Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

RISC-V をベースアーキテクチャとする 自動メモ化プロセッサの設計

宮川 晃輔[1]　中原 博研[1]　津邑 公暁[1]　中島 康彦[2]

概要：我々は計算再利用に基づいた高速化手法を採用した自動メモ化プロセッサを提案している．自動メモ化プロセッサは再利用対象である関数の実行時に，その関数の入出力を記憶する．その後，同一関数を同一入力により再実行しようとした際に，過去に記憶した出力を再利用することでその実行自体を省略する．これまで我々は，自動メモ化プロセッサのシミュレーションによる評価を行ってきた．しかし，その評価は一部の限定的なアーキテクチャでのシミュレーションにとどまっており，自動メモ化プロセッサの汎用性や実用性を十分に評価できていない可能性がある．そこで，本稿では，オープンかつ標準的な命令セットアーキテクチャとして注目を集めている RISC-V を新たに自動メモ化プロセッサのベースアーキテクチャとして採用し，その実装や評価を通じて自動メモ化プロセッサの汎用的実用性や改良の余地を検討する．RISC-V をベースアーキテクチャとする自動メモ化プロセッサを設計し，評価を行った結果，その実行サイクル数は，自動メモ化機構を実装しない場合と比較して，最大 49.0%減少，平均 6.2%増加となった．実行サイクル数の削減に成功した場合が存在したことから，RISC-V をベースアーキテクチャとする自動メモ化プロセッサは，既存の自動メモ化プロセッサと同様に性能向上が期待できることが確認できた．

1. はじめに

これまで，ムーアの法則およびデナード・スケーリングを指針として，集積回路の微細化によりクロック周波数を向上させることで，マイクロプロセッサは高速化を続けてきた．しかし，ゲート遅延に対する配線遅延の相対的な増大や，行き過ぎた微細化に伴うリーク電力および発熱量の増大といった問題から，クロック周波数の向上によるマイクロプロセッサの高速化は困難になってきている．

こうした中で，SIMD やスーパスカラなどの細粒度な並列性に基づく高速化手法が注目されてきた．しかし，一般にプログラム中から細粒度な並列性を抽出することは容易ではなく，またプログラム中に存在する細粒度な並列性にも限りがあることから，これらの手法による高速化にも限界がある．よって現在では，消費電力や発熱量の問題を解決しつつ，プロセッサあたりの処理能力を向上させるため，クロック周波数を抑えたコアを 1 つの CPU に複数搭載したマルチコアプロセッサが広く普及している．そして，このマルチコアプロセッサを有効活用するために，スレッドレベル並列性（TLP: Thread Level Parallelism）に着目した様々な高速化手法が研究されている．これらの手法は，プログラムを複数スレッドで処理できるように分割し，それらをそれぞれのコアに割り当てることで高速化を図っている場合が多い．しかし，そもそも並列性を持たず TLP を抽出できないプログラムが存在することや，プログラマが明示的に並列処理プログラムを記述することは容易ではないことなど，様々な問題がある．これらの高速化技術は，いずれも複数の処理を並列実行することで，処理の総量は変化させずに高速化を図るものである．

これに対し，過去の実行結果を記憶しておき，その結果を再利用することで処理の総量自体を削減し，高速化を図る計算再利用と呼ばれる高速化手法がある．我々は，専用ハードウェアを用いてプログラム中の一部の処理に計算再利用を自動的に適用する，**自動メモ化プロセッサ** [1] に関する研究を行ってきた．自動メモ化プロセッサは関数を計算再利用の対象区間とし，実行時に関数の入力と出力の組を再利用表と呼ばれる専用表へ登録する．そして，再び同一関数を同一入力を用いて実行しようとした際に，再利用表に登録されている過去の出力を再利用することで，その関数の実行を省略する．

さて，これまで我々は，自動メモ化プロセッサのシミュレーションによる評価を行ってきた．しかし，その評価は一部の限定的なアーキテクチャでのシミュレーションにと

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

どまっており，自動メモ化プロセッサの汎用性や実用性を十分に評価できていない可能性がある．そこで，本稿では，オープンかつ標準的な命令セットアーキテクチャとして注目を集めている RISC-V[2] を新たに自動メモ化プロセッサのベースアーキテクチャとして採用し，その実装や評価を通じて自動メモ化プロセッサの汎用的実用性や改良の余地を検討する．

以下，2 章では我々が提案している自動メモ化プロセッサについて説明する．3 章では，RISC-V で自動メモ化プロセッサを実現するために必要な設計について説明し，4 章では設計した自動メモ化プロセッサの実装と動作を説明する．5 章では実装した自動メモ化プロセッサの評価を行い，最後に 6 章で結論を述べる．

2. 自動メモ化プロセッサ

本章では，我々が提案している自動メモ化プロセッサの動作原理とその構成について概説する．

2.1 自動メモ化プロセッサの概要

計算再利用（Computation Reuse）とは，プログラム中で定義された関数において，その入力の組（**入力セット**）と出力の組（**出力セット**）を実行時に記憶し，再び同じ入力セットでその関数が実行されようとした場合に，過去の記憶された出力セットを再利用することで，その関数の実行自体を省略する高速化手法である．また，この計算再利用を各関数に適用することを**メモ化（Memoization）**[3] と呼ぶ．メモ化は元来，高速化のためのプログラミングテクニックである．しかし，メモ化を適用するためには，プログラムを記述しなおす必要があり，既存のロードモジュールやバイナリをそのまま高速化することはできない．その上，ソフトウェアによるメモ化 [4] はオーバーヘッドが大きく，限られたプログラムでしか性能向上が得られていない．

そこで，ハードウェアを用いて自動的にメモ化を行うプロセッサとして，**自動メモ化プロセッサ（Auto-Memoization Processor）**が提案されている．自動メモ化プロセッサは，プログラムの実行時に動的に関数を検出し，メモ化を行うことで，既存のバイナリを変更することなく高速に実行できる．なお自動メモ化プロセッサは，関数呼び出し命令による遷移先の命令から，関数からの復帰命令までの区間を関数として検出する．

自動メモ化プロセッサのハードウェア構成の概略を**図 1** に示す．自動メモ化プロセッサは，一般的なプロセッサと同様に，ALU，レジスタ（Reg），1 次データキャッシュ（D$1），2 次データキャッシュ（D$2）を持つ．また，自動メモ化プロセッサ独自の機構として，関数およびその入力と出力の組（入出力セット）を記憶しておく表である**再利用表（MemoTbl）**と**メモ化制御機構（Memoize engine）**，および MemoTbl への書き込みバッファとして働く**再利用バッファ（MemoBuf）**を持つ．MemoTbl はサイズが大きく，コアからのアクセスコストが大きいため，入出力を検出する度に MemoTbl へアクセスするとオーバーヘッドが大きくなる．そこで，このオーバーヘッドを軽減するために，作業用の小さなバッファである MemoBuf をコアの内部に設けている．なお，自動メモ化プロセッサでは，レジスタやメモリの参照を「入力」，レジスタやメモリへの書き込み，および返り値を「出力」として扱う．

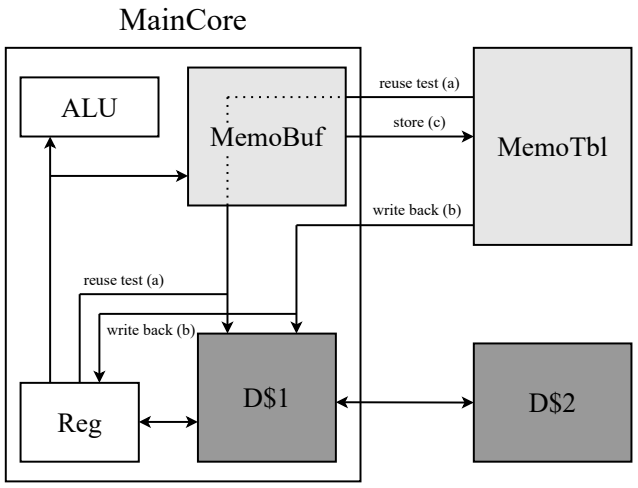

図 1 自動メモ化プロセッサのハードウェア構成

自動メモ化プロセッサは関数を検出すると，MemoTbl を参照し，現在の入力セットと MemoTbl に記憶されている過去の入力セットとを比較する．これを**再利用テスト（reuse test）**（図 1(a)）と呼ぶ．もし，現在の入力セットが MemoTbl に記憶されたいずれかの入力セットと一致する場合，メモ化制御機構はその入力セットに対応する出力セットをレジスタやキャッシュに書き戻し（write back）（図 1(b)），関数の実行を省略する．一方，現在の入力セットが MemoTbl のいずれの入力セットとも一致しない場合，自動メモ化プロセッサはその関数を通常実行しながら，その入出力を MemoBuf に登録し，実行終了時に MemoBuf の内容を MemoTbl に登録（store）（図 1(c)）することで将来の再利用に備える．

