システムＬＳＩ設計技術　96－5
（2000. 5. 12）

ASIP 開発システム PEAS-III のための命令セットレベルシミュレータの自動生成

前田 友英　武内 良典　今井 正治　北嶋 暁

大阪大学 大学院基礎工学研究科 情報数理系専攻
〒 560-8531 大阪府豊中市待兼山町 1-3
TEL: 06-6850-6626
FAX: 06-6850-6627
E-mail: peasv@vlsilab.ics.es.osaka-u.ac.jp

あらまし　ASIP 開発システム PEAS-III で入力されたパイプラインプロセッサの仕様記述から命令セットレベルシミュレータを生成する手法を考案した．本手法では，各パイプラインステージを同一のモデルで表現し，そのモデルを SW 言語で記述することによりシミュレータを実現する．モデルの記述言語には SystemC を用いる．これにより，並列動作するパイプラインステージの SW 言語によるモデリングが容易に行え，またサイクル単位のシミュレーションを行うカーネルを利用することで，パイプライン動作を正確かつ短時間でシミュレート可能となる．評価実験により，PEAS -III の生成したプロセッサに対し，本手法により生成されたシミュレータは，VHDL シミュレーションと比較し実行サイクル数の見積りを約 1/7 の時間で行えることを示した．

キーワード　ASIP，シミュレータ生成，SystemC，PEAS-III

Automatic Generation of an Instruction-Set Level Simulator with the ASIP Development System PEAS-III

Tomohide Maeda　Yoshinori Takeuchi　Masaharu Imai　Akira Kitajima

Department of Informatics and Mathematical Science
Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka
Osaka, 560-8531 Japan
TEL: +81 6 6850 6626
FAX: +81 6 6850 6627
E-mail: peasv@vlsilab.ics.es.osaka-u.ac.jp

Abstract　An automatic generation method of an instruction-set level simulator with the ASIP development system PEAS-III is proposed. In this method, the pipeline processor model which is used in PEAS-III is represented in a software language and the program derived from the model can be execute as a simulator for a given processor description. Since SystemC is the language for simulators in our method, modeling of pipeline stages as concurrent processes is easy. In addition, with the cycle-based simulation kernel enables accurate and shorter-time simulation for pipeline processors. Experimental results show that simulation time of our simulator for a processor generated with PEAS-III is around 7 times shorter than a VHDL simulator's.

Key words　ASIP, Simulator Generation, SystemC, PEAS-III

1 はじめに

特定用途向き命令セットプロセッサ (ASIP) の設計では，その用途に応じてプロセッサの適切なアーキテクチャや命令セットを決定する必要があるが，一般にアーキテクチャと命令セットの組み合わせは膨大な数となるため，従来手法を用いた場合，設計工数が増大する可能性がある．そこで，ハードウェア/ソフトウェア・コデザイン手法を用いて，最適な構成を持つASIPの設計をより短期間で効率良く行うことが試みられている [1][2][3].

このようなASIP設計の工程には，適切な命令セットを決定するためのプロセッサの命令セットレベルシミュレーションが含まれる．従来では，このプロセッサ・シミュレーションのために，C言語やVHDLのビヘイビア記述などを用いてシミュレーション・モデルを作成していた．

そこで，本研究では，ASIP 開発システムであるPEAS-III (Practical Environment for ASIP Development type III) で入力されたプロセッサの仕様記述から，命令セットレベル・シミュレータを自動生成する手法を提案する．本手法を用いることにより，パイプライン段数，命令セット，演算器構成などのプロセッサ仕様が変更された場合でも，シミュレータの記述を人手により修正することなく，変更後の仕様に従うシミュレータを得られる．また，PEAS-III の生成するプロセッサのモデルに準拠したシミュレーション・モデルを採用することにより，本手法により生成されるシミュレータの実行結果と，PEAS-III の生成するプロセッサの動作結果との間に矛盾が生じないようにする．生成されるシミュレータは，プロセッサのパイプライン動作をクロックサイクル単位でシミュレートする．これによりパイプラインを流れる各命令の様子が正確に再現されるため，実行サイクル数の正確な見積りが可能となる．本手法では，シミュレータの記述言語としてSystemC [4] を用いた．SystemCは，並列動作するシステムのC++によるモデリングを可能にするため，プロセッサにおいて並列動作するパイプライン・ステージの記述に用いることができる．またSystemCは，記述されたモデルをサイクル単位でシミュレートするカーネルをライブラリとして提供しているので，本手法により生成されるシミュレータの制御部として用いることにした．

