Vol.2017-ARC-224 No.34
Vol.2017-SLDM-178 No.34
2017/1/25

STRAIGHTにおける投機メモリフォワーディングの実装の検討

酒井 一憲[1]　中江 哲史[1]　入江 英嗣[1]　坂井 修一[1]

概要：今日のプロセッサでは，様々な特性のコアを組み合わせることによって総合的な性能を高めることが一般的になっている．しかし，不規則な制御依存やデータ依存を持つプログラムの実行性能はシングルスレッドの処理能力に制約されてしまう．そこで，我々は新しい命令セットとその支援コンパイラを導入し，レジスタ・リネーミングを必要としない STRAIGHT アーキテクチャを開発している．STRAIGHT アーキテクチャでは，レジスタリネーミングや例外発生時の複雑な回復を必要とせず，多ポート RAM を必要とするユニットが削減されているため，スケーラビリティが高い利点がある．一方で，LSU 等のメモリアクセスに関する機構は，アドレスが動的に定まる性質上，コンパイラ支援だけでは効率化が難しい．大きなビット幅での CAM アクセスが必要な LSQ は，スケーラビリティがないため，他のボトルネックが解消された STRAIGHT ではクリティカルとなりやすく，マイクロアーキテクチャによる対策が有効である．そこで本論文では，STRAIGHT の命令セットの特徴を活かしつつ投機メモリフォワーディングを適用し SQ を省略する手法について検討する．

1. はじめに

かつて，プロセッサの性能向上のアプローチとして，半導体プロセスの微細化による利用可能なトランジスタ数の増加は ILP(Instruction Level Parallelism) 抽出に充てられてきた．しかし，ILP 向上に伴い必要とされるトランジスタ資源量と面積が指数的に増加する問題により，このアプローチは転換を余儀なくされた．そのため，現在はコア数増加などによって TLP(Thread Level Parallelism) を享受するアプローチが取られるようになっている．しかし，アムダールの法則 [1] に指摘されるように，充分に並列化が可能となっても，性能はシングルスレッドの性能に制約されてしまう．

そこで，我々はレジスタ割り当てと解放を従来の方法で行うことでレジスタリネーミングを必要としないアーキテクチャである，STRAIGHT アーキテクチャ [2]（以下 STRAIGHT と記述する）を提案している．

STRAIGHT は，ソースレジスタを命令位置の差分で指定する [3] 命令形式を用いることで，各命令の結果を保存するレジスタが一度しか書き込みされないような機構となっている．それによって，コアの性能向上の障害となっていた複雑なレジスタ・リネーミングの処理が不要となり，同時処理命令数の拡張に限らず、例外回復のペナルティの削減とスケーラビリティの向上を実現している．しかし，STRAIGHT の特性を利用して命令ウィンドウを拡張すると，in-flight な命令が増加し，ロード・ストア・キュー (LSQ) のエントリ数・ポート数が他の汎用プロセッサと比べ大きくなるため，これがクリティカルパスとならないようにアーキテクチャを工夫する必要がある．

LSQ とは，ロード/ストア命令の依存制約を守りつつ，out-of-order に実行する役割を担っている．依存による先行制約を果たすには，依存元ストア命令の発見，あるいは，メモリ・アクセス順序違反検出のための，64bit に及ぶ動的なターゲット・アドレスの比較が必須となる．ターゲット・アドレスの比較は，従来 CAM(Countent Addressable Memory) を用いた LSQ を構成することによって行われてきた．しかし，RAM/CAM の面積はポート数の 2 乗に比例して大きくなることから，in-flight なロード/ストア命令を増やすことは，面積の増加を招くことになる．このような現状から，トランジスタの微細化が進んでも，CAM ロジックの複雑さによって LSQ のスケーリングが阻害されると予想されている．[4][5]

そのため，スケーラビリティのない SQ を排除するために，投機メモリフォワーディング [6] をすべてのロードに適用する手法が提案されている．[7] 投機メモリフォワーディングとは，メモリ依存予測を用いることによって，in-flight なロード/ストア間のデータの授受にレジスタを利用する

