設 計 自 動 化 60－4
（1991．12．12）

高位仕様記述からの
専用プロセッサ設計における機能設計について

北畠 宏信　　白井 克彦
早稲田大学　理工学部

LSIの製造技術の進歩に伴って、様々な専用VLSIプロセッサが実現されている。その反面、システム設計もより複雑化，多様化し、設計期間の長期化とコストが増加し、より容易に設計可能な設計支援システムが必要不可欠となる。本論文では、高位仕様記述を入力として、種々の解析を行い、シミュレーションを通してそのアルゴリズムの特徴的な部分を機能ブロックとして合成して高速化を行い、資源と実行速度を考慮した専用プロセッサ設計の機能設計を中心に述べる。そして、ソート・アルゴリズム群に対して本システムを適用して設計を行い、その評価を行う。その結果、シミュレーションと合成を繰り返しながら機能構成要素を設計していく機能設計が有効であることが示された。

Function Design in Special Purpose Processor Design from High Level Specification Description

Hironobu Kitabatake and Katsuhiko Shirai
Department of Electrical Engineering, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo 169, Japan

According to the advance of the LSI production technology, various kinds of special purpose LSIs have been produced. However, the complexity and the variety of such integrated circuit systems are increasing and the period and the cost for the design are growing. Then, effective support systems to design more easily have been eagerly desired. This paper discusses the functional design method which receives the specification described by high level language and makes an optimal synthesis using the method to find an optimal combination of functional blocks by the iterative simulation. The method is applied to several kinds of sorting algorithms. It is verified through the iterative process of the simulation and of the combination the functional blocks.

1 はじめに

LSIの製造技術の進歩に伴い、様々なシステムのLSI化が可能となり、特定のアルゴリズムをより高速に実行する専用VLSIプロセッサ[1]の応用分野も広がっている。その反面、規模の増大に伴い、システム設計もより複雑化，多様化し、設計期間の長期化とコストの増加が指摘されている。今後、LSIの多様な分野への応用が進む中で、専用LSIプロセッサへの要求は増加していくと考えられるが、そのためには、より容易により短期間にLSIを設計することのできる設計支援システムが必要不可欠となる。このような要求に応える設計支援システムの1つの形態として、ユーザ自身が容易に設計を行えるソフトウェアの概念に近い高位仕様記述を入力としたシステムが考えられる。

このような背景から、我々は、幅広いユーザを対象とした高位仕様記述を入力とする専用プロセッサ設計支援システム[2][3]の提案と研究を行っている。このシステムは、ソフトウェアの開発に用いられている高級言語をアルゴリズム仕様記述として用い、そのアルゴリズムに対して種々の解析[4]を行い、高速で効率の良い専用プロセッサの設計を支援することを目的としている。

本論文では、アルゴリズムのシミュレーションを通してそのアルゴリズムの特徴的な部分を機能ブロックとして合成して高速化を行い、資源と実行速度を考慮した機能設計を中心に本システムの設計指針と処理概要を述べる。そして、適用例として代表的な信号処理アルゴリズムであるPARCOR格子型フィルタを取り上げてシステムの機能設計方法とその効果を論ずる。さらに、ソート・アルゴリズム群に対して本システムを適用して設計を行い、アルゴリズムの相違や表記の違いに対する依存性とシステムの有効性、および問題点を論ずる。

2 本システムの専用プロセッサの設計指針

専用プロセッサの機能設計においては、特定のアルゴリズムをどのような機能構成要素（命令の種類、レジスタ、RAMなど）を用いて、それをどのように有効に利用するかが重要である。特に、専用プロセッサの機能設計においては、与えられたアルゴリズムを実行するために、機能構成要素を必要最小限に抑えて、効率よく実行しなければならない。この場合に、ハードウェアの実行速度に対する要求を満たすと同時に、ハードウェア資源を必要最小限に抑え、安価で製造が容易なものとする事が最終目的となる。

