東海大学 工学部 通信工学科 1996 年度 卒業研究論文

Java Virtual Machine 命令互換 ロセッサ　高速演算回路　研究

東海大学 工学部通信工学科

30ET3114　黒子 周作

指導教員　清水 尚彦 講師

目 次

Abstract

Java は異機種間ネットワークによる分散環境のもとで、アプリケーション開発作業を支援する目的で Sun Microsystems 社によって設計された。そして、プログラムが特定のマイクロプロセッサー向けにコンパイルされず、「Java Virtual Machine（JVM）」で定義された中間コードに変換することによって、JVM が動く異なるプラットフォームで実行可能であるという特徴を持つ。この JVM の実装方法の一つに命令実行をハードウェアで備える Javachip がある。本稿では、この Javachip をハードウェア記述言語である SFL を用いて実装するとともに、主に演算関連の高速化について検討を行った。

Chapter 1

序論

パルテノン研究会主催　第３回 ASIC デザインコンテストにおいて、東海大学工学部通信工学科第７研究室のメンバーである黒子・青柳・榎田・河野の４名によって Javachip の実現方法を研究し「Java Virtual Machine 命令互換を持つプロセッサの設計」（Javachip Version1）として発表した。しかし、ハードウェアの完成を優先したために、高速化についてはほとんど検討しておらず改善するべき点がたくさんあった。本研究では、より幅広い Java Virtual Machine 命令互換のサポート・浮動小数点演算のサポート・算術演算ユニット（ALU）の高速化によって、より高性能な Javachip の実現を目差した。
以下に高速化について簡単な概要を示す。

1.1　算術演算ユニットの高速化に関して

Javachip Version1 では加算器に順次桁上げ方式（Ripple Carry Adder）を用いたために遅延時間がかなりおおきくなっていた。よって、加算器をより高速な回路に変更するだけでもクロック周波数の向上が望めるため、高速手法の１つである桁上げ先見方式（Carry-Lookahead Adder）を用いることによって桁上げの高速化、式の複雑性を抑えハードウェアの単純化を図った。

1.2　浮動小数点演算に関して

Javachip Version1 でサポートされていなかった浮動小数点演算をハードウェアで実装し、内部で使用するシフタにバレルシフタ、乗算に Booth のアルゴリズムを用いてさらなる互換性・高速化を図った。
以下に、今回実装したものを示す。

1. 浮動小数点加算
2. 浮動小数点乗算

3. 浮動小数点演算関連の型変換命令

Chapter 2

Javaについて

2.1 Javachip

Javaは Sun Microsystems 社によって開発された。Javaの重要な要素として Java 言語と Java Virtual Machine（JVM）がある。Java 言語は C++に似たオブジェクト指向プログラミング言語で、信頼性の高いソフトウェア生成と異機種間ネットワーク環境に展開するアプリケーションをサポートすることを目的に設計されており、アプリケーションは様々なハードウェア・アーキテクチャ上で実行可能でなければならない。そのため Java で書かれたコードは、Java コンパイラでバイトコードと呼ばれるアーキテクチャに依存しない中間コードに変換され、特定のハードウェア／ソフトウェア・プラットフォームに実装された JVM 上で実行することによってコンピュータの命令セット・アーキテクチャや OS に依存せずに実行することができる。

JVM の実装方法として、

- 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（Javachip）

などが挙げられる。
しかし、「インタプリタ」の JVM は命令をソフトウェアでフェッチし、デコードし、実行するというループを繰り返すため実行性能がプロセッサ本来の性能よりかなり低くなる。そこで、実行性能を高めるために Java のバイト・コードを実行する前にプロセッサー向けのバイナリー・コードにコンパイルし実行する「ジャスト・イン・タイム・コンパイラ」と、ハードウェアで実行する「Javachip」という方法がある。（Figure 2.1）

現在 Sun Microsystems 社は組み込み用途に３種類の Javachip を開発している。（Table 2.1）これらはまだ概要しかわかってないが、JVM に互換性を持つプロセッサであり pico

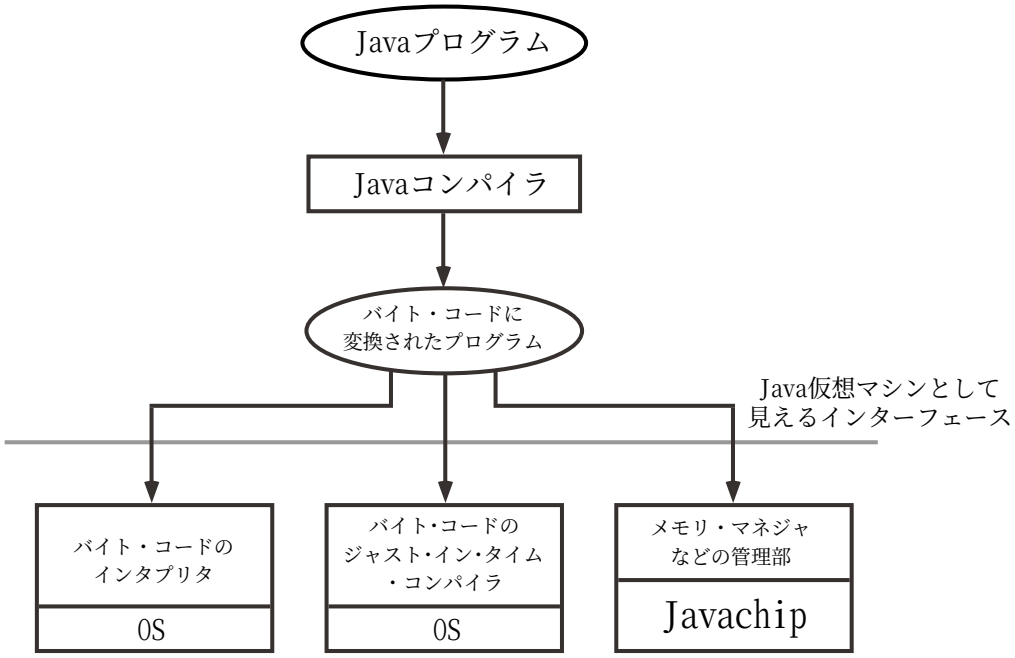

Figure 2.1: Java Virtual Machine

JAVA の場合その性能はジャスト・イン・タイム・コンパイラ使用した同クロックの Pentium より 5 倍の性能になると言われている。

pico JAVA の主な特徴

- RISC ライクなアーキテクチャ
- マイクロコード
- シンプルな 4 ステージパイプライン
- ハードウェアスタック

2.2 Java Virtual Machine の概要

Sun Microsystems 社が提供している資料「The Java Virtual Machine Specification」[3] と、日経エレクトロニクスで特集された「Javachip のすべて」[6] を参考に、JVM のアーキテクチャについて簡単に説明する。

