ストア命令：レジスタ-メモリ間のデータ転送命令およびレジスタへの定数ロードを行う．データ・タイプは，i)符号付き/符号なし8ビット整数，ii)符号付き/符号なし16ビット整数，iii)符号付き/符号なし32ビット整数，iv)符号付き/符号なし64ビット整数，v)IEEEフォーマット単精度浮動小数点数(32ビット)，vi)IEEEフォーマット倍精度浮動小数点数(64ビット)を取り扱う．定数ロード命令では，命令中の18ビット定数値をレジスタにロードする．

Table 5.1 Firing latency and result latency of Functional Units.

Instruction Type	Firing Latency (cycles)	Result Latency (cycles)
Load	2	4/6 †
Store	2	4/6 †
Integer Add/Subtract/Shift	2	2
Integer Multiply	2	2
Floating-point Add/Subtract/Shift	2	4/5 ‡
Floating-point Multiply	2	4/6 ‡
Branch	2	4

† Integer/Floating-point
‡ Single precision/Double precision

(b) レジスタ間移動，型変換命令

(c) 整数演算命令：除算命令は演算器を数十サイクルに渡って占有するため提供しない．

(d) 論理演算命令

(e) ビット演算，シフト・ローテイト命令

(f) 浮動小数点数演算命令：整数演算命令同様，除算命令は提供しない．

(g) 先行条件決定，分岐，CALL命令：先行条件決定および分岐命令は，DDUSにおける命令の中で特長的なものの1つである．詳細は5.4.3節で述べる．

(h) システム制御命令：トラップ命令など．

5.4.3 分岐命令の仕様

先行条件決定命令および分岐命令の組合せによる条件分岐はDDUSの命令セット・アーキテクチャにおける特長の1つである．まず，条件分岐処理過程についてまとめ，従来の条件分岐方式と先行条件決定方式との違いを明らかにする．

(1) 条件分岐処理過程

一般に条件分岐を行う場合，以下の処理を行う必要がある．

(a) コンディション生成：条件分岐の元になるコンディション (状態) を生成する．このコンディションは，算術・論理演算などを実行することにより生成される．

(b) 条件決定 (condition decision)：(a) で生成したコンディションを分岐条件でテストし，分岐するか否かを決定する．

上記 (a)(b) は分岐するしないに関わらず行う．分岐する場合は，さらに以下の処理が必要である．

(c) 分岐先アドレス生成：分岐先の命令アドレスをアドレッシング・モードに従って計算する．なお，アドレッシング・モード次第で，この処理は省ける．

(d) 分岐処理：(c) で生成した分岐先アドレスをプログラム・カウンタ *(PC : Program Counter)* に設定する．

上記の 4 処理には，その先行制約に関し，(a)→(b)→(d)，および，(c)→(d) という半順序関係 (→：先行制約) が存在する．

(2) 従来の条件分岐方式

従来のプロセッサで主に用いられている条件分岐方式には，compare-and-branch 方式と branch-on-condition 方式がある．

- compare-and-branch 方式：Table 5.2に示すように，条件分岐処理過程の (a)(b)(c)(d) のすべてを 1 個の compare-and-branch 命令で行う．次に述べる branch-on-condition 方式における問題点がない反面，クリティカルパスが (a)→(b)→(d) と長いため分岐コストが大きくなる欠点がある．

- branch-on-condition 方式：コンディション・コード (CC : Condition Code) を用いて，(a) と (b)(c)(d) とを別々の命令で行うようにする．Table 5.2 に示すように，(a) は通常の算術・論理演算命令における CC 設定で，(b)(c)(d) は 1 個の branch-on-condition 命令で行う．branch-on-condition 命令自身のクリティカルパスが (b) →(d) ないし (c)→(d) と短くなるため，compare-and-branch 命令に比べて分岐コストは小さくなる．しかし，CC を導入したことで，以下の問題点が生じる．

  – CC に関するフロー依存関係：CC を設定する演算命令と CC を参照する branch-on-condition 命令との間にフロー依存関係が生じるので，一般の命令間依存関係

Table 5.2 Branch Schemes.

Scheme	(a) Condition Generation	(b) Condition Decision	(c) Address Generation	(d) Branching
compare-and-branch	compare-and-branch instruction			
branch-on-condition	ALU instruction	branch-on-condition instruction		
advanced conditioning	ALU instruction	advavnced conditioning instruction	branch instruction	

と同様にこれを保証する必要がある．単純なパイプライン・インターロックでこれは解決可能だが，パイプラインに乱れが生じる．乱れを抑えるには，CC のスコアボーディングなどの機構が必要である．

– CC に対するアクセス競合：CC を設定/参照する命令間で，CC に対するアクセス競合が生じ得る．よって，競合関係にある命令間の逆依存および出力依存関係を保証するために，パイプラインに乱れが生じる．アクセス競合の頻度を軽減するには，CC を複数個設けるなどの手段が必要となる．ただし，この場合，CC を設定/参照するすべての命令に CC 番号を指定するフィールドが必要となる．