さて，一般に関数内では，複数の入力値が順に参照され，使用される．しかし，同じ関数でも，その入力アドレスの列は分岐していく場合がある．例えば，条件分岐命令を実行すると，次に参照されるアドレスはその条件分岐命令の分岐結果によって変化してしまう．そこで，自動メモ化プロセッサは，全入力パターンを木構造で表現し，MemoTbl に登録する．例えば，自動メモ化プロセッサが**図 2** に示すサンプルプログラムを実行する場合，関数 calc の全入力セットは**図 3** に示すような木構造で表現されることになる．なお，図 3 のノードは関数の入力値を，エッジは入力値と次に参照される入力値との対応関係を示しており，End はそれ以上の入力値が存在しないことを示す．また，

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

```
1  int a = 3, b = 4, c = 8;
2  int calc(x){
3    int tmp = x + 1;
4    tmp = tmp + a;
5    if(tmp < 7)
6      tmp = tmp + b;
7    else
8      tmp = tmp + c;
9    return(tmp);
10 }
11 int main(void){
12   int ans = 0;
13   calc(2); /* x = 2, a = 3, b = 4 */
14   b = 5; ans = calc(2); /* x = 2, a = 3, b = 5 */
15   a = 5; ans = calc(2); /* x = 2, a = 5, c = 8 */
16   a = 3; ans = calc(2); /* x = 2, a = 3, b = 5 */
17   return(0);
18 }
```

図 2 サンプルコード

図 3 の (i) は図 2 の 13 行目，(ii) は 14 行目および 16 行目，(iii) は 15 行目における関数呼び出し時の入力セットに対応する．この例において，入力セット (i) および入力セット (ii) では，変数 b が 3 番目に参照されるのに対して，入力セット (iii) では，変数 c が 3 番目に参照される．これは，2 番目に参照される変数 a の値が異なるため，入力セット (i) と入力セット (ii) では，プログラムの 5 行目における if 文の条件を満たし，if 文に含まれるコードを実行したのに対して，入力セット (iii) では，if 文の条件を満たさず，else 文に含まれるコードを実行したためである．

次に，この木構造で表現された全入力パターンを登録するための表である MemoTbl の構成を**図 4** に示す．MemoTbl は，関数を記憶する**関数表（FuncTbl）**，入力を記憶する**入力表（InTbl）**，入力アドレスを記憶する**アドレス表（AddrTbl）**，および出力を記憶する**出力表（OutTbl）**の 4 つの表から構成される．FuncTbl，AddrTbl，OutTbl は RAM で実装し，InTbl は **3 値 CAM（Ternary Content Addressable Memory）**で実装する．CAM は全てのエントリに対して，一致比較を同時に行うことができるため，RAM と比較して高速な連想検索が可能である．

FuncTbl は 1 エントリが 1 関数に対応しており，その行番号（Index）を各関数の識別番号とする．

InTbl の各エントリは FuncTbl の行番号 Index に対応するインデクス（FuncTbl idx）を持ち，この値を用いてどの関数の入力値を記憶しているかを判別する．また，関数の全入力パターンを木構造で管理するために，入力値（input values）に加えて親エントリのインデクス（parent idx）を持つ．この input values は，入力を含むキャッシュライン全体を記憶する．このとき，キャッシュライン中の，入力として参照される値が格納されている部分以外については，3 値 CAM のドントケアを用いて登録する．なお，レジスタの値が入力値となる場合も同様に，複数レジスタの入力値をまとめて 1 つの入力エントリに記憶する．

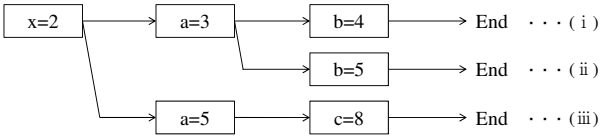

図 3 入力パターンの木構造

AddrTbl は InTbl と同数のエントリを持ち，各エントリは 1 対 1 に対応する．AddrTbl の各エントリは入力値検索のために，次に参照すべきアドレス（next addr）を持つ．また，入力セットの終端エントリか否かを保持するフラグ（ec flag）を持ち，そのエントリが終端エントリである場合，出力を記憶している表である OutTbl のエントリを指すインデクス（OutTbl idx）も持つ．

OutTbl の各エントリは FuncTbl のインデクス（FuncTbl idx）に加えて，関数の出力先のアドレス（output addr），出力値（output values）を持つ．また，出力セットの各エントリをリスト構造で管理するため，次に参照すべきエントリのインデクス（next idx）を持つ．

このような MemoTbl への書き込みバッファである MemoBuf の詳細な構成を**図 5** に示す．MemoBuf は複数のエントリを持ち，1 エントリが 1 入出力セットに対応する．各エントリは，どの関数に対応しているかを示すインデクス（FuncTbl idx），その関数の開始アドレス（Start Addr），その関数の実行開始時のスタックポインタ（SP），関数の戻りアドレス，またはループの場合はその終端アドレス（retOfs），関数の入力セット（Read）および出力セット（Write）のためのフィールドを持つ．なお，Read フィールドおよび Write フィールドは，複数の入出力値を保持できるようになっている．また，入れ子構造になった関数もメモ化対象とするために，MemoBuf は現在実行中の関数に対応する各エントリをスタック構造として保持する．そのため，MemoBuf は現在使用しているエントリをポインタ（MemoBuf_top）で指しており，関数の検出時にそのポインタをインクリメントし，関数の実行終了時にデクリメントすることで入れ子構造に対応している．

2.2 自動メモ化プロセッサの動作例

計算再利用を適用するためには，現在の入力と MemoTbl に記憶しておいた過去の入力とを比較する必要がある．MemoTbl の検索手順を**図 6** に示す．ここで，図 6 中の MemoTbl は図 2 のサンプルプログラムを 15 行目まで実行

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

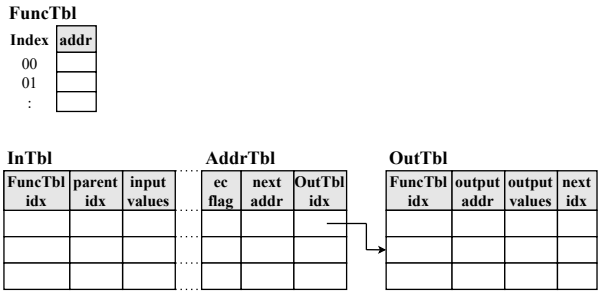

図 4 MemoTbl の構成

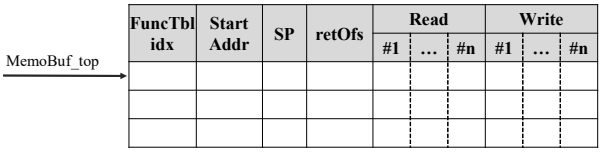

図 5 MemoBuf の構成

した状態を表している．なお，図中の InTbl における X はドントケアを示しており，キャッシュライン内のその値は検索時に比較されない．また，図中の InTbl の parent idx における “N/A” は親エントリが存在しないこと，すなわち，parent idx の値が “N/A” であるエントリは，図 3 で示したような入力パターンの木構造の根に相当するエントリであることを示している．また，図 6 内の矢印は図 2 の 16 行目における関数呼び出し時の MemoTbl 検索フローを示している．

次に，どのような手順で再利用を適用するのかを，図 2 の 16 行目を実行する場合を例に説明する．16 行目を実行し関数 calc が呼び出され，再利用区間の実行開始アドレスが検出されると，まず，FuncTbl が関数 calc の開始アドレス（0x100）をキーとして検索される．これにより得られた index 値を FuncTbl idx フィールドに持ち，input values が現在のレジスタ上の入力値と一致し，かつ parent idx が “N/A” であるようなルートエントリが検索される（図 6(a)）．次に，該当するエントリがライン 00 で発見され，対応する AddrTbl の next addr が 0x200 番地を指しているため，そのアドレスに対応するキャッシュラインを参照する（図 6(b)）．そして，得られた値を input values として持ち，かつ parent idx が 00 であるエントリを InTbl から検索する（図 6(c)）．以降，同様に検索を続ける（図 6(d)(e)）．その後，検索対象のエントリがライン 03 で発見されたとき，当該エントリの ec flag が 1 であることが検出される．これは，当該エントリが入力セットの終端エントリであることを表しているため，一連の入力セットの比較を終える．なお，検索が終端エントリに達するのは，現在の入力セットと過去の入力セットが全て一致した場合のみであるため，自動メモ化プロセッサは再利用テストに成功したと分かる．このとき，終端エントリに対応する AddrTbl エントリの OutTbl idx に登録されているインデクスが指す

