

# 卒業研究報告書

題目

## V P S Mの構築

指導教員

石水 隆 助手

---

報告者

02-1-47-014

西谷 政博

---

近畿大学工学部情報学科

平成 18 年 2 月 10 日提出

## 概要

並列アルゴリズムの設計およびその計算量の評価は多くの場合 PRAM(Parallel Random Access Machine) 上で行われる。PRAM は共有メモリ型並列計算モデルであり、個々の演算による実行時間の違いや通信や同期のコストを無視した単純なモデルであるため、アルゴリズムの設計および評価を行い易いためである。しかし、大規模なプロセッサでのメモリの共有化や、通信や同期の高速化には様々な問題があるので、PRAM 自体の実現は困難であるため、PRAM アルゴリズムの実験的な評価を行うために PRAM シミュレータが必要とされる。

PRAM アルゴリズムの実行をシミュレートする PRAM シミュレータは以下の 4 要素から成る

- (1) PRAM 用並列言語
- (2) 並列アセンブラ
- (3) PRAM コンパイラ
- (4) VPSM (Virtual Parallel Stack Machine)

PRAM 用言語は、C や JAVA のような人間にとって記述し易い高級言語であり、PRAM 上での並列処理を記述できる言語である。並列アセンブラは並列処理を記述できる低級言語である。PRAM コンパイラは PRAM 用並列言語を並列アセンブラに変換するコンパイラであり、VPSM は並列アセンブラの実行をシミュレートするインタプリタである。

PRAM シミュレータのユーザは、PRAM 用並列言語で PRAM 上での並列処理を記述し、PRAM コンパイラにより並列アセンブラに変換する。次に VPSM 上で並列アセンブラを実行することにより PRAM の動作をシミュレートできる。

本研究では、VPSM インタプリタを JAVA 言語を用いて設計を行った。本研究で設計した VPSM インタプリタは、並列アセンブラの実行をシミュレートすることができる。また、PRAM 用並列言語を PRAM 上で実行させたときの実行時間を出力する機能を持ち、PRAM アルゴリズムの計算量を実験的に評価することができる。

# 目次

1	序論	1
1.1	並列アルゴリズム	1
1.2	並列計算機	1
1.3	並列計算モデル	1
1.4	PRAM シミュレータ	1
1.5	本報告書の構成	2
2	準備	3
2.1	PRAM (Parallel Random Access Machine)	3
2.2	PRAM シミュレータ	4
3	PRAM シミュレータ	6
3.1	PRAM 用並列言語	6
3.2	並列アセンブラ	6
3.3	VPSM(Virtual Parallel Stack Machine) インタプリタ	6
4	結果および考察	9
5	結論および今後の課題	9
6	謝辞	10
付録 A	付録	12
A.1	並列アセンブラの文法	12
A.2	VPSM インタプリタプログラム	16
A.3	$n$ 個のデータの和を計算する PRAM 用並列言語プログラム	57
A.4	付録 3 を PRAM コンパイラによって変換した並列アセンブラプログラム	59

# 1 序論

## 1.1 並列アルゴリズム

地球規模の気象シミュレーションや天体の軌道計算など、計算量の大きな問題を短時間で解く必要のある分野は多岐に渡っている。これらの問題に対して、従来の1台のプロセッサから成る逐次計算機を用いた逐次処理では非常に大きな時間が掛かる。このため、これらの問題を解く手法として、複数のプロセッサを持つ並列計算機 (Parallel Computer) による並列処理 (Parallel Processing) が現在注目されている。複数のプロセッサが協調してデータを処理することにより、問題を短時間で解け、またより複雑な問題を解くことができるようになる。しかし、並列処理を行うためには、プロセッサ間のデータのやり取りや複数のプロセッサによるメモリへの同時アクセス、プロセッサ間の同期等、並列特有の問題を解決せねばならない。このため、従来の逐次処理で用いられてきた逐次アルゴリズムをそのまま並列処理に用いることはできず、並列処理専用のアルゴリズム、すなわち並列アルゴリズム (Parallel Algorithm) が必要となる。そのため、現在様々な分野で、高速に処理を行う並列アルゴリズムが求められている。

## 1.2 並列計算機

並列計算機は複数のプロセッサを持ち並列処理を行うことができる計算機である。並列計算機は、全てのプロセッサが共通したメモリに対して読み書きを行い、プロセッサ間の通信はメモリを通して行う共有メモリ型並列計算機 (Shared Memory Parallel Computer) と、それぞれのプロセッサが局所メモリを持ち、プロセッサ間の通信はネットワークを通じて行う分散メモリ型並列計算機 (Distributed Memory Parallel Computer) に大別される。共有メモリ型計算機はプロセッサ間の通信を高速に行うことができ、プロセッサ間での同期も取り易いため、通信および同期にかかるコストを気にせずに高速化を得ることができる。一般的にプロセッサ数の増加に従い、1つの共有メモリに全てのプロセッサを繋ぐことは困難となる。このため、現在、プロセッサ数の多い並列計算機では分散メモリ型が主流となっている。また、複数の計算機をネットワークで繋ぎ、それ全体を仮想的な計算機として扱うクラスター (Cluster) 処理やグリッド (Grid) 処理も幅広く行われている。

## 1.3 並列計算モデル

並列アルゴリズムの設計およびその計算量の評価は多くの場合 PRAM (Parallel Random Access Machine) 上で行われる。

PRAM は共有メモリ型並列計算モデルであり、全ての演算がその種類に関係なく1単位時間で行われる、1命令毎に同期が取られる、通信のコストが一切発生しない、等の仮定が設けられた理想的なモデルである。PRAM は個々の演算による実行時間の違いや通信や同期のコストを無視した単純なモデルであるため、PRAM 上ではアルゴリズムの設計および評価を比較的容易に行うことができる。しかし、大規模なプロセッサでのメモリの共有化や、通信や同期の高速化には様々な問題があるため、PRAM 自体の実現は困難である。

## 1.4 PRAM シミュレータ

多くの並列アルゴリズムは、PRAM 上で設計・解析が行われる。しかし、PRAM 自体の実現は困難であるため、PRAM アルゴリズムの正当性および時間計算量を実験的に評価するのは容易ではない。そのため、

PRAM アルゴリズムの実験的な評価を行うために PRAM シミュレータ (PRAM Simulator) が必要となる。

PRAM シミュレータは、高級言語である PRAM 用並列言語を低級言語である並列アセンブラに変換する PRAM コンパイラと、並列アセンブラを実行する VPSM(Virtual Parallel Stack Machine) インタプリタから成る。

本研究では、JAVA 言語を用いて VPSM インタプリタの設計を行う。本研究で設計した VPSM インタプリタは、実行する並列アセンブラプログラムの原始プログラムである PRAM 用並列言語プログラムの PRAM 上での実行をシミュレートし、その結果を出力する。また、本研究で設計した VPSM インタプリタは、PRAM 用並列言語プログラムを PRAM 上で動作させた場合の実行時間を計測する機能を持ち、PRAM アルゴリズムの計算量を実験的に評価することができる。

## 1.5 本報告書の構成

本研究の構成を以下に述べる。2 章では本研究が対象とする並列計算モデルである PRAM および PRAM シミュレータを定義する。3 章では本研究で使った PRAM 用並列言語と、設計した VPSM インタプリタの説明をしている。4 章では本研究で設計した VPSM インタプリタが正しく PRAM アルゴリズムの動作をシミュレートしている事を検証する。5 章で本研究の結論を述べている。

## 2 準備

### 2.1 PRAM (Parallel Random Access Machine)

図 1 のように、複数のプロセッサがメモリを共有した並列計算機を共有メモリ型並列計算機 (Shared Memory Parallel Computer) という。共有メモリ型並列計算機を抽象化した並列計算機モデルが、並列ランダムアクセスマシン (PRAM, Parallel Random Access Machine) である。RAM と同様に PRAM では共有メモリの読み書きも含めて全ての命令は 1 単位時間で行われると仮定されている。また、1 命令ごとに全てのプロセッサで同期が取られる。共有メモリであるため PRAM アルゴリズムではデータの局所性を考える必要がない。またプロセッサ間の通信にかかるコストも無視されている。このため PRAM 上では並列アルゴリズムの設計開発を簡易に行うことができ、並列アルゴリズムの解析も行い易い。加えて、PRAM 上で設計したアルゴリズムは、多くの場合他の並列設計モデル上でも効率良く動かせる。このため、並列アルゴリズムの設計は多くの場合 PRAM 上で行われる。以下は PRAM のプロセッサ台数を  $p$  とし、各プロセッサを  $P_i (1 \leq i \leq p)$  と表す。

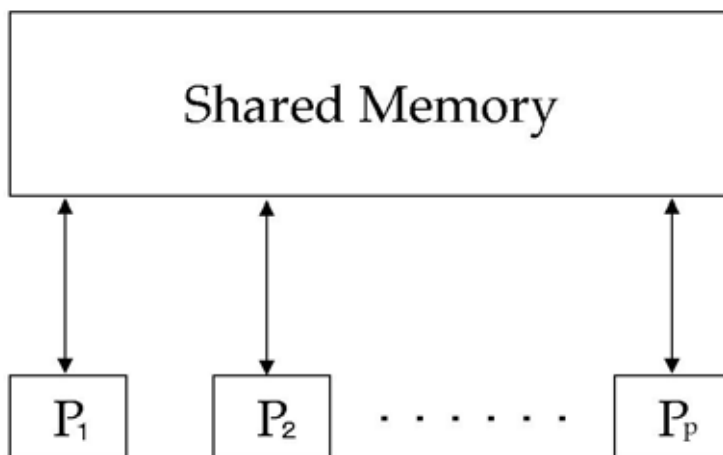


図 1 PRAM(Parallel Random Access Machine)

PRAM 上でのプロセッサ間の通信は、共有メモリ (SM, Shared Memory) を通して行う。プロセッサ  $P_i$  がプロセッサ  $P_j$  にある数値を送りたい場合、これは 2 ステップで実行される。まず、プロセッサ  $P_i$  が、プロセッサ  $P_j$  の知っている所定の位置の共有メモリに、その数値を書き込む。その後、プロセッサ  $P_j$  がその位置から先ほどの数値を読み出せばよい。並列アルゴリズムの実行中、 $p$  個のプロセッサは、入力データを読んだり、中間結果を読んだり書いたり、最終結果を書いたりするために、共有メモリにアクセスする。PRAM ではプロセッサが読み出したり書き込んだりするメモリ位置が異なっているならば、全てのプロセッサが共有メモリに同時にアクセスできる。一方、二つ以上のプロセッサが同じメモリ位置に同時にアクセスする場合は、それができかどうかによって、PRAM は、次のような 4 種類に分類される。

- EREW(Exclusive-Read Exclusive-Write)PRAM どの二つのプロセッサも同じメモリ位置から同時に読み出したり書き込んだりできない。

- CREW(Concurrent-Read Exclusive-Write)PRAM 複数のプロセッサが同じメモリ位置から読み出すことができるが、どの二つのプロセッサも同じメモリ位置に同時に書き込むことはできない。
- ERCW(Exclusive-Read Concurrent-Write)PRAM 複数のプロセッサが同じメモリ位置に書き込むことができるが、どの二つのプロセッサも同じメモリ位置に同時に読み込むことはできない。
- CRCW(Concurrent-Read Concurrent-Write)PRAM 複数のプロセッサが同じメモリ位置に読み出すことも書き込むこともできる。

