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

五　島　　正　裕

目 次

第 1 章　緒論 **3**
1.1 LSI の微細化とマイクロアーキテクチャの変化 3
1.1.1 非演算器ユニットの遅延の増大 4
1.1.2 パイプライン化 6
1.1.3 非集中化 8
1.1.4 パイプライン化，非集中化と投機 11
1.2 投機と Out-of-Order 命令スケジューリング 13
1.2.1 投機と命令スケジューリング 13
1.2.2 0 次キャッシュと動的命令スケジューリングの効果 15
1.3 本稿の内容 19

第 2 章　CMOS 回路の基礎 **23**
2.1 ダイナミック論理 23
2.1.1 ダイナミック・ゲート 23
2.1.2 ドミノ論理 25
2.1.3 ダイナミック回路のメリット，ディメリット 25
2.2 一致比較器 27
2.2.1 *n* ビット一致比較器 27
2.2.2 1ビット一致比較器 28
2.3 RAM 29
2.3.1 RAM のロジック 31
2.3.2 多ポート RAM 34
2.3.3 シングル・ビットライン 34
2.3.4 ビットライン・ドライバのサイジング 37
2.4 CAM 39
2.4.1 CAM のロジック 39
2.4.2 多ポートの CAM 41
2.4.3 RAM との関係 41
2.5 CMOS のスケーリング 43

第 3 章　Out-of-Order 命令スケジューリング　47
3.1 Out-of-Order 命令スケジューリングの原理 47
3.1.1 スーパースカラ・プロセッサの基礎 48
3.1.2 レジスタ・リネーミング 49
3.1.3 命令ウィンドウ 52
3.1.4 レジスタ・リネーミングと Out-of-Order 実行 53
3.1.5 命令スケジューリング 54
3.1.6 Out-of-Order スケジューリングの 5 フェーズ 56
3.2 Out-of-Order 命令スケジューリングと命令パイプライン 58
3.2.1 命令パイプラインにおける 5 フェーズ 58
3.2.2 スーパースカラ・プロセッサの命令パイプラインの乱れ 59
3.2.3 命令ウィンドウのパイプライン動作 61
3.2.4 5 フェーズのパイプライン化 63
3.2.5 3.2 節のまとめ 64
3.3 命令ウィンドウ・エントリと物理レジスタの寿命 64
3.4 *Rename* ロジック 67
3.4.1 *Rename* ロジックの処理 67
3.4.2 *Rename* ロジックの実装 69
3.4.3 *Rename* ロジックの動作タイミング 71
3.4.4 CAM 方式 *Rename* ロジック 72
3.5 *Select* ロジック 74
3.5.1 *Select* ロジックの動作 74
3.5.2 カスケード方式 *Select* ロジックの実装 75
3.6 *Wakeup* ロジック 77
3.6.1 RAM 方式 *Wakeup* ロジック 77
3.6.2 連想方式 *Wakeup* ロジック 79
3.6.3 連想方式 *Wakeup* ロジックの構成 80
3.6.4 連想方式 *Wakeup* ロジックの遅延 86
3.7 命令ウィンドウの非集中化 86
3.7.1 ロジック実効サイズの縮小 87
3.7.2 クリティカル・パスの分離 88

第 4 章　間接方式　89
4.1 間接方式の原理 89
4.1.1 間接方式のデータ構造 90
4.1.2 行列アクセス 92

4.1.3 行列アクセスの式 93
4.2 間接方式のロジック 93
4.2.1 デスティネーション行列 93
4.2.2 ソース行列 95
4.2.3 *rdy* レジスタ 97
4.3 連想方式との関係 97
4.3.1 RAM 方式との関係 97
4.3.2 連想方式との関係 98

第 5 章 Dualflow アーキテクチャ 99
5.1 Dualflow アーキテクチャの実行モデル 99
5.1.1 Dualflow アーキテクチャの実行モデルの概要 100
5.1.2 実行例 101
5.2 Dualflow アーキテクチャの命令スケジューリングの原理 103
5.2.1 命令スケジューリングのためのデータ構造 103
5.2.2 コンシューマ行列アクセス 106
5.2.3 コンシューマ行列アクセスの式 107
5.3 Dualflow アーキテクチャの *Wakeup* ロジック 108
5.3.1 *grant* のホールド 108
5.3.2 列リセット方式 110
5.3.3 ロジックの構成 110
5.4 Dualflow アーキテクチャの実装 110
5.5 Dualflow アーキテクチャのコード生成 112
5.5.1 GCC 112
5.5.2 Dualflow アーキテクチャ専用パス群 113
5.5.3 条件分岐に関する最適化 114
5.6 Dualflow アーキテクチャの性能評価 116

第 6 章 直接方式 121
6.1 直接方式の原理 121
6.1.1 直接方式のデータ構造 121
6.1.2 プロデューサ行列アクセス 123
6.1.3 Dualflow アーキテクチャとの関係 124
6.2 直接方式のロジック 125
6.3 依存行列アクセスの高速化 125
6.3.1 依存行列の非集中化 126

6.3.2 依存行列の 2 階層化 126
6.3.3 L-1 行列の狭幅化 128
6.4 間接方式に対する高速化手法の適用 129
6.5 IPC の評価 129

第 7 章 回路の評価 **133**
7.1 評価方法 133
7.2 Out-of-Order 命令スケジューリングのロジック 133
7.2.1 連想方式のデスティネーション RAM 134
7.2.2 連想方式のソース CAM 135
7.2.3 間接/直接方式の依存行列 136
7.2.4 *Select* ロジック 137
7.3 回路面積 140
7.4 回路遅延 141

第 8 章 結論 **145**

謝辞 **145**

参考文献 **149**

著者発表論文 **155**

図目次

1.1 CINT95 の実行時間と最上位キャッシュのミス率 18

2.1 ドミノ論理 24
2.2 スタティック OR ゲートとダイナミック OR ゲート 26
2.3 *n* ビット一致比較器 27
2.4 ダイナミックな XOR ゲートを用いた一致比較器 28
2.5 ダイナミックなセレクタを用いた一致比較器 28
2.6 SRAM のパラメタの比較 29
2.7 RAM の回路図 31
2.8 RAM セル・アレイ 32
2.9 マルチポートの RAM セルと CAM セル 35
2.10 シングルエンドのセンスアンプ 36
2.11 ポート数とセル面積 38
2.12 CAM のロジック 40

3.1 レジスタ・リネーミング前と後のコード 50
3.2 図 3.1 のコードのデータ・フロー・グラフ 50
3.3 命令ウィンドウ 50
3.4 命令パイプライン 58
3.5 バックエンドのパイプライン動作 61
3.6 *Wakeup* フェーズに 1.5 サイクル充てた場合のバックエンドのパイプライン動作 . . 62
3.7 I_x — I_l 間の実効レイテンシが 2 サイクルの場合のバックエンドのパイプライン動作 62
3.8 レジスタ・リネーミング前（左）と後（右）のコード（図 3.1 の再掲） 68
3.9 RAM 方式レジスタ・マップ・テーブル 68
3.10 CAM 方式レジスタ・マップ・テーブル 68
3.11 *Rename* ロジックのブロック図 70
3.12 *Rename* ロジックのタイミング・チャート 72
3.13 CAM 方式 RMT の本体 CAM のブロック図 74
3.14 カスケード方式 *Select* ロジックのブロック図 76

3.15 RAM 方式 *Wakeup* ロジックのブロック図 78
3.16 連想方式 *Wakeup* ロジック 80
3.17 連想方式 *Wakeup* ロジックのブロック図 81
3.18 連想方式 *Wakeup* ロジック 82
3.19 連想方式 *Wakeup* ロジックにおける *rdy* と *rdyL*/*rdyR* の関係 84
3.20 *Wakeup*，*Select*ロジックの非集中化 87

4.1 レジスタ・リネーミングされたコード 91
4.2 間接方式の命令ウィンドウ 91
4.3 間接方式の概念図 91
4.4 デスティネーション行列のロジック 94
4.5 ソース行列のロジック 96

5.1 Dualflow アーキテクチャの命令フォーマット例 100
5.2 |*a* – *b*| を計算するコード 102
5.3 図 5.2 のコードを実行する際の命令ウィンドウの状態 102
5.4 コード 104
5.5 Dualflow アーキテクチャの命令ウィンドウ 104
5.6 Dualflow アーキテクチャの概念図 104
5.7 命令ウィンドウの非集中化 111
5.8 条件分岐の処理 115
5.9 参照回数と寿命の割合とその累積 117
5.10 動的なコードの増加率 118

6.1 レジスタ・リネーミングされたコード 122
6.2 直接方式の命令ウィンドウ 122
6.3 直接方式の概念図 122
6.4 行列の非集中化 127
6.5 L-1 行列の狭幅化 128
6.6 L-1 行列の幅 *w* に対する IPC の変化 131

7.1 Prefix-Sum Thicket 138
7.2 s3 を求める論理回路 138
7.3 回路面積 139
7.4 各方式の回路遅延 142

表 目 次

1.1 MIPS R10000 プロセッサの命令パイプライン構成と主要なユニット 4
1.2 各投機技術の 2 つの事象 14
1.3 キャッシュ，メモリのパラメタ 17

2.1 SRAM のパラメタの比較 30
2.2 RAM セル・アレイと CAM セル・アレイの入出力ライン 42

3.1 命令スケジューリングのフェーズ 57
3.2 ロジックのパラメタ 67

5.1 連想方式と Dualflow アーキテクチャの *rdy*，*rdyL*/*rdyR*，*irdy* フィールドの関係 105
5.2 静的なコードの増加率 118

6.1 SPEC CINT95 ベンチマーク・プログラム 129
6.2 キャッシュ，メモリのパラメタ 130

7.1 CS80A CMOS プロセスの諸元 133
7.2 各モデルのパラメタ 134
7.3 発行要求数のエンコーディング 137
7.4 各ロジックのパラメタ 139
7.5 セル面積 139
7.6 各バンクのパラメタ 141

第1章　緒論

「スーパースカラ・プロセッサの商業的な成功は，一重にそのバイナリ互換性のためだ．しかし，まさにそのバイナリ互換性のための動的命令スケジューリング・ロジックの『複雑さ』が，クロック速度向上の足枷となる．いずれ，より高い性能を達成できるVLIWプロセッサによって置き換えられるだろう」—— 1995年頃は，このような意見が大勢を占めていた[16, 17].

しかし，そのような考えは誤りであったことが徐々に明らかにされつつある．LSIの微細化が進むにつれ，プロセッサのデザインは大きな転換点を向かえている．以前のLSIではゲート遅延がロジックの遅延の大部分を占めていたのに対して，最近のLSIでは配線遅延の影響が無視できなくなってきたためである．配線遅延の影響が大きいLSIを用いて製造されるプロセッサでは,『複雑』な動的命令スケジューリングは，バイナリ互換性のためではなく，高い性能を維持するために不可欠なものとなる．そのため，もし本当に動的命令スケジューリング・ロジックの『複雑さ』がプロセッサのクロック速度を制限することになれば，情報産業の発展を下支えしたマイクロプロセッサの性能向上は，いずれ頭打ちとなるであろう．

以下 本章では，このような流れについて述べる．まず1.1節では，LSIが微細化されるにつれて，スーパースカラ・プロセッサのような動的命令スケジューリングが重要になることを述べる．

1.1　LSIの微細化とマイクロアーキテクチャの変化

動的命令スケジューリングには，動的な事象に対応してIPCを高める正の効果がある．しかし90年代前半においては，動的命令スケジューリングの正の効果についての議論はほとんど見られなかった．ただしそれは，動的命令スケジューリングに正の効果があることが知られていなかった訳ではなく，過小に評価されていたのだと思われる．実際，1.2.2項で述べるように，当時のプロセッサではその効果はそれほど大きくはなかった．しかし1.2.2項で示すように，LSIの微細化にしたがい，その効果は徐々に増大していく．当時は，LSIの微細化によるマイクロプロセッサのデザインの変化を正しく予見できなかったのだと考えられる．

さて，ILPプロセッサを構成するユニットは，LSIの微細化に関連して，ALU，シフタ，アドレス計算機，浮動小数点加減乗除算器などの**演算器** (functional unit) と，それ以外のユニットとに大別することができる．以下では，演算器以外のユニットを**非演算器ユニット**と呼ぶことにする．なお1次データ・キャッシュは，実行ユニットであるロード / ストア・ユニットの一

部であるが，非演算器ユニットに含める．

演算器と非演算器ユニットに分けるのは，それらの遅延がLSIの微細化に対してそれぞれ異なる傾向を示すためである．LSIが微細化されると，非演算器ユニットの遅延は，演算器の遅延に対して相対的に増大するのである．非演算器ユニットの遅延の増大は，より細分化されたアーキテクチャの採用を促す．そして，アーキテクチャが細分化されるにつれ，動的命令スケジューリングの可否による性能の差は拡大することになる．

以下，1.1.1項でLSIの微細化が非演算器ユニットの遅延の相対的な増大を招く理由について，そして，1.1.2項と1.1.3項で，非演算器ユニットの遅延の増大がアーキテクチャの細分化を促す理由について述べる．

1.1.1 非演算器ユニットの遅延の増大

微細化によって，LSIの特性も変化する．それらの特性の変化は，演算器の遅延に対する非演算器ユニットの遅延の相対的な増大を招く．

ILPプロセッサの主要なユニット

まず，ILPプロセッサの主要なユニットを列挙し，非演算器ユニットにはどのようなものがあるのかを明らかにしよう．表1.1に，MIPS R10000プロセッサの命令パイプラインのステージ構成と，各ステージを構成する主要なユニットを示す[23]．同表に挙げたユニットのうち，◦を付したレジスタ・リネーミング・ロジックと命令ウィンドウはスーパースカラ・プロセッサに特

#	ステージ	ユニット
0	命令フェッチ	● 分岐予測器 ● 命令キャッシュ
1	レジスタ・リネーミング	◦ レジスタ・リネーミング・ロジック
	ディスパッチ	◦ 命令ウィンドウ
2	命令スケジューリング	◦ 命令スケジューリング・ロジック
3	発行	◦ 命令ウィンドウ
	レジスタ読み出し	● レジスタ・ファイル
4	実行（アドレス計算）	● 実行ユニット ● オペランド・バイパス
5	1次データ・キャッシュ・アクセス	● 1次データ・キャッシュ
6	ライトバック	● レジスタ・ファイル
	—	—

表1.1: MIPS R10000プロセッサの命令パイプライン構成と主要なユニット

有のものであるが，それ以外の，● を付したユニットは，VLIW プロセッサにも存在する．

非演算器ユニットの遅延の増大

微細化によって引き起こされる LSI の特性の変化のうち，以下の 2 点が特に ILP プロセッサのデザインに影響を与える：

1. トランジスタ数の増大
2. 配線遅延の増大

一方，演算器の遅延と非演算器ユニットの遅延には，以下のような違いがある：

a. **ウェイ数の影響**　演算器の遅延はウェイ数とは独立だが，非演算器ユニットの遅延はウェイ数の増加関数で与えられる．

b. **配線遅延の影響**　演算器の遅延は主にゲート遅延のみからなるが，非演算器ユニットの遅延は配線遅延を多く含む．

a. ウェイ数の影響 は，上記 1. トランジスタ数の増大 に関連して; b. 配線遅延の影響 は，同じく 2. 配線遅延の増大 に関連して，それぞれ非演算器ユニットの遅延を増大させることになる．以下，それぞれについて詳しく述べる．

1. トランジスタ数とウェイ数

最近までは，ウェイ数の増加を阻む要因は，ほとんどダイ上に集積可能なトランジスタの数だけであった．実際 商用のスーパースカラ・プロセッサは，LSI の微細化が進み，集積可能なトランジスタ数が増加すると，ウェイ数を 2 から 3，そして 4 へと増加させてきた．

ウェイ数の増加は，非演算器ユニットの遅延の相対的な増加を招く．個々の演算器それ自体の遅延は，当然のことながら，ウェイ数とは独立である．それに対して非演算器ユニットの遅延は，3 章以降で詳しく述べるように，ウェイ数の増加関数で与えられる．そのため，ウェイ数の増加に伴い，非演算器ユニットの遅延は，演算器の遅延に対して相対的に増大することになる．

ただしウェイ数の増加は，集積可能なトランジスタ数という LSI の性質だけではなく，ILP プロセッサがプログラムから抽出可能な ILP にによっても制限される．抽出可能な ILP を大きく越えるようなウェイ数を持つプロセッサを設計しても，IPC は向上しない．むやみに増加させれれば，非演算器ユニットの遅延の増大のため，かえって性能を悪化させることになる．実際 現在のスーパースカラ・プロセッサでは，トランジスタ数には余裕があるにも関わらず，ウェイ数は 4〜5 程度に留められている．トランジスタ数の増分は，ウェイ数の増加にではなく，オンチップ・キャッシュの増量などに充てられている．

このように，ウェイ数の増加には一定の歯止めがかかるから，非演算器ユニットの遅延の増加の要因としては，以下で述べる配線遅延の影響の方がより重要である．

2. 配線遅延の影響

ALU などの演算器の遅延は主にゲート遅延からなるのに対して，前述した非演算器ユニットの遅延は配線遅延を多く含む．命令 / データ・キャッシュ，レジスタ・ファイルなどを構成す

るRAMは，ビットライン，ワードラインなどの長い配線を持つ．また，3章で述べるように，スーパースカラ・プロセッサのレジスタ・リネーミング・ロジックや命令ウィンドウ・ロジックなどは，主にテーブル参照によって実現されるため，テーブルを構成するメモリへのアクセスが必要となる．オペランド・バイパスの遅延は，ほぼ完全に配線遅延のみからなる．

そのため，LSIの微細化によって配線遅延がゲート遅延に対して相対的に増大すると，専らゲート遅延からなる演算器の遅延に対して，配線遅延を多く含む非演算器ユニットの遅延が相対的に増大することになるのである．

ウェイ数の場合とは異なり，配線遅延の影響は，アーキテクチャの側から制御することはできない．アーキテクチャの側で非演算器ユニットの遅延の増大を招くような変更をしなくても，新しいプロセスに移行するだけで，配線遅延の影響は増大してしまう．アーキテクトの視点からは，非演算器ユニットの遅延は，『自動的に』増大していくように見えることになる．

後述するように，キャッシュなどの一部のユニットでは，既にその遅延がALUのそれより長くなってしまっている．もし何も対策を採らなければ，いずれこれらのユニットが**クリティカル** (critical) になる，すなわち，これらのユニットの遅延がシステム全体のクロック速度を制限することになる．2.5節で述べるように，配線遅延はほとんどスケーリングされないため，配線遅延に支配されるユニットの遅延が実際にクリティカルになれば，クロック速度もほとんどスケーリングされなくなってしまう．

このような傾向は，細分化されたアーキテクチャの採用を促す．あるユニットがクリティカルでなくなるように，そのユニットの1クロック・サイクルあたりの遅延を減らすには，つまるところ，サイクル数を増やすか，遅延を減らすしかない．サイクル数を増やすことは**パイプライン化** (pipelining) によって，遅延を減らすことは**非集中化** (decentralization) によって，それぞれ達成できる．どちらの技術も，プロセッサを多くのより小さなユニットへと細分化することになる．以下，1.1.2項と1.1.3項で，それぞれについて詳しく述べる．

1.1.2 パイプライン化

表1.1に示したユニットの多くは，**パイプライン化可能** (pipelinable) である．あるユニットをパイプライン化するとは，そのユニットにより多くのパイプライン・ステージを割り当てることを指す．ユニットをパイプライン化すれば，1クロック・サイクルあたりの遅延時間を大幅に削減でき，そのユニットをクリティカルでなくすることができる．

パイプライン化の実際

表1.1に示したユニットのうち，命令キャッシュやレジスタ・ファイル，スーパースカラ・プロセッサのレジスタ・リネーミング・ロジック，命令ウィンドウ・ロジックなど，すべての命令の実行に関わるユニットをパイプライン化することは，命令パイプラインを深化することと等価である．

前述したように，配線遅延に支配されるユニットの遅延が実際にクリティカルになれば，クロック速度もほとんどスケーリングされなくなってしまう．LSI の微細化に伴って高いクロック速度を得ようとするならば，命令パイプラインの深化はほとんど避けることができない．

実際，500nm 世代あたりから，命令パイプラインのステージ数は増え続けている．500nm 世代以前は，配線遅延の影響はほとんど無視することができた．当時主流であった RISC スカラ・プロセッサのほとんどは，1. 命令フェッチ，2. 命令デコード / レジスタ読み出し，3. 実行 / アドレス計算，4. 1次キャッシュ・アクセス，5. 書き戻し からなる，典型的な 5 ステージの命令パイプラインを採用していた．各ステージの遅延は，論理ゲートのステージ数によってほぼ決まっていたため，最小加工寸法が $1/S$ に縮小されると比例的に $1/S$ に縮小される．そのため，命令パイプラインの構成を変更することなく，クロック速度を S 倍にすることができた．しかし，500nm 世代あたりからは，次第に配線遅延の影響を無視することができなくなり，命令パイプラインのステージ数は以来徐々に増え続けている．

各ユニットの遅延がほぼ完全に配線遅延に支配された場合，最小加工寸法が $1/S$ に縮小されたとき，S 倍のクロック速度を得るためには，命令パイプラインのステージ数を S 倍にしなければならなくなる．

パイプライン化と投機

ただしパイプライン化は，何らかの**投機** (speculation) と組み合わせることが重要である．ユニットにより多くのパイプライン・ステージを割り当てると，その分だけそのユニットのレイテンシ（単位はサイクル）が増加することになり，それによる IPC の低下が避けられない．この IPC の低下がクロック速度による性能向上と相殺するようでは，パイプライン化を施した意味がない．投機には，このレイテンシの増加による IPC の低下を緩和する効果がある．

例えば，表 1.1 の，0〜4 のステージの 1 つまたは複数にパイプライン化を施した場合，それによるレイテンシの増加は分岐予測ミス・ペナルティ（単位はサイクル）の増加として現れることになる．この場合，性能に影響を与えるのは，予測ミス・ペナルティそのものではなく，予測ミス・ペナルティと予測ミスの発生頻度との積である．したがって，予測ミス率を抑え，予測ミスの発生頻度を十分小さくできれば，パイプライン化に伴う IPC の低下を小さく抑えることができる．

また，表 1.1 のステージ 3 にパイプライン化を施した場合には，次項で述べる，実行レイテンシ予測ミス・ペナルティが増加することになる．この場合も，予測ミスの発生頻度を十分小さくすることでパイプライン化に伴う IPC の低下を小さく抑えることができる．

換言すれば，様々な投機技術の進歩がパイプライン化を受け入れさせる要因となっていると言えよう．すなわち，非演算器ユニットのパイプライン化と投機技術の進歩は，2 つ同時に考える必要がある．

1.1.3 非集中化

1クロック・サイクルあたりの遅延を減らすもう1つの方法は，非集中化である．非集中化とは，集中化された (centralized) 単一のロジックで構成されていたユニットを，複数のサブユニットに分解することである．分解されたサブユニットの中には，機能を限定することで低遅延化したサブユニットを1個または複数個含む．割り当てられたサイクルに低遅延のサブユニットだけを動作させることで，1サイクルあたりの遅延を削減することができる．

キャッシュは，非集中化の最も代表的な例である．キャッシュ付きのメモリ・システムでは，実行ステージではキャッシュのみがアクセスされる．そのため1サイクルあたりの遅延は，主記憶の遅延からキャッシュの遅延にまで削減することができる．

非集中化には，水平方向のもの，垂直方向のもの，そして，その2つを組み合わせたものが考えられる．水平方向の非集中化とは，ユニットを複数の対等な小型のサブユニットに分解することを指す．一方，垂直方向の非集中化とは，ユニットを，普段用いられる小型で低遅延なサブユニットと，時々用いられる大型で高遅延なサブユニットに分解することを指す．

垂直方向の非集中化

垂直方向の非集中化は，水平方向のそれに比べて適用範囲が広く，特に階層化とも呼ばれる．前述したキャッシュは，垂直方向の非集中化にあたる．

垂直方向の非集中化の場合の，大型で高遅延なサブユニットとは，もともとの集中化されたユニットそのものであることが多く，小型で低遅延なサブユニットとは，それを簡略化したものであることが多い．また，大型で高遅延なサブユニットには，しばしばパイプライン化が施される．

キャッシュに限らず，小型で低遅延のサブユニットだけで処理できること / できないことを，しばしばヒット / ミスと呼ぶ．ミスの場合には一般に何らかのミス・ペナルティが発生するため，非集中化を施す際にはヒット率が十分に高いことが必要である．

非集中化の実例

前項で述べたパイプライン化が比較的 適用の対象を選ばないのに対して，非集中化は特に実行ユニット，および，その周辺に対して用いられる．非集中化には，前述したキャッシュの他，以下のような応用例がある：

命令ウィンドウの非集中化 スーパースカラ・プロセッサの命令ウィンドウは，整数，ロード / ストア，浮動小数点といった実行ユニットの系統ごとに，別個のサブウィンドウに分解することができる．命令ウィンドウの非集中化については，3.7 節で詳しく述べる．

命令ウィンドウの非集中化は水平方向の非集中化であり，以下の例はすべて垂直方向の非集中化である．

演算器の可変レイテンシ (variable latency) **化** [24, 25] 特に整数加減算命令のため，キャリーの伝搬する桁数が少ない場合に限り正しい結果を与える，より低遅延な演算器を用意する．通常の加減算器 (ALU) の動作を 2 サイクルにパイプライン化するとともに，この

演算器を 1サイクルで動作させることにより，1サイクルあたりの遅延を短縮することができる．

計算結果再利用 (value reuse) [25, 26, 27, 28, 29, 30, 31, 32] 以前の計算の結果を表に記録しておくことによって，再び計算することなく，表を読み出すことで結果を得る．この表は，最近 同一の計算を実行していた場合に限り正しい結果を与える，低遅延の演算器とみなすことができる．

実行ユニットのクラスタ化 (clustering) [33, 34] オペランド・バイパスは，多くの実行ユニットの出力と入力とを接続するバスであり，その配線長はほぼ実行ユニット数に比例する．そのため，実行ユニット数の増加とともに，配線遅延の影響を強く受け，スケーリングされにくくなる．

実行ユニットのクラスタ化は，DEC* Alpha 21264 プロセッサで採用された，オペランド・バイパスの遅延を短縮する技術である．実行ユニットのクラスタ化では，実行ユニットを**クラスタ** (cluster) と呼ばれるいくつかのグループに分割し，クラスタ間にまたがるオペランド・バイパスを省略する．依存する 2 つの命令が異なるクラスタに割り当てられた場合には，オペランド・バイパスが利用できないため，引き続くサイクルに実行することができなくなる．一方でオペランド・バイパス自体は，各クラスタ内の実行ユニット間のみを接続すればよいので，その配線長はおおよそ 1/（クラスタ数）に短縮される．2.5 節で述べるように，配線遅延は配線長の 2 乗に比例するため，2 つのクラスタに分割するだけでも，大幅な遅延の短縮が可能となる．

クラスタ化自体は，マルチプロセッサの分野では既にありふれた技術であり，配線遅延の影響の増大とともにプロセッサの内部にまで浸透してきたのだと考えられる．

なお，クラスタ化は，複数のクラスタへの水平方向の非集中化と考えてもよいし，オペランド・ネットワークのクラスタ内とクラスタ間へと垂直方向の非集中化と考えてもよい．

キャッシュの多階層化 キャッシュは，チップ上で最も大容量の RAM であり，配線遅延の影響を最も受けやすいユニットである．近年では，実際に 1次データ・キャッシュがクリティカルになりつつあり，更なる多階層化が検討されている．

上述したように，パイプライン化が比較的 適用の対象を選ばないのに対して，実行ユニットに対しては非集中化の適用が重要である．プログラムの実行時間の下限を与えるのは，プログラムのクリティカル・パス上にある命令が実行される実行ユニットの遅延の総和である．そのため，パイプライン化によって 1 サイクルあたりの実行ユニットの遅延を短縮して，クロック速度が向上したとしても，実行時間の短縮にはつながらない．

さて，上述した非集中化の例の中では，キャッシュの多階層化が後の議論で特に重要になるので，以下で詳しく説明する．

---

* 発表当時.

キャッシュの多階層化

キャッシュは，チップの面積の大きな部分を占める，チップ上で最も大容量のRAMであり，そのワードライン，ビットラインは裸眼で見えるほど長い．そのためキャッシュは，配線遅延の影響を最も受けやすいユニットであり，近年では1次データ・キャッシュが実際にクリティカルになっている．

1次データ・キャッシュの容量は，オンチップ化されて以降，16KB程度のまま，ほとんど変化していない*．にもかかわらず，180nm世代では，とうとう1サイクルでアクセスすることが困難になったてきた．

そのため，まずデータ・キャッシュ・アクセスに対してパイプライン化が施された．過去何十年にも渡って，ロード/ストア命令はアドレス計算と1次キャッシュ・アクセスの2ステージ構成であった．しかし，180nmプロセスで製造されるIntel Pentium 4プロセッサ[35]やAMD Athlonプロセッサでは，アドレス計算に1ステージ，1次キャッシュ・アクセスに2ステージの3ステージ構成となっている†．

しかし，1次キャッシュのアクセス・レイテンシが3サイクルともなると，それによるIPCの低下は受け入れがたいものとなる．そこで，Williamsらはライン・バッファを提案している[36, 37, 38, 39]．ライン・バッファは，ロード/ストア・ユニットごとに用意された小容量のバッファで，1次キャッシュと同時にアクセスされる．その容量を十分に小さく抑えることにより，1サイクルでアクセスできるように設計する．65nm世代以降のプロセスにおいて，1サイクルでのアクセスを可能とするためには，その容量は1KB程度以下にする必要がある．後述するようにそのミス率は，小容量のため，最大30%程度にものぼる．それでもライン・バッファは，1次キャッシュのアクセス・レイテンシが3サイクルを越えるようなスーパースカラ・プロセッサに対して，数割程度の性能向上をもたらす．

ライン・バッファ自体には，そのままの形で採用するにはいくつかの問題がある．しかし，Williamsらの功績は，1KB程度以下のごく小容量のキャッシュであっても，相応の参照の局所性を抽出できることを示した点にある．65nm世代以降のプロセッサには，1KB程度以下の0次キャッシュが有効であることは間違いない．

上述したように，1KB程度以下の0次キャッシュは，数割程度の性能向上をもたらすものの，そのミス率が最大30%程度にもなる．これは，1次キャッシュのミス率が数%程度であるのと比べると1桁近くも大きな値である．次節で述べるように，この0次キャッシュのミス率の高さは，動的命令スケジューリングの重要性に大きな影響を与えることになる．

非集中化と投機的命令スケジューリング

キャッシュや可変レイテンシの演算器など，実行ユニットに対して非集中化を施した場合には，その実行ユニットのレイテンシがヒット/ミスに応じて変わることになる．

このような可変レイテンシの実行ユニットに対しては，そのレイテンシを予測して，投機的に後続の命令をスケジューリングすることが不可欠である．先行命令のヒット/ミスが判明し

---

* 本稿でも用いている，最も標準的なベンチマーク集であった，SPEC'95ベンチマークのワーキング・セットがおよそ16KBであるためという説がある．

† ただし，総合的な性能よりもクロック速度が高い方が商売上有利であるとの判断もあるかも知れない．

た後で後続の命令をスケジューリングしたのでは間に合わないからである．先行命令がヒットと分かった後で後続の命令のスケジューリングを開始したのでは，後続命令が実行されるのは先行命令が実行されてから何サイクルか後になる．これでは，先行命令の実効的なレイテンシがそれだけ増加してしまい，非集中化を施した意味がない．最も単純な場合でも，すべてヒットと想定して後続の命令をスケジューリングしなければならない．このことは，すべての命令がヒットすると静的に予測したと考えることができる．

このような静的な予測に基づく投機的な命令スケジューリングは，既に普通に行われている．例えばロード命令は，すべてキャッシュにヒットすると予測して後続命令を投機的にスケジューリングすることが普通である．

ただし，ヒットと予測して後続の命令を投機的にスケジューリングすると，実際にはミスであった場合に，後続命令は先行命令から正しい結果を得られないことになる．その場合，後続命令の実行をキャンセルして，再スケジューリングする必要が生じる．

この場合のミス・ペナルティは，命令がスケジューリングされてから実際に実行されるまでのレイテンシで与えられる（3.2.4 節）．表 1.1 では，ステージ 3 の 1 サイクルがこのミス・ペナルティにあたる．前項で述べたパイプライン化をこの部分に対しても施すと，分岐予測に加えて，この投機的スケジューリングのミス・ペナルティも増加することになる．

この投機的スケジューリング・ミスのコストを低減するため，分岐予測で培われた技術を応用した，実行履歴に基づくより高度な実行レイテンシ予測技術も採用されている [33, 38, 39].

1.1.4 パイプライン化，非集中化と投機

前々項と前項で述べたように，パイプライン化と非集中化は，投機と組み合わせて用いることが重要である．近年では，以下のようなさまざまな投機技術が開発されている．

パイプライン化と条件分岐

分岐予測をはじめとする以下の技術は，制御依存による先行制約を解消し，パイプライン化に伴うレイテンシの増大の影響を緩和する効果がある：

分岐予測 (branch prediction) [40] 条件分岐命令の結果が判明する以前に，その下流の一方のパス上の命令を先行実行する．

分岐予測の予測ヒット率は非常に高いが，それでも 90% を割るプログラムもある．そのようなプログラムに対しては，以下のような技術によって分岐予測を補完することができる．

複数パス実行 (multi-path execution) 条件分岐命令の結果が判明する以前に，その下流の両方のパス上の命令を先行実行する．複数パス実行は，主にスーパースカラ・プロセッサ向けに研究されている．

プレディケーション (predication) [17, 19] プレディケーションは，主に VLIW プロセッサ向けに開発された技術である．指定されたプレディケート（predicate，述語）が真の場合にのみ

マシン状態を更新する，プレディケート付き命令を命令セット・アーキテクチャに導入する．コンパイラは，if (P) then $\cdots$ else $\cdots$ という構造に対して，then 部の各命令にプレディケート P を，else 部の各命令にプレディケート $\overline{P}$ を付し，条件分岐を省略して then 部 / else 部両方の命令が実行されるようにコードを生成する．このような操作は，IF 変換 (IF-conversion) と呼ばれる．プロセッサは，then 部 / else 部，両方の命令を投機的に実行しておき，P の値が決まった後，その値に基づいて一方の結果を破棄すればよい．

非集中化と実行レイテンシ予測

前項で述べたように，非集中化に伴う実行ユニットの可変レイテンシ化に対しては，以下のような予測が行われる：

実行レイテンシ予測 (execution latency prediction) [24, 25, 41]　低遅延の演算器のみで処理可能かどうかを予測する．予測には，主に分岐予測のために開発された，各命令の実行履歴 (history) に基づいた技術が応用可能である．

キャッシュ・ヒット / ミス予測 (cache hit/miss prediction) [33, 38, 39]　ロード命令の実行レイテンシを予測することは，キャッシュ・ヒット / ミスを予測することと等価である．

その他の投機技術

また近年では，更なる IPC の向上を目指して，データ依存による先行制約を解消する以下のような技術が多数提案されている：

値予測 (value prediction) [42, 43]　通常 実行履歴に基づいて，ほとんどすべての命令の結果を予測する．データ依存関係にある後続の命令を，先行命令の実行を待つことなく実行することができる．

アドレス予測 (address prediction) [44]　そのアドレスを予測することにより，ロード命令を先行実行する．

アドレス一致 / 不一致予測 (address match prediction) [44, 21]　先行するストア命令と後続のロード命令のアドレスの一致 / 不一致を予測し，不一致と予測した場合に後続のロード命令を先行実行する．

なお，アドレスが不明であることによる偽のデータ依存を解消する技術は，メモリ曖昧性解消 (memory disambiguation) と呼ばれる．

本節のまとめ

本節の内容は，以下のようにまとめられる：

- LSI の微細化に伴って，配線遅延がゲート遅延に対して相対的に増大すると，非演算器ユニットの遅延が演算器の遅延に対して相対的に増大し，配線遅延を多く含む非演算器ユニットがクリティカルになる可能性が高くなる．配線遅延はほとんどスケーリングされ

ないため，配線遅延に支配されている非演算器ユニットがクリティカルになると，LSI の微細化に見合ったクロック速度の向上が得られなくなる．

- 非演算器ユニットがクリティカルになることは，1. パイプライン化 と 2. 非集中化 によって回避できる．どちらの技術も，プロセッサを多くのより小さなユニットへと細分化することになる．
- 1. パイプライン化 と 2. 非集中化 に対しては，以下のように，投機技術が本質的な役割を果たす:
  1. パイプライン化に対しては，分岐予測や実行レイテンシ予測によって IPC の低下を緩和することが重要である．
  2. 実行ユニットに対して非集中化を施した場合には，実行レイテンシ予測を組み合わせなければ意味がない．
- 1. パイプライン化 や 2. 非集中化 のうち，0 次キャッシュが特に動的命令スケジューリングの重要性に影響を与える．
- 近年では，更なる IPC の向上を目指して，データ依存による先行制約を解消する投機技術が多数提案されている．

次節では，投機技術が本質的な役割を果たす細分化されたアーキテクチャにおいて，動的命令スケジューリングが重要になる理由について述べる．

1.2 投機と Out-of-Order 命令スケジューリング

前節で述べたように，非演算器ユニットがクリティカルになることを回避するために 1. パイプライン化 と 2. 非集中化 を施して細分化されたアーキテクチャでは，投機技術が本質的な役割を果たす．このような投機技術の採用は，動的命令スケジューリングの重要性に強い影響を及ぼすことになる．本節では，投機と命令スケジューリングの関係について論ずる．

以下，1.2.1 項では，投機と命令スケジューリングの関係について定性的な議論を行う．1.2.2 項では，非集中化の例として 0 次キャッシュを採用した ILP プロセッサにおける動的命令スケジューリングの可否が IPC に与える影響の評価結果を示す．

1.2.1 投機と命令スケジューリング

前節で挙げた投機技術のうち，プレディケーション以外のすべての技術では，対象となる命令は通常 2 つの**事象**を生起し，この 2 つの事象のどちらが実際に生起するかを選択することになる．前述した各技術の事象を表 1.2 にまとめる．なおここで言う事象とは，予測のヒット / ミスとは異なることに注意されたい．事前に予測した事象と実際に起こった事象が同じ場合が予測ヒット，そうでない場合が予測ミスである．

これらの技術では，それぞれの事象によって，命令そのもの，あるいは，命令の実効的なレイテンシが互いに異なる．その結果，最適な命令スケジューリングの結果も，予測の結果によって動的に変化することになる．

プレディケーション以外のほとんどの技術は，もともとスーパースカラ・プロセッサ向けに開発されたものである．スーパースカラ・プロセッサの場合，動的命令スケジューリングによって，この命令やレイテンシの動的な変化に自然に対応することができる．一方，動的命令スケジューリングを行わない VLIW プロセッサには，命令やレイテンシが動的に変化するこれらの技術のすべてが応用可能な訳ではない．スーパースカラ・プロセッサの場合，これらの投機技術が有効であるためには，予測ヒット率が十分に高ければよい．一方 VLIW プロセッサの場合には，予測ヒット率がいくら高くてもうまくいかない場合がある．

これらの投機技術を VLIW プロセッサに応用するには，1. 静的予測 と 2. プレディケーション の 2 つの方法がある：

1. **静的予測**　事象の生起確率に大きな偏りがある場合には，コード生成時に静的に予測を行うことにより，これらの投機技術を VLIW プロセッサにも応用することができる．事象の生起確率に大きな偏りがあれば，生起確率の高い事象のみを考慮して最適化を施す一方で，もう一方の事象を例外的に扱えばよい．もう一方の事象の扱い方には，以下のようなものがある：

   - 条件分岐の分岐の方向に大きな偏りがある場合，トレース・スケジューリング系の最適化が効果的であることはよく知られている [17]．すなわち，実行される確率が高いトレースに着目して最適化を施す一方で，それ以外のパスでその辻褄を合わせる (book-keeping) ようにすればよい．
   - アドレス一致 / 不一致予測では，不一致である確率が非常に高いと静的に予測可能な場合が多い．その場合，不一致であると静的に予測して最適化を施す一方で，アドレスの一致を検出して後続のロード以降の命令を再実行できるしくみを用意すればよい [19, 20]．
   - キャッシュ・ヒット / ミス予測では，キャッシュ・ヒットと静的に予測して最適化を施す一

技術	事象 1	事象 2
分岐予測	条件分岐が taken	not taken
複数パス実行	複数パス実行を行う	行わない
実行レイテンシ予測	低遅延の演算器で処理可能	不能
キャッシュ・ヒット / ミス予測	キャッシュ・ヒット	ミス
値予測，アドレス予測	予測器が正しい結果を出力する	しない
計算結果再利用	再利用可能	不能
アドレス一致 / 不一致予測	アドレスが不一致	一致

表 1.2: 各投機技術の 2 つの事象

方で，ミスであった場合には，ミス処理の終了までパイプラインをストールすればよい．

ただし，後述するように，この手法が提供できるのは 1.1.3 で述べたような 1KB 程度の 0 次キャッシュ を採用**しない**場合に限られる．

2. **プレディケーション**　条件分岐の場合には，事象の生起確率に大きな偏りがなくても，すなわち，分岐の方向に大きな偏りがなくても，プレディケーションによって対応可能である．プレディケーションでは，2 つの事象に対応する命令が 1 つのコードにまとめられており，動的に変化する要素がないため，動的命令スケジューリングを行わない VLIW プロセッサでも対応可能になっている．

しかし上述以外の場合には，以下に示すように，上述の方法を用いて VLIW プロセッサに応用することは困難である．なお上述以外の場合とは，具体的には，実行レイテンシ予測，0 次キャッシュに対するヒット / ミス予測，値予測，アドレス予測などである：

1. **静的予測**　上述以外の場合でも，表 1.2 における事象 1 の生起確率はそこそこ高いので，事象 1 が生起すると静的に予測して最適化を施すことは可能である．

   しかし上述以外の場合，事象の生起確率に大きな偏りがあるとは言えないため，静的に事象 1 を予測すると，上述したような高い予測ヒット率は得られない．例えば，キャッシュ・ヒット / ミス予測では，16KB 程度の 1 次キャッシュに対するミス率は 1% 程度以下と低く，ミスの度にパイプラインをストールさせても IPC は大きく低下しないと期待できる．逆に，1KB 程度の 0 次キャッシュ を採用した場合には，そのミス率は最大 40% 程度になり，ミスの度にパイプラインをストールさせたのでは IPC の低下が大きくなり過ぎる可能性がある．なお次項では，この場合の IPC の低下を定量的に評価する．

2. **プレディケーション**　条件分岐**以外**の場合には，プレディケーションと類似の方法によって VLIW プロセッサに応用することは不可能である．条件分岐の場合には，事象によって変わるのは命令そのものであるのに対して；それ以外の場合には，事象によって命令の実効的なレイテンシが変わるためである（囲み：プレディケーションと可変レイテンシの命令）．

結局 動的命令スケジューリングを行わない VLIW プロセッサでは，事象の生起確率に大きな偏りがない場合にも，IPC が低下するのを承知のうえで，静的予測に頼らざるを得ない．次項では，0 次キャッシュを例に，その IPC の低下を定量的に評価した結果を示す．

1.2.2　0 次キャッシュと動的命令スケジューリングの効果

本項では，0 次キャッシュを例に，事象の生起確率に大きな偏りがない投機技術を採用した場合に，動的命令スケジューリングの可否によってどの程度の性能差が生じるのかを評価した結果を示す．

1.1.3 項で述べたような 1KB 程度以下の 0 次キャッシュを装備した場合には，後述するように，そのミス率は最大 40% 程度にもなる．16KB 程度の容量を持つ従来の 1 次キャッシュ

に比べ，キャッシュ・ミスの発生頻度は1桁程度も増加することになる．

動的命令スケジューリングを行うスーパースカラ・プロセッサでは，ロード命令がキャッシュ・ミスを起こしたときも，そのロード命令に依存しない命令を発見し，実行することができる．一方，動的に命令をスケジューリングしないVLIWプロセッサでは，キャッシュ・ミスの処理が終了するまで，後続の命令を実行することができない．キャッシュ・ミスの度に発生するパイプライン・ストールは，ミス頻度の上昇とともに増加することになる．

0次キャッシュを持たない従来のILPプロセッサでは，1次キャッシュのミス・ペナルティ（2次キャッシュのレイテンシ）は，4〜6サイクル程度もある．たしかに動的命令スケジューリングには，それほどのレイテンシを完全に隠蔽する (hide) 能力はないかも知れない．しかし，0次キャッシュのミス・ペナルティ（1次キャッシュのレイテンシ）は，たかだか2〜3サイクルに過ぎ

プレディケーションと可変レイテンシの命令

ある可変レイテンシの命令 I があり，その実効的なレイテンシは，事象1より事象2が n サイクル ($n > 1$) 長いとしよう．すると，命令 I に依存するすべての命令の最適な位置も，事象1と事象2で n サイクルだけ異なることになる．そのため，プレディケートによって事象1と事象2を1つのコードにまとめるには，以下のような不都合が生じる：

- I に依存する**すべての**命令を二重化し，それぞれ n サイクルだけずらして配さなければならない．二重化を止め，レイテンシの長い方に合わせると，その時点で可変レイテンシ化した効果自体が完全に失われる．
- 予測をミスした事象に対応する命令はすべて，プレディケートによって実行結果が破棄される．すなわち，二重化された命令の半分は，無駄な計算を行うことになる．
- ループなどの繰り返し構造に対しては，このような二重化されたコードを生成する手段すら明らかでない．
- 通常は，I に依存する命令の中にも別の可変レイテンシの命令があり，問題は更に複雑になる．

このように，どの一点をとっても十分に非現実的であり，プレディケーションによって可変レイテンシの命令に対処することは不可能であると結論づけられる．

なお，通常の条件分岐に対するプレディケーションの場合に，このような不都合が生じないのは以下の理由による：条件分岐に対するプレディケーションでは，通常，then部/else部より下流の命令は，then部/else部のどちらかレイテンシの長い方に合わせてスケジューリングされる．そのため短い方が正しかった場合には長さの差の分だけ無駄なサイクルが生じる．しかし，それによってプレディケーションの効果のすべてが失われる訳ではない．then部とelse部で —— 事象1と事象2で命令自体が異なるから，二重化したことにはならないからである．

ないから，そのレイテンシのかなりの部分を動的命令スケジューリングによって隠蔽できると期待できる．

評価方法

SimpleScalar ツールセット [45, 46] (ver. 2.0) の sim-outorder シミュレータに 0 次キャッシュを付加し，表 6.1 (p. 129) に示す SPEC ベンチマーク [47] CINT95 の 8 つのプログラムの実行時間を計測した．コンパイラは，gcc (ver. 2.7.2.3) を用いた．最適化オプションは，-O6 -funroll-loops である．gcc は，大域的な命令スケジューリングは行わないが，命令のレイテンシを考慮した局所的なスケジューリングに関しては，ほぼ満足いく性能を持っている．

評価モデル

ベース・モデルは，基本的には，MIPS R10000 プロセッサ [23] に準ずる．R10000 プロセッサは，整数演算，ロード / ストア，浮動小数点演算のそれぞれを毎サイクル 2 命令ずつ実行できるスーパースカラ・プロセッサである．

0 次キャッシュのある / なしに関して，以下の 2 つのオプションを設定した：

L1 180nm 世代を想定したモデル．0 次キャッシュは持たず，1 次キャッシュのレイテンシは 2 サイクルとしている．

L0 65nm 世代以降を想定したモデル．1 次キャッシュのレイテンシを 4 サイクルとし，1 サイクルでアクセス可能な 1KB の 0 次キャッシュを備える．

0 次，および，1 次キャッシュ以外のパラメタは，両モデルで同一とした．表 1.3 に，キャッシュ，メモリのパラメタをまとめる．本来であれば，モデル L0 の 2 次キャッシュ，および，メモリは，1 次キャッシュと同様，モデル L1 のそれより低速にすべきであるが，後述するように，それらの速度差の影響はごくわずかであり，ほとんど無視できる．

動的命令スケジューリングに関しては，以下の 2 つのモデルを設定した：

SS ベース・モデルである，MIPS R10000 プロセッサに準じたスーパースカラ・プロセッサ．

VLIW 前項で述べた静的予測により，最上位キャッシュがすべてヒットするとして最適化を施したコードを実行する VLIW プロセッサ．キャッシュ・ミス時には，ミス処理の完了までストールする．

	容量	ライン サイズ	ウェイ数	レイテンシ (cycles)
0 次キャッシュ	1KB	64B	1	1
1 次キャッシュ	64KB	↑	2	2/4
2 次キャッシュ	1MB	↑	8	10
メモリ	—	—	—	32

表 1.3: キャッシュ，メモリのパラメタ

モデル VLIW の実行時間は，モデル SS と同じスーパースカラ・プロセッサのシミュレータを用いて，以下のように求めた：シミュレータでキャッシュ・ミスのペナルティをすべて 0 サイクルとすることにより，ストールしていない時間を測定する．こうすると，シミュレータの動的命令スケジューラがスケジューリングした結果は，最上位キャッシュがすべてヒットすると静的に予測して最適化を施した結果とほぼ同じになる．それと同時に，各階層ごとのキャッシュ・ミス回数を記録しておき，それらに各階層のミス・ペナルティをそれぞれ乗ずることにより，ストールしている時間を求める．

以上，2×2 = 4 つのモデルが考えられ，それぞれを，L1-SS，L1-VLIW，L0-SS，L0-VLIW と呼ぶことにする．

評価結果

図 1.1 に結果を示す．グラフには，2 本 × 2 対 × 8 組のバーがある．各組は CINT95 の 8 つのプログラムに対応する．各組は，2 本 × 2 対のバーからなり，左の対はモデル L1，右の対はモデル L0 のものである．各対は，左 / 右 2 本のバーからなり，左はモデル SS，右はモデ

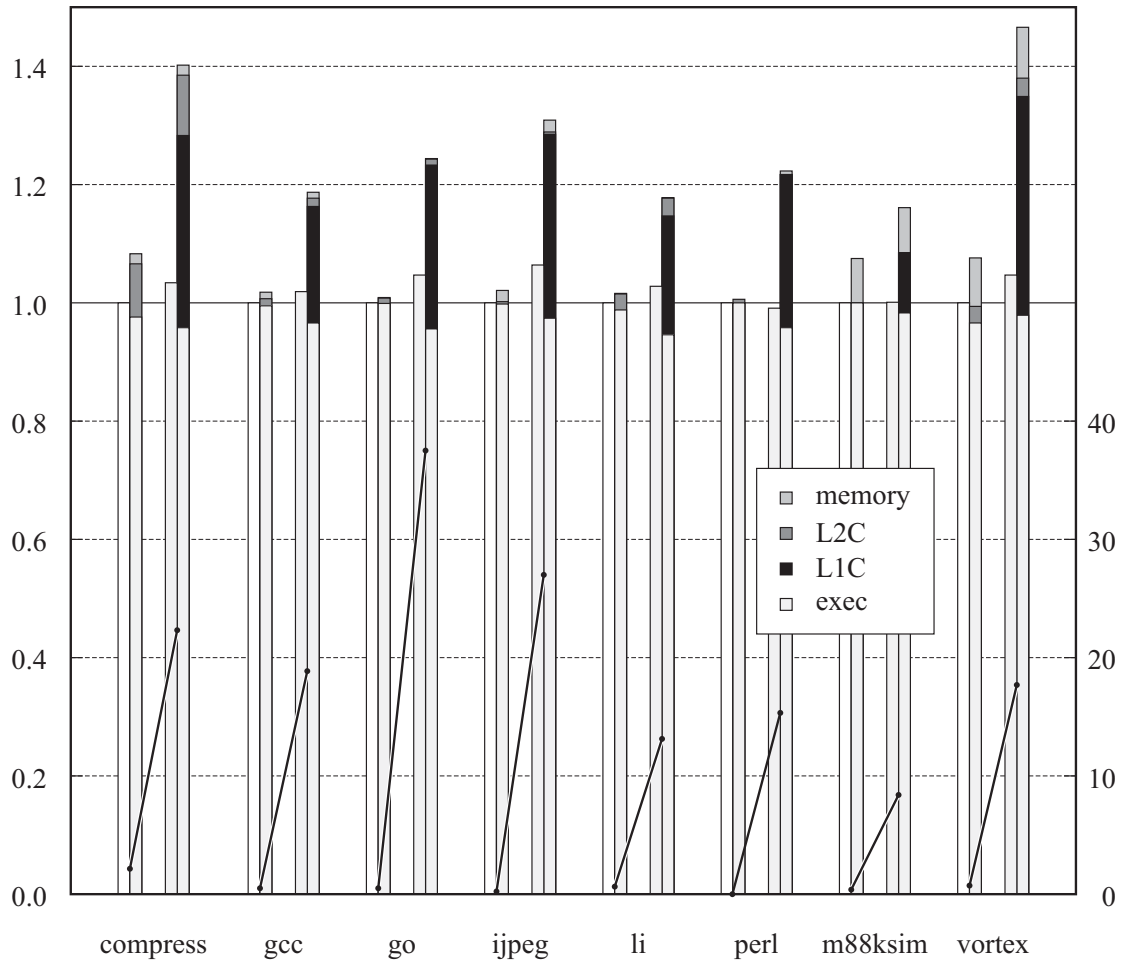


図 1.1: CINT95 の実行時間と最上位キャッシュのミス率

ル VLIW の実行時間（単位はサイクル）を表す．各組のバーは，その組のモデル L1-SS の実行時間を 1 として正規化してある．モデル VLIW のバーは，3 ないし 4 つの部分からなるが，上から順に，メモリ，2 次キャッシュ，1 次キャッシュからのラインの到着を待ってストールしている時間を表し，一番下はそれ以外の時間を表す．なお，0 次キャッシュを持たないモデル L1-VLIW のバーには，1 次キャッシュを待ってストールする時間はない．

また，同図下部には，最上位キャッシュ（L1 では 1 次，L0 では 0 次）のミス率（単位は %）を，折れ線で示した．

グラフから，180nm 世代を想定したモデル L1（各組，左の 2 本）では，L1-SS/L1-VLIW の実行時間の差は 1 割程度以下しかなく，動的命令スケジューリングによる性能向上はそれほど大きくないことが分かる*．

しかし，65nm 世代以降を想定したモデル L0（各組，右の 2 本）では，動的命令スケジューリングの可否による性能差は最大 40% 程度にも達する．モデル L0-VLIW では，特に，1 次キャッシュを待ってストールしている時間，すなわち，0 次キャッシュ・ミスの処理時間が多い．結局，0 次キャッシュを搭載する 65nm 世代以降の ILP プロセッサでは，0 次キャッシュのミス率が高いため，静的予測では対処しきれていないと言える．

逆に言えば，モデル L0-SS では，動的命令スケジューリングによって 0 次キャッシュ・ミスの処理時間の多くの部分が隠蔽できていることが分かる．モデル L1 からモデル L0 へ，1 次キャッシュのレイテンシは 2 サイクルから 4 サイクルへと増加しているにも関わらず，モデル SS の実行時間は大きく増加していない．プログラムによっては，逆に減少してるものもある．動的命令スケジューリングは，LSI の微細化に伴う IPC の低下を最小限に抑える強力な武器になっていると言える．

1.3 本稿の内容

前節で述べたように，LSI の微細化が進むにつれ，動的命令スケジューリングの重要性が高まる．しかし，動的命令スケジューリングのためのロジックが本当にクロック速度向上の足枷となるのなら，その効果は相殺されてしまうかも知れない．逆に言えば，もしクロック速度を制限しないような動的命令スケジューリングの方式が開発できれば，LSI の微細化に伴うアーキテクチャ上の問題を解決できることになる．

本稿では，実際にそのような技術について述べる．研究成果は，スーパースカラ・プロセッサの動的命令スケジューリング・ロジックが，クロック速度に影響を与えないことを示す．

まず，**Dualflow** と呼ぶ命令セット・アーキテクチャについて述べる [1, 2, 3, 4, 5]．Dualflow アーキテクチャは，制御駆動とデータ駆動の両方の性質をあわせ持つ命令セット・アーキテクチャである．制御駆動型アーキテクチャと同様のプログラム・オーダを定義するが，制御駆動型アーキテクチャようなレジスタを定義しない．命令間のデータの受け渡しは，制御駆動型アーキテクチャのようにレジスタを介して間接的に行われるのではなく，データ駆動型

* 現在のマイクロプロセッサ研究の水準からすれば，1 割の性能向上は十分に大きい．

アーキテクチャのように定義側の命令が使用側の命令を直接的に指定することで行われる．Dualflow アーキテクチャでは，命令間の依存関係を表す行列を用いて命令スケジューリングを実現することになる．その結果，スーパースカラ・プロセッサと同様の動的命令スケジューリングを行いながら，小型の RAM 1個でロジックを構成することができる．

その後，Dualflow アーキテクチャのために考案された依存行列に基づく命令スケジューリング・ロジックは，通常の制御駆動型アーキテクチャを持つスーパースカラ・プロセッサにも適用可能であることが分かった [6, 7, 8, 9, 10, 11, 12, 13]．その上，制御駆動型アーキテクチャの特徴を活かした高速化技術がいくつか考案された．その結果，4 word × 4 bit，1-read × *IW*-write という，小容量の RAM を読み出す処理によって動的命令スケジューリングを実現することができる．

次章以降の本稿の内容は以下のとおりである：

2章 CMOS 回路の基礎 3 章以降で述べる動的命令スケジューリングのためのロジックは，プロセッサの中でも最もクリティカルな部分の 1 つであり，ダイナミック・プリチャージ回路をはじめとする，やや技巧的な回路技術が利用されている．それらの回路に対する理解なくしては，3 章以降の議論はほとんど成り立たない．本章では，それらの CMOS 回路の基礎についてまとめる．また，LSI のスケーリングについてまとめ，配線遅延がほとんどスケーリングされない理由について説明する．

3章 Out-of-Order 命令スケジューリング プログラムに記述された順序とは関係なく命令を実行できる out-of-order な動的命令スケジューリングの原理について説明する．同時に，従来の out-of-order 命令スケジューリング・ロジックの一般的な実装法である連想方式を紹介する．連想方式は，RAM を読み出した結果を CAM によって連想検索するという構造を持つことを示し，これらのメモリの遅延が LSI の微細化に伴ってクリティカルになる理由を説明する．

4章 間接方式 DEC* Alpha 21264 プロセッサで採用されている out-of-order 命令スケジューリング・ ロジックについて述べる [48, 41, 49]．間接方式は，5 章で述べる dualflow アーキテクチャや 6 章で述べる直接方式と同様に，依存行列を用いた方式であるが，従来の連想方式との連続性が高い．そのため，間接方式について知っておくことは，後述する dualflow アーキテクチャや直接方式の理解の助けとなるだろう．

5章 Dualflow アーキテクチャ Dualflow アーキテクチャは，out-of-order 命令スケジューリング・ロジックを簡単化することを目的として開発された out-of-order ILP プロセッサ向けの命令セット・アーキテクチャであり，その実行モデルを素直に実装することにより，高速なロジックを得ることができる．

6章 直接方式 制御駆動型の命令セット・アーキテクチャに準拠する通常のスーパースカラ・プロセッサの out-of-order 命令スケジューリング・ロジックを高速化する直接方式について述べる．直接方式は，5 章で述べた dualflow アーキテクチャの命令スケジューリング・

* 発表当時.

