Vol.2010-ARC-188 No.3
2010/3/1

CoreSymphony アーキテクチャのための物理レジスタ管理手法

若杉祐太[†1] 坂口嘉一[†2]
三好健文[†1,†3] 吉瀬謙二[†1]

CMP の逐次性能の向上，および逐次性能と並列性能のバランシングを目的として，スーパースカラの協調動作を実現する CoreSymphony アーキテクチャを提案している．CoreSymphony は発行幅の狭いスーパースカラを複数個協調動作させることで，発行幅の広いスーパースカラを仮想的に形成する技術である．

本稿では，CoreSymphony の実装を効率化するために，(1)2-way リネーミングと (2)CoreSymphony 向け物理レジスタ分散手法の 2 つの要素技術を提案する．(1) は RMT の複雑度を緩和する．(2) は物理レジスタのエントリ分散を実現する．評価の結果，これらの要素技術により，性能にほとんど影響を与えずに RMT と物理レジスタの HW 複雑度を軽減できることが分かった．

An Efficient Physical Register Management Scheme for CoreSymphony Architecture

Yuhta Wakasugi,[†1] Yoshito Sakaguchi,[†2]
Takefumi Miyoshi[†1,†3] and Kenji Kise[†1]

We previously proposed CoreSymphony, a cooperative superscalar processor architecture to improve sequential performance in Chip Multi-Processors. CoreSymphony enables some narrow-issue cores to fuse into one wide-issue core.

In this paper, we propose two techniques to improve hardware complexity of CoreSymphony. 2-way renaming reduces the number of read/write ports of Register Map Table. Physical register distribution scheme for CoreSymphony realizes decentralized Physical Register File. Our evaluation results show that these techniques improve hardware complexity of CoreSymphony with low performance overhead.

1. はじめに

プロセッサの処理性能および電力効率の向上を目指し，CMP(Chip Multi-Processor) が広く普及している．CMP はスレッドレベルの並列性を利用し，複数のスレッドを複数のコアで並列に実行することで性能向上を得る．半導体技術の持続的な進歩により，チップ当たりに集積されるコア数は今後も増加する見通しである．

並列処理の普及と並列度のさらなる向上は，CMP に 2 つの重要な課題を提起する．(1) プログラム中の並列化できない処理 (逐次処理) が並列プログラム全体の性能を制限すること，(2) 逐次性能と並列性能を両立すること，の 2 点である．(1) は深刻な問題である[1)]．仮に，プログラムの 90%が並列化可能であったとする．この場合，たとえ 100 コアを使用して並列処理したとしても，1 コア時に対する性能向上は僅か 10 倍程度である．(2) は CMP の設計方針に関する問題である．高速で面積の大きいコアを少数搭載するアプローチと，低速だが軽量なコアを多数搭載するアプローチにはそれぞれ得失が存在する．前者は並列性能，後者は逐次性能の低下が問題となる．

我々は，複数個のコアを協調動作させることで，1 つの逐次性能の高いコアを仮想的に形成する CoreSymphony アーキテクチャ[2),3)] を提案している．CoreSymphony は，複数個の発行幅の狭いスーパースカラプロセッサを，1 つの発行幅の広いスーパースカラとして動作させることを可能にするアーキテクチャ技術である．協調動作するコア数は動的に変更することができる．そのため，本手法を実装する CMP は (1) 並列プログラム中の逐次処理の高速化，(2) 逐次性能と並列性能の両立，という CMP の課題を解決することができる．

本稿では，我々が過去に提案した CoreSymphony ver.0.2[3)] において問題であった (a)RMT(Register Map Table) の複雑化，(b) 物理レジスタのエントリ数/ポート数の増大に取り組む．(a) に対しては，旧来の多ポートの RMT を 2 種類の軽量な RMT に置き換える，**2-way リネーミング**を提案する．(b) に対しては，不要なレジスタ・コピーをフィルタリングすることで物理レジスタの分散を実現する，**CoreSymphony 向け物理レジスタ分散手法**を提案する．そして，これらの要素技術をまとめた **CoreSymphony ver.0.3** を

†1 東京工業大学 大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†2 東京工業大学 工学部情報工学科
Department of Computer Science, Tokyo Institute of Technology
†3 独立行政法人 科学技術振興機構 CREST
CREST, Japan Science and Technology Agency

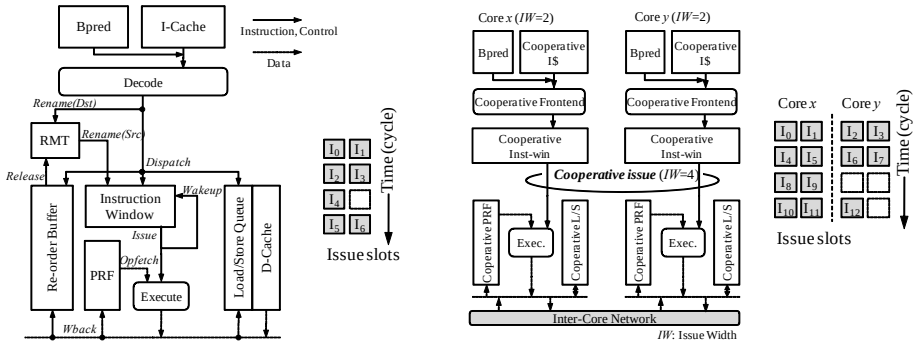

図 1 ベースのスーパースカラのブロック図 (左)．協調動作を実現するスーパースカラの概念図 (右)．

定義する．

本稿の構成を述べる．2 章では CoreSymphony アーキテクチャの実現に向けたこれまでの試みと，その問題点についてまとめる．3 章と 4 章では，本稿で新たに提案する 2-way リネーミングと CoreSymphony 向け物理レジスタ分散手法について詳しく述べる．5 章ではこれまでに提案した要素技術をまとめ，CoreSymphony ver.0.3 を定義する．6 章では提案手法の性能を評価し，最後に 7 章で本稿をまとめる．

2. 協調可能スーパースカラの実現に向けたこれまでの試み

本章では，これまでの試みである CoreSymphony ver.0.2 の実装を簡潔にまとめる．

2.1 ベースのスーパースカラの構成

まず，議論のベースとするスーパースカラの構成について述べる．**図 1**(左) は対象とするスーパースカラのブロック図である．構成方式として，物理レジスタファイル (PRF) をバックエンドで読み出す方式のアウトオブオーダを採用する．レジスタリネーミングのための RMT，コミットのインオーダ性を確保するための Re-order Buffer(ROB) を備える．命令供給部には標準的な分岐予測器 (Bpred) を備えるものとする．発行幅は，INT 2 命令/cycle，FP 1 命令/cycle 程度を想定する．

