ディジタル信号処理向けプロセッサのための シミュレータ生成手法

笠原亨介[†] 戸川望[††,†††]
柳澤政生[†] 大附辰夫[†]

本稿では，DSP を対象としたハードウェア/ソフトウェア協調合成システムにおいて，対象プロセッサの構成を評価し，アセンブリコードの検証を行うシミュレータの生成手法を提案する．本システムは異なるアプリケーションを対象とすると，異なるプロセッサ構成を出力するため，シミュレータはプロセッサ構成に対応するようにリターゲッタブルに生成されなければならない．提案手法はプロセッサの機能ユニット，パイプライン構成，命令セット，フォワーディングユニットの記述から，プロセッサに含まれる機能ユニットの種類と個数，パイプライン段数や各ステージおける動作，命令のパイプライン内の動作，フォワーディングユニットの動作を解析し，これらを C++記述で表し，この記述をもとにパイプラインシミュレータおよび命令セットシミュレータを生成する．いくつかのプロセッサ構成について生成したシミュレータの計算機実験結果を示し，提案手法の有用性を示す．

Simulator Generation for Digital Signal Processors

Kyosuke Kasahara,[†] Nozomu Togawa,[††,†††] Masao Yanagisawa[†] and Tatsuo Ohtsuki[†]

This paper proposes a methodology to generate instruction-set and pipeline simulators in hardware/software cosynthesis system for digital signal processors. Since the system synthesizes an application-specific processor core based on a given application program and its data, simulators must be configured so that they can be applied to the synthesized processor core, i.e., retargetable simulators for a variety of processor cores are required. In the proposed methodology, type and number of functional units, pipeline behavior and number of pipeline stages, instruction behavior in pipeline, and forwarding unit behavior are analyzed from functional unit definitions, pipeline stage definitions, forwarding unit definitions and instruction set. Simulators are generated by transforming them into C++ language description. The generated simulators evaluate processor performance and verify assembly codes for the processor core synthesized by the system. The experimental result shows the effectiveness of the proposed methodology.

1. まえがき

特定のディジタル信号処理アプリケーションに対し，プロセッサあるいはシステムのハードウェア部分とソフトウェア部分を同時に設計することにより，設計期間を短縮し，システム全体のハードウェア部分とソフトウェア部分のトレードオフを考慮した最適化を図る設計手法としてハードウェア/ソフトウェア協調設計がある．アプリケーションプログラムから適当なプロセッサ構成を得るハードウェア/ソフトウェア協調設計システムに関しては，COACH[1]，PEAS[2),3)]，ASIA[4)] の研究例がある．

現在，我々はディジタル信号処理向けプロセッサを対象としたハードウェア/ソフトウェア協調合成システム[5)]を構築している．システムは C 言語記述のアプリケーションプログラム，アプリケーションデータを入力とし，プロセッサコアのハードウェア記述，およびソフトウェア環境としてシミュレータ，コンパイラ，アセンブラを出力する．

ハードウェア/ソフトウェア協調合成システム[5)]に

† 早稲田大学理工学部電子・情報通信学科
Department of Electronics, Information and Communication Engineering, Waseda University
†† 北九州市立大学国際環境工学部情報メディア工学科
Department of Infomation and Media Sciences, The University of Kitakyushu
††† 早稲田大学理工学総合研究センター
Advanced Research Institute for Science and Engineering, Waseda University

よって合成されるプロセッサは，システムに入力されるアプリケーションプログラム，アプリケーションデータに対して最適化されている．しかし，アプリケーションプログラムやアプリケーションデータの一部を変更した場合，合成されたプロセッサによってアプリケーションの要求が満足されるかをシミュレーションによって評価し直す必要がある．

シミュレーションでは，対象プロセッサのパイプライン構造，機能ユニット，および命令セットによってアプリケーションの要求が満たされるかを評価するとともに，アプリケーションのアセンブリコードが対象プロセッサ上でハザードを起こさずに実行されるか検証することが求められる．また，異なるプロセッサ構成に対して正確なシミュレーションを行うために，シミュレータは合成されるプロセッサ構成に応じて，リターゲッタブルに生成される必要がある．

本稿では，ハードウェア/ソフトウェア協調合成システムにおいて，RISC 型から DSP 型まで多数のプロセッサ構成に対応可能なシミュレータの生成手法を提案する．シミュレータ生成系はプロセッサのパイプライン構成，機能ユニット，命令セット，フォワーディングユニットの VHDL 記述から，パイプラインレベルのシミュレータと命令セットレベルのシミュレータを生成する．

パイプラインシミュレータは次のように生成される．パイプライン構成記述より，パイプラインの段数，およびパイプライン各ステージで実行される動作を読み込み，その動作を C++記述に変換する．機能ユニット記述より，プロセッサに含まれる機能ユニットを読み込み，シミュレータ上で使用されるユニットを宣言する．命令セット記述から，各命令がパイプラインの各ステージで，どの機能ユニットによってどの動作を実行するか読み込み，ライブラリに用意された機能ユニット動作関数を指すポインタを生成する．ライブラリには各機能ユニットの動作が C++で記述されている．フォワーディングユニットの VHDL より，C++のフォワーディングユニットの動作記述を生成する．パイプラインレベルにおいて，プロセッサと同様に動作するフォワーディングユニットを持つことによって，命令間の複雑な依存関係を記述することなくハザードの検証を行うことができる．

命令セットシミュレータは命令セット記述から生成される．命令を読み込み，ライブラリ内の命令の動作を記述した関数を指すポインタを生成する．パイプラインハザードを考慮しないが，高速なシミュレーション実行を可能にする．

2. 既存研究

任意のプロセッサ構成の記述を入力とし，プロセッサ構成に基づいたシミュレータを生成するようなシステムは文献 6)~12) 等で報告されている．プロセッサを対象としたシミュレータには，主に命令セットシミュレータとパイプラインシミュレータがある．命令セットシミュレータは，命令レベルの動作をシミュレーションする．シミュレーション実行速度は比較的高速であるが，パイプライン構成で起こる遅延分岐や遅延レジスタ書き込みによるハザードのシミュレーションをするのが難しい．パイプラインシミュレータは，パイプライン各ステージ内の動作をシミュレートする．シミュレーション実行速度は比較的低速であるが，より詳細なシミュレーションができるため，遅延分岐や遅延レジスタ書き込みによるハザードをシミュレーションしやすい．

JACOB [6]や SuperSim [8]は，命令セットのモデルから，命令セットシミュレータを生成するシステムである．命令セットシミュレータであるため，パイプライン構成のプロセッサを対象とした場合，ハザードの検出が困難である．