[1] 東京大学大学院情報理工学系研究科

Vol.2017-ARC-224 No.34
Vol.2017-SLDM-178 No.34
2017/1/25

手法であり，SQ を用いることなくフォワーディングをすることが出来る．投機メモリフォワーディングを実現するためには，主に 2 つの機構が必要となる．ひとつは，高精度なメモリ依存予測器，もうひとつは軽量な検査機構である．予測されたメモリ依存関係にもとづいてフォワーディングされ実行されるが，この予測が正しいかどうかを確認する必要がある．そこで，[7] では SVW[4] (Store Vulnerability Window) とロード再実行を組み合わせることで，軽量な検査機構を実現している．当然予測を間違える確率が高いほど性能に与える影響が大きくなるため，高精度なメモリ依存予測器が望まれる．

投機メモリフォワーディングは，基本的に in-flight なロード/ストア間でのみ行われてきたが，ROB(Reorder Buffer) と物理レジスタの解放を遅らせることで，命令ウィンドウ幅以上の投機メモリフォワーディングの適用を可能とする手法が提案されている．[8] 一般的な out-of-order スーパ・スカラ・プロセッサの ROB の大きさは 192 エントリ程度であり，投機メモリフォワーディングが適用できるのは高々 192 命令前までのデータである．一方，STRAIGHT はレジスタファイルに各命令の結果が In-order に並べてあり，レジスタファイルは out-of-order での ROB を大きくしたものであると疑似的に捉えることが出来る．STRAIGHT では 1 つの命令が参照できるレジスタファイルが 1024 個存在するため，従来の out-of-order スーパ・スカラ・プロセッサに投機メモリフォワーディングを適用するのに比べ，投機メモリフォワーディングが適用できる命令の距離が 5 倍近く存在するといえる．

本稿の構成

本稿では，第 2 節で STRAIGHT アーキテクチャについて説明し，第 3 節で SQ の排除を実現するための機構について説明する．その後，第 4 節で提案手法について述べ，最後に第 5 節で本稿をまとめる．

2. STRAIGHT アーキテクチャ

2.1 レジスタ管理方式

STRAIGHT では命令の表現形式として，ソースオペランドを命令間の距離で指定する方式 (以下逆 Dualflow 形式と呼ぶ) を採用している．このようなフォーマットは逆 Dualflow アーキテクチャ [3] の中間表現で用いられている．逆 Dualflow 形式は，各命令毎にディスティネーションレジスタがひとつ割り当てられ，実行結果を格納する．そのため，レジスタファイルには In-order に命令の実行結果が並ぶことになる．**図 1** と**図 2** は STRAIGHT での命令形式とアセンブリコードを表している．ロードとストアのアドレスはそれぞれ 10 命令前に指定されているものとする．ソースオペランドとして命令間の変位を持てば，自身のレジスタ番号と変位を減算することでソースオペランドの

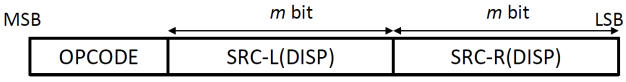

図 1 STRAIGHT アーキテクチャの命令形式

```
s = a + b + c + d;
```

```
LD [10]          #ld a
LD [10]          #ld b
LD [10]          #ld c
LD [10]          #ld d
ADD [4] [3]      # s = a + b
ADD [1] [3]      # s = s + c
ADD [1] [3]      # s = s + d
ST [1][10]       #st s
```

図 2 STRAIGHT アーキテクチャのアセンブリコード

レジスタ番号を得ることが出来る．レジスタ番号が一巡した際にレジスタに割り当てられた命令が解放し，新たな命令にそのレジスタ番号を割り当てる．そのため，レジスタファイルの大きさは，ソースオペランドとして利用できる変位と命令ウィンドウの大きさの和より大きくなくてはならない．

以上のようにレジスタの割り当てから解放までに値が一度しか書き込まれない，ライトワンス性を保証することで，逆依存と出力依存をレジスタリネーミングなしで解消している．

2.2 プロセッサの構成

STRAIGHT では RMT のようなレジスタの集中管理を行うハードウェアが存在しないため，分散管理を行い大容量のレジスタファイルを構成することができる．大容量のレジスタをコンパイラ側から扱えるようにすることによってスピルイン・スピルアウトであったり，配列へのメモリアクセスを減少させることが期待される．ただし，レジスタファイルの分散管理を行うことによって，レジスタアクセスのレイテンシは増大するため，パイプライン段数は増加することになる．

2.3 リネームロジック

STRAIGHT では各命令に 1 つレジスタが割り当てられるため，そのアドレスをリネームステージで与えられることになる．ディスティネーションレジスタ，ソースレジスタを指定するために，リネーム時に RP(Register Pointer)