2.2 目標とする協調可能スーパースカラの構成

次に，本研究が目標とする協調可能スーパースカラについて概説する．図 1(右) は協調動作を実現するスーパースカラの概念図である．複数個の 2 命令発行スーパースカラが協調動作し，あたかも 1 つの発行幅の広いコアのようにふるまう．図 1(右) は 2 コアの協調により 4 命令発行を実現する例である．一次キャッシュや命令ウインドウといった各コアに属するモジュールは，アーキテクチャ技術により協調性を備え，協調動作によってスレッドあたりのエントリ数を増加させる．フロントエンドはコアのレベルまで完全に分割され，クラスタ型アーキテクチャのように制御の集中化や密な通信をおこなわない．コア間のデータの授受は，主に各コアのバックエンドを接続するコア間ネットワークを用いておこなう．

2.3 CoreSymphony ver.0.2 のアプローチ

本節では，CoreSymphony のこれまでの実装について述べる．その際には，各ステージの課題を明確にし，それに対し我々がどのようなアプローチをとったかを示す．また，これまでのアプローチに問題点がある場合には，それを明らかにする．

本節では，発行幅の広いスーパースカラを複数個の発行幅の狭いスーパースカラに分割するという視点から説明をおこなう場合がある．

2.3.1 命令フェッチ

フェッチステージにおける課題は，(a) 分散可能命令キャッシュを構築すること．(b) データ依存関係を解決するために十分な情報を後段に供給すること．(c) 協調中の全コアの制御フローを同期させることである．

(a)，(b) に対して，我々は**ローカル命令キャッシュ**[2),3)] を提案している．CoreSymphony では，協調中のコア数 ×4 命令を最大長とする命令トレースを 1 単位としてフェッチ/ステアリングをおこなう．この命令トレースを**フェッチブロック**(以降 FB と表記する) と呼ぶ．ローカル命令キャッシュは，FB 中の自コアにステアリングされた命令と，FB 間の依存関係の解決に用いる制御情報を格納するステアリング済み命令トレースキャッシュである．制御情報には，**Destination vector** と呼ぶ，論理レジスタ数と同長のベクトルが含まれる．これは，FB 中の命令のデスティネーションレジスタの位置を示すベクトルである．Destination vector の第 n ビットは，論理レジスタ R_n に結果を書き込む命令が当該 FB 中に存在するか否かを示す．ローカル命令キャッシュの実装は文献 3) に詳しい[⋆1]．

(c) に関しては，投機ミスや例外に関する処理を全コアで多重化して実行することで対処する．

2.3.2 命令ステアリング

命令ステアリングは，クラスタ型アーキテクチャにおいて，どの命令をどのクラスタで実

⋆1 ただし，文献 3) では，Destination vector は未実装である．Destination vector はローカル命令キャッシュにミスした場合に，デコードステージで生成される．

行するかを決定する操作である．本ステージにおける課題は，(a)Wakeup レイテンシ短縮のために依存関係にある命令をまとめること，(b)Select レイテンシ短縮のために各クラスタの負荷を均等にすることである．

CoreSymphony の場合，ステアリング先はクラスタではなくコアとなる．また，ステアリング済み命令トレースキャッシュの採用により，あるトレースのステアリング結果は毎回同じであるという制約が加わる．よって，ステアリングアルゴリズムはより一層重要となる．このため，我々は**リーフノードステアリング**[3] と呼ぶ CoreSymphony 向けステアリングアルゴリズムを提案している．これは，複数の命令の依存元となる命令を複数のコアに重複してステアリングすることで，(a) と (b) の要求を同時に満たすアルゴリズムである．さらに，重要な性質として，本アルゴリズムは **FB 内のコア間通信を一切発生させない**．この性質は，3 章で述べる RMT の軽量化に必要不可欠である．リーフノードステアリングの実装は文献 3) に詳しい．

2.3.3 レジスタリネーミング

リネームステージにおける課題は RMT の分割である．一般的な RMT は，命令間の依存関係を表すタグを一元管理するという性質上，集中化を必要とする．そのため，協調可能スーパースカラにとって RMT の分割は非常に困難な課題となる．

我々の過去の実装では，デスティネーションの登録と解放を全コアで重複させておこなう[⋆1]ことで，全コアの RMT の内容を同一に保つ方式を使用していた．そのため，2 命令発行 ×4 コアの協調をサポートするためには，RMT に 12R8W ものポート数を必要とした．

この問題に対し，本稿ではコア内の依存とコア間の依存を異なるタグを用いて表現し，別々のテーブルで管理することで RMT の軽量化を図る 2-way リネーミングを 3 章で提案する．

2.3.4 物理レジスタの構成

物理レジスタファイルはポート数，エントリ数ともに大きく遅延が大きい．そのため，各コアが保有する物理レジスタを全体のサブセットとし，ポート数，エントリ数を抑える必要がある．

フロントエンドを完全に分割する CoreSymphony では，それぞれの命令の結果を明示的に各コアに分散することは難しい．そのため，あるコアで生成された値は全コアで共有する結果バスにブロードキャストする．我々の過去の実装では，全コアの物理レジスタの内容を同一に保つために，各コアは結果バスから全ての結果を自身の物理レジスタに取り込んでいた．このため，物理レジスタは 8-way 相当のエントリ数と非常に多いポート数を必要とした．

この問題に対し，本稿では (1) 結果バスに放送する値のフィルタリング，(2) 結果バスから物理レジスタに取り込む値のフィルタリング，(3) 書き込み時に物理レジスタのエントリを束縛することで必要最低限のエントリを確保する機構，を組み合わせた CoreSymphony 向け物理レジスタ分散手法を 4 章で提案する．これにより，物理レジスタのエントリ数/ポート数を現実的な範囲に抑える事を目指す．

2.3.5 ロード/ストアユニット

ロード/ストアユニットにおける課題は，データキャッシュと LSQ の分割である．データキャッシュおよび LSQ を実効アドレスでバンク分けするアプローチ[4),5)] は，LSQ 内のストアデータフォワーディングおよび，メモリあいまい性除去を局所化できるという点で都合がよい．しかし，各命令がアクセスするバンクは実効アドレスの計算後に判明する．このため，ステアリング時にメモリバンク予測[6)] をおこなう手法が採用されることが多いが，バンク予測ミスの取り扱いが煩雑になるという欠点がある．

