修士論文

スーパースカラ型自動メモ化プロセッサにおける再利用テストタイミングの改良

指導教員　　松尾 啓志 教授
　　津邑 公暁 准教授

名古屋工業大学大学院工学研究科
修士課程創成シミュレーション工学専攻
平成 21 年度入学　21413520 番

加藤 拡

平成 23 年 2 月 3 日

スーパースカラ型自動メモ化プロセッサにおける再利用テストタイミングの改良

加藤 拡

内容梗概

これまで，科学技術計算だけでなく医療分野，工業製品の設計等といった様々な場で CPU には高度な計算処理能力が求められており，その高速化のために様々な研究が行われてきた．動作クロックを向上させるためにパイプラインの多段化は進み，それに伴い増大するハザードによるペナルティを緩和するために命令レベル並列性についての研究がさかんに行われてきた．しかし，動作クロックの向上による消費電流の増大を無視できなくなり，CPU はマルチコア化の道を進むことになる．マルチコア CPU を有効に活用するためにスレッドレベル並列性の研究が行われてきたが，依然としてシングルタスクの高速化は非常に困難な状況である．

一方，これら並列化による CPU の高速化とはまったく異なったパラダイムである計算再利用という高速化手法が存在する．計算再利用の実行モデルの一つに自動メモ化プロセッサというものがある．自動メモ化プロセッサは過去に実行した関数やループの入出力を表に記憶しておき，後に同じ関数やループが過去と同じ入力で実行されようとする時に，過去の実行結果を再利用することにで高速化を図る．

この自動メモ化プロセッサの既存モデルに，スーパースカラ型 ARM をベースとしたものが存在する．これにはいくつかの問題点が存在し，その 1 つに再利用成功時のオーバヘッドの大きさが挙げられる．このオーバヘッドは再利用テスト成功時に発生するパイプラインフラッシュによるものである．これによりバブルとなってしまう命令の数を削減し高速化を図るため，再利用テストを開始するタイミングをパイプラインの最終ステージである RETIRE ステージからより前段へと変更した．

本提案手法をサイクルベースのシミュレーションにより評価を行ったところ，stanford ベンチマークで通常実行時に比べて平均で 67.4%，最大で 72.3%のバブル削減を確認できた．

スーパースカラ型自動メモ化プロセッサにおける再利用テストタイミングの改良

目次

1　はじめに

現在までに，プログラムの実行を高速化する手法として，スーパースケーラのように命令レベル並列性 (Instruction-Level Parallelism:ILP) に着目したものが研究されてきた．しかしながら，プログラム自体に存在する ILP には限界があり，命令レベルの並列化を行うだけでは，プロセッサの性能向上が頭打ちになりつつある [1]．また，並列性を高めるための命令のスケジューリング制御部分はハードウェアに負担をかけている．この流れを受け，CPU がマルチコア化されるとともに，命令レベル並列性より粒度の粗い並列性であるスレッドレベル並列性 (Thread-Level Parallelism:TLP) が注目されることとなる．並列性の抽出にはマルチスレッドライブラリや，高度なコンパイラが用いられる．しかし，ライブラリを用いた場合，プログラム内の並列化できる部分を明示的にスレッドの形で書き下す必要があり，プログラマに負担がかかる．一方，コンパイラによる抽出はその精度に問題がある．よって TLP によるプログラム実行の高速化もまた難しい状況であるといえる [2]．さて，これらの高速化手法は粒度の違いはあれど，いずれもプログラムの持つ並列性に着目したものである．一般にプログラムとして記述される対象処理自体は半順序構造を成しており，その構造は時間軸に沿った縦方向への広がりと，時間軸に直交する横方向への広がり（並列性）を持っている．処理のこの横方向への広がりに着目し，その処理量を圧縮しようとするのがこれらの高速化手法であると言える．

一方，従来の高速化手法とは着眼点が全く異なるものとして，上記の処理の半順序構造における縦方向の広がりを圧縮しようと試みる手法である計算再利用がある。．計算再利用には，ハードウェアによるものとソフトウェアによるもの，またその両方によるものなど，様々なものが提案されている．そのハードウェア実装の 1 つに，専用のハードウェアを用いることによりバイナリの変更無しに既存のプログラムを実行できる自動メモ化プロセッサが存在する．自動メモ化プロセッサは実行した関数の入出力を表に記録する．そして再び同じ関数が呼び出されたときにその入力と過去に実行した関数の入力とを比較し，一致すれば過去の出力を利用して高速化を図る [3].

この自動メモ化プロセッサには 2 つの既存モデルがある．1 つは、単命令発行型の SPARC アーキテクチャをベースとしたモデルである．しかし，現在市場に流通しているプロセッサは通常パイプライン化されており，自動メモ化プロセッサの実際的なパフォーマンスについて，SPARC ベースモデルの評価結果だけでは十分とはいえない．

以上を受けて提案されたのが，もう 1 つの既存モデルである．これは，スーパース

カラ型 ARM をベースアーキテクチャとしている．このモデルでは，計算再利用による高速化をパイプラインプロセッサ上で行っている．その実現のためのハードウェアは SPARC ベースモデルと比べて複雑となり，また ARM の仕様から純粋にハードウェアのみでの命令区間の検出は困難であり，ソフトウェア支援が必要となったが，パイプラインプロセッサ上で計算再利用を行った際のパフォーマンスを測ることが可能となっている．

しかし，この ARM ベースモデルには SPARC ベースモデルには存在しないオーバーヘッドが存在する．ARM ベースモデルでは，再利用が適用されるたびにパイプラインがフラッシュされる．これは再利用を適用したときに後続の命令を一旦破棄するための処理であるが，そのためにパイプライン中にバブルが発生してしまう．加えて既存の ARM ベースモデルでは，再利用が適用可能かどうかの検査をパイプラインの最終ステージである RETIRE ステージまで命令が到達するのを待って行っているために，バブルが多くなってしまっている．本研究では再利用が適用可能かどうかの検査のタイミングを改良し，このバブルによるオーバーヘッドの削減を図る．

以下，2 章では本研究が扱う自動メモ化プロセッサの，SPARC ベースモデルと ARM ベースモデルの動作モデルと実装をそれぞれ概説する．3 章では，本論文の提案モデルと，その実現のために必要な事項について述べ，4 章で提案モデルの実装手法について詳細に説明する．5 章でその評価を行い，最後の 6 章において結論を述べる．

2　既存手法

2.1　自動メモ化プロセッサ

本章では，自動メモ化プロセッサの概略と，その既存モデル 2 つの動作を概説する．

2.1.1　概略

計算再利用（Computation Reuse）とは，主に関数などの命令区間に対してその入力と出力の組を実行時に記憶しておき，再び同じ入力によりその命令区間が実行されようとした場合に，過去に記憶された出力を利用することで命令区間の実行自体を省略し，高速化を図る手法である．また，それら命令区間に計算再利用を適用することをメモ化（Memoization) と呼ぶ．このメモ化をハードウェアにより自動的に行うプロセッサとして，自動メモ化プロセッサが提案されている．自動メモ化プロセッサは，関数及びループの実行を省略することによりプログラムの実行の高速化を図る．自動メモ化プロセッサには，単命令発行型の SPARC アーキテクチャ[4] をベースとしたモデルとスーパースカラ型の ARM アーキテクチャ[5] をベースとしたモデルの 2 種類が存在

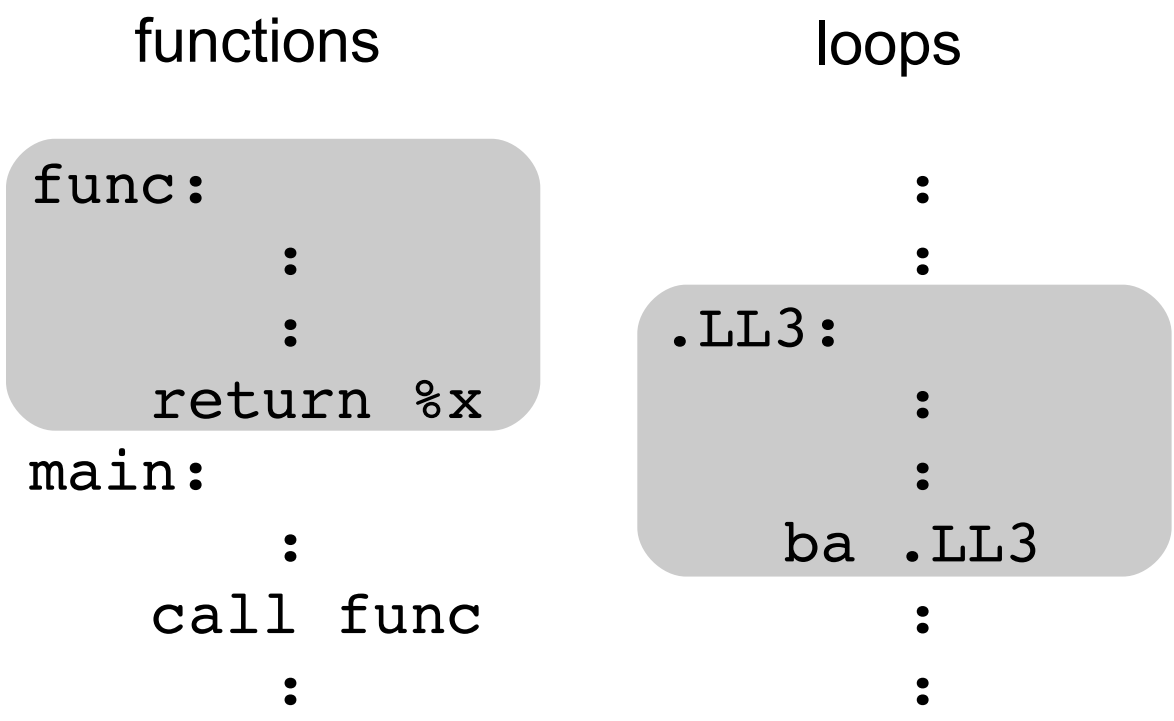


図 1: 関数とループの検出方法と登録される再利用区間

する．

2.1.2 SPARC ベースモデル

SPARC ベースモデルを例に，自動メモ化プロセッサの動作を説明する．自動メモ化プロセッサは，プログラム実行時に動的に関数およびループイタレーションを再利用可能命令区間として検出する．再利用可能命令区間の検出は，その開始アドレスと終了アドレスを検出することによって可能である．関数呼び出し時の命令の流れとループ時の命令の流れを**図 1** に示す．関数の区間は，図 1 左に示すように call 命令のターゲットから return 命令までの区間である．またループイタレーションは，図 1 右のように後方分岐命令のターゲットからその後方分岐命令までの区間である．これらの区間を対象として，自動メモ化プロセッサはメモ化を行う．

自動メモ化プロセッサはまず，再利用対象となる命令区間の実行時にその入力と出力を MemoTbl と呼ばれる表に登録する．関数及びループ区間の入力は，関数の引数および関数内で参照される大域変数であり，出力は関数の戻り値および関数内で書き換えられる大域変数である．さらにループ区間の場合，1 回のループイタレーションの実行中に出現する大域変数に加え，そのループ区間を含む関数の局所変数もループの

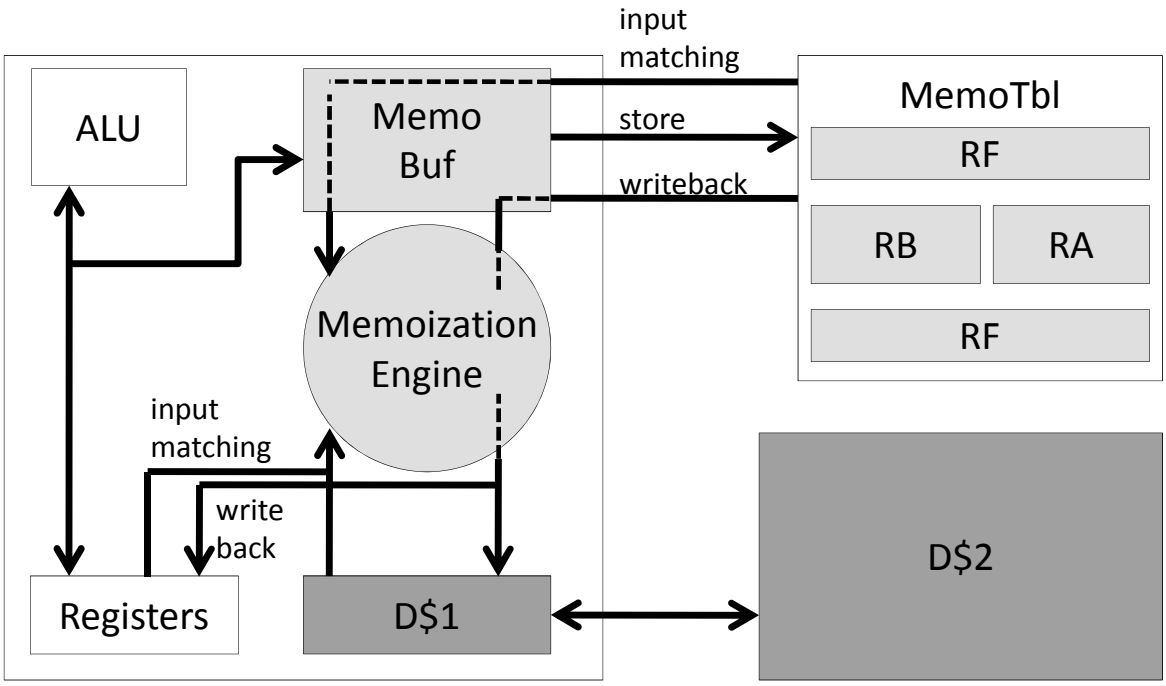


図 2: 自動メモ化プロセッサ構成図

入出力として扱う．その後，再び同じ関数が実行されるときにその入力を過去の入力と比較し，一致した時には対応する出力をレジスタ，キャッシュに書き戻し，当該関数の実行を省略する．

自動メモ化プロセッサの構成図を**図 2** に示す．MemoTbl はその内部に 4 つの表を持つ．このうち RF，RA，W1 は RAM で構成される表であり，RB は連想検索が可能な CAM(Content Addressable Memory) で構成される表である．これら構成要素の詳細は以下の通りである．

RF: 登録済みの命令区間に関する情報を記憶する表である．RF に記憶する情報は多岐に渡る．まず，命令区間のアドレスとして関数の開始アドレス (関数アドレス) やループの分岐命令のアドレス及びループのジャンプ先命令のアドレスを記憶する．それ以外にも，過去に計算再利用が成功又は失敗した回数や，各命令区間に対して最近登録した入出力値の履歴等を記憶する．

RB: 命令区間の入力値を記憶する表である．入力値を高速に検索可能とするため，CAM で構成される．

RA: 命令区間の入力アドレスを記憶する表である．

W1: 命令区間の出力値を記憶する表である．

このモデルが関数を対象としたメモ化を行う際の動作の概略は以下の通りである．

ある関数への call 命令が検出されたとすると，その関数の先頭アドレスが RF 上に登録されているか検索を行う．登録されていた場合，プロセッサは入力一致比較を行う際に最初に比較する RB エントリへのポインタを RF から取得する．次に入力値の一致比較を RB と RA を用いて行う．これらの，RF 上からの命令区間の検索から入力値の一致比較までの一連の処理の流れを**再利用テスト**と呼ぶ．詳細は 2.2.1 項で述べる．最後に，再利用テストの結果入力がすべて一致すれば，登録済みの出力が W1 から書き戻され関数の実行は省略される．

もし RF に当該関数が見つからなかったりすべての入力が一致せず再利用が行われなかった場合，関数を実行し MemoBuf 上に関数の入出力を登録していく．そして当該関数の実行終了時，すなわち return 命令を検出した時，MemoBuf に登録された内容を MemoTbl 本体である RF，RB，RA，W1 に登録する．

関数再利用における call 命令と return 命令の検出と同様に，どの命令からどの命令までがループ区間であるかを認識することでループ再利用が可能となっている．命令列がループを成すことの判定は，一度後方分岐命令が実行され成立した後に再度同じ後方分岐命令に到達したことを検出することで行う．つまり，後方分岐命令の分岐先から，再度現れる同一の後方分岐命令までの入出力を MemoBuf に登録しておくことで，関数同様にループ再利用が可能となる．ループの再利用に必要な入力は，ループ内で行われる全てのレジスタ及びメモリ参照である．また，出力はループ内で行われる全てのレジスタやメモリへの値の格納である．これは，関数と異なりループには局所変数が規定されないためである．

この他に，ループ再利用を行う上での注意点として出力書き戻し後の復帰アドレスが挙げられる．関数再利用の場合，出力書き戻し後の復帰アドレスは必ず call 命令の次の命令 (厳密には遅延スロットがあれば，スロット分の命令後) のアドレスである．しかしループ再利用では，分岐方向 (taken or untaken) によって復帰アドレスは異なる．そのため，再利用適用時に復帰アドレスを得るために，ループ毎に RB に分岐方向を記憶させておく必要がある．

さて，一般的に関数は関数を呼び出すことにより入れ子構造をとる．この入れ子構造にある関数を再利用可能とするような MemoTbl への登録の仕組みを多重再利用と呼ぶ．多重再利用時の登録の様子を以下に説明する．

図 3 は多重再利用時において、入れ子構造の関数の入出力がどのように MemoBuf に登録されるのかを示している．図 3 の右側半分は実行中のプログラムの関数の入れ子構造を表しており，関数 `f` が関数 `f_s` を呼び出し，関数 `f` は関数 `f_p` に呼び出されてい

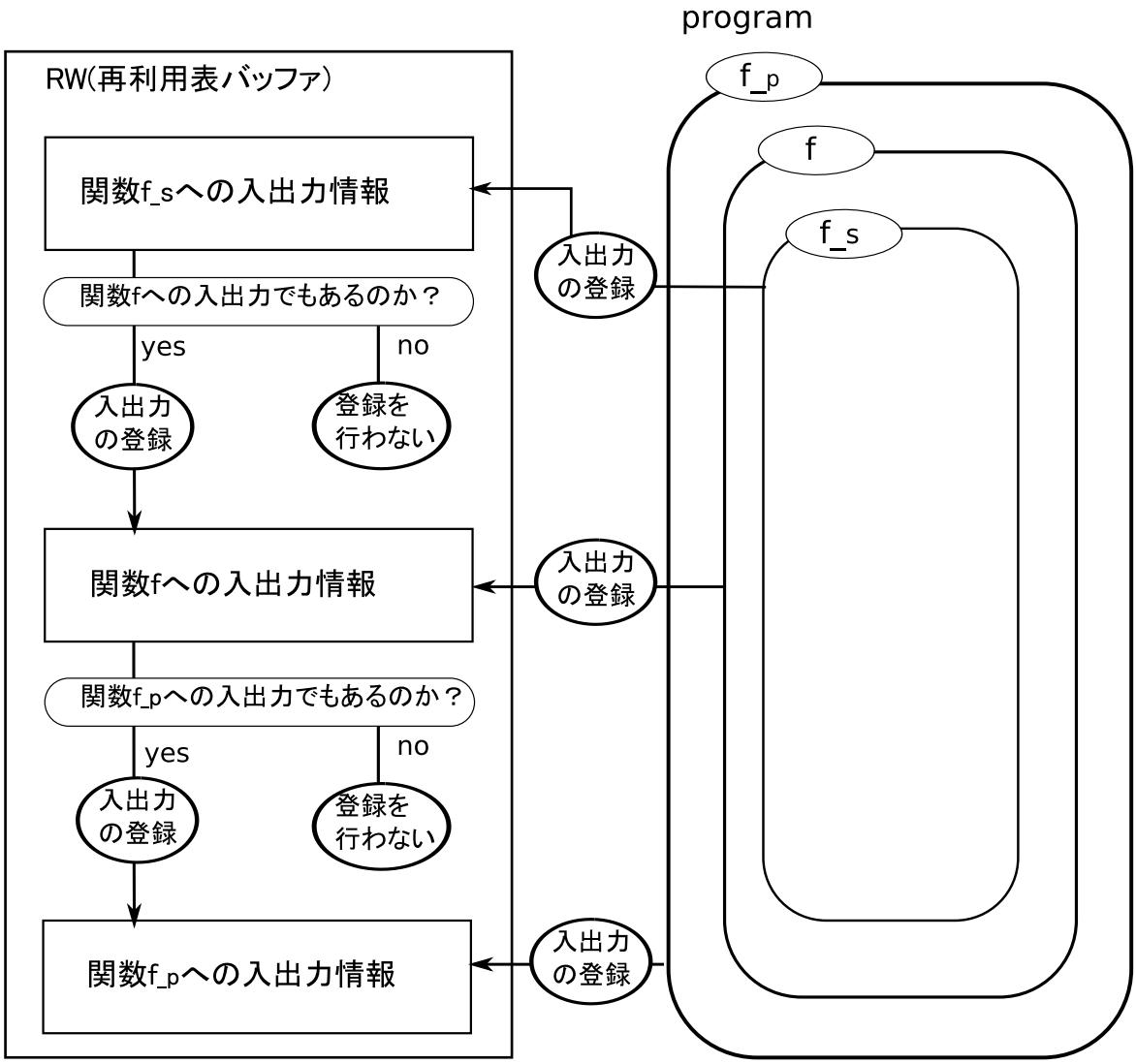


図 3: 多重再利用を実現する MemoBuf の構造

る．さて，再利用が行われず，関数 `f_s` が通常実行される場合について考える．関数 `f_s` の処理が進むにつれ，MemoBuf には関数 `f_s` の入出力が登録されるが，関数 `f_s` の入出力の中で，大域変数や関数 `f` を呼び出した関数である `f_p` の局所変数の読み書きによるものは，関数 `f` の入出力でもあるため，関数 `f` の入出力としても登録する必要がある．つまり，関数の入れ子構造において，入れ子の内側の関数の入出力は入れ子の外側の関数の入出力となり得るということである．