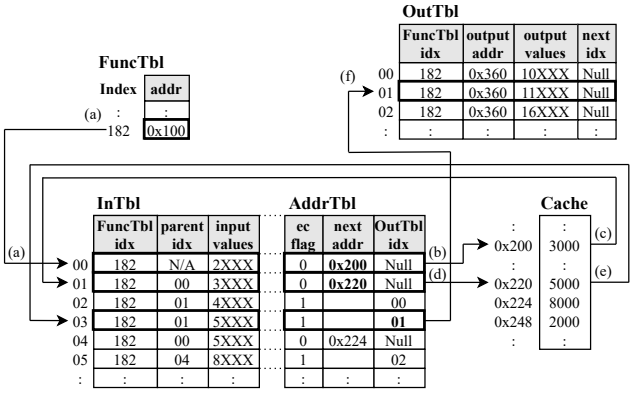

図 6 MemoTbl の検索手順

OutTbl エントリを参照し（図 6(f)），読み出した出力値をレジスタやメモリに書き戻すことで関数の実行を省略することができる．

以上の手順で関数に対して計算再利用を適用するが，再利用テストを行う際には MemoTbl を検索するコストが，再利用テスト成功時には出力セットを MemoTbl からレジスタやメモリに書き戻すコストが，オーバーヘッドとして発生する．そこで，再利用テストを行っている間は，命令パイプラインをストールさせず，現在実行中の関数を，再利用テストとオーバラップして引き続き実行させることによって，再利用テストの MemoTbl を検索する際に発生するオーバヘッドを緩和する．ただし，再利用テスト中のパイプライン実行では，リタイアステージでの処理を禁止している．これは，関数内の命令がリタイアされてしまうと，一致比較対象の入力値が上書きされる可能性があり，その結果，過去と入力が異なるにも関わらず一致比較に成功してしまうことで，不正な再利用が適用されてしまう可能性があるからである．また，再利用テストに成功した場合，出力セットを MemoTbl からレジスタやメモリに書き戻すと同時にパイプラインをフラッシュする．これは，既にパイプラインに投入されている命令は再利用対象区間である関数内の命令であり，実行する必要がないためである．

このように自動メモ化プロセッサは，再利用可能な区間の実行を省略することで高速化を図る手法であるが，これまで自動メモ化プロセッサは，限定的なアーキテクチャへの実装を想定した評価しか行われていない．

3. RISC-V で自動メモ化プロセッサを実現するために

本章では，RISC-V を自動メモ化プロセッサのベースアーキテクチャとして採用する経緯，および RISC-V ABI が自動メモ化プロセッサの実装に与える影響について述べる．

3.1 ベースアーキテクチャの決定

これまで我々は，自動メモ化プロセッサのシミュレーションによる評価を行ってきた．しかし，その評価は一部

の限定的なアーキテクチャでのシミュレーションにとどまっている．

まず，単命令発行でパイプライン化されていない単純なSPARC-V8 アーキテクチャ [5] がベースアーキテクチャとして採用され，研究されてきた．しかし，この自動メモ化プロセッサには 2 つの問題点が存在する．1 つは，自動メモ化プロセッサのベースアーキテクチャとして採用されている，この単命令発行のアーキテクチャが，現在市場に流通しているプロセッサのベースアーキテクチャとして広く採用されているスーパスカラアーキテクチャと，ハードウェア構成が大きく異なっている点である．もう 1 つは，この自動メモ化プロセッサが，SPARC アーキテクチャに強く依存した実装になっているという点である．例えば，SPARC の命令セットや ABI（Application Binary Interface）は，命令区間の検出が容易であるなど，メモ化を行う上で非常に都合が良い．

次に，ARM アーキテクチャ [6] がベースアーキテクチャとして採用され，研究されてきた．しかし，この自動メモ化プロセッサにも問題点がある．それは，このプロセッサが純粋な ARM をベースとしていない点である．そのため，その実装が汎用的であるか否か，またその評価結果が妥当であるか否かについては疑問が残る．

そこで，SPARC および ARM に代わる自動メモ化プロセッサのベースアーキテクチャとして RISC-V アーキテクチャを採用する．RISC-V は，2010 年に開発された新しい命令セットアーキテクチャである．RISC-V はオープン標準であるという特徴があり，今後より一層 RISC-V を利用した研究や開発が行われていくことが予想される．また，RISC-V は RISC-V International という非営利団体が所有している．RISC-V International には多数の企業が参加しており，社会的にも注目を集めている．さらに，RISC-V の命令セットには，中核となる小規模の基本命令セットと，任意に選択できる拡張命令セットが存在する．このようなモジュール性のおかげで，ユーザはアプリケーションに応じた拡張拡張をハードウェアに組み込むか否かを自由に選択することが可能であり，拡張が容易である．また，他のアーキテクチャと比較し，RISC-V は特定のマイクロ・アーキテクチャへ過度に依存しない設計になっており，より汎用的かつ実用的な自動メモ化プロセッサの設計に寄与すると考える．

本稿では，この RISC-V をベースとしたプロセッサ上に自動メモ化機構を実装し，その性能を評価することで，自動メモ化プロセッサの汎用的実用性を確認する．

3.2 RISC-V ABI がメモ化に与える影響

自動メモ化機構を実装する対象であるアーキテクチャの ABI が定める関数呼び出し規約は，自動メモ化プロセッサのハードウェア機構に大きな影響を与える．本節では，

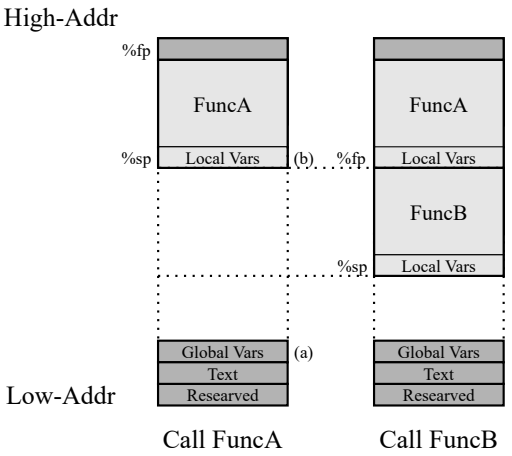

図 7 RISC-V の関数スタックフレーム

RISC-V をベースアーキテクチャとする自動メモ化プロセッサを設計するにあたり，RISC-V ABI が自動メモ化機構に与える影響を検討する．

3.2.1 関数の入出力検出

自動メモ化プロセッサは，関数を再利用するために，実行時にその入出力を記憶しておく必要がある．具体的に，入力の対象となるのは，関数に与えられる引数，大域変数および当該関数の呼び出し元関数の局所変数への参照であり，出力の対象となるのは，返り値，大域変数および当該関数の呼び出し元関数の局所変数への書き込みである．

まず，主記憶参照時における関数の入出力の検出について，**図 7** を用いて説明する．なお，FuncB が計算再利用を適用する関数，FuncA が FuncB の呼び出し元関数であるとする．図 7 は RISC-V ABI が定める関数スタックフレームの構造を示し，各スタックフレームは，左から FuncB 呼び出し前の FuncA での状態，FuncB 実行開始時の状態を示す．主記憶を参照する関数では，大域変数および当該関数の呼び出し元関数の局所変数が入出力となりうる．このうち，大域変数は FuncA の実行より以前に，グローバル領域に静的データとして既に格納されている（図 7(a)），また，当該関数の呼び出し元関数の局所変数は，FuncA の実行時に FuncA のスタック領域に格納される（図 7(b)）．その後，FuncA は FuncB を呼び出す．このように，FuncB における入出力は，両者とも FuncB のスタック領域外にデータが格納されており，FuncB 実行時のスタックポインタ，およびフレームポインタを参照することで両変数が格納されているアドレスが当該関数のスタック領域外であるか判断することができる．よって，当該関数のスタック領域外への参照および書き込みが発生した場合に，それを大域変数もしくは呼び出し元関数の局所変数を対象とする入出力として検出する．

次に，レジスタ参照時における入出力の検出について説明する．RISC-V ABI が定める関数呼び出し規約によると，関数呼び出しの際に，引数を格納するためのレジスタ

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

```
1   addi sp,sp,−32
2   sw s0,28(sp)
3   addi s0,sp,32
4   sw a0,−20(s0) /* a0:arg1 */
5   sw a1,−24(s0) /* a1:arg2 */
6   lw a4,−20(s0)
7   lw a5,−24(s0)
8   add a5,a4,a5
9   mv a0,a5
10  lw s0,28(sp)
11  addi sp,sp,32
12  ret
```

