高級言語により記述されたアルゴリズムを実現する専用プロセッサ設計支援システム†

池永 剛††* 白井克彦††

LSI 技術の進歩により，様々な専用 LSI の実現が可能になり，その優れた支援システムの実現が重要となっている．そこで，幅広い分野のユーザに対してプロセッサ設計環境を与えることを目的として，高級言語によるアルゴリズム記述を入力とすることにより，これを実行する LSI 設計を可能にする専用プロセッサ設計支援システムの構築を行った．本システムの特徴は，専用 LSI とすることでハードウェアが小規模になるだけでなく，設計された LSI を利用するソフトウェア開発環境にも優れたプロセッサ設計が可能なことである．本論文では，システムの主眼点，処理概要について説明した後で，専用プロセッサの設計例を示し，その評価を行う．複数の信号処理アルゴリズムを実行する専用プロセッサを設計し，汎用 DSP と比較する．

1. はじめに

LSI 技術の進歩に伴い，その規模は年々増大し，応用分野も急激に広がりつつある．このため，信号処理プロセッサ[1]のような様々な専用プロセッサが製作され，実用に供されるようになってきている．

その一方で，これらの LSI は複雑化，多様化し，そのシステム設計に多くの時間とコストが必要となってきている．現在，システムの大規模化による大量のデータ処理，リアルタイム処理等の要求が高まる中で，高速処理を可能とする特定用途向け LSI は必要不可欠な存在になりつつあり，今後，その需要はますます高まっていくと考えられる．そこで，このような専用 LSI を容易に設計可能な環境を実現することが重要になってくると考えられる．また，もう一つ重要なポイントとして，設計された LSI についてテストも含め，利用者自身が使用しやすいソフトウェア環境も同時に生成されることがあげられる．これらの実現の一つの形態として，その LSI の利用者自身の仕様記述に基づく LSI 設計支援システムが考えられる．このためには，ユーザが通常用いているソフトウェアの概念に近い，高位の入力記述を持ったシステムが望まれる．

われわれは，このような背景をもとに，幅広い分野のユーザを対象とした，専用プロセッサ設計支援システムを提案し，研究を行っている[2)-4)]．このシステムは，通常の高級言語で書かれたアルゴリズム記述を入力とし，そのアルゴリズムを効率よく実行するマイクロプロセッサの設計支援を行うことを目的としている．また，それと同時に，設計されたプロセッサに対する最適化コンパイラを自動合成することにより，ハードウェアのみならず，ソフトウェアを含めた統合システムの実現を目的としている．

本論文では，専用プロセッサ設計支援システムの主眼点，処理概要，設計アプローチについて説明する．そして，このシステムを用いて，様々な専用プロセッサを設計し，その評価を行う．

適用例としては，代表的なディジタル信号処理アルゴリズムとクイックソートの例を取り上げる．まず，PARCOR 格子型フィルタを用いて，プロセッサ設計過程を示す．次に複数の信号処理アルゴリズムに対しプロセッサ設計を行い，ハードウェア要素などの評価を行う．さらに，本システムで設計されたプロセッサの評価のために，代表的な DSP である TMS 320C 25 を取り上げ，ハードウェアとソフトウェアの面から比較検討する．また，本システムによる設計結果は，アルゴリズム記述の方法にかなり依存してくるので，クイックソートの設計例を通して，その影響について考察する．

2. 専用プロセッサ設計支援システムの主眼点

本システムは，LSI 設計を支援するシステムであるが，従来の LSI CAD のようにハードウェア設計としての枠組みに縛られず，アーキテクチャ，ソフトウェアを含めた統合システムを考える．LSI CAD[5)]，プロセッサ[6),7)]，コンパイラ[8),9)] 研究の現状を踏まえ

† Special Purpose Processor Design System Realizing Algorithm Described by Higher Level Language by TAKESHI IKENAGA and KATSUHIKO SHIRAI (Department of Electrical Engineering, School of Science and Engineering, Waseda University).

†† 早稲田大学理工学部電気工学科
* 現在 NTT LSI 研究所

て，本システムの基本となる三つの主眼点について述べる．

2.1 幅広いユーザに対するプロセッサ設計環境

ハードウェアやプロセッサの専門家に限らず，システムの高速化を望む幅広い分野のユーザに対して，プロセッサ設計環境を与えることを本システムの一番の主眼点に掲げる．ただし，現在のところはある限定された範囲のアルゴリズムを実行することに特化したプロセッサの設計を目的としている．

（1） 高位の入力記述

LSI CAD の上位レベルにおける仕様記述を可能にするために，VHDL のような専用言語が作られ，その処理系も発展しつつあるが，現状では RT レベルが一般的である．RT レベルからのプロセッサ設計を行うにはハードウェアの構成要素やタイミング等に関する専門知識が必要である．これがハードウェアの専門家以外のユーザをプロセッサ設計の現場に近づきにくくする要因になっている．よって，対象ユーザを決めるのに，システムの入力を何にするかが重要なポイントになる．

本研究では，ユーザの要求仕様を比較的容易に，生産性高く記述できるものとして，Cや Pascal といった通常の手続き型の高級言語を考えた．これらの言語は，現在既に，システム設計を行う際の仕様記述言語として，多くのユーザに受け入れられており，ユーザに LSI 設計のための特別な知識の修得が少なく，幅広い分野を対象とした入力言語としては，最も優れていると考えられる．これらの言語は，現状では，コンパイラを用いて特定の汎用のプロセッサのマシン語に変換され，その上で実行されると仮定されており，ソフトウェア記述という限定された用途に用いられている．しかし，そこに記述されているアルゴリズムから，プロセッサの仕様情報を抽出することによって，本システムのようなハードウェア設計システムの入力としても有効であると考える．そこで，本システムでは，Pascal を入力記述とした．

（2） プロセッサ記述

プロセッサ設計は，命令セット，内部構成，動作仕様等多くの点を考えなければならないが，それにはプロセッサに関する高度な専門知識が要求される．本システムは Pascal を入力記述としているが，これを用いて上に示したプロセッサ仕様をすべて与えなければならないとすると，やはり対象ユーザを限定してしまう．

本論文では，限定されているが，かなり広い一般性を持つプロセッサ構成を考える．ここで設計されるプロセッサの基本的な枠組みは，命令部とデータ部を分離したハーバード・アーキテクチャとして，演算に関してはレジスタ間を基本とする．したがって，RAM に対するアクセスはロード・ストアのみとする．固定要素は，プログラムカウンタ，インストラクション ROM，インストラクションレジスタ，レジスタファイル，ALU，入出力ポートであり，可変要素は，ALU で行える演算等の命令の種類（命令セット），全体のデータ型，RAM，ROM，レジスタファイル等の個数，Band width である．内部動作のタイミングは，フェッチ・デコード，命令実行，結果格納の3マシンサイクルを基本とする．ただし，複合命令については命令実行時間が2マシンサイクルになることもある．バスは，制御用の Ctr-Bus が1本，データ用バスが転送用に必要に応じた本数，結果用R-Bus 1本を想定されている．したがって，可変要素は，命令，記憶要素（レジスタ，RAM 等），データ型に限定することができ，その他の部分は固定要素として，どのプロセッサでも，ほぼ共通と考える．したがって，本システムではプロセッサの詳細設計をゼロから行うのではなく，プロセッサの枠組み（固定要素）はテンプレートとして用意し，それに対して入力アルゴリズムを解析し，可変要素を決め，チューニングしていくという形をとっている．