ロジックを通常のスーパースカラ・プロセッサに応用したものであるが，制御駆動型アーキテクチャの性質を利用して，更なる高速化を達成することができる．

7 章　回路の評価　6 章までで述べた各方式の回路の評価は，7 章でまとめて行う．富士通株式会社から提供された実在する CMOS プロセスのデザイン・データを用いて，連想方式，間接方式，直接方式それぞれのロジックの回路面積と回路遅延の評価を行う．

8 章　結論　本稿の内容をまとめる．

第2章　CMOS回路の基礎

Out-of-order 命令スケジューリングのためのロジックは，プロセッサの中でも性能上最もクリティカルな部分の 1つである．そのため，ダイナミック・プリチャージ回路をはじめとする，やや技巧的な回路技術が利用されている．それらの回路に関する理解なくしては，次章以降で述べる out-of-order 命令スケジューリング・ロジックに関する議論はほとんど成立しない．

本章では，それらの CMOS 回路の基礎についてまとめる．以下，まず，2.1 節では，ダイナミック・プリチャージ回路についてまとめる．2.2 節では，ダイナミック・プリチャージ回路を用いた一致比較器について述べる．一致比較器は，2.4 節で述べる CAM で用いられる他，ダイナミック・プリチャージ回路の利用によって著しく高速化される好例ともなっている．Out-of-order 命令スケジューリングのほとんどは，各種のテーブルの読み書きによって実現される．2.3 節と 2.4 節では，それらのテーブルを構成する RAM と CAM について述べる．最後に，2.5 節では，CMOS のスケーリングについてまとめる．

2.1　ダイナミック論理

ダイナミック・プリチャージ回路 (dynamic precharged circuit) は，高性能な CMOS LSI において重要な役割を果たしている．

2.1.1　ダイナミック・ゲート

図 2.1 に，ダイナミック・ゲートの例を示す．同図は，2 段のダイナミック・ゲートを示している．まず，図中央のノード *y* より左の，1段目のゲートに注目しよう．

ダイナミック・ゲートは，図では *x* と記された，*p*MOS トランジスタと *n*MOS トランジスタに挟まれた**プリチャージ・ノード**にプリチャージされた電荷がディスチャージされるか否かによって出力が決定される．プリチャージ・ノード *x* の電源側にある *p*MOS トランジスタは，**プリチャージ・トランジスタ**と呼ばれ，プリチャージ・ノードに電荷をプリチャージする役割を担う．プリチャージ・ノード *x* のグラウンド側にある *n*MOS トランジスタのうち，最も電源に近いものは**フット** (foot) と呼ばれることがある．プリチャージ・ノード *x* とフットの間にある *n*MOS トランジスタが，このダイナミック・ゲート全体の論理を決定する．図に示すように，プリチャージ・ノード *x* と出力の間には，通常，インバータの出力バッファが配される．また，**キーパー** (keeper) と呼ばれる *p*MOS トランジスタが付加されることがある．

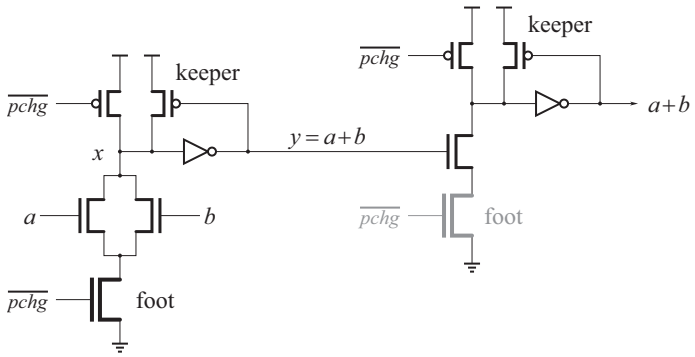


図 2.1: ドミノ論理

ダイナミック・ゲートは，**プリチャージ期間**と**評価期間**の 2 つの期間の繰り返すことで動作する．すなわち，以下のように，プリチャージ期間でプリチャージ・ノード x にプリチャージされた電荷が，評価期間でディスチャージされるか否かによって出力が決定される：

プリチャージ期間 プリチャージ期間には，プリチャージ・クロック *pchg* が high となり，プリチャージ・トランジスタは ON，フットは OFF になるため，入力 a，b の値によらず，プリチャージ・ノード x は high にプリチャージされる．インバータの出力バッファにより，プリチャージ期間中のゲートの出力 y は low となる．

評価期間 プリチャージ期間が終わると，プリチャージ・クロック *pchg* が low となる．その後の入力の値により，ノード x に蓄えられた電荷がディスチャージされるか否かが決定される．同図 2.1 の場合，a，b のどちらか一方が high であれば，ディスチャージが行われ，インバータの出力バッファにより，出力 y は high にドライブされる．逆に，a，b 共に low であれば，プリチャージ・ノード x は high に保たれ，出力 y は low のままとなる．すなわち，1段目のゲートは全体として，OR ゲートとして動作する．

フットとキーパーは，以下のような役割を担う付加的なトランジスタである：

フット フットは，プリチャージ期間中に，入力の値によらず，プリチャージ・ノードをグラウンドから切り放す役割を担う．

フットは，ゲートの論理を担う *n*MOS トランジスタの数倍程度の大きさとすることが普通である．ゲートの論理を担う *n*MOS トランジスタを大きくすると，入力容量が増加し，前段のゲートの動作速度が低下する．そのたため，むやみに大きくすると，回路全体の遅延がかえって悪化するおそれがある．このようなトランジスタは，回路全体のバランスから個々のトランジスタの大きさを決定する必要がある．一方フットは，プリチャージ・クロックによってドライブされるため，当該ゲートの動作速度に与える影響だけからその大きさを決定すればよい．クロックの分配は，それ自体難しい問題であるが，いずれ解決しなければならない．

キーパー キーパーは，リークによる誤動作を防ぐために付加されることがある．評価期間中，入力 a，b が共に low である場合にも，プリチャージ・ノード x に蓄えられた電荷が

リークすることにって誤動作が発生する怖れがある．キーパーは，このような場合にONとなっており，リークによって失われた電荷を補償する．

キーパーの駆動能力は，わずかなリークを補償するだけあればよい．また，キーパーの駆動能力が高すぎると，ディスチャージを妨害して，ゲートの動作速度を低下させることになる．そのためキーパーの駆動能力は，ゲート幅を最小化した上でゲート長を敢えて長くとるなどして，ごく小さく抑えられる．

2.1.2 ドミノ論理

図2.1に示した回路では，インバータの出力バッファにより，プリチャージ期間中には出力がlowとなっている．評価期間に，1段目のゲートでディスチャージが起こってその出力がhighに変わると，2段目のゲートでもディスチャージが起こってその出力がhighに変わる．このように，同図2.1のような形式のダイナミック回路では，ドミノが倒れるように，次から次へとディスチャージが伝搬していくことになる．そのため，このような形式のダイナミック回路は，特に，**ドミノ論理 (domino logic)** と呼ばれる．

ドミノ論理では，前段の出力が，プリチャージ期間中にはlow (normally low) であることが保証されるため，2段目以降のゲートでは，フットを省略することができる．図2.1では，薄く示してあるフットを省略しても，プリチャージ期間中ノード y はlowであることが保証されるため，2段目のゲートでは，フットがなくても正しくプリチャージを行うことができる．フットを省略することは，高速化に対してかなりの効果が期待できる．この例の場合だと，2個直列に接続されていた *n*MOSトランジスタを1個にでき，等価抵抗の値を半減できることになる．

逆に，プリチャージ期間中にhigh (noramlly high) であるシグナルは，ドミノ論理に入力することができない．フットを付加したとしても，プリチャージ期間の終了直後には，両方がONになり，ディスチャージが行われてしまうためである．その入力が最終的にlowになる場合にも，一旦ディスチャージされてしまった電荷は元に戻らない．プリチャージ期間中にhighであるシグナルを入力とできないことの影響については，3.6.2項でより詳しく述べる．

2.1.3 ダイナミック回路のメリット，ディメリット

図2.2に，2入力ORゲートのスタティック回路による実装とダイナミック回路による実装を示す．ダイナミック回路は，プリチャージ・トランジスタ，フット，キーパーなど，いくつかの付加的なトランジスタを必要とするものの，基本的には，論理的に等価なスタティック回路から *p*MOSトランジスタを省いたものと言える．*p*MOSトランジスタのキャリア移動度は *n*MOSトランジスタのそれの半分程度であるため，*p*MOSトランジスタは通常 *n*MOSトランジスタの倍程度のゲート幅を必要とする．したがってダイナミック回路では，以下のようなメリットが得られる：

- コンパクトである．

  付加的なトランジスタを除けば，面積は 1/3 程度になる．

- 高速である．

  入力のゲート容量が 1/3 程度になるため，前段ゲートの動作が高速になる．

  また，スタティック回路における *p*MOS トランジスタのドレイン容量にあたる分だけプリチャージ・ノード（図 2.1 の *x*）の容量が減少するため，ディスチャージも高速になり，当該ゲートのスイッチングが高速になる．

- 多入力の OR が構成しやすい．

  スタティック回路の場合，*n* 入力 OR ゲートでは，*n* 個の *p*MOS トランジスタが直列に接続されるため，特に高速化が困難になる．

ただし，以下のようなディメリットもある：

- 入力のグリッチや，静的ハザードが許されないため，設計がより難しくなる．
- ドミノ論理では，基本的には，NOT のない AND/OR ゲートしか実現できない．

  NOT を含む回路を実現するには，2 レール (double-rail，2-rail) 化する，すなわち，反転出力を求める回路を別途用意する必要がある．次項では，2 レールの回路の例を示す．ただし，2 レール化を広範囲に行うと，回路規模は倍増し，ダイナミック論理のコンパクトさというメリットはほぼ失われてしまう．

- スタティック回路に比べて，消費電力が大きくなりがちである．

  スタティック CMOS 回路では，基本的には，出力が変化しなければ電力は消費されない．一方ダイナミック回路では，出力が low のままならば問題ないが，出力が high のままである場合には，毎サイクル，プリチャージ / ディスチャージが繰り返されるため，消費電力が大きくなりがちである．

これらのディメリットのため，ダイナミック論理は万能ではないが，特に次節以降で述べるアレイのような規則的な回路では，そのコンパクトさ，高速性は不可欠なものとなっている．

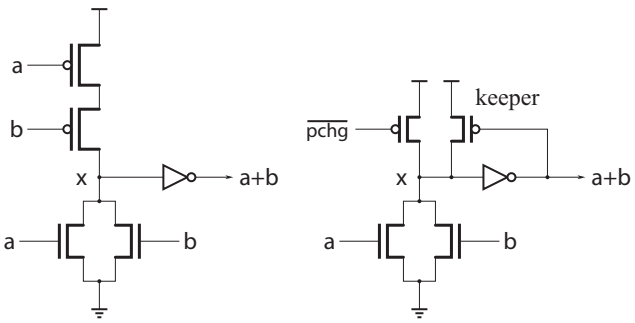


図 2.2: スタティック OR ゲート（左）とダイナミック OR ゲート（右）

2.2 一致比較器

Out-of-order 命令スケジューリングのロジックでは，一致比較器は，物理レジスタ番号の比較などに用いられる．特に，2.4 節で述べる CAM に用いられる．一致比較器はまた，ダイナミック・ロジックの好例でもある．

以下，まず 2.2.1 項で，n ビットの一致比較器の全体的な構成について述べた後，2.2.2 項で，n ビット一致比較器の構成要素となる 1 ビットの一致比較器について述べる．

2.2.1 n ビット一致比較器

一致比較器は，通常，やや変則的なダイナミック回路によって実現される．比較する 2 つの入力の第 i ビットをそれぞれ $a[i]$，$b[i]$ とすると，全体の一致は $\prod_{i=0}^{n-1} \overline{a[i] \oplus b[i]}$ と，n 入力の AND で表される．n 入力の AND では，nMOS ゲートが n 個直列に接続されるため，n が大きいと高速化が難しい．n は，最大 4 程度に制限されることが普通である．

スタティック回路による実装では，通常トゥリー状に変形することによって，多入力の AND を実現するであろう．一方，ダイナミック回路による実装では，以下で詳しく述べるように，$\prod_{i=0}^{n-1} \overline{a[i] \oplus b[i]} = \sum_{i=0}^{n-1} (a[i] \oplus b[i])$ と変形し，n 入力の OR として実現できる．

図 2.3 に，ダイナミック・ロジックを用いた，n ビット一致比較器の回路図を示す．**マッチライン** *match* は，アクセスに先だって high にプリチャージされ，インバータの出力バッファにより出力 $\overline{match}$ は low となる．*match* の下には，XNOR ゲートのシンボルで表されている 1 ビット一致比較器と，ドライバ nMOS ゲートのペアが並べられている．1 ビット一致比較器のいずれかが入力の不一致を検出すると，*match* に蓄えられた電荷がドライバによってディスチャージされ，match は low に，$\overline{match}$ は high に変化する．逆に，入力の各ビットがすべて一致する場合，*match* は high，$\overline{match}$ は low のままとなる．

マッチライン *match* は，プリチャージ・ノードであり，normally high，すなわち，プリチャージ期間中 high であるため，ドミノ論理に対する入力とはできないことに注意されたい（2.1 項）．例えば，いくつかの一致比較器のすべてが一致を検出したことを知りたい場合，$\prod match = \sum \overline{match}$ を求める必要があるが，この論理は，normally low である $\overline{match}$ を入力とするため，ドミノ論理で実現することができる．しかし，いくつかの一致比較器のいずれかが一致を

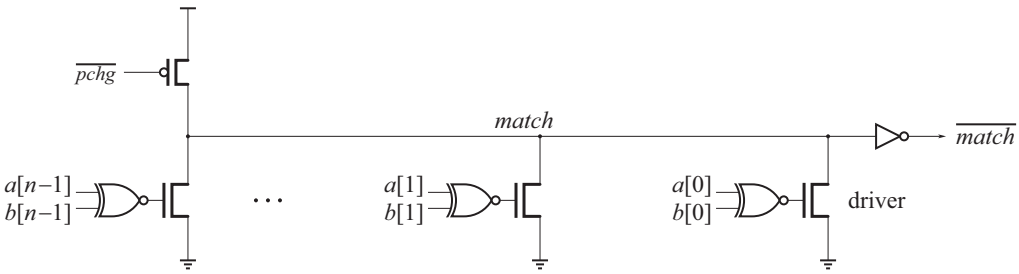


図 2.3: n ビット一致比較器

検出したことを知りたい場合には，$\sum match$ を求める必要があるが，この論理は，normally high である *match* を入力とするため，ドミノ論理では実現することができない．このことは，3.6 節で述べる連想方式 *wakeup* ロジックの実装において，やや強い制約条件となる．

2.2.2　1ビット一致比較器

前項で述べた *n* ビット一致比較器を構成する 1 ビット一致比較器には，いくつかの実装方法が考えられる．図 2.3 から想像されるとおりのスタティックな XNOR ゲートを用いる方法がある．ただしその場合には，各ドライバ・ゲートの下にフットが必要である（2.1 節参照）．

文献 [50] では，ダイナミック XOR ゲートを用いて，一致比較器全体を 1 段のダイナミック・ゲートとして実現する方式が示されている．図 2.4 に，その回路図を示す．

なお，同図 2.4 中の各 1ビット比較器のように，XOR/XNOR のほとんどの実装は相補的な入力を必要とする．

このような 1段の回路は単純で，高速動作が可能であるように見えるが，2 個直列に接続された *n*MOS スタックでマッチラインをドライブする必要があるため，ディスチャージに時間がかかる．

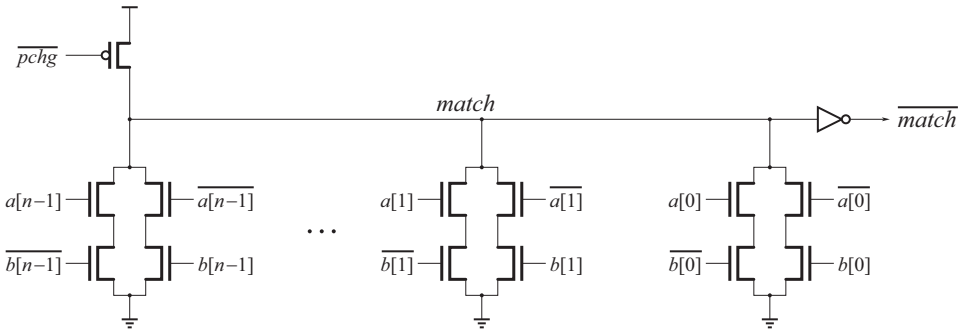


図 2.4: ダイナミックな XOR ゲートを用いた一致比較器

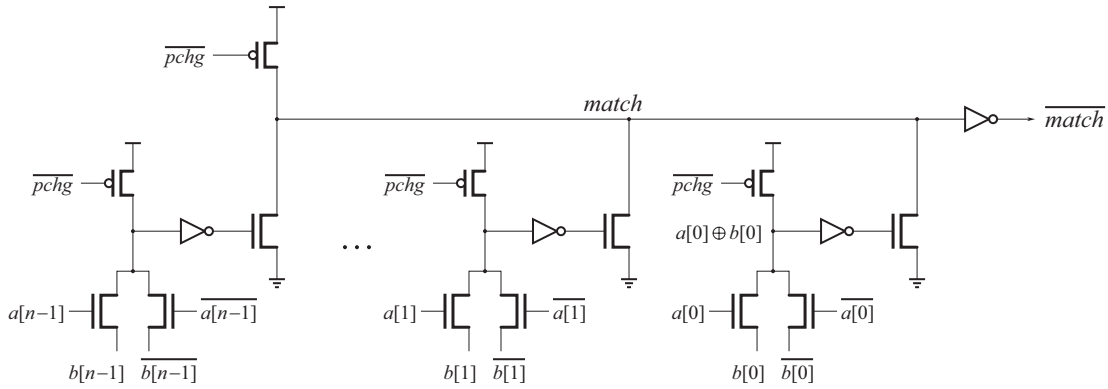


図 2.5: ダイナミックなセレクタを用いた一致比較器

図 2.5 に，別体のマッチライン・ドライバを持つ一致比較器を示す．この方式では，2 個 1 組の *n*MOS パス・ゲートによって構成したダイナミックなセレクタを用いている．*a*[*i*] の値によって，2 個のパス・ゲートのうちどちらか一方が ON になる．*a*[*i*] と *b*[*i*] が異なる場合には，ON となっているパス・ゲートを通じてノード *x* の電荷がディスチャージされる．その結果，2 段目のマッチライン・ドライバがマッチラインをディスチャージする．

同図の方式では，ゲート段数が増えるため，その部分で遅延が増大するが，別体のマッチライン・ドライバによって，マッチラインのディスチャージが高速化されるため，ビット数 *n* が多い場合に有利である．このような最適化はもちろん，上述したダイナミック XOR ゲートを用いる方式でも可能である．

2.3 RAM

Out-of-order 命令スケジューリングの処理の多くは，テーブルの読み書きによって実現される．本節では，それらのテーブルを構成する RAM —— SRAM (Static RAM) の論理回路について述べる．以下，特に断りのない限り，RAM と言えば SRAM のことを指すことにする．

命令スケジューリングのロジックで用いられる各種の RAM は，汎用 SRAM チップやオンチップ・ キャッシュなどの通常の SRAM とは，そのパラメタが大きく異なる．図 2.6，および，表 2.1 に，通常の RAM，命令スケジューリング・ロジックの RAM，および，スーパースカラ・プロセッサの整数系物理レジスタ・ファイルのパラメタを示す．命令スケジューリング・ロジックの RAM の特徴は，以下のようにまとめられる：

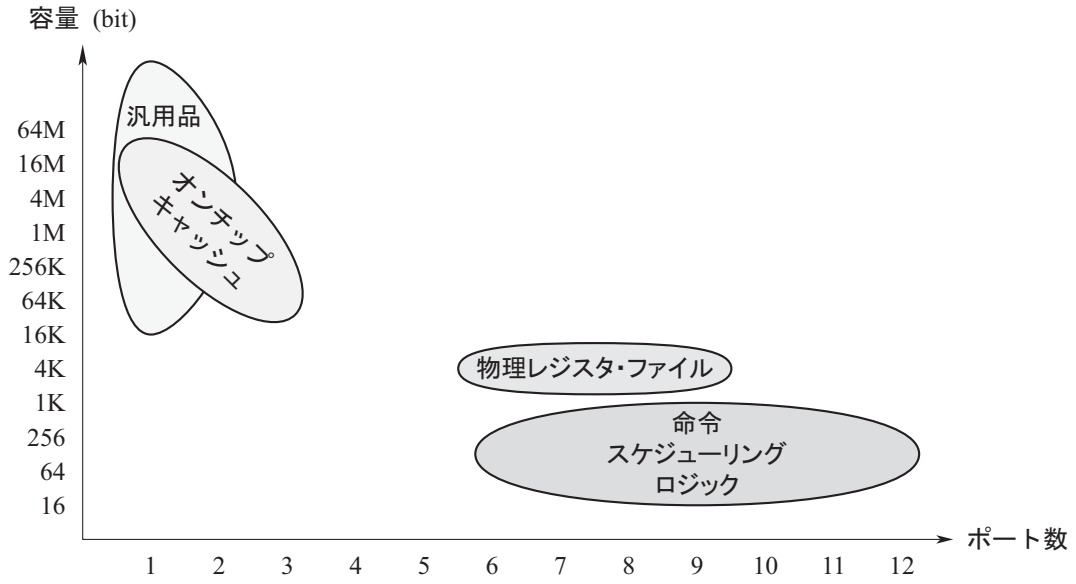


図 2.6: SRAM のパラメタの比較

種 別	ワード数	ビット幅	ポート数
汎用品	2^{10}〜2^{20}	2^{0}〜2^{7}	1〜 2
オンチップ・キャッシュ	2^{10}〜2^{20}	2^{5}〜2^{6}	1〜 3
物理レジスタ・ファイル	2^{5} 〜2^{6}	2^{5}〜2^{6}	6〜 9
命令スケジューリング・ロジック	2^{4} 〜2^{5}	2^{0}〜2^{6}	6〜12

表 2.1: SRAM のパラメタの比較

1. **浅い (shallow)**，すなわち，ワード数が少ない．

   通常の RAM は深い (deep)，すなわち，ワード数はビット幅より 1〜2 桁大きい．それに対して，命令スケジューリング用の RAM では，ワード数はビット幅と同程度である．

2. **小容量である**．

   特に，浅いことによって，通常の RAM に比べて 1〜4 桁程度小容量で，物理レジスタ・ファイルと比べてもやや小さい程度である．例えば MIPS R10000 プロセッサでは，物理レジスタ・ファイルの容量は，64 word × 64 bit = 4Kb である．それに対して，命令ウィンドウを構成する RAM は，16 word × 60 bit = 960b と，物理レジスタ・ファイルの 1/4 程度である．

3. **ポート数が多い**．

   命令スケジューリング・ロジックの RAM のポート数は，大抵の場合，命令のフェッチ幅，ディスパッチ幅，または，発行幅の増加関数で与えられる．また，1つの RAM に対する読み出しと書き込みは，通常，パイプラインの別のステージで同時に行われるため，リード / ライト・ポートではなく，リード・ポートとライト・ポートを別個に持つ．これらの理由により，命令スケジューリング・ロジックの RAM のポート数は，大きいものでは 10 を越える．

4. **アドレス・デコーダが不要な場合がある**．

   通常の RAM では，アドレスは 2 進数で与えられるため，それをデコードするデコーダが必要となる．一方，命令スケジューリング・ロジックの RAM の一部では，前段の回路からアドレスが最初からデコードされた形で与えられることがある．その場合，当然のことながら，アドレス・デコーダは不要である．

5. **シングル・ビットラインの場合がある**．

   通常の RAM では，相補的な 2 本のライン，すなわち，2 レールのビットラインを用いてリード / ライトを行うダブル・ビットライン方式を採る．一方，命令スケジューリング・ロジックの RAM の一部では，1 本のビットラインを用いてリードを行うシングル・ビットライン方式を採らざるを得ない．

以下，命令スケジューリング・ロジックの RAM のこれらの特長を考慮しつつ，RAM について説明する．

2.3.1 RAM のロジック

図 2.7 に，RAM の回路図を示す．RAM は，基本的には，1ビットを記憶する RAM セルの 2 次元のアレイ（配列，array）によって構成される．

セル

RAM セルは，2 個のインバータからなるループが持つ双安定 (bi-stable) な状態よって 1ビットの情報を記録する．CMOS RAM の場合，各インバータは CMOS インバータとなる．各インバータの *p*MOS トランジスタは**負荷トランジスタ (load transistor)**，*n*MOS トランジスタは**ドライバ・トランジスタ (driver transistor)** と呼ばれる．この 2 つのインバータを構成する 4 つのトランジスタからなる部分回路を **4T セル (4T-cell)** と呼ぶ．4T セルの基本的な動作は主にドライバ・トランジスタによって担われており，負荷トランジスタは主に直流を遮断する役割を果たす．4T セルは，左 / 右インバータの入 / 出力，および，出 / 入力にあたる，2 つの相補的なアクセス・ノードを持つ．

通常，4T セルの 2 つのアクセス・ノードは，**アクセス・トランジスタ (access transistor)** と呼

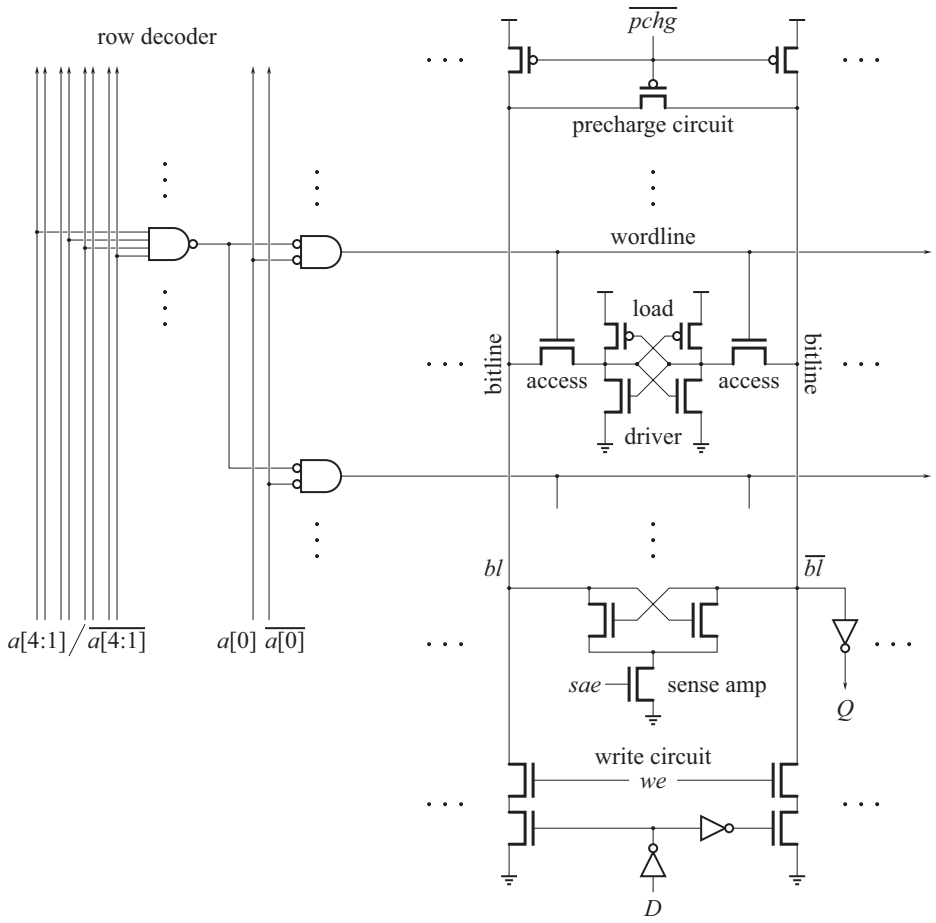


図 2.7: RAM の回路図

ばれる *n*MOS パス・トランジスタを介して，相補的 (complementary) な**ビットライン (bitline)** に接続されている．アクセス・トランジスタは，**ワードライン (wordline)** によって制御される．

図 2.7 に示すような 1 ポートの RAM の場合，セルは，1 個の 4T セルと 2 個アクセス・トランジスタの，合計 6 個のトランジスタからなるため，これを特に **6T セル (6T-cell)** と呼ぶ．

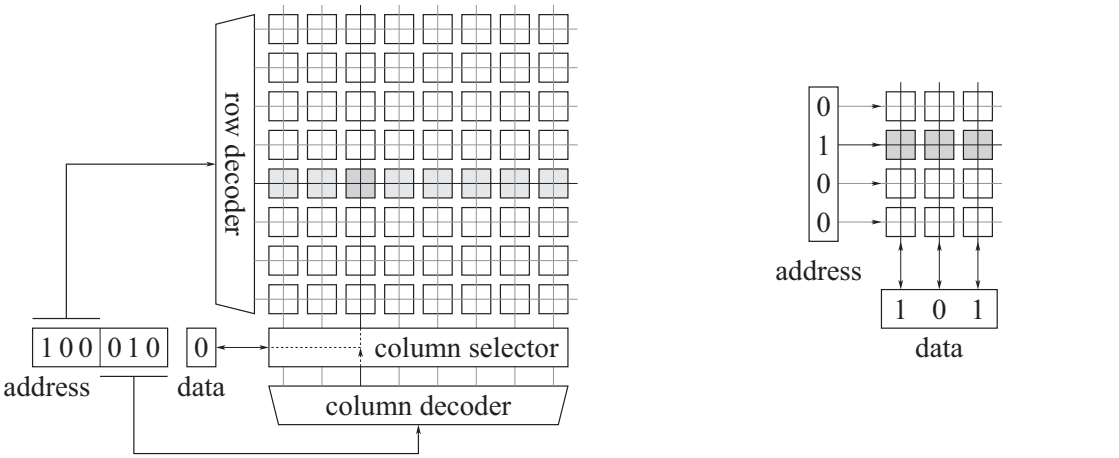
セル・アレイ

汎用品では通常，語構成に関わらず，セル・アレイは正方に近くする．一方，命令スケジューリング用の RAM では，アレイの構成は語構成と一致させればよい．

RAM の語構成を m word × n bit としよう．命令スケジューリング用 RAM では，アレイの構成は，語構成と同じく m 行 n 列とすればよい．しかし，汎用品の場合には，そのようにするわけにはいかない．前述のとおり，汎用品は非常に深く，ワード数 m はビット数 n より 1〜2 桁大きい．そのため，アレイの構成を語構成と同じく m 行 n 列とすると，アレイ全体は，列方向に細長い，非常にいびつな形状となってしまう．その場合，特にビットラインが非常に長くなってしまう他，場合によってはチップ内に収めることすら難しくなる．

そこで通常の RAM では，語構成とはある程度独立に，アレイを正方に近づける．図 2.8 (a) に，典型的な $q2^n$ word × 1 bit の RAM のセル・アレイを示す．n bit のアドレスは，上位 r bit と下位 c bit に分けられ，上位 r bit は**行デコーダ** (row decoder) に，下位 c bit は**列デコーダ** (column decoder) に，それぞれ入力される．上位 r bit/下位 c bit によってそれぞれ指定された行 / 列のセルが，アクセスされる．行デコーダは，選択された行のワードラインをドライブする．列選択回路 (column selector/multiplexer) は，列デコーダの出力にしたがい，ワードラインによって選択された 1 行，2^c bit の中から，更に必要な 1bit を選択する．セル・アレイは，2^r 行 2^c 列となり，$r = c$ のとき，ワードラインとビットラインの長さの和は最小となる．

一方，前述したように，命令スケジューリング用の RAM は，汎用品と比べると極めて浅く，ワード数 m とビット数 n は同程度のオーダである．そのため，語構成と同じ m 行 n 列の



(a) Conventional RAM Cell Array　　(b) RAM Cell Array for Scheduling

図 2.8: RAM セル・アレイ

アレイはもともとオーダ的に正方に近い．アレイの構成と語構成が一致している場合には，図 2.8 (b) に示されているように，列デコーダ，列選択回路は不要であり，アクセスは各列に対して直接行われる．なお図 2.7 の回路図でも，列デコーダ，列選択回路は描かれていない．

行デコーダ

通常の RAM では，アクセスされる行のアドレスは 2 進数で与えられる．行デコーダ (row decoder，wordline docoder) がそれをデコードし，対応するワードラインをドライブする．

RAM の行デコーダは，通常，2 段デコードを行う．図 2.7 の左には，4bitプリデコードを行う 5-bit 行デコーダの回路図を示した*．同図では，5bit のアドレス a[4:0] のうち，a[4:1] の 4bit が 1 段目の NAND ゲートでデコードされ，残り 1bit a[0] が 2 段目の 2 入力 NOR ゲートによってデコードされている．2 段目の NOR ゲートがワードライン・ドライバを兼ねている．

最初 a[0]/$\overline{a[0]}$ は，両方とも high にプリチャージされており，ワードラインは low に保たれる．1段目のデコードが終了した後，a[0]/$\overline{a[0]}$ の一方をディスチャージすると，指定されたワードラインが high にドライブされる．

ビットラインとセンスアンプ

行デコーダによって，指定された行のワードラインが high にドライブされると，各列では指定された行の 1 つのセルがビットラインに接続される．

RAM の読み出しでは，ビットラインに接続されている多数のセルのアクセス・トランジスタのドレイン容量のために，ビットラインの電位は非常にゆっくりとしか変化しない．そこで，ごく小さい電位変化を検出するセンスアンプが使用される．

通常の RAM では，2 レールのビットラインを持つため，それらの間の電位差を検出するダブルエンドのセンスアンプが用いられる．2 レールのビットラインとダブルエンドのセンスアンプは，同相ノイズをキャンセルする働きがある．

図 2.7 のセンスアンプでは，読み出しに先だって，両方のビットラインが high にプリチャージされる．ワードラインの一本がアサートされると，選択されたセルのドライバ・ゲートのどちらか一方が，アクセス・トランジスタを通して，接続された一方のビットラインの電荷をゆっくりとディスチャージする．

センスアンプは，2 本のビットライン間に十分な電位差が生じた後に，アクティブにする．*sae* が high となって，センスアンプがアクティブになると，*n*MOS ゲートのペアが両方とも ON になり，ビットラインの電位は急速に下がりはじめる．このとき，より速く低電位になった方の反対側の *n*MOS ゲートがクロス結合によって OFF になるため，最初の電位差が急速に増幅されることになる．

なお，上述した行デコーダにおけるアドレス a[4:0] や，ビットラインなどの 2 レールの回路は，2.1 項で述べた，NOT を含むダイナミック・ロジックの実現例となっている．

---

* この回路を『1bitプリデコードする』と表現する文献 [51] もある．

2.3.2 多ポートRAM

表2.1に示したように，命令スケジューリング・ロジックで用いられるRAMのポート数は，6〜12と非常に多い．

RAMセルをマルチポート化するには，基本的には，ポートを構成する要素，すなわち，ワードライン，ビットライン，および，アクセス・トランジスタを，ポートごとに複製すればよい．図2.9 (a)に，2-read × 1-writeのRAMのセルの回路図を示す．

ただし，同図2.9 (a)のように，それらの要素を単に複製した場合には，以下のような不具合が生じる：

- 同図2.9 (a)の回路では，2つのリード・ポートで1つのドライバ・トランジスタを共有していることになる．そのため，2つのリード・ポートにおいて同時に同じワードを読み出した場合，読み出し速度が低下する．1つのドライブ・トランジスタが，2つのポートの2本のビットラインをドライブすることになるためである．
- アクセス・トランジスタのソース容量のため，アクセス・ノードb，b'の容量が増加し，書き込み速度が低下する．

そのため，2〜3ポート程度ならばまだしも，更にポート数を増やす場合には，同図2.9 (b)のようにした方がよい．同図2.9 (b)では，同図2.9 (a)に対して，以下のような改良がなされている：

- ドライバ・トランジスタとは別に，リード・ポートごとにビットライン・ドライバが用意されている．そのため，同時に読み出しても読み出し速度は低下しない．

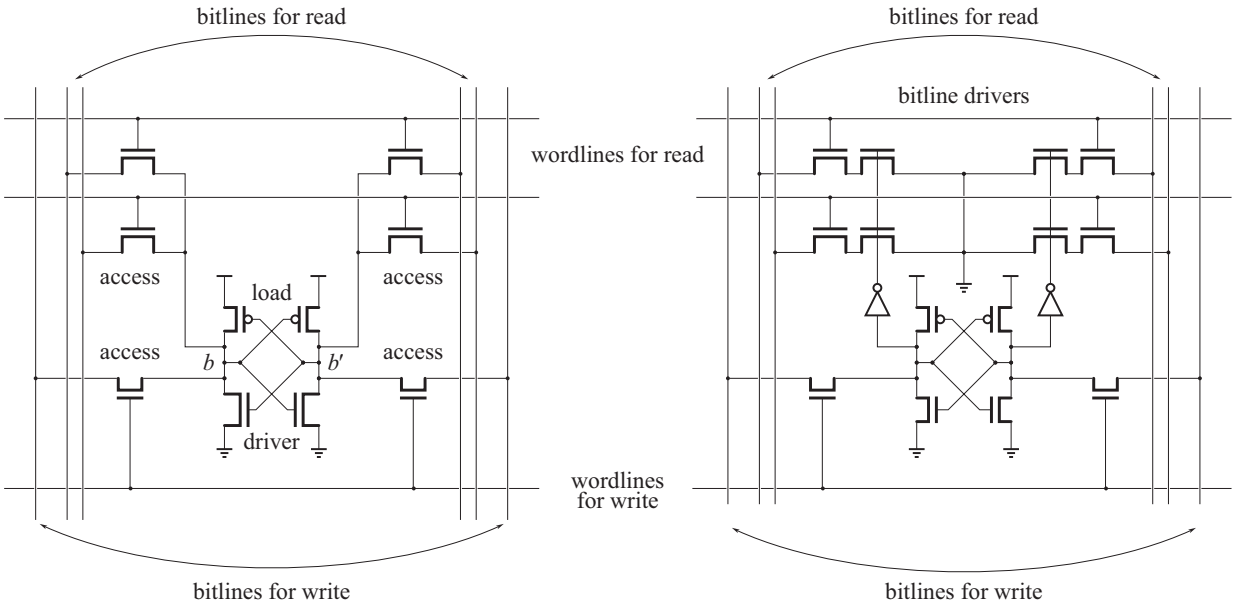
  なお，このビットライン・ドライバのように，2個直列に接続されたnMOSゲートからなる構造は，**nMOSスタック**と呼ばれる．
- アクセス・ノードb，b'と上記ビットライン・ドライバの間に，インバータのバッファが挿入されている．このインバータによりアクセス・ノードの容量が制限されるので，書き込み速度が向上する．

また，これらの改良により4Tセル内のドライブ・トランジスタの負荷容量が制限されるため，ドライバ・トランジスタ自体はできる限り小型化してもよい．

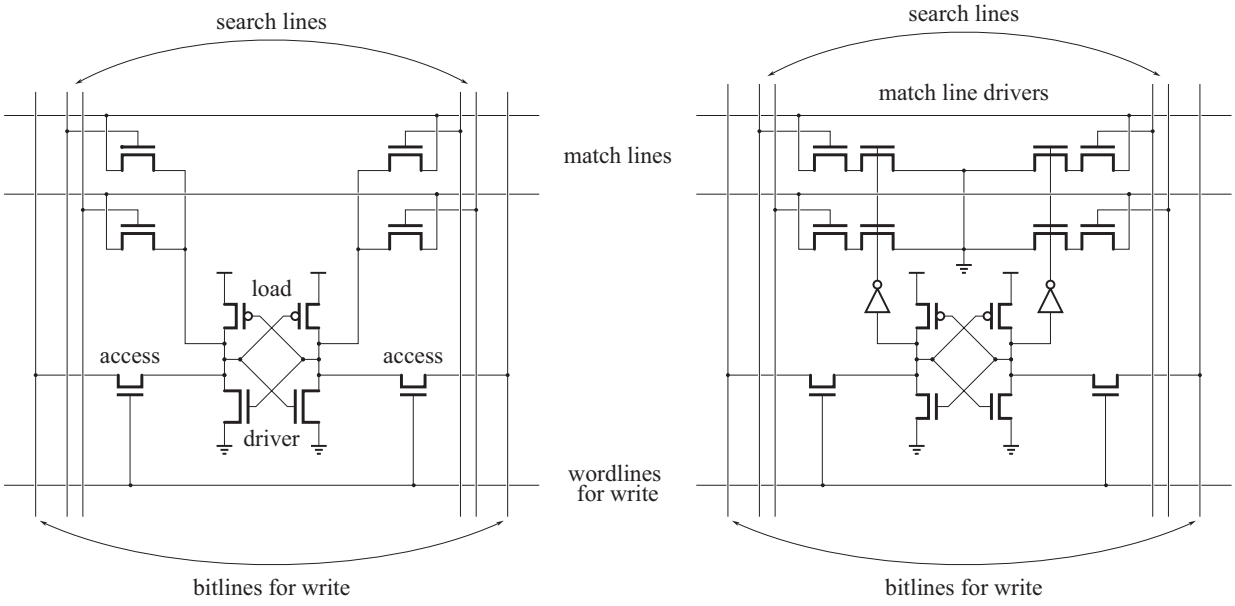
2.3.3 シングル・ビットライン

命令スケジューリング用RAMでは，方式によっては，原理的に2レールの出力が得られないことがある．その場合には，1本のビットラインを用いて読み出しを行う必要がある．1本のビットラインを用いる方式は，**シングル・ビットライン**と呼ばれる．

ダブル・ビットラインでは，ビットラインとそれ用のアクセス・トランジスタのペアが，各ポートに2組ずつある．シングル・ビットラインは，そのどちらか1組を省略したものである．



(a) RAM Cell w/o Separate Drivers (b) RAM Cell with Separate Drivers



(c) CAM Cell w/o Separate Drivers (d) CAM Cell with Separate Drivers

図 2.9: マルチポートの RAM セルと CAM セル

ダブル・ビットラインでは，2本あるビットラインのどちらがディスチャージされたかを検出するが；シングル・ビットラインでは，唯一のビットラインがディスチャージされたか，されなかったかを検出することになる．

シングル・ビットラインの応用分野

シングル・ビットラインは，シングル・ポートの汎用SRAMなどでは一般的ではないが，ROMなどでは一般的である[51].