このことから，多重再利用を実現するために MemoBuf はスタック構造をとっている．新たに関数が call される度に，MemoBuf にその関数の入出力を記録するためのエントリを push する．そして，return 命令で関数の実行が終わると pop し，その入出力

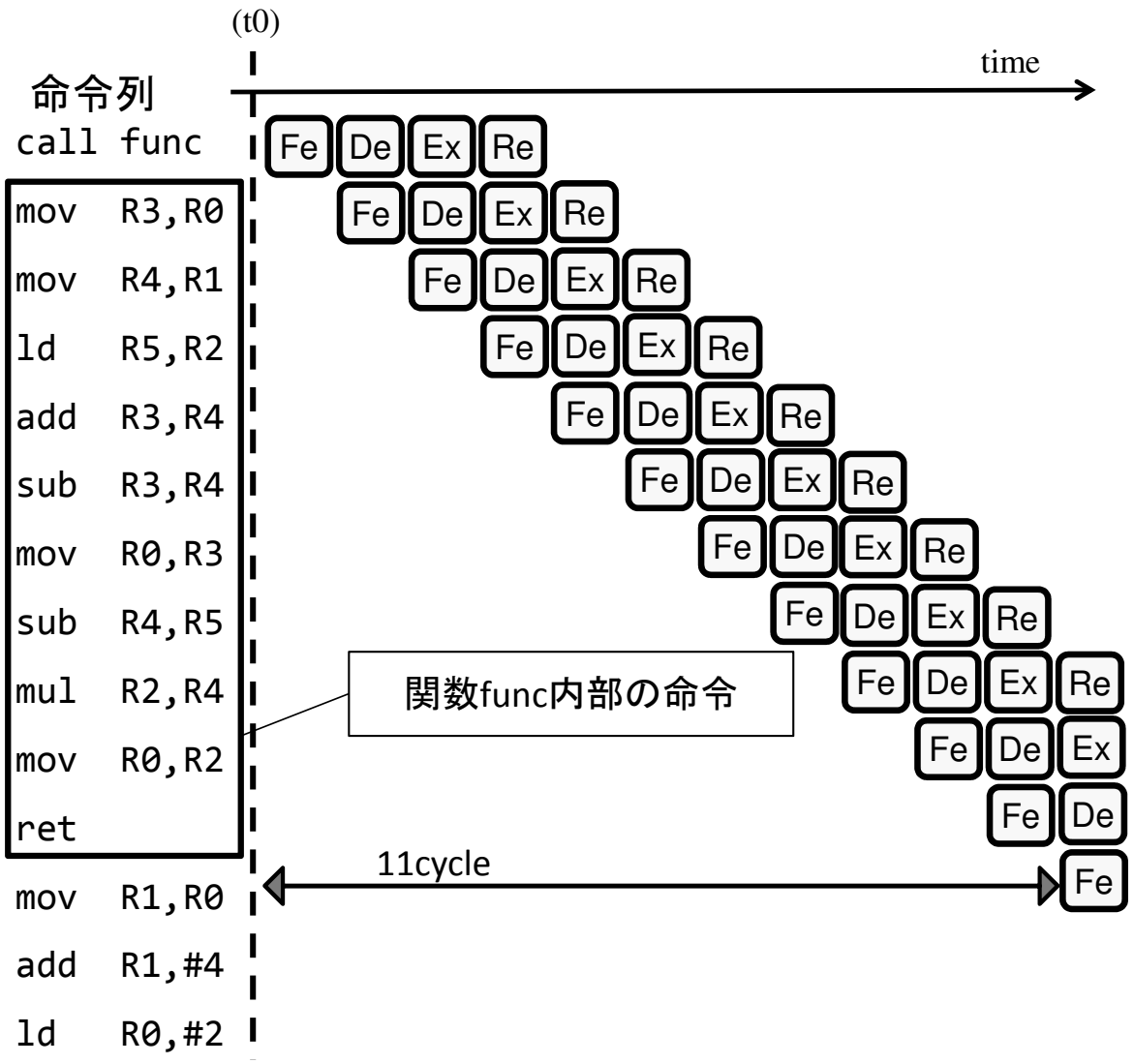


図 4: ARM ベースモデルの動作モデル:通常実行時

内容を MemoTbl へ登録する．関数の入出力が検出された時，その関数の入出力の記録を担当する MemoBuf エントリに入出力は記録される．そして次に，該当 MemoBuf エントリより下に積まれている MemoBuf エントリ各々に対して，入出力が記録するべき入出力なのかの判断が行われ，該当エントリが管理する関数の入出力であると判断された場合，その入出力は記録される．

2.1.3 ARM ベースモデル

本項では，スーパースカラ型 ARM をベースとする自動メモ化プロセッサの動作モデルを述べる．このモデルでは，現在は関数を対象としたメモ化のみが実装されている．

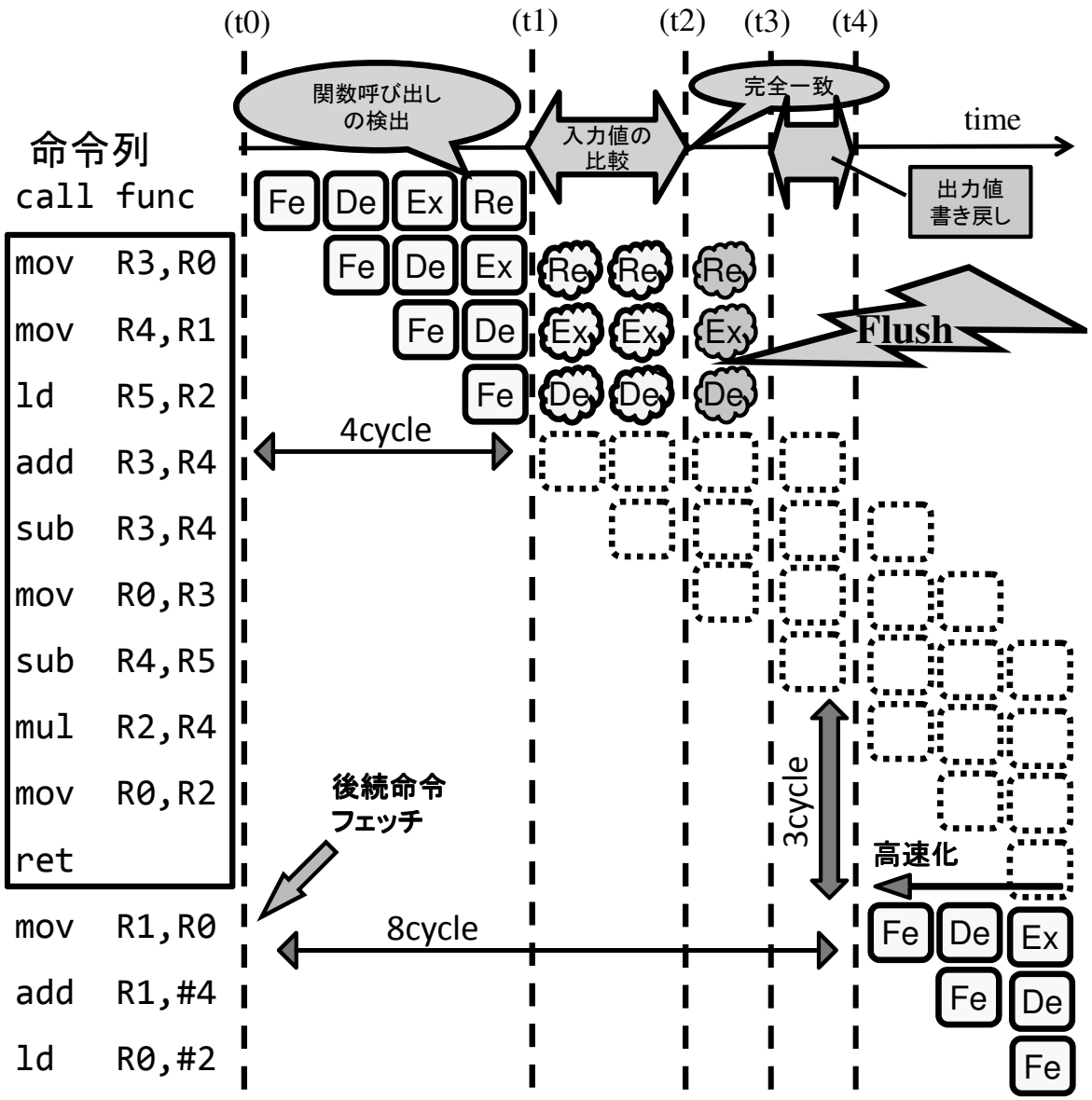


図 5: ARM ベースモデルの動作モデル:再利用成功時

図 4 に説明に用いるプログラム例とそれが実行された時のパイプラインの様子を示す．図 4 左が実行中のプログラムであり，図右が横軸を時間軸としたパイプラインの様子である．プログラム中の四角で囲ってある部分は func 内の命令列である．なお，説明の簡単化のため，例に示すベースアーキテクチャのパイプラインは，Fe(命令フェッチ)，De(デコード)，Ex(命令実行)，Re(リタイア) のシンプルな 4 段構成を取っている．また，キャッシュミス等によりパイプラインがストールすることがない，理想的な実行時の様子を示している．また，図中 (t0) は関数呼び出し命令をフェッチした時刻であ

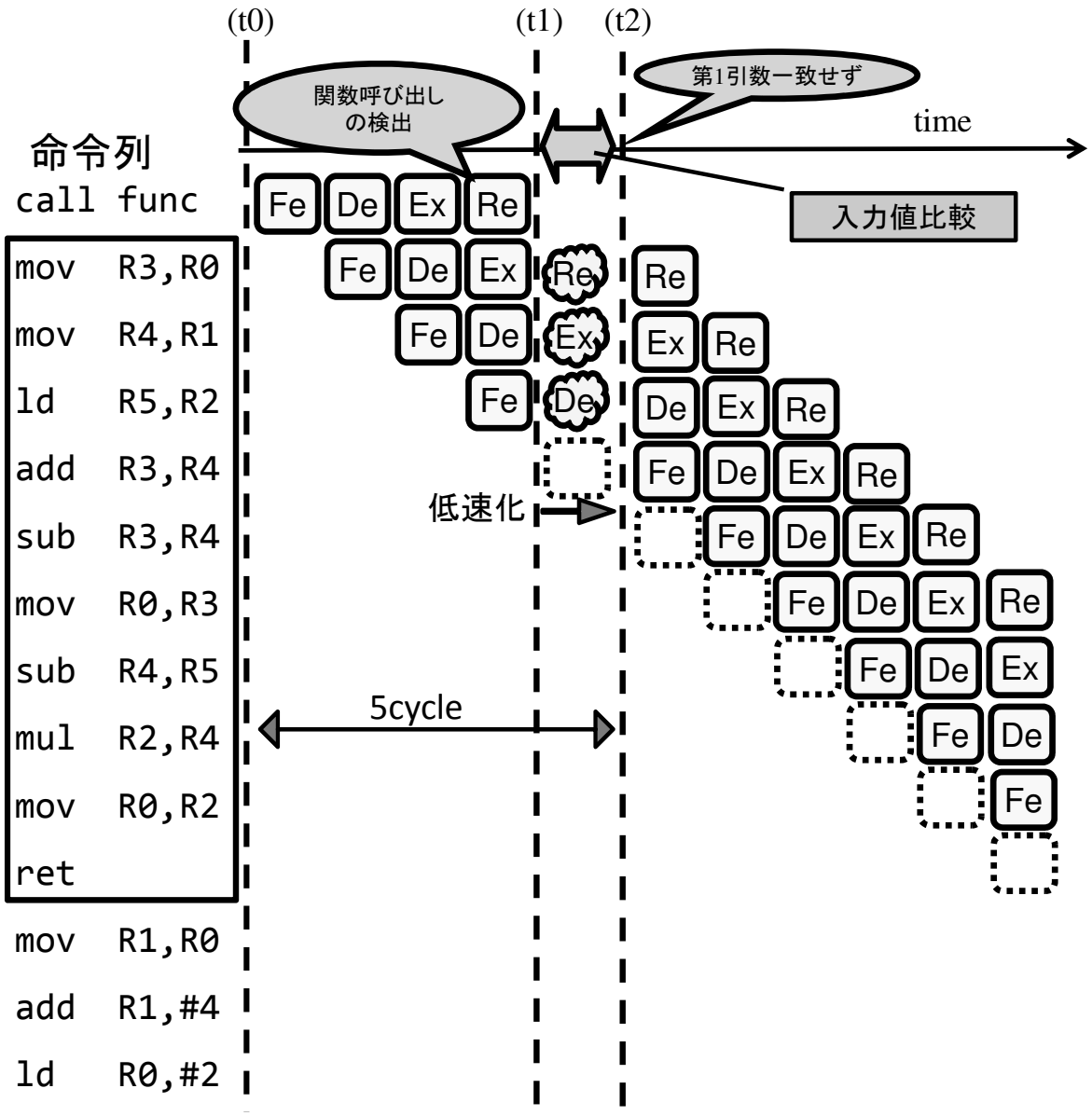


図 6: ARM ベースモデルの動作:再利用失敗時

り，この時刻から再利用を終了するまでが，再利用に要した時間となる．なお，関数 func を再利用を行わずに通常通り実行すると，関数復帰までに 11 サイクルを要する．

図 5 に ARM ベースモデルの再利用成功時の様子を示す．図右にある点線の四角は，ベースアーキテクチャでプログラムを実行した際のプログラムの進行状況を表す．関数呼び出し命令がコミットされるタイミングで，入力値の検索は開始される (図 5 中 (t1))．入力値の検索とは，論理レジスタ，及びキャッシュの値が該当関数を過去に実行した時と同じものであるかどうかを比較することである．この比較を正しく行うには，

関数呼び出し命令より前の命令列による論理レジスタやキャッシュへの書き込みが既に終了されている必要がある．関数呼び出し命令がコミットされていれば，それより以前の命令が終了していることが保証されるので，入力値の検索は関数呼び出し命令がコミットされたタイミングで行う．そのため，関数呼び出し命令がフェッチされてから入力値の検索が始まるまで，4 サイクルを要する．入力値の比較を行う間，パイプラインは停止している．これは，関数呼び出し命令の後続命令がレジスタやキャッシュへと値を書き込むような命令であった場合に，その命令によって一致比較対象となる入力値が別の値で上書きされてしまう可能性があるためである．もし一致比較を行う前に入力値が別の値で上書きされると，本来とは異なる値を用いて一致比較がされてしまい，誤った再利用が適用される可能性がある．これを防ぐために，ARM ベースモデルでは入力値の一致比較を行う間はパイプラインをストールさせる．その後，入力値が過去の入力値と全て一致することが確かめられ，再利用が適用可能であることが判明する (図 5 中 (t2))．関数の実行を省略するために，過去の関数実行時出力をレジスタ，キャッシュに書き戻すのだが，その前にパイプラインのフラッシュを行う (図 5 中 (t2)-(t3))．すでにパイプラインに投入されている命令列は，再利用の対象である関数内の命令列であり，無効化する必要があるためである．パイプラインのフラッシュ後，あらためて出力を書き戻し (図 5 中 (t3))，その後，後続命令をフェッチしてくる (図 5 中 (t4)) ．以上の関数実行省略により，図 5 の場合では関数呼び出し命令がフェッチされてから関数復帰後の命令がフェッチされるまで 8 サイクルとなり，再利用を適用しなかった場合と比べ 3 サイクル高速化する．

次に，ARM ベースモデルの再利用失敗時の動作の様子を**図 6** 示す．実行するプログラムは図 5 と同じである．まず，先程の例と同様に，RETIRE ステージにて関数呼び出し命令がコミットされるのを検知し，入力値の検索が始まる (図 6 中 (t1))．レジスタ，キャッシュの値と RB 内の値が一致せず，再利用の適用が不可能なことが判明すると，その後は再び命令の実行を継続する (図 6 中 (t2))．この例では，入力値検索の途中で入力値の不一致を検知し，入力値検索を途中で中止している．このように，再利用が失敗する場合でも入力値比較のためにパイプラインをストールさせる必要があるため，プログラムの実行は低速化する．また，一部の入力値が異なるために再利用が出来ない場合でも，すべての入力が比較可能となるまで一致比較を開始できない．そのため，入力値一致比較を途中で中止するとしても，関数が再利用出来ないことが判明するまでに合計で 5 サイクルかかる．

例 1：自動メモ化プロセッサの動作モデル説明用プログラム

```
 1:  int weight = 1;
 2:  int array_mul(int num, int a[], int b[]) {
 3:      int i, v = 0;
 4:      for(i = 0; i < num; i++)
 5:          v += a[i] * b[i];
 6:      return (v / weight);
 7:  }
 8:  int main(void) {
 9:      int x, y, z, a[3] = {1, 2, 3}, b[3] = {4, 5, 6};
10:      x = array_mul(3, a, b);
11:      weight++, b[0] = 7;
12:      y = array_mul(3, a, b);
13:      weight--, b[0] = 4;
14:      z = array_mul(3, a, b);
15:      return (0);
16: }
```

2.2 自動メモ化プロセッサの実装

本節では，自動メモ化プロセッサの既存モデルに共通する MemoTbl の動作と，特に ARM ベースモデルの実装について詳細を述べる．

2.2.1 再利用テスト

本項では，MemoTbl が再利用テストを行う際の詳細な動作を述べる．例 1 のプログラム内の関数 `array_mul` は，2 つの配列について添字の一致する要素同士の積を求め，第 1 引数 `num` で指定された回数だけ配列の添字 0 から順に同様の処理を行い，それらの和を求め，その値を局所変数 `v` に代入する．その後局所変数 `v` を大域変数 `weight` で除算した結果を返す．また，関数 `array_mul` は `main` 関数の中で複数回呼び出される．

さて，一般に関数やループ等の命令区間は複数の異なる入力セットにより実行される．その入力セットは，ある途中の入力値によって次に参照するべき入力アドレスが変化する．よってある命令区間の全入力パターンは，入力アドレスを節，入力値を枝と見なした木構造で表すことができる．そして，その命令区間の 1 入力セットは，その

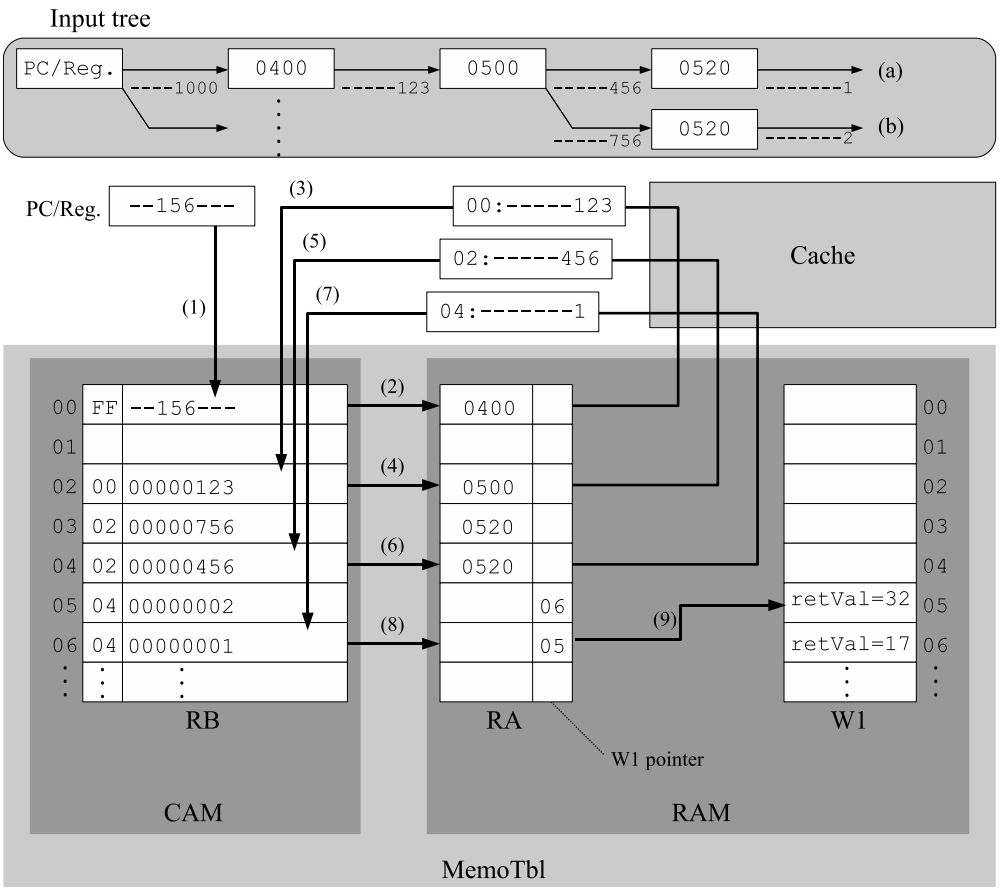


図 7: 入力のツリー構造と入力値検索の手順

木構造におけるルートからリーフまでの 1 本のパスとして表現できる．**図 7** の上部に，例 1 の関数 `array_mul` を呼び出した際に当該関数の入力が成す木構造の上での，10 行目及び 12 行目で呼び出された際の入力が成すパスを，それぞれ (a)，(b) として示す．引数 `num` の値及び配列 `a` の値は 2 つの入力セット (a) 及び (b) で共通している．しかし配列 `b` の値及び大域変数 `weight` の値が異なっているため，図 7 に示したように 3 つ目の入力から入力ツリーが分岐する．入力セットは RB 及び RA に木構造を保持したまま登録され，RB が節に，RA が枝に対応する．命令区間の入出力セットを，その木構造を保ったまま RB 及び RA エントリに登録する事で，限られた容量の RB 及び RA エントリを有効に使用できる．

