設計自動化　45—9
(1988. 12. 16)

制御論理合成の一手法

A METHOD ON CONTROL SYNTHESIS

若林　一敏　　吉村　猛
Kazutoshi WAKABAYASHI　Takeshi YOSIMURA
(日本電気(株)　C&Cシステム研究所)
C&C Systems Research Laboratories, NEC Corporation

あらまし　本論文では、汎用プログラミング言語C，及び動作記述言語ＢＤＬで記述された動作アルゴリズムからテクノロジディペンデントな論理回路接続情報を合成するシステムにおいて、特に制御論理をステートマシンによって合成する手法について述べる。

本手法は、条件分岐やループ、ジャンプ等を含んだアルゴリズム記述から、冗長なステップを含まない高速なステートマシンを合成する。まず従来のスケジューリング手法と同様にデータフローグラフを作成、解析し、演算の並列実行可能性を求める。次にその結果を木構造のコントロールフローグラフに反映し、その木構造グラフを制御構造に着目して構造を変換する。構造変換されたコントロールグラフから、各ステップで実行される動作の組を決定し、ステートマシンを生成する。

Abstract　This paper describes automatic degital circuits design system. This synthesis system generates technology dependent netlists from programming language C and Behavior Description Language BDL. And In this paper, a new method on contol synthesis is presented. Using this method, the system synthesizes controller composed of fast finite state machine(FSM).

This system can synthesize a fast FSM from the complex behavior description contained conditional branch, loop, jamp and so on. First, this system generates Data Flow Graph, and analyzes it in order to determine the parallelism of operators. Then, it constructs tree structured control flow graph, and reconstructures that graph. In this way, a FSM is synthsized.

1．はじめに

半導体製造技術の進歩にともない、大規模な論理回路を１チップ上に実現することが可能となってきた。一方、設計技術も進歩してきてはいるが十分ではなく、既に大規模ＬＳＩの製品価格の大半は設計コストとなってきている。特に、回路の大規模化複雑化に伴い、論理設計工程に要する時間が急速に増大しており、計算機支援による効率化が必須となっている。

機能設計工程の効率化のため、従来の設計スタイルから離れ、ソフトウェアによって記述された動作アルゴリズムから制御論理・データパスを自動生成し、ハードウェア化する研究が行なわれている。[1,2,3] このような研究の実用化により、ハードウェア設計に従来の膨大なソフトウェア資産や、ソフトウェア作成技術が利用できるようになる。将来的には、ＯＳや各種応用プログラムが複数のＩＣカードからなるようなシステムが可能となることも考えられる。我々は、このようなＬＳＩをソフトウェアチップと呼んでいる。

Ｃ等のシーケンシャルなアルゴリズムから、並列に実行するハードウェアを合成するためには、シーケンシャルなアルゴリズムから並列性を抽出し、並列動作アルゴリズムに変換するスケジューリング手法が重要である。従来のスケジューリング手法では、記述から得られたデータフローグラフ（DFG）をハードウェア資源の制約に基づいて「並列実行可能な演算動作の組」を含むステージに分割し、各ステージに１状態を割り当てていた。[2,4,5] しかし、このような手法は、４章で示すように分岐を含まない動作系列には良い解を与えるが、分岐やループ、ジャンプを含む複雑な動作系列では、冗長な動作ステップを与えるという問題があった。

本稿では、Ｃまたは、動作記述言語ＢＤＬ（Behavior Description Language）[5,6]で記述された動作アルゴリズムからテクノロジディペンデントな論理回路接続情報を合成するシステムにおいて、特に制御論理をステートマシン（FSM:Finite State Machine）で合成する手法について述べる。

本手法は、条件分岐やループ、ジャンプ等を含んだアルゴリズム記述から、従来のスケジューリング手法と同様にDFG解析により、演算の並列実行可能性を求めた後、更にその結果を木構造の制御フローグラフ（tCFG）[6]に反映し、そのtCFGの制御構造に着目することによってその構造を変換し、各ステップで実行される動作の組を決定することにより、上述した問題点を解決するような高速なステートマシンを生成する。

以下、２章でシステム構成について、３章で入力となる動作記述言語ＢＤＬについてそれぞれ簡単に述べる。４章で制御論理生成手法について、５章で合成実験の結果とその評価、６章でまとめを述べる。