これらのことをふまえ、本システムでは、次のような3つの設計指針から専用プロセッサの設計支援を行う。

2.1 高位仕様記述によるプロセッサ設計

専用プロセッサの設計においては、ハードウェアに関する専門知識が絶対不可欠のものであり、それがユーザを限定してしまっている。しかし、専用プロセッサに対する開発要求が増大するにつれて、ハードウェアの専門知識を持たないユーザであっても、専門家と同等のレベルの設計が可能になることが望まれる。

そのため、本システムでは、一般のソフトウェア開発に広く用いられている高級言語（Pascal）を用いてアルゴリズムの仕様記述のみを入力とし、ハードウェアの詳細な設計はシステムが行っていくものとする。 システムは、与えられたアルゴリズム記述に対して、現在のソフトウェアのコンパイラの種々の手法を用いて、抽象度の高い記述から無駄なコードを削除し、ハードウェア合成に必要な情報を抽出し、機能構成要素を割り付けて、効率の良いハードウェアを設計していくという方針を取る。

2.2 シミュレーションに基づいた設計変更

機能構成要素を選択する際、用意された機能構成要素が必ずしもアルゴリズムの実行に適切であるかどうかはわかりにくい。また、機能構成要素を全て有効に選択し、自動的にユーザの全ての要求を満足するプロセッサを設計するのは、きわめて複雑で困難である。

そのため、本システムでは、シミュレーションを通してプロセッサの実行速度と資源の選択をユーザに示し、機能構成要素の変更とシミュレーションを繰り返して無駄な機能構成要素をできるだけ取り除きながら設計を進めていく。そして、ユーザが介入し、速度向上に有効な機能ブロックを選択して、それを用いてシステムが自動的に割付を行う。さらにシミュレーションと機能構成要素の選択を繰り返しながらユーザの速度に対する要求と資源に対する要求に応じた最適なプロセッサの設計を進めていくという方針を取る。

2.3 実行速度の重視

本システムでは、並列処理、機能ブロック化を行った特殊命令の導入によって速度性能を上げることを考える。本システムでは、現在の上位設計の入力であるRTレベルのハードウェア記述言語による設計がモジュールを中心として設計することを考慮し、ターゲットアーキテクチャとして命令の並列実行が可能なレジスタ演算を基本とするアーキテクチャを考える。

並列処理においては、与えられた命令を用いてできるだけ並列に実行することを考えて、速度の向上をはかる。また、データの依存する部分では、並列実行は不可能なので基本的には速度の向上は望めない。しかし、アルゴリズム特有の構造や、高頻度に実行されている部分のうち、データの依存している命令群を抽出して、レジスタやメモリ転送の無駄の削除や、その部分だけ特別に回路を合成するなどの方法を用いて、一つの機能ブロックとして合成し、高速化をはかることは可能である。

そこで、データの依存性をグラフ構造を用いて解析し、頻度解析の情報から特徴的な命令群を機能ブロックとして合成することで速度の向上をはかっていく。

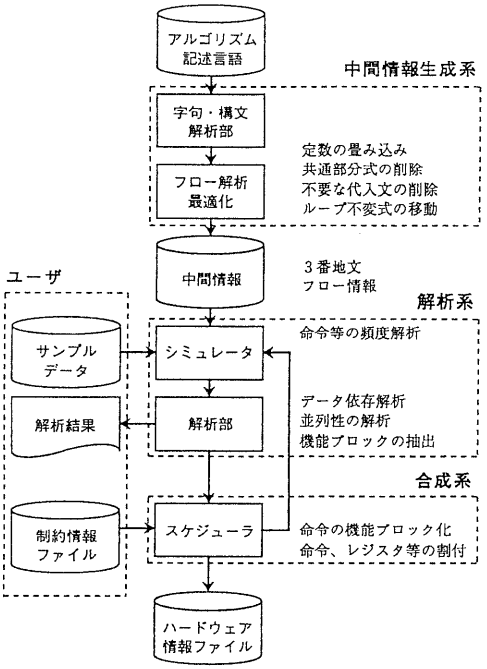

図 1: システム構成図

3 処理概要

本システムは、中間情報生成系、解析系、合成系の3つの系からなる (図 1)。ユーザは、ほぼ Pascal の仕様にしたがって、ハードウェアとして実現したいアルゴリズムの仕様記述を記述し、実行するサンプルデータとともに入力する。