2.2 速度性能を重視したマイクロプロセッサ

専用プロセッサを考える上で，速度性能は最重要項目であり，本システムでも，プロセッサの速度性能を向上させることを主眼点に考える．一般に速度性能をあげるための技術として，セマンティクスギャップの縮小，並列処理，ウェアの強化，高速スイッチング素子の採用がある．本システムで，これらの項目をどの様に考慮しているかについて述べる．

（1） セマンティックギャップの縮小

ソフトウェアと命令セットのセマンティックギャップを縮小することによって，速度性能向上が可能である．このため，ソフトウェア概念に近い高機能命令が望まれる．しかし，汎用プロセッサの場合それらの高機能命令は実際に使われる頻度は小さいため，高機能命令導入によってハードウェアが複雑になる割には効果が得られていない．

本システムでは専用プロセッサ設計としての特徴を生かし，有効な高機能命令は積極的に取り入れ，セマ

ンティックギャップの縮小をはかる．よって，命令セットの設計においては，最初入力アルゴリズムを実行するのに必要最低限のものとし簡略化を図り，その上でアルゴリズムのセマンティクスを解析し，有用で高頻度な高機能命令を取り入れていくこととした．

（2） 並 列 処 理

本システムでは，与えられたアルゴリズムが持つ並列性を必要なだけ生成することを考える．具体的には，本システムではプロセッサの制御方法として，プログラムカウンタ制御を行っているが，並列性を引き出すために高頻度に実行される部分をデータフロー制御にすることなどを考える．

（3） ウェアの強化

一般にソフトウェアよりもファームウェア，ファームウェアよりもハードウェアを用いたほうが高速である．このため，機能をできるだけハードウェア寄りに実現することを想定しており，命令は布線論理を用いて実現し，それらをマイクロ命令を用いて直接実行（RISC 制御）させることを基本と考える．

（4） 高速スイッチング素子

高速スイッチング素子として，ECL，GaAs 等がある．これらの素子を利用するためには，ゲート数をできるだけ抑える必要がある．本システムは，専用プロセッサとして特定アルゴリズムを実行するのに必要な最低限のハードウェア構成（命令数，レジスタ数，RAM の大きさ等）とするのでゲート数は汎用プロセッサと比べて少なくとすみ，高速素子を利用できる可能性がある．また，この要因を考慮しながら，高機能命令導入等のトレードオフを評価することができる．

2.3 ソフトウェアを含めた総合システム

本システムは，単に専用プロセッサの性能向上のみを考えるのではなく，ソフトウェアを含めた利用環境の利便を考慮した総合評価の向上を考える．よって，専用プロセッサを設計支援するのと同時に，設計されたプロセッサに対する最適化コンパイラを自動合成し，ソフトウェア開発のため環境として提供する．このことによって，最初に与えたアルゴリズムのわずかな変更や類似のアルゴリズムに対して対応が容易になり，設計されたプロセッサのテストも含め，利用上の柔軟性が高くなる．

3. 処 理 概 要

本システムは，中間情報生成系，解析系，合成系，コード生成系の四つの系から構成される（**図1**）．

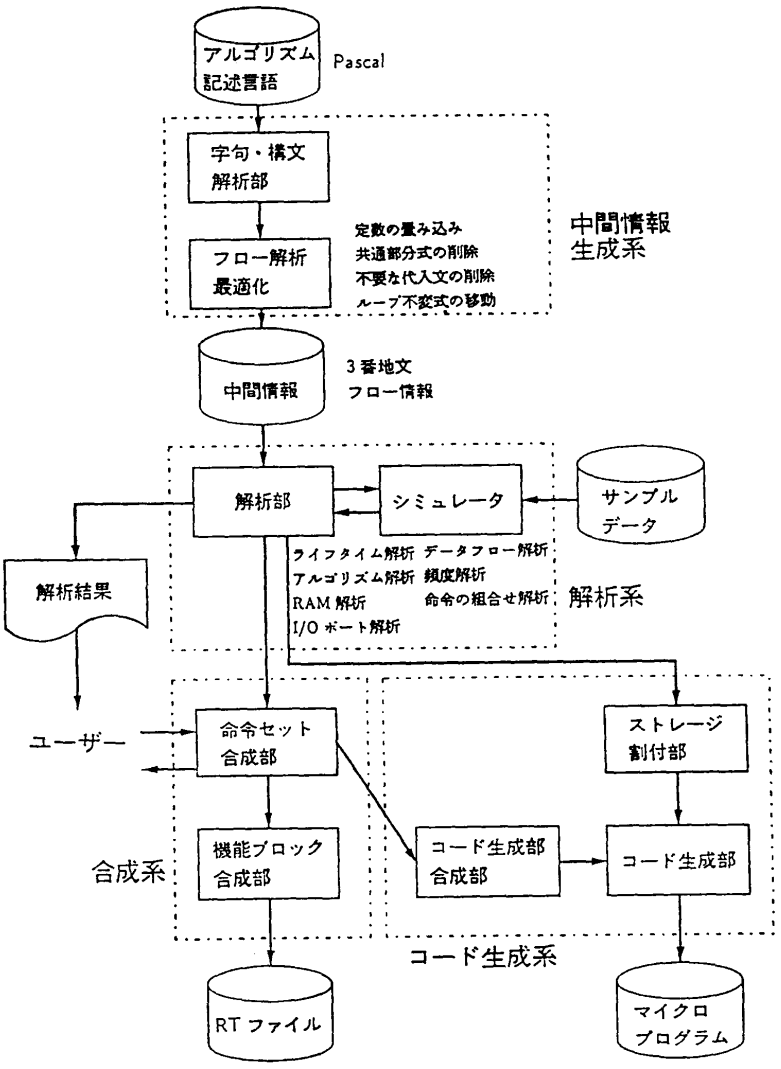


図 1 システムの構成
Fig. 1 Block diagram of the system.