続いて，例 1 のプログラムを例に入出力セットの登録手順について述べる．例 1 のプログラム中の `main` 関数が最初から実行され，10 行目で関数 `array_mul` が検出される．関数を検出すると，MemoTbl 内の RF，RB，RA を参照し，当該関数に対応する

入力セットがMemoTbl内に存在するか否かを確認する．これを**入力値検索**と呼ぶ．関数array_mulは10行目の時点で初めて呼び出されるので，RFを検索しても対応する関数アドレスがまだRF上に存在しない．そのため関数array_mulのアドレスをRFに登録し，2行目から7行目までを通常通り実行する．自動メモ化プロセッサは，関数の実行中に出現した入出力をMemoBufに登録しながら命令の実行を進める．6行目のreturn文を検出し，関数の実行を終了する際，MemoBufに記憶した入出力セットをMemoBufからMemoTblへと一括して登録する．同様に12行目の関数array_mulの入出力セットもMemoBufに登録され，当該関数の実行終了時にMemoTblへと一括して登録される．

次に図7を用いて入力値検索の具体的な手順について述べる．プログラム例1では，10行目及び12行目で関数array_mulに対する入力値検索が失敗し，当該関数の入力セットが各RBエントリに登録されているとする．引き続き14行目まで例1のプログラムを実行し，14行目の関数array_mulの呼び出し命令を検出すると，命令実行を一時停止し入力値検索を開始する．14行目の関数呼び出しarray_mul(3，a，b)に対する入力値検索の手順は図7中にある数字(1)から(8)で示したものに対応する．まず，関数のアドレス及び引数の値をキーとしてRBを検索する(1)．本例ではRBエントリ00の値と一致する事が分かる．その後，今一致したRBエントリと同一エントリ番号のRAを参照し(2)，次の入力値が格納されているレジスタ番号又は主記憶アドレスを得る．本例では配列aの値を記憶している主記憶アドレス0400をRAエントリ00から得た後，検索キー00及び主記憶アドレス0400から読み出した値を用いて再度RBを検索する(3)．検索キーは命令区間の開始アドレス，もしくは入力を木構造として見た時の親ノードのRAエントリ番号によって一意に決定される．検索の結果，主記憶アドレス0400から読み出した値はRBエントリ02と一致するため，先程と同様にRAエントリ02から次の入力アドレス0500を得る(4)．同様の手順で検索を行い(5)(6)(7)，全ての入力が一致する事を確認すると(8)，入力値検索は成功となる．入力値検索が成功すると，終端エントリのW1ポインタを用いて出力セットをW1から読み出し(9)，レジスタやキャッシュへ書き戻す．以上の手順により，14行目の関数呼び出しarray_mul(3，a，b)に対して計算再利用を適用することができる．

2.2.2 オーバヘッドフィルタ

入力値の検索は，高速に連想検索可能なCAMを用いているとはいえ，そのオーバヘッドは大きい．また，再利用に関するオーバヘッドは入力値の検索に要する時間だけでなく，RBからキャッシュへの出力の書き戻しにかかる時間も存在する．これらのよ

うなオーバーヘッドを再利用オーバヘッドと呼ぶ．さて，再利用によって得られる効果が小さいような短い命令区間については，削減されるサイクル数よりも再利用オーバヘッドの方が大きい場合がある．このような命令区間に対しては，計算再利用を適用しないようにするため，SPARC ベースモデルの自動メモ化プロセッサには再利用オーバヘッドを動的に評価し，それに基づいて入力値検索の実施を決定する機構が備わっている．この機構をオーバヘッドフィルタと呼ぶ．

この機構は一定期間 T の RB へのエントリの登録回数 N を記録している．これらの値と当該命令区間の過去の省略サイクル数 S から，実際に削減できたサイクル数は

$$N \times (S - Ovh^R - Ovh^W) \tag{1}$$

として計算できる．ここで，Ovh^R,Ovh^W はそれぞれ，過去の履歴より概算した，入力値の検索によるオーバヘッドと，RB からキャッシュへの書き戻しオーバヘッドである．また，再利用が成功しなかった場合でも，入力値の検索によるオーバヘッドは存在する．このオーバヘッドは，

$$(T - N) \times Ovh^R \tag{2}$$

として計算できる．

このとき，発生したオーバヘッド (式 (2)) よりも削減できたサイクル数 (式 (1)) が大きいような命令区間は，再利用の効果が得られると考えられる．そこで，RF にいくつかのカウンタを付加することによってこれらを記録及び比較し，効果が得られないと判断された命令区間は再利用の対象とせず，RB への登録を行わない．

2.2.3 ARM モデルのベースアーキテクチャ

本項では，ARM ベースモデルを実装する上で用いられたベースアーキテクチャの詳細を説明する．ユーザモードでアクセス可能な論理レジスタの一覧を**表 1** に示す．これは，ユーザモードすなわちアプリケーション実行時に通常使用される保護モードにおいて，アクティブなレジスタを示している．ユーザーモードでアクセス可能な論理レジスタは 18 個であり，データレジスタは 16 個，プログラムステータスレジスタは 2 個である．プログラマは R0-R15 のレジスタを使用することができる．その中でも R0-R3 の 4 つのレジスタは関数の引数渡し及び返り値受取りに用いられる．それとは別に ARM プロセッサでは特別な役割を担うデータレジスタが 3 つある．それは R13，R14，R15 のレジスタである．順に R13 はスタックポインタ (sp) の格納場所であり，R14 はリンクレジスタ (lr) と呼ばれ，サブルーチンを呼び出す時に現在のプログラムカウンタ (pc) を格納し，サブルーチンからの復帰時に使用する．R15 はプログラムカウンタ格納レ

表 1: ARM の汎用レジスタ

レジスタ番号	使用用途
R0-R3	引数渡しレジスタ 1/作業レジスタ/返り値格納レジスタ
R4-R8	一時レジスタ
R9	一時レジスタ/静的ベースレジスタ
R10	一時レジスタ/スタック制限/スタックチャンク処理
R11	一時レジスタ/フレームポインタ
R12	ip(instruction pointer)/作業レジスタ
R13	sp(stack pointer) スタックフレームの最下部
R14	lr(link register)/作業レジスタ
R15	pc(program counter)
cpsr	条件コードレジスタ
spsr	条件コードレジスタの内容を待避するレジスタ

図 8: プログラムステータスレジスタ

ジスタであり，プロセッサが次にフェッチするアドレスを保持する．

プログラムステータスレジスタには cpsr（current program status register）と spsr(セーブド・プログラム・ステータス・レジスタ) の 2 つがある．spsr は cpsr を保存するために用いられる．一般的なプログラムステータスレジスタの基本レイアウトを**図 8** に示す．cpsr はフラグ用，ステータス用，拡張用，コントロール用の 4 つのフィールドに分かれ，それぞれが 8bit 幅である．フラグフィールドには条件フラグがある．ステータスフィールドとコントロールフィールドは将来の使用に備えて予約されているコン

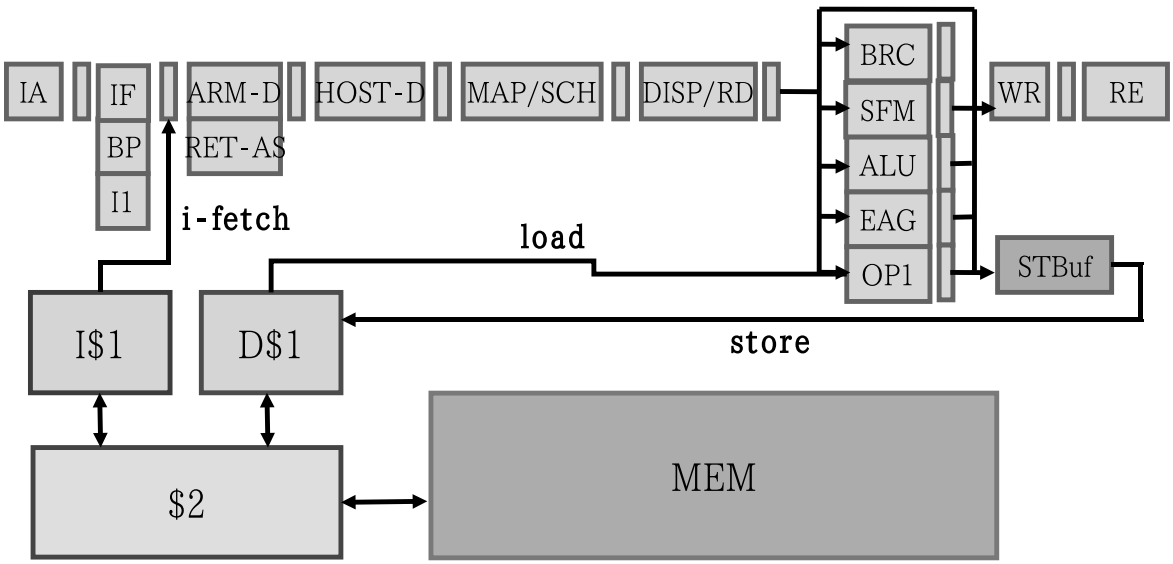


図 9: ベースアーキテクチャのハードウェア構成図

トロールフィールドには，プロセッサモードビット，Thumb ステートビット，割り込みマスクビットがある．プロセッサ・モードはどのレジスタがアクティブで，cpsr レジスタ自体へのアクセス権を持つかを決定する．

次に，**図 9** にベースアーキテクチャのハードウェア構成を示す．ベースアーキテクチャには「OROCHI[6]」を用いている．OROCHI は VLIW 命令と ARM 命令を同時実行可能なスーパースカラ型プロセッサである．一般に，プログラムが持つ ILP には限界があり，高性能なコンパイラを用いたとしても VLIW 命令内には多数の NOP 命令が混入してしまい，プロセッサの使用率は低下してしまう．そこで OROCHI は，NOP 命令により空いている演算器に元々ILP が期待できないようなスレッド (例えば Kernel スレッド) の命令を投入することにより，プロセッサ全体の使用率を向上させるということを意図して設計された．つまり，OROCHI プロセッサは VLIW 命令と ARM 命令それぞれ別に，命令フェッチ，デコード等を行うフロントエンドを持ち，実行ステージを含むバックエンドは 1 つというハードウェア構成になっている．この VLIW 命令用のフロントエンドは，スーパースカラ型自動メモ化プロセッサを実装する上では必要ないため無効化されており，図 9 でも省略している．ベースアーキテクチャのパイプラインはパイプラインレジスタを含め全 19 段で構成され，2 次キャッシュは命令キャッシュと 1 次データキャッシュに共有される．次に，各パイプラインステージについて簡易的に説明する．

IA(Instruction Address)

プログラムカウンタの計算を行う．

IF(Instruction Fetch)・BP(Branch Prediction)

IF ステージでは命令キャッシュから命令をフェッチする．BP ステージでは g-share による分岐予測を行い，その結果を IA ステージへと伝達する．

ARM-D(Arm-Instruction Decode)

ARM 命令を高速に実行可能な内部命令に分解する．例えば，マルチ LD 命令は，複数の 4byte の LD 命令に，乗算命令などは，複数の加算命令とシフト命令に分解される．

HOST-D(Host-Instruction Decode)

条件実行命令を実行ステージで処理可能なように分解する．条件実行命令とは，命令の実行に条件を付加する ARM 独自の命令である．ARM のすべての命令は，実行時に条件フラグを参照し，フラグが立っているときのみ実行するように出来る．OROCHI では，条件実行命令は 2 命令に分解される．分解後命令は，一方は条件が真の場合に実行される命令であり，もう一方はその命令が実行される条件をチェックする CSL 命令である．実行条件が満たされていない場合，前者の命令は破棄される．

MAP/SCH(Register mapping/Schedule)

各命令のオペランドに対して，汎用レジスタ (EREG) と拡張レジスタ (IREG) で構成される論理レジスタから，命令ウィンドウと物理レジスタを兼ねる RoB(Reoder Buffer) へマッピングされる．また，同時に各命令のスケジューリングも行われる．

DISP/RD(Dispatch and Read)

命令の発行を行うステージである．各演算ユニットからオペランドのバイパスが可能かどうか調べ，依存関係にあるオペランドを持つ命令の発行を制御する．

IE(Instruction Execution)

ベースアーキテクチャは 5 並列の実行ステージを持つ．また，本ステージの各ユニットは前段に 5 段の入力 FIFO を，後段に 5 段の出力 FIFO を持っている．以下，各ユニットについて記述する．

BRC

分岐の taken/untaken を判定する．

SFM

シフト演算を処理する．また，積和補助演算の処理も担当する．

ALU

加減算命令を処理する．

EAG

アドレス計算を処理する．また，積和補助演算の処理も担当する．

OP1

ロード，ストア命令を処理する．

WR(WriteBack)

RoB への書き込み処理を行う．

RE(Retire)

先行命令がすべて完了した命令を RoB から論理レジスタへ書き戻し処理を行う．なお，分岐予測ミス時等に実行再開をすみやかに行うために，命令がコミットされる順番は，分解前命令列と同じであることが保証されている．

OROCHI では ARM 命令を RISC 型内部命令へ分解する．ARM 命令セットは RISC 設計でありながら，1 命令で複雑な処理も可能な命令形態となっている．その一方で，パイプラインで命令を効率よく実行するには，命令が常に同じサイクルで実行されるほうが良いとされている．そのため，個々の命令は簡単な処理のみを行い命令ごとの処理量が均等であることが望ましい．そこで OROCHI では，ARM 命令を簡単な処理を行う内部命令へと変換，分解し，パイプラインの各ステージの処理量が均等になるようにしている．

さて，一般的な ARM プロセッサが持つ論理レジスタについては先述したが，ベースアーキテクチャには，これらのレジスタに加えて，6 個の拡張論理レジスタが存在する．この拡張論理レジスタは，ARM 命令分解後の内部命令が使用する論理レジスタである．この内部命令用のレジスタを IREG（Inplicit Register）と呼び，対して ARM プロセッサで規定される R0-R15 までの汎用論理レジスタを EREG（Explicit Register）と呼ぶ．IREG は，内部命令がデータを一時保存するために用いられる．

2.2.4　ARM ベースモデルの実装

ARM ベースモデルは，以上のアーキテクチャをベースとして実装されている．**図 10** に ARM ベース自動メモ化プロセッサのハードウェア構成を示す．MemoTbl 及び MemoBuf の基本的な構造は SPARC ベースモデルに準ずる．ARM ベースモデルには，SPARC ベースモデルで実装された MemoTbl や MemoBuf の他に，メモ化に必要な独自の追加ハードウェアとして関数の入出力登録及び入力値検索開始ユニットと関数呼び出し及び復帰検知デコーダがある．以下，それぞれについて説明する。

関数の入出力登録及び入力値検索開始ユニット

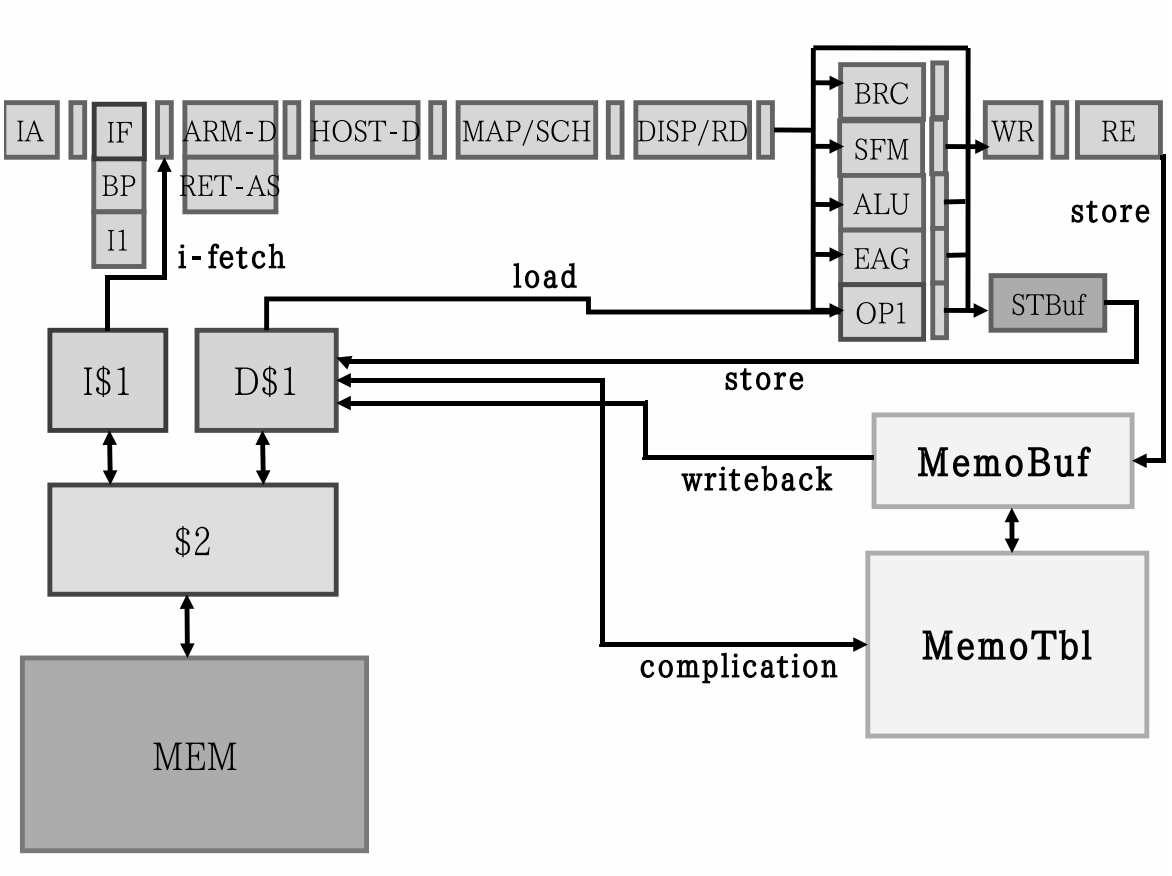
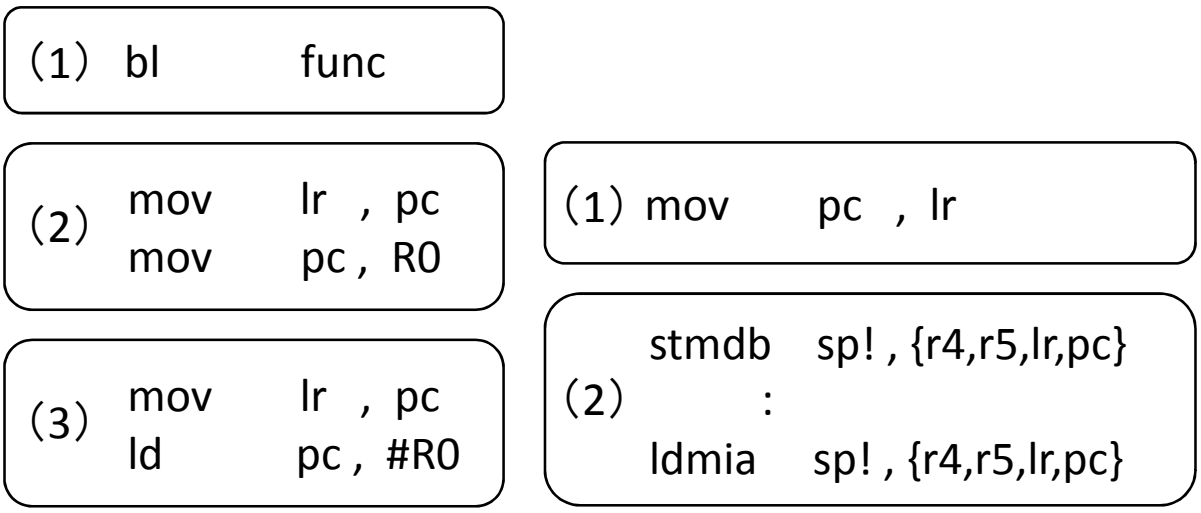


