JAIST Repository
https://dspace.jaist.ac.jp/

Title	軽いハードウェアによるJava高速化手法に関する研究
Author(s)	吉兼，寛
Citation	
Issue Date	2004-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1775
Rights	
Description	Supervisor:田中 清史，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

軽いハードウェアによる Java 高速化手法に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

吉兼 寛

2004 年 3 月

修 士 論 文

軽いハードウェアによる Java 高速化手法に関する研究

指導教官　田中清史 助教授

審査委員主査　田中清史 助教授
審査委員　日比野靖 教授
審査委員　井口寧 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

210102 吉兼 寛

提出年月: 2004 年 2 月

Copyright © 2004 by Yoshikane Hiroshi

概 要

近年，組み込みシステム向けの言語としてマルチプラットホーム，ネットワーク親和性，安全性などの点で Java 言語が注目されている．Java 仮想機械は，アプリケーションをインタプリタ形式で実行するため実行性能が低く計算パワーが必要なアプリケーションの実行には問題がある．この問題に対する解決手法はいくつか存在するが，組み込み機器に適用するにはメモリ使用量が多過ぎる，Java のバイトコード以外を直接実行できないなどの問題が 生じる．本論文では，Java 仮想機械の実行領域の一部をハードウェア上にレジスタとして実現し，命令の最適化及びメモリアクセス数の軽減をおこなうことで高速化をおこなう手法を提案し評価する．その結果，従来手法と比べ最大で局所的な命令の最適化は 1350%，総サイクル数は 43．38%の性能向上を得ることができた．

目 次

第 1 章　はじめに **1**
1.1 背景と目的 1
1.2 本論文の構成 1

第 2 章　Java の概要 **3**
2.1 Java の特徴 3
2.1.1 プラットホーム独立性 3
2.1.2 オブジェクト指向言語 4
2.1.3 スタックマシンアーキテクチャ 6

第 3 章　Java 仮想機械 (JavaVM) **7**
3.1 Java 仮想機械実行フロー 7
3.2 クラスファイル 7
3.2.1 コンスタントプールエントリ 11
3.2.2 フィールドエントリ 11
3.2.3 メソッドエントリ 11
3.3 データ構造 11
3.3.1 メソッド領域 11
3.3.2 ヒープ領域 12
3.3.3 スレッド領域 12
3.4 バイトコード (ByteCode) 13

第 4 章　提案手法 **15**
4.1 ハードウェアレジスタ 15
4.2 Java 専用レジスタ (Java Exclusive Register FILE) 16
4.3 提案手法の命令セット 18
4.4 パイプライン化 19
4.4.1 専用命令解析ステージ 19
4.4.2 専用命令実行ステージ 23

第 5 章　評価 **25**
5.1 提案手法による命令最適化 25

5.2 評価方法 27
5.3 評価結果 32
5.4 追加ハードウェア量評価 33

第 6 章 関連研究 **35**

第 7 章 最後に **37**
7.1 まとめ 37
7.2 今後の課題 37

第1章　はじめに

1.1　背景と目的

従来，組み込みソフトウェアはC言語やアセンブラ言語を用いて開発，運用されてきた．これらの言語は，各モジュールがプログラム全体に影響を及ぼす可能性が大きく，ソフトウェア開発の向上が困難である．また，複数の機器を連係させた新たな利便性を実現するために多種多様で複雑な処理を搭載する要求が高まっている．

これらのことから，組み込みシステム向けの言語としてマルチプラットホーム，ネットワーク親和性，安全性などの点からJava言語が注目されている．Java言語はプログラムをネットワークからダウンロードして実行することも可能となってきており，Java言語の処理機構を組み込んだ携帯端末や家電製品などへの応用が急速に進んでいる．

Java言語は，Javaソースファイルを Javaコンパイラによってコンパイルすることによりクラスファイルを生成し，そのクラスファイルを Java仮想機械が読み込み実行する．しかし，Java仮想機械は読み込まれたクラスファイル中のバイトコード実行をインタプリタ形式で実行する形式のものが一般的であり，その実行性能は高いものではない．この実行速度の問題を解決するために，JITコンパイラ [1]，ホットスポット [4] などの提案がなされてきたが，これらの提案は再コンパイルをおこなうことにより高速な処理を実現するものである．そのため，組み込み機器においてはその処理を実行するために必要なメモリの確保が困難である．

また，Javaのバイトコードを直接実行することが可能な Javaチップ [2] も提案されたが，Javaのバイトコード以外のアプリケーションを直接実行できない問題がある．

本研究では，Java仮想機械においてバイトコードの実行に必要なオペランドスタック及びローカル変数がメモリ上に動的に確保されていることに着目し，それらをCPU上でレジスタとして実現し命令の最適化をおこない高速化を達成する手法を提案する．

1.2　本論文の構成

本論文の構成を以下に示す．

第2章 Java言語の特徴及びオブジェクト指向について述べる．

第3章 Javaクラスファイルの構造，及び Java仮想機械の仕様について述べる．

第 4 章 本研究で提案するハードウェアレジスタ化をおこなった CPU の仕様，及びその動作について述べる．

第 5 章 評価対象のプログラムを実行したサイクル数を従来方式と比較し，その結果の考察，及び追加したハードウェア量に関する考察について述べる．

第 6 章 本研究の関連研究について報告する．

第 7 章 まとめと今後の課題について述べる．

第2章　Javaの概要

本章では，Java の特徴，及びそのオブジェクト指向について述べる．

2.1　Javaの特徴

Java には以下のような特徴が挙げられる．

- プラットホーム依存が無く，それぞれの実行環境が独立しており，プログラマが実行環境の違いを意識する必要がない．
- オブジェクト指向言語である．
- JavaVM は，スタックマシンアーキテクチャ方式を採用している．

以下にそれぞれの特徴について詳しく述べる．

2.1.1　プラットホーム独立性

分散環境下においては，あるプログラムを様々なプラットホームで実行させたい場合が生じる．実行をさせたいプラットホーム用の実行形式プログラムが入手することが出来ない場合，そのソースコードを入手し移植が可能であれば，そのソースコードをコンパイルすることにより，そのプラットホームでの実行が可能である．つまり，そのソースコードが入手不可能であったり，コンパイル，移植が不可能であった場合にはそのプラットホームでは実行できない．

Java 言語は，プラットホームに依存しない言語である．これは，プラットホームに依存しない仮想機械 (Java 仮想機械) を導入することにより，プラットホームの違いを吸収することを可能としたからである (図 2.1)．Java 言語のプログラムはこの仮想機械用の実行形式にコンパイル (クラスファイル化) し，Java 仮想機械が実行をおこなうので，分散環境などの異なったプラットホーム下において実行する場合，再コンパイルなどの複雑な処理をせずに実行をおこなうことが可能である．

そのため，Java 言語ではソースコードレベルでの移植性の問題を注意しながらプログラミングする必要はない．また，同じパッケージを用いてさまざまなプラットホームで実行可能なマルチプラットホームアプリケーションをを開発することも可能となっている．これは，分散処理環境を構築する上で非常に大きな利点となる．

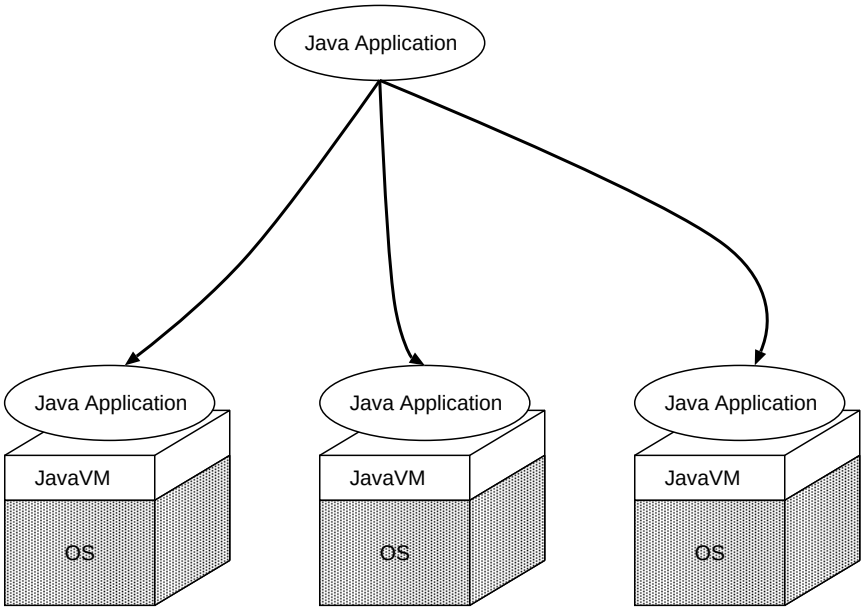

図 2.1: Java アプリケーションのマルチプラットホーム性

2.1.2 オブジェクト指向言語

本節では，オブジェクト指向の特徴について述べる．オブジェクトとは実世界に存在するものや概念のことである．オブジェクトは，属性 (data) と手続き (method) を持つ．

抽象化 (Abstraction)

抽象化とは，複数のオブジェクトの属性や手続きのうち，共通の部分を抽出することを指す．つまり複数のオブジェクトの原型 (クラス) を作ることに相当する．また，これをインスタンス (実体) にすることをインスタンス化という (図 2.2).

継承 (Inheritance)

継承とは，ある抽象化されたクラスを基に拡張したクラスを作ることを意味する．基となるクラスをスーパークラス (親クラス)，拡張したクラスをサブクラス (子クラス) と呼ぶ (図 2.3).

