計算機アーキテクチャ　101－10
（１９９３．　８．　２０）

スーパスカラプロセサにおけるリカバリー方式

安里　彰　　志村　浩也

富士通研究所

非順序実行を行うスーパスカラプロセサは、正確な割り込みを実現するためのリカバリー機構を備えていることが必要であり、その一手法として従来よりリオーダバッファやフューチャファイルを用いた方式が提案されている。しかし性能向上のための有力な手段であるレジスタリネーミングを採用したプロセサで従来通りのリカバリー機構を実装すると、ハードウェア量が非常に大きくなる問題がある。我々は本論文で、レジスタリネーミングを行うプロセサにおいても少ないハードウェアコストで実現できるリカバリー方式を提案し、従来方式とのハードウェア量の比較を試みる。

A Recovery Method for Superscalar Processors

Akira Asato　　Kouya Shimura

FUJITSU Laboratories Ltd.
1015 Kamikodanaka Nakahara-ku, Kawasaki, Kanagawa 211

A processor using out-of-order execution should provide some recovery mechanism which can support precise interrupts, and a method with reorder-buffer and future-file has been proposed. But the hardware cost of such method increases too much in processors which adopt the register renaming mechanism. In this paper we propose a recovery method which needs lower hardware cost even in processors with register renaming. Then we estimate the actual hardware cost and compare it with other methods.

1 はじめに

汎用プロセサの実行並列度を向上させるためのアーキテクチャの一つに、複数の命令実行パイプラインを用いて命令の並列実行を可能にしたスーパスカラアーキテクチャがある。その枠組の中でも、より高い性能を得るための種々の工夫が数多く提案されている [1]。その中でも効果の大きいものの一つに命令の out-of-order 実行がある。これはプログラム上の命令順序に拘らずに、オペランドが有効になり次第、命令の実行を開始する方式である。

out-of-order 実行方式を採用すると、命令の終了がプログラム順通りに行なわれる保証がなくなるため、いわゆる precise interrupt の問題が発生する。この問題を回避するためには、命令 A で割り込みが起こったとすれば、命令ストリーム上で A に先行する命令は全て実行が終了し、A 以降の命令は一つも終了してないようなプロセサの状態 (レジスタやメモリの内容) を導く機構が必要になる。これをリカバリー機構と呼ぶ。out-of-order 実行を採用したプロセサでは、何らかのリカバリー機構が必須であり、いくつかの方式が提案されている。[2][3]

一方、スーパスカラプロセサは複数のパイプラインを有することや、制御が複雑であることなどから、一般にハードウェアコストが大きいと考えられるので、性能向上のための機構であっても実装上のコストを十分に意識して設計することが重要である。特に上記のリカバリー機構はハードウェア量が相対的に大きいので、コンパクトなインプリメントが望まれる。

我々は本論文でリカバリーのための一方式を提案し、他方式とのハードウェア量の比較を試みる。2 章では従来より提案されているリオーダバッファおよびフューチャファイルを用いたリカバー方式を簡単に紹介し、3 章で我々の方式を説明する。4 章ではリカバリーに用いるテーブル群の最適なエントリ数についてシミュレーションの結果を踏まえて議論し、5 章では 4 章で求めた値に基づいてハードウェア量を算定し他方式と比較する。

2 従来方式と問題点

2.1 文献 [2] のリカバリー方式

本節では基本的なリカバリー方式であり我々の方式にも関連深い、文献 [2] で提案されている、リオーダバッファおよびフューチャファイルを用いたリカバリーについて述べる。

図 1 に、リオーダバッファの概念を示す。リオーダバッファの 1 個のエントリが 1 個の命令に対応し、top と tail の 2 個のレジスタによって指される間の領域が valid である。一つの命令がデコードされると、top レジスタの値がその命令を識別する id として与えられ、リオーダバッファ中の top で指されるエントリの $< regN >$ 領域にその命令のデスティネーションレジスタの番号が格納され、top レジスタはインクリメントされる。やがて命令が実行されて結果が得られると、その結果はレジスタファイルに書き込まれずに、リオーダバッファ中のその命令の id をアドレスとするエントリの $< val >$ 領域に格納される。また命令実行中に例外が発生すると、対応するエントリの $< exp >$ 領域がそれを示す値に書き換わる。