Table 2.1: Javachip

開発コード名	pico JAVA	micro JAVA	Ultra JAVA
主な機能	整数演算ユニット FPU、キャッシュ	整数演算ユニット FPU、キャッシュ メモリ・コントローラ バス・インタフェース 入出力制御回路	整数演算ユニット FPU、キャッシュ メモリ・コントローラ マルチメディア 命令処理ユニット

2.2.1 JVM のデータ型

Java 言語はオブジェクト指向言語であり、プログラムの実行時にならないとデータ型を決められない。JVM では Table 2.2に示すデータ型が定義されており、それぞれのデータ型のオペランドに対して、ロード・ストアあるいは算術／論理演算が行われる。JVM のデータ型では、整数の long 型と浮動小数点の double 型が 64bit であり、符号なし型やポインタ型は定義されていない。浮動小数点形式は IEEE754 浮動小数点規格 (IEEE754 floating-point standard) に準拠した数値表現を採用している。

JVM はデータの処理や演算などをすべてスタックを介して行うスタックマシンであり、スタックは32bit 幅である。よって、64bit の long/double 型の値を格納するためには、64bit の値を 2 つの 32bit の値に分解して格納する。8bit/16bit の値の場合は、32bit に拡張して格納する。

Table 2.2: JVM のデータ型

データ型	長さ (byte)	説明
byte	1	符号付き整数（2 の補数表現）
short	2	符号付き整数（2 の補数表現）
int	4	符号付き整数（2 の補数表現）
long	8	符号付き整数（2 の補数表現）
float	4	IEEE754 フォマットの単精度浮動小数点
double	8	IEEE754 フォマットの倍精度浮動小数点
char	2	文字列（符号なしユニコード）

2.2.2 JVM の命令セット

JVM は、バイトコードなどを含んだ class ファイルと呼ばれる実行形式のファイルを実行する。このバイトコードは JVM で規定された 203 個の命令によって構成されており、記

憶装置にはこれらの命令が単純にバイトごとに並べられた形で格納されている。Table 2.3 に JVM の命令セットを示す。

Table 2.3: JVM の命令セット

種類	命令数		
定数プッシュ命令	20	ロード／ストア命令	52
スタック命令	10	数値演算命令	24
論理演算命令	12	分岐命令	22
比較命令	5	例外処理命令	1
配列管理命令	20	リターン命令	7
スイッチ命令	2	型変換命令	15
スレッド制御命令	2	フィールド操作命令	4
メソッド呼びだし	4	その他	3
合計		203	

Chapter 3

Javachip Version1

我々はJavaチップについて研究しパルテノンによる設計を試み「Java Virtual Machine命令互換を持つプロセッサの設計」(Javachip Version1)として発表した。このプロセッサは、Table 3.1に示すようにJVM互換命令62種類、独自命令7種類の合計69種類の命令を実装している。その他の未実装命令については、ソフトウェアのエミュレーションによって行われる。Javachip Version1の論理合成結果をTable 3.2に示す。

我々は、これらの結果を参考に高速化について検討した。そして、以下のような特徴を持つJavachip Version2を共同研究者である青柳氏が開発し「TRAJA (Tokai Research Approach for JVM Architecture)」として研究・開発している。

1. パイプライン
2. スタックキャッシュ
3. 命令セット
4. エミュレーションへの対応

Table 3.1: Javachip Version1 で実装した命令

種類	命令数
スタック管理命令	20
演算命令	20
型変換命令	5
分岐命令	16
比較命令	1
独自命令	7
合計	69

Table 3.2: Javachip Version1 の論理合成結果

ゲート数		12158
面積	[Kμm^2]	10180
消費電力	[μW/Hz]	42561.3
最大遅延時間	[ns]	493.718
クロック周波数	[MHz]	2.025

Chapter 4

算術演算ユニットの高速化

4.1 Javachip Version1 での加算器

Javachip Version1 では、Figure 4.1の順次桁上げ加算器（ripple carry adder）を使用していた。これは、キャリーインが前隣の 1bit 加算器からの結果を待つために 64 個の 1bit 加算器を順々に処理しないと全体の和が得られない。よって、キャリーの伝播で遅延時間が増大してしまった。加算器関連を除いた論理合成結果（Table 4.1）と比較すると、加算器関連で約 400[ns] の遅延時間になっている事がわかる。そこで、遅延時間を短縮するために桁上げの処理を高速化した。その方法として桁上げ保存法・桁上げ飛越し法などがあるが、比較的容易に記述でき高速な加算ができる桁上げ先見方式（carry-lookahead adder）を使用し、桁上げの高速化、式の複雑性を抑えハードウェアの単純化を図った。

Table 4.1: 加算器関連を除いた論理合成結果

ゲート数	9281
最大遅延時間 [ns]	107
クロック周波数 [MHz]	9.36

4.2 桁上げ先見加算器

桁上げ先見法とは、下の論理式 4.1で各 bit のキャリーを求めることにより、データ ab と各 bit のキャリー情報の排他的論理和を求めることによって加算する方法である。この方法を用いれば、データ ab が入力された時点で任意のキャリーを求めることが出来るためキャリーの伝播で待たされることがなく処理を高速化する事が出来る。

