Java Virtual Machine 命令互換プロセッサの高速化に関する研究

東海大学 工学部 通信工学科

提出日 平成 9 年 3 月 25 日

指導教員 清水 尚彦 講師

研究者 30et2101 青柳 真佐樹

Abstract

Sun Microsystems 社が開発した JVM(Java Virtual Machine) はどのようなプラットフォームにおいても同様に動作することが可能であるという特徴を持つ。JVM の性能向上のためには多くのアプローチが存在し、そのひとつに JVM の命令実行をハードウェアで備える Javachip というものが提案されている。

しかし Javachip は既存のプロセッサとは大きく異なり、複雑な制御を要求する。本稿では高速に動作する JVM 命令互換プロセッサの開発とともに、ハードウェアでは実装できない命令に対するソフトウェアエミュレーションのための検討を行った。ハードウェアの開発に際しては、ハードウェア記述言語である SFL を使用している。

目 次

1 はじめに **1**

2 Java Virtual Machine のアーキテクチャ **3**

2.1 データ型 3
2.2 Java Virtual Machine の命令 4
2.3 レジスタ 6
2.4 ローカル変数 6
2.5 オペランド スタック 7
2.6 実行環境 8
2.6.1 動的リンク 8
2.6.2 ノーマルメソッド リターン 8
2.6.3 例外とエラーの伝搬 8
2.6.4 追加情報 9
2.7 メソッド エリア 9
2.8 制限 9

3 Java Virtual Machine 命令セット **11**

3.1 Java Virtual Machine 互換命令 11
3.1.1 ローカル変数をスタックにロード する命令 11
3.1.2 ローカル変数にスタックの値をストアする命令 12
3.1.3 定数プッシュ命令 13
3.1.4 ローカル変数のインデックスを拡張する命令 13
3.1.5 配列管理命令 14
3.1.6 スタック命令 15
3.1.7 数値演算命令 16
3.1.8 分岐命令 17
3.1.9 リターン命令 18
3.1.10 比較命令 18
3.1.11 論理演算命令 18
3.1.12 型変換命令 19
3.1.13 スイッチ命令 19

3.1.14 フィールド操作命令 19
3.1.15 メソッド呼び出し命令 20
3.1.16 例外処理命令 20
3.1.17 その他のオブジェクト操作命令 20
3.1.18 スレッド制御命令 20
3.2 Java Virtual Machine 非互換命令 21

4 TRAJA Version 2 のアーキテクチャ 23
4.1 TRAJA Version 2 について 23
4.2 パイプライン 24
4.2.1 ステージの構成 24
4.2.2 IF ステージの動作 26
4.2.3 ID ステージの動作 27
4.2.4 OC ステージの動作 29
4.2.5 EX ステージの動作 31
4.3 スタックキャッシュ 33
4.3.1 スタックキャッシュの概要 33
4.3.2 スタックキャッシュのアーキテクチャ 34
4.4 命令セット 41
4.4.1 ハードウェアで実装した命令セット 41
4.4.2 実行クロック 41
4.4.3 ソフトウェアエミュレーションのための検討 41

5 プログラムについて 43

6 結論 44

7 謝辞 45

Chapter 1

はじめに

Java とは Sun Microsystems 社が開発したオブジェクト指向言語である。この言語で書かれたコードはバイトコードと呼ばれる中間言語に翻訳される。バイトコードはどのようなプラットフォームにおいても、各々の OS 及びマイクロプロセッサのために開発された Java Virtual Machine と呼ばれる仮想計算機上で同様に実行することができる。JVM の実装方法としては、インタプリタ、ジャスト・イン・タイム・コンパイラ、ハードウェアなどがあげられる。

ただし通常のバイトコードを翻訳しながら実行するインタプリタ型の JVM では、その処理速度が問題になる。この解決策としてはいくつかのアプローチがある。たとえば、ジャスト・イン・タイム・コンパイラは実行環境のマイクロプロセッサと OS に合ったコードに変換してから実行する。そして、バイトコードをハードウェアで直接実行するプロセッサである Javachip というものも提案されている。Figure 1.1 は JVM 実装のための 3 つの手段を示している。

我々はパルテノン研究会主催の第 3 回 ASIC デザインコンテストにおいて Javachip の実現方法について研究し、JVM 互換命令 62 種類及び独自命令 7 種類をハードウェアで実装した「Java Virtual Machine 互換命令を持つプロセッサ」(TRAJA[1] Version 1) として発表した。しかし、ハードウェアの性能及び完成度については改善すべき点が多くあった。

今回の研究では、より幅広い JVM 互換命令のサポート及びパイプライン、スタックキャッシュの実装などによって高性能な Javachip の実現をめざし、TRAJA Version 2 として新たに開発を行った。

しかし、JVM は既存のプロセッサと比較して多くの点で特殊な処理を要求する。したがって前回の研究を踏襲し、ハードウェアで対応しきれない場合にはソフトウェアによるエミュレーションを以て対応する。前回不明瞭だったエミュレーションの実現方法についても研究を行っている。

[1] Tokai Research Approach for JVM Architecture の略

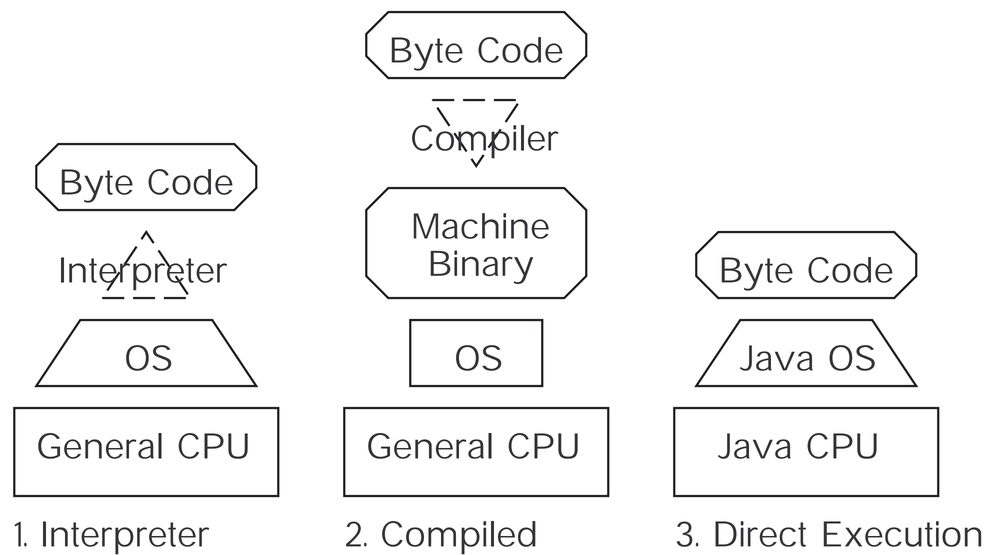


Figure 1.1: JVM 実装のための 3 つの手段

Chapter 2

Java Virtual Machine のアーキテクチャ

以下に JVM の概要を説明する。これは今回設計した Javachip と特に関連の深い事項を、Sun Microelectronicsが提供している「The Java Virtual Machine Specification」から抜粋したものである。直訳では理解しづらい部分については修正を加えている。また説明不足な点については多少の説明を加えている。

2.1 データ型

JVM のデータ型は以下に示される Java 言語の基本データ型を含んでいる。

データ型	意味
byte	1 バイト符号付き整数 (2 の補数による表現)
short	2 バイト符号付き整数 (2 の補数による表現)
int	4 バイト符号付き整数 (2 の補数による表現)
long	8 バイト符号付き整数 (2 の補数による表現)
float	4 バイト単精度浮動小数点数 (IEEE754 フォーマット)
double	8 バイト倍精度浮動小数点数 (IEEE754 フォーマット)
char	2 バイト符号なしユニコード (Unicode) 文字

ほとんどの型はコンパイル時にチェックが行われる。Java の実行のためには、上に示したプリミティブな型のデータはハードウェアに依存しない。そのかわりに、プリミティブな値を操作するバイトコードはオペランドの型に依存する。例えば、iadd、ladd、fadd、dadd などの命令はそれぞれ int、long、float、double の型を持つ 2 つの値を加算する。

JVM は boolean 型に対して特別な命令を持たない。そのかわりに、整数の戻り値を持つ整数命令が boolaen 型の操作に使用される。boolean の配列に対してはバイト配列が使用される。また、JVM は浮動小数点数を単精度、倍精度ともに IEEE754 フォーマットで表

現する。

Java Virtual Machine のデータ型は他には次のようなものがある。

object	Java のオブジェクトに対する 4 バイトのリファレンス
returnAddress	jsr,ret,jsr_w,ret_w 命令で使用される (4 バイト)

バイトコードはデータ型の規則を厳守し、その規則を破るようなプログラムは実行が拒否される。

2.2 Java Virtual Machine の命令

Java の命令セットにおける命令は実行されるための操作を特徴づける 1 バイトの opcode(オペコード) と、操作に使われるパラメータやデータを供給する 0 個またはそれ以上のオペランドで構成される。多くの命令はオペランドを持たず、1 つの opcode のみで構成されている。

Figure 2.1 は、バイトコードがメモリ上にどのように格納されているかを図で表したものである。1 ワードは 4 バイトであり、1 つの命令が複数のワードにまたがって格納されることもある。

仮想マシンの実行における内部ループは実質的に以下のようになる。

```
do {
     opcode をフェッチする
     opcode に値にしたがって実行
} while (他にすることがあるならばループを繰り返す);
```

付属するオペランドの数とサイズは opcode によって決定される。もし、付属するオペランドのサイズが 1 バイト以上ならば、big-endian order(高い順序のバイトは最初にくる) で格納される。例として、16 ビットのパラメータは 2 バイトとして格納され、その値は以下のようになる。

```
first_byte * 256 +second_byte
```

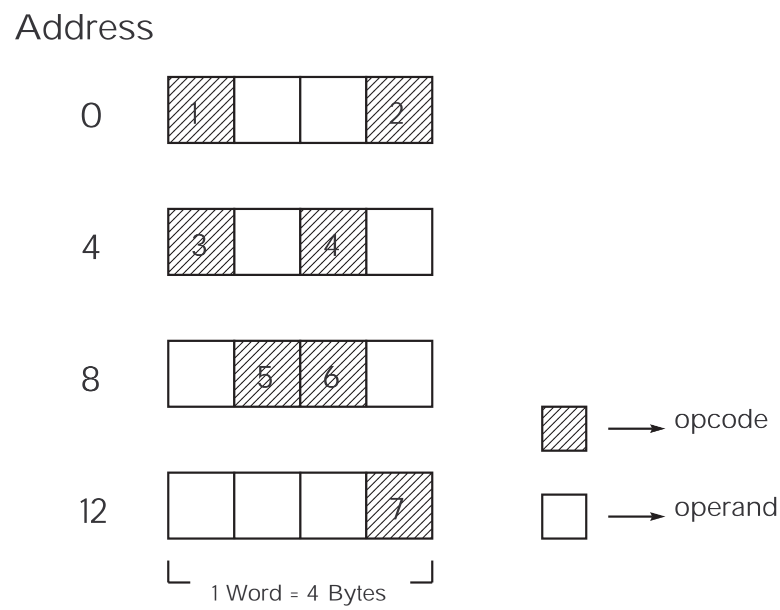


Figure 2.1: バイトコード格納の様子

バイトコードの命令の流れは単純にバイト順である。(tableswitch や lookupswitch を除く。これらは強制的に命令の内部を 4 バイトの境界でならべる。)

