計算機アーキテクチャ 110－1
設 計 自 動 化 73－1
（1995. 1. 19）

情報抽出技術を用いたアーキテクチャ評価用シミュレーション・モデルの生成

赤星 博輝　安浦 寛人

九州大学 大学院 総合理工学研究科 情報システム学専攻
〒816 春日市春日公園6—1
Email: {akaboshi, yasuura }@is.kyushu-u.ac.jp

あらまし
RT レベルでプロセッサのシミュレーションを行う場合，シミュレーション・スピードが遅いという問題があった．そのために，高速なシミュレーション・モデルが必要とされていた．本稿は，RT レベルのプロセッサ記述からアーキテクチャ評価用シミュレーション・モデルの自動生成の手法について述べ，その実験結果を報告する．自動生成されたシミュレーション・モデルは，プログラムを実際の計算機上で実行した場合に比べ，約 1/100 のスピードで実行可能であった．

和文キーワード： シミュレーション・モデルの生成，アーキテクチャ設計，情報抽出，CAD

Simulation Model Generation for Architecture Evaluation using an Information Extraction Technique

Hiroki AKABOSHI and Hiroto YASUURA

Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
6-1 Kasuga-koen, Kasuga-shi, Fukuoka 816 Japan

E-mail:{akaboshi,yasuura}@is.kyushu-u.ac.jp

Abstract One of major problems in evaluation of computer architecture is that simulation speed is not enough to simulate a microprocessor in RT desgin. To simulate a microprocessor with practical programs, a fast simulation model is required. In this paper, an algorithm is proposed for fast simulation model generation and experimental results are shown. Simulation speed of a generated simulation model is one-hundred times slower than a real machine.

英文 Key Words: Simulation Model Generation, Architecture Design, Inforomation Extraction, CAD

1 はじめに

アーキテクチャ設計では，性能評価にかなりの時間が費やされる．実際に性能評価を行うために，実際のプロセッサ上で動作が予定されるプログラムや一般的なベンチマーク・プログラムを用いてシミュレーションし，性能評価を行うといった方法が用いられる．今までの設計支援ツールは，RT レベルからの論理/レイアウト合成といった支援および，シミュレーションといった支援を行っていた．しかし，設計支援ツールに付随するシミュレータは，もともと，論理的な動作や遅延時間の評価を用いるためのものであり，プログラムを用いたアーキテクチャの評価を行うためのものではない．細かい部分まで精度の高いシミュレーションを行っており，シミュレーション時間がかかり過ぎる．そのため，実用プログラムを用いて性能評価を行うことは難しい [5]．そのために高速なシミュレーションを行うために，ハードウェア記述言語による設計とは別に，新たにシミュレーション・モデルの作成などが行われている [6]．

我々は，ハードウェア記述言語によるプロセッサ記述からの情報抽出という技術の研究をすすめており，この技術を，命令レベルのシミュレーション・モデルの自動生成に用いる．この情報抽出された情報からシミュレーション・モデルを自動生成することで，設計者はプロセッサ記述を行うだけで複数のレベルのシミュレーション・モデルを利用することができる．既存のシミュレータではスピードの点から，大規模なシミュレーションすることができなかった．ここで自動生成するシミュレーション・モデルの目的は，大規模なプログラムをシミュレーション可能にすることである．

本稿では，2章で 命令レベル・シミュレーションの定義および情報抽出の説明を行い，3章でシミュレーション・モデルの生成法について述べ，4章で実験結果を示す．

2 アーキテクチャ評価用シミュレーション・モデル

2.1 シミュレーションのレベル

実際にプロセッサの設計を行う場合，いくつかのレベルのシミュレーションが必要である．われわれは，シミュレーションのレベルを表1の レイアウト，ゲート，RT，パイプライン，命令 レベルの5つに分類した．レイアウト・レベルでは，トランジスタを中心とした設計を行い，扱う信号はアナログ信号を扱い，基本的には連続時間でシミュレーションを行う．レイアウト・レベルでは，レイアウトの配線を考慮したシミュレーションを行う．

ゲート・レベルでは，ゲートが設計の中心となり，扱う信号も 0,1 といったデジタルとなり，ゲートの遅延時間を考慮したシミュレーションを行う．

RT レベルでは，レジスタに対する動作を論理式で与え，扱う信号はデジタル，基本的な時間はレジスタに対するクロックとなる．

ターゲット・アーキテクチャをプロセッサとした場合に，次の 2 つのレベルが必要となる．パイプラインでは，時間的にはクロックを最小単位とするが，まとまりのある信号については 1 ビットの信号をまとめたワードとして扱い，ワード単位の処理を行う機能ユニットが設計時の基本的な素子となる．