2．システム構成

本合成システムの構成を図１に示す。本システムは、自動機能合成システムと論理合成システムFUSION[7]からなり、自動機能合成システムはさらにトランスレータ、並列化処理部、ステートマシン生成部、データパス生成部（演算器・変数・データ転送路割当部）、制御信号生成部、FDL生成部からなる。

与えられたC記述、またはBDL記述はトランスレータによってともに同一の中間コード形式に変換される。中間コードは、データフローグラフ（DFG）と木構造コントロールフローグラフ（tCFG）からなる。

並列化処理部は、使用可能な演算器制約のもとでDFGの変数の依存性をもとに並列化を行なう。このとき演算器の併合が行なわれる。並列化結果はtCFGに反映され、ステートマシン生成部がこの木構造グラフを基に状態遷移テーブルを作成する。

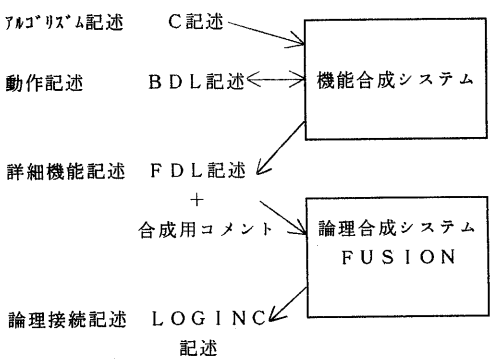


図１．システム構成

次に、データパス生成部で、必要な演算は利用可能な演算器に割り当てられ、変数は併合された後、レジスタまたはメモリに割り付けられ、更にデータパス転送路が併合されバスに割り付けられる。最後に、演算器やデータ転送等データパスの制御信号が生成される。

機能合成システムと論理合成システムの中間段階として機能記述言語FDL（Functional Description Language）[8]が用意されている。FDLは、構造の記述力を強化した機能記述言語であり、機能モジュール毎にその動作を記述し、動作はレジスタや端子へのデータ転送をIF THEN ELSE、CASE形式の条件と共に記述するものである。オートマトン等の時間の概念は含まず、データパス構造を明確に表現することが可能である。FDL生成部は、中間コードをFDLに変換する部分である。但し、BDLはFDLと等価な記述が可能であるから必ずしもFDLに変換する必要はなく、全ての工程をBDLによって表現することは可能である。

論理合成システムは、FDL記述を論理ゲート回路に変換し、論理的に最小化し、更にテクノロジマッピングを行い、所望のテクノロジの論理接続記述を出力する。

3. 動作記述言語ＢＤＬ（Behavior Description Language）

本システムではC及びBDLを入力としているが、本章ではBDLについて簡単に述べる。BDLは、Cに並列動作記述や動作タイミング記述等のハードウェア記述用の拡張を行った言語であり、基本的な構文及び演算記法はCと同様である。よって、CとBDLは、トランスレータによって同一の中間コードに変換することが容易である。

DDL[9],HSL-FX[10]等多くのRTレベル言語では、詳細な時間の概念を表現するのにオートマトンの枠組みを用いるため、動作のステップをすべて状態の遷移として記述しなければならず、大規模な動作シーケンスの記述には適していない。一方、ISPS[11]の「NEXT」のように１ステップの時間の経過を示す順次実行シンボルにより順序動作を表現する言語は、大規模な動作シーケンスでも比較的制御の流れが分かりやすく記述できる。しかし、IF文やループ文が入れ子になったときなど同時に実行される動作の集合が分かりづらく、「NEXT」のみでは並列に実行される動作を簡潔には記述できない場合があった。そこで、BDLは、Cの制御構造をそのまま残しつつ、"$"と"^"という２つの順次実行シンボルにより、「NEXT」では、判別性良く記述できなかった条件判断用の演算と実行文の演算の関係等も含め正確にかつ簡潔にクロックに同期した動作ステップを記述できる。"$"は、ISPSの「NEXT」に近い意味であり、"^"は、BDL独自の表現であり、意味は次章で述べられる。つまり、Cの形式のまま"$"と"^"記号によりオートマトンと同等以上の厳密な動作表現が可能である。その他、レジスタ・端子転送表現や動作タイミング等のハードウェアよりの記法は、DDL,HSL-FX等のRTレベル言語とほぼ同様である。（記述例２）に、BDLの記述例を示す。