図 10: ARM ベースモデルのハードウェア構成

プログラムに関数再利用を適用するためには，入力値の検索を行ってから実際に関数が呼び出されるまでの間に，その入力値が異なる値で上書きされることがないことを保証しなくてはならない．もし上書きが起こったとすると，上書き後の値が関数の入力値として適正ということになり，上書き前の値を用いた入力値検索は誤った処理だということになる．

そこで，入力値検索は関数呼び出し命令に先行する命令が完了した後に行うことにする．これにより，入力値検索をしてから関数が呼び出されるまでの間に入力が別の値で上書きされることがなくなる．関数呼び出し命令の先行命令が完了しているとは，すなわち関数呼び出し命令がコミット可能であるということである．つまり，関数呼び出し命令がコミットされるとき，その先行命令はすべて完了しているということである．以上から，ARM ベースモデルでは関数呼び出し命令がコミットされた時点で入力値検索を行うこととしている．その際，もし関数呼び出し命令の後続命令がコミット可能になっていたとしても，後続命令のコミットはキャンセルされる．それは，その



(a) 関数呼び出し　　(b) 関数復帰

図 11: ARM の関数呼び出し，復帰コード

命令をコミットしてしまうと入力値を上書きしてしまい，再利用テストが正しく行えない可能性があるためである．入力値検索の後，入力値が全て一致し再利用テストが成功すれば，パイプラインはフラッシュされる．再利用テストが失敗した場合も，コミットがキャンセルされた後続命令は RoB に残り，次サイクル以降にコミットされる．

関数の入出力登録もまた，RETIRE ステージにて，関数の入出力を格納する命令がコミットされた時点で行う．命令がコミットされるより前に入出力の登録を行うと，分岐予測ミスなどで実行された命令がキャンセルされた場合，それに応じて MemoBuf に登録した入出力もキャンセルする必要があり，ハードウェアが複雑となってしまうためである．以上のように，関数の入出力登録，及び入力値検索開始ユニットは，RETIRE ステージを拡張することで実装されている．

関数呼び出し，復帰検知デコーダ

関数をメモ化するためには，関数の呼び出し及び復帰を検知する必要がある．SPARC ベースモデルでは，SPARC の ABI により関数呼び出しは call 命令，復帰命令は ret 命令と規定されているため，call 命令と ret 命令の実行を監視することにより，メモ化対象区間を特定している．

一方，ARM ABI では関数呼び出し及び復帰に特定の命令を使用しなくてはならないと規定されているわけではない．主に関数呼び出しに用いられる命令として bl(branch and link) 命令が存在するが，関数呼び出し時には必ずしもこの命令を用いなければならないわけではない．**図 11**(a)，(b) に ARM の関数呼び出し，及び復帰コードの様々

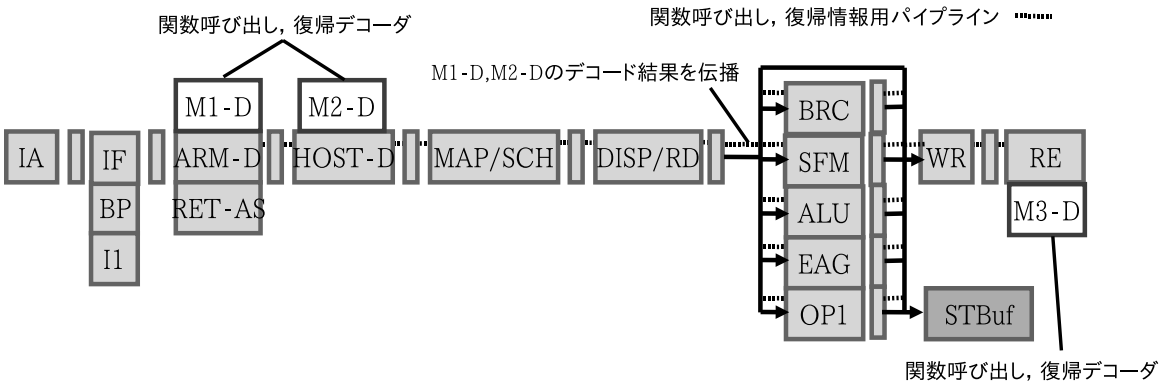


図 12: ARM の関数呼び出し，復帰デコーダ

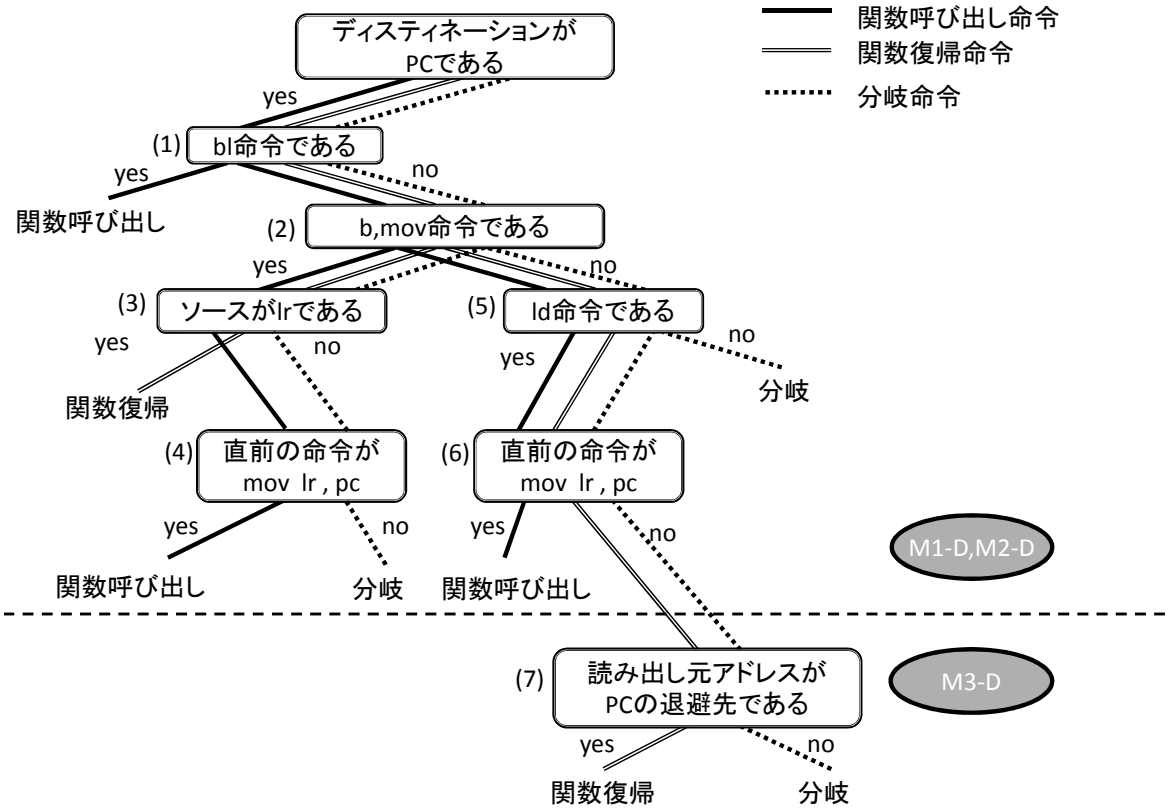


図 13: 関数呼び出し，復帰検知デコーダの動作アルゴリズム

な形態を示す．このように，ARM の関数呼び出し，復帰コードは多岐に渡るのため，SPARC モデル同様に単純に特定の命令を監視するだけでは，メモ化対象区間を特定することができない．そのため，メモ化対象区間特定のために，SPARC ベースモデルに比べ複雑な順序回路を用いる必要がある．

ARM ベースモデルでは，関数呼び出しと関数復帰を検知するために ARM-DECODE

ステージ，HOST-DECODE ステージ，RETIRE ステージに関数呼び出し及び復帰検知回路とパイプラインレジスタを**図 12** に示すように追加している

この機構の動作アルゴリズムを**図 13** に示す．このアルゴリズムは，ディスティネーションレジスタがプログラムカウンタの命令を対象として，それが関数呼び出し命令，関数復帰命令，分岐命令のいずれであるかを判定する．各命令の判定までパスを，図中右上の書式に従って示す．本機構は以下のように動作する．まず，ARM-DECODE ステージと HOST-DECODE ステージの検知回路 (M1-D,M2-D) はプログラムカウンタを上書きする命令を監視する．検知された命令が bl 命令ならば，これは図 11(a)(1) のパターンに該当するので，関数呼び出し命令であると判定される (図 13(1))．この結果は RETIRE ステージまでパイプラインレジスタを介して伝えられる．bl 命令でない場合，次にその命令が b 命令または mov 命令であるどうかと，ld 命令であるかどうかを調べる (図 13(2)，(5))．b 命令または mov 命令であるならば，次にソースオペランドレジスタを調べる (図 13(3))．ソースオペランドレジスタが関数復帰アドレスを保持する lr であるならば，これは図 11(b)(1) のパターンに該当するため，関数復帰命令であると判定される．そうでなかった場合，次に直前の命令を調べる (図 13(4))．それが関数復帰アドレスを待避する命令 (mov lr，pc 等) であれば，これは図 11(a)(2) のパターンに該当するので，関数呼び出し命令であると判定される．そうでない場合，関数呼び出し及び復帰命令のいずれにも該当しないため，その命令は単なる分岐命令であると判定される．図 13(5) での判定で ld 命令と判明した場合，図 13(4) と同様に直前の命令が関数復帰アドレスを待避する命令かどうかを調べる (図 13(6))．そうであるならば，これは図 11(a)(3) のパターンに該当するので，関数呼び出し命令であると判定される．そうでなかった場合，最後にその ld 命令の読み出し元アドレスが，関数プロローグにてプログラムカウンタが退避されたアドレスと一致するかどうかを比較する (図 13(7))．これは，図 11(b)(2) のパターンのような，リンクレジスタを介さずに関数プロローグにて関数復帰アドレスをスタックへ退避し，それを関数エピローグにて ld 命令で直接プログラムカウンタを関数復帰アドレスで上書きする関数復帰パターンが存在するためである．この比較を行うため，RETIRE ステージの検知回路 (M3-D) は関数プロローグにてプログラムカウンタの値が退避されたアドレスを記憶する．そのアドレスから値を読み出す ld 命令を検知したとき，その ld 命令は図 11(b)(2) に示す関数復帰命令だと判定される．以上のいずれにも該当しないプログラムカウンタを上書きする命令は，すべて単なる分岐命令だと判定される．

表 2: SPARC ベースモデルの評価パラメータ

L1 cache	miss penalty	10 cycles
	size	32 Kbytes
	line size	32 bytes
	way 数	4 ways
L2 cache	miss penalty	100 cycles
	size	2 Mbytes
	way 数	4 ways
	ロードレイテンシ	2 cycles
	Register Window 数	4 sets
	Window ミスペナルティ	20 cycles
演算レイテンシ	整数乗算	8 cycles
	整数除算	70 cycles
	浮動小数点加減乗算	4 cycles
	単精度浮動小数点除算	16 cycles
	倍精度浮動小数点除算	19 cycles
CAM	size	4 Klines
	line size	32 bytes
入力値検索コスト	Register	1 cycle
	L1 cache	2 cycle

2.3 予備評価

2.1 節と 2.2 節で述べた 2 つの既存モデルの評価を行った．評価時の SPARC ベースモデルの各パラメータを**表 2.3** に，ARM ベースモデルのものを**表 2.3** に示し，**図 14** に SPEC CPU95 による評価結果を示す．横軸がプログラム名を示し，縦軸はメモ化無しの通常実行を 1 として正規化した実行時間を示している．各実行プログラム毎の 2 本のグラフは左が SPARC ベースモデル，右が ARM ベースモデルを示す．

プロセッサ実行時間は，キャッシュミスペナルティやレジスタウィンドウミスを含むプロセッサ実行時間を表す．再利用オーバーヘッドは入力値の比較にかかった時間を示し，レジスタと CAM との比較，キャッシュと CAM との比較時間を意味する．再利用ストールは ARM ベースモデル独自のオーバーヘッドで，再利用テスト時に発生する

表 3: ARM ベースモデルの評価パラメータ

A1 cache	miss penalty	8	cycles
	size	16	Kbytes
	line size	64	bytes
	way 数	4	ways
L1 cache	miss penalty	8	cycles
	size	32	Kbytes
	line size	64	bytes
	way 数	4	ways
L2 cache	miss penalty	40	cycles
	size	2	Mbytes
	line size	64	bytes
	way 数	4	ways
pipeline	命令フェッチ (IF)	1	命令/cycle
	デコード (ID)	2	命令/cycles
	命令分解能 (AD)	4	内部命令/cycle
	命令分解能 (HD)	4	内部命令/cycle
	レジスタマッピング (MAP)	4	内部命令/cycle
	命令発行 (SEL/READ)	4	内部命令/cycle
	命令実行 (EXE)	1	内部命令/cycle
	物理レジスタ書き込み (WR)	1	内部命令/cycle
	論理レジスタ書き込み (RE)	4	内部命令/cycle
	Reorder buffer	32	entry
CAM	size	4	Klines
	line size	64	bytes
入力値検索コスト	Register	1	cycle
	L1 cache	2	cycle

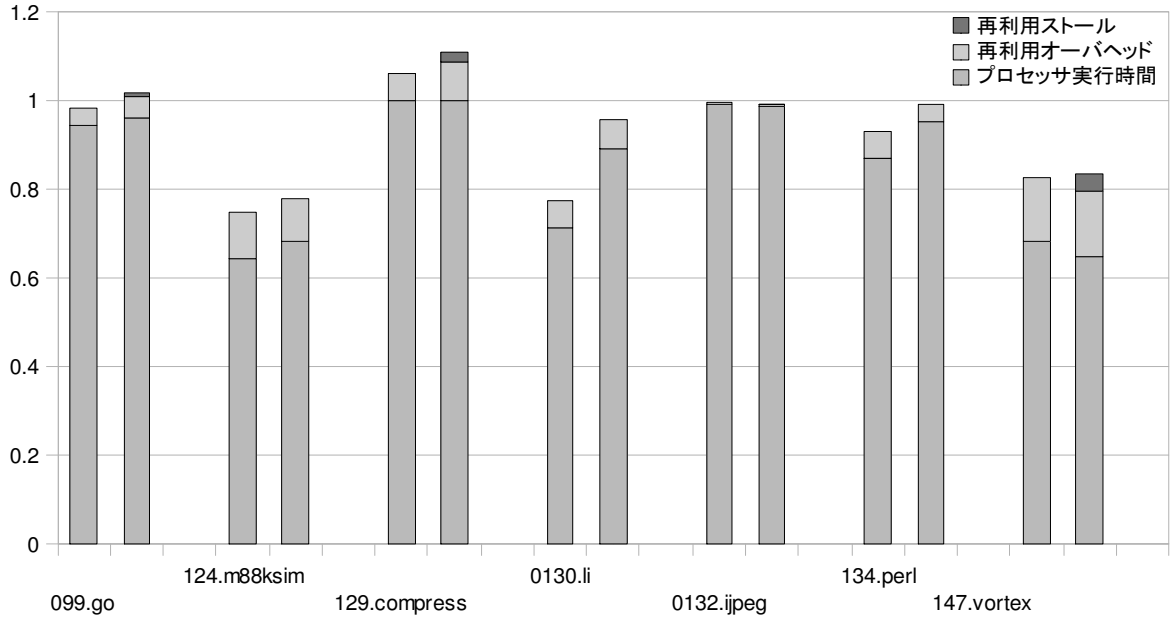


図 14: 予備評価

ストールや，その成功によって発生するバブルによるペナルティを意味する．SPARC ベースモデルでは平均 5.4%，124.m88ksim において最大 17.5%の高速化を確認した．ARM ベースモデルでは平均 2.1%，SPARC ベースモデルと同じく 124.m88ksim において最大 16.8%の高速化を確認した．再利用による実行時間の削減率に関して，両モデルは同様の傾向を示していることがグラフから読み取れる．しかし ARM ベースモデルでは，再利用適用時に発生するパイプラインフラッシュの影響から，プログラム全体の実行時間は増加してしまっている．このフラッシュにより増加したオーバヘッドを隠蔽することが出来れば，ARM ベースモデルは平均 5.8%，最大 21.0%の実行サイクル数を削減でき，SPARC ベースモデルの削減率を上回る．

3 提案手法

3.1 既存モデルの問題点

本節では，既存の ARM ベースモデルの問題点とその原因を考察し，それを解決する提案手法を述べる．2.3 節で述べた通り，既存の ARM ベースモデルでは再利用成功時にバブルが発生する．これは，再利用テストを行うステージが RETIRE ステージであることに起因する．関数呼び出し命令が RETIRE ステージまで到達し，関数の再利用が成功したとする．関数の再利用が成功した時点でパイプライン中を流れている命令は，本来呼び出されるはずだった関数の内部の命令である．それらの命令を破棄す

るため，パイプラインはフラッシュされ，その後関数復帰後の状態からプログラムの実行が再開される．そのために，すでにフェッチ済みであった命令はバブルとなり，その分だけオーバーヘッドが発生してしまう．

再利用テストを RETIRE ステージで行っているのは，先行命令をすべてコミットすることによって，再利用テストを行うのに必要な入力情報がすべて揃っていることを保証するためである．MemoTbl に登録されている関数の入力値は，過去の関数呼び出し命令がコミットされたときの値である．そのため，再利用テストの際に比較する各入力値もまた，関数呼び出し命令がコミットされるときのものでなくてはならない．入力がこの状態であることを，本稿では**入力が Ready である**状態と呼ぶ．既存モデルでは，関数呼び出し命令が実際にコミットされるのを待つことによって，全ての入力が Ready であることを保証している．

先に述べた通り，入力が Ready であることを保証するために，既存モデルでは再利用テストを RETIRE ステージで行っている．これを逆説的に考えると，入力が Ready であることが保証出来れば，再利用テストはより前段のステージで行うことが可能となる．そこで本研究では，再利用テストを行うタイミングを改良することによって再利用成功時に発生するオーバヘッドを削減することを目的とし，先行命令のコミット以外によって入力 Ready を保証する方法を提案する．

3.2　提案する再利用テスト

本節では，提案する再利用テストを実現するための方法を，関数検索，入力値検索，出力書き戻しの 3 つの観点から考える．

3.2.1　関数検索

再利用テストには，MemoTbl からの関数検索と入力値一致比較の 2 つの処理がある．これらのうち関数の入力値を用いる処理は入力値一致比較だけである．つまり，入力の Ready は入力値一致比較のときに満たされていればよく，関数アドレスの検索に対しては当てはまらない．関数の検索は，呼び出される関数のアドレスが判明した時点で可能となる．関数のアドレスが判明するタイミングは 2.2.4 項で述べたように関数を呼び出す命令ごとに異なる．その一方で，詳細は後述するが，再利用テストが可能となるのはその関数呼び出し命令が発行可能になってからである．以上から，関数のアドレスの検出は命令発行ステージ (DISP ステージ) 以降の各ステージで行うものとする．

また，再利用テストを DISP ステージ以降にて行うということは，再利用テストを行う時点で関数呼び出し命令が DISP ステージに到達しているということである．これに

