システムＬＳＩ設計技術　107－3
(2002. 11. 28)

Java言語を使ったプロセッサのサイクルアキュレイト・モデリング

嶋　正利† 篠崎　新† 佐藤　格†

†会津大学コンピュータ理工学部 〒965-8580 福島県会津若松市一箕町鶴賀
E-mail: †(shima, m5061114, m5061112)@u-aizu.ac.jp

あらまし

本論文では、論理構成に近いモデリングとトップダウン設計に適した、Java 言語を使用したサイクルアキュレイト・プロセッサ・モデルのモデリング手法を提案する。本研究では、以前開発した5段のパイプライン・プロセッサのRTLモデルと同一の論理構造を採用し、オブジェクト指向言語であるJava言語を使用してモデルの構築を行う。モデリング時に生じるモジュール間の依存関係やモデルの階層構造に適したモデリングルールを導入し、Java 言語のオブジェクト指向性を使った論理構成に近いモデリングを行い、Java言語のモデリング言語としての適用性を評価する。PC上でプロセッサ・モデルの動作検証を行い、シミュレーション結果を報告し、Java言語のモデリング言語としての有用性を示す。

キーワード　プロセッサ、プロセッサモデリング、サイクルアキュレイトモデル、Java言語

Cycle-Accurate Processor Modeling Written in Java Language

Masatoshi SHIMA† Arata SHINOZAKI† and Tadashi SATO†

† Faculty of Computer Science and Engineering, University of Aizu, Tsuruga Ikki-machi, Aizuwakamatsu-City, Fukushima 965-8580 Japan
E-mail: †(shima, m5061114, m5061112)@u-aizu.ac.jp

Abstract

This paper proposes a modeling methodology for a cycle-accurate processor model, written in Java Language, suitable for top-down development and modeling close to the logical structure of a processor. In this research, the same logical structure used for a previously developed RTL model of five-stage pipelined processor is adopted, and the processor model is designed in Java Language, which is an object-oriented language. First, modeling rules suitable for the hierarchy of the model and the dependency among modules triggered by the modeling description are introduced. Then, the processor model is designed, focusing on the object-orientation of Java Language. Finally, the suitability of Java Language as a modeling language is evaluated. The function of this processor model is validated on a PC. The result of simulation shows the effectiveness of Java Language as a modeling language.

Keyword Processor, Processor Modeling, Cycle-Accurate Model, Java Language

1. はじめに

プロセッサのモデルを作成し、シミュレーションすることにより、プロセッサの機能検証と性能評価を行うことが可能である。モデルにはシステムの検証を行うRTLモデル、応用プログラムの検証を行うサイクルアキュレイトモデル[1][2]、アルゴリズムの検証を行う命令モデルがある。ハードウェア記述言語で記述したRTLモデルはハードウェアに忠実な検証が可能であるが、RTLシミュレーションでは結果を波形表示するため、検証に要する時間が非常に長く、応用プログラムの検証が難しい。一方、SOC (System-On-a Chip)時代に向け、アプリケーションに特化したプロセッサ指向が強まり、数十万から数百万クロックを必要とする検証の頻度が高くなっている。そのため、RTLモデルと完全に等価で、ハードウェアに忠実かつ高速な検証が可能なサイクルアキュレイトモデルが必要となる。

サイクルアキュレイトモデルの記述には、ハードウェア・コンポーネントとオブジェクトを一対一で記述可能な、オブジェクト指向言語が最適である。現在、主流のオブジェクト指向言語はC++とJavaである。実行速度を重視したモデルには、コンパイラによって最適化されたオブジェクト指向言語であるC++が広範囲に用いられている。

本論文では、サイクルアキュレイトモデルの記述言

語としてJava言語を提案する[3]。Java言語を採用するのは、性能ではなく、開発効率と論理の明確性を重視するためである。Java言語の有用性[4][5]を二つ示す。一つは、Java言語にはガーベジコレクション、ポインタの削除などの機構があり、C++より利便性に優れ、プログラミング効率が大幅に向上している。二つ目は、Java言語はC++よりもオブジェクト指向性が強い。