中間情報生成系では、仕様記述に対して字句・構文解析、冗長なコードを削除する種々の解析を行い、中間情報として3番地文によるフロー情報を生成する。

解析系では、サンプルデータに基づいてシミュレーションを行い、ブロック・命令ごとの頻度解析を行い、機能ブロックとして合成できる部分をグラフ構造として取り出し、ユーザに結果を出力する。ユーザは、この解析結果から命令の選択を行う。

合成系では、ユーザより与えられた機能ブロック・命令・レジスタの諸制限の中で割付を行い、システムの出力として、使用した命令・機能ブロック・レジスタの情報とそのフロー情報をハードウェア情報として出力する。

ユーザは、解析系での結果と合成系より得られたコードの再度のシミュレーションにより、速度と資源の制約を満たしたハードウェアを合成していく。

4 適用例

適用例として、代表的な信号処理アルゴリズムである PARCOR 格子型フィルタの設計例を示す。

```
program  parcor_lattice_filter (s_in, rk_in, e_out);
const    m = 15;
         m1 = 14;
         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, gt, fto, gto : array [m] of integer;
         i, j : integer;
         s, e : array [n] of signed bit 12;
         rk : array [m] of signed bit 8;

begin

   { data input }
   for i := 0 to n do
     s[i] := s_in;
   for i := 0 to m do
     rk[i] := rk_in;

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

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

図 2: 仕様記述 (PARCOR 格子型フィルタ)

アルゴリズム仕様記述として図2に示すように記述される [5]。

ハードウェアの仕様をより細かく明確に表現できるように、入出力変数宣言 (input,output)、ビット型による変数宣言を用いている。入出力変数は、ハードウェアとして実現された場合の外部とのデータのやり取りを行う入出力端子に相当する。また、ビット型による変数宣言は、計算値を保持する記憶素子の精度を厳密に決めるためのものである。

仕様記述に冗長なコードの削除・移動を中心にフロー解析・最適化を行った中間情報が出力される (図 3)。

次に、解析系での命令の機能ブロック化を中心とした機能設計について述べる。

図 4は、基本ブロックとして分割したブロックの No.14 の一部分を DAG(Directed Acyclic Graph) に命令の時間情報を持たせたグラフ構造で示している。図では、実行ステップを全て1ステップとした場合のものである。実際には、命令ごとにステップ数を変えることが可能である。