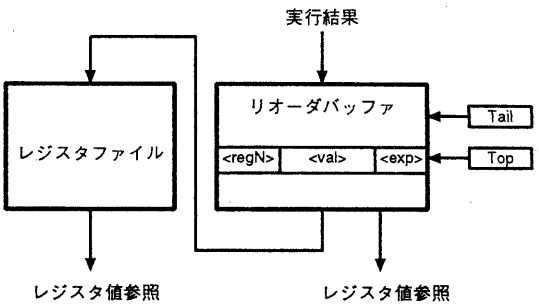

図 1 リオーダバッファの構成

また、tail レジスタによって指されるエントリにある命令の結果が既に得られていて、かつ例外が起こっていなければ、その命令はコミット可能と見做され、結果がレジスタファイルに書き戻されるとともに、tail レジスタがインクリメントされる。即ち、top と tail によってリオーダバッファは FIFO の如く振舞うと考えれば良い。

一方、tailエントリの命令で例外が起こっていた場合は、topとtailを共に0にしてリオーダバッファをフラッシュすれば良く、これによってレジスタファイル上に直前の命令終了時の正しいプロセサ状態が保証される。何故なら、エントリの割り当ては命令ストリームの順に行なわれるので、書き戻しも命令の順序通りに行なわれることになり、ある命令の結果が書き戻された時点のレジスタファイルの内容は、その命令までプログラムの実行が進んだ段階のプロセサの状態を正しく反映しているからである。

同一レジスタを更新する複数個の命令が同時にリオーダバッファ中に存在する可能性がある。そのレジスタ値を参照する場合はこれらの中から最新の値を選び出す必要があるが、こういった連想的な参照のインプリメントは実装が困難である。この問題を解決するために提案されたのがフューチャファイルである。

図2にフューチャファイルの概念を示す。フューチャファイルはアーキテクチャで与えられたレジスタ個数分のエントリを持ち、各エントリの$< val >$領域には、そのエントリ番号をデスティネーションレジスタの番号とする命令群のうち最新の命令の結果を格納する。フューチャファイルは普通のレジスタファイル構成をしているので、リオーダバッファから参照する代わりにフューチャファイルからレジスタ値の参照を行なうようにすれば、前述のような連想的な参照の必要がなくなり、インプリメントは容易になる。

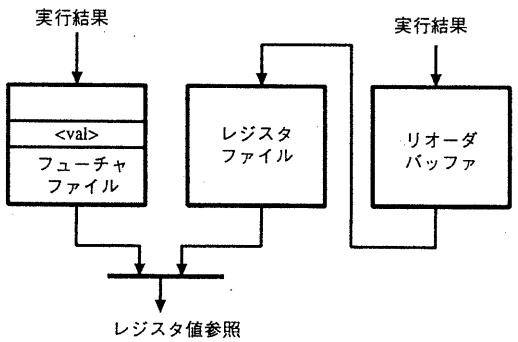

図2 フューチャファイルの構成

2.2 レジスタリネーミング

プロセサの高速化手法の一つにレジスタリネーミングがある。レジスタリネーミングは、命令間の逆依存や出力依存を解消させてプロセサの性能を向上させる技術である。後で示すようにその効果は特に浮動小数点演算命令を多用するプログラムで顕著なので、是非とも取り入れたい手法である。

レジスタリネーミングをインプリメントするためには、アーキテクチャで与えられた個数以上の(通常は2のべき乗倍の)レジスタをハードウェアとして用意する必要がある。しかし、文献[2]のリカバリー方式を採用していると、フューチャファイルのエントリ数をレジスタ数の増加に合わせて増やさねばならず、実装上のネックになる可能性がある。

3 我々の提案する方式

我々はレジスタリネーミングを行なうプロセサにおいて、より小さいハードウェアコストでリカバリーを実現するための方式を本章で提案する。提案の骨子は、リネーミング前後のレジスタ番号の組でプロセサの状態を表現するということである。

