動的命令変更機構を持つ組み込み向けJavaプロセッサの設計と評価

鈴木真人[†]　木村晋二[†]　渡邉勝正[†]

近年，携帯情報端末にJavaの実行環境を搭載することが一般的になりつつある．Javaバイトコードを直接実行できるJavaプロセッサはソフトウェアと比較して実行速度及び省メモリ性に優れており，組み込み分野への応用が期待されている．ここでは，Javaプロセッサの実行効率を向上するための手法として，動的命令変更機構を提案し，その評価を行なう．Javaでは演算の前後にスタック操作が必要となるが，バイトコード実行中にこれらの命令列をRISC型の拡張命令に動的に変更して命令キャッシュを書き換えることで，繰り返し実行での効率を向上させる．この動的命令変更機構を付加したJavaプロセッサを設計し，その性能評価を行なう．

Design and Evaluation of Java Processor with Dynamic Instruction Conversion Mechanism for Embedded Systems

Masato Suzuki,[†] Shinji Kimura[†] and Katsumasa Watanabe[†]

Java processors are key for executing Java bytecodes in embedded systems, and are expected low hardware consumption and high execution efficiency. Inthe paper, we propose a dynamic instruction conversion mechanism for a Java processor. Java bytecode is based on stack operations, and we can raise the efficiency by changing these codes into extended codes corresponding to RISC-like operations. Rewriting is done in cache and done in parallel with the direct execution of bytecode. By performing the execution and the conversion in parallel, we can manipulate complex conversions with low hardware cost. The paper shows the design and evaluation of the Java processor with the dynamic instruction conversion mechanism.

1. はじめに

近年，携帯電話を始めとする携帯情報端末の高機能化が著しい．その一つとして携帯情報端末にJavaの実行環境を搭載することが一般的になりつつある．Javaはオブジェクト指向プログラミング言語であり[1)]，マルチプラットフォームによる開発効率，安全性，ネットワーク親和性に優れている．今後は携帯端末ばかりでなく，情報家電向きの組み込みシステムのための標準言語になると思われる．組み込みシステムでのJavaの実行環境としては，インタプリタによるソフトウェア実行が一般的である．しかし，組み込みシステムに用いられるCPUは低速である事が多く，今後のJavaアプリケーションに必要とされるパフォーマンスを得ることができない．一般のパーソナルコンピュータではJITなどコード最適化を行なうことで処理速度を向上できるが，搭載メモリ量の少ない組み込みシステムにおいては困難である．そこで，Javaバイトコードを直接実行できるJavaプロセッサおよびその高速化手法が注目を集めている．

組み込みシステムにJava実行環境を実装する場合は，実行速度と消費メモリ量とのトレードオフを考慮する必要がある．インタプリタより高速で，ソフトウェアによる最適化に比べてメモリを消費しないJavaプロセッサは組み込みJavaに最も適していると考えられる．Javaプロセッサの研究開発は広く行なわれており，Sun社のpicoJava[5)]が広く知られている．最近では，通常のRISCへのJava処理系の組み込み[6)]や，Javaプロセッサに再構成可能部を導入する手法の研究[3)4)]も行なわれている．

本稿では，Javaプロセッサの実行効率をより向上するための手法として，動的命令変換機構を提案する．Javaのバイトコードは，スタック上のデータに対する処理を基本としている．このため，演算の前後にスタック操作が必要となり，これが実行効率を低下させている．提案する手法では，バイトコード実行中にこれらの複数命令をRISC型の拡張コードに書き換えることにより，従来より少ないサイクルでの実行を可能としている．このコード書き換えは命令キャッシュ

† 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

上で行ない，書き換えられた拡張コードはキャッシュが無効になるまでは何度も利用されるため，繰り返し実行での高速化が期待できる．バイトコードの実行と命令変更を並行して行なうことで，コード変換のハードウェア量を抑えたままで，複雑な命令変更に対応できる．

2では，研究背景として Java 及び Java 実行方式についての説明を行なう．3では，提案する動的命令変更機構の構成について説明する．4では，シミュレーションにより動的命令変更機構を持つ Java プロセッサの評価を行なう．