メモリ内の同じアドレスに多重の読み出しの権利を認めることは、原理的に何の問題も引き起こさない。概念的には、その位置から読み出すいくつかのプロセッサは、そのメモリの内容のコピーを作り、各プロセッサの局所メモリに格納する。一方アドレスに対する書き込みは複数のプロセッサが異なる値を書き込もうとした時に、どのプロセッサを優先するかという問題が発生する。

この問題への対処法により、CRCW PRAM は以下のような3種に細分化される。

- 共通型 (Common)CRCW PRAM 書き込みを行おうとした全てのプロセッサが同じ値を書き込もうとした時のみ値が成立する。
- 任意型 (Arbitrary)CRCW PRAM 書き込みを行おうとしたプロセッサのどれか一つが書き込みに成功する。
- 優先型 (Priority)CRCW PRAM 書き込みを行おうとしたプロセッサのうち、最も優先順位が高いプロセッサの書き込みが成功する。

## 2.2 PRAM シミュレータ

PRAM シミュレータは PRAM 上での並列アルゴリズムの実行をシミュレートし、その実行結果を出力しおよび実行時間を計測する機能を持つプログラムである。PRAM シミュレータは以下の4要素から成る。

- PRAM 用並列言語
- 並列アセンブラ
- PRAM コンパイラ
- VPSM(Virtual Parallel Stack Machine) インタプリタ

PRAM 用並列言語は、JAVA や C 等の高級言語に並列処理を行うための命令を加えたものであり、並列アセンブラは並列処理を行うための命令を加えたアセンブラである。PRAM コンパイラは、PRAM 用並列言語で記述されたプログラムを並列アセンブラプログラムに変換するコンパイラであり、VPSM インタプリタは並列アセンブラプログラムを実行できるインタプリタである。図2に PRAM シミュレータの実行の流れを示す。ユーザは PRAM 用並列言語を用いて PRAM アルゴリズムを記述する。次に PRAM 用並列言語プログラムを PRAM コンパイラを用いて並列アセンブラプログラムに変換し、VPSM インタプリタを用いて並列アセンブラプログラムを実行する。

リスト1から3に PRAM シミュレータによる実行例を示す。図3は各プロセッサが、そのプロセッサ番号を出力する PRAM 用並列言語プログラムである。このプログラムを PRAM コンパイラによりコンパイルすると、図4に示す並列アセンブラプログラムが得られる。これを VPSM インタプリタにより実行すると図5の出力が得られる。

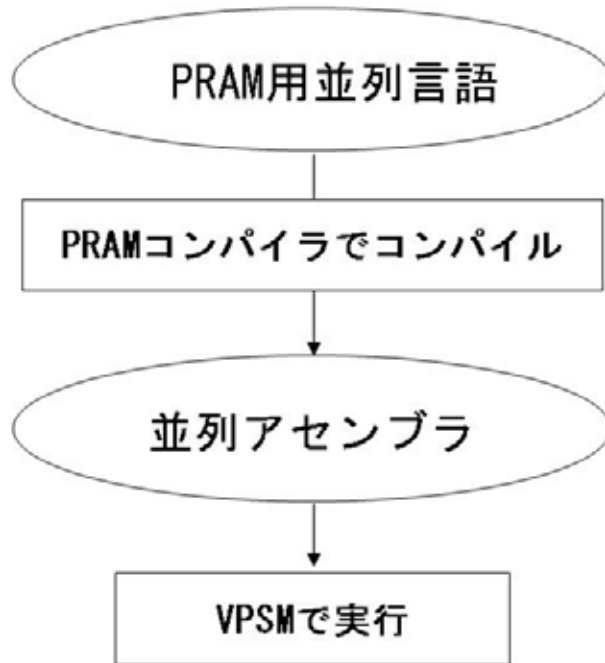


図2 PRAM シミュレータによる実行の流れ

```

main() {
  parallel (1, 10) {
    writeint ($ p);
  }
}
  
```

図3 PRAM 用並列言語プログラムの例

```

PUSHI 1
PUSHI 10
PARA
PARA
OUTPUT
SYNC
HALT
  
```

図4 並列アセンブラプログラムの例

1 2 3 4 5 6 7 8 9 10

図5 VPSM の出力結果



### 3 PRAM シミュレータ

#### 3.1 PRAM 用並列言語

本研究で使用する PRAM 用並列言語は、C 風の手続き型言語である。この言語は、KC05 言語 [1] に並列処理を行う `parallel` 文および実行中のプロセッサ番号を表す特殊変数  $\$p$  を加えたものである。`parallel` 文の文法は以下のように定義される

$$\text{parallel} ( \text{exp1} , \text{exp2} ) \text{ statement}$$

$\text{exp1}$  および  $\text{exp2}$  は `int` 型の評価値を持つ式であり、 $\text{statement}$  は `parallel` 文を含まない任意の文である。`parallel` 文を実行すると、プロセッサ番号  $\text{exp1}$  から  $\text{exp2}$  までのプロセッサが  $P_{\text{exp1}}, P_{\text{exp1}+1}, P_{\text{exp1}+2}, \dots, P_{\text{exp2}}$  が後に続く  $\text{statement}$  を並列に実行する。また、 $\text{statement}$  中に特殊変数  $\$p$  を記述すると  $\$p$  は実行中のプロセッサ番号の値を持つ変数として処理される。

PRAM 用並列言語プログラムは、プロセッサ  $P_0$  のみが命令を実行する逐次モードと、`parallel` 文により指定された複数のプロセッサが命令を実行する並列モードの 2 つのモードを持つ。プログラム開始時は逐次モードであり、`parallel` 命令中の  $\text{statement}$  は並列モードとして実行され、それ以外の命令は逐次モードとして実行される。

#### 3.2 並列アセンブラ

本研究で使用する並列アセンブラは、司令部オペレータと値部オペランドから成る 2 個組命令で構成される。この言語は、KC05 言語の目的言語として用いられるアセンブラ [1] の命令セットに以下の命令を加えたものである。

- **PUSHP** :プロセッサ番号。スタック挿入命令: スタックトップに命令を実行しているプロセッサのプロセッサ番号を挿入する
- **PARA** :並列開始命令。逐次モードから並列モードに移行する。逐次モード実行中に **PARA** 命令を読み込んだ場合、スタックから 2 個のデータ  $d_1, d_2$  が取り出される。この命令以降は、プロセッサ  $P_{d_1}, P_{d_1+1}, P_{d_1+2}, \dots, P_{d_2}$  が並列に命令を実行する。並列モード中に **PARA** 命令を読み込んだ場合、実行時エラーとなる。
- **SYNC** :同期命令。並列モード実行中のプロセッサ間で同期を取り、逐次モードに移行する。並列モード中に **SYNC** 命令を読み込んだ場合、並列モード実行中の全てのプロセッサが **SYNC** 命令に到達するまで実行を中断する。全てのプロセッサが **SYNC** に到達すると、それ以降はプロセッサ  $P_0$  のみが命令を実行する。逐次モード中に **SYNC** 命令が読み込まれると実行時エラーとなる。

付録 A.1 に本研究で使用する並列アセンブラの命令セットを示す。

#### 3.3 VPSM(Virtual Parallel Stack Machine) インタプリタ

本研究では JAVA 言語を用いて VPSM インタプリタの設計を行った。付録 A.2 に本研究で作成した VPSM インタプリタプログラムを示す。

### 3.3.1 VPSM インタプリタの構成

VPSM インタプリタは以下の 3 つの要素から成る。

1. データセグメント (Data Segment) : PRAM 用並列言語プログラム中で使用する変数の値を保存する。データセグメントは全てのプロセッサで共有される。
2. 命令セグメント (Instruction Segment) : 並列アセンブラプログラムを保存する。Instruction Segment は全てのプロセッサで共有される。
3. 仮想スタックマシン (Virtual Stack Machine) : 1 個のスタックおよびプログラムカウンタから成る。仮想スタックマシンは各プロセッサに 1 台ずつ作成される。

各仮想スタックマシンは、PRAM の各プロセッサの動作をシミュレートする。各仮想スタックマシンは、プログラムカウンタが指す番地にある命令セグメントの命令を読み込み、読み込んだ命令に応じてデータセグメントへのデータの読み書き、各自のスタックへのデータの出し入れを行い、次の命令へ移る。

VPSM インタプリタは逐次状態と並列状態の 2 つの状態を持つ。逐次状態では、識別番号 0 を持つ 1 台の仮想スタックマシンのみが動作を行い、並列状態では PRAM 用並列言語プログラム中で指定された識別番号を持つ複数のスタックマシンが動作を行う。初期状態においては VPSM インタプリタは逐次状態にある。命令 `PARA` を読み込むと並列状態に以降し、複数の仮想スタックマシンが起動される。並列状態において命令 `SYNC` を読み込むと、全ての仮想スタックマシンが `SYNC` に到達するまで処理を中断し、その後逐次状態に移行する。

### 3.3.2 VPSM インタプリタの仕様

以下に本研究で作成した VPSM インタプリタの仕様を示す。

(VPSM インタプリタの使用方法)

`foo.pram` を PRAM 用並列アルゴリズム言語によって記述された PRAM アルゴリズムとする。以下のコマンドを実行すると、`foo.pram` が並列アセンブラにコンパイルされ、`outputfile` に出力される。`outputfile` を省略した場合は `OpCode.asm` に出力される。

```
> java Pram [-option] foo.pram [outputfile]
```

`foo.asm` を並列アセンブラによって記述された PRAM アルゴリズムとする。以下のコマンドを実行すると、PRAM アルゴリズムの PRAM 上での実行がシミュレートされる。`foo.asm` を省略した場合は `OpCode.asm` を入力ファイルとして実行する。

```
> java VSM [-option] foo.asm
```

(VPSM インタプリタのオプション)

VPSM インタプリタは実行時に以下のオプションを指定できる。

- `d`: VPSM をデバグモードで実行する。
- `t`: VPSM をトレスモードで実行する。
- `n`: 並列アセンブラプログラムの文法チェックのみを行い実行しない。
- `c`: 実行後に実行命令数、データセグメントサイズ、最大スタック深度を表示する。

- o: 実行前に並列アセンブラプログラムを表示する。
- p: 実行前に並列アセンブラプログラムを OpCopde.asm に出力する。

c オプションを付けて VPSM を実行することにより、PRAM アルゴリズムを PRAM 上で実行した場合の実行時間を表示することができる。

## 4 結果および考察

本研究で作成した VPSM インタプリタが正しく PRAM アルゴリズムの動作をシミュレートしていることを確認するため、付録 A.3 に示す PRAM 用並列言語プログラムを用いて出力および実行時間を検証した。

付録 A.3 の PRAM 用並列言語プログラムは、 $n$  個のデータの和を並列に求めるプログラムであり、PRAM 上では  $p$  プロセッサを用いて  $O(\frac{n}{p} + \log p)$  時間で求めることができる。付録 A.3 の PRAM 用並列言語プログラムは、PRAM コンパイラにより付録 A.4 に示す並列アセンブラに変換される。本研究で作成した VPSM インタプリタを用いて付録 A.4 を実行すると、正しい解が出力され、また、PRAM 上での実行時間も出力された。図 6 に VPSM インタプリタが出力した PRAM アルゴリズムの実行時間のシミュレート結果と、理論値との比較を示す。図 6 より、実行時間のシミュレート結果は理論値とほぼ一致する。理論値と実行時間のシミュレート結果の間にデータ数に関わらず定数の差が出るのは変数の初期値設定時間や出力時間が加わるためである。