評価実験では，試作した処理系により生成されたシミュレータの性能評価を行った．PEAS-III システムは，入力されたプロセッサ仕様記述から，論理合成可能なプロセッサのVHDL記述を生成するので，VHDLシミュレータを用いてプロセッサ・シミュレーションを行うことができる．このシミュレーション方法と比較して，自動生成されたシミュレータは，約1/7の実行時間で応用プログラムの実行サイクル数の見積りが行えることを示した．

2 PEAS-III システムにおけるプロセッサモデル

PEAS-III システムは，ASIP を開発するためのワークベンチである．PEAS-III を用いることにより，設計者はプロセッサを抽象度の高いレベルで記述でき，設計の早期段階で，面積，動作周波数，消費電力などのプロセッサの性能を見積ることができる．

本章では，まず，PEAS-III で設計可能なプロセッサのアーキテクチャについて述べる．次に，設計者がPEAS-III に入力する設計仕様記述について述べ，その仕様記述からPEAS-III が生成するプロセッサのモデルを示す．

2.1 PEAS-III で設計可能なプロセッサ・アーキテクチャ

PEAS-III で設計可能なプロセッサ・アーキテクチャは次のようになっている．

- パイプライン・アーキテクチャ
- ハーバード・アーキテクチャ
- ハードウェア・インタロック機構
- 固定長命令
- 汎用レジスタ方式

なお，パイプライン段数は任意の数にすることができ，ハードウェア・インタロック機構を有しないアーキテクチャも設計可能である．

2.2 入力する設計仕様記述

設計者は，以下に示す情報を PEAS-III に与えることにより，設計対象とするプロセッサのアーキテクチャを指定する．

- アーキテクチャ・パラメータ
- 命令セット
- 使用するハードウェア・リソース
- 命令の動作定義

アーキテクチャ・パラメータとして与える情報は，パイプライン段数，バス幅，レジスタ数，ハードウェア・インタロックの有無などである．命令セットには，各命令のニーモニック，命令形式，および命令のオペコードを定義する．使用するハードウェア・リソースには，プログラム・カウンタやレジスタファイルなどの基本的なリソースの他に，乗算器やシフタなどを必要に応じて宣言する．なお，使用するリソースは，FHM (Flexible Hardware Model)[5] データベースに登録されているものの中から選択する．このデータベースは，登録されているリソースのモデル記述を格納しており，これらの記述はパラメタ化されているため，様々な種類のリソース・インスタンスの記述を得ることが可能である．

以上の情報を与えた後，定義した各命令に対して動作定義を行う．ここで定義する命令動作はクロックサイクル単位で記述し，各クロックサイクルでは，リソースに対する制御または演算の指示，データ転送，および，条件判定の操作 (以後，これらの操作をマイクロ動作と呼ぶ) を記述する．図 1 は条件付き分岐命令の動作定義例である．この例において，$ で始まる文字列は変数名を，英大文字のみの文字列はリソース・インスタンス名を，英小文字のみの文字列は命令語のフィールド名を，それぞれ表している．