2. Java 実行方式とその実装手段

ここでは，研究の背景となる Java について，その特徴と実行方式について述べる．また，Java をハードウェア化して組み込みシステムに利用する場合の利点及びその実装法を述べる．

2.1 Java 言語

Java は，並列処理が可能な汎用目的のオブジェクト指向プログラミング言語である．C 言語に近い構文でありながら，ポインタ機能などの複雑かつ安全でない言語的特徴を省いたものとなっている．不要リソースの解放もガーベッジ・コレクタによる自動管理で行なわれるため，安全性が高い．

2.1.1 クラスファイルとその構造

Java 言語で記述したプログラムを Java コンパイラでコンパイルすると，クラスファイルが生成される．これは，プログラム実行に必要な静的情報をまとめたバイナリデータである．基本的に 8 ビットのバイトストリームから成り，データ構造は C 言語に似た疑似構造体で定義される．

2.1.2 Java バイトコード

Java バイトコードはクラスファイルのメソッド構造体中に収められている Java 独自の命令セットである．Java バイトコードは，1 バイトの命令コードと命令毎にバイト長の異なるオペランドから構成される 201 種類の可変長命令である．命令の粒度は様々であり，一般的にオブジェクト指向をサポートする命令ほど粒度が大きい．また，コード自身がデータ型情報を持っている（データ型別にバイトコードが用意されている）．Java 仮想マシンはクラスファイル中の Java バイトコードを実行イメージとしてメモリに展開し，処理を行なう．

2.2 Java 実行方式

2.2.1 Java 仮想マシン

一般に Java の実行環境は Java 仮想マシンとして提供されている．Java 仮想マシンは汎用レジスタを持たず，データの演算及び処理などにはすべてスタックを用いる．仮想マシン方式を取ることで安全性の確保とプロセッサアーキテクチャからの独立を実現している．Java 仮想マシンの仕様[1] ではバイトコードの命令セットは規定されているが実装の詳細については規定されておらず，クラスファイルを仕様通りに処理することができれば良い．

Java 仮想マシンの機能として，以下のものがある．

- バイトコードの実行
  Java バイトコードの実行を行なう．Java 仮想マシンの機能の中心である．
- クラスのロード
  外部からクラスファイルを動的にロードする．クラスファイルの持つ情報を動的にメモリに展開する機能が必要である．
- バイトコードのベリファイ
  Java はネットワーク等を通じて外部からクラスファイルをロードすることができる．そのため，ベリファイアは新しくロードしたクラスファイルが Java 仮想マシンで正常に動作することを保証する必要がある．ベリファイアを搭載できない小容量の Java 仮想マシンにおいては，クラスファイルを事前にベリファイしておかなければならない．
- マルチスレッド管理
  Java はマルチスレッドをサポートしている．Java 仮想マシンにはマルチスレッドの管理機能が求められる．
- ガーベッジコレクション
  Java では，不要となったメモリの解放をガーベッジコレクタが行なう．

2.2.2 Java の高速化手法

Java 仮想マシンはバイトコードの実行をインタプリタ方式で行なう．この時の実行速度面での問題を解決するために様々な高速化手法が用いられてきた．各実装方式は以下のように分類できる．

- インタプリタ方式
  バイトコードを一命令ずつ解釈して実行する．低速ではあるが，メモリ消費量が少なくすむため，現時点では組み込み Java 実装方式の主流である．
- JIT(Just-In-Time) コンパイラ方式
  実行時にバイトコードをネイティブコードに変換する．実行速度はインタプリタ方式の数倍早くなるが，メモリの消費量も数倍になる．また，最適化を強くする程，立ち上がりが遅くなる．
- AOT(Ahead-Of-Time) コンパイラ方式

実行前に予めバイトコードをネイティブコードに変換する．JIT コンパイラ方式同様，実行速度は早くなるが，メモリの消費量も多くなる．コンパイルを別の計算機で予め行ない，生成したネイティブコードを利用する方法もあるが，Java の持つ柔軟性は失われる．