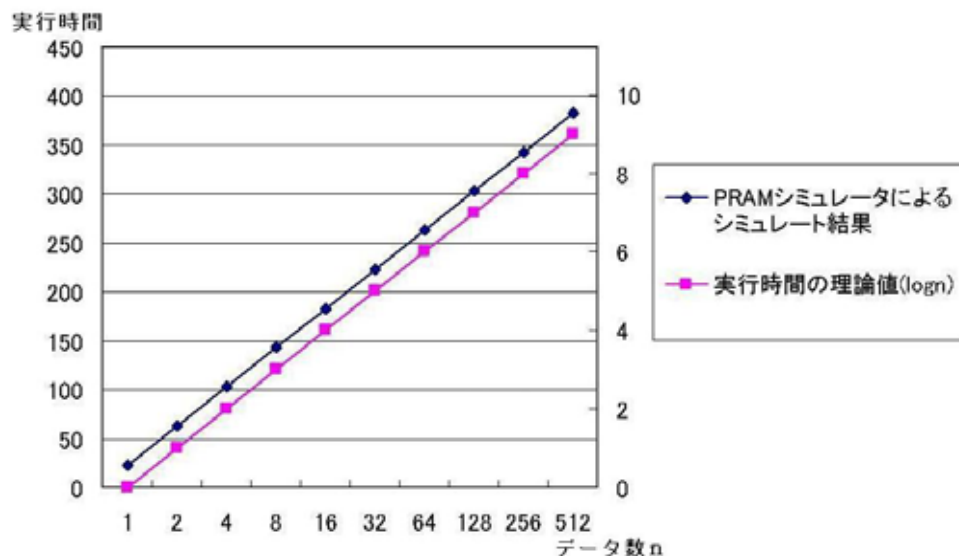


図 6 論理値と実行結果の比較

## 5 結論および今後の課題

本研究では、JAVA 言語を用いて VPSM インタプリタの設計を行った。本研究で設計した VPSM インタプリタは、PRAM 上での並列アルゴリズムの実行をシミュレートすることができる。また、PRAM 上での並列アルゴリズムの実行時間を出力する機能を持ち、PRAM アルゴリズムの計算量を実験的に評価することができる。

今後の課題としては、操作性を上げたり、並列に実行するプロセッサ数を自動的に最適な数に設定できるプログラムなど、VPSM インタプリタの機能の拡張が挙げられる。

## 6 謝辞

卒業論文の作成にあたり、石水隆助手には数多くの御指導や御助言を頂きました。多忙な中、夜まで見て頂いたりどんな質問も丁寧に教えて頂き、本当にありがとうございました。また、同じ研究室の皆には色々とお世話になり、感謝申し上げます

## 参考文献

- [1] 加藤暢, ”平成 17 年度第 5 セメスター 情報・システムプロジェクト I 指導書,” 近畿大学工学部情報学科, (2005)
- [2] 辻野嘉宏, ”情報工学入門選書 10 コンパイラ,” 昭晃堂, (1996)
- [3] 疋田輝雄, 石畑清, ”コンパイラの理論と実現,” 共立出版, (1988)
- [4] J.JáJá to Parallel Algorithms, Addison- Wesley Publishing Company, (1992)

## 付録 A 付録

### A.1 並列アセンブラの文法

以下に並列アセンブラの命令セットを示す。ただし、各命令実行中にプログラムカウンタが変更されなかった場合は、命令実行後プログラムカウンタが1増やされる。

(算術演算命令)

- **ADD** : 加算命令。スタックより値  $val1, val2$  を取り出し、値  $val2 + val1$  をスタックに挿入する。
- **SUB** : 減算命令。スタックより値  $val1, val2$  を取り出し、値  $val2 - val1$  をスタックに挿入する。
- **MUL** : 乗算命令。スタックより値  $val1, val2$  を取り出し、値  $val2 * val1$  をスタックに挿入する。
- **DIV** : 除算命令。スタックより値  $val1, val2$  を取り出し、 $val1 \neq 0$  であれば値  $val2/val1$  をスタックに挿入する。 $val1 = 0$  のときは実行時エラーとなる。
- **MOD** : 剰余算命令。スタックより値  $val1, val2$  を取り出し、 $val1 \neq 0$  であれば値  $val2 \% val1$  をスタックに挿入する。 $val1 = 0$  のときは実行時エラーとなる。
- **INC** : インクリメント命令。スタックトップの値  $val$  を1増やす。
- **DEC** : デクリメント命令。スタックトップの値  $val$  を1減らす。
- **CSIGN** : 単項負号命令。スタックトップの値  $val$  を  $-val$  に変更する。

(論理演算命令)

- **AND** : 論理積命令。スタックより値  $val1, val2$  を取り出し、 $val1 \neq 0$  かつ  $val2 \neq 0$  ならば1を、それ以外ならば0をスタックに挿入する。
- **OR** : 論理和命令。スタックより値  $val1, val2$  を取り出し、 $val1 \neq 0$  または  $val2 \neq 0$  ならば1を、それ以外ならば0をスタックに挿入する。
- **NOT** : 否定命令。スタックより値  $val$  を取り出し、 $val = 0$  ならば1を、 $val \neq 0$  ならば0をスタックに挿入する。
- **XOR** : 排他的論理和命令。スタックより値  $val1, val2$  を取り出し、 $val1 \neq 0$  かつ  $val2 = 0$  または  $val1 = 0$  かつ  $val2 \neq 0$  ならば1を、それ以外ならば0をスタックに挿入する。
- **COMP** : 比較命令。スタックより値  $val1, val2$  を取り出し、 $val1 = val2$  ならば0を、 $val1 < val2$  ならば1を、 $val1 > val2$  ならば-1をスタックに挿入する。

(スタックへの値の出し入れおよびデータセグメントへの読み書き命令)

- **ASSIGN** : 代入命令。以下の操作を行う。
  1. スタックより値  $val$  および値  $addr$  を取り出す。
  2. データセグメントの  $addr$  番地に値  $val$  を書き込む。
  3. スタックに値  $val$  を挿入する。
- **PUSH  $addr$**  : 参照値スタック挿入命令。データセグメントの  $addr$  番地の値  $val$  をスタックに挿入する。
- **PUSHI  $val$**  : スタック挿入命令。値  $val$  をスタックに挿入する。

- **POP *addr*** : スタック取り出し命令。スタックより値 *val* を取り出し、データセグメントの *addr* 番地に値 *val* を書き込む。
- **LOAD** : 参照値読み込み命令。スタックより値 *addr* を取り出し、データセグメントの *addr* 番地の値 *val* をスタックに挿入する。
- **REMOVE** : スタック除去命令。スタックよりデータを 1 つ取り除く。
- **COPY** : スタック複写命令。スタックトップのデータ *val* をスタックに挿入する。
- **PUSHP** : プロセッサ番号スタック挿入命令。スタックに命令を実行しているプロセッサのプロセッサ番号を挿入する。

(算術演算およびデータセグメントへの読み書き命令)

- **ADDLHS** : 加算代入命令。以下の操作を行う。
  1. スタックより値 *val1* および値 *addr* を取り出す。
  2. データセグメントの *addr* 番地の値 *val2* を読み込む。
  3. データセグメントの *addr* 番地に値  $val2 + val1$  を書き込む。
  4. スタックに値  $val2 + val1$  を挿入する。
- **SUBLHS** : 減算代入命令。以下の操作を行う。
  1. スタックより値 *val1* および値 *addr* を取り出す。
  2. データセグメントの *addr* 番地の値 *val2* を読み込む。
  3. データセグメントの *addr* 番地に値  $val2 - val1$  を書き込む。
  4. スタックに値  $val2 - val1$  を挿入する。
- **MULLHS** : 乗算代入命令。以下の操作を行う。
  1. スタックより値 *val1* および値 *addr* を取り出す。
  2. データセグメントの *addr* 番地の値 *val2* を読み込む。
  3. データセグメントの *addr* 番地に値  $val2 * val1$  を書き込む。
  4. スタックに値  $val2 * val1$  を挿入する。
- **DIVLHS** : 除算代入命令。以下の操作を行う。ただし、 $val1 = 0$  ならば実行時エラーとなる。
  1. スタックより値 *val1* および値 *addr* を取り出す。
  2. データセグメントの *addr* 番地の値 *val2* を読み込む。
  3. データセグメントの *addr* 番地に値  $val2 / val1$  を書き込む。
  4. スタックに値  $val2 / val1$  を挿入する。
- **DIVLHS** : 剰余算代入命令。以下の操作を行う。ただし、 $val1 = 0$  ならば実行時エラーとなる。
  1. スタックより値 *val1* および値 *addr* を取り出す。
  2. データセグメントの *addr* 番地の値 *val2* を読み込む。
  3. データセグメントの *addr* 番地に値  $val2 \% val1$  を書き込む。
  4. スタックに値  $val2 \% val1$  を挿入する。
- **PREINC *addr*** : 前置インクリメント代入命令。以下の操作を行う。
  1. データセグメントの *addr* 番地の値 *val* を読み込む。
  2. データセグメントの *addr* 番地に値  $val + 1$  を書き込む。
  3. スタックに値  $val + 1$  を挿入する。
- **PREDEC *addr*** : 前置デクリメント代入命令。以下の操作を行う。



1. データセグメントの *addr* 番地の値 *val* を読み込む。
  2. データセグメントの *addr* 番地に値  $val - 1$  を書き込む。
  3. スタックに値  $val - 1$  を挿入する。
- **POSTINC *addr*** : 後置インクリメント代入命令。以下の操作を行う。
    1. データセグメントの *addr* 番地の値 *val* を読み込む。
    2. データセグメントの *addr* 番地に値  $val + 1$  を書き込む。
    3. スタックに値 *val* を挿入する。
  - **POSTDEC *addr*** : 後置デクリメント代入命令。以下の操作を行う。
    1. データセグメントの *addr* 番地の値 *val* を読み込む。
    2. データセグメントの *addr* 番地に値  $val - 1$  を書き込む。
    3. スタックに値 *val* を挿入する。

(ジャンプ命令)

- **JUMP *addr*** : 無条件ジャンプ命令。プログラムカウンタの値を *addr* に変更する。
- **BEQ *addr*** : 条件付ジャンプ命令。スタックから値 *val* を取り出し、 $val = 0$  ならばプログラムカウンタの値を *addr* に変更する。
- **BNE *addr*** : 条件付ジャンプ命令。スタックから値 *val* を取り出し、 $val \neq 0$  ならばプログラムカウンタの値を *addr* に変更する。
- **BLT *addr*** : 条件付ジャンプ命令。スタックから値 *val* を取り出し、 $val < 0$  ならばプログラムカウンタの値を *addr* に変更する。
- **BLE *addr*** : 条件付ジャンプ命令。スタックから値 *val* を取り出し、 $val \leq 0$  ならばプログラムカウンタの値を *addr* に変更する。
- **BGT *addr*** : 条件付ジャンプ命令。スタックから値 *val* を取り出し、 $val > 0$  ならばプログラムカウンタの値を *addr* に変更する。
- **BGE *addr*** : 条件付ジャンプ命令。スタックから値 *val* を取り出し、 $val \geq 0$  ならばプログラムカウンタの値を *addr* に変更する。
- **CALL *addr*** : サブルーチン呼び出し命令。スタックに現在のプログラムカウンタの値を挿入し、プログラムカウンタの値を *addr* に変更する。
- **RET** : サブルーチン帰還命令。スタックから値 *addr* を取り出し、プログラムカウンタの値を *addr* に変更する。

(入出力命令)