プロセッサコアの命令拡張のための言語，TIE よりプロセッサコアおよびソフトウェア環境を生成する Xtensa システム [12)~14)]はシミュレータとして命令セットシミュレータを生成する．生成されるシミュレータは，入力される命令列を命令セットレベルの動作で実行し，サイクルカウントを求めることを目的としているため，ハザードが起きる場合を想定していない．

プロセッサを記述するための言語，nML [15)]や LISA [11),16)]からシミュレータを生成する手法がある．nML や LISA は命令レベルの動作を記述する言語であるため，遅延分岐や遅延レジスタ書き込みのシミュレーションが難しい．命令どうしの依存関係やパイプライン構成，命令タイプごとのパイプライン内動作を記述に追加することで，遅延分岐や遅延レジスタ書き込みのシミュレーションを実現するが，パイプライン構成が複雑になった場合，命令の依存関係が複雑化するため，LISA や nML の記述は自動生成が困難で，人手による記述を前提としている．

PD-file [9)]と呼ばれるプロセッサの動作を記述したファイルからシミュレータを生成する手法がある．入力には，各命令が何サイクル目にどのような動作を実行するかが定義されている．生成されるシミュレータはパイプラインシミュレータでありハザードの検出は容易だが，ハードウェアループのように，制御用の

機能ユニットを必要とするような命令は対象としていない．

既存手法はいずれも，命令セットレベル，またはパイプラインレベルの命令の動作記述を基本にシミュレータを生成する．既存手法はプロセッサに含まれる機能ユニットを考慮していないため，フォワーディングユニットやハードウェアループ等を持つプロセッサを対象とした場合，入力に特別な記述が必要となり，それによって対応アーキテクチャが制限される可能性がある．

シミュレータ生成系によって生成されるパイプラインシミュレータは，プロセッサの機能ユニットの動作を基準にシミュレーションを行う．パイプラインシミュレータは，フォワーディングユニットやハードウェアループユニット，アドレッシングユニットのように特別な動作をする機能ユニットの記述も持つことができるため，プロセッサがフォワーディングをする場合やハードウェアループユニットを使う命令を持つ場合でも，命令の依存関係等特別な記述を追加する必要がない．

3. アーキテクチャモデルと命令セット

3.1 アーキテクチャモデル

プロセッサのターゲットアーキテクチャとして，RISC アーキテクチャを持つ汎用のマイクロプロセッサからディジタル信号処理プロセッサにわたるものを考える．市販のディジタル信号処理プロセッサ[17]をもとに，**図 1** にプロセッサカーネルおよびプロセッサカーネルに付加されるハードウェアユニットの一部を示す．プロセッサカーネルに，いくつかのハードウェアユニットが付加されたものをプロセッサコアと呼ぶ．以下，ハードウェア/ソフトウェア協調合成において，合成の対象となるプロセッサカーネルおよびプロセッサカーネルに付加されるハードウェアユニットを定義する．

3.1.1 プロセッサカーネル

プロセッサカーネルとして，(i) RISC 型および，(ii) DSP 型のうちいずれかをとる．RISC 型は，文献 18) にならい，IF（命令フェッチ），ID（命令デコード），EXE（命令の実行），MEM（メモリアクセス），WB（書き込み）という 5 ステージのパイプライン構成をとる．DSP 型は，文献 17)，19)，20) にならい，IF，ID，EXE という 3 ステージのパイプライン構成をとる．プロセッサカーネルの各候補は，ハーバードアーキテクチャをとる．内部に，(c-i) 1 個の命令メモリ，(c-ii) 1 個のデータメモリ（X データメモリ），(c-iii) レジスタファイル RF_1，(c-iv) バレルシフタ，ALU（算術論理演算ユニット）を持つ．命令メモリおよび X データメモリのデータバス幅は変化させることができる．さらに，レジスタファイル RF_1 のレジスタ数およびビット幅を変化させることができる．命令メモリおよび X データメモリのアドレスバス幅は 16 ビットに固定される．ただし，X データメモリのデータバス幅はレジスタファイル RF_1 のビット幅と同一に設定される．命令メモリのデータバス幅は，命令セットから決定される．

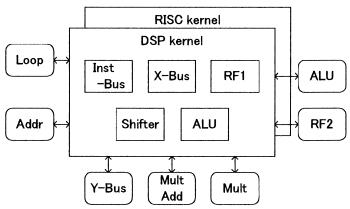

図 1 プロセッサカーネルおよび付加されるハードウェアユニット
Fig. 1 Processor kernels and hardware units added to them.

このような構成により，3.2 節に示す**表 1** の必須命令を実行可能となる．さらに，プロセッサカーネルの各候補は，複数の命令を同時に実行することができる．同時に実行可能な命令数は，あらかじめ与えられ固定とする．

3.1.2 ハードウェアユニット

プロセッサカーネルに次のようなハードウェアユニットを付加することができる．

Y データメモリおよびこれに付随する Y バスを付加できる．Y データメモリを使用する場合，X データメモリおよび Y データメモリを用いることで，データメモリから（に）並列に 2 個の値をロード（ストア）

表 1 基本命令（下線のある命令は必須命令）
Table 1 Basic instructions (underlined instructions are necessary instructions).

基本命令 1（算術命令）	ADD, SUB, SRA, SRL, SLL, AND, OR, XOR, MUL, DIV, SLT, SEQ, SNE, COM2, MAC, INC, DEC, ADDI, SUBI, SRAI, SRLI, SLLI, ANDI, ORI, XORI, MULI, DIVI
基本命令 2（ロード/ストア命令）	LDX, LDY, STX, STY, LDRX, LDRY, STRX, STRY, LDXI, LDYI, STXI, STYI, LDIX, LDIY, STIX, STIY, MV, IMM
基本命令 3（ジャンプ命令ほか）	BEQ, BNE, BZ, BNZ, JP, LOOP, RPT, CALL, RET, NOP, HLT
基本命令 4（並列ロード/ストア命令）	LDPX, STPX

することができる．Yデータメモリのデータバス幅およびアドレスバス幅は，Xデータメモリと同一とする．

演算ユニットの種類および個数を変化することができる．演算ユニットとして，シフタ，ALU，乗算器，乗加算器をとることができる．

アドレッシングユニットとして，(i) no operation, (ii) post increment, (iii) post decrement, (iv) index add, (v) modulo add, (vi) bit reverseのアドレス演算を実現できるものを考える[19),20)]．アドレス演算は，アドレスレジスタ（(ii), (iii)のアドレス演算），インデックスレジスタ（(ii)–(iv), (vi)のアドレス演算），モジュロレジスタ（(v)のアドレス演算）によって実現される．