2.3 生成されるプロセッサのモデル

PEAS-III のサブシステムの一つである HDL 記述生成系は，設計者の入力した仕様記述から，論理合成可能なプロセッサの HDL 記述を自動生成する．この自動生成手法は文献 [6] に述べられているが，この手法ではプロセッサは図 2 のようにモデル化される．すなわち，図 3 に示すパイプライン・ステージのモデルをパイプライン段数分連結し，プロセッサをモデル化している．なお，ハードウェア・インタロックを行うためには，すべてのステージを制御するための制御部が必要となるため，図 3 のステージ・モデルを単純に連結するだけではプロセッサ全体を構成することはできない．

```
clk(1){
  "$pc := PC;
   IR := IMEM[PC];
   PC.inc();"
},
clk(2){
  "DECODE(IR);
   $rs := GPR.read0(rs);
   $rt := GPR.read1(rd);
   $imm := EXT.sign(immediate);"
},
clk(3){
  "$offset := $imm(29 downto 0) & \"00\";
   $target := ADDER.add( $pc, $offset );
   $flag := ALU.cmp( $rs, $rt );"
},
clk(4){
  "if ( $flag(2) = '1' ) then
     PC.write( $target );
   end if;"
},
clk(5){ "" }
```

図 1: 条件付き分岐命令の動作定義例

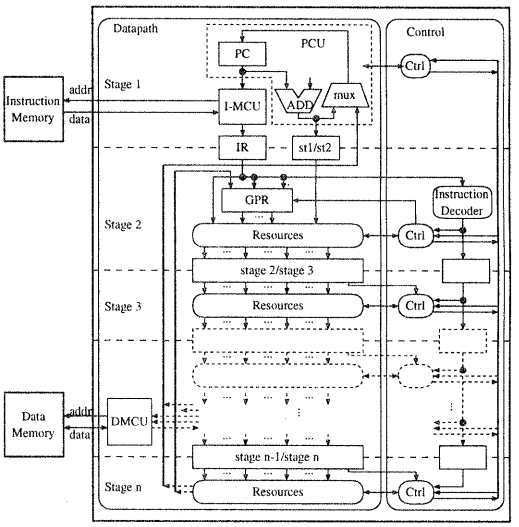

図 2: PEAS-III により生成されるプロセッサの構造

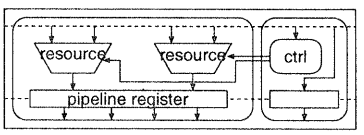

図 3: PEAS-III により生成されるパイプライン・ステージのモデル

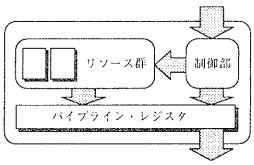

図 4: パイプライン・ステージのシミュレーション・モデル

3 シミュレータ生成

本章では，PEAS-III に入力されたプロセッサ仕様記述から命令セットレベル・シミュレータを生成する方法について述べる．

3.1 シミュレーション・モデル

実行サイクル数を正確に見積るために，図 2 と同様の構造をシミュレーション・モデルとして用いることにするが，シミュレータ生成系の試作を容易にするために，PEAS-III で設計可能なプロセッサのアーキテクチャに対し，以下の制約を置く．

- ハードウェア・インタロック機構を有さない
- パイプラインの各ステージの処理は常に 1 クロックサイクルの時間で完了する

この制約により，シミュレーション・モデルは，図 4 で示されるパイプライン・ステージのモデルを単純に連結したものとして表せる．なお，図 4 において，角の丸い四角形は処理を，影付きの四角形はデータ構造を，矢印はデータの流れを，それぞれ表している．

3.2 シミュレータの構成

シミュレータの記述言語はソフトウェア・プログラミング言語とし，生成されたシミュレータの記述をコンパイルして得られる実行可能ファイルを実行すればシミュレーションが行えるようにする．このようにすれば，専用シミュレータにモデル記述を読み込ませてシミュレーションを行う場合と比較し，速度の向上が期待できる．

しかし，並列に動作する各パイプライン・ステージをソフトウェア・プログラミング言語を用いてシミュレートするためには，シミュレータ内部にシミュレーション制御部が必要となる．シミュレーション制御部は，パイプライン・ステージ毎の動作を適当な順序で実行する．ここで，ステージの実行順に依存した動作を防ぐために，パイプライン・ステージ間で共有されるデータ，すなわちパイプライン・レジスタの内容はシミュレーション制御部に管理させ，共有データへの値の読み書きを制御する．このような制御を実現するシミュレータの構成を図 5 に示す．この図ではプロセスという言葉を用いているが，ここで用いるプロセスの意味は次の通りである．