これらの決まりによって、コンパイルされた Java プログラムのための仮想マシンコードをコンパクトにし、実行におけるコストは小さくなる。

2.3 レジスタ

JVM はどの時点においてもシングルメソッドのコードを実行している。そして、pc レジスタは次に実行されるべきバイトコードのアドレスを指し示している。
どのメソッドも以下のレジスタを所有するためのメモリ空間が割り当てられている。

- **vars** レジスタによって参照されるローカル変数のセット
- **optop** レジスタによって参照されるオペランドスタック
- **frame** レジスタによって参照される実行環境構造

この空間は一度に全て割り当てられることができる。なぜなら、コンパイル時にはローカル変数とオペランドスタックのサイズはコンパイル時に知ることができ、実行環境構造のサイズはインタプリタがすでに知っているからである。

2.4 ローカル変数

全てのメソッドは固定サイズのローカル変数のセットを使用する。これは vars レジスタからのワードオフセットとしてアドレッシングされる。ローカル変数は全て 32 ビット幅である。
long 型と double 型は 2 つのローカル変数をとると考えられる。このとき、そのアドレスは最初のローカル変数のインデックスである。例えば、double 型でインデックスが n のローカル変数は実際には、n と n+1 のインデックスを持つ領域を占有する。JVM 仕様書では、64 ビットのローカル変数の値が 64 ビットで割り当てられる必要はないとされている。実装者が long 型及び double 型を適切な方法で 2 つのワードに分配して良い。
JVM 命令セットでは、ローカル変数のオペランドスタック上へのロード及びオペランドスタックからローカル変数へのストアを行う命令などが提供されている。

2.5 オペランド スタック

ほとんどの命令はオペランドをオペランド スタックから取り出し、それらを操作し、そしてスタックに結果を返す。

オペランド スタックの領域は 32 ビット幅で分割される。これは操作のためのパラメータを供給したり、調査の結果を保存したりすることのほかにも、パラメータをメソッドに渡したり、メソッドの結果を受け取ることにも使用される。

例えば、iadd 命令は 2 つの整数を加算する命令である。加算される整数はオペランド スタック上の上から 2 つであり、それらはその前の命令によって push されている。オペランド スタックから 2 つの整数が pop され、加算され、その結果はオペランド スタックに pop される。

どのプリミティブなデータ型もその型のオペランドでどのように操作するかを知る特化した命令を持つ。どのオペランドもスタック上に 1 つの場所を要求する (ただし、long、double 型を除く。これらは 2 つに場所を要求する)。

オペランドはそれらのデータ型に適合する命令によって操作されなければいけない。例えば、2 つの int 型を push して、それを long 型として扱うことは許されない。Sun のインプリメンテーションでは、この制限はバイトコード ベリファイヤにより強制される。しかしながら、少数のオペレーション (swap や dup) は型に関係なく、未加工の値としてランタイムデータエリア上で操作を行う。

下の JVM の命令の記述においては、オペランド スタック上の命令実行の結果はテキストで表現している。ここでは、左から右に積まれていくスタックを用いる。スタックのアドレスは右にのびていくにしたがって大きくなる。

```
Stack: ...,value1,value2=>...,value3
```

これはスタックの最上位にある value2 とそのすぐ下にある value1 を得ることによって始まるオペレーションを示している。命令実行の結果として、value1 と value2 は pop されて、命令によって計算された値である value3 に置き換わる。この命令は、スタックの他の部分に影響しない。

long 型や double 型の場合は、以下に示すようにオペランド スタック上に 2 ワードをとる。

```
Stack: ...,value-word1,value-word2
```

JVM 仕様書では、64 ビットの値から 2 つの語がどのように選択されるかについては言及していない。独自の実装において矛盾がなければよいのである。

2.6 実行環境

実行環境に含まれる情報は動的リンクを行うために使用されるノーマルメソッドリターンと例外の伝搬(Exception Propagation)である。

2.6.1 動的リンク

実行環境はカレントメソッドやカレントクラスに対するインタプリタのシンボルテーブルへのリファレンスを含んでいる。これはメソッドコードの動的リンクを支援するものである。メソッドに対するクラスファイルコードは呼び出されるべきメソッドとシンボルでアクセスされる変数を参照する。動的リンクはこれらのシンボリックメソッドコールを実際のメソッドコールに変換する。必要ならば、as-yet-undefined のシンボルを決定するためにクラスをロードする。そして変数のアクセスをこれらの変数の実行時における場所に基づいて、記憶構造の中の適切なオフセットへ変換する。

実行時のメソッドや変数のバインディングは、他のクラスにおいてメソッドに壊れたコードの使用を回避させる。

2.6.2 ノーマルメソッドリターン

通常、カレントメソッドの実行が完了すると呼び出し元のメソッドに値を返す。これはメソッドが戻り値の型に適切なリターン命令を実行したときに発生する。

この場合、実行環境は呼び出し側のレジスタをリストアするために使用される。このとき同時に、呼び出し側のプログラムカウンタをメソッドコール命令をスキップするようにインクリメントしておく。そして、呼び出し側の実行環境で実行が続く。

2.6.3 例外とエラーの伝搬

throwable のサブクラスであり、Java ではエラーまたは例外として知られている例外状況はプログラム中において以下のことが原因で発生する可能性がある。

- 動的リンクの失敗(クラスファイルが見つからないなど)
- ランタイムエラー(ヌルポインタによる参照など)
- 非同期イベント(他のスレッドから Thread.stop によって throw されるなど)
- throw ステートメントを使用しているプログラム

例外が発生するのは以下の時である。

- カレントメソッドに関連している catch clauses のリストが実行される。catch 条項はそれがアクティブとなる命令の範囲を記述し、処理するべき例外のタイプを記述し、そしてそれを扱うためにそのコードのアドレスを持つ。
- もし例外を発生させた命令が適切な範囲にあるならば、例外は catch 条項に一致する。そして、例外のタイプは catch 条項が処理する例外のタイプのサブタイプである。もし一致する条項が見つかった場合は、システムは指定されたハンドラに分岐する。そのハンドラが見つけられない場合は、カレントメソッドの全てのネストされた catch 条項を使いきるまでプロセスが繰り返される。
- リスト中の catch 条項の順序は重要である。JVM は最初に一致する catch 条項で実行を続けていく。なぜなら、java のコードは構造化され、一つのメソッドのために全ての例外ハンドラを一つのメソッドにソートすることが可能である。そしてそのリストはどの可能なプログラムカウンタの値に対しても、プログラムカウンタの値において発生する例外に対して線形順序で正規の例外ハンドラを探すために検索されることが可能である。
- もし一致する catch 条項がない場合には、カレントメソッドはその結果として uncought という例外があることを伝える。このメソッドを呼び出したメソッドの実行状態が実行環境からリストアされ、そして呼び出し側で例外が発生したように例外の伝搬が続く。

2.6.4 追加情報

実行環境は"Additional Implementation-specific Information"によってデバッギング情報として拡張される。

2.7 メソッドエリア

メソッドエリアは従来の言語におけるコンパイル済みのコードのストアや UNIX のプロセスにおけるテキストセグメントに類似している。これはメソッドのコード (コンパイルされた Java コード) やシンボルテーブルをストアする。将来的には計画されているのであるが、現在の Java の実装においては、メソッドコードはガーベジコレクテッドヒープの一部ではない。

2.8 制限

JVM 内部の制限として以下のことがあげられる。

- クラスごとの定数プールは最大で 65535 エントリである。
- メソッドごとのコードの総和は、例外テーブル、ラインナンバーテーブル、ローカル変数テーブルのコード中のインデックスのサイズによって 65535 バイトに制限される。
- メソッドコールにおける変数のワード数は 255 に制限される。

Chapter 3

Java Virtual Machine 命令セット

3.1 Java Virtual Machine 互換命令

以下に示す命令は Java Virtual Machine 互換命令である。現段階でハードウェアでの実装が可能な命令とエミュレーションを必要とする命令とに分類した。

3.1.1 ローカル変数をスタックにロードする命令

名称	命令長	機能	ハードウェアでの実装
iload	2	int 型のローカル変数をロードする	実装済み
iload_n	1	int 型のローカル変数をインデックスを n としてロードする	実装済み
lload	2	long 型のローカル変数をロードする	実装済み
lload_n	1	long 型のローカル変数をインデックスを n としてロードする	実装済み
fload	2	float 型のローカル変数をロードする	実装済み
fload_n	1	float 型のローカル変数をインデックスを n としてロードする	実装済み
dload	2	double 型のローカル変数をロードする	実装済み
dload_n	1	double 型のローカル変数をインデックスを n としてロードする	実装済み
aload	2	オブジェクトリファレンス型のローカル変数をロードする	実装済み
aload_n	1	オブジェクトリファレンス型のローカル変数をインデックスを n としてロードする	実装済み

3.1.2 ローカル変数にスタックの値をストアする命令

名称	命令長	機能	ハードウェアでの実装
istore	2	int 型のローカル変数をストアする	実装済み
istore_n	1	int 型のローカル変数をインデックスを n としてストアする	実装済み
lstore	2	long 型のローカル変数をストアする	実装済み
lstore_n	1	long 型のローカル変数をインデックスを n としてストアする	実装済み
fstore	2	float 型のローカル変数をストアする	実装済み
fstore_n	1	float 型のローカル変数をインデックスを n としてストアする	実装済み
dstore	2	double 型のローカル変数をストアする	実装済み
dstore_n	1	double 型のローカル変数をインデックスを n としてストアする	実装済み
astore	2	オブジェクトリファレンス型のローカル変数をストアする	実装済み
astore_n	1	オブジェクトリファレンス型のローカル変数をインデックスを n としてストアする	実装済み
iinc	3	ローカル変数に定数を加算する	実装済み