CoreSymphony では Simha らによって提案された Unordered Late-Binding Load/Store Queue(**ULB-LSQ**)[7)] を用いる．この LSQ はエントリの確保を実効アドレスの計算後におこなうことができる．よって，バンク予測をおこなう必要がない．実効アドレスの計算後に，当該バンクを有するコアの ULB-LSQ のエントリを確保し命令をディスパッチする．リモートのコアへロード/ストア命令をディスパッチするためのコア間ネットワークを用意する．

2.3.6 投機ミスの処理

投機ミスの処理における課題は，(a) 投機ミスおよび例外に関する情報を共有すること，(b) 投機ミスおよび例外からの回復のためのインオーダステート[⋆2]を分散して管理することである．

(a) に関しては，CoreSymphony では分岐予測の正否等の投機ミスに関する情報，および発生した例外の種類を全コアの ROB に重複して保存することで対処する．(b) のインオーダステートの分散管理に関しては，検討不十分のため本稿では実装を見送る．暫定的な実装として，物理レジスタファイルとは別に，確定済みの値を格納する論理レジスタファイル (Logical Register File:LRF) を用意する．コミット時に物理レジスタファイルを読み出し，全コアの論理レジスタファイルに値をコピーする．すなわち，インオーダステートは全コア

⋆1 リネームのための読み出しは分散される．

⋆2 非投機状態にある命令の，各論理レジスタに対する最新の代入操作によって構成されるステート[8)]．

が重複して保有するものとする．インオーダステートの分散は今後の重要な課題とし，本稿には含まない．

3. 2-way リネーミングの提案

レジスタリネーミングの主要ハードウェアである RMT は，非常にポート数が多く遅延が大きい．本章では，協調時の発行幅の広いスーパースカラが必要とする多ポートの RMT を，複数の小規模な RMT に置き換えるための 2-way リネーミングを提案する．

3.1 基本のアイデア

リネームステージの役割は，(1) 論理レジスタ番号から物理レジスタ番号への対応を与えること，(2) ある命令が依存する命令を表すタグを与えること，の 2 つである．一般的なアウトオブオーダでは，タグ＝物理レジスタ番号であるため，(1) と (2) は同義である．ここで，(2) のタグはその性質上グローバルな管理を必要とする．これにより，アウトオブオーダにおける RMT は集中化の必要が生じる．集中化はポート数が爆発する原因となる．しかし，(1) と (2) の役割は独立しているため，必ずしもタグ＝物理レジスタ番号でなくともよい．2-way リネーミングはこの観察に基づく．

2-way リネーミングの本質は，コア内の依存とコア間の依存を異なるタグで表現し，別々のテーブルで管理することにある．グローバルに管理する必要があるのはコア間の依存のみである．そこで，コア間の依存に関しては物理レジスタ番号をタグとして用いず，より圧縮された情報を用いる．グローバルな情報を管理するテーブルは全コアでコピーをもつ必要があるが，圧縮された情報を用いるため，ポート数を抑える効果が期待できる．ここで，コア内の依存を表すタグを Local tag(**Ltag**)，コア間の依存を表すタグを Global tag(**Gtag**) と呼ぶ．あるリネーム前の論理レジスタ R_n に対して Ltag，Gtag がとりうる値の集合は次のようになる．

$$Ltag: \quad \{P_m : m = 0, 1, ..., NP-1\}$$
$$Gtag: \quad \{R_{n\_t} : t = 0, 1, ..., NF\}$$

NP は物理レジスタの総数を，NF は FB 番号の最大値を表す．FB 番号とは in-flight な FB にサイクリックに割り当てられる 4 ビット程度のタグである．すなわち，FB_t[⋆1]中の命令のソースレジスタが R_n である場合，R_n の Gtag は R_{n_t} で与えられる．Gtag は依存先の命令を特定せず，依存先のフェッチブロックのみを特定する．これは，コア間にまたがる依存関係を表現するための情報として十分である．なぜならば，ある FB にとってコア間通信を発生させる可能性があるデスティネーションレジスタは，各論理レジスタについて高々1 個であるためである．このことは，我々の提案するリーフノードステアリングが，当該 FB 内で依存関係にある 2 命令を必ず同じコアにステアリングすることによって保証される．

⋆1 FB 番号が t である FB を FB_t と表す．

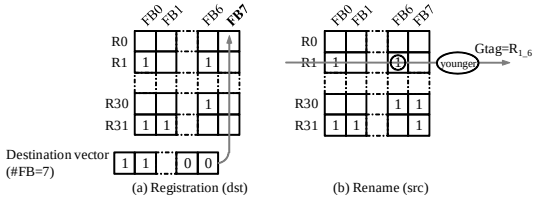

図 2 GRMT の動作．(a) デスティネーションの登録．(b) ソースのリネーム．

3.2 2-way リネーミングの実装

Ltag と Gtag は別々のテーブルで管理される．Ltag を管理するテーブルを Local RMT(**LRMT**)，Gtag を管理するテーブルを Global RMT(**GRMT**) と呼ぶ．LRMT は通常の RMT に相当するハードウェアである．よって，LRMT のポート数は基本的には 2-way アウトオブオーダの RMT に準じる．ただし，各エントリを { 物理レジスタ番号，生産者の FB 番号 } に拡張する．

一方，GRMT の構成は少々特殊である．ISA が規定する論理レジスタの本数を N，パイプライン中に保持可能な最大 FB 数を M とすると，GRMT は $N \times M$ のビット行列として構成される．GRMT の i 行 j 列のビットは，FB_j が，論理レジスタ R_i をデスティネーションとする命令を保持するか否かを表す．

GRMT によるリネームは通常の RMT と同様，(a) デスティネーションの登録，(b) ソースのリネームの 2 ステップに分けられる．**図 2** に GRMT の各ステップの動作を示す．(a) では各 FB に付随する Destination vector を GRMT の FB 番号列に書き込む．Destination Vector は各コアのローカル命令キャッシュにコピーが保存されているため，GRMT は全コアで同一に保たれる．(b) では LRMT と同様に論理レジスタ番号で GRMT の行を読みだす．GRMT の第 x 行は，論理レジスタ R_x をデスティネーションとする命令を保持する FB の FB 番号を与える．ここから図 2 中 younger で示すロジックにより，最も若い FB 番