図 8 引数検出の例

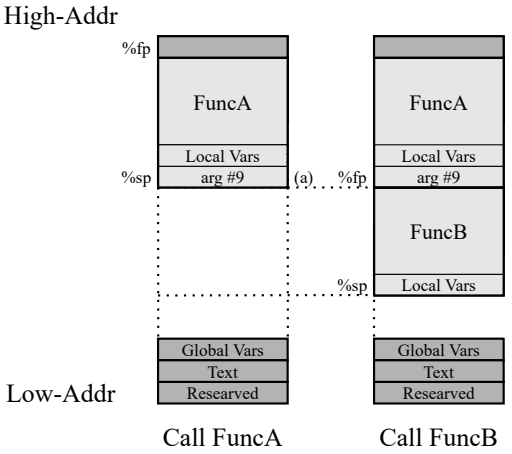

図 9 引数レジスタ数を超えた引数を持つ関数のスタックフレーム

は a0 レジスタから a7 レジスタまでの 8 個と規定されている．本稿では，これらのレジスタを**引数格納レジスタ**と表記する．関数の入出力の対象のうち，引数は，関数が呼び出された際に，引数格納レジスタを介して渡される．

しかし，引数の数を関数の実行前に取得するのは困難であるため，8 個の引数格納レジスタのうち，どれが引数渡しのために使用されているかは分からない．また，引数格納レジスタは呼び出された関数内で引数を参照する以外の用途で使用される可能性がある．そのため，ただ引数格納レジスタの参照を監視するだけでは，そのレジスタに格納されている値が引数であるか否かを判断することはできない．引数を正確に検出するには，引数格納レジスタのうち，どれが引数渡しのために使用されるかを判別する必要がある．そこで，関数の実行開始後に引数格納レジスタへの書き込みが行われるより前にそのレジスタへの参照が発生するような場合は，引数を渡すために使用されていると判断し，それを入力として検出する．

ここで，関数のアセンブリ命令列から，引数を入力として検出する例を**図 8** に示す．なお，図 8 は関数呼び出しのための命令以降のアセンブリ命令列である．また，図 8 において，引数格納レジスタと定められているのは a0 レジスタ，a1 レジスタ，a4 レジスタ，および a5 レジスタであり，当該関数の引数は，a0 レジスタ，および a1 レジスタを介して渡される．図 8 の 4 行目の命令は，a0 レジスタへの書き込みがまだ行われていない状態で，a0 レジスタが初めてソースオペランドに指定されている．よって a0 レジスタに格納されている値を引数として検出する．同様に，5 行目の命令における a1 レジスタに格納されている値も引数として検出する．次に，8 行目の命令では，a4 レジスタおよび a5 レジスタがソースオペランドに指定されているが，6 行目の命令，および 7 行目の命令でデスティネーションオペランドに指定されている，つまり，既にこれらのレジスタへの書き込みが行われているため，a4 レジスタ，および a5 レジスタに格納されている値は引数として検出しない．

3.2.2 引数格納レジスタ数を超えた引数渡し

前項で述べた通り，RISC-V の引数格納レジスタは，a0 レジスタから a7 レジスタまでの 8 個である．したがって，関数呼び出しの際は，8 つ目までの引数はレジスタに，9 つ目以降の引数はスタックに順次格納されて渡される．そのため，前項で述べた引数の検出方法では，9 つ目以降の引数を検出することができない．ここで，引数格納レジスタ数を超えた引数を持つ関数が呼び出された際のスタックの様子を**図 9** に示す．なお，図 9 の各スタックフレームは，左から FuncB 呼び出し前の FuncA での状態，FuncB 実行開始時の状態をそれぞれ表す．FuncB の 9 つ目の引数は FuncB が呼び出される前にスタックに格納される．そのため，9 つ目の引数は FuncB の呼び出し元である FuncA のスタック領域に存在する（図 9(a)）．よって，引数格納レジスタ数を超えた引数渡しが発生する場合においても，前項で述べた主記憶参照時と同様に，当該関数のスタック領域外への参照となるため，引数として検出することができる．

3.2.3 関数呼び出し・関数復帰検出

再利用対象区間である関数を特定するためには，関数呼び出し，および関数復帰を検出する必要がある．これまでに評価を行ってきた自動メモ化プロセッサにも，アーキテクチャごとに関数呼び出し，および関数復帰を検出するための仕組みが用意されている．SPARC アーキテクチャの ISA（Istruction Set Architecture）では，関数呼び出しのための命令は `call`，関数復帰のための命令は `ret` と規定されている．そのため，既存の SPARC をベースとしている自動メモ化プロセッサは，これら二つの命令を監視することにより，関数を検出している．一方，ARM アーキテクチャの ISA では，関数呼び出し，および関数復帰に特定の命令を使用すべきであるといった規定はない．ARM で

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

(C-a) `jal ra, offset`　　(R-a) `jalr zero, ra, 0`

(C-b) `jalr ra, rs, 0`

(C-c)
```
auipc ra, offset
jalr ra, ra, offset
```

図 10　関数呼び出し・関数復帰コード

は関数呼び出し，および関数復帰コードの種類は多岐にわたるため，それら複数のコードを監視することで関数呼び出し，および関数復帰を検出している．このように，ベースアーキテクチャごとに関数呼び出し，および関数復帰に使用される命令の特徴に応じた関数検出が必要である．

そこで，RISC-V の関数呼び出し，および関数復帰に使用される命令の特徴を分析する．RISC-V は，SPARC のような関数呼び出し，および関数復帰のための専用命令を持っていない．そのため，専用命令の代わりに通常の無条件分岐命令と同じ命令が使用されている．RISC-V バイナリにおける関数呼び出し，および関数復帰コードを**図 10** に示す．なお，図は左側に関数呼び出しコード，右側に関数復帰コードを示し，“ra” は関数復帰アドレスを保持するリンクレジスタを，“rs” は任意のレジスタを，“zero” は常に 0 を保持するゼロ・レジスタを示す．また，“offset” は関数の開始アドレスを指定するための値を示す．図 10 に示すように，関数呼び出しには無条件分岐命令である `jal` 命令，および `jalr` 命令が使用され，関数復帰には `jalr` 命令が使用される．関数呼び出し，および関数復帰のための命令であるかは，`jal` 命令，および `jalr` 命令のソースオペランド／デスティネーションオペランドに，どのレジスタがどのような組み合わせで指定されているかで判断できる．このように，RISC-V についても，ARM と同様に関数呼び出し，および関数復帰コードの種類は多岐にわたる．また，RISC-V に限らず，一般的なアーキテクチャは関数呼び出しおよび関数復帰のための専用命令を備えていない場合が多数である．そのため，RISC-V の関数を検出する方法は，他のアーキテクチャに適用できる可能性がある．

そこで，RISC-V の関数呼び出し，および関数復帰の特徴を考慮し，専用の関数検出ユニットを設計する．関数検出ユニットは，関数呼び出し，および関数復帰を検出するフローに従って関数を検出する．ここで，**図 11** に関数検出ユニットによる関数呼び出し，および関数復帰検出フローを示す．まず，関数検出ユニットは，無条件分岐命令を監視する (1)．次に，検知した命令の種類によって，その後の関数呼び出し，および関数復帰検出フローが分岐する．まず，その検知した命令が `jal` 命令である場合 (2)，その命令のデスティネーションオペランドを調べる．デスティネーションオペランドが関数復帰アドレスを保持するリンクレジスタ（ra）である場合 (3)，その命令は図 10(C-a) に該当するため，関数検出ユニットはその命令を関数呼び出しのための命令として検出する．次に，検知した命令が `jalr` 命令である場合 (4)，その命令のデスティネーションオペランドを調べる．デスティネーションオペランドが関数復帰アドレスを保持するリンクレジスタである場合 (5)，その命令のソースオペランドを調べる．ソースオペランドが RISC-V のもう一つのリンクレジスタ（t0）でない場合 (6)，その命令は図 10(C-b)，および (C-c) に該当するため，検知した命令を関数呼び出しのための命令として検出する．これは，`jalr` のソースオペランド，およびデスティネーションオペランドに 2 種類のリンクレジスタを指定した命令には，特別な意味が割り当てられており，関数呼び出しには使用されないためである．一方で，デスティネーションオペランドが常に 0 を保持するゼロ・レジスタ（zero）である場合 (7)，その命令のソースオペランドを調べる．ソースオペランドが関数復帰アドレスを保持するリンクレジスタである場合 (8)，その命令は図 10(R-a) に該当するため，検知した命令を関数復帰のための命令として検出する．以上のように関数呼び出し，および関数復帰のための命令を検出することで，再利用対象区間である関数を検出することができる．