- 動的コンパイラ方式
  最初はバイトコードのままインタプリタで実行する．インタプリタで 実行中にプロファイラがアプリケーションのボトルネックを検出する．その後，ボトルネック部分のみを JIT で変換するのだが，この時にプロファイル情報も利用して強力な最適化をかけて高速なコードを生成する．JIT コンパイラ方式より立ち上がりが早く，より効率的な高速化が行なわれる．しかしメモリの消費量は多い．
- Java プロセッサによる直接実行方式
  バイトコードをネイティブコードとして直接実行するプロセッサを用いる．汎用プロセッサに比べて数倍高速であり，メモリ消費が極めて小さい．しかしながら汎用性は無く，また開発コストがかかる．

2.3 Java バイトコードとハードウェア実現容易性

Java 仮想マシンに Java プロセッサを適用することを考える．Java 仮想マシンとして要求される機能すべてをハードウェア化することは困難であるので，Java プロセッサは Java バイトコードの実行を高速化するコプロセッサとして用い，ハードウェア実行できない部分はホスト CPU 上でソフトウェアによる処理を行なうことにする．

Java プロセッサを含んだ Java 仮想マシンは図 1 のフローに示される流れに沿って Java プログラムの実行を行なう．まず，ホスト CPU 側でクラスファイルのロードを行なう．ロード後，必要となるバイトコードをメモリに展開し，Java プロセッサ側に実行権を移す．Java プロセッサ側で処理不可能な命令があった場合，Java プロセッサ側から割り込みがかかる．そのときは，要求された処理をホスト CPU 側で行なう．

Java プロセッサがサポートすることは Java バイトコードの実行である．しかしすべての Java バイトコードを Java プロセッサ上で実行することは難しい．なぜなら，Java バイトコードは命令毎に異なった粒度を持っており，粒度が非常に大きい命令 (例えばメソッド呼び出し，新規インスタンス生成など) はワイヤードロジックでの処理が困難だからである．また，浮動小数点演算のような大きな回路を必要とする演算命令

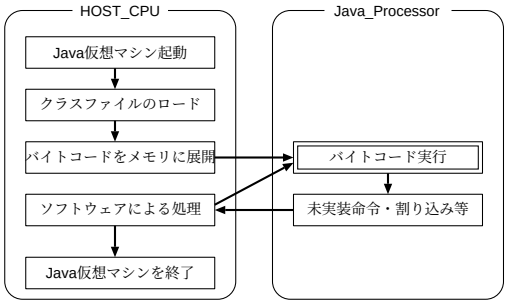


図 1 Java 仮想マシンの動作フロー

もソフトウェアで処理を行なうことが多い．Java バイトコード種類別でのハードウェア実現容易性は以下のように分類される．

- 実現が比較的容易
  ロード，ストア，定数プッシュ，32 ビット整数演算，インクリメント，分岐
- ハードウェアの拡張により実現可能
  64 ビット整数演算，浮動小数点演算，型変換，スタック直接操作，フィールドアクセス，コンスタントプールアクセス，配列操作，wide 命令
- 実現困難
  メソッドの呼び出しとリターン，インスタンス生成，多次元配列生成，例外，オブジェクト型検査，モニタ

3. 動的命令変更機構

以下では，Java プロセッサの問題点を述べるとともに，それを解決する一つの手法として動的命令変更機構の提案を行なう．

3.1 複数命令の一括実行

前章でも述べた通り，JavaVM ではスタックマシンを採用している．スタックマシンはハードウェアを簡潔にでき，命令コードをコンパクトにすることが出来る．しかしスタックマシンはレジスタマシンに比べて，プロセッサの性能を制限する原因にもなる．

Java バイトコードでは演算命令の前後にスタック操作命令が入る．例えば「iload_1, iload_2, iadd, istore_3」という命令列があるとする．この命令列はローカル変数 1 番地及び 2 番地の値をそれぞれスタックへロードし，加算した結果をローカル変数 3 番地にストアするものである．この命令列を実行すると，図 2 のように 4 サイクルかけて加算を行なう．