カプセル化 (Encapsulation)

カプセル化とは，あるオブジェクトの属性に対して決められたインターフェース以外の方法では，データ操作を許可しないようすることである．このモジュール化によりプログ

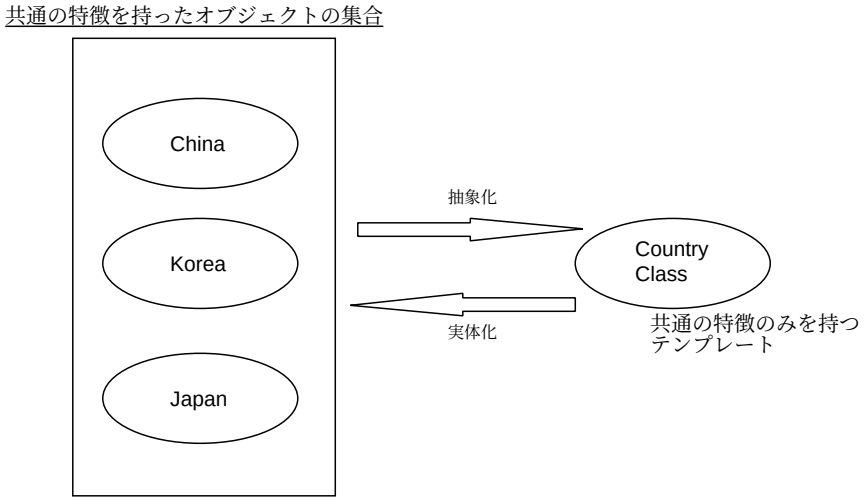

図 2.2: オブジェクトの抽象化例

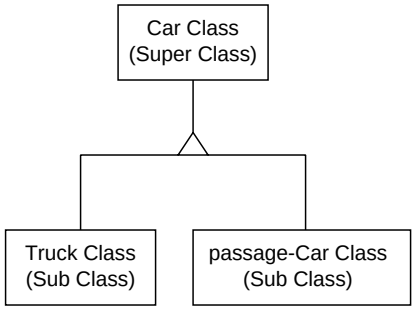

図 2.3: 継承

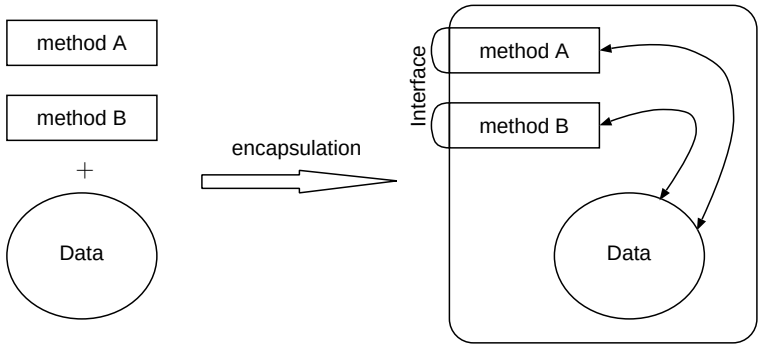

図 2.4: カプセル化

ラムコードの保守性を高めることができる (図 2.4).

多相性 (Polymorphism)

多相性とは，異なるオブジェクトに対して同じメッセージを送った場合に，そのオブジェクトの特性に応じた振舞を設定できることをいう.

2.1.3 スタックマシンアーキテクチャ

Java 仮想機械は，スタックマシンアーキテクチャ方式を採用している．他方，現代の CPU はレジスタマシンアーキテクチャ方式を大多数が採用している．レジスタマシンアーキテクチャ方式では，非常に高速なアクセスが可能であり容量の小さな記憶装置であるレジスタを CPU 内に保持し，演算結果の保持などに極力利用することによって高速な処理を実現している．このレジスタの数量，及び形態等については各プラットホームによって異なる.

Java は，各プラットホーム上に Java 仮想機械を構築する必要が生じる．Java 仮想機械のアーキテクチャとしてレジスタマシンアーキテクチャを採用した場合，それぞれのプラットホームのレジスタの数量，及びアクセス形態の違いが問題となる．その問題はスタックマシンアーキテクチャ方式を採用することで回避できる．スタックマシンアーキテクチャ方式では，プログラムの実行を進めていく上で，使用するレジスタの存在を仮定しない．そのため，レジスタが存在しない，もしくはレジスタ数の少ないプラットホームでも Java 仮想機械を実行することが可能となっている.

第3章　Java仮想機械(JavaVM)

本章では，最初にJava仮想機械の実行フローの概要を示す．その後，Java仮想機械の実行に必要なクラスファイルの構造について示し，最後にそのクラスファイル中に存在するバイトコードがJava仮想機械でどのように使用され, オペランドスタック上でどのようにデータを動作させるかを示す.

3.1　Java仮想機械実行フロー

Java仮想機械は，それ自身がアプリケーションであり，実行時に読み込ませたクラスファイルの情報を基にバイトコードの内容を忠実に実行する (図3.1).

Java仮想機械の動作は大きくわけて以下のように分類される.

- 指定された (もしくは, 必要に応じて) クラスファイルの情報を正確に読み込み，その情報を格納する領域を動的に確保する
- 読み込まれたクラスファイルの内容に従い, バイトコードを正しく実行する

3.2　クラスファイル

本節では，Java仮想機械の入力部に相当するクラスファイルの構造を示す. クラスファイルはいくつかのブロックで構成され，そのひとつひとつのブロックがさらに小さなブロックより構成され，そのブロックが更に細部構造を持つものである．これらの各構造は入れ子関係，もしくは継承関係となっている (図3.2).

図3.2では,ClassFileはクラスファイルを示し,Constant_Pool_infoはコンスタントプールエントリを示すが, クラスファイルとコンスタントプールエントリは1対nの関係にあり, その個数はクラスファイルによって異なるものである.

クラスファイルは大きく分けて以下の3つの部分に分けることができる.