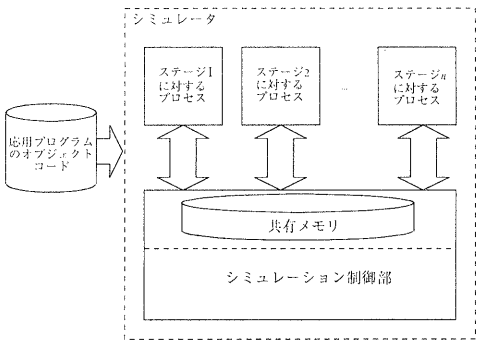

図 5: シミュレータの構成

- ある一連の処理を逐次的に実行する手続き
- 処理が最後に達した時点で，実行は停止する．次に起動されると最初から処理を開始する
- すべてのプロセスは並列に動作する

プロセスの内部は逐次的に実行されるため，個々のパイプライン・ステージのモデルはプロセスとして記述することができる．また，すべてのプロセスは並列に動作するため，プロセスとして記述されたすべてのパイプライン・ステージは並列に動作することになる．そこで，各パイプライン・ステージはプロセスとして記述することにする．

3.3 シミュレータの記述言語

本研究では，シミュレータを記述するために用いる言語として SystemC を選択した．SystemC は，並列性，同期，および時間を C++ プログラムで扱えるようにするためのクラスライブラリである．SystemC は

```
void ProcessStage;()
{
   「一時変数の宣言」
   「共有メモリからの値の読出」
   「マイクロ動作の実行」
   「共有メモリへの値の格納」
}
```

図 6: パイプラインステージ・モデルのプロセス記述

また，シミュレーションの制御を行うシミュレーション・カーネルもライブラリとして提供している．

そこで，前節で述べたシミュレーション制御部にはSystemC のシミュレーション・カーネルを利用することにし，各パイプライン・ステージのプロセス記述を自動生成することにする．

3.4 パイプライン・ステージのモデルのプロセス記述

3.1 節で示したパイプライン・ステージのモデルをプロセスの記述で表すことを考える．各パイプライン・ステージで行われる処理は次のようになる．アクティブなクロック・エッジが現れる毎に，前段のステージのパイプライン・レジスタの内容を読み出し，その内容に基づいて制御部はマイクロ動作を実行し，演算結果や制御値をこのステージのパイプライン・レジスタに設定する．ここで設定される値は，次のアクティブ・エッジが現れる時点でそのパイプライン・レジスタにロードされる．

この処理をプロセスとして記述すると図 6 のようになる．なお，本節以降，プロセスの記述例を示す際に C++ 言語の構文を用いて図示する．

図中，「一時変数の宣言」の部分には，パイプライン・レジスタから読み出した値を一時的に格納する局所変数を宣言するが，このステージで行った演算等の結果を一時的に格納する変数もここで宣言する．「共有メモリからの値の読出」の部分には，前段のステージが出力するパイプライン・レジスタの値を読み出し，一時変数に格納するコードが入る．パイプライン・レジスタの値は，シミュレーション・カーネルによって管理されている共有メモリに格納されている．「マイクロ動作の実行」の部分には，このステージで行うマイクロ動作を実行するコードが入る．この部分のコードは各命令の動作記述に基づいて生成される．「共有メモリへの値の格納」の部分には，このステージのリソースの出力や前段のステージから渡されたパイプライン・レジスタの値をそのまま次のステージに引き渡すコードが入る．ここで引き渡すべき値はすべて一時変数に格納されているので，実際には，一時変数の値を共有メモリに格納するコードが入ることになる．

以上に述べたようにして，パイプライン・ステージのモデルのプロセス記述の全体を構成する．

3.5 命令動作記述からのコード生成

図 6 で示したプロセス記述中の具体的なコードは，設計者の記述した命令動作記述から生成する．本節では，図 6 におけるプロセス記述の「マイクロ動作の実行」部分のコード生成について述べる．

3.5.1 実行する命令の識別

生成すべきプロセス記述は，ステージ毎に用意するため，各プロセス記述の中には，そのプロセスに対応するパイプライン・ステージで処理している命令を識別するコードを生成する必要がある．