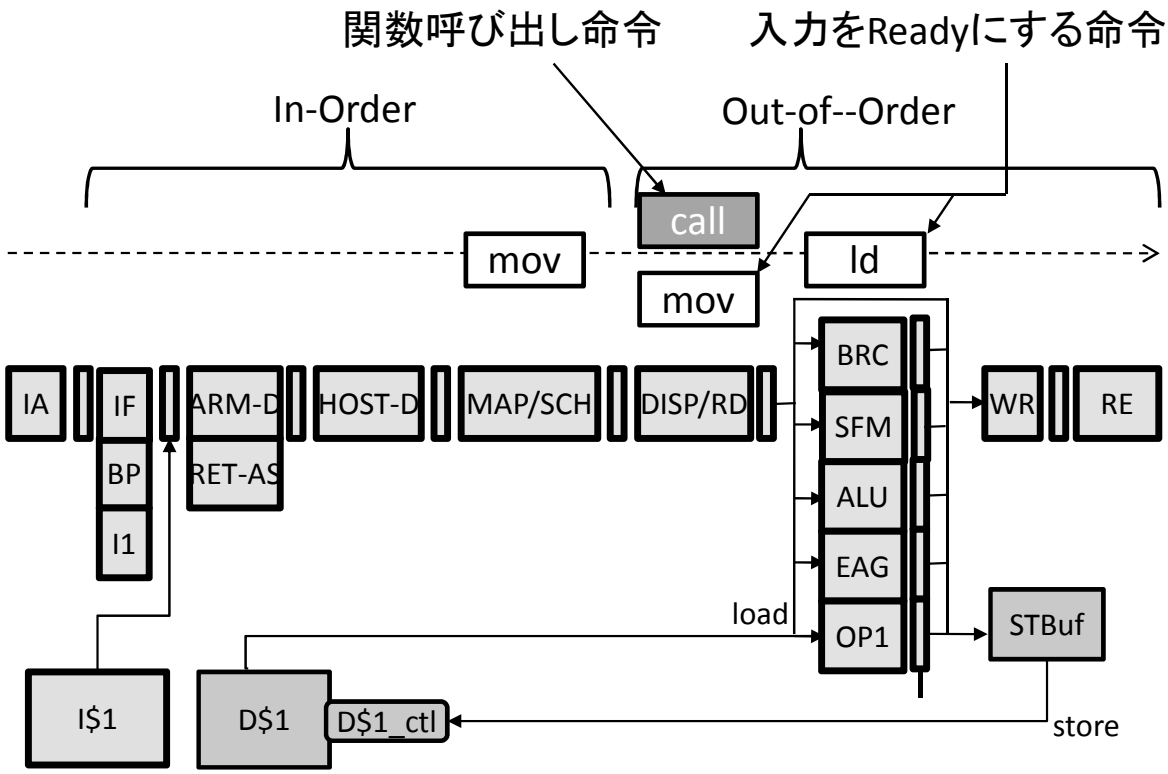


図 15: パイプライン中の命令の配置

よって，入力を Ready にする命令が必ず DISP ステージ以降に存在することを保証できる．パイプライン中の命令の配置の様子を**図 15** に示す．図中の下部にベースアーキテクチャのパイプラインを，上部にパイプラインを流れる命令を示す．図 15 は関数呼び出し命令が DISP ステージに到達した直後の様子を示している．ベースアーキテクチャのパイプラインは Out-of-Order 実行であるが，実際に Out-of-Order に実行されるのは DISP ステージ以降のみであり，MAP ステージ以前では命令は In-Order で進む．よって，DISP ステージまで到達している命令の先行命令は必ず DISP ステージより後段に存在する．入力を Ready にする命令は In-Order では必ず関数呼び出し命令に先行するので，関数呼び出し命令が DISP ステージに到達しているのであれば，入力を Ready にする命令もまた図 15 のように DISP ステージ以降に存在する．

3.2.2 入力値検索

既存モデルでは入力が Ready であることを，全先行命令のコミットによって保証していた．ここで，DISP ステージで再利用テストのための関数アドレス検出を行うことを考える．仮に入力値 Ready の保証方法を変更しないまま再利用テストを行うステージを変更すると，関数呼び出しに先行する命令のコミットを待つ必要がある．関数呼

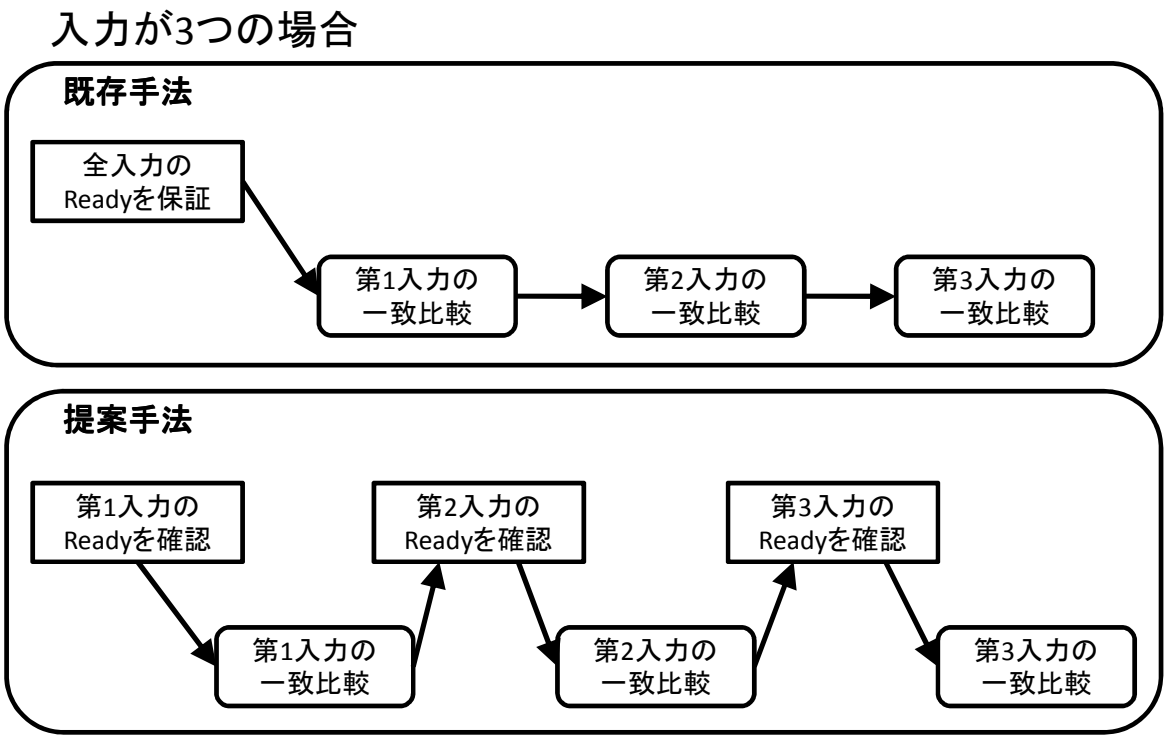


図 16: 入力値一致比較の手順

び出し命令とその後続命令をストールさせる場合，先行命令のコミットが完了するまでの間にバブルが発生し続けることになる．また，この手法は関数の入力値とは関係の無い命令のコミットも待つ必要がある．

そこで，関数の入力値の格納箇所ごとに，それをディスティネーション (DST) オペランドとする命令が存在しないかを確認することとする．それにより，各入力が Ready であることを保証する．既存手法と提案手法のそれぞれについて入力値一致比較の手順を**図 16** に示す．この図では，入力が 3 つの場合を考えている．既存手法では，最初にすべての入力値が Ready であることを予め保証する。具体的には、すべての先行命令をコミットさせることによって，その保証を取っている。その後、各入力の一致比較を行う．この方法は工程数が少なく機構が単純で済む一方で，すべての先行命令のコミットを待たなくてはならない．またそのために，関数入力が Ready となるのに特に関係が無い先行命令のコミットを待つ必要がある．一方提案手法では，最初は第 1 入力の Ready のみ確認し，それが確認でき次第第 1 入力の一致比較を行う．それが一致したならば第 2 入力の Ready を確認し，その後第 2 入力の一致比較を行う．第 3 入力についても同様に，入力が Ready であることの確認と入力値の一致比較を行う．このように入力ごとに，その入力が Ready であることの確認とその入力値と MemoTbl に登

録されている入力値の一致比較を繰り返す．こうすることにより，手順は多くなり機構は複雑となるものの，可能なかぎり早いタイミングで入力の一致比較が行える．加えて，関数入力が Ready となるのに特に関係が無い命令のコミットを待つ必要が無くなる．

さて，入力が Ready であるとは，即ち関数が実際に呼び出されるまでの間に，入力値を書き換える命令が存在しない状況である．前項で示したように再利用テストのための関数アドレス検出を DISP ステージで行う場合，DISP ステージから入力値を検出する箇所 (従来は RETIRE ステージ) までの間に，そのような命令が存在しないことが確認出来れば，入力が Ready であることを保証できる．また，既存モデルではどの先行命令が入力を Ready にするのかを確認していなかったため，本質的にはコミットを待つ必要が無かった命令についてもコミットを待っていた．本手法では入力ごとに Ready であるかどうかの確認を取るため，関数の入力とは関係の無い命令について待つ必要が無くなる．

3.2.3 出力値書き戻し

従来は再利用テストを行うのは関数呼び出し命令のコミット時であった。このとき，再利用テストが行われた関数は，本来は呼び出されることが確定している．そのため再利用テストが成功したならば，無条件で再利用を適用し，プログラムをその関数の実行が完了した状態にしても問題がなかった．しかし，提案モデルでは関数の出力を書き戻してもプログラムの正しい実行を維持できるかを確認する必要がある．なぜなら，コミットしていない命令はコミットせず破棄される可能性があるためである．

再利用テストを DISP ステージにて開始する場合，関数呼び出し命令が発行されながらもコミットされない状況が存在するため，再利用対象の関数が実際に呼び出されるかどうかを考慮しなくてはならない．そのような状況は 2 通り存在する．

1 つは，先行する分岐命令の分岐予測が失敗していた場合である．その場合，その分岐命令の後続命令は実行されないため，通常実行時には関数呼び出し命令は破棄され，関数呼び出しは起こらない．もう 1 つは，関数呼び出し命令が条件実行命令であり，かつその実行される条件が満たされていない場合である．この場合もまた関数呼び出し命令は破棄され関数呼び出しは起こらない．これらのような場合に関数の再利用を適用してしまうと，本来は呼び出されないはずの関数が呼び出されたあとの状態になってしまう．これはプログラムの実行として正しくないため，このようなケースでは再利用テストが成功しても関数の出力を書き戻さないようにする必要がある．そこで，再利用が成功しても，関数が呼び出されることが保証されるまでは関数の出力

を書き戻さないようにする．これによって，誤った再利用の適用を防ぐ．

またこの他にも，関数再利用を適用できない場合が存在する．それは，関数復帰アドレスが分からない場合である．自動メモ化プロセッサは，関数再利用を適用したとき PC の値を関数復帰アドレスで上書きする．その関数復帰アドレスが正しく参照できなくては，関数の出力が書き戻されたとしてもプログラムを誤ったアドレスから再開する可能性がある．そのため，関数復帰アドレスがリンクレジスタへと格納されるまでは関数再利用を適用できない．そこで，リンクレジスタへ PC を退避する命令を監視する．これにより，誤った関数復帰アドレス参照を防ぐ．

3.3 動作モデル

本節では，提案モデルの動作の様子を述べる．まずはじめに，再利用テストに成功した場合の動作から述べる．**図 17** に，提案モデルが再利用テストに成功した際の動作の様子を示す．図の見方やパイプラインの構成ステージは，2.1.3 項の図 4 に準ずる．実行されるプログラムで呼び出される関数の入力値は，引数レジスタ R0 と R1 を介して入力される．また，再利用テストを開始するステージは，この図では De ステージに相当するものとしている．これは，提案モデルで関数検索を行っている DISP ステージが命令実行の直前のステージであるためである．

さて，プログラムが実行され，関数呼び出し命令がデコードステージまで到達し関数検索が行われたとする (図 17 中 (t1))．まずはじめに，入力が Ready であるかの確認と，入力値の一致比較が行われる．2.1.3 項で示した既存モデルではここまでに 4 サイクルを要したが，提案モデルでは 2 サイクルで済む．すべての入力が Ready でありそれぞれについて入力値が一致したとすると，関数再利用が適用される (図 17 中 (t2))．その後，関数呼び出し命令とその後続命令のみを無効化する (図 17 中 (t3))．

このとき，パイプライン中には関数呼び出し命令に先行する mov 命令がコミットを完了しないまま残留している．この命令は再利用テストの正否に関らず実行されなくてはならず，既存モデルのようにパイプライン中の全命令を破棄するとプログラムの動作に異常をきたす．そのため，この命令をこの先行命令はパイプライン上で有効なまま残す．その後関数の出力が書き戻され，関数復帰後の後続命令がフェッチされる (図 17 中 (t4))．ここで，再利用成功時に発生するバブルに着目をする．既存モデルでは，再利用成功時にはパイプライン中の全ての命令を無効化しなくてはならなかった．しかし提案モデルでは関数検索を行うステージを早めたことにより，再利用成功時に発生するバブルの量が低減している．

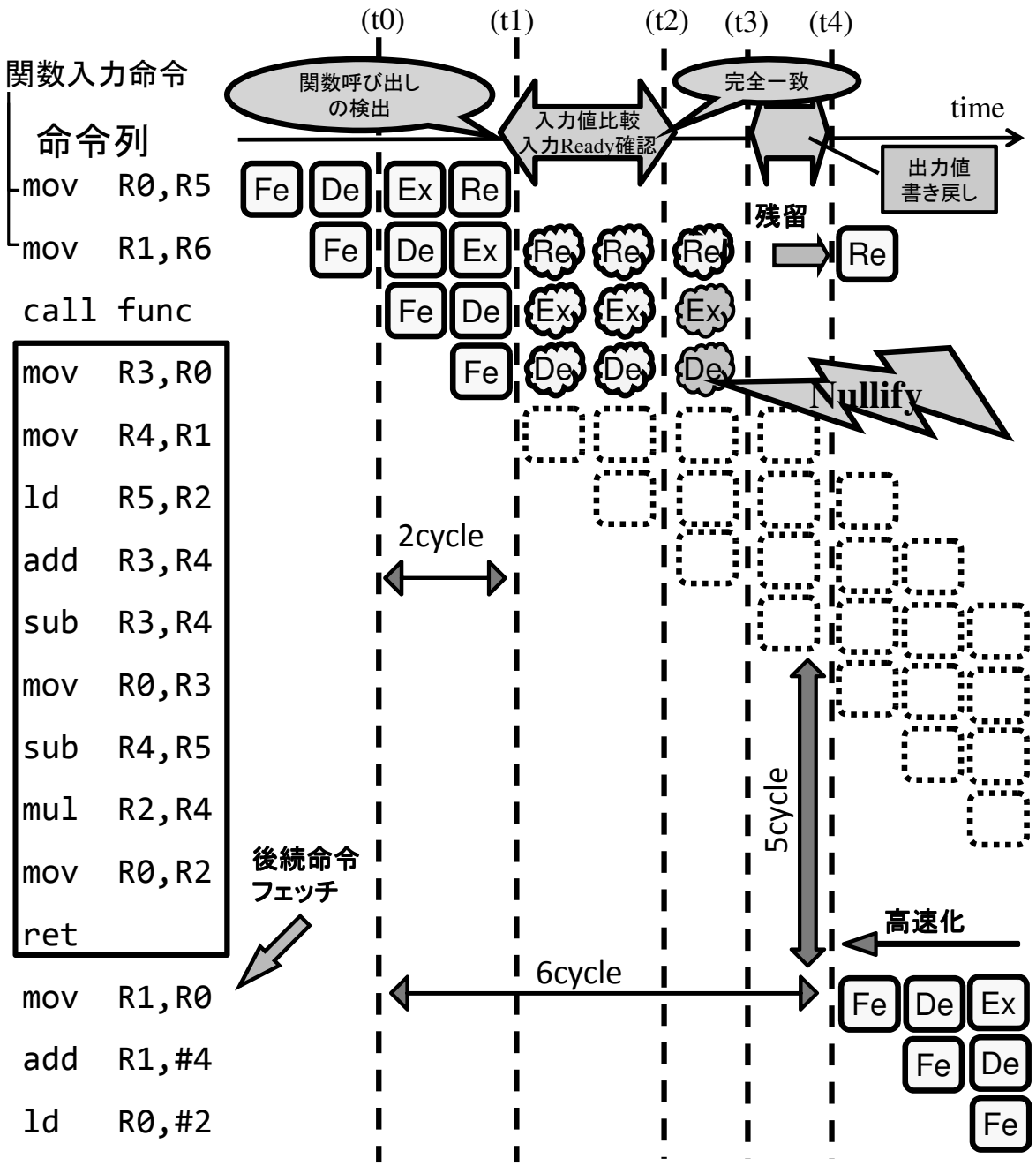


図 17: 提案モデルの動作の様子:再利用成功時

また，既存モデルと比べて早い時刻から再利用テストを行うことにより，図 17 の例では，関数呼び出し命令をフェッチしてから関数復帰後の後続命令をフェッチするまでにかかる時間 (図 17(t0)-(t4)) は 6 サイクルとなり，通常実行時と比べて 5 サイクル，2.1.3 項の図 5 の既存モデルと比べて 2 サイクル短縮されている．加えて，図 17(t4) の時点で先行命令がパイプライン上に存在しており，既存モデルよりもスループットが

上昇している．関数出力を書き戻した時点でパイプライン上に残っている命令が多いほど，このスループット向上は大きくなる．図 17 では，コミットが完了していない関数の入力となる命令が残っている．この他にパイプライン上に残り得る命令としては，関数呼び出しとは関係の無い命令が考えられる．関数の入力がすべて Ready になったあとにパイプライン上に関数呼び出し命令の先行命令が存在する場合，提案モデルではそのような命令をパイプライン上に残したまま再利用を行う．

次に，再利用テストの失敗を再利用テストの途中で検知した際の動作を**図 18** に示す．まず，関数呼び出し命令がデコードされたときに再利用テストが開始されたとする (図 18 中 (t1))．次に入力が Ready であるかの確認と入力値の比較を行うのだが，そこで再利用テストが失敗することが判明すると，再利用テストを中止し，プログラムの実行を再開する (図 18 中 (t2))．このように再利用テストが途中で中止される場合も，既存モデルではすべての入力が Ready 状態になるのを待っていた．しかし提案モデルでは，入力ごとにその入力が Ready かどうかを確認して再利用テストを行う．そのため，余分に命令の実行を待つ必要が無くなっている．そのため，関数呼び出し命令をフェッチしてから再利用テスト失敗が判明するまでにかかる時間 (図 18(t0)-(t2)) は 3 サイクルとなり，既存モデルの図 6 の例と比べて短縮されている．

4　実装

4.1　関数検索

本節では，再利用テストのうち関数検索に必要な情報である関数アドレスの検出方法を解説する．2.2.4 節で述べたとおり，ARM には関数呼び出し命令に様々なパターンが存在する．それらのコードパターンはそれぞれ関数アドレスの格納箇所が異なり，そのために関数アドレスが検出可能なステージも異なる．これらの各コードパターンと，それに応じた関数アドレス検出箇所，それらと MemoTbl の接続の様子を**図 19** に示す．

1 つ目は bl 命令などのように，関数アドレスがプログラムに記述されている場合である (図 19(1))．この場合，命令が関数呼び出しであることを検知した時点で必ず関数アドレスを検出できる．その一方で，本提案では再利用テストを行う時点で関数の入力を Ready にする命令が DISP ステージよりも後段にあることを保証する必要がある．このために，DISP ステージにて関数アドレスを読み出すものとする．

2 つ目は mov 命令などのように，関数アドレスがレジスタに格納されている場合である (図 19(2))．この場合関数アドレスをレジスタから読み出す必要があり，その読み

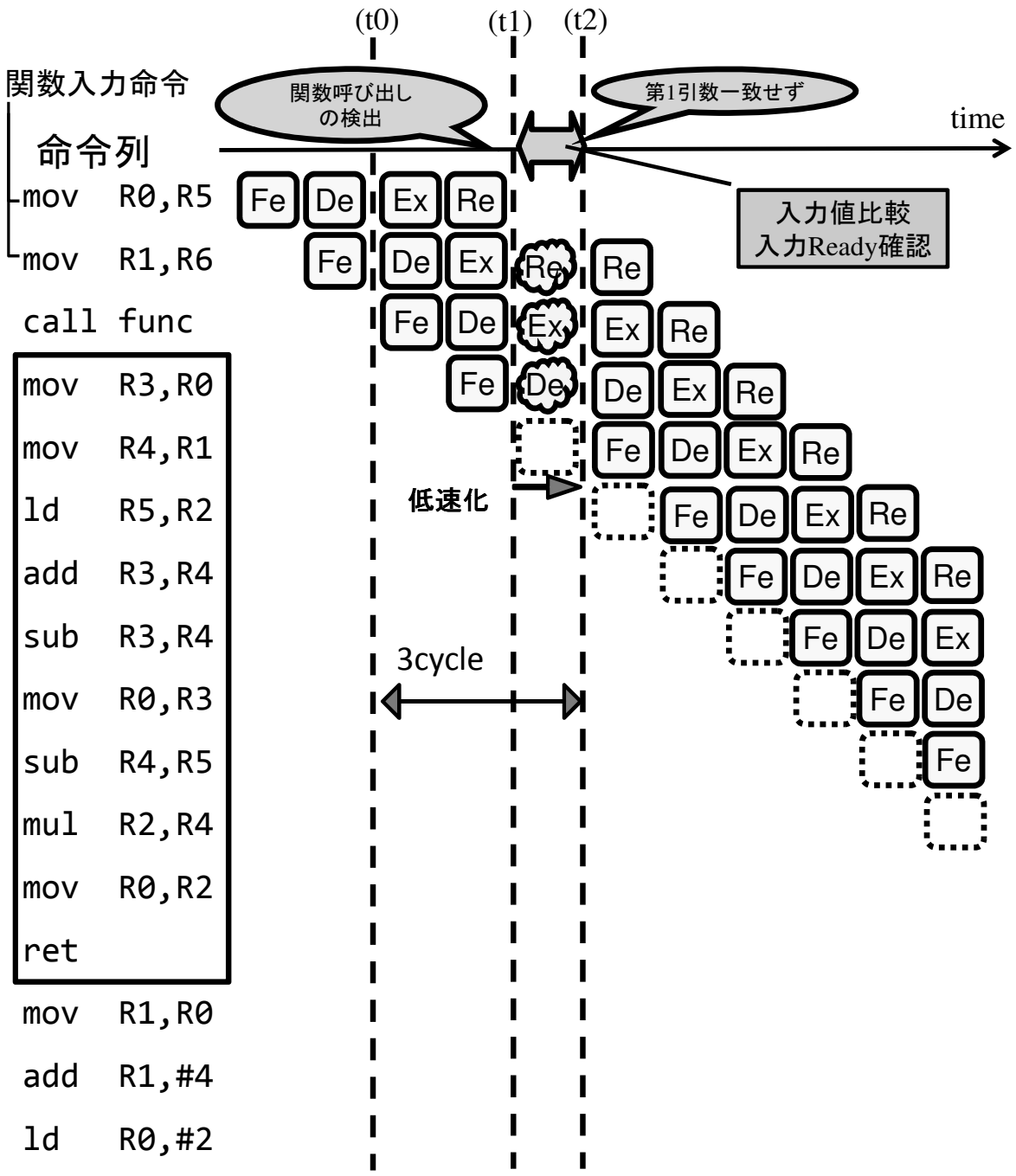


図 18: 提案モデルの動作の様子:再利用失敗時

出し元は関数呼び出し命令のソースオペランドとして指定されている．関数アドレスが読み出し可能な状態とは，すなわち関数呼び出し命令が実行されるまでソースオペランドが他の命令によって上書きされない状態である．これは関数呼び出し命令が発行可能かどうかで判別できる．