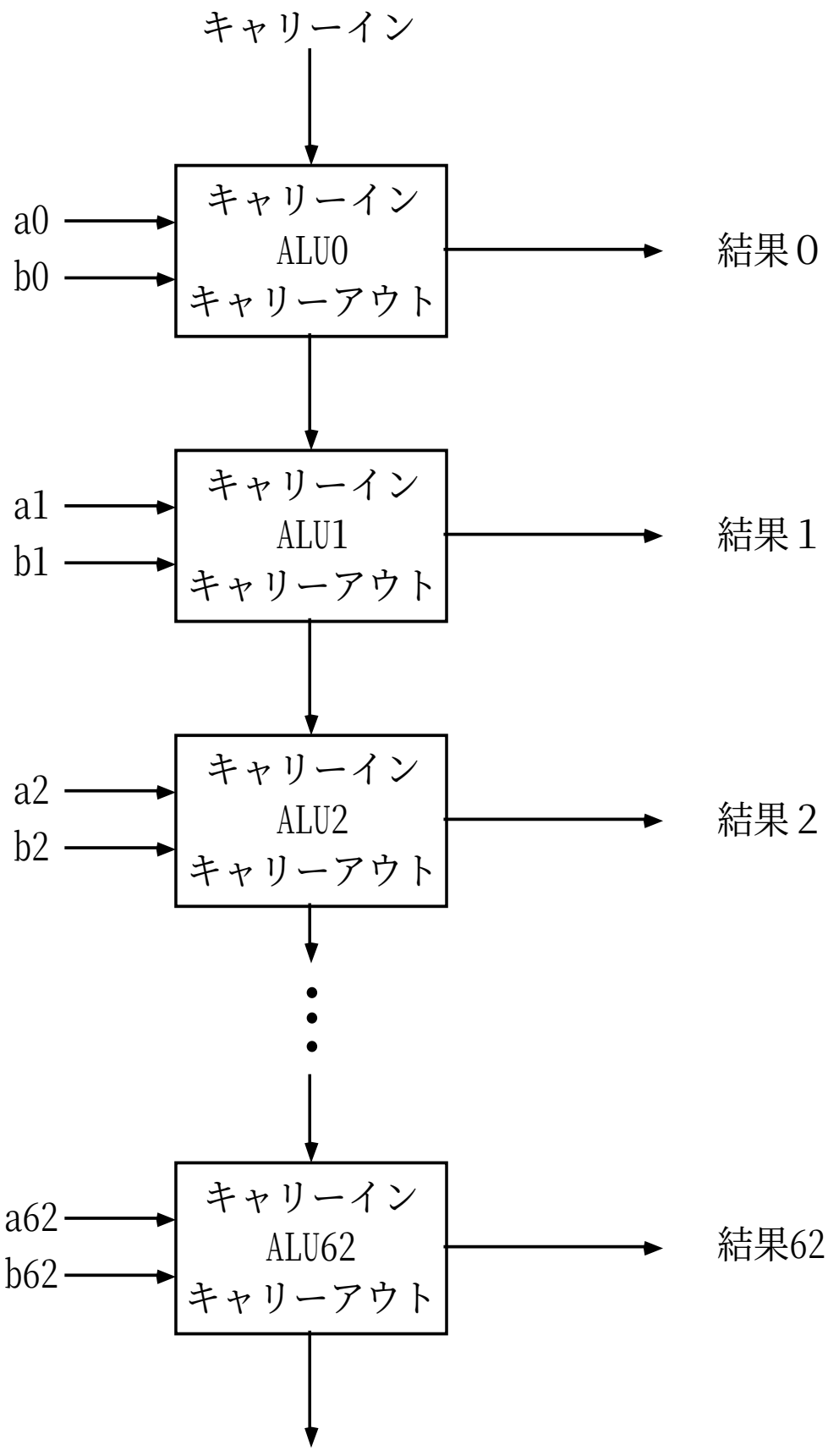

Figure 4.1: 順次桁上げ加算機

Figure 4.2に c_3の桁上げ先見出力回路を示す。

論理式

$$
\begin{array}{lcllll}
c_0 & = & a_0b_0 & + & c_{in} & (a_0+b_0) \\
c_1 & = & a_1b_1 & + & a_0b_0 & (a_1+b_1) \\
 & & & + & c_{in} & (a_0+b_0)(a_1+b_1) \\
c_2 & = & a_2b_2 & + & a_1b_1 & (a_2+b_2) \\
 & & & + & a_0b_0 & (a_1+b_1)(a_2+b_2) \\
 & & & + & c_{in} & (a_0+b_0)(a_1+b_1)(a_2+b_2) \\
c_3 & = & a_3b_3 & + & a_2b_2 & (a_3+b_3) \\
 & & & + & a_1b_1 & (a_2+b_2)(a_3+b_3) \\
 & & & + & a_0b_0 & (a_1+b_1)(a_2+b_2)(a_3+b_3) \\
 & & & + & c_{in} & (a_0+b_0)(a_1+b_1)(a_2+b_2)(a_3+b_3) \\
 & & & & & \vdots \\
 & & & & & \vdots
\end{array}
\tag{4.1}
$$

4.3 JavaALU

Javachip の算術論理演算命令を処理する回路を、JavaALU として作成した。JavaALU では、算術論理演算命令として Table 4.2に示す命令を持っており、入力端子の「mode」によって、それぞれの命令を実行する。

Table 4.2: JavaALU の算術論理演算命令

命令	mode	SFLでの記述方法
SBC	000	$a = a + \hat{}\,b + \hat{}\,c_{in}$
ADC	001	$a = a + b + c_{in}$
SUB	010	$a = a + \hat{}\,b + 0b1$
ADD	011	$a = a + b$
EOR	100	$a = a \,@\, b$
OR	101	$a = a \mid b$
AND	110	$a = a \,\&\, b$
CMP	111	$a + \hat{}\,b + 0b1$