命令レベル (詳細は，2.3節で述べる) では，基本素子は MPU 本体となり，信号はワードが基本となり，基本的な時間は 1 命令を実行する時間 (命令サイクル) になる．

2.2 シミュレーション・モデルの生成

これまでの設計支援ツールの進歩により，RT レベルの設計から論理合成を行うことによってゲート・レベルのシミュレーション，さらにレイアウト合成を行うことでレイアウト・レベルのシミュレーションを行うことが可能になった．

これは，設計者からの立場からすると，観測する対象に応じて使用するシミュレーション・モデルを選択できることになる．シミュレーションの精度とシミュレーションに要する時間はトレードオフの関係にあり，精度の高いシミュレーションを行うとシミュレーションに要する時間は増大し，シミュレーションの時間を短くするためには精度を下げたシミュレーションを行う必要がある．精度の高い レイアウトの配線遅延を考慮した性能評価を行うためにはレイアウト・レベル，論理的な動作のみを評価したい場合には RT レベルと切替えることで，設計者は用途に応じたシミュレーションを行うことが可能となる．これは，今までの論理/レイアウト合

表 1: シミュレーションのレベル

レベル	レイアウト	ゲート	RT	パイプライン	命令
基本素子	トランジスタ	ゲート	論理式	機能ユニット	MPU
信号値	アナログ	デジタル	デジタル	ワード	ワード
単位時間	連続時間	離散時間	クロック	クロック	命令サイクル

成ツールがシミュレーションのレベルの変換を行っていると考えられる (図 1). 論理/レイアウト合成ツールでは精度の高いシミュレーションを行っているが, 我々は, シミュレーション時間の短縮を図るために精度を落した命令レベル・シミュレーションを行う. そのために, HDL からの動作の抽出を行う情報抽出という技術 [1] を用いて, 命令レベルのシミュレーション・モデルの自動生成を行う.

情報抽出によってプロセッサの動作を抽出し, その抽出した動作から命令レベル・シミュレーション・モデルを自動生成することによって, さらに設計者が自由に使えるシミュレーション・モデルのレベルが増えるということになる.

この命令レベル・シミュレーションによって, 精度が落ちるものの高速なシミュレーションが可能となり さまざまなプログラムを用いて性能評価を行うことが可能になると考えられる.

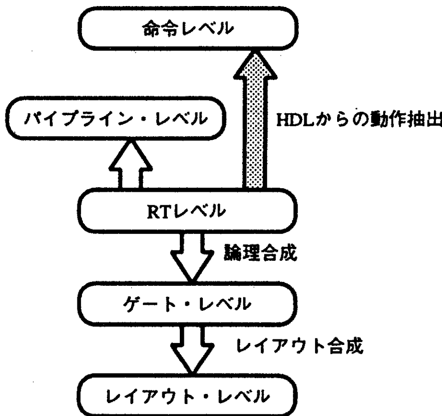


図 1: シミュレーション・レベルの変換

2.3 命令レベル・シミュレーション

命令レベル・シミュレーションを定義するために, まず, 用語の定義を行う.

プログラマブル・レジスタ プログラマが直接設定・参照可能なレジスタおよびメモリのこと (例えば, 汎用レジスタ, メモリなど)

ノンプログラマブル・レジスタ プログラマが直接設定・参照不可能なレジスタやメモリのこと (例えば, フラグやパイプライン・レジスタなど)

全レジスタ プログラマブル・レジスタ および ノンプログラマブル・レジスタ (すべての記憶資源は, 全レジスタに属するものとする)

プログラム中の i 番目の命令の動作 f_i を次のように定義する.

$$R_{t+1} = f_i(R_t) \qquad (1)$$

ここで, R_t は, 命令 f_i が実行される前の全レジスタの内容, R_{t+1} は命令 f_i が実行された後の全レジスタの内容を示す.

プログラムとは, 命令の集合のなかから (重複を許して) 命令を並べた, 任意の有限列である.

