設計自動化 43－1
（1988. 7. 12）

論理合成エキスパートシステム（CLS）について

浜崎 良二 ，今村 真人 ，北村 幸太 ，白木 昇

沖電気工業（株）超ＬＳＩ開発センタ

ＶＬＳＩの論理設計における設計期間の短縮を目的として、論理合成エキスパートシステム（ＣＬＳ）を開発した。
本システムは、ＬＳＩの動作・機能記述より、ライブラリへ割付済の論理回路を合成し、さらにフロアプランを出力する。
合成対象は、カスタム回路全般であり、ＬＳＩの動作・機能記述は、ＨＳＬ－ＦＸ（Hierarchical Specification Language Function Extension）を用いている。
また、合成処理においては、人手設計における割付作業、最適化作業の特徴をルール型言語を用いて表現することで、最適化、フロアプラン等の処理負担の軽減、レイアウト性能の向上を図っている。

Custom LSI Logic Synthesis Expert System CLS for VLSI Design

Ryoji HAMAZAKI. Masato IMAMURA. Kohta KITAMURA and Noboru SHIRAKI

VLSI Research & Development Center, Oki Electric Industry Co., Ltd.,
550-1. Higashiasakawa-cho. Hachioji-shi. Tokyo. 193 Japan

This paper presents Custom LSI Logic Synthesis Expert System (CLS) developed for the purpose of shortening VLSI logic design TAT.

CLS synthesizes a logic circuit assigned to the cell library and its floor plan from register transfer level functinal description. We adopt Hierarchical Specification Language Function Extension (HSL-FX) as functional description language.

This system can be applied to all types of digital custom circuits. Its main feature lies in the adoption of a rule-based approach able to capture design expertise in a straight-forward manner. Experimental results show that CLS synthesizes several thousand gate circuits in reasonable computational time.

1．はじめに

近年のＶＬＳＩ製造技術の進歩により、ＶＬＳＩの高集積化・大規模化が急速に進んでいる。そのため、ＣＰＵ等の１チップ化が可能となり、100KゲートにもおよぶデジタルＶＬＳＩが現われてきている。

しかし一方で、これら製造技術の進歩はＶＬＳＩの設計を複雑化し、設計期間の増大という問題を引き起こしている。この設計期間の増大は、設計データ量の増大に起因するところが大きい。

設計データ量の増大を回避するためには、よりデータ量が少ない抽象度の高い、機能設計・論理設計レベルからのＣＡＤ化が必要不可欠である。

我々は、そのためのアプローチとして、論理設計のＣＡＤ化を目的とする論理合成エキスパートシステムＣＬＳ（Custom LSI Logic Synthesis Expert System）の開発[1][2]を行なった。

本システムは、ＬＳＩの動作・機能を定義した機能記述言語を入力して、それらを実現するための論理回路を合成し、ＬＳＩのフロアプランを出力する。合成対象は、カスタム回路全般であり、入力の機能記述言語にはＨＳＬ－ＦＸ（拡張階層仕様記述言語）[3]を採用している。また、合成処理は、人手設計作業の特徴を取り込む観点からＯＰＳ系のルール型言語を適用し構築した。

本稿では、システムの開発方針、システム構成、合成処理概要、及びその適用結果について報告する。

2．システムの開発方針

本章では、ＣＬＳの開発方針を基本方式、入力設計言語、及び開発言語の点から述べる。

2.1 基本方式

論理合成は、合成対象の動作・機能を定義した機能表現より、その機能を実現するための論理回路間の接続関係を定義した構造表現への変換処理である。その処理過程では、論理回路への割付、及び最適化等が行われる。

現在までに報告されている論理合成システムをその基本方式で分類すると、以下の３つに大別できる。

ⅰ）合成対象を組み合わせ回路に限定した方式[4]

論理を表わすブール表現を入力し、ＰＬＡ、ゲートアレイ等でレイアウトを実現するものであり、100ゲート程度の小規模回路に向いている。

ⅱ）特定のアーキテクチャ・フロアプランのみを合成対象とした方式[5][6]

設計対象を特定の機能をもつ回路（例えばＤＳＰ）に固定し、レイアウトの実現方法（割付対象、フロアプラン等）を限定したものである。そのため、割付、最適化、フロアプラン等の処理負担が小さくなり、アルゴリズムレベルの設計言語からでも人手設計に近いレベルの集積度を有するレイアウト結果を得ることができる。

ⅲ）カスタム回路全般を合成対象とした方式[7][8]

広範囲な機能をもつ回路を設計対象にしたものであり、ＲＴＬ表現の設計言語を入力とするものが多い。ただし、汎用性を追求しているため、割付、最適化、フロアプラン等の処理負担が大きくなる。また、入力の設計言語に構造化への割付情報が不充分な場合、回路の小規模レイアウトを実現せざるを得ないことから、最終的に得られるレイアウト結果よりの集積度・性能等が人手設計に比べ大きく劣る傾向にある。