ここでレジスタを利用できるようにすると，図 3 のように，ロード・演算・ストアを 1 サイクルで一括実行可能となる．以上の様に Java プロセッサにレジス

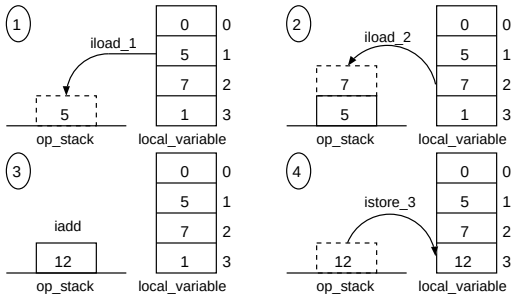


図 2 スタックを用いた演算処理

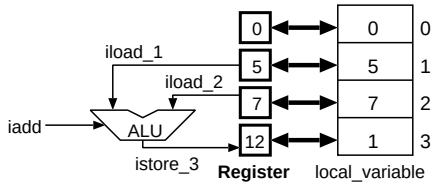


図 3 レジスタを用いた演算処理の一括実行

タを付加し，スタック操作を含む命令列を一括実行できれば効率の向上が見込まれる．

ここで，どのような命令列が一括実行可能かを考える．まずバイトコードの中で，頻出かつ命令一括実行が可能なものを選び，それを 5 種類に分類した．

- **Load** レジスタ→スタックへのロード
  iload, iload_0. . .
- **Const** スタックへの定数プッシュ
  iconst_0, . . ., bipush, sipush
- **Store** スタック→レジスタへのストア
  istore, istore_0, . . .
- **Exec** 演算実行
  iadd,isub,iand,ishl など
- **Branch** 分岐命令
  if_icmpeq,if_ne など

これらの命令をレジスタが有効活用できるパターンで組み合わせる（表 1）．

Const + Const + Exec + Store(わざわざ 2 つの定数を足してからレジスタにストアする) のような普通プログラムされないパターン，何度も実行されないであろうパターンに関しては組合せから除外した．

3.2 拡張コードと動的命令変更

レジスタを用いた複数命令一括実行を行なうためには，命令デコーダを拡張して複数命令のデコードを可能にすればよい．しかし，Java バイトコードは各命令のバイト長が異なる．例えば，iload 命令は通常命令とローカル変数アドレスの 2 バイトから成るが，0〜3 番地までのローカル変数読み込み用に 1 バイト命令 iload_0〜iload_3 が用意されている．例えば，iload_1, iload_2, isub, istore_3 という命令列は 4 バイトであるが，iload 5, iload 6, isub, iload 7 という命令列は 7 バイトである．そのためフェッチした命令を順次解析しながら読み込まなければならず，命令デコーダが複雑化する問題がある．

そこで命令デコーダの複雑化を避けるため，拡張コードを用意する．拡張コードは，それぞれが表に分類される 2〜4 命令の組合せから成っている．iload_0〜iload_3 と iload，istore_0〜istore_3 と istore をそれぞれ区別して命令を作成した． 拡張コード毎にフェッチした命令列の読み込み元が固定されているため，命令解析の複雑性を解消できる．例えば「iload_2, iload, 演算, istore」という命令列を表す拡張コードを用意し，それを「X」とする（図 4）．X という命令をフェッチした場合のデコード処理は，プロセッサに実装されている．図 5 は拡張コード X を含む命令列の一括デコードの例である．命令デコーダはまずフェッチした命令列の先頭を見る．先頭は「X」なのでまずレジスタの 2 番地からのロードであることがわかる．次にもう一つのロードすべきレジスタの番地は，命令列の 3 番目の値である．同様に演算の種類を 4 番目，ストアすべきレジスタの番地は 6 番目からそれぞれ引けば良い．

拡張コードの総数は 205 である．バイトコードの空き分だけでは収まらないため，バイトコードを 1 ビット拡張して 9 ビットとする．命令の書き換えはメモリで無く，命令キャッシュに対して行なう．命令キャッシュをバイト毎に 1 ビットずつ拡張することで，容易に拡張コードへの書き換えが可能となる．命令キャッシュ内の拡張コードがループなどで繰り返し実行されるとき，効率向上が期待できる．