号 Fy を求める．結果，Gtag=R_{x_y} としてタグ付けが完了する．以上まとめると，GRMT のポート数は (a) デスティネーションの登録に 1 ポート (列方向の Write)，(b) ソースのリネームに 4 ポート (行方向の Read) となり，比較的少ないポート数の RAM で構成できる．

2-way リネーミングについてまとめる．本手法ではソースのリネーム時に Ltag と Gtag の 2 種類のタグが生成される．ただし，実際にスケジューリングに使用するのはどちらか一方である．LRMT によるリネーム時に Ltag と同時に読み出した FB 番号と，Gtag に含まれる FB 番号を比較し，FB 番号が若い (プログラム順で新しい) 方のタグを採用する．採用されなかった方のタグは invalid とする．

3.3 2-way リネーミング向け命令ウインドウ

Ltag と Gtag の 2 種類のタグに対応するため，命令ウインドウに変更を加える．命令ウインドウの構成方式として，CAM による連想方式を用いる．**図 3** に，CoreSymphony の命令ウインドウの 1 エントリを示す．命令ウインドウのスケジューラ部は次のエントリをもつ．左オペランドのフィールドを添え字 l，右オペランドのフィールドを添え字 r で区別する．

- LR_l/LR_r
  Ltag フィールドが Rdy あるいは invalid であることを示す 1bit のフィールド．
- $Ltag_l/Ltag_r$
  Ltag フィールド．CAM で構成される．
- GR_l/GR_r
  Gtag フィールドが Rdy あるいは invalid であることを示す 1bit のフィールド．
- $Gtag_l/Gtag_r$
  Gtag フィールド．CAM で構成される．

Ltag フィールドは自コアで実行された先行命令によって wakeup される．Gtag フィールドは他コアで実行された命令によって wakeup される．LR_l，LR_r，GR_l，GR_r の 4 つのフィールドが全て 1 になる時，エントリは発行可能となる．

3.4 2-way リネーミングの関連研究

2-way リネーミングは，FB 間の依存を，(1) のようなタグで表現することで RMT の軽量化を図る．ここでは，同様の表現方法を採用するいくつかの研究についてまとめる．

$$Gtag = \{\text{論理レジスタ番号, suffix(FB 番号)}\} \quad (1)$$

Hopkins ら[9] は，ILP プロセッサの軽量化と性能向上を目的として DIF(Dynamic Instruction Format) という手法を提案している．この手法は，狭帯域のアウトオブオーダ発行エンジンで命令グループの依存を解析し，スケジュールが済んだ状態で DIF キャッシュという特別な命令キャッシュに保存する．DIF キャッシュにヒットする場合は，スケジュール済みの命令グループが得られるので，広帯域な VLIW 発行エンジンで高速に実行する．この手法には，命令グループ間で物理レジスタのマッピングが異なるため，通常のリネーミングではグループ間の依存が正しく表せないという問題がある．これに対し，Hopkins らは，命令のタグを (1) と同様のフォーマットで表現する方法を提案している．

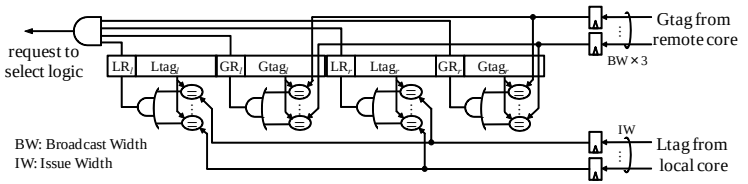

図 3 2-way リネーミング向け命令ウインドウのウェイクアップ部．

Talpes ら[10] も ILP プロセッサの低消費電力化を目的とし，類似する手法を提案している．

これらの文献では，物理レジスタの構成そのものを命令タグのフォーマットに合わせて変更している．物理レジスタは，各論理レジスタに対して数個のプールをもつ構成となる．この実装は，物理レジスタの利用効率を著しく低下させ，性能低下を引き起こす．

一方，2-way リネーミングでは性能低下の問題は発生しない．2-way リネーミングにおける Gtag は単なるタグであり，物理レジスタ番号ではない．Gtag のとり得る空間は，物理レジスタの構成とは無関係に拡大することができる．物理レジスタの構成も従来と同様のものを採用できるため，2-way リネーミングは性能低下の直接的な要因にはならない．その点において，タグと物理レジスタ番号を分けて考える 2-way リネーミングは，より進んだ実装であるといえる．

4. CoreSymphony 向け物理レジスタ分散手法の提案

CoreSymphony において，あるコアで実行された命令の結果およびタグは協調動作中の全コアにブロードキャストされる．しかし，他コアからの結果を全て自コアの物理レジスタに格納することは，物理レジスタのポート数やエントリ数の面から現実的でない．本節では，この問題に対処する CoreSymphony 向け物理レジスタ分散手法について述べる．

4.1 コア間のオペランド通信とそのフィルタリング

まず，CoreSymphony におけるコア間のオペランド通信について述べる．コア間のオペ

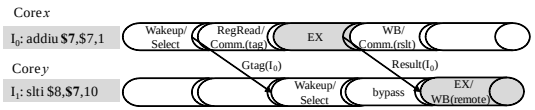

図 4 コア間のオペランド通信のタイミング.

ランド通信には，ブロードキャストモデルを採用する．**図 4** はコア x で実行された命令 I_0 の結果を，コア y の命令 I_1 が利用する場合のパイプラインの動作である．オペランド通信は 2 フェーズに分けられる．まず，コア x では I_0 が発行されるタイミングで I_0 の Gtag をブロードキャストする．その後，I_0 の実行完了と同時に結果をブロードキャストする．一方コア y では，先に放送されてきたタグによって I_1 のウェイクアップがおこなわれる．ここで，I_1 がセレクトされた場合には，続けて到着する I_0 の結果をバイパスにより取り込んで実行ステージへ移行する．

一般に，ブロードキャストモデルはハードウェアの複雑度に与える影響が大きい．1 サイクルあたりに他コアから到来する Gtag の数は，命令ウインドウの Gtag フィールドを構成する CAM のサーチポート数に直結し，物理レジスタの書き込みポート数にも影響する．そのため，CoreSymphony では，各コアが 1 サイクルあたりに発生させるブロードキャストの数 (**Broadcast Width:***BW*) を制限する．*BW* は設計パラメータである．*BW* を小さくすることは，HW の複雑度を抑える一方で，性能低下につながる．そこで，CoreSymphony は次の場合にブロードキャストをフィルタリングする．