本システムでは、広範囲な設計対象の論理設計の効率向上を目的とすることから、基本的には上記ⅲ）の方式をとる。ただし、最適化・フロアプランの処理負担の増大、割付単位からくる処理負担の増大、割付単位からくるレイアウト性能の劣下等の問題に対処するため、以下に述べる様な人手設計作業の特徴を取り入れ、その高性能化・高集積化を図ることとした。

１）割付処理

人手による機能設計作業では、機能のかたまりやデータ信号の流れに着目しながら、設計対象のブロック図作成を行う。この作業では、バス信号、ブロック内

の構造の繰り返し性の抽出等、レイアウトにおけるブロック間配線、ブロック自体の面積最小化の工夫を行っている。

この手法を取り入れるため、機能記述言語の記述内容から機能のかたまり（以下、機能セットと記す）を抽出し、それをより大きなブロック（ＡＬＵ、レジスタアレイ等のデータパス、乗算器、ＲＡＭ、ＲＯＭ等）へ割付ることに重点をおく。

２）最適化処理

人手による最適化作業は、面積、遅延、消費電力等のトレードオフを行いながら進められる。また、その方法は、最適化するブロックの種類等により異なる。そこで、最適化の内容をその目的、対象により細分化し、それぞれに応じた処理を行う。

図１にその基本方式の概念図を示す。

２．２　入力設計言語

合成システムの入力となる設計言語の開発／選定は、システム全体の性能・処理能力に大きな影響を与える。抽象度の高い言語では、記述性は高いがその解釈があいまいとなり処理負担が増大し実現性が低下する。逆に抽象度の低い言語では、システムの処理負担は軽減されるが、設計者の機能設計期間は増大する。

本システムでは、入力データとなる設計言語にＨＳＬ－ＦＸを採用した。

ＨＳＬ－ＦＸは、ＶＨＤＬ(VHSIC Hardware Description Language)[9]同様、機能記述・構造記述の両方がサポートされているが、機能記述はＲＴＬ表現中心の設計言語である。ただし、その言語仕様は、ＶＬＳＩの動作をその実現方法を念頭において記述したり、構造化に対する指針を記述したりできるよう、工夫がなされている。そのため、記述の抽象度の高さは、ＶＨＤＬに比べやや劣るものの、論理合成向きの言語であるといえる。図２にＨＳＬ－ＦＸの簡単な記述例とその回路モデルを示す。

２．３　開発言語

開発言語は、２．１節で述べた処理を効率良く実現するため、主にＯＰＳ系のルール型言語を利用した。それは、最適化処理や割付処理には、パターンマッチング処理が多く要求されること、及びその開発効率が極めて良好であることからである。また、入力言語の解析処理等は、文法定義と変換処理が比較的容易に記述できるProlog系のＤＣＧ（Definite Clause Grammars）[10]を利用し、ファイルの入出力等の逐次処理に関しては、Ｃ言語ライクのものを用いた。