また，シングル・ビットラインは，特にポート数が多い場合に，回路面積の削減，ひいては回路遅延の短縮に効果がある．前項で述べたように，ポート数の非常に多いSRAMでは，ビットライン，および，ワードラインの配線領域によってセル面積が決まる．そのため，ビットライン数を半減できることは回路面積の削減効果が高い．また，基板によってセル面積が決まる，ポート数が2〜3程度の場合でも，ビットライン・ドライバの数を半減できるため，回路面積の削減に一定の効果がある．

そのためシングル・ビットラインは，物理レジスタ・ファイルなど，ロジックLSI内部の，ポート数の非常に多いSRAMなどで利用が進みつつある．

センスアンプ

シングル・ビットラインでは，2.3.1項で述べたような，ダブルエンドのセンスアンプをそのまま用いることはできない．そのための対処法には，以下の2つの方法が考えられる；1つはダブルエンドのセンスアンプを転用する方法であり，もう1つは**シングルエンド** (single-end) のセンスアンプを用いる方法である．

シングルエンドのセンスアンプの例を図2.10に示す[51]．このセンスアンプは，ビットライン x の電位を，インバータI1の高利得領域である $V_{DD}/2$ 付近にプリチャージすることによって，I1の応答性を高めるものである．

sae がhighになり，センスアンプがイネーブルされると，*p*MOSトランジスタM3がビットラ

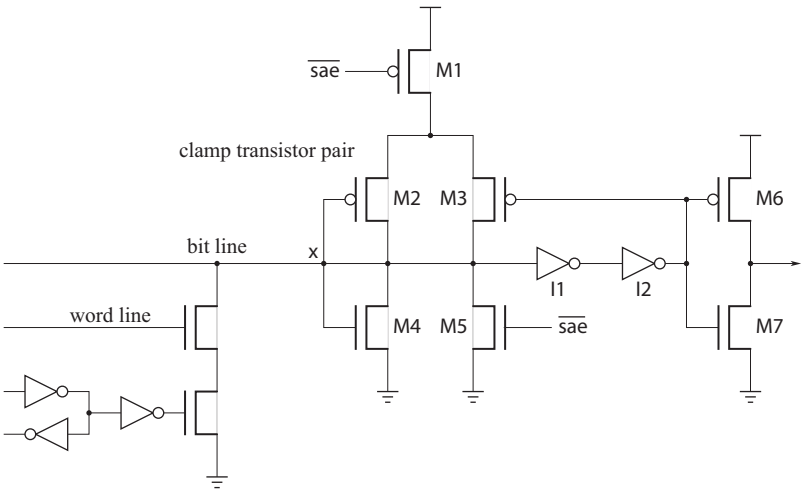


図 2.10: シングルエンドのセンスアンプ

イン x をプルアップしようとする．しかし，ビットライン x の電位が I1 の論理閾値に近づくと，I1，I2 からなる負のフィードバック・ループにより，M3 は OFF になり，x の電位の上昇はそこで止まる．このとき，M6 も M3 と同様ぎりぎり OFF になり，出力は low となる．

この状態で，選択されたセルが入力ビットライン x をわずかにプルダウンすると，I1 は敏感に反応し，出力が切り替わる．

なお，M2 と M4 は，クランプ・トランジスタ・ペアと呼ばれ，ビットライン x の電圧振幅を $V_{DD}/2$ 近辺に固定 (clamp) して，再び $V_{DD}/2$ にプリチャージし直す時間を短縮している．

2.3.4 ビットライン・ドライバのサイジング

個々のトランジスタのサイズ，特にゲート幅を最適化することを，トランジスタの**サイジング** (sizing) という．サイジングにおける目的関数は，本稿で特に問題としている回路遅延の他，面積，あるいは，消費電力などによって与えられる．

トランジスタのゲート幅を拡大すると，それ自体の遅延は短縮される一方で，そのゲート容量が増加するため，前段のトランジスタの遅延は増加することになる．したがって通常のロジックの遅延を最小化するには，主にファン・インとファン・アウトから最適なゲート幅を決定することになる．

RAM セルでは，特にリードに遅延が問題となるため，主にビットラインをドライブするトランジスタがサイジングの対象となる．図 2.9 (a) のロジックではドライバ・トランジスタとアクセス・トランジスタが，同図 2.9 (b) のロジックでは別体ビットライン・ドライバを形成する nMOS トランジスタが，それにあたる．ロード・トランジスタや，図 2.9 (b) のロジックにおけるインバータのバッファなどは，リードの遅延には関係ないので，可能な限り小さく設計すればよい．

最適なドライバ・サイズは，RAM の容量，および，ポート数によって決まる．以下，それぞれについて述べる．

容量とドライバ・サイズ

RAM セルのドライバのサイジングには，以下の 2 つの相対する方針が考えられる：

1. **小さいドライバ**

   ドライバをできる限り小さく設計してセル・アレイ全体の面積を抑え，ワードライン，ビットライン長を短縮することによって，特に配線遅延の短縮を図る．

2. **大きいドライバ**

   セル・サイズをある程度犠牲にして，ゲート幅の広いドライバをセル内に配置することによって，特にゲート遅延の短縮を図る．

どちらの方針を採るべきかは，全体の遅延に占めるゲート遅延と配線遅延のバランスによる．そのバランスは，主に RAM の容量によって決まる．

汎用品やオンチップ・キャッシュなどの通常の RAM では，セルは可能な限り小さく設計される．大容量の RAM では，回路面積を最小化すること自体非常に重要であるうえ，小さい

セルは遅延の上でも有利だからである．大容量のRAMでは，ワードライン，ビットラインが非常に長く，配線遅延が支配的である．そのため，全体の遅延は，ほぼ配線長のみによって決まり，ドライバをいくら大きくしてもほとんど短縮されない（2.5節）．

一方，命令スケジューリング・ロジックのRAMのように，容量が十分に小さく，ゲート遅延の影響が大きいときには，大きいドライバがより有利になる．

また，最小加工寸法が縮小されると，配線遅延の影響が増大するため，より少ない容量でも小さいドライバがより有利になる．

ポート数とドライバ・サイズ

図2.11に，ポート数とセル面積の関係を示す．セルの面積は，概略的には，トランジスタが形成される基板(substrate)上の領域と，配線が形成される配線層の領域の，どちらか大きい方によってバウンドされる．それぞれの領域の面積は，以下のように，ポート数 p の増加関数で与えられる：

基板 4Tセルやインバータのバッファなどの面積は $O(1)$ である．アクセス・トランジスタ，あるいは，ビットライン・ドライバの数は $2p$ 個であり，その面積は $O(p)$ である．したがって，全体としては，$O(p)$ となる．

配線層 1つのセルには，行方向に p 本のワードラインと2本の電源バス，列方向に $2p$ 本のビットラインを通す必要がある．そのため配線層の領域は，配線のピッチを P とすると，縦×横が $(p+2)P \times 2pP$ の矩形より大きい．したがってその面積は，$O(p^2)$ となる．

したがって，特に p が大きい場合には，セル面積は配線層によってバウンドされる．その意味において，RAMセルの，ひいては，RAMの面積のオーダは，$O(p^2)$ である．

通常のRAM

汎用品オンチップ・キャッシュなどの通常のRAMでは，その容量が大きいため，小さいドライバの方が有利である．

RAMの面積のオーダは $O(p^2)$ であるが，実際に配線層によってバウンドされるのは十分に大きな p に対してのみであり，ポート数が十分に少ない場合には基板によってバウンドされる．特に $p = 1$，すなわち，シングル・ポートのRAMの場合，セル面積は6Tセルの面積によって与えられる．そのため，各RAMベンダは，6Tセルの面積が何 F^2 になるかによっ

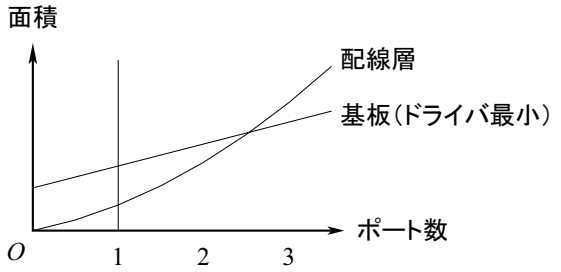


図2.11: ポート数とセル面積

て，自社製品の優秀性をアピールすることになる．

ポート数が多い場合には，実際に配線層によってバウンドされることになる．その場合には，配線層によって決まる $(p+2)P \times 2pP$ の矩形領域の中に，可能な限り大きくトランジスタを作り込めばよい．

命令スケジューリング・ロジックの RAM

命令スケジューリング・ロジックの RAM では，汎用品などの通常の RAM とは様相が大きく異なる．命令スケジューリング・ロジックの RAM は，通常の RAM と比べて極めて小容量であり，ワードライン，ビットラインは非常に短く，ゲート遅延の影響が大きいため，大きいドライバが有利である．

7 章の評価では，配線遅延は無視することはできないが，支配的とは程遠い．ビットライン・ドライバのゲート幅を増加させるにしたがい，セル面積が増加し，ビットライン，ワードラインの配線長も増大するにも関わらず，リードの遅延は改善されていった．最終的には，セル面積の半分程度をビットライン・ドライバが占めることとなった．

また，リード・ポートがシングル・ビットラインであることも手伝って，配線層ではなく，基板がセル面積をバウンドすることとなった．

2.4 CAM

Out-of-order 命令スケジューリングのロジックで用いられるテーブルでは，そのエントリに対する連想検索が必要なものがある．そのようなテーブルは，**CAM** (Content Addressable Memory) によって実現される．

2.4.1 CAM のロジック

通常の CAM の使い方は，以下のようなものである; CAM の各エントリには，**キー**と**バリュー**（value，値）のペアが格納される．そして，与えられた検索キーに一致 (match) するキーが検索 (search) される．この検索が，格納されたキーの個数 n に対して，$O(1)$ で行われるとき，この検索を**連想検索** (associative search) と呼ぶ．検索キーと一致するキーがあれば，そのキーと対になっているバリューに対して，リード / ライトが行われる．このように，検索キーを与えてバリューにアクセスすることを，**連想アクセス**と呼ぶ．

図 2.12 に，最小限の機能を持つ CAM のロジックを示す．同図中，$key[i]$，$val[i]$ は，キー，および，バリューの第 i ビットをそれぞれ示している．CAM のロジックは，キーを格納する部分，バリューを格納する部分，そして，それらの間を接続するドライバ回路からなる．

さて，この『CAM』という用語だが，その指す内容は分野によって若干異なっている．より応用よりの分野では，上述のように，キーとバリューのペアを格納するロジック全体を CAM と呼ぶのが普通である．しかし，回路の実装に近い分野では，キーを格納する部分のみを CAM と呼ぶことが多い．バリューが格納される部分は，図 2.12 から分かるように，単

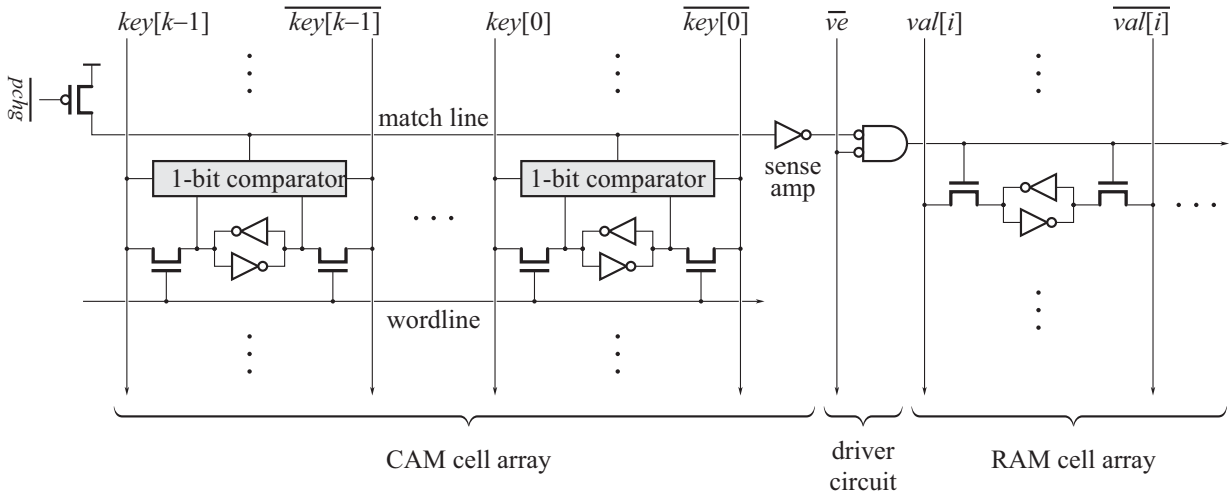


図 2.12: CAM のロジック

なる RAM セルのアレイによって構成されるため，それを RAM と呼んで区別すると都合がよいからであろう．

本稿では，より正確に，全体を CAM，そして，キーを格納する部分を **CAM セル・アレイ**，バリューを格納する部分を **RAM セル・アレイ**と呼ぶことにする．

RAM セル・アレイは前節で述べたのと同様であるので，以下ではそれ以外の部分について説明する．

CAM セル・アレイ

CAM セルは，基本的には，キーの 1bit を格納する RAM セルに対して，入力された検索キーの 1bit と格納されたキーの 1bit とを比較する 1-bit 一致比較器を付加したものである．同図 2.12 のロジックにおいて，各 CAM セルの下部にある 6T セル，および，リード / ライト・ポートは，前節で述べた RAM セルのものと同様である．上部には，1-bit 一致比較器が配されている．

同図 2.12 に示したロジックは 1 ポートであり，連想アクセス時の検索キーの入力にも，キーのリード / ライトに用いるのと同じビットライン *key*[*k*−1:0] を用いる．一致比較器は 2.2 節で述べたものと同様である．マッチライン *match* はプリチャージされ，入力された検索キーが格納されているキーと一致しなければディスチャージされる．一致していれば，high に保たれる．

センスアンプ

RAM におけるビットラインほどではないが，マッチラインにも比較的多数のセルが接続されるため，その電位はゆっくりとしか変化しない．そこで，RAM におけるビットラインと同様，センスアンプが使用されることがある．

ただしマッチラインは，原理的に相補的な出力が得られなため，RAM におけるシングル・ビットラインと同様，シングル・エンドのセンスアンプを用いる必要がある．

ドライバ回路

同図 2.12 に示したロジックでは，マッチラインは NOR ゲートのドライバを介して，バリューを格納する RAM セル・アレイのワードラインに接続されている*．一致を検出**しなかった**行のマッチラインの値が low に変化したのを見計らってバリュー部へのアクセス・イネーブル $\overline{ve}$ をアサートすると，一致を検出**した**行のワードラインが high に遷移する．

2.2 節で述べたように，マッチラインは normally high であるため，このようなタイミング制御を行うドライバ回路が必要となることに注意されたい．RAM セル・アレイのワードラインは noramlly low でなければならず，normally high であるマッチラインを直接接続することはできない．

2.4.2 多ポートの CAM

命令スケジューリングで用いられる CAM は，前項で述べた命令スケジューリング用 RAM の場合と同様，多数のポートを持つ．ポート数は，大抵の場合，命令のディスパッチ幅，または，発行幅の増加関数で与えられる．また，1 つの CAM に対するライト・アクセスとサーチ・アクセスは，通常，パイプラインの別のステージで同時に行われるため，ライト・ポートとサーチ・ポートを別個に持つ．これらの理由により，命令スケジューリング用 CAM のポート数は，大きいものでは 10 を越える．

図 2.9 (p. 35) (c)，(d) に，2 つのサーチ・ポートと 1 つのライト・ポートを持つマルチポートの CAM セルのロジックを示す．サーチ・ポートにおける，検索キーを入力するための相補的なラインは，**サーチライン** (search-line) と呼ばれることがある．

同図 2.9 (c) は，一致比較器として，2.2 節で述べたドミノ XOR ゲートを使用している．4T セル内のドライバ・トランジスタが，ドミノ XOR ゲートの一方の入力に対する *n*MOS ゲートを兼用している．同図 2.9 (d) は，RAM セルのビットライン・ドライバの場合と同様に，マッチライン・ドライバを別体化したものである．

2.4.3 RAM との関係

一般に,『CAM は（RAM より）遅い』と言われる．実際，5 章以降で述べる命令スケジューリング方式のいくつかでは，CAM を RAM に変換すべく工夫を凝らしている．また，3.4 節で述べるように，写像を保持するマッピング・テーブルでは，一般に，RAM ベースの構成と CAMベースの構成が互いに交換可能である．そこで本項では，RAM と CAM と遅延の差について考察する．

---

* このロジックでは，最初にバリューを格納する際にも，まずキーを格納した上で，連想アクセスによって RAM 部にバリューを書き込む必要があり，使い勝手が悪い．最初にキーとバリューのペアを同時に書き込む機能があるとよい．

一般的な RAM と CAM の比較

一般に『CAM は RAM より遅い』と言うときには，m word × n bit の RAM に対するアクセスと，バリュー部が同じく m word × n bit である CAM に対する連想アクセスとを比較していると思われる．

その場合，RAM の遅延は行デコーダの遅延と RAM セル・アレイの遅延に分けられ，CAM の遅延は CAM セル・アレイの遅延と同じく RAM セル・アレイの遅延に分けられる．ここで，双方の RAM セル・アレイの語構成はともに m word × n bit で同じであるから，その遅延も互いに等しい．

したがって，この場合の『CAM は RAM より遅い』とは，CAM セル・アレイの遅延は RAM の行デコーダの遅延より大きいという，ある意味当然のことを述べているに過ぎない．ここで本当に比較したかったのは，双方において中心的な役割を果たす，RAM セル・アレイと CAM セル・アレイではないだろうか．

命令スケジューリング・ロジックにおける RAM と CAM の比較

実際，命令スケジューリング・ロジックにおける RAM と CAM の交換では，5 章以降で詳しく述べるように，RAM と CAM ではなく，RAM セル・アレイに対するリード・アクセスと CAM セル・アレイに対するサーチ・アクセスが互いに交換可能になる．より具体的な考察はそれらの章に譲り，本項では RAM セル・アレイと CAM セル・アレイの違いを定性的に明らかにしておく．

同図 2.9 (a)，(b) に示した RAM セルのロジックと，同図 (c)，(d) に示した CAM セルのロジックを比較すると，RAM のリード・ポートと CAM のサーチ・ポートでは，入出力ラインがちょうど逆になっていることが分かる．すなわち，RAM のリード・ポートでは，行方向に走るワードラインを入力として，セル内に配置されたドライバが ON となり，列方向に走るビットラインがドライブされる（横 → 縦）一方で，CAM のサーチ・ポートでは，列方向に走るサーチラインを入力として，行方向に走るマッチラインがドライブされる（縦 → 横）．表 2.2 に，RAM セル・アレイと CAM セル・アレイの入出力ラインをまとめる．

そのため，セルの回路上では，アクセス・トランジスタのゲート電極とドレイン電極を局所的に付け換えることによって，RAM セルと CAM セルを互いに交換することができる．同図 2.9 の (a) と (c)，(b) と (d) を比較されたい．したがって，ゲートのファン・インやゲート段数で表される論理回路の複雑さには，RAM セル・アレイと CAM セル・アレイで違いは全くない．RAM セル・アレイと CAM セル・アレイの遅延の差は，それぞれの入出力ラインの長さの差によって生じるのである．

構成が同じ RAM セル・アレイと CAM セル・アレイを比較すると，むしろ CAM セル・アレ

セル・アレイ	入力ライン （方向）	出力ライン （方向）
RAM	ワードライン （行）	ビットライン （列）
CAM	サーチライン （列）	マッチライン （行）

表 2.2: RAM セル・アレイと CAM セル・アレイの入出力ライン

イの方が高速であることが多い．構成が同じであれば，両者の入出力ライン長の和は互いに等しい，すなわち,（ワードライン長）+（ビットライン長）=（サーチライン長）+（マッチライン長）である．しかし，面積に関する制約が厳しいセル内部のドライバでドライブされる出力ラインが短い方が，遅延の上では有利である．通常，ビット幅よりワード数の方が多く，セル・アレイは列方向に長い形状となる．そのため，出力ラインが行方向に走る CAM セル・アレイの方が有利になるのである．

したがって，単に CAM セル・アレイを RAM セル・アレイに置き換えたというだけでは，どれほどの高速化が達成されたのか全く明らかでない．『RAM は CAM より速い』というのは大抵の場合正しいが,『RAM セル・アレイは CAM セル・アレイより速い』かどうかは，詳しく評価してみなければ分からない．少なくとも，それぞれの入出力ラインの長さを定量的に評価する必要がある．

2.5 CMOS のスケーリング

回路の遅延は，**ゲート遅延** (intrinsic gate delay) と**配線遅延** (wire delay) とに分けて考えることができる．純粋なゲート，あるいは，純粋な配線などというものは LSI 上に存在しないから，両者を厳密に区別することはできない．しかし，例えば，1 個の NAND ゲートをドライブする NOR ゲートの遅延はほぼゲート遅延だけからなり，RAM のビットライン，ワードラインなどの長い配線の遅延は，ドライバのゲート遅延と配線遅延からなると考えることができる．

CAM の消費電力

図 2.12 のような CAM では，その消費電力が問題にされることがある．通常の CAM の使用法では，連想アクセスで検索キーに一致するのはたかだか 1 行である．残りのほとんどすべての行では一致が起こらず，マッチラインはディスチャージされることになる．すなわち，連想アクセスの際には，ほとんどすべての行のマッチラインに対してプリチャージとディスチャージが行われ，その容量分の電力が消費されることになる．

本文で述べているように，CAM のマッチライン は RAM のワードラインと対比することができる．アクセスされる行だけで充放電が行われるワードラインと比較すると，ほとんどすべての行で充放電が行われるマッチラインの消費電力の多さが分かる．

なお，CAM のサーチライン は RAM のビットラインと対比することができるが，電力消費の状況は双方で変わりがない．双方とも，アクセスの際には，すべての列において 2 レールのラインの一方がディスチャージされる．

ゲート遅延

ゲート遅延は，以下の式で表すことができる：

$$Delay_{gate} = \frac{C_L \times V}{I} \tag{2.1}$$

C_L はゲートの負荷容量，V は供給電圧，I は平均の充放電電流である．I は，ゲートを形成するデバイスの飽和ドレイン電流 I_{dsat} の関数で与えられる．

最小加工寸法がスケーリングされるにしたがい，消費電力を管理可能なレベルに抑えるため，電源電圧 V を低下させる必要がある．そのため，電源電圧 V は任意にスケーリングすることはできず，最小加工寸法とは別のスケーリング曲線を描く．[52] によれば，サブミクロンのデバイスでは，スケーリング・ファクタを S，電源電圧のスケーリング・ファクタを U として，C_L，V，I は，それぞれ，$1/S$，$1/U$，$1/U$ にスケーリングされる．したがって，ゲート遅延のスケーリング・ファクタ S_{gate} は

$$\begin{aligned} S_{gate} &= \frac{(1/S) \times (1/U)}{1/U} \\ &= \frac{1}{S} \end{aligned} \tag{2.2}$$

となる．

配線遅延

配線遅延 (intrinsic RC delay) は，分布 RC 回路の終端における遅延の 1 次の近似として，以下の式で表すことができる：

$$Delay_{wire} = \frac{1}{2} \times R \times C \times L^2 \tag{2.3}$$

R，C は，配線の単位長さあたりの抵抗と容量，L は配線の長さである．

また，R，C は，それぞれ，以下の式で表すことができる：

$$R = \frac{\rho}{w \times h} \tag{2.4}$$

$$C = 2 \times k \times \epsilon_0 \times \frac{h}{w} + 2 \times k \times \epsilon_0 \times \frac{w}{t} \tag{2.5}$$

ρ は配線の抵抗率，w は配線の幅，h は配線の高さ，k は層間絶縁膜の比誘電率，ϵ_0 は真空の誘電率，t は配線間の絶縁膜の厚さ ($\simeq h$) である．

式 2.5 の第 1 項は，左右に隣接する配線との間に生じる容量を (fringe capacitance) 表し；第 2 項は，上下に隣接する配線，あるいは，基盤との間に生じる容量 q(paralle-plate capacitance) を表している．

McFarland と Flynn は，ローカル配線に対するスケーリングの方法を調べ，準理想的な (quasi-ideal) スケーリングが将来のディープ・サブミクロンのテクノロジによく合致すると結論づけている [53]．準理想的なスケーリングでは，水平方向には理想的なスケーリングを，垂直

方向にはよりゆっくりとしたスケーリングを施す．実際，これまでの数世代の間，配線幅が $1/S$ にスケーリングされてきたのに対して，配線層の厚さ h，t はほぼ一定であった．

したがって，最小加工寸法が縮小されるにつれ，第 1 項が支配的になることが分かる．Rahmat らは，最小加工寸法が 100nm より小さい場合には，第 1 項が C 全体の 90% を占めることを示している [54].

そこで，C として第 1 項のみを考慮すると，式 2.3 の右辺は，式 2.4，2.5 をそれぞれ代入して：

$$\begin{aligned} \frac{1}{2} \times R \times C \times L^2 &= \rho \times k \times \epsilon_0 \times \left(\frac{L^2}{w^2} + \frac{L^2}{h \times t} \right) \\ &\simeq \rho \times k \times \epsilon_0 \times \frac{L^2}{w^2} \end{aligned} \tag{2.6}$$

となる．結局，配線遅延のスケーリング・ファクタ S_{wire} は：

$$\begin{aligned} S_{wire} &= (1/S)^2/(1/S)^2 + (1/S)^2/(1 \times 1) \\ &= 1 + \frac{1}{S^2} \\ &\simeq 1 \end{aligned} \tag{2.7}$$

となる．

式 2.2 と 2.7 から，最小加工寸法のスケーリング・ファクタ $1/S$ に対して，ゲート遅延は $1/S$ にスケーリングされる一方で，配線遅延は一定のままであることが分かる．このことは，配線遅延の影響が次第に増加していき，最終的には遅延全体を支配するようになることを示している．

実際には，以下の 2 つの理由により，事態はより深刻である：第 1 に，すべての配線が完全に $1/S$ に短縮されるわけではない．第 2 に，クロックなどの一部のグローバル配線の長さは，ダイ・サイズの増加にしたがって，むしろ長くなる傾向にある．

なお，銅配線，低誘電率層間絶縁膜の採用は配線遅延の短縮に一定の効果があるが，式 2.6 からも分かるように，配線遅延の影響の増大を数世代分先延ばしにする効果があるものの，本質的な解決にはならない．

第3章　Out-of-Order命令スケジューリング

本章では，out-of-order スーパースカラ・プロセッサの命令スケジューリングについて詳しく述べる．Out-of-order スーパースカラ・プロセッサは，VLIW プロセッサであれば必要のない，out-of-order 命令スケジューリングのためのロジックを持つ．本章の目的は，LSI の微細化にともなって，そのようなロジックのどの部分がクリティカルになるか，すなわち，プロセッサのクロック速度を制限する可能性があるのか，明らかにすることである．

以下まず，3.1 節では，out-of-order 命令スケジューリングの原理について述べる．Out-of-order 命令スケジューリングは，1. *rename*，2. *dispatch*，3. *wakeup*，4. *select*，5. *issue* と呼ぶ，5 つのフェーズに分けることができる．3.1 節では，これらのフェーズについて説明する．

これらの 5 つのフェーズは，実際には，命令パイプラインにおいてパイプライン的に動作する必要がある．各フェーズのパイプライン動作については，3.2 節で改めて説明する．これら 5 つのフェーズのうちのどのフェーズのロジックがクリティカルになるか考察するには，まず，各フェーズの**パイプライン化可能性 (pipelinability)** を考慮する必要がある．一般にあるフェーズの遅延がクリティカルである場合には，1 章で述べたように，パイプライン化を施すことによって非クリティカルにすることができる．しかし，すべてのフェーズがパイプライン化可能であるわけではない．同 3.2 節では，5 つのフェーズのうち，3. *wakeup* と 4. *select* の 2 つのフェーズが実際上パイプライン化不能であることを示す．

したがって，これら 2 つのフェーズのロジックの遅延がクリティカルである場合，実際にプロセッサのクロック速度を制限することになる．3.4 節で各フェーズを実現するロジックについて説明するが，興味の中心は，3. *wakeup* と 4. *select* フェーズのロジックのどちらがクリティカルであるのかという点にある．3.6 節の最後には，3. *wakeup* ロジックは配線遅延を多く含み，LSI の微細化にともなってよりよりクリティカルになることを示す．

3.1　Out-of-Order 命令スケジューリングの原理

本節では，out-of-order 命令スケジューリングの原理について説明する．以下まず，3.1.1 項で，スーパースカラ・プロセッサの基本的な構造についてまとめ，3.1.2 項以降で，命令パイプラインの上流から順に，命令スケジューリングの処理について説明する．

3.1.1　スーパースカラ・プロセッサの基礎

命令ウィンドウ

動的命令スケジューリングを行う ILP プロセッサは，スケジューリングの対象となる命令を格納する論理的なバッファを持つ．フェッチ (fetch) された命令はすべて一旦このバッファに格納され，各サイクルでこのバッファに格納されている命令がスケジューリングの対象となる．すべての命令の中でこのバッファに『見えている』範囲の命令のみがスケジューリングの対象となるため，このバッファは**命令ウィンドウ** (instruction window) と呼ばれる．

ディスパッチ，発行

フェッチされた命令を命令ウィンドウに格納することを，命令を命令ウィンドウに**ディスパッチ** (**dispatch**) するという．命令ウィンドウ内から実行可能な命令を見つけ出し，それらのうちのどれを実際に実行するのか決定することを，命令を**スケジューリング** (**scheduling**) するという．また，スケジューリングされた命令を命令ウィンドウから読み出して実行ユニットに送ることを，命令を実行ユニットに対して**発行** (**issue**) するという*.

同時にフェッチ，ディスパッチ，および，発行できる命令の数を，それぞれ**フェッチ幅** (**fetch width**)，**ディスパッチ幅** (**dispatch width**)，および，**発行幅** (**issue width**) と言う．本稿では，それぞれを，記号 $\boldsymbol{FW}$，$\boldsymbol{DW}$，および，$\boldsymbol{IW}$ で表す．$FW = DW = IW$ であるときには，それらの値を特に**ウェイ数** (way number) と言い，例えば『4-way スーパースカラ・プロセッサ』などと言う†.

フロントエンド，バックエンド

命令パイプラインは，バッファである命令ウィンドウを境に，上流と下流に分離 (decouple) されている‡．本稿では，命令パイプラインの命令ウィンドウより上流を**フロントエンド** (**frontend**)，命令ウィンドウおよび，命令ウィンドウより下流を**バックエンド** (**backend**) と呼ぶことにする．命令ウィンドウ自体をバックエンドに含めるのは，その方が議論に都合がよいからである．命令パイプラインのステージで言えば，命令フェッチからディスパッチまでがフロントエンド，それより下流のステージがバックエンドに属する．

プログラム・オーダ

IA-32 (x86) [55]，MIPS [23]，SPARC [56]，PA-RISC [57]，Alpha [33] など，現存するほとんどすべてのスーパースカラ・プロセッサが準拠する命令セット・アーキテクチャ (Instruction Set Architecture: **ISA**) は，逐次的な実行モデルを採用している．これらの ISA は，メモリ上の命令の並びと分岐命令の実行結果によって，命令の実行順序を規定する．この順序を**プログラム・オーダ** (program order) と言う．プログラム・オーダは，命令間の全順序 (total order)

---

* これとは逆に，特に in-order スーパースカラ・プロセッサにおいて，命令を（命令ウィンドウから）実行ユニットへ送る処理をディスパッチと呼ぶこともある．

† 最近では，$FW = DW < IW$ とすることが普通で，スーパースカラ・プロセッサの規模をウェイ数で代表することは難しくなった．

‡ このようなアーキテクチャをデカップルド (decoupled) アーキテクチャと呼ぶことがある [23] が，ほとんどのスーパースカラ・プロセッサは何らかの命令バッファによって『デカップル』されている．

関係を規定する．プログラム・オーダにしたがって命令を 1 つずつ実行する，逐次的なマシン・モデルを規定することができる．これらの ISA に準拠するプロセッサは，この逐次的なモデルと同一の実行結果を与えることを要求される．スーパースカラ・プロセッサの場合には，この制約の範囲内で動的命令スケジューリングを行う必要がある．

In-Order，Out-of-Order

プログラム・オーダ上で，先行する命令 $\boldsymbol{I_{pred}}$ と後続の命令 $\boldsymbol{I_{succ}}$ を考える．I_{pred} と I_{succ} は，連続していてもしていなくてもよい．複数の命令を同時に処理する ILP プロセッサでは，I_{pred} と I_{succ} が同時に命令パイプラインの同じステージに進むことがある．更に，プロセッサによっては，I_{pred} があるステージに留まっているのに，I_{succ} がその下流のステージに進んでしまうことがある．このように，I_{succ} が I_{pred} を追い越して下流のステージに進んでいるとき，これらの 2 命令は **out-of-order** に処理されていると言う．その反対に，I_{succ} が I_{pred} より上流か，あるいは同じステージにあって，追い越しが起こっていないとき，これらの 2 命令は **in-order** に処理されていると言う．

処理が out-of-order になることを許すスーパースカラ・プロセッサを out-of-order スーパースカラ・プロセッサ，許さないスーパースカラ・プロセッサを in-order スーパースカラ・プロセッサという*.

ただし，out-of-order スーパースカラ・プロセッサであっても，命令パイプラインのすべてのステージで処理が out-of-order になるわけではない．Out-of-order スーパースカラ・プロセッサでも，フェッチされた命令を命令ウィンドウにディスパッチするまでは in-order に処理する．その後，命令は命令ウィンドウ内でスケジューリングされ，命令ウィンドウから発行され，実行される訳だが，out-of-order スーパースカラ・プロセッサは専らこの部分を out-of-order に行うのである．すなわち，out-of-order スーパースカラ・プロセッサでは，命令は，**フロントエンドでは in-order に，バックエンドでは out-of-order に，処理される**のである．

3.1.2 レジスタ・リネーミング

Out-of-order スケジューリングを効率よく行うためには，**レジスタ・リネーミング**が重要である．Out-of-order に命令を発行する場合，逆依存関係が主な関心事となる．というのは，プロセッサが普通に動作している間は，発行を止められた命令のソース・オペランドが後続命令によって破壊されるという状況が多いからである [18].

以下，図 3.1 に示したコードを例に，レジスタ・リネーミングについて説明する．同図中，label はラベルを表す．同図には，ラベル I_x，I_l，I_r，および，I_c が付された，連続する 4 つの命令が示されている．また，opcode はオプコードを，opD，および，opL/opR は，それぞれ，デスティネーション，および，左 / 右のソース・オペランドを表す．$0，$1, ··· は，レジスタ番号である．

---

* Out-of-order スーパースカラ・プロセッサをダイナミック－と呼ぶことがある．しかし，in-order スーパースカラ・プロセッサをスタティック－と呼ぶことは稀である．

label	*opcode*	*opD*		*opL*	*opR*
I_x :	op_x	\$8	=	\$7,	0
I_l :	op_l	\$9	=	\$8,	1
I_r :	op_r	\$8	=	\$7,	2
I_c :	op_c	\$8	=	\$9,	\$8

label	*opcode*	*prD*		*prL*	*prR*	*immed*
I_x :	op_x	%1	=	%0,		0
I_l :	op_l	%2	=	%1,		1
I_r :	op_r	%3	=	%0,		2
I_c :	op_c	%4	=	%2,	%3	

図 3.1: レジスタ・リネーミング前（左）と後（右）のコード

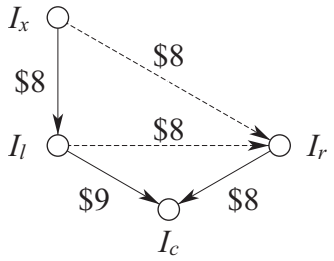


図 3.2: 図 3.1 のコードのデータ・フロー・グラフ

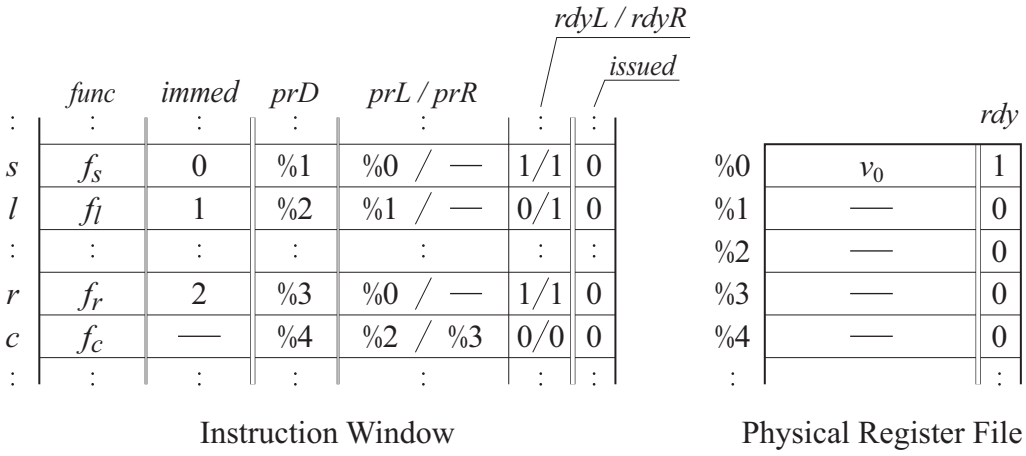


図 3.3: 命令ウィンドウ

図 3.1 のコードのデータ・フロー・グラフを図 3.2 に示す．同図では，ノードが命令を，実線のエッジがフロー依存を，破線のエッジが逆依存，出力依存を表している．エッジに付された記号 $7〜$9 は，その依存の原因となるレジスタ番号を示す．同図から分かるように，I_x と I_r，および，I_l と I_r の間には，それぞれフロー依存がない．したがって，データ・フローの観点からは，I_r は I_x，あるいは，I_l と同時に実行可能である．しかし，I_x と I_l は同一のレジスタ $8 をそれぞれ定義しており，出力依存の関係にある．また，I_l と I_r は，同一のレジスタ $8 をそれぞれ使用，定義しており，逆依存の関係にある．

逆依存，出力依存は，同一のレジスタ $8 をそれぞれ使用，および，定義しているために生じている．したがって，各命令の実行結果をそれぞれ別のロケーション (location) に保存 (save) することで，この問題を解消することができる．Out-of-order スーパースカラ・プロセッサはこの目的のため，**物理レジスタ**と呼ぶ物理的なロケーションを持つ．ISA が定義するレジスタ，いわゆるアーキテクチャ・レジスタは，この物理レジスタと区別して，特に**論理レジスタ**と呼ばれる．以降では，論理レジスタ番号 $0，$1, ··· と区別するため，物理レジスタ番号は %0，%1, ··· と表すことにする．

物理レジスタは，各命令のデスティネーション・オペランドに動的に (dynamically) 割り当てられる．空いている物理レジスタは**プール** (pool) によって管理されており，命令がフェッチされると，空いている物理レジスタが 1 つプールから取り出され，割り当てられる．

各命令は，そのデスティネーション・オペランドに割り当てられた物理レジスタに実行結果を格納する．その結果，実行に必要なソース・オペランドは，依存元の命令のデスティネーション・オペランドに割り当てられた物理レジスタから読み出すことになる．例えば，命令 I_x のデスティネーション・オペランド $8 に物理レジスタ %1 が割り当てられるとしよう．I_x は，実行結果を物理レジスタ %1 に格納することになる．命令 I_l の左ソース・オペランド $8 は命令 I_x のデスティネーション・オペランドを指しているため，I_l はこの %1 から左ソース・オペランドのデータを読み出すことになる．

したがって，必要なソース・オペランド・データがどの物理レジスタに格納されるか予め知っておくと便利である．逆に言えば，この物理レジスタの番号さえ知っておけば，論理レジスタ番号は忘れてしまって構わない．このことは，あたかも各命令のオペランドの論理レジスタ番号を物理レジスタ番号に付け換えた (rename) と見なすことができる．そのためこの処理は**レジスタ・リネーミング**と呼ばれるのである．

図 3.1 左に示したコードが同図右のようにリネームされる過程は，以下のようになる．命令 I_x のデスティネーション・オペランドには物理レジスタ %1 が，命令 I_l，I_r，I_c のデスティネーション・オペランドには，物理レジスタ %2，%3，%4 がそれぞれ割り当てられるものとする：

1. 命令 I_x には，物理レジスタ %1 が割り当てられるため，そのデスティネーション・オペランドである $8 は %1 にリネームされたと考える．
2. それに合わせて，命令 I_l の左ソース・オペランドの論理レジスタ $8 も，物理レジスタ %1 にリネームする．
3. 一方，命令 I_r のデスティネーション・オペランドは I_x と同じく $8 であるが，$I_r$ には物理レ

ジスタ %3 が割り当てられるため，以降 $8 は %3 にリネームし直されたと考える．

4. そのため，命令 I_c の左ソース・オペランドの論理レジスタ $8 は，%1 ではなく，%3 にリネームする．

このような一貫した手順によって，元のプログラムの意味を変えることなく，論理レジスタ番号を物理レジスタ番号にリネームすることができる．

この処理の結果，逆依存の元凶であった $8 は，それぞれ別の物理レジスタ，%1 と %3 にリネームされており，リネーミング後のコードでは逆依存は解消されている．

なお，レジスタ・リネーミングは in-order に行われていることに注意されたい．上記の説明では，命令 I_r に物理レジスタ %3 が割り当てられて『以降』，$8 は %3 にリネームされたと考えると述べた．『以降』とは，厳密には,『プログラム・オーダ上で下流の命令に対しては』，という意味である．このようなプログラム・オーダにしたがった処理は，in-order に実行することで自然に実現することができる（3.4 節）．

レジスタ・リネーミングは，実際のスーパースカラ・プロセッサでは，命令がフェッチされた直後に行われる．各命令は，レジスタ・リネーミングが施された後，命令ウィンドウにディスパッチされる．

3.1.3　命令ウィンドウ

各命令は，スーパースカラ・プロセッサのフロントエンドにおいて，in-order にレジスタ・リネーミングされた後，命令ウィンドウにディスパッチされる．図 3.3 左に，図 3.1 に示した 4 命令がディスパッチされた後の命令ウィンドウの様子を示す．命令 I_i $(i = x, l, r, c)$ は命令ウィンド

レジスタ・リネーミングとマシン・ステート

すべての論理レジスタが物理レジスタにリネームされてしまうのだとしたら，特定の論理レジスタの値が知りたい場合はどのようにすればよいのか？ 例えば，コンテクスト・スイッチ時に，OS はどのように論理レジスタの内容を待避すればよいのだろう？

答えは『普通にやればよい』だ．制御が OS に移った後，OS はユーザ・プロセスの論理レジスタの内容をスタックに待避するため，以下のようなコードを実行するだろう：

```
st   [$sp + OFFSET] = $1
```

このコードにおいて，$1 はその時点で最後に$1 に書き込みを行った命令に割り当てられた物理レジスタにリネームされる．その結果，ユーザ・プロセスの$1 の内容が正しく待避されるのである．

すなわち，プログラムの側では，物理レジスタの存在は一切気にする必要はないのである．

ウの *i* 番エントリに格納されたとする．命令ウィンドウ・エントリは，以下のフィールドを持つ：

func	オプコードをデコードして得た，実行ユニットの機能を示すコード．
immed	即値 (immediate)．
prD*，*prL*/*prR	それぞれ，デスティネーション，および，左 / 右のソース・オペランドの論理レジスタ番号からリネームされた物理レジスタ番号．
rdyL*/*rdyR	それぞれ，*prL*/*prR* によって示される物理レジスタ・ファイルの *rdy* フィールドの値．
issued	当該命令が発行されたことを表すフラグ．

rdyL/*rdyR* および，*issued* に関しては，後で詳しく述べる．なお実装によっては，この他にもいくつかのフィールドが付加されることがある．

以降では，これらのフィールドの値を，以下のように C 言語風に表すことにする：命令ウィンドウの *i* 番エントリを ***iwe*[*i*]**，そのフィールドの値を *iwe*[*i*].*prD* のように表す．また，紛れのない場合には，*iwe*[*i*].*prD* は，*prD*[*i*] のように省略することにする．例えば，命令 I_x のデスティネーション・オペランドに関しては，*prD*[*x*] = %1 のように表すことができる．

命令ウィンドウでは，元のプログラムには存在した以下の 2 つの情報が失われている：

1. **プログラム・オーダ**　図 3.1 に示した連続する 4 命令は，図 3.3 では，命令ウィンドウの連続する 4 エントリには格納されていない．このように，命令ウィンドウ上の命令の格納場所は，必ずしもプログラム・オーダのとおりでなくてよい．後述するように，命令スケジューリング・ロジックは，プログラム・オーダに関する情報が全くなくても，命令を『正しく』スケジューリングすることができる．
2. **論理レジスタ番号**　命令は，命令ウィンドウにディスパッチされるときにはリネーミング済みであり，命令ウィンドウには，通常，論理レジスタ番号を格納するフィールドは存在しない．命令ウィンドウより下流では，専ら物理レジスタ番号に基づいて命令が処理されることになる．

命令スケジューリング・ロジックは，これらの失われた情報に依存せず，命令ウィンドウにある情報のみから命令スケジューリングを行うことになる．次項で述べるように，このことが out-of-order スケジューリングにとって重要である．

3.1.4 レジスタ・リネーミングと Out-of-Order 実行

Out-of-order スーパースカラ・プロセッサでは，各命令は，フロントエンドで in-order に，レジスタ・リネーミングされ，命令ウィンドウにディスパッチされる．それ以降，すなわちバックエンドでは，各命令は，out-of-order にスケジューリングされ，命令ウィンドウから発行され，実行される．3.1.2 項で述べたレジスタ・リネーミングの手続きは，単に逆依存を解消するだけでなく，この out-of-order な命令スケジューリングにおいて重要な役割を果たすことになる．

3.1.2 項で述べたレジスタ・リネーミングの手続きでは，フェッチされた各命令に対して，空いている物理レジスタが割り当てられる．このとき，この物理レジスタはデータが『**ない**』状態に初期化される．その後，当該命令が実行されて，その実行結果がこの物理レジスタに書き込まれると，この物理レジスタはデータが『**ある**』状態となる．データが『ある』物理レジスタ，あるいは，**利用可能 (ready)** であるという．逆に，データが『ない』物理レジスタは**利用不能 (not ready)** であるという*．当然のことながら，この物理レジスタを使用する命令は，物理レジスタが利用可能なら，すなわち，データが『ある』なら実行を開始することができる．

各命令は，その左 / 右のソース・オペランドに割り振られた物理レジスタの双方にデータが『ある』とき，すなわち，ソース・オペランドに割り振られた物理レジスタが利用可能であるとき，**実行可能 (ready)** であるという．実行可能な命令は，プログラム・オーダとは無関係に，すなわち，out-of-order に，実行を開始することができる；なぜなら，命令間の先行制約を引き起こすデータ依存は，以下のように解消され，また，守られるからである：

- 逆依存のような偽のデータ依存は，レジスタ・リネーミングによって既に解消されている．
- 真の依存であるフロー依存は，明らかに，ソース・オペランドに割り振られた物理レジスタにデータが『ある』ことによって守られる．

このように，out-of-order スーパースカラ・プロセッサのバックエンドでは，プログラム・オーダとは関係なく，データの『ある』/『なし』によって，プログラムの実行が進められる．すなわち，**out-of-order スーパースカラ・プロセッサのバックエンドは，制御駆動型ではなく，データ駆動型の駆動方式** [58] **に基づいていると考えることができる**．また，レジスタ・リネーミングは，バックエンドにおいて out-of-order なデータ駆動的計算が可能になるように，プログラムを変形する操作と考えることもできる．

3.1.5　命令スケジューリング

前項では，命令ウィンドウにディスパッチされた命令のうち，ソース・オペランドに割り振られた物理レジスタが利用可能であるものが実行可能であると述べた．しかし，では実際にいつ，どの命令を実行するのか決定する方法については触れなかった．本項では，実際にいつ，どの命令を実行するのか決定する方法，すなわち，具体的な命令スケジューリングの方法について説明する．

命令スケジューリングは，以下に述べる 2 つのフェーズからなる：命令をスケジューリングするにはまず，命令ウィンドウの中から実行可能な命令を検出しなければならない．このフェーズを ***wakeup*** と呼ぶ．次に，検出された実行可能な命令の中から，どの命令を実際に実行するのか選択する必要がある．このフェーズを ***select*** と呼ぶ．

---

* あるいは，full/empty)，有効 / 無効 (valid/invalid) と呼ばれることもある．

rdy*，*rdyL*/*rdyR

Wakeup フェーズでは，実行可能な命令を検出するために，物理レジスタの利用可能性を表すフラグを用いる．図 3.3 では，物理レジスタ・ファイル (physical register file) の各物理レジスタの横に示した 1 ビットのフィールド ***rdy*** が，この物理レジスタの利用可能性を示している．同図では，物理レジスタ %0 の *rdy* フィールドが最初からセットされている．一方，%1〜%4 の *rdy* フィールドは，*rename* フェーズにおいて物理レジスタが割り付けられるときに 0 に初期化されている．

また，同図 3.3 左に示した命令ウィンドウには，*prL*/*prR* によって示される物理レジスタが利用可能かどうかを表すそれぞれ 1 ビットのフィールド ***rdyL***/***rdyR*** がある．*rdyL*/*rdyR* は，上述した *rdy* と以下のような関係がある：

$$\begin{cases} rdyL[i] &= rdy[\ prL[i]\ ] \\ rdyR[i] &= rdy[\ prR[i]\ ] \end{cases} \quad (i = 0, 1, \cdots, WS-1) \tag{3.1}$$

rdyL/*rdyR* フィールドは，命令のディスパッチ時に，*rdy* フィールドの内容にしたがって初期化される．例えば，同図 3.3 では，命令 I_x の *prL* が %0 を指している，すなわち，$prL[x] = \%0$ であるので，$rdyL[x]$ は，$rdyL[x] = rdy[\ prL[x]\ ] = rdy[\%0] = 1$ に初期化される．命令 I_r の $rdyL[r]$ も同様に 1 に初期化される．なお，命令が即値オペランドを持つ場合には，*rdyR* フィールドを 1 に初期化しておけばよい．同図 3.3 では，命令 I_x，I_l，I_r が即値を持つから，$rdyR[x] = rdyR[l] = rdyR[r] = 1$ に初期化されている．

rdyL/*rdyR* フィールドの存在は，少々冗長に感じられるかも知れない．実際，*rdyL*/*rdyR* の値は，式 3.1 にしたがって，*rdy*，*prL*/*prR* から完全に再現することができる．それでも *rdyL*/*rdyR* フィールドが必要であるのは，実装上のことではあるものの，本質的な制約による．そのあたりの事情については 3.6 節で詳しく述べる．

***Wakeup* フェーズ**

命令ウィンドウにディスパッチされた命令は，左 / 右のソース・オペランドに割り振られた物理レジスタが利用可能であれば実行可能である．逆に，左 / 右のソース・オペランドに割り振られた物理レジスタのいずれかが利用可能ではない場合，実行を開始することができない．このような命令は，物理レジスタにデータが到着するのを待って，命令ウィンドウの中で『眠る (sleep)』ことになる．やがて依存元の命令が実行されると，この命令は『眠っ』ている命令を『起こす (wakeup)』ことになる．

Wakeup フェーズの処理は，実際には，*rdyL*/*rdyR* に基づいて行われる．図 3.3 の状態で，命令 I_x が実行された場合，*rdy*，および，*rdyL*/*rdyR* は以下のように更新される：

1. I_x が実行されると，$prD[x] = \%1$ より，その実行結果が物理レジスタ %1 に格納されると同時に，$rdy[\ prD[x]\ ] = rdy[\%1]$ もセットされる．
2. $prL[l] = \%1$ であるから，式 3.1 から，$rdyL[l] = rdy[\ prL[l]\ ] = rdy[\%1]$ である．したがって，$rdyL[l]$ もセットされる．

命令 I_l は即値を持ち，*rdyR*[*l*] は最初から 1 であったから，*rdyL*[*l*] / *rdyR*[*l*] ともに 1 となり，I_l は実行可能であると分かる．

なお，out-of-order スーパースカラ・プロセッサのバックエンドはデータ駆動型の駆動方式に基づいていると述べたが，この *wakeup* フェーズの処理はデータ駆動型計算機における**発火** (**firing**) とほぼ等価である [58].

***Select* フェーズ**

実行ユニットなどの計算資源の数には限りがあるから，実行可能な命令のすべてがすぐさま実行できる訳ではない．*Select* フェーズは，*wakeup* フェーズで検出された実行可能な命令の中から実際に発行する命令を選択するフェーズである．*Select* フェーズは，いつどの命令を実行するか最終的に決定するフェーズであり，狭義の命令スケジューリングということができる．

実行する命令を選択するにあたっては，以下の点を考慮する必要がある：

1. **構造ハザードの解消**　*Rename* フェーズで偽の，*wakeup* フェーズで真のデータ・ハザードが，それぞれ解消されている．*Select* フェーズの主たる役割は，実行ユニットなどの演算資源に関する構造ハザードを解消することにある．演算資源の空き状況を見て，同一の演算資源を要求する命令が複数あれば，それらを調停する．

2. **IPC**　データ・ハザードは既に解消されているから，構造ハザードさえ解消すれば，実行可能な命令のどれを選択してもプログラムは正しく動作する．ただし，高い IPC のためには，何らかの戦略に基づいて適切な命令を選択することが必要である．

   そのような戦略のうち，最も単純なものはできるだけ古い命令を選ぶというものであろう．より高度なものとしては，プログラムのクリティカル・パス上の命令を優先するような方式も提案されている [59, 60, 61, 14, 15].

3.1.6　Out-of-Order スケジューリングの 5 フェーズ

前項までの議論から，out-of-order スーパースカラ・プロセッサの命令スケジューリングは，以下の 5 つのフェーズに分解できる：1. ***rename***，2. ***dispatch***，3. ***wakeup***，4. ***select***，5. ***issue***．これらのフェーズは，命令パイプラインにおいては，命令フェッチ / デコード (decode) から，実行 (execution) の間にある．

図 3.1，3.2 に示したコードにおいて，2 番目の命令 I_l が実行されるタイミングに着目すると，命令の処理の流れは以下のように説明できる．図 3.3 を同時に参照されたい：

1. ***Rename***　I_l にレジスタ・リネーミングが施され，I_l は I_x の実行結果が物理レジスタ %1 に書き込まれることを知る (*prL*[*l*] = %1).

2. ***Dispatch***　その後 I_l は，命令ウィンドウに格納される．図 3.3 は，このときの状態を示している．

位置	処理順序	計算方式	フェーズ
フロントエンド	in-order	制御駆動	1. *rename*，2. *dispatch*
バックエンド	out-of-order	データ駆動	3. *wakeup*，4. *select*，5. *issue*

表 3.1: 命令スケジューリングのフェーズ

I_l は即値を持つので，*rdyR* は 1 に初期化される (*rdyR*[l] = 1).

prL[l] で示される物理レジスタ %1 にデータがまだ『なく』，*rdyL* は 0 に初期化される (*rdyL*[l] = *rdyL*[l] = *rdy*[*prL*[l]] = *rdy*[%1] = 0). したがって，命令 I_l は実行可能ではない.

3. *Wakeup* やがて命令 I_x が実行されると，その *prD* で示される物理レジスタ %1 にデータが到着する (*rdy*[*prD*[x]] = *rdy*[%1] = 1).

命令 I_l の *prL* が同じく %1 であるので，その *rdyL* がセットされる (*rdyL*[l] = *rdy*[*prL*[l]] = *rdy*[%1] = 1). すると I_l は，*rdyL*/*rdyR* がともにセットされるので，実行可能になる.

4. *Select* I_l の使用する演算資源に空きがあれば，I_l が選択される.

5. *Issue* すると，その情報が命令ウィンドウから読み出され，実行ユニットに送られる.

I_l が実行されると，その結果は *prD*[l] で示される物理レジスタ %2 に書き込まれ，今度は %2 を参照する命令 I_c が *wakeup* されることになる.

3.1 節のまとめ

Out-of-order スーパースカラ・プロセッサの命令スケジューリングの骨子は，以下のようにまとめることができる:

- Out-of-order スーパースカラ・プロセッサの命令スケジューリングは，以下の 5 つのフェーズに分解できる: 1. *rename*，2. *dispatch*，3. *wakeup*，4. *select*，5. *issue*. これらのフェーズは，命令パイプラインにおいては，命令フェッチ / デコードから，実行の間にある.
- これらのフェーズのうち，命令フェッチから，1. *rename*，2. *dispatch* までは，フロントエンドに; 3. *wakeup*，4. *select*，5. *issue*，および，実行ステージ以降はバックエンドに属する.
- フロント，および，バックエンドでは，端的に言えば，以下のような処理が行われる:

  フロントエンド 制御駆動方式に従い，in-order に，プログラムをデータ駆動的実行が可能なように変換する.

  バックエンド データ駆動方式に従い，out-of-order に命令を実行する.

以上を，表 3.1 にまとめる.