- コンスタントプールエントリ
- フィールドエントリ
- メソッドエントリ

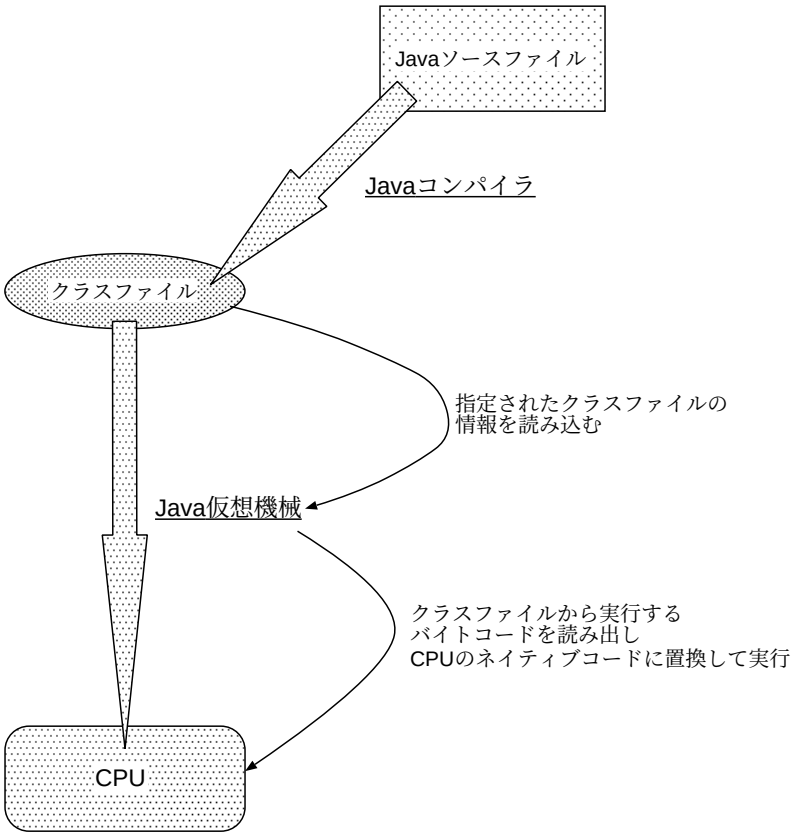

図 3.1: Java 実行フロー

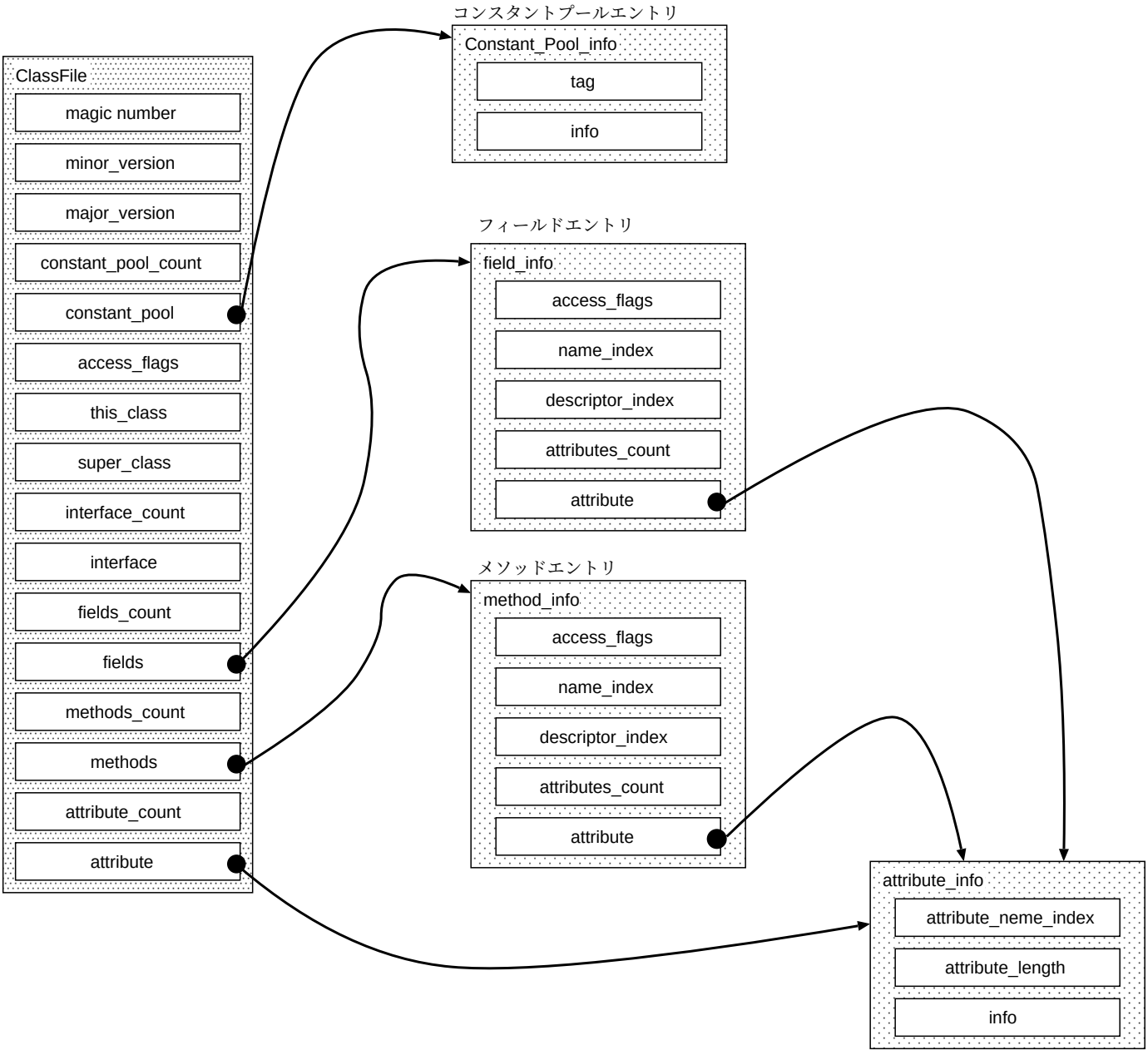

図 3.2: クラスファイル構造の入れ子関係

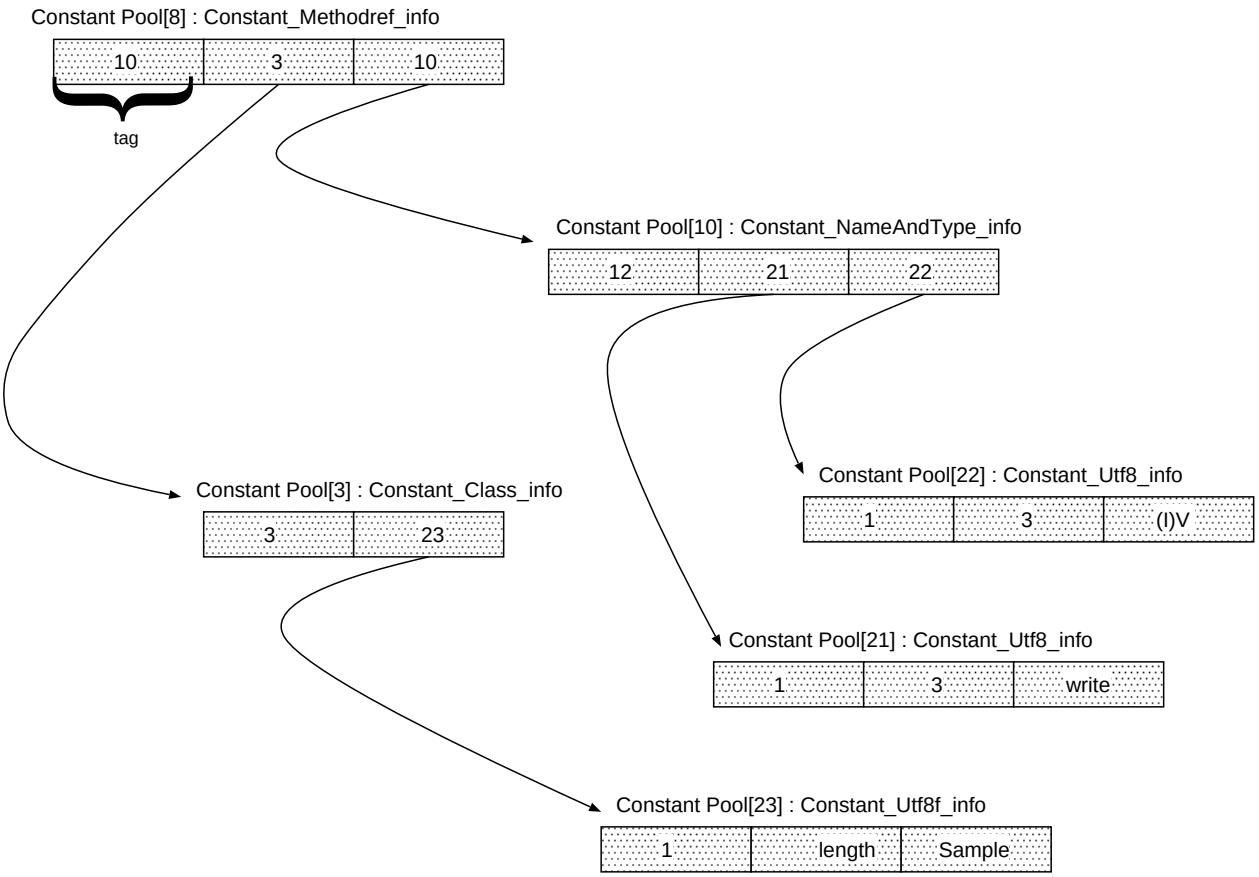

図 3.3: コンスタントプールを辿り Sample クラスの write メソッドを参照する例

コンスタントプールエントリを辿り，メソッド情報を参照する例を示す (図 3.3).

図 3.3 では，メソッド呼び出しのバイトコード命令が読み込まれ，コンスタントプールエントリインデックスとして値 8 が与えられメソッド情報を解決するためにコンスタントプールのエントリを辿った場合の例である．[ ] 内の数字はコンスタントプールエントリのインデックス番号である．また，それぞれのコンスタントプールエントリのそれぞれの内容は可変長であり，共通するのは最初の tag 部分だけである．この tag 情報により，そのコンスタントプールエントリの内容を Java 仮想機械は知ることができるのである．図では，最初のコンスタントプールエントリから tag 以降の情報を辿っていくことにより，クラス名，メソッド名，メソッドディスクリプターに対応する情報を解決することができる．

クラスファイルはこの他に，そのクラスファイルの直接のスーパークラスであるクラスファイルの情報をコンスタントプールへのインデックスとして保持している．これにより，Java 仮想機械はその情報を利用してクラス間の継承関係を知ることができる．

3.2.1 コンスタントプールエントリ

コンスタントプールエントリは，自身のクラスファイルが使用する全ての定数の情報を保持する．この情報はフィールドエントリやメソッドエントリがコンスタントプールを介して間接的に参照することができる．

3.2.2 フィールドエントリ

フィールドエントリは，Java 言語のクラスにおけるフィールド定義に対応している．それぞれのフィールドエントリの中に名前，型，アクセスフラグのコンスタントプールエントリのインデックスが保持されている．

3.2.3 メソッドエントリ

メソッドエントリは，Java 言語のクラスにおけるメソッド定義に対応している．フィールドエントリと同様に名前，型，アクセスフラグのコンスタントプールエントリへのインデックスが保持されている．また，メソッドエントリはメソッドに対応するバイトコード情報を保持している．

3.3 データ構造

本節では,Java 仮想機械のデータ構造について述べる. データ構造は以下の 3 つに分類することができる (図 3.4).

- メソッド領域 (クラス)
- ヒープ領域 (インスタンス)
- スレッド領域 (Java スタック)

これらのデータ領域について詳しく述べる.

3.3.1 メソッド領域

Java 仮想機械は，実行時にクラスファイルを ClassLoader によって読み込みその情報をメソッド領域に格納する．Java 環境では，ガベージコレクションの機能により不必要になった情報は廃棄されるが，このメソッド領域は一般的にその対象には含まれない．