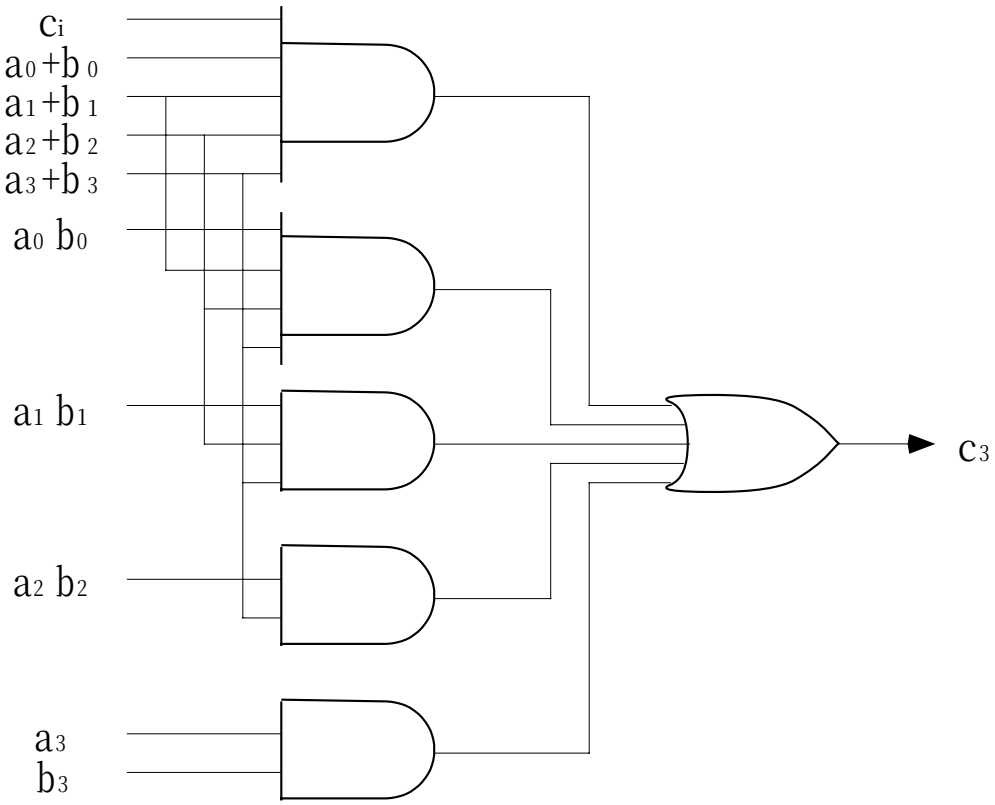

Figure 4.2: 桁上げ先見出力回路

JavaALU では、桁上げ先見方式を用いた加算器を使用しており、Figure 4.3のように 4bit 単位の桁上げ先見加算器 16 個、16bit 単位の桁上げ先見加算器 4 個を用いて 64bit の加算が出来るようになっている。これらの桁上げ先見加算器は並列に動作するため、高速に加算することが出来る。

論理合成の結果は Table 4.3のようになり、Javachip Version1 の加算器と比べると半分以下に遅延時間を短縮することができた。

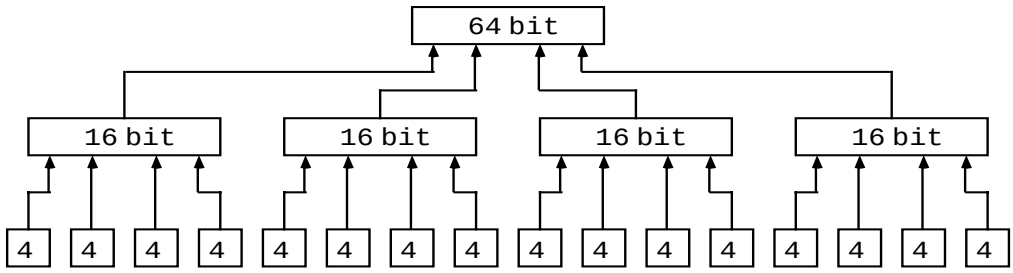

Figure 4.3: 桁上げ先見加算器

Table 4.3: 桁上げ先見加算器の論理合成結果

ゲート数		1769
面積	[Kμm^2]	1494.84
消費電力	[μW/Hz]	6870.2
最大遅延時間	[ns]	130

4.4 SFL 言語での実装

JavaALU を SFL 言語で記述した際、64bit の加算器を用いるため各 bit のキャリーを式 (4.1) で求めようとすると相当な長さになってしまう。そこで、式 4.2のように生成 g_iと伝播 p_iの２つの共通項に分けて簡素化した。しかし、それでも相当な長さになってしまうため、4bit 桁上げ先見加算器を単一の構成要素としたモジュールを用意した。モジュールとは SFL 言語における記述の基本単位であり、階層化の単位となっている。モジュールは、他のモジュールを使うことができ、階層表現をする事ができるので、これらの加算器を対象にした桁上げ先見加算器を作成することによって階層化し、また抽象化することによって記述が容易になった。また、加算器は他のモジュールでも頻繁に用いるため、必要な bit

数の加算器を容易に作成できた方がよい。4bit 桁上げ先見加算器をモジュールとして作成した事によって、16bit の加算器を作成する場合は、4bit の加算器 4 個、32bit の加算器を作成する場合は、4bit の加算器 8 個・16bit の加算器 2 個といったように流用することができ、容易に必要な bit 数の加算器を作ることができる。

$$
\begin{aligned}
g_i &= ai \cdot bi \\
p_i &= ai + bi
\end{aligned}
\qquad\qquad
\begin{aligned}
c_0 &= g_0 + (c_{in} \cdot p_0) \\
c_1 &= g_1 + (g_0 \cdot p_1) \\
&\quad + (c_{in} \cdot p_0 \cdot p_1) \\
c_2 &= g_2 + (g_1 \cdot p_2) \\
&\quad + (g_0 \cdot p_1 \cdot p_2) \\
&\quad + (c_{in} \cdot p_0 \cdot p_1 \cdot p_2) \\
c_3 &= g_3 + (g_2 \cdot p_3) \\
&\quad + (g_1 \cdot p_2 \cdot p_3) \\
&\quad + (g_0 \cdot p_1 \cdot p_2 \cdot p_3) \\
&\quad + (c_{in} \cdot p_0 \cdot p_1 \cdot p_2 \cdot p_3)
\end{aligned}
\tag{4.2}
$$