という特殊レジスタを用いる．各命令は，リネームステージに到達した時の RP の値を参照し，ディスティネーションレジスタとソースレジスタを決定する．RP は読み出されるたびにインクリメントされるため，次の命令には連続したディスティネーションレジスタが与えられる．

2.4 例外からの回復

STRAIGHT には，RP の他に PC(Program Counter)，SP(Stack Pointer)，FP(Frame Pointer) といった特殊レジスタが存在する．これらの特殊レジスタは RP を用いてレジスタ割り当てをされないため，依存関係が解決できない．そのため，フロントエンド・パイプライン上で演算される．

コミット時に分岐予測ミスなどの例外が発生した場合，正しい例外を保証し，正しく命令が再実行するために各特殊レジスタの値を例外発生時に巻き戻すことが出来るようにしておく必要がある．そのため，STRAIGHT には Active List が実装されている．Active List には PC(32bit)，SP(32bit)，FP(32bit)，RP(物理レジスタ数分の bit)，Rbit(1bit)，ECode(2bit) が各命令毎に格納される．

一般的なレジスタでは，例外の回復時に Active List に記憶されたレジスタ番号を用いて RMT を巻き戻す操作が必要となり，数〜十数サイクルのペナルティがある．一方 STRAIGHT で巻き戻すべきプロセッサ状態は，単一のレジスタである特殊レジスタのみであるため 1 サイクルで回復が可能となる．

3. 投機メモリフォワーディングを用いた SQ の省略

データを生成し，メモリを介して利用する場合の一般的なデータの流れとして，値を生成する命令である Producer，その値をメモリに格納するストア，メモリから値を読み出すロード，その値を利用する Consumer という 4 つの段階が存在する．しかし，ストアとロードは，アーキテクチャの資源的な問題によって生じた余分な命令であり，これを排除できれば実行速度を早めることが出来る．そこで，投機メモリフォワーディング [6] は，in-flight なロード/ストア間のデータの授受にレジスタを利用する．レジスタリネーミングを利用することで，Producer-ストア-ロード-Consumer という一般的な流れを Producer-Consumer と短くすることを可能としている．Tingting Sha らは，この投機メモリフォワーディングをすべてのロード/ストアに実装することで SQ を排除する手法を提案した．[7]

投機メモリフォワーディングを実現するためには大きく 2 つの要素が必要となる．1 つはメモリ依存予測器，もう 1 つは検査機構である．

3.1 メモリ依存予測器

メモリ依存予測器の目標は，ロードと依存関係にあるストア (もしくはそのストアが利用する値を生成した命令) と結びつけることである．投機メモリフォワーディングにおけるメモリ依存予測器の特徴として，

(1) アドレス比較なしにストアの値をロードにフォワーディングする
(2) すべてのロードに対して依存予測を行う

が挙げられる．高精度な予測精度が求められることから，確信度カウンタやパス情報を利用して，予測した際の精度を上げている．また，予測器の学習はロードのコミット時に行われるため，メモリ順序違反検出機構はコミット時にフォワーディングされた元のストア命令が参照できる機構である必要がある．

3.2 検査機構

投機メモリフォワーディングは，投機であることからフォワーディングされたデータが本当に正しいかどうかを確認するための機構が必要となる．そのため，コミット時にロード再実行を行い，データのマッチングを取ることによってフォワーディングミスを発見することになる．このロード再実行 [9] と SVW(Store Vulnerability Window) [4] を用いたメモリ順序違反検出を組み合わせることによってロード再実行を有効に利用した軽量な検査機構を構築している．

3.3 SQ の省略

以上の機構を組み合わせることによって，SQ を省略することが可能となる．SQ を省略する利点として，レイテンシ・クリティカルな out-of-order ロード実行のパスを簡略化出来ること，ストアの out-of-order 実行を飛ばすことによってイシューキューのエントリとイシュースロットを開放できることなどが挙げられる．

また，ストアすべてとフォワーディングされると予測されたロードは out-of-order 実行をスキップしてコミットパイプライン上で実行されることになるため，コミットパイプラインにステージを追加する必要がある．

これらの発展形として，Arther Perais らは，DDT(Data Dependence Table) と CRM(Commit Rename Map) を組み合わせて距離を計測し，予測器として TAGE-like なものを提案している．また，ROB のポインタに release head を追加して，物理レジスタをフリーリストに返すのを遅らせることで，コミットされたものに対しても投機メモリフォワーディングを適用出来るようにしている．[8]