以下の各節では、我々の提案する方式による通常の命令実行時の動作および例外発生時のリカバリー動作について説明する。

3.1 デコード

図3にデコード時の動作を示す。同時にデコードできる命令数をN_{dec}として、デコードの際には各命令にシリアル番号SN_0, SN_1, , SN_{Ndec-1}を割り当てる。SNは命令ストリーム中の順序に従って割り当てられ、命令がコミットするまでの個々の命令の識別子として用いられる。SN空間の要素数をN_{SN}とする。SNの割り当てはSN管理ブロックが行なう。

デコード時にはまたソースレジスタおよびデスティネーションレジスタのリネーミングも行なう。リネーミングは、アーキテクチャで与えられた個数($2^{R_{arc}}$個)の要素を持つアーキテクチャ空間(S_{arc})から、$2^{R_{ren}}$個の要素を持つリネーミング空間(S_{ren})へのマッピングであり、RN管理ブロックが制御す

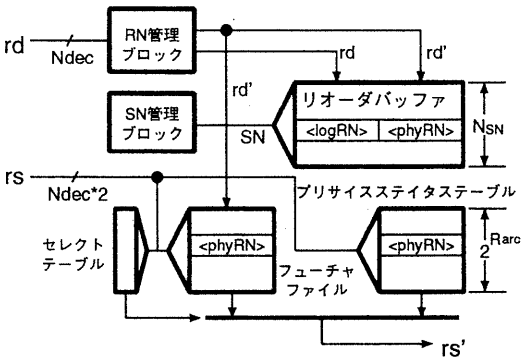

図3　我々の方式(デコード時)

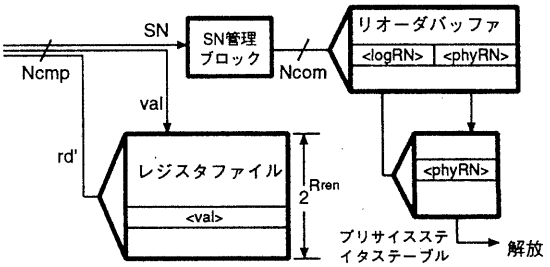

図4　我々の方式(終了・コミット時)

る。

RN管理ブロックはS_{ren}の中からフリーなN_{dec}個の番号を、デコードする各命令のデスティネーションレジスタ$rd_i(0 \leq i < N_{dec})$それぞれに1個ずつ割り当てる。割り当てた番号をrd'_iとする。そして、フューチャファイルのrd_iで指されるエントリの$< phyRN >$領域にrd'_iを登録する。フューチャファイルは$2^{R_{arc}}$個のエントリを持ち、エントリはS_{ren}の要素を表現するためのR_{ren}ビットの領域である。

また、それと同時にリオーダバッファのSN_iで指されるエントリの$< logRN >$領域にrd_iを、$< phyRN >$領域にrd'_iを格納する。リオーダバッファはN_{SN}個のエントリを持つテーブルで、エントリ番号をSNとする命令のrdとrd'の組を保持するようにする。$< logRN >$領域はR_{arc}ビット、$< phyRN >$領域はR_{ren}ビットの長さである。

ソースレジスタのリネーミングは、ソースレジスタ番号rs_iを、フューチャファイルまたはプリサイスステイタステーブルの、対応するエントリの値rs'_iに変換することで行なう。プリサイスステイタステーブルはフューチャファイルと同じ構成をしている。両者の使い分けは、セレクトテーブルのrs_iに対応するエントリの値を用いる。セレクトテーブルの値の設定に関しては後述する。ただし図には現れていないが、同時にデコードされる命令間に依存関係があった場合は、その中で先行する命令のrd'がrs'として用いられることもある。

3.2　オペランドフェッチ

レジスタファイルのrs'で指される内容を読み込む。レジスタファイルは、$2^{R_{ren}}$個のエントリを持つ。先行命令との間に依存関係があれば、それが解消するまでフェッチは待たされる。インプリメントによっては、演算結果を直接フォワーディングすることも当然考えられる。

3.3　命令実行の終了

図4に命令の実行終了およびコミット時の動作を示す。命令を実行して結果が生成されると、直ちにその命令のrd'で指されるレジスタファイルのエントリにライトする。この時に実行終了順序を意識する必要はない。1サイクルに終了できる命令数N_{cmp}はレジスタファイルのライトポート数で抑えられる。また、レジスタライトと同時にSN管理ブロックに終了した命令のSNを通知する。