- **INPUT** : 整数値入力命令。標準入力より整数 *i* を読み込み、*i* をスタックに挿入する。
- **INPUTC** : 文字入力命令。標準入力より文字 *c* を読み込み、*c* の文字コードをスタックに挿入する。
- **OUTPUT** : 整数値出力命令。スタックより整数値 *i* を取り出し、*i* を標準出力に出力する。
- **OUTPUT** : 文字出力命令。スタックより整数値 *i* を取り出し、文字コード *i* の文字 *c* を標準出力に出力する。
- **OUTPUTL** : 改行出力命令。標準出力に改行コードを出力する。

(フラグレジスタ操作命令)

- **SETFR** *addr* : レジスタセット命令。フラグレジスタの値を *addr* に変更する。
- **INCFR** *addr* : レジスタ加算命令。フラグレジスタの値に *addr* を加える。
- **DECFR** *addr* : レジスタ減算命令。フラグレジスタの値から *addr* を引く。

(並列命令)

- **PARA** : 並列開始命令。逐次モードから並列モードに移行する。逐次モード実行中に **PARA** 命令を読み込んだ場合、スタックから 2 個のデータ  $d_1, d_2$  が取り出される。この命令以降は、プロセッサ  $P_{d_1}, P_{d_1+1}, P_{d_1+2}, \dots, P_{d_2}$  が並列に命令を実行する。並列モード中に **PARA** 命令を読み込んだ場合、実行時エラーとなる。
- **SYNC** : 同期命令。並列モード実行中のプロセッサ間で同期を取り、逐次モードに移行する。並列モード中に **SYNC** 命令を読み込んだ場合、並列モード実行中の全てのプロセッサが **SYNC** 命令に到達するまで実行を中断する。全てのプロセッサが **SYNC** に到達すると、それ以降はプロセッサ  $P_0$  のみが命令を実行する。逐次モード中に **SYNC** 命令が読み込まれると実行時エラーとなる。

(その他の命令)

- **NOP** : 何も行わない。
- **HALT** : 停止命令。VPSM を停止する。
- **EOF** : 実行時エラーとなる。
- **ERROR** : 実行時エラーとなる。

## A.2 VPSM インタプリタプログラム

VPSM インタプリタは以下に示す 10 個のプログラムより成る。

- VSM.java :メインプログラム
- DataSegment.java :データセグメントを構成する
- InputFile.java :入出力を扱う
- Instraction.java :命令コードを定義する
- InstractionSegment.java :命令セグメントを定義する
- Opetor.java :アセンブラ命令セットを定義する
- PramMode.java :PRAM のモードを定義する
- Stack.java :スタックを構成する
- VirtualStackMachine.java :仮想スタックマシンを構成する
- VSMLexer.java :字句解析を行う

### A.2.1 VSM.java

```
import ioTools.*;

public class VSM implements Operators, PramMode {
    static final int PROCESSORMAX = 1024;
    static final int MODE = M_EREW;          // 同時読み書きモード
    static final boolean FREGP = false;     // フレームレジスタ修飾

    static VSMLexer lexer;                  // 字句解析機
    /* Lexer は並列対応時でも 1 個のまま変更しない */

    static VirtualStackMachine vsm[];      // Virtual Stack Machine
    /* 1 台のプロセッサは 1 台の VSM である */
    /* 並列対応時は VirtualStackMachine 型の配列にする */
    // static VirtualStackMachine vsm[];

    static InstractionSegment iseg;        // Instraction Segment
    static DataSegment dseg;               // Data Segment
    /* Iseg, Dseg は並列対応時でも各 1 個ずつのまま変更しない */

    static boolean inParallel;
    // true: 並列処理中 false: 逐次処理中
    static int low_processor;
    static int high_processor;
}
```

```

/* 並列処理時は vsm[low_processor] ~ vsm[high_processor] が動く
   逐次処理時は vsm[0] が動く */

static boolean execSW = true;          /* コンパイルだけ */
static boolean objOutSW = false;      /* 目的コードの表示 */
static boolean objPrtSW = false;     /* 目的コードの表示 */
static boolean traceSW = false;      /* トレースモード */
static boolean statSW = true;        /* 実行データの表示 */
static boolean debugSW = false;     /* デバッグモード */
static boolean symPrtSW = false;     /* 記号表の表示 */

public static void main (String[] args) {
    String sourceFile = setUpOption(args);
    dseg = new DataSegment(MODE);      /* data segment */
    lexer = new VSMLexer(sourceFile); /* 字句解析器 */
    iseg = lexer.makeIseg(debugSW);   /* instruction segment */
    lexer.inFile.closeFile();
    /* 解析が終了したらファイルを閉じる */

    if (objOutSW) iseg.dump(); // Iseg の内容を表示する
    if (objPrtSW) iseg.dumpToFile();
    // Iseg の内容をファイルに出力する

    if (execSW) {
        vsm = new VirtualStackMachine[PROCESSORMAX];
        for (int i=0; i<PROCESSORMAX; i++)
            vsm[i]= new VirtualStackMachine(iseg, dseg, i, FREGP);
        /* vsm[0] ~ vsm[PROCESSORMAX] を作成する */
        /* vsm[i] のコンストラクタの引数は (iseg, dseg, i, FREGP) */
        startVSM(0);                    /* VSM 開始 */

        if (statSW) execReport();
        else System.out.println("Execution finished");
    }
}

public static void startVSM(int startAddr) {
    int[] operator;
    operator=new int[PROCESSORMAX];
}

```

```

vsm[0].setProgramCounter(startAddr);
vsm[0].awake();
inParallel = false;
low_processor = 0;
high_processor = 0;
boolean syncck=true;

while (true) { // 並列処理か逐次処理かで分岐
    if(inParallel==true){ // 並列処理の場合
        syncck=true; // SYNC 到達チェック用変数を true に初期化
        for(int i=low_processor;i<=high_processor; i++){
            // vsm[low_processor] ~ vsm[high_processor] に対して以下を実行する
            operator[i]=vsm[i].execute(traceSW);
            // execute() を実行する
            if (operator[i] == HALT)
                // 命令が PARA または HALT ならばエラー
                syntaxError(vsm[i].pctr);
            // (並列時にさらに並列はできない
            // また並列時にプログラムは終われない)
        }
        else if(operator[i] == PARA)
            syntaxError(vsm[i].pctr);
    }
    for (int i=low_processor;i<=high_processor; i++){
        if(operator[i]!=SYNC)
            // 命令が SYNC でなければ SYNC 到達チェック用変数を false に
            syncck=false;
        // (このとき 1 つでも SYNC 未到達の vsm があればチェック
        // 用変数が false に書き換えられる)
    }
    if(syncck==true){
        // SYNC 到達チェック用変数が true ならば逐次処理へ移行
        vsm[0].pctr=vsm[low_processor].pctr;
        // vsm[0] のプログラムカウンタを vsm[low_processor]
        // のプログラムカウンタに合わせる
        vsm[0].incProgramCounter();
        // vsm[0] のプログラムカウンタを 1 増やす
        inParallel=false;
        low_processor=0;
        // inParallel, low_processor, high_processor の
        // 値を適当なものに設定
    }
}

```

```

        high_processor=0;
        vsm[0].awake();// vsm[0] を起こす
    }
}
else { // 逐次処理の場合 vsm[0] に対して以下を実行する
    operator[0]=vsm[0].execute(traceSW);
    // execute() を実行する
    if (operator[0] == HALT) break;
    // 命令が HALT なら while ループ脱出 プログラム終了へ
    if (operator[0] == SYNC) // 命令が SYNC ならエラー
        syntaxError(vsm[0].pctr);
        // (逐次時に同期は取れない)
    else if (operator[0] == PARA) {
        // 命令が PARA なら並列処理に移行
        inParallel=true;
        // inParallel を適当な値に設定
        int proc=vsm[0].pctr;
        for(int i=low_processor; i<=high_processor; i++){
            //vsm[low_processor] ~ vsm[high_processor] のプログラムカウンタを vsm[0] の
            //プログラムカウンタに合わせる
            vsm[i].pctr=proc;
            vsm[i].incProgramCounter();// vsm[low_processor] ~
            //vsm[high_processor] のプログラムカウンタを 1 増やす
        }
        for(int i=low_processor; i<=high_processor; i++)
            vsm[i].awake();
            //vsm[low_processor] ~ vsm[high_processor] を起こす
    }
}
dseg.set();// Dseg の内容を更新する
// 各 VSM が Dseg に対して書き込みを行っても、
// dseg.set() を実行するまでは Dseg から読み出される値は
// 書き込みを行う前のものになっている
}
}
// 実行時の引数に応じてオプションをセットする
public static String setUpOption (String[] args) {
    int i;
    for (i=0; i<args.length; i++) {
        if (args[i].charAt(0) == '-') {

```

```

        if (args[i].indexOf('c') != -1) statSW = true;
        /* 実行データの表示 */
        if (args[i].indexOf('d') != -1) debugSW = true;
        /* デバッグモード */
        if (args[i].indexOf('n') != -1) execSW = false;
        /* コンパイルだけ */
        if (args[i].indexOf('o') != -1) objOutSW = true;
        /* 目的コードの表示 */
        if (args[i].indexOf('p') != -1) objPrtSW = true;
        /* 目的コードの表示 */
        if (args[i].indexOf('s') != -1) symPrtSW = true;
        /* 記号表の表示 */
        if (args[i].indexOf('t') != -1) traceSW = true;
        /* トレースモード */
    } else break;
}
if (i<args.length) return args[i];
else return "OpCode.asm";
}
// 並列処理に移行する
// このメソッドは VSM.java から直接呼び出すのではなく、
// 逐次処理時に vsm[0] が VirtualStackMachine.java で PARA 命令を
//実行したときに VirtualStackMachine.java から呼び出される
public static void setParallel(int l, int h) {
    low_processor = l;
    high_processor = h;
    inParallel = true;
}
// 実行データの表示
public static void execReport() {
    System.out.println();
    System.out.println("Object code size: "+iseg.size);
    // Iseg は共通なので並列対応時でもこの部分は同じ
    System.out.print("Max stack depth: ");
    int maxstdp =0;
    for(int i=0;i<PROCESSORMAX;i++){
        if (vsm[i].stack.maxSp > maxstdp)
            maxstdp=vsm[i].stack.maxSp;
    }
    System.out.println(maxstdp);
}

```

```

System.out.print("Max frame size: ");
// 並列対応時は、vsm[0].stack.maxSp ~ vsm[PROCESSORMAX]
//.stack.maxSp の最大値を表示する
System.out.println(dseg.size - vsm[0].minFreg);
System.out.print("Function calls: ");
// 並列対応時は、dseg.size-vsm[0].minFreg ~ dseg.size.vsm
//[PROCESSORMAX].minFreg の最大値を表示する
int Fccl=0;
for(int i=0;i<PROCESSORMAX;i++){
    Fccl=Fccl+vsm[i].callCtr;
}
System.out.println(Fccl);
// 並列対応時は、vsm[0].callCtr ~ vsm[PROCESSORMAX].
//callCtr の和を表示する
System.out.print("Executiin count : ");
System.out.println(vsm[0].insCtr);
// 並列対応時は、vsm[0].insCtr ~ vsm[PROCESSORMAX].insCtr の
//最大値を表示する
// (アセンブリプログラムが正しければ常に vsm[0].insCtr が
//最大値となっている)
}

public static void syntaxError (int addr) { /* 文法エラー */
    System.out.println("Syntax error at line " + addr);
    iseg.print(addr);
    System.out.println();
    System.exit(1);
}

public static void syntaxError (String err_mes, int addr) { /* 文法エラー */
    System.out.println("Systax error at line " + addr);
    System.out.println(err_mes);
    iseg.print(addr);
    System.out.println();
    System.exit(1);
}

public static void executeError (int addr) { /* 実行時エラー */
    System.out.println("Execute error at line " + addr);
    iseg.print(addr);
}