プロセッサカーネルに対し，**ハードウェアループユニット**を付加できる．ハードウェアループは，ネストレジスタによって実現され，その数は変化させることができる．

プロセッサカーネルに対し，**レジスタファイル** RF_1 のレジスタ数，およびビット幅を変化させることができる．また，レジスタファイル RF_2 を付加できる．$RF2$ のレジスタ数およびビット幅は変化させることができる．

3.2 命令セット

合成されるプロセッサは命令セットとして基本命令および複合命令を持つ．基本命令は汎用のディジタル信号処理プロセッサの命令セット[17),19),20)]をもとにしており，プロセッサカーネルを構成するハードウェアに対応した命令である．基本命令を表1に示す．複合命令は，基本命令を複数個並列実行する命令である．基本命令のあらゆる組合せが複合命令となるのではなく，アプリケーションプログラムに依存して，どのような組合せの基本命令を複合命令とすべきかを決定する．異なるハードウェアユニットによって実行される命令が複合命令となりうる．

ハードウェア/ソフトウェア協調合成によって合成されるプロセッサコアがプロセッサとして動作するためには，いくつかの必要最小限の命令セットを持つ必要がある．基本命令の中でこのような命令を必須命令と呼び，アプリケーションプログラムによらず合成されるものとする．表1で下線がある命令が必須命令である．ハードウェア/ソフトウェア協調合成では，必須命令に加えアプリケーションプログラムに応じて，必要な基本命令を抽出あるいは複合命令を合成する．

4. ハードウェア/ソフトウェア協調合成システム

ディジタル信号処理向けプロセッサのハードウェア/ソフトウェア協調合成システムは，C言語で書かれたディジタル信号処理アプリケーションプログラム，アプリケーションデータを入力とし，プロセッサコアのハードウェア記述，プロセッサコア上で動作するオブジェクトコード，ソフトウェア環境（コンパイラ，シミュレータ）を出力する．アプリケーションプログラムを実行する時間を制約とし（時間制約と呼ぶ），面積最小のプロセッサコア合成を目的とする．プロセッサコアの面積は，図1で表されたプロセッサカーネルおよびプロセッサカーネルに付加される各ハードウェアユニットのうち，プロセッサコアに使用されるものの面積の和によって与えられる．アプリケーションプログラムの実行時間は，アプリケーションプログラムを実行するのに必要なクロックサイクル数とクロック周期との積によって与えられる．

図2にシステムの概要を示す．システムは，コンパイラ，ハードウェア/ソフトウェア分割，ハードウェア生成，ソフトウェア生成から構成される．**コンパイラ**はアプリケーションの実行に必要とされるすべてのハードウェアユニットを持つと仮定したプロセッサコ

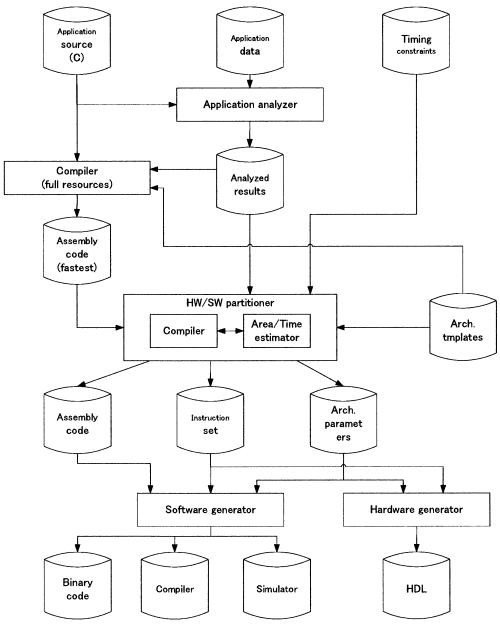

図2 ディジタル信号処理向けプロセッサのハードウェア/ソフトウェア協調合成システム

Fig. 2 The hardware/software cosynthesis system for processor cores of digital signal processing.

ア上で，アプリケーションの持つ最大限の並列性を抽出したオブジェクトコードを生成する．**ハードウェア/ソフトウェア分割**では，ハードウェアによる実現部の一部を徐々にソフトウェアによって代替することで，プロセッサコアの面積削減を図る．この操作は時間制約を満たす限り繰り返され，最終的にアプリケーションの実行時間が時間制約を満たし小面積のプロセッサコアを合成することが可能となる．**ハードウェア生成**，**ソフトウェア生成**ではハードウェア生成は，ハードウェア/ソフトウェア分割の結果から VHDL 記述を自動生成する．**ソフトウェア生成**は，ハードウェア/ソフトウェア分割の結果から，対象プロセッサ向けのシミュレータ，コンパイラを自動生成する．このようにして最終的に，プロセッサコアのハードウェア記述，プロセッサコア上で動作するアプリケーションプログラムのオブジェクトコードおよびソフトウェアを生成する．

5. シミュレータ生成手法

ハードウェア/ソフトウェア協調合成システム[5]は，合成されるプロセッサ向けのソフトウェア環境として，シミュレータ，コンパイラを生成する．ハードウェア/ソフトウェア協調合成システムのように，異なるアプリケーションに対し，異なるプロセッサ構成を生成する場合，シミュレータの構成は生成されるプロセッサ構成に対応しなければならない．シミュレータ生成系は，合成されるプロセッサの命令セット，パイプライン構成，付加機能ユニットに関する記述より，シミュレータをリターゲッタブルに生成することによってプロセッサコアの構成に対応する．

シミュレータ生成系によって生成されるシミュレータは，合成されるプロセッサ構成上で，対象アプリケーションがどの程度の時間で実行されるかを評価し，プロセッサの論理合成前に合成されたプロセッサの構成によってアプリケーションの要求が満たされるかを確認することを目的とする．また，シミュレータ生成系によって生成されるシミュレータはアセンブリコードが，合成されたプロセッサ上でハザードを起こさないか確認することも目的とする．

シミュレータ生成系は，パイプラインシミュレータと命令セットシミュレータの 2 種類のシミュレータを生成する．パイプラインシミュレータは，機能ユニットの動作を基準に，パイプラインステージ内の動作をシミュレーションすることによって，総クロックサイクル，データメモリ，レジスタファイル，およびパイプラインレジスタの値を出力できる．また，ハザードが起きた場合，シミュレーション結果にハザードが反映される．命令セットシミュレータはフォワーディング等は考慮せず，シミュレーションを高速に実行することを目的とする．総クロックサイクルとデータメモリ，レジスタファイルの値を出力できるが，ハザードの検証をしないため，ハザードが起こるようなアセンブリコードに対してもハザードを起こさずに実行してしまい，正確なシミュレーションをすることができない場合がある．