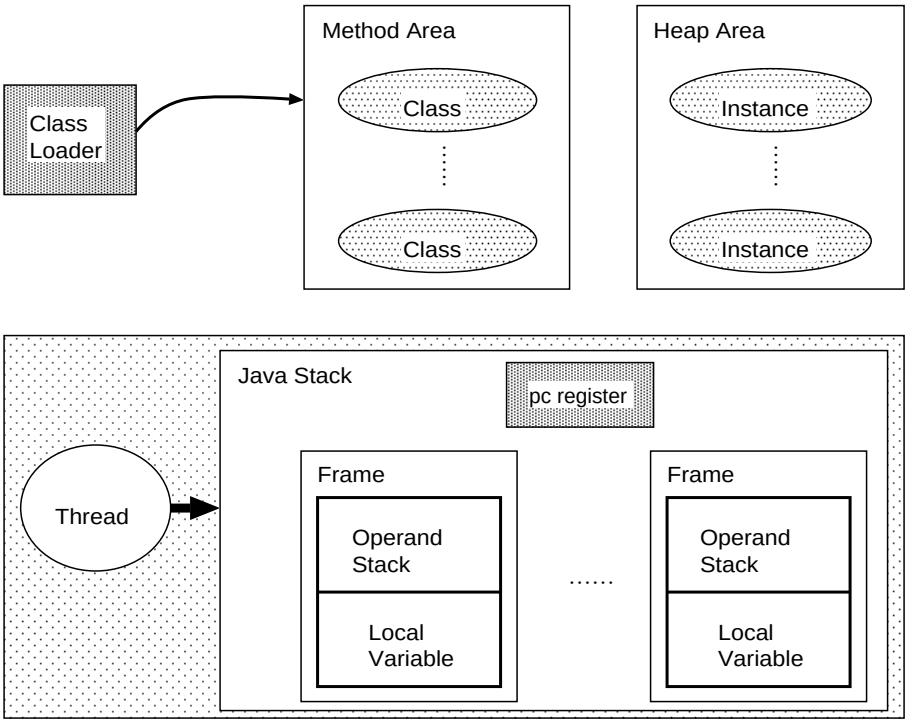

図 3.4: Java 仮想機械のデータ構造

3.3.2 ヒープ領域

クラスファイルは抽象化されたものであり，実際にその領域に書き込むことはできない．従って，それを実体化したものをインスタンスと呼ぶ．Java 仮想機械では，その実体化されたインスタンスの情報をヒープ領域に格納する．この領域はガベージコレクションの対象であり，不必要になったインスタンスは回収され再び利用される．

3.3.3 スレッド領域

スレッドのデータ構造は，複数のフレームから成る．この領域もガベージコレクションの領域に含まれる．また，スレッド領域をヒープ領域内に構成することも仕様書 [3] によって許可されている．

Java スタック

Java スタックは一つのスレッドを実行するために必要なデータの集合であり，以下に示すものから構成されている．

- フレーム

フレームとはメソッドが実行時に使用するメモリ領域であり，Java 言語のメソッドと１対１に対応する．実行中にメソッド呼び出しがおこなわれると，対応する新たなフレームが生成されて Java スタックに領域が確保される．生成されたフレームはメソッド終了時に Java スタックから除去される．フレーム内のデータは他のメソッドからアクセスすることは許されない．フレームは更に細かく，

- ローカル変数
- オペランドスタック

の二つのメモリ領域から構成され，メソッドはこれらの領域にアクセスしながら実行をおこなう．

ローカル変数は，変数の配列であり，その大きさはコンパイル時に決定されクラスファイルにその情報が与えられる．Java のローカル変数は型が固定されており，それ以外の型の値を代入することはできないが，Java 仮想機械のローカル変数は任意の型の値を保持できる．一般的には Java 仮想機械のローカル変数は Java 言語のローカル変数とは一致しない．

オペランドスタックは，メソッドが実行時に一時的に計算結果を記憶するために使用される．オペランドスタックはフレームの生成時点では空であり，オペランドスタックの各エントリは任意の型を格納することができる．また，オペランドスタックの最大使用量は Java 仮想機械のローカル変数と同様にコンパイル時に決定されクラスファイルにその情報が与えられる．

- pc レジスタ

pc レジスタは，プログラムカウンタを保持する．Java 仮想機械は実行中カレントメソッドと呼ばれる単一のメソッドのコードを実行している．pc レジスタはこの現在実行中の命令のアドレスが格納されている．メソッド呼び出し，もしくは終了時に次に実行すべきメソッドのプログラムカウンタに変更される．

3.4 バイトコード (ByteCode)

本節では，Java のクラスファイル中のメソッドエントリに示されるバイトコードの概要を示す．Java バイトコードは Java 仮想機械の命令セットである．Java コンパイラは Java プログラムの個々のメソッドをバイトコードの配列にコンパイルする．そして，それをクラスファイル中のメソッド部分に情報として埋め込むのである．

バイトコードの直接的な操作対象はフレーム内のオペランドスタック及びローカル変数，ヒープ領域である．バイトコードは Java 言語が持つオブジェクト指向をサポートするため一般的に使用されるマシンコードよりも抽象度の高い命令が存在する．例えば，invoke

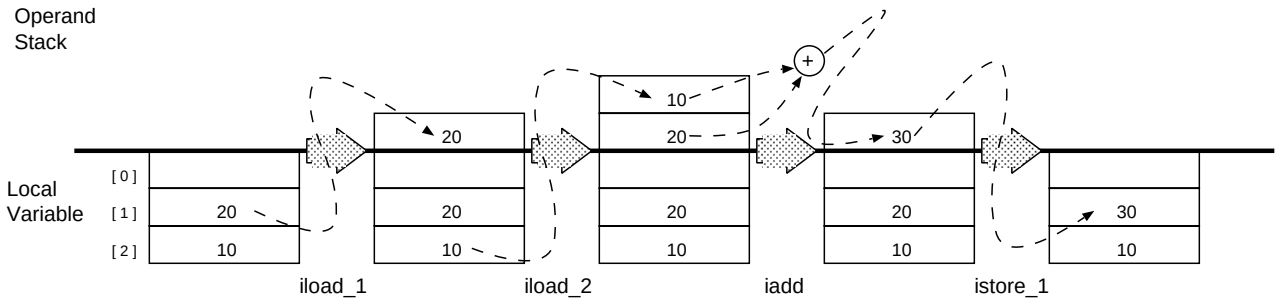

図 3.5: バイトコードの実行フロー例

系命令，new 系命令は，単独でメソッドの呼び出し及びインスタンスオブジェクトを生成する命令である．

次に，バイトコードの動作例を示す．以下のバイトコードがクラスファイル中に存在した場合の動作例を示す (図 3.5)．

```
//sample Bytecode
iload_1
iload_2
iadd
istore_1
```

1. iload_1

   インデクス 1 のローカル変数の値をオペランドスタックのトップに push する．

2. iload_2

   iload_1 命令と同様にローカル変数の 2 番目のインデックスの値をオペランドスタックのトップに push する．

3. iadd

   オペランドスタックのトップの値とその一つ下のオペランドスタックの値を pop し，足し合わせた結果をオペランドスタックのトップに push する．

4. istore_1

   オペランドスタックのトップの値をインデクス 1 のローカル変数に，格納する．

第4章　提案手法

本章では，第 3 章で示した Java 仮想機械のオペランドスタック及びローカル変数をハードウェア上でレジスタとして実現し高速化をおこなう手法を示す．最初に，ハードウェア上に実装する利点と問題点，その問題点に対する解決法を示す．その後，提案する Java 専用レジスタと Java 専用命令の概要を示し，最後に実装する CPU の基本仕様を示す．

4.1　ハードウェアレジスタ

第 2 章で述べたように，Java 仮想機械はスタックマシンアーキテクチャで，インタプリタ形式でバイトコードを実行する．従って，Java 仮想機械のメソッド呼び出し時にオペランドスタック及びローカル変数は動的に領域が確保される．そして，バイトコードの命令に応じて確保された領域 (この場合は，オペランドスタック及びローカル変数) から必要なデータを読み込み，演算をおこない結果を確保された領域に書き込む．つまり，あるバイトコード命令で演算をおこなった結果のデータが，次のバイトコード命令で必要なデータであっても，一度オペランドスタックに書き込む必要が生じる．

本研究では，Java 仮想機械のメソッド実行時に動的に確保されるオペランドスタック及びローカル変数をハードウェア上にレジスタとして実現し高速化をおこなう手法を提案する．この手法を用いることにより，以下の利点を得ることができる．

- ハードウェア上にオペランドスタックを実装するため，オペランドスタックを示すオペランドスタックポインタの更新を Java 仮想機械側で処理する必要がなくなる．これにより命令数の削減が期待される．
- オペランドスタック及びローカル変数をレジスタとして実装するので，オペランドスタック及びローカル変数の情報が必要な場合，従来手法のようにメモリアクセスの為のアドレス計算が必要がない．これにより命令の最適化がおこなわれる．

また，以下の点を考慮する必要がある．

- レジスタ数は有限である．従って，使用可能数に限界がありオーバーフローが起こる場合の処理が必要である．
- 本研究は組み込み機器での高速実行を目的としているため，追加すべきハードウェア量は極力少ない必要がある．

まず，レジスタ数が有限であるためオーバーフローが起こりうる問題については，Java 仮想機械に現在の使用レジスタ数を管理させる機能を追加することで回避する．第 3 章で示したように，クラスファイルの情報の中には各メソッドの実行に必要なオペランドスタック及びローカル変数の個数が示されている．これを利用して，新しくメソッドを起動する度に必要なレジスタ数を加え，メソッドが終了する度に使用したレジスタ数を減らす．そして，新しくメソッドを起動した時に必要となるレジスタ数と現在の使用レジスタ数の合計が，用意しているレジスタ数を越える場合には，そのメソッドは従来の Java 仮想機械と同様に必要な領域を動的に確保し実行をおこなうことで解決する．

次に，組み込み機器を研究の対象としているため，追加すべきハードウェア量を極力少なくする方法であるが，これは通常の CPU 資源を極力利用することで補う．具体的な設計については後述するが，本研究の手法を実現するために必要な機構は，Java 専用命令を検知，制御する機構と Java 専用レジスタ，そしていくつかのマルチプレクサを追加するのみである．これにより，メモリ制約の厳しい組み込み機器への実装も可能である．

4.2 Java 専用レジスタ (Java Exclusive Register FILE)

本節では，Java 専用レジスタ (Java Exclusive Register FILE) について述べる．これは，前節で提案したオペランドスタック及びローカル変数を格納するメソッド実行領域を実現するものである．(図 4.1) また，Java 仮想機械と同様の動作を実現するために以下のレジスタを追加する (図 4.2).

- OPSTP(Operand Stack Top Pointer)

  現在のメソッドのオペランドスタックのトップを示すレジスタ番号を保持するためのレジスタ．常に増減を繰り返す.