```



```

        System.out.println();
        System.exit(1);
    }

    public static void executeError (String err_mes, int addr) { /* 実行時エラー */
        System.out.println("Execute error at line " + addr);
        System.out.println(err_mes);
        iseg.print(addr);
        System.out.println();
        System.exit(1);
    }
}

```

#### A.2.2 DataSegment.java

```

public class DataSegment {
    static final int DSEGSIZE = 1024;
    int[] dseg;
    int[] tempDseg;      // 作業用配列
    boolean[] readCheck; // 同時読みチェック
    boolean[] writeCheck; // 同時書きチェック
    int maxAddr;        // 使用アドレスの最大値
    int mode;           // PRAM mode
    int size;           // サイズ

    public DataSegment(int m) {
        size = DSEGSIZE;
        dseg = new int[DSEGSIZE];
        tempDseg = new int[DSEGSIZE];
        readCheck = new boolean[DSEGSIZE];
        writeCheck = new boolean[DSEGSIZE];
        maxAddr = -1;
        mode = m;

        for (int i=0; i<DSEGSIZE; i++) { // データの初期化
            dseg[i] = 0;
            tempDseg[i] = 0;
            readCheck[i] = false;
            writeCheck[i] = false;
        }
    }
}

```

```

}

// アドレス addr のデータを読む
public int read(int addr) {
    if (addr < 0 || addr >= size)
        executeError("Illegal dseg address : " + addr);
    if (addr>maxAddr) maxAddr = addr;
    readCheck[addr] = true;
    return dseg[addr];
}

// アドレス addr にデータ val を書く
public void write(int addr, int val) {
    if (addr < 0 || addr >= size)
        executeError("Illegal dseg address : " + addr);
    if (addr>maxAddr) maxAddr = addr;
    writeCheck[addr] = true;
    tempDseg[addr] = val;          /* 更新データは tmp 配列に */
    return;
}

// データ更新し、チェックフラグを初期化する
public void set() {
    for(int i=0; i<=maxAddr; i++) {
        dseg[i] = tempDseg[i];    /* tmp 配列のデータをコピー */
        readCheck[i] = false;
        writeCheck[i] = false;
    }
    return;
}

public void executeError (String err_mes) { /* 実行時エラー */
    System.out.println("Processor "+VSM.vsm.pr);
    System.out.println("Execute error at line " + VSM.vsm.pctr);
    System.out.println(err_mes);
    VSM.iseg.print(VSM.vsm.pctr);
    System.out.println();
    System.exit(1);
}
}

```

### A.2.3 InputFile.java

```
import ioTools.*;
import java.io.*;

public class InputFile {
    BufferedReader buffer;    /* 入力ファイルのバッファ */
    String line;             /* 入力ファイルの 1 行分の文字列 */
    int linenum;            /* 入力ファイルの行番号 */
    int columnnum;         /* 入力ファイルの列番号 */
    char currentc;         /* 読み込んだ文字 */
    char nextc;            /* 次に読み込む文字 */

    /* コンストラクタでは, inputFileName というファイルを開き
       そのファイルを今後 buffer で参照する . また linenum,
       columnnum, currentc, nextc を初期化する */
    public InputFile(String inputFile_name) {
        buffer = FileIo.fRead(inputFile_name);
        linenum = 0;
        columnnum = 0;
        //入力ファイルから一行読む
        readInputFile();
        //最初の一文字目を読んで, その文字を nextc に格納する .
        nextc = ' ';
        nextChar();
    }

    //buffer から一行読み, 文字列変数 line にその行を格納するメソッド .
    public void readInputFile() {
        try {
            line = buffer.readLine();
        } catch(IOException error_report) {
            /* 読み込みエラーが発生したら, キャッチした例外を表示し,
               ファイルを閉じ, 処理系を終了させる */
            System.out.println(error_report);
            closeFile();
            System.exit(1);
        }
    }
}
```

```

}

//入力ファイルを閉じるメソッド.
public void closeFile() {
    try {
        buffer.close();
    } catch(IOException error_report) {
        System.out.println(error_report);
        System.exit(1);
    }
}

//次の文字を得るメソッド.
public char nextChar() {
    if (line == null) { //ファイル末に達したら'\0' を返す .
        currentc = nextc;
        nextc = '\0';
        return currentc;
    } else if (columnnum >= line.length()) { //行末に達したら'\n' を返す
        readInputFile();
        currentc = nextc;
        nextc = '\n';
        linenum++;
        columnnum = 0;
        return currentc;
    } else { //通常の動作 . 読んだ一文字を nextc に格納し, その値を返す .
        currentc = nextc;
        nextc = line.charAt(columnnum);
        columnnum++;
        return currentc;
    }
}
}
}

```

#### A.2.4 Instraction.java

```

class Instraction implements Operators {
    int operator;        /* オペレータ */
    int operand;        /* オペランド */
    boolean register;   /* アドレス修飾 */
}

```

```

public Instruction(int i, int j) {
    operator = i;
    operand = j;
    register = false;
}

public Instruction(int i) {
    operator = i;
    operand = -1;
    register = false;
}

public Instruction(int i, int j, boolean r) {
    operator = i;
    operand = j;
    register = r;
}

public Instruction(int i, boolean r) {
    operator = i;
    operand = -1;
    register = r;
}

public String toString() {
    switch(operator) {
    case PUSH:
    case PUSHI:
    case POP:
    case BEQ:
    case BNE:
    case BLE:
    case BLT:
    case BGE:
    case BGT:
    case JUMP:
    case CALL:
        return opName() + "\t" + operand + "\t";
        /* オペランドを持つ場合 */

```

```

default:
    return opName() + "\t\t";
        /* オペランドを持たない場合 */
    }
}

// オペランドコードをオペランド名に変換
public String opName() {
    switch(operator) {
        case NOP:      return "NOP   ";    // no operation
        case ASSGN:    return "ASSGN ";    // assign
        case ADD:      return "ADD   ";    // +
        case ADDLHS:   return "ADDLHS ";    // +=
        case SUB:      return "SUB   ";    // -
        case SUBLHS:   return "SUBLHS ";    // -=
        case MUL:      return "MUL   ";    // *
        case MULLHS:   return "MULLHS ";    // *=
        case DIV:      return "DIV   ";    // /
        case DIVLHS:   return "DIVLHS ";    // /=
        case MOD:      return "MOD   ";    // %
        case MODLHS:   return "MODLHS ";    // %=
        case CSIGN:    return "CSIGN ";    // 単項-
        case AND:      return "AND   ";    // and
        case OR:       return "OR    ";    // or
        case NOT:      return "NOT   ";    // not
        case XOR:      return "XOR   ";    // exclusive or
        case COMP:     return "COMP  ";    // comp
        case COPY:     return "COPY  ";    // copy
        case PUSH:     return "PUSH  ";    // push
        case PUSHI:    return "PUSHI ";    // push integer
        case POP:      return "POP   ";    // pop
        case REMOVE:   return "REMOVE";    // remove
        case LOAD:     return "LOAD  ";    // load
        case INC:      return "INC   ";    // ++
        case DEC:      return "DEC   ";    // --
        case PREINC:   return "PREINC";    // 前置++
        case PREDEC:   return "PREDEC";    // 前置--
        case POSTINC:  return "POSTINC";    // 後置++
        case POSTDEC:  return "POSTDEC";    // 後置--
        case SETFR:    return "SETFR ";    // set frame register
    }
}

```

```

    case INCFR:    return "INCFR ";    // inc frame register
    case DECFR:    return "DECFR ";    // dec frame register
    case JUMP:     return "JUMP  ";    // jump
    case BEQ:      return "BEQ   ";    // == ?
    case BNE:      return "BNE   ";    // != ?
    case BLT:      return "BLT   ";    // < ?
    case BLE:      return "BLE   ";    // <= ?
    case BGT:      return "BGT   ";    // > ?
    case BGE:      return "BGE   ";    // >= ?
    case CALL:     return "CALL  ";    // call
    case RET:      return "RET   ";    // return
    case INPUT:    return "INPUT ";    // input integer
    case INPUTC:   return "INPUTC";    // input character
    case INPUTS:   return "INPUTS";    // input string
    case OUTPUT:   return "OUTPUT ";    // output integer
    case OUTPUTC:  return "OUTPUTC";    // output character
    case OUTPUTL:  return "OUTPUTL";    // output line
    case OUTPUTS:  return "OUTPUTS";    // output string
    case RAND:     return "RAND  ";    // random
    case HALT:     return "HALT  ";    // halt
    case ASSGNS:   return "ASSGNS ";    // assign string
    case LOADS:    return "LOADS ";    // load string
    case PARA:     return "PARA  ";    // parallel
    case SYNC:     return "SYNC  ";    // synchronous
    case PUSHHP:   return "PUSHHP ";    // push processor number
    case EOF:      return "EOF   ";    // end of file
    default:       return "ERROR ";    // error
}
}
}

```

#### A.2.5 InstractionSegment.java

```

import ioTools.*; //ファイル入出力用
import java.io.*; //ファイル入出力用
import java.util.Vector; //ベクトル用

public class InstractionSegment implements Operators {
    Vector iseg;
    int isegPtr;

```

```

int size;

boolean debugSW;

public InstructionSegment(boolean dsw) {
    iseg = new Vector();
    isegPtr = 0;
    size = 0;
    debugSW = dsw;
}

// Iseg に命令を加える
public int appendCode(Instruction inst) {
    if (size == isegPtr) {
        if (debugSW) System.out.println(isegPtr+": "+inst);
        iseg.addElement(inst);
        size++;
    } else {
        Instruction oldInst = ((Instruction) iseg.get(isegPtr));
        if (debugSW) System.out.print(isegPtr+": "+oldInst);
        iseg.removeElementAt(isegPtr);
        iseg.insertElementAt(inst, isegPtr);
        if (debugSW) System.out.println("-> "+inst);
    }
    isegPtr++;
    return isegPtr-1;
}

public int appendCode(int operator, int operand, boolean register) {
    if (size == isegPtr) {
        Instruction inst = new Instruction(operator, operand, register);
        if (debugSW) System.out.println(isegPtr+": "+inst);
        iseg.addElement(inst);
        size++;
    } else {
        Instruction inst = ((Instruction) iseg.get(isegPtr));
        if (debugSW) System.out.print(isegPtr+": "+inst);
        inst.operator = operator;
        inst.operand = operand;
        inst.register = register;
    }
}

```



```

        if (debugSW) System.out.println("-> "+inst);
    }
    isegPtr++;
    return isegPtr-1;
}

public int appendCode(int operator, boolean register) {
    if (size == isegPtr) {
        Instruction inst = new Instruction(operator, register);
        if (debugSW) System.out.println(isegPtr+": "+inst);
        iseg.addElement(inst);
        size++;
    } else {
        Instruction inst = ((Instruction) iseg.get(isegPtr));
        if (debugSW) System.out.print(isegPtr+": "+inst);
        inst.operator = operator;
        inst.operand = -1;
        inst.register = register;
        if (debugSW) System.out.println("-> "+inst);
    }
    isegPtr++;
    return isegPtr-1;
}

public int appendCode(int operator, int operand) {
    if (size == isegPtr) {
        Instruction inst = new Instruction(operator, operand);
        if (debugSW) System.out.println(isegPtr+": "+inst);
        iseg.addElement(inst);
        size++;
    } else {
        Instruction inst = ((Instruction) iseg.get(isegPtr));
        if (debugSW) System.out.print(isegPtr+": "+inst);
        inst.operator = operator;
        inst.operand = operand;
        inst.register = false;
        if (debugSW) System.out.println("-> "+inst);
    }
    isegPtr++;
    return isegPtr-1;
}