3.1.3 定数プッシュ命令

名称	命令長	機能	ハードウェアでの実装
bipush	2	1 バイトの符号付整数をスタックにプッシュする	実装済み
sipush	3	2 バイトの符号付整数をスタックにプッシュする	実装済み
ldc1	2	クラス定数プールへのインデックス (8bit) をスタックにプッシュする	不可
ldc2	3	クラス定数プールへのインデックス (16bit) をスタックにプッシュする	不可
ldc2w	3	クラス定数プールへのインデックス (long 型または double 型) をスタックにプッシュする	不可
aconst_null		オブジェクトリファレンス型の null をスタックにプッシュする	実装済み
iconst_m1	1	int 型の-1 をプッシュする	実装済み
iconst_n	1	int 型の定数 n をプッシュする (n は 0 から 5 までの整数)	実装済み
lconst_l	1	long 型の定数 l をプッシュする (l は 0 または 1)	実装済み
fconst_f	1	float 型の定数 f をプッシュする (f は 0 から 2 までの浮動小数点数)	実装済み
dconst_d	1	double 型の定数 d をプッシュする (d は 0 または 1)	実装済み

3.1.4 ローカル変数のインデックスを拡張する命令

名称	命令長	機能	ハードウェアでの実装
wide	1	ローカル変数のインデックスを 16bit に拡張する	不可

3.1.5 配列管理命令

名称	命令長	機能	ハード ウェアでの実装
newarray	2	新しい配列を割り当てる	不可
anewarray	3	オブジェクトの配列を割り当てる	不可
multianewarray	4	多次元の配列を割り当てる	不可
arraylength	1	配列の長さを得る	不可
iaload	1	配列から int 型の値をロード する	不可
laload	1	配列から long 型の値をロード する	不可
faload	1	配列から float 型の値をロード する	不可
daload	1	配列から double 型の値をロード する	不可
aaload	1	配列からオブジェクト リファレンス型の値をロード する	不可
baload	1	配列から byte 型の値をロード する	不可
caload	1	配列から char 型の値をロード する	不可
saload	1	配列から short 型の値をロード する	不可
iastore	1	配列に int 型の値をストアする	不可
lastore	1	配列に long 型の値をストアする	不可
fastore	1	配列に float 型の値をストアする	不可
dastore	1	配列に double 型の値をストアする	不可
aastore	1	配列にオブジェクト リファレンス型の値をストアする	不可
bastore	1	配列に byte 型の値をストアする	不可
castore	1	配列に char 型の値をストアする	不可
sastore	1	配列に short 型の値をストアする	不可

3.1.6 スタック命令

名称	命令長	機能	ハード ウェアでの実装
nop	1	何もしない	実装済み
pop	1	スタックの最上位をポップする	実装済み
pop2	1	スタックの最上位と 2 番目をポップする	実装済み
dup	1	スタックの最上位をコピーする	実装済み
dup2	1	スタックの最上位と 2 番目をコピーする	実装済み
dup_x1	1	スタックの最上位をコピーし、2 つ下の領域に挿入する	実装済み
dup2_x1	1	スタックの最上位と 2 番目をコピーし、2 つ下の領域に挿入する	実装済み
dup_x2	1	スタックの最上位をコピーし、3 つ下の領域に挿入する	実装済み
dup2_x2	1	スタックの最上位と 2 番目をコピーし、3 つ下の領域に挿入する	実装済み
swap	1	スタックの最上位と 2 番目を交換する	実装済み

3.1.7 数値演算命令

名称	命令長	機能	ハード ウェアでの実装
iadd	1	int 型の加算をする	実装済み
ladd	1	long 型の加算をする	実装済み
fadd	1	float 型の加算をする	実装予定
dadd	1	double 型の加算をする	実装予定
isub	1	int 型の減算をする	実装済み
lsub	1	long 型の減算をする	実装済み
fsub	1	float 型の減算をする	実装予定
dsub	1	double 型の減算をする	実装予定
imul	1	int 型の乗算をする	実装予定
lmul	1	long 型の乗算をする	実装予定
fmul	1	float 型の乗算をする	実装予定
dmul	1	double 型の乗算をする	実装予定
idiv	1	int 型の除算をする	実装予定
ldiv	1	long 型の除算をする	実装予定
fdiv	1	float 型の除算をする	実装予定
ddiv	1	double 型の除算をする	実装予定
irem	1	int 型の除余算をする	実装予定
lrem	1	long 型の除余算をする	実装予定
frem	1	float 型の除余算をする	実装予定
drem	1	double 型の除余算をする	実装予定
ineg	1	int 型のデータ符号を反転させる	実装済み
lneg	1	long 型のデータ符号を反転させる	実装済み
fneg	1	float 型のデータ符号を反転させる	実装済み
dneg	1	double 型のデータ符号を反転させる	実装済み

3.1.8 分岐命令

名称	命令長	機能	ハードウェアでの実装
ifeq	3	スタックの最上位が 0 であれば分岐する	実装済み
ifnull	3	スタックの最上位が null であれば分岐する	実装済み
iflt	3	スタックの最上位が負であれば分岐する	実装済み
ifle	3	スタックの最上位が 0 以下であれば分岐する	実装済み
ifne	3	スタックの最上位が 0 でなければ分岐する	実装済み
ifnonnull	3	スタックの最上位が null でなければ分岐する	実装済み
ifgt	3	スタックの最上位が正であれば分岐する	実装済み
ifge	3	スタックの最上位が 0 以上であれば分岐する	実装済み
if_icomeq	3	スタックの最上位と 2 番目が等しければ分岐する	実装済み
if_icmpne	3	スタックの最上位と 2 番目が等しくなければ分岐する	実装済み
if_icmplt	3	スタックの最上位が 2 番目よりも大きければ分岐する	実装済み
if_icmpgt	3	スタックの最上位が 2 番目よりも小さければ分岐する	実装済み
if_icmple	3	スタックの最上位が 2 番目以上であれば分岐する	実装済み
if_icmpge	3	スタックの最上位が 2 番目以下であれば分岐する	実装済み
if_acmpeq	3	スタックの最上位がオブジェクトリファレンス型の null であれば分岐する	実装済み
if_acmpne	3	スタックの最上位がオブジェクトリファレンス型の null でなければ分岐する	実装済み
goto	3	16 ビットのオフセット値で分岐する	実装済み
goto_w	5	32 ビットのオフセット値で分岐する	実装済み
jsr	3	16 ビットのオフセット値でサブルーチンに分岐する	実装済み
jsr_w	5	32 ビットのオフセット値でサブルーチンに分岐する	実装済み
ret	2	サブルーチンから間接分岐する	実装済み
ret_w	3	サブルーチンから間接分岐する (wide index)	実装済み

3.1.9 リターン命令

名称	命令長	機能	ハードウェアでの実装
ireturn	1	int 型の値をリターンする	不可
lreturn	1	long 型の値をリターンする	不可
freturn	1	float 型の値をリターンする	不可
dreturn	1	double 型の値をリターンする	不可
areturn	1	オブジェクトリファレンス型の値をリターンする	不可
return	1	リターンする	不可
breakpoint	1	ブレークポイントハンドラを呼び出す	不可

3.1.10 比較命令

名称	命令長	機能	ハードウェアでの実装
lcmp	1	long 型の値を比較する	実装済み
fcmpl	1	float 型の値を比較する	実装予定
fcmpg	1	float 型の値を比較する	実装予定
dcmpl	1	double 型の値を比較する	実装予定
dcmpg	1	double 型の値を比較する	実装予定

3.1.11 論理演算命令

名称	命令長	機能	ハードウェアでの実装
ishl	1	int 型の値を左へシフトする	実装済み
ishr	1	int 型の値を右へ算術シフトする	実装済み
iushr	1	int 型の値を右へ論理シフトする	実装済み
lshl	1	long 型の値を左へシフトする	実装済み
lshr	1	long 型の値を右へ算術シフトする	実装済み
lushr	1	long 型の値を右へ論理シフトする	実装済み
iand	1	int 型の値の論理積を求める	実装済み
land	1	long 型の値の論理積を求める	実装済み
ior	1	int 型の値の論理和を求める	実装済み
lor	1	long 型の値の論理和を求める	実装済み
ixor	1	int 型の値の排他的論理和を求める	実装済み
lxor	1	long 型の値の排他的論理和を求める	実装済み

3.1.12 型変換命令

名称	命令長	機能	ハードウェアでの実装
i2l	1	int 型の値を long 型に変換する	実装済み
i2f	1	int 型の値を float 型に変換する	実装予定
l2d	1	int 型の値を double 型に変換する	実装予定
l2i	1	long 型の値を int 型に変換する	実装済み
l2f	1	long 型の値を float 型に変換する	実装予定
l2d	1	long 型の値を double 型に変換する	実装予定
f2i	1	float 型の値を int 型に変換する	実装予定
f2l	1	float 型の値を long 型に変換する	実装予定
f2d	1	float 型の値を double 型に変換する	実装予定
d2i	1	double 型の値を int 型に変換する	実装予定
d2l	1	double 型の値を long 型に変換する	実装予定
d2f	1	double 型の値を float 型に変換する	実装予定
int2byte	1	int 型の値を符号付きの byte 型に変換する	実装済み
int2char	1	int 型の値を char 型に変換する	実装済み
int2short	1	int 型の値を符号付きの short 型に変換する	実装済み

3.1.13 スイッチ命令

名称	命令長	機能	ハードウェアでの実装
tableswitch	不明	ジャンプ命令を用いて分岐する	不可
lookupswitch	不明	キーを使ったジャンプ命令を用いて分岐する	不可

3.1.14 フィールド操作命令

名称	命令長	機能	ハードウェアでの実装
putfield	3	オブジェクトのフィールドの値を変更する	不可
getfield	3	オブジェクトのフィールドの値を読み出す	不可
putstatic	3	クラスの静的フィールドの値を変更する	不可
getstatic	3	クラスの静的フィールドの値を読み出す	不可

3.1.15 メソッド呼び出し命令

名称	命令長	機能	ハードウェアでの実装
invokevirtual	3	実行時にメソッドを呼び出す	不可
invokenonvirtual	3	コンパイル時にメソッドを呼び出す	不可
invokestatic	3	クラスメソッドを呼び出す	不可
invokeinterface	5	インターフェースメソッドを呼び出す	不可

3.1.16 例外処理命令

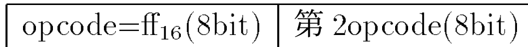
名称	命令長	機能	ハードウェアでの実装
athrow	1	例外処理を起こす	不可

3.1.17 その他のオブジェクト操作命令

名称	命令長	機能	ハードウェアでの実装
new	1	新しいオブジェクトを生成する	不可
checkcast	1	オブジェクトの型を調べる	不可
instanceof	1	オブジェクトの型を比較する	不可

3.1.18 スレッド制御命令

名称	命令長	機能	ハードウェアでの実装
monitorenter	1	他のスレッドにオブジェクトの使用を禁止する	不可
monitorexit	1	他のスレッドにオブジェクトの使用を解放する	不可



opcode=ff_{16}(8bit)	第 2opcode(8bit)