3.4　命令のコミット

SN管理ブロックが各サイクルにコミット可能な命令をin-orderに決定し、それらのSNを出力する。1サイクルにコミットできる命令数をN_{com}とし、各SNをSN_{com_0}, SN_{com_1},のように表す。各SN_{com_i}を受けて、リオーダバッファ中のSN_{com_i}で指されるエントリの$< logRN >$領域の値を$logRN_{com_i}$、$< phyRN >$領域の値を$phyRN_{com_i}$として、プリサイスステイタステーブルの$logRN_{com_i}$で指されるエントリに$phyRN_{com_i}$を書き込み、同時にそれ

まで書かれていた値をフリーな番号としてRN管理ブロックに通知する。これによって、命令ストリーム上でそのサイクルにコミットした命令まで実行が進んだ段階におけるプロセサの状態が、リネーミング前後のレジスタ番号の組として、ブリサイスステイタステーブル上に保証される。

3.5 リカバリー

例外の発生はその命令のSNとともにSN管理ブロックに通知される。SN管理ブロックは、それに先行する命令が全てコミットした段階で、リカバリー処理の指示を出す。上述したように最近コミットした命令までのプロセサ状態がブリサイスステイタステーブルに保証されているので、リカバリー処理にあたっては、リオーダバッファをフラッシュするとともに、図3におけるセレクトテーブルの値を全てのレジスタについてブリサイスステイタステーブル側に向けるだけで良い。

セレクトテーブルは、立ち上げ時およびリカバー時には全ビットがブリサイスステイタステーブル側の値をとり、デスティネーションのリネーミングを一旦行なうと、対応するビットの値をフューチャファイル側に変更するように制御する。

4 テーブルサイズに関する考察

本章では、リカバリー機構を実現するために必要な各種テーブルの仕様を整理し、テーブルサイズを決定するためのいくつかのパラメタについて性能の観点から議論する。

4.1 各テーブルの仕様

文献[2]のリカバリー方式を採用し、レジスタリネーミングを行なわないモデル(M1)、同様にレジスタリネーミングを行なうモデル(M2)、我々のリカバリー方式を採用したモデル(M3)のそれぞれについて、レジスタファイル(RF)、リオーダバッファ(ROB)、フューチャファイル(FF)、ブリサイスステイタステーブル(PST)の仕様を表1にまとめた。ただし、M1, M2のリオーダバッファについては、インプリメントを考慮して、$< regN >$領域と$< val >$領域を別個のモジュールで実現することにした。なお、$< exp >$領域は無視した。また、M1とM2にはPSTは存在しないが、M2に関してはPSTと同様の構成の変換テーブルが明らかに必要なので、それをPSTの欄に記した。表中のWLは語長を意味する。また、1個の命令について2個のソースオペランドを仮定している。

これらのテーブルは全てレジスタファイル形式のSRAMで、エントリ数、エントリ長、リードポート数、ライトポート数をパラメタとして持つ。また、前章までの議論では整数レジスタと浮動小数点レジスタを特に区別しなかったが、以降では分けて扱うこととし、パラメタ上では-int, -fpなどを付加して区別する。両者のうち大きい方という意味で-maxという記述も用いる。上記のテーブルのうち、リオーダバッファ以外は全て整数用と浮動小数点用のものが存在する。

表1から、M3ではROBやFFのエントリ長が大幅に削減されていることが分かる。

4.2 シミュレーションによる評価

本節では、前節でまとめた各パラメタのうち、リネーミング空間のサイズである$2^{R_{ren}}$とシリアル番号空間のサイズであるN_{SN}の適切な値について、シミュレーション結果に基づいて考察する。シミュレーションは我々が開発したトレースベースシミュレータParatool[4]を用いて、以下のようなマシンモデル上でSPEC92ベンチマークを走らせた場合のIPCを測定した。