– 静的コードスケジューリングへの影響：branch-on-condition 方式は，CC 設定を行う演算命令と CC 参照を行う branch-on-condition 命令とを離せる場合にのみ，compare-and-branch 方式に対する優位性がある (Figure 5.9(a)(b) 参照)．この場合，当該 2 命令の間に 1 個以上の命令が存在することになるが，そのいかなる命令も CC を変更してはいけない．このとき，CC 設定方式の相違により，次の影響が生じる．

  * 暗黙的設定：CC を設定するか否かを暗黙的に定める．CC を設定しない命令しか当該 2 命令間に挿入できないので，コードスケジューリングの自由度が低くなる欠点がある．

  * 明示的設定：命令フィールド中で CC を設定するか否かを明示的に指定する．暗黙的設定に比べると自由度は高いが，CC が 1 個しかない場合には，CC 設定－参照関係の入れ子ないし交差が許されないという制限がつく．

(3) 先行条件決定方式

これまでに述べた条件分岐方式以外に，Figure 5.9に示す先行条件決定と呼ぶ方式が提案されていることは4.3節でも述べた．DDUSプロセッサでは本方式を採用する．本方式はTable 5.2に示すように，(a) はbranch-on-condition 方式同様，通常の算術・論理演算命令におけるCC設定で行うが，(b) と (c)(d) とは別々の命令で行う．(c) (d) は1個の“分岐”命令で行い，そのクリティカルパスは (c) →(d) と短い (さらに，アドレッシング・モード次第では，(c) が省けて (d) のみとなる)．

本方式では，先行条件決定命令の実行結果である“分岐するか否か (TF : True/False)”の1ビット情報はTF レジスタに格納して分岐命令に伝達される．このTF は，branch-on-condition 方式のCC に比べて，以下の特徴がある．

- TF に関するフロー依存関係：CC の場合と同様．
- TF に対するアクセス競合：複数のTF レジスタを設けることで，アクセス競合を緩和できる．この場合，TF レジスタを設定/参照する命令は先行条件決定命令と分岐命令だけであるから，他の命令のフォーマットには影響を与えない．
- 静的コードスケジューリングへの影響：複数のTF レジスタを設けることで，TF設定-参照関係の入れ子および交差を含めて，自由度の高いコードスケジューリングが可能である (Figure 5.9(c)(d) 参照)．
- ハードウエア・コストが小さい．

しかしながら，本方式でも，(a) のコンディション生成と (b) の条件決定との間ではCC を用いるので，branch-on-condition 方式で述べたCC に関する問題はそのまま残る．この問題は，CC を独立した単一のレジスタとしてではなく，個々の汎用/浮動小数点レジスタに付随するタグとして定義することで解決する．つまり，CC を設定する命令を実行すると，そのデスティネーション・レジスタのタグにCC が設定される．この改良されたCC は，通常のCC に比べて以下の特長がある．

- CC に関するフロー依存関係：レジスタ・スコアボーディング機構により同時に解決される．
- CC に対するアクセス競合：汎用/浮動小数点レジスタと同数個のCC が存在し，アクセス競合が緩和される．しかも，CC の指定はデスティネーション・フィールドを兼用するので，新たなフィールドは必要ない．

- 静的コードスケジューリングへの影響：CC への設定は暗黙的に行う．ただし，branch-on-condition 命令と異なり先行条件決定命令は任意の位置に置けることから (branch-on-condition 命令は基本ブロックの底部のみ)，CC が変更される前に条件決定を済ませることが可能である (Figure 5.9(c) 参照)．

(4) 関連レジスタ

先行条件決定方式の導入に関連して，次の2種類のレジスタを導入する．

(a) TF レジスタ (TF : True/False register)：32個の1ビット長レジスタで，“分岐が成立する (True=TAKEN) か否 (False=NOT_TAKEN) か” を保持する．ただし，TF0の値は常に1(TAKEN) で，無条件分岐の際にはこれを指定する．

(b) タグ付き汎用/浮動小数点レジスタ：個々の汎用/浮動小数点レジスタに4ビットのタグを設ける．算術・論理演算命令の実行により生成されたCC は，そのデスティネーション・レジスタのタグに格納される．

(5) 関連命令および処理過程

条件分岐に関連して，次の3種類の命令を定義する (Figure 5.10参照)．

- テスト (test) 命令：ソースレジスタのタグ内のCC に対して分岐条件と合致するか否かのテストを行い，分岐する/しないのtrue/false 値をデスティネーション・レジスタに設定する．条件分岐処理過程の (b) に相当する．
- 比較&テスト (compare-and-test) 命令：2つのソースオペランド間の算術・論理比較を行い，その結果に対して分岐条件と合致するか否かのテストを行い，true/false 値をデスティネーションTF レジスタに設定する．条件分岐処理過程の (a)(b) に相当する．
- 分岐 (branch) 命令：ソースTF レジスタのtrue/false 値に従って分岐結果 (TAKEN/NOT-TAKEN) を決定する．TAKEN の場合さらに，プログラム・カウンタ (PC) に分岐先アドレスを設定する．分岐先アドレスはアドレッシング・モード (PC 相対/GR 相対/PC+GR) に従って計算する．条件分岐処理過程の (c)(d) に相当する．