```

```

}

public int appendCode(int operator) {
    if (size == isegPtr) {
        Instraction inst = new Instraction(operator);
        if (debugSW) System.out.println(isegPtr+": "+inst);
        iseg.addElement(inst);
        size++;
    } else {
        Instraction inst = ((Instraction) iseg.get(isegPtr));
        if (debugSW) System.out.print(isegPtr+": "+inst);
        inst.operator = operator;
        inst.operand = -1;
        inst.register = false;
        if (debugSW) System.out.println("-> "+inst);
    }
    isegPtr++;
    return isegPtr-1;
}

public int operator(int addr) {      /* オペレータを返す */
    if (addr < 0 || addr >= size)
        executeError("Illegal iseg address ", addr);
    return ((Instraction) iseg.get(addr)).operator;
}

public int operand(int addr) {      /* オペランドを返す */
    if (addr < 0 || addr >= size)
        executeError("Illegal iseg address ", addr);
    return ((Instraction) iseg.get(addr)).operand;
}

public boolean register (int addr) { /* レジスタを返す */
    if (addr < 0 || addr >= size)
        executeError("Illegal iseg address ", addr);
    return ((Instraction) iseg.get(addr)).register;
}

// 命令のジャンプ先のアドレスをチェック
public void checkAddress(int addr) {

```

```

Instruction inst = (Instruction) iseg.get(addr);
switch(inst.operator) {
    case JUMP:
    case BEQ:
    case BNE:
    case BLT:
    case BLE:
    case BGT:
    case BGE:
    case CALL:
        if(inst.operand < 0 || inst.operand >= size)
            syntaxError("Illegal iseg address : " + inst, addr);
        break;
    default:
        break;
}
}

// 指定したアドレスの命令を表示
public void print(int addr) {
    System.out.print(addr + ": " + (Instruction) iseg.get(addr));
}

// Iseg を表示
public void dump() {
    for (int i=0; i<isegPtr; i++)
        System.out.println(i + ": " + (Instruction) iseg.get(i));
}

// Iseg をデフォルトファイルに出力
public void dumpToFile() {
    PrintWriter outputFile = FileIo.fWrite("OpCode.asm", false);
    for (int i=0; i<isegPtr; i++)
        outputFile.println((Instruction) iseg.get(i));
    outputFile.close();
}

// Iseg を指定したファイルに出力
public void dumpToFile(String fileName) {
    PrintWriter outputFile = FileIo.fWrite(fileName, false);

```

```

    for (int i=0; i<isegPtr; i++)
        outputFile.println((Instraction) iseg.get(i));
    outputFile.close();
}

// addr 番目の命令の オペレータ, オペランド を operator, operand に変更する
public void replaceCode(int addr, int operator, int operand) {
    Instraction inst = ((Instraction) iseg.get(addr));
    if (debugSW)
        System.out.print(addr + ": " + inst);
    inst.operator = operator;
    inst.operand = operand;
    if (debugSW)
        System.out.println("-> " + inst);
}

// addr 番目の命令のオペランド を operand に変更する
public void replaceCode(int addr, int operand) {
    Instraction inst = ((Instraction) iseg.get(addr));
    if (debugSW)
        System.out.print(addr + ": " + inst);
    inst.operand = operand;
    if (debugSW)
        System.out.println("-> " + inst);
}

// addr 番目の命令を inst に変更する
public void replaceCode(int addr, Instraction inst) {
    Instraction oldInst = ((Instraction) iseg.get(addr));
    if (debugSW) System.out.print(addr+": "+oldInst);
    iseg.removeElementAt(addr);
    iseg.insertElementAt(inst, addr);
    if (debugSW) System.out.println("-> "+inst);
}

public void syntaxError(String err_mes, int addr) {    /* 文法エラー */
    System.out.println("Syntax error at line " + addr);
    System.out.println(err_mes);
    System.out.println((Instraction) iseg.get(addr));
    System.exit(1);
}

```

```

}

public void executeError (String err_mes, int addr) { /* 実行時エラー */
    System.out.println("Execute error at line " + addr);
    System.out.println(err_mes);
    System.exit(1);
}
}

```

#### A.2.6 Operators.java

```

interface Operators {
    static final int NOP      = 0; // no operation
    static final int ASSGN    = 1; // assign
    static final int ADD      = 2; // +
    static final int ADDLHS   = 3; // +=
    static final int SUB      = 4; // -
    static final int SUBLHS   = 5; // -=
    static final int MUL      = 6; // *
    static final int MULLHS   = 7; // *=
    static final int DIV      = 8; // /
    static final int DIVLHS   = 9; // /=
    static final int MOD      = 10; // %
    static final int MODLHS   = 11; // %=
    static final int CSIGN    = 12; // 単項-
    static final int AND      = 13; // and
    static final int OR       = 14; // or
    static final int NOT      = 15; // not
    static final int XOR      = 16; // exclusive or
    static final int COMP     = 17; // comp
    static final int COPY     = 18; // copy
    static final int PUSH     = 19; // push
    static final int PUSHI    = 20; // push integer
    static final int REMOVE   = 21; // remove
    static final int LOAD     = 22; // load
    static final int POP      = 23; // pop
    static final int INC      = 24; // ++
    static final int DEC      = 25; // --
    static final int PREINC   = 26; // 前置++
    static final int PREDEC   = 27; // 前置--
}

```

```

static final int POSTINC = 28; // 後置++
static final int POSTDEC = 29; // 後置--
static final int SETFR   = 30; // set frame register
static final int INCFR   = 31; // inc frame register
static final int DECFR   = 32; // dec frame register
static final int JUMP    = 33; // jump
static final int BLT     = 34; // < ?
static final int BLE     = 35; // <= ?
static final int BEQ     = 36; // == ?
static final int BNE     = 37; // != ?
static final int BGE     = 38; // > ?
static final int BGT     = 39; // >= ?
static final int CALL    = 40; // call
static final int RET     = 41; // return
static final int INPUT   = 42; // input integer
static final int INPUTC  = 43; // input character
static final int OUTPUT  = 44; // output integer
static final int OUTPUTC = 45; // output character
static final int OUTPUTL = 46; // output line
static final int RAND    = 47; // random
static final int HALT    = 48; // halt
static final int PUSHS   = 49; // push string
static final int POPS    = 50; // pop string
static final int ASSGNS  = 51; // assign string
static final int LOADS   = 52; // load string
static final int INPUTS  = 53; // input string
static final int OUTPUTS = 54; // output string
static final int PARA    = 55; // parallel
static final int SYNC    = 56; // synchronous
static final int PUSHPP  = 57; // push processor number
static final int EOF     = 255; // end of file
static final int ERROR   = -1; // error
}

```

### A.2.7 PramMode.java

```

interface PramMode {
    static final int M_EREW = 0;
}

```

```

static final int M_CREW = 1;
static final int M_COMMON_CWCW = 2;
static final int M_ARBITRARY_CRCW = 3;
static final int M_PRIORITY_CRCW = 4;
}

```

#### A.2.8 Stack.java

```

public class Stack {
    static final int STACKSIZE = 128;
    int[] st;          // Stack
    int sp;           // StackPointer
    int size;         // StackSize
    int maxSp;       // Max Stack Pointer

    public Stack () {
        size = STACKSIZE;
        st = new int[STACKSIZE];
        sp = -1;
        maxSp = -1;
    }

    public int pop () {
        if (sp < 0) executeError("Stack Underflow");
        int val = st[sp];
        sp--;
        return val;
    }

    public void push (int val) {
        sp++;
        if (sp >= size) executeError("Stack Overflow");
        st[sp] = val;
        if (sp > maxSp) maxSp = sp;
        return;
    }

    public String toString() {
        if (sp >= 0) return sp + "\t" + st[sp] + "\t";
    }
}

```

```

        else return "\t\t";
    }

    public void executeError (String err_mes) { /* 実行時エラー */
        System.out.println("Processor "+VSM.vsm.pr);
        System.out.println("Execute error at line " + VSM.vsm.pctr);
        System.out.println(err_mes);
        VSM.iseg.print(VSM.vsm.pctr);
        System.out.println();
        System.exit(1);
    }
}

```

#### A.2.9 VirtualStackMachine.java

```

import ioTools.*;

public class VirtualStackMachine implements Operators, PramMode {
    InstructionSegment iseg;// Instranciton Segment
    DataSegment dseg;      // Data Segment
    Stack stack;          // Stack
    int pctr;             // Program Counter
    boolean isRunning;    // status
    boolean fregP;
    int freg;            // Frame Register
    int minFreg;
    int callCtr;        // Call Counter
    int insCtr;        // Instrument Counter
    int pr;            // Processor Number

    public VirtualStackMachine
        (InstructionSegment is, DataSegment ds, int p, boolean fp) {
        iseg = is;
        dseg = ds;
        stack = new Stack();
        pctr = 0;
        isRunning = false;
        callCtr = 0;
        insCtr = 0;
        pr = p;
    }
}

```



```

    freg = 0;
    fregP = fp;
    minFreg = ds.size;
}

public int execute(boolean traceSW) {
    int operator = iseg.operator(pctr);
    int operand = iseg.operand(pctr);
    if (iseg.register(pctr) && fregP)
        operand += freg;

    if (isRunning) {
        int val, val1, val2, addr;
        int i;
        String str;
        insCtr++;

        if (traceSW) iseg.print(pctr);

        switch(operator) {
            case NOP:      /* No operation */
                pctr++;
                break;
            case ASSGN:    /* assign */
                val = stack.pop();
                addr = stack.pop();
                dseg.write(addr, val, pr);
                stack.push(val);
                pctr++;
                break;
            case ADD:      /* + */
                val1 = stack.pop();
                val2 = stack.pop();
                stack.push(val2 + val1);
                pctr++;
                break;
            case ADDLHS:   /* += */
                val1 = stack.pop();
                addr = stack.pop();
                val2 = dseg.read(addr, pr);

```

```

    val = val2 + val1;
    dseg.write(addr, val, pr);
    stack.push(val);
    pctr++;
    break;
case SUB:      /* - */
    val1 = stack.pop();
    val2 = stack.pop();
    stack.push(val2 - val1);
    pctr++;
    break;
case SUBLHS:   /* -= */
    val1 = stack.pop();
    addr = stack.pop();
    val2 = dseg.read(addr, pr);
    val = val2 - val1;
    dseg.write(addr, val, pr);
    stack.push(val);
    pctr++;
    break;
case MUL:      /* * */
    val1 = stack.pop();
    val2 = stack.pop();
    stack.push(val2 * val1);
    pctr++;
    break;
case MULLHS:   /* *= */
    val1 = stack.pop();
    addr = stack.pop();
    val2 = dseg.read(addr, pr);
    val = val2 * val1;
    dseg.write(addr, val, pr);
    stack.push(val);
    pctr++;
    break;
case DIV:      /* / */
    val1 = stack.pop();
    if (val1 == 0)
        executeError("Zero divider detected");
    val2 = stack.pop();