3.2　Out-of-Order 命令スケジューリングと命令パイプライン

前節で述べたように，out-of-order スーパースカラ・プロセッサの命令スケジューリングは，5 つのフェーズからなる．本節以降では，どのフェーズのロジックがクリティカルになるのか，すなわち，どのフェーズのロジックの遅延がシステムのクロック速度を制限する可能性があるのか考察する．それにはまず，各フェーズの**パイプライン化可能性** (pipelinability) を考慮する必要がある．

1 章で述べたように，あるフェーズの遅延がクリティカルである場合には，パイプライン化を施すことによって非クリティカルにすることができる．しかし，5 つのフェーズのすべてがパイプライン化可能である訳ではない．あるフェーズをパイプライン化することによって IPC がひどく悪化する場合，そのフェーズは実際上パイプライン化不可能である．

本節では，各フェーズのパイプライン化可能性について述べる．まず，3.2.1 項で，命令パイプライン中の 5 フェーズの位置づけについてまとめ，3.2.4 項でそれらのパイプライン化可能性について述べる．

3.2.1　命令パイプラインにおける 5 フェーズ

前項で述べた out-of-order スケジューリングの 5 フェーズは，実際にはスーパースカラ・プロセッサの命令パイプラインの中で，パイプライン動作する必要がある．各フェーズは，命令パイプラインのステージと，必ずしも 1 対 1 に対応するわけではない．その遅延が十分に短ければ，1サイクル未満で済む．逆に，上述したように，あるフェーズをパイプライン化した場合には，そのフェーズには複数のパイプライン・ステージが割り当てられることになる．

図 3.4 に，命令パイプラインの例を示す．命令パイプラインの構成は，MIPS R10000 プロ

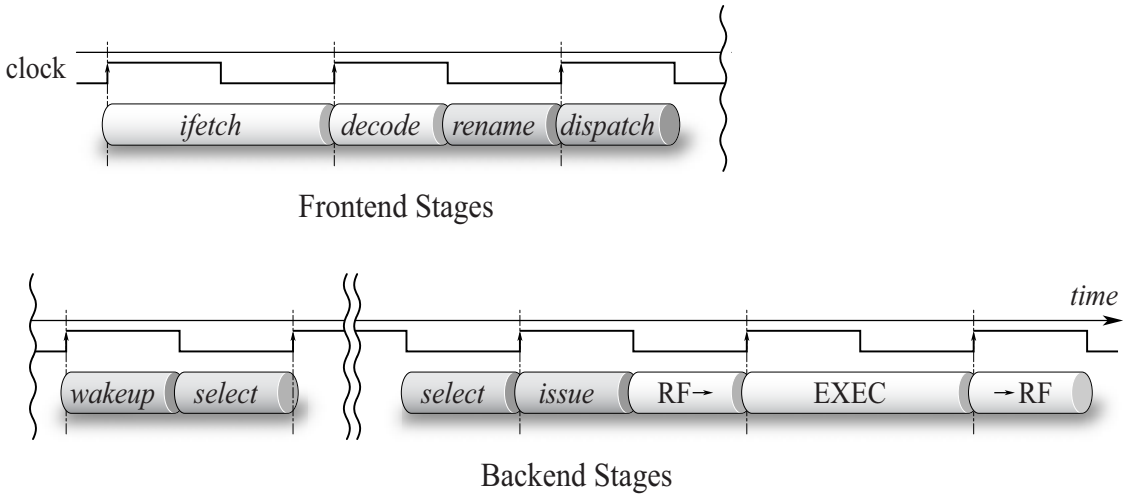


図 3.4: 命令パイプライン

セッサのそれに準ずる [23]．同図には，1つの命令が命令パイプラインの各ステージを通過していく様子を上下二段に分割して描かれている．上段がフロントエンド，下段がバックエンドにあたる．同図中，IF，および，ID は，命令フェッチ，および，デコードを表す．EXEC は実行を，RF → と → RF は物理レジスタ・ファイルに対する読み出しと書き戻しを表す．

図 3.4 から分かるように，R10000 では，5 フェーズのそれぞれに半サイクルずつを充てている．

スーパースカラ・プロセッサの命令パイプラインの特徴は，命令が命令ウィンドウによってバッファリングされるところにある．命令ウィンドウの中で何サイクル滞留するかは，プログラムを実行してみないと分からない．図 3.4 で言えば，*dispatch* されてから *wakeup* されるまで，また，*wakeup* されてから実際に *select* されて *issue* されるまで，何サイクル経過するか分からない．

3.2.2 スーパースカラ・プロセッサの命令パイプラインの乱れ

スーパースカラ・プロセッサにおいて，同時にフェッチされた（最大）FW 個（3.1.1 項参照）の命令の集合を，**フェッチ・グループ** (fetch group) と言う．フェッチ・グループは，VLIW プロセッサの長命令語 (VLIW) と対比することができる．ただし，1つの長命令語に属する命令が最後まで同時に処理されるのとは異なり，フェッチ・グループに属する命令は最後まで同時に処理されるわけではない．動的命令スケジューリングによって，フェッチ・グループは分割 (break) され，別々のフェッチ・グループに属していた命令が同時に実行されることになる．そのため，VLIW プロセッサのそれと比べると，スーパースカラ・プロセッサの命令パイプラインの動きは複雑になる．

ただし，このフェッチ・グループの分割は，命令ウィンドウにおいてのみ行われる．そこで，スーパースカラ・プロセッサでは，命令パイプラインをフロントエンド・パイプラインとバックエンド・パイプラインに分けて考えると分かりやすい．フロントエンド・パイプラインは *dispatch* フェーズで終わり，バックエンド・パイプラインは *issue* フェーズから始まる．フェッチ・グループの分割は命令ウィンドウにおいてのみ行われるので，フロントエンド・パイプライン，および，バックエンド・パイプライン，それぞれの動きは，VLIW プロセッサのそれと比べても，むしろ単純化することができる．それぞれ，基本的にはパイプラインの乱れは生じない．

フロントエンド・パイプライン

スーパースカラ・プロセッサのフロントエンド・パイプラインでは，通常，物理レジスタ，あるいは，命令ウィンドウの空きエントリが不足するとき，またそのときのみ，処理を進めることができなくなる．これは，構造ハザード (structural hazard) の一種である．フロントエンド・パイプラインでは，命令は実行されるわけではないので，データ・ハザードは生じない．また，命令デコーダなどの各ステージに固有の計算資源に関しては，FW 個の命令を同時に処理できるだけ用意するのが普通である．したがって，物理レジスタ，あるいは，命令ウィンドウのエントリ不足以外の構造ハザードも生じない．

物理レジスタ，あるいは，命令ウィンドウのエントリが不足した場合には，フロントエンド・パイプライン全体をストールさせればよい．例えば，物理レジスタ，あるいは，命令ウィンドウの空きエントリ数が n $(1 \leq n < FW)$ であるときには，そこでフェッチ・グループを分割して，最初の n 命令だけでも処理するという実装も不可能ではない．しかし，そのようにしたとしても，複雑さの増加に見合う性能向上が得られる可能性は低い．このような状況は，物理レジスタ，あるいは，命令ウィンドウのエントリ数をそれぞれ FW 個ほど増やすだけで回避することができる．それでもストールが頻発するようであれば，そもそもプロセッサ全体の設計のバランスを見直すべきである．実際，現存するスーパースカラ・プロセッサでは，フロントエンド・パイプラインにおいてフェッチ・グループを分割するような実装は極めて稀である．

結局スーパースカラ・プロセッサのフロントエンド・パイプラインは，全体がストールするだけで，インターロックすら発生しない．乱れのない，フェッチ・グループを処理する FW 命令幅の単一のパイプラインのように振る舞う．

バックエンド・パイプライン

スーパースカラ・プロセッサのバックエンドでは，実際に命令が実行されるため，データ・ハザードを回避する必要がある．また，以下のように，演算資源に関する制約が厳しいため，構造ハザードの回避の問題も複雑になる：

- 命令のタイプごとに異なる演算器を使用する必要がある．
- すべての演算器に対して物理レジスタ・ファイルのポートが割り当てられていないこともある．その場合，物理レジスタ・ファイルのポートを共有する演算器は同時には使用できない．
- いくつかの演算器はパイプライン化されておらず，毎サイクル命令を投入することができない．また，演算がいつ終了するか分からないこともある．

これらの制約にも関わらず，スーパースカラ・プロセッサのバックエンド・パイプラインの振る舞いは，基本的には VLIW プロセッサのそれと同程度に単純になる．

同時に発行される（最大）IW 個の命令の集合を**発行グループ (issue group)** と呼ぶことにしよう．発行グループは，フェッチ・グループと同様，VLIW プロセッサの長命令語と対比することができる．VLIW プロセッサでは，フェッチ・グループと発行グループが同じになると考えることができる．一方スーパースカラ・プロセッサでは，発行グループは，一般にフェッチ・グループとは異なる．

スーパースカラ・プロセッサのバックエンド・パイプラインは，発行グループを処理する IW 命令幅の単一のパイプラインのように振る舞う．バックエンド・パイプラインにおける発行グループは，フロントエンド・パイプラインにおけるフェッチ・グループと同様，分割されることはない．前述したように，フロントエンド・パイプラインは，物理レジスタ，あるいは，命令ウィンドウのエントリの不足によって，全体がストールすることがある．一方バックエンド・パイプラインは，基本的には，全体がストールすることすらない．

このようにスーパースカラ・プロセッサのバックエンド・パイプラインの動作が単純化されるのは，命令のスケジューリング時にすべてのハザードを解消するためである．すべてのハザー

ドが解消されている命令のみを *select* するため，一旦 *issue* された命令はハザードの回避のために更に待たされるということはない．

したがって，命令が *select* された時点で，その命令が実行されるまでの動作は**決定的** (deterministic) となる．図 3.4 で言えば，命令が *select* されると，次のサイクルに *issue* され，その次のサイクルに実行が開始されることが決定する．したがって，命令をスケジューリングすること，*select* すること，発行すること，および，実行（を開始）することは，この意味において等価であると言える．

さて，高い IPC を実現するためには，オペランド・バイパスを利用するなどして，互いに依存関係にある命令を連続して実行することが重要である．次節で述べるように，依存する命令を引き続くサイクルにスケジューリングするにあたって，バックエンド・パイプラインの動作が決定的であることを利用している．

3.2.3 命令ウィンドウのパイプライン動作

Out-of-order スケジューリングの 5 つのフェーズのうち，*wakeup* と *select* は，それらの間にフィードバック・ループが存在するため，それらのパイプライン動作は複雑になる．その他のフェーズには，そのようなフィードバック・ループがなく，単純にパイプライン動作させればよい．以下では，主に *wakeup* と *select* フェーズのパイプライン動作について詳しく述べる．

命令パイプラインにおける *Wakeup* と *Select* フェーズ

図 3.5 に，バックエンドのパイプライン動作を示す．同図には，図 3.1 に示した，フロー依存の関係にある 2 つの命令，I_x と I_l が描かれている．同図では，命令パイプラインのは以下のように動作している：

A_2　I_x が *select* される．I_l はまだ実行可能でないので，*select* の対象となっていない．

B_1　I_x が *issue* されると同時に，I_l が *wakeup* され，実行可能になる．

B_2　I_l も *select* の対象となり，実際に *select* されいている．

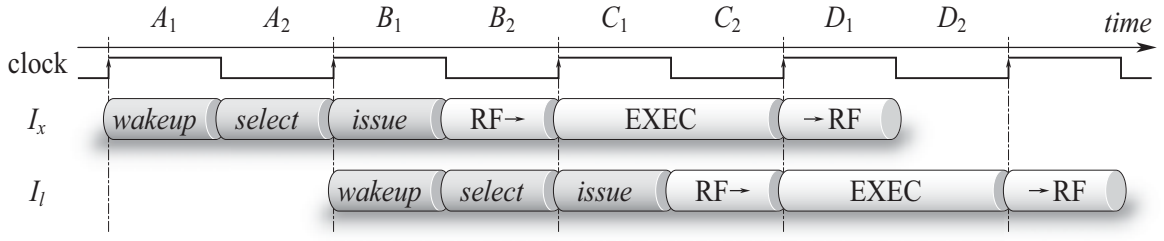


図 3.5: バックエンドのパイプライン動作

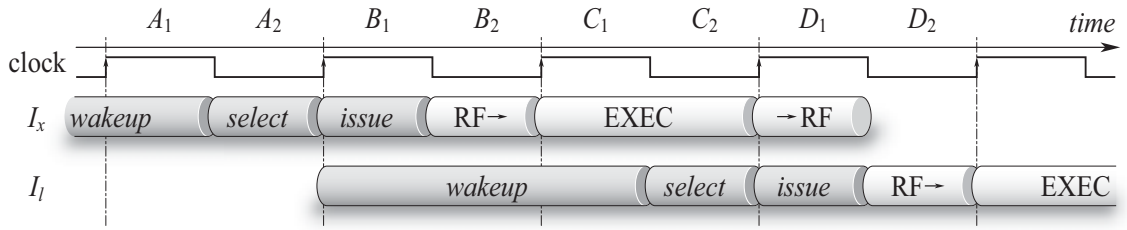


図 3.6: *Wakeup* フェーズに 1.5 サイクル充てた場合のバックエンドのパイプライン動作

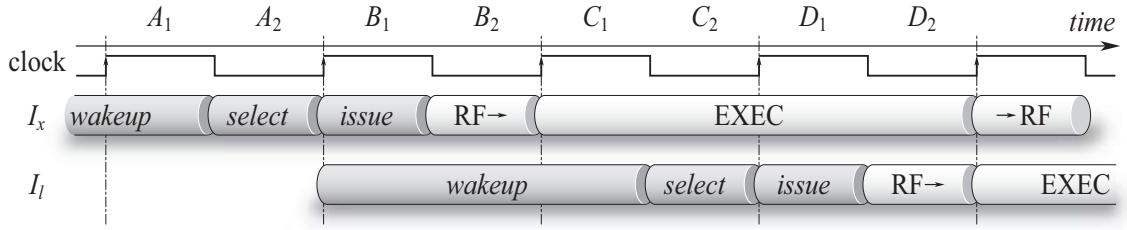


図 3.7: I_x — I_l 間の実効レイテンシが 2 サイクルの場合のバックエンドのパイプライン動作

I_x の実行 (EXEC) ステージが開始されてから，その実行結果を I_l が利用可能になるまでのレイテンシを，I_x — I_l 間の**実効レイテンシ (effective latecy)** と呼ぶことにする．実効レイテンシは，I_x の**実行レイテンシ** (execution latency)，すなわち，実行ステージのサイクル数，および，I_x — I_l 間のオペランド・バイパスの利用可 / 不可によって決まる．同図では，I_x が使用する実行ユニットの実行レイテンシは 1 サイクルで，I_x が生成したデータはオペランド・バイパスを通って I_l の実行に使用されているので，I_x — I_l 間の実効レイテンシは 1 サイクルとなっている．なお，この場合のように，依存関係にある 2 命令の実行ステージが時間的に連続している場合，この 2 命令は **back-to-back** に実行されているという．

以下では，I_x — I_l 間の実効レイテンシは決定的である，すなわち，静的に 1 サイクルに決まっているものとする．実際のスーパースカラ・プロセッサでは，通常，整数演算器の実行結果を実行する場合，実効レイテンシは 1 サイクルに決まっている．

Wakeup フェーズの開始タイミング

ここで注意しなければならないのは，I_x が *select* されるサイクル A_2 の直後のサイクル B_1 から，I_l に対する *wakeup* が開始できることである．実際に I_x の実行結果が得られるのはサイクル C の最後であるが，I_l に対する *wakeup* はそれまで待つ必要はない．これは，以下の 2 つの理由による：

1. バックエンド・パイプラインの動作が決定的である．具体的には，I_x が *select* されると，I_x は既知の時刻に実行されることが決まる．

2. 加えて，I_x — I_l 間の実効レイテンシも決定的である．

この 2 つの理由により，I_x が *select* された時点で，I_x の実行結果がサイクル D に利用可能になることが分かる．したがって，そのサイクル D から I_l の実行が開始できるように，I_l に対する *wakeup* フェーズを予め開始しておくことが可能になるのである．

I_l に対する *wakeup* を開始するサイクルは I_x — I_l 間の実効レイテンシだけから決まる．I_x — I_l 間の実効レイテンシが L_e サイクルである場合，I_l に対する *wakeup* は，I_x の *select* から $L_e - 1$ サイクル後に開始すればよい．特に，図 3.5 のように，I_x — I_l 間の実効レイテンシが 1 サイクルである場合には，I_l に対する *wakeup* フェーズは，I_x の *select* フェーズの直後から開始することになる．

逆に，I_x — I_l 間の実効レイテンシが非決定的 (nonderteministic) である場合には，単純にサイクル B_1 から I_l に対する *wakeup* を開始する訳にはいかない．そのような例としては，整数乗除算など，定義側の命令の実行レイテンシが可変長である場合や，オペランド・バイパス利用の可否が動的に決定される場合などが考えられる．ただし，このような命令の実装が性能に影響を与えることはほとんどない．

前述したように，I_x — I_l 間の実効レイテンシが 1 サイクルである場合には，I_l に対する *wakeup* は，I_x の *select* の直後のサイクル B_1 から開始することができる．逆に，I_x がまだ *select* されるかどうか決っていないサイクル A_2 では，I_l に対する *wakeup* を開始することはできない．すなわち，*wakeup* フェーズの処理は，*select* フェーズの結果に依存する*．つまり，***select* フェーズから *wakeup* フェーズには，フィードバック・ループが存在する**ことになる．次項で述べるように，このフィードバック・ループの存在は，これらのフェーズのパイプライン化可能性に大きく影響する．

3.2.4 5 フェーズのパイプライン化

命令スケジューリングの 5 つのフェーズのうち，フロントエンドにある *rename* と *dispatch* は，必要であれば，パイプライン化することができる．これらのフェーズのパイプライン化の代償は，分岐予測ミス・ペナルティの増加であり，分岐予測ヒット率の高さによって補償することができる．実際，現存するスーパースカラ・プロセッサでは，これらのステージに数サイクルを充てることが多い [62, 22, 63, 23, 57, 33, 48].

一方，*wakeup* と *select* フェーズのパイプライン化の代償は，*rename*，*dispatch* フェーズの場合とは大きく異なる．図 3.5 **(L-2)** に，*wakeup* フェーズに 1 サイクル余分にかけた場合の命令パイプラインの様子を示す．前述した *select* から *wakeup* フェーズへのフィードバック・ループのため，I_l に対する *wakeup* は B_1 からしか開始できないことに注意されたい．その結果，I_l の発行は 1 サイクル遅れ，I_x と I_l は back-to-back に実行できなくなる．同図では，I_x が生成したデータは，レジスタ・ファイルを介して I_l の実行に使用されればよく，オペランド・バイパスを使用する必要はなくなっている．すなわち，*wakeup* と *select* フェーズに 1 サイクルより多

* もちろん，何らかの予測によって，この依存を断ち切ることは可能である [49].

くを割り当てることは，IPC の観点からは，実行レイテンシが 1 サイクルである実行ユニット —— 通常の構成では ALU，シフタ —— からのオペランド・バイパスを取り除くことと等価である．6.5 項で述べるように，それによる IPC の悪化は最大 15%程度にもなり，クロック速度の向上に見合わない可能性が高い．したがって次のように結論づけることができる：すなわち，**実効レイテンシが 1 サイクルである命令間では，*wakeup* と *select* フェーズは合わせて 1 サイクル以内に実行しなければならない**．

なお *issue* フェーズも，*rename*，*dispatch* フェーズと同様，予測ミス・ペナルティの増加を代償にパイプライン化することができる．ただし，その様子は，*rename*，*dispatch* フェーズとは若干異なる．図 3.5 では，*issue* に半サイクルが充てられているが，より多くのサイクルを充てた場合でも，I_x と I_l が back-to-back に実行できなくなることはない．その場合，I_x，I_l の双方において，*issue* より下流のステージがより下流に移動することになる．しかし，それら移動は平行に起こるので，I_x と I_l が back-to-back に実行できることに変わりはない．実際，最近では，*issue* フェーズに数サイクルを充てるようなプロセッサも現れつつある [35]．また *issue* フェーズのパイプライン化は，分岐予測に加えて，実行レイテンシ予測のミス・ペナルティの増加も招くが，やはり予測ミス率の低減によってその影響を緩和することができる（1.1.4 項）．

3.2.5　3.2 節のまとめ

本節では，out-of-order スーパースカラ・プロセッサの命令スケジューリングの 5 つのフェーズのパイプライン動作と，パイプライン化可能性について述べた．本節の内容は，以下のようにまとめられる：

- 5 つのフェーズのうち，*rename*，*dispatch*，および，*issue* は，分岐予測ミス・ペナルティの増加を代償に，パイプライン化可能である．
- 一方，*wakeup* と *select* は，フィードバック・ループのため，実際上パイプライン化不能である．合わせて 1 サイクル以内に実行しなければ，IPC に深刻な悪影響がある．

3.3　命令ウィンドウ・エントリと物理レジスタの寿命

前述したように，out-of-order スーパースカラ・プロセッサの命令ウィンドウより下流，すなわち，命令ウィンドウとバックエンドは，データ駆動型計算機とほぼ同じものと考えてよい．Out-of-order スーパースカラ・プロセッサの命令ウィンドウは，データ駆動型計算機では**待ち合わせ記憶** (matching memory) に相当する．しかしスーパースカラ・プロセッサは，基本的には制御駆動型の計算原理に従うものであるから，全く同じという訳ではない．

データ駆動型計算機では，基本的に，すべてのデータは一時的 (temporal) である．すなわち，データはプロデューサからコンシューマに渡される間だけ存在すればよい．コンシューマが実行されてしまえば，そのデータは消してしまって構わない．一方，制御駆動型計算機

では，データは永続的 (persistent) である．すなわち，同一のロケーションに対する上書きによって明示的に消去されない限り，将来の参照に備えてデータはいつまでも取って置かなければならない．

この違いは，命令ウィンドウ・エントリの**寿命** (lifetime) と，命令の実行結果を格納する物理レジスタの寿命の差として現れる．

命令ウィンドウ・エントリと物理レジスタの寿命

Out-of-order スーパースカラ・プロセッサでは，命令ウィンドウ・エントリと物理レジスタは共にプールによって管理される．すなわち，命令がフェッチされると，その命令それ自体の格納場所として命令ウィンドウ・エントリが，その命令の実行結果の格納場所として物理レジスタが，それぞれ動的に割り当てられる．使用後はそれぞれプールに返却され，別の命令によって再利用される．

なお，スーパースカラ・プロセッサのバックエンドにおいて命令に割り当てられるメモリ資源は，命令ウィンドウ・エントリと物理レジスタですべてである．バックエンドで用いられるメモリ資源はすべて，命令ウィンドウ・エントリか物理レジスタのいずれかのフィールドである．

さて，1つの命令に割り当てられた命令ウィンドウ・エントリと物理レジスタは，しかし，それぞれ異なる寿命を持つ．命令ウィンドウ・エントリと物理レジスタは，フェッチされた命令に対してほぼ同時に割り当てられるが，解放のタイミングはそれぞれ異なる．命令ウィンドウの構成方式や，投機失敗時の回復の方式などにも依存するが [41]，原理的には，それぞれ以下のタイミングで解放できる：

命令ウィンドウ・エントリ　命令が発行されれば解放してよい．

物理レジスタ　当該物理レジスタを使用する命令が存在する間はもちろんのこと，そのような命令が将来現れる可能性がある場合にも，解放することはできない．

割り当てられた論理レジスタに対して上書きを行う命令の実行が**完了**すると，プログラム・オーダ上でそれより下流には，当該物理レジスタを参照する命令が現れないことが保証される．図 3.1 (p. 50) の例では，命令 I_x のデスティネーション・オペランドである $8 に対して，命令 I_r が上書きを行っている．そのため，命令 I_r の実行が完了すると，I_x のデスティネーション・オペランド $8 に割り当てられている物理レジスタ %1 に対する参照が以降行われないことが保証される．

MIPS R10000 プロセッサの物理レジスタ・プールは，この議論を素直に実装している．その実装方法は，[23] に詳しい．

待ち合わせ記憶のエントリの寿命

一方，データ駆動型計算機では，1つの命令には待ち合わせ記憶の 1 エントリが動的に割り当てられる．待ち合わせ記憶エントリもプールによって管理されている．

待ち合わせ記憶エントリは，命令それ自体を格納する命令フィールドと，データを格納するデータ・フィールドを持つ．データ駆動型計算機の待ち合わせ記憶のエントリの命令フィールド

は，スーパースカラ・プロセッサの命令ウィンドウ・エントリと，同データ・フィールドはスーパースカラ・プロセッサの物理レジスタと，それぞれ対比することができる．

しかし，待ち合わせ記憶エントリのデータ・フィールドに格納されるのは，そのエントリに格納された命令の実行結果ではなく，その命令の実行に必要となる左 / 右のソース・オペランド・データである．命令が実行されると，その実行結果は，その命令が指示するコンシューマが格納された待ち合わせ記憶のエントリに送られ，そのエントリのデータ・フィールドに格納される．そして，左 / 右のソース・オペランド・データが揃った命令から実行されることになる．実行された命令の待ち合わせ記憶エントリは，直ちに解放してよい．

命令フィールドとデータ・フィールドは，待ち合わせ記憶エントリの一部として，同時に割り当てられ，同時に解放されるから，それらの寿命は全く同じである．

プール

スーパースカラ・プロセッサにおける命令ウィンドウ・エントリと物理レジスタの寿命の違いは，それが別々のプールによって管理されるべきであることを意味する；もし仮に，データ駆動型計算機の待ち合わせ記憶のように，物理レジスタを命令ウィンドウ・エントリの一部として単一のプールによって管理すると，以下のような不都合が生じる．ある命令ウィンドウ・エントリの物理レジスタが割り当てられた論理レジスタを上書きする命令が延々と現れない場合，その命令ウィンドウ・エントリも延々と再利用することができない．すなわち，命令ウィンドウ・エントリにも物理レジスタの寿命が強制されることになる．

命令ウィンドウ・エントリは，本稿全体を通して問題としているように，物理レジスタに比べ極めて高価な資源である．後述するように，命令ウィンドウ・エントリ数は，*wakeup*，*select* ロジックの遅延に直接影響するからである．

以上の理由から，命令ウィンドウ・エントリと物理レジスタを単一のプールによって一括管理することは受け入れ難い．実際，現存するプロセッサでは，その管理コストにも関わらず，命令ウィンドウ・エントリと物理レジスタを個別に管理している．物理レジスタの数 NR は，$NR = 2 \cdot WS$ とすることが多い [23, 48]*.

次節からはいよいよ，out-of-order スケジューリングの各フェーズの動作を実現するロジックについて説明する．前節で述べた out-of-order スケジューリングの 5 つのフェーズのうち，*dispatch* と *issue* は命令ウィンドウを構成するペイロード RAM に対する単なる書き込みと読み出しであるので省略し，*rename*，*wakeup*，および，*select* ロジックについて詳しく述べる．まず 3.4 節では，*rename* ロジックについて説明する．次いで 3.5 節では，順序を変えて，先に *select* ロジックについて説明することにする．*Select* ロジックの方が理解が容易であり，また，*wakeup* ロジックの実装にあたっては，*select* ロジックとのインタフェースが重要な境界条件となるためである．そして，最後に 3.6 節で，本稿の主眼である *wakeup* ロジックについて詳しく述べる．

なお，各ロジックの規模は，表 3.2 に示すスーパースカラ・プロセッサの基本的なパラメタ

* Farkas らは，多重度 4 では 80，8 では 120 の物理レジスタが必要だとしている [64].

	記号	値	説 明
フェッチ幅 (fetch width)	FW	4	同時にフェッチ可能な命令の数
ディスパッチ幅 (dispatch width)	DW	4	同時にディスパッチ可能な命令の数
発行幅 (issue width)	IW	4	同時に発行可能な命令の個数
ウィンドウ・サイズ (window size)	WS	32	命令ウィンドウに格納できる命令の数
物理レジスタ数 (no. of physical regs)	NR	64	物理レジスタの個数
タグ幅 (tag width)	TW	6	タグのビット幅 $\lceil \log_2 NR \rceil$

表 3.2: ロジックのパラメタ

によって記述される．同表中の値は，ごく基本的な 4-way スーパースカラ・プロセッサのものであり，次節以降での説明の参考にされたい．より実際的な値に関しては，3.7 節以降で述べる．

3.4 *Rename* ロジック

本節では，レジスタ・リネーミングを行う *rename* ロジックについて詳しく述べる．以下ではまず 3.4.1 項で処理の逐次的な流れについて述べた後，3.4.2 項その実装の方法について説明する．

3.4.1 *Rename* ロジックの処理

レジスタ・リネーミングでは，論理レジスタ番号から物理レジスタ番号への現在の写像 (current mapping) を記録する**レジスタ・マップ・ テーブル** (**Register Map Table**:**RMT**) が中心的な役割を果たす.

リネーミングの処理は，以下のように，1. *prL*/*prR* の解決 (resolution) と，2. *prD* の割り当て (allocation) とに分けて考えることができるが，それぞれは，基本的には，RMT に対する参照と更新によって実現される:

1. ***prL*/*prR* の解決**　RMT を参照し，左 / 右のソース・オペランドである論理レジスタにそれぞれ割り当てられている物理レジスタを求める．それらの物理レジスタの番号が，それぞれ *prL*/*prR* である．

2. ***prD* の割り当て**　未使用の物理レジスタをプールから 1 つ取り出して，命令のデスティネーション・オペランドに割り当てる．その物理レジスタの番号が *prD* である．

   また，新たな割り当てにしたがって，RMT を更新する．

図 3.8 左に示したコードが同図右のようにリネーミングされるときの RMT の様子を，図 3.9 に示す．なお，図 3.8 のコードは，図 3.1 に示したものと同一である．以下に，各命令に対

label	*opcode*	*opD*		*opL*	*opR*
I_x :	op_x	\$8	=	\$7,	0
I_l :	op_l	\$9	=	\$8,	1
I_r :	op_r	\$8	=	\$7,	2
I_c :	op_c	\$8	=	\$9,	\$8

label	*opcode*	*prD*		*prL*	*prR*	*immed*
I_x :	op_x	%1	=	%0,		0
I_l :	op_l	%2	=	%1,		1
I_r :	op_r	%3	=	%0,		2
I_c :	op_c	%4	=	%2,	%3	

図 3.8: レジスタ・リネーミング前（左）と後（右）のコード（図 3.1 の再掲）

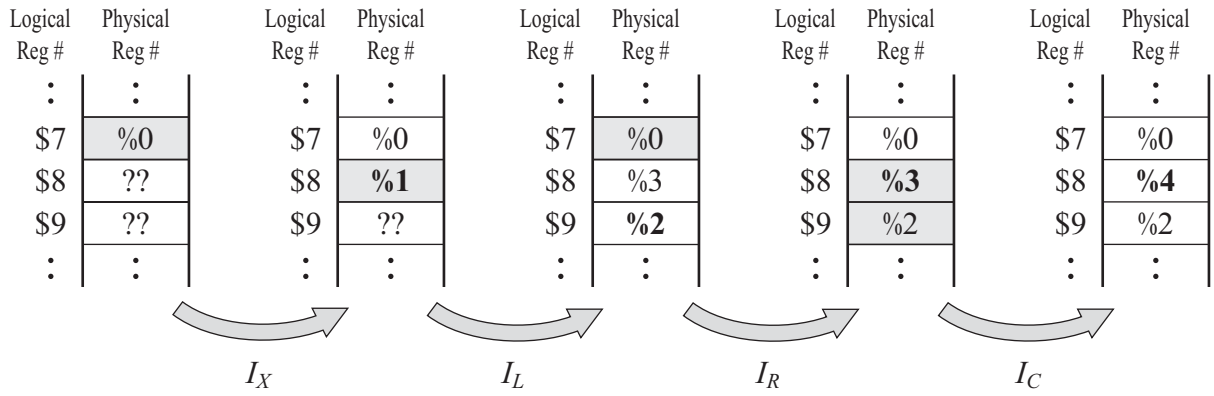


図 3.9: RAM 方式レジスタ・マップ・テーブル

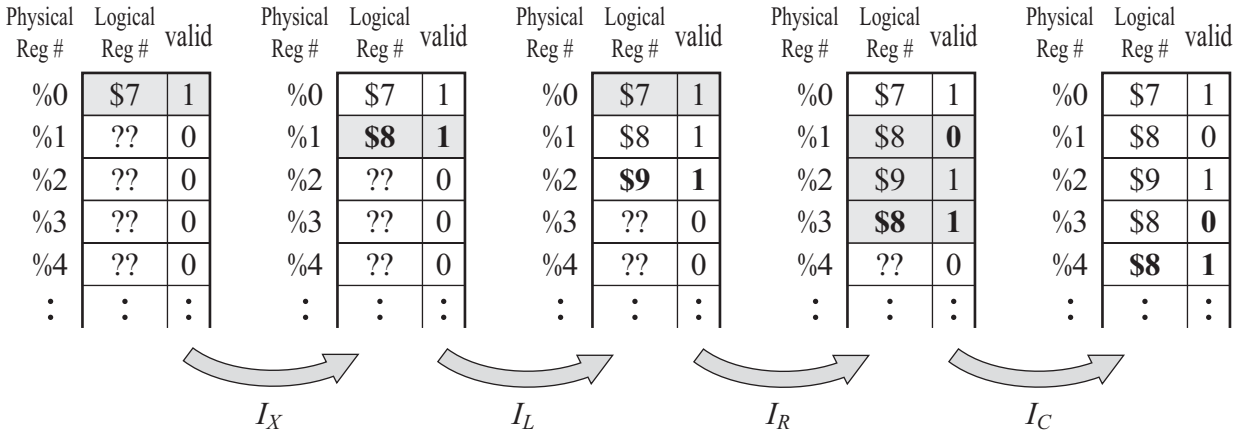


図 3.10: CAM 方式レジスタ・マップ・テーブル

する処理の流れを説明する．前述したように，各命令はプロセッサのフロントエンドにある *rename* ステージを in-order で通過することに注意されたい：

I_x RMT を参照すると，命令 I_x の左ソース・オペランド $7 には，物理レジスタ %0 が割り当てられていることが分かるので，命令 I_x の *prL* は %0 とする．

命令 I_x のデスティネーション・オペランド $8 には物理レジスタ %1 が割り当てられるので，RMT の $8 行を %1 に更新する．図 3.9 では，更新される部分を網掛けで示した．

I_l 次の命令 I_l の左ソース・オペランド $8 に対しては，つい今し方 RMT の $8 行に書き込まれた物理レジスタ %1 が読み出され，*prL* は %1 となる．

I_r 命令 I_r では，I_x と同様に，*prL* は %0 となる．

デスティネーション・オペランド $8 には新たに物理レジスタ %3 が割り当てられるので，RMT の $8 行は上書きされ，%1 から %3 に更新される．

I_c したがって，次の命令 I_c の左ソース・オペランド $8 に対しては，物理レジスタ %1 ではなく，%3 が読み出されることになる．

なお，命令 I_c のように，ソース・オペランドとデスティネーション・オペランドに同一の論理レジスタを指定することがあるため，RMT の更新は，RMT の参照後に行う必要がある．さもないと，自分自身が書き込む物理レジスタを参照することになってしまうからである．

3.4.2 *Rename* ロジックの実装

RMT の実装にはいくつかの方式があるが，本稿ではまず，その自然さから，RAM を用いる **RAM 方式 (RAM scheme)** について述べる [50].

RAM 方式では，前項で述べた RMT をそのまま，論理レジスタ番号をアドレス，物理レジスタ番号をを内容とする RAM として実装すればよい．この RAM の構成は，$2 \cdot FW$-read, FW-write，WS word $\times$ TW bit となる．基本的には，この RAM に対して，デスティネーション・オペランドの論理レジスタ番号をアドレスとして，割り当てられた *prD* を書き込んでおけば，左 / 右ソース・オペランドの論理レジスタ番号をアドレスとする読み出しによって *prL*/*prR* を得ることができる．

ただし，前項の説明は各命令を逐次的に処理する場合のものであるから，実際にスーパースカラ・プロセッサに実装する場合には同時に複数の命令を処理することを考慮する必要がある．そのためには，単に RMT を構成する RAM をマルチポート化するだけでなく，同時にリネーミングされる命令間の依存に対処する必要がある．

同時にリネーミングされる命令間に依存がある場合には，RMT からは『古い』*prL*/*prR* が読み出されてしまう．例えば，図 3.1 に示したコードにおいて，最初の 2 命令 I_x と I_l が同時にリネーミング処理を受ける場合を考えよう．命令 I_l の左ソース・オペランド $8 に割り当てられるべき *prL* は，命令 I_x のデスティネーション・オペランド $8 に割り当てられる *prD* %3 であ

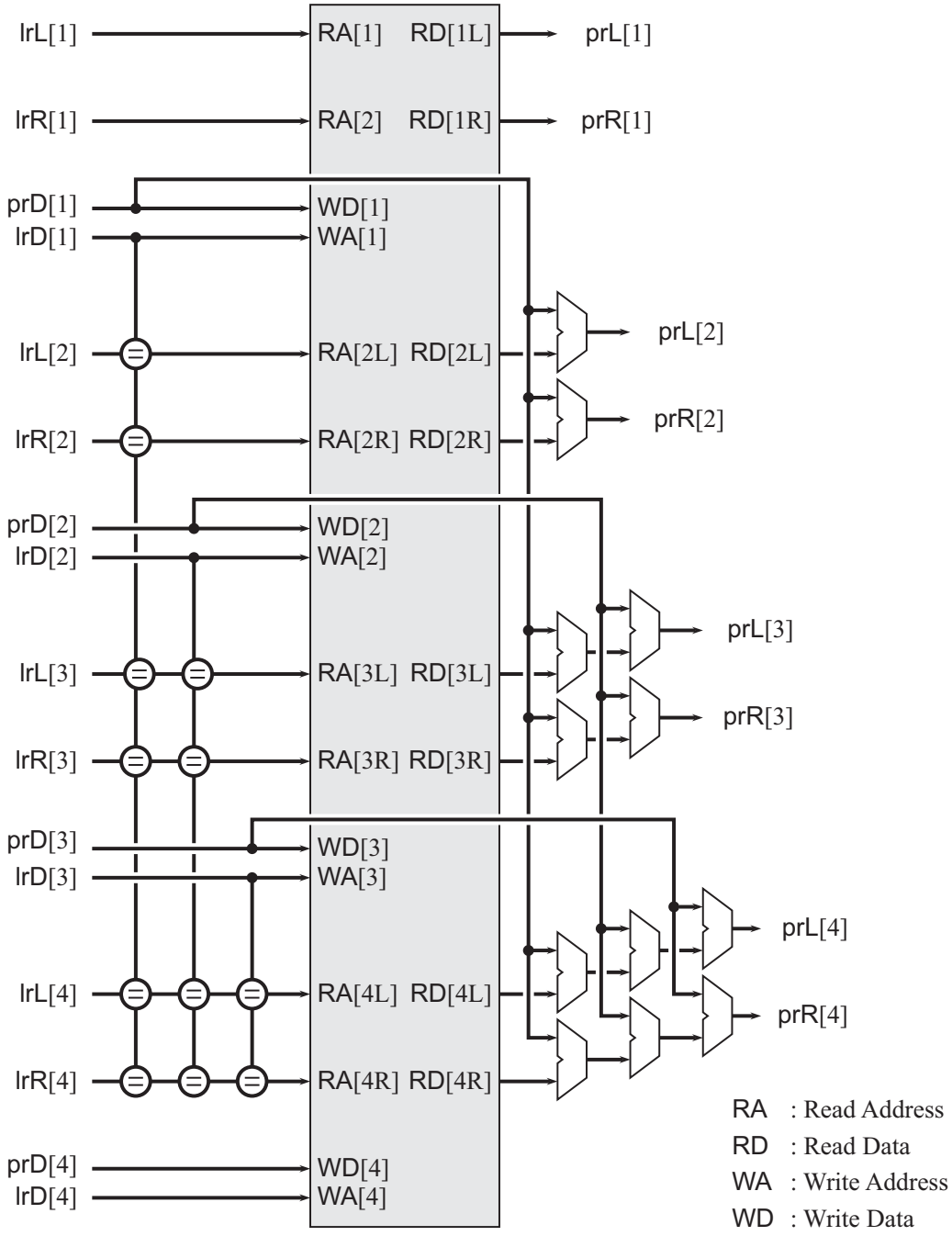


図 3.11: *Rename* ロジックのブロック図

る．しかし，I_x と I_l が同時に処理される場合には，図 3.9 で最左の状態で RMT を読み出すことになり，$8 行からは直前まで $8 に割り当てられていた物理レジスタ番号が読み出されてしまう．

図 3.11 に，それに対する対策を施した RMT のブロック図を示す．同図は 4 命令を同時にリネーミングする場合のものである．同図中央にある本体 RAM の左右には，それぞれ，同時にリネーミングされる命令間の依存関係を検出する一致比較器のアレイと，検出された依存関係にしたがって正しい *prL*/*prR* を選択するセレクタのアレイが置かれる．

一致比較器のアレイは，各命令の論理レジスタ番号の比較を行い，命令間の依存を検出する．依存が検出された場合には，RMT から読み出される『古い』 *prL*/*prR* の代わりに，RMT に書き込まれようとしている『新しい』 *prD* を選択すればよい．図 3.11 中，右側にあるセレクタは，左側にある，位置的に対応する一致比較器 ⊜ が一致を検出した場合，上側の入力を選択する．なお同図は，説明のため，セレクタを直列に接続しているが，実際にはトゥリー状に接続して，遅延を短縮することができる．

前述の，I_x と I_l が同時にリネーミング処理を受ける場合には，I_x のデスティネーション・オペランドと I_l の左ソース・オペランドの論理レジスタ番号が一致するので，RMT 本体 RAM から読み出される物理レジスタ番号ではなく，I_x のデスティネーション・オペランドに割り当てられ，今まさに本体 RAM に書き込まれようとしている *prD* %3 が I_l の *prL* として出力される．

3.4.3 *Rename* ロジックの動作タイミング

RMT の読み出し側では，以下の 3 つの処理の結果が，出力部にあるセレクタへの入力になる：

1. RAM Read　RMT 本体 RAM から読み出された *prL*/*prR*

2. Comparator　セレクタの選択信号となる論理レジスタ番号の比較結果

3. ***prD*** **Allocation**　新たに割り当てられた *prD*

これら 3 つの処理のうち，どれがクリティカルであるか考えよう．

1. RAM Read と，2. Comparator は，デコードされた命令の論理レジスタ番号に依存するため，命令デコード後，すなわち，*rename* ステージの最初からしか開始することができない．

1. RAM Read と 2. Comparator では，1. RAM Read の方がクリティカルであると考えられている [50]．1. RAM Read では，本体 RAM を構成するワードライン，ビットラインなどの配線遅延が支配的であるため，LSI の微細化にともなっていっそうクリティカルになっていくと考えられる．

一方，3. *prD* Alloocation は，*rename* ステージ以前から開始することができる．ある命令に割り当てるべき物理レジスタは，命令がどんなものであるかには依存しないからである．したがって，1. RAM Read と 2. Comparator とは異なり，*rename* ステージの開始以前から

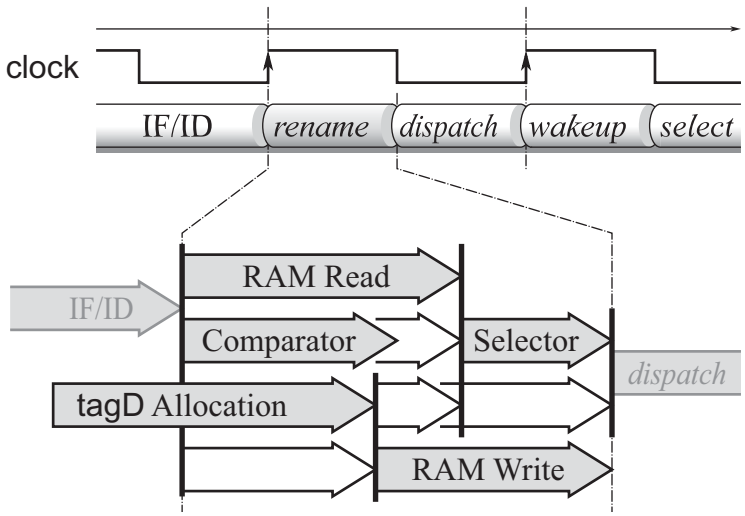


図 3.12: *Rename* ロジックのタイミング・チャート

開始することが可能である．ただしもちろん早めに決めれば決めるほど物理レジスタを無駄に消費することになるが，物理レジスタ数をある程度多くとっておくことによって，その不足によってフロントエンドがストール (stall) する機会を十分に小さくすることができる．

以上から，*rename* ロジックのタイミング・チャートは図 3.12 のようになる．同図中，灰色の矢印は各処理の遅延を，矢印の接続関係は処理間の依存を表している．また，白色の矢印は時間的な余裕を表す．同図から，RAM Read と Selector の遅延の和が，*rename* フェーズの遅延を決定することが分かる．

この遅延は，フロントエンドのレイテンシ，つまり，分岐予測ミスのペナルティを直接増加させるため，短いに越したことはない．しかし，3.2 節で述べたとおり，適切にパイプライン化を施すことによってクリティカル・パスからはずすことができるため，システムのクロック速度には直接影響しない．

Rename はこのように，基本的には，RMT の本体メモリを読み書きするだけで実現することができる．これは，*rename* フェーズのあるフロントエンドが in-order で命令を処理するためである．各命令が *rename* ステージを in-order で通過するため，プログラム・オーダ上で最新の状態を容易に維持することができるのである．

3.4.4　CAM 方式 *Rename* ロジック [63, 33]

前項までは，RMT の本体として RAM を用いる RAM 方式 RMT について述べてきたが，CAM を用いる **CAM 方式(CAM scheme)** も提案されている．CAM 方式は，RAM 方式に比べ，複雑でスケーラビリティ (scalability) が低いが，後で詳しく述べるようにチェックポインティング (checkpointing) 用のバックアップ・ メモリが少なくてすむため，より多くの分岐

を越えて投機を行うことができるというメリットがある．例えば，RAM 方式の多くのプロセッサがたかだか 4 程度の分岐しか越えられない [62, 22, 63, 23, 57, 33, 48] のに対し，HAL SPARC64 [63] は 16 もの分岐を越えることができる．しかし現在の分岐予測の精度では，その複雑さに見合うほどのメリットが得られるかどうか疑わしい．本稿では主に，次項から述べる *wakeup* ロジックとの対比のため，CAM 方式 RMT について説明する．*Wakeup* ロジックもやはり CAM によって構成されるからである．

CAM 方式 RMT は，本体に RAM ではなく CAM を用いる他は，RAM 方式と全く同じと考えてよい．

本体メモリとして用いられる CAM は，RAM 方式の本体 RAM とは逆に，論理レジスタ番号を内容とし，物理レジスタ数に等しいワード数を持つ．論理レジスタ番号をキーとして連想読み出しを行うことによって *prL*/*prR* を得る．図 3.8 に示したコードが，CAM 方式によってリネーミングされる様子を図 3.10 に示す．図 3.9 に示した RAM 方式の様子も同時に参照されたい．CAM の各エントリには，論理レジスタ番号の他，エントリの有効性を示す valid ビットがある．I_x に対するリネーミング処理では，I_x の左ソース・オペランド $7 をキーとして連想検索する．すると，%0 行の内容が $7 であり，その validビットが 1 であるので，*prL* は %0 であることが分かる．デスティネーション・オペランド $8 には %1 が割り当てられるので，%1 行に $8 を書き込み，同時に valid ビットを 1 とする．

チェックポインティング

この validビットは，論理レジスタに対する『上書き』に対応するために用意されている．命令 I_r のデスティネーション・オペランドは，命令 I_x と同じく $8 であるので，論理レジスタ $8 は，%1 から %3 へと写像し直す必要がある．この時，%1 行の validビットをリセットすることによって，$8 — %1 間の写像が無効化するのである．命令 I_c の右ソース・オペランドの $8 をキーとする連想検索では，%1 行と %3 行に $8 が見つかるが，%3 行の validビットだけが 1 であるため，%3 が最新の写像であることが分かる．

RMT では，分岐予測ミス時の回復 (recovery) のため，チェックポインティングを行う，すなわち，投機開始直前の状態を保存 (save) しておく．RAM 方式では本体 RAM 全体を保存しておく必要があるのに対して，CAM 方式では valid ビットのみを保存すればよい．再び図 3.10 を参照されたい．どの状態からでも，validビットだけを回復すれば，写像全体を回復できることが確認できよう．

これは，新しい写像の追加によって，古い写像の論理レジスタ番号欄が上書きされないことによる．例えば命令 I_r に対するリネーミング時には，論理レジスタ $8 に新たに物理レジスタ %3 が割り当て直されるが，%1 行の内容である $8 も破壊されずに残っている．RAM 方式では，$8 行の %1 が %3 に上書きされ，古い内容が破壊されてしまうため，RMT 全体を保存する必要が生じるのである．

CAM 方式の実装

図 3.13 に，本体 CAM のブロック図を示す．$2 \cdot FW$ 本のサーチ・ポートから入力される論理レジスタ番号が，各エントリに放送 (broadcast) される．各エントリでは，入力された論理レ

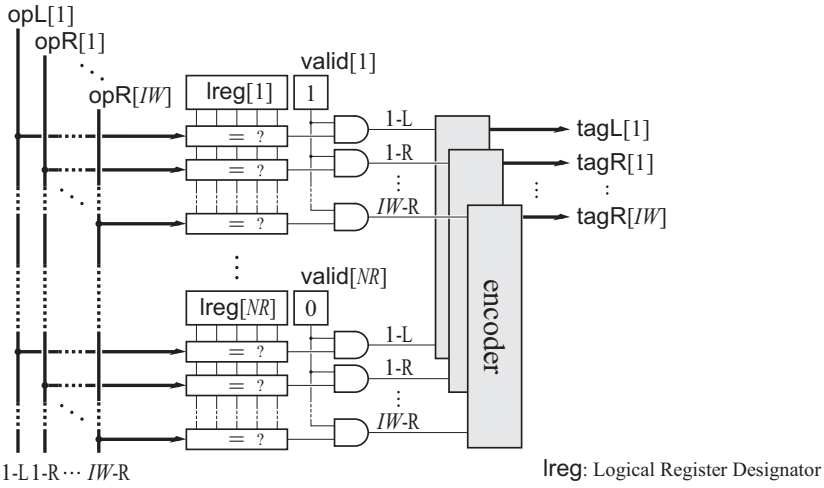


図 3.13: CAM 方式 RMT の本体 CAM のブロック図

ジスタ番号が各々の論理レジスタ番号（図中 lreg）と比較される．各エントリの比較結果が，validビットと AND され，エンコーダに送られる．1つのエンコーダに各エントリから送られて来る *NR* 本の入力のうち，1本だけがアサートされることになる．エンコーダは，これを TW b の 2 進値にエンコードする．

3.5 *Select* ロジック

Select ロジックは，命令ウィンドウ内で実行可能となった命令の中から，実際に発行する命令を決定する．*Select* ロジックは，発行可能な命令からの発行要求を調停する単なるアービタ (arbiter) である．*Select* ロジックは，適当な発行戦略に基づき，実行ユニットの空き状況を見て，発行要求を出している命令の中から実際に発行する命令を選択し，その命令に対して発行許諾を与える．

3.5.1 *Select* ロジックの動作

Select ロジックへの入力である発行要求信号は，通常，命令ウィンドウの各エントリごとのビット・ベクトルで与えられる．命令ウィンドウの i ($i = 0, 1, \cdots, WS-1$) 番エントリは，格納された命令が発行可能であれば，*select* ロジックに対して発行要求信号 ***req[i]*** をアサートする．

一方，*Select* ロジックからの出力である発行許諾信号は，単にその命令が選ばれたかどうかだけではなく，その命令が何番目に選ばれたのかを表す必要がある．例えば，ALU が 2 個ある場合には，選ばれた 2 つの命令のどちらがどちらの ALU を使用するかを区別するためである．2 つの ALU に 1 番，2 番と番号を振り，1 番目に選ばれた命令が 1 番 ALU を，2 番目に選ばれた命令が 2 番 ALU を使用するようにすればよい．このため発行許諾信号は，

通常，2 次元の配列で与えられる．命令ウィンドウの i 番 ($i = 0, 1, \cdots, WS-1$) エントリが j 番目 ($j = 0, 1, \cdots, IW-1$) に選ばれると，発行許諾信号 ***grant[i][j]*** がアサートされる．

Select ロジックは，適当な戦略に基づき，資源の空き状況を見て，*req* をアサートした命令ウィンドウ・エントリの中から最大 IW 個のエントリを選択する．発行の戦略は，*req* の各要素にどのような優先順位を付けるかということに帰着する．IPC のためには，できるだけ古い命令，あるいは，クリティカル・パス上の命令を選択することが望ましい [60, 14, 15]．しかし実際には，0.5 サイクル程度の間に選択を行わなければならないため，複雑なものを実装することは困難である．

最も一般的的なものは，物理番号 0 番のエントリが常に最高の優先順位を持つような固定優先順位の方式である．本節では，このような固定優先順位を持つアービタを紹介する．このような方式であっても，どのエントリに命令をディスパッチするかによって，実行順序をある程度制御することができる．例えば DEC* Alpha 21264 プロセッサでは，古い命令が高い優先順位を持つようにエントリのコンパクションを行っている [48]．7.2.4 項では，サイクリックな優先順位を付けられるアービタを紹介する．

3.5.2 カスケード方式 *Select* ロジックの実装

Select ロジックの実装方法としては様々なものが提案されている [65] が，本節では最も基本的なカスケード方式 [50] について述べる．カスケード方式は，性能的には満足いくものではないが，*select* ロジックの基礎について理解するには十分である．

カスケード方式は，1 命令を選択する回路を IW 個カスケード接続することによって IW 個の命令を選択する方式である．図 3.14 に，$(IW, WS = 3, 16)$ の場合のカスケード方式 *select* ロジックのブロック図を示す．

同図では，1 命令選択回路として，階層型アービタを用いている．1 つの階層型アービタは，図 3.14 中左上の矩形で表される．階層型アービタは，通常 4 つ程度の要求から 1 つを選択する回路をセルとして，そのセルを木状に接続することによって全体を構成する．各セルの論理回路を同図中右上部の矩形に示す．4 つ程度の要求を処理する回路をセルとするのは，セル内の論理ゲートの fan-in を 4 程度に制限するためである．

命令ウィンドウの各エントリに対して，1 本の発行要求信号線 *req* と，それに対する発行許諾信号線 *grant* が，それぞれ接続されている．要求信号 *req* は，命令が発行可能になった時にアサートされる．その要求が許諾される場合には，対応する許諾信号 *grant* がアサートされる．