中間情報生成系は，Pascal で記述された入力アルゴリズムに対し字句・構文解析を行い，種々の最適化を行った後，中間情報（3番地文等）を生成する．

解析系は，中間情報に対しライフタイム解析，アルゴリズム解析，RAM 解析，I/O ポート解析，頻度解析，命令の組合せ解析，データフロー解析等の解析を行う．前者四つは入力アルゴリズムの実現に最低限どれだけの命令，ハードウェア要素が必要かの情報を調べ，後者三つは命令高機能化のための情報を調べる．

合成系は，解析系で得られた諸情報を基に命令セットを合成する．この際，ユーザは解析結果をもとに命令セット，記憶要素（レジスタ，RAM 等）を変更することができる．そして，合成された命令セットを基に機能ブロックを合成し，プロセッサ情報（ハードウェア要素，マイクロ命令のフォーマット等）を出力する．

コード生成系は，合成系で得られた命令セットを基にコード列を生成し，マイクロプログラムを出力する．

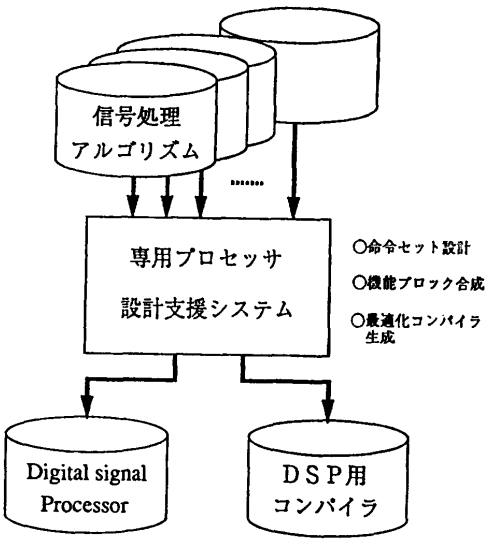


図 2 複数アルゴリズムを実行するプロセッサ設計
Fig. 2 Processor design executing multiple algorithm.

4. 専用プロセッサの設計アプローチ

ここでは，設計支援システムを用いて専用プロセッサを設計する際の，設計アプローチについて述べる．

4.1 汎用性を持ったプロセッサ設計

専用プロセッサ設計を考えた場合，対象をできるだけ狭い範囲に限定するほうが，より高性能なプロセッサ設計が可能である．しかしながら，個々のアルゴリズムごとに，数多くの LSI を設計するよりも，速度性能を多少犠牲にしても，複数のアルゴリズムを実行可能なある程度の汎用性を持ったプロセッサ設計が望ましい場合もある．この場合，あまり幅広いアルゴリズムを対象とすると，無駄な要素が多く生成され，そのための性能低下は避けられないので，専用プロセッサを設計する意味が薄れてきてしまう．しかし，アルゴリズムの特徴が似ているものに限定すれば，本システムのようなプロセッサ設計では，布線論理を用いた LSI 設計と違って多くのハードウェア要素を共有できるため，それほど無駄な要素が生成されることはなく，性能低下も少ないと考えられる．

特徴が似たアルゴリズム群の代表例としては，音声，画像処理等で用いられるディジタル信号処理アルゴリズムが考えられる．本システムを用いて，複数のアルゴリズムを実行する信号処理プロセッサを設計する際の処理概要を**図 2** に示す．システムは入力された信号処理アルゴリズムそれぞれに対して，解析を行い，命令セット，記憶要素の最適設計を行う．そしてそれらの OR 要素を取って全体を統合し，合成したプロセッサを RT 情報として出力する．また，同時にそのプロセッサ向けの最適化コンパイラを生成する．ここで設計されるプロセッサには，積和演算などの，汎用 DSP にみられる信号処理アルゴリズムの特徴が，共通のものとして生ずるだけでなく，やはりアルゴリズムを限定することで，通常の汎用 DSP よりはかなり簡潔なものとなってくる．

4.2 専用性を追求したプロセッサ設計

4.1 節とは逆に，対象を特定アルゴリズムに絞り，できる限り速度性能向上をめざしたプロセッサ設計に対する要求も高い．本システムでは，単一アルゴリズムを入力した場合，このアルゴリズムを実現するのに必要最低限のプロセッサ構成にすることにより，簡潔化を行い，性能向上をはかっている．これにより，専用性を追求したプロセッサ設計が可能であるが，プログラムの規模が大きくなるにつれて，複数のアルゴリズムを考える場合と同様にプロセッサ構成が複雑化し，速度性能が低下してくるのは避けられない．このため，モジュールごとに階層化を行ったプロセッサ設計が有効であると考えられる．これを実現するのに，入力記述である Pascal の手続き，関数といった階層化のための手段をそのまま利用するのが有効である．

手続きで記述された部分は，入出力部を持った新しいプロセッサとして，メインプロセッサの入れ子の形

で実現される．命令セット，機能ブロック設計等は，各手続きごとにまったく独立して行い，それぞれが異なったハードウェア構成を持ったプロセッサとして実現される．また，それぞれの動作も非同期で行われる．

関数で記述された部分は，関数器として布線論理を用いて実現される．この際，関数器は，データフロー解析を行いアルゴリズムの並列性を十分引き出す形で実現される．ただし，SIN，COS 等の標準関数はあらかじめライブラリとして関数器が登録してあるものとしている．

5. 信号処理のアルゴリズムへの適用例

本システムを用いた専用プロセッサ設計と，その評価に用いる入力アルゴリズムとして，次に示す 13 個の代表的な信号処理アルゴリズムを取り上げる．

- ○ PARCOR 格子型フィルタ … PAR
- ○ 自己相関関数 … COR
- ○ デルタ変調方式の符号化 … CDM
- ○ デルタ変調方式の復合化 … DDM
- ○ ADPCM 符号化 … CAD
- ○ ADPCM 復合化 … DAD
- ○ CEPSTRUM 算出 … CEP
- ○ Durbin-Levinson-板倉法 … DUR
- ○ FFT（高速フーリェ変換） … FFT
- ○ Pitch 抽出 … PIT
- ○ Biquad フィルタ … BIQ
- ○ FIR フィルタ（1次） … FIR
- ○ IIR フィルタ（3次） … IIR

これらのアルゴリズムを用いて，種々のプロセッサの設計，評価を行う．

5.1 PARCOR 格子型フィルタ

PARCOR 格子型フィルタのアルゴリズム記述の例を**図 3** に，頻度解析結果を**図 4** に，命令の組合せ解析結果を**図 5** に，合成された命令セットフォーマットを**図 6** に，コード生成結果を**図 7** に示す．