5.1 シミュレータ生成

5.1.1 入　　力

シミュレータ生成系には，**図 3**，**4**，**5**，**6** に示すような，ハードウェア/ソフトウェア分割された後のプロセッサコアに関する記述が入力される．また，フォワーディングユニットに関する情報として**図 7** に示すような VHDL 記述が入力される．

```
KERNEL kernel01 {
      if{
      imem_1.read(i_adrs, i_inst, i_rdy);
      pc_1.inc(i_adrs, me2ex_halt);
      PREG if2id(clock_sig, reset_sig);
      };
      id{
      DEC decode(i_inst);
      PREG id2ex(me2ex_halt,me_flush,clock_sig,reset_sig);
      };
      ex{
      EXU ex_unit_1();
      fwd_1.id_source(op1, op2, dat1, dat2);
      fwd_1.sink(fw_dat1, fw_dat2);
      PREG ex2me(clock_sig, reset_sig);
      };
      me{
      dmem_1.halt(me2ex_halt:direct);
      fwd_1.me_source(op_dst, ex_rslt, w_ctrl);
      PREG me2wb(clock_sig, reset_sig);
      };
      wb{
      fwd_1.wb_source(op_dst, me_rslt, w_ctrl);
      };
}
```

図 3　入力（パイプライン構成）
Fig. 3　Input (pipeline stage definitions).

```
UNIT {
     DMEM : dmem_1(group 1);
     IMEM : imem_1(group 1);
     CLOCK : clock_1(group 1);
     HALT : halt_1(group 1);
     PC : pc_1(group 1);
     FORWARD : fwd_1(group 1);
     ALU : alu_1(group 1),alu_2(group 1);
     ADD16 : add_2(group 1);
     SHIFT32 : shift_1(group 1);
     REGFILE : reg_1(type 1);
     PREG : if2id(type 2), id2ex(type 1),
            ex2me(type 2), me2wb(type 2);
     SYSCALL : syscall(type 1);
}
```

図 4　入力（機能ユニット）
Fig. 4　Input (functional units).

```
–alu type
code 1 {
     (31:26, inst);
     (25:24, op1);
     (23:22, op2);
     (21:20, op_dst);
}
–alui ls st beq imm type
code 2 {
     (31:26, inst);
     (25:24, op1);
     (23:22, op2);
     (21:20, op_dst);
     (15:0, imm);
}
–operation
operation {
     add(code 1, inst 1_1(op1, op2, op_dst));
     ldx(code 2, inst 26_1a(op1, op_dst, imm));
     beq(code 2, inst 30_1(op1, op_dst, imm));
}
```

図 5 入力（命令フォーマット）
Fig. 5 Input (instruction format).

```
– add r1, r2, r_dst
inst 1_1(r1, r2, r_dst) {
     if{};
     id{reg_1.read_w(r1, r2, dat1, dat2, w_ctrl)};
     ex{alu_2.add(fw_dat1, fw_dat2, ex_rslt)};
     me{syscall.assign(me_rslt, ex_rslt)};
     wb{reg_1.write(r_dst, me_rslt)};
} end;
– ldx r1, r_dst, imm
inst 26_1a(r1, r_dst, imm) {
     if{};
     id{reg_1.read_w(r1, r_dst, dat1, dat2, w_ctrl)};
     ex{add_2.add(fw_dat1, imm, ex_adrs)};
     me{dmem_1.read(me_rslt, ex_adrs)};
     wb{reg_1.write(r_dst, me_rslt)};
} end;
– beq r1, r_dst, imm
inst 30_1(r1, r_dst, imm) {
     if{};
     id{reg_1.read(r1, r_dst, dat1, dat2)};
     ex{add_1.zout(fw_dat1, fw_dat2, zout);
        add_2.add(i_adrs, imm, ex_adrs)};
     me{pc_1.beq(zout, ex_adrs, me_flush)};
     wb{};
} end;
```

図 6 入力（命令のパイプライン内動作）
Fig. 6 Input (instruction behaviors in pipeline).

図 3 は，パイプライン構成の記述である．図 3 は 5 ステージのパイプラインステージを持つ RISC 型プロセッサコアの例である．プロセッサコアは IF，ID，EX，ME，WB の 5 ステージからなる．IF ステージ中の，imem_1.read は命令メモリから命令を読み込むことを表す．imem_1.read のような機能ユニットの動作記述は命令に依存せずに必ず実行される．PREG if2id は IF ステージと ID ステージ間のパイプラインレジスタを表す．

図 4 は，プロセッサコアに含まれる機能ユニットの記述である．DMEM，IMEM 等はデータメモリ，命令メモリ等の機能ユニットの種類を表す．ALU の記

```
PORT ( id2ex_rs1 : in opnd;
     id2ex_rs2 : in opnd;
     id2ex_op1 : in data;
     id2ex_op2 : in data;
     ex2me_dst : in opnd;
     me2wb_dst : in opnd;
     ex2me_w : in std_logic;
     me2wb_w : in std_logic;
     me2ex_rslt: in data;
     wb2ex_rslt: in data;
     fw2ex_op1 : out data;
     fw2ex_op2 : out data;
     reset : in std_logic);

forward_unit_1 : process(ex2me_w,me2wb_w,eq1,eq2)
begin
     if((ex2me_w='1')and(eq1='1'))
       then alu_sel1_sig<="01";
     elsif((me2wb_w='1')and(eq2='1'))
       then alu_sel1_sig<="10";
     else alu_sel1_sig<="00";
     end if;
end process;
select_unit_1 :
  process(alu_sel1_sig,id2ex_op1,me2ex_rslt,wb2ex_rslt)
begin
     case alu_sel1_sig is
     when "00" => fw2ex_op1 <= id2ex_op1;
     when "01" => fw2ex_op1 <= me2ex_rslt;
     when "10" => fw2ex_op1 <= wb2ex_rslt;
     when others => fw2ex_op1 <= id2ex_op1;
     end case;
end process;

fwd_1_prcs : process(ex_op1, ex_op2, ex_dat1, ex_dat2)
BEGIN
     fwd_1_id2ex_rs1 <= ex_op1;
     fwd_1_id2ex_rs2 <= ex_op2;
     fwd_1_id2ex_op1 <= ex_dat1;
     fwd_1_id2ex_op2 <= ex_dat2;
END process;
```

図 7 入力（フォワーディングユニットの VHDL 記述）
Fig. 7 Input (forwarding unit (VHDL)).