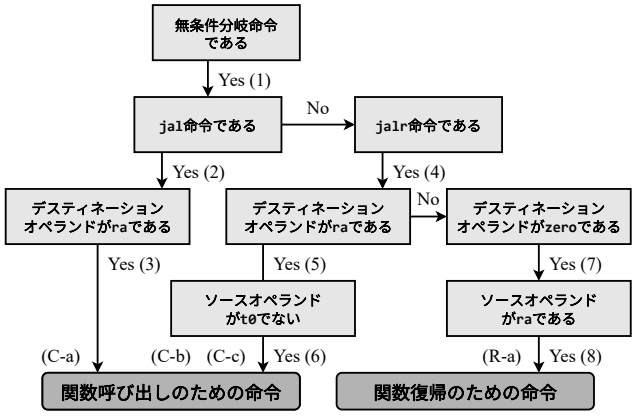

図 11　関数呼び出し・関数復帰検出フロー

3.2.4　動作モードと CSR

RISC-V の動作モードには，ユーザモード，スーパバイザモード，マシンモードの 3 つが存在する．また，RISC-V では CSR（コントロール・ステータス・レジスタ）というシステムレジスタがそれぞれの動作モード毎に定義されている．ユーザモード中でプログラムの実行中に例外が発生した場合，動作モードをスーパバイザモード，またはマシンモードへ移行するために，専用の命令を利用して CSR にアクセスする．

RISC-V では以上のような仕組みで例外を処理しているが，スーパバイザモード，およびマシンモードでの例外処理が自動メモ化機構へ悪影響を及ぼす可能性がある．ここで，例外処理のためにマシンモードへ遷移するアセンブリ例を**図 12** に示す．なお，図中の `csrrw` 命令は CSR

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

```
1    csrrw a0,mscratch,a0 −(a)
2    sw a1,0(a0) −(b)
3    sw a2,4(a0) −(c)
4        ...
5    lw a2,4(a0) −(d)
6    lw a1,0(a0) −(e)
7    csrrw a0,mscratch,a0 −(f)
8    mret −(g)
```

図 12 例外処理のための命令列

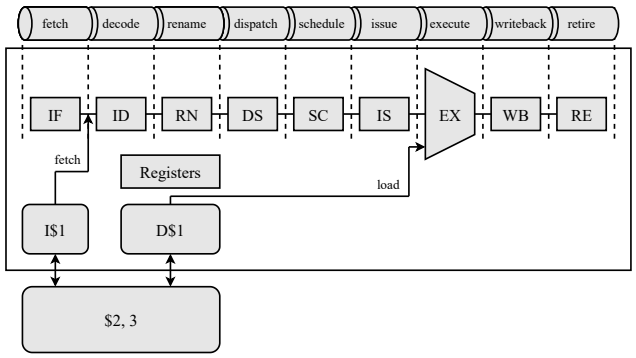

図 13 プロセッサのハードウェア構成

にアクセスするための専用命令の一つであり，mscratch レジスタはマシンモードで定義されている CSR である．ユーザモード中でプログラムの実行中に例外が発生した場合，整数レジスタへの上書きを避けるために，まず整数レジスタと mscratch レジスタの内容をスワップする（図 12(a)）．mscratch レジスタは，割り込み前のデータを退避するための一時ストレージとして 1 データ語を保持するレジスタであり，この例では a0 レジスタに一時ストレージのアドレスを保存している．その後，設定した一時ストレージに，整数レジスタに格納されている値を保存する（図 12(b)(c)）．マシンモードに移行し，例外を処理した後，一時ストレージに保存しておいたデータをレジスタに書き戻す（図 12(d)(e)）．最後に，再び a0 レジスタと mscratch レジスタをスワップして，2 つのレジスタを例外発生前の状態に復元し（図 12(f)）．マシンモードからユーザモードへ制御を戻すための専用命令である `mret` 命令を用いてユーザモードへ復帰する（図 12(g)）．ここで，この一連の処理が関数の実行中に行われる場合を考える．図 12（b）の命令における a1 レジスタについて，関数の実行中に書き込みがまだ行われておらず，関数の実行開始後，初めて同レジスタを参照した場合，3.2.1 項で述べたレジスタ参照時における入力検出の対象になる可能性がある．同様に，図 12（c）の命令における a2 レジスタも入力検出の対象になる可能性があり，それらが誤って関数の入力として検出された場合，本来入力ではない値が再利用表に登録されてしまい，再利用テストが正確に行われない．そのため，例外が発生してからユーザモードに制御が戻るまでは関数の入出力の検出を中止することとした．具体的には，CSR にアクセスするための命令が実行される際に関数の入出力の検出を中止し，ユーザモードに制御を戻すための専用命令が呼び出された後に，関数の入出力の検出を再開する．

4. 自動メモ化プロセッサの実装

本章では，設計した自動メモ化プロセッサの実装について説明する．まず，ベースアーキテクチャにどのように自動メモ化機構を実装するのかについて述べる．その後，実装した自動メモ化プロセッサの動作モデルを示す．

4.1 ハードウェア構成

本稿では，自動メモ化機構を実装するプロセッサとして，現在広く普及しているアウトオブオーダ実行プロセッサを想定する．**図 13** に自動メモ化機構を実装するプロセッサのハードウェア構成を示す．このプロセッサは，1 次命令キャッシュ，1 次データキャッシュ，共有 2 次キャッシュ，および共有 3 次キャッシュを持つ．フロントエンドでは命令フェッチ，デコード，リネーム，ディスパッチを行い，バックエンドでは命令をアウトオブオーダで実行する．また，各パイプラインステージは以下の通りである．

IF (Instruction Fetch)
PC（Program Counter）の値に従って命令キャッシュにアクセスし，命令を読み出す．PC の更新は分岐予測に基づいて行われ，分岐予測器は TAGE-SC-L[7]，および ITTAGE[8] を採用している．

ID (Instruction Decode)
命令を解析し，命令の種類やオペランドなどの情報を得る．

RN (Rename)
命令間の偽の依存を取り除くためにレジスタ・リネーミングを行う．また，リオーダバッファ，および発行キューのエントリを確保し，リオーダバッファに命令の情報を書き込む．

DS (Dispatch)
命令のオペランドがレディか調べた後，命令を発行キューに書き込む．

SC (Schedule)
発行キューにディスパッチされた命令について，依存関係が解決されて実行可能となった命令を起こし，次に実行する命令を選択する．これを交互に繰り返すことで命令の実行タイミングを決定する．

IS (Issue)
次に実行する命令を発行キューからから読み出し，以降のステージに送る．また，同時に発行キューのエントリを解放する．

EX (Execute)
ALU やシフタなどの演算器で演算を行う．実行ステー

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

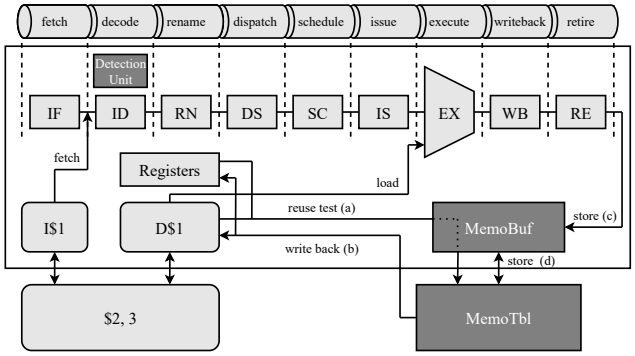

図 14 自動メモ化プロセッサのハードウェア構成

ジでは，演算器がそれぞれ並列に処理を実行する．

WB (WriteBack)

実行ステージで処理された命令の情報をリオーダバッファに記録する．

RE (Retire)

先行命令がすべて完了した命令の実行結果をリオーダバッファから論理レジスタへ書き戻す．なお，分岐予測ミスなどの際に速やかに実行を再開するために，命令がリタイアされる順番はプログラムの命令列と同じであることが保証されている．

4.2 メモ化に必要なハードウェアの追加

本節では設計した自動メモ化プロセッサの実装について説明する．設計した自動メモ化プロセッサの構成図を**図 14** に示す．MemoBuf や MemoTbl などの基本的な構造は，2 章で述べた既存の自動メモ化プロセッサと同様に実装する．

また，RISC-V をベースアーキテクチャとする自動メモ化プロセッサを実現するために，3.2.3 項で述べた関数検出ユニットを追加する．関数検出ユニットによる関数呼び出し，および関数検出フローには命令の種類，ソースオペランド，およびデスティネーションオペランドの情報が必要である．そのため，関数検出ユニットは，デコードステージに追加する．フェッチステージでフェッチされた関数呼び出し，および関数復帰のための命令は，デコードステージで処理される際に，関数検出ユニットにより検出される．

関数呼び出しのための命令を検出した場合，再利用テストを行う（図 14(a)）．再利用テストに成功した場合，再利用表から再利用バッファを介して，レジスタやキャッシュに計算結果を書き込む（図 14(b)）．また，再利用テストに失敗した場合，当該関数の関数復帰のための命令が実行されるまでに，その入出力を登録する（図 14(c)）．その後，関数復帰のための命令を検出した際に，再利用バッファから再利用表へ入出力を書き込む（図 14(d)）．