アルゴリズム記述は，ビット型による変数宣言，入出力変数宣言（input, output）以外は，ほぼ Pascal の言語仕様に基づいて記述できる．ビット型は，ユーザがデータの精度をより厳密に記述できる手段を与えるものである．入出力変数は，外部とのデータのやり取りを明確にするためのもので，ハードウェア化された場合，入出力端子として実現される．アルゴリズム記述中で，入力変数からデータを受け取ったとき，外部からデータを読み込む．出力変数にデータを書き込んだとき，外部へデータを出力する．転送方法は，send，ack 信号を介して非同期で行われることを仮定している．割り込み処理などを加えることは容易であるが，言語仕様の拡張を要し現在その機能はない．

```
program parcorfilter (s_in, rk_in, e_out);
const   m1 = 13;
        n  = 128;
type    integer = signed bit 16;
input   s_in   : signed bit 12;
        rk_in  : signed bit 8;
output  e_out  : signed bit 12;
var     ft, gto : array [m1] of integer;
        i, j    : integer;
        s  : array [n] of signed bit 12;
        rk : array [m1] of signed bit 8;
        e  : array [n] of signed bit 12;
begin
  { data input }
  for i := 0 to n-1 do
    s[i] := s_in;
  for i := 0 to m1-1 do
    rk[i] := rk_in;

  { Filter main program }
  for i := 0 to m1-1 do
    begin
      gto[i] := 0;
      ft[i]  := 0
    end;
  e[0] := s[0];
  for j := 1 to n-1 do
    begin
      ft[0] := s[j];
      gto[0] := s[j-1];
      for i := 1 to m1-1 do
        begin
          ft[i] := ft[i-1] - rk[i-1] * gto[i-1];
          gto[i] := gto[i-1] - rk[i-1] * ft[i-1]
      end;
      e[j] := ft[m1-1]
    end;

  { data output }
  for i := 0 to n-1 do
    e_out := e[i]
end.
```

図 3 アルゴリズム記述の例（PARCOR 格子型フィルタ）
Fig. 3 Algorithm description (PARCOR lattice filter).

また，信号処理の分野で本格的にこのシステムを用いていく場合は，丸め込みなどの演算結果の処理が記述できることが望まれる場合が多いが，本システムにはまだその機能はない．

頻度解析は，シミュレータにサンプルデータを与え動的解析を行った結果，各基本ブロックが何回参照されたかをそのフローグラフと共に示したものである．全 19 ブロック中，1回しか実行されないブロックもあれば，ループが多重になり 1,500 回以上も実行されているブロック（B0013, B0014）もあることを示している．

命令の組合せ解析は，頻度解析の結果，高頻度に実

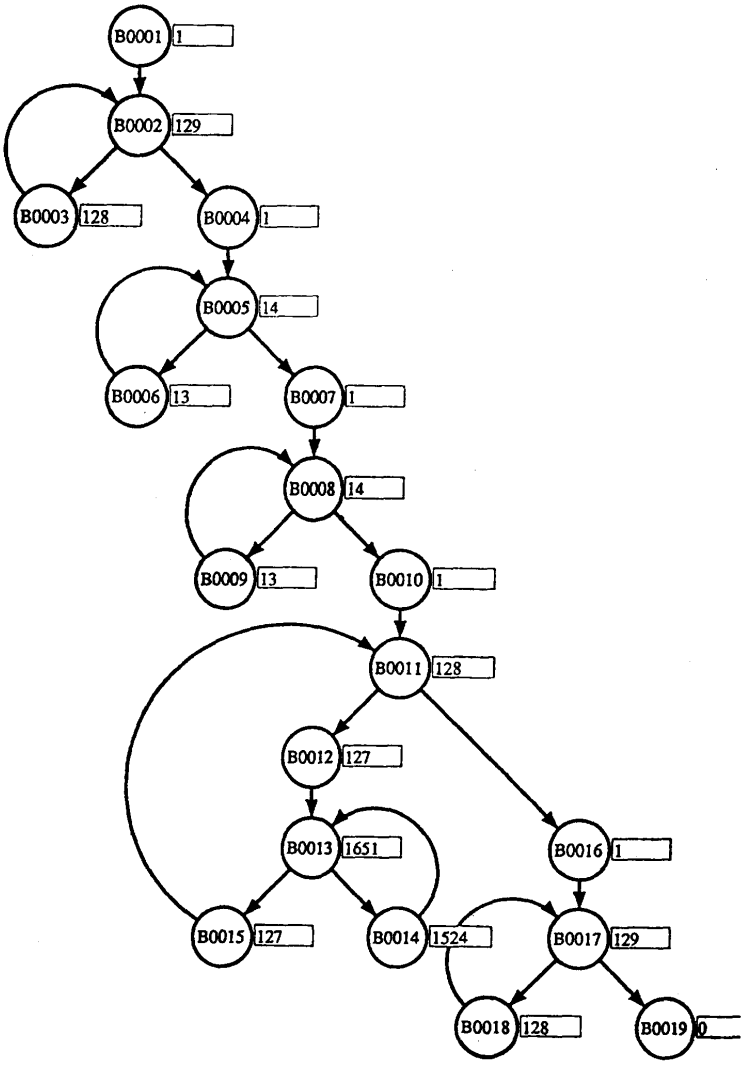


図 4 頻度解析の例 (PARCOR 格子型フィルタ)
Fig. 4 Frequency analysis (PARCOR lattice filter).

行されている B0014 に対して，命令の複合化を行ったものである．文 32 と 33，36 と 38 が同じ積差演算の形をしており，それらの高機能化を行った命令として OP 1 を合成している．また，文 39，40 はインクリメントと無条件分岐と互いに無関係な命令の組合せであり，その並列化を行った命令として OP 2 を合成している．

ライフタイム解析により変数割付に必要なレジスタ数を決めるが，実際の変数とレジスタの割付もそのライフタイム解析の結果によって行う．複合命令に対してもレジスタ割付は同様である．ただし，複合命令によってはレジスタに2つ同時にアクセスするという場合が生じるので，これに関しては工夫が必要である．

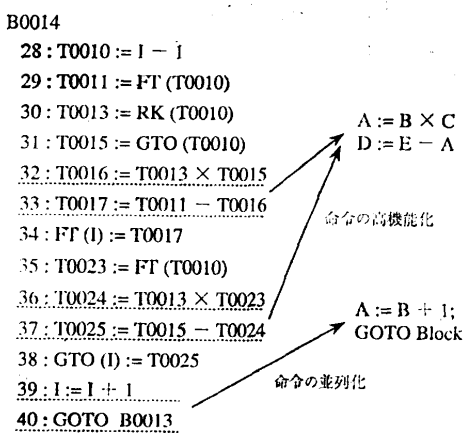


図 5 命令の組合せ解析
Fig. 5 Combinatory analysis of operation.