そこで，各命令に対して一意な識別子をあらかじめ定義しておき，命令デコード・ステージに対応するプロセスは，解釈した命令語に対する識別子を，次のステージに対応するプロセスに共有メモリを通じて渡し，次のステージ以降のプロセスは，受け取った識別子の値に基づいて実行するコードを選択し，実行完了後にその識別子を次のステージのプロセスに渡すコードを生成する．例を示すと，図 7 に示すような命令とその動作が定義されている場合，第 3 ステージに対するプロセス記述の一部は図 8 のように生成される．図 8 中，instrid_2，instrid_3 は共有メモリ上の変数であり，それぞれ第 2，第 3 ステージのパイプライン・レジスタに格納されている命令識別子の値を表している．

3.5.2 命令語から命令識別子への変換

3.5.1 節では，命令デコード・ステージ以降の各ステージに対するプロセスにおける命令の識別方法を述べた．しかし，命令デコード・ステージでは，命令

```
INSTR_A:

  clk(2){
    "DECODE(IR);
     $rs := GPR.read0(rs);
     $rt := GPR.read1(rt);"
  },
  clk(3){
    "($result, $flag) := ALU0.add($rs, $rt);"
  },


INSTR_B:

  clk(2){
    "DECODE(IR);
     $rs := GPR.read0(rs);
     $rt := GPR.read1(rt);"
  },
  clk(3){
    "($result, $flag) := ALU0.sub($rs, $rt);"
  },
```

図 7: 例として用いる命令の動作記述

```
void ProcessStage3()
{
  int instrid = instrid_2;
      ...

  switch ( instrid ) {
    case i_INSTR_A:
      result = alu0.add( rs, rt ); break;
    case i_INSTR_B:
      result = alu0.sub( rs, rt ); break;
  }   ...

  instrid_3 = instrid;
}
```

図 8: 第 3 ステージのプロセス記述の生成例

フェッチ・ステージから渡される命令語から命令識別子を得る必要がある．そこで，命令デコード・ステージでは，受け取った命令語と，設計者の定義した各命令のオペコードとを順次比較し，一致したオペコードを持つ命令の識別子の値をパイプライン・レジスタに格納することにする．また，命令デコード・ステージでは，その他のステージと同様に，識別した命令に応じて，実行すべきマイクロ動作を選択する必要があるため，まず最初に命令語から命令識別子に変換し，次に，3.5.1 節で述べたことと同様の処理を行う．

以上より，命令デコード・ステージに対するプロセス記述は図9のようになる．図9において，imem_dout は，命令フェッチ・ステージから渡される命令語の値を持つ共有変数を，instrid_ID は，命令デコード・ステージのパイプライン・レジスタに格納する命令識別子の値を持つ共有変数である．

```
void ProcessStageID()
{
  int      instrid;
  unsigned instrcode = imem_dout;
           ...

  if ( imem_dout と命令 0 のオペコードの比較 )
    instrid = i_INSTR0;
  else if ( imem_dout と命令 1 のオペコードの比較 )
    instrid = i_INSTR1;
  else...

  switch ( instrid ) {
    case i_INSTR0:
         ...
    case i_INSTR1:
         ...
  }   ...

  instrid_ID = instrid;
}
```

図 9: 命令デコード・ステージのプロセス記述

3.5.3 マイクロ動作のコード生成

マイクロ動作として，データ転送，リソースに対する制御，および条件判定の 3 種類の操作を行うことができる．このうち，データ転送と条件判定に関しては，ソフトウェア・プログラミング言語を用いて容易に実現できるため，本稿では説明しない．本節では，リソースに対する制御を行うコードの生成のみについて述べる．

プロセッサの論理合成可能なハードウェア記述を生成する場合，各パイプライン・ステージの制御部は，実行する命令に応じて適切な制御信号をリソースに与える必要がある．また，リソースの各ポートとの接続関係を解決する必要がある．しかし，プロセッサのシミュレーションを目的とした場合，リソースの機能，すなわち振舞いが重要であるから，具体的な制御信号の値や接続関係を意識する必要はない．

そこで，命令動作記述におけるリソースに対する制御は，C++ におけるクラス・オブジェクトへの操作として捉えることにする．そうすると，リソース・インスタンス *Res* に対して *func* という機能の実行を指示するマイクロ動作記述:

Res.func(...)