4.3 自動メモ化プロセッサの動作モデル

本節では，設計した自動メモ化プロセッサの，再利用テストが実行される際の動作を**図 15** のサンプルプログラムの実行に沿って説明する．なお，図 15 における sum 関数のアセンブリ命令列を**図 16** に示す．

```
1  int c = 4;
2  int sum(int a, int b) {
3      return a+b+c;
4  }
5  int main(void) {
6      int x = sum(1,2);
7      x = sum(2,3);
8      x = sum(1,2);
9      return 0;
10 }
```

図 15 サンプルプログラム

```
1   addi sp,sp,−32
2   sw s0,28(sp)
3   addi s0,sp,32
4   sw a0,−20(s0)
5   sw a1,−24(s0)
6   lw a4,−20(s0)
7   lw a5,−24(s0)
8   add a4,a4,a5
9   lui a5,0x12
10  lw a5,−1404(a5)
11  add a5,a4,a5
12  mv a0,a5
13  lw s0,28(sp)
14  addi sp,sp,32
15  jalr zero,ra,0
```

図 16 sum 関数のアセンブリ命令列

まず，図 15 の 6 行目を実行した際の自動メモ化プロセッサの動作を**図 17** に，その動作に対応するパイプラインの様子を**図 18** に示す．なお，以降の説明を簡単にするために，図 17 において，自動メモ化プロセッサの FuncTbl，InTbl，AddrTbl，および OutTbl をまとめて 1 つの MemoTbl として示し，1 つの関数の入出力情報を 1 エントリで表すこととする．また，図 18 における横軸はサイクル数を表しており，自動メモ化プロセッサのパイプラインのウェイ数，およびリタイア幅は 2 とし，各ステージは 1 サイクルで処理されると仮定する．さらに，キャッシュミスなどによりパイプラインがストールすることなく，理想的な状態で各ステージでの処理が実行されるものとする．また，図 18 の左の命令列のうち，`jal` 命令は関数呼び出しのための命令，`jalr` 命令は関数復帰のための命令であり，3 行目から 18 行目までの区間が図 15 の sum 関数に相当するとする．

まず，sum 関数が呼び出されると，図 18 の時刻 t0 で `jal` 命令がフェッチされ，デコードステージにて関数検出ユニットは関数呼び出しのための命令を検出し，自動メモ化プロセッサは再利用テストを行う．この再利用テストを正しく行うためには，関数呼び出しのための命令によるレジスタやキャッシュへの書き込みが完了している必要がある．ここで，関数呼び出しのための命令がリタイアされていれば，それ以前の命令がリタイアされており，関数の入力となる値がレジスタやキャッシュに存在していることが保証される．そのため，再利用テストは関数呼び出しのための命令がリタイアされたタイミングである t1 で行う

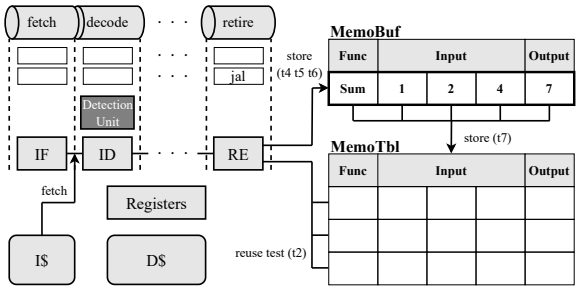

図 17 再利用テスト失敗時の動作モデル

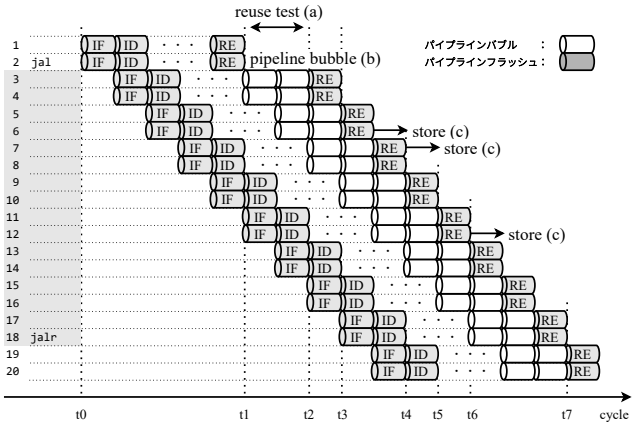

図 18 再利用テスト失敗時のパイプライン動作

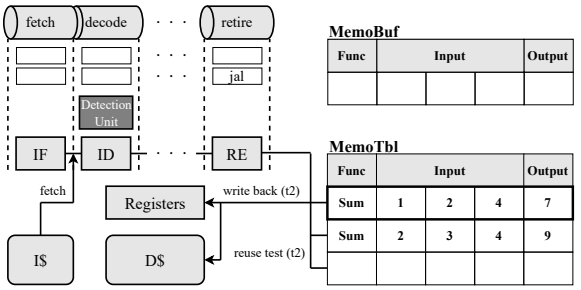

図 19 再利用テスト成功時の動作モデル

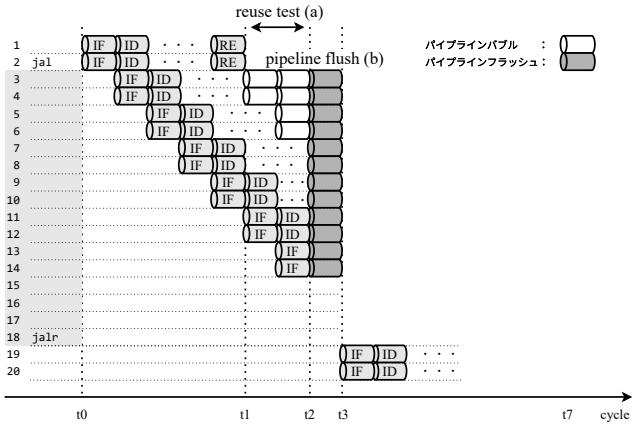

図 20 再利用テスト成功時のパイプライン動作

(図 18(a)). また，2.2 節で述べた通り，再利用テストを行っている間はパイプラインをストールさせないが，リタイアステージでの処理を禁止している．そのため，3 行目以降の命令は，リタイアステージの直前にパイプラインバブルが生じる（図 18(b)).

再利用テスト開始時点で，過去に計算した結果が MemoTbl に登録されていないため，再利用テストに失敗する (図 17(t2)). 自動メモ化機構が再利用テストに失敗し，この関数に計算再利用を適用できないことが判明すると，禁止していたリタイアステージでの処理が再開され，通常通り命令の実行が完了する．図 18 の 3 行目の命令は，再利用テストを行わない場合は t2 の時点でリタイアを完了しているはずであったが，再利用テストによりリタイアステージでの処理が禁止されていたため，処理が再開した後，t3 でリタイアを完了する．さらに，当該関数の実行中に検出した入出力情報を MemoBuf に登録する（図 17(t4 t5 t6)). なお，関数の入出力の登録は，入出力検出の対象となる命令をリタイアしたタイミングで行う．これは，当該命令が挿入されたパイプラインが，分岐予測ミスなどでフラッシュした場合，それに応じて再利用バッファに登録した入出力を無効にする必要があり，ハードウェアが複雑になるためである．ここで，3.2.1 項で述べた入出力検出の対象となる命令が，図 18 では，6 行目，7 行目，および 12 行目であるとすると，それらの命令がリタイアしたタイミングである，t4，t5，および t6 で MemoBuf に登録する（図 18(c)). その後，当該関数の実行が進み，最後に `jalr` 命令がフェッチされ，デコードステージにて関数検出ユニットは関数復帰のための命令を検出する．検出した命令のリタイアが完了するタイミングで，自動メモ化プロセッサは関数の入出力情報を MemoBuf から MemoTbl に書き込む（図 17(t7)).