- LVP(Local Valiable Pointer)

  現在のメソッドが使用しているローカル変数のベースとなるレジスタ番号を保持するレジスタ．新たなメソッドの呼出し，または現在実行中のメソッドの終了時のみ変化する．

Java 専用レジスタは，最初のメソッド (main メソッド) を実行するときに，OPSTP レジスタの値は 0 番レジスタを指すようにセットされ，LVP レジスタは Java 専用レジスタの最後の番号 (本研究では 31 番レジスタとしている) がセットされる．新たなメソッドが呼び出されると現在の OPSTP レジスタと LVP レジスタの値を Java 仮想機械側に伝え，Java 仮想機械側から与えられる次のメソッド用の新しい値をセットする．メソッドが終了したときは，呼び出しと逆の手順で呼び出し側の使用していた OPSTP レジスタと LVP レジスタの値を復元する．

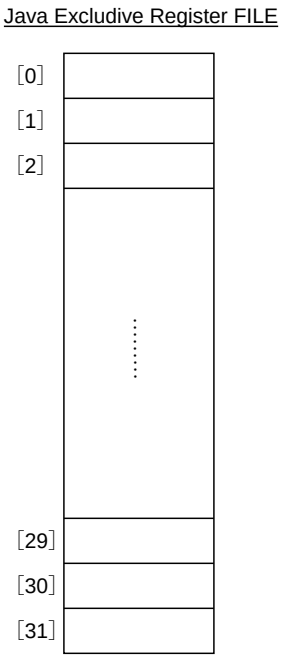

図 4.1: Java 専用レジスタ

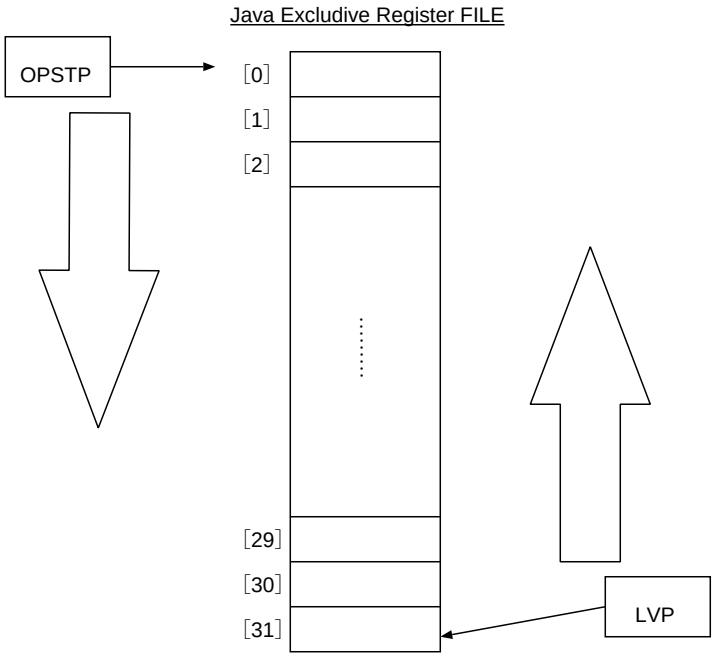

図 4.2: Java 専用レジスタファイルと OPSTP レジスタ,LVP レジスタの関係図

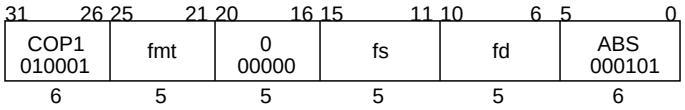

図 4.3: 実装依存命令基本フォーマット

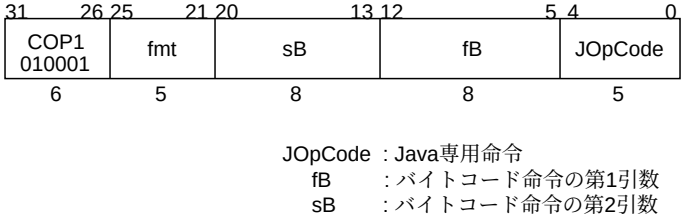

図 4.4: Java 専用命令基本フォーマット

4.3 提案手法の命令セット

本節では，Java 専用命令の命令セットを示す．Java 専用命令は MIPS[6] の実装依存命令の一部を使用して実装する．

図 4.3 のフォーマットでは，MIPS の実装依存命令を利用しているために上位 11 ビットを使用することはできない．従って，32 ビットから上位 11 ビットを除いた下位 21 ビットで Java 専用命令はデコードされる (図 4.4)．

Java 仮想機械の命令はバイトコード命令と呼ばれ最小 1 バイトの可変長命令であり，176 命令が定義されている．Java 仮想機械では全く同じ動作を行う命令であっても異なるバイトコードとして定義されているものがある．例えば，iload_1 命令と aload_1 命令は，どちらもオペランドスタックのトップに対して，ローカル変数 1 番の変数の値を push する命令である．これは，Java 仮想機械がソフトウェアであるために生じる定義である．本研究においては，動作的に同じバイトコード命令は一つの命令として実装することとした．

バイトコード命令の中には Java 仮想機械のプログラムカウンタの値を変更する命令が存在する．しかし，プログラムカウンタはハードウェア上に実現していないため不都合が生じる．例えば，オペランドスタックの値によってプログラムカウンタを書き換える if_icmplt 命令などである．オペランドスタックの値は Java 専用レジスタにあり，直接参照することはできない．従って，Java 専用レジスタの任意のレジスタの値を通常のレジスタに移動させる命令と，逆に任意の通常のレジスタの値を Java 専用レジスタに移動させる命令を実装することで，その問題を解決する．

実装した命令を表 4.1 に示す．なお，操作の項目に記述しているオペランドスタックとローカル変数は Java 専用レジスタの任意の場所を指す．

4.4 パイプライン化

本節では，提案手法である Java 専用レジスタ，Java 専用命令の選定機構を従来のパイプラインに追加しその整合性を示す．また，本章の最初に述べたが，通常のパイプラインの資源を有効に使用するため数個のマルチプレクサを追加した部分についても示す．図 4.5 は，提案手法を加えたパイプラインの全体図である．以下，提案手法の詳細な図を示し，その動作を示す.

4.4.1 専用命令解析ステージ

本節では，Java 専用命令の解析と Java 専用レジスタの動作の詳細を示す．図 4.5 で示した Java 専用ハードウェアの部分の詳細図を図 4.6 に示す.

命令解析ステージで追加した主な Java 専用ハードウェアを以下に示す.

- Java 専用レジスタ
- Java コントロールユニット
- OPSTP レジスタ更新加算器
- LVP レジスタ更新加算器
- 読み出しデータ選択用のマルチプレクサ

Java コントロールユニットはパイプラインのコントロールユニットが Java 専用命令を検知した時，Java コントロールユニットに信号を送ることで実行される．それ以外の CPU のネイティブな命令の実行の際には実行されない.

Java コントロールユニットが制御する信号線を以下に示す.

- ID ステージ
  - jselect_1
    ALU への第 1 オペランド入力を選択するための信号
  - jselect_2
    ALU への第 2 オペランド入力を選択するための信号
  - jDOP_sig
    Destination レジスタを決定するための信号
- EX ステージ