Figure 3.1: 独自命令

3.2 Java Virtual Machine 非互換命令

以下に示す命令は Java Virtual Machine 互換ではなく、未実装命令のサポートのために独自に実装した命令である。これらはすべてハードウェアで実装した。TRAJA Version 1 では未使用の opcode に割り当てていたが、現在空いている opcode は特殊命令を除いても 53 種類しかなく、今後拡張が行われる可能性もあるのでこの方法は好ましくない。

そこで、今回開発したプロセッサでは Figure 3.1 のように独自命令の opcode を 255(ff_{16}) に限定し、それに付随する 1 バイトのオペランド (第 2opcode) との組み合わせで 1 命令を表現する方式をとった。命令の実行に必要なパラメータはすべてスタックから供給するため、命令長はすべて 2 バイトである。

名称	**第 2opcode**	**機能**
ch_sp	01_{16}	スタックの最上位を sp に代入する
ch_vars	02_{16}	スタックの最上位を vars レジスタに代入する
ch_expt	03_{16}	スタックの最上位を expt レジスタに代入する
load	10_{16}	スタックの最上位に格納されたアドレスが示すデータ (32bit) をプッシュする
load_w	11_{16}	スタックの最上位に格納されたアドレスが示すデータ (64bit) をプッシュする
store	12_{16}	スタックの最上位が示すアドレスに、2 番目に格納されたデータ (32bit) をストアする
store_w	13_{16}	スタックの最上位が示すアドレスに、2 番目及び 3 番目に格納されたデータ (64bit) をストアする
push_pc	20_{16}	次に実行するアドレスの値をプッシュする
push_sp	21_{16}	sp の値をプッシュする (sp の値はこの命令を実行する直前のもの)
push_vars	22_{16}	vars レジスタの値をプッシュする
push_expt	23_{16}	expt レジスタの値をプッシュする
jump	30_{16}	スタックの最上位に格納されたアドレスに分岐する
jump_sr	31_{16}	次に実行するアドレスの値をプッシュし、スタックの最上位に格納されたアドレスに分岐する
nload	40_{16}	スタックの最上位に格納されているアドレスから始まるメモリ領域を、2 番目に格納されているワード数だけスタックにロードする。
nstore	41_{16}	スタックの最上位に格納されているアドレスから始まるメモリ領域に、2 番目に格納されているワード数だけスタックからストアする。

Chapter 4

TRAJA Version 2 のアーキテクチャ

4.1 TRAJA Version 2 について

第 3 回 ASIC デザインコンテスト作品として開発した TRAJA Version 1 は JVM 互換命令の実行に最低限必要な機能だけを備えたシンプルなプロセッサであった。そのため、性能に関しては改善すべき点が多く残されていた。今回開発した TRAJA Version 2 では、以下に説明するようなパイプライン、スタックキャッシュなどによって、特に CPI の減少を目指している。

TRAJA Version 2 は TRAJA Version 1 と比較して以下のような特徴を持つ。

- **パイプライン**
  命令の実行を 4 つのステージに分割し、それぞれが平行動作するため、ピーク時にはほとんどの命令を 1 クロックで実行することが可能である
- **スタックキャッシュ**
  スタックマシンである JVM の高速な演算をサポートするために、自動制御のスタックキャッシュを実装した
- **命令セット**
  前回実装した 62 種類の JVM 互換命令に加え、ローカル変数命令、型変換命令など幅広い命令をサポートしている
- **エミュレーションへの対応**
  独自命令の改善とともに、未実装命令をサポートするための機構を新たに考案した

4.2 パイプライン

4.2.1 ステージの構成

TRAJA Version 2 は以下に示す 4 ステージのパイプライン構造を持っている。それぞれのステージの概略は以下の通りである。

1. **IF(Instruction Fetch) ステージ**
   命令が格納されているワードをプログラムカウンタに先行してフェッチする
2. **ID(Instruction Decode) ステージ**
   IF ステージでフェッチされたワードを組み合わせて実行データとして整形し、実行データバッファに格納する
3. **OC(Operand Creation) ステージ**
   実行データバッファからデータを取り出し、命令実行に必要なデータを先行して計算する
4. **EX(EXecution) ステージ**
   命令データを解読し、実行する

Figure 4.1 はステージの構成と命令実行の流れを示したものである。

今回設計したアーキテクチャの特徴として、通常のプロセッサとは異なり、データを書き戻すためのライトバックステージを持たないことがあげられる。これはスタックキャッシュの構造上の特徴から、1 クロックで命令実行に必要なデータの読み出しと書き戻しができるためである。そして、スタックキャッシュの操作ができるのは EX ステージに限定している。これによって、フォワーディングを使用しない簡潔なアーキテクチャが実現できた。

また、可変長命令を持つ JVM では命令フェッチ/デコードに 2 ワードの命令データを必要とするが、これは命令プリフェッチバッファに命令データを蓄積して、任意の 2 ワードを取り出せるようにして対応している。IF ステージや ID ステージは処理が終了したデータを格納する専用のバッファを持つ。そのため、命令実行がストールしたり、複数クロックを消費する命令を実行しているときでもしているときでも独立して動作し、性能の向上が期待できる。

OC ステージは TRAJA Version 2 独自のものであるが、実行時に必要なデータを先行して計算するため、一部の命令の実行サイクルを確実に短縮できる。

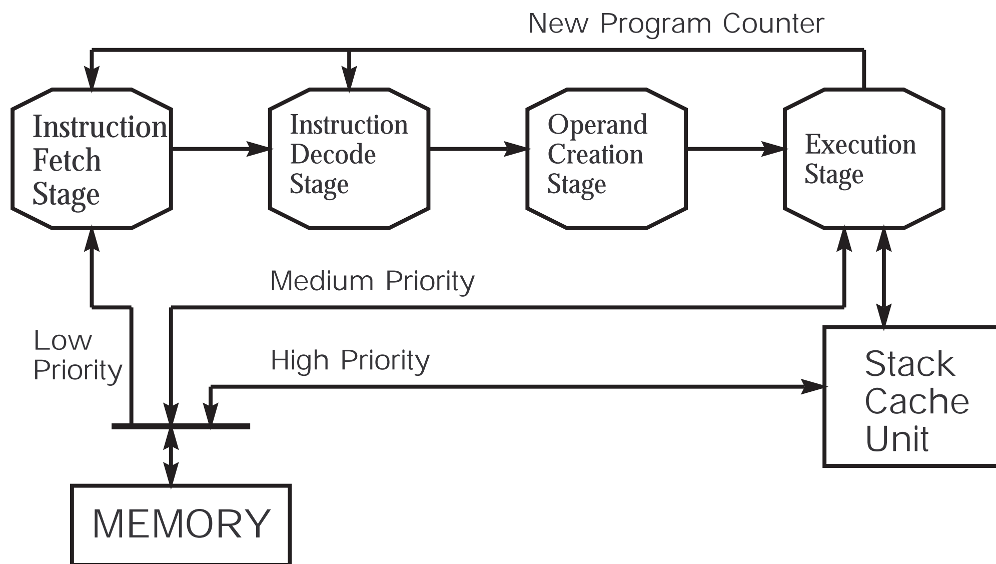


Figure 4.1: パイプラインステージ

4.2.2 IF ステージの動作

IF ステージではプログラムカウンタから連続するデータを、全 4 エントリの命令プリフェッチバッファに自動的に記憶していく。フェッチしたデータを格納するエントリは決まっていて、フェッチしたアドレスのワードオフセットを除く下位 2bit に対応したダイレクトマップ方式である。使用済みのエントリは ID ステージで解放されるので、IF ステージ専用プログラムカウンタ (pcif) に 4 を加算すると同時に、解放されているエントリに順番に書き込んでいく。各エントリが空であるかどうかはそのエントリの Valid bit から判断できる。各エントリの bit 幅は Valid bit を含めて 33bit である。

ただし、IF ステージはメモリアクセスに関する優先度を最も低く設定しているため、スタックキャッシュ及び EX ステージがメモリにアクセスしているときには、そのクロックでは何もできない。しかしバイトコードにおける命令長は、1 バイトが平均して約 1.8 バイトなので、IF ステージは 2 クロックに 1 回動作すれば特に問題はないと言える。

分岐が発生した場合 (内部端子:branch がアサートされたとき) には、EX ステージから新しいプログラムカウンタの値が与えられるので、すべてのエントリを消去し、pcif を更新する。Figure 4.2 は IF ステージの構成を示したものである。

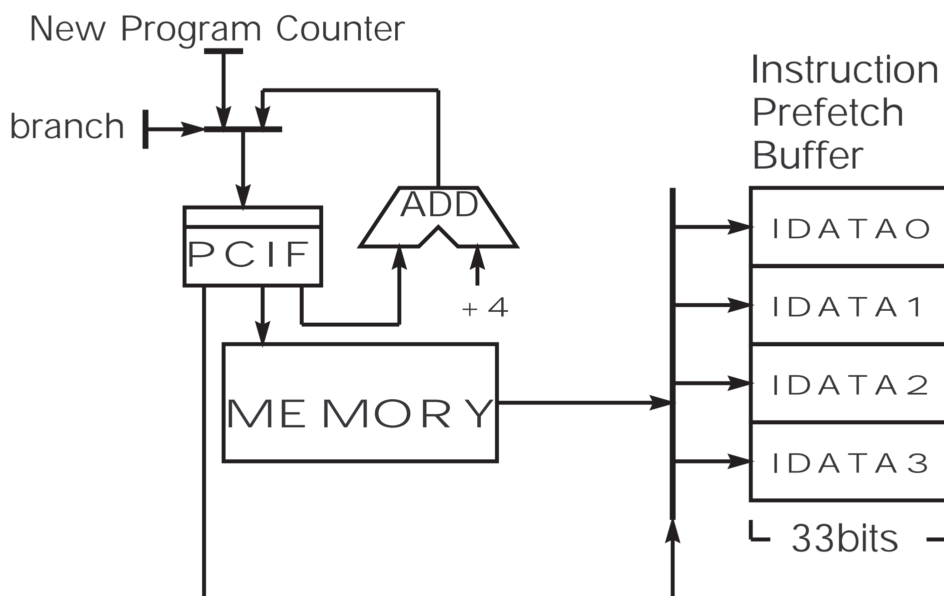


Figure 4.2: IF ステージ

4.2.3 ID ステージの動作

ID ステージではまず、命令プリフェッチバッファをチェックして、プログラムカウンタが示すアドレスのワードとその直後のワードの両方がそろっていることを確認する。両方そろっているならば、1〜5 バイトの命令長を持つ命令はこの 2 ワード中に絶対に格納されている。JVM には 6 バイト以上の命令長を持つ命令も若干存在するが、その数はきわめて少ないため、これらは例外的な処理によってソフトウェアエミュレーションを行う。

そして実行すべき命令を格納する命令実行バッファに書き込むべきエントリ (レジスタ :wcounter が示しているエントリ) が空いているとき、命令デコードが発動する。