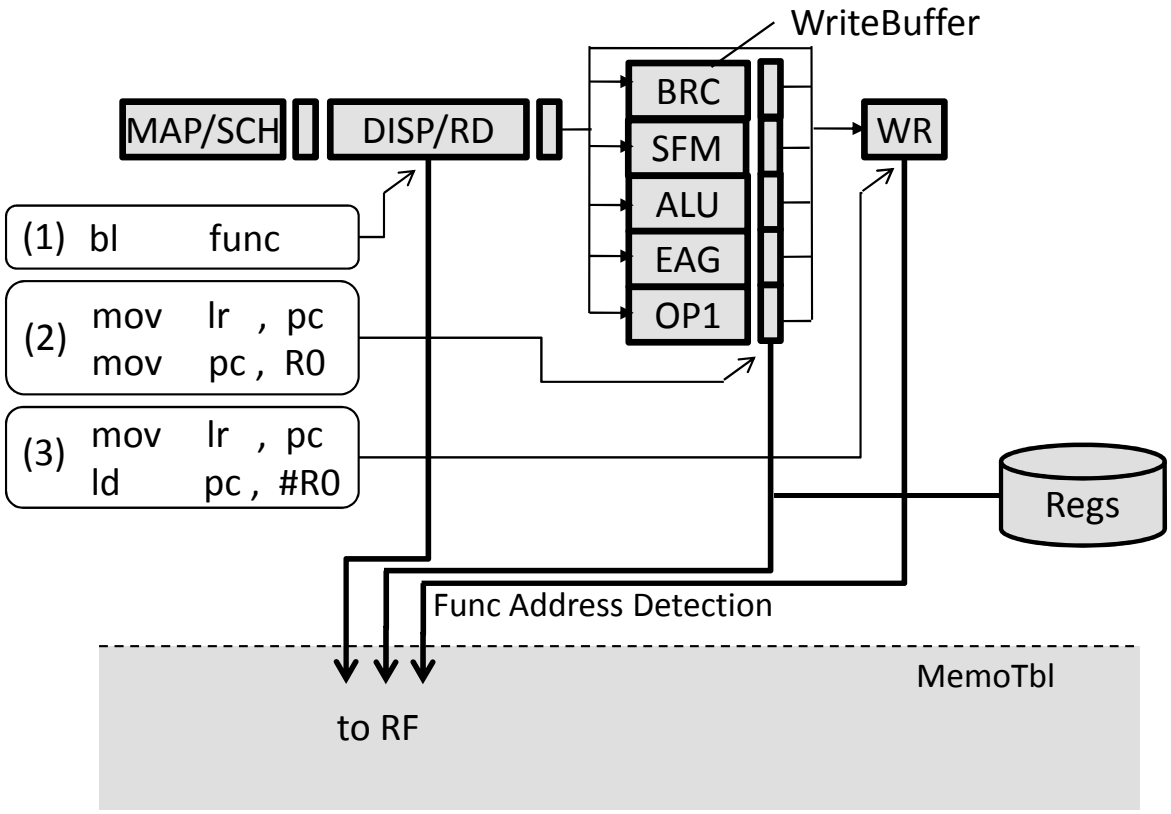


図 19: 再利用テスト時の関数アドレスの検出

発行可能な場合，関数呼び出しアドレスを読み出し関数検索を行う．関数アドレスの読み出し元は，命令ウィンドウに備わるバイパス機構を用いて検出される．ソースオペランドレジスタへの書き戻しが完了しているならばレジスタから読み出し，完了してないならば Write バッファからバイパスする．これを利用することにより，素早く関数アドレスを読み出すことが出来る．

3 つ目は ld 命令などのように，関数アドレスがメモリに格納されており，それをメモリからロードしなくてはならない場合である (図 19(3))．この際関数アドレスが読み出し可能かの判別については，基本的にレジスタに格納されている場合と同様である．ただし，メモリアクセスを伴う命令の場合，関数アドレスを取得するまで時間が掛かるため，前述 2 パターンのような命令発行直後の関数検索は不可能である．またこの場合，キャッシュミスの有無などにより値を取得できるまでの時間が場合によって変化するため，関数アドレスの取得が完了するタイミングを実行ステージにて知ることは困難である．そのため，WR ステージにその命令が到達したタイミングで関数アドレスを検出する．これは，関数アドレスをロードする命令を実行している実行ステー

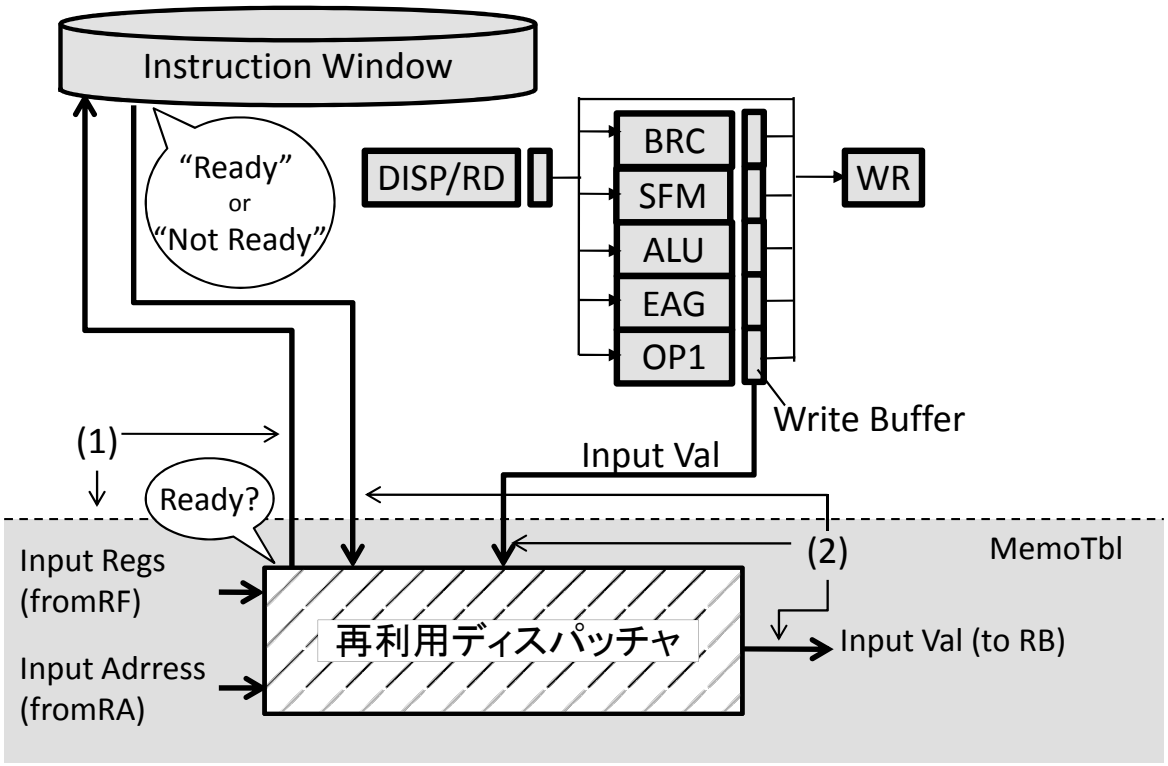


図 20: 再利用ディスパッチャ

ジに応じた WR ステージの Write ポートを監視することによって実現可能である．その命令がロードしてきた関数アドレスをレジスタへ書き戻す際に，そのアドレスは取得され RF へと伝達される．

以上のように，コードパターンに応じて関数アドレスを検出する箇所を変化させる．検出された関数アドレスは MemoTbl 内の関数情報が格納されている RF に伝達され，これを基に関数検索を行う．これにより，各コードパターンにとって最も早いタイミングでの再利用テスト開始を実現する．

4.2 入力値検索

本節では，入力値検索を DISP ステージで行うための方法とその実現のための追加ハードウェアについて説明する．入力値検索を行うためには，入力が Ready 状態かどうかを判別する必要がある．その判別のための機構として**再利用ディスパッチャ**を設置する．再利用ディスパッチャが他のハードウェアに対してどのように接続されているかを**図 20** に示す．再利用ディスパッチャは，関数の引数レジスタが登録されている RF，入力アドレスが登録されている RA，過去の入力値が登録されている RB に接続

されている．

再利用ディスパッチャの動きは 2 段階に分けられる．1 段階目は命令ウィンドウへの入力 Ready の問い合わせである．再利用テストの中で，再利用ディスパッチャが引数レジスタや入力アドレスを RF や RA から受け取ったとする．既存モデルでは，その入力が Ready であることが再利用テスト開始時点で保証できていたので，このまま RB にて一致比較を行うことが出来た．提案モデルでは，このときに再利用ディスパッチャから命令ウィンドウへとその入力が Ready 状態かどうかの問い合わせを行う．命令ウィンドウは各記憶領域が Ready 状態か Wait 状態かを知っているため，この問い合わせによって入力が Ready であることを知ることが出来る．

2 段階目は，命令ウィンドウからの返答の受け取りと RB への一致比較指示である．1 段階目の処理の後，命令ウィンドウはレジスタやメモリの依存関係を調べ，入力が Ready 状態かどうかを再利用ディスパッチャへ返信する．入力が Ready であった場合，その入力格納箇所の値も伝達する．その値の WriteBack が完了している場合，レジスタやキャッシュから再利用ディスパッチャへと伝達される．その一方，WriteBack が完了していない場合，入力を Ready とする命令が実行されていた実行ステージの Write バッファからバイパスして値を取得する．こうすることにより，値の WriteBack を待つことなく入力値を取得できる．その値もまた，再利用ディスパッチャへと伝達される．再利用ディスパッチャは以上の結果を命令ウィンドウから受け取ると，入力が Ready であるならば，同時に受け取った入力値を RB へと伝達する．入力値は，RB に登録されている過去の入力値と比較される．なお既存モデルでは，ここでの比較は RB に登録されている値と，レジスタやキャッシュに格納されている値を直接比較していた．入力が Ready でない場合，何もせずに待機する．待機中パイプライン上では命令の実行が進み，入力が Ready になり次第入力値を RB へ伝達する．

4.3　出力値書き戻し

再利用テストが成功した際に，関数出力のレジスタやキャッシュへの書き戻しを制御する機構として，本提案モデルでは**W1 コントローラ**を設置する．W1 コントローラと他のハードウェアとの接続の様子を**図 21** に示す．W1 コントローラは MemoTbl の RA から入力値がすべて一致したこと知らせる信号を受け取り，書き戻し可能な条件が満たされるまでその信号を保留する機構である．条件が満たされないことが判明した場合，RA から発せられた信号は W1 に届かず，再利用は中止となる．再利用テストが成功しても出力を書き戻してはならないパターンには以下の 3 種類がある．

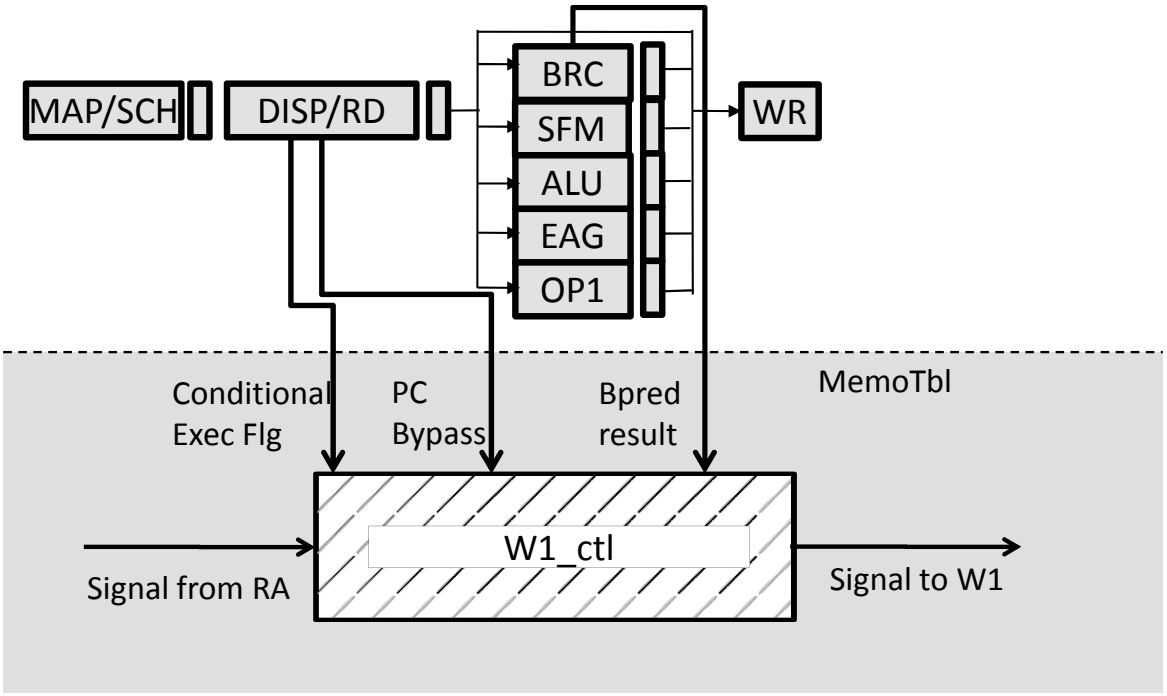


図 21: W1 コントローラ

1つ目は先行する分岐命令の分岐予測が失敗していた場合である．**図 22** に，先行する分岐命令の分岐予測が失敗しながらも後続の関数を再利用した場合の命令の流れを示す．この例では，1行目の分岐命令は untaken であると予測されたものとする．図 22(t0) で関数呼び出し命令がフェッチされるが，このときにはまだ分岐予測の成否は判明していない．分岐予測が成功したかどうかの判定は，ベースアーキテクチャでは BRC ステージと RETIRE ステージが連携して行う．分岐が taken か untaken かの判定を BRC ステージが行い，その結果を用いて RETIRE ステージにて分岐予測が成功したか失敗したかを判定する．失敗したならば，パイプラインはフラッシュされる．図 22 では，実行ステージとリタイアステージで行われるものとする．図 22(t1) にて分岐予測の成否が判明し，分岐予測が失敗していたとする．このときの後続命令のフラッシュは，分岐命令がコミットされたあとに行われる．さて，分岐予測の成否の判定の後，図 22(t2) で関数呼び出しが検出され，再利用テストが行われる．このとき再利用テストが成功すれば再利用が適用され，図 22(t3) でプログラムの状態はこの関数が実行された際のものとなる．しかしこの関数は，先行する分岐命令の分岐予測が失敗しているため，本来であれば呼び出されない関数である．つまり，このように再利用を適用してしまうと，プログラムの正しい動作から外れることになる．これは逆説的には，先行する分

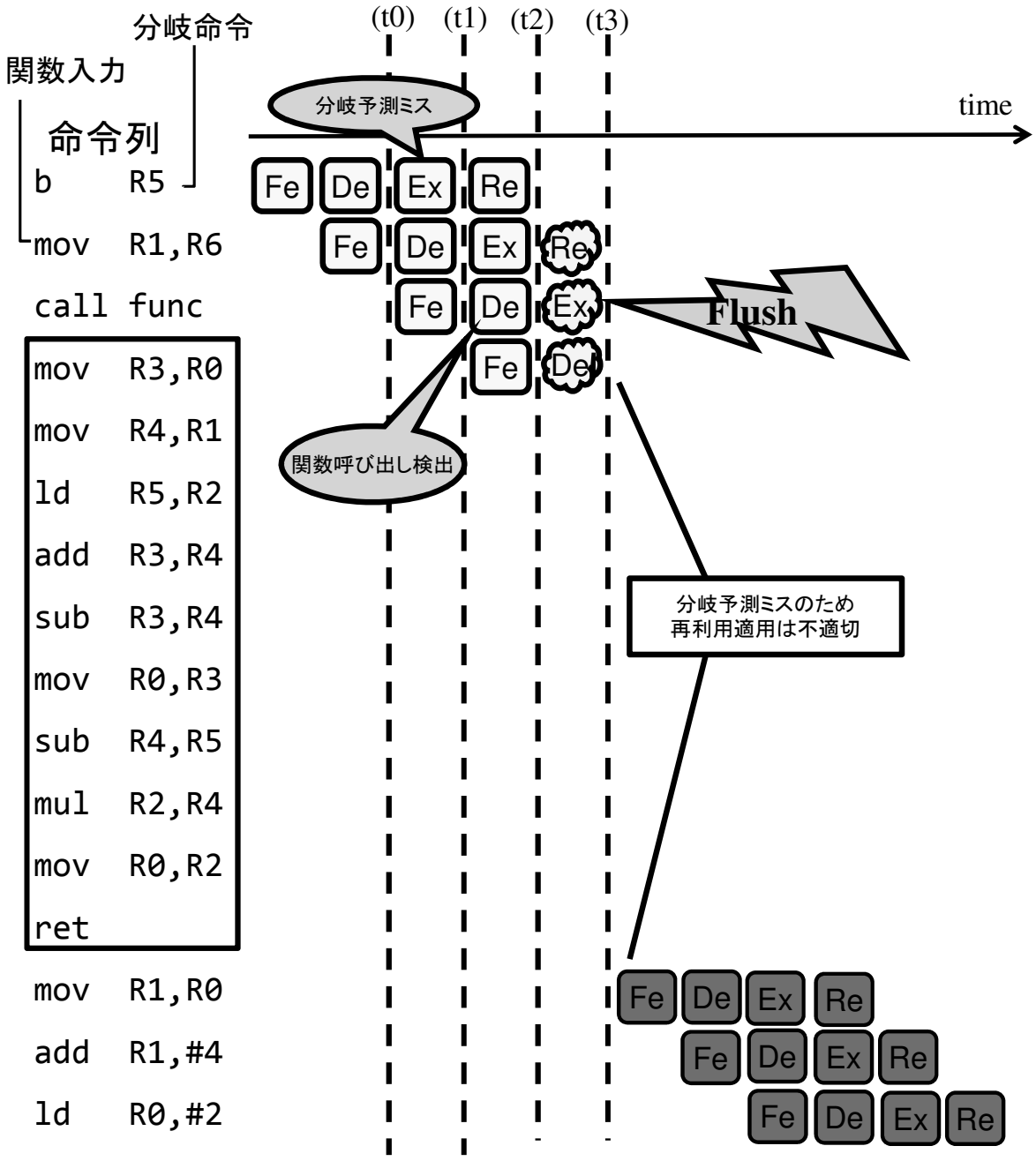


図 22: 不適切な再利用の例 1

岐命令の分岐予測の成功が判明するまでは，関数の出力は書き戻してはならないということである．

さて，先述のとおりベースアーキテクチャでは分岐予測の成否を RETIRE ステージで判定しているが，この判定は分岐予測方向と実際の分岐方向が判明すれば可能なため，実際には BRC ステージで分岐方向が判明した時点で可能である．そこで提案モデ