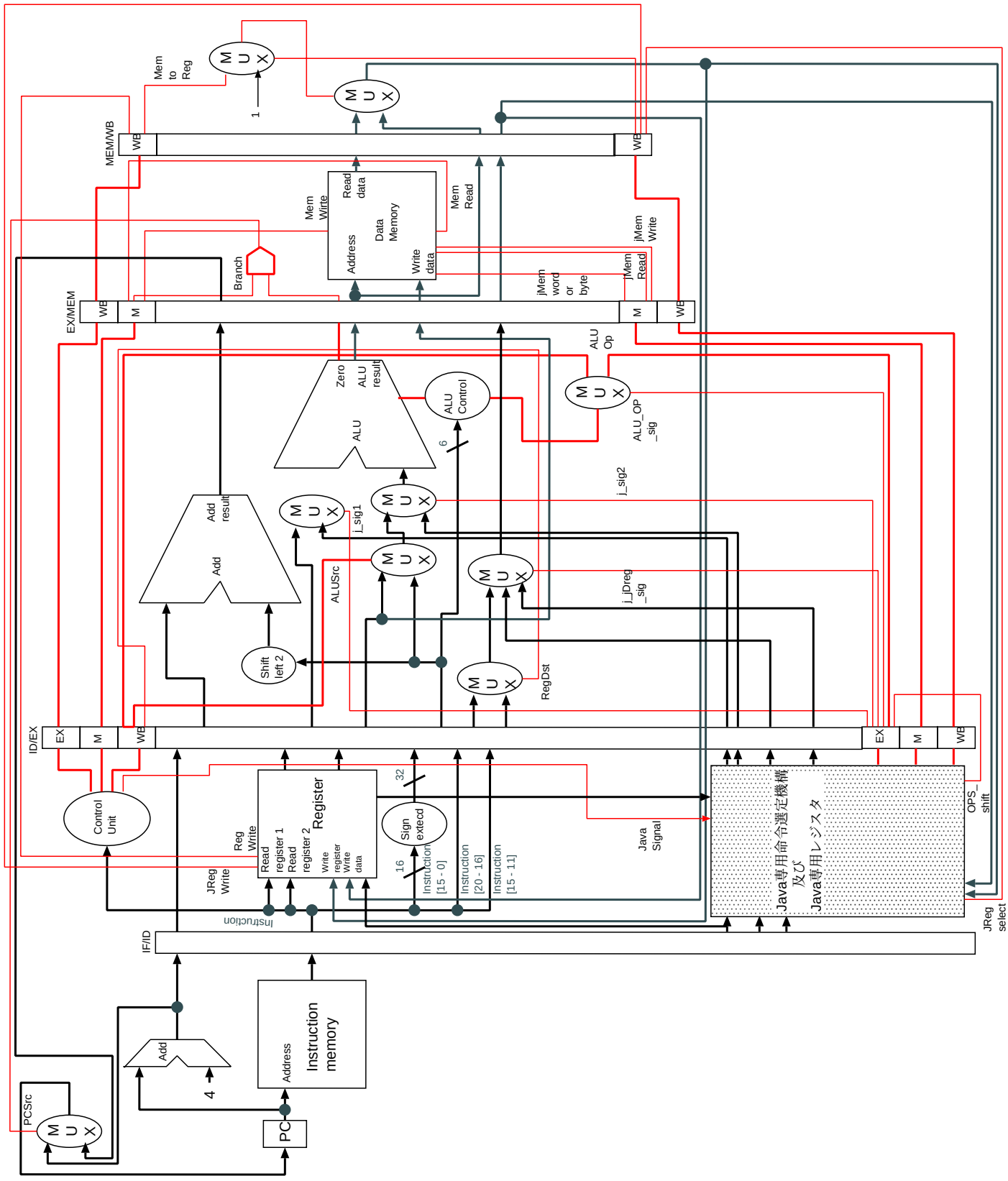

図 4.5: Java 専用ハードウェアを追加したパイプライン

- OPS_shift
  OPSTP の増減値を示す信号
- ALU_OP_sig
  ALU 制御信号を選択するための信号
- j_sig1
  ALU への第 1 オペランド入力を選択するための信号
- j_sig2
  ALU への第 2 オペランド入力を選択するための信号
- j_jDreg_sig
  Destination レジスタを決定するための信号
- j_mem
  ストアデータを選択するための信号

- MEM ステージ
  - jmem_write_sig
    メモリ書き込み許可を選択するための信号
  - jmem_read_sig
    メモリ読み込みを許可するための信号
  - jmem_word_or_byte
    word 単位か byte 単位かを選択するための信号

- WB ステージ
  - JRegWrite_sig
    通常のレジスタに書き込むかを選択する信号
  - JReg_select
    Java 専用レジスタに書き込むかを選択する信号
  - select_MemtoReg_sig
    レジスタからのデータかメモリから読み込んだデータかを選択する信号

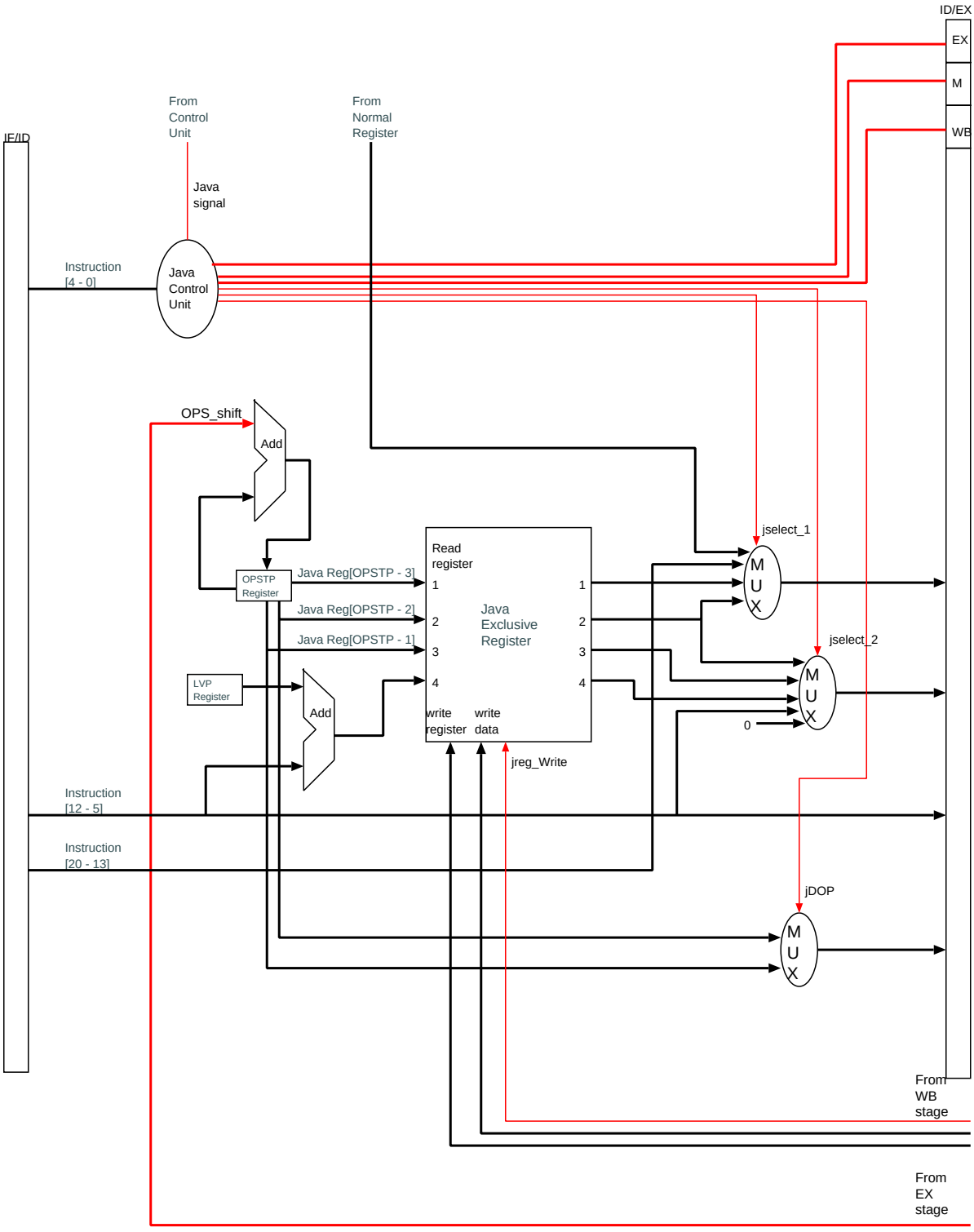

図 4.6: Java 専用ハードウェア詳細図

4.4.2 専用命令実行ステージ

Java 専用命令をデコードした場合，Java コントロールユニットの制御信号によりそれ以降のステージは全ての CPU 資源を利用する．従って，各ステージには Java 専用レジスタのデータと通常のレジスタのデータのどちらを使用するかを判断する必要があるためマルチプレクサを追加した．図 4.5，図 4.6 からわかるように追加したハードウェア資源は非常に小量である．

表 4.1: Java 専用命令

JOpCode	Java 専用命令	操作
00000	Jiconst	オペランドスタックのトップに fB の値を積む
00001	Jistore	命令の fB の数値をインデックスとする ローカル変数にオペランドスタックのトップの値を格納する
00010	Jiload	命令の fB の数値をインデックスとする ローカル変数の値をオペランドスタックのトップに積む
00011	Jiinc	命令の sB の数値をインデックスとする ローカル変数に fB の値を加算する
00100	Jiadd	オペランドスタックのトップの 2 値を加算し 結果をオペランドスタックのトップに積む
00101	Jislt	オペランドスタックのトップの 2 値を比較し 条件を満たせば 1，満たさなければ 0 を fB の値の通常のレジスタに格納する
00110	Jiasw	オペランドスタックのトップの値を, オペランドスタックのトップを除いた上位 2 値の値を 用いて表現される配列要素に格納する
00111	Jialw	オペランドスタックのトップの 2 値を用いて 表現する配列要素の値をオペランドスタックのトップに積む
01000	Jnmove	オペランドスタックのトップの値を fB の値の通常のレジスタに格納する
01001	Jimul	オペランドスタックのトップの 2 値を乗算し 結果をオペランドスタックのトップに積む
01010	Jidiv	オペランドスタックのトップの 2 値を除算し 結果をオペランドスタックのトップに積む
01011	Jmove	命令の sB の数値をインデックスとする 通常のレジスタの値を オペランドスタックのトップに格納する

第5章　評価

本章では, 通常の Java 仮想機械と通常の CPU との組と提案手法を採用した Java 仮想機械と CPU との組との実行サイクル数を比較する. 最初に提案手法により通常の Java 仮想機械のアセンブラ命令がどの程度最適化されたかの例を示す．次に評価に使用したプログラムを示し, 最後にその評価結果と追加したハードウェア量を示す.

5.1　提案手法による命令最適化

本節では，Java 仮想機械のアセンブラ命令が，第 4 章で示した Java 専用命令によりどの程度最適化されたかを示す.