は，プロセス記述においても全く同様の形で実現できる．

なお，この実現のためには，リソースの種類と同数のリソース・クラスをあらかじめ定義しておく必要がある．

3.6 SystemC によるプロセスの記述

3.4，3.5 節で図示したプロセスの記述は，説明のために C++ の構文を用いて表したが，シミュレーショ

ン制御部としてSystemCのシミュレーション・カーネルを用いることにしているので，実際に生成するプロセスの記述にもSystemCを用いることになる．

SystemCは，システムをモデリングする際に有用なデータ・タイプも提供しているので，このデータ・タイプを用いることにより，プロセス記述の生成が容易になる．例えば，命令デコード・ステージに対するプロセスでは，変数の値とオペコードのビット・パターンを比較する処理が行われるが，SystemCの提供するビット・ベクタ型を用いると，ビット・レンジの切り出しやビット列の比較，連結が簡単に記述できるため，Cのビット演算子を複雑に組み合せなければ表現できないような条件式を素直に表現できる．また，ビット・ベクタ型を用いると，任意のビット幅のデータの表現が可能になるため，どのようなバス幅を持つプロセッサに対してもシミュレータの生成が可能となる．SystemCの詳細については，文献[7]を参照されたい．

表1: 実験で用いるプロセッサの命令セット

命令	実行内容
add R1,R2,R3	R1 ← R2 + R3
sub R1,R2,R3	R1 ← R2 - R3
addi R1,R2,*imm*	R1 ← R2 + *imm*
and R1,R2,R3	R1 ← R2 bitwise-and R3
or R1,R2,R3	R1 ← R2 bitwise-or R3
andi R1,R2,*imm*	R1 ← R2 bitwise-and *imm*
ori R1,R2,*imm*	R1 ← R2 bitwise-or *imm*
lw R1,*disp*(R2)	R1 ← dmem[R2 + *disp*]
sw R1,*disp*(R2)	dmem[R2 + *disp*] ← R1
beq R1,R2,*ofs*	relative jump by *ofs* if R1 = R2
slt R1,R2,R3	R1 ← 1 when R2 < R3 0 otherwise
slti R1,R2,*imm*	R1 ← 1 when R2 < *imm* 0 otherwise

(R*n*: レジスタ; *imm*, *disp*, *ofs*: 即値)

表2: シミュレーション結果の比較

シミュレーション方法	方法A (VHDL)	方法B (C++)
実行サイクル数	527,018	527,018
実行時間 (秒)	1,593	211
処理能力 (命令/秒)	331	2,497

4 評価実験

本章では，本研究で試作したシミュレータ生成系によって生成される命令セットレベル・シミュレータを用いて応用プログラムの実行サイクル数の見積りが可能であることの確認，およびシミュレーション速度の評価を目的として実験を行った．

4.1 実験方法

評価サンプルとして用いるプロセッサはMIPS R3000[8]に準拠したプロセッサである．このプロセッサに定義した命令セットを表1に示す．このプロセッサに対するシミュレーションを2通りの方法で行う．1つは，このプロセッサの仕様記述からPEAS-IIIシステムを用いて生成したVHDL記述を，VHDLシミュレータを用いてシミュレーションを行う方法(方法Aとする)であり，もう一つは，それと同一のプロセッサ仕様記述から，本研究で試作した処理系によって生成されるシミュレータを用いる方法(方法Bとする)である．この2通りの方法で，同一の応用プログラムの実行をそれぞれシミュレートし，シミュレーション結果を比較することにより，本研究で作成した処理系によって生成されるシミュレータがプロセッサの仕様通りの動作をシミュレートし，実行サイクル数の見積りが行えることを確認する．