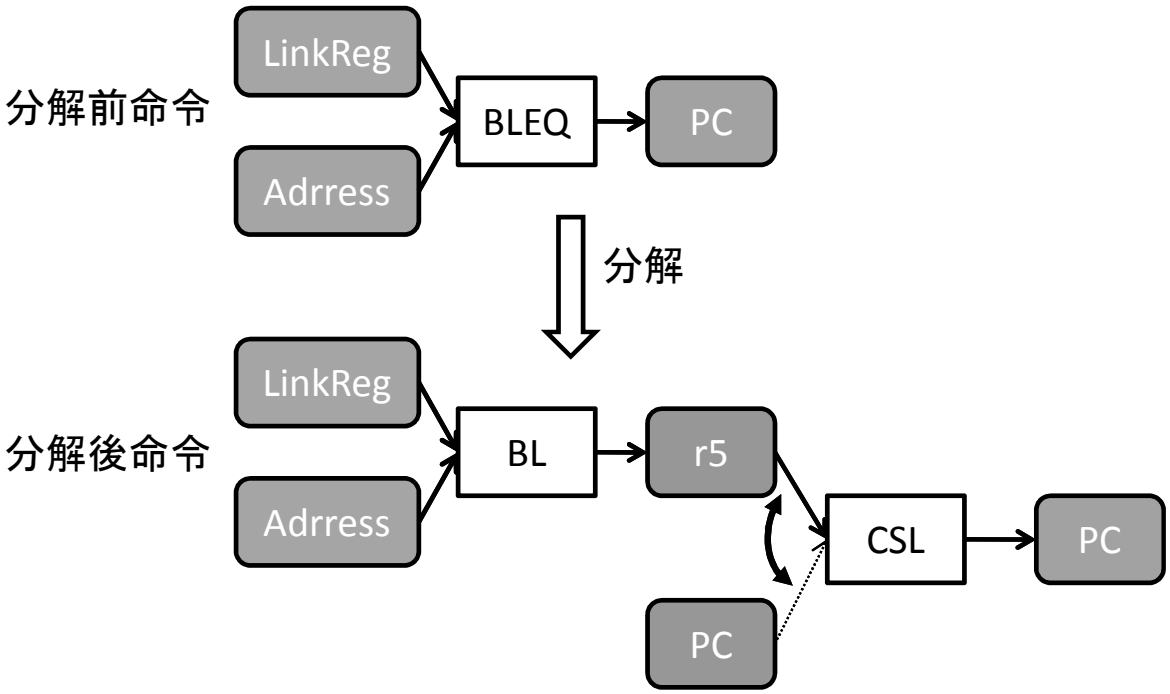


図 23: 条件実行命令の分解の様子

ルでは，BRC ステージにて分岐予測の成否を判定する．その判定で分岐予測が成功したとされると，W1 コントローラにそのことが伝達される．失敗したと判定されるならば，再利用テストは中止される．このようにすることで，本来呼び出されないはずの関数の再利用が適用されることを防ぐ．

2 つ目は関数呼び出し命令が条件実行命令であり，かつその実行される条件が満たされていない場合である．ここでまず，条件実行命令の仕様について述べる．関数呼び出し命令が条件実行命令であった場合の命令分解の様子を，**図 23** に示す．ここでは関数呼び出し命令の例として BL 命令を EQ フラグが立っているときに実行する場合を挙げる．ここに示す分解は，プログラムカウンタをディスティネーションレジスタとする命令全般に適用される．条件実行命令は，HOST-DECODE ステージにて，元の命令から実行条件を除いた命令と，条件判定を行う CSL(conditional-select) 命令に分解される．この時，実行条件が除かれた命令のディスティネーションレジスタは，本来のプログラムカウンタから条件実行の際の結果一時保管用のレジスタに変更される．後続の CSL 命令は実行時に条件判定を行い，条件が合致し命令が実行されると判定されるならば一時保管用レジスタの値を本来のデスティネーションレジスタであるプログラムカウンタへと格納する．以上から条件実行命令が実際に実行されるかは，ベース

アーキテクチャでは CSL 命令の実行時に判定していることが分かる．その CSL 命令が条件判定のために参照するフラグをバイパスすれば，CSL 命令の発行時には条件実行命令が実行されるかが分かる．この判定の結果，関数呼び出し命令が実行されるならば関数出力の書き戻しを許可し，実行されないならば出力書き戻しを中止する．

さて，この処理を正しく行うためには関数呼び出し命令が条件実行命令であったかどうかの判定をしなくてはならない．この判定は，パイプラインレジスタに存在する命令終端ビットを参照することで可能である．ベースアーキテクチャではフェッチ時では 1 つだった命令を ARM-DECODE ステージや HOST-DECODE ステージで複数の命令に分解する．命令終端ビットは，分解後の命令が分解前命令の終端であることを表すビットフィールドである．条件実行命令の場合，図 23 のように終端命令は CSL 命令となる．つまり，条件実行命令でない通常の関数呼び出し命令は終端命令である一方で，条件実行の関数呼び出し命令は終端命令でなくなる．これは逆説的には，終端命令である関数呼び出し命令は必ず実行され，終端命令でない関数呼び出し命令は条件実行命令であるということである．このことから，終端命令でない関数呼び出し命令は実際には実行されない可能性があるものとし，実際に実行されるかの判定を待つ必要がある．

3 つ目は，関数復帰アドレスが分からない場合である．通常実行時に関数復帰が行われる際，関数復帰アドレスはリンクレジスタから取得される．PC のリンクレジスタへの退避は関数呼び出し時に行われるため，この退避命令を監視すれば関数復帰アドレスをバイパス出来る．そのような命令は 2 種類あり，1 つは関数呼び出し命令自体がリンクレジスタへ PC を退避する場合である．これには，4.1 節の図 19(1) に示すような，bl 命令が相当する．bl 命令は，PC をリンクレジスタへ退避するのと PC を関数アドレスで上書きするのを同時に行う．もう 1 つは，直前の命令がリンクレジスタへ PC を退避している場合である．これには，4.1 節の図 19(2)(3) それぞれの 1 つ目の命令のような mov 命令が相当する．これは，bl 命令が行っていた処理を明示的に行っているものである．いずれの場合も，その命令のソースオペランドを参照すれば，関数復帰アドレスが分かる．どの命令が後者の命令に該当するかの判定が必要であるが，これは関数呼び出し検知デコーダのうち HOST-DECODE ステージに設置されたユニットで予め検出することで可能である．これらのような命令が発行可能であるとき，関数出力は書き戻し可能である．発行可能かどうかは DISP ステージにて命令ウィンドウに問い合わせることで検知する．

以上のように関数出力の書き戻しを制御することによって，プログラムの実行を正

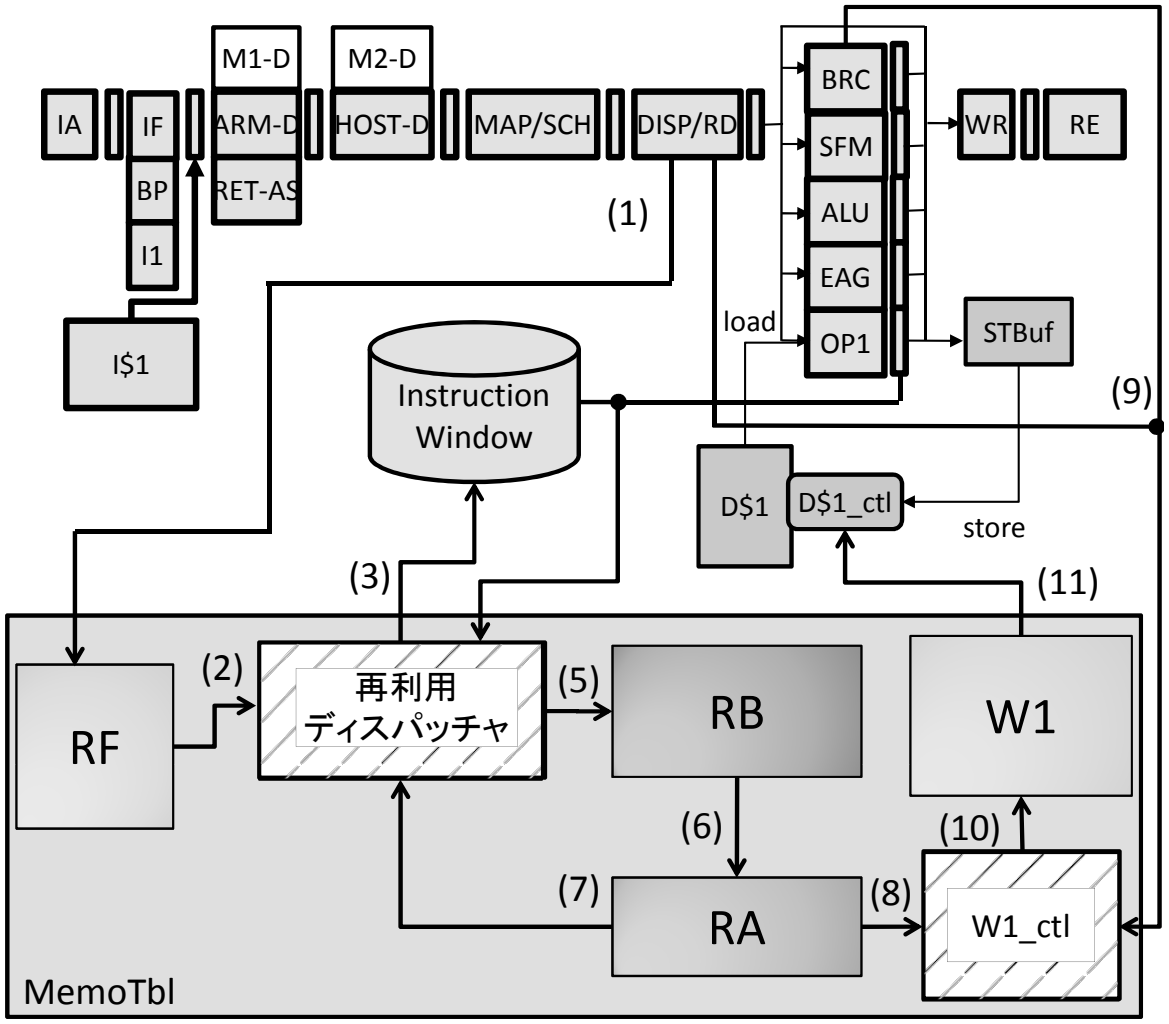


図 24: 提案アーキテクチャ全体図

常に行うことを可能にしている．このとき，出力を書き戻さなかった場合も，再利用テストは行っている．既存モデルでは呼び出されることが保証されている関数に対してのみ再利用テストを行っていたが，提案モデルでは呼び出されることが保証されていない関数呼び出しに対しても再利用テストを行う必要がある．そのため，既存モデルと比べ提案モデルでは再利用テストを行う回数は増加している．その増加した再利用テストを行うための時間が，関数出力書き戻しの制御のオーバヘッドとなる．

4.4 提案アーキテクチャ

以上に述べた機能を総括した，提案アーキテクチャの全体図を**図 24** に示す．提案アーキテクチャが関数呼び出し命令を検知してから関数の出力がレジスタやメモリへ

書き戻されるまでの動作を説明する．まず，関数呼び出し命令が検知され，DISP ステージまで到達したとする．ここでまず，再利用表の RF から関数を検索するために関数アドレスを検出する．関数アドレスの検出方法は 4.1 節で述べた通りである．関数アドレスが検出されると，RF から関数が検索される (図 24 中 (1))．検索の結果，一致する関数を発見したとする．RF は関数アドレスと、それと共に登録されている入力レジスタを再利用ディスパッチャへと伝達する (図 24 中 (2))．再利用ディスパッチャの動作は 4.2 節で述べた通りである．入力レジスタを受け取ると，再利用ディスパッチャは命令ウィンドウへ入力が Ready かどうかを問い合わせる (図 24 中 (3))．入力が Ready でないならば再利用テストは一旦保留され，パイプライン上で命令の実行が進む．再利用ディスパッチャは，入力が Ready となるまで命令ウィンドウへと問い合わせを続ける．入力が Ready であることが判明した場合，再利用ディスパッチャは入力値を受け取りそれを RB へ伝達する (図 24 中 (4))．RB では入力値の一致比較が行われ，もしも一致するならば最初の入力アドレスを RA から検索する (図 24 中 (5))．検索の結果入力アドレスが発見されると，そのアドレスは再利用ディスパッチャへと伝達される (図 24 中 (6))．再利用ディスパッチャは，その入力アドレスについて命令ウィンドウに Ready かどうかを問い合わせ，Ready であるならば入力レジスタの際と同様に入力値を受け取り RB へと伝達する．この処理を繰り返していき (図 24 中 (3)-(6))，全ての入力値が一致したならば，該当する出力値を読み出すことを試みる (図 24 中 (7))．その読み出しは W1 コントローラによって制御され，関数出力書き戻しの可否が決定されるまで保留される．関数出力の書き戻しが可能であることがパイプラインから伝達されると (図 24 中 (8))，関数出力が W1 から読み出されレジスタやメモリへと書き戻される (図 24 中 (9)-(10))．書き戻しが不可能と判定されると，再利用は中止される．

4.5　簡易モデル

提案モデルを簡略化したものとして，再利用ディスパッチャを取り除いたモデルを実装した．簡易モデルのアーキテクチャ図を**図 25** に示す．簡易モデルには，DISP ステージにおける関数アドレスの検出と，再利用ストッパーによる再利用結果の書き戻しの抑制が実装されている．その一方で，入力が Ready であるかの確認を行う再利用ディスパッチャは実装されていない．再利用ディスパッチャの代わりに，既存モデルで行われていた先行命令の全コミットよる入力 Ready の確認を行う．

図 26 に，簡易モデルが再利用に成功した際の動作モデルを示す．再利用テストに必要な各操作を行うステージは，3.3 節に示したものに準拠する．まずはじめに，図 26(t0)

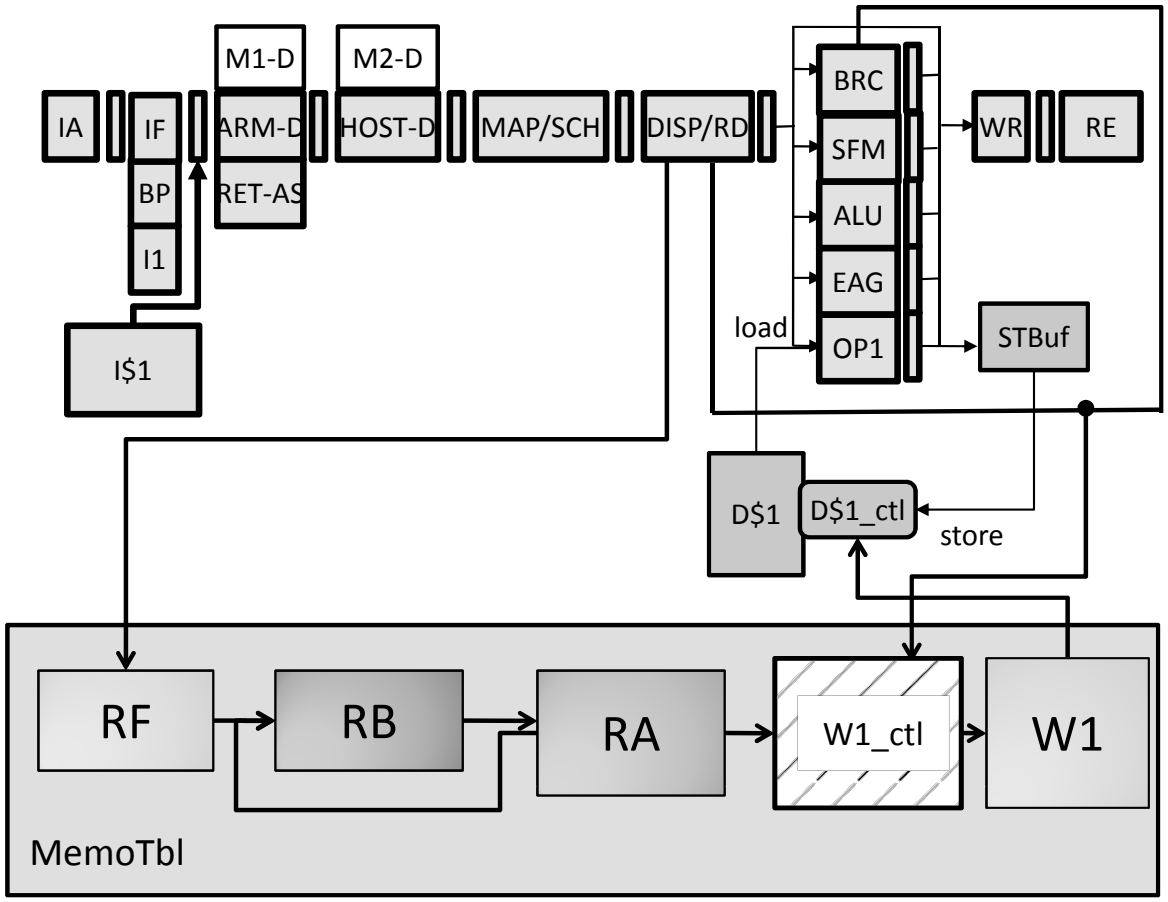


図 25: 簡易モデルアーキテクチャ図

で関数呼び出し命令がフェッチされ，図 26(t1) で関数呼び出しを検出したとする．ここで，関数呼び出し検出を行うステージ及びそれよりも前段のステージを停止させ，先行命令が全てコミットされるまで，関数呼び出し命令とその後続命令をストールさせる．先行命令がすべてコミットされ入力値がすべて Ready であることが保証されると，図 26(t2) で再利用テストを開始する．入力値の一致比較が行われ再利用が成功すると，図 26(t3) で関数の出力値がレジスタやキャッシュへと書き戻される．図 26(t4) で書き戻しが完了すると，関数の後続命令がフェッチされる．

以上の動作の際，図 26 の例では関数呼び出し命令のフェッチ (t0) から再利用テストの開始 (t2) まで 3 サイクルを，関数の後続命令のフェッチ (t4) まで 6 サイクルを要しする．これは，3.3 節で示した 4.4 節のアーキテクチャの動作モデルの例 (図 17) の 2 サイクルに対し 1 サイクル増加している．後続命令のフェッチは再利用を行わない通常実行に対して 5 サイクルの高速化をしており，これは図 17 と同様であるが，これらの後続命令にオーバーラップする先行命令は既にコミット済みであり，スループットは低

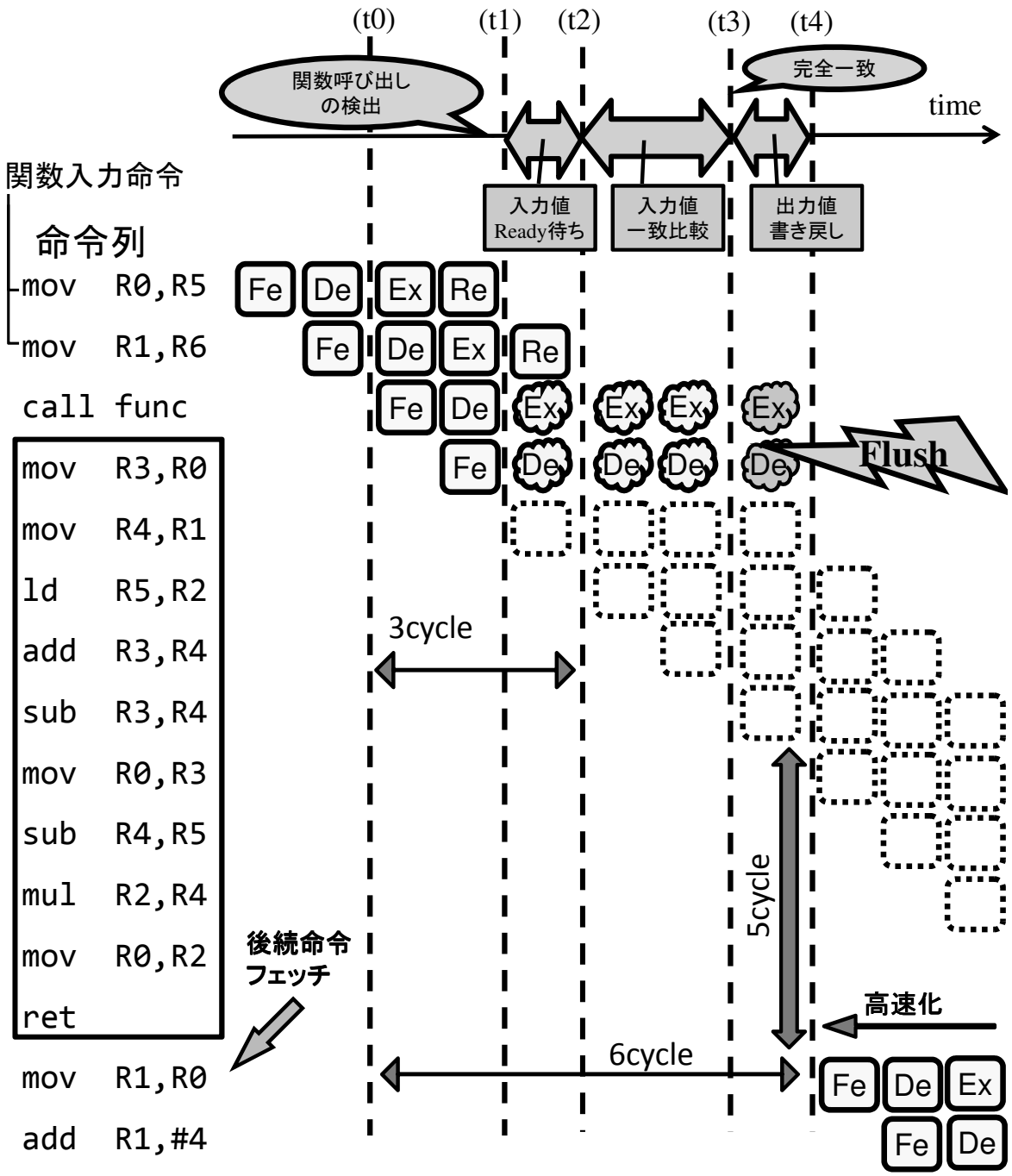


図 26: 簡易モデルの動作：再利用成功時

下している.