Chapter 5

浮動小数点演算

Javachip Version1 では浮動小数点演算がサポートされていなかった。よって浮動小数点演算のうち加減算と乗算について研究し実装を試みた。また浮動小数点演算関連の型変換命令についても数種類、実装している。

5.1 浮動小数点加減算

5.1.1 浮動小数点加減算のアルゴリズム

浮動小数点の加減算は次のような操作からなる、

1. 2 つの指数を比較し小さい方の仮数を右にシフトして、大きい方と指数を一致させる
2. 仮数を加える
3. 和を正規化し、必要な数だけ右、または左にシフトする
4. 丸めの処理を行う
5. 正規化数でないなら 3 にもどり正規化を行う

Figure 5.1に浮動小数点数加算回路の基本構成図を示す。
最初に ALU によって一方の演算対象の指数が他方の演算対象の指数から引かれて、どちらがどれだけ大きいかが判定される。その差によって、3 つのマルチプレクサが制御される。左から、大きい方の指数の選択、小さい方の数値の仮数の選択、大きい方の数値の仮数の選択となり、小さい方の仮数は右にシフトされる。それから ALU によって 2 つの仮数が加算され、正規化するため和が左または右にシフトされて指数が増減される。最後に丸めの処理をして結果が得られる。

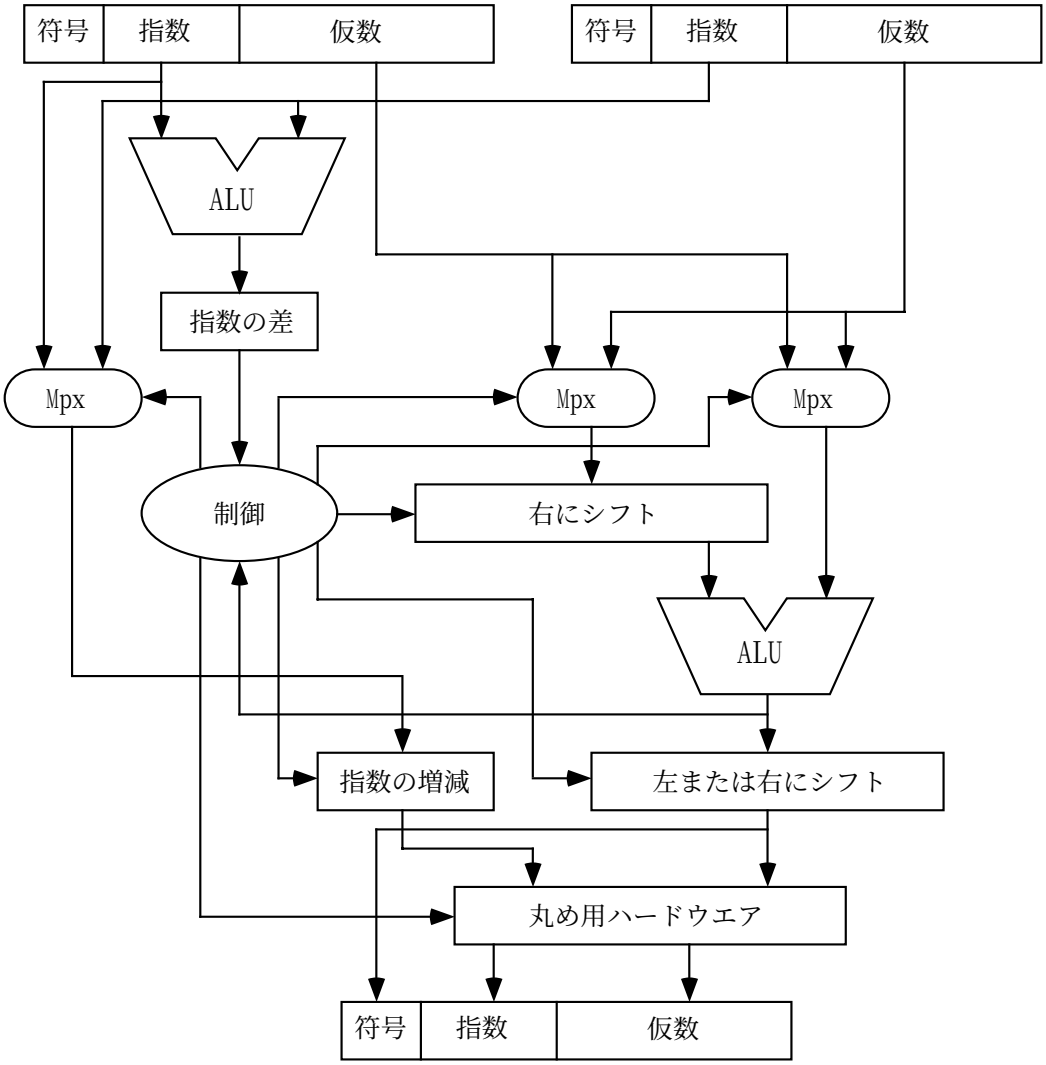

Figure 5.1: 浮動小数点数加算回路の基本構成図

5.1.2 Javachiv 用浮動小数点加減算回路

本演算回路は次のような仕様になっている。

1. IEEE754 標準規格に準拠した数値表現を採用
2. モジュールが取り扱う浮動小数点数の精度は倍密度 64bit（Figure 5.2）
   （符号 1bit・指数部 11bit・仮数 52bit）
3. モジュール内部では別の数値表現に変換（Figure 5.2）
   （符号 1bit・桁上げ用 1bit・暗黙の 1 1bit・仮数 52bit・ガード桁 9bit）
4. 指数のゲタは 1023（指数の最大値 1023・最小値-1022）
5. 入出力は正規化数
6. パイプライン化
7. シフト操作にバレルシフタを使用