各セルの処理は，以下のように，上りと下りの 2 つのフェーズからなる：

上り 下流からの要求があれば，上流のセルへ要求を出す．

下り 上流へ出した要求が許諾されたら，下流からの 4 つの要求のうちから 1 つを選び，許諾する．

---

* 発表当時．

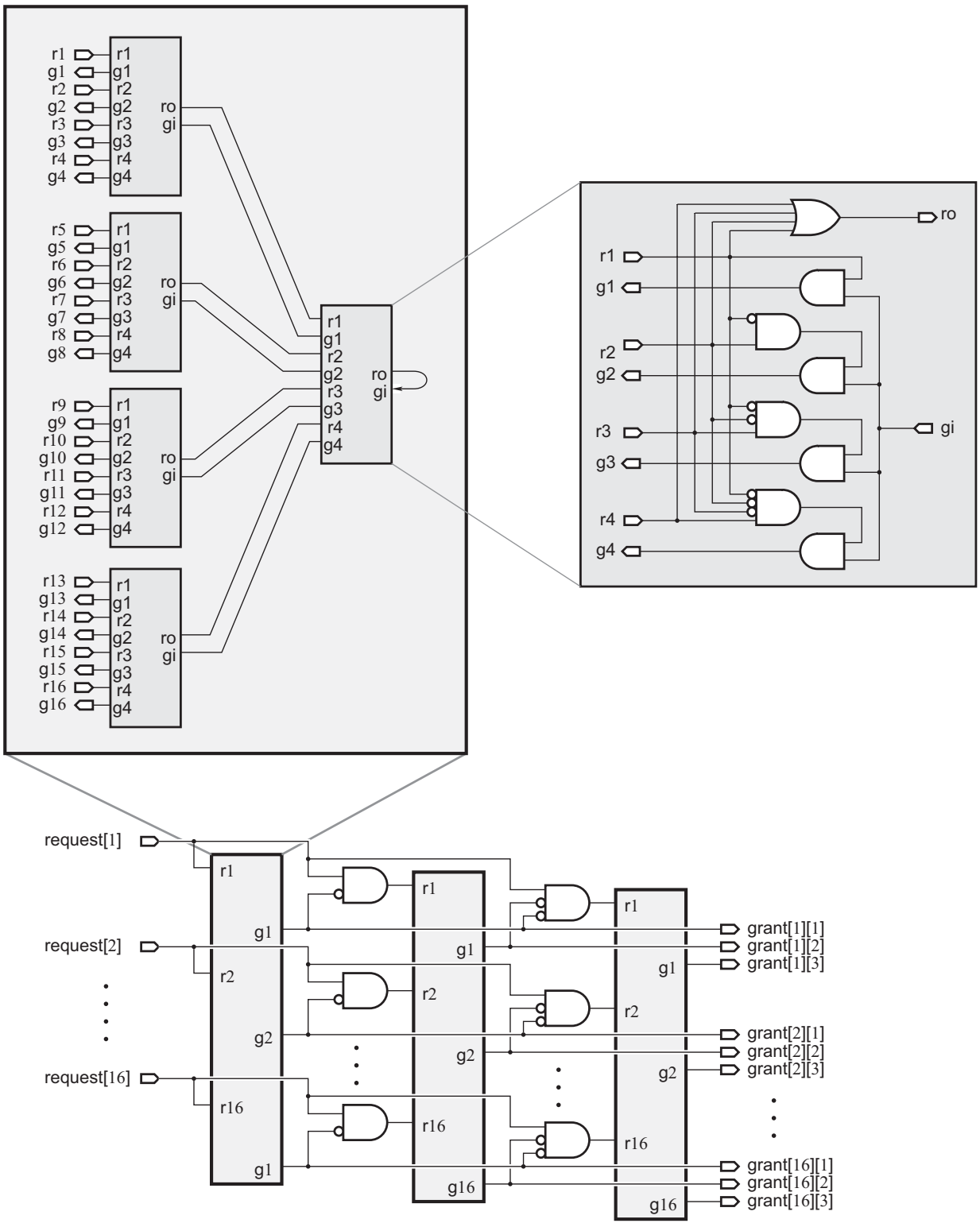


図 3.14: カスケード方式 *Select* ロジックのブロック図

最上流のセルでは，図 3.14 に示されるように，「要求があれば許諾される」とすればよい．

Select ロジック全体は，この階層型アービタを *IW* 個カスケード接続することによって得られる．図 3.14 下部に，接続の方法を示す．*n* 段目の選択回路への要求を，*n* – 1 段目までの発行許諾信号 *grant*[1]〜*grant*[*n*−1] で抑制することによって，*n* 段目の選択回路に *n* 番目の命令を選択させることができる．

このようにカスケード方式の遅延は，$O(\log_4 WS \times IW)$ で与えられる．

次節では *wakeup* ロジックについて述べる．3.2 節で見たように，*wakeup* ロジックは，本節で述べた *select* ロジックと，緊密なフィードバック・ ループを形成する．両者のインタフェースは，シグナル *req*/*grant* による．したがって，これらのシグナルの形式が *wakeup* ロジックのデザインの境界条件となる．

3.6 *Wakeup* ロジック

Wakeup ロジックは，*select* ロジックによる命令の選択にともなって実行可能になる命令を検出するロジックである．前節で述べたように，*wakeup* ロジックと *select* ロジックは，緊密なフィードバック・ループを形成している．*wakeup* ロジックの入力 / 出力は，*select* ロジックの出力 / 入力である，*grant*/*req* である．*Wakeup* ロジックは，*grant* を受け取って，次のサイクルの *req* を求めるロジックとすることができる．

Wakeup の処理は，物理レジスタ・ファイルの *rdy* フィールド，命令ウィンドウの *rdyL*/*rdyR* フィールドの更新と参照によって行われる．以下では，*rdy* を格納する RAM を用いる RAM 方式と，*rdyL*/*rdyR* を格納する CAM を用いる連想方式について述べる．RAM 方式は実装が困難な机上の方式であり，実際には専ら連想方式が用いられている．しかし，RAM 方式を知っておくことは，連想方式をはじめ，次章以降で述べるその他の *wakeup* の方式をより深く理解するための助けとなる．以下，3.6.1 項で RAM 方式について述べた後，3.6.2 項以降で連想方式について述べる．

3.6.1 RAM 方式 *Wakeup* ロジック

3.1.3 項で述べたように，*wakeup* の処理は，物理レジスタ・ファイルのフィールド *rdy* と，命令ウィンドウのフィールド *rdyL*/*rdyR* によって実現される．*rdyL*/*rdyR* は *rdy* と，式 3.1，$rdyL[i] = rdy[\,prL[i]\,]$，および，$rdyR[i] = rdy[\,prR[i]\,]$ $(i = 0, 1, \cdots, WS-1)$ なる関係がある．また，*wakeup* ロジックの出力である *req* は，*rdyL*/*rdyR* を用いて，$req[i] = rdyL[i] \cdot rdyR[i]$ と求められる．

これらの関係式を素朴 (naive) に実装すると，図 3.15 のようなロジックが得られる．

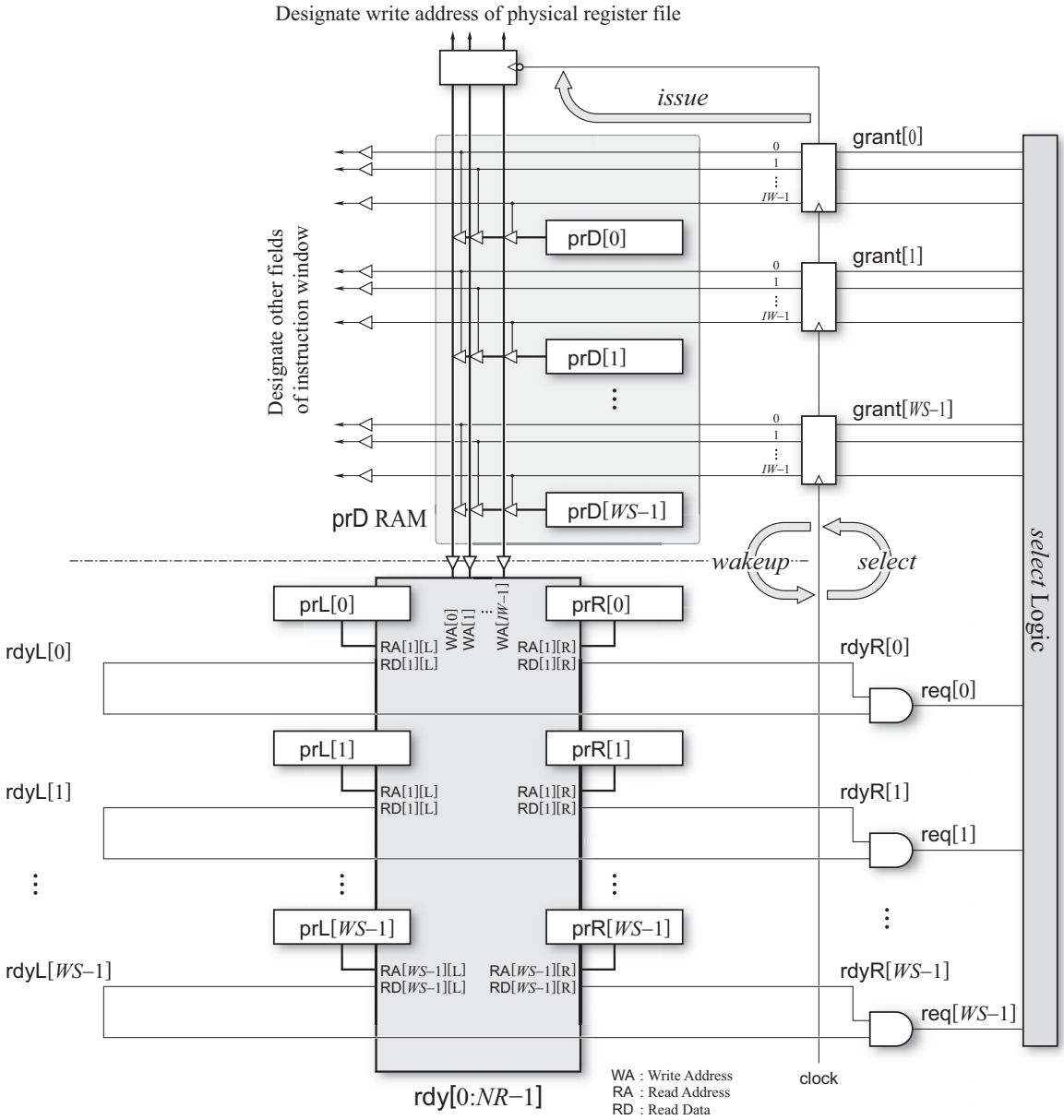


図 3.15: RAM 方式 *Wakeup* ロジックのブロック図

***rdy* RAM**

物理レジスタ・ファイルの *rdy* フィールドは，図 3.3 (p. 50) のように物理レジスタ・ファイルに物理的に付随している必要はない．図 3.15 では，物理レジスタ番号をインデクス，対応する物理レジスタの *rdy* フィールドの値を内容とする，NR word × 1 bit の独立した RAM として実装されている．この RAM を ***rdy* RAM** と呼ぶことにする．図 3.15 下中央の矩形がこの *rdy* RAM を示している．

以下に示すように，このロジックは，基本的には，*rdy* RAM に対する更新と参照によって *grant* から *req* を求める：

更新 *grant* によって実行が許諾された命令が指示されると，まず，図 3.3 上部から，その命令の *prD* が読み出される．

次いで，読み出された *prD* をアドレスとして，*rdy* RAM に 1 が書き込まれる．

参照 命令ウィンドウ中の各命令の *prL*/*prR* フィールドをアドレスとして *rdy* RAM を読み出すと，*rdyL*/*rdyR* が求められる．

req は，*rdyL*/*rdyR* を AND することによって得られる．

このように，*rdy* RAM が中心的な役割を果たすため，この方式を **RAM 方式**と呼ぶことにする．

なお，この RAM 方式における *rdyL*/*rdyR* は，組み合わせ回路的出力であることに注意されたい．したがって，必ずしも *rdyL*/*rdyR* を命令ウィンドウの一部とみなす必要はない．そのため RAM 方式は，後述する連想方式に比べ，冗長性が低く，シンプルである．

しかし RAM 方式は，実際には，実装することが極めて困難である．*rdy* RAM のリード・ポートに着目しよう．このリード・ポートは，$2 \cdot WS$ 本必要であり，例えば WS が 8 程度のごく小規模の out-of-order スーパースカラ・プロセッサであっても，$2 \cdot WS = 16$ 本にもなる．RAM の面積はポート数の 2 乗に比例するため（2.3.2 節），このような多ポートの RAM は実際上実装することができない．

3.6.2 連想方式 *Wakeup* ロジック

rdy RAM のインデクスは物理レジスタ番号である．一方，*select* ロジックに送らなければならない *req* のインデクスは，命令ウィンドウのエントリ番号である．そのため，*rdy* から *req* の間には，なんらかの変換が必要になる．

連想方式 *Wakeup* ロジックの原理

前項で述べた RAM 方式では，まず *prD* によって *rdy* RAM を更新し，次いで *prL*/*prR* によって *rdy* RAM を読み出すことによってこの変換を実現していた．例えば，命令 I_x の *prD*[x] = %1 によって *rdy* RAM の %1 行を *rdy*[%1] = 1 に更新し，次いで，I_l の *prL*[l] = %1 によって *rdy* RAM の %1 行を参照して，*rdyL*[l] = *rdy*[*prL*[l]] = 1 を得ている．

ここで，*prD*[*x*] と *prL*[*l*] の内容がともに %1 であることに気が付くだろう．そこで，*rdy* RAM を経由するのではなく，*prD* と *prL*/*prR* の内容を比較することによって *rdyL*/*rdyR* フィールドを更新することが考えられる．この例で言えば，*prD*[*x*] = *prL*[*l*] = %1 であることから，*rdyL*[*l*] をセットすることができる．

ただし，そのためには，定義側の命令の *prD* に一致する使用側の命令の *prL*/*prR* をすべて検出する，すなわち，*prD* に一致する *prL*/*prR* を連想検索する必要がある．その結果，*rdyL*/*rdyR* を更新するロジックは，一種の CAM によって実現されることになる．

連想方式 *Wakeup* ロジックの動作

図 3.8 (p. 68) に示したコードが実行される時の連想方式 *wakeup* ロジックの動作を，図 3.16 に示す．図 3.16 では，命令 I_x と I_r が最初から実行可能になっている．ここで，I_x が *select* ロジックによって選択されると，*prD*[*x*] = %1 が読み出される．*prL*/*prR* フィールドから %1 を連想検索すると，命令 I_l の *prL*[*l*] = %1 が見つかるので，*rdyL*[*l*] がセットされる．命令 I_l は即値を持ち，*rdyR*[*l*] は最初から 1 になっているため，I_l は実行可能になる．

3.6.3　連想方式 *Wakeup* ロジックの構成

図 3.17 に，連想方式 *wakeup* ロジックのブロック図を示す．

Wakeup ロジックへの入力は，前節で述べた *select* ロジックからの発行許諾シグナル *grant* である．*Wakeup* ロジックは，*grant* を受け取って，実行可能になる命令を検出する．一方，その出力は実行可能な命令を表すシグナル，すなわち，*select* ロジックへの発行要求シグナル *req* となる．

図 3.15 中央の太矢印によって端的に表されているように，*select* と *wakeup* ロジックは，シグナル *req* と *grant* を介して緊密 (tight) なループを形成している．3.2 節で述べたフィードバック・ループはロジックのレベルでは，このように現れる．

連想方式の *wakeup* ロジックは，前項で述べた方式と同様，*prD* を格納する上部の RAM 様のロジックと，下部の CAM 様のロジックの 2 つの部分からなる．前者を**デスティネーション RAM**，後者を**ソース CAM** と呼ぶ．

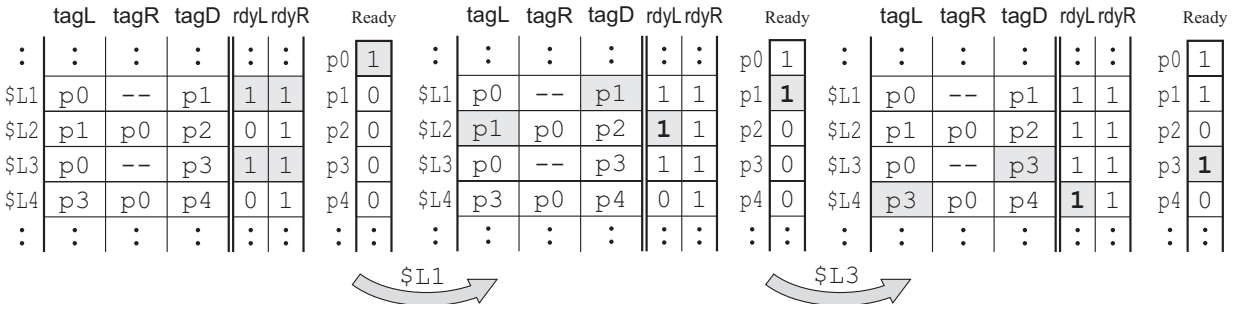


図 3.16: 連想方式 *Wakeup* ロジック

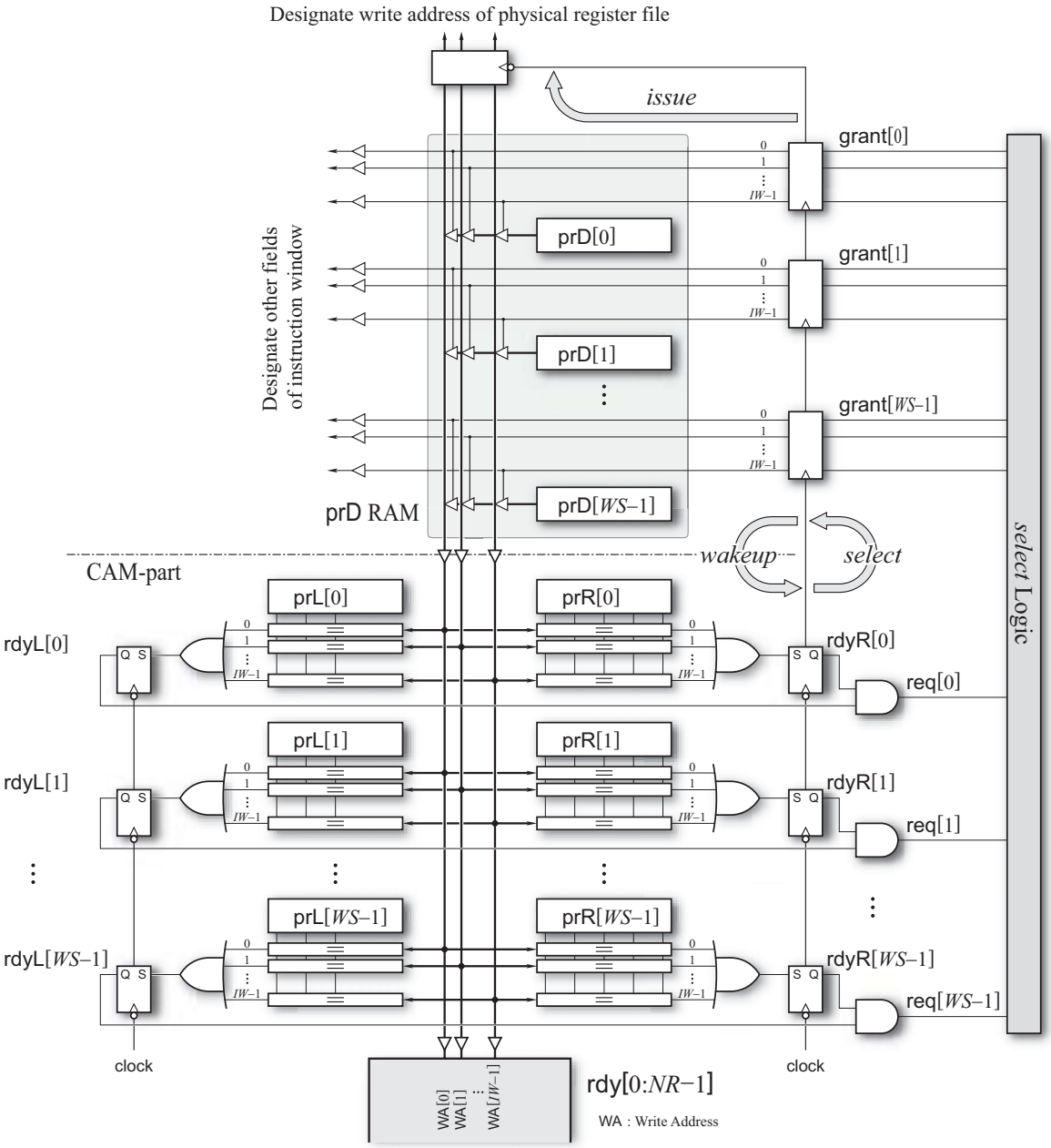


図 3.17: 連想方式 *Wakeup* ロジックのブロック図

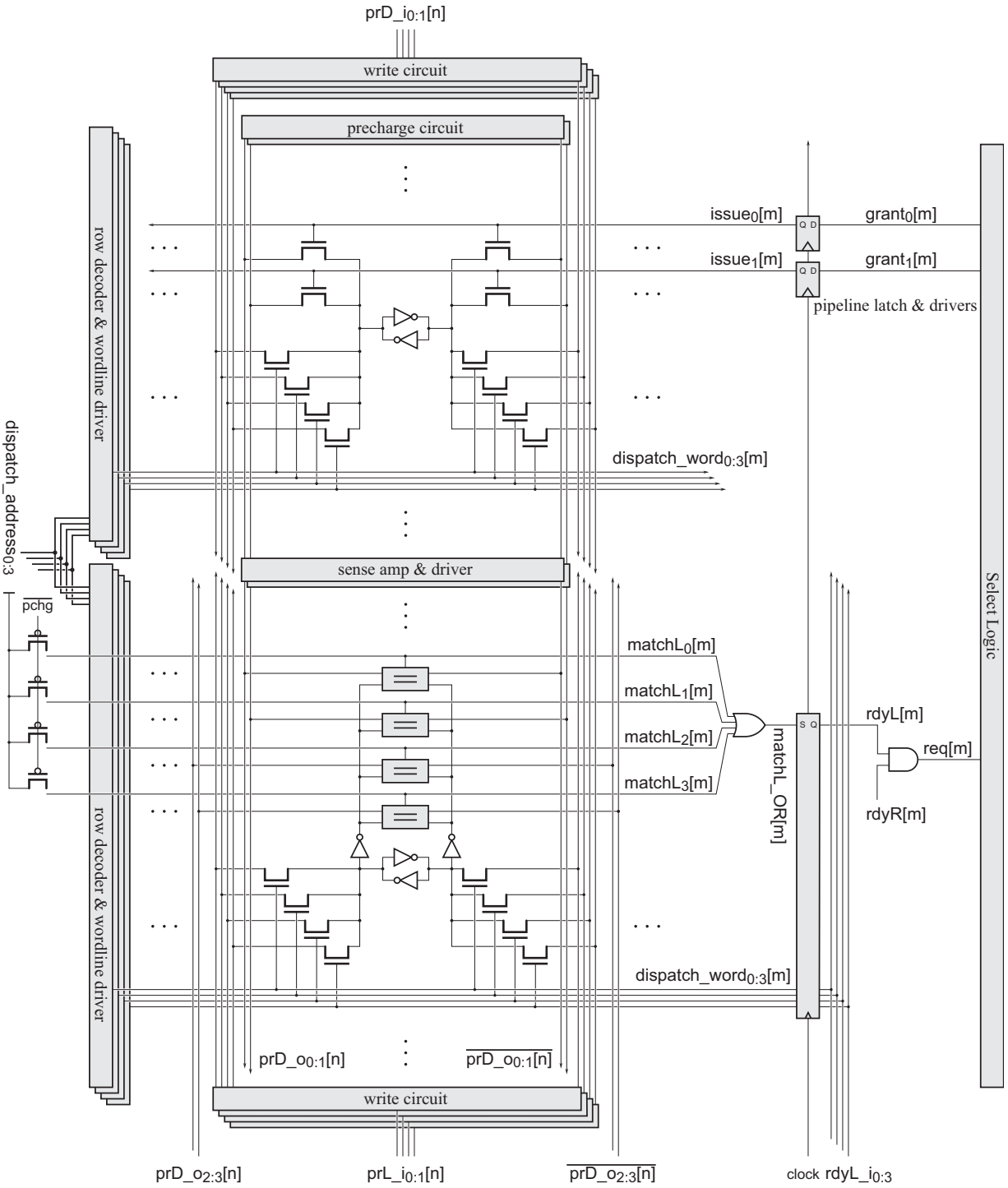


図 3.18: 連想方式 *Wakeup* ロジック

デスティネーション RAM

図 3.17 上部の RAM 様 のロジックは，命令ウィンドウのデスティネーション・オペランドに割り当てられた物理レジスタの番号，*prD* フィールドを格納する．同図 3.17 では，個別のレジスタの配列のように描かれているが，このロジックは多ポートの RAM として実装可能である．

ただしこのデスティネーション RAM のリード・ポートでは，通常の RAM とは異なり，行デコーダ (row docoder) が必要ない．デスティネーション RAM のリード・アドレスとなる *select* ロジックからの発行許諾シグナル *grant* が，前節で述べたように，元々デコードされた形で与えられるためである．リード・ポートのワードラインには，*select* ロジックからの *grant* がパイプライン・ラッチを介してワードラインに接続される．

また，前節では，*select* ロジックの *grant* は，単にどのエントリが選ばれたがだけではなく，そのエントリが何番目に選ばれたかを表していると述べた．このことは，デスティネーション RAM の *IW* 本のリード・ポートを使い分けるために便利である．$grant[i][j]$ $(i = 0, 1, \cdots, WS-1, j = 0, 1, \cdots, IW-1)$ は，*i* 行の *j* 番目のワードラインに接続されている．ある命令が *j* 番目に選ばれた場合には，*j* 番目のリード・ポートによって *prD* の読み出しが行われる．

同図 3.17 では *wakeup* 処理に用いられるリード・ポートのみが描かれているが，デスティネーション RAM は命令ウィンドウの一部でもあるから，ディスパッチのための *DW* 本のライト・ポートも必要である．

ソース CAM

図 3.17 下部の本体部分は，*prL*/*prR* をキー，*rdyL*/*rdyR* をバリューとする CAM 様のロジックとなる．*prL*/*prR*，*rdyL*/*rdyR* はともに，命令ウィンドウのフィールドでもある．このソース CAM では，デスティネーション RAM から読み出された *prD* がサーチ・ポートに入力される．入力された *prD* と一致する *prL*/*prR* が連想検索され，一致したエントリの *rdyL*/*rdyR* がセットされる．

サーチ・ポートは，やはり *IW* 本あり，デスティネーション RAM の *IW* 本のリード・ポートのデータ出力と 1 対 1 に接続されている．したがって，デスティネーション RAM のリード・ポートと同様に，ある命令が *j* 番目に選ばれた場合には，*j* 番目のサーチ・ポートによって連想検索が行われる．どの命令が何番目に選ばれるかは非決定的であるので，*IW* 本のサーチ・ポートはそれぞれ対称で，機能的な差異はない．いずれかのサーチ・ポートへ入力された *prD* が *prL*/*prR* と一致すれば，対応する *rdyL*/*rdyR* をセットすればよい．そのため，一致比較器は *prL*/*prR* のそれぞれに対して *IW* 個必要であり，それらの出力は OR されて，*rdyL*/*rdyR* を格納するレジスタに送られる．

rdyL/*rdyR* は，一致比較器のいずれかが一致を検出するとセットされる．その出力側では，各エントリの *rdyL* と *rdyR* の AND が *select* ロジックへの *req* となっている．

なお，簡単のため同図 3.17 には描かれていないが，厳密には，発行が許諾されたエントリの *req* をディアサートするためのロジックが必要である．具体的には，発行が許諾されたエントリ *i* では，*issued*[*i*] がセットされ，*req*[*i*] はディアサートされる．すなわち，$req[i] = rdyL[i] \cdot rdyR[i] \cdot \overline{issued[i]}$ である．

また，デスティネーション RAM の場合と同様，同図 3.17 には描かれていないが，*prL*/*prR*, *rdyL*/*rdyR* は命令ウィンドウの一部でもあるから，ディスパッチのための *DW* 本のライト・ポートも必要である．

rdyL/_rdyR_ と _rdy_ RAM の関係

rdyL/*rdyR* は，*prD* と *prL*/*prR* の一致が検出されるとセットされ，それ以降，少なくとも，発行要求が許諾されるまでの間，その出力を維持する必要がある．したがって連想方式の *rdyL*/*rdyR* は，RAM 方式の場合とは異なり，組合わせ回路的出力ではなく，図 3.17 に示されるように，任意の期間その内容を保持できるようなメモリの出力でなければならない．

rdyL/*rdyR* は，情報としては冗長であるが，*rdy* RAM を置き換えることはできない．*rdy* RAM は，当然のことながら，すべての物理レジスタに関する情報を持つ．一方 *rdyL*/*rdyR* は，命令ウィンドウに現在格納されている命令が使用する物理レジスタに関する情報しか持っていない．例えば，図 3.3 (p. 50) の状態では，*rdyL*/*rdyR* は全体として物理レジスタ %0〜%3 に関する情報しか持っていない．そのため，連想方式でも，RAM 方式と同じ *rdy* RAM が必要である．図 3.17 では，*rdy* RAM は図中最下部に配置してある．

情報の完全性から考えると，*rdy* RAM の方がむしろ本体であり，ソース CAM の *rdyL*/*rdyR* は *rdy* RAM の一部のビットの複製 (duplicate) だと考えた方が都合がよい．

rdy RAM，および，*rdyL*/*rdyR* レジスタに対する更新は，同時に行われる．命令が *select* され，その *prD* が得られると，ソース CAM では *prD* の連想アクセスにより *rdyL*/*rdyR* がセットされる．それと同時に，*rdy* RAM では，RAM 方式の場合と同様に，この *prD* をアドレスとして 1 を書き込むことで，対応するビットがセットされる．

このようにして，*rdy* RAM は，すべてのレジスタに対する最新の値を常に保持する．入力された *prD* は必ずしもソース CAM にヒットするとは限らないが，*rdy* RAM の *prD* で示されるビットは必ず 0 になっている．また，命令ウィンドウ・エントリが解放されるときには，*rdyL*/*rdyR* の内容は捨ててしまって構わない．

rdy RAM は，すべてのレジスタに対する最新の値を常に保持するので，命令が *dispatch* されるとき，*rdy* RAM は *prL*/*prR* の初期値を提供する．式 3.1，$rdyL[i] = rdy[\,prL[i]\,]$, $rdyR[i] = rdy[\,prR[i]\,]$ $(i = 0, 1, \cdots, WS-1)$ にしたがって，*dispatch* の直前に，*prL*/*prR* をアドレスとして *rdy* RAM を読み出すことで，*rdyL*/*rdyR* の初期値が得られる．

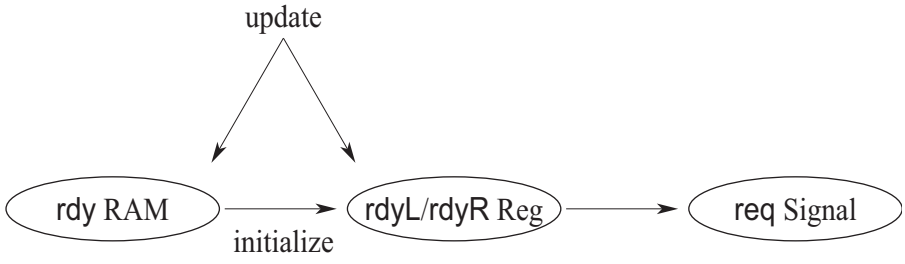


図 3.19: 連想方式 *Wakeup* ロジックにおける *rdy* と *rdyL*/*rdyR* の関係

図 3.19 に，連想方式 *wakeup* ロジックにおける *rdy* RAM と *rdyL*/*rdyR* レジスタの関係を示す．連想方式における *rdy* RAM は，*req* を生成するためには用いられず，すべての物理レジスタの利用可能性を常に保持ことによって，専ら *rdyL*/*rdyR* の初期値を提供する役割を果たしている．

初期値を提供するためのリード・ポートは，たかだか $2 \cdot DW$ 本，すなわち，8〜16 本程度で済む．また，更新のためのライト・ポートも，たかだか IW 本，すなわち，4〜8 本程度で済む．これらの値は，十分に小さいとは言えないものの，RAM 方式のリード・ポートが $2 \cdot WS$，すなわち，64〜128 本であったのと比べれば，十分に実現可能になっている．

Issue フェーズとの関係

3.2.3 項において，図 3.5 (p. 61) に示したように，定義側の命令 I_x が *select* されると，使用側の命令 I_l に対する *wakeup* と同時に，I_x に対する *issue* が開始される．I_x に対する *issue* では，命令ウィンドウの各フィールドが読み出されるわけだが，その中には，*prD*，*prL*/*prR* も含まれる．

特に *prD* は，*wakeup* フェーズでも読み出されるため，*prD* の読み出しは，*wakeup* と *issue* の両方に含まれることになる．デスティネーション RAM から読み出された *prD* は，*wakeup* のために図 3.15 下に示したソース CAM に送られると同時に，書き込むべき物理レジスタを指示するために同図 3.15 上に示したパイプライン・ラッチにも送られる．このパイプライン・ラッチに送られた *prD* は，ライトバック・ステージ（図 3.4 では →RF で示した）まで遅延され，実行結果が物理レジスタ・ファイルにライトバックされる時に，その書き込みアドレスとして使用される．

なお，本稿の論旨とは直接関係がないが，命令ウィンドウの *prD* 以外のフィールドを格納するロジックも，デスティネーション RAM と同様の，RAM セル・アレイによって構成されるこ

rdyL/*rdyR* は *rdy* のキャッシュ

ソース CAM の *rdyL*/*rdyR* は，*rdy* RAM に対する，変則的なキャッシングととらえると，分かりやすいかも知れない．以下のように，*rdy* RAM の方が本体であり，ソース CAM の *rdyL*/*rdyR* は *rdy* RAM の一部のビットのキャッシュ・コピー (cached copy) だと考えるのである：

- *rdyL*/*rdyR* の値は，命令の *dispatch* 時に *rdy* RAM からフェッチされる．
- 以降では，*req* シグナルを生成するための参照は，キャッシュ・コピーである *rdyL*/*rdyR* に対して行われる．
- *Wakeup* のための更新は，*rdyL*/*rdyR* と同時に *rdy* RAM に対しても行われるから，ライト・スルー (write through) によって一貫性の維持が行われていると考えられる．そのため，命令ウィンドウのエントリが解放されるときには，*rdyL*/*rdyR* の値は捨ててしまって構わない．

とを指摘しておく．ただし，デスティネーション RAM の読み出しは *wakeup* フェーズにも含まれるためクリティカルになり得るのに対して，その他のフィールドの読み出しは純粋に *issue* フェーズのみに含まれるため，パイプライン化可能であり，クリティカルにはならない（3.2 節）．

また，*prL*/*prR* は，ソース CAM のキー部でもあるため，*issue* ためのリード・ポートによってソース CAM の遅延の増加を招く．したがって，*wakeup* ロジックの遅延がクリティカルである場合には，*prL*/*prR* を *issue* 用と *wakeup* 用に複製 (duplicate) する方法が考えられる．すなわち，図 3.17 に示した *prL*/*prR* とは別に，デスティネーション RAM と同様の行デコーダのない RAM を用意し，*dispatch* フェーズではその RAM にも *prL*/*prR* を書き込んでおき，*issue* フェーズではその RAM から *prL*/*prR* を得るのである．同図 3.17 の *prL*/*prR* 部にはリード・ポートは付加せずに済むため，*wakeup* ロジックの面積を抑え，遅延を短縮することができる．7 章のロジックの評価では，実際にこの最適化を施している．

3.6.4　連想方式 *Wakeup* ロジックの遅延

Wakeup ロジックと *select* ロジックは，*req* と *grant* を介して，緊密 (tight) なフィードバック・ループを形成している．3.2 節で述べたように，このループ 1 周の遅延が，1 サイクル未満でなければならない．クリティカルである場合，この遅延がシステムのクロック速度を規定することになる．

3.7　命令ウィンドウの非集中化

ウィンドウは，実際には，前節まで説明してきたような集中した単一のロジックとして実装されるのではなく，q1.1.3 項で触れたように，複数の**サブウィンドウ**に**非集中化** (decentralization) されることが多い．実際，最近のスーパースカラ・プロセッサの多くは，整数 (**INT**)，ロード / ストア (**LS**)，浮動小数点 (**FP**) といった命令の系統ごとに，別個の**サブウィンドウ**を持つ．例えば，DEC* Alpha 21264 プロセッサは (INT+LS) と FP の，HP PA-8000 プロセッサは (INT+FP) と LS の，それぞれ 2 つのサブウィンドウを持つ．また，MIPS R10000 プロセッサは INT，LS，FP のそれぞれに対して，合計 3 つのサブウィンドウを持つ [62, 22, 63, 23, 57, 33, 48].

このような命令の系統ごとのウィンドウの非集中化は，ごくわずかな IPC のペナルティを犠牲に，ロジックの実効サイズを大幅に縮小できるため，非常に有効である．非集中化した場合，ウィンドウの断片化（フラグメンテーション，fragmentation）のために，IPC が低下する可能性がある．すなわち，命令ウィンドウ・エントリが総量としては不足していないのに，特定のサブウィンドウのエントリが不足するために，フロントエンドがストールすることがあり得る．しかし実際には，このことが問題になることはほとんどない．その一方で，命令ウィンドウの非集中化には，1. ロジックの実効サイズの縮小 と，2. クリティカル・パスの分離 の 2 つの効果がある．以下，3.7.1 項と 3.7.2 項で，それぞれについて述べる．

---

* 発表当時.

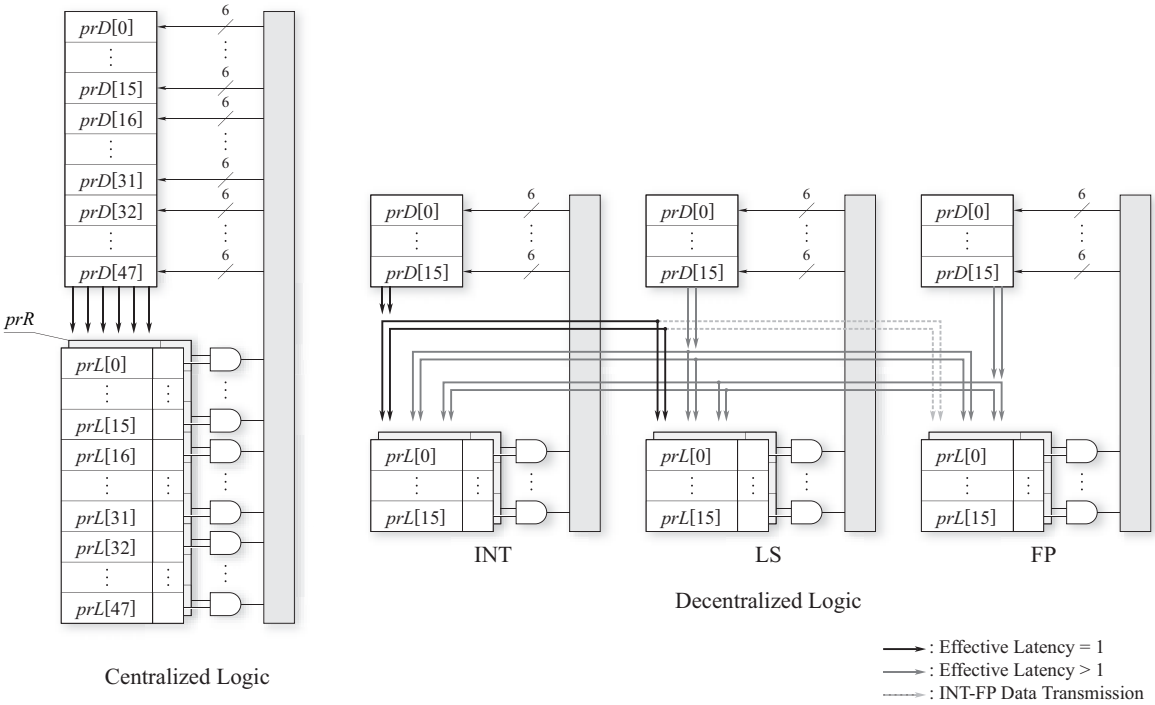


図 3.20: *Wakeup*，*Select*ロジックの非集中化

3.7.1 ロジック実効サイズの縮小

図 3.20 に，MIPS R10000 プロセッサにおける命令ウィンドウ，すなわち，*wakeup*，*select* ロジックの非集中化の様子を示す．R10000 は，INT，LS，FP の 3 つのサブウィンドウ*を持つ．以降では，サブウィンドウのパラメタを，′ (prime) を付けて，$\boldsymbol{IW'}$，$\boldsymbol{WS'}$ のように表すことにする．R10000 では，各サブウィンドウの命令発行幅とサイズはそれぞれ等しく，$IW' = IW/3 = 2$，および，$WS' = WS/3 = 16$ となっている．

前節までに見てきたように，命令ウィンドウを構成する *wakeup* ロジックや，*select* ロジックのほとんどすべてののパラメタは，*IW*，*WS* の増加関数によって与えられる．そのため，非集中化によって命令ウィンドウを構成するロジックのほとんどのパラメタがそれぞれ 1/3 に縮小されることになる．

ただし，連想方式の *wakeup* ロジックにおけるソース CAM のサーチ・ポート数は，例外的に，縮小することができない．例えば，INT 命令の実行によって LS 命令が *wakeup* される場合などがあるため，基本的には，各サブウィンドウのソース CAM はすべてのサブウィンドウのデスティネーション RAM から *prD* を受け取る必要がある．そのため，図 3.20 のように，各サブウィンドウのデスティネーション RAM とソース CAM の接続は完全結合が基本となる．したがって，デスティネーション RAM のリード・ポート数は *IW* から IW' へと縮小できるが，

∗ R10000 では，命令キュー (instruction queue) と呼称される [23].

ソース CAM のサーチ・ポート数は，IW のまま，IW' に縮小することはできない．

ただし実際には，デスティネーション RAM→ソース CAM 間の接続は，個々の事例ごとに対応せざるを得ない．例えば，SPARC など，INT→FPレジスタ間の転送命令が無い命令セット・アーキテクチャ [56] では，同図中破線で記した INT→FP サブウィンドウ間の接続は不要である．MIPS など，INT→FP レジスタ間の転送命令がある命令セット・アーキテクチャであっても，同時に発行できる命令の数を制限するなどして，一部を省略することができる [23]．一方，FP→INT 間の接続は，FP→INT レジスタ間の転送命令だけでなく，FP 命令から FP 条件分岐命令に FP 条件コードを受け渡すために用いられることがある．

3.7.2　クリティカル・パスの分離

同図 3.20 中実線で示した，INT→INT，INT→LS 間の接続は，3.2 節で述べた，命令間の実効レイテンシが 1 サイクルである接続である．この接続を利用する *wakeup* は，*select* と合わせて 1 サイクルで実行する必要がある．逆に言えば，それ以外の接続を利用する *wakeup* フェーズは適当にパイプライン化してもよい．

図 3.7 (p. 62) に，例えば I_x がロード命令である場合など，I_x — I_l 間の実効レイテンシが 2 サイクルの場合のバックエンドのパイプライン動作の様子を示す．図 3.6 の場合と同様，*wakeup* フェーズに 1.5 サイクルを充てているが，図 3.6 の場合と異なり，I_x と I_l は back-to-back に実行できている．

したがって，3 つのサブウィンドウ中では INT サブウィンドウが最もクリティカルであり，*wakeup*，*select* の遅延を考える最には INT サブウィンドウのみを考慮すればよい．

本章のまとめ

本章の結論は，以下のようにまとめられる：

- 3.2 節で述べたように，*wakeup* フェーズと *select* フェーズは，実際上パイプライン化不能であり，合わせて 1 サイクルで実行する必要がある．
- これら 2 つのフェーズのロジックのうち，*select* ロジックの遅延は専らゲート遅延からなるため，LSI の微細化に伴って順調に短縮されると予測される．

  一方 *wakeup* ロジックの遅延は，RAM のワードラインとビットライン，および，CAM のサーチラインとマッチラインといった長い配線の遅延からなるため，LSI の微細化の恩恵を受けにくい．
- 以上の理由により，*wakeup* ロジックは，LSI の微細化にともなっていっそうクリティカルになっていくと予測されている [50, 34, 11, 12]．

次章からは，*wakeup* ロジックを簡単化する，命令間の依存関係を表す行列を用いた *wakeup* の方式について述べる．

第4章　間接方式

本章と次章では，命令間の依存関係を表す行列を用いた命令スケジューリングの方式について述べる．*Wakeup* フェーズにおいて *wakeup* されるべき命令は，発行が決まった命令に依存する命令である（3.1.5 項）．そのため，これらの方式では，命令パイプラインのフロントエンドにおいて検出した命令間の依存関係を行列の形に表しておき，*wakeup* フェーズにおいてこの行列を読み出すことで *wakeup* すべき命令を検出するのである．依存行列を用いたスケジューリング方式では，従来の連想方式で用いられるような連想検索（3.6.2 項）を省略することで，*wakeup* 処理の高速化を図る．

依存行列を用いたスケジューリング方式には，我々の研究室で提案されたものの他，DEC* Alpha 21264 プロセッサで採用されいてるものがある [48, 41, 49].

Wakeup フェーズでは，*select* ロジックからの発行許諾シグナル *grant* を受けて，*select* ロジックへの次のサイクルの発行要求シグナル *req* を求める（3.6 項）．21264 プロセッサで採用されている方式では，*grant* から *req* を求めるにあたって，物理レジスタが利用可能かどうかを表す物理レジスタのフィールド *rdy*（3.1.4，3.1.5 項）を介する．すなわち，まず *grant* にしたがって *rdy* を更新し，次いで更新された *rdy* から次のサイクルの *req* を求めるのである．*grant* から *rdy*，*rdy* から *req* を求めるには，それぞれ別の行列を用いる．

一方，我々の方式では，命令間の依存関係を表す単一の行列を用い，*rdy* を介することなく，直接 *grant* から *req* を求める．

そのため本稿では，我々の方式を**直接方式**，21264 で採用されいてる方式を**間接方式**と呼ぶことにする．

本章では，この間接方式について説明する．間接方式は前章で述べた従来の連想方式との連続性が高いため，間接方式について知っておくことは後述する dualflow アーキテクチャや直接方式の理解の助けとなるだろう．以下，4.1 節で間接方式の原理について説明した後に，4.2 節で間接方式のロジックについて述べる．最後に 4.3 節では，間接方式と連想方式との関係についてまとめる．

4.1　間接方式の原理

本節では，間接方式の原理を説明する．まず，4.1.1 項で間接方式のデータ構造について述べた後，4.1.2 項で，それに対する *wakeup* 時のアクセスについて述べる．

---

* 発表当時.