```
(PROGRAM PARCOR_LATTICE_FILTER NIL (S_IN RK_IN E_OUT)
 (INPUT
  (S_IN (SIGNED BIT 12))
  (RK_IN (SIGNED BIT 8)))
 (OUTPUT
  (E_OUT (SIGNED BIT 12)))
 (VAR
  (FT (ARRAY (15) OF (SIGNED BIT 16)))
            省略
 (TEMPORARY
  (T0004 (SIGNED BIT 12))
 ;;
 (PROCESS
  (B0001
   (1 (I := 0)))
            省略
  (B0014
   (32 (T0010 := I - 1))
   (33 (T0011 := FT (T0010)))
   (34 (T0013 := RK (T0010)))
   (35 (T0015 := GTO (T0010)))
   (36 (T0016 := T0013 * T0015))
   (37 (T0017 := T0011 - T0016))
   (38 (FT (I) := T0017))
   (39 (T0023 := FT (T0010)))
   (40 (T0024 := T0013 * T0023))
   (41 (T0025 := T0015 - T0024))
   (42 (GTO (I) := T0025))
   (43 (I := I + 1))
   (44 (GOTO B0013)))
            省略
  (B0022
   (98 (E_OUT := E (I)))))
```

図 3: 中間情報

さて、1つの命令が同時に複数実行可能であれば、同じステップ内に並ぶ命令列は、同時に実行が可能なので、同じコントロールステップに割り付けることができる。

また、ステップ方向の命令の並びは、下位の命令列の実行が終わらなければ実行が不可能なので、基本的には、スピードを上げることができないことになる。

しかし、この部分を新しい機能を持った機能ブロックとして1命令として実現し、速度を短縮することができれば、さらにスピードを上げることが可能である。そこで、このグラフ構造より速度の向上が期待できる命令群を命令のグラフとして抽出する。そして、それぞれの命

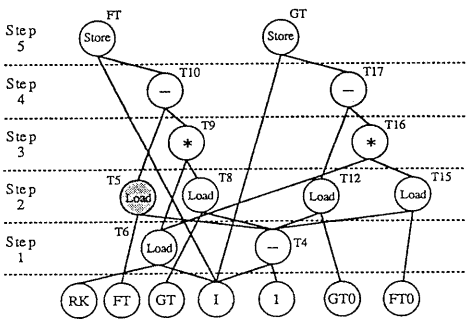

図 4: ブロック14のDAG(一部)

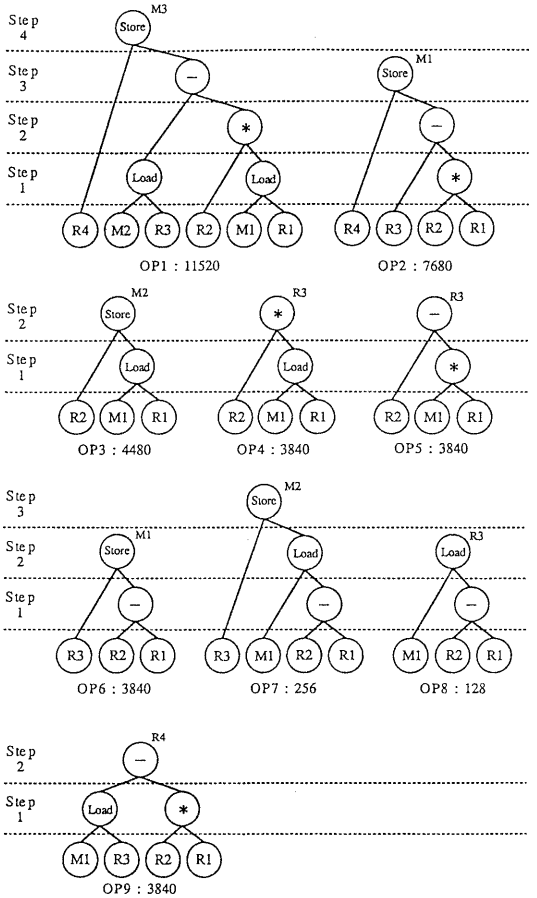

図 5: 抽出されたグラフ

令群に頻度解析の結果を加えてユーザに解析結果を表示する。

実際には、次の条件に会うグラフ構造を中間情報として分割されたブロックごとに探索しながら再帰的に抽出していく。ある Node を中心として、

(1) 自分の左の Node と右の Node の実行されるステップが異なる場合、その間に含まれる Node を全て取り出す。

(2) 自分の左の Node と右の Node の実行されるステップが同じ場合には、それらをまとめて取り出す。

(3) 抽出された命令群の外から途中のデータの参照が行われる場合や、他のブロックから参照される場合は抽出しない。

ただし、抽出される命令群は、下位の設計支援システムでの設計が行い易いように、入力が複数で、出力が1つの構造のみに限り、複数の命令を1つの機能ブロック

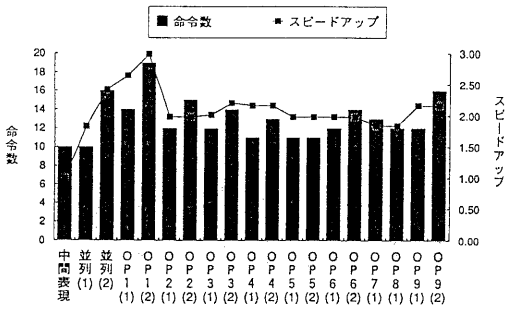

図 6: 命令数とスピードアップとの関係

として合成することを考え、その中身は、実際には、どのように実現されるかは限定しない。そのため、次のような場合には、抽出を行わない。

(1) の例として、図 4の Step5 における Store の Node を中心に考えると図 5の OP1、OP2、OP6 が抽出される。