4. STRAIGHT における投機メモリフォワーディングの実装

STRAIGHT に投機メモリフォワーディングを実装するにあたって，一般的な out-of-order スーパ・スカラ・プロセッサと区別する点が幾つか存在する．まず，STRAIGHT にはレジスタリネーミングが行われないということが挙げられる．このことによって，物理レジスタと論理レジスタ

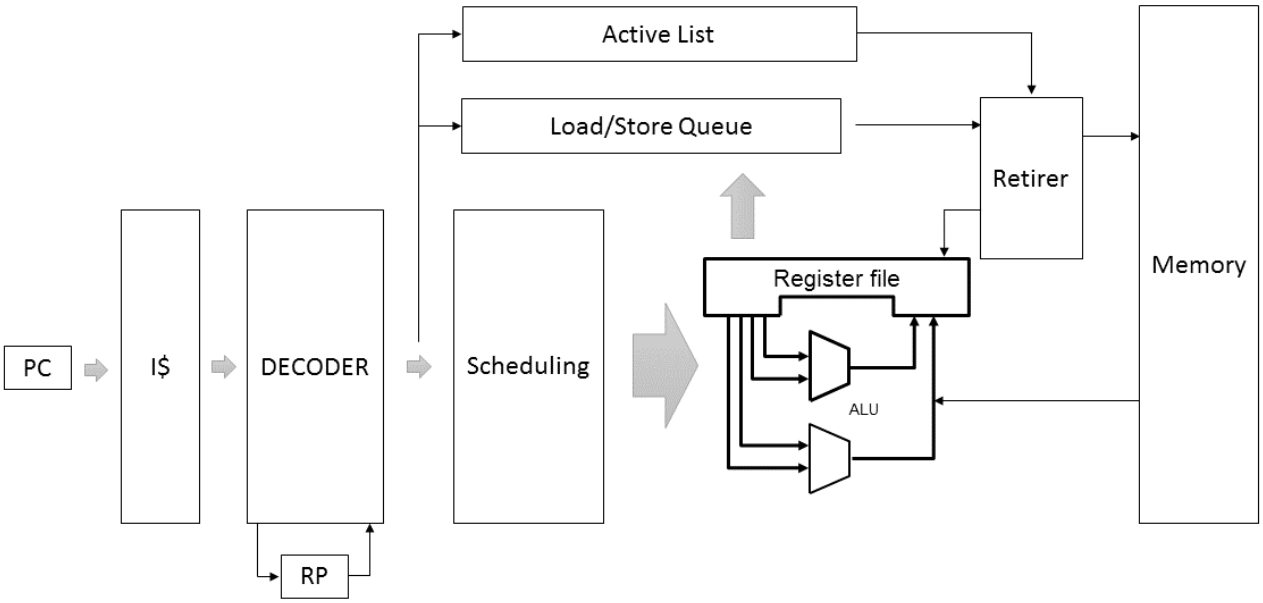

図 3　STRAIGHT のブロック図

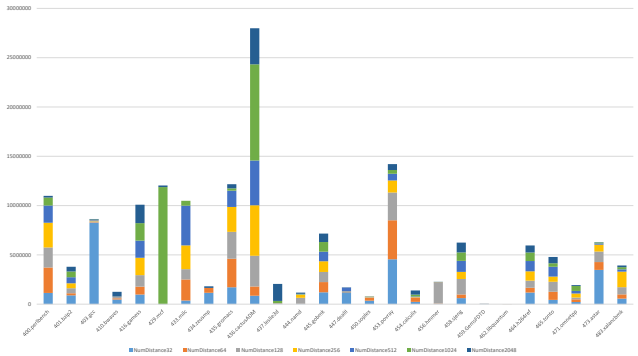

図 4　フォワーディング出来る数

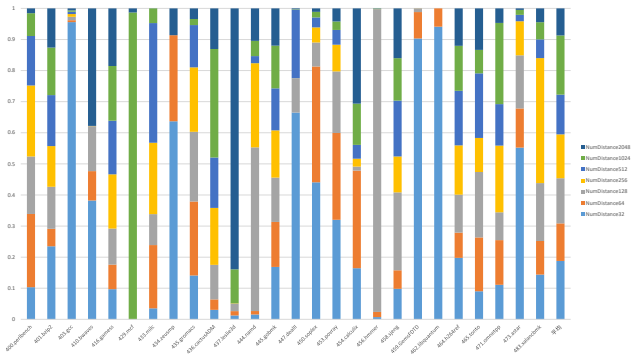

図 5　フォワーディング出来る割合

の間のマッチングをとるためのテーブルが必要なくなり，投機メモリフォワーディングの実装が容易になる．また，ソースレジスタが距離の変位で与えられることで，ループ構造でのメモリ依存関係を表しやすくなっていること．そして，ディスティネーションレジスタが各命令に 1 つずつ与えられるため，通常レジスタに書き戻さない命令であっても，ディスティネーションレジスタを利用できることが挙げられる．このように，レジスタファイルには各命令の結果が In-order に並んでいることからロード命令からの距離で依存元を指し示す方式を導入しやすい．

図 4 と**図 5** は，SPEC2006[10] を cycle-accurate なプロセッサ・シミュレータである鬼斬弐 [11] を用いて実行し，フォワーディング出来る数と割合を命令距離別に集計したものである．従来の SQ のサイズではフォワーディング不可能であった依存関係が多く存在していることが確認できる．256 命令前までのフォワーディング可能な命令に比べて，1024 命令前までのフォワーディング可能な命令は約 1.5 倍に増えていることがわかる．つまり，1.5 倍近くの命令に投機メモリフォワーディングを適用することができ，その分の性能向上も見込むことが出来る．

以上に留意して，以下では投機メモリフォワーディング実装のために追加で必要となる機構，命令の変更，そして実際の実装について述べていく．

4.1　追加する機構

投機メモリフォワーディングを実装するには高精度なメモリ依存予測器が必要となる．そのため，TAGE-like なメモリ依存予測器と，予測器を学習する機構を追加する．従来のメモリ依存予測器であれば，メモリ・アクセス順序違反が起こった場合のみ学習すれば充分であった．しかし，投機メモリフォワーディングでは積極的にメモリ依存予測器を学習したいため，すべてのロード命令に対してフォワーディングが可能であったかを検証する．そのため，フォワーディング可能な命令数分のストアの履歴をもつテーブルを持ち，ロードがコミットする際にそれを参照することで，フォワーディング可能であったかどうかを検証し，学習することが可能となる．このテーブルを SHT(Store History Table) と呼ぶことにする．SHT はアドレスをキー，RP をバリューとするテーブルである．ロードはコミットの際に