図 5.1 は，Java 仮想機械のバイトコードを解析し実行する部分の概念図である．Java 仮想機械は,

1. pc レジスタが示すアドレスからバイトコードを読み出す
2. バイトコードに該当するラベルのアセンブラ命令を実行する
3. pc レジスタの値を更新する
4. バイトコードが終了するまで，1 に戻る

以上の操作を繰り返す．以下にオペランドスタックに値が一つ積まれた状態で実行されるバイトコード命令の istore_2 命令に対応するアセンブラ命令を示す.

```
//istore_2 Bytecode Instruction
$L286:
        lw      $2,608($fp)
        lw      $3,8($2)
        addi    $2,$3,8
        lw      $3,608($fp)
        lw      $4,4($3)
        addi    $3,$4,-4
        lw      $4,0($3)
```

```
        sw      $4,0($2)
        lw      $3,608($fp)
        lw      $2,608($fp)
        lw      $4,4($3)
        addi    $3,$4,-4
        lw      $3,4($2)
        b       $L206
```

通常の Java 仮想機械のアセンブラ命令は 14 命令である．次に Java 専用命令をアセンブラに埋め込んだものを示す.

```
//istore_2 Bytecode Instruction
$L286:
        jistore 2
        b       $L206
```

次に，オペランドスタックに 3 つの値が積まれた状態で実行されるバイトコード命令 iastore 命令に対応するアセンブラ命令を示す.

```
//iastore Bytecode Instruction
$L287:
        lw      $2,608($fp)
        lw      $3,4($2)
        addi    $2,$3,-4
        lw      $3,0($2)
        sw      $3,16($fp)
        lw      $2,608($fp)
        lw      $3,4($2)
        addi    $3,$4,-8
        lw      $3,0($2)
        sw      $3,20($fp)
        lw      $2,608($fp)
        lw      $3,4($2)
        addi    $2,$3,-12
        lw      $3,0($2)
        sw      $3,592($fp)
```

```
        lw      $2,592($fp)
        lw      $3,20($fp)
        move    $4,$3
        sll     $3,$4,2
        lw      $4,0($2)
        addu    $3,$4,$4
        lw      $3,16($fp)
        sw      $3,0($2)
        lw      $3,608($fp)
        lw      $2,608($fp)
        lw      $4,4($3)
        addi    $3,$4,-12
        sw      $3,4($2)
        b       $L206
```

これは，Java 専用命令を埋め込むことにより以下のように最適化される．

```
//iastore Bytecode Instruction
$L287:
        jiasw
        b       $L206
```

Java 専用命令を適用することによりアセンブラ命令 2 命令で istore_2 命令，iastore 命令を実行することができる．

他のバイトコードもほぼ同様に最適化をおこなうことが可能であり (5.1，5.2)，通常の Java 仮想機械のアセンブラ命令数と比較したとき，最大で 1350%，最小で- 70.97%の命令数の最適化となっている．このうち，最適化の結果がマイナスになっているものはバイトコードの命令が 2 バイト以上のものである．それらの命令は，後続するバイトコードの内容をローカル変数のインデクスとして利用している命令などであり，その内容を読み出すまで Java 専用命令を発行できない．例えば，iload 命令は，後続するバイトコードの内容をローカル変数のインデクスとして利用し任意のローカル変数にアクセスする命令でありインデクスが決定するまで Java 専用命令を発行できない．

5.2 評価方法

本節では，評価方法についての詳細を示す．

表 5.1: バイトコードに対応するアセンブラ命令数の比較 (1)

ByteCode 命令	アセンブラ命令数 (数)		実行効率 (%)
	Normal	Java Exclusive	
nop	1	1	0.00
iconst_m1	11	2	450.00
iconst_0	11	2	450.00
iconst_1	11	2	450.00
iconst_2	11	2	450.00
iconst_3	11	2	450.00
iconst_4	11	2	450.00
iconst_5	11	2	450.00
bipush	15	7	114.29
iload	9	31	-70.97
aload	9	31	-70.97
iload_0	14	2	600.00
iload_1	14	2	600.00
iload_2	14	2	600.00
iload_3	14	2	600.00
aload_0	14	2	600.00
aload_1	14	2	600.00
aload_2	14	2	600.00
aload_3	14	2	600.00
iaload	28	2	1300.00
istore	9	31	-70.97
astore	9	31	-70.97
istore_0	15	2	650.00
istore_1	15	2	650.00
istore_2	15	2	650.00
istore_3	15	2	650.00

表 5.2: バイトコードに対応するアセンブラ命令数の比較 (2)

ByteCode 命令	アセンブラ命令数 (数)		実行効率 (%)
	Normal	Java Exclusive	
astore_0	15	2	650.00
astore_1	15	2	650.00
astore_2	15	2	650.00
astore_3	15	2	650.00
iastore	29	2	1350.00
iadd	20	2	900.00
isub	20	2	900.00
imul	21	2	950.00
idiv	21	2	950.00
irem	21	2	950.00
ishl	20	2	900.00
ishr	20	2	900.00
iushr	20	2	900.00
iand	20	2	900.00
ior	20	2	900.00
ixor	20	2	900.00
iinc	31	50	-38.00
if_icmpl	43	33	30.30
goto	27	27	0.00
return	1	1	0.00
newarray	37	33	12.12

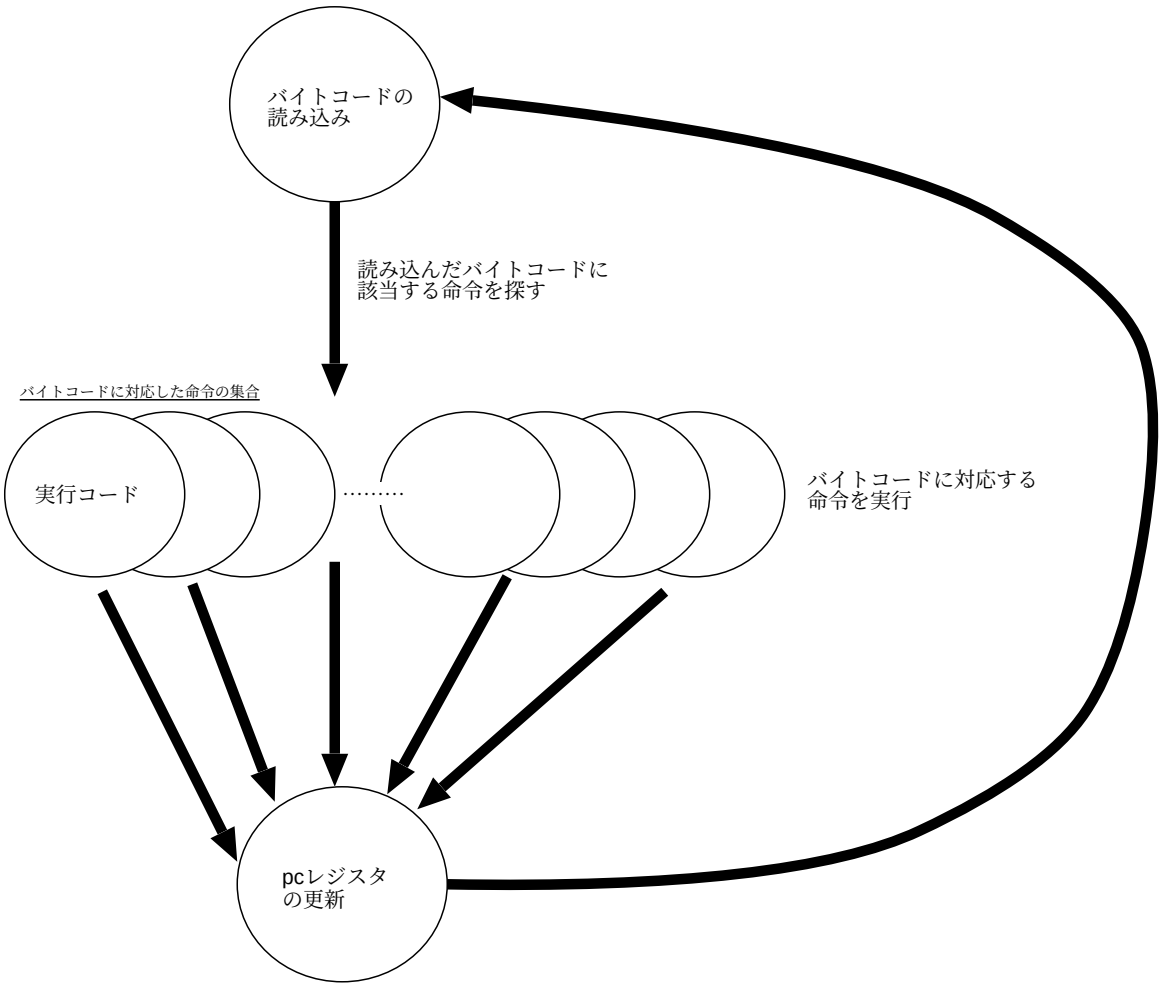

図 5.1: バイトコード実行概念図

入力として与える Java 仮想機械は，クラスファイルの情報はすでに読み込んでいるものと仮定し，Java ソースの main 関数に相当するメソッドの実行をおこなう段階からの総実行サイクル数を計測し比較するものとする．

実行サイクル数を計測するパイプラインシミュレータの性能を示す．

- C 言語で記述した Java 仮想機械をコンパイルして生成したアセンブリコード (MIPS 命令セット) を入力とする
- 分岐スロットは考慮しない
- Forwading Unit 実装
- Hazard Detection Unit 実装
- キャッシュ機構
  - 二次キャッシュは考慮しない
  - Write Back 方式
  - way 数 ： 4-way
  - キャッシュライン ： 32byte
  - キャッシュアクセス
    - キャッシュヒット時 … 1cycle
    - キャッシュミス時 … 10cycle

次に，Java 仮想機械に計測前に事前に読み込ませるクラスファイルを生成する Java ソースコードを表 5.3 に示す．

表 5.3: 評価用プログラムリスト

Java ソースコード
inner_product.java(Livermore Loops code)
banded_linear_euations.java(Livermore Loops code)
tri-diagonal_elimination.java(Livermore Loops code)
first_sum.java(Livermore Loops code)
first_difference.java(Livermore Loops code)

表 5.4: 総実行サイクル数測定結果

評価プログラム	実行総サイクル数 (cycle)		実行効率 (%)
	Normal	Java Exclusive	
inner_product.java(Livermore Loops code)	115121	86204	33.54
banded_linear_euations.java(Livermore Loops code)	112768	84483	33.48
tri-diagonal_elimination.java(Livermore Loops code)	161865	112889	43.38
first_sum.java(Livermore Loops code)	127909	93241	37.18
first_difference.java(Livermore Loops code)	125829	91371	37.71