フェッチされた 2 ワードは命令データ専用シフタ (irshift) を通し、pc の下位 2bit に 8 をかけた bit 数だけ左にシフトする。この動作で、その命令がどのような動作をするかを判断するための opcode を最上位バイトに格納させる。最高 32bit のオペランドは opcode の直後のバイトに連結されている。

そして、命令データとプログラムカウンタ及び Valid bit(エントリが有効であることを示す bit) を連結して実行データとして整形し、実行データバッファの wcounter が示すエントリに書き込み、wcounter をインクリメントする。実行データがプログラムカウンタを含んでいるのは分岐命令がオペランドとして使用するためである。

opl は opcode を入力すると、その命令長を出力するモジュールである。これによって得られた命令長は pc に加算され、次の命令のアドレスにセットされる。同時に、命令長とプログラムカウンタの下位 2bit から不要になる命令プリフェッチバッファのエントリ数を算出して、エントリの削除を行う。

分岐が発生した場合は IF ステージと同様に、プログラムカウンタ (この値は IF ステージと共通) が更新され、実行データバッファのエントリをすべて無効化する。

opcode が判断不能な命令であるときには、その命令長は 0 バイトとして EX ステージで例外処理を行う。

Figure 4.3 は ID ステージの構成を示したものである。

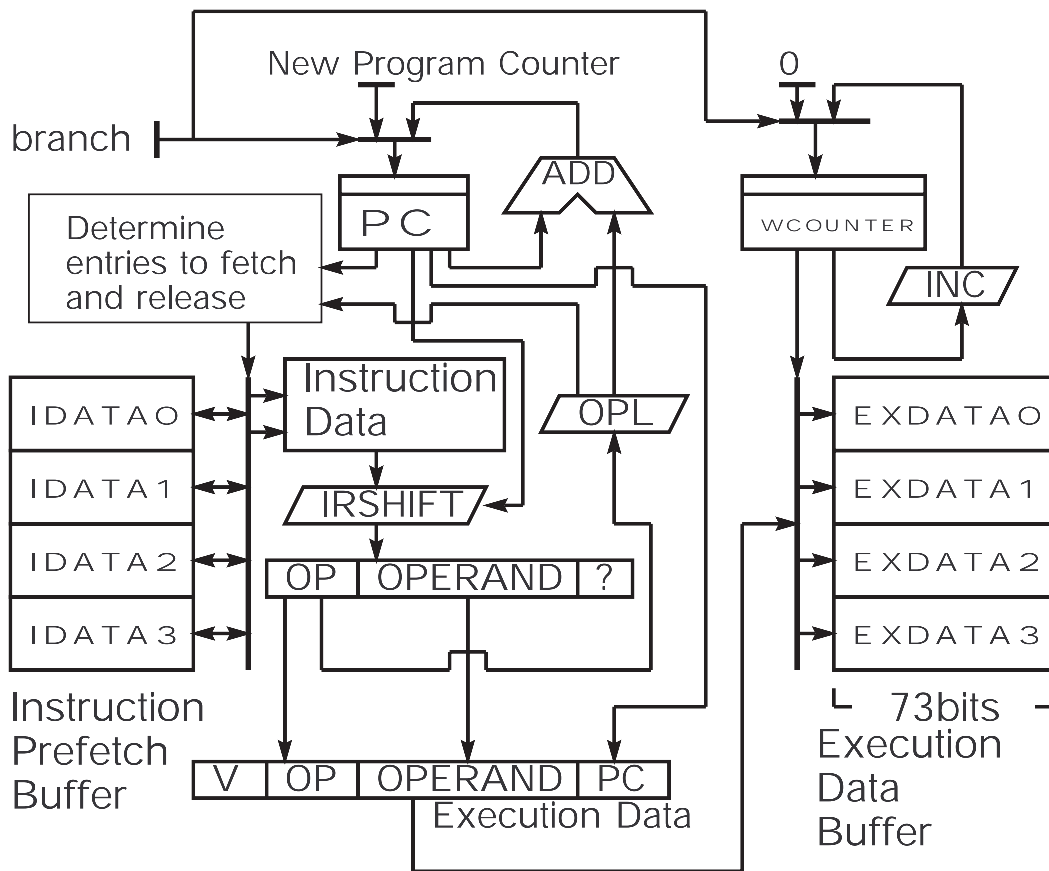


Figure 4.3: ID ステージ

4.2.4 OC ステージの動作

OC ステージは EX ステージから次の命令の準備の要求があったとき (内部端子:exrls がアサートされたとき) に発動する。OC ステージの役割はオペランド (exoperand) から導き出される実行に必要なデータを第 2 オペランド (exoperand2) として EX ステージに提供することである。ただし EX ステージで vars レジスタ変更命令を実行中のときは強制的にストールさせる。第 2 オペランドの種類は次の 3 つである。

- **分岐命令**
  第 2 オペランド =分岐先のアドレス (プログラムカウンタ+オペランド)
- **ロード /ストア命令**
  第 2 オペランド =メモリ参照アドレス (vars レジスタ+オペランド または定数)
- **上記以外の命令**
  第 2 オペランド =0

第 2 オペランドを生成した後、EX ステージが命令を実行するために必要な以下の 4 つのパラメータを更新する。このパラメータは次のクロックで有効となり、実行が開始される。

- **opcode(exop)**
  命令の種類を識別するために使用
- **第 1 オペランド (exoperand)**
  opcode に付随するオペランド
- **第 2 オペランド (exoperand2)**
  実行時に計算する値
- **プログラムカウンタ (expc)**
  命令が格納されているアドレス

もし次の命令の要求があったが、レジスタ:excounter が示す実行データバッファのエントリが有効ではないときには、そのエントリが有効になるまで nop 命令を EX ステージに送出し続ける。nop 命令は 1 サイクルの間何も実行せずに待機する命令であり、JVM 命令セットに含まれている。

Figure 4.4 は OC ステージの構成を示したものである。

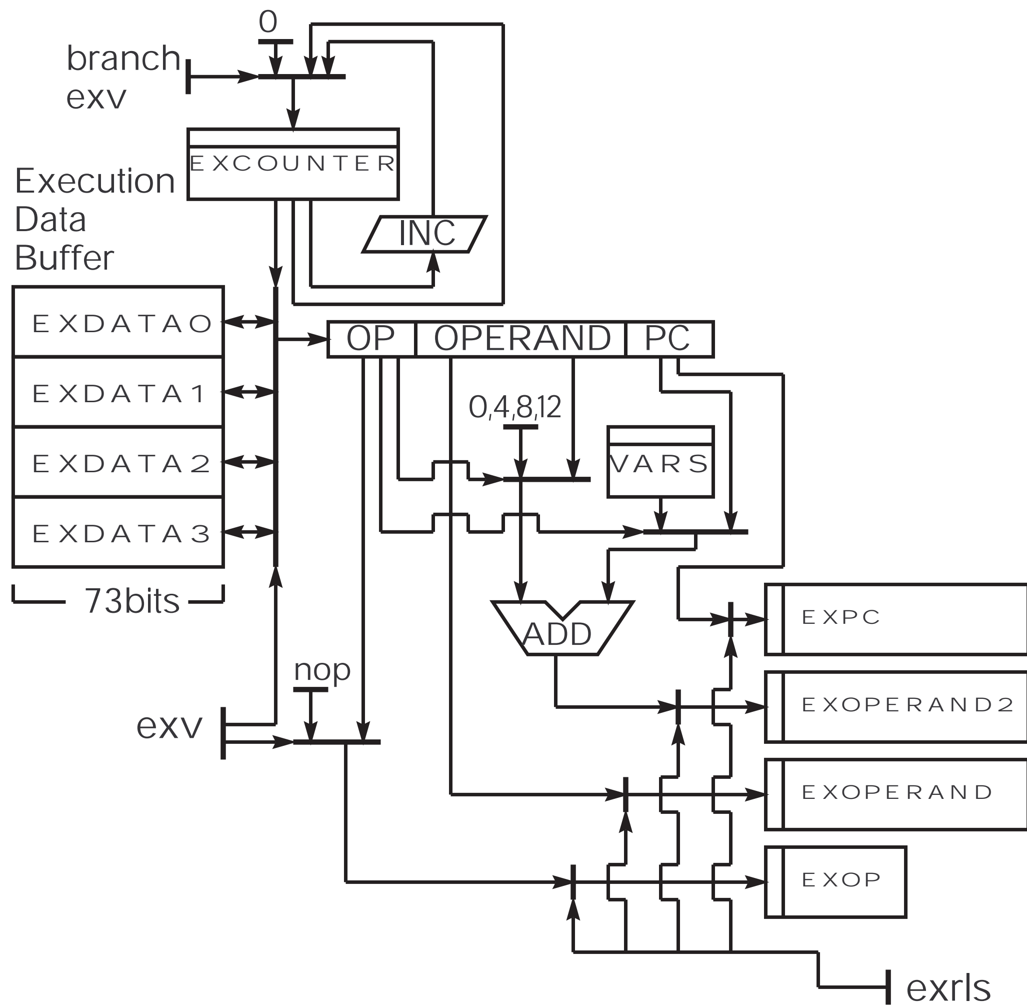


Figure 4.4: OC ステージ

4.2.5 EX ステージの動作

EX ステージは OC ステージから渡された 4 つのパラメータによって命令を実行する。JVM 命令セットではほとんどの命令がスタックを使用するため、スタックキャッシュが使用不可能であるときには、EX ステージの動作はほぼストールしてしまう。

スタックを使用する命令は、まずスタックキャッシュが使用可能かどうかを調べ、使用可能ならばスタックキャッシュ制御パラメータを渡す。スタックキャッシュから受け取った出力は、演算終了後に直接スタックキャッシュへ戻す。スタックキャッシュは前述の制御パラメータを解読して、演算中にエントリの再配置動作を行うので EX ステージはスタックのエントリを直接操作する必要はない。

演算はすべて 64bit の値を扱えるモジュールを使用している。これは JVM に 64bit の値を扱う命令が多いことが理由である。スタックキャッシュは 2 本の 64bit 出力端子 (または 32bit × 4 本) と 1 本の 64bit 入力端子 (または 32bit × 2 本) を持っているので、JVM の命令セットにある ladd などの命令を 1 クロックで実行できる。

命令の実行中に、現在のクロックで命令実行が終了すると判断されたときには内部端子:exrls をアサートする。OC ステージはこれを感知して、命令実行パラメータである exop、exoperand、exoperand2、expc の 4 つをレジスタ更新する。次のクロックにはこれらのパラメータが有効になり、次の命令の実行を開始する。

分岐命令によって分岐が発生したときには、内部端子:branch をアサートして、分岐先のアドレスを内部端子:newpc_tmp に出力する。この動作で、IF ステージ及び ID ステージのプログラムカウンタは自動的に更新されて、IF ステージは次のクロックで分岐先のアドレスから命令読み出しを始める。ID ステージも同様に命令データ 2 ワードが揃うまで待機する。分岐が発生すると IF に 2 クロック、ID に 1 クロック、OC に 1 クロックかかるため、EX ステージは少なくとも 4 クロックの間ストールしてしまう。