述のように複数個のユニットを記述することもできる．

図 5 は，命令フィールド，命令セットに含まれる命令とその命令の種類の記述である．図 5 の code 1 は，31〜26 ビット目がオペコード，25〜24 および 23〜22 ビット目がオペランド，21〜20 ビット目がデスティネーションであるような命令フィールドを表している．operation 以下は，命令セットに add，ldx，beq，··· が含まれ，それぞれ code 1，code 2，code 2 の命令フィールドであり，図 6 の inst 1_1，inst 26_1a，inst 30_1 に従って，パイプラインの各ステージで動作することを表す．

図 6 は，各命令のパイプラインステージでの動作を表している．inst 1_1 は，ID ステージでレジスタ 1 から 2 つのデータを読み込み，EX ステージで ALU2 において加算を行い，ME ステージで実行結果を次のパイプラインレジスタに書き込み，WB ステージでレジスタ 1 に実行結果を書き込むということを表す．

図 7 は，フォワーディングユニットの VHDL 記述で

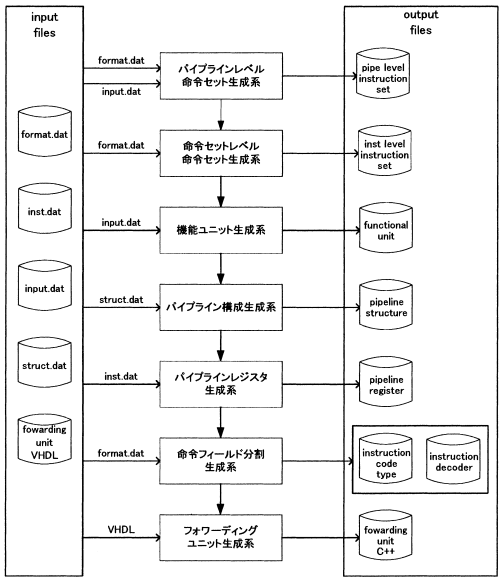

図 8 シミュレータ生成フロー
Fig. 8 Simulator generator.

ある．生成されるプロセッサコアはハザード回避のためにフォワーディングユニットを持つ．フォワーディングユニットの構成はプロセッサコアの並列度やパイプラインステージ数によって異なるため，フォワーディングユニットの VHDL 記述を入力とする．図 7 の上段はフォワーディングユニットのポートを，中段はフォワーディングを行う条件を，下段はパイプラインステージにおいてフォワーディングの対象となるデータをフォワーディングユニットに取り込む動作を表している．

5.1.2 生成フロー

シミュレータ生成系は**図 8** に示す手順で，C++によるシミュレータ記述を生成する．シミュレータ生成系はパイプラインレベル命令生成系，命令セットレベル命令生成系，機能ユニット生成系，パイプライン構成生成系，パイプラインレジスタ生成系，命令フィールド分割生成系，フォワーディングユニット生成系からなる．各生成系の処理を表したフローチャートを**図 9〜15** に示す．

パイプラインレベル命令セット生成系は，図 5 と図 6 を入力とし，以下のステップを経て**図 16** を生成する．パイプラインレベル命令セット生成系は図 5 の operation 以下の命令セットの記述より，命令セットに含まれる命令の動作を読み込む **(Step A1)**．たとえば，図 5 の add は inst 1_1 に従って動作する．続いて，読み込んだ命令と一致するパイプライン内の動作を図 6 より捜し出す **(Step A2)**．命令とパイプライン内の動作を関係付ける図 16 のようなポインタ関数の記述を生成する **(Step A3)**．図 16 の add_if から add_wb は，add 命令の各パイプラインステージにおける動作を表している．パイプラインステージにおける動作は，機能ユニットの動作，または機能ユニットの動作の組合せによって表される．機能ユニットの動作はあらかじめライブラリとして用意されている．

命令セットレベル命令セット生成系は，図 5 を入力とし，以下のステップを経て記述を生成する．命令セットレベル命令セット生成系は，図 5 の operation 以下の記述より命令セットに含まれる命令を読み込む **(Step B1)**．読み込んだ命令に対し，ライブラリにある命令セットレベルの命令動作を表した関数へのポインタ関数を生成する **(Step B2)**．

機能ユニット生成系は，図 4 を入力とし，以下のステップを経て**図 19** を生成する．図 4 の UNIT 以下に記述されている機能ユニットの種類とその機能ユニットの数を読み込む **(Step C1)**．読み込んだ機能ユニットより，図 19 のような C++のクラス宣言を生成する **(Step C2)**．

パイプライン構成生成系は，図 3 を入力とし，**図 17** を生成する．図 3 から，フォワーディングユニットを除いた機能ユニットの動作記述，フォワーディングユニットに関する動作記述，パイプラインレジスタの記述を分類して読み込む **(Step D1)**．読み込んだ記述のうち，機能ユニットの動作記述を図 17 の imem_1.read，pc_1.inc のように読み込んだ順に書き出す **(Step D2)**．続いて，フォワーディングユニットの動作記述を，図 17 における fwd_1 の記述のように fwd_1.sink が最後になるように書き出す **(Step D3)**．fwd_1 の id_source 関数，me_source 関数，wb_source 関数は，ID と EX の間，EX と ME の間，ME と WB の間のパイプラインレジスタから，フォワーディングユニットにレジスタ番地とデータを取り込む関数である．sink 関数はフォワーディングの条件に従い，フォワーディングユニットから演算器にデータを返す関数である．sink 関数が最後になるように書き出すことによって，該当クロックのデータに対しフォワーディングを行うことができる．次に，パイプラインステージの数だけ，ポインタ関数（op_code_ex 関数）を書き出す **(Step D3)**．op_code_ex は，図 16 の各命令のパイプラインステージ内動作を表した関数へのポインタ関数である．最後に，パイプラインレジスタの記述をパイプラインステージの逆順になるように書き出す **(Step D4)**．

パイプラインレジスタ生成系は，図 6 を入力とし，

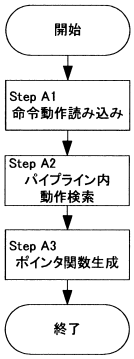

図 9 パイプラインレベル命令セット生成フロー
Fig. 9 Pipelien level instruction set generation flow.

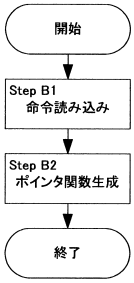

図 10 命令セットレベル命令セット生成フロー
Fig. 10 Instruction level instrcution set generation flow.

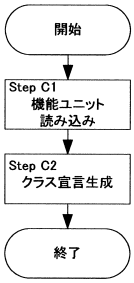

図 11 機能ユニット生成フロー
Fig. 11 Functional unit generation flow.

図 12 パイプライン構成生成フロー
Fig. 12 Pipeline structure generation flow.

図 13 パイプラインレジスタ生成フロー
Fig. 13 Pipeline register generation flow.