(1) 当該命令が結果を生成しない場合．(ex. 分岐命令，ストア命令)
(2) 当該命令が生成する結果が，同 FB 中の同じデスティネーションをもつ他の命令によって上書きされる場合[⋆1]．

これらの情報は事前に解析され，broadcast flag としてローカル命令キャッシュに保存されている．スケジューラの Select ロジックは，このフラグを加味して，ブロードキャストを発生させる命令がサイクルあたり *BW* 個以下になるように発行命令を選択する．*BW* が性能に与える影響は 6 章で評価する．

4.2 物理レジスタへの書き込みフィルタリング

$BW = 1$ としても，他コアから到来するオペランドは，4 コア協調時で最大 3 命令/cycle にもなる．そこで，物理レジスタへの書き込みをフィルタリングする．他コアで実行された命令の結果を，自コアの物理レジスタに書き込む必要があるのは次の場合に限定される．

(1) 自コアの命令ウインドウ中に存在する命令が，その結果を利用する場合.
(2) これから自コアにディスパッチされる命令が，その結果を利用する場合.

すなわち，これらに当てはまらない場合は物理レジスタに書き込む必要はない．CoreSymphony では，これらに当てはまらないブロードキャストを検出し，物理レジスタへの書き込みをフィルタリングすることで，物理レジスタのポート数及びエントリ数を現実的な範囲に抑えることを目指す.

まず，(1) の検出方法について示す．これは，先行して放送されてきた Gtag に依存する命令が命令ウインドウ中に存在するか否かを調べることで検出する．このために，命令ウインドウのウェイクアップロジックを一部変更する．**図 5** に変更を加えた命令ウインドウのウェイクアップ部を示す．ソースの Gtag を格納する CAM のマッチ線を引き出し，全てのエントリについて OR をとることで，マッチするエントリ (=放送されてきた命令の結果を利用するエントリ) の有無を検出する.

次に，(2) の検出方法について示す．これは，フィルタリング対象の命令がアーキテクチャステートに含まれるか否かを調べることで検出できる．なぜならば，既にアーキテクチャステートでない結果は，投機ミスが起こらない限り，今後ディスパッチされる命令によって利用されることはないためである．対象命令がアーキテクチャステートを生成するか否かは，GRMT を利用することで容易に検出できる．対象命令のデスティネーションの論理レジスタ番号で GRMT の行を読み出すことで，その論理レジスタに対してアーキテクチャステートを与える FB 番号 FB_x を知ることができる．FB_x と対象命令の FB 番号を比較することで，対象命令がアーキテクチャステートか否かを検出できる．この機構のために，GRMT に $BW \times (NC - 1)$ と同数の読み出しポートを追加する必要がある．*BW* は各コ

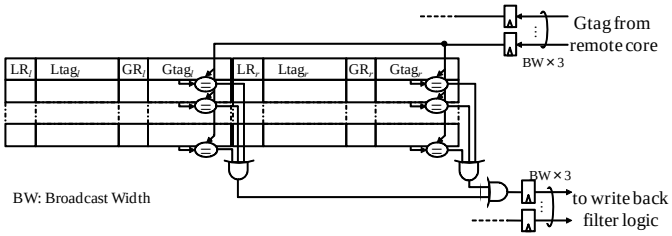

図 5 リモートの命令に依存する命令を検出するために変更を加えた命令ウインドウのウェイクアップ部.

⋆1 リーフノードステアリングは FB 内のコア間通信を発生させないため，上書きされる命令の結果が他コアで利用されることはない.

アのブロードキャスト幅，NC は協調をサポートする最大のコア数である．

リモートからの物理レジスタの書き込みについてまとめる．物理レジスタへの書き込みは (1) と (2) の検出によってフィルタリングされる．物理レジスタのリモート用の書き込みポートの数を **Remote writeback Width**(RW) と表す．書き込みが許諾された命令は，物理レジスタのエントリを確保し，LRMT にマッピングを登録する．そのため，LRMT に RW と同数の書き込みポートを追加する．RW は設計パラメータである．RW が性能に与える影響は 6 章で評価する．

4.3 リモートのライトバックによるデッドロックの発生と対処

リモートの命令が自コアの物理レジスタに値を書き込む場合，その命令は自コアの物理レジスタをアウトオブオーダに束縛する．物理レジスタのアウトオブオーダな束縛はデッドロックという問題を引き起こす[11]．デッドロックは，書き込みを発生するリモートの命令 I_x が in-flight な命令中で最も古く，かつ物理レジスタが枯渇している場合に発生する．なぜならば，I_x に物理レジスタが割り当てられない限り，命令の実行は進まず物理レジスタが解放されることもないためである．

CoreSymphony は非常にシンプルな方式でこの問題を回避する．CoreSymphony はリモート命令の物理レジスタの束縛用に，あらかじめ物理レジスタのエントリを確保しておく．確保する物理レジスタの数を **Number of Reserved Physical register** (NRP) と呼ぶ．NRP は設計パラメータである．物理レジスタの空きエントリ数が NRP を下回った時はフロントエンドをストールさせ，物理レジスタの束縛を止める．NRP によって物理レジスタの空きを確保していても，場合によってはリモートのライトバック時に物理レジスタに空きがない場合がある．この場合には物理レジスタの枯渇を引き起こした命令以降をフラッシュし，再実行する．すなわち，NRP の決定にはフロントエンドのスループット向上とパイプラインフラッシュの増加のトレードオフが存在する．

5. CoreSymphony ver.0.3 の構成

これまでに提案した要素技術をまとめた CoreSymphony ver.0.3 の全体構成を**図 6** に示す．影付きのモジュールはベースのスーパースカラ (図 1) に追加，あるいは大きな変更を加えたモジュールである．各モジュールの入出力ポートには，**表 1** と対応する番号を併記する．表 1 は，CoreSymphony ver.0.3 の HW の複雑度をまとめたものである．比較用に典型的な 2-way および 8-way アウトオブオーダの値を併記した．BW および RW は設計パラメータで，典型的には $BW = 1$，$RW = 1$ である．本稿で提案した 2 つの要素技術に

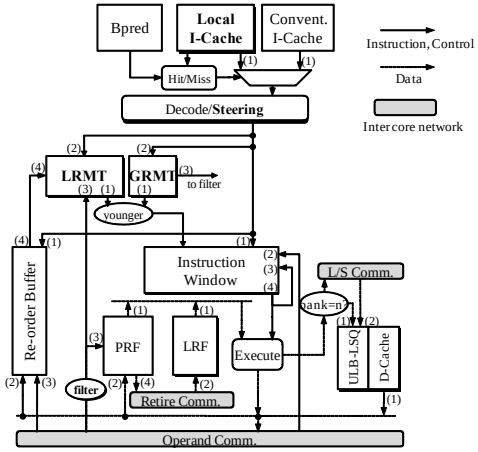

図 6 CoreSymphony ver.0.3 のブロック図.