本研究では、Java言語を使ってサイクルアキュレイト・プロセッサ・モデルを構築し、モデルの動作検証と性能評価を行い、Java言語のモデリング言語としての有用性を検討する。プロセッサには、システムへの組み込みを前提に、IF、ID、EX、MA、WBの5ステージで構成するパイプライン・プロセッサを、命令セットにはMIPS整数命令を採用する。各ユニットはデータパス本体部と制御本体部で構成する。データパス本体部はレジスタ、マルチプレクサ、演算器、シフタなどのコンポーネントから構成する。制御本体部はデータパスと同様の論理コンポーネントと新規に開発した真理値表セルライブラリ[6]を使った主制御部で構成する。

本論文ではプロセッサを例に挙げ、Java言語の持つオブジェクト指向性を使って、論理構成により近いモデリング手法を提案する。はじめにモデリングの標準化を行う3つのモデリングルールについて述べる。第2章ではモデルに時間の概念を導入し、RTLモデルとの互換性を保持する信号やレジスタのネーミングルールの定義について述べ、第3章でJava言語の記述とハードウェア・コンポーネントの対応付けと、ハードウェアの階層構造の実現について述べ、第4章ではモデルの抽象度を高める抽象クラスを用いた、基本クラスの導入による、機能と実装の分離について述べる。第5章ではモデリングルールを用いて実装したモデルの開発について述べる。第6章ではプロセッサ全体の実装について述べる。第7章では実装したモデルのシミュレーション結果を報告し、評価について述べる。

2. ネーミングルールの定義

各クロックでプロセッサが実行する機能、信号、レジスタ、演算結果の詳細な検討を行うため、クロック・サイクル精度を必要とするサイクルアキュレイトモデルでは時間の概念が必要である。モデリングには時間の概念がないJava言語などの高級言語を用いるため、各クロック・サイクルを一回のループに対応付けてモデリングを行い、時間の概念を導入する。各ループでは、機能を構成するすべてのコンポーネントを逐次的に実行するため、相互のコンポーネントの依存関係に起因した実行順序の制約を考慮し、時間の概念を明確にする、ハードウェア・コンポーネントと等価なモデリングを行うネーミングルールを定義した。

表1．モデルのネーミングルール

名前	対象となる要素
修飾子なし	そのクロックで使用するレジスタとフリップフロップ出力信号。同一クロック内では、すべてのコンポーネントにおいて使用できる。
next_	次のクロックで使用するレジスタとフリップフロップ出力信号。次のクロックに実行が遷移する、ループ終了直前に、修飾子のない信号へ代入する。
thru_	レジスタとフリップフロップ出力以外の通常の出力信号。実行順序に依存し、実行順序を遡って用いることはできない。
_int	モジュール内部で帰納的に用いる信号。
my***Reg	レジスタ自体を指す。信号とレジスタを区別する。
my***FF	フリップフロップ自体を指す。

ネーミングルールを表1に示す。信号やレジスタなど、各コンポーネントの名称は以前開発したRTLモデルと互換性を保持した。通常の出力にはthru_という接頭辞を、レジスタ自身にはmyという接頭辞とRegという接尾辞をそれぞれRTLモデル上の名前に付加する。レジスタに格納する値のネーミングは特に重要である。格納した値を次のクロックで使用するレジスタ出力は、next_という接頭辞を付加して区別する。

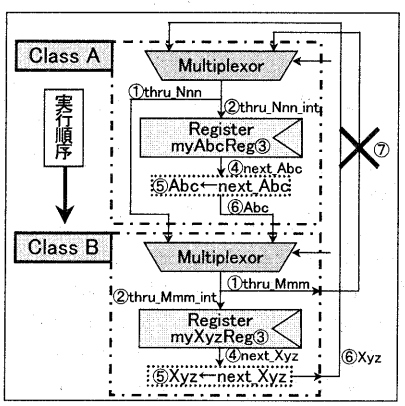

図1．ネーミングルールと実行順序

信号の受け渡しの様子と実行順序を図1に示す。マルチプレクサとレジスタで構成する二つのモジュールに対応したクラスAとBを宣言する。マルチプレクサによって選択された信号は通常の信号(thru_信号)とする(①)。同一の信号をクラスの内部と外部両方で使用する場合に、配列として値を出力する外部信号と区別し、整数値として送信する内部信号に_intという接尾辞をつける(②)。レジスタ(③)に格納した値は次のクロックで使用する信号(next_信号)として発行する(④)。すべてのコンポーネントを逐次処理し、ループを終了すると、next_信号を修飾子のないレジスタ出力へ代入して、レジスタ出力の更新を行う(⑤)。レジスタ出力は次の更新まで値を変更しないため、実行順序