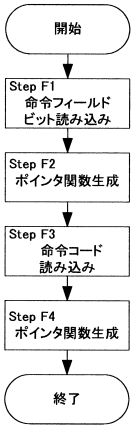

図 14 命令フィールド分割生成フロー
Fig. 14 Instruction field division generation flow.

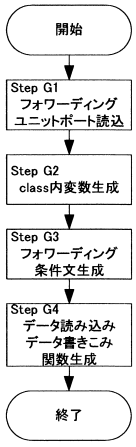

図 15 フォワーディングユニット生成フロー
Fig. 15 Forwarding unit generation flow.

以下のステップを経てパイプラインレジスタを表した構造体の記述を生成する．図 6 のパイプライン内動作の引数からパイプラインレジスタに必要な変数を読み込む **(Step E1)**．読み込んだ変数を含む構造体の記述を生成する **(Step E2)**．

命令フィールド分割生成系は，図 5 を入力とし，以下のステップを経て**図 20**，**図 21** のような記述を生成する．ハードウェア/ソフトウェア協調合成システム[5]は，合成されるプロセッサコアの命令数やレジスタ数によってオペコードやオペランドのビット数を決めるため，アプリケーションごとに命令フィールドが異なる．したがって，シミュレータ生成系はプロセッサコアごとに命令フィールドからオペコード，オペランドを取り出す機構を生成する必要がある．命令フィールド分割生成系は，図 5 の code 以下の記述から各 code のオペコード，オペランド，即値等が何ビット目にあたるのか読み込む **(Step F1)**．読み込まれた命令フィールドの情報より図 20 のようなポインタ関数の記述を生成する **(Step F2)**．続いて，図 5 の operand 以下の記述より，各命令の code の種類を読み込む **(Step F3)**．読み込まれた命令の code の種類より，図 21 のようなポインタ関数を生成する **(Step F4)**．

フォワーディングユニット生成系は，図 7 のようなフォワーディングユニットの VHDL 記述を入力とし，以下のステップを経て**図 22** のような class 記述を生成する．フォワーディングユニット生成系は，図 7 よりフォワーディングユニットのポートを読み込む **(Step G1)**．読み込まれたポートより，図 22 のような class

```
void add_if(PREG& signal)={ };
void add_id(PREG& signal)=
   { reg_1.read_w(r1, r2, dat1, dat2, w_ctrl); };
void add_ex(PREG& signal)=
   { alu_2.add(fw_dat1, fw_dat2, ex_rslt); };
void add_me(PREG& signal)=
   { syscall.assign(me_rslt, ex_rslt); };
void add_wb(PREG& signal)=
   { reg_1.write(r_dst, me_rslt); };

void sub_if(PREG& signal)={ };
void sub_id(PREG& signal)=
   { reg_1.read_w(r1, r2, dat1, dat2, w_ctrl); };
void sub_ex(PREG& signal)=
   { alu_1.sub(fw_dat1, fw_dat2, ex_rslt); };
void sub_me(PREG& signal)=
   { syscall.assign(me_rslt, ex_rslt); };
void sub_wb(PREG& signal)=
   { reg_1.write(r_dst, me_rslt); };
...

void (*add[])(PREG& signal)=
   {add_if, add_id, add_ex, add_me, add_wb, };
void (*sub[])(PREG& signal)=
   {sub_if, sub_id, sub_ex, sub_me, sub_wb, };
...

void (**op_code_ex[])(PREG& signal)={ add, sub, ··· };
```

図 16 命令セットとパイプライン内動作の関連
Fig. 16 Relationship between instruction set and instruction behavior in pipeline.

```
inline void kernel(void){
     imem_1.read(i_adrs, i_inst, i_rdy);
     pc_1.inc(i_adrs, me2ex_halt);
     fwd_1.id_source(id2ex);
     fwd_1.me_source(ex2me);
     fwd_1.wb_source(me2wb);
     fwd_1.sink(id2ex);

     op_code_ex[i_inst.op_code][if](i_inst);
     op_code_ex[if2id.op_code][id](if2id);
     op_code_ex[id2ex.op_code][ex](id2ex);
     op_code_ex[ex2me.op_code][me](ex2me);
     op_code_ex[me2wb.op_code][wb](me2wb);

     me2wb=ex2me;
     ex2me=id2ex;
     id2ex=if2id;
     if2id=i_inst;
}
```

図 17 パイプライン構成の C++記述
Fig. 17 Pipeline structure (C++).

記述内の変数を書き出す **(Step G2)**．続いて，図 7 の中段より，フォワーディングする条件を読み込み，図 22 における sink 関数の if 文のような記述を生成する **(Step G3)**．次に，図 7 の下段のような記述より，パイプラインステージ内のデータをフォワーディングユニット内の変数に読み込む記述を生成する **(Step G4)**．図 7 の下段からは，図 22 の id_source 関数のような記述を生成する．また，図 22 の sink 関数の下 2 行のように，フォワーディングユニットからパイプラインステージにデータを返す記述も生成する．

```
class ALU{
public:
     add(int dat1, int dat2, int& ex_rslt);
     sub(int dat1, int dat2, int& ex_rslt);
     ···
};

ALU::add(int dat1, int dat2, int& ex_rslt){
     ex_rslt=dat1+dat2;
}
ALU::sub(int dat1, int dat2, int& ex_rslt){
     ex_rslt=dat1-dat2;
}
...
```

図 18 機能ユニットの C++記述
Fig. 18 Functional units (C++).

```
DMEM dmem_1;
IMEM imem_1;
CLOCK clock_1;
HALT halt_1;
PC pc_1;
FORWARD fwd_1;
ALU alu_1,alu_2;
ADD16 add_2;
SHIFT32 shift_1;
REGFILE reg_1;
PREG if2id,id2ex,ex2me,me2wb;
SYSCALL syscall;
```

図 19 機能ユニットの宣言
Fig. 19 Functional unit definitions.

```
int code1(int *binary_oprand, PREG& decimal_oprand){
     decimal_oprand.r1=
        transform_2to10_u(binary_oprand,25,24);
     decimal_oprand.r2=
        transform_2to10_u(binary_oprand,23,22);
     decimal_oprand.r_dst=
        transform_2to10_u(binary_oprand,21,20);
     return 0;
};
...

int (*transform_binary_oprand_to_decimal_oprand[])
     (int *binary_oprand,PREG& decimal_oprand)=
{code0,code1,code2,code3,code4,code5,code6,};
```

図 20 命令フィールド分割
Fig. 20 Instruction field division.

```
int add_code(void)return 1;;
int sub_code(void)return 1;;

int (*search_code_type[])(void)={
add_code,sub_code, ···};
```

図 21 命令コード定義
Fig. 21 Instruction field definitions.