より，物理レジスタ (PRF) と RMT(LRMT,GRMT) の複雑度が現実的なものになったことがわかる．しかし，LRF および ROB に 8-way 相当の複雑度の部分が存在する．これは，インオーダステートの分散が未検討であることによる．インオーダステートの分散は今後の課題である．

6. CoreSymphony ver.0.3 の評価

6.1 評価環境

評価環境を示す．シミュレータとして，独自開発の実行駆動サイクルレベルシミュレータ SimMips/SS を用いる．SimMips/SS は Linux の動作する MIPS システムレベルシミュレータ SimMips[12] をサイクルアキュレートに拡張したものである．SimMips/SS は標準的なアウトオブオーダスーパースカラを模倣する．よって CoreSymphony のシミュレーションのために拡張を施した．

ベンチマークには SPEC2006 ベンチマークより INT5 種類と FP5 種類を用いる．データセットには train を用い，1G 命令をスキップ後の 100M 命令を評価に使用する．ベンチマークプログラムのコンパイルには MIPS32 用に構築した gcc4.3.3(最適化オプション-O2) を用いる．ただし，遅延分岐最適化をおこなわないように gcc に変更を加えている．これは，CoreSymphony 用のシミュレータが遅延分岐に対応していないためである．

Vol.2010-ARC-188 No.3
2010/3/1

表 1 CoreSymphony ver.0.3 の 1 コアの HW 複雑度.

Module	function	**CoreSymphony**	2-way OoO	8-way OoO
I-cache	(1) fetch	2 inst/cycle	2 inst/cycle	8 inst/cycle
Local I-cache	(1) fetch	2 inst/cycle	-	-
Decoder	Decode	2 inst/cycle	2 inst/cycle	8 inst/cycle
Steering Unit	Steering	2 inst/cycle	-	-
LRMT	(1) Rename(Src)	4 Read	4 Read	16 Read
	(2) Rename(Dst, Local)	2 Write	2 Write	8 Write
	(3) Rename(Dst, Remote)	RW Write	-	-
	(4) Release	2 Read	2 Read	8 Read
GRMT	(1) Rename(Src)	4 Read	-	-
	(2) Rename(Dst)	1 Write	-	-
	(3) State check	$3BW$ Read	-	-
Inst window	(1) Dispatch	2 inst/cycle	2 inst/cycle	8 inst/cycle
	(2) Wakeup(Ltag CAM)	2 Search	2 Search	8 Search
	(3) Wakeup(Gtag CAM)	$3BW$ Search	-	-
	(4) Issue	2 inst/cycle	2 inst/cycle	8 inst/cycle
ROB	(1) Allocate	8 entry/cycle	2 entry/cycle	8 entry/cycle
	(2) Update exec-flag(Local)	2 Write	2 Write	8 Write
	(3) Update exec-flag(Remote)	6 Write	-	-
	(4) Release	8 entry/cycle	2 entry/cycle	8 entry/cycle
PRF	(1) Opfetch	4 Read	4 Read	16 Read
	(2) Writeback(Local)	2 Write	2 Write	8 Write
	(3) Writeback(Remote)	RW Write	-	-
	(4) Commit	2 Read	-	-
LRF	(1) Opfetch	4 Read	-	-
	(2) Commit	8 Write	-	-
ULB-LSQ	Allocate	-	2 inst/cycle	8 inst/cycle
	(1) Dispatch(Local)	1 inst/cycle	1 inst/cycle	4 inst/cycle
	(2) Dispatch(Remote)	1 inst/cycle	-	-
	Issue	1 inst/cycle	1 inst/cycle	4 inst/cycle
D-cache	(1) Load/Store	1 inst/cycle	1 inst/cycle	4 inst/cycle

BW: Broadcast Width, RW: Remote writeback Width

表 2 評価に用いたプロセッサのパラメータ.

(1) Pipeline	9 stage (~issue: 5 stage)
(2) Pipeline width	fetch,issue:2, commit:8
(3) FUs	2 iALU, 1 LD/ST, 1 fpALU
(4) Inst-window	24/24 ent. for INT/FP
(5) PRF	48/48 entries for INT/FP, NRP=18/18
(6) LSQ	96 entries ULB-LSQ$^{\star 2}$
(7) ROB	96 entries
(8) Memory disamb.	1k entries LWT
(9) Branch prediction	1K entries Bimode
(10) BTB	1k entries, 2-way
(11) L1-D$	16KB,2-way,1 cycle,non-blocking
(12) L1-I$(Convent.)	8KB, 2-way, 1 cycle
(13) L1-I$(Local)	8KB, 4-way, 2 cycle
(14) Shared L2$	2MB, 4-way, 10 cycle
(15) Main memory	100 cycle
(16) Inter core latency	1 cycle
(17) BW	1/1 for INT/FP
(18) RW	1/1 for INT/FP

評価に用いるプロセッサのパラメータは**表 2** の通りである$^{\star 1}$．各命令のレイテンシと発行間隔については MIPS R10000[13] と同様に設定した．

$^{\star 1}$ LSQ のエントリ数がコアあたり 96 と多いのは，ULB-LSQ に特有のデッドロック問題を回避するためである．実際には，フロー制御等の実装により，性能へほとんど影響を与えずにエントリ数を一般的な LSQ と同等以下に抑えることができる[7]．

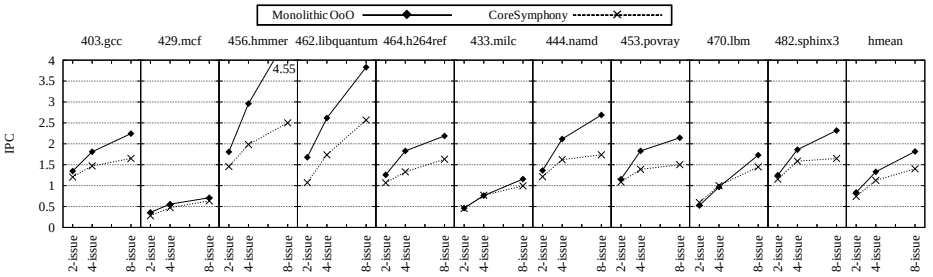

図 7 協調動作による発行幅の増加と IPC の関係.