次に**図 27** に，簡易モデルが再利用に失敗したときの動作モデルを示す．基本的な動作は 3.3 節の図 18 と同様である．図 27(t0) で関数呼び出し命令がフェッチされ，図 27(t1) で関数呼び出しを検出し，(t2) で先行命令がすべてコミットされるまで関数呼び出し

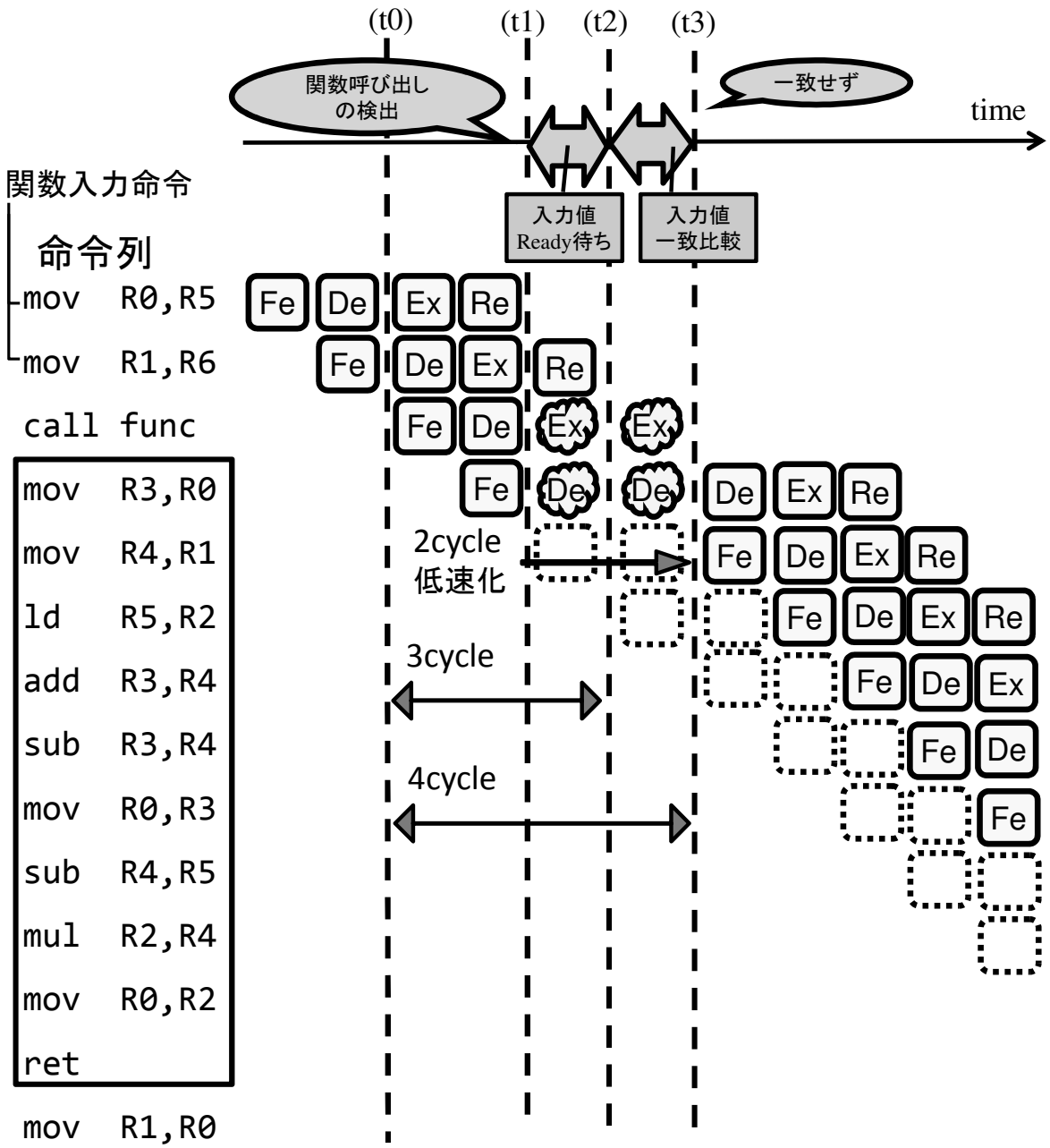


図 27: 簡易モデルの動作：再利用失敗時

命令とその後続命令をストールさせる．その後，関数の入力値の一致比較が開始され，図 27(t3) にて入力値が一致しないことが判明すると再利用テストは中止され，停止していた各ステージが再び動作し実行が再開される．

このように，上述の再利用成功時の場合と同様に再利用テストを行う前に先行命令をすべてコミットさせる必要がある．そのためのパイプラインストールにより，図 27

の例では再利用テスト開始まで 3 サイクル，プログラム実行の再開まで 4 サイクルを要し，図 18 の場合よりも 1 サイクル，再利用を行わず通常実行する場合よりも 2 サイクル遅い．

以上のように，このモデルは既存モデルと比べて，再利用テストが完了するステージが変更されているため，再利用テスト成功時のパイプラインフラッシュによって無効化される命令の数が減少している．その一方で，再利用テスト時に関数呼び出し命令の先行命令がすべてコミットされるまで後続命令をすべてストールさせているため，再利用テストを始めるためのオーバヘッドが新たに発生している．

5 評価

5.1 評価環境

評価時の各パラメータは 2.3 節の表 2.3 に準ずる．ベンチマークプログラムとして，stanford ベンチマークを gcc-4.1.1(-O2 -msoft-flaot -march=armv4) によりコンパイルし，スタティックリンクにより生成したロードモジュールを用いた．

5.2 評価

評価結果を**図 28** に示す．グラフの横軸はプログラム名を表している．縦軸は実行サイクル数を表しており，ベースアーキテクチャ「OROCHI」を用いて実行した場合の実行サイクル数を 1 として正規化している．各グラフの左のものが既存の ARM ベースモデルを，右のものが提案した簡易モデルの実行サイクル数を表す．r_step はキャッシュミスレイテンシを含むプロセッサ実行時間を表す．t_step，m_step，w_step は MemoTbl へのアクセスレイテンシであり，順にレジスタと CAM との比較時間，キャッシュと CAM との比較時間，MemoTbl から出力を書き戻すのに要した時間を意味する．reuse_hit_bubble は再利用成功時にパイプラインフラッシュによってバブルへと変化した命令が，変化する前にパイプライン中に存在した時間を表す．cycle_Ih，cycle_Im は関数呼び出し命令を DISP ステージにて検出してから入力の Ready を検知するまでに要した時間であり，前者は再利用成功時のもの，後者は再利用失敗時のものを表す．cycle_Sh，cycle_Sm は W1 コントローラが W1 からの出力書き戻しを待機させている間の時間であり，cycle_I と同様に前者は再利用成功時のもの，後者は再利用失敗時のものを表す．

結果，総実行サイクル数は平均 5.7%，Perm で最大 14.8%の低速化となった．これは，提案した簡易モデル独自のオーバヘッド cycle_I，cycle_S が大きかったためである．cycle_I は実行サイクル数のうち平均で 16.8%，Perm で最大 34.5%を，cycle_S は実行

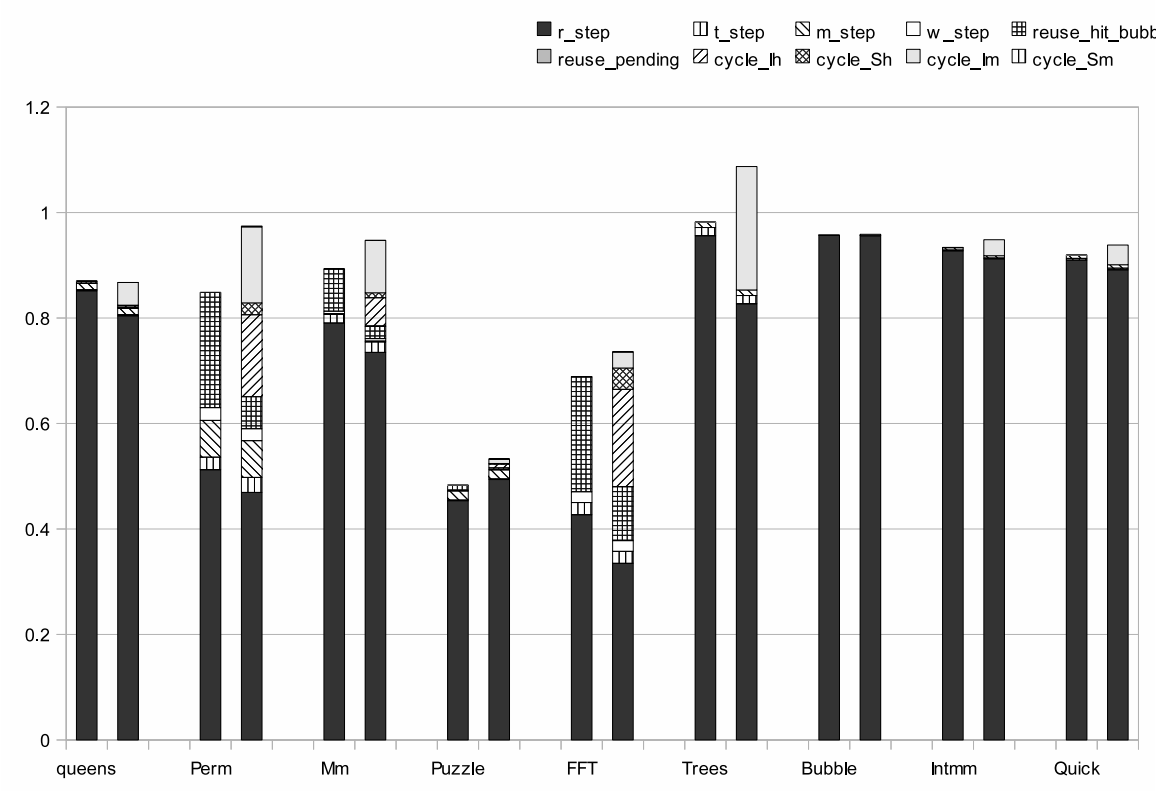


図 28: 評価結果

サイクル数のうち平均で 1.8%，FFT で最大 4.7%を占める．しかしその一方で，既存モデルで問題となっていた reuse_hit_bubble は平均 67.4%，Perm で最大 72.3%削減されている．この削減率は総実行サイクル数の 3.5%に相当する．また，Puzzle を除く全てのプログラムで r_step が削減されている．

5.3 考察

提案手法では，既存モデルで問題となっていた reuse_hit_bubble は低減されている．これは，提案手法で再利用テストを行うステージが RETIRE ステージから DISP ステージへとより前段に変更されたためと考えられる．再利用成功時にバブルに変化する命令を低減するのに，提案手法が有効であることが分かる．

また提案手法では，r_step が削減されている．この原因として，r_step 以外の時間にも命令の処理が進む点が考えられる．入力値が Ready となるまでのオーバヘッドである cycle_Ih，cycle_Im，cycle_Sh，cycle_Sm の間，パイプラインの MAP ステージよりも前段はストールするが後段は動いており，そこに存在する命令の実行は進む．そのため，一部の命令は cycle_Ih，cycle_Im，cycle_Sh，cycle_Sm の間に実行されている．こ

れらのオーバヘッド cycle_Ih，cycle_Im，cycle_Sh，cycle_Sm の発生によって，r_step が減少する一方で総実行サイクル数は増加している．

ここで，4.2 節で示した再利用ディスパッチャを実装した際にどのようなパフォーマンスとなるかを考察する．そのためにまず，図 28 の各凡例についてそれぞれ考察する．

r_step

r_step はプログラムが完了するまでのプロセッサ実行時間であり，この間の行程はすべて行われなくてはならず，また必要以上に行程が増えることも無い．そのため，これは決して増加することは無い．その一方で，3.3 節の図 17 に示すように，既存モデルよりもスループットが向上する可能性がある．以上から再利用ディスパッチャを用いたモデルでは，r_step は最大で既存モデルと同じになると考えられる．

t_step，m_step，w_step

t_step，m_step，w_step の 3 つは，再利用テストと関数出力書き戻しの際に発生するオーバヘッドである．再利用テストの回数は，提案簡易モデルでは既存モデルに比べて増加している．これは，先行する分岐命令の分岐予測が失敗して通常実行時には関数が呼び出されないなど，関数呼び出しが検出されながらも実際には呼び出されない場合についても，提案簡易モデルでは再利用テストを行うためである．既存モデルでは RETIRE ステージで関数呼び出し命令がコミットされたときに再利用テストを開始していたために，実際には実行されない関数の再利用テストはまったく行われなかった．しかし提案簡易モデルでは，実際に分岐予測の結果が判明する前に再利用テストを行うために，再利用テストを行いかつそれが成功しつつも再利用は適用されない場合が存在する．そのために，再利用テスト時にレジスタやキャッシュ，メモリなどにアクセスする際のオーバヘッド t_step，m_step は増加している．その一方で関数出力の書き戻しの回数は増減しないため，そのためのオーバヘッド w_step は変化しない．これは再利用ディスパッチャを用いたモデルでも同様であり，このモデルの t_step，m_step，w_step は提案簡易モデルのものと同じとなる．

reuse_hit_bubble

reuse_hit_bubble は再利用テストが成功し再利用が適用される際に発生するバブル量である．提案簡易モデルではこのバブル量を，フェッチステージから関数出力書き戻し処理が行われた際に関数呼び出し命令が存在していたステージまでの段数を計測することで取得している．このステージ段数は，関数呼び出しを検出した

直後に再利用を適用した際のステージ段数と同等である．再利用ディスパッチャを用いたモデルでは，再利用テストが可能となった直後に再利用テストが行われる．関数呼び出し検出から再利用テスト開始までの時間差は存在する可能性があるが，この時間差は r_step に計上される．これについては，cycle_Ih，cycle_Im，cycle_Sh，cycle_Sm の項目で詳細に述べる．このことにより，関数呼び出し検出から再利用テスト開始までのオーバヘッドは実質的には存在しなくなる．以上から，再利用ディスパッチャを用いたモデルの reuse_hit_bubble は，提案簡易モデルのものと同じになる．

reuse_pending

reuse_pending は，再利用テストを行う際に，キャッシュコントローラが空になるのを待つストール時間である．この時間は既存モデルでは他の命令の実行時間にオーバーラップしている．再利用テストを行う前に実行される命令の数が減少するため，再利用ディスパッチャを用いたモデルでは reuse_pending は増加すると考えられる．しかし，入力値の Ready を待つ時間にオーバーラップするとも考えられ，今後の評価が必要である．しかし，reuse_pending の量は実行サイクル数全体に対し総じて非常に少なく，概ね無視できると予測される．

cycle_Ih，cycle_Im，cycle_Sh，cycle_Sm

cycle_Ih，cycle_Im，cycle_Sh，cycle_Sm は，入力値の Ready を保証するために関数呼び出し命令の先行命令のみを実行する，提案簡易モデルが独自に持つストール時間である．これらのストール時間の間に，大量のバブルが発生している．このバブルは，入力値の Ready を保証するためのストールが無ければ発生しない．また，それらの時間に実行されている先行命令の実行時間は，既存モデルでは r_step に計上されている．これは再利用ディスパッチャを用いたモデルでも同様である．なお，この先行命令を実行する時間は，再利用ディスパッチャを用いたモデルでは関数呼び出し検出から再利用テストを行うまでの時間に相当する．以上から，cycle_Ih，cycle_Im，cycle_Sh，cycle_Sm は無くなり，その間に実行される命令の実行時間は r_step に計上される．

以上から，再利用ディスパッチャを用いたモデルの実行サイクル数は，既存モデルの r_step，提案簡易モデルの t_step，m_step，w_step，reuse_hit_bubble の合計値に概ね沿うと考えられる．その削減比率は，既存モデルに比べて平均 3.5%，Perm で最大 18.4%の削減となっており，更なる高速化が期待できる．よって，再利用ディスパッチャによる入力 Ready の確認の実現が必要であると考えられる．

6　おわりに

自動メモ化プロセッサは，メモ化をハードウェアにより自動的に行い，プログラム実行の高速化を達成する．このうちスーパースカラ型 ARM をベースとするモデルには，再利用適用時にパイプラインをフラッシュするためにバブルが発生し，オーバヘッドが増加してしまうという問題があった．本研究では，このバブルの削減を目的に，再利用テストのタイミングの改良を図った．その過程で，再利用テストを行いメモ化を正しく適用するための，様々な条件が存在することがわかった．そこで本研究では，それらの条件を満たすための方法を関数検索，入力値比較，結果書き戻しの 3 つの観点から提案した．これらのうち関数検索と結果書き戻しの条件を変更したモデルをサイクルベースのシミュレーションにより stanford ベンチマークを用いて評価したところ，平均 67.4%のバブル削減となったが，実行サイクル数は平均 5.7%の増加となった．この結果から，今後入力値比較の方法を改良することによって，今回評価したモデルで発生したオーバヘッドを隠蔽する必要があることがわかった．またそこから，今後追加実装されるべきハードウェアについての知見を得た．

今後の課題としては，再利用ディスパッチャの実装によるオーバヘッド削減がまず挙げられる．これが実現されれば，平均 3.5%の実行サイクル数削減となる．次に，再利用機構そのものの改良により，より高速に再利用テストを行う方法の提案が挙げられる．また，再利用テストをパイプライン上での命令実行と並列に行うことも考えられる。これらが実現されると、再利用テストを行うのに要するオーバヘッドが削減ないし隠蔽可能である．またこのほか，SPARC ベースモデルに実装済みの機能を ARM ベースモデルに移植し，そのパフォーマンスを測ることが挙げられる．具体的には，loop 区間を対象とした再利用，オーバヘッドフィルタやキャッシュの書き換え検知による入力値検索削減手法などである．以上はハードウェアの改良による高速化であるが，その一方でソフトウェア支援による更なる高速化も考えられる．自動メモ化プロセッサの ARM ベースモデルでは，プログラムにメモ化を適用するためにプログラムソースから必要な情報を抽出する必要がある．これを発展させてプログラムそのものを自動メモ化プロセッサに適するように変換，最適化すれば，より高いパフォーマンスが得られると考えられる．

謝辞

本研究の為に多大な御尽力をいただき，日頃から熱心な御指導を賜った名古屋工業大学の津邑公暁准教授，奈良先端科学技術大学院大学の中島康彦教授に深く感謝いたします．また，本研究のために多大な御協力をいただいた新美明仁さん，高木伴彰さんに深く感謝致します．さらに，本研究に対しての熱心な御協力をいただきました，名古屋工業大学の松尾啓志教授に深く感謝致します．加えて，たびたびご検討・助言をしていただいた同大学の齋藤彰一准教授や松井俊浩助教授にも深く感謝致します．最後に，本研究の際に多くの助言・協力をしていただいた松尾・津邑研究室ならびに齋藤研究室のみなさまに対しても深く感謝致します．

参考文献

[1] W.Wall, D.: Limits of Instruction-Level Parallelism, *WRL Research Report,* No. 93/6 (1993).

[2] Akio Nakajima, Ryotaro Kobayashi, H. A. T. S.: Limits of Thread-Level Parallelism in Non-numerical Programs, *IPSJ Degital Courier*, Vol. 2, pp. 280–288 (2006).

[3] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).

[4] Sun Microsystems: *UltraSPARC III Cu User's Manual* (2002).

[5] ARM Limited: *"ARM Architecture Reference Manual"ARM DDI 0100E* (2000).

[6] KOJIMA, T. and NAKASHIMA, Y.: Design of a Centralized Instruction Window Superscalar for Evaluation of OROCHI, *IPSJ SIG Notes*, Vol. 2006, No. 127, pp. 61–66 (20061128).