5.2 シミュレータ動作

シミュレータ生成系は，合成されるプロセッサコア向けのシミュレータとして，パイプラインシミュレータと命令セットシミュレータを生成する．

```
class FORWARD{
    int id2ex_rs1,id2ex_rs2,id2ex_op1,id2ex_op2,
    ex2me_dst,me2wb_dst, ex2me_w,me2wb_w,me2ex_rslt,
    wb2ex_rslt,fw2ex_op1,fw2ex_op2,reset;
public:
    id_source(PREG signal);
    sink(PREG& signal);
    me_source(PREG signal);
    wb_source(PREG signal);
};
FORWARD::sink(PREG& signal){
    if(id2ex_rs1==id2ex_rs1) fw2ex_op1 = id2ex_op1;
    if(id2ex_rs1==ex2me_dst) fw2ex_op1 = me2ex_rslt;
    if(id2ex_rs1==me2wb_dst) fw2ex_op1 = wb2ex_rslt;
    if(id2ex_rs2==id2ex_rs2) fw2ex_op2 = id2ex_op2;
    if(id2ex_rs2==ex2me_dst) fw2ex_op2 = me2ex_rslt;
    if(id2ex_rs2==me2wb_dst) fw2ex_op2 = wb2ex_rslt;
    signal.dat1 = fw2ex_op1;
    signal.dat2 = fw2ex_op2;
}
FORWARD::id_source(PREG signal){
    id2ex_rs1 = signal.r1;
    id2ex_rs2 = signal.r2;
    id2ex_op1 = signal.dat1;
    id2ex_op2 = signal.dat2;
}
...
```

図 22 フォワーディングユニット記述
Fig. 22 Forwarding unit description.

パイプラインシミュレータは，シミュレータ生成系によって生成された記述と，あらかじめ用意した，**図 18** に示す各機能ユニットの class 記述と，シミュレーションの最も外側のループ等，プロセッサ構成に依存しない基本的な部分の記述によって構成される．

パイプラインシミュレータには，対象プロセッサ専用にコンパイル，アセンブルされたアプリケーションのバイナリコードが入力される．入力されたバイナリコードは，図 20，図 21 によってオペランド，即値等に分割され，10 進表現に変換される．変換されたデータは，パイプラインレジスタ生成で生成された PREG 型の構造体に格納される．

パイプラインシミュレータは，図 17 のパイプライン構成記述を繰り返し実行し，繰返し 1 回を 1 クロックとしてシミュレーションを行う．図 17 上段の機能ユニット，フォワーディングユニットの動作は命令にかかわらず実行される．図 17 中段の op_code_ex は，図 16 で宣言した命令のパイプラインステージ内動作を表す関数へのポインタ関数である．EX ステージに add 命令がある場合，op_code_ex[id2ex.op_code][ex] は図 16 の add_ex を指す．op_code_ex が，各パイプラインステージについて同様に動作することによって，命令の動作をシミュレートする．図 17 下段では，命令の実行によって変更されたパイプラインレジスタの値を次のパイプラインレジスタへ書き込む．

命令の各パイプラインステージでの動作は，図 18 に示す機能ユニットの動作を表した関数を基準とする．プロセッサコアに含まれる機能ユニットは図 19 のように宣言される．

フォワーディングは，図 22 のフォワーディングユニットの class 表現とパイプライン構成記述内の fwd 以下の関数によって実現される．フォワーディングの対象となるデータは，図 17 の source 関数によってフォワーディングユニットに取り込まれる．sink 関数はフォワーディングの条件に従い，EX ステージにデータを返す．フォワーディングユニット生成系によって生成されるフォワーディングユニットは，合成されるプロセッサコアのフォワーディングユニットと同じデータに対し，同じ条件でフォワーディングを行う．これによって，命令間の依存関係を定義することなくハザードの検証が可能になる．

命令セットシミュレータには，パイプラインシミュレータ同様，アプリケーションのバイナリコードが入力され，図 20，図 21 によってオペランド，即値等に分割され，PREG 型の構造体に格納される．命令セットシミュレータは命令セットレベルの動作を表した関数へのポインタ関数を繰り返し実行し，繰返し 1 回を 1 クロックとする．命令セットレベルの動作関数はライブラリに用意されている．ハザードは起こらないことを前提としているため，命令の動作のみ実行する．

6. 計算機実験結果

シミュレータ生成系は C 言語を用いて実装されている．シミュレータ生成系を，**表 2** に示す 3 種類のプロセッサについて適用した．各プロセッサ構成のシミュレータ生成時間は 0.3 秒から 0.6 秒程度であり，十分高速に生成されている．

シミュレータ生成系によって生成されたシミュレータに，1024 次 FIR フィルタを実行させた際のシミュレーション実行速度を**表 4** に示す．シミュレーションは ULTRA SPARC2 400 MHz，メモリ 1024 MB，gcc 2.8.1，最適化オプション-O1 の環境で行った．既存手法のシミュレーション実行速度を**表 3** に示す．本手法によって生成されたシミュレータは，既存手法と比較して同等，またはそれ以上の性能を実現することができている．

提案手法は，以上に示したように，単純な命令からなる RISC 型のプロセッサから，ハードウェアループユニットやアドレッシングユニットといったディジタル信号処理特有の機能ユニットを有する DSP 型のプロセッサまで，幅広いアーキテクチャのプロセッサを生成できる．提案手法の入力は，ハードウェア/ソフ

表 2 生成したシミュレータのプロセッサ構成
Table 2 Processor structures of generated simulators.

	パイプ段数	命令数	並列度	HW ループ	アドレッシング
RISC	5 段	39 命令	1 並列	無	無
DSP1	3 段	33 命令	1 並列	有	有
DSP2	3 段	36 命令	2 並列	有	有

表 3 既存手法によるシミュレータ実行速度
Table 3 Simulation execution time.

手法	実験環境	速度 (命令/sec)
JACOB [6]	SPARC Station 20, Memory 256 MB	$0.1\sim0.5\times10^6$
SuperSim [8]	Sun-10, Memory 64 MB	$0.4\sim2.5\times10^6$
文献 7)	Ultra SPARC	$0.02\sim0.15\times10^6$
Checkers [10]	Sun Ultra-5	0.1×10^6
LISA [11),16)]	SPARC Ultra 10	$0.2\sim0.5\times10^6$
PD-file [9]	Sun	$0.2\sim0.3\times10^6$

表 4 本手法によるシミュレション実行速度
Table 4 Simulation execution time (proposed).