4.1.1　間接方式のデータ構造

間接方式の命令ウィンドウ

図 4.2 に，図 4.1 の 4 命令がディスパッチされた直後の間接方式の命令ウィンドウの様子を示す．なお，図 4.1 のコードは，3 章で用いた図 3.1 (p. 50) と同じものである．また，図 4.2 は，同じく 3 章で用いた図 3.3 (p. 50) と同じ状況のものである．間接方式の命令ウィンドウは，図 3.3 に示した連想方式のそれと比べると，*rdyL*/*rdyR* フィールドがない点が異なる．

デスティネーション行列，ソース行列

前述したように間接方式では，物理レジスタが利用可能かどうかを表す，物理レジスタ・ファイルのフィールド *rdy* を介して，*grant* から *req* を求める．*grant* から *rdy*，*rdy* から *req* を求めるには，それぞれ別の行列を用いる．本稿では，それぞれを**デスティネーション行列**，**ソース行列**と呼ぶことにする．以下，デスティネーション行列，ソース行列の *i* 行 *j* 列の要素を，それぞれ $\boldsymbol{DST[i][j]}$，$\boldsymbol{SRC[i][j]}$ と表す．ただし，*i*，*j* は，$i = 0, 1, \cdots, WS-1$，$j = 0, 1, \cdots, NR-1$ とする．図 4.1 のコードが図 4.2 のように命令ウィンドウにディスパッチされた直後のデスティネーション行列，ソース行列の状態を図 4.3 の 1. に示す．デスティネーション行列，ソース行列は，それぞれ以下のような行列である：

デスティネーション行列　デスティネーション行列は，命令とそのデスティネーション・オペランドとなる物理レジスタの依存関係を表している．

デスティネーション行列の *i* 行 *DST*[*i*] は，命令ウィンドウの *i* 行の命令のデスティネーション・オペランドの物理レジスタ番号，*prD*[*i*] をデコードしたものである．*DST*[*i*] は，したがって，命令ウィンドウの *i* 行のフィールドの 1 つである．

例えば，図 4.1 では，*prD*[*x*] = %1 であるので，図 4.3 の *x* 行では，%1 列要素が 1，それ以外の要素が 0 となっている．

ソース行列　ソース行列は，命令とそのソース・オペランドとなる物理レジスタの依存関係を表している．

ソース行列の *i* 行 *SRC*[*i*] は，命令ウィンドウの *i* 行の命令のソース・オペランドの物理レジスタ番号，*prL*[*i*] と *prR*[*i*] をそれぞれデコードし，それらを列ごと (column-wise) に OR したものである．したがって *SRC*[*i*] は，デスティネーション行列の場合と同様，命令ウィンドウの *i* 行のフィールドの 1 つである．

例えば，*prL*[*c*] = %2，*prR*[*c*] = %3 であるので，同図 4.3 の *c* 行では，%2 列と %3 列の要素がそれぞれ 1，それ以外の要素が 0 となっている．

デスティネーション行列，ソース行列の *i* 行 *j* 列の要素，*DST*[*i*][*j*]，*SRC*[*i*][*j*] は，それぞれ以下のように書ける：

$$DST[i][j] \;=\; \begin{cases} 1 & (j = prD[i]) \\ 0 & (j \neq prD[i]) \end{cases}$$

label	*opcode*	*prD*	*prL*	*prR*	*immed*
I_x :	op_x	%1 =	%0,		0
I_l :	op_l	%2 =	%1,		1
I_r :	op_r	%3 =	%0,		2
I_c :	op_c	%4 =	%2,	%3	

図 4.1: レジスタ・リネーミングされたコード

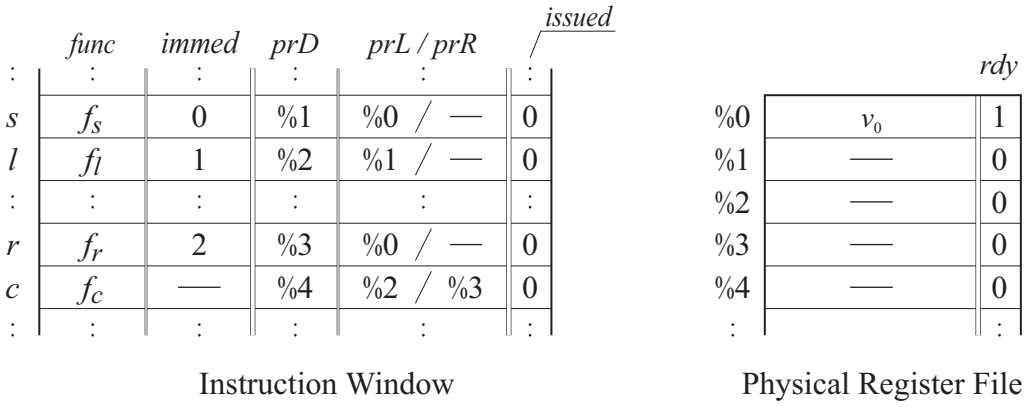


図 4.2: 間接方式の命令ウィンドウ

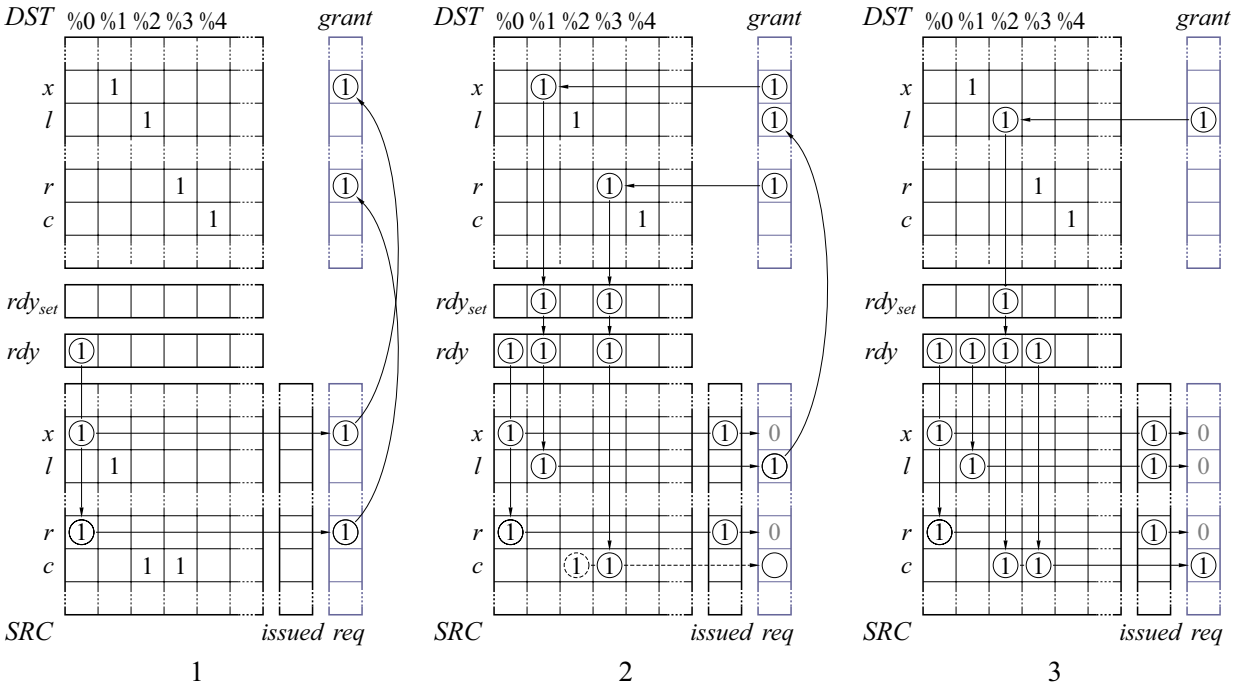


図 4.3: 間接方式の概念図

$$SRC[i][j] \quad = \quad \begin{cases} 1 & (j = prL[i] \vee j = prR[i]) \\ 0 & (j \neq prL[i] \wedge j \neq prR[i]) \end{cases}$$

4.1.2　行列アクセス

図 4.3 に，図 4.1 に示したコードが実行される際の，間接方式の動作を示す．同図 4.3 中，薄く示した格子は組合わせ回路的なシグナルであり，濃く示した格子はメモリの出力である．間接方式の *wakeup* 処理は，以下のように進む:

1. 最初，*rdy*[%0] が 1 であるとすると，物理レジスタ %0 を使用する命令が実行可能である．ソース行列の %0 列を見ると，*x* 行と *r* 行がそれぞれ 1 であるので，I_x と I_r が実行可能であることが分かる．したがって，*req*[*x*] と *req*[*r*] がアサートされる．

   命令 I_x と I_r の発行が同時に許諾されたとしよう．

2. すると，次のサイクルに，*grant*[*x*] と *grant*[*r*] がそれぞれアサートされ，命令 I_x と I_r のデスティネーション・オペランドに割り当てられた物理レジスタが利用可能になるので，対応する *rdy* をセットする．デスティネーション行列の *x* 行と *r* 行を見れば，セットすべき *rdy* の要素が分かる．*x* 行によって *rdy*[%1] が，*r* 行によって *rdy*[%3] がそれぞれセットされる．

   さて，新たにセットすべき *rdy* の要素を表す行ベクトルを，$\boldsymbol{rdy_{set}}$ とする，すなわち，$rdy[j] = rdy[j] + rdy_{set}[j]$ である．今の場合 rdy_{set} は，%1 列と %3 列が 1，それ以外が 0 であればよい．すなわち，rdy_{set} は，*x* 行と *r* 行の列ごとの OR (column-wise OR) によって求められる．

   次いで，*rdy*[%1] と *rdy*[%3] が新たにセットされたことによって，%0，%1，%3 を使用する命令が実行可能になることが分かる．ソース行列の %1 列を見ると，*l* 行が 1 であるので，I_l が実行可能であると分かり，*req*[*l*] がアサートされる．

   一方，%3 列を見ると，*c* 行が 1 であるが，I_c はこのときには実行可能に**ならない**．ソース行列の *c* 行では，%2 列も 1 にセットされている．したがって，物理レジスタ %3 に加え，物理レジスタ %2 も利用可能にならなければ，I_c は実行可能にならない．

   命令 I_l の発行が許諾されたとしよう．

3. *grant*[*l*] がアサートされると，デスティネーション行列の *l* 行の %2 列要素が 1 であるから，*rdy*[%2] がセットされる．

   すると，*rdy*[%2] と *rdy*[%3] がともにセットされたので，今度こそ命令 I_c は実行可能になり，*req*[*c*] がアサートされる．

4.1.3 行列アクセスの式

Wakeup フェーズにおけるデスティネーション行列，および，ソース行列へのアクセスは，それぞれ以下のようにまとめられる：

デスティネーション行列 $grant[i] = 1$ である i 行において，$DST[i][j] = 1$ であれば，$rdy_{set}[j]$ をアサートする．

ソース行列 i 行において，$SRC[i][j] = 1$ であるすべての j に対して，$rdy[j] = 1$ であれば，$req[i]$ がアサートされる．

各アクセスは，以下の式で表すことができる：

$$\begin{aligned} rdy_{set}[j] &= \sum_i DST[i][j] \cdot grant[i] && (4.1) \\ req[i] &= \prod_j (SRC[i][j] \Rightarrow rdy[j]) \\ &= \prod_j \left(\overline{SRC[i][j]} + rdy[j]\right) \\ &= \overline{\sum_j SRC[i][j] \cdot \overline{rdy[j]}} && (4.2) \end{aligned}$$

それぞれの式は，行列 — ベクトル積を用いて，以下のように書ける．ただし，$\overline{A}$ は，ベクトル A の各要素を反転したものとする：

$$\begin{aligned} rdy_{set} &= grant^T \cdot DST && (4.3) \\ req &= \overline{SRC \cdot \overline{rdy^T}} && (4.4) \end{aligned}$$

これらの式からは，デスティネーション行列，ソース行列へのアクセスは，いずれも行列 — ベクトル積であるが，『アクセスの方向が 90° ずれている』ことが分かる．

4.2 間接方式のロジック

前項で見たように，間接方式の *wakeup* におけるデスティネーション行列，ソース行列へのアクセスは，行列 — ベクトル積である．行列 — ベクトル積は，連想方式のデスティネーション RAM のような RAM セル・アレイによって実現できる．

4.2.1 デスティネーション行列

図 4.4 に，デスティネーション行列のロジックを示す．

ベクトル *grant* の各要素の値が，ワードラインによって，図では右から左へと，各列にブロードキャストされる．そして，同図に詳細に描いてある j 列では，出力の行ベクトル $grant \cdot DST$ の j 番目の要素，すなわち，ベクトル *grant* と行列 *DST* の j 列との内積を計算する．より具体

$$\begin{pmatrix} grant[0] & \cdots & grant[WS-1] \end{pmatrix} \begin{pmatrix} \cdots & DST[0][j] & \cdots \\ & \vdots & \\ \cdots & DST[WS-1][j] & \cdots \end{pmatrix} = \begin{pmatrix} \cdots & \sum_i grant[i] \cdot DST[i][j] & \cdots \end{pmatrix}$$

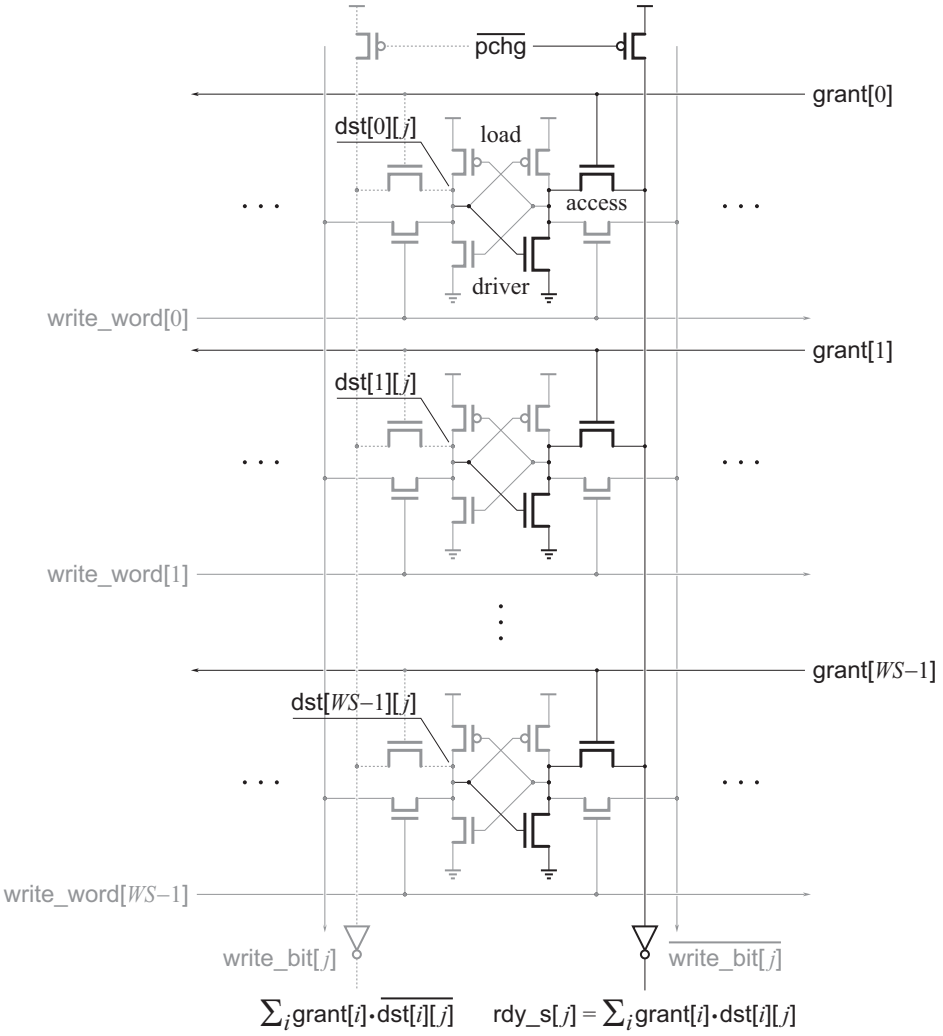


図 4.4: デスティネーション行列のロジック

的には，$\sum_i grant[i] \cdot DST[i][j] = grant[0] \cdot DST[0][j] + grant[1] \cdot DST[1][j] + \cdots + grant[WS-1] \cdot DST[WS-1][j]$ なる積和計算を実行する．

同図 4.4 中，黒色で描かれた部分回路が積和計算を行うダイナミック AND − OR ゲートを形成している（2.1 節）．右側の負論理のビットラインが，プリチャージ・ノードの役目を果たす．このビットラインに接続された 4T セルでは，ドライバ・ゲートとアクセス・ゲートが直列に接続され，2 個の *n*MOS ゲートからなるスタックを形成している．そしてこのビットラインには，*j* 列の各セルのスタックが並列に接続されている．この直列接続が論理積を，並列接続が論理和をそれぞれ実現し，黒色で描かれた部分回路が全体として積和計算を実現するのである．

同図 4.4 から分かるように，このロジックは，構造的には，デスティネーション RAM と同様の RAM セル・アレイである（2.3 節）．

ただし，通常の RAM の RAM セル・アレイとは動作が異なる．通常の RAM では，行デコーダの働きによって，同時には 1 本のワードラインのみがアサートされる．一方このロジックでは，ワードラインにはベクトル *grant* の各要素が入力されるので，同時に複数のワードラインがアサートされることになる．

その結果このロジックでは，構造上は RAM セル・アレイであるにも関わらず，2 レールの出力が得られないことになる．同図 4.4 中，破線で描かれた左側のビットラインに出力されるのは，行ベクトル $grant \cdot \overline{DST}$ であり，右側のビットラインに出力される所望の行ベクトル $grant \cdot DST$ の反転 $\overline{grant \cdot DST}$ **ではない**．

なお，このように，左 / 右の出力の論理式が相補的にはならないのは，ロジックの構造によることであり，通常の RAM にも当てはまることに注意されたい．通常の RAM では，同時には 1 本のワードラインのみがアサートされる結果として，2 つのビットラインに相補的な出力が得られるのである．このロジックの場合で言うと，*grant* の *i* 番目要素 $grant[i]$ のみが 1 であるとすると，右側のビットラインの出力は $grant \cdot DST = DST[i]$，左側のビットラインの出力は $grant \cdot \overline{DST} = \overline{DST[i]}$ となり，相補的になる．

ともかく，このロジックでは 2 レールの出力が得られれるわけではないので，図 4.4 で破線で描かれた，左側のビットライン，および，それに接続されたアクセス・ゲートは不要である．すなわち，行列 — ベクトル積が行われるこのロジックのリード・ポートは，シングル・ビットラインとなる（2.3 節）．

なお，シングル・ビットラインとなるのは行列 — ベクトル積が行われるリード・ポートのみであり，ライト・ポートをどうするかはまた別の問題であることを付記しておく（2.3 節）．

4.2.2 ソース行列

図 4.5 に，ソース行列のロジックを示す．

ソース行列アクセスも，行列 — ベクトル積 $SRC \cdot \overline{rdy}$ であり，デスティネーション行列と同様，RAM セル・アレイによって実装できる．ただし，デスティネーション行列では，行ベクト

$$\begin{pmatrix} \vdots & & \vdots \\ SRC[i][0] & \cdots & SRC[i][NR-1] \\ \vdots & & \vdots \end{pmatrix} \begin{pmatrix} \overline{rdy[0]} \\ \vdots \\ \overline{rdy[NR-1]} \end{pmatrix} = \begin{pmatrix} \vdots \\ \sum_j SRC[i][j] \cdot \overline{rdy[i]} \\ \vdots \end{pmatrix}$$

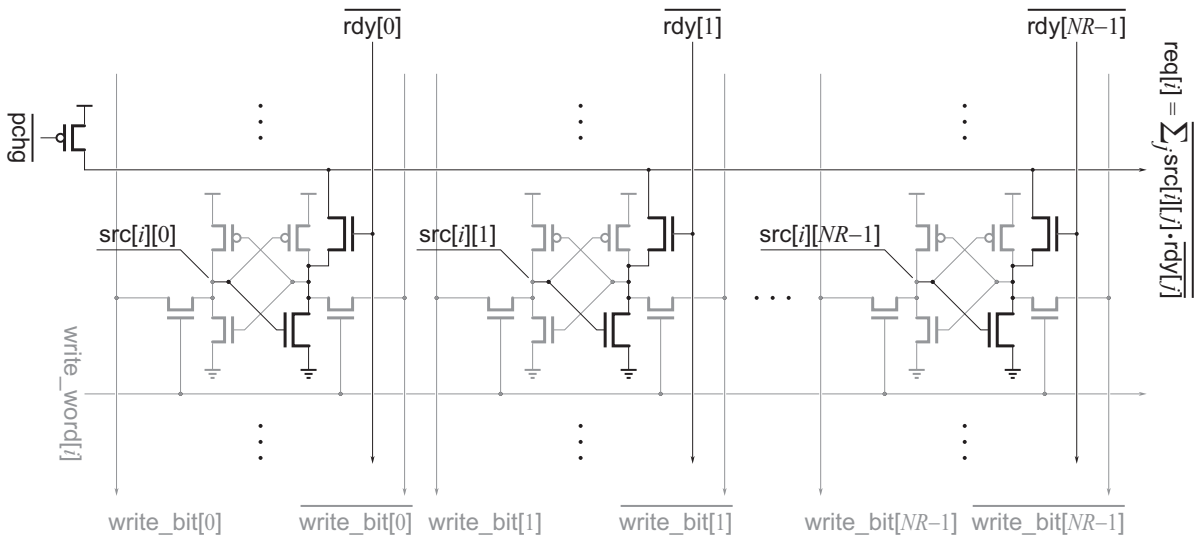


図 4.5: ソース行列のロジック

ル×行列であったものが，ソース行列では行列×列ベクトルとなるので，計算の『向き』が90°ずれることになる．そのため，ソース行列のロジックのリード・ポートでは，図4.5に示されているように，ワードライン $\overline{rdy}$ が列（縦）方向に，ビットライン *req* が行（横）方向に，走ることになる．それ以外の点は，デスティネーション行列と全く同じである．

ソース行列のロジックは，デスティネーション行列のロジックのリード・ポートのアクセス・ゲートのビットラインとワードラインに対するコンタクトを局所的に付け換えることで得られる．したがって，ワードラインが列方向に，ビットラインが行方向に走ること自体のディメリットはない．ただしもちろん，ワードライン，ビットラインの長さが入れ替わるので，遅延時間は変化する．具体的には，ソース行列は *WS* 行 *NR* 列と，行（横）方向に長い形状をしているので，ワードラインが短く，ビットラインが長くなる．7章で詳しく述べるが，このことによって遅延は若干長くなる傾向にある．

4.2.3 *rdy*レジスタ

これまで述べてきたとおり，*rdy* とは，物理レジスタ・ファイルの *rdy* フィールドのことである（3.1.3項）．しかし間接方式における *rdy* は，3.6.1項で述べたRAM方式とは異なり，RAMによって構成されるのではない．図4.3中央に示されているように，各ビットに入出力を備えた1ビット・レジスタの単なるアレイである．このアレイへの入力は，rdy_{set} である．各ビットを記憶するメモリ素子は，機能的にはSR-FFと等価である．

その一方で，ソース行列と *select* ロジックの間には，連想方式における *rdyL*/*rdyR*（図3.17）のようなレジスタは必要なく，単なるパイプライン・ラッチを置けばよい．

4.3 連想方式との関係

デスティネーション行列，および，ソース行列の各行は，基本的には，*prD*，および，*prL*/*prR* をデコードしたものである．したがって，デスティネーション行列，および，ソース行列は，連想方式のデスティネーションRAM，および，ソースCAMとおおよそ対応している．しかし，連想方式より，3.6.1項で述べたRAM方式との方が，ロジックの構成要素間により厳密な対応関係がある．

4.3.1 RAM方式との関係

間接方式のデスティネーション行列，*rdy* レジスタ，および，ソース行列は，RAM方式の以下に示す構成要素と完全な対応関係がある：

デスティネーション行列　デスティネーションRAMと *rdy* RAMのライト・ポート

*rdy*レジスタ　*rdy* RAMのRAMセル・アレイ

ソース行列　*rdy* RAM のリード・ポート

RAM 方式では，*prD* をアドレスとして *rdy* RAM を更新し，*prL*/*prR* をアドレスとして *rdy* RAM を読み出して *rdyL*/*rdyR* を得ていた．一方間接方式では，同様に，デスティネーション行列によって *rdy* レジスタを更新し，ソース行列によって *rdy* を得る．

3.6.1 項では，RAM 方式は，*rdy* RAM のリード・ポート数が多過ぎて実装できないと述べた．したがって間接方式は，RAM 方式における *rdy* RAM の読み出し方法を工夫した方式ととらえることができる．

4.3.2　連想方式との関係

連想方式では，*IW* 個の物理レジスタ番号に対する連想処理を，それぞれ *IW* 個の一致比較器で行っていた．一方間接方式では，物理レジスタ番号をデコードすることによって，1 個の回路でまとめて処理している．間接方式のロジックは，連想方式のロジックに対して，以下のように簡単化されている：

デスティネーション RAM とデスティネーション行列　リード・ポート数が *IW* 本から 1 本に削減される．

ソース CAM とソース行列　一致比較が積和に変わる．ポート数は，*IW* 本から 1 本に削減される．

しかしその一方で，ビット幅は $\lceil \log_2 NR \rceil$ b から *NR* b へと大幅に増加している．これら得失が遅延に与える影響については，7 章で詳しく述べる．

第5章　Dualflowアーキテクチャ

3章で述べたように，従来のスーパースカラ・プロセッサの連想方式の *wakeup* ロジックは，LSIの微細化にともなってクリティカルになっていくと考えられる．

一方，データ駆動型計算機では，スーパースカラ・プロセッサの *wakeup* と同様の処理を，待ち合わせ記憶において発火という形で実現している．しかし，直接待ち合わせ方式のデータ駆動型計算機の待ち合わせ記憶は，スーパースカラ・プロセッサの *wakeup* ロジックのような連想検索を必要としていない [58, 66, 67, 68].

本節では，**dualflowアーキテクチャ**と呼ぶ命令セット・アーキテクチャを紹介する．Dualflowアーキテクチャは，スーパースカラ・プロセッサにデータ駆動的性質を導入することで，スーパースカラ・プロセッサと同様の out-of-order 命令スケジューリングを行いながら，連想方式のような連想処理を省略することができる．

本章では，以下，5.1 節から dualflow アーキテクチャについて述べる．5.1 節で dualflow アーキテクチャの実行モデルについて述べた後，5.4 節以降で実装方法について説明し，導入されたデータ駆動的性質がどのように out-of-order スケジューリングから連想処理を取り除くのかを明らかにする．

5.1　Dualflowアーキテクチャの実行モデル

Dualflow アーキテクチャは，以下のように，制御駆動とデータ駆動の両方の性質を合わせ持つ命令セット・アーキテクチャである：

制御駆動的性質　通常の制御駆動型アーキテクチャと同様のプログラム・オーダを定義する．すなわち，プログラム・カウンタ (PC: Program Counter) を持ち，メモリ上の命令の並びと分岐命令の実行結果によって実行すべき命令を決定する．

データ駆動的性質　通常の制御駆動型アーキテクチャのようなレジスタを定義しない．命令間のデータの授受は，制御駆動のようにレジスタを介して間接的に行われるのではなく，データ駆動型アーキテクチャのように命令間で直接的に行われる．

以下では，まず 5.1.1 項でモデルの全体像について述べた後，5.1.2 項で実行例を用いて説明を行う．

5.1.1 Dualflow アーキテクチャの実行モデルの概要

命令フォーマット

まず，Dualflow アーキテクチャの命令フォーマットについて簡単に触れておく．図 5.1 に命令フォーマットの例を示す．命令中には，通常の制御駆動型プロセッサのような，ソースやデスティネーション・オペランドとなるレジスタを示すフィールドはない．代わりに，命令の実行結果の宛先を示す $\boldsymbol{d_1}/\boldsymbol{d_2}$ フィールドがある．d_1/d_2 フィールドについての詳細は後述する．

各命令は，プログラム・オーダ上で先行する命令から送りつけられたデータを使用して実行を行い，実行結果を d_1/d_2 フィールドが示すプログラム・オーダ上で後続の命令に送りつける．この動作がドミノ倒しのように伝搬して，プログラムが実行される．

各命令は，送りつけられたデータを使用する，すなわち，各命令は自らは使用するデータを選らばないので，命令フォーマットにはソース・オペランドを指定するフィールドは存在しない．

実行モデル

Dualflow アーキテクチャでは，その実行モデルのレベルから out-of-order 実行が想定されており，そのため**命令ウィンドウ**が導入されている．命令ウィンドウの各エントリは，基本的には，3 つのフィールドを持つ．そのうちの 1 つは命令を格納する**命令フィールド**であり，残りの 2 つは命令の左 / 右のソース・オペランド・データを格納する**データ・フィールド**である．

命令ウィンドウの各エントリには，命令と，そのソース・オペランドとなるデータが，別々に届く．命令と必要なデータが揃った命令ウィンドウ・エントリの命令が，プログラム・オーダとは独立に，すなわち，out-of-order に実行される．

以下のように，命令は制御駆動的に，データはデータ駆動的に命令ウィンドウ・エントリに届く：

命　令　通常の制御駆動型アーキテクチャと同様に，メモリ上の命令の並びとと分岐命令の実行結果によってプログラム・オーダが定義される．命令は，プログラム・オーダにしたがって，メモリからフェッチされ，フェッチされた順に命令ウィンドウの**連続する**エントリの命令フィールドに格納される．

データ　命令が実行されると，その実行結果は，データ駆動型アーキテクチャと同様に，命令中 d_1/d_2 フィールドで示される宛先に送られる．ただし，宛先の指定の方法は一般のデータ駆動型アーキテクチャとは異なる．データ駆動型アーキテクチャでは，実行結果の宛先は命令であり，各命令は宛先の命令のアドレスを指示する．一方 dualflow アーキテクチャでは，宛先は命令では**なく**，後続の命令ウィンドウ・エントリである．

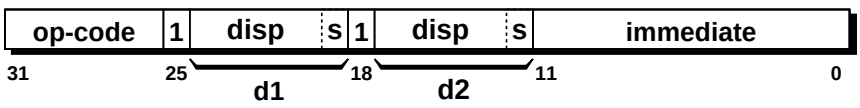


図 5.1: Dualflow アーキテクチャの命令フォーマット例

d_1/d_2 フィールド

ここで再び命令フォーマットに話を戻そう．図 5.1 に示した命令フォーマットにおいて，d_1/d_2 フィールドは，宛先となる命令ウィンドウ・エントリのデータ・フィールドを示す．d_1/d_2 の各フィールド中，*disp* サブフィールドは自命令が格納された命令ウィンドウ・エントリと宛先命令ウィンドウ・エントリ間の変位 (displacement) を示し，*s* サブフィールドは宛先データ・フィールドの左 / 右の別を表す．

なお，ハードウェア量とのトレードオフから，宛先の数は 2，*disp*サブフィールドは 5b を想定している．その場合，ある命令の実行結果を送ることができるのは，距離が 31 命令以内にある最大 2 つのデータ・フィールドに制限される．その制限の妥当性については，5.6 節で検証する．

5.1.2 実行例

では，図 5.2 に示す $|a - b|$ を計算するコードを例に，dualflow アーキテクチャにおける実行の様子を具体的に説明しよう．このコードは，まず，3 行の sub 命令で $d = a - b$ を求める．d が負である場合には，4 行の bneg 命令からラベル NEG が付された 7 行の命令に分岐し，更に $0 - d$ を計算して最終的な結果とする．

図 5.3 に，図 5.2 のコードを実行後の命令ウィンドウの様子を示す．このコードは以下のように実行される：

1. 最初 PC は 1 行を指しているとする．この時点で 4 行の条件分岐命令 bneg までの制御の流れは確定しているので，1〜4 行の各命令を，命令ウィンドウ・エントリ 1〜4 の命令フィールドにそれぞれ格納することができる．

2. 1/2 行の imm 命令は即値を生成する命令で，実行に際してデータを必要としない．したがってフェッチ後直ちに実行されて，値 *a*/*b* を 2L/1R で示される命令ウィンドウ・エントリに送る．1 行の命令の宛先 2L は命令ウィンドウ・エントリ 1 + 2 = 3 の左データ・フィールドを，2 行の命令の宛先 1R は命令ウィンドウ・エントリ 2 + 1 = 3 の右データ・フィールドを，それぞれ示す．

3. 命令ウィンドウ・エントリ 3 は，3 行の sub 命令と，1/2 行の imm 命令から送られたデータの到着によって実行可能となる．その実行結果 *D* は，1L/2L で示される命令ウィンドウ・エントリ 4/5 それぞれの左データ・フィールドに送られる．

   ここで，宛先 2L で示される命令ウィンドウ・エントリ 5 に格納される命令は，条件分岐命令 bneg の結果に依存するので，この時点ではフェッチできないことに注意されたい．したがってこの sub 命令は，**そこにどのような命令が来るかに関わらず**，命令ウィンドウ・エントリ 5 に実行結果を送りつけることになる．一方命令ウィンドウ・エントリ 5 には，命令より先にデータが届くことになる．

4. 命令ウィンドウ・エントリ 5 の命令フィールドには，命令 bneg の実行結果が not taken であれ

```
line label  instruction
  1         imm   a     2L
  2         imm   b     1R
  3         sub         1L, 2L
  4         bneg NEG
  5         mov         2L
  6         b END
  7   NEG:  subr 0      1L
  8   END:  mov         X
```

図 5.2: |*a* – *b*| を計算するコード

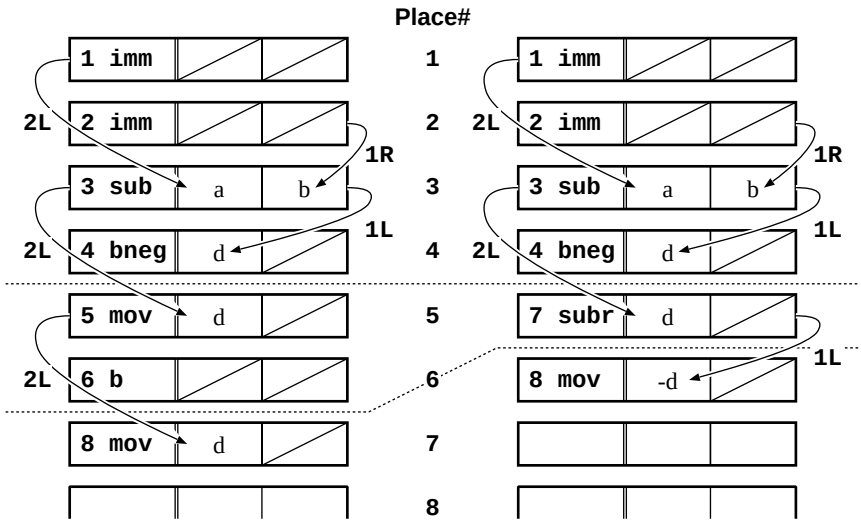


図 5.3: 図 5.2 のコードを実行する際の命令ウィンドウの状態
左は 4 行の条件分岐が not taken の，右は taken の場合

ば 5 行の mov 命令が，taken であれば 7 行の subr 命令が格納される．

以降は taken であった場合（図 5.3 では右）について説明する．この時点で制御の流れはすべて確定する．

5. subr 命令は，sub 命令とは逆に，右オペランドから左オペランドを減ずる命令である．この場合は即値 0 を持っているので，0 – *d* を計算することになる．subr 命令が命令ウィンドウ・エントリ 5 の命令フィールドに格納される時点で，データ *d* は既に到着している．したがってこの命令ウィンドウ・エントリの命令は，フェッチ後，直ちに実行される．
6. 実行結果 –*d* は命令ウィンドウ・エントリ 7 の mov 命令によって X で示される宛先に送られる．4 行の条件分岐 bneg の結果によって実行命令数が異なるため，この mov 命令が格納される命令ウィンドウ・エントリも異なる．not taken の場合は 7，taken の場合は 6 に，それぞれ格納されることになる．

制御駆動型アーキテクチャでは命令が，データ駆動型アーキテクチャではデータが，それぞれ計算の主体であると言われる．そのような観点から言えば，dualflow アーキテクチャでは，命令とデータのどちらかが主でどちらかが従であるということはない．

本節で示したように，dualflow アーキテクチャはデータ駆動的な命令間のデータ受け渡しモデルを採用しているが，それは out-of-order 命令スケジューリングのハードウェアを簡単化するためである．次節以降では，実際にこのことがどのようにハードウェアを簡単化するのかを説明する．

5.2 Dualflow アーキテクチャの命令スケジューリングの原理

本節では，dualflow アーキテクチャにおける out-of-order 命令スケジューリングの原理について説明する．4.1 節と同様に，5.2.1 項で命令スケジューリングのためのデータ構造について述べた後，5.2.2 項で *wakeup* 処理時の動作について述べる．

5.2.1 命令スケジューリングのためのデータ構造

Dualflow アーキテクチャの命令ウィンドウ

前節で述べた実行モデルでは命令ウィンドウのサイズについて言及していなかったが，ハードウェアとして実現するためには有限の命令ウィンドウを考える必要がある．この問題は，命令ウィンドウ・エントリを再利用することによって解決できる．実行を完了した命令を命令ウィンドウ上から順次削除し，空いた命令ウィンドウ・エントリを再利用すればよい．

ただし，命令ウィンドウのサイズ *WS* の最小値は命令フォーマット中の *disp* サブフィールドのビット幅の制約を受ける．例えば *disp* が 5b であるとすると，*WS* は 32 以上とすることが望ましい．*WS* が 32 あれば，たとえすべての命令が 31 命令先の命令ウィンドウ・エントリを宛先として指定したとしても，最も古い命令から順に 1 命令ずつ実行することができる．

label	*opcode*	d_1 *disp*	d_1 *s*	d_2 *disp*	d_2 *s*	*immed*
I_x :	op_x	1	L			0
I_l :	op_l	2	L			1
I_r :	op_r	1	R			2
I_c :	op_c	?	?			

図 5.4: コード

	func	prD_1 / prD_2	*irdy*	*issued*	*dataL*	*rdyL*	*dataR*	*rdyR*
:	:	:	:	:	:		:	
x	f_x	*l*-L / —	1	0	—	1	0	1
l	f_l	*c*-L / —	0	0		0	1	1
r	f_r	*c*-R / —	1	0	—	1	2	1
c	f_c	?? / ??	0	0		0	—	0
:	:	:	:	:	:		:	

Instruction Fields　　Data Fields

図 5.5: Dualflow アーキテクチャの命令ウィンドウ

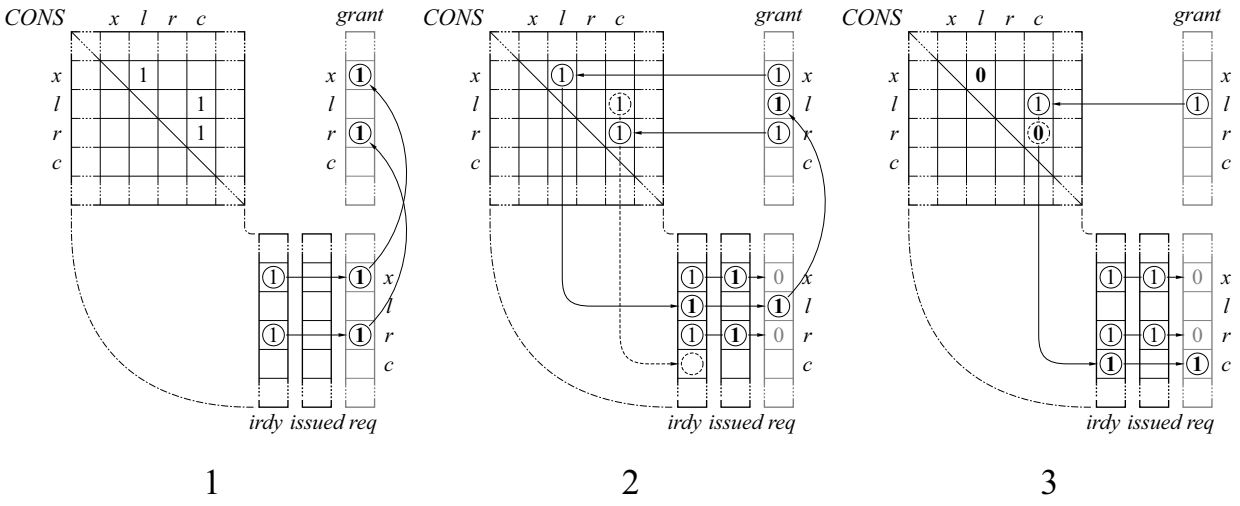


図 5.6: Dualflow アーキテクチャの概念図

図 5.5 に，図 5.4 の 5 命令がディスパッチされた直後の命令ウィンドウの様子を示す．なお，図 5.4 のコードは，3 章で用いた図 3.1 (p. 50) と同じ，図 3.2 に示したデータ・フロー・グラフを持つ．また，図 4.2 は，同じく 3 章で用いた図 3.3 と同じ状況のものである．

Dualflow アーキテクチャの命令ウィンドウは，図 3.3 (p. 50) に示した連想方式や，図 4.2 (p. 91) に示した間接方式の命令ウィンドウと比べると，以下の点が異なる．以下，i は，$i = 0, 1, \cdots, WS-1$ とする：

- 前節で述べたように，命令ウィンドウ・エントリに左 / 右のソース・オペランド・データを格納するデータ・フィールド ***dataL*/*dataR*** がある．
- 即値は *dataL*/*dataR* フィールドに格納すればよいので，*immed* フィールドも必要ない．通常の命令では，即値は *dataR* フィールドに格納する．前節で述べた実行例中の subr 命令のように，左 / 右のソース・オペランドを逆に操作する命令では，即値は *dataL* フィールドに格納する．
- プログラム・オーダ上で連続する命令は連続するエントリに格納される．通常のスーパースカラ・プロセッサの場合には，必ずしも連続するエントリに格納されている必要はない（3.5 節）．
- ソース・オペランドを示す *prL*/*prR* フィールドがない．その代わり，実行結果の宛先を示す ***prD*$_1$/*prD*$_2$** フィールドがある．prD_1/prD_2 フィールドは，宛先データ・フィールド *dataL*/*dataR* の物理的なアドレスである．prD_1/prD_2 は，

$$\begin{cases} prD_1[i].disp &= (i + d_1[i].disp) \bmod WS \\ prD_2[i].disp &= (i + d_2[i].disp) \bmod WS \end{cases}$$

  により，容易に求めることができる．ただし，mod はモジュロ演算を表す．
- 連想方式と同様に，*rdyL*/*rdyR* フィールドがある．*rdyL*/*rdyR* フィールドは，*dataL*/*dataR* フィールドに格納されるソース・オペランド・データの利用可能性を表す．
- *rdyL*/*rdyR* とは別に，当該命令が発行可能であることを示す ***irdy*** と呼ぶフィールドがある．*irdy* と *rdyL*/*rdyR* の関係については，以下で詳しく述べる．

irdy* と *rdyL*/*rdyR

Dualflow アーキテクチャの命令ウィンドウの *rdyL*/*rdyR*，および，*irdy* フィールドは，連想方式では，物理レジスタ・ファイルの *rdy* フィールド，および，命令ウィンドウの *rdyL*/*rdyR* フィールドと同様に，ソース・オペランド・データの利用可能性，および，命令の実行可能性を表す．

	ディスパッチ時に参照	*req* の計算
連想方式	物理レジスタ・ファイルの *rdy*	命令ウィンドウの *rdyL*/*rdyR*
Dualflow	命令ウィンドウの *rdyL*/*rdyR*	命令ウィンドウの *irdy*

表 5.1: 連想方式と Dualflow アーキテクチャの *rdy*，*rdyL*/*rdyR*，*irdy* フィールドの関係

ただし，それらの役割は互いに異なっている．表 5.1 に，これらのフィールドの関係をまとめる．連想方式では，命令が命令ウィンドウにディスパッチされるとき，物理レジスタ・ファイルの *rdy* フィールドが命令ウィンドウの *rdyL*/*rdyR* の初期値を提供し，以降 *rdyL*/*rdyR* から *req* が生成される（3.6.2 項）．それに対して dualflow アーキテクチャでは，命令が命令ウィンドウにディスパッチされるときには，命令ウィンドウの *rdyL*/*rdyR* が *irdy* の初期値を提供し，*irdy* から *req* が生成される．すなわち，$irdy[i] = rdyL[i] \cdot rdyR[i]$，$req[i] = irdy[i] \cdot \overline{issued[i]}$ である．

つまり，dualflow アーキテクチャの *rdyL*/*rdyR* フィールドは，命令がディスパッチされたときに既にデータが揃っているかどうかを判定するためにのみ，存在している．

ディスパッチ時に既にデータが揃っていた場合には，命令ウィンドウの *rdyL*/*rdyR* の両方がセットされており，それを見て *irdy* もセットされる．ディスパッチ時にデータが揃っていなかった場合には，命令ウィンドウの *rdyL*/*rdyR* のいずれかがセットされておらず，*irdy* もセットされない．その場合，*irdy* をセットする，すなわち，命令を *wakeup* するのは，以下に述べるコンシューマ行列の役割となる．

コンシューマ行列

Dualflow アーキテクチャにおける *wakeup* の処理は，命令間の依存関係を表す行列に基づいて行われる．依存行列の各行は，各命令のコンシューマを表すため，**コンシューマ行列**と呼び，***CONS*** で表す．

図 5.5 の命令ウィンドウと同時刻のコンシューマ行列の様子を図 6.3 の 1. に示す．コンシューマ行列は命令間の依存関係を表す．したがって，間接方式のデスティネーション行列，ソース行列がそれぞれ WS 行 × NR 列であったのに対して，コンシューマ行列は WS 行 × WS 列の正方行列となる．また，対角要素は使用しない．命令ウィンドウの p 行の命令 I_p の実行結果を，同じく c 行の命令 I_c が使用する場合 $(p, c = 0, 1, \cdots, WS-1)$，$CONS[p][c] = 1$ となる．

命令 I_i が，命令ウィンドウの i 番エントリ $(i = 0, 1, \cdots, WS-1)$ にディスパッチされるとき，命令 I_i のコンシューマを表すベクトルが $CONS[i]$ に書き込まれる．$CONS[i]$ の内容は，$prD_1[i]$/$prD_2[i]$ から容易に求めることができる．$prD_1[i].disp$/$prD_2[i].disp$ をそれぞれデコードし，ビットごとの OR (bit-wise OR) をとればよい．

ここで振り返ってみると，4 章で述べた間接方式のデスティネーション行列とソース行列は，命令間の依存関係を物理レジスタを介して間接的に表現したものとみなすことができる．実際，コンシューマ行列 $CONS$ は，間接方式のデスティネーション行列 DST とソース行列 SRC を用いて，以下のように表すことができる：

$$CONS = DST \cdot SRC^T \ . \tag{5.1}$$

5.2.2　コンシューマ行列アクセス

図 5.6 に，図 5.4 に示したコードが実行される際の，コンシューマ行列の様子を示す．同図 5.6 中，薄く示した格子は組合わせ回路的なシグナルであり，濃く示した格子はメモリの

出力である．Dualflow アーキテクチャの *wakeup* 処理は，以下のように進む:

1. 最初，$irdy[x]$，$irdy[r]$ がセットされており，命令 I_x と I_r が実行可能であることを示している．そのため，$req[x]$，$req[r]$ がアサートされる．

   ここで，命令 I_x と I_r の発行が同時に許諾されたとしよう．

2. すると，次のサイクルに，$grant[x]$ と $grant[r]$ がそれぞれアサートされる．

   x 行を見ると，l 列が 1 である．したがって，命令 I_l は I_x に依存しており，I_x の発行に伴って実行可能になると分かる．そこで，$irdy[l]$ がセットされ，$req[l]$ がアサートされる．

   r 行を見ると，c 列が 1 である．しかし，I_c は実行可能に**ならない**．c 列では，r 行の他に l 行もセットされており，命令 I_c は，I_r の他に I_l にも依存していることが分かる．命令 I_l の発行はまだ許諾されていないので，I_c はこのサイクルでは実行可能にならない．

   ここでは，命令 I_l の発行が許諾されたとしよう．

3. $grant[l]$ がアサートされると，今度こそ I_c が実行可能になり，$req[c]$ がアサートされる．

5.2.3 コンシューマ行列アクセスの式

以下，4.1 節の rdy_{set} の場合と同様に，ベクトル $\boldsymbol{irdy_{set}}$ を用い $irdy = irdy + irdy_{set}$ とする．*Wakeup* フェーズにおける *CONS* へのアクセスは，以下のようにまとめられる:

コンシューマ行列 j 列において，$CONS[i][j] = 1$ であるすべての i に対して，$grant[i] = 1$ であれば，$irdy_{set}[j]$ がアサートされる．ただし，$i, j = 0, 1, \cdots, WS-1$ である．

4.1 節(p. 89) で述べた，以下の間接方式のソース行列へのアクセスと比較されたい:

ソース行列 i 行において，$SRC[i][j] = 1$ であるすべての j に対して，$rdy[j] = 1$ であれば，$req[i]$ がアサートされる．ただし，$i = 0, 1, \cdots, WS-1$，$j = 0, 1, \cdots, NR-1$ である．

これから分かるように，dualflow アーキテクチャにおけるコンシューマ行列 *CONS* へのアクセスは，間接方式におけるソース行列 *SRC* へのアクセスと相似である．基本的には，ソース行列アクセスにおける *req*，*SRC*，および，*rdy* を，$irdy_{set}$，$CONS^T$，および，$grant^T$ に置換すれば，コンシューマ行列アクセスを得ることができる．

コンシューマ行列 *CONS* へのアクセスは，以下の式で表すことができる．やはり，4.1 節で述べた，間接方式のソース行列へのアクセスの式 4.2 と比較されたい:

$$\begin{aligned} req[i] &= \prod_j (SRC[i][j] \Rightarrow rdy[j]) \\ &= \prod_j \left(\overline{SRC[i][j]} + rdy[j]\right) \\ &= \overline{\sum_j SRC[i][j] \cdot \overline{rdy[j]}} \\ irdy_{set}[j] &= \prod_i (CONS[i][j] \Rightarrow grant[i]) \end{aligned} \tag{4.2}$$

$$= \prod_i \left(\overline{CONS[i][j]} + grant[i]\right)$$
$$= \overline{\sum_i CONS[i][j] \cdot \overline{grant[i]}}\,. \tag{5.2}$$

式 5.2 は，行列 — ベクトル積を用いると以下のように書ける．同様に，4.1 節で述べた，間接方式のソース行列へのアクセスの式 4.4 と比較されたい：

$$req = \overline{SRC \cdot \overline{rdy^T}} \tag{4.4}$$
$$irdy_{set} = \overline{CONS^T \cdot \overline{grant}}\,. \tag{5.3}$$

5.3　Dualflow アーキテクチャの *Wakeup* ロジック

前節で述べたように，コンシューマ行列 *CONS* に対するアクセスは，間接方式のソース行列に対するアクセスと似ている．コンシューマ行列は，ソース行列のそれとほぼ同じロジックによって実装される．

ただし，*CONS* に対する入力 *grant* は，組合わせ回路的出力であることに注意する必要がある．それに対して，間接方式のデスティネーション行列に対する入力である *rdy* は，*rdy* レジスタの出力である．4.1 節の図 4.3 (p. 91) と前節の図 5.6 の例で，命令 I_l の発行が許諾され，命令 I_c が実行可能になるサイクル 3 における *r* 列の入力をそれぞれ比較されたい．間接方式のソース行列への入力である *rdy*[*r*] は，その前のサイクル 2 から引き続きアサートされ続けている．サイクル 3 では，*rdy*[*l*] と *rdy*[*r*] が揃ってアサートされるので，出力 *req*[*c*] を正しくアサートすることができる．それに対して，直接方式のコンシューマ行列 *CONS* への入力である *grant*[*r*] は，サイクル 3 ではアサートされていない．そのため，ソース行列と同じロジックを用いると，サイクル 3 でも $irdy_{set}$ はアサートされなくなってしまう．

この問題に対する対応策のうち，最も単純に思い付くものは，単に *grant*[*r*] をアサートし続けるようにするというものであろう．しかし，この方法は実際にはあまりうまくいかない．前節では述べなかったが，図 5.6 では，サイクル 2 において *CONS*[*r*][*c*] をリセットしておくことによって，この問題に対処している．以下，2 つの対処法について順に述べる．

5.3.1　*grant* のホールド

必要なサイクルまで *grant* をアサートし続けるため，間接方式の *rdy* と同様，依存行列の前にレジスタを設置することを考える．このレジスタ，および，その出力を ***granted*** と呼ぶことにする．

そうした場合，このレジスタ *granted* をいつまでセットしておけばよいかという疑問が生じる．

granted[*r*] と *iwe*[*r*]

granted をいつまでセットしておけばよいかという問題は，命令ウィンドウ・エントリの寿命の問題に帰着する．命令ウィンドウ・エントリは，可能な限り速やかに解放することが望ましい．さもないと，命令ウィンドウ・エントリの不足により IPC が低下してしまうからである．本稿全体を通じて問題にしているとおり，命令ウィンドウ・エントリは極めて高価な資源であり，無駄にはできない．

投機失敗時の回復の方式などにも依存するが，原理的には，命令ウィンドウ・エントリの寿命は，格納された命令が発行されるまでである（3.3 節）．前節の例で言えば，*iwe*[*r*] は，*grant*[*r*] がアサートされたサイクル 2 の最後には解放してよく，サイクル 3 には別の命令のために再利用してもよい．

しかし *granted* レジスタを用いる場合には，命令が発行された後も解放 / 再利用することができなくなる．*iwe*[*r*] を再利用するためには，*granted*[*r*] を初期状態に戻す，すなわち，一旦リセットする必要がある．逆に言えば，命令 I_r が他の命令を *wakeup* するために *granted*[*r*] がセットされている間は，別の命令が *iwe*[*r*] を再利用することもできない．すなわち，*granted*[*r*] は *iwe*[*r*] の 1フィールドと考えることができ，*iwe*[*r*] の寿命は *granted*[*r*] のそれによってバウンドされることになる．

granted[*r*] の解放 / 再利用のタイミング

では，*granted*[*r*] は，いつになれば解放 / 再利用できるのだろうか．*granted*[*r*] がその役割を果たすためには，以下に述べる期間中セットされている必要がある；すなわち，依存行列の *r* 列にセットされている要素があり，その要素によって *wakeup* される命令が命令ウィンドウ内に存在している．逆に，そのような命令がすべて *wakeup* されてしまえば，*granted*[*r*] とともに *iwe*[*r*] を解放 / 再利用してもよい．前節の例では，*granted*[*r*] は，サイクル 3 にセットされていて *req*[*c*] をアサートする役割を果たし，それ以降のサイクルでは（新たに I_r に依存する命令がディスパッチされなければ）解放 / 再利用してよい．

解放 / 再利用のタイミングの検出

しかし，依存する命令がすべて *wakeup* されたことを検出することは，実際には容易ではない．例えば，コンシューマ行列を用いてこのタイミングを検出するには，*wakeup* された列の要素すべてをリセットするとともに，各行にセットされている要素があるかどうかを検出する方法が考えられる．その場合，前者のためには，列をリセットするリセット・ポートが必要となる．各行にセットされている要素があるかどうかを検出するには，間接方式のデスティネーション行列のように各行を読み出すリード・ポートが必要となる（4.2.1 項）．

結局，レジスタ *granted* を用いる方法は，後述する列リセット方式より高コストになる．その上，たとえ最速で解放できたとしても，命令ウィンドウ・エントリの寿命は，*granted* がセットされている期間の分だけ長くなるため，IPC の低下が避けられない．

間接方式の *rdy* レジスタ

なお，間接方式の *rdy* レジスタの場合には，*granted* の場合のような問題は発生しないことに注意されたい．*rdy* はそもそも物理レジスタ・ファイルのフィールドであり，*rdy* によって

寿命が延びるとするなら，それは，命令ウィンドウ・エントリではなく物理レジスタである．

また，*granted* のために命令ウィンドウ・エントリの寿命が延びたように，*rdy* のために物理レジスタの寿命が延びることは**ない**．ある物理レジスタが解放されるのは，その内容を参照する命令が存在しなくなったからである（3.3 節）．端的に言えば，*rdy* フィールドはその物理レジスタが利用可能であることを表すのだから，そのレジスタを使用する命令が存在しないのにセットされている必要はない．設計の流儀に従って，物理レジスタが解放されるときにリセットしてもよいし，他の命令に割り当てられるときにリセットしてもよい．

5.3.2 列リセット方式

2 つ目の対処法として，*granted* によって入力をホールドするのとは逆に，入力を受ける行の要素をリセットしてしまうことが考えられる．前節で用いた，*grant*[*r*] と *grant*[*l*] がそれぞれサイクル 2 とサイクル 3 にアサートされた例の場合，以下のようになる：

2. （*grant*[*x*] と）*grant*[*r*] がアサートされている．

   r 行を見ると，*c* 列が 1 である．しかし *c* 列では，*r* 行の他に *l* 行もセットされており，命令 I_c は，I_r の他に I_l にも依存していることが分かる．命令 I_l の発行はまだ許諾されていないので，I_c はこのサイクルでは実行可能にならない．

   その後，*r* 行をリセットする．この場合，実際には *c* 列要素だけが，1 から 0 に変わる．

3. *grant*[*l*] がアサートされている．

   l 行を見ると，やはり，*c* 列が 1 である．*r* 行要素はサイクル 2 においてリセットされたので，*c* 列では *l* 行要素のみが 1 であり，命令 I_c は命令 I_l にのみ依存しているかのように見える．そのため，今度こそ I_c は実行可能になり，*req*[*c*] がアサートされる．

5.3.3 ロジックの構成

行をリセットするには，専用のポートを用意するのが最も低コストであろう．各セルに対しては，小型の *n*MOS トランジスタを 1 つ追加するだけでよい．

なおこの方法は，行列上に保存された依存関係を発行時に破壊するため，投機失敗時の状態回復の方法が制限される．しかし，このことに依存しない効率のよい回復法も提案されている [41].

5.4 Dualflow アーキテクチャの実装

本節では，dualflow アーキテクチャの基本的な実装方法について説明し，前節で述べた実行モデルが実装におよぼす効果を明らかにする．本節では主に，out-of-order スケジューリン

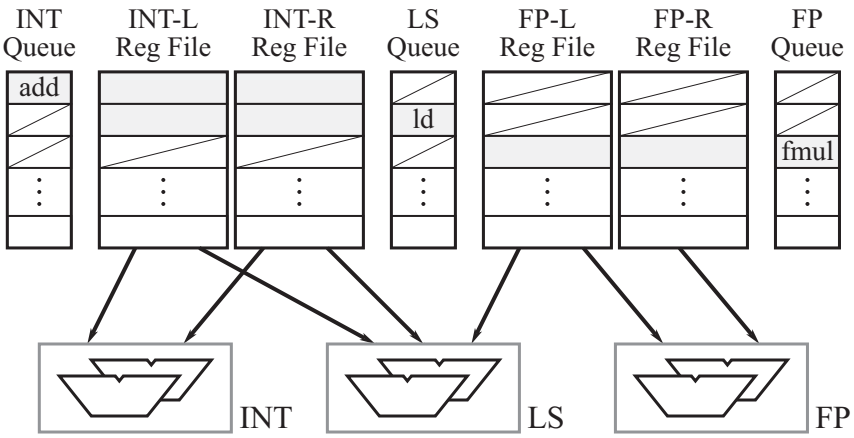


図 5.7: 命令ウィンドウの非集中化

グを行うロジックについて述べる．それ以外の部分はスーパースカラ・プロセッサと同じと考えてよい．

Dualflow アーキテクチャに基づくプロセッサの実装は，原理的には，図 5.5 に示した命令ウィンドウをほぼそのままの形でハードウェア化すればよい．ただしもちろん，単に 1 個の RAM を用いて実装したのでは，ポートの数が多くなりすぎて現実的ではない．3.7 節で述べたように，適切に非集中化することが望ましい．

図 5.7 に，命令ウィンドウの実装例を示す．この例では，MIPS R10000 プロセッサにならい，INT，LS，FP の命令の系統ごとの 3 つのサブ命令ウィンドウと，INT，FP の 2 つのレジスタ・ファイルに非集中化されている．

非集中化された各サブ命令ウィンドウ，および，各レジスタ・ファイルでは，同一の ID をもつエントリは 1 つの命令によって占有されるという点に注意する必要がある．図 5.7 の例では，最上段のエントリには INT 命令の add が格納されているため，LS，FP サブ命令ウィンドウ，および，FP レジスタ・ファイルでも，最上段エントリは使用することができない．

レジスタ・ファイル

INT/FP の各レジスタ・ファイルは，dualflow アーキテクチャでは更に，左 / 右のソース・オペランドごとに自然に分割することができる．通常のデータ駆動型の命令セット・アーキテクチャでは，実行結果が左 / 右のどちらのソース・オペランドとして使用されるか，定義側の命令には分からないのに対して，dualflow アーキテクチャでは，それが d_1/d_2 フィールドに明示されるためである．

レジスタ・ファイルを左 / 右に分割することで，リード・ポート数をそれぞれ 1/2 に縮小することができる．その一方で dualflow アーキテクチャでは，1 つの命令が実行結果を最大 2 箇所のデータ・フィールドに送るため，ライト・ポート数は 2 倍必要になる．結局，個々のレジスタ・ファイルのポートの総数は，スーパースカラ・プロセッサと同じになる．

5.5　Dualflow アーキテクチャのコード生成

Dualflow アーキテクチャの評価のため，C コンパイラとインタープリタを作成した．以下，まず 5.5.1 節では，ベースとした用いた GNU C コンパイラ，GCC についてまとめる．次いで，5.5.2 節で Dualflow アーキテクチャ専用パスについて述べた後，5.5.3 節で実装した最適化の手法について説明する．

5.5.1　GCC

Dualflow アーキテクチャの C コンパイラを作成するにあたっては，GCC (ver.2.8.1) をベースにした．GCC を通常の制御駆動のプロセッサにポーティングするには，基本的には，Machine Description File と Target Macro という 2 つのファイルをプロセッサの構成に合わせて記述するだけですむ．Dualflow アーキテクチャにポーティングするには，それに加えて専用のパスを追加する必要があるが，多くのパスはそのまま流用することができる．

RTL と疑似レジスタ

GCC は，入力ファイルを解析し，**Register Transfer Language**（**RTL**）と呼ぶ内部表現を生成する．以下のパスでの作業は，RTL に対して行われる．

GCC はまず，無限の数の **疑似レジスタ** があると仮定して RTL を生成する．疑似レジスタは，静的ではあるが，個々のデータを識別できるタグである．疑似レジスタは，下流のパスで，ハードウェアのレジスタに変換される，すなわち，レジスタ割り当てが行われる．

スーパースカラ・プロセッサは，タグから変換されたレジスタ番号から，また動的にタグを求める．一方 dualflow アーキテクチャでは，疑似レジスタは直接的に宛先に変換され，実行時には宛先をやはり直接的に参照する．

疑似レジスタから宛先への変換は，ごく基本的には，ある疑似レジスタを定義する命令から参照する命令までの命令数を数えればよい．

逆に，一旦レジスタ割り付けが行われてしまった後では，データの同一性に関する情報が失われてしまうので，そこから静的に宛先を求める作業は簡単ではない．Dualflow アーキテクチャの実行コードを得るにあたって，既存のマシンのアセンブリ・コードからのトランスレータを製作するのではなく，GCC を流用する方法を選んだのは，主にこの理由による．

パス

GCC は，以下のパスにしたがって処理を行う：

1. 構文解析．RTL の生成.
2. 最適化．データ・フロー解析.
3. レジスタ割り当て．Spill-out 処理.
4. Peep-hole 最適化.