5.3 評価結果

本節では，5.2 節で示した条件の下でおこなった評価を示す．各評価プログラムで計測した実行サイクル数を表 5.4 に示す．

実行総サイクル数は，従来方式 (Normal)，提案手法 (Java Exclusive) の意味を示し，実行効率は従来手法と比較した提案手法の性能向上率を示す．

総実行サイクル数は，従来方式よりも最大で 43.38%，最小で 37.06%の向上率を示した．平均で 36.69%の向上が見られる．バイトコード 1 命令辺りのアセンブラ命令数の向上率を考えると少ない向上率であるが，Java 仮想機械の評価部分はメソッドの起動準備や pc レジスタの更新をおこなうことも含んでいる．従って，バイトコードの最適化は局所的なものであり，表のような結果になった．また，評価用プログラムとして使用したプログラムは，配列を用いるものが多い．本研究では，Java 仮想機械のオペランドスタック及びローカル変数をレジスタ化することにより高速化をおこなう．しかし，それ以外の動的に領域を確保する部分は従来方式と同様である．配列やインスタンスは，その領域をバイトコード命令によって動的に確保するものである．従って，その領域を確保する処理を高速化するとこはできず，向上率の低下に結び付く原因となる．

各評価プログラムで計測した実行アセンブラ命令数及び Java 専用命令数を表 5.5 に示す．アセンブラ命令数は，従来方式 (Normal)，提案方式 (Java Exclusive) の実行アセンブラ命令数である．また，提案方式は通常アセンブラ命令 (MIPS)，Java 専用命令 (Java)，通常アセンブラ命令と Java 専用命令の合計 (Total) となっている．

評価プログラムで計測した総命令中の Java 専用命令の内訳を表 5.6，表 5.7 に示す．表の数字は命令の実行された回数を示す．

表 5.5: 実行命令数の内訳

評価プログラム	アセンブラ命令数			
	Normal	Java Exclusive		
		MIPS	Java	Total
inner_product.java	56728	25423	1200	26623
banded_linear_euations.java	34623	23962	1192	25154
tri-diagonal_elimination.java	72459	32164	1980	34144
first_sum.java	51386	27310	1440	28750
first_difference.java	50301	26830	1380	28210

表 5.6: Java 専用命令の内訳 (1)

プログラム名	Jiconst	Jistore	Jiload	Jiinc	Jiadd	Jislt
inner_product.java	21	124	521	110	0	121
banded_linear_euations.java	102	175	481	50	100	101
tri-diagonal_elimination.java	111	14	921	110	200	121
first_sum.java	31	14	741	110	100	121
first_difference.java	11	14	721	110	0	121

5.4 追加ハードウェア量評価

本節では，提案手法により追加したハードウェア量を評価する．
パイプライン全体でハードウェアに追加した資源を以下に示す．

- Java 専用レジスタ
- Java コントロールユニット
- 4 ビット加算器 … 2 個
- マルチプレクサ … 8 個
- パイプラインレジスタの追加 … 4 個

これらの追加は，従来の CPU のハードウェア量と比較したとき非常に小量であり，追加に問題はない．

表 5.7: Java 専用命令の内訳 (2)

プログラム名	Jiastore	Jiaload	Jnmove	Jisub	Jimul
inner_product.java	0	200	3	0	100
banded_linear_euations.java	20	120	3	0	40
tri-diagonal_elimination.java	100	300	3	0	100
first_sum.java	110	210	3	0	0
first_difference.java	100	200	3	100	0

第6章　関連研究

本章では，Java 実行の高速化手法に関する二つの研究を示す．

JIT コンパイラ

JIT コンパイラ [8] は，Java 実行時に入力のバイトコードを部分的にネイティブコードに再コンパイルして高速化をおこなう手法である．

JIT コンパイラの高速化手法を以下に示す．

- オペランドスタック及びローカル変数のレジスタ割り付け

  動的に確保される Java 仮想機械のオペランドスタック及びローカル変数を再コンパイル時にレジスタに割り付ける．

- peephole 最適化

  上記のレジスタ割り付けをおこなった後，レジスタアクセスをおこなうネイティブコードの冗長な命令を最適化する．

- 動的リンク

  バイトコード命令は，クラスやインスタンスの変数，メソッド情報などを使用する場合はクラスファイルのコンスタントプールへのシンボル情報がバイトコード内に与えられている．このシンボルの検索処理がおこなわれた時，一度目は通常の検索処理をするが，その際，検索処理をおこなったバイトコードを quick 命令に書き換え，二度目以降の検索処理のオーバヘッドを削減している．

JIT コンパイラによる高速化手法は，再コンパイル時に必要な時間により実行性能が従来手法よりも遅くなる場合がある．また，再コンパイルにより生成されるネイティブコードが必要とするメモリ量は膨大で，メモリ制約の厳しい組み込み機器には重大な問題となる．本研究では，再コンパイル手法を用いることなく実行領域のレジスタ割り付けをおこなうことが可能である．

Javaチップ

Javaチップ [7] とは，Javaのバイトコードを機械語とするスタックマシンのプロセッサである．バイトコードを高速に実行するだけでなく，従来からのCプログラムも実行することが可能である．

Javaチップの高速化手法を以下に示す．

- 命令フォールディング機構

  ある一定の条件を満たすバイトコード命令列のとき，それら複数の命令をまとめて一つの命令で実行することができる．

Javaチップによる高速化手法は，スタックマシンのプロセッサに対しての手法である．Cプログラムの実行はJavaチップ用のCコンパイラを提供することで実現している．この手法では，Java実行は高速化されるが，ネイティブコードの実行には問題が残る．本研究では，従来のプロセッサにJava用のレジスタを実装することで，従来からのネイティブコードの実行に影響を与えることなく高速化を実現することが可能である．

第7章　最後に

7.1　まとめ

本論では，Java 実行における高速化手法として，Java 仮想機械が動的に確保するオペランドスタック及びローカル変数をレジスタとしてハードウェア上に実現し，バイトコードに対応するアセンブラ命令を最適化し高速化する手法を提案した．

提案した Java 専用命令及び Java 専用レジスタ機構はシミュレータ実行において，従来の Java 実行よりも高速に実行させることが可能である．特にオペランドスタック及びローカル変数をアクセスするバイトコード命令が多いほどその効果は大きいものとなる．

本提案手法によるハードウェアの追加量は十分に少量であり，メモリ制約の厳しい組み込み機器への応用は可能である．また，本提案では Java 実行の高速化について述べたが，Java 専用命令及び Java 専用レジスタは非常に汎用性に優れており他の用途への利用も可能である．

7.2　今後の課題

今後の課題として以下の点を挙げる．

1. シミュレーションの高精度化
2. Java 仮想機械の高精度化
3. Java 専用レジスタの利用法
4. 他の高速化手法との連係

1 では，評価をする際，クラスファイルを読み込む所要サイクル数を考慮していなかった．今後はクラスファイルを読み込むシミュレータに改良したい．

2 では，今回実装した Java 仮想機械は，マルチスレッド機能などいくつかの機能をを実装していない．今後は，Java 仮想機械の全ての機能を実装した後評価をおこないたい．

3 では，本研究で提案した Java 専用レジスタ及び Java 専用命令は Java 実行にのみ特化したものではない．従って，他の利用方法も考案したい．

4 では，本研究の提案は，関連研究で紹介したような他のソフトウェアアプローチによる高速化手法と連係して動作することが可能である．今後は，連係して動作する手法について提案したい．

参考文献

[1] T.Wilkinson, KAFFE v0.5.5-A JIT and interpreting virtual machine to run Java code, October 1996.

[2] M.O'Connor, M.Tremblay, picoJava-I : The Java virtual machine hardware, IEEE Micro,pp.45-53, March-April 1997.

[3] T.Lindholm, F.Yellin, The Java Virtual Machine Specification(2nd Ed), Addison-Wesley, 1999.

[4] Java Hotspot performance engine, http:java.sun.com/products/hotspot/, 1999.

[5] J.Meyer, T.Downing, JAVA Virtual Machine The JAVA SERIES(tm), 1997

[6] MIPS Technologies, Inc, MIPS64(TM) Architecture For Programmers Volume 2: The MIPS64(TM) Instruction Set, March 12, 2001

[7] 青木 孝, ハードウェアソリューションによる Java Virtual Machine, 電子情報通信学会研究報告，CPSY-2000-95 March，2001

[8] 志村 浩也，木村 康則, Java JIT コンパイラの試作, 情報処理学会研究報告，96-ARC-120，pp.37-42 November 31, 1996

謝辞

本研究をおこなうにあたり，日頃から熱心にかつ的確な御指導を頂きました田中清史 助教授に心より感謝致します.

貴重な御助言をしていただきました日比野 靖 教授，井口 寧 助教授に深く感謝致します.

同じ研究室で苦楽を共にした 大崎 哲弥 氏，越前 智史 氏，馬場 久 氏，山本 達也 氏，荻野 雅 氏 に深く感謝します.
年齢的には年上で後輩の今井 俊晴 氏には，色々と愚痴を聞いてもらいました.
また，同じ研究室の後輩であり，色々とお世話になった高橋 礼彦 氏 ，上田 憲吾 氏，島田 信行 氏，南里 洋亮 氏，森田 充 氏もありがとう.

最後に，私が大学院での生活を送るにあたって心の支えとなってくれた大島の祖父と祖母と両親に心より感謝致します.