6.2 協調動作による性能の変化

図 7 は協調動作により発行幅を増加させた場合の IPC の変化をベンチマーク毎に示したものである．2，4，8-issue はそれぞれ 1，2，4 コアの協調動作を意味する．比較対象として，標準的な単一コアデザインのアウトオブオーダの IPC を同時に示す．

比較対象に用いる標準的なアウトオブオーダについては，表 2 の (2) の値を発行幅と同じ値とし，その他のバッファやキャッシュのエントリ数は CoreSymphony の協調動作時の実容量と同等とした．例えば，CoreSymphony は協調動作により一次データキャッシュの用量を純粋に増加させることができる．4 コア協調時にスレッドあたりの実用量は 64KB である．そのため，4 コアの比較対象である 8-way アウトオブオーダは 64KB の一次データキャッシュを持つと設定する．ただし，レイテンシは CoreSymphony における 16KB のキャッシュと同等に設定する．このように，図 7 のアウトオブオーダの IPC は協調のオーバヘッドを排した，理想値としての性格を持つ．

結果を見ると，CoreSymphony は協調動作をおこなうコア数の増加に伴い，IPC が向上することが分かる．1 コア時からの性能向上という観点では，10 種のベンチマークの調和平均 (hmean) において，2 コアの協調で 1.51 倍，4 コアの協調で 1.88 倍の性能を示す．最も性能向上の幅が大きかった lbm では，2 コアで 1.68 倍，4 コアで 2.46 倍という IPC を達成した．

単一コアのアウトオブオーダとの比較では，協調動作のオーバヘッドが分かる．8 命令発行時の調和平均で，CoreSymphony は単一コアのアウトオブオーダに比べ，23.3%低い IPC を示した．

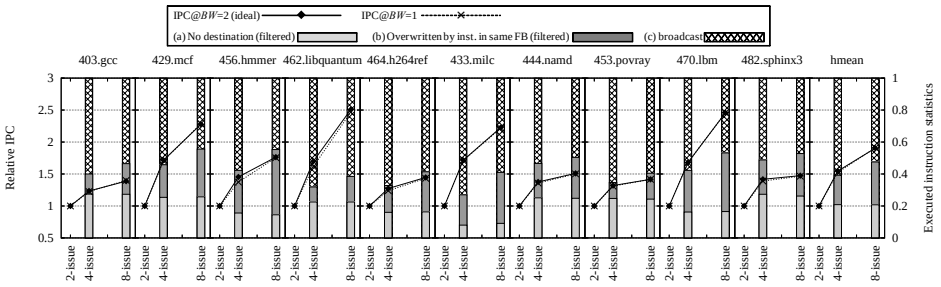

図 8 コアあたりのブロードキャストの幅 (BW) と性能の関係 (左軸)．実行された命令に関する統計情報 (右軸)．

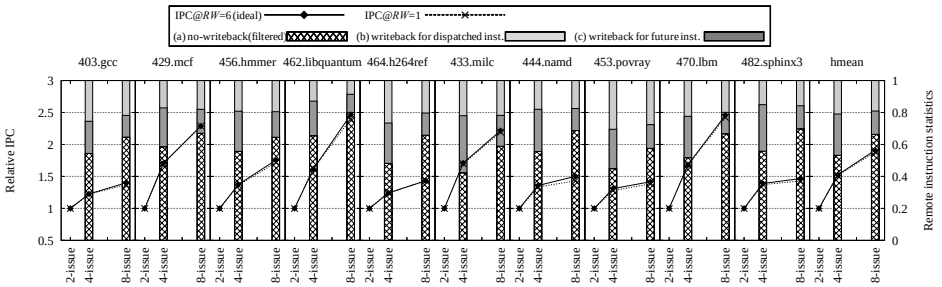

図 9 物理レジスタのリモートからのライトバックのポート数 (RW) と性能の関係 (左軸)．ブロードキャストされた命令に関する統計情報 (右軸)．

6.3 ブロードキャストの幅と性能の関係

図 8 は，1 サイクルあたりにブロードキャストをおこなうことができる命令の数 (BW) と性能の関係をまとめたものである．典型的な値である $BW = 1$ と，理想的な値である $BW = 2$ の 2 種類の設定で，$BW = 1$ の 1 コア時に対する相対 IPC を測定した．同時に，実行された命令に関する統計情報を示す．(a) はレジスタへの書き込みをおこなわない命令，(b) は同 FB 内かつ同コアの他の命令によって値が上書きされる命令，(c) はそれ以外のブロードキャストされる命令の割合である．(a) と (b) の場合にブロードキャストをフィルタできる．命令の統計情報は，協調中の各コアについて集計し平均をとった．

結果を見ると，BW を理想化してもほとんど性能は変化しないことが分かる．この結果は，命令の統計情報を見ることで納得ができる．ブロードキャストがフィルタされる命令の割合は，2 コア時には 4 割弱，4 コア時には 5 割弱にも及ぶ．特に，(b) によってフィルタされる命令の割合が協調コア数が増えるほど増加しており，非常に良い傾向を示す．これには，リーフノードステアリングの特徴である，依存関係にある命令を同じコアにまとめる効果が寄与していると考えられる．協調コア数の増加により FB が長くなるほど，リーフノードステアリングの効果は大きくなる．

結論として BW は 1 命令/cycle で十分であり，協調するコアの数に依存しないパラメータであることが分かった．

6.4 リモートのライトバックのポート数と性能の関係

図 9 は，物理レジスタのリモートからのライトバックのポート数 (RW) と性能の関係をまとめたものである．典型的な値である $RW = 1$ と，理想的な値である $RW = 6$ の 2 種類の設定で，$RW = 1$ の 1 コア時に対する相対 IPC を測定した．同時に，ブロードキャストされた命令に関する統計情報を示す．(a) は物理レジスタへのライトバックがフィルタされた命令，(b) は命令ウインドウ内の命令のためにライトバックをおこなった命令，(c) はこれからディスパッチされるであろう命令のためにライトバックをおこなった命令である．命令の統計情報は，協調中の各コアについて集計し平均をとった．