ハードウェアを意識した記述をするには、Ｃは不向きであり、ハードウェアを意識して設計を進める場合は、ＢＤＬを用いる。

4. 制御論理生成手法

本章では、機能合成システムの機能記述合成手法について、特に制御論理の生成手法を中心に述べる。

従来のスケジューリング手法は分岐点を持たない連続的な演算の最適化に重点がおかれていた。そのため、最も高速になるようにASAP手法[2]によりDFG上の演算をスケジューリングした場合でも、各ステージに状態を割り当てると、分岐を含んだ記述に対しては冗長なステップや演算器の浪費が起こることがある。[6]提案する手法では、DFGと木構造のコントロールグラフtCFGを用いる手法により、このような分岐やループ、ジャンプを含むアルゴリズムから冗長なステップを生じないようなステートマシンを簡潔な処理で生成することができる。

以下、各サブステムの処理内容を順に簡単な例を用いて述べる。なお、本稿では説明を簡単にするため、各演算はすべて１クロックで実行できることと仮定する。また、マルチプロセスや、関数呼び出し等に対する扱いについても触れない。

4．1 トランスレータ

まず、与えられたC記述、BDL記述をデータフローグラフ（DFG）、及び木構造コントロールフローグラフ（tCFG）に変換する。Cの記述例を図１に、対応するDFGとtCFGの例を図２と図３にそれぞれ示す。DFGでは、実線がデータの依存関係を示している。if文の分岐は、図のようにthen節の出力とelse節の出力を(x<y)の出力によって選択する事で実現する。tCFGは、C（BDL）の制御構造をPAD図式を利用して表わしたものである。但し、else節の演算は２項演算に分解され、必要な中間変数t1,t2が生成している。図でBOX1の二分木がif文を表わしており、BOX3または、BOX7が実行されると次に、BOX8が実行される。

```
if ( x < y ) { a = x + y;
               x = a - z;  }
else         { x = z + w;
               a = x*(z-w)-y; }
out = a + w;
```

図２．Ｃの記述例

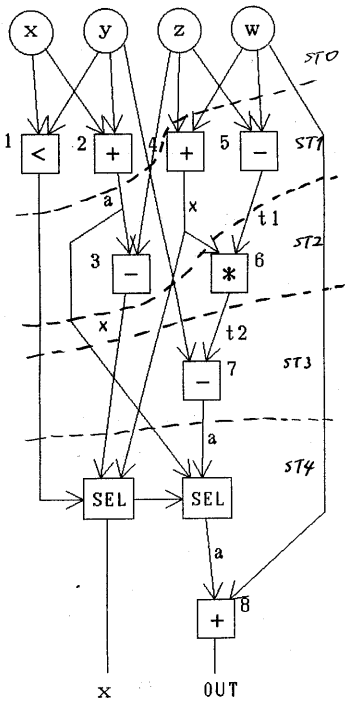


図３．　図２のＤＦＧ

4．2 並列化処理部

次に、与えられた演算器の制約条件のもとで、DFGをもとに演算器同士の並列実行可能性を解析する。本処理のステージは、実現されるステートマシンの状態とは必ずしも対応しない。演算器の制約としては、単にALUの個数を与えるのではなく、各ALUが実行できる演算の種類も指定する。これにより、乗算器と加算器を同一演算器に割り付ける等というハードウェア構成上好ましくない組合せを避けている。本稿では、基本的な処理手順を簡単に述べる。

1）DFGのデータ依存性に基づいてASAP(As Soon As Possible)手法[2]によってスケジューリングする。

2）1)で得られた全てのステージに対してデータの入力側のステージから以下の処理を繰り返す。

ｉ)もし、ステージ内の演算の組が、与えられた演算器の制約条件を満たしていれば終了。次のステージへ。満たされなければ、ii)へ

ii)制約条件を満たすまでそのステージで実行する演算を次のステージへ移す。次のステージに移される演算は、次のステージの演算とデータ依存性の無いものを優先して選ぶ。それ以外は、ステージ内の演算であって、tCFGを深さ優先探索（Deapth First Search）した時最も最後に出現する演算が選ばれる。次のステージに移された演算が次のステージの演算と並列に実行できない場合は、更にその演算を次のステージに移すといった工程を矛盾が生じないように行なう。