命令レベルの実行モデルとは, 式 (2) から (6) の動作を有限回 繰り返すモデルとする.

$$PC = get_pc(R_t); \qquad (2)$$

$$INST = instruction_fetch(PC, R_t); \qquad (3)$$

$$PC = PC++; \qquad (4)$$

$$f_i = decode(INST); \qquad (5)$$

$$R_{t+1} = f_i(R_t); \qquad (6)$$

式 (2) から (6) 迄の実行は分割できないものとする.

式 (2) では, 全レジスタからプログラム・カウンタの値を取り出す. 式 (3) で, PC が示すアドレスのメモリの内容を読みだし, $INST$ に代入する. 式 (4) で, PC をインクリメントする. 式 (5) で命令をデコードし, 式 (6) 命令 f_i の実行を行う.

命令レベルのシミュレーションは, “命令レベルの実行モデル上でプログラムを実行した結果とプ

ログラマブル・レジスタの内容が同値であること”と定義する．

2.4 情報抽出

一般的な設計では，RT レベルからゲートレベルというように上位レベルから下位レベルへ変換を行うものである．より高位のレベルのシミュレーション・モデルを作成するためには，下位レベルから上位レベルへ記述の変換を行う必要がある．

下位レベルから上位レベルへの変換としては，レイアウト・レベルからゲート・レベルへの変換を行う部分が実用的な技術として用いられている [2, 3]．また，ゲート・レベルから RT レベルへの変換などの研究も行われている [4]．プロセッサの設計では，RT レベルの設計が行われることが多いために，RT レベルのプロセッサの設計から命令の動作を取り出す必要がある．現在，その RT レベルの設計に付加情報を与えることで，命令の動作を取り出す研究が行われており，[1] の技術を利用する．基本的なアイディアとしては，プロセッサとして重要なレジスタ (命令レジスタ，プログラム・カウンタ，汎用レジスタなど) を付加情報として与え，プロセッサをその付加情報を用いて各命令の動作を記述している部分に分け，全ての命令についてプログラマブル・レジスタの内容をどのように変更するかを調べるというものである．これにより RT レベルの記述から，命令の動作を自動で取り出すことができる．

3 命令レベル・シミュレーション・モデルの生成

本手法では，HDL によるプロセッサ記述から抽出した情報を用いて高速シミュレーション・モデルを生成する．図 2に概要を示す．ハードウェア記述言語 (HDL) の記述から情報抽出をすることで，命令の動作を取り出し，その他に付加情報を与えることで，シミュレーション・モデル・ジェネレータの入力とする．その入力からシミュレーション・モデルを作成するが，シミュレーション・モデル・ジェネレータの出力は C++のコードであるために，そのコードをコンパイルすることでシミュレーション・モデルを生成する．

HDL による設計からプロセッサの動作の抽出に

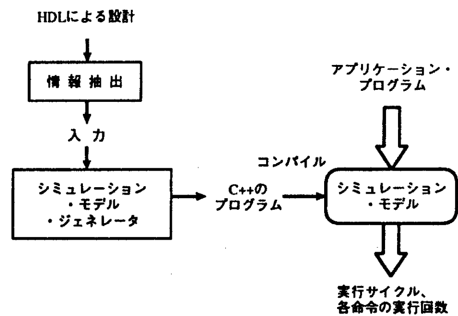


図 2: シミュレーション・モデルの生成

は，[1] で述べた手法を用いる．命令の動作としては，全ての基本レジスタの対応ではなくて，実際の変化に関係あるものだけが明示的に示される．

3.1 入力

入力としては，全ての命令の op_code と命令の動作，および，汎用および浮動小数点レジスタの数，ROM，RAM のサイズ，さらに，命令レジスタのビットの割り当てに関する情報を与える．

図 3に入力の例を示す．図 3(a) が命令の op_code と命令の動作を示している．ここで，*GPRd* はデスティネーション・レジスタを指し，*GPR*1 はソースレジスタ 1，*GPR*2 はソースレジスタ 2，*Imm* は即値データ，*.mem* はアクセスするメモリを示している．

op_code が *0X00000000* の時は，NOP を表している．op_code が *0X10000000* の時は，ソースレジスタ 1 とソースレジスタ 2 の加算を行い，デスティネーション・レジスタに書き込むことを示す．op_code が *0X40000000* の時は，ソースレジスタ 1 と即値の値を足したアドレスのメモリの内容を，デスティネーション・レジスタに書き込むことを示す．