次に，プログラムの実行が進み，図 15 の 8 行目を実行した際の自動メモ化プロセッサの動作モデルを**図 19** に，その動作に対応するパイプラインの様子を**図 20** に示す．なお，図 20 の横軸のサイクル数は図 17 の横軸のサイクル数に対応している．また，この時点でプログラム 7 行目の sum 関数における再利用テストにも失敗しているため，MemoTbl の 2 行目にその入出力情報が登録されている．まず，sum 関数が呼び出されると，プログラム 6 行目の sum 関数実行時と同様に，図 20 の t0 で `jal` 命令がフェッチされ，デコードステージにて関数検出ユニットは関数呼び出しのための命令を検出し，この命令がリタイアされたタイミングでである t1 で再利用テストを開始する（図 20(a)). また同様に，再利用テストを行っている間，リタイアステージでの処理が禁止された状態で，現在実行中の関数が再利用テストとオーバラップして引き続き実行される．再利用テスト開始時点で，過去に計算した結果が MemoTbl の 1 行目に登録されているため，再利用テストに成功する（図 19(t2)). 自動メモ化機構が再利用テストに成功し，この関数に計算再利用を適用できることが判明すると，再利用テストに成功したタイミングで，過去の実行結果を MemoTbl からレジスタやキャッシュに書き

込む（図 19(t2)）．さらに，既にパイプラインに投入されている命令は再利用対象区間である関数内の命令であり，実行する必要がないため，パイプラインをフラッシュする（図 20(b)）．計算再利用の結果，sum 関数の処理が完了するのは t3 のタイミングであるため，この例では，計算再利用を適用できない場合に比べて (t7 – t3) サイクル削減される．

5. 評価

本章では，設計した自動メモ化プロセッサをシミュレータに実装し，ベンチマークプログラムを用いて評価した結果について考察する．

5.1 評価に使用するシミュレータ

設計した自動メモ化プロセッサをシミュレータ上に実装し，評価する．評価には Championship Value Prediction（CVP）シミュレータ [9] というトレースドリブンシミュレータを使用する．トレースドリブンシミュレーションは，他のシミュレーション方式のシミュレータ，または実機から，アドレストレースと呼ばれるメモリアクセス履歴を取得し，それを基にプロセッサのメモリアクセスを発行させて評価を行う方法である．アドレストレースの取得には RISC-V 向けの命令セットシミュレータである，Spike シミュレータ [10] を使用した．

アドレストレースとして，PC の値，命令の種類，ソースオペランドのレジスタ番号，デスティネーションオペランドのレジスタ番号，およびデスティネーションオペランドのレジスタに格納されている値の情報を，命令ごとに出力させる．さらに，命令の種類によっては付加的な情報が必要になる．命令の種類がロード命令，およびストア命令である場合，アクセス対象のメモリアドレス，およびその範囲の情報も併せて出力させる．命令の種類が分岐命令である場合は分岐が成立したか否か，およびその分岐先のメモリアドレスの情報も併せて出力させる．

また，アドレストレースとして出力させる情報のうち，命令の種類の判別は命令のオペコードを解析することで行う．しかし，3.2.3 項で述べた通り，RISC-V の関数呼び出し，および関数復帰のための命令には無条件分岐命令が使用されるため，オペコードの解析による命令の分類だけでは関数呼び出し，および関数復帰のための命令であるということが判別できない．そこで，関数呼び出し，および関数復帰のための命令を判別するために，3.2.3 項で設計した関数検出ユニットの動作をアドレストレース生成時に再現する．そして，関数呼び出し，および関数復帰のための命令である場合，命令の種類に加えて，関数呼び出し，および関数復帰のための命令であるという情報をアドレストレースとして出力させる．

表 1 自動メモ化プロセッサのシミュレータ諸元

Core	1 core
MemoBuf	128 KiBytes
MemoTbl CAM	256 KiBytes
Comparison	
register and CAM	4 cycles/64Bytes
Cache and CAM	4 cycles/64Bytes
Write back (MemoTbl to Reg/Cache)	1 cycles/64Bytes
L1 I-cache	128 KiBytes
line size	64 Bytes
ways	8 ways
miss penalty	12 cycles
L1 D-cache	64 KiBytes
line size	64 Bytes
ways	8 ways
miss penalty	12 cycles
L2 cache	1 MiBytes
line size	64 Bytes
ways	8 ways
miss penalty	60 cycles
L3 cache	8 MiBytes
line size	64 Bytes
ways	16 ways
miss penalty	150 cycles
pipeline depth	9
pipeline way	2 ways
Reorder Buffer	32 entries

5.2 評価環境

前節で述べた，自動メモ化プロセッサの評価に用いたシミュレータの仕様を**表 1** に示す．なお，MemoTbl 内の InTbl に用いる CAM の容量は 64Bytes 幅× 4096 行の 256KiB とした．また，CAM のクロック周波数は数百 MHz[11] から 1GHz[12] と想定した．一方で，今日の CPU のクロック周波数が 2GHz から 3GHz であることを考慮し，CPU のクロック周波数が CAM のクロック周波数の 4 倍であると仮定して，CAM の検索オーバヘッドを見積もった．

5.3 評価結果

RISC-V をベースアーキテクチャとする自動メモ化プロセッサが，既存の自動メモ化プロセッサの評価結果と同等のパフォーマンスを示すか否かについて検証するために，以下の 2 つのプロセッサを評価した．

(N) 自動メモ化機構を実装していないプロセッサ（baseline）

(R) RISC-V をベースアーキテクチャとする自動メモ化プロセッサ

ここで，自動メモ化機構を実装していないプロセッサ (N) とは，自動メモ化機構を実装していない RISC-V のプロセッサのことである．また，ベンチマークプログラムに

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

表 2 実行命令数の減少率

	Bubble	FFT	Intmm	Mm	Queens
(-O0)	0%	0.4%	0%	0%	55.5%
(-O1)	0%	0%	0%	0%	67.0%
	Towers	Puzzle	Quick	Trees	Perm
(-O0)	51.4%	53.9%	0%	0%	24.5%
(-O1)	27.1%	6.8%	0%	0%	0%

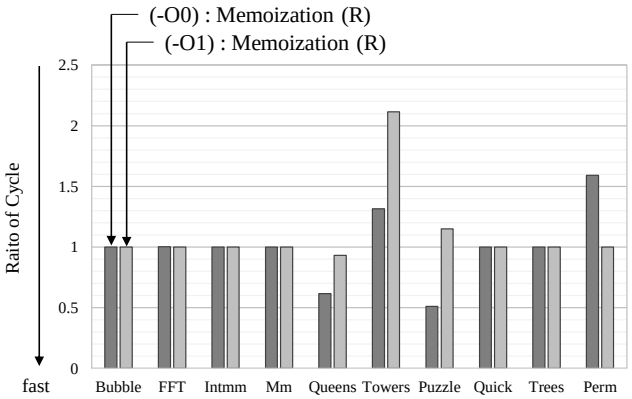

図 21 実行サイクル数比

は，stanford を使用した．なお，stanford はプログラムサイズが小さく，コンパイル時の最適化で関数呼び出し自体が省略されてしまう場合がある．そこで，gcc の最適化オプションを (-O0)，および (-O1) に設定し，ロードモジュールを生成した．なお，以降の説明では，両ロードモジュールを (-O0)，(-O1) と表記する．

まず，RISC-V をベースアーキテクチャとする自動メモ化プロセッサ (R) における実行命令数の減少率を**表 2** に示す．なお，表の各行には，それぞれのロードモジュールにおける実行命令数の，(N) に対する (R) の減少率を示している．この結果から，いくつかのプログラムでは実行命令数が低減しており，関数の実行が省略できていることがうかがえる．プログラム全体では，実行命令数の減少率は (-O0) が平均で 17.9%，(-O1) が平均で 11.2%となり，これが実行サイクル数の削減や IPC の向上につながることが期待できる．

そこで次に，上述した 2 つのプロセッサの実行サイクル数を評価した．その結果を**図 21** に示す．図は各ベンチマークプログラムの実行サイクル数を 2 本のグラフで示しており，左から最適化オプション (-O0)，最適化オプション (-O1) で生成したロードモジュールにおける，RISC-V をベースアーキテクチャとする自動メモ化プロセッサ (R) が要したサイクル数を示している．なお，各グラフは自動メモ化機構を実装していないプロセッサ (N) の実行サイクル数を 1 として正規化している．

評価の結果，RISC-V をベースアーキテクチャとする自動メモ化プロセッサ (R) は，自動メモ化機構を実装していないプロセッサ (N) と比較して，実行サイクル数を最大 49.0%削減したものの，平均では 6.2%増加してしまった．各ベンチマークプログラムに注目すると，実行サイクル数がほとんど変化していないプログラム，削減されているプログラム，および増加しているプログラムの 3 つに大別できる．

まず，実行サイクル数がほとんど変化していないプログラムである Bubble，FFT，Intmm，Mm，Quick，Trees，および (-O1) の Perm について，表 2 中の結果を見ると，実行命令数がほとんど，または全く削減されていないことが確認できる．このことから，計算再利用を適用できる関数がほとんど存在しないために，再利用テストに失敗する場合が多いことがわかる．そのため，再利用テスト失敗時に発生するオーバヘッドが実行サイクル数の増加に影響を及ぼすはずである．しかし，実行サイクル数の増加に至らなかったのは，再利用テスト失敗時のレイテンシが隠蔽されているからであると考える．ここで，再利用テスト失敗時のレイテンシが隠蔽される様子を**図 22** に示す．図は再利用テストを開始してから，失敗が判明し後続命令の処理が再開された際の，パイプラインの様子を示す．図では 2 行目の `jal` 命令がリタイアされた後，再利用テストが開始され，リタイアステージでの処理が禁止される．その際に発生するパイプラインバブルが実行サイクル数の増加に影響を及ぼすことが予想される．しかしここで，7 行目の命令を実行する際にデータキャッシュミスが発生し，パイプラインがストールしたとすると，その遅延がボトルネックとなり，リタイアステージでのストールによる遅延は隠蔽される．このことを確かめるために，実際にベンチマークを実行した際のパイプラインの様子を，パイプライン可視化ツール Konata[13] を用いて調査した．その結果を**図 23** に示す．図は (-O0) の Bubble を実行した際のパイプラインの様子を表しており，(i) は関数呼び出しのための命令である．(i) がリタイアすると，再利用テストが開始され，後続命令である (ii) のリタイアステージでの処理が禁止されるため，(ii) のリタイアするタイミングは計算再利用を適用しないで実行する場合よりも遅くなる．しかし，(iii) の命令でデータキャッシュミスが発生し，その大きなレイテンシによって，(ii) のリタイアでのレイテンシは隠蔽されていることがわかる．