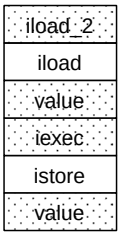
拡張コードに動的変換可能な命令列を発見するために，逐次的な命令パターン解析を行なう．プログラム稼働時には Java バイトコードをそのまま通常実行する．Java バイトコードのデコード時にその命令がどのような種類の命令かを判定して，結果をキューに入れる．命令列解析ユニットは，クロック毎にキューと規定した命令列パターンのマッチングを行なう．パターンと一致した場合，対応する拡張コードでその命令列の先頭を書き換える．先頭のコードのみを書き換えるため，書き換え後に命令列の途中へジャンプする場合でも問題なく動作する．(図 6, 7, 8, 9)

3.3 Java プロセッサのアーキテクチャ

本節では，設計する Java プロセッサの仕様について述べる．

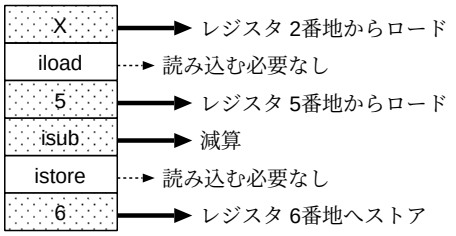
4 命令	Load+Load+Exec+Store Load+Const+Exec+Store Const+Load+Exec+Store	レジスタの値同士もしくは, レジスタの値と定数の演算結果を レジスタに書き込む
3 命令	Load+Load+Exec Load+Const+Exec Const+Load+Exec	レジスタの値同士もしくは, レジスタの値と定数の演算結果を スタックにプッシュする
	Load+Load+Branch Load+Const+Branch Const+Load+Branch	レジスタの値同士もしくは, レジスタの値と定数との比較を行ない, 成功した場合に分岐判断する
2 命令	Load+Store	レジスタからレジスタへの値コピー
	Const+Store	レジスタに定数をストア
	Load+Branch (Branch はゼロ比較のみ)	レジスタの値とゼロとの比較を行ない, 成功した場合に分岐する

表 1 一括実行可能な命令列パターン



X=iload_2,iload,iexec,istore

図 4 拡張コードの一例



フェッチした命令列

図 5 拡張コードを含む命令列のデコード

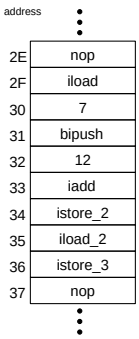


図 6 拡張コードに変換前の命令列

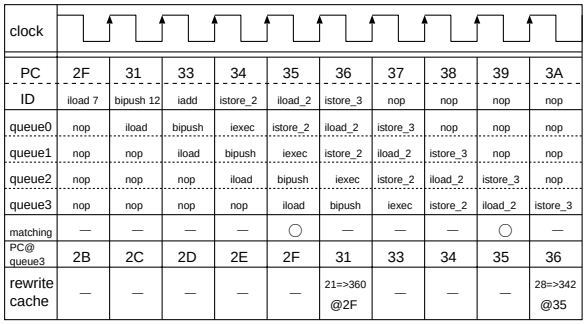


図 7 命令変換のフロー例

3.3.1 基 本 構 造

DLX アーキテクチャを拡張した 6 段パイプライン (IF,ID,SW,EX,MEM,WB) 構造とした (図 10)．SW 段は ID 段の役割を分割したものである．

未実装命令は内部割り込みを発生させソフトウェア処理で対処させることにした．オペランドスタックは 32 ビット 16 段とし，オーバーフロー・アンダーフローが発生した際にはメモリへの退避及び復帰を行なう．

パイプラインに乱れが生じた際にはフォワーディングまたは最小限のパイプラインストールで対処する．

命令フェッチ (IF)

メモリ上のバイトコードを 4 バイトずつプリフェッチして命令キャッシュに読み込む．また，命令変更機構による命令キャッシュの実行時変換も行なう．命令キャッシュの容量は 288 バイト（(8 ビット+1 ビット)*256）である

命令デコード (ID)

Java の全命令 201 中 127 個を実装した．未実装命令及び未定義命令は，割り込みによりソフトウェアで処理を行なう．

データ制御 (SW)