シミュレータ上で実行するプログラムは，0から199までの200個の整数データを昇順にソートするものである．シミュレーション開始前には，データメモリの先頭番地から200語分の領域に0から199までの整数値をランダムに格納しておく．そして，シミュレーションを開始しプログラムの実行が完了すれば，データメモリの内容は，先頭番地から上位番地に向かって，0, 1, 2, ... , 199という配置になることが期待できる．したがって，シミュレーション終了時にデータメモリの内容をテストすることにより，このプログラムが正しく実行されたかどうかを判定することができる．

4.2 実験結果

前節で述べた実験方法を用いて実験を行った．この実験の結果を表2に示す．

なお，シミュレーションを実行したハードウェア環境，使用したVHDLシミュレータ，および，試作した処理系の生成したシミュレータ記述をコンパイルする際に用いたC++コンパイラを，それぞれ以下に示す．

ハードウェア環境：SUN WS, CPU: Ultra Sparc II 296MHz, Memory: 1.1GB

VHDL シミュレータ：Synopsys VHDL System Simulator (VSS)

C++コンパイラ：GNU C Compiler (GCC) version 2.95.1

4.3 考察

表 2 より，本研究で試作した処理系によって生成されるシミュレータを用いることにより，応用プログラムの実行サイクルの正確な見積りが行えることがわかった．また，シミュレーションの実行時間は，VHDL シミュレータを用いる方法に比べ，約 1/7.5 となった．

本手法では 3.1 節で述べた制約を満たすプロセッサのみのシミュレーションが行えるが，それらの制約を無くした場合でも，拡張は可能であると思われる．ハードウェア・インタロックに関しては，PEAS-III で用いているインタロックの論理を導出し，シミュレータのパイプライン・モデルを拡張すればよい．複数のクロックサイクルを要するステージがある場合も同様である．

5 おわりに

本稿では，プロセッサの仕様記述から命令セットレベル・シミュレータを生成する手法について述べた．そして，本手法を用いてプロセッサの命令セットレベル・シミュレータを生成できること，および，生成されたシミュレータを用いて応用プログラムの実行サイクルの正確な見積りが行えることを評価実験により確認した．

今後の課題としては，応用プログラムの開発ツールである，コンパイラ，アセンブラを自動的に生成する処理系を実現することが考えられる．

謝辞

本研究を進めるにあたり，貴重なコメントを頂いた大阪大学 VLSI システム設計研究室の諸氏に感謝致します．なお，本研究の一部は (株) 半導体理工学研究センターとの共同研究による．

参考文献

[1] Sato,J., Alomary,A.Y., Honma,Y., Nakata,T., Shiomi,A., Hikichi,N. and Imai,M., "PEAS-I: A Hardware/Software Codesign System for ASIP Development", IEICE Trans. Fundamentals, Vol.E77-A, No.3, pp.483-491, 1994.

[2] 濱辺 雅哉, 能勢 敦, 戸川 望, 柳澤 政生, 大附 辰夫, "パイプラインプロセッサのハードウェア記述自動生成手法", 信学技報 VLD 97-117, pp.33-40, 1997.

[3] Takayuki Morimoto, Kazushi Saito, Hiroshi Nakamura, Taisuke Boku and Kisaburo Nakazawa, "Advanced Processor Design Using Hardware Description Language AIDL", Proc. of ASP-DAC'97, pp.387-390, 1997.

[4] Stan Liao, Steve Tjiang and Rajesh Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment", Proc. of DAC'97, pp.70-75, 1997.

[5] T.Morifuji, Y.Takeuchi, J.Sato, and M.Imai, "Flexible hardware model: Implementation and effectiveness", Proc. of SASIMI'97, pp.83 - 289, Dec 1997.

[6] 伊藤 真紀子, 檜垣 茂明, 塩見 彰睦, 佐藤 淳, 武内 良典, 今井 正治, "パイプライン・ハザードを考慮した合成可能なプロセッサの HDL 記述生成手法の提案", DA シンポジウム'99 論文集, pp.201-206, 1999.

[7] Synopsys, Inc., "SystemC Reference Manual" Release 0.9, Synopsys, Inc., 1999.

[8] Gerry Kane, "mips RISC アーキテクチャ — R2000 / R3000 —", 共立出版, 1992.