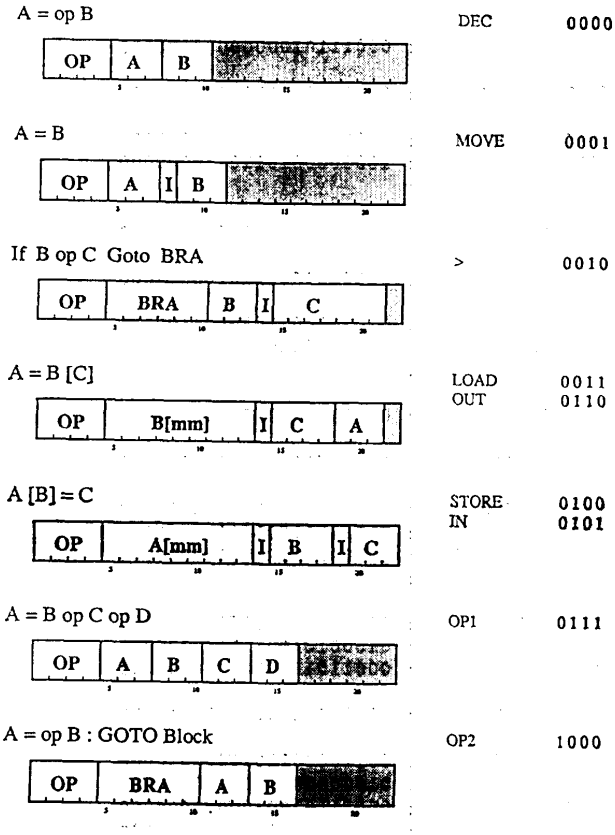


図 6 合成された命令セット (PARCOR 格子型フィルタ)
Fig. 6 Synthesized instruction set (PARCOR lattice filter).

命令セット合成は，解析系で得られた情報をもとに，演算命令，分岐命令，メモリアクセス命令，複合命令ごとに規則性を保った上で，命令長ができるだけ短くなるように決定する．図の左側は，命令の型ごとの命令フォーマット，図の右側は，生成された命令とその命令に割り付けられたビット列を示している．

コード生成は，自動作成されたコード生成部を用いて行われる．図の左側は中間表現（基本ブロックによって構成された3番地文の列），中央は生成されたコードに対するニーモニック，右側はビット列を示している．この例では，複合命令の導入により，全 49 個の3番地文に対して，生成されたコード数は 41 である．

5.2 複数アルゴリズムを実行するプロセッサ

5.1 節に示した，PARCOR 格子型フィルタ同様に，残りの 12 個の信号処理アルゴリズムに対しても，プロセッサ設計を行う．また，同時に 13 個のアルゴリズムを入力し，複数のアルゴリズムを実行するプロセッサ設計を行い，それぞれを比較する．

ストレージは，マイクロインストラクション格納用のプログラム ROM，配列データ格納用の内部データ RAM と，テンポラリデータ格納用のレジスタであるが，各アルゴリズムに対するレジスタ数の関係を**図 8** に，データ RAM の関係を**図 9** に，プログラム ROM の関係を**図 10** に，命令数の関係を**図 11** に示す．

レジスタ数は，入力アルゴリズムが，その変数格納にどれだけのレジスタが必要かの度合を示す．アルゴリズムによって，必要なレジスタは 2 から 32 までグループ分けされる．全体の数は，最も多い 32 (FFT) となる．

RAM 数は，やはりアルゴリズムによってグループ分けされる．全体量は，最も多くの配列データを扱うアルゴリズム PIT (4096 W) によって決められる．

ROM 数は，入力アルゴリズムを実行するプログラムが，どれくらいの長さになるかを示す．やはりアルゴリズムによるグループ分けが可能である．全体は，アルゴリズムの長い CAD, FFT, PIT (128 W) によって決められる．

命令数は，入力アルゴリズムがどれくらいの複雑さを持つかの度合を示す．全体は，すべての命令の OR 要素を取って決められる．

ここで取り上げたアルゴリズムの中でも，どれくらいのレジスタ，RAM，ROM，命令が必要となるかは，アルゴリズムによって異なるが，ある程度グループ分けができる．したがって，複数のアルゴリズムを実行するプロセッサを設計する場合，より高性能を望むなら，アルゴリズムの特徴によって，さらにグループ分けを行った設計が有効である．

5.3 評　価

本システムを用いて設計された，複数の信号処理アルゴリズムを実行するプロセッサと信号処理プロセッサ TMS 320 C 25 (以下 C 25) の比較検討を行う．**表 1** にハードウェアとソフトウェアの両面からの比較を示す．

ハードウェアにおいて，命令数は C 25 と比較して少ないが，これは本システムの場合，汎用性を持たせたといっても，完全な汎用 DSP でないためである．本システムでは，設計途中で命令の追加等のユーザの介入を許しており，これによりさらに汎用性を追求したプロセッサ設計も可能である．

データ型は，両者とも符号付きの固定小数点を用いている．最近の DSP は，浮動小数点を取り扱えるも

```
B0001 :
  1 : I := 0                    : MOVE WR6 0           : 00011101000000000000000
B0002 :
  2 : IF I > 127 GOTO B0004     : > WR6 127 B0004      : 00100001011101111111110
B0003 :
  3 : S (I) := S_IN             : IN S WR6 IPORT       : 01010000111010001100111
  4 : I := I + 1                : OP2 B0002 WR6 WR6    : 10000000101101100000000
  5 : GOTO B0002                :                      :
B0004 :
  6 : I := 0                    : MOVE WR6 0           : 00011101000000000000000
B0005 :
  7 : IF I > 12 GOTO B0007      : > WR6 12 B0007       : 00100010011101000110000
B0006 :
  8 : RK (I) := RK_IN           : IN RK WR6 IPORT      : 01010100111010001100111
  9 : I := I + 1                : OP2 B0005 WR6 WR6    : 10000001101101100000000
  10 : GOTO B0005               :                      :
B0007 :
  11 : I := 0                   : MOVE WR6 0           : 00011101000000000000000
B0008 :
  12 : IF I > 12 GOTO B0010     : > WR6 12 B0010       : 00100011101101000110000
B0009 :
  13 : GTO (I) := 0             : STORE GTO WR6 0      : 01000000011010011010000
  14 : FT (I) := 0              : STORE FT WR6 0       : 01000000000000011010000
  15 : I := I + 1               : OP2 B0008 WR6 WR6    : 10000010101101100000000
  16 : GOTO B0008               :                      :
B0010 :
  17 : T0004 := S (0)           : LOAD WR0 S 0         : 00110000111010100000000
  18 : E (0) := T0004           : STORE E 0 WR0        : 01000101001111000000000
  19 : J := 1                   : MOVE WR5 1           : 00011011001000000000000
B0011 :
  20 : IF J > 127 GOTO B0016    : > WR5 127 B0016      : 00101001101011111111110
B0012 :
  21 : T0006 := S (J)           : LOAD WR0 S WR5       : 00110000111010001010000
  22 : FT (0) := T0006          : STORE FT 0 WR0       : 01000000000001000000000
  23 : I := 1                   : MOVE WR6 1           : 00011101001000000000000
  24 : T0007 := J - 1           : DEC WR0 WR5          : 00000001010000000000000
  25 : T0008 := S (T0007)       : LOAD WR0 S WR0       : 00110000111010000000000
  26 : GTO (0) := T0008         : STORE GTO 0 WR0      : 01000000011011000000000
B0013 :
  27 : IF I > 12 GOTO B0015     : > WR6 12 B0015       : 00101000101101000110000
B0014 :
  28 : T0010 := I - 1           : DEC WR0 WR6          : 00000001100000000000000
  29 : T0011 := FT (T0010)      : LOAD WR1 FT WR0      : 00110000000000000000010
  30 : T0013 := RK (T0010)      : LOAD WR2 RK WR0      : 00110100111010000000100
  31 : T0015 := GTO (T0010)     : LOAD WR3 GTO WR0     : 00110000011010000000110
  32 : T0016 := T0013 * T0015 : OP1 WR2 WR3 WR1 WR1 : 01110100110010010000000
  33 : T0017 := T0011 - T0016 :                        :
  34 : FT (I) := T0017          : STORE FT WR6 WR1     : 01000000000000011000001
  35 : T0023 := FT (T0010)      : LOAD WR0 FT WR0      : 00110000000000000000000
  36 : T0024 := T0013 * T0023 : OP1 WR2 WR0 WR0 WR3 : 01110100000000011000000
  37 : T0025 := T0015 - T0024 :                        :
  38 : GTO (I) := T0025         : STORE GTO WR6 WR0    : 01000000011010011000000
  39 : I := I + 1               : OP2 B0013 WR6 WR6    : 10000110001101100000000
  40 : GOTO B0013               :                      :
B0015 :
  41 : T0027 := FT (12)         : LOAD WR0 FT 12       : 00110000000001110000000
  42 : E (J) := T0027           : STORE E WR5 WR0      : 01000101001110010100000
  43 : J := J + 1               : OP2 B0011 WR5 WR5    : 10000100011011010000000
  44 : GOTO B0011               :                      :
B0016 :
  45 : I := 0                   : MOVE WR6 0           : 00011101000000000000000
B0017 :
  46 : IF I > 127 GOTO B0019    : > WR6 127 B0019      : 00101010001101111111110
B0018 :
  47 : E_OUT := E (I)           : OUT OPORT E WR6      : 01100101001110011011110
  48 : I := I + 1               : OP2 B0017 WR6 WR6    : 10001001111101100000000
  49 : GOTO B0017               :                      :
B0019 :
```