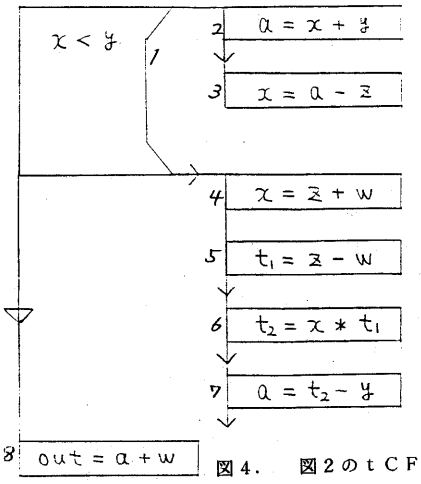


図４．　図２のｔＣＦＧ

なお、上記の処理で、if文のthen節とelse節の演算は相互に排他的に実行されるから、別々にスケジューリングされる。なお、本手法は、DFGの最小ステージ分割という意味では必ずしも良い解を保証しておらず、今後の検討が必要である。

図３で、加減算と比較演算ができる二つのALUと、一つの乗算器が使用可能ならば、破線で示されたようなステージに分けられる。並列に実行可能な演算の組は同一ステージにあるものの他、3と8も異なるステージにあるが並列に実行可能である。また、3,4,5は同一ステージにあるが3と4,5は相互に排他な関係にあるので、ALU二つという演算器の制約条件を満たしている。

ここで、従来法によるステートマシン生成を考える。[6]で従来法により生成されたステートマシンの問題点を指摘したが、この程度の問題であればif文のthen節とelse節を区別する様にDFGを改良する事によって従来法でも解決でき、改善された状態遷移は以下のようになる。

```
(FSM1)
  ST0: fire <(1),+(2)  goto ST1;
  ST1: fire -(3), goto ST4;
       or fire +(4),-(5) goto ST2;
  ST2: fire *(6) goto ST3;
  ST3: fire -(7) goto ST4;
  ST4: select a between the outputs of +(2)
and -(7), fire +(8), exit;
```

しかし、更に以下のようなより高速な状態遷移が存在する。

```
(FSM2)
  ST0: fire <(1),+(2)
       if (the output of < ) goto ST5;
       else                  goto ST1;
  ST5: fire -(3),+(8),exit;
  ST1: fire +(4),-(5) goto ST2;
  ST2: fire *(6) goto ST3;
  ST3: fire -(7) goto ST4;
  ST4: fire +(8) exit;
```

つまり、演算+(8)が二つの状態に割付られることによってif節が実行された場合に、より高速になる。このような状態の割付は、DFG分割によって得られたステージをステートマシンの状態に対応させるような従来の割付方法では困難である。特に、分岐やジャンプが複雑に入れ子になっている場合に最適なステートマシン処理はより困難であり、冗長なステップを生じてしまう。そこで、我々は演算器の並列性解析の後さらに、その結果をtCFGに反映し、tCFGを解析・操作し、最適な動作の組をステートマシンの各状態に割り付けることによって、最適なステートマシンを生成する手法を提案した。次節にその詳細を述べる。