自身のターゲットアドレスを SHT に投げることで，直近の同じターゲットアドレスをもつストアの RP を得ることが出来る．その後，ロードの RP とストアの RP の差を取ることで，メモリ依存予測器の学習をすることが可能となる．

また，投機メモリフォワーディングされた命令のデータが正しいかどうかを検証する機構を追加する．今回は NoSQ で採用された SVW とロード再実行を組み合わせることで，確認機構とする．先行研究では，SVW ではなく他のフィルタを用いて，かつストアのコミット時にメモリ・アクセス順序違反を検出することで無駄な学習テーブルを排除する手法がある．しかし，今回は命令ウィンドウ幅より大きい命令間でのフォワーディングについても学習をする必要があるため，この手法は採用しない．

ロード再実行を実装するにあたって，ストアについてもコミット時に実行することにする．ロード再実行とコミット時のストア実行のために，新たにターゲットアドレスと値を格納している RP を記憶するテーブルと Free List を追加する必要がある．この RP を記憶するテーブルを LSList(Load/Store List) と呼ぶ．LSList は Active List にロード/ストアを書き込む際に Free List から値を得て，ストアとロードのソースオペランドを保持する．Active List からコミットされるときに，Active List に格納していた LSList のポインタを用いてアクセスすることで，自身のソースオペランドを得る．その後，自身の RP から除算することで，レジスタアクセスをし，その後 Free List にポインタを戻すことになる．つまり，Active List にはストアまたはロードであるかどうかを判別する 1bit と，投機メモリフォワーディングされたかどうかを表す 1bit，そして Free List のポインタ分の bit をエントリに増やす必要がある．

4.2 命令形式の変更

4.2.1 ストアの命令形式の変更

STRAIGHT では，各命令にディスティネーションレジスタが割り当てられる．しかし，ストア命令は，ターゲットアドレスと値を得ることでメモリアクセスをするだけの命令であり，本来は ALU を通す必要はない．NoSQ では，ストアのコミット時に実行することでクリティカルパスを避けることを提案している．本方式では，ストアのコミット時に実行することになるが，それに加えてストア命令をレジスタムーブ命令に変更する．ストアのディスティネーションレジスタに値を格納することによって，ロードはメモリ依存予測機が予測したストアのデータを利用することが容易になる．ストア自体はコミット時にソースレジスタを読み出し，architecture state を更新すれば良い．