ID 段からの制御信号に従って，EX 段に入力する値を選択して出力する．SW 段の出力は定数・即値・スタック値・データレジスタ値・プログラムカウンタ値があり，レジスタ A,B,E に格納される．また分岐の

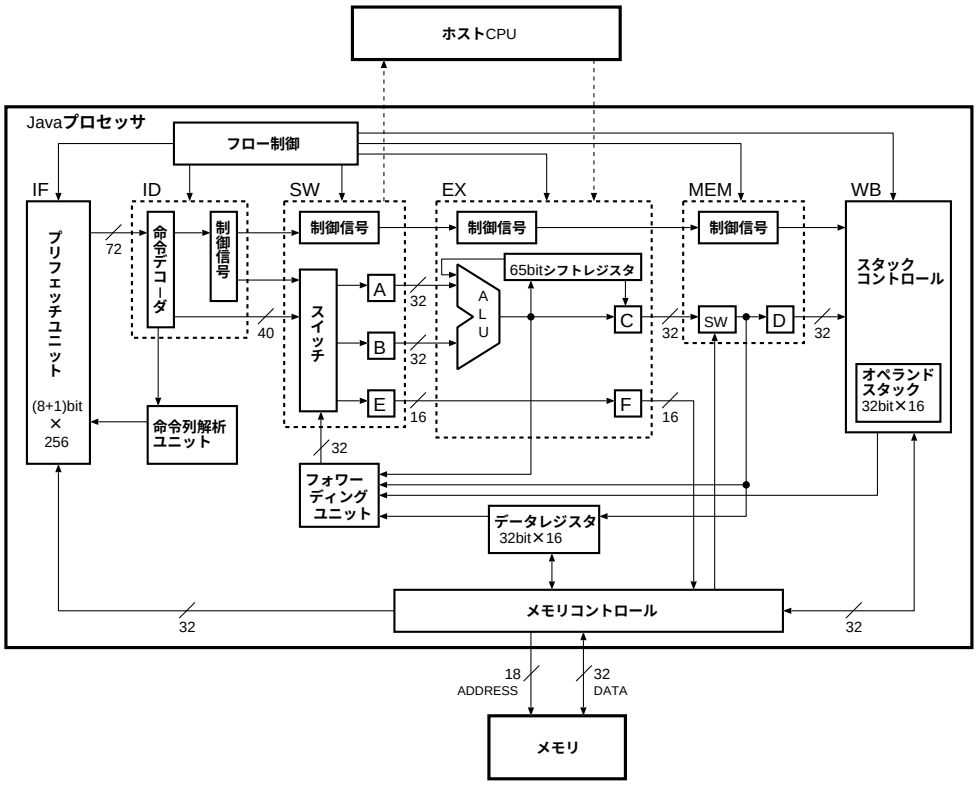


図 10 Java プロセッサのアーキテクチャ

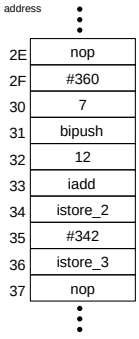


図 8 拡張コードに変換後の命令列

clock						
PC	2F	35	37	38	39	3A
ID	360	342	nop	nop	nop	nop
queue0	nop	nop	nop	nop	nop	nop
queue1	nop	nop	nop	nop	nop	nop
queue2	nop	nop	nop	nop	nop	nop
queue3	nop	nop	nop	nop	nop	nop
matching	—	—	—	—	—	—
PC@ queue3	2B	2C	2D	2E	2F	35
rewrite cache	—	—	—	—	—	—

図 9 拡張命令実行のフロー例

条件判断も行なう．

演算実行 (EX)

演算種類情報に基づき演算を行なう．内部演算器は，32 ビット整数加減算，論理及び算術シフト，論理演算，符号拡張機能，整数比較機能を有している．さらに，65 ビットシフトレジスタを追加することにより 32 ビット整数の乗算，除算，剰余算を行なうことができる．乗算は booth 法を用いて 16 クロックで，除算と剰余算については引き戻し法を利用して 33 クロックで結果を出力する．