4．3 ステートマシン生成部

ステートマシン生成部は、並列化処理部で得られた演算器の並列実行可能性に基づいて、演算の実行順序を決定し、制御回路をステートマシンにより生成する部分である。

（１） tCFGの並列化：順次実行シンボル（"v"、"▽"）の挿入

4.2で求めた並列実行可能性に基づいて、tCFGの並列に実行できないboxとboxの間の枝に順次実行シンボル"v"をつける。これは、BDLの"$"と同義である。tCFGでは"v"で区切られた部分も必ず木となり、この部分木につながる動作はすべて並列に実行される。以下、実線でつながり、"v"で区切られた並列動作集合を「１ステップ部分木」と呼ぶ。記述例１を前節で述べた並列実行可能性結果に基づいて順序化した結果が、図４である。図で"▽"は、即時実行シンボルと呼ばれ、BDLの"^"記号と同義である。これは、同一ステップ内の動作間の区別を示すシンボルであり、出現する位置（分脈）によって意味が異なる。図＊の場合は、IF文の直後についているが、この場合は、直前のIF文が終了したと同時に実行されることを示している。即ち、"▽"の直前の分岐した枝の最終ステップ（図４ではBOX3とBOX7）と"▽"の直後のステップ（BOX8）の間の実行順序を示しており、BOX3とBOX8は同時に実行され、BOX7とBOX8は順次実行されることを表わす。よって、(x<y)が真の時は実行に3ステップ必要であり、偽の時は5ステップ必要である。このように、"v"（"$"）と"▽"（"^"）により動作の実行タイミングが正確に表現できる。

以上の議論から明らかなようにBDLとtCFGは表現形式が違うだけで等価である。よって、BDLは、近代的流れ制御機構を持ちながら各動作の厳格な実行タイミングが記述できることが分かる。また、設計者が詳細な動作をBDLで記述すれば、本節の（２）以降の処理のみで即ち、DFG解析を行なうこと

なく記述通りの動作をするステートマシンを生成できる。

（２） 基本的な状態割当手法

tCFGがループや、即時実行シンボル"▽"を含まず、if文とgoto文、順次実行シンボル"v"とからなるとき、それをtCFG基本構造と呼ぶ。tCFG基本構造は、[6]に示したように簡単な手順でステートマシンに変換することができる。

【変換手順】

1) "v"で区切られた１ステップ部分木につながる動作の集合は、並列に実行され１ステップで動作可能であるから、１ステップ部分木に１つの状態を割り当てる。

2) tCFGをツリーウォークすることによって各状態の次状態を決定し、それぞれの状態遷移条件を求めることにより、ステートマシンを生成する。

このような機械的で簡単な処理によって基本構造のtCFGから最適なステートマシンを生成できる。

goto文を含む時の状態割当法は、説明の都合上（３）の2)で述べる。

（３） 木構造変換

即時シンボル"▽"や、ループ、並列ジャンプ[5,6]等を含むtCFGの木構造を変換する部分である。これら特殊な動作を表わす記述を含むtCFGは、前節で示された基本的な状態割当手法では処理できない。そこで、これらを含むtCFGをtCFG基本構造に変換し、この変換結果を前節で述べた状態割当部によって処理することにより、（FSM2）の様な高速なステートマシンが得られる。

【木構造変換処理】

1) IF文の直後の即時実行シンボル"▽"の処理

（１）で述べたように"▽"記号は特別な動作シーケンスを表わす。このとき、"▽"の直後にくる動作は"▽"の直前のIF_BOXにつながる動作と並列または、順次に実行される。このような例として先ほどのtCFG（図４）を、基本構造に変換した後のtCFG（図５）を示す。ここでは、BOX8がBOX3の下に並列に動作するようにコピーされ、BOX7の次のステップでBOX8が動作するようになっている。この基本構造（図５）から前節の手法で生成されるステートマシンは（FSM2）で表わされる最適なものである。

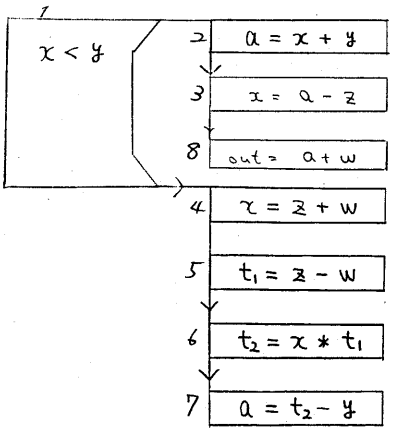


図５． ｔＣＦＧの基本構造

次に、より一般的な例（図６）を用いて変換アルゴリズムを示す。

i ）"▽"の直後にくる部分木１を、"▽"の前にあるIF_BOXをルートとする木２の葉で"v"記号の無いBOXに並列に動作するようにコピーする。"v"記号のついたBOXの"v"記号は削除する。