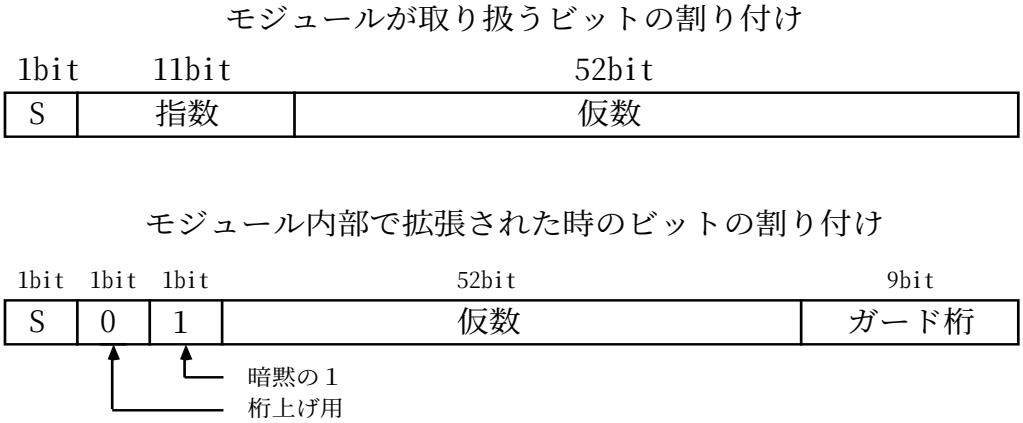

Figure 5.2: 浮動小数点数加減算回路の仕様

Figure 5.2のようにモジュールでのビット割付けが入出力時と内部演算時で違うのは、モジュール内部での演算のしやすさ、そしてガード桁を取ることによって演算結果をより正確な値に近付けるためである。

5.1.3 パイプライン

今回の浮動小数点数加算回路は 4 ステージのパイプライン構造になっていおり、Figure 5.3のような流れになっている。

ステージの構成

ステージ 1

- モジュール内部に用いる数値表現に変換
- 仮数と被仮数の指数の差を求める
- 大きい方の数を指数を仮の指数とする
- 指数が小さい方の仮数を指数の差だけ右にシフトする

ステージ 2

- 仮数の加減算を行う
- 答えの正負の符号を定める

ステージ 3

- 演算結果の仮数の正規化に必要なシフト数を求める
- 仮数をシフトするとともに丸めを行う
- シフト数だけ仮の指数を補正する

ステージ 4

- 出力

5.1.4 バレルシフタ

浮動小数点の加減算と乗算では、桁合わせのために任意のビット数のシフト操作が必要になる。Javachip Version1 で使用していたシフト命令は、1bit 単位のシフトしか出来ないシフタを使用していたためにシフト量に比例して遅くなっていた。これを高速に行うのが Figure 5.4のバレルシフタと呼ばれる回路である。これはビットのシフトを並列に行う回路を多段に重ねて構成されており、各段では制御信号により 2^n ビットのシフトを行うのか行わないのかを選択する。６段の重ねの回路を用いれば 1〜63bit までの任意の bit 数のシフトを一括しておこなうこどが出来るので、このバレルシフタを用いる事によって多機能化及び高速化を試みた。
Table 5.1に論理合成結果を示す。