図 3(b) では，図 3(c) のその他の情報を示している．

- 汎用および浮動小数点レジスタの数
- ROM および RAM のサイズを与える．
- 命令レジスタの op_code，デスティネーション・レジスタ，ソースレジスタ，即値を指定するフィールドの情報として，それらのフィールドを 1 でマスクする値を与える．
- 各フィールドの最下位ビットの指定

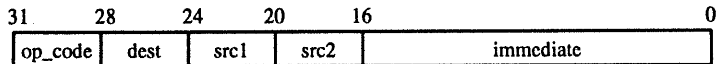
```
0X00000000 :  ;
0X10000000 : GPRd := ADD( GPR1, GPR2 );
0X20000000 : GPRd := SUB( GPR1, GPR2 );
0X30000000 : GPRd := MULT ( GPR1, GPR2 );
0X40000000 : GPRd := LOAD( .mem, ADD(GPR1, IMD));
0X50000000 : STORE( .mem , ADD(GPR1,IMD), gpr2);
0X60000000 : GPRd :=  IMD;
0X70000000 : PC   := ADD(GPR1, IMD);
```

(a) op_codeと命令の動作

```
int NUM_OF_GPR = 16;
int NUM_OF_FPR = 0;
int ROM_SIZE   = 65536; /* Words */
int RAM_SIZE   = 65536; /* Words */
int op_code    = 0xF0000000; /* OP_CODE */
int op_dest    = 0x0F000000; /* Destination */
int op_src1    = 0x00F00000; /* Source 1 */
int op_src2    = 0x000F0000; /* Source 2 */
int op_imm     = 0x0000FFFF; /* Immediate */

int code_shift = 28;
int dest_shift = 24;
int src1_shift = 20;
int src2_shift = 16;
int imm_shift  = 0;
```

(b) その他の情報



(c) この例での命令のフィールド

図 3: 入力例

3.2 コード生成アルゴリズム

シミュレーション・モデルは，*YACC*を用いて生成する．基本的なアルゴリズムを *YACC*プログラムの一部を使って説明する．

図 4(a) が，命令の動作と生成されるコードの一部を示している．基本的にループの 1 イタレーションで 1 命令のシミュレーションを行う．そして，命令を関数 `get_instruction()` で ir にいれ，その ir の内容を見て実行する命令を case 文によって決定する．

命令の動作の Op-Code の部分がシミュレーション・モデルの case 文の分岐に用い，プログラム・カウンタのインクリメントおよびcase 文からの break のコードを生成する．そして，抽出された命令の動作のシミュレーションを行うコードは，プログラム・カウンタのインクリメントと break 文の間に挿入する．コードの生成法を，図 4(b) の *0X10000000*の場合を用いて用いて説明する．まず，関数の説明をしておくと，regi.write(A,B) はレジスタ番号 A に B の値を代入する関数，regi.read(A) はレジスタ番号 A の値を返す関数，get_dest(ir) は命令レジスタの内容からからデスティネーションレジスタの番号を返す関数，get_src1(ir) は命令レジスタの内容からソースレジスタ 1 の番号を返す関数，get_src1(ir) は命令レジスタの内容からソースレジスタ 2 の番号を返す関数である．

*YACC*では，まず，GPR1，GPR2 が (3) で処理される．`$1` にソースレジスタ 1 という情報が入っており，ソースレジスタ 1 の値を読み出して，中間変数に値を代入するコードを出力する．そして，返り値として中間変数の番号 (num) を返す．

次に (2) の処理を行う，`$3`，`$5` に中間変数の番号が与えられるので，指定された中間変数の加算を行い，さらにその結果を新しい中間変数に代入するコードを出力する．

最後に (1) の処理を行い，$3 に加算の結果を収納した番号が与えられるので，そのレジスタの値をデスティネーションレジスタに書き込むコードを生成し，さらにこの命令のシミュレーションが終るという break 文を出力する．このように命令の動作に応じてシミュレーション・モデルを生成する．

4 実験結果

今回作成した高速シミュレーション・モデル・ジェネレータは，取り出された情報から C++のコードを生成し，そのコードをコンパイルすることでシミュレーション・モデルの生成を行った．今回生成したプロトタイプでは，ターゲット・アーキテクチャに次の制約を加えている．