ii ）部分木１の"v"でつながる先にラベルをつけ、コピーした部分木には対応するラベルへのジャンプ文をつなげる。

上記の様に変換することにより、if文の直後にくる"▽"記号を含むtCFGは基本構造に変換できる。

2) ループ文・ジャンプ文の処理

ループ文とgoto文の処理について説明する。並列化処理の終わった後のtCFGをBDLで表わした記述例２を例としてもちいる。

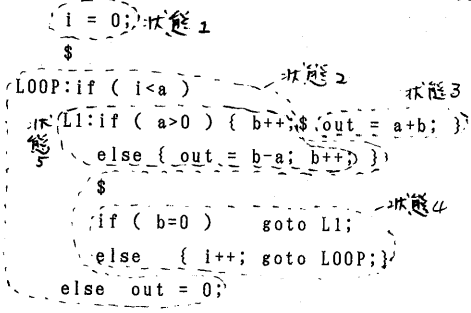
（記述例２）

```
i=0;
$
while (i < a) {
L1: if ( a>0 ) { b++;$ out = a+b; }
    else       { out = b-a; b++; }
    $
    if ( b=0 ) goto L1;
    ^
    i++;
}
^
out = 0;
```

【ループ文(while,for,do-while)の処理】

ループ文はまずif文とgoto文に変換される。このとき、ループの前後の動作や、goto文、for文の変数の再設定動作等を冗長なステップを生じず最速に実行できるように木構造を変換する。変換された結果をBDLで表わすと以下のようになる。破線で囲まれた動作集合が並列に動作する。

(if文とgoto文によって変換した結果のBDL表現)



【goto文の処理】 =状態割付手法=

goto文は基本的には状態の遷移で実現できる。例えば、"∨"の直後のラベルへのジャンプであるLOOPへのジャンプは、状態2への遷移で実現できる。しかし、"∨"の直後でないラベルL1へのジャンプは状態2への状態遷移では実現できない。なぜなら、状態2では(i<a)の判断と(a>0)の判断を並列に実行するが、L1へのジャンプではL1以下の動作だけを実行し、(i<a)の判断は行なわないからである。そこで、L1以下の動作集合に新しい状態を割り当て(状態5)れば、goto文を新しく生成した状態5への遷移で実現できる。

このようなgoto文に対する処理手法を用いると、複雑に入り組んだ制御構造をもつアルゴリズムを統一的に最速なステートマシンで合成することができる。

3) ループ文の前後動作処理

ループ文の前後に"▽"があると、前後でそれぞれ異なった意味を表わす。ループの下方にある場合は"▽"から"∨"までの動作をループから抜けたタイミングで実行することを示す。記述例2では

```
i++; と  out=0;
```

がwhile文を抜けるタイミングで実行するように変換されている。

ループの上方に"▽"がある時は、"▽"より上方にある演算はループに入る前に一度だけ実行されることを意味する。これは、"∨"の直後でないラベルへのgoto文へ変換される。

4) switch文や ?演算子、代入演算子(+=等)等

これらの表現も木構造変換によって基本構造に変換される。特にswitch文では、動作を実行する条件が変更される。

5. 合成結果、評価

提案した手法を評価するためのプロトタイプシステムを、C言語でワークステーション上に作成した。現在、トランスレータ部がyacc,lexで約5,000行、機能合成システムが約10,000行である。

ソフトウェアチップの例として、二分木ソートのアルゴリズムを本システムによって合成した。変数宣言部を含めて約80行の記述を変換するのに、トランスレータも含め機能合成全体で約2秒であり、論理合成が約5分であった。作成したメモリー付きゲートアレイによるチップ写真を示す。(写真1)

同じ二分木ソートアルゴリズムをSUN2(CPU68000)上でコンパイルして同一のテストパターンについて実行させたところ約300倍の高速性が確認された。これは、コンパイラの生成した機械語を68000マイクロコンピュータ上で実行した場合と、専用のソフトウェアチップの実行速度を比較したことを意味する。

また、いくつかのアルゴリズムについて実験した結果、数値演算をあまり含まず、分岐やジャンプを多く含む記述に対してはかなりよい結果を出すが、信号処理演算など連続的に多くの数値演算を行なうときには明確な効果は得られなかった。これは、並列化処理部のアルゴリズムに問題があるためだと考えられる。