実行中の opcode である exop が認識できない命令 (未実装命令) であるときには、レジスタ:expt が示すアドレスに分岐する。

Figure 4.5 は EX ステージの構成を示したものである。ただし、この図で示した ALU は、型変換、シフタ、整数/浮動小数点数演算器などの演算装置全体を示すものである。整数演算命令はすべて 1 クロックで実行が終了するが、浮動小数点演算、型変換、メモリアクセスを要する命令などは複数クロックに分割して実行する。

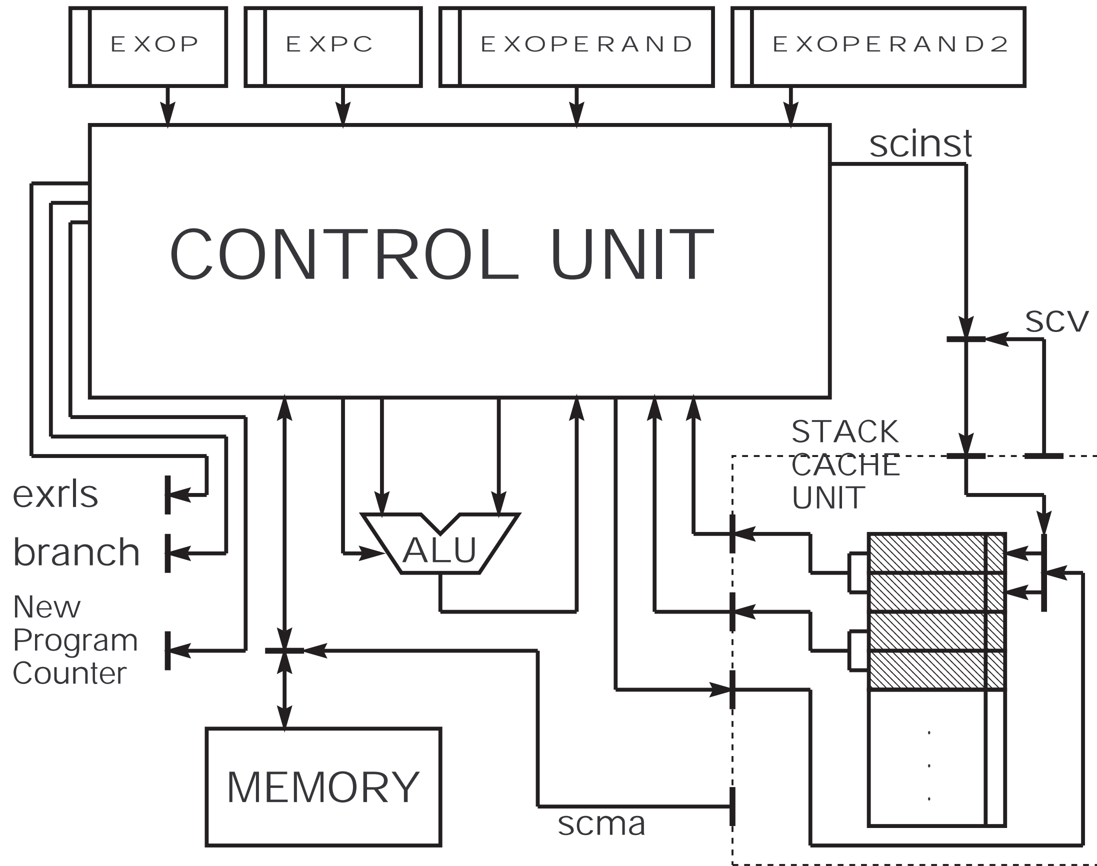


Figure 4.5: EX ステージ

4.3 スタックキャッシュ

4.3.1 スタックキャッシュの概要

TRAJA Version 1 では、スタックを用いた命令の実行のために複数サイクル消費していた。例えば、long 型の和を求める「ladd」命令では実行サイクルは以下に示すようになる。

1. 命令を読み出し、整形する (3 サイクル)
2. 必要になるオペランドのデータをメモリから 32bit ごとに読み出す (4 サイクル)
3. 演算を実行する (1 サイクル)
4. 演算結果をメモリに書き戻す (2 サイクル)

したがって、実行サイクルは合計で 10 サイクルになる。命令実行のオーバーラップもしていない。JVM はスタック型のアーキテクチャを採用しているので、ほとんどの命令がスタックを使用する。そのため、この多大な実行サイクルは性能向上の足枷になる。そこで、TRAJA Version 2 では高速化の手段として独自のスタックキャッシュを実装している。前述の ladd 命令は以下に示す手続きを同時に行うことによって 1 サイクルで実行できる。また、メモリへのアクセスも最小限にとどめている。

1. スタックキャッシュが使用可能であるとき、scinst に 1010010_2を出力する (外部命令、出力が 32bit × 4 本、入力が 32bit × 2 本という意味)。
2. スタックキャッシュの出力端子を ALU の入力端子に、入端子を出力端子に接続する。
3. ALU に加算をさせる
4. 命令の実行が成功するとき (スタックがしよう可能である、つまり内部端子:scv がアサートされているとき)、次に実行すべき命令を要求する

名称	**bit 幅**	**機能**
scinst	7bit	スタックキャッシュ制御命令端子
scin0〜1	各 32bit	データ入力端子
scout0〜3	各 32bit	データ出力端子
scma	1bit	スタックキャッシメモリアクセスフラグ
scv	1bit	スタックキャッシュ有効フラグ

Table 4.1: スタックキャッシュの外部から見える端子

4.3.2 スタックキャッシュのアーキテクチャ

Figure 4.6 はスタックキャッシュの構成図である。スタックキャッシュは 32 個のエントリを持ち、1 つのエントリは 32bit のデータ部と 1bit の Valid bit から構成されるレジスタである。Valid bit はそれぞれのエントリに格納されている値が有効であるかを示す。

各エントリはそれぞれ 1 つのメモリアドレスに対応し、スタックの最上位のエントリのアドレスはスタックポインタに格納される。最上位以外のエントリのアドレスはスタックポインタからのオフセットで知ることができる。ただし、スタックの各要素はメモリ上には 32bit ごとに区切られているため、スタックポインタの下位 2bit はワードオフセットとして無視される。

スタックキャッシュの操作はすべて専用の制御命令によって行う。これによって、スタックキャッシュを使用する命令はスタックのエントリを直接操作する必要がなくなり、構造を簡単にすることができた。

スタックキャッシュの外部から見える端子は Table 4.1 のようになっている。このほかにも制御用のカウンタやエントリ増減用の制御端子などが存在するが、スタックキャッシュの外部からは操作できない。

スタックキャッシュは内部端子:scinst に入力されたデータによって各エントリを操作する。ただし、scinst が有効になるのは内部端子:scv が 1(ON) の時だけである。スタックキャッシュは 1 つのステージとして動作しているため、他のステージがスタックキャッシュのエントリを勝手に参照したり更新することは禁止されている。データの読み出し、書き込み、内部命令の実行などを行うためには、scinst によってスタックキャッシュに要求を出さなければいけない。

スタックのエントリ数が足りないとき (4 番目のエントリの Valid bit が 0 のとき) は自動的にデータフェッチ状態になり、メモリからデータを取り出す。逆にオーバーフローしたとき (31 番目のエントリの Valid bit が 1 のとき) にはライトバック状態になり、データをメモリに書き戻す。データフェッチ状態やライトバック状態などでは scv が 0(OFF) にセッ