- データ幅 32 ビット
- ハーバード・アーキテクチャ
- 汎用レジスタモデル
- 割込みなどは考えない，ただ単純な命令流のみを実行するモデルとする

そして，入力に用いるハードウェア記述言語としては *UDL/I* を利用している．

16 本の汎用レジスタ，13 個の命令を持つプロセッサを持つプロセッサのシミュレーションを行った．シミュレーションを行う際には，キャッシュ・ヒット率は 100%としてシミュレーションを行った．

まず，現在までに情報抽出を行った簡単な RISC プロセッサに対する高速シミュレーション・モデルを生成し，市販されているの CAD ツールのシミュレータとシミュレーション・スピードの比較を行った．また，命令セットやレジスタ数は異なるが，実際の計算機上でプログラムを実行した場合と比べてどの程度スピードが遅くなるかを比較するため，Sparc Station での実行時間も計測した．次の 4 通りについて評価を行った．

SPARC: Sparc で直接ベンチマーク・プログラムを実行した場合

本手法: 本手法で生成したシミュレータを用いてシミュレーションを行った場合

V-SYSTEM: VHDL を入力言語とする Model Technology 社のシミュレータ *V-System/Workstation* を用いてシミュレーションを行った場合

QuickSim: VHDL を入力言語とする Mentor Graphics 社のシミュレータ *QuickSim II* を用いてシミュレーションを行った場合

全ての比較は，同じ Sparc Station 10 上で行い，シミュレーション時間は Unix の time コマンドで計測を行った．**SPARC** の場合，実行時間が短いため time コマンドでは有効な時間が測定できない．**SPARC** については，**SPARC** と本手法について最外ループのループ回数を増やして実行し，本手法との相対時間を示している．

実際に，ハノイの塔 と エラトステネスのふるいの 2 つのプログラムを評価用プログラムとして使用した．

また，これらのシミュレーション・モデルの生成しするのに要した時間は，6.5 秒程度であった．

本手法は，実機上でプログラムを実行した場合に比べて，数 100 倍という時間でシミュレーションを行うことが可能であった．それに対し現在良く使われている市販のシミュレータは，10^5 から 10^7 倍のシミュレーション時間を要する．これは，本手法と比べても 10^2 から 10^4 倍の時間を要する．この差は，命令レベルのシミュレーションでは，内部の細かな動作をシミュレートする必要がないために発生したと考えられる．特に，プログラマに対して見えないレジスタのシミュレーションを行わないことで，大幅にシミュレーション時間を削減することができた．また，本手法は RT レベルの設計から命令レベルのシミュレーション・モデルを作成したために，より細かな部分を観測するためには，設計を行った RT レベルやレイアウト・レベルのシミュレーションを行うことで観測することができる．

表 3 では，命令の実行頻度を示している．現段階では，各命令の総実行命令数などを計測することができる．このような評価を短時間で実行可能な命令レベルのシミュレーションは，有効であると考えれれる．

5 おわりに

情報抽出という技術を用いることで，RT レベルのプロセッサの設計から，命令レベルのシミュレーション・モデルの自動生成を行う手法について述べた．命令レベルのシミュレーション・モデルは，シミュレーションの精度を落すことで，高速なシミュ

表 2: シミュレーション時間

	SPARC	本手法	V-SYSTEM	QuickSim
Sieve (秒)	0.00394	0.894	374.468	4589.642
Hanoi (秒)	0.00221	0.238	83.548	926.302

表 3: 命令実行回数 (回)

	ADD	SUB	LOAD	LOADI	STORE	JUMP	BEQZ	BNEZ	SGT	合計
Sieve	154,547	0	8,191	208,798	37,864	46,055	8,191	54,248	54,248	572,142
Hanoi	20,473	1	28,661	16,385	24,570	10,239	0	4,095	0	104,424

レーションを行うことを行うことが可能となった．実際の計算機上で実行した場合に比べ，数100倍程度の実行時間の遅れでシミュレーションが可能であった．今までは，実際に用いられるプログラムを用いて性能評価を行うことは困難であったが，シミュレーションの目的によっては本手法が有効であることを示した．

RTレベルの設計を基本としているために，今までの，CADツールはそのまま利用可能である．そのために，より細かなレベルの観測を行うためには，今までのCADツールなどが利用できる．このような環境の中で，設計者は，大雑把なシミュレーションを行う場合には命令レベルのシミュレーション，詳細なシミュレーションを行うためにはレイアウト・レベルのシミュレーションと用途に応じて自由に選ぶことが可能となる．今回の実験では，小規模なプログラムを利用したが，今後より大規模なプログラムに関してもシミュレーションを実際に行って評価をしていく予定である．