- 8命令フェッチ。アライン機能なし。
- 4命令issue。
- ALUとロードストアが2個、分岐ユニット, SFT, FADD, FMUL, FDIVが各1個。
- レイテンシはロードが3, FADD, FMULが4, FDIVが20, その他は全て1(サイクル)。
- BTBは128エントリ,4way, 2bit予測。
- キャッシュI,Dとも32KB, ブロックサイズ64B, 4way。

表1 各テーブルの仕様

	エントリ数			エントリ長(ビット)		
	M1	M2	M3	M1	M2	M3
RF_{int}	$2^{R_{arc-int}}$	$2^{R_{ren-int}}$	$2^{R_{ren-int}}$	WL	WL	WL
RF_{fp}	$2^{R_{arc-fp}}$	$2^{R_{ren-fp}}$	$2^{R_{ren-fp}}$	WL	WL	WL
ROB_{reg}	N_{SN}	N_{SN}	N_{SN}	R_{arc}	$R_{ren-max}$	$R_{arc}+R_{ren-max}$
ROB_{val}	N_{SN}	N_{SN}	なし	WL	WL	なし
FF_{int}	$2^{R_{arc-int}}$	$2^{R_{ren-int}}$	$2^{R_{arc-int}}$	WL	WL	$R_{ren-int}$
FF_{fp}	$2^{R_{arc-fp}}$	$2^{R_{ren-fp}}$	$2^{R_{arc-fp}}$	WL	WL	R_{ren-fp}
PST_{int}	なし	$2^{R_{arc-int}}$	$2^{R_{arc-int}}$	なし	WL	$R_{ren-int}$
PST_{fp}	なし	$2^{R_{arc-fp}}$	$2^{R_{arc-fp}}$	なし	WL	R_{ren-fp}

	リードポート数			ライトポート数		
	M1	M2	M3	M1	M2	M3
RF_{int}	$N_{dec-int}*2$	$N_{dec-int}*2$	$N_{dec-int}*2$	N_{com}	N_{com}	$N_{cmp-int}$
RF_{fp}	$N_{dec-fp}*2$	$N_{dec-fp}*2$	$N_{dec-fp}*2$	N_{com}	N_{com}	N_{cmp-fp}
ROB_{reg}	N_{com}	N_{com}	N_{com}	N_{dec}	N_{dec}	N_{dec}
ROB_{val}	N_{com}	N_{com}	なし	N_{cmp}	N_{cmp}	なし
FF_{int}	$N_{dec-int}*2$	$N_{dec-int}*2$	$N_{dec-int}*2$	$N_{cmp-int}$	$N_{cmp-int}$	$N_{dec-int}$
FF_{fp}	$N_{dec-fp}*2$	$N_{dec-fp}*2$	$N_{dec-fp}*2$	N_{cmp-fp}	N_{cmp-fp}	N_{dec-fp}
PST_{int}	なし	$N_{dec-int}*2$	$N_{dec-int}*2+N_{com}$	なし	$N_{dec-int}$	N_{com}
PST_{fp}	なし	$N_{dec-fp}*2$	$N_{dec-fp}*2+N_{com}$	なし	N_{dec-fp}	N_{com}