メモリアクセス (MEM)

ロード，ストア命令が実行されたとき，メモリとのデータ入出力を行なう．メモリのアクセスが重複した場合，決められた優先順位に従って処理を行ない，他のアクセスはストールさせて処理を行なう．また，命令列一括実行のためのデータレジスタを備えており 0〜15 番地（64 バイト分）までのデータを高速に読み書きすることができる．

ライトバック (WB)

演算結果，もしくはメモリからロードした値をオペランドスタックに積む．同時に SW 段で消費された分を消去する．スタックへの書き込みは半クロックずれ

て実行される．また，スタックアンダーフロー・オーバーフローの処理も行なう．

フォワーディングユニット

データハザードによるパイプラインストールを解消するため，必要なデータがスタックへまたはデータレジスタから読み出せるのを待たずに，パイプライン中から取りだして使用できるようにする．命令によって最大 4 オペランドのフォワーディングを行なう．

命令列解析ユニット

3.2 節で述べた命令列解析及び拡張命令の発行を行なう．デコードされた命令をパイプライン的処理を行ない，一括実行可能な命令列を検出する．検出時には拡張命令及び変換アドレス情報をプリフェッチユニットに送る．

フロー制御ユニット

各種ハザード発生時にパイプラインの制御を行なう．本プロセッサはメモリバスが 1 つしかなく，各種メモリアクセスが競合した場合に構造ハザードが発生する．その場合，次の優先順位に従って実施し，その他の命令はストールを行なうことで対処する．

(1)　内部スタックオーバー・アンダーフロー
(2)　ローカル変数（メモリ）の読み書き
(3)　命令フェッチ

また，フォワーディングで対処できないデータハザードが発生した場合や，複数クロックを必要とする演算命令（乗・除・剰余算）や分岐命令（条件判断及びジャンプアドレス計算）の処理時にもパイプラインを適切にストールさせて対処する．

4. シミュレーションによる動的命令変更機構の評価

提案した動的命令変更機構をソフトウェアシミュレーションにより評価する．

4.1 実 験 環 境

提案する Java プロセッサについてレジスタ転送レベル (Register Transfer Level, RTL) での設計を行ない，C 言語を用いてハードウェア記述言語 (Hardware Description Language, HDL) 風に記述したシミュレータを用いて実験を行なった．バイトコードを実行すると，終了時に実行に要したクロック数を表示する．このクロック数表示機能を用いて，動的命令変更機構の有無による速度差を調べる．テスト用プログラムとしては，Embedded CaffeineMark benchmark3.0[7] を使用する．これには 6 種のベンチマークが含まれている．

- **Sieve**…エラストテネスのふるいにより素数を求める
- **Loop**…フィボナッチ数列のソート
- **Logic**…論理演算による決定木の操作
- **String**…文字列連結，文字列検索
- **Float**…3D 物体の回転シミュレーション
- **Method**…再帰的なメソッド呼びだし

今回は，この中から Sieve, Loop, Logic ベンチマークについてシミュレーションを行なった．String, Float, Method についてはハードウェアで実行できる命令が少なく，動的命令変更機構の評価には向いていないと判断したためである．

表 2　実行クロック数及び高速化率

CaffeineMark	変換なし	変換あり	高速化率
Sieve	531,214clock	481,596clock	9.4%
Loop	216,618clock	155,395clock	28.3%
Logic	456,085clock	330,193clock	28.7%

表 3　CaffeineMark/MHz 値

CaffeineMark	本手法	J2ME[2]	A[2]	HotSpot[8]
Sieve	5.75	0.52	9.65	8.00
Loop	17.32	0.50	14.65	25.77
Logic	11.55	0.49	8.35	17.47

4.2 実 験 結 果

各ベンチマークにおいて，動的命令変更を行なうときと行なわないときそれぞれの実行クロック数を計測し，それを基に高速化率 $(= 1 - \frac{\text{変換ありでのクロック数}}{\text{変換なしでのクロック数}})$ を求めた．結果を表 2 に示す．