に依存せず、すべてのクラスで使用可能である(⑥)。一方、thru_信号は信号を生成したコンポーネントより時間的に後で実行するコンポーネントでのみ使用可能で(⑦)、次のクロックで無効となる。

3. Java 言語による階層構造の記述

オブジェクト指向言語である Java 言語はクラス単位で記述する。クラスは、変数にあたるフィールドと関数にあたるメソッドで構成する。クラス内のフィールドやメソッドを直接使用できるが、クラスの実体としてインスタンスを生成し、インスタンスを介してフィールドとメソッドを使用する。

C++よりもオブジェクト指向性に優れた Java は以下の 3 つの特徴と有用性を持つ。

Java は継承に対する制約が厳しく簡潔な階層構造を持つ。Java では子クラスが親クラスの属性を受け継ぎ、継承を行う。継承には、一つの子クラスが一つの親クラスからのみ継承する単一継承を採用している。Java は、C++と異なり、多重継承を行わず、単一継承の機構を用いて階層的な記述を行うため、クラスの階層構造が複雑にならず、常に明確である。

また、Java はクラス自体の結合が強い。C++はメンバ関数をクラス外で宣言できるが、Java はクラス外にフィールドやメソッドを定義できないため、クラスの結合が強化され、オブジェクト指向性が強くなる。従って、曖昧なクラス定義が不可能となり、クラスと対応付けたモジュールの再利用性が高くなる。

C++はモデルを作成する際に多くの制約を定める必要があったが、Java 言語では、言語そのものに制約があるため、別途に制約を設ける負担が軽減する。

基本的に階層構造の記述は次のように行う。任意のモジュールに対応するクラスは任意のコンポーネントや下位階層のモジュールで構成する。任意のモジュールは、更に上位階層のモジュールの一部としてインスタンス化し、記述できる。モデルではモジュール毎にクラスを割り当てて宣言する。クラス内で下位のモジュールを使用する場合は対応するクラスのインスタンスを生成し、使用する。

モジュールは各クラスに実行用のメソッドを設け、インスタンスを介してメソッドを呼び出し、実行する。従って、モジュールの機能はすべてメソッド内に記述する。入出力はすべてメソッドの引数として送受信する。

各々の信号やレジスタはすべてフィールドとして宣言する。原則として、信号には整数値を用いるが、真理値表のデータは文字列として入出力を行い、必要に応じて整数値に変換する。

複数のモジュールをまとめ、グループを形成する場合は、パッケージを用いる。パッケージとは、プログラムで使用する名前の衝突を防ぎ、アクセス制御を行い、プログラムの再利用性を高める Java 言語の機構である。クラスの記述は階層的に行うが、物理的にはクラスは階層的に配置せず、パッケージでグループを形成し、同一階層上に構築する。

例えば命令フェッチ(IF)ユニットを形成するすべてのモジュールを一つのグループとしてまとめるために、IF パッケージを生成する。IF ユニットに関係のないクラスは IF パッケージのクラスに対してアクセス権が限定され、パッケージ外のクラスはパッケージ内の public というキーワードが付いたクラスやメソッド、フィールドにのみアクセス可能である。パッケージ毎に独立を保ち、インタフェースのみを公開した IP としてのモジュールの構築が可能となる。

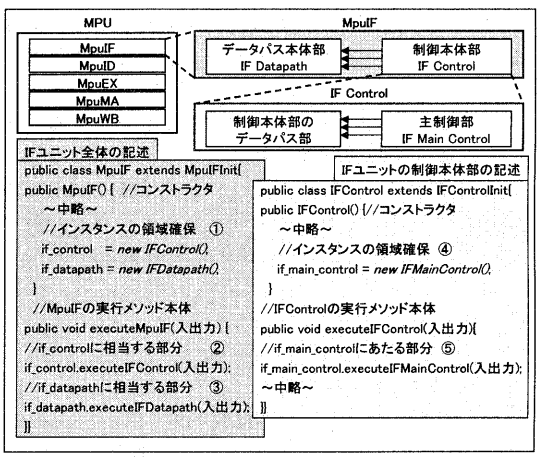

図２．階層構造の記述例(IF ユニット)