RF: レジスタファイル、ROB: リオーダバッファ、FF: フューチャファイル、PST: プリサイスステイタステーブル

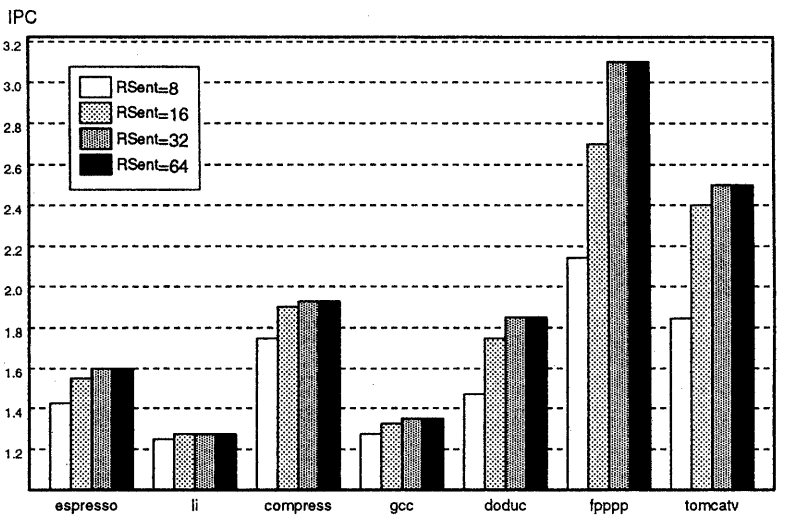

図5 RSエントリ数とIPC

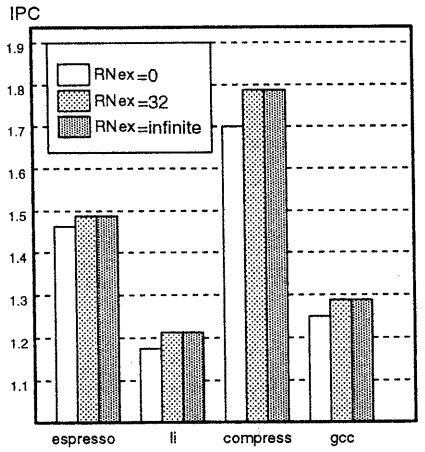

図6　リネーミングの効果(integer)

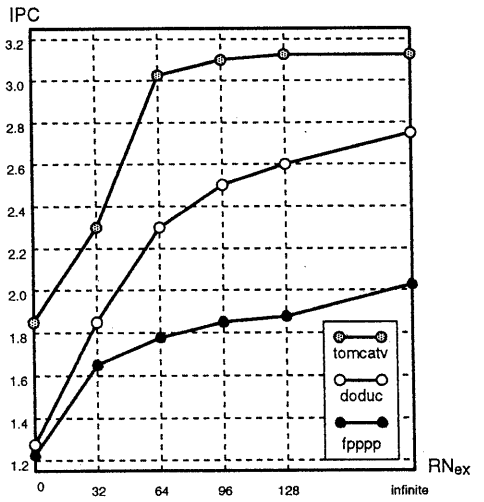

図7　リネーミングの効果(floating)

- キャッシュミスペナルティは3サイクル。(2次キャッシュを想定)

そして、評価のために次の項目を可変にした。

- リザベーションステーションのエントリ数 (RS_{ent})
- リネーミングのための余分なレジスタ数 (RN_{ex-int}, RN_{ex-fp})

この2項目を前章の説明に照らして考えれば、前者は S_{SN} の要素数とほぼ等価であり、後者は $2^{R_{ren}}$ と $2^{R_{arc}}$ の差分に相当する数である。ここで、Sparcアーキの SS2 で測定した関係上 $2^{R_{arc}}$ は 32 なので、最適な RN_{ex} の値に 32 を加えた数を $2^{R_{ren}}$ として用いることにする。

はじめに、RS_{ent} を 8, 16, 32, 64 と変化させた場合のシミュレーション結果を図5に示す (RN_{ex} は 32 に固定)。この結果から、RS_{ent} は 32 あれば性能向上は飽和すると考えられるので、S_{SN} の値として 32 を採用することにする。

次に整数ベンチマークにおいて、RN_{ex-int} を、0, 32, 無限と変化させた場合の結果を図6に示す (RS_{ent} は 32 に固定)。この結果から、RN_{ex-int} は 32 あれば充分であることが分かるので、$2^{R_{ren-int}}$ は 64 とする。

次に浮動小数点ベンチマークにおいて、RN_{ex-fp} を 0 から 32 刻みに 128 までと、更に無限にした場合についてシミュレーションした結果を図7に示す。ベンチマークによって振舞いが若干異なるが、RN_{ex-fp} は 64 ないし 96 が適当であると考えられる。リネーミング空間は2のべき乗が望ましいので、$2^{R_{ren-fp}}$ は 128 とする。

5　テーブルサイズの比較

本章では表1にまとめたテーブルのサイズを算定し、モデル毎の比較を試みる。簡単のため WL 以外のパラメタは表2に示す値に固定する。表2の中で、N_{SN}, $2^{R_{ren-int}}$, $2^{R_{ren-fp}}$ は前章の議論から導いた値であり、その他は一般的と思われる値にした。WL は、32bit アーキ、64bit アーキの双方について評価するため、32, 64 の2通りの値を与えることにする。