結果を見ると，hmmer や libquantum，namd といった IPC の大きなベンチマークで，RW を狭幅化した影響が僅かに表れているものの，総合的にはほとんど影響がないことが分かる．命令の統計情報を見ると，ライトバックがフィルタされる命令は 2 コア時に 5 割強，4 コア時には 6 割強にもおよぶ．特に，協調動作するコア数が増えるにつれ (b) の割合が減少し，フィルタできる命令数増加に寄与している．

結論として，RW は 1 ポートで十分であることが分かった．

6.5 物理レジスタの構成と性能の関係

次に，リモートからの束縛用に確保する物理レジスタ数 NRP が性能に与える影響を評価する．物理レジスタ数を INT/FP ともに 48 エントリとし，NRP を 14，18，22 と変化させた．**図 10** は $NRP = 18$ の 1 コア時に対する相対 IPC をまとめたものである．10 種のベンチマークのうち，特徴的な傾向を示した hmmer と lbm，および全ベンチマークの調和平均を示す．同時に，100M 命令を実行する間に発生したパイプラインフラッシュのうち，リモート命令の物理レジスタ束縛時における物理レジスタ枯渇に起因する物の回数を示す．

結果をみると hmmer と lbm では逆の傾向を示していることが分かる．hmmer は NRP を少なく設定したほうが良い IPC が得られ，lbm は NRP を大きめに設定したほうが IPC が高い．これは，hmmer が NRP の不足によるパイプラインフラッシュを全く発生させな

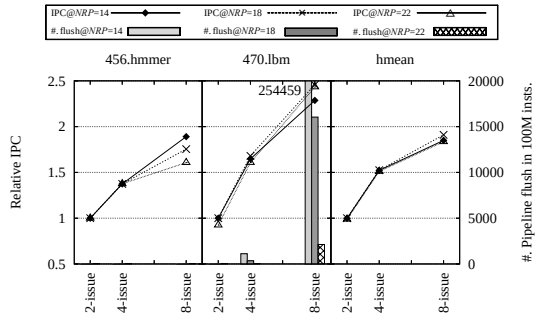

図 10 リモート命令用に確保する物理レジスタ数 (NRP) と性能の関係 (左軸)．物理レジスタの枯渇によるパイプラインフラッシュの発生回数 (右軸)．

いベンチマークであるのに対し，lbm は非常に多く発生するという，プログラムの性質の違いによるものである．基本的に NRP は小さく設定する程，ディスパッチのストールが起こりにくくなるため，性能が向上する．しかし，lbm のように，リモートからの束縛時に物理レジスタ不足を引き起こしやすいベンチマークでは，パイプラインフラッシュによるオーバーヘッドがディスパッチのスループット向上によるメリットを上回る．

このトレードオフの存在は，CoreSymphony のさらなる性能向上の可能性を示す．図 10 から分かるように，最適な NRP はアプリケーションによって異なる．プログラム中の実行フェーズによっても異なると考えられる．よって，NRP を動的に変更することでさらなる性能向上が得られる可能性が高い．

7. まとめと今後の課題

CMP の逐次処理能力の向上および逐次処理能力と並列処理能力のバランシングを目的とし，協調動作可能なスーパースカラアーキテクチャCoreSymphony の研究を進めている．本稿では，これまでの実装で問題となっていた，RMT の複雑化と物理レジスタのエントリ数/ポート数増大を解決すべく，次の 2 つの要素技術を提案した：

- RMT の軽量化を実現する **2-way リネーミング**.
- 物理レジスタのエントリ分散とポート数の抑制を実現する **CoreSymphony 向け物理レジスタ分散手法**.

また，これらの要素技術を実装する，CoreSymphony ver.0.3 を定義した．CoreSymphony ver.0.3 は一部の未検討部分に例外はあるものの，現実的なハードウェア増加量でスーパースカラの協調動作を実現する．また，シミュレーションによる性能評価の結果，4 コアの協調動作時に，1 コア時と比較して平均 1.88 倍，最大 2.46 倍の IPC が得られることが分かった．

協調可能スーパースカラの実現に向けた今後の課題として次のことがあげられる：

- 未検討部分であるインオーダステートの分散.
- より詳細なハードウェア量の見積もり.
- NRP の動的最適化等による，さらなる性能向上.

参 考 文 献

1) M. D. Hill and M. R. Marty: Amdahl's law in the multicore era, *IEEE Computer*, Vol.41, No.7, pp.33–38 (2008).
2) 若杉　他：メニーコアプロセッサに向けたシンプルで柔軟なコア融合機構 CoreSymphony，先進的計算基盤システムシンポジウム (SACSIS-2008)，pp.411–419 (2008).
3) 若杉　他：CoreSymphony アーキテクチャの高効率化，情処研報 2009-ARC-184，pp. 1–12 (2009).
4) V. V. Zyuban and P. M. Kogge: Inherently Lower-Power High-Performance Superscalar Architectures, *IEEE Transactions on Computers*, Vol.50, No.3, pp.268–285 (2001).
5) E. Ipek, et al.: Core Fusion: Accommodating Software Diversity in Chip Multiprocessors, *Proc. of 34th ISCA*, pp.186–197 (2007).
6) A. Yoaz, et al.: Speculation Techniques for Improving Load Related Instruction Scheduling, *Proc. of 26th ISCA*, pp.42–53 (1999).
7) S. Sethumadhavan, et al.: Late-binding: enabling unordered load-store queues, *Proc. of 34th ISCA*, pp.347–357 (2007).
8) M. Johonson: Superscalar Microprocessor Design, Prentice hall, Englewood Cliffs (1990).
9) M. E. Hopkins and R. Nair: Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups, *Proc. of 24th ISCA*, pp.12–25 (1997).
10) E. Talpes and D. Marculescu: Execution cache-based microarchitecture power-efficient superscalar processors, *IEEE Transactions on VLSI*, Vol. 13, No. 1, pp. 14–26 (2005).
11) T. Monreal, et al.: Delaying physical register allocation through virtual-physical registers, *Proc. of 32nd MICRO*, pp.186–192 (1999).
12) 藤枝　他：教育・研究に有用な MIPS システムシミュレータ SimMips，情報処理学会論文誌, Vol.50, No.11，pp.2665–2676 (2009).
13) K. C. Yeager: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol.16, No.2, pp.28–40 (1996).