5. アセンブリ・コードへの変換.

Dualflow アーキテクチャ専用パス群は，疑似レジスタに基づく RTL を入力とする必要がある．また，4. Peep-hole 最適化は，ハードウェア・レジスタに基づいて行われるので利用しづらく，利用価値も低い．したがって Dualflow アーキテクチャ専用パス群は，3. レジスタ割り当てと 4. Peep-hole 最適化の代わりに組み込む．5. アセンブリ・コードへの変化を行うパスは，そのまま利用できる．

レジスタ割り当てより前のパス 1.，2. は，dualflow アーキテクチャ向けコンパイラでも全くそのまま利用することができる．特に，主要な最適化処理のほとんどとデータ・フロー解析がこの部分で行われていて，非常に都合がよい．ここで行われる最適化の中には，定数の畳み込み，計算強度の軽減，分岐のスレッド化 (jump threading)，無用命令の削除，共通部分式の削除，ループ最適化，局所命令スケジューリングなどがある．

したがって Dualflow アーキテクチャ専用パスには，最適化，データ・フロー解析済の RTL が渡されることになる．Dualflow アーキテクチャ専用パスついては，次節で詳しく述べる．

5.5.2 Dualflow アーキテクチャ専用パス群

Dualflow アーキテクチャは，データ駆動的性質を導入することによって out-of-order 命令スケジューリング・ロジックを大幅に簡略化するが，その代償として，コードはデータの授受に関する制約を受ける．それは，宛先データ・フィールドを静的に計算するという制約である．制約は，以下の 3 つに分類できる：

数と距離 2 個を越えるデータ・フィールド，あるいは，32 以上離れたデータ・フィールドにはデータを送れない．

条件分岐 データの授受が条件分岐を越える場合にも，分岐の結果によって宛先を変えることはできない．

基本ブロック データの授受が 1 個以上の基本ブロックを越える場合には，命令間の距離，すなわち，宛先データ・フィールドの値を静的に求めるとができない．関数呼び出し，if-then-else 構造，ループなどを越えたデータの授受が，これにあたる．

上記 3 つ制約は，それぞれ以下のようにして満たすことができる：

数と距離 データを中継するための mov 命令を挿入する．

各 mov 命令は宛先を 2 つ持てるので，2 分木を組むことができる．2 分木の形状は，数と距離を考慮して決める．

条件分岐 taken/not taken 側それぞれの基本ブロック内部で適当に mov を挿入することによって受け取るデータ・フィールドの配置を一致させる．

基本ブロック 関数呼び出しをまたぐ場合には，距離を静的に決めることは原理的にできない．そのため，メモリを介してデータの授受を行う必要がある．これはちょうど，制

御駆動型アーキテクチャにおいてすべてのレジスタを caller-save とした場合と同じと考えればよい．

その他の場合には，データを授受する命令間にある各基本ブロックに中継のための mov 命令を挿入する．図 5.2 に示したコードでは，5 行の mov がこれにあたる．

なおいずれの場合でも，距離が長くなりすぎるようであれば，メモリを介してデータの授受を行う方法を採る．

それぞれの処理において，単純な実装では大量の mov 命令が挿入されてしまうだろう．このような mov に対しては，さまざまな最適化手法が考えられる．今回は，最も基本的と思われる条件分岐に対する最適化のみを実装した．次節では，それについて詳しく述べる．

5.5.3 条件分岐に関する最適化

データの授受が条件分岐をまたぐ場合，データを受け取るデータ・フィールドの配置を，taken 側と not taken 側で同一にしなければならない．

例えば，図 5.3 に示したコードの実行例の場合，命令ウィンドウ・エントリ 3 の実行結果のうち 2L で示されるものは，命令ウィンドウ・エントリ 4 の条件分岐命令 bneg を越えて，命令ウィンドウ・エントリ 5 の左データ・フィールドに渡される．この命令ウィンドウ・エントリ 5 の命令フィールドには，bneg 命令の結果によって 7 行か 5 行の命令がディスパッチされるが，どちらの場合でも正しく動作するように 7 行，5 行に置く命令を調整する必要がある．

最も簡単には，データのそれぞれに対して，それを受け取る mov を各基本ブロックの先頭に配置すればよい．もちろんそのような方法では受け入れ難い数の mov が挿入されてしまうので，それをどう削減するかが問題である．

アルゴリズム

では図 5.8 に示す例を用いて，今回実装した最適化技法について説明しよう．図中の上側に示す命令列が入力である．図中の A，B，C は，それぞれに対応するデータを上流の基本ブロックから受け取るデータ・フィールドを表す．したがって，A，B，C の配置を not taken 側/taken 側で合わせればよい．

処理は，以下の条件下で実行される：

- このパスを含む Dualflow アーキテクチャ専用パス群の入出力は，実際には，疑似レジスタに基づいた RTL である．前述したように入力される RTL は既にデータ・フロー解析済である．
- 前節で述べた 3 種の制約のうち，に対する処理はこのパスの後で実行される．したがってこのパスでは，の制約がないものとして処理を行えばよい．
- 分岐確率に偏りがある場合とない場合では最適化の方法を変えた方がよいのは明白であるが，今回はまず分岐確率に偏りがあるものとして実装した．

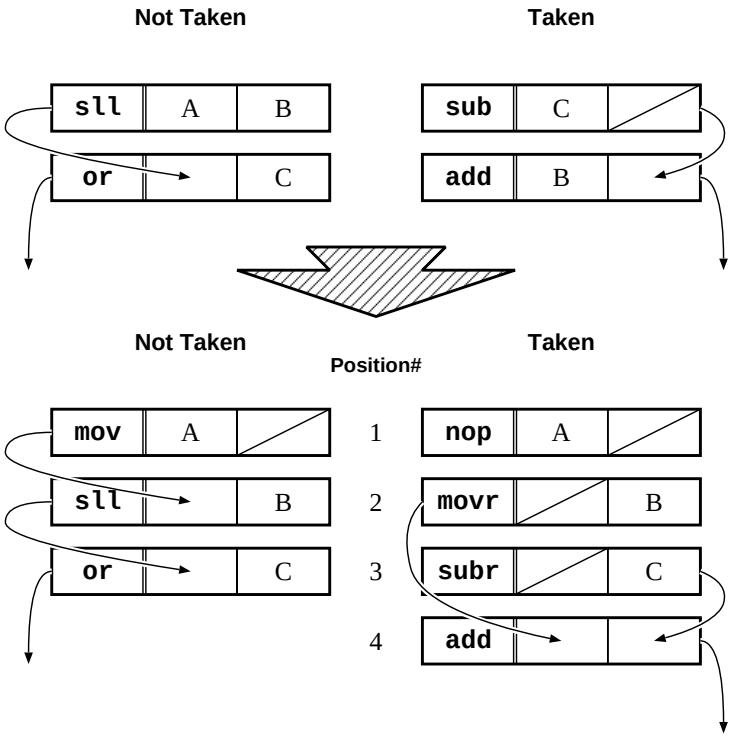


図 5.8: 条件分岐の処理

基本的には，分岐確率が高い側に最適化し，低い側でつじつまを合わせる．例では，not taken 側が分岐確率が高いものとする．処理は，以下のように進められる：

1. **競合の解消**

   受け取り位置に競合がある場合，まずその状態を mov 命令により解消する．図 5.8 では，命令 sll はデータ A，B を；命令 add は B のみを受け取っている．not taken 側に A を受け取る mov を挿入し，A と B を受け取る命令を分離する．

2. **再スケジューリング**

   分岐確率が高い方の基本ブロックに対し局所命令スケジューリングを施し，1. で挿入された mov 命令の位置を最適化する．

   実際には，GCC の命令スケジューリング・パスを呼び出し，コード中の全ての基本ブロックに対して再び命令スケジューリングを施すこととした．

3. **データ・フィールドの配置の整合**

   分岐確率が高い側の基本ブロックの受け取りデータ・フィールドの配置に合わせて，低い側の基本ブロック内の命令を並び替える．

   ソース・オペランドの左右を入れ換えられる命令では，必要に応じて入れ換えを行う．入れ換えは分岐確率の高い側でも行ってよい．なお，ソース・オペランドの左右を入れ換え

られる命令には，ロード（アドレス計算），加減乗算（整数および浮動小数点数），論理演算（and，or，xor），そして，mov，nop がある．

図 5.8 では，taken 側の命令を再配置する．位置 1 には，データ A を握り潰すための nop 命令を置く．左右を入れ換えれば位置 2 に add 命令を置くことは可能だが，そうすると sub 命令との先行制約が満たせない．したがって add 命令は後にまわして，位置 2 には movr 命令を置く．また位置 3 には，sub 命令の左右ソース・オペランドを入れ換えた subr 命令を置くことができる．

この処理の結果，図 5.8 下に示す命令列を得る．

5.6 Dualflow アーキテクチャの性能評価

Dualflow アーキテクチャは，明らかに一時的なデータに対して極端に最適化されているので，一時的ではない —— 参照回数が多く寿命が長いデータが多いと，データのコピーを行う命令 (mov) や，ロード / ストア命令の増加に起因する性能の悪化を招く畏れがある．問題になりそうな場面としては，5.5 節で述べたループや関数呼び出しといった動的な構造が挙げられる．

予備評価

SPARC プロセッサで SPEC CPU95 の各プログラムを実行したトレース・データから，参照回数と寿命を求めた．なお寿命とは，定義からその最後の参照までの実行トレース上の命令数とする．

- コンパイラは gcc-2.7.2，最適化オプションは-O4.
- ゼロ・レジスタに対する定義 / 参照は数えない．
- スライディング・レジスタ・ウィンドウの操作に関しては，save 命令は最後の参照，restore 命令は定義とする．
- 参照されなかったデータは数えない．

結果を図 5.9 に示す．一本の線は，1 つのプログラムに対応している．各図中の上のグラフは分布を，下のグラフはその累積を表している．

参照回数が 3 回以上，寿命が 32 以上であるようなデータはすべてのデータのうちの，平均で 10%程度，悪い場合でも 20%程度である．これらのデータのために実行される mov 命令の増加も同程度であると予想される．このような目的で挿入される mov 命令はクリティカル・パス上にはないので，その程度の増加であれば，命令発行多重度の向上によって隠蔽できる可能性が高いと思われる．

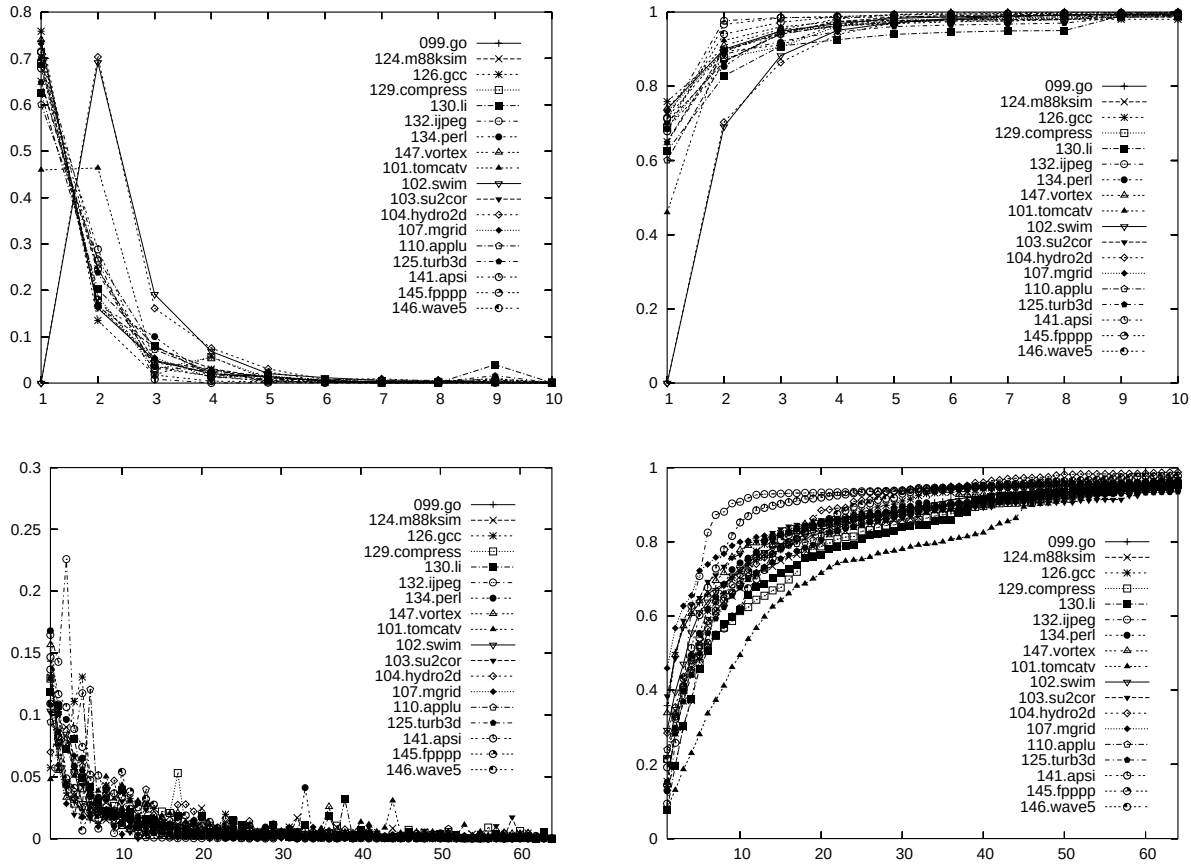


図 5.9: 参照回数（上）と寿命（下）の割合（左）とその累積（右）

コンパイラの評価

前章で述べたコンパイラによって，SPECint95 の 126.gcc 以外のプログラムをコンパイルし*，mov と nop の数を数えた．それ以外の有効な命令に対する mov と nop によるコードの増加率を計測した．表 5.2 に静的なものを，図 5.10 に動的なものを示す．後者は，インタープリタを用い，最初の 1G 命令に対して計測した．

図 5.10 中の内訳は以下のとおりである：

bb　基本ブロックの制約のために挿入される mov.

d32　距離が 32 以上ある場合に挿入される mov.

nd3　宛先が 2 個を越える場合に挿入される mov.

gcc　GCC が勝手に挿入してしまう mov.

cb　条件分岐の制約のために挿入される mov と nop.

triv　コンパイラの単純な不備によって挿入されている mov．今後簡単に削除することができる．

最適化がほとんど行われていないため，最大で 150%もの mov，nop が実行されている．性能について論じられる段階にはないが，d32 と nd3 は，他に比べて最適化との関係が薄い

Program	(%)	Program	(%)
099.go	93.5	132.ijpeg	67.1
124.m88ksim	42.0	134.perl	57.5
129.compress	72.3	147.vortex	55.5
130.li	40.9		

表 5.2: 静的なコードの増加率

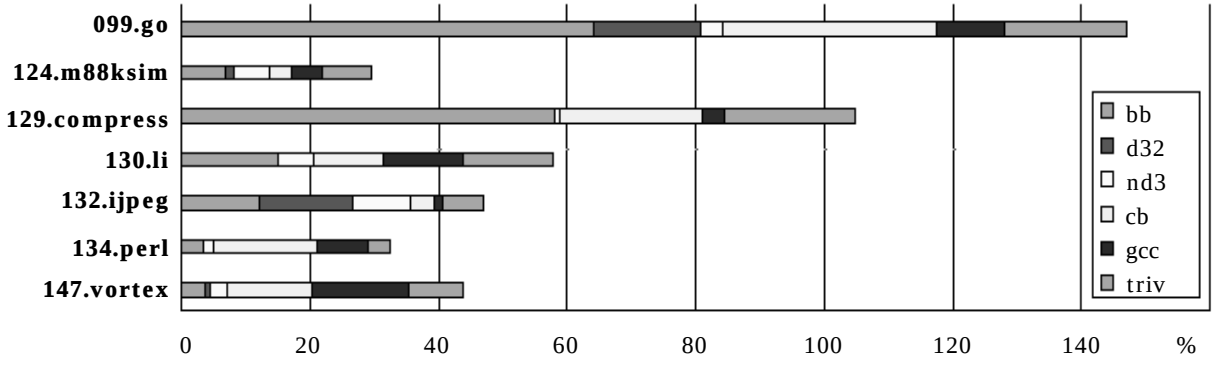


図 5.10: 動的なコードの増加率

* 126.gcc が使用する *alloca()* 関数が実装できていないため．

ため，一定の議論が可能である．d32 と nd3 は，*wakeup* ロジックの遅延と直接的なトレードオフの関係にあるため，重要である．

d32 は，5 つのプログラムでは非常に少ないが，090.go と 132.ijpeg では多い．090.go では，他の要因による無用な命令 —— 特に bb が大量に挿入された結果として，命令間の距離が実際以上に伸びてしまっていることが原因と推定される．また現在のコンパイラは，一時変数の数が 32 に近づいた時に，安易に `mov` を挿入して距離を長くするという悪循環に陥っている．132.ijpeg で d32 が多いのは，主にこの理由による．bb の削減と命令の並び替えによる距離の最小化によって，d32 は大幅に削減できる可能性がある．

nd3 は，全てのプログラムで 5〜10%程度である．nd3 は，コンパイラの最適化によっては除去することができない．しかし，nd3 によってクリティカル・パスが伸びてしまうのか，このデータからは判断できない．クリティカル・パス上にはなく命令フェッチのバンド幅が浪費されるだけであるのなら，*IW* が大きい場合には，この程度の増加は許容できる可能性が高い．

第6章　直接方式

前章で述べた間接方式では，デスティネーション行列，ソース行列の 2 種類の行列を逐次的にアクセスして，*req* から *rdy*，そして，*rdy* から *grant* を求めていた．その意味において間接方式は，従来の連想方式と変わりがない．一方，[6, 7, 8, 9, 10, 11, 12, 13] で提案した直接方式では，単一の行列を用い，直接 *grant* から *req* を求める．

また，[6, 7, 8, 9, 10, 11, 12, 13] では，行列の実効サイズを縮小しアクセスを高速化する手法も併せて提案している．

6.1　直接方式の原理

本節では，直接方式の原理を説明する．4.1 節と同様に，6.1.1 項で直接方式のデータ構造について述べた後，6.1.2 項で，それに対する *wakeup* 時のアクセスについて述べる．また，6.1.3 項では，5 章で述べた dualflow アーキテクチャとの関係について述べる．

6.1.1　直接方式のデータ構造

直接方式の命令ウィンドウ

図 6.1 のコードの 4 命令がディスパッチされた直後の直接方式の命令ウィンドウの様子を図 6.2 に示す．なお，図 6.1 のコードは，3 章，4 章で用いた図 3.1 (p. 50)，図 4.1 (p. 91) と同じものである．また，図 6.2 の命令ウィンドウは，同じく 3 章，4 章で用いた図 3.3 (p. 50)，図 4.2 (p. 91) と同じ状況のものである．直接方式の命令ウィンドウは，図 3.3 に示した連想方式や，図 4.2 に示した間接方式のそれと比べると，*rdyL*/*rdyR* フィールドがなく，代わりに ***irdy*** と呼ぶフィールドがある点が異なる．

irdy フィールドは，当該命令が発行可能であることを示す．連想方式の *rdyL*/*rdyR* を用いると，$irdy[i]$ $(i = 0, 1, \cdots, WS-1)$ は $rdyL[i] \cdot rdyR[i]$ と論理的に等価である．

プロデューサ行列

直接方式における依存行列の各行は，各命令のプロデューサを表すため，**プロデューサ行列**と呼び，***PROD*** で表す．命令 I_i $(i = 0, 1, \cdots, WS-1)$ が命令ウィンドウの i 番エントリに格納されるとき，その命令 I_i のプロデューサを表すベクトルが $PROD[i]$ に書き込まれる．

図 6.2 の命令ウィンドウと同時刻のプロデューサ行列の様子を図 6.3 の 1. に示す．プロデュー

label	*opcode*	*prD*		*prL*	*prR*	*immed*
I_x :	op_x	%1	=	%0,		0
I_l :	op_l	%2	=	%1,		1
I_r :	op_r	%3	=	%0,		2
I_c :	op_c	%4	=	%2,	%3	

図 6.1: レジスタ・リネーミングされたコード

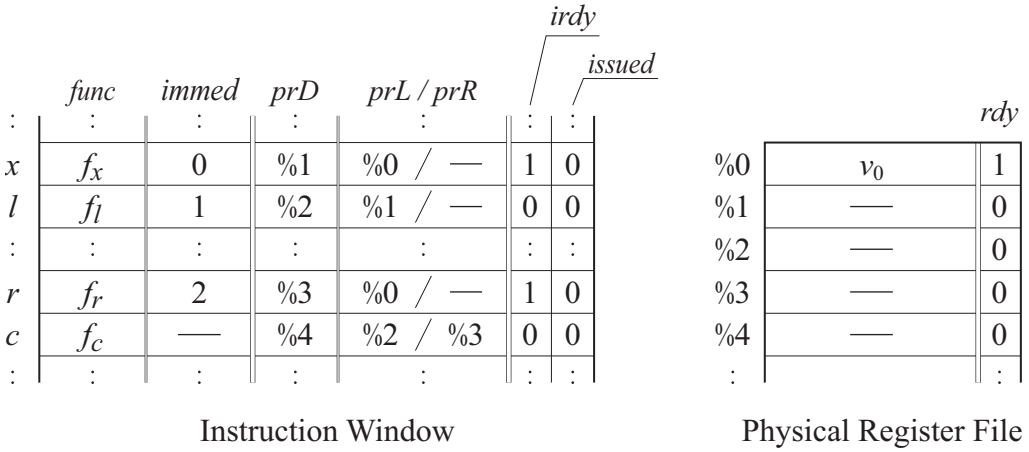


図 6.2: 直接方式の命令ウィンドウ

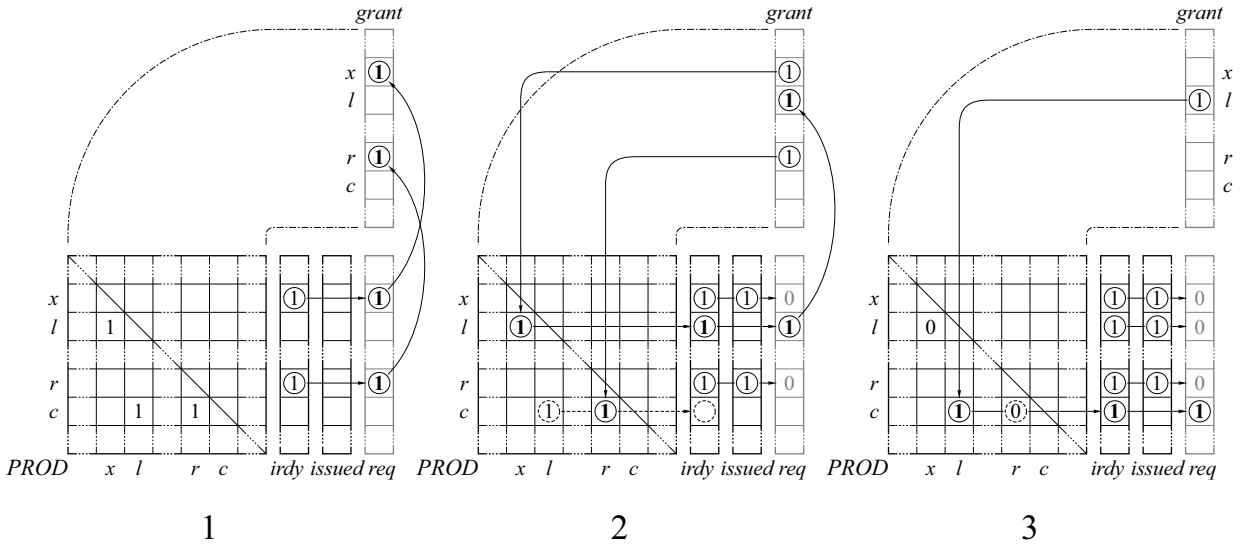


図 6.3: 直接方式の概念図

サ行列は，5 章で述べた dualflow アーキテクチャのコンシューマ行列と同様に，命令間の依存関係を表す．したがって，間接方式のデスティネーション行列，ソース行列がそれぞれ WS 行 × NR 列であったのに対して，直接方式のプロデューサ行列は WS 行 × WS 列の正方行列である．また，dualflow アーキテクチャのコンシューマ行列と同様に，対角要素は使用しない．

ただし，dualflow アーキテクチャのコンシューマ行列の各行が当該命令のコンシューマを表していたのに対して，直接方式のプロデューサ行列の各行は当該命令のプロデューサを表す．すなわち，命令ウィンドウの p 行 ($p = 0, 1, \cdots, WS-1$) の命令 I_p の実行結果を，同じく c 行 ($c = 0, 1, \cdots, WS-1$) の命令 I_c が使用する場合，dualflow アーキテクチャのコンシューマ行列 $CONS$ では $CONS[p][c]$ を 1 としたのに対して，直接方式のプロデューサ行列 $PROD$ では $PROD[c][p]$ を 1 とする．すなわち，直接方式のプロデューサ行列は dualflow アーキテクチャのコンシューマ行列と転置 (transposition) の関係にある．また，式 5.1 より：

$$PROD^T = CONS = DST \cdot SRC^T \ . \tag{6.1}$$

なお，dualflow アーキテクチャとの関係については，6.1.3 項で詳しく述べる．

6.1.2 プロデューサ行列アクセス

図 6.3 に，図 6.1 に示したコードが実行される際の，直接方式の動作を示す．同図 6.3 中，薄く示した格子は組合わせ回路的なシグナルであり，濃く示した格子はメモリの出力である．直接方式の *wakeup* 処理は，以下のように進む：

1. $irdy[x]$，$irdy[r]$ は最初からセットされており，$req[x]$，$req[r]$ がアサートされる．

   ここで，命令 I_x と I_r の発行が同時に許諾されたとしよう．

2. すると，次のサイクルに，$grant[x]$ と $grant[r]$ がそれぞれアサートされる．

   x 列を見ると，l 行が 1 である．したがって，命令 I_l は I_x に依存しており，I_x の発行に伴って実行可能になると分かる．そこで，$irdy[l]$ がセットされ，$req[l]$ がアサートされる．

   r 列を見ると，c 行が 1 である．しかし，I_c は実行可能に**ならない**．c 行では，r 列の他に l 列もセットされており，命令 I_c は，I_r の他に I_l にも依存していることが分かる．命令 I_l の発行はまだ許諾されていないので，I_c はこのサイクルでは実行可能にならない．

   その後，dualflow アーキテクチャと同様に，x 列と r 列をリセットしておく（5.3 節）．この場合，実際には c 行要素だけが，1 から 0 に変わる．

   ここでは，命令 I_l の発行が許諾されたとしよう．

3. すると，$grant[l]$ がアサートされる．l 列を見ると，やはり，c 行が 1 である．r 列要素はサイクル 2 においてリセットされたので，c 行では l 列要素のみが 1 であり，命令 I_c は命令 I_l にのみ依存しているかのように見える．そのため，今度こそ I_c は実行可能になり，$req[c]$ がアサートされる．

4.1 節の rdy_{set} の場合と同様に，ベクトル $\boldsymbol{irdy_{set}}$ を用い $irdy = irdy + irdy_{set}$ とする．

Wakeup フェーズにおける直接方式のプロデューサ行列 *PROD* へのアクセスは，以下のようにまとめられる：

プロデューサ行列　i 行において，$PROD[i][j] = 1$ であるすべての j に対して，$grant[j] = 1$ であれば，$irdy_{set}[i]$ がアサートされる．ただし，$i, j = 0, 1, \cdots, WS-1$ である．

4.1 節(p. 89) で述べた，以下の間接方式のソース行列へのアクセスと比較されたい：

ソース行列　i 行において，$SRC[i][j] = 1$ であるすべての j に対して，$rdy[j] = 1$ であれば，$req[i]$ がアサートされる．ただし，$i = 0, 1, \cdots, WS-1$，$j = 0, 1, \cdots, NR-1$ である．

これから分かるように，直接方式におけるプロデューサ行列 *PROD* へのアクセスは，間接方式におけるソース行列 *SRC* へのアクセスと相似である．基本的には，ソース行列アクセスにおける *req*，*SRC*，および，*rdy* を，$irdy_{set}$，*PROD*，および，$grant^T$ に置換すれば，プロデューサ行列アクセスを得ることができる．

プロデューサ行列 *PROD* へのアクセスは，以下の式で表すことができる．やはり，4.1 節で述べた，間接方式のソース行列へのアクセスの式 4.2 と比較されたい：

$$\begin{aligned} req[i] &= \prod_j (SRC[i][j] \Rightarrow rdy[j]) \\ &= \prod_j \left(\overline{SRC[i][j]} + rdy[j]\right) \\ &= \overline{\sum_j SRC[i][j] \cdot \overline{rdy[j]}} \qquad (4.2) \\ irdy_{set}[i] &= \prod_j (PROD[i][j] \Rightarrow grant[j]) \\ &= \prod_j \left(\overline{PROD[i][j]} + grant[j]\right) \\ &= \overline{\sum_j PROD[i][j] \cdot \overline{grant[j]}}\,. \qquad (6.2) \end{aligned}$$

式 6.2 は，行列 — ベクトル積を用いると以下のように書ける．同様に，4.1 節で述べた，間接方式のソース行列へのアクセスの式 4.4 と比較されたい：

$$\begin{aligned} req &= \overline{SRC \cdot \overline{rdy^T}} \qquad (4.4) \\ irdy_{set} &= \overline{PROD \cdot \overline{grant}}\,. \qquad (6.3) \end{aligned}$$

6.1.3　Dualflow アーキテクチャとの関係

前述したように，直接方式のプロデューサ行列 *PROD* は，dualflow アーキテクチャのコンシューマ行列 *CONS* と転置の関係にある．Dualflow アーキテクチャの依存行列の各行が当該命令のコンシューマを表していたのに対して，直接方式の依存行列の各行は当該命令のプロデューサを表す．すなわち，命令ウィンドウの p 行の命令 I_p の実行結果を c 行の命令 I_c が使

用する場合，dualflow アーキテクチャのコンシューマ行列では $CONS[p][c]$ を 1 としたのに対して，直接方式のプロデューサ行列では $PROD[c][p]$ を 1 とする．

このことは，dualflow アーキテクチャにおけるデータの授受がデータ駆動的であるのに対して，直接方式の適用の対象である通常のスーパースカラ・プロセッサの命令セット・アーキテクチャが制御駆動型であることに起因する．Dualflow アーキテクチャでは，各命令の中にそのコンシューマが明示的に埋め込まれているため，ディスパッチ時にコンシューマ行列の各行を自然に求めることができる．一方，通常の制御駆動型の命令セット・アーキテクチャを持つスーパースカラ・プロセッサでは，命令のディスパッチ時にその命令のコンシューマをすべて求めることは原理的に不可能である．しかし，命令のソース・オペランド・データのプロデューサであれば，容易に求めることができる．次節では，その具体的な方法について詳しく述べる．

6.2 直接方式のロジック

前節で述べたように，直接方式のプロデューサ行列 *PROD* に対するアクセスは，間接方式のソース行列に対するアクセスと似ている．プロデューサ行列は，ソース行列のそれとほぼ同じロジックによって実装される．

ただし，dualflow アーキテクチャのコンシューマ行列と同様に，直接方式の *PROD* に対する入力 *grant* は，組合わせ回路的出力であるため，それに対処するための機能が必要となる．コンシューマ行列では，行リセット機能を追加した（5.3 節）．コンシューマ行列とプロデューサ行列は転置の関係にあるから，列リセットの機能を追加することになる．

なおこの方法は，行列上に保存された依存関係を発行時に破壊するため，投機失敗時の状態回復の方法が制限される．しかし，このことに依存しない効率のよい回復法も提案されている [41].

6.3 依存行列アクセスの高速化

本節では，1. 非集中化，2. 2 階層化，および，3. 狭幅化 の，依存行列アクセスの 3 つの高速化手法について述べる．以下，6.3.1，6.3.2，および，6.3.3 項で，それぞれの手法につ

ソース・オペランドごとの依存行列

当初，文献 [6, 7, 8, 9, 10, 11] で示した直接方式では，ソース・オペランドごとに 1 つの行列を用いるというものであった．その後，間接方式の影響などから，文献 [12] で，それらソース・オペランドごとの依存行列を単一の依存行列に単一化する方法を提案するに至った．

いて順を追って説明する．

これらの手法のうち，1. 非集中化 は，連想方式でも一般的に用いられている手法を依存行列に応用したものである．2. 2 階層化，および，3. 狭幅化 は，直接方式の高速化手法として新たに考案したものである．また，1. 非集中化 と 2. 2 階層化 は，間接方式にも適用可能である．その方法については 6.4 節で述べる．

6.3.1　依存行列の非集中化

3.7 でも述べたように，実際のマイクロプロセッサでは，命令ウィンドウは，集中化された単一のロジックとして実装されるのではなく，命令の系統ごとに非集中化されることが多い．最近の out-of-order スーパースカラ・プロセッサの多くは，整数 (**INT**)，LS (**LS**)，および，浮動小数点 (**FP**) の各命令の系統ごとに，命令ウィンドウを非集中化している．DEC* Alpha 21264 プロセッサは INT，LS 用と FP 用の，HP PA-8000 プロセッサは INT，FP 用と LS 用の，それぞれ 2 つの命令ウィンドウを持つ．MIPS R10000 プロセッサは INT 用，LS 用，FP 用の 3 つの命令ウィンドウを持つ [62, 22, 63, 23, 57, 33, 48]．このような命令の系統ごとの命令ウィンドウの非集中化は，ほとんど必須と言ってもよい．ごくわずかな IPC のペナルティを代償に，ロジックの実効サイズ (effective size) を大幅に縮小することができるからである．

MIPS R10000 プロセッサと同様の非集中化を施した場合の依存行列の様子を図 6.4 **(b)** に示す．なお，SPARC など，INT–FP レジスタ間の転送命令が無い ISA では，同図中破線で記した部分行列を省略できる．R10000 プロセッサでは，INT，LS，FP の各サブウィンドウの発行幅とサイズはそれぞれ，$\boldsymbol{IW'} = IW/3 = 2$，$\boldsymbol{WS'} = WS/3 = 16$ である．

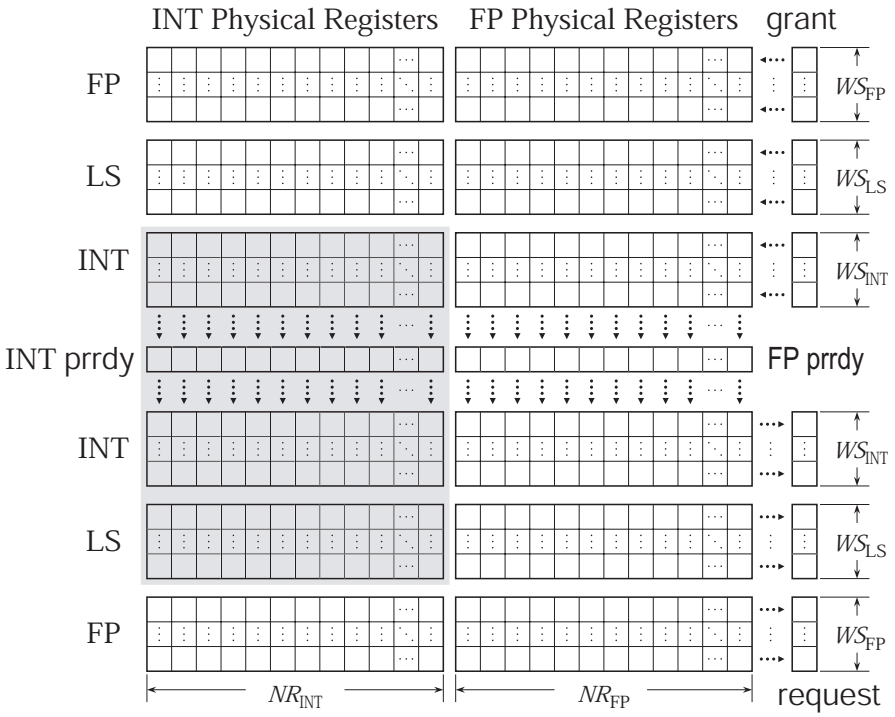
非集中化が行列に与える第一の効果は，ライト・ポート数の削減である．行列に対する書き込みは，ディスパッチ時に，命令ウィンドウ・エントリに対応する行に対して行われる．非集中化を行わない場合，ディスパッチされる命令は，命令の系統によらず，どの命令ウィンドウ・エントリにも書き込まれ得る．そのため，行列の各行は，IW 本のライト・ポートが必要となる．非集中化した場合には，例えば FP 命令をディスパッチするときに書き込まれる行は，図 6.4 (b) では下から WS' 行に制限することができる．したがってこの部分のセルのライト・ポート数は，IW' 本でよく，IW から IW' へと 1/3 に削減することができる．INT，LS についても同様である．

また，R10000 プロセッサのように INT のみのサブウィンドウを持つ場合，次項で述べる多階層化の適用が可能になる．

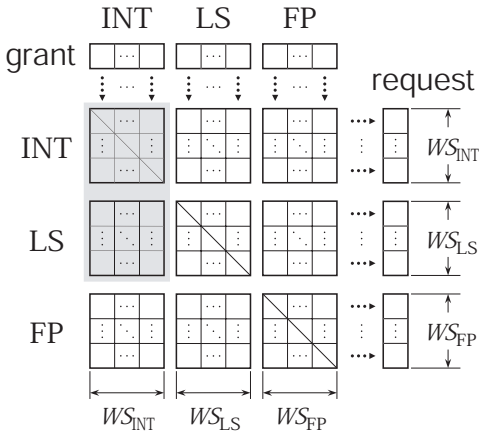
6.3.2　依存行列の 2 階層化

MIPS R10000 プロセッサの構成では，レイテンシが 1 であるパスは，INT から INT，INT から LS の 2 つである．図 6.4 (b) では，影を付けた部分がこれに相当する．この部分を取

* 発表当時.



(a) Indirect Scheme



(b) Direct Scheme

: L-1 Matrix

図 6.4: 行列の非集中化

りだし，これを **L-1 行列**，残りを **L-2 行列**と呼ぶ.

L-1 行列 アクセスは *select* と合わせて 1 サイクル以内に行う必要があるが，L-2 行列 アクセスはパスのレイテンシに合わせて適当にパイプライン化してよい．図 3.4 (L-2 行列) は，ちょうど，レイテンシが 2 サイクルの場合にあたる．同図では，L-2 行列 アクセスには 1.5 サイクル，すなわち，L-1 行列 の 3 倍の時間をかけており，L-2 行列 がクリティカルになる可能性は極めて低い.

一方 L-1 行列 は，元の 依存行列から比べると格段に少量化され，その分だけ遅延も短縮される．更に，L-1 行列 に対しては次節で述べる狭幅化を適用することができる.

6.3.3　L-1 行列の狭幅化

依存する命令間の距離は短い場合が多い．この性質を利用して，L-1 行列を更に縮小することができる．各行において，先行する w 命令に対するビットだけを残し，それ以外を L-2 行列に移すのである．*Wakeup* 処理は，命令間の距離が w 以下の場合には L-1 行列によって，そうでない場合には L-2 行列によって行われる．前者を L-1 行列**ヒット**，後者を**ミス**と呼ぶ．ミスの場合には，1 サイクルのペナルティが生じる.

図 6.5 に，L-1 行列の縮小の様子を示す．左は元々の，右が縮小後の L-1 行列である．L-2 行列に移されるセルは，左図中で薄く示してある．メモリのワードライン，ビットラインのうちでは，ライト・ポートのビットラインが $2 \cdot IW'$ 本と最も本数が多い．したがって図 6.5 では，それらが直線になるように，L-1 行列に残されるセルを矩形領域に集めている．w は，L-1 行列を構成するアレイの幅にあたる．そのため，この技術を **L-1 行列の狭幅化**と呼ぶ.

図 6.5 右から明らかなように，狭幅化された L-1 行列 のワードライン，ビットラインは，それぞれ w 個のセルにしか接続されていない．したがって *wakeup* ロジックの遅延は，WS と

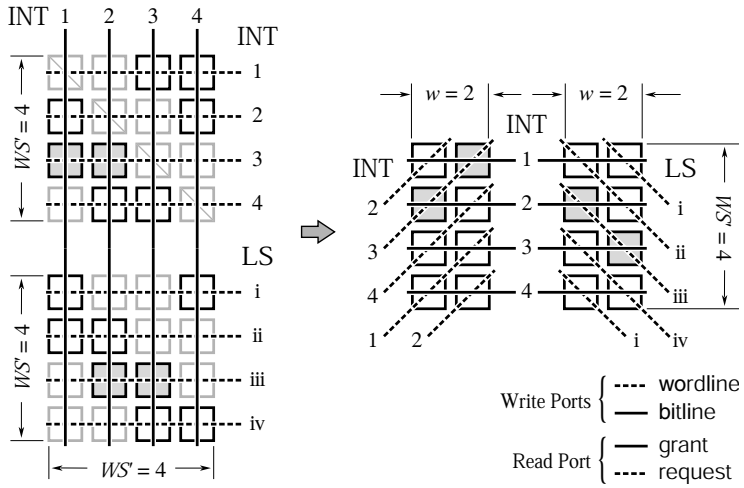


図 6.5: L-1 行列の狭幅化

は独立に，w によって決まる．

Pace-Keeping

なお，狭幅化を非集中化と同時に施した場合，各サブウィンドウ間で命令ウィンドウ・エントリの消費の歩調を揃える **pace-keeping** と呼ぶ操作が必要となり，命令ウィンドウ・エントリの使用効率が若干悪化する．

6.4 間接方式に対する高速化手法の適用

間接方式のデスティネーション行列，ソース行列に対しても，直接方式と同様の分散化と多階層化を適用ことができる．図 6.4 (a) に，その様子を示す．同図 (a) は，同図 (b) と同様，R10000 の構成に対するものである．

物理レジスタは，INT，FP のそれぞれに分けられ，それぞれの個数は，$\boldsymbol{NR}_{\mathbf{INT}} = 2(WS_{\mathrm{INT}} + WS_{\mathrm{LS}})$，$\boldsymbol{NR}_{\mathbf{FP}} = 2(WS_{\mathrm{FP}} + WS_{\mathrm{LS}})$，すなわち，$NR_{\mathrm{INT}} = NR_{\mathrm{FP}} \equiv \boldsymbol{NR'} = 4 \cdot WS'$ とすることが多い [62, 22, 63, 23, 57, 33, 48].

しかし，L-1 行列 の狭幅化は，間接方式には適用することができない．直接方式と同様に，割り当てられる命令ウィンドウ・エントリと物理レジスタとの関係を制限して行列を縮小することは可能だが，直接方式における命令間の近さのような有効なヒューリスティクスが，命令ウィンドウ・エントリと物理レジスタの間には存在しないからである．

6.5 IPC の評価

SimpleScalar ツールセット [45, 46] (ver. 2.0) に対して，6.3.3 節で述べた L-1 行列の縮小を実装し，SPEC ベンチマーク [47] を用いて L-1 行列の幅 w に対する IPC の変化を測定した．表 6.1 に示す CINT95 の 8 つのプログラムを実行した．

プログラム	入力セット	実行命令数
099.go	9 9	132M
124.m88ksim	dcrand.big	120M
126.gcc	genrecog.i	122M
129.compress	10000 q 2131	35M
130.li	train.lsp	183M
132.ijpeg	vigo.ppm -GO	26M
134.perl	primes.in	10M
147.vortex	persons.250	157M

表 6.1: SPEC CINT95 ベンチマーク・プログラム

キャッシュ		容量	ライン サイズ	連想度	レイテンシ (cycles)
1次	命令	8K命令 *	8命令 *	2	1
	データ	32KB	32B	2	1
2 次	（統合）	1MB	64B	2	6
メモリ		—	—	—	18

*: SimpleScalar Toolset では 1 命令が 8B.