このように，ほとんどの場合において，再利用テストに失敗した際に発生するパイプラインバブルによる遅延は，全体の実行サイクル数の増加にほとんど影響を及ぼさないことがわかった．結果として，自動メモ化プロセッサの実行サイクル数の増加に大きな影響を及ぼすのは，再利用テストに成功した際に発生するレイテンシがほとんどであると考えられる．

次に，Queens，Towers，Puzzle，および (-O0) の Perm は，表 2 中の結果を見ると，実行命令数が削減されていることが確認できる．しかし，これらの実行サイクル数を見ると，実行サイクル数が削減されているプログラムと，増

Vol.2022-ARC-248 No.2
Vol.2022-SLDM-198 No.2
Vol.2022-EMB-59 No.2
2022/3/10

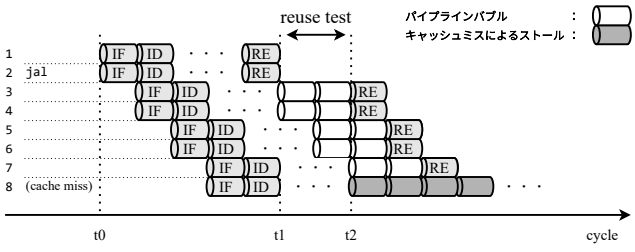

図 22 再利用テスト失敗時のレイテンシが隠蔽される様子

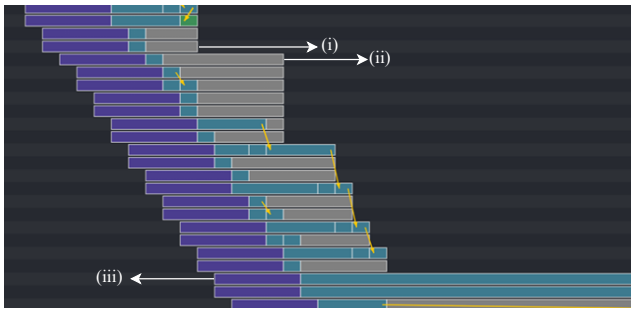

図 23 Bubble におけるパイプラインの様子

加しているプログラムが存在する．このように，プログラムによって示す傾向が異なるのは，再利用テストの際に発生するオーバヘッドの大きさと，再利用テストに成功することで削減されるサイクル数の大きさが関係している．再利用テストの際に発生するオーバヘッドは，MemoTbl を検索するレイテンシ，および入力が一致したエントリに対応する出力を MemoTbl からレジスタやメモリに書き戻すレイテンシからなる．もし，一致比較を行う入力の数や，レジスタやメモリに書き戻す出力の数が多い関数に計算再利用を適用する場合，これらのレイテンシが大きくなり，再利用テストの際に発生するオーバヘッドが，再利用テストに成功することで削減されるサイクル数よりも大きくなる可能性がある．このとき，再利用テストに成功したのにもかかわらず，実行サイクル数は増加してしまい，かえって性能は悪化してしまう．実行サイクル数が削減されているプログラムでは，このような関数が少なかったため，再利用テストに成功することで性能が向上したが，実行サイクル数が増加しているプログラムでは，このような関数が多かったため，再利用テストに成功することで，かえって性能が悪化してしまった．そのため，再利用テストに成功すると性能が悪化する関数については，計算再利用の適用を中止する必要がある．なお，既存の自動メモ化プロセッサには，計算再利用の適用を中止する機構である，**オーバヘッドフィルタ（Overhead Filter）**が実装済みであり，この機構が自動メモ化プロセッサの性能低下を抑制できることを確認している．そのため，RISC-V をベースアーキテクチャとする自動メモ化プロセッサにおいても，既存の自動メモ化プロセッサと同様に性能低下を抑制できると考えられる．さらに，従来の機構では，再利用テストの成功率，過去の再利用テストによるオーバヘッド，および削減サイクル数の履歴を基に，計算再利用を関数に適用するか否かを推測していた．しかし本稿で，再利用テストに失敗した際のオーバヘッドはほぼ考慮しなくてもよいことが明らかになったため，オーバヘッドフィルタによる計算再利用を適用するか否かの判定が，従来よりも簡単かつ正確に行えると考えられる．

以上の評価結果より，実行サイクル数の削減に成功した場合が存在したことから，RISC-V をベースアーキテクチャとする自動メモ化プロセッサは，既存の自動メモ化プロセッサと同様に性能向上が期待できることが確認できた．

6. おわりに

本稿では，自動メモ化プロセッサの汎用的実用性と改良の余地を検討するために，RISC-V を新たに自動メモ化プロセッサのベースアーキテクチャとして採用し，RISC-V アーキテクチャベースのプロセッサ上に自動メモ化機構を実装する場合に発生しうる問題を検討し，自動メモ化プロセッサを設計した．

設計した自動メモ化プロセッサの有効性を検証するために，stanford ベンチマークを用いてシミュレーションにより評価した．評価の結果，RISC-V をベースアーキテクチャとする自動メモ化プロセッサは，自動メモ化機構を実装しない場合と比較して，実行サイクル数を最大 49.0%削減したものの，平均では 6.2%増加してしまった．しかし，実行サイクル数の削減に成功した場合が存在したことから，RISC-V をベースアーキテクチャとする自動メモ化プロセッサは，既存の自動メモ化プロセッサと同様に性能向上が期待できることが確認できた．

今後の課題として，既存の自動メモ化プロセッサが備えている，性能向上のための機構を実装することが挙げられる．例えば，5.3 節で述べたオーバヘッドフィルタを実装することで，再利用テストによる性能低下を最小限にとどめることができる．また，現在の実装では再利用表に登録されている入力のエントリが一杯になった場合，新たな入力の登録を中止しているため，本来計算再利用による実行の省略が期待できた関数を省略できなくなってしまう可能性がある．そのため，適切なアルゴリズムを用いて再利用表のエントリを消去するように実装することで，性能の向上が期待できる．さらに，再利用テストにおけるオーバヘッドを削減するために，可能な限りの処理のパイプライン化を図ることが挙げられる．具体的には，MemoTbl を検索するオーバヘッドは，現在の入力をキャッシュから読み出すのにかかるレイテンシ，およびキャッシュから読み出した入力と MemoTbl に記憶されている入力とを一致比較するのにかかるレイテンシからなる．現在の実装ではその 2 つを逐次処理しているが，キャッシュから読み出した入力と MemoTbl に記憶されている入力とを一致比較している

間，次の入力をキャッシュから読み出す処理を並行して行える可能性がある．

謝辞 本研究の一部は，JSPS 科研費 21H03408 の助成を受けたものである．

参考文献

[1] 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).

[2] Waterman, A.: Design of the RISC-V Instruction Set Architecture, PhD Thesis, EECS Department, University of California, Berkeley (2016).

[3] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).

[4] Huang, J. and Lilja, D. J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc. 5th Int'l Symp. on High-Performance Computer Architecture (HPCA-5)*, pp. 106–114 (1999).

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

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

[7] Seznec, A.: TAGE-SC-L Branch Predictors Again, *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, (online), available from ⟨https://hal.inria.fr/hal-01354253⟩ (2016).

[8] Seznec, A.: A 64-Kbytes ITTAGE indirect branch predictor, *JWAC-2: Championship Branch Prediction*, San Jose, United States, JILP, (online), available from ⟨https://hal.inria.fr/hal-00639041⟩ (2011).

[9] Championship Value Prediction: Championship Value Prediction (CVP) simulator, `https://github.com/eric-rotenberg/CVP`.

[10] RISC-V International: Spike, a RISC-V ISA Simulator, `https://github.com/riscv-software-src/riscv-isa-sim`.

[11] Xilinx Inc.: *Ternary CAM Search v2.2 LogiCORE IP Pruduct Guide*, 2.2 edition (2021).

[12] Agrawal, B. and Sherwood, T.: Ternary CAM Power and Delay Model: Extensions and Uses, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 16, No. 5, pp. 554–564 (online), DOI: 10.1109/TVLSI.2008.917538 (2008).

[13] Ryota Shioya: Konata, `https://github.com/shioyadan/Konata`.