4.2.2 ロードの命令形式の変更

STRAIGHT において，ロード命令はターゲットアドレスのみをソースとするため，ソースオペランドを 1 つしか持たない．メモリ依存予測器の確信度カウンタが飽和している場合はロードはフォワーディングされることになる．そのため，STRAIGHT では，フォワーディングされると予測されたロード命令はリネーム時にレジスタムーブ命令に変更するようにする．ただし，Active List にはロードとして登録し，コミット時にロード再実行を行うことによって，正当性を確認する．

また，メモリ依存予測器の確信度カウンタが飽和していない場合は，命令の変更を行われないが，予測した距離はロードのソースオペランドとして格納する．そのようにすることで，スケジューラが予測した距離の命令が解決されるまで Issue することはなくなるため，メモリ・アクセス順序違反が起こるのを未然に防ぐ効果が期待される．

4.2.3 ロード/ストア間予測とロード/プロデューサー間予測の比較

STRAIGHT ではディスティネーションレジスタが各命令に割り当てられるため，従来では複雑になりがちであった，ロード/プロデューサー間のフォワーディングが容易に実装可能となる．すなわち，ストアのソースオペランドとして値を入ったレジスタが指定されていることから，SHT の RP にそのレジスタを記録すれば良い．以上で説明してきたロード/ストア間フォワーディングとロード/プロデューサー間フォワーディングには多少の違いが存在する．まず，ロード/ストア間フォワーディングでは，すべてのロードに予測距離をソースオペランドとして与えることでメモリ・アクセス順序違反を防ぐ事ができる．しかし，ストア命令はレジスタムーブ命令として out-of-order コアに送られてしまうため，クリティカルパスの圧迫が避けられず，また，レジスタを余分に叩くことになるため，電力的に好ましいとはいえない．一方で，プロデューサー/ロード間フォワーディングはメモリ・アクセス順序違反を未然に防ぐ効果はなくなるものの，ストアを out-of-order コアに通す必要はなくなる．ただし，ストアを out-of-order コアに通さなくなることにより，SVW などのフィルタを用いた順序違反検出をするには実装を工夫する必要がある．

どちらの手法が優れているかについては，実装・評価をする必要がある．

4.3 ブロック図

以上の機構を追加したブロック図が**図 6** である．赤いユニットや線がメモリ依存予測のために追加した機構，青いユニットが投機メモリフォワーディングを適用し SQ を排除するために追加した機構である．Retirer に着目すると，ストアのコミット時の実行と，ロード再実行のためにレジスタファイルから読み出しポートが，ロード再実行時のデータ比較のためにメモリからの読み出しポートが増えていることがわかる．