IF ユニットの階層構造の記述例を図２に示す。IF ユニットは、データパス本体部と制御本体部で構成する。IF ユニットの記述では、別のクラスで定義したデータパス本体部(IFDatapath)と制御本体部(IFControl)のインスタンスを、コンストラクタ内で new というキーワードを使って生成する(①)。IF ユニットの実行メソッドではインスタンスを介してデータパス本体部と制御本体部の実行メソッドを実行順序に従って呼び出す(②、③)。制御本体部は簡単なデータパス部と主制御部で構成する。主制御部は別のクラス(IFMainControl)で実装するため、制御本体部のコンストラクタでインスタンスを生成し(④)、実行メソッドを呼び出す(⑤)。

4. 基本クラスの導入

C++と異なり Java にはヘッダが存在しない。C++で

はヘッダを定義し、信号やレジスタの宣言を行う。Javaではヘッダという曖昧なプログラミング構造によるオブジェクト指向性の崩壊がなく、フィールドとメソッドをすべてクラス内で宣言し記述するため、論理構造が簡潔になる。

モデルを抽象化し、プロセッサの構造と互換性を保持したプログラミングを行うために、ヘッダの代替として抽象クラス(abstract class)を導入する。抽象クラスは Java 言語に用意されている実体のないクラスで、プログラムの抽象度を高め、実装を行うクラスに継承させて、クラス階層を整える。抽象クラスではレジスタや信号に対応するフィールド、下位階層のモジュールに対応するクラスのインスタンス、モジュールの機能に対応する実行メソッドを宣言する。実装するフィールド、インスタンス、メソッドを宣言し、クラスの基本的な構造を記述するため、このクラスを基本クラス (Basic Class) と呼ぶ。それに対して、基本クラスを継承し、実装を行うクラスを本体部クラスと呼ぶ。

基本クラスの導入によって、C++モデルで曖昧であった機能と実装の分離が容易となる。基本クラスを使って、設計すべき対象とリソースの定義と割り当てを行い、ハードウェア・アーキテクチャの設計を行う。本体部クラスではアーキテクチャの実装を行う。

5. モデルの実装

モデルの実装は JDK1.3 (Java Development Kit Version 1.3) を使用して行った。開発環境としてバージョン管理やデバッキング環境が整った Borland 社の JBuilder 5 Enterprise を使用した。開発には Pentium III 1 GHz と 512MB のメモリを搭載した PC と、OS として Windows 2000 Professional を使用した。

IF ユニットの制御本体部の RTL モデル(図３)、Java 言語を使ったモデルの基本クラス (図４)と本体部クラス(図５)を例として述べる。IF ユニットの制御本体部は簡単なデータパス部と真理値表セルライブラリを実装した主制御部で構成した。データパス部は命令プリフェッチバッファ、命令のソースを選択するマルチプレクサ、および命令レジスタで構成し、主制御部で発行する制御信号に従って処理を行う。

基本クラスでは本体部クラスのアーキテクチャ設計を行い、レジスタ(a)、レジスタ出力(b)、通常出力(c)などのフィールド、内部で受け渡しを行う信号の領域(e)、下位モジュールのインスタンス(f)、本体部クラスで実装する実行メソッド(g)などを宣言する。特に次のクロックで使用するレジスタの出力値を一時的に保持するフィールドを next_信号として宣言する(d)。