図 7 コード生成
Fig. 7 Code generation.

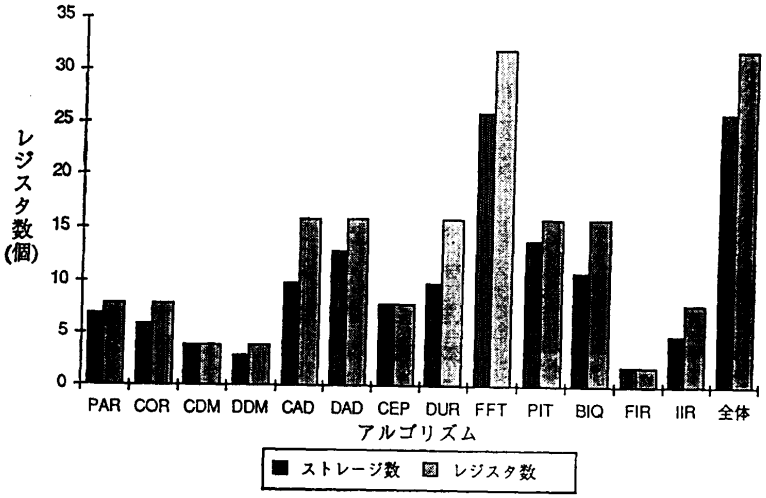


図 8 各アルゴリズムに対するレジスタ数
Fig. 8 The number of registers in each algorithm.

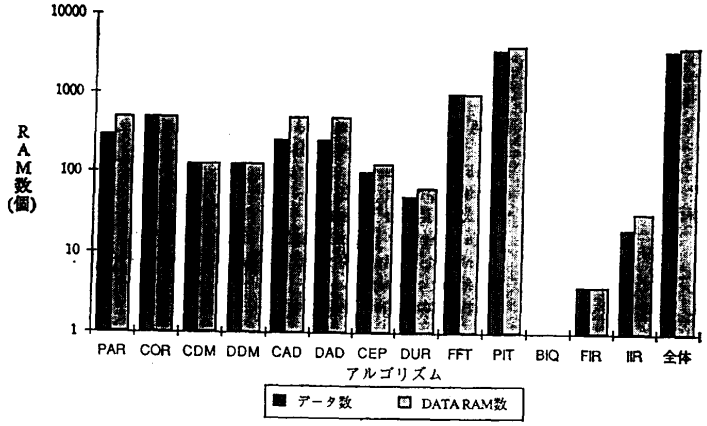


図 9 各アルゴリズムに対する RAM 数
Fig. 9 The number of RAM in each algorithm.

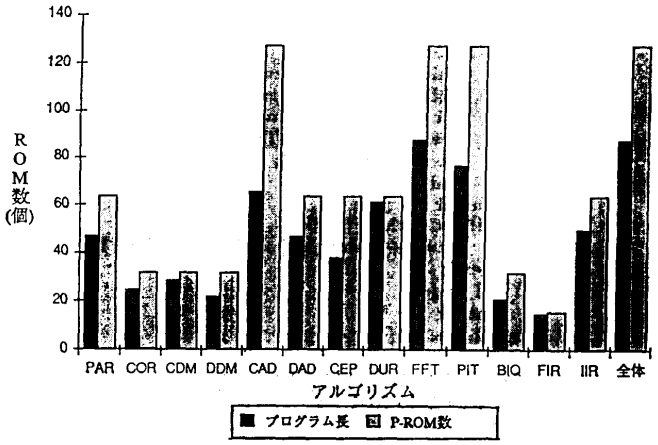


図 10 各アルゴリズムに対する ROM 数
Fig. 10 The number of ROM in each algorithm.

のが増えてきているが，これについては検討中である．

特殊命令は，両者とも信号処理アルゴリズム特有な積和，積差といった演算を行うものを備えている．

内部データ RAM は C25 と比較して多いが，これはピッチ抽出など多くのデータ領域を必要とするアルゴリズムが含まれるためである．このアルゴリズムを実行する場合，C25 では，データ格納のために内部 RAM と外部 RAM を使い分けなければならず，効率が悪い．

プログラム ROM は C25 と比較して少ないが，これは入力アルゴリズムの規模がそれほど大きくないのと，コンパイラが効率のよいコード生成を行うためである．C25 では，ROM 領域の大部分は使用されず，無駄なハードウェア要素となる．

レジスタは多いが，これは本システムが RISC のようにレジスタ演算を基本としているためである．C25 はレジスタが少ないため，メモリ演算が多くなり効率が悪い．

ソフトウェア開発面は，本システムでは Pascal，C25 ではアセンブリ言語を用いて行う．C25はC言語によるソフトウェア開発も可能であるが，このコンパイラでは，現状ではプロセッサの性能を十分に引き出せない．