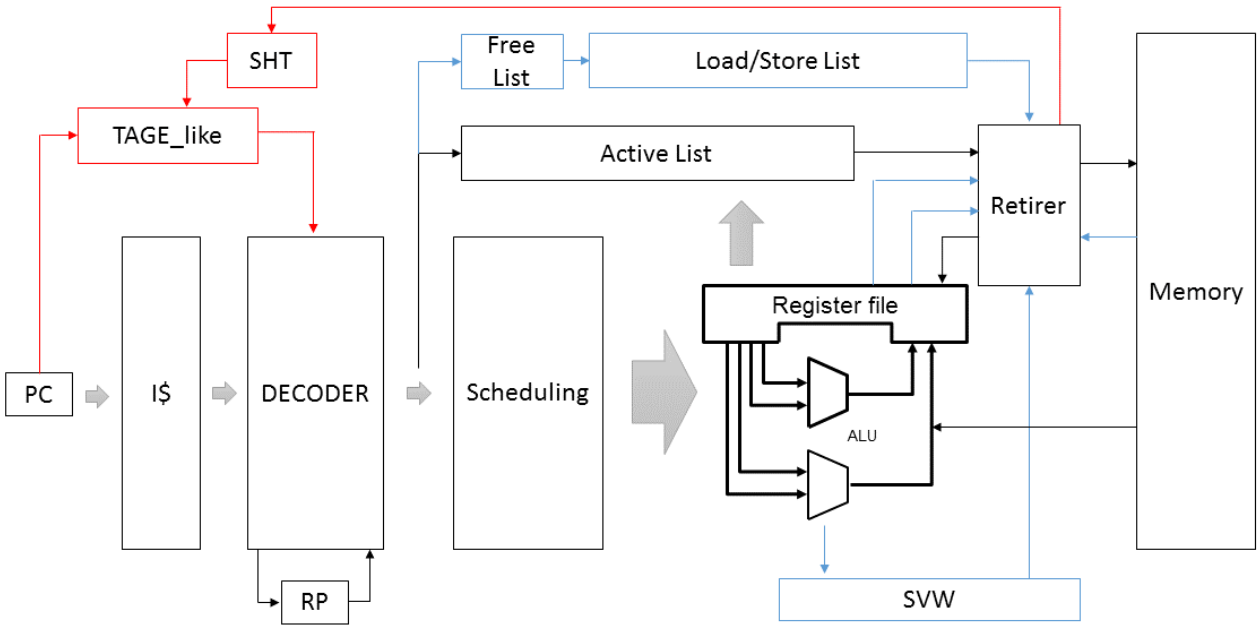

図 6　STRAIGHT に投機メモリフォワーディングを実装した場合のブロック図

5. まとめと今後の課題

本稿では，STRAIGHT に投機メモリフォワーディングを適用した場合のアーキテクチャについて検討した．STRAIGHT では，レジスタリネーミングが省略されていること，そしてレジスタファイルに命令順に値が格納されていることから，距離を用いたメモリ依存予測を取り入れることが容易であることがわかった．また，一般的なプロセッサでは投機メモリフォワーディングが参照できる命令の距離は高々 ROB のサイズであるのに対し，STRAIGHT では，命令が指定できる変位分の距離を参照し，フォワーディングすることができる．

今後の課題

今後の課題としては以下のことが挙げられる：

- TAGE-like メモリ依存予測器の実装と評価．
- SVW 以外のフィルタとの比較検討．
- プロデューサー/ロード間フォワーディングと，ロード/ストア間フォワーディングの性能の比較検討．
- 面積評価を行う．

謝辞

本論文の研究は一部，文部科学省科学研究費補助金 No. 25730028 による．

参考文献

[1] Amdahl, G. M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), New York, NY, USA, ACM, pp. 483–485 (online), DOI: 10.1145/1465482.1465560 (1967).

[2] 入江英嗣，山中崇弘，佐保田誠，吉見真聡，　吉永努：もし ILP プロセッサのレジスタファイルが分散キーバリューストアになったら，研究報告計算機アーキテクチャ（ARC），Vol. 2013, No. 5, pp. 1–10（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110009587847/⟩ (2013).

[3] 一林宏憲，英嗣入江，正裕五島，修一坂井：逆 Dualflow アーキテクチャ，Vol. 2006, No. 88, 一般社団法人情報処理学会，pp. 37–42（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110004824125/⟩ (2006).

[4] Roth, A.: Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 458–468 (2005).

[5] Castro, F., Pinuel, L., Chaver, D., Prieto, M., Huang, M. and Tirado, F.: Using age registers for a simple load-store queue filtering, *Journal of Systems Architecture: the EUROMICRO Journal*, pp. 79–89 (2009).

[6] Tyson, G. S. and Austin, T. M.: Improving the accuracy and performance of memory communication through renaming, *Proceedings of 30th Annual International Symposium on Microarchitecture*, pp. 218–227 (online), DOI: 10.1109/MICRO.1997.645812 (1997).

[7] Sha, T., Martin, M. M. K. and Roth, A.: NoSQ: Store-Load Communication Without a Store Queue, *39th International Symposium on Microarchitecture (MICRO'06)*, pp. 285–296 (2006).

[8] Perais, A. and Seznec, A.: Cost effective physical register sharing, *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 694–706 (online), DOI: 10.1109/HPCA.2016.7446105 (2016).

[9] Cain, H. W. and Lipasti, M. H.: Memory ordering: A value-based approach, *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, IEEE, pp. 90–101 (2004).

[10] The Standard Performance Evaluation Corporation: SPEC CPU2006 suite.

[11] 塩谷亮太，五島正裕，坂井修一：プロセッサ・シミュレータ「鬼斬弐」の設計と実装，先進的計算基盤システムシンポジウム SACSIS，pp. 120–121 (2009).