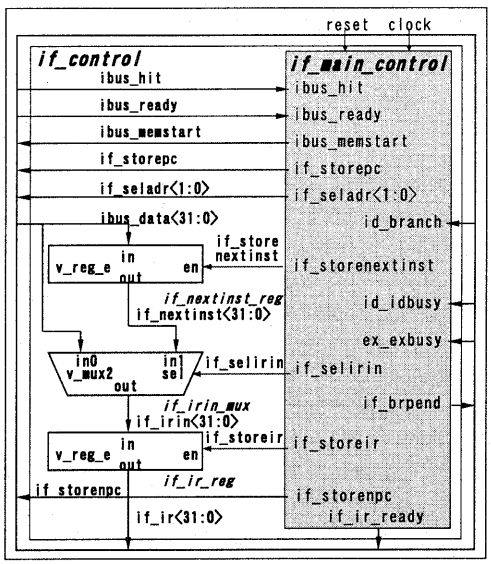

図３．IF ユニットの制御部の RTL モデル

```
abstract class IFControlInit {
  int myIFNextinstReg; //レジスタ          (a)
  public int myIFIrReg;  //レジスタ
  int if_nextinst;          //レジスタ出力 (b)
   int thru_if_irin;        //通常出力     (c)
  //レジスタ出力の一時記憶フィールド       (d)
  int next_if_nextinst;
  //内部で受け渡しを行う信号               (e)
  short[] thru_if_storenextinst;
  short[] thru_if_selirin;
  short[] thru_if_storeir;
  //モジュールインスタンスの宣言           (f)
  IFMainControl if_main_control;
   //IFControl実行メソッドの宣言           (g)
  abstract void executeIFControl(
   short clock,                  //入力
     ～中略～
    short[] next_if_brpend); //出力
}
```

図４．基本クラスの記述例

本体部クラスは各モジュールに対応付けて宣言した(①)。本体部クラスは基本クラスを継承し、拡張(extends)したものである。基本クラスから継承したフィールドのうち、他のクラスのインスタンスからの出力を入力として受け取る信号は、コンストラクタで領域確保を行う。この例では下位のモジュールである主制御部で発行し、データパス部のレジスタやマルチプレクサで使用する制御信号の領域を確保する(②)。主制御部のインスタンスはコンストラクタ内で領域確保を行う(③)。

モジュールの機能はメソッドによって記述する。この機能実行用のメソッドを execute メソッドと呼ぶ(④)。execute メソッドの引数に、対応するモジュールの入出力信号を記述する。入力は整数値の引数として渡す。JDK1.3 では引数の参照渡しが不可能であるため、出力は配列として受け渡す。下位のモジュールで

ある主制御部は別のクラスで実装するため、そのクラスのインスタンスを作成し、インスタンスを介してexecuteメソッドを実行する(⑤)。

データパス部の記述は言語への依存度が低い。Javaによる記述もHDL言語による記述と同様に、レジスタにはif文(⑥)を用い、マルチプレクサにはswitch文(⑦)を用いて記述した。レジスタ出力の更新はexecuteメソッドの末尾で行う(⑧)。

```
public class IFControl extends IFControlInit{  ①
//コンストラクタ 各フィールドの領域を確保する
 public IFControl() {
  //outputとしてわたすフィールドの領域確保   ②
  thru_if_storenextinst = new short[1];
  thru_if_selirin       = new short[1];
  thru_if_storeir       = new short[1];
  //モジュールインスタンスの領域確保          ③
  if_main_control = new IFMainControl();
 }
 //IFControlの実行メソッド本体               ④
 public void executeIFControl( short clock,  //入力
                        ～中略～
                    short[] next_if_brpend ){//出力
 //if_main_controlの実行                     ⑤
 if_main_control.executeIFMainControl( reset, //入力
                        ～中略～
                thru_if_storenpc ); //出力
 //if_nextinst_regに相当する部分            ⑥
 if(thru_if_storenextinst[0] == 1){
  myIFNextinstReg = thru_ibus_data;
  next_if_nextinst = myIFNextinstReg;
 }
 //if_irin_muxに相当する部分                ⑦
 switch(thru_if_selirin[0]) {
  case 0:  thru_if_irin = thru_ibus_data; break;
  case 1:  thru_if_irin = if_nextinst;    break;
  default:                                break;
 }
 //if_ir_regに相当する部分                  ⑥
 if(thru_if_storeir[0] == 1){
  myIFIrReg  = thru_if_irin;
  next_if_ir[0] = myIFIrReg;
 }
 //レジスタ出力の値を更新                   ⑧
 if_nextinst = next_if_nextinst;
}
```