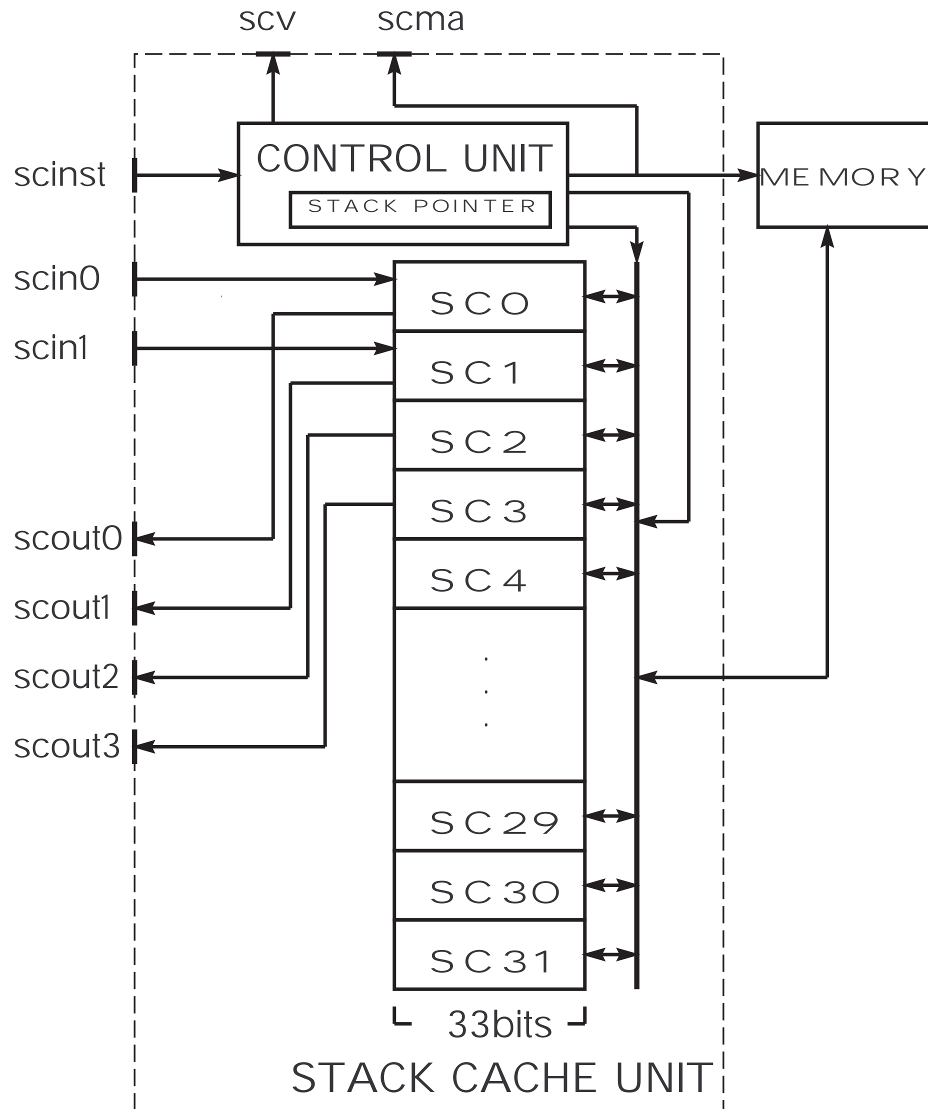


Figure 4.6: スタックキャッシュの構成図

トされ、スタックキャッシュの操作は禁止される。この状態遷移によって、スタックキャッシュの有効エントリ数は常に 4〜30 個に保たれる。

Figure 4.7 はスタックキャッシュの状態遷移を図で表したものである。

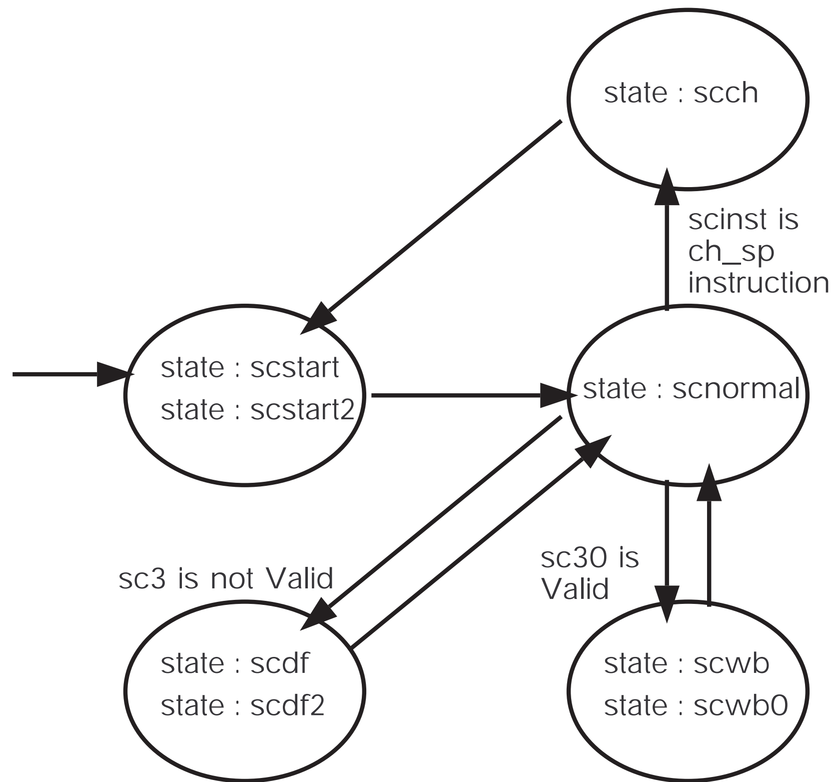


Figure 4.7: スタックキャッシュの状態遷移

scinst<6:5> 命令モード	scinst<4:2> 出力エントリ数	scinst<1:0> 入力エントリ数	エントリ数の増減	使用命令
10_2	000_2	00_2	0	nop
10_2	000_2	01_2	+1	bipush,iconst,jsr など
10_2	000_2	10_2	+2	lconst,dconst など
10_2	001_2	00_2	-1	ifeq など
10_2	001_2	01_2	0	ineg,i2f など
10_2	001_2	10_2	+1	i2l など
10_2	010_2	00_2	-2	if_icmpeq など
10_2	010_2	01_2	-1	l2i,iadd,fcmpg,ishl など
10_2	010_2	10_2	0	lneg,l2d など
10_2	011_2	00_2	-3	
10_2	011_2	01_2	-2	
10_2	011_2	10_2	-1	lshl など
10_2	100_2	00_2	-4	
10_2	100_2	01_2	-3	lcmp など
10_2	100_2	10_2	-2	ladd など

Table 4.2: スタックキャッシュの外部命令

スタックキャッシュの外部命令

スタックキャッシュの外部のデータやモジュールを使用する操作を、ここではスタックキャッシュの外部命令と呼ぶ。

外部命令は scinst の値によってスタックからパラメータを出力し、入力端子に入ってきたデータをスタックの最上位に格納する。scinst の bit 幅は 7bit であり、外部命令のときは最上位 2bit が 10_2になる。それに続く 3bit と 2bit の値によって、出力端子数と入力端子数を決定する。出力端子と入力端子の数が決定すれば、エントリの増減数も決定するので、入力端子の値が有効になるまでの間に無効になるエントリを予測して、エントリを詰めたり新しいエントリを挿入するためにエントリの移動を行う。同時にスタックの最上位を示すスタックポインタの値を更新する。これら動作によって、スタックキャッシュは次のクロックには使用可能な状態になる。

Table 4.2 は scinst の外部命令の機能を示したものである。そして、スタックキャッシュの外部命令を使用する例として、ladd 命令を figure 4.8 に示す。

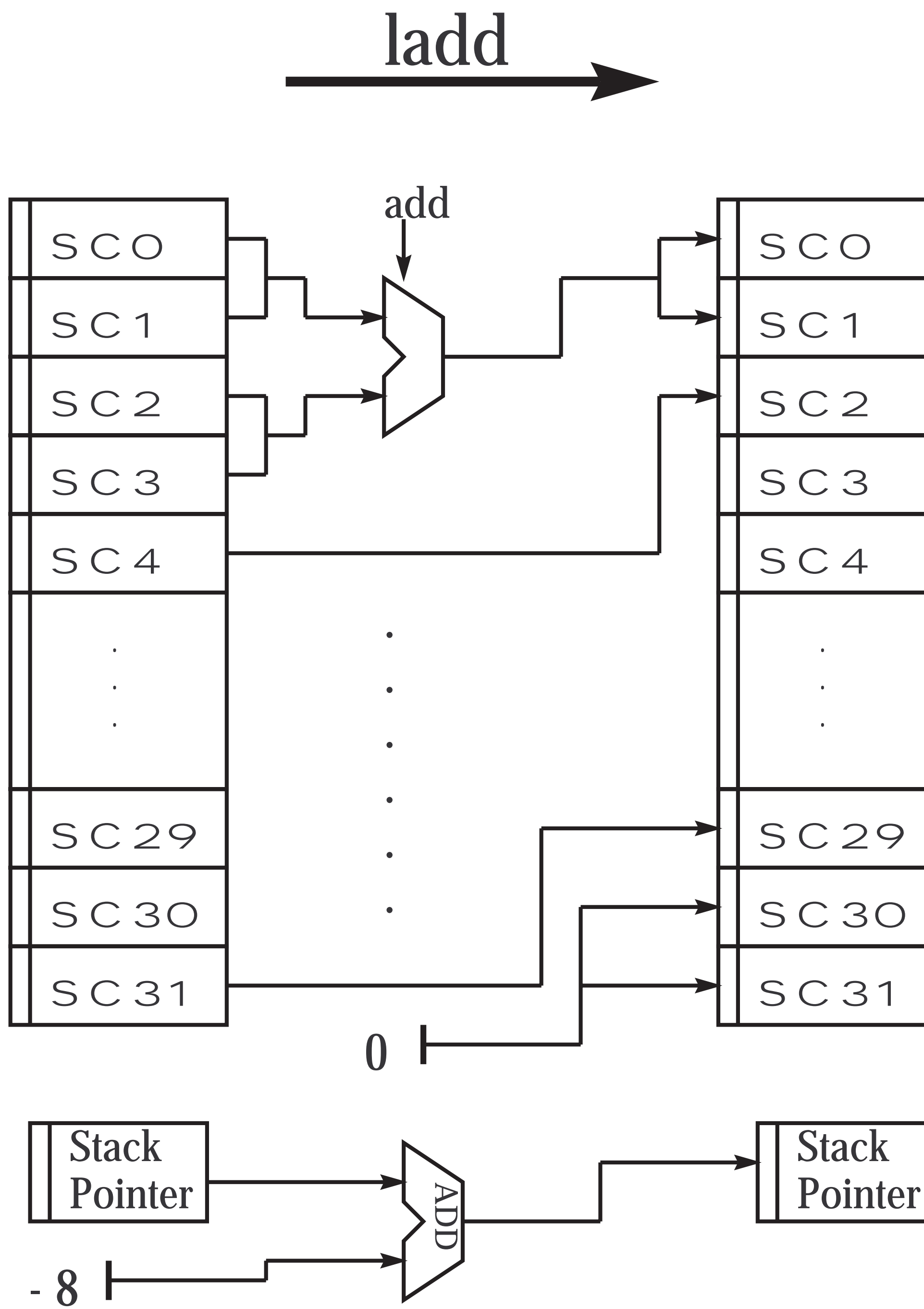


Figure 4.8: 外部命令の使用例 (ladd)

scinst<6:5> 命令モード	scinst<4:0> 命令識別コード	エントリ数の増減	使用命令
11_2	00000_2	-1	pop
11_2	00001_2	-2	pop2
11_2	00010_2	+1	dup
11_2	00011_2	+2	dup2
11_2	00100_2	+1	dup_x1
11_2	00101_2	+2	dup2_x1
11_2	00110_2	+1	dup_x2
11_2	00111_2	+2	dup2_x2
11_2	01000_2	0	swap
11_2	01100_2	-1	ch_sp

Table 4.3: スタックキャッシュの内部命令

スタックキャッシュの内部命令

スタックキャッシュの内部のデータだけを使用する操作を、スタックキャッシュの内部命令と呼ぶ。

内部命令のときは最上位 2bit が 11_2になる。それに続く 5bit の値によって命令の種類を決定する。内部命令は JVM の命令セットにある複雑なスタック管理命令を少ないクロック数で実行するために実装したものである。今回実装した内部命令によってすべてのスタック管理命令を 1 クロックで実行することが可能になった。内部命令も外部命令と同様に、自動的にエントリ、スタックポインタの操作を行う。

Figure 4.9 はスタックキャッシュの内部命令を使用する dup2_x2 命令を図で表したものである。この命令を前述の外部命令だけで実行することも可能であるが、スタックキャッシュには入力端子が 2 つしかないため、6 エントリを操作するこの命令を実行するためには 3 クロックを要する。また一時的にエントリ数が 2 個減少/増加するので、スタックキャッシュの有効エントリ不足でデータフェッチ状態になり、命令実行がストールする可能性もある。しかし、内部命令を使用すれば、エントリ数は 2 個増加するだけであり、1 クロックで実行できるため、確実に性能は向上する。