```

```

    val = val2 / val1;
    stack.push(val);
    pctr++;
    break;
case DIVLHS: /* /= */
    val1 = stack.pop();
    if (val1 == 0)
        executeError("Zero divider detected");
    addr = stack.pop();
    val2 = dseg.read(addr, pr);
    val = val2 / val1;
    dseg.write(addr, val, pr);
    stack.push(val);
    pctr++;
    break;
case MOD: /* % */
    val1 = stack.pop();
    if (val1 == 0)
        executeError("Zero divider detected");
    val2 = stack.pop();
    val = val2 % val1;
    stack.push(val);
    pctr++;
    break;
case MODLHS: /* %= */
    val1 = stack.pop();
    if (val1 == 0)
        executeError("Zero divider detected");
    addr = stack.pop();
    val2 = dseg.read(addr, pr);
    val = val2 % val1;
    dseg.write(addr, val, pr);
    stack.push(val);
    pctr++;
    break;
case CSIGN: /* 単項 */
    val = stack.pop();
    stack.push(-val);
    pctr++;
    break;

```

```

case AND:      /* and */
    val1 = stack.pop();
    val2 = stack.pop();
    if (val2 != 0 && val1 != 0) stack.push(1);
    else stack.push(0);
    pctr++;
    break;
case OR:       /* or */
    val1 = stack.pop();
    val2 = stack.pop();
    if (val2 != 0 || val1 != 0) stack.push(1);
    else stack.push(0);
    pctr++;
    break;
case NOT:      /* not */
    val = stack.pop();
    if (val == 0) stack.push(1);
    else stack.push(0);
    pctr++;
    break;
case XOR:      /* exclusive or */
    val1 = stack.pop();
    val2 = stack.pop();
    if ((val2 != 0 && val1 == 0) || (val2 == 0 && val1 != 0)) stack.push(1);
    else stack.push(0);
    pctr++;
    break;
case COMP:     /* comp */
    val1 = stack.pop();
    val2 = stack.pop();
    if (val2 == val1) stack.push(0);
    else if (val2 > val1) stack.push(1);
    else stack.push(-1);
    pctr++;
    break;
case COPY:     /* copy */
    val = stack.pop();
    stack.push(val);
    stack.push(val);
    pctr++;

```

```

    break;
case PUSH:    /* push */
    stack.push(dseg.read(operand, pr));
    pctr++;
    break;
case PUSHI:  /* push integer */
    stack.push(operand);
    pctr++;
    break;
case PUSHHP: /* push processor number */
    stack.push(pr);
    pctr++;
    break;
case POP:    /* pop */
    dseg.write(operand, stack.pop(), pr);
    pctr++;
    break;
case REMOVE: /* remove */
    stack.pop();
    pctr++;
    break;
case LOAD:   /* load */
    addr = stack.pop();
    stack.push(dseg.read(addr, pr));
    pctr++;
    break;
case INC:    /* ++ */
    val = stack.pop();
    val++;
    stack.push(val);
    pctr++;
    break;
case DEC:    /* -- */
    val = stack.pop();
    val--;
    stack.push(val);
    pctr++;
    break;
case PREINC: /* 前置 + */
    addr = stack.pop();

```

```

    val = dseg.read(addr, pr);
    val++;
    dseg.write(addr, val, pr);
    stack.push(val);
case PREDEC: /* 前置-- */
    addr = stack.pop();
    val = dseg.read(addr, pr);
    val--;
    dseg.write(addr, val, pr);
    stack.push(val);
case POSTINC: /* 後置++ */
    addr = stack.pop();
    val = dseg.read(addr, pr);
    stack.push(val);
    val++;
    dseg.write(addr, val, pr);
case POSTDEC: /* 後置-- */
    addr = stack.pop();
    val = dseg.read(addr, pr);
    stack.push(val);
    val--;
    dseg.write(addr, val, pr);
case SETFR: /* set frame register */
    freg = operand;
    if (freg > dseg.size)
        executeError("Freg overflow");
    else if (freg < minFreg) minFreg = freg;
    pctr++;
    break;
case INCFR: /* inc frame register */
    freg += operand;
    if (freg > dseg.size)
        executeError("Freg overflow");
    pctr++;
    break;
case DECFR: /* dec frame register */
    freg -= operand;
    if (freg < minFreg) minFreg = freg;
    pctr++;
    break;

```

```

case JUMP:    /* jump */
    pctr = operand;
    break;
case BEQ:    /* ==0 ? */
    val = stack.pop();
    if (val == 0) pctr = operand;
    else pctr++;
    break;
case BNE:    /* !=0 ? */
    val = stack.pop();
    if (val != 0) pctr = operand;
    else pctr++;
    break;
case BLT:    /* <0 ? */
    val = stack.pop();
    if (val < 0) pctr = operand;
    else pctr++;
    break;
case BLE:    /* <=0 ? */
    val = stack.pop();
    if (val <= 0) pctr = operand;
    else pctr++;
    break;
case BGT:    /* >0 ? */
    val = stack.pop();
    if (val > 0) pctr = operand;
    else pctr++;
    break;
case BGE:    /* >=0 ? */
    val = stack.pop();
    if (val >= 0) pctr = operand;
    else pctr++;
    break;
case CALL:   /* call */
    stack.push(pctr);
    pctr = operand;
    callCtr++;
    break;
case RET:    /* return */
    pctr = stack.pop();

```

```

        break;
    case INPUT: /* input integer */
        System.out.print("Integer : ");
        val = Console.ReadInteger();
        stack.push(val);
        pctr++;
        break;
    case INPUTC: /* input character */
        do {
            System.out.print("Character : ");
            str = Console.ReadString();
        } while (str.length()==0);
        val = (int) str.charAt(0);
        stack.push(val);
        pctr++;
        break;
    case OUTPUT: /* output integer */
        val = stack.pop();
        System.out.print(val + " ");
        pctr++;
        break;
    case OUTPUTC: /* output character */
        val = stack.pop();
        if (traceSW)
            switch(val) {
                case '\0':
                    System.out.print("\\0");
                    break;
                case '\b':
                    System.out.print("\\b");
                    break;
                case '\n':
                    System.out.print("\\n");
                    break;
                case '\r':
                    System.out.print("\\r");
                    break;
                case '\t':
                    System.out.print("\\t");
                    break;
            }

```



```

        default:
            System.out.print((char) val);
            break;
    }
    else System.out.print((char) val);
    pctr++;
    break;
case OUTPUTL: /* output lne */
    if (traceSW) System.out.print("\n");
    else System.out.println();
    pctr++;
    break;
case RAND: /* random */
    val = (int) (stack.pop() * Math.random());
    stack.push(val);
    pctr++;
    break;
case PUSHHS:
    str = "";
    for(i=0; true; i++) {
        str += (char) dseg.read(operand+i, pr);
        if (str.charAt(i) == '\0') break;
    }
    for(i=str.length()-1; i>=0; i--)
        stack.push((int) str.charAt(i));
    pctr++;
    break;
case POPS:
    str="";
    for(i=0; true; i++) {
        str += (char) stack.pop();
        dseg.write(operand+i, (int) str.charAt(i), pr);
        if (str.charAt(i) == '\0') break;
    }
    pctr++;
    break;
case ASSGNS:
    str = "";
    for(i=0; true; i++) {
        str += (char) stack.pop();

```

```

        if (str.charAt(i) == '\0') break;
    }
    addr = stack.pop();
    for(i=0; i<str.length(); i++)
        dseg.write(addr+i, (int) str.charAt(i), pr);
    stack.push(str.length());
    pctr++;
    break;
case LOADS:
    str = "";
    addr = stack.pop();
    for(i=0; true; i++) {
        str += (char) dseg.read(addr+i, pr);
        if (str.charAt(i) == '\0') break;
    }
    for(i=str.length()-1; i>=0; i--)
        stack.push((int) str.charAt(i));
    pctr++;
    break;
case INPUTS:
    System.out.print("String : ");
    str = Console.ReadString();
    stack.push('\0');
    for(i=str.length()-1; i>=0; i--)
        stack.push((int) str.charAt(i));
    pctr++;
    break;
case OUTPUTS:
    str = "";
    for (i=0; true; i++) {
        str += (char) stack.pop();
        if (str.charAt(i) == '\0') break;
        if (traceSW)
            switch(str.charAt(i)) {
                case '\0':
                    System.out.print("\\0");
                    break;
                case '\b':
                    System.out.print("\\b");
                    break;
            }
    }

```

```

        case '\n':
            System.out.print("\\n");
            break;
        case '\r':
            System.out.print("\\r");
            break;
        case '\t':
            System.out.print("\\t");
            break;
        default:
            System.out.print((char) str.charAt(i));
            break;
    }
    else System.out.print((char) str.charAt(i));
}
pctr++;
break;
case PARA:
    if (VSM.inParallel)
        executeError("Already in parallel mode");
    int h = stack.pop();
    int l = stack.pop();
    VSM.setParallel(l, h);
    VSM.inParallel = true;
    // PARA ではプログラムカウンタを増加させない
    break;
case SYNC:
    if (!VSM.inParallel)
        executeError("Already in sequential mode");
    isRunning = false;
    // SYNC ではプログラムカウンタを増加させない
    break;
case HALT:
    isRunning = false;
    // HALT ではプログラムカウンタを増加させない
    break;
case EOF:
    executeError("Illegal end of file");
    break;
default:

```

```

        syntaxError("Illegal operator");
    }
    if (traceSW)
        System.out.println(stack);
    }
    return operator;
}
public void setProgramCounter(int addr) {
    pctr = addr;
}

public void incProgramCounter() {
    pctr++;
}

public void awake() {
    isRunning = true;
}

public void syntaxError () {                                /* 文法エラー */
    System.out.println("Processor "+pr);
    System.out.println("Syntax error at line " + pctr);
    iseg.print(pctr);
    System.out.println();
    System.exit(1);
}

public void syntaxError (String err_mes) {                /* 文法エラー */
    System.out.println("Processor "+pr);
    System.out.println("Systax error at line " + pctr);
    System.out.println(err_mes);
    iseg.print(pctr);
    System.out.println();
    System.exit(1);
}

public void executeError () {                              /* 実行時エラー */
    System.out.println("Processor "+pr);
    System.out.println("Execute error at line " + pctr);
    iseg.print(pctr);
}

```

```

        System.out.println();
        System.exit(1);
    }

    public void executeError (String err_mes) { /* 実行時エラー */
        System.out.println("Processor "+pr);
        System.out.println("Execute error at line " + pctr);
        System.out.println(err_mes);
        iseg.print(pctr);
        System.out.println();
        System.exit(1);
    }
}

A.2.10 VSMLexer.java
import java.util.Vector; //ベクトル用

public class VSMLexer implements Operators {
    InputFile inFile; /* InputFile クラスのインスタンス (入力ファイル) */

    //コンストラクタでは、入力ファイルの読み込みと、各種初期化を行う。
    public VSMLexer(String fname) {
        //入力ファイルを開く
        inFile = new InputFile(fname);
    }

    public InstructionSegment makeIseg(boolean debugSW) {
        InstructionSegment iseg = new InstructionSegment(debugSW);
        while (true) {
            Instruction inst = nextOperator();
            if (inst.operator == EOF) break;
            iseg.appendCode(inst);
        }
        for (int i=0; i<iseg.size; i++) iseg.checkAddress(i);
        return iseg;
    }

    public Instruction nextOperator() { /* 字句解析 次のオペレータを得る */
        int operator = NOP;