図5．本体部クラスの記述例

主制御部には真理値表セルライブラリを使用する。主制御部では、他のモジュールと同様に入力を受け付け、真理値表の評価を行うクラスのメソッドへ入力値と真理値表のデータやサイズ情報を渡す。真理値表と入力値の比較を行いヒットした行の出力値を返す。ヒットした行が複数ある場合は出力値の論理和を出力する。

6. プロセッサ・モデルの実現

各ユニットの実行順序と信号やデータの流れを図6に示す。数字は実行順序を表す。各ユニット間のインタフェースとしてパイプライン・レジスタを設けた。データパス本体部には演算のオペランドなど各種データ用、制御本体部にはマイクロ命令用のパイプライン・レジスタを設けた。制御本体部は、マイクロ命令を発行する準備が整うとマイクロ命令実行開始許可信号(Ready 信号)と共に、マイクロ命令を後続ユニットの制御本体部へ発行する。これらのパイプライン・レジスタに格納した値は実行順序に依存しないので、IFユニットからWBユニットへ処理を行うことができる。後続のユニットから先行するユニットに向かってフィードバックする信号やデータは通常の出力（thru_信号）として扱う。従って、5つのユニットを用いて構成したプロセッサではWBユニットから実行を開始し、IFユニットで実行を終了させる。

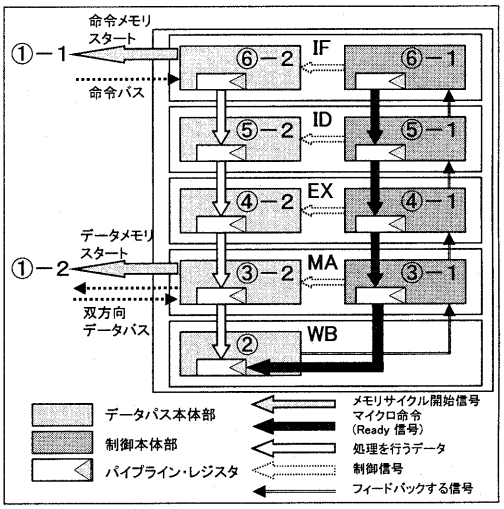

図6．プロセッサ内の各ユニットの実行順序

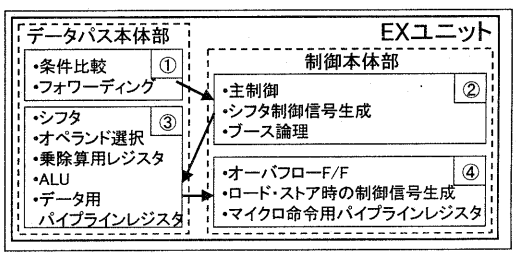

図7．EXユニットの実行順序

ライトバック(WB)以外のユニットはデータパス本体部と制御本体部で構成した。原則として、制御本体部で生成した信号をデータパス本体部へ発行し、データパス本体部の処理を行う。受け渡す信号の種類と実行順序によって原則が適用できない場合は、実行順序にあわせて分割した複数のメソッドを作成し、実行順序に応じて、上位モジュールのクラスから各メソッドを呼び出した。各メソッドはデータパス本体部や制御本体部などのクラスに属し、ハードウェア・コンポーネントとの対応を保持した。

分割したメソッドの例としてEXユニットのexecuteメソッドと実行順序を図7に示す。EXユニットは、条

件比較、フォワーディング、シフタ、ALU などで構成するデータパス本体部と、主制御部、各種制御信号生成部、ブース論理、オーバーフロー・フリップフロップなどで構成する制御本体部の 2 つで構成した。

主制御部で比較結果を使用するため、データのフォワーディングと、条件分岐時の条件比較を先行して実行する必要がある(①)。主制御部での制御が確定すると、データパス本体部へ制御信号を発行する(②)。次に、シフタと ALU での処理を行う(③)。最後に ALU で生成したオーバーフローやアドレス情報を用いて、オーバーフロー検出信号を格納し、次段へのマイクロ命令を生成する(④)。EX ユニットのクラスから①～④のメソッドを逐次呼び出し、実行する。①と③のメソッドはデータパス本体部のクラスに、②と④のメソッドは制御本体部のクラスに記述し、ハードウェア・コンポーネントとの対応を保った。