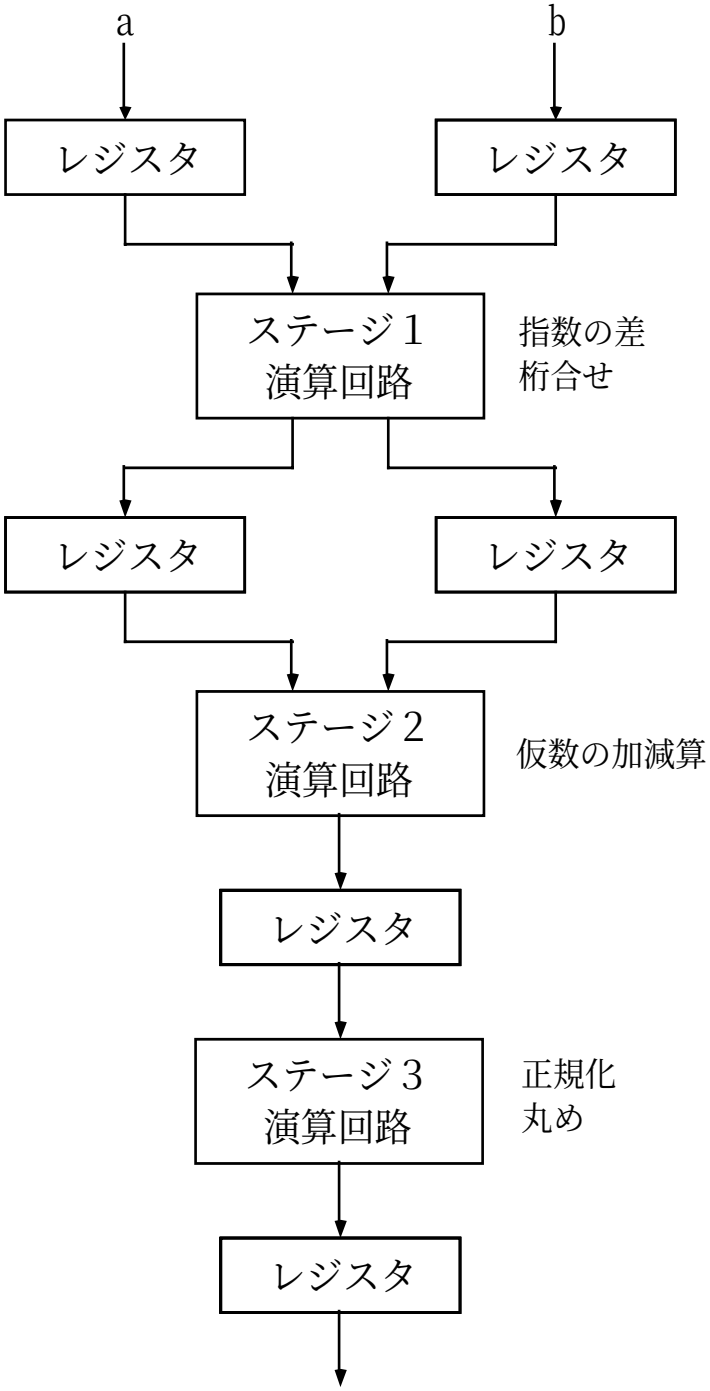

Figure 5.3: パイプライン浮動小数点数加減算回路の流れ

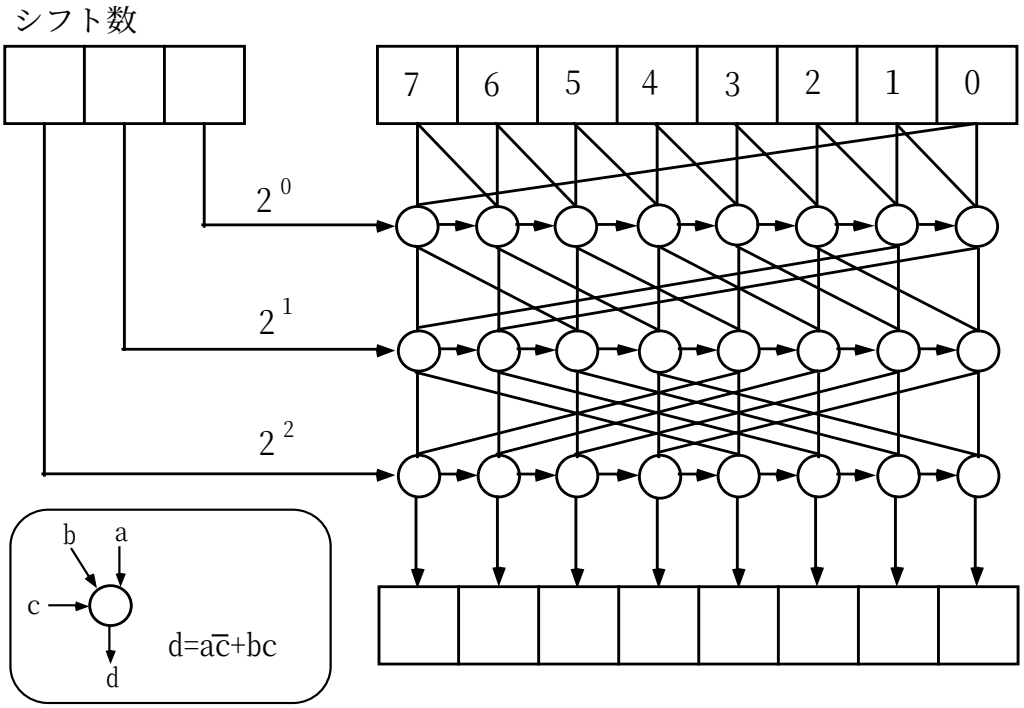

Figure 5.4: バレルシフタ

Table 5.1: バレルシフタの論理合成結果

ゲート数		463
面積	[Kμm^2]	440.156
消費電力	[μW/Hz]	2044.9
最大遅延時間	[ns]	37.524

5.2 浮動小数点乗算

5.2.1 浮動小数点加減算のアルゴリズム

浮動小数点の乗算は次のような操作からなる、

1. 2 つの指数を加え、その和からゲタを引いて指数を求める
2. 仮数を掛ける
3. 積を正規化して、積を右にシフトし指数を増やす
4. 丸めの処理を行う
5. 正規化数でないなら 3 にもどり正規化を行う

5.2.2 Javachiv 用浮動小数点乗算回路

本演算回路は次のような仕様になっている。

1. IEEE754 標準規格に準拠した数値表現を採用
2. モジュールが取り扱う浮動小数点数の精度は倍密度 64bit（Figure 5.2）
   （符号 1bit・指数部 11bit・仮数 52bit）
3. 指数のゲタは 1023（指数の最大値 1023・最小値-1022）
4. 入出力は正規化数
5. パイプライン化
6. 乗算器に Booth のアルゴリズムを使用

5.2.3 パイプライン

今回の浮動小数点数乗算回路は 4 ステージのパイプライン構造になっていおり、Figure 5.5のような流れになっている。

ステージの構成

ステージ 1

- モジュール内部に用いる数値表現に変換
- 2 の指数を加え、その和からゲタを引いて指数を求める
- 答えの正負の符号を求める

ステージ 2

- Booth のアルゴリズムで仮数の乗算を行う

ステージ 3

- 結果に応じて仮数を 1 ビットシフトするとともに丸めを行う
- 結果に応じて指数を 1 だけ補正する

ステージ 4

- 出力

5.2.4 Booth のアルゴリズム

符号付きの数値を高速に乗算する方法の一つに Booth のアルゴリズムがある。これは、加算よりもシフトの方が速く処理できるという考えの元に考案され、数値のパターンによって速く乗算する事が出来る。Booth のアルゴリズムでは、Table 5.2のように乗数の 2 つのビットを見て処理を行う。そして、右にシフトし同様に最後まで繰り返す。

Table 5.2: Booth のアルゴリズム

左のビットの値	左のビットの値	処理	例
1	0	何も行わない	001**10**
1	1	被乗数を積の左半分に加える	00**11**0
0	1	被乗数を積の左半分に引く	0**01**10
0	0	何も行わない	**00**110

5.3 SFL 言語での実装

Javachip の浮動小数点数加減算を処理する回路を、fp_adder として作成し、浮動小数点数乗算を処理する回路を、fp_multi として作成した。負の数値の処理において、fp_adder は IEEE754 標準規格に準拠した数値表現を採用しているので、負の数値を表す場合は最上位ビット (MSB) を 1 にしなければならない。仮数部の加減算を行う場合、加算器は負の数値を 2 の補数で表さなければならないので、そのままでは計算できない。よって、まず正負の符号を判別し、負であれば全ビットを反転し 1 を加えて 2 の補数表現に変換する。符号が正であれば、そのまま元の数値を使い加減算の処理を行っている。fp_multi においては、