また多方向分岐用に，2個のTF レジスタ間の論理演算を行う命令も備えている．

(6)　先行条件決定方式の利点

先行条件決定方式を採用したことにより，以下の利点が期待できる．

- テスト命令または比較&テスト命令と分岐命令との間の距離を大きくするようにスケジューリングすることで，条件決定 (分岐するか否か) に伴う分岐遅延を隠蔽できる．
- TF レジスタ間の論理演算命令を用いることによって，複数の分岐命令を 1 個の分岐命令にまとめることができる．すなわち，分岐命令数の削減が可能である．

5.5　まとめ

本節では，DD 型スーパースカラ方式を採用した試作プロセッサの開発方針および設計方針について述べた．試作プロセッサである DDUS プロセッサは，最適化コンパイラでも対処できない依存関係に対し，高度な動的コード・スケジューリング・アルゴリズムにより対処し性能向上を目指す．以下に，DDUS の特長をまとめる．

(a) 高度な動的コード・スケジューリングの採用．

(b) 分岐先バッファを用いた動的分岐予測の採用．

(c) 先行条件決定命令の採用．

(d) 選択的命令無効化の採用．

(e) 正確な割り込み，および，分岐を保証する reorder buffer の採用．

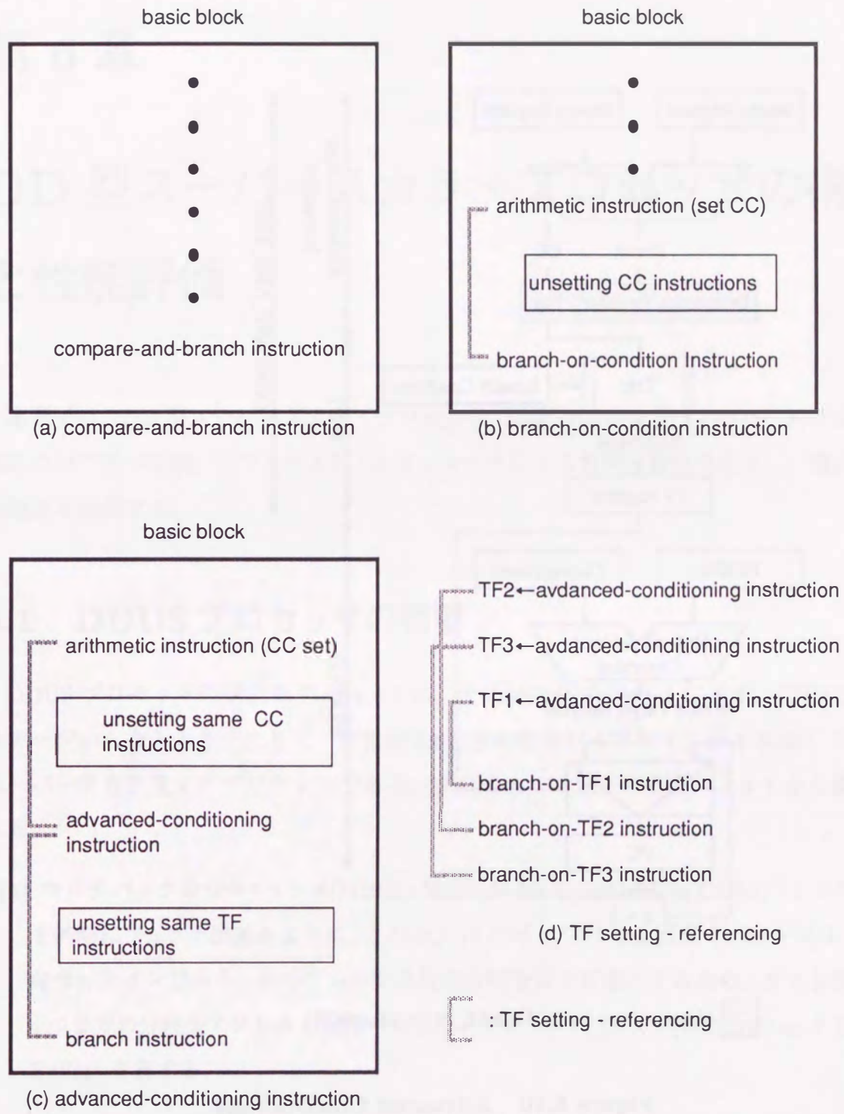


Figure 5.9　Branch Schemes.