DAG の構成時点では、as soon as possible の考えに従い、資源の量を考慮せずに、図 4の、Load 命令 (斜線部) のように Step2、Step3 のどちらのステップでも実行が許される場合には、Step2 に一時的に割り付けられる。(2) では、この点を考慮し、結果として Step4 の減算命令では、(1)、(2) の両方が適用されることになり、図 5からは、OP5、OP9 が抽出される。

結果として、上記の条件にしたがって PACOR 格子型フィルタからは、図 5の 9 通りのグラフ構造が抽出されることになる。すなわち、9 種類の機能ブロック化が可能であることを示している。それぞれのグラフの下の数値は、速度の短縮が見込まれるステップ数と頻度を掛け合わせたものをすべてのブロックから抽出し、合計した値である。

シミュレーションによってそれぞれの機能ブロックの導入時の命令数とスピードアップの関係を図 6に示す。

各命令の個数と機能ブロックとして合成された命令に含まれる基本の命令数の総計を命令の資源と考えて命令数とする。スピードアップは、逐次実行を基本とした場合にどれくらいの速度が向上したかを示している。

中間表現は、中間情報において 1 ステップ 1 命令で逐次実行を行ったもの、並列 (1) は、各命令数を 1 つに限定した上で並列実行を可能としたもの、並列 (2) は各命令を複数使用して並列実行を可能としたものである。また、その他のものは、図 5の機能ブロックに対応し、括弧内の数字は、合成された機能ブロックの個数であり、基本とする命令は 1 つに限定したものである。

基本的には、命令群を機能ブロックとして合成すれば、それに応じて実行速度はあがるはずである。しかし、図からわかるように OP7 や OP8 の利用はあまり実行速度の向上につながらない。これは、データの依存関係や使

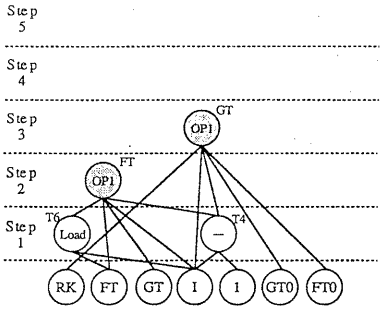

図 7: OP1 を利用した DAG

用できる命令の数によって、必ずしも機能ブロック化された新しい命令が有効に働かないためである。結果として、機能ブロックを 1 種類のみ利用した場合、命令数を 10 から 19 の範囲で選択、実行回数を 18763 ステップから 24781 ステップの範囲で選択が可能なことがわかる。

このようにユーザは、自分の資源と速度に対する要求を考慮しながら設計を進めることが可能である。また、この例では、機能ブロックは 1 ステップで実行されるものとして計算を行っているが、本システムでは機能ブロックの実行ステップ数の変更も設定が可能である。

例として OP1 を使用して合成を行うと、図 4のグラフは、図 7のようになる (斜線部)。

さらにシミュレーションを繰り返しながら設計を進める。結果的に OP1 を使用して合成を行い、出力されたハードウェア情報が図 8である。

このようにして、必要に応じて機能ブロックの選択を行いながら、ユーザは自分の要求にあったハードウェアの合成が可能である。

5 評価

高位の仕様記述に基づいた専用プロセッサの設計においては、実際に VLSI を作製するにはまだ様々な問題を含んでおり、実際に作製されているわけではない。そのため、市販の専用プロセッサと直接比較することは極めて困難であるため、実際的な評価を下すのは難しい。そこで、本論文では、アルゴリズムとデータについて種々の要素を含む問題として、ソート・アルゴリズム (7 種類) を取り上げる。本システムを適用した場合の設計の相違を中心にして、本システムの評価を行うことにする。適用したアルゴリズムは、下記のものである。

ソート・アルゴリズム

- ○ クイック・ソート (QS1、ランダムデータ)
- ○ クイック・ソート (QS1'、昇順データ)
- ○ クイック・ソート (QS1"、降順データ)
- ○ クイック・ソート (QS2)
- ○ クイック・ソート (QS3)
- ○ ヒープ・ソート (HS)