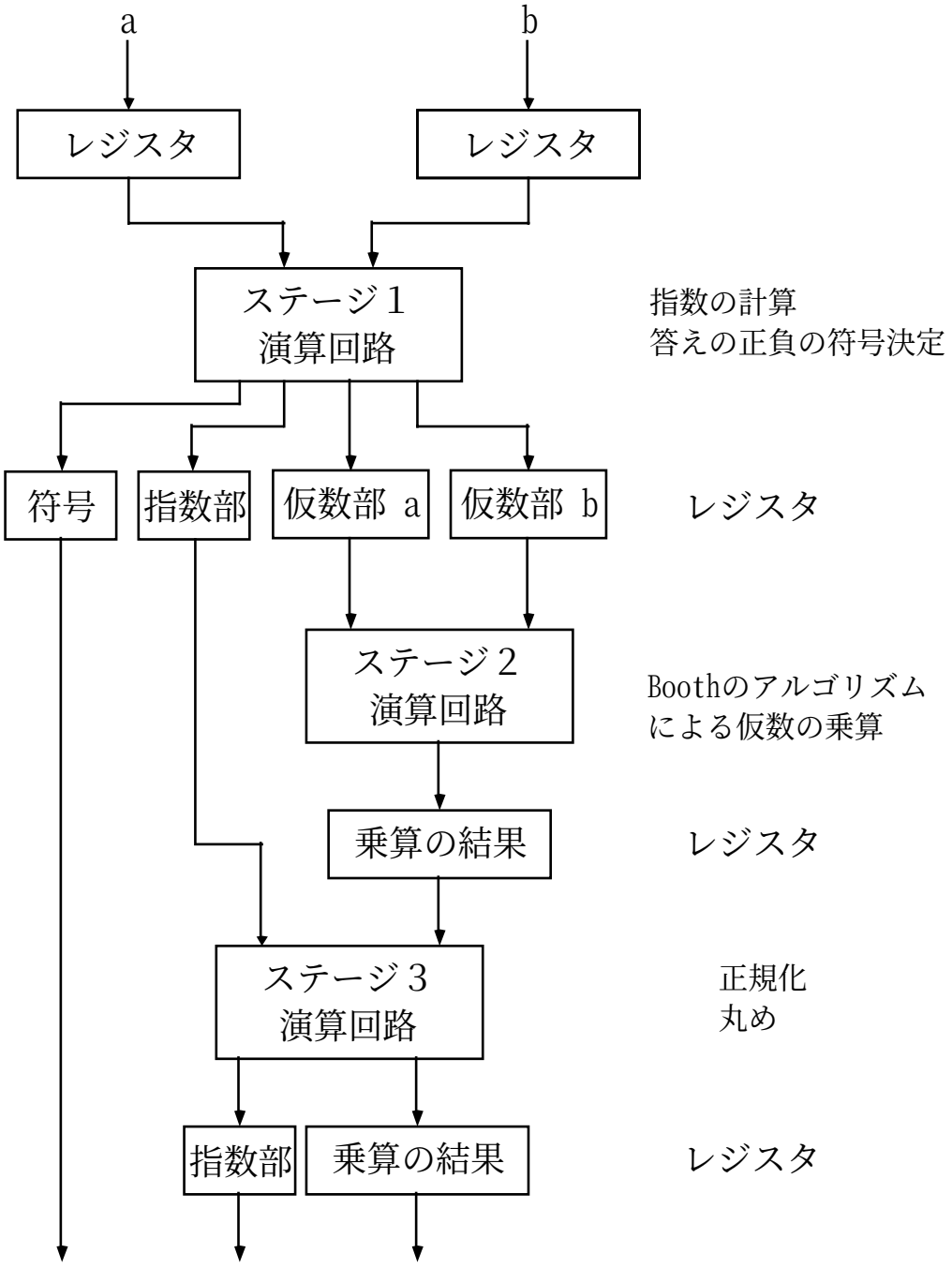

Figure 5.5: パイプライン浮動小数点数乗算回路の流れ

乗数と被乗数の符号が異なる場合のみ負の数値となるため、そのまま仮数部の乗算を行うことができる。

SFL 言語には状態遷移を容易に記述をできる「ステージ」や「タスク」の構文があり、浮動小数点数加減算・乗算のパイプラインの動作はこれらの構文を用いて作成している。ステージの移動については、SFL 言語での「relay」文を用いて、ジョブを転送して行っている。ステージ内には複数の状態と状態遷移を記述できるため、必ずしも 1 ステージが１クロックで終了するとは限らない。

Chapter 6

まとめ

本論文では、Java Virtual Machine 命令互換を持つ Javachip について研究し、Javachip Version1 で問題だった算術演算回路の高速化・そして、浮動小数点演算回路の実装を試みた。算術演算回路に関しては、加算器において問題となる桁上げ処理を高速化するために、桁上げ先見方式（carry-lookahead adder）を用いて遅延時間の短縮を図った。その結果、大幅な高速化がおこなえた。これにより、算術演算回路以外で用いる加算器においても高速化することができ、クロック周波数を向上する事が出来た。しかし、まだ高速化について改良すべき点があり、さらに複雑な加算方式を用いることによって高速化できると思われる。このような加算方式については、様々な論文で発表されている。

浮動小数点演算回路部分に関しては、完成を優先したために高速化についてはほとんど検討できなかった。しかし、パイプライン化することにより単位時間当たりの処理量が増え、浮動小数点演算の多い計算の場合を考えれば有効な高速化となった。また、バレルシフタ・Booth のアルゴリズムを用いて高速化について試みたが、まだ高速化の余地が残っている。たとえば、乗算の高速化手法の 1 つである可変長桁移動方式 (variable length shift method) を用いることによって、さらなる高速化が望めると思える。

Bibliography

[1] David A.Patterson/John L.Hennessy. コンピュータの構成と設計. 上. 日経 BP 社, 4 月 19 日 1996. (翻訳：成田光影).

[2] David A.Patterson/John L.Hennessy. コンピュータの構成と設計. 下. 日経 BP 社, 4 月 19 日 1996. (翻訳：成田光影).

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley Publishing Co., September 1996.

[4] 伊藤規之. デジタル回路. 日経理工出版.

[5] 清水賢資. デジタル情報回路. 森北出版.

[6] 北郷達郎. Javachip のすべて, pp. 155–185. 日経エレクトロニクス, No. 664. 日経 BP 社, 6 月 17 日 1996.