次に，CaffeineMark 値を求める．これは各ベンチマークの処理速度から求められる．今回の実験では動的命令変更を行なった側のベンチマークの結果を元に CaffeinMark 値を求めた．さらにこれを動作周波数 (MHz) で割ったものを求め，他実装方式と比較を行なった．「J2ME」は Sun UltraSparc450MHz 上でインタプリタ実行される仮想マシンである[2]．「A」は 2) で設計された Java プロセッサである．「HotSpot[8]」は 900MHz で動作する PC 上の Linux において動的コンパイル方式で高速化された JDK1.3 を使用したものである．それぞれの CaffeineMark/MHz 値をまとめたものを表 3 に示す．

4.3 考　　察

動的命令変更機構により最大 28.7%の高速化率を得た．また，インタプリタ型と比べクロック当たりで 10 倍以上の処理速度が出ることを確認した．繰り返しの多い Loop ベンチマークと Logic ベンチマークにおいては，プロセッサ A よりも良い結果となった．

Logic ベンチマークの結果は命令キャッシュ容量により大きく変わる．命令キャッシュ容量を半分の 144 バイトにすると，書き換えた命令が実行される前に他アドレスの命令に上書きされてしまうため，高速化率が 0%となった．逆にキャッシュ容量を倍の 576 バイトにすると，高速化率は 30.8%となった．

Sieve ベンチマークにおいて，乗算・除算が遅いため全体のクロック数が増えた．その結果，高速化率が低く抑えられてしまっている．

全体的な問題としては，配列処理やコンスタントプールへのアクセス時間が挙げられる．データレジスタをキャッシュとして利用できるように拡張する必要性がある．データキャッシュを使えば一括実行可能な命令列の種類が増えると考えられる．

今回実験を行なわなかった String,Float,Method ベンチマークについて，高速化率は 1%程度となることがわかった．これはメソッド呼び出しや浮動小数点演算をソフトウェアで処理したことで，全体の高速化率が低下したものと考えられる．

5. おわりに

本稿では，Java プロセッサの実行効率を向上するための手法として，動的命令変換機構を提案した．また，動的命令変更機構を持つ Java プロセッサを設計し，ソフトウェアシミュレーションによる評価を行なった．動的命令変換機構を用いることで，繰り返しの多い命令の実行を効率良く高速化することができる．バイトコードの実行と命令変更を並行して行なうことで，命令デコーダのハードウェア量を抑えたままで，複雑な命令変更に対応できる．今後，今回行なわなかったベンチマークの詳細テスト及び，プロセッサとしての実装を行なう予定である．

謝辞

日頃から御討論，御助言頂く堀山貴史助手，中西正樹助手をはじめとする本学言語設計学講座の皆様に深く感謝します．また，研究に際し資料等を御提供いただき，御討論も頂いた名古屋大学の高木一義先生，鬼頭秀明氏に深く感謝します．

なお，本研究は一部文部科学省科学研究補助金および平成 12 年度稲盛財団助成金による．

参考文献

1) ティム・リンドホルム, フランクリン・イェリン（村上雅章 訳）," Java^{VM} 仮想マシン仕様 第 2 版", ピアソンエデュケーション,2001.
2) Motoki Kimura, Morgan Hirosuke Miki, Takao Onoye, Isao Shirakawa, “High Performance Java Execution for Embedded Systems”, SASIMI2001 pp346-350,2001.
3) Shinji Kimura, Hiroyuki Kida, Kazuyoshi Takagi, Tatsumori Abematsu, Katsumasa Watanabe, “An Application Specific Java Processor with Reconfigurabilities”, ASP-DAC2000,2000.
4) 鬼頭秀明, 高木一義，木村晋二, 高木直史, “再構成可能部を持つ Java プロセッサにおけるハードウェア JIT 機構の検討” 信学技報, CPSY2000-65,2000.
5) “picoJava”, http://www.sun.co.jp/tech/whitepapers/
6) “Jazelle”, http://www.arm.com/armtech/Jazelle_Tech
7) “CaffeineMark 3.0”, http://www.webfayre.com/pendragon/cm3/index.html
8) “HotSpot”, http://java.sun.com/products/hotspot/