今後の課題としては，割込み，周辺回路などのサポートを考えている．また，パイプライン・レベルのシミュレーション・モデルに関しても考察を行っている．

謝辞

QuickSim IIなどの提供頂くメンター・グラフィクス・ジャパンの皆様，V-SYSTEMを利用させて頂いたソリトン システムズの皆様に感謝致します．日頃御討論頂く，安浦研究室の諸氏に感謝致します．

参考文献

[1] Hiroki Akaboshi and Hiroto Yasuura. “Behavior Extraction of MPU from HDL Description”. In *Second Asian Pacific Conference on Hardware Description Languages*, pp. 67–74, Oct. 1994.

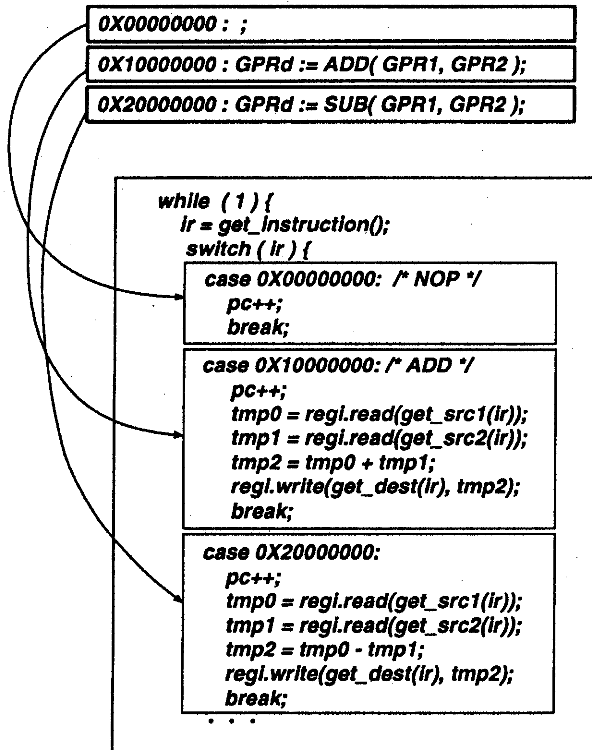
[2] David T. Blaauw, Daniel G. Saab, Robert B. Mueller-Thuns, Jacob A. Abraham, and Joseph T. Rahmeh. “Automatic Generation of Behavioral Models from Switch-Level Descriptions”. In *26th ACM/IEEE Design Automation Conference*, pp. 179–184, Jun. 1989.

[3] Randal E. Bryant. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. In *Proc. ICCAD 91*, pp. 350—353, Nov. 1991.

[4] Masahiko Ohmura, Hiroto Yasuura, and Keikichi Tamaru. “Extraction of Functional Information from Combinational Circuits”. In *Proc. ICCAD-90*, pp. 176–179, Nov. 1990.

[5] Minoru Shoji, Fumiyasu Hirose, and Koichiro Takayama. “VHDL Compiler of Behavioral Descriptions for Ultrahigh-Speed Simulation”. In *Second Asian Pacific Conference on Hardware Description Languages*, pp. 85–88, Oct. 1994.

[6] 那須隆, 岩田俊一, 清水徹, 斎藤和則. “16 ビットマイクロコントローラ (M16) の高速シミュレータの開発”. *Technical Report of IEICE VLD93-88*, pp. 25–32, Dec. 1993.



(a) 生成のながれ

```
statement : GPR  ASSIGN expression ';' {  <- - - - - - - - - - - - - - -  (1)
        fprintf(fp,"regi.write(get_dest(ir), tmp%d);\n", $3);
        fprintf(fp,"break;\n");
    }
    ;

expression : ADD '(' expression ',' expression ')' {  <- - - - - - - - - -  (2)
          fprintf(fp, "tmp%d = tmp%d + tmp%d; \n",num, $3, $5);
          $$ = num++;
      }
    | GPR {  <- - - - - - - - - - - - - - - - - - - - - - - - - - - -  (3)
          fprintf(fp, "tmp%d = regi.read(get_src%d(ir));\n", num, $1);
          $$ = num++;
      }
    ;
```

(b) シミュレーション・モデル・ジェネレータの一部

図 4: コード生成プログラム