本手法で合成される回路は、DFG解析によって並列化された回路に比べ、変数名等の対応がつき易いことも分かった。

6. まとめ

汎用プログラミング言語C及び、動作記述言語BDLにより記述されたアルゴリズムを高速なステートマシンで実現する手法を提案し、プロトタイプシステムの作成を行なった。木構造のコントロールフローを構造解析・構造変換することにより、分岐やループ、ジャンプ等を含んだ記述に対して、冗長なステップを生じない高速なステート

マシンが生成できることを確認した。

提案した手法に基づいて二分木ソートのアルゴリズムを実チップ化した結果、68000マイクロコンピュータ上でコンパイラの生成した機械語を実行させた場合と比べ百倍以上の速度があり、人手設計に比べても遜色ない実行速度のものが合成できた。

謝辞

最後に、この研究の機会を与えてくださったC&Cシステム研究所石黒所長、応用システム研究部後藤部長、いろいろとご指導を頂いた応用システム研究部石川主任、並びに本システムのプログラム作成に御尽力頂いた日本電気技術情報システム（株）麻野氏、田中氏、岡田氏に感謝致します。

【参考文献】

[1]Tseng,C.J. et al.,"Facet: A Procedure for the Automated Synthesys of Digital Systems," Proc. of 20th DA conf. pp490-496 (1983)

[2]P.Paulin and J.Knight,"Force-Directed Scheduling in Automatic Data Path Synthesis," Proc.of 24th DAC pp.195-202 (1987)

[3]F.Maruyama et al.,"Prolog-Based Expert System For Logic Design," FGCS'84 pp.563-571(1984)

[4]田中、小林：「演算器の自動機能割付法の検討」情処第36回全大5X-7 (1987)

[5]若林：「ハードウェア記述言語とステートマシン生成手法に関する一考察」 情処第35回全大3F-3 (1987)

[6]若林、吉村：「複雑な制御構造を持つアルゴリズムからの高速なステートマシン合成手法」 情処第37回全大1U-5 (1988)

[7]T.Yoshimura et al.,"A Rule-based and Algorithmic Approach for Logic Synthesis," International Workshop on Logic Synthesis Reserch", May 12-15,(1987)

[8]Kato,S.,Sasaki,T.,"FDL:A Structural Behavior Description Language," CHDL'83 pp137-152 (1983)

[9]Duley,J.R., Dietmeyer,D.L. "A Digital System Design Language (DDL),IEEE Trans. Comput.,Vol.C-17,No.9,(1968)

[10]Hoshino,T.Karatsu,O.,Nakashima,T. "HSL-FX:A Unified Language for VLSI Design, Proc.of 7th CHDL (1985)

[11]Barbacci,M.R. "Instrunction Set Processor Specifictions(ISPS):The Notation and Its Applications", IEEE TRans.Comput., Vol.C-30,No.1,pp24～40,(1981)

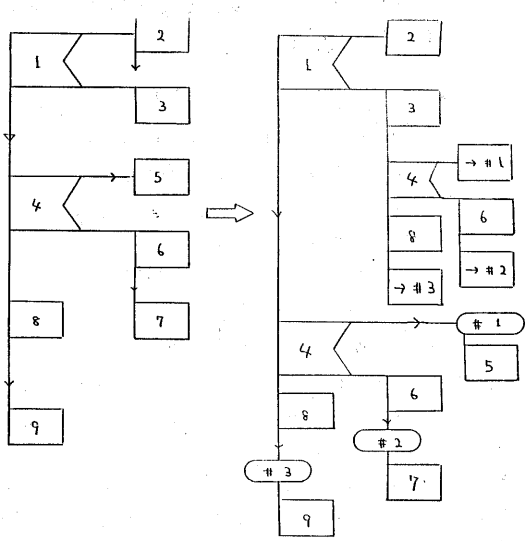


図6.　一般的なｔＣＦＧの記述例

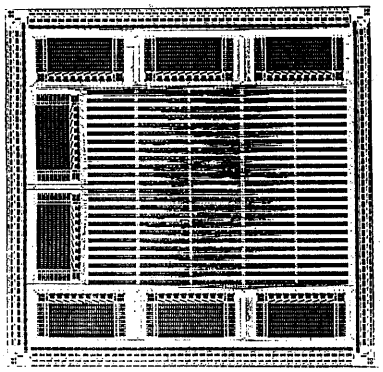


写真1.　二分木ソータチップ