Table 4.3 は scinst の内部命令の機能を示したものである。

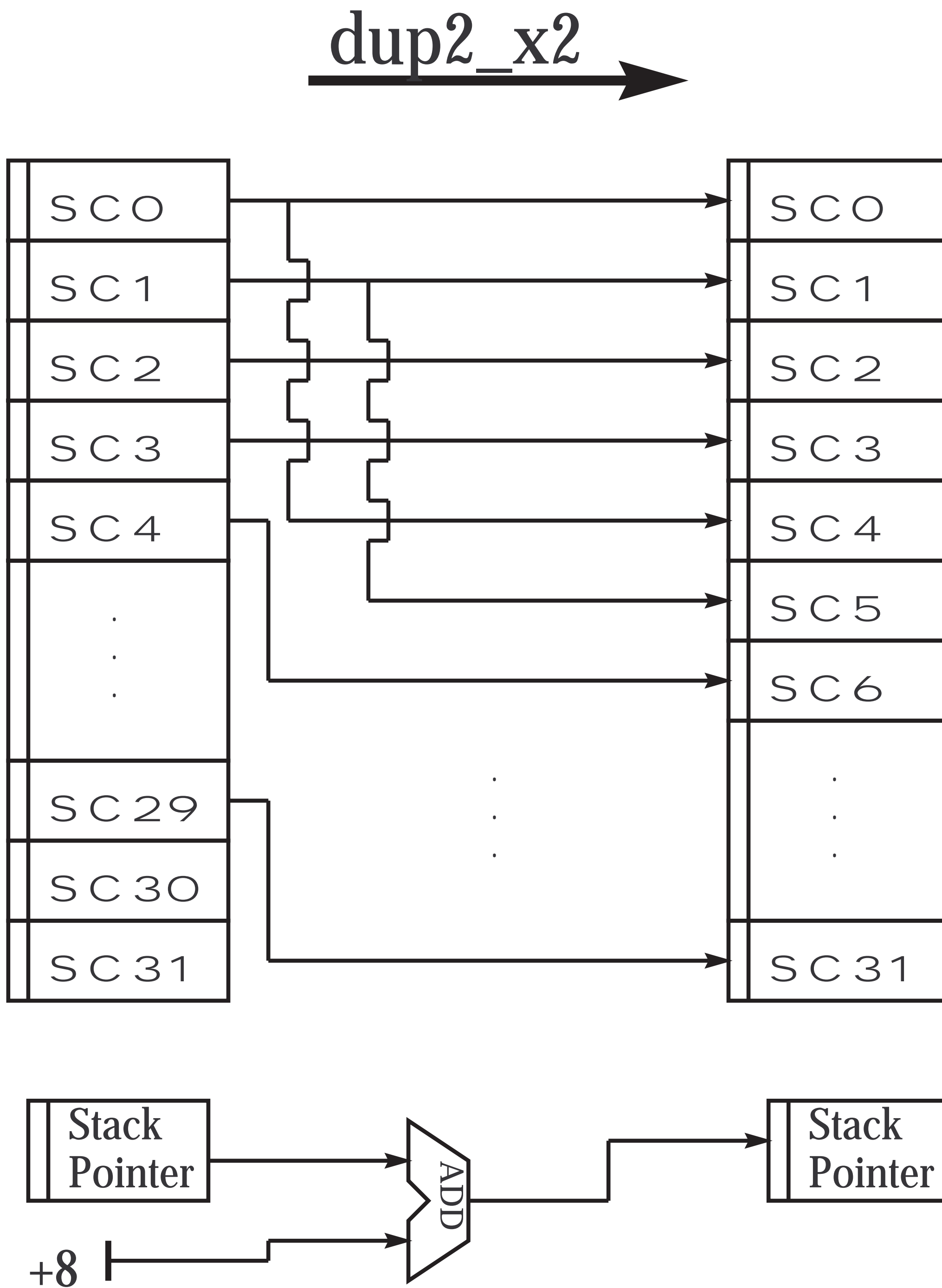


Figure 4.9: 内部命令の使用例 (dup2_x2)

実行クロック数	命令数
1	105
2	20
3	1

Table 4.4: 命令の実行クロック数

4.4 命令セット

4.4.1 ハードウェアで実装した命令セット

前回、TRAJA Version 1 でサポートしていた JVM 完全互換命令は 62 種類であった。TRAJA Version 2 ではさらに広範囲の JVM 完全互換命令の実装及び高速化を目指し、以下のような改良を行った。

- **ローカル変数命令**
  JVM には 50 種類以上のローカル変数命令が存在するが、TRAJA Version 2 ではローカル変数命令をすべて実装しているため、実装命令数は大幅に向上している。
- **演算命令**
  スタックキャッシュ、高速演算回路により CPI は大幅に減少した。また、前回実装できなかった浮動小数点などの命令を追加した
- **独自命令**
  JVM に不足している命令を補うために、独自命令を追加/改良した。

4.4.2 実行クロック

現時点でのハードウェアで実装した JVM 互換命令は 126 種類である。ほとんどの命令は、スタックキャッシュが使用可能な状態にあれば 1 クロックで命令実行が終了する。しかし、若干ではあるが複数クロックを消費する命令もある。Table 4.4 に命令を実行するために必要なクロック数をまとめた。

4.4.3 ソフトウェアエミュレーションのための検討

未実装命令を実行しようとすると、無条件に決まったアドレス (レジスタ:expt で指定される) へ分岐する。その際、どのような例外が起きたのかを分岐先で判断するために、ス

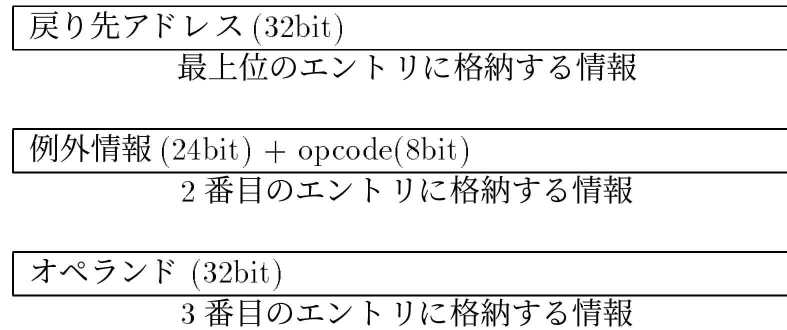


Figure 4.10: 例外発生時に待避する情報

タックキャッシュに以下の 3 エントリを push する。それぞれのエントリの内容は Figure 4.10 に示すようになっている。

最上位のエントリには、実行中の命令のアドレスにその命令の命令長を加算したアドレス、つまり、例外処理終了後の戻り先アドレスを格納する。

2 番目のエントリには例外情報を格納する。これはどのような例外が発生したかを示す情報であり、未実装命令を実行したときには、例外情報として 0 が格納される。opcode は実行中の命令の opcode である。

3 番目のエントリにはオペランドを格納する。オペランドについては opcode に付随するオペランドをすべて格納する。ただし、スイッチ命令については命令長が不定のため、オペランドはその上位 4 バイトまで格納する。

スタックキャッシュには同時に 2 エントリまでしか push できないため、分岐には 2 クロックを要する。分岐先のアドレスでは、これらのデータを解読してソフトウェアエミュレーションを行う。

Chapter 5

プログラムについて

以下に示すファイルは今回作成した TRAJA Version2 のソースファイルである。コードはすべて SFL と呼ばれるハードウェア記述言語で記述した。

ファイル名	内容
javachip.sfl	TRAJA Version 2 の本体
board.sfl	32bit メモリボード
opl.sfl	opcode・命令長変換器
irshift.sfl	命令データ専用シフタ
shift64.sfl	64bit バレルシフタ
javaalu.sfl	64bit 整数演算器
inc.sfl	32bit 整数加算器
add8.sfl	8bit 整数加算器

Chapter 6

結論

今回の研究は特にCPIの減少に注目した。一般的なアプローチとして、現在ではほとんどのプロセッサで使用されているパイプラインをJavachipに適した形にして、独自の手法で実装した。また、Sun Microelectronics社のpicoJavaやPatriot Scientific社のShBoomなどのJavachipと総称されるプロセッサは、どれもスタックキャッシュによって命令実行を高速化している。今回開発したプロセッサのスタックキャッシュは独自のアーキテクチャであり、上記の2つのものとは多少異なるが、複数のレジスタによってスタックを表現する点では共通している。CPIの減少に関してはこれらの手法で大きく改善できると考えている。

しかし、高速化の余地はまだ残されている。たとえば、JVM互換命令をハードウェアで処理しやすいプロセッサ内部の命令に変換したり、複数の命令を1つの命令で置き換えるFolding Logicなどがあげられる。また、複雑な命令が多いJVMではソフトウェアエミュレーションを補助するための独自命令も重要な要素である。

まだ論理合成ができていないため、回路規模や消費電力は不明である。また、クロック周波数に関しては、演算器の改良によりTRAJA Version 1と比較してかなりの向上が見込めるが、詳細は不明である。今後はそれらについても検討し、プロセッサとしての完成度を上げる必要がある。

Chapter 7

謝辞

本論文は多くの人の協力によって完成することができました。本研究の前身であるTRAJA Version 1の開発プロジェクトにおいて共に研究をした黒子周作氏、榎田裕一氏、河野賢一氏、SFL言語にふれる契機を与えていただいたパルテノン研究会の皆様、様々な面で協力してくださった清水研究室のメンバーの方々、そして本研究を円滑に進めることができたのは、東海大学工学部通信工学科の専任講師であられる清水尚彦先生のご指導のおかげであると考えています。

多くの方々からのご支援には本当に感謝しています。

Bibliography

[1] D.A.Patterson and J.L.Hennesy: "Computer Organization and Design -The Hardware/Software Interface", Morgan Kaufmann Publishers Inc., June 1, 1993.

[2] James Gosling, Bill Joy and Guy Steele: "The Java Language Specification", Addison-Wesley Pub. Co., 1996.

[3] "The Java Virtual Environment *A White Paper*", Sun Microsystems Computer Corporation, October,1995.

[4] "Java OS : A Standalone Java Environment *A White Paper*", Java Soft, May,1996.

[5] Tim Lindholm and Frank Yellin: "The Java Virtual Machine Specification", Addison-Wesley Pub. Co., September 1, 1996.

[6] 「Javachip のすべて」, 日経 BP 社, 日経エレクトロニクス 1996 年 6 月 17 日号, no.664, pp.155.

[7] Linley Gwennap: "T9000 Transputer Begins Sampling *Complexity Causes Schedule Delays, Increased Transistor Count*", Ziff Davis Pr., Understanding RISC Microprocessors, pp.13-29, July 1, 1993.

[8] Jim Turley: "New Embedded CPU Goes ShBoom *Patriot Scientific's Unusual 32bit Stack Machine Has 8-Bit Instruction Word*", MicroDesign Resources, Micro Processor Report, Volume 10, Number 5, April 15, 1996.

[9] Brian Case: "Implementing the Java Virtual Machine *Java's Complex Instruction Set Can Be Built in Software or Hardware*", MicroDesign Resources, Micro Processor Report, Volume 10, Number 4, March 25, 1996.

[10] Brian Case: "Java Performance Advancing Rapidly *Just-In-Time Compilers Show Java Can Compete with Compiled Code*", MicroDesign Resources, Micro Processor Report, Volume 10.

[11] Mark Lentczner, Glyphic Technology: "Java's Virtual World *Java Components Include High-Level Language and Virtual Machine*", MicroDesign Resources, Micro Processor Report, Volume 10, Number 4, March 25, 1996.