```
(PROGRAM PARCOR_LATTICE_FILTER NIL (S_IN RK_IN E_OUT)
        (INPUT
         (S_IN  (SIGNED BIT 12))
         (RK_IN (SIGNED BIT 8)))
        (OUTPUT
         (E_OUT (SIGNED BIT 12)))
        (REGISTER
         (R0    (SIGN . S) (BIT . 12) (TYPE . INT))
           省略
        (MEMORY
         (FT    (ARRAY . 15) (SIGN . S) (BIT . 16) (TYPE . INT))
           省略
        (FUNC
         (OP1   (LOAD TR1 MEM0 TR0)
                (MUL TR3 TR2 TR1)
                (LOAD TR5 MEM1 TR4)
                (SUB TR6 TR5 TR3)
                (STORE MEM2 TR7 TR6))))
        ;;
        (PROCESS
         (B0001
          (S1   (MOVE R5 1)))
           省略
         (B0014
          (S16  (DEC R0 R5)
                (LOAD R2 RK R5))
          (S17  (OP1 FT R5 FT R0 R2 GT R0))
          (S18  (OP1 GT R5 GT0 R0 R2 FT0 R0)
                (INC R5 R5)
                (JMP B0013)))
           省略
         (B0022
          (S44  (DEC R0 R5))))
```

図 8: ハードウェア情報

- ○ シェル・ソート (SS)
- ○ 単純挿入法 (SIS)
- ○ 単純選択法 (SSS)
- ○ 分布数えソート (DS)
- ○ バブル・ソート (BS)

QS1、QS1'、QS1"は、同じ仕様記述に対して3種類のデータ (ランダム、昇順、降順に並べられたデータ) を用いたもので、他の例では、ランダムに並べられたデータを利用している。QS2 は QS1 とソートの比較相手の選択法を変えたものであり、アルゴリズムが若干異なるもので、QS3 はアルゴリズムは QS1 と同じで記述の仕方が異なるものである。

それぞれの仕様記述に対する命令数、生成されるコード列の数、実行ステップ数、逐次実行を基準としたスピードアップを図 9、図 10、図 11、図 12に示す。

QS1、QS1'、QS1"の命令数、コード列の数は、アルゴリズムが同じなので、最大と最小の変化は、当然同じとなる。

図 9は、それぞれのアルゴリズムに対する命令数の設計の範囲を示すもので、最小は、基本命令のみを 1 つに限定した場合に用いた命令数、最大は、実行速度の向上が見込める範囲での機能ブロックを最も多く利用した場合の命令数である。