表 6.2: キャッシュ，メモリのパラメタ

ベース・モデルとしては，MIPS R10000 プロセッサ [23] を用いた．6.3.1 項で述べたように，R10000 プロセッサは，INT，LS，FP のそれぞれにサブ命令ウィンドウを持つ．各サブウィンドウのパラメタは，それぞれ $(IW', IW, WS', TW) = (2, 4, 16, 6)$ である．表 6.2 に，キャッシュ，メモリのパラメタをまとめる．命令 / データ分離 1 次，および，統合 2 次キャッシュの，容量，ライン・サイズ，レイテンシは，それぞれ，32KB，32B，1 サイクル，および，1MB，64B，6 サイクルである．2 次キャッシュ・ミス時には，最初のワードに 18 サイクル，後続ワードには 2 サイクル / ワードが必要である．分岐予測には，同ツールセットに用意されている 2b 飽和型カウンタによるもの (bimod) を用いた．このモデルを **R10K×1** と呼ぶ．更に，命令ウィンドウのすべてのパラメタを 2 倍，すなわち，$(IW', IW, WS', TW) = (4, 8, 32, 7)$ としたモデル **R10K×2** も合わせて測定した．命令ウィンドウのパラメタに関わる，フェッチ幅，実行ユニット数などもそれぞれ 2 倍にしてある．また，w の上限を求めるため，すべての資源を無限大にし，キャッシュ，分岐予測を完全としたモデル **R10K×∞** も測定した．

L-1 行列の幅に対する IPC の比率を図 6.6 に示す．R10K×1 では $w \leq WS1 = 16$，R10K×2 では $w \leq WS' = 32$ であることに注意されたい．なお，$w = WS'$ で pace-keeping を行わない場合には，IPC の低下は原理的に起こらず，その場合の IPC を 100% としている．グラフでは，$w = WS'$ のとき，本来不要な pace-keeping を行った場合の IPC を表示している．そのときの IPC の低下は，pace-keeping の影響を示している．

R10K×1，R10K×2 の結果からは，$w \geq WS'/4$（R10K×1 では 4，R10K×2 では 8）では $w = WS'$ の場合からほとんど IPC は低下せず，1〜2% 程度以下に抑えられることが分かる．この部分における IPC の低下は，ほとんど pace-keeping の影響だけによると推定できる．

また，$w = 0$ のときの IPC はちょうど，3.2 節で述べた，*wakeup* と *select* フェーズに 2 サイクルをかけた場合に相当する．同節で述べたように，その場合の IPC の低下は最大 15% 程度にもなる．

R10K×∞ の結果からは，w は 16 あれば IPC の大きな落ち込みはなく，64 あれば十分であることが分かる．

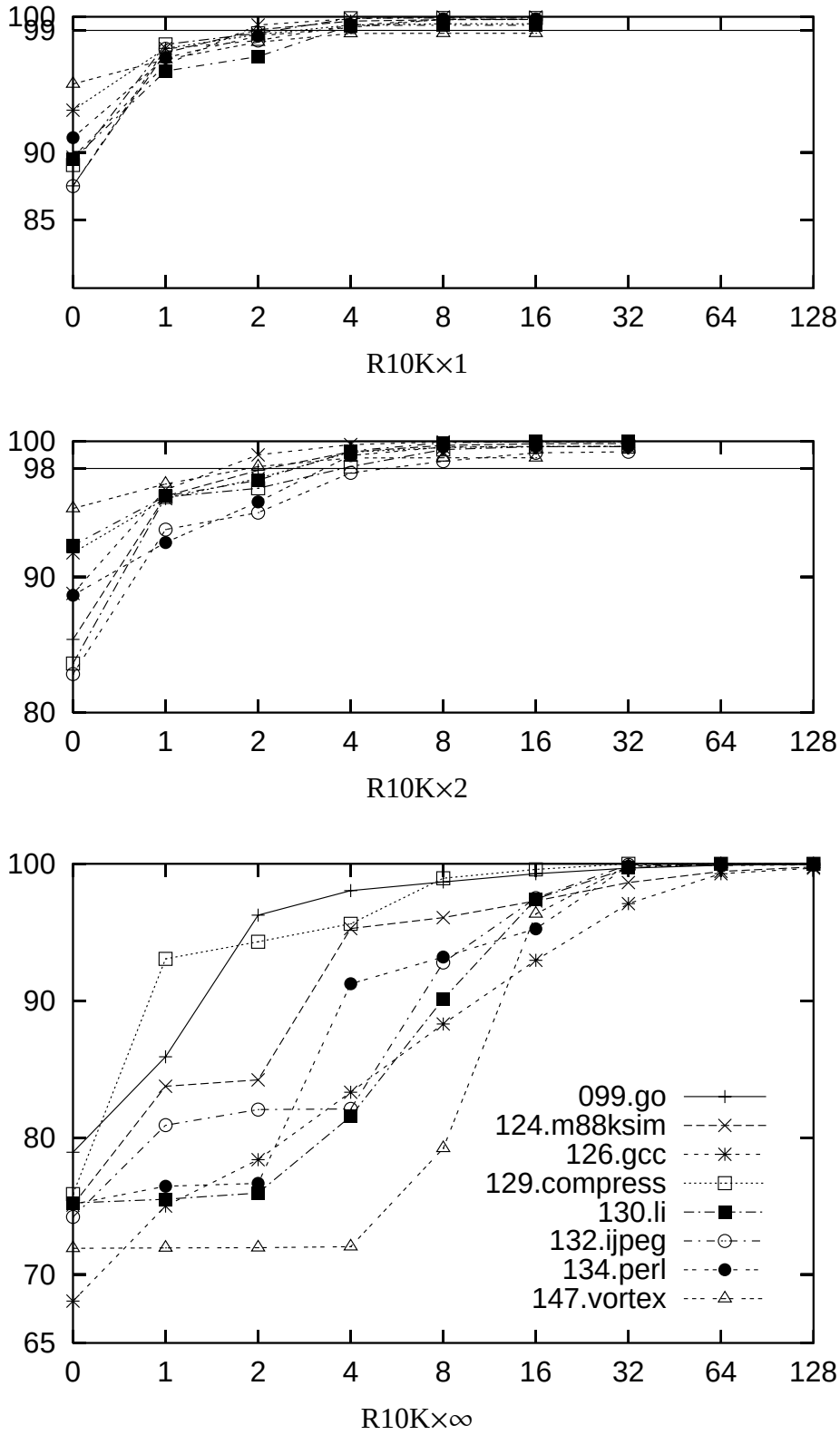


図 6.6: L-1 行列の幅 w に対する IPC の変化

第7章　回路の評価

本章では，実在するCMOSプロセスのデザイン・ルールを用いて，従来方式と提案方式の主要な回路の面積と遅延を評価する．

7.1　評価方法

評価には，富士通株式会社から提供されたCS80A CMOSプロセスを用いた．表7.1に，CS80Aの主要な諸元を示す．

CS80Aのデザイン・ルールに基づいて，整数命令キューにおける従来方式と提案方式の主要な回路のレイアウトをスクラッチから設計した．なお，CS80Aの配線層は6層あるが，今回は下3層のみを用いている．得られたレイアウトから回路面積を求めた．また，プロセス・パラメタに基づいてRCデータを抽出し，Hspiceシミュレーションによって回路遅延を求めた．

以下，7.2節では設計した回路の説明を行い，7.3節と7.4節で評価した回路の面積と遅延をそれぞれ示す．

7.2　Out-of-Order命令スケジューリングのロジック

以下，順に，連想方式のデスティネーションRAMとソースCAM，次いで間接方式，および，直接方式の依存行列，最後に*select*ロジックの説明を行う．

各回路の評価では，前章のIPCの評価で用いたのと同じモデル×1と×2に加えて，MIPS R10000プロセッサの半分の資源を持つモデル$\times\frac{1}{2}$を用いた．表7.2に，それぞれのパラメタをまとめる．

パラメタ名	値
ゲート長	180nm
基盤	バルク
ゲート絶縁膜	SiO_2
層間絶縁膜	SiO_2
メタル配線	6層アルミニウム

表7.1: CS80A CMOSプロセスの諸元

パラメタ	記号	モデル		
		$\times\frac{1}{2}$	×1	×2
命令フェッチ幅	FW	2	4	8
サブウィンドウ ディスパッチ幅	DW'	1	2	4
サブウィンドウ 発行幅	IW'	1	2	4
サブウィンドウ サイズ	WS'	8	16	32
物理レジスタ数（INT，FP それぞれ）	NR	32	64	128
タグ幅 ($\log_2 NR$)	TW	5	6	7
狭幅化後の行列の幅 ($1 \le w \le WS'$)	w	1〜8	1〜16	1〜32

表 7.2: 各モデルのパラメタ

7.2.1　連想方式のデスティネーション RAM

連想方式におけるデスティネーション RAM は，命令ウィンドウの一部として *prD* を格納する．フェッチ，デコードされた命令は，*rename* フェーズにおいて，*prD* が割り当てられる（3.4 節）．割り当てられた *prD* は，命令がディスパッチされるときにデスティネーション RAM に書き込まれ，*wakeup* フェーズにおいて読み出される．読み出された *prD* は，同じく *wakeup* フェーズにおいて次節で述べるソース CAM 部に入力される他，ライトバック・ステージまで遅延された上で物理レジスタ・ファイルの書き込みアドレスとして使用される（3.6 節）．

構成

連想方式におけるデスティネーション RAM は，基本的には通常のマルチポート RAM である．ただし，3.6.2 節で述べたように，リード・ポートには行デコーダがなく，*select* ロジックからの *grant* 信号が直接的にワードラインに接続される．

命令ウィンドウの一部として *prD* を格納するため，デスティネーション RAM の語構成は，INT キューにおいては，WS' word × TW bit となる．

prD は，*dispatch* フェーズで書き込まれ，*wakeup* フェーズで読み出されるから，ライト・ポート数は DW' 本，リード・ポート数は IW' 本必要になる．

フロア・プラン

レイアウトのフロア・プランは，図 3.17 (p. 81) 上部の回路図とほぼ対応する．通常の RAM では，ワードラインとビットラインの配線長の和を最小化するため，セルは正方形に近い形状とする（2.3 節）．しかしデスティネーション RAM では，以下の理由により，セルは行（横）方向に長い矩形とした：

1. ビット数よりワード数の方が大きい
2. 次節で述べるように，ソース CAM とセルの幅を合わせる必要がある．

ビットラインはメタル 1 層，ワードラインと電源バスはメタル 2 層に配した．

セルの高さは，ワードラインが通過する配線エリアによって決まる．前述したように，セ

ル面積の半分以上はビットライン・ドライバによって占められている．セルの幅は，ビットライン・ドライバのゲート幅によって決まっている．そのため，ワードラインとビットラインの遅延の間のトレードオフとなる．

7.2.2 連想方式のソース CAM

連想方式のソース CAM は，命令ウィンドウの一部として *prL*/*prR* を格納し，連想検索により *rdyL*/*rdyR* を更新する．フェッチ，デコードされた命令は，*rename* フェーズにおいて，*prL*/*prR* が割り当てられる（3.4 節）．割り当てられた *prL*/*prR* は，命令がディスパッチされるときにソース CAM に書き込まれる．*Wakeup* フェーズでは，前節で述べたデスティネーション RAM から読み出された *prD* がサーチ・ポートに入力され，格納されている *prL*/*prR* に対する連想検索が行われる．一致するエントリがあれば，対応する *rdyL*/*rdyR* がセットされる．

構成

CAM のキー部には *prL*/*prR* が，バリュー部には *rdyL*/*rdyR* が，それぞれ格納される．*prL* — *rdyL* ペアと，*prR* — *rdyR* ペアは，それぞれ独立であるので，それぞれ別のバンクとすることができる．図 3.17 (p. 81) 下の回路図では，*prL* — *rdyL* のバンクだけを示してある．

命令ウィンドウの一部として格納された *prL*/*prR* は，物理レジスタ・ファイルの読み出しアドレスとしても使用されるため，*issue* フェーズで命令ウィンドウの他のフィールドと共に読み出す必要がある．しかし，*wakeup* ロジックがクリティカルであるとの仮定のもとに議論しているので，ソース CAM のキー部には *prL*/*prR* を読み出すリード・ポートは用意しないものとする．別途用意した RAM に *prL*/*prR* の複製を格納し，*issue* フェーズではそこから読み出す．

INT キューでは，キー部として，語構成がそれぞれ WS' word × TW bit となるものが 2バンク必要となる．

セルの下半分は比較器のアレイである．図では，$IW' = 2$ としている．そのため，ライト・ポート数は 2 でよいが，比較器は 4 組必要である（3.7 節）．図中，$prD\_o_{3:4}$ は，LS キューから送られてきた *prD* を示している．

一致比較器

連想方式のソース CAM では，連想アクセスに際して，入力された *IW* 個の *prD* のうちのいずれか 1つが，*prL*/*prR* に一致すれば *rdyL*/*rdyR* をセットする．したがって，*IW* 本のマッチラインの出力の OR を求める必要がある．

この *IW* 入力の OR は，しかし，入力であるマッチラインが normally high であるため，原則ドミノ論理では実装することができない（2.1 節）．

そのためこの OR は，通常スタティック・ゲートで実現される*ため，その遅延は無視できないものとなる．

∗ *p*MOS ドミノ論理で実現するという解もある．

フロア・プラン

フロア・プランは，やはり図 3.17 下の回路図とほぼ対応する．

ライト・ポートのビットラインをメタル 1 層に，ライト・ポートのワードラインとサーチ・ポートのマッチライン，電源バスをメタル 2 層に，サーチ・ポートのサーチラインをメタル 3 層に配した．

セル・サイズは，2，3 層の配線領域によって決まる．

なお，前節で述べたように，*prD* 用のデスティネーション RAM のセルとできるだけ横幅が一致するようにレイアウトするとよい．

7.2.3　間接/直接方式の依存行列

構成

依存行列のセルの構成は，図 4.5 (p. 96) に示した．依存行列 セルは，7.2.1 節で述べた連想方式のデスティネーション RAM のセルと，基本的に同じである．ただし，以下の点が異なる：

1. リード・ポート数は，*IW′* にかかわらず，1 でよい．

   7.2.1 節で述べたように，今回用いた構成では，ビットライン・ドライバがセル面積の半分以上を占める．したがって，リード・ポート数の少なさは，セル面積，ひいては，読み出しの遅延に大きな影響を与える．

2. 2 レールの出力が得られない．

   6.2 節で述べたように，依存行列では同時に複数のワードラインが ON になるため，原理的に 2 レールの出力が得られない．そのため，2.3.3 節で述べたような，シングルエンドのセンスアンプを用いる必要がある．

更に，直接方式の依存行列では，以下の違いがある：

1. ビットラインは横方向に，ワードラインは斜め方向に引かれる．

   6.3.3 節で述べたように，狭幅化の結果，ビットラインは横方向に，ワードラインは斜め方向に引かれる．

   また，ワードライン長，ビットライン長は，*WS′*，および，*TW* とは独立に，依存行列の幅 *w* によって決まる．

2. 列リセットの機能が必要になる．

   6.2 節で述べたように，直接方式の依存行列では，列を破壊的に読み出すため，列リセットの機能が必要になる．

フロア・プラン

斜めになるワードライン長を最小化するため，外形が正方形に近くなるようにレイアウトした．CS80A プロセスではメタルを斜めに配線することは許されていないので，ワードライン長は (依存行列の幅) × ((セルの高さ) + (セルの幅)) となる*.

リード・ポート数が 1 でよく，長い配線の数が少ないため，セル・サイズはトランジスタによって決まった.

7.2.4 *Select* ロジック

Select ロジックは，最大 WS' 個の *req* を調停し，そのうちの最大 IW' 個に *grant* を与える．

3.5 節で挙げたカスケード方式は，同節で説明したように，1 命令選択回路を IW' 個カスケード接続したもので，IW' が 2 程度であればよいが，3〜4 となるとその遅延が問題となる．

そこで今回は，**prefix-sum 方式**を評価した．この方式は，自分より優先順位の高い発行要求の数，すなわち，発行要求数の prefix sum を求めることにより，自らが選らばれるかどうか，また，選ばれるならば何番目に選ばれるかを判断する．自分より優先順位の高い *req* が n ($n < IW'$) 個あれば，自らは $n+1$ 番目に選択されると判断できる．

なお今回は，簡単のため，優先順位は固定としている．文献 [65] には，命令ウィンドウ・エントリをサイクリックに使用する方式に向けて，優先順位を同様にサイクリックに変化させる方法が示されている．

Prefix-Sum Thicket

Prefix-sum は，prefix-sum thicket*によって計算できる．図 7.1 に，Prefix-sum thicekt を示す．図中 ⊕ は 2 入力加算器を，◯ は単なるバッファを表す．発行要求があれば 1 を，そうでなければ 0 を r[n] に入力すると，s[n] に $\sum_{k=1}^{n}$ r[k] が出力される．

加算器

ここで用いられる加算器では，IW' 以上の値を正確に計算する必要はない．このことを利用した最適化が可能である．今回は，変則的な one-hot エンコーディングによって，回路の

リクエスト数	s0	s1	s2	s3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4 以上	0	0	0	0

表 7.3: 発行要求数のエンコーディング

* 最近では，斜め配線を許すプロセスがある．
* やぶ, 茂み, 雑木林.

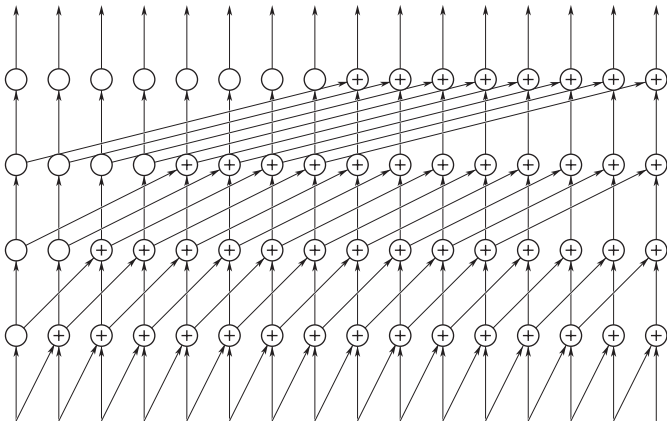


図 7.1: Prefix-Sum Thicket

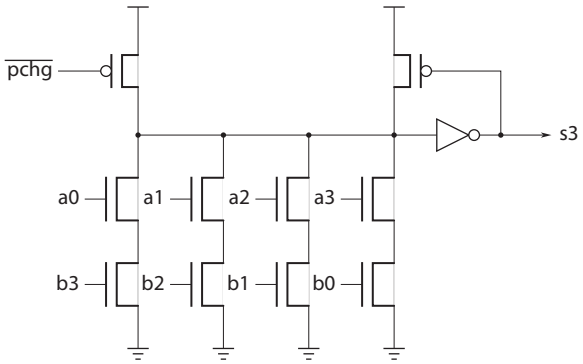


図 7.2: s3 を求める論理回路

簡単化を図った．表 7.3 に，$IW' = 4$ の場合のリクエスト数のコーディングの方法を示す．このエンコーディングを用いると，リクエスト数の加算は以下の論理式で表される．ただし，a0〜a3，b0〜b3，および，s0〜s3 は，それぞれ，2 つの入力，および，出力を表す:

$$
\begin{aligned}
s0 &= a0 \cdot b0 \\
s1 &= a0 \cdot b1 + a1 \cdot b0 \\
s2 &= a0 \cdot b2 + a1 \cdot b1 + a2 \cdot b0 \\
s3 &= a0 \cdot b3 + a1 \cdot b2 + a2 \cdot b1 + a3 \cdot b0
\end{aligned}
$$

各式は，それぞれ，1 段のダイナミック・ゲートで実現することができる．図 7.2 に，s3 を求めるダイナミック・ゲートを示す．

したがって全体の遅延は，たかだか $\log_2 WS'$ 段のダイナミック・ゲートの遅延で与えられる．

方式	ロジック	ビット数	ワード数	ライトポート数	リードポート数	サーチポート数
連想	デスティネーション RAM	TW	WS'	DW'	IW'	—
	ソース CAM	↑	$2 \cdot WS'$	↑	—	$2 \cdot IW'$
間接	デスティネーション行列	NR	WS'	↑	1	—
	ソース行列	↑	↑	↑	↑	—
直接	—	WS'	↑	↑	↑	—

表 7.4: 各ロジックのパラメタ

	x1/2	x1	x2
連想方式	960 (24×40)	1360 (34×40)	4320 (54×80)
間接方式	360 (20×18)	400 (20×20)	572 (26×22)
直接方式	400 (20×20)	400 (20×20)	572 (26×22)

表 7.5: セル面積 (F^2) (縦 (F) × 横 (F))

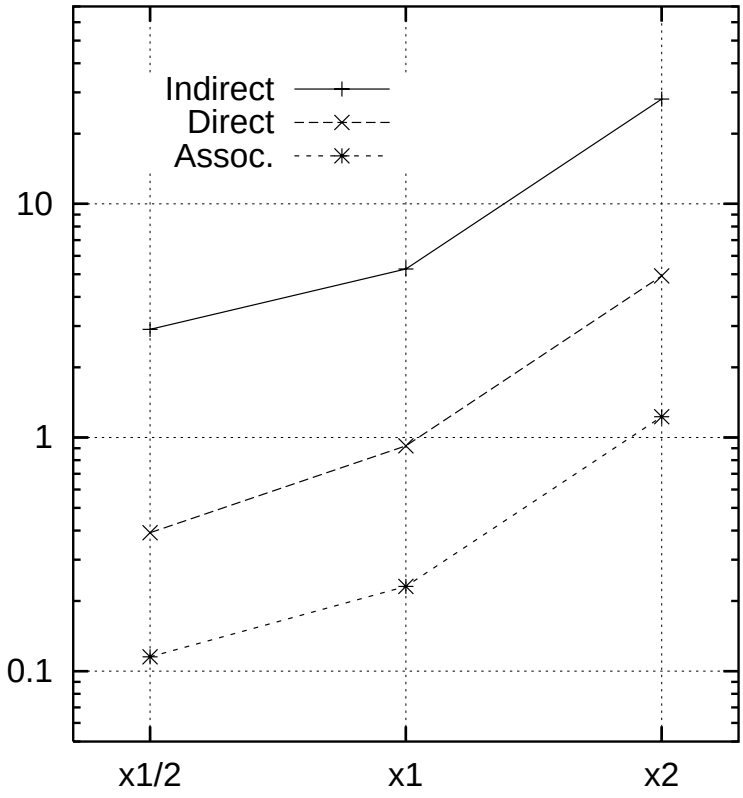


図 7.3: 回路面積 ($\times 10^6$ F^2)

7.3　回路面積

間接/直接方式共に，命令ウィンドウの一部として，ライトバックすべき物理レジスタを指示する *prD* を格納するため，連想方式のデスティネーション RAM と同じものを別途必要とする（4.1 項，6.1 項）．したがって，連想方式のソース CAM と，間接/直接方式の依存行列の面積を比較することになる．表 7.4 に各方式の各ロジックのパラメタをまとめる．

表 7.5 にセルの面積を，図 7.3 に回路の総面積を示す．

セル面積

セルの面積は，トランジスタが形成される基盤上の領域の面積と，ビットライン，ワードライン，電源バスなどの配線が形成される配線層の領域の面積の，どちらか大きい方で決まる．これらの領域の面積は，それぞれ，ポート数の増加関数で与えられる．

4T セルの面積は，基本的には，ポート数によらず一定である．それ以外の，4T セルとポートを結ぶ *n*MOS トランジスタや，ビットライン・ドライバなど，ポートに関連するトランジスタの面積は，ポート数に比例する．

それに対して，配線層の面積は，ポート数の 2 乗に比例する．例えば，ビットラインを第 1 層に縦に，ワードラインを第 2 層に横に配線した場合，配線層の横幅はビットライン数に，高さはワードライン数に，それぞれ比例する．そのため，配線領域の面積はポート数の 2 乗に比例することになる．

したがって，各方式のセルの面積のオーダはポート数の 2 乗で与えられるが，実際の面積は，ポート数の絶対値に強く依存する．

間接，直接方式のセルでは，リード・ポート数はモデルによらず 1 本でよいため，4Tセルに加えて，ビットライン・ドライバの面積も定数となる．これらの定数成分が支配的であるため，x1/2 から x2 へ，ライト・ポート数が 1 本から 4 本へ増加しても，面積の増加は 4 割程度に留まっている．

一方，連想方式のソース CAM のセルでは，実際に，面積がポート数の 2 乗に比例する部分が見られる．セル面積は，x1/2 から x1 では 4 割程度の増加に留まっているが，x1 から x2 では 4 倍近く増加している．これは，主に，サーチ・ポート数の増加による．INT サブウィンドウのソース CAM では，ライト・ポートは間接/直接方式のセルと同じ IW' 本でよいが，サーチ・ポートはその倍の $2 \cdot IW'$ 本必要となる；INT サブウィンドウのソース CAM のサーチ・ポートは，INT サブウィンドウと LS サブウィンドウから *prD* を受け付けるためである（3.7 節）．そのため，x1 では 4 本，x2 では 8 本ものサーチ・ポートが必要となり，セルの面積は実際に配線領域によってバウンドされている．

セル数

セルの総数は，連想方式では $WS \log NR'$，間接方式では $2 \cdot WS \cdot NR = 16/3 \cdot WS^2$（6.4 節），直接方式では WS^2 である．

総面積のオーダ

連想方式のセル面積のオーダを $O(IW'^2)$ とすると，連想方式の総面積のオーダは $O(IW'^2)\times O(WS\log NR')$ となる．$WS \propto IW'$ とすると，$O(IW'^2)\times O(WS\log NR') = O(IW'^3\log NR')$ とできる．ただし，$\log NR' = 5$，6，7 程度であるので，その影響は小さい．

一方，間接/直接方式のセルの面積のオーダを $O(IW'^2)$ とみなすと，総面積のオーダは，やはり $WS \propto IW'$ として，$O(IW'^2)\times O(WS^2) = O(IW'^4)$ となる．これは，連想方式の総面積のオーダ $O(IW'^3\log NR')$ より大きい．このことは，連想方式ではタグが 2 進数にエンコーディングされていることによる．

しかし実際には，上述したように，間接/直接方式のセル面積のオーダを $O(IW'^2)$ とみなすことは現実とはそぐわない．セル面積のオーダを $O(IW')$ としても，総面積のオーダは $O(IW')\times O(WS^2) = O(IW'^3)$ となる．

したがって間接/直接方式の総面積のオーダは，実用領域では，連想方式のそれより若干小さい．図 7.3 でも，間接，直接方式の曲線の傾きは連想方式のそれよりも若干緩やかになっている．

総面積

結局，どのモデルにおいても，各方式の総面積はほぼ比例関係にある．直接方式の面積は連想方式の面積のほぼ 2 倍，間接方式の面積は直接方式の 16/3 倍となっている．

7.4 回路遅延

高速化のため，それぞれの回路にはバンク分割を施し，それぞれのバンクの遅延を測定した．連想方式のソース CAM は *prL*/*prR* のそれぞれに 1 つのバンクを割り当てた．間接/直接方式では，図 6.4 に示したように，INT — INT，INT — LS の 2 つの小行列をそれぞれ 1 つのバンクとしている．更に，間接方式のデスティネーション行列，ソース行列は，それぞれ縦に（ビット幅が半分になるように），2 バンクに分割した [48]．表 7.6 に，各バンクのパラメタをまとめる．

図 7.4 に，直接 (Direct)，間接 (Indirect)，連想 (Assoc) の各方式の *wakeup* ロジックの回路

方式	ロジック	ビット数	ワード数	ライトポート数	リードポート数	比較入力ポート数
連想	デスティネーション RAM	TW	WS'	DW'	IW'	—
	ソース CAM	↑	WS'	↑	—	$2\cdot IW'$
間接	デスティネーション行列	$NR/2$	WS'	↑	1	—
	ソース行列	↑	↑	↑	↑	—
直接	—	w	w	↑	↑	—

表 7.6: 各バンクのパラメタ

遅延を示す．参考として，*select* ロジックの回路遅延を合わせて示した．縦軸は，F.O.4 インバータの遅延 (60.0ps) で正規化してある．

連想/間接/直接方式の各バーは，それぞれ，5 つ，4 つ，2 つの部分からなる．下から n 番目の部分は，それぞれ，3.18 (p. 96)，4.4 (p. 94) および図 4.5 (p. 96)，図 4.5 (p. 96) に示した各方式の回路図中の，丸数字の n 番から $n+1$ 番で示した点の間の遅延を表す．白色の部分は主にゲート遅延からなり，灰色の部分は配線遅延の影響を受ける．

なお直接方式で，x1 の $w = 32$，x1/2 の $w = 16$，32 のバーが描かれていないのは，そのような構成を採り得ないためである．x1 では，$WS' = 16$ なので，$w = 32$ という構成を採り得ない．x1/2 でも同様である．

間接方式

ソース行列の遅延は，連想方式のソース CAM のそれに比べて大幅に短縮されている．その一方で，デスティネーション行列の遅延は連想方式のデスティネーション RAM のそれに比べて大幅に増加してしまっている．

ソース行列の遅延が短縮されるのは，連想方式のソース CAM に比べて回路が大幅に簡単

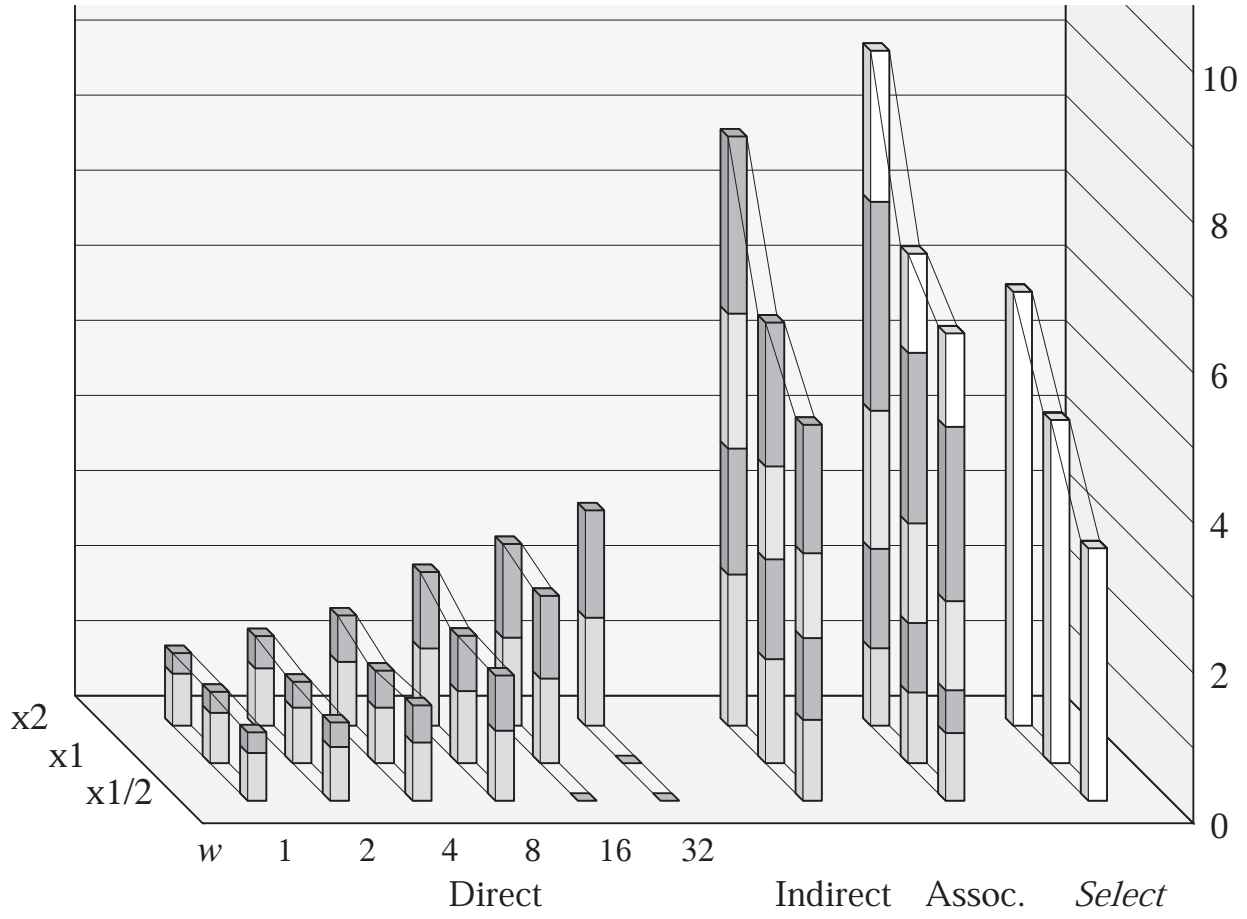


図 7.4: 各方式の回路遅延

化されているためである．連想方式のソースCAMの連想検索が，ソース行列では積和演算に置き換わっている．

一方，デスティネーション行列の遅延が大幅に増加してしまっているは，ビット幅が$\log_2 NR'$bからNR'bへと大幅に増大しているためである．例えばx2では，連想方式で7bであるものが，間接方式では128bと，約18倍にもなっている．

結局，連想方式に対する*wakeup*全体の遅延の改善はごくわずかである．遅延の点でも，スケーラビリティが低い方式と言わざるを得ない．

直接方式

それに比べて直接方式の遅延は，連想/間接方式の両者に対して大幅に短縮されている．L-1行列の狭幅化を行わなくても，その遅延は両方式の1/2以下である．$w = WS'/4 = 8$まで狭幅化した場合，IPCは約2%程度低下するが（6.5節），遅延は更に半減される．

また，x1/2からx1，x2へと，IW，WS，NRなどのパラメタが倍増していくにしたがい，連想/間接方式の遅延は急激に増大しているが，直接方式の遅延の増大の割合は比較的緩やかである．

直接方式では，遅延に影響を及ぼすセルの個数はwで与えられる．同一のwに対してパラメタを増減させたときの遅延の変化は，したがって，ライト・ポート数の増減によるセル面積の増減の影響による．グラフから，その影響はわずかであることが分かる．

総合評価

本来であれば，本節で得られた回路遅延と，6.5節で述べた狭幅化によるIPCの低下とを総合的に評価するのが通常であろう．回路遅延からクロック速度を求め，IPCとの積をとることで，総合的な性能を評価することができる．しかし今回の場合には，そのようにしても意味のある数値を得ることはできない．それは，狭幅化を行うまでもなく，直接方式の回路遅延が十分に短いことによる．

直接方式の遅延が連想方式のそれの半分であることは，その程度まで削減しないとクリティカルではなくならない，ということを意味しない．モデルx1では，連想方式の*wakeup*ロジックと*select*ロジックの遅延の和は，F.O.4インバータ10個分程度であり，ぎりぎりクリティカルになるかならないかといった程度である．直接方式の遅延が連想方式のそれの半分であることは，したがって，現状では設計に対して大幅なマージンを提供することになる．

また，更なる低遅延化を達成する狭幅化は，今のところは無用である．しかし，配線遅延の影響が増大する将来には，直接方式*wakeup*ロジックの遅延も相対的に増大することになる．そのとき狭幅化は，それを非クリティカルにする強力な武器となるだろう．

したがって，直接方式によって，*wakeup*ロジックがクリティカルになる可能性は，将来に渡って，極めて低くなったと結論づけることができる．

第8章　結論

Out-of-order 命令スケジューリングは，配線遅延が支配的となる将来の ILP プロセッサにおいて，高い IPC を維持するために不可欠な技術となる．しかし，その複雑さゆえに，特に *wakeup* と呼ばれるロジックがクリティカルになると考えられてきた．本稿は，この *wakeup* ロジックを高速化するための研究の成果についてまとめたものである．得られた主要な成果は以下のとおりである：

Dualflow アーキテクチャ

Dualflow は，制御駆動とデータ駆動の両方の性質をあわせ持つ命令セット・アーキテクチャである．制御駆動型アーキテクチャと同様のプログラム・オーダを定義するが，制御駆動型アーキテクチャようなレジスタを定義しない．命令間のデータの受け渡しは，制御駆動型アーキテクチャのようにレジスタを介して間接的に行われるのではなく，データ駆動型アーキテクチャのように定義側の命令が使用側の命令を直接的に指定することで行われる．

Dualflow アーキテクチャでは，命令間の依存関係を表す行列を用いて命令スケジューリングを実現することになる．その結果，スーパースカラ・プロセッサと同様の動的命令スケジューリングを行いながら，行列を格納する小型の RAM 1 個を読み出すことで *wakeup* ロジックを構成することができる．

直接依存行列を用いた命令スケジューリング方式

直接方式は，dualflow アーキテクチャで開発された依存行列を用いた *wakeup* ロジックを，通常の制御駆動型アーキテクチャを持つスーパースカラ・プロセッサに応用したものである．

DEC* Alpha 21264 プロセッサで採用されている間接方式と，この直接方式を定量的に比較した．

間接方式は，2 つの行列，デスティネーション行列とソース行列を逐次的にアクセスして *wakeup* を実現する．間接方式では，回路は単純化されるものの，回路面積が大幅に増加してしまう．その結果，従来の連想方式に対してほとんど遅延の改善が見られなかった．

一方直接方式は，間接方式の 2 つの行列の行列積を 1つの RAM にまとめたものと考えることができる．そのため，間接方式に対して，面積は 3/16 以下，遅延は 1/2 以下になることが分かった．

更に，狭幅化と呼ぶ最適化を施した場合，IPC は約 2%程度低下するが，遅延は更に半減され，その絶対値は F.O.4 インバータ2 個分を切る．したがって，*wakeup* ロジックの遅延がクリティカルになる可能性は極めて低くなったと結論づけることができる．

---

* 発表当時.

謝辞

本論文の執筆にあたり，ご指導，ご鞭撻を賜った京都大学 富田 眞治 教授に深謝いたします．

本論文をまとめることができましたのは，在学中から現在に至るまで，数多くの方々からご指導，ご助言，ご協力をいただいたおかげです．

在学中は，豊橋技術科学大学 中島 浩 教授，京都大学 森 眞一郎 助教授にご指導を賜りました．

当時研究室の協力スタッフであった，京都大学 経済学研究科 中島 康彦 助教授，広島市立大学 北村 俊明 教授には，ご助言，ご協力をいただきました．

当時研究室の学生であったグエン ハイ ハー氏，縣 亮慶 氏，西野 賢悟 氏には，評価データの収集をしていただきました．

ここに深甚なる謝意を表します．

富士通株式会社には，LSI の設計情報をご提供いただきました．この情報のおかげで，研究結果がずいぶんと説得力のあるものになったと思います．また，実際の設計情報を用いた設計は，大学人としては得難い，貴重な経験となりました．ここに深謝いたします．

また，本研究の重要なアイデアの多くは，学会会場などにおいて多くの方々からご教示いただいたものです．

名古屋大学 安藤 秀樹 助教授には，博士論文 [17] をとおして，ILP プロセッサの研究を始めるきっかけをいただきました．

JSPP 2000 の会場では，筑波大学 久野 靖 教授，九州工業大学 佐藤 寿倫 助教授に，dualflow アーキテクチャ用の *wakeup* ロジックが通常のスーパースカラ・プロセッサにも応用可能であることをご指摘いただきました．

SWoPP 2001 の会場では，豊橋技術科学大学 中島 浩 教授に，直接方式の行列が，物理レジスタ番号からではなく，論理レジスタ番号から直接更新可能であることをご指摘いただきました．

MICRO-34 の会場では，Intel Corp. の Eric Sprangle 氏に，ソース・オペランドごとの行列を 1 つにまとめることが可能であることをご指摘いただきました．

これらのご指摘がなければ，本研究は実用性の低いものに留まっていたと思います．ここに深甚なる謝意を表します．

参考文献

[1] 五島正裕, グェンハイハー, 森眞一郎, 富田眞治: Dual-Flow: 制御駆動とデータ駆動を融合したプロセッサ・アーキテクチャ, 情報処理学会研究報告 98–ARC–130 (SWoPP'98), pp. 115–120 (1998).

[2] 五島正裕, グェンハイハー, 縣亮慶, 森眞一郎, 富田眞治: Dualflow アーキテクチャとそのコード生成手法, 情報処理学会研究報告 99–ARC–134 (SWoPP'99), pp. 163–168 (1999).

[3] 五島正裕, グェンハイハー, 縣亮慶, 森眞一郎, 富田眞治: Dualflow アーキテクチャの提案, 並列処理シンポジウム JSPP 2000, pp. 197–204 (2000).

[4] グェンハイハー, 縣亮慶, 五島正裕, 中島康彦, 森眞一郎, 富田眞治: Dualflow アーキテクチャの命令発行機構, 情報処理学会研究報告 2000–ARC–139 (SWoPP 2000), pp. 103–108 (2000).

[5] 五島正裕, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: Dualflow アーキテクチャの命令発行機構, 情報処理学会論文誌, Vol. 42, No. 4, pp. 652–662 (2001).

[6] 縣亮慶, グェンハイハー, 五島正裕, 中島康彦, 森眞一郎, 富田眞治: Superscalar における低遅延な命令スケジューリング方式, 情報処理学会研究報告 2000–ARC–139 (SWoPP 2000), pp. 109–114 (2000).

[7] 五島正裕, 西野賢悟, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 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, 情報処理学会研究報告 2001–ARC–142 (HOKKE 2001), pp. 121–126 (2001).

[8] 西野賢悟, 五島正裕, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 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の改良, 並列処理シンポジウム JSPP 2001, pp. 137–138 (2001). （ポスター）.

[9] 五島正裕, 西野賢悟, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 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, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, Vol. 42, No. SIG 9(HPS 3), pp. 77–92 (2001).

[10] 西野賢悟, 小田累, 五島正裕, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: スーパースケーラのための高速な命令スケジューリング方式の IPC の評価, 情報処理学会研究報告 2001–ARC–144 (SWoPP 2001), pp. 171–176 (2001).

[11] Goshima, M., Nishino, K., Nakashima, Y., Mori, S., Kitamura, T. and Tomita, S.: A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, *34th Annual Int'l Symp. on Microarchitecture (MICRO-34)*, pp. 225–236 (2001).

[12] 五島正裕, 西野賢悟, 小西将人, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 行列を用いた Out-of-Order スケジューリング方式の評価, 並列処理シンポジウム JSPP 2002, pp. 11–18 (2002).

[13] 五島正裕, 西野賢悟, 小西将人, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 行列に基づく Out-of-Order スケジューリング方式の評価, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, Vol. 43, No. SIG 6(HPS5), pp. 13–23 (2002).

[14] 小西将人, 小田累, 西野賢悟, 五島正裕, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: クラスタ化スーパースケーラ・プロセッサにおける直接依存行列型スケジューリング方式, 並列処理シンポジウム JSPP 2002, pp. 19–26 (2002).

[15] 小西将人, 西野賢悟, 五島正裕, 中島康彦, 森眞一郎, 富田眞治: 直接依存行列型命令スケジューリングを適用したクラスタ化スーパースケーラ・プロセッサの評価, 情報処理学会研究報告 2002–ARC–149 (SWoPP 2002), pp. 151–156 (2002).

[16] 浅見直樹, 枝洋樹: 次世代マイクロプロセッサ, 日経エレクトロニクス, No. 627, pp. 67–150 (1995).

[17] 安藤秀樹: 投機的実行を行うマイクロプロセッサに関する研究, 博士論文, 京都大学 (1996).

[18] Johnson, M.: *Superscalar Micorprocessor Design*, Prentice-Hall (1991).

[19] Schlansker, M. S. and Rau, B. R.: EPIC: Explicitly Parallel Instruction Computing, *IEEE Computer*, Vol. 33, No. 2, pp. 37–45 (2000).

[20] Ditzel, D. R.: Transmeta's Crusoe: A Low-Power x86-Compatible Microprocessor Built with Software, *COOL Chips III* (2000). (Keynote Presentation).

[21] Transmeta Corp.: http://www.transmeta.com/efficeon/.

[22] Tremblay, M. and O'Connor, J. M.: UltraSPARC I: A Four-Issue Processor Supporting Multimedia, *IEEE Micro*, No. 4, pp. 42–49 (1996).

[23] Yeager, K. C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol. 16, No. 2, pp. 28–40 (1996).

[24] 佐藤寿倫, 有田五次郎: 命令レベル並列プロセッサにおける可変レイテンシパイプラインの効果, 情報処理学会研究報告 99–ARC–135 (1999).

[25] 佐藤寿倫, 有田五次郎: 可変レイテンシパイプライン技術と演算結果再利用技術の併用による演算レイテンシ削減, 電子情報通信学会論文誌, Vol. J85-DI, No. 12, pp. 1103–1113 (2002).

[26] 中島康彦, 緒方勝也, 正西申悟, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 関数値再利用および並列事前実行による高速化技術, 並列処理シンポジウム JSPP 2002, pp. 269–276 (2002).

[27] 中島康彦, 緒方勝也, 正西申悟, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 関数値再利用および並列事前実行による高速化技術, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, Vol. 43, No. SIG 6(HPS 5), pp. 1–12 (2002).

[28] 津邑公暁, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: ステレオ画像処理を用いた曖昧再利用の評価, 先進的計算基盤システムシンポジウム SACSIS 2003, pp. 97–104 (2003).

[29] 中島康彦, 津邑公暁, 五島正裕, 森眞一郎, 富田眞治: 動的命令解析に基づく多重再利用および並列事前実行, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 10(ACS 2), pp. 1–16 (2003).

[30] 清水雄歩, 中島康彦, 津邑公暁, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 距離画像生成処理におけるメディアプロセッサの評価, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp. 257–267 (2003).

[31] 津邑公暁, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: ステレオ画像処理を用いた曖昧再利用の評価, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp. 246–256 (2003).

[32] 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 並列事前実行機構における主記憶値テストの高速化, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 12(ACS 4) (2003).

[33] Keller, J.: The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, *9th Annual Microprocessor Forum* (1996).

[34] Palacharla, S., Jouppi, N. P. and Smith, J. E.: Complexity-Effective Superscalar Processors, *Proc. 24th Int'l Symp. on Computer Architecture (ISCA24)* (1997).

[35] Intel Corp.: *A detailed look inside the NetBurst micro-architecture of the Intel Pentium 4 processor* (2000).

[36] Wilson, K. M., Olukotun, K. and Rosenblum, M.: Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors, *23rd. Int'l Symp. on Comp. Architecuture*, pp. 147–157 (1996).

[37] Wilson, K. M. and Olukotun, K.: High Bandwidth On-Chip Cache Design, *IEEE Trans. Comp.*, Vol. 50, No. 4, pp. 292–307 (2001).

[38] 福田祥貴, 片山清和, 安藤秀樹, 島田俊夫: ライン・バッファ・ヒット/ミス予測を利用した動的命令スケジューリング, 情報処理学会研究報告 2003–ARC–149 (SWoPP 2003), pp. 139–144 (2003).

[39] 福田祥貴, 片山清和, 島田俊夫: ライン・バッファ・ヒット/ミス予測を利用した動的命令スケジューリングの高精度化手法, 先進的計算基盤システムシンポジウム SACSIS 2003, pp. 227–234 (2003).

[40] McFarling, S.: Combining Branch Predictors, Technical Report WRL Technical Note TN-36, Digital Equipment Corp. (1993).

[41] Morancho, E., Llabería, J. M. and Olivé, A.: Recovery Mechanism for Latency Misprediction, *2001 Int'l Conf. of Parallel Architectures and Compilation Techniques (PACT2001)* (2001).

[42] Lipasti, M. H. and Shen, J. P.: Exceeding the Dataflow Limit via Value Prediction, *29th. Int'l Symp. on Microarchitecture (MICRO-29)*, pp. 226–237 (1996).

[43] Sazeides, Y. and Smith, J. E.: The Predictablity of Data Values, *30th. Int'l Symp. on Microarchitecture (MICRO-30)*, pp. 248–257 (1997).

[44] Chrysos, G. and Emer, J.: Memory Dependence Prediction using Store Sets, *Proc. 25th Int'l Symp. on Computer Architecture (ISCA)*, pp. 142–153 (1998).

[45] Burger, D., Austin, T. M. and Bennett, S.: Evaluating Future Microprocessors: The SimpleScalar ToolSet, Technical Report CS-TR-1308, Univ. of Wisconsin-Madison (1996).

[46] SimpleScalar LLC: http://www.simplescalar.com/.

[47] SPEC Corp.: http://www.spec.org/.

[48] Farrell, J. A. and Fischer, T. C.: Issue logic for a 600-MHz out-of-order execution microprocessor, *IEEE Journal of Solid-State Circuits*, Vol. 33, No. 5 (1998).

[49] Brown, M. D., Stark, J. and Patt, Y. N.: Select-Free Instruction Scheduling Logic, *34th Annual International Symposium on Microarchitecture (MICRO-34)*, pp. 204–213 (2001).

[50] Palacharla, S., Jouppi, N. P. and Smith, J. E.: Quantifying the Complexity of Superscalar Processors, Technical report, Univ. of Wisconsin-Madison (1996).

[51] Bakogulu, H. B.: *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley (1990).

[52] Rabaey, J. M.: *Digital Interconnection Circuits — A Design Perspective*, Prentice Hall Electronics and VLSI Series (1996).

[53] McFarland, G. and Flynn, M.: Limits of Scaling MOSFETS, Technical Report CSL-TR-95-662, Stanford University (1995).

[54] Rahmat, K., Nakagawa, O. S., Oh, S.-Y. and Moll, J.: A Scaling Scheme for Interconnection in Deep-Submicron Processes, Technical Report HPL-95-77, Hewlett-Packard Laboratories (1995).

[55] Intel: *IA-32 Intel(R) Architecture Software Developer's Manual* (2002).

[56] SPARC International Inc.: *The SPARC Architecture Manual Version 9* (1994).

[57] Kumar, A.: The HP PA-8000 RISC CPU, *Hot Chips VIII*, pp. 9–20 (1996).

[58] 弓場敏嗣, 山口喜教: データ駆動型並列計算機, オーム社 (1993).

[59] Tune, E., Liang, D., Tullsen, D. M. and Calder, B.: Dynamic Prediction of Critical Path Instructions, *7th Int'l Symp. on High Performance Computer Architecture (HPCA7)* (2001).

[60] 小林良太郎, 安藤秀樹, 島田俊夫: データフロー・グラフの最長パスに着目したクラスタ化スーパースカラ・プロセッサにおける命令発行機構, 並列処理シンポジウム JSPP 2001, pp. 31–38 (2001).

[61] Fields, B. and Blodik, S. R. R.: Focusing Processor Policies via Critical-Path Prediction, *28th Int'l Symp. on Computer Architecture (ISCA-28)*, pp. – (2001).

[62] Sun Microsystems, inc.: *UltraSPARC User's Manual* (1997).

[63] Asato, C., Montoye, R., Gmuender, J., Simmons, E. W., Ike, A. and Zasio, J.: A 14-port 3.8ns 116-word 64b Read Renaming Register File, *1995 IEEE International Solid-State Circuit Conference Digest of Technical Papers*, pp. 104–105 (1995).

[64] Farkas, K. I., Jouppi, N. P. and Chaw, P.: Register File Design Consideration in Dynamically Scheduled Processors, *the Second IEEE Symposium on High-Performance Computer Architecture*, pp. 40–51 (1996).

[65] Henry, D. S., Kuszmaul, B. C., Loh, G. H. and Sami, R.: Circuits for Wide-Window Superscalar Processors, *Proc. 27th Int'l Symp. on Computer Architecture (ISCA 2000)* (2000).

[66] Sakai, S., Kodama, Y. and Yamaguchi, Y.: Prototype Implementation of a Highly Parallel Dataflow Machien EM-4, *5th International Parallel Processing Symposium*, pp. 278–286 (1991).

[67] Sakai, S., Kodama, Y., Hiraki, K. and Yamaguchi, Y.: Design of the Dataflow Single-Chip Processor EMC-R, *Journal of Information Processing*, Vol. 13, No. 2, pp. 165–173 (1990).

[68] Papadopoulos, G. M. and Culler, D. E.: Monsoon: an Explicit Token-Store Architecture, *IEEE*, pp. 82–91 (1990).

著者発表論文

1. Araki, S., Goshima, M., Mori, S., Nakashima, H., Tomita, S., Akiyama, Y. and Kanehisa, M.: Application of Parallelized DP and A∗ Algorithm to Multiple Sequence Alignment, *Proc. Genome Information Workshop IV*, pp. 94–101 (1993).
2. Mori, S., Saito, H., Goshima, M., Yagihata, M., Tanaka, T., Fraser, D., Joe, K., Nitta, H. and Tomita, S.: A Distributed Shared Memory Multiprocessor: ASURA —Memory and Cache Architectures—, *Proc. Int'l Conf. on Supercomputing '93*, pp. 740–749 (1993).
3. Mori, S., Goshima, M., Nakashima, H. and Tomita, S.: A Proposal of Self-Cleanup Cache, *Proc. Int'l Conf. on Parallel Architecture and Compiler Techinique, PACT'95*, pp. 298–301 (1995).
4. 五島正裕, 松本重光, 森眞一郎, 中島浩, 富田眞治: Virtual Queue : 超並列計算機向きメッセージ通信機構, 並列処理シンポジウム JSPP'95, pp. 225–232 (1995).
5. 森眞一郎, 五島正裕, 福島直人, 中島浩, 富田眞治: Self-Cleanup Cache の提案, 並列処理シンポジウム JSPP'95, pp. 265–272 (1995).
6. 五島正裕, 森眞一郎, 富田眞治: Virtual Queue : 超並列計算機向きメッセージ通信機構, 情報処理学会論文誌, Vol. 37, No. 7, pp. 1399–1408 (1996).
7. 森眞一郎, 五島正裕, 福島直人, 中島浩, 富田眞治: Self-Cleanup Cache の提案, 情報処理学会論文誌, Vol. 38, No. 2, pp. 321–331 (1997).
8. 後藤慎也, 窪田昌史, 田中利彦, 五島正裕, 森眞一郎, 中島浩, 富田眞治: 並列化コンパイラ TINPAR による非均質計算環境向けコード生成手法, 並列処理シンポジウム JSPP'97, pp. 205–212 (1997).
9. Kubota, A., Tatsumi, S., Tanaka, T., Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: A Technique to Eliminate Redudant Inter-Processor Communication on Paralleliz-ing Compiler TINPAR, *Proc. Int'l Symp. on High Performance Computing (ISHPC'97)*, Springer, pp. 195–204 (1997). Springer LNCS 1336, High Perfromance Computing.
10. Ohno, K., Ikawa, M., Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: Improvement of Message Communication in Concurrent Logic Language, *Proc. 2nd. Int'l Symp. on Parallel Symbolic Computation, PASCO'97*, pp. 156–164 (1997).

11. Ohno, K., Ikawa, M., Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: Efficient Goal Scheduling in Concurrent Logic Language using Type-Based Dependency Analysis, *Proc. 4th Asian Computing Science Conf. (ASIAN'97)*, pp. 268–282 (1997). Springer LNCS 1345, Advances in Computing Science —ASIAN'97.

12. Goto, S., Kubota, A., Tanaka, T., Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: Optimized Code Generation for Heterogeneous Computing Environment using Parallelizing Compiler TINPAR, *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, Paris, France, pp. 426–433 (1998).

13. Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: The Intelligent Cache Controller of a Massively Parallel Processor JUMP-1, *IWIA'97, Int'l Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 116–124 (1997).

14. Kubota, A., Tatsumi, S., Tanaka, T., Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: A Technique to Eliminate Redundant Inter-Processor Communication on Parallelizing Compiler TINPAR, *Int'l Journal of Parallel Programming*, Vol. 27, No. 2, pp. 97–109 (1999).

15. 津邑公暁, 三輪忍, 五島正裕, 富田眞治: 記憶構造観測のための神経網シミュレーション, 第 20 回計測自動制御学会システム工学部会研究会『人工生命の新しい潮流』, 計測自動制御学会, pp. 111–114 (2000).

16. 秤谷雅史, 小西将人, 五島正裕, 森眞一郎, 富田眞治: 並列計算機 JUMP-1 における分散共有メモリ管理, *JSPP2000*, pp. 67–74 (2000).

17. 五島正裕, グェンハイハー, 縣亮慶, 森眞一郎, 富田眞治: Dualflow アーキテクチャの提案, 並列処理シンポジウム JSPP 2000, pp. 197–204 (2000).

18. 五島正裕, 斎藤康二, 小西将人, 秤谷雅史, 森眞一郎, 富田眞治: 並列計算機 JUMP-1 における分散共有メモリ・システム, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, Vol. 41, No. SIG 8(HPS 2), pp. 15–27 (2000).

19. 五島正裕, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: Dualflow アーキテクチャの命令発行機構, 情報処理学会論文誌, Vol. 42, No. 4, pp. 652–662 (2001).

20. 小西将人, 五島正裕, 森眞一郎, 富田眞治: 並列計算機 JUMP-1 における分散共有メモリ管理の実装, 情報処理学会論文誌, Vol. 42, No. 4, pp. 674–682 (2001).

21. 山田克樹, 尼崎央典, 中島康彦, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: Java バイトコード実行におけるデータ再利用の分析, *JSPP 2001*, pp. 173–180 (2001).

22. 西野賢悟, 五島正裕, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 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の改良, 並列処理シンポジウム JSPP 2001, pp. 137–138 (2001). （ポスター）.

23. 五島正裕, 西野賢悟, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 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, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, Vol. 42, No. SIG 9(HPS 3), pp. 77–92 (2001).

24. Le Moal, D., Masuda, M., Goshima, M., Mori, S., Nakashima, Y., Kitamura, T. and Tomita, S.: Priority Enhanced Stride Scheduling, *Int'l Conf. on High-Performance Computing in the Asia-Pacific Region (HPCAsia'01)* (2001). (online proceedings).

25. Goshima, M., Nishino, K., Nakashima, Y., Mori, S., Kitamura, T. and Tomita, S.: A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, *34th Annual Int'l Symp. on Microarchitecture (MICRO-34)*, pp. 225–236 (2001).

26. 中島康彦, 緒方勝也, 正西申悟, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 関数値再利用および並列事前実行による高速化技術, 並列処理シンポジウム JSPP2002, pp. 269–276 (2002).

27. 小西将人, 小田累, 西野賢悟, 五島正裕, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: クラスタ化スーパースケーラ・プロセッサにおける直接依存行列型スケジューリング方式, 並列処理シンポジウム JSPP 2002, pp. 19–26 (2002).

28. 五島正裕, 西野賢悟, 小西将人, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 行列を用いた Out-of-Order スケジューリング方式の評価, 並列処理シンポジウム JSPP 2002, pp. 11–18 (2002).

29. 木村篤彦, 中島康彦, 宮田佳昭, 中川伸二, 北村俊明, 五島正裕, 森眞一郎, 富田眞治: 低電力 Java プロセッサのための投機的クロック制御, 情報処理学会論文誌, Vol. 43, No. 6, pp. 1956–1967 (2002).

30. 五島正裕, 西野賢悟, 小西将人, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 行列に基づく Out-of-Order スケジューリング方式の評価, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, Vol. 43, No. SIG 6(HPS5), pp. 13–23 (2002).

31. 中島康彦, 緒方勝也, 正西申悟, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 関数値再利用および並列事前実行による高速化技術, 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム, Vol. 43, No. SIG 6(HPS 5), pp. 1–12 (2002).

32. Le Moal, D., Ikumo, M., Tsumura, T., Goshima, M., Mori, S., Nakashima, Y., Kitamura, T. and Tomita, S.: Priority Enhanced Stride Scheduling, *IPSJ Trans. High Performance Computing Systems*, Vol. 43, No. SIG 6(HPS 5), pp. 99–111 (2002).

33. 津邑公暁, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: ステレオ画像処理を用いた曖昧再利用の評価, 先進的計算基盤システムシンポジウム SACSIS 2003, pp. 97–104 (2003).

34. 額田匡則, 小西将人, 五島正裕, 中島康彦, 富田眞治: 参照の空間局所性を最大化するボリューム・レンダリング・アルゴリズム, 先進的計算基盤システムシンポジウム SACSIS 2003, pp. 333–340 (2003).

35. 小西将人, 五島正裕: リザベーション・ステーションと物理レジスタ・ファイルを併用するスーパースカラ・プロセッサの構成方式, 先進的計算基盤システムシンポジウム SACSIS 2003, pp. 191–192 (2003). （ポスター）.

36. 中島康彦, 津邑公暁, 五島正裕, 森眞一郎, 富田眞治: 動的命令解析に基づく多重再利用および並列事前実行, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 10(ACS 2), pp. 1–16 (2003).

37. 清水雄歩, 中島康彦, 津邑公暁, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 距離画像生成処理におけるメディアプロセッサの評価, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp. 257–267 (2003).

38. 津邑公暁, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: ステレオ画像処理を用いた曖昧再利用の評価, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp. 246–256 (2003).

39. 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 並列事前実行機構における主記憶値テストの高速化, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 12(ACS 4) (2003).

40. 額田匡則, 小西将人, 五島正裕, 中島康彦, 富田眞治: 参照の空間局所性を最大化するボリューム・レンダリング・アルゴリズム, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp. 137–146 (2003).