```

```

int operand = -1;
String str = "";
char c;
boolean csign = false;
boolean existOp = false;

do {
    c = skipSpace();
} while(c == '\n');          /* 空白をスキップ */

if (c == '\0') operator = EOF;          /* End of file */
else if (Character.isUpperCase(c)) {
    str = extractWord(c);
    switch (c) {
        case 'A':
            if (str.equals("ADD")) operator = ADD;          /* ADD */
            else if (str.equals("ADDLHS")) operator = ADDLHS; /* ADDLHS */
            else if (str.equals("AND")) operator = AND;      /* AND */
            else if (str.equals("ASSGN")) operator = ASSGN;  /* ASSGN */
            else if (str.equals("ASSGNS")) operator = ASSGNS; /* ASSGNS */
            else syntaxError("Unknown operator : "+str);
            break;
        case 'B':
            if (str.equals("BEQ")) operator = BEQ;          /* BEQ */
            else if (str.equals("BGE")) operator = BGE;      /* BGE */
            else if (str.equals("BGT")) operator = BGT;      /* BGT */
            else if (str.equals("BLE")) operator = BLE;      /* BLE */
            else if (str.equals("BLT")) operator = BLT;      /* BLT */
            else if (str.equals("BNE")) operator = BNE;      /* BNE */
            else syntaxError("Unknown operator : "+str);
            break;
        case 'C':
            if (str.equals("CALL")) operator = CALL;        /* CALL */
            else if (str.equals("COMP")) operator = COMP;    /* COMP */
            else if (str.equals("COPY")) operator = COPY;    /* COPY */
            else if (str.equals("CSIGN")) operator = CSIGN;  /* CSIGN */
            else syntaxError("Unknown operator : "+str);
            break;
        case 'D':
            if (str.equals("DEC")) operator = DEC;          /* DECFR */

```

```

    else if (str.equals("DECFR")) operator = DECFR;      /* DECFR */
    else if (str.equals("DIV")) operator = DIV;          /* DIV */
    else if (str.equals("DIVLHS")) operator = DIVLHS;    /* DIVLHS */
    else syntaxError("Unknown operator : "+str);
    break;
case 'E':
    if (str.equals("ERROR")) operator = ERROR;          /* ERROR */
    else syntaxError("Unknown operator : "+str);
    break;
case 'H':
    if (str.equals("HALT")) operator = HALT;            /* HALT */
    else syntaxError("Unknown operator : "+str);
    break;
case 'I':
    if (str.equals("INC")) operator = INC;              /* INC */
    else if (str.equals("INCFR")) operator = INCFR;     /* INCFR */
    else if (str.equals("INPUT")) operator = INPUT;     /* INPUT */
    else if (str.equals("INPUTC")) operator = INPUTC;   /* INPUTC */
    else if (str.equals("INPUTS")) operator = INPUTS;   /* INPUTS */
    else syntaxError("Unknown operator : "+str);
    break;
case 'J':
    if (str.equals("JUMP")) operator = JUMP;            /* JUMP */
    else syntaxError("Unknown operator : "+str);
    break;
case 'L':
    if (str.equals("LOAD")) operator = LOAD;            /* LOAD */
    else if (str.equals("LOADS")) operator = LOADS;     /* LOADS */
    else syntaxError("Unknown operator : "+str);
    break;
case 'M':
    if (str.equals("MOD")) operator = MOD;              /* MOD */
    else if (str.equals("MODLHS")) operator = MODLHS;   /* MODLHS */
    else if (str.equals("MUL")) operator = MUL;         /* MUL */
    else if (str.equals("MULLHS")) operator = MULLHS;   /* MULLHS */
    else syntaxError("Unknown operator : "+str);
    break;
case 'N':
    if (str.equals("NOP")) operator = NOP;              /* NOP */
    else if (str.equals("NOT")) operator = NOT;         /* NOT */

```

```

    else syntaxError("Unknown operator : "+str);
    break;
case 'O':
    if (str.equals("OR")) operator = OR;           /* OR */
    else if (str.equals("OUTPUT")) operator = OUTPUT; /* OUTPUT */
    else if (str.equals("OUTPUTC")) operator = OUTPUTC; /* OUTPUTC */
    else if (str.equals("OUTPUTL")) operator = OUTPUTL; /* OUTPUTL */
    else if (str.equals("OUTPUTS")) operator = OUTPUTS; /* OUTPUTS */
    else syntaxError("Unknown operator : "+str);
    break;
case 'P':
    if (str.equals("PARA")) operator = PARA;       /* PARA */
    else if (str.equals("POP")) operator = POP;    /* POP */
    else if (str.equals("POPS")) operator = POPS;  /* POPS */
    else if (str.equals("POSTDEC")) operator = POSTDEC; /* POSTDEC */
    else if (str.equals("POSTINC")) operator = POSTINC; /* POSTINC */
    else if (str.equals("PREDEC")) operator = PREDEC; /* PREDEC */
    else if (str.equals("PREINC")) operator = PREINC; /* PREINC */
    else if (str.equals("PUSH")) operator = PUSH;  /* PUSH */
    else if (str.equals("PUSHI")) operator = PUSHI; /* PUSHI */
    else if (str.equals("PUSHP")) operator = PUSHP; /* PUSHP */
    else if (str.equals("PUSHS")) operator = PUSHS; /* PUSHS */
    else syntaxError("Unknown operator : "+str);
    break;
case 'R':
    if (str.equals("RAND")) operator = RAND;      /* RAND */
    else if (str.equals("RET")) operator = RET;   /* RET */
    else if (str.equals("REMOVE")) operator = REMOVE; /* REMOVE */
    else syntaxError("Unknown operator : "+str);
    break;
case 'S':
    if (str.equals("SETFR")) operator = SETFR;    /* SETFR */
    else if (str.equals("SUB")) operator = SUB;   /* SUB */
    else if (str.equals("SUBLHS")) operator = SUBLHS; /* SUBLHS */
    else if (str.equals("SYNC")) operator = SYNC; /* SYNC */
    else syntaxError("Unknown operator : "+str);
    break;
case 'X':
    if (str.equals("XOR")) operator = XOR;       /* XOR */
    else syntaxError("Unknown operator : "+str);

```



```

        break;
    default:
        syntaxError("Unknown operator : "+str);
        break;
    }
} else syntaxError("Illegal character : "+c);

c = skipSpace();                                /* 空白をスキップ */

if (c == '-') {                                  /* 符号 */
    csign = true;
    c = inFile.nextChar();
}
if (c == '0') {                                  /* 符号無し整数 (0) */
    operand = 0;
    c = skipSpace();
    if ( c != '\n' && c != '\0' ) syntaxError("Illegal character : "+c);
    existOp = true;
} else if (Character.isDigit(c)) {               /* 符号無し整数 */
    operand = extractIntValue(c);
    c = skipSpace();
    if ( c != '\n' && c != '\0' ) syntaxError("Illegal character : "+c);
    existOp = true;
} else if (c != '\n' && c != '\0' )
    syntaxError("Illegal character : "+c);

if (csign) operand *= -1;
checkOperand(operator, operand, existOp);

return new Instraction(operator, operand);
}

public int extractIntValue(char c) {             /* c で始まる整数を得る */
    int v = Character.digit(c, 10);             /* 文字を整数に変換 */
    while (Character.isDigit(inFile.nextc)) {
        c = inFile.nextChar();
        v = v * 10 + Character.digit(c, 10);
    }
    return v;
}
}

```

```

public String extractWord(char c) {          /* cで始まる文字列を得る */
    String s = String.valueOf(c);
    while (Character.isUpperCase(inFile.nextc)) {
        c = inFile.nextChar();
        s = s + c;
    }
    return s;
}

```

```

public char skipSpace() {                  /* 空白をスキップ */
    char c;
    do {
        c = inFile.nextChar();
        if (c == '#')
            do {
                c = inFile.nextChar();
            } while (c != '\n' && c != '\0');
        if (c == '\n' || c == '\0') break;
    } while (c == ' ' || c == '\t');
    return c;
}

```

```

public void checkOperand(int operator, int operand, boolean e) {
    switch(operator) {
        case NOP:
        case ASSGN:
        case ADD:
        case ADDLHS:
        case SUB:
        case SUBLHS:
        case MUL:
        case MULLHS:
        case DIV:
        case DIVLHS:
        case MOD:
        case MODLHS:
        case CSIGN:
        case AND:
        case OR:

```

```
case NOT:
case XOR:
case COMP:
case COPY:
case PUSH:
case REMOVE:
case LOAD:
case INC:
case DEC:
case PREINC:
case PREDEC:
case POSTINC:
case POSTDEC:
case RET:
case INPUT:
case INPUTC:
case OUTPUT:
case OUTPUTC:
case OUTPUTL:
case RAND:
case ASSGNS:
case LOADS:
case INPUTS:
case OUTPUTS:
case PARA:
case HALT:
case SYNC:
case EOF:
    if (e)
        syntaxError("Illegal operand" + operand);
    break;
case PUSH:
case PUSHI:
case POP:
case SETFR:
case INCFR:
case DECFR:
case JUMP:
case BEQ:
case BNE:
```

```

    case BLT:
    case BLE:
    case BGT:
    case BGE:
    case CALL:
    case PUSHHS:
    case POPS:
        if (!e)
            syntaxError("Illegal operand");
        break;
    default:
        syntaxError("Illegal operator");
        break;
    }
}

public void syntaxError() { /* 文法エラー */
    System.out.println("Systax error at line " + inFile.linenum);
    System.out.println(inFile.line);
    inFile.closeFile();
    System.exit(1);
}

public void syntaxError(String err_mes) { /* 文法エラー */
    System.out.println("Systax error at line " + inFile.linenum);
    System.out.println(err_mes);
    System.out.println(inFile.line);
    inFile.closeFile();
    System.exit(1);
}
}
\newpage

```

### A.3 $n$ 個のデータの和を計算する PRAM 用並列言語プログラム

```

main(){
    int a[16];
    int i;

    parallel(0,15)

```

```
a[$p]=$p+1;

for(i=1; i<16; i*=2)
  parallel(0,16/2/i)
    a[$p]=a[2*$p]+a[2*$p+1];

write(a[0]);
}
```

#### A.4 付録3をPRAMコンパイラによって変換した並列アセンブラプログラム

0: PUSHI	0	30: PUSHI	8
1: PUSHI	15	31: PUSH	16
2: PARA		32: DIV	
3: PUSHI	0	33: PARA	
4: PUSHHP		34: PUSHI	0
5: ADD		35: PUSHHP	
6: PUSHHP		36: ADD	
7: PUSHI	1	37: PUSHI	0
8: ADD		38: PUSHI	2
9: ASSGN		39: PUSHHP	
10: REMOVE		40: MUL	
11: SYNC		41: ADD	
12: PUSHI	16	42: LOAD	
13: PUSHI	1	43: PUSHI	0
14: ASSGN		44: PUSHI	2
15: REMOVE		45: PUSHHP	
16: PUSH	16	46: MUL	
17: PUSHI	16	47: PUSHI	1
18: COMP		48: ADD	
19: BGE	56	49: ADD	
20: JUMP	29	50: LOAD	
21: PUSHI	16	51: ADD	
22: COPY		52: ASSGN	
23: LOAD		53: REMOVE	
24: PUSHI	2	54: SYNC	
25: MUL		55: JUMP	21
26: ASSGN		56: PUSH	0
27: REMOVE		57: OUTPUT	
28: JUMP	16	58: HALT	
29: PUSHI	0		