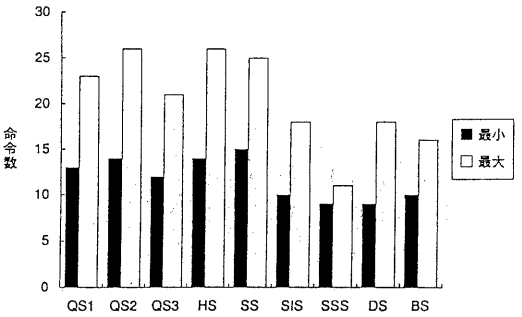

図 9: ソート・アルゴリズムと命令数の関係

図 10は、最終的に出力されるハードウェア情報によるフロー情報におけるコード列の数であり、並列化、機能ブロック化を行なうことで、ステップ数が減り、結果としてコード列の数も減ことになる。

図 11は、実行回数の幅を示すもので、最大は、基本命令のみを 1 つに限定して並列に実行を行ったもの、最小は、命令・機能ブロックを複数使って得られるもっとも速い実行回数のものである。

図 12は、並列化・機能ブロック化によって得られる逐次実行に対するスピードアップを示す。最小は、基本命令のみを 1 つに並列実行を行った場合、最大は、命令・機能ブロックを複数使って得られるもっとも速いものの速度向上の度合いである。

QS1、QS1'、QS1"によるサンプルデータの値の変化は、ループ回数等に影響してくるので、実行ステップ数には影響を与えている。また、合成品質に関しては、ブロックごとの頻度の変化により、選択する機能ブロックによって、変化することが考えられる。

そのため、ユーザは、ループ回数などの変化するアルゴリズムを扱う場合には、数種類のサンプルデータを与える必要がある。しかし、最大の機能ブロックを導入し

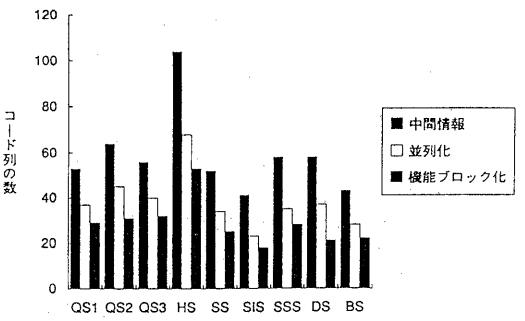

図 10: ソート・アルゴリズムとコード列の関係

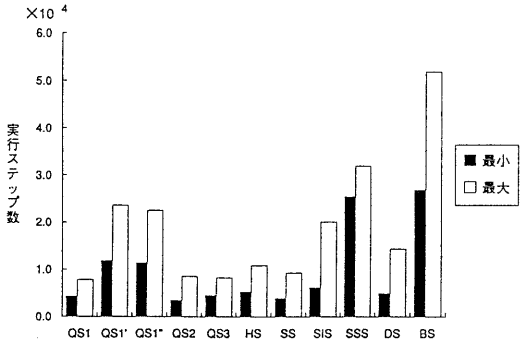

図 11: ソート・アルゴリズムと実行ステップ数の関係

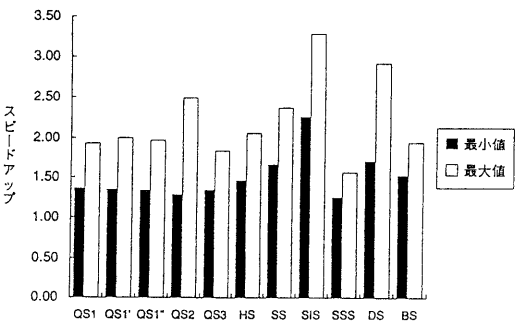

図 12: ソート・アルゴリズムとスピードアップの関係

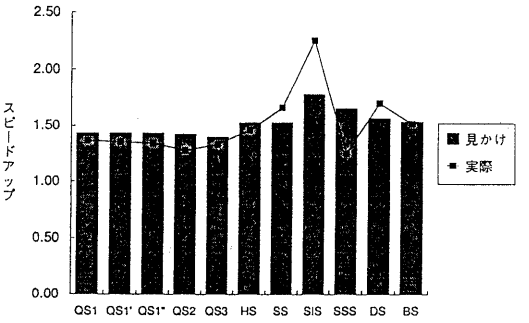

図 13: 逐次実行に対する並列実行 (各命令 1) のスピードアップ

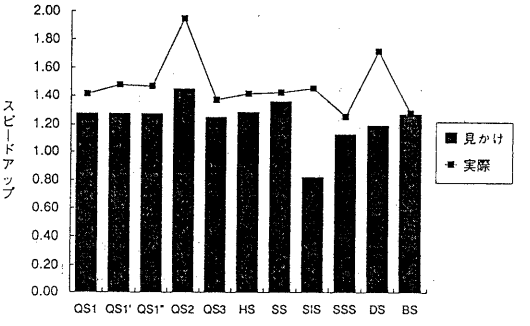

図 14: 並列実行 (各命令 1) に対する機能ブロック化によるスピードアップ

た場合には、スピードアップにはあまり影響していないのは、サンプルデータにあまり依存せずに全体の速度性能が向上したためといえる。

QS1、QS2 によるアルゴリズムの変化は、実行ステップ数、コード列の数にはあまり影響を与えない。しかし、アルゴリズムの変化が使用する命令に影響を与えるため、グラフ構造も変化し、命令数、スピードアップに影響している。

QS1、QS3 による仕様記述に対するものは、命令の選択の幅は変化するが、コードの最適化により、冗長な部分が吸収され、ほとんど合成品質に影響しない。

そのほかのソートアルゴリズムにおいては、機能ブロックの合成によって、ある程度差のあるプログラムでも、スピードをあげることができるといえる。つまり、ある程度ののユーザによるアルゴリズムの変化や記述の違いは、コードの最適化と命令の機能ブロック化によって対処が可能であるといえる。

コード列の比は、コード列の数が減った分、実行回数が減少すると考えられるので、見かけ上どのくらいスピードが上がったのかを示している。しかし、実際は、制御構造によって、複数回実行されるステップが存在するので、スピードがどれくらい上がるかは、シミュレータで調べることになる。