表1のコード生成欄のステップ数は，同じアルゴリズムに対し，コンパイラを用いてコード生成を行い，コード数を比較したものである．図3の PARCOR 格子型フィルタを等価なC言語に書き直し，C25 用のCコンパイラを用いてコード生成を行う．最も高頻度に参照される B 0014 のコード数を比較した結果，10 ステップと 111 ステップになった．このブロックは 1,500 回以上も参照さ

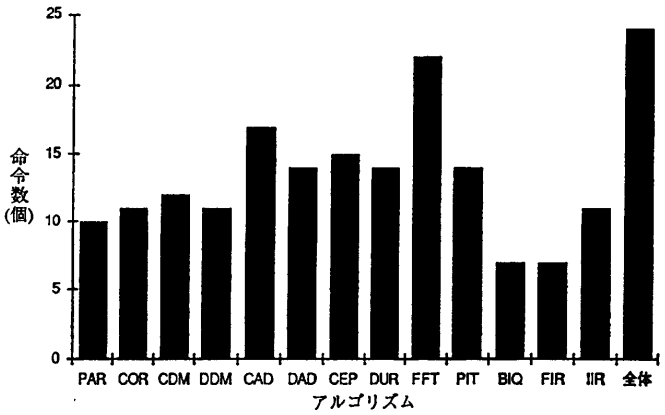


図 11 各アルゴリズムに対する命令数
Fig. 11 The number of instruction in each algorithm.

表 1 TMS 320 C 25 との比較
Table 1 Comparison with TMS 320 C 25.

		本システム	TMS 320 C 25
ハードウェア	命令数	24 個	133 個
	データ型	32 Bit 固定小数点	16 Bit 固定小数点
	特殊命令	積和，積差命令	積和，積差命令
	内部データ RAM	4096W	544W
	プログラム ROM	128W	4096W
	レジスタ	32W	8W
ソフトウェア	開発環境	Pascal	アセンブリ言語 (C言語)
	コード生成 PARCOR (B14)	10 ステップ	111 ステップ

れるので，プログラム実行時間に大きな差が出ると思われる．C 25 の場合，アセンブリ言語を使用すれば，かなりこのステップの短縮が期待できることから，C コンパイラによるソフトウェア開発は必ずしも十分でないと考えられる．

一般に，信号処理プロセッサのソフトウェア開発は低次元のアセンブリ言語を用いて行わねばならず，効率が悪い[10]．本システムでは，設計された LSI の利用に際しても高級言語によるソフトウェア開発を前提としており，命令セット等もコンパイラが効率のよいコードを生成できるように決められる．このように，ソフトウェア開発環境は本システムのほうが優れている．

6. クイックソート・アルゴリズムへの適用

信号処理とは異なる特徴を持つアルゴリズムの例としてクイックソートの例を示す．本システムでは，同じアルゴリズムであっても，その記述の仕方によって設計結果が異なり，合成品質にも差が生じることが避けられない．ただし，本システムでは中間情報生成系における冗長部の削除などの最適化や，合成部において必要最低限になるように資源の決定がなされるので，相当にアルゴリズム記述による差を吸収できる．ただし，配列宣言にメモリを割り当てるので，メモリ容量を少なくするためには，配列を少なくする工夫が必要となる．このように本システム使用上の留意点はある．ここでは紙面の都合上あまり豊富な例題を示すことはできないが，クイックソートのアルゴリズムについて**図 12** に示すような 3 種類の異なる表現記述，QS 1，QS 2，QS 3 を与えた結果を示す．主要な設計結果は**表 2** に示すとおりである．QS 1，QS 2 は，比較相手（プログラム中の変数 x）の選択を変えたもので，アルゴリズムが若

```
program Quick_Sort1 (iport,oport);
const N = 100;
      M = 100;
type integer = unsigned bit 16;
     real = unsigned bit 32;
input iport : real;
output oport : real;
var i, j, k, l : integer;
    e : array [N] of real; { 64Kw * 32bit }
    stack_h, stack_l : array [M] of integer; { 4Kw * 16bit }
    x, w : real;
    p : unsigned bit 12;

begin
{ input data from port(iport) to RAM(e) }
  for i := 0 to N-1 do
    e[i] := iport;

{ body of quick sort }
  p := 0; stack_h[0] := 0; stack_l[0] := N-1;
  repeat
    k := stack_h[p]; l := stack_l[p]; p := p-1;
    repeat
      i := k; j := l; x := e[i];
      repeat
        while e[i] < x do i := i+1;
        while x < e[j] do j := j-1;
        if i <= j then begin
          w := e[i]; e[i] := e[j]; e[j] := w;
          i := i+1; j := j-1;
        end
      until i > j;
      if i < l then begin
        p := p+1; stack_h[p] := i; stack_l[p] := l
      end;
      l := j
    until k >= l
  until p = -1;

{ output data from RAM(e) to port(oport) }
  for i := 0 to N-1 do
    oport := e[i]
end.
```

(a)

干異なるものである．QS3は，QS1と同じアルゴリズムを制御構造を変えて記述したものである．アルゴリズムの実行に必要なレジスタ数，RAM数は，ほとんど同じ結果が得られている．ROM数，命令数においては，プログラムの書き方の違いはあまり合成品質に影響しないが，QS2のほうがQS1，QS3よりもプログラムが長くなり，アルゴリズムの違いはプログラム長にかなり影響することを示している．表2では，設計結果の微妙な差を示すためにレジスタとROMの数は必要最小数を示したが，実際上は2のべき乗個に丸められると差が見られなくなったり，1ビット分の違いとなる場合もある．総ステップ数は，実際にデータを与えて，シミュレーションを行った場合の実行されるステップの総数を示しているが，QS1よりQS2のほうがステップ数が多いのは，選択相手の選び方のアルゴリズムと与えたデータに依存したものである．同じアルゴリズムに対してQS3のほうがステップ数

表2 クイックソートの異なる仕様記述による設計結果の比較
Table 2 Comparison of synthesized results given by three different algorithm descriptions that execute quick sort.