表2 パラメタ値

N_{SN}	32	$2^{R_{ren-int}}$	64	$2^{R_{ren-fp}}$	128
N_{dec}	4	$N_{dec-int}$	4	N_{dec-fp}	2
$N_{cmp-int}$	4	N_{cmp-fp}	2	N_{com}	4

5.1 サイズ算定方法

CMOSテクノロジを想定して、配線ピッチを$p(\mu m)$とし、リードポート数とライトポート数の合計がn、エントリ数が2^w、語長が$b(bit)$のテーブルの面積を算定する。

まず、メモリセル1ビットの面積について述べる。我々が使用するようなポート数の多いテーブルでは、トランジスタの面積よりも配線の占める面積が支配的になる。配線は縦方向にビット線がn本、横方向にワード線がn本必要である。また、異なる配線層間の橋渡しや電源等のため約n本の配線が別途必要になる。これより、1ビットのメモリセルは、横が$2np$縦がnpの矩形とする。

テーブル全体は図8のような構造であるとする。1エントリを形成するメモリセルは横方向に左右のメモリセル領域に半分ずつ並べることにする。従って、$L_1 = np * 2^w$、$L_2 = np * b$になる。

ワード線ドライバの横幅L_3、アドレスデコーダの縦幅L_4、センスアンプの縦幅L_5については、経験的に以下の近似値を用いることにする。

$L_3 = L_2$、$L_4 = nwp$、$L_5 = 50p$

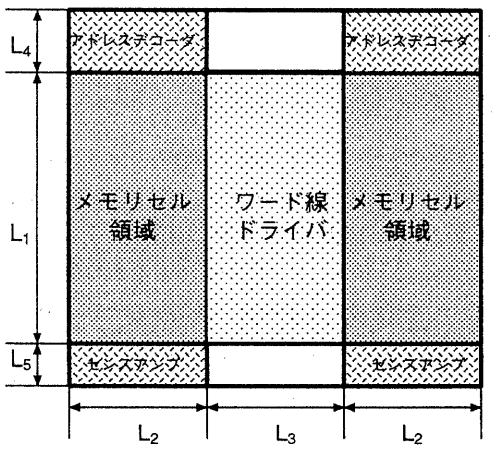

図8　テーブルの構造

5.2 算定結果と考察

上記の算定方法に基づき、各テーブルの面積を計算して、モデル毎に合計した結果を表3にまとめた。

表3 テーブルサイズの合計 (単位: mm^2)

	32ビットアーキ	64ビットアーキ
M1	$1.87p^2$	$3.69p^2$
M2	$4.46p^2$	$8.87p^2$
M3	$2.07p^2$	$3.59p^2$

テーブルサイズのみの評価であり、それ以外のランダムな制御回路を考慮に入れていないという意味で不完全な面もあるが、文献[2]の方式のままレジスタリネーミングを行なうと(M2)リカバリー機構のハードウェアが倍以上になってしまうが、我々の方式(M3)に従えばリネーミングしない場合(M1)と同等のコストで実現できることが示せたと考える。

6 おわりに

レジスタリネーミングを行なうスーパスカラプロセサにおけるハードウェアコストの少ないリカバリー方式を提案し、そこで用いるテーブル群のサイズを算定した。

本論文ではディレイに関して全く触れなかったが、ディレイは面積と並んでインプリメント時に考慮すべき重要なファクターであることは言うまでもない。実際、我々の方式ではデコードステージに行なうべき仕事が相対的に大きく、ここのディレイが全体のサイクルタイムを引き下げる可能性がある。今後はこうしたことを含めた総合的な評価を行なって、バランスのとれたプロセサの姿を追求していきたいと考えている。

参考文献

[1] M. Johnson, Superscalar Microprocessor Design, Prentice Hall, 1991.

[2] J.E. Smith and A.R. Pleszkun, "Impremention of Precise Interrupts in Pipelined Processors.", Proc. of 12th ISCA(1985), pp 36-44.

[3] W.W. Hwu and Y.N.Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", Proc. of 14th ISCA(1987), pp 18-26.

[4] 志村他, "スーパスカラプロセッサの性能評価", 計算機アーキテクチャ研究会投稿中.