プロセッサ	シミュレーションレベル	速度 (命令/sec)
RISC	パイプラインレベル	0.21×10^6
	命令セットレベル	6.15×10^6
DSP1	パイプラインレベル	0.23×10^6
	命令セットレベル	1.54×10^6
DSP2	パイプラインレベル	0.22×10^6
	命令セットレベル	1.54×10^6

トウェア協調合成システムが対象とする，すべてのプロセッサの表現が可能であるため，提案手法によって，システムが対象とする RISC 型，または DSP 型のパイプライン構成を持つプロセッサ，任意の機能ユニットを有するプロセッサ，任意の命令セットを実行可能なプロセッサ，任意の並列度のプロセッサについて，シミュレータの生成が可能になった．

また，本手法によって生成されるシミュレータは既存手法と同等以上のシミュレーション実行速度を実現できた．

7. むすび

本稿では，ディジタル信号処理プロセッサを対象としたハードウェア/ソフトウェア協調合成システムにおけるシミュレータ生成手法を提案した．また，シミュレータ生成系によって生成されるシミュレータは RISC 型から DSP 型まで幅広いアーキテクチャに対応し，既存手法と比較して遜色ない実行速度を実現することを示した．

今後は，コンパイル手法によるシミュレーション実行時間の高速化を図るとともに，SIMD 型命令や複数サイクル命令に対応し，対応アーキテクチャの拡大を目指す．また，入力するデータに誤りが含まれないように，入力を検証する機構の導入を検討する．

参考文献

1) Akaboshi, H. and Yasuura, H.: COARCH: A computer aided design tool for computer architects, *Trans. IEICE*, Vol.E76–A, No.10, pp.1760–1769 (1993).
2) Alomary, A., Nakata, T., Honma, Y., Imai, M. and Hikichi, N.: An Adaptive Noiseless Coding for Sources with Big Alphabet Size, *ICCAD-93*, pp.526–532 (1993).
3) Bình, N.N., Imai, M., Shiomi, A. and Hikichi, N.: A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate count, *33rd DAC*, pp.527–532 (1996).
4) Huang, I.-J. and Despain, A.M.: Synthesis of instruction sets for pipelined microprocessors, *31st DAC*, pp.5–11 (1994).
5) Togawa, N., Yanagisawa, M. and Ohtsuki, T.: A hardware/software cosynthesis system for digital signal processor cores, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E82-A, No.11 (1999).
6) Leupers, R., Elste, J. and Landwehr, B.: Generation of Interpretive and Compiled Instruction Set Simulators, *ASP-DAC '99*, pp.339–342 (1999).
7) Hartoog, M.R., Rowson, J.A., Reddy, P.D., Desai, S., Dunlop, D.D., Harcourt, E.A., Khullar, N. and Alta Group of Cardence Design Systems, I.: Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign, *34th DAC* (1997).
8) Zivojnovic, V. and Meyr, H.: COMPILED HW/SW CO–SIMULATION, *33rd DAC* (1996).
9) Engel, F., Nuhrenberg, J. and Fettweis, G.P.: A Generic Tool Set for Application Specific Processor Architectures, *CODES 2000* (2000).
10) Target Compiler Technologies: *The Chess/Checkers: A Retargetable DSP Compilation Environment — Technical Write Paper* (2000).
11) Peens, S., Hoffmann, A., Zivojnovic, V. and Meyr, H.: LISA-Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures, *36th DAC* (1999).
12) Wang, A., Killian, E., Maydan, D. and Rowen C.: Hardware/Software Instruction Set Cinfigurability for System-on-Chip Processors, *DAC*

2001 (2001).
13) Tensilica, Inc.: Xtensa Microporcessor overview handbook, http://www.tensilica.com/dl/hand-book.pdf
14) Gonzalez, R.: XTENSA: A Configurable and Extensible Processor, *IEEE micro*, Vol.20, No.2, pp.60–70 (2000).
15) M. Freericks: *The nML Machine Description Formalism* (1993).
16) Peens, S., Hoffmann, A. and Meyr, H.: Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language, *The Design, Automation and Test in Europe* (2000).
17) NEC：信号処理LSI（DSP/音声）データブック (1996).
18) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, Morgan-Kaufman (1990).
19) Lapsley, P., Bier, J., Shoham, A. and Lee, E.A.: *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology (1996).
20) Madisetti, V.K.: *Digital Signal Processors*, IEEE Press (1995).

(平成13年9月15日受付)
(平成14年3月14日採録)

笠原　亨介

2000年早稲田大学理工学部電子・情報通信学科卒業．2002年同大学大学院修士課程修了．同年ソニー（株）入社．電子回路の設計自動化，特にハードウェア/ソフトウェア協調設計に関する研究に従事．

戸川　望（正会員）

1992年早稲田大学理工学部電子通信学科卒業．1994年同大学大学院修士課程修了．1997年同後期課程修了．博士（工学）．1997年早稲田大学理工学部電子・情報通信学科助手．2000年早稲田大学理工学総合研究センター講師を経て，現在北九州市立大学国際環境工学部情報メディア工学科助教授，早稲田大学理工学総合研究センター客員助教授．電子回路の設計自動化，計算幾何学，グラフ理論等の研究に従事．1996年第9回安藤博記念学術奨励賞受賞．1997年度（第21回）丹羽記念賞受賞．IEEE，電子情報通信学会各会員．

柳澤　政生（正会員）

1981年早稲田大学理工学部電子通信学科卒業．1984年同大学大学院博士前期課程修了．1986年同後期課程修了．工学博士．1984年早稲田大学情報科学研究教育センター助手．1986年カリフォルニア大学バークレー校研究員．1987年拓殖大学工学部情報工学科助教授を経て，現在早稲田大学理工学部電子・情報通信学科教授．LSI設計と設計自動化技術，ゲノム解析等の研究に従事．1988年度丹羽記念賞受賞．1990年安藤博学術奨励賞受賞．IEEE，ACM，電子情報通信学会，日本OR学会各会員．

大附　辰夫（正会員）

1963年早稲田大学理工学部電気通信学科卒業．1965年同大学大学院修士課程修了．同年日本電気（株）入社．1980年同退社．現在，早稲田大学理工学部電子・情報通信学科教授．博士（工学）．システムLSIおよびこれに関連した基礎研究に従事．1969年度電子情報通信学会論文賞受賞．1994年度第32回電子情報通信学会業績賞受賞．IEEE CAS Societyより Guillmin-Cauer Prize Award（1974年），Meritorious Service Award（1995年），Golden Jubilee Medal（2000年）受賞．2000年IEEEより3rd Millennium Medal受賞．共著「VLSIの設計I」（岩波書店），編共著「Layout Design and Verification」（North-Holland）．IEEEフェロー，電子情報通信学会フェロー，電気学会，プリント回路学会各会員．