	QS1	QS2	QS3
レジスタ数	7W	8W	7W
RAM数	512W	512W	512W
ROM数	53W	64W	56W
命令数	15個	18個	15個
総ステップ数	7772ステップ	8541ステップ	8240ステップ

```
program Quick_Sort2 (iport,oport);
const N = 100;
      M = 100;
type integer = unsigned bit 16;
     real = unsigned bit 32;
input iport : real;
output oport : real;
var e : array [N] of real; { 64Kw * 32bit }
    ls, rs : array [M] of integer; { 4Kw * 16bit }
    i, j, l, r, p, x : integer;
    t : real;

begin
{ input data from port(iport) to RAM(e) }
  for i := 1 to N do
    e[i] := iport;

{ body of quick sort }
  p := 1; ls[1] := 1; rs[1] := N;
  repeat
    l := ls[p]; r := rs[p]; p := p-1;
    repeat
      x := e[(l+r) div 2];
      i := l-1; j := r+1;
      repeat
        repeat i := i+1 until x <= e[i];
        repeat j := j-1 until e[j] <= x;
        t := e[i]; e[i] := e[j]; e[j] := t;
      until i >= j;
      e[j] := e[i]; e[i] := t;
      if i - l > r - j then begin
        if l < i then begin
          p := p+1; ls[p] := l; rs[p] := i-1
        end;
        l := j+1
      end else begin
        if j < r then begin
          p := p+1; ls[p] := j+1; rs[p] := r
        end;
        r := i - 1
      end
    until l >= r;
  until p = 0;

{ output data from RAM(e) to port(oport) }
  for i := 1 to N do
    oport := e[i]
end.
```

(b)

```
program Quick_Sort3 (iport,oport);
const N = 100;
      M = 100;
type integer = unsigned bit 16;
     real = unsigned bit 32;
input iport : real;
output oport : real;
var e : array [N] of real; { 64Kw * 32bit }
    stack_h, stack_l : array [M] of integer; { 4Kw * 16bit }
    i, j, k, l : integer;
    x, w : real;
    p : unsigned bit 12;

begin
{ input data from port(iport) to RAM(e) }
  for i := 0 to N-1 do
    e[i] := iport;

{ body of quick sort }
  p := 0; stack_h[0] := 0; stack_l[0] := N-1;
  while p <> -1 do begin
    k := stack_h[p]; l := stack_l[p]; p := p-1;
    while k < l do begin
      i := k; j := l; x := e[i];
      while i <= j do begin
        while e[i] < x do i := i+1;
        while x < e[j] do j := j-1;
        if i <= j then begin
          w := e[i]; e[i] := e[j]; e[j] := w;
          i := i+1; j := j-1;
        end;
      end;
      if i < l then begin
        p := p+1; stack_h[p] := i; stack_l[p] := l;
      end;
      l := j
    end;
  end;

{ output data from RAM(e) to port(oport) }
  for i := 0 to N-1 do
    oport := e[i];
end.
```

(c)

図12 クイックソートのアルゴリズムの異なる記述
Fig. 12 Different descriptions of quick sort algorithm.

が多いのは，主に中間情報生成時に分割されるブロックの数が多くなり，分岐命令の頻度が増したことによる．

7. おわりに

高級言語で書かれたアルゴリズム記述を入力とする専用プロセッサ設計支援システムの，基本となる三つの主眼点，処理概要，設計アプローチについて述べた．このシステムに複数の代表的なディジタル信号処理アルゴリズムおよびクイックソートのアルゴリズムを与えて，種々の専用プロセッサ設計を試みた結果，次の三つの結論を得た．

（1） 高級言語によるアルゴリズム記述を入力とすることにより，通常のソフトウェアの開発を行うのと同じ感覚で，専用プロセッサ設計が可能である．

（2） 設計されたプロセッサの特徴を生かした最適化コンパイラを自動合成することにより，優れたソフトウェア開発環境を持ったプロセッサ設計が可能である．

（3） 仕様記述の違いにより，設計されるプロセッサに差が生ずるが，その差は設計過程でかなり吸収され，記述を極端に変形しない限り，設計結果はほぼ同等のものが得られる．

LSI CAD は論理設計，機能設計とボトムアップ的に発展してきており，それぞれの設計ツールも実用化されつつある．しかし，その上位のシステム設計，方式設計といったレベルになると，まだ明確な方向付けはなく，現在様々なシステムが試行錯誤されている状態である．こういった中で，本システムのように，設計された LSI に対するソフトウェア環境も含めて考える立場に立った設計ツールは，今後の LSI CAD の流れの中で，一つの新しいパスになるのではないかと思う．

最後に今後の課題を述べる．今後は，本システムで得られたプロセッサ情報を，既存の RT レベルのハードウェア記述言語に変換し，それから LSI のマスクパタンまでの設計を行うハードウェア合成システムと共にトータルな LSI CAD の実現を試みたい．その合成システムから，ゲート数，命令実行時間等の情報をフィードバックされることにより，本システムの LSI 設計の質をさらに向上させることができる．

謝辞 本研究を進めにあたり，貴重な討論と協力をいただいた，竹沢寿幸氏，上野聡氏，北畠宏信氏ほか，白井研 CAD グループの皆様に感謝します．

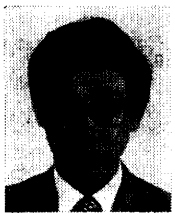
参考文献

1) 持田侑宏：DSP の現状と動向，情報処理，Vol. 30, No. 11, pp. 1291-1299 (1989).
2) 竹沢寿幸，上野 聡，白井克彦：特定用途向け回路アーキテクチャ設計支援システム，電子情報通信学会論文誌D分冊，Vol. J 71-D, No. 10, pp. 1931-1938 (1988).
3) 池永 剛，竹沢寿幸，白井克彦：高位の仕様記述に基づく専用プロセッサ設計支援システム，第39回情報処理学会全国大会論文集，4 X-7 (1989).
4) 池永 剛，白井克彦：複数のアルゴリズムを実行する専用プロセッサのアーキテクチャ設計，情報処理学会設計自動化研究会，51-9 (1990).
5) 高木 茂：VLSI 設計 CAD の最近の動向，情報処理，Vol. 28, No. 5, pp. 581-589 (1987).
6) 宇都宮公訓：命令セットアーキテクチャの評価，情報処理，Vol. 29, No. 12, pp. 1474-1481 (1988).
7) Colwel, R. P. et al.: Computers, Complexity, and Controversy, *Computer*, Vol. 18, No. 9, pp. 8-19 (1985).
8) Wulf, W. A.: Compilers and Computer Architecture, *Computer*, Vol. 14, No. 7, pp. 41-47 (1981).
9) Ganapathi, M. et al.: Retargetable Compiler Code Generation, *Comput. Surv.*, Vol. 14, No. 4, pp. 573-592 (1982).
10) 小野定康：DSP のプログラム開発環境，情報処理，Vol. 30, No. 11, pp. 1307-1314 (1989).

（平成3年3月 7 日受付）
（平成3年9月 12日採録）

池永 剛（正会員）

昭和39年生．昭和63年早稲田大学理工学部電気工学科卒業．平成2年同大学大学院理工学研究科修士課程修了．同年日本電信電話(株)入社．LSI 研究所設計システム研究部勤務．以来，性能指向型大規模 ASIC 構成法の研究，特にそのテスト設計法の研究に従事．



白井 克彦（正会員）

昭和14年生．昭和38年早稲田大学理工学部電気工学科卒業．昭和43年同大学大学院博士課程修了．同年早稲田大学理工学部電気工学科講師．昭和50年同教授．平成3年同情報学科教授．音声情報処理を中心に人間の情報処理を主な研究対象としているが，LSI アーキテクチャ設計にも興味を持っている．