図 13、図 14に見かけ上のスピードアップと実際のスピードアップの差異を示す。図 13は、逐次実行に対する並列実行 (各命令 1) のスピードアップの関係。図 14は、並列実行 (各命令 1) に対する、並列実行と機能ブロックを最大に利用したものとのスピードアップの関係である。

2 つの図から、並列化を行なうと見かけのスピードアップに比べて実際のスピードアップはだいたい同じ値となっている。これは、実際の実行頻度によらず、全体が平均的に並列化が行なわれていることを示すが、SS、SIS、DS は、頻度の高い部分での並列化が有効に働いているために、見かけより実際のスピードアップの方が大きい。また、SSS は、逆にあまり有効に働いていないといえる。これに対して、機能ブロック化を行なうと、見かけよりも実際のスピードアップの方が高い値が出ている。これは、機能ブロックの実行速度が 1 ステップになるようにしてシミュレートしたことも影響しているが、頻度の高い部分で有効に機能していることを示している。

これらの見かけと実際の違いが存在するため、あまり有効でない機能ブロックは選択しない設計方針が必要となる。本システムの設計では、機能ブロックのステップの設定や、サンプルデータの違いによって変化することが考えられるが、これらの見えない部分のシミュレートは、無だな機能構成要素の削除に有効である。

6 おわりに

高位仕様記述からの専用プロセッサ設計における機能設計を中心として、本システムの設計指針、処理概要を述べた。そして、複数のソート・アルゴリズムへの適用を試みた。

その結果、シミュレーションを通してそのアルゴリズムの特徴的な部分を機能ブロックとして合成して高速化を行い、資源と実行速度を考慮した機能設計が可能なことが示された。また、設計された専用プロセッサのデータの比較によって、シミュレータの頻度解析によって機能ブロックを選択していく設計が有効であることが示された。

おわりに、今後の課題について述べる。本システムでは、機能ブロックとして合成した命令列を利用して、速度の高速化を行っている。そして、その機能ブロックに対してその内容に速度の重みをかけ、ステップ単位での速度の情報を持たせてシミュレーションを行い、できるだけ実際に近い評価を可能にし、それを繰り返すことでより良い設計結果を得ていくという方法を取っている。しかし、より実際的なシステムとして利用するためには、下位の設計支援システムから、速度性能等の情報をフィードバックする必要性があると考えられる。

より実際的な情報を用いて、質の良い設計を行えるシステムを構築したい。

参考文献

[1] 田丸啓吉：専用 VLSI プロセッサの現状と動向、情報処理、Vol.31、No.4、(Apr.1990)

[2] 北畠、雨坪、白井：高位仕様記述に基づく専用プロセッサの機能設計について、情報処理学会第 43 回全国大会、4R-3、(1991-10)

[3] 池永、白井：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、情報処理学会論文誌、Vol.32、No.11、(1991-11)

[4] Alfred V.Aho, Ravi Sethi, and Jeffrey D. Ullman "Compilers", Addison Wesley,1986

[5] 上野、竹沢、白井：「回路自動設計のためのアルゴリズム記述とフロー解析」、情報処理学会第 33 回全国大会、3R-4、(1986-10)