例外的に、プロセッサのクラス内に記述した IF ユニットと MA ユニットからメモリサイクル開始信号を発行するメソッドは、プロセッサの各ユニットに先立って実行する。メモリサイクルの開始信号をループの先頭で発行して、メモリサイクルを開始し、最終的にプロセッサで入力データを処理する。

データバスは双方向であるため、モデリングは入力と出力の 2 つのバスに分割して行った。入力データの制御はバスリゾルバを設けて行った。プロセッサからのデータ出力は直接出力する。内部データ用 SRAM や BIU(Bus Interface Unit)からプロセッサへの入力データは、ドライブ情報を付加してバスリゾルバに発行し、ドライブ情報を使用して選択したデータを、入力データとしてプロセッサへ送出した。

7. 評価

本論文では、オブジェクト指向性を持つ Java 言語を用いて、論理構成に近い、サイクルアキュレイトモデルのモデリング手法を提案した。

ユニット毎の 5 つのモデルとプロセッサ・モデルに対してテストベンチを作成し、テキストベースで動作検証を行った。開発したサイクルアキュレイト・プロセッサモデルのシミュレーション性能を表 2 に示す。1 GHz の PC を使用して、1,746 クロック/秒の性能を得た。330MHz のワークステーション上で動作させた RTL システムモデルに対して約 2 桁の性能向上が見られる。以前開発した C++モデルと比較すると動作性能は約 1/5 となるが十分に実用的な性能を示した。

ネーミング、階層構造の記述、基本クラスに関する 3 つのモデリングルールの導入により、モデリングを一般化し、Java 言語のトップダウン設計への適用性を示した。Java 言語のオブジェクト指向性を生かした、基本クラスによる機能と実装の分離により、論理構成に忠実なモデリングを行い、Java 言語のモデリング言語としての有用性を示した。

表 2．サイクルアキュレイトモデルの性能
(clocks/sec)

	PC	WS
プロセッサ型名	Pentium III	UltraSparc IIi
動作周波数(MHz)	1,000	330
主メモリ容量(MB)	512	256
プロセッサ・モデル	1,746	
RTL システムモデル		9

8. おわりに

Java 言語によるモデルは性能の低さが指摘されていたが、プロセッサ・モデルでは 1GHz の PC で 1,746 クロック/秒の性能を得た。実用に十分な性能に、論理構成に近いモデリングやプログラミング時の生産性、簡潔な論理構造を加味すると、Java 言語は開発効率に優れ、モデリングに適した言語である。

モデリングルールの確立により、モデルの記述を統一し、メモリや、システムバス、BIU (Bus Interface Unit) を含んだシステム・モデルへの拡張が容易となる。Swing などの GUI コンポーネントが充実した Java 言語のライブラリを使用すれば、GUI を搭載したシステム・モデル実現への道が開け、Java 言語のみを使用して、システム・モデルのシミュレーション効率の向上が可能となる。

文　献

[1] Maturana, G., Ball, J.L., Gee, J., Iyer, A. and O' Conner, J.: Incas: A Cycle Accurate Model of UltraSparc, Proc. 1995 International Conference Computer Design, pp. 130-135, 1995.

[2] Hangal, S., and O' Conner, M.: Performance Analysis and Validation of The PicoJava Processor, IEEE Micro, pp.66-72 May-June, 1999.

[3] 嶋正利、大田優、篠崎新、伊藤和哉、" Java 言語を使ったサイクルアキュレイト・プロセッサ・モデリング," 電気関係学会東北支部連合大会・講演論文集, p.125, 2002 年 8 月

[4] 結城 浩," Java 言語プログラミングレッスン(上・下)," ソフトバンク パブリッシング, 東京, 1999.

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, 村上雅章 訳 "Java™言語仕様 第 2 版," ピアソンエジュケーション, 東京, 2000.

[6] 伊藤和哉、篠崎新、大田優、嶋正利," Development of Truth Table Cell Library Suitable for the Logic Design of a Control Block," 電気関係学会東北支部連合大会・講演論文集, p.21, 2002 年 8 月