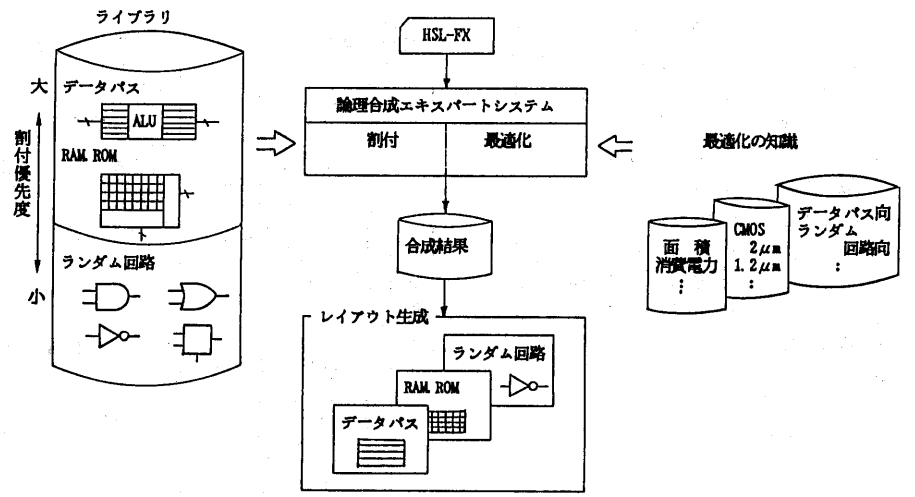


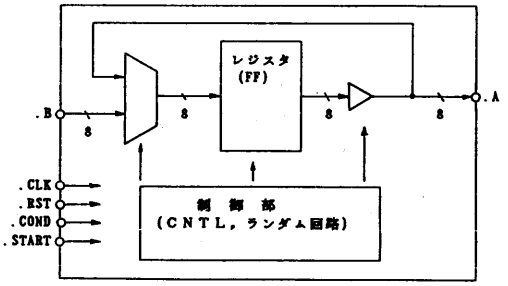
図１　論理合成の基本方式概念図

```
NAME     : CLSTEST   ;
PURPOSE  : EXSAMPLE  ;
LEVEL    : BLOCK     ;
EXT      : CLK. RST. COND. START. A<0:7>. B<0:7>  ;
INPUT    : . CLK. . RST. . COND. . START. . B  ;
BUS      : . A  ;
REGISTER : FF<0:7>  ;

BEHAVIOR-SECTION  ;
    BEGIN
        IF . RST THEN REST(FF) END-IF  ;
        IF CNTL*STATE2 THEN
            AT . CLK DO
                IF . COND THEN FF := . A ;
                          ELSE FF := . B ;  END-IF  ;
            END-DO  ;
        END-IF  ;
        IF CNTL*STATE3 THEN
            . A :- FF OFFSTATE Z  ;  END-IF  ;
    END :

    AUTOMATON : CNTL : . RST: . CLK ;
          STATE1 : WAIT(. START) : -> STATE2 ;
          STATE2 : -> STATE3  ;
          STATE3 : -> STATE2  ;
    END-AUTO  ;
END-SECTION  ;
END  ;
```

（a）ＨＳＬ－ＦＸの記述例



（b）回路モデル

図２ ＨＳＬ－ＦＸの記述例とその回路モデル

３．システム構成

本システムは、論理合成用データベースと段階的に合成処理を行う合成処理部とから構成されており、合成結果は、レイアウト自動生成システムへのインタフェースデータとなる。図３にシステムの概略構成を示す。

３．１論理合成用データベース

論理合成用データベースには、合成対象のＨＳＬ－ＦＸ記述の解析結果、各処理部における合成過程の結果、及び割付られる機能ブロック、論理ブロックを登録した機能ブロックライブラリ、論理ブロックライブラリのデータが格納されている。

１）機能ブロックライブラリ

機能ブロックライブラリは、システムがあらかじめ用意した機能ブロックの集合である。

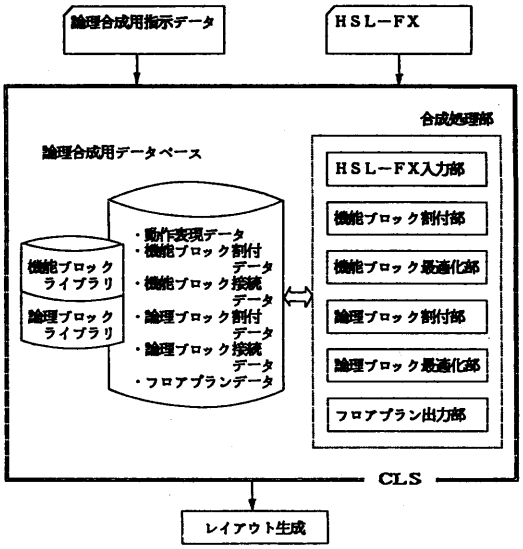


図３ ＣＬＳの概略構成

機能ブロックとは、合成対象の動作・機能を実現する上で必要とされる機能を、各機能単位に定義したブロックで、実際の回路、レイアウトとは独立している。

機能ブロックの一部を、ＨＳＬ－ＦＸ記述の対応とともに、図４に示す。

２）論理ブロックライブラリ

論理ブロックライブラリは、システムが用意した論理ブロック、及びユーザが必要に応じて登録した論理ブロックの集合である。

機能ブロックライブラリ			ＨＳＬ－ＦＸ記述
ＦＳＭ／ＰＬＡ			オートマトン／CASE文
ＲＯＭ，ＲＡＭ			メモリ宣言された資源
サブモジュール			サブモジュール定義
ランダム回路／データパスブロック	基本ブロック	インバータ バッファ AND, OR, EXOR 3-STATEゲート セレクタ	代入文、演算子
	レジスタ（同期・非同期）		レジスタ宣言された資源
	演算ブロック	加減算器 乗算器 除算器 比較器 シフタ	システム関数、演算子

図４ 機能ブロックライブラリの例

論理ブロックとは、実際の回路、レイアウトと1対1の関係にあるブロックで、デザインルール等に依存している。そのため、システム中には機能ブロックライブラリが唯一であるのに対し、論理ブロックライブラリは、そのデザインルールごとに複数存在する。

3．2 合成処理部

合成処理部は、ＨＳＬ－ＦＸを入力し、構文・意味解析を行うＨＳＬ－ＦＸ入力部、解析結果からデザインルールに依存しない機能ブロックのネットワークを生成する機能ブロック割付部、同最適化部、さらに、デザインルールに依存した論理ブロックのネットワークを生成する論理ブロック割付部、同最適化部、及び合成対象ブロックのフロアプランを出力するフロアプラン出力部より構成されている。

4．合成処理概要

本章では、ＣＬＳにおける合成処理の概要について述べる。

ＣＬＳでは、動作記述を制御系動作とデータパス系動作に分離し、それら回路の特徴に応じた割付・最適化処理を行っている。また、割付・最適化処理を、デザインルール等に依存しない処理とデザインルールに依存した処理に分割して、処理負担の軽減を図っている。

以下に、各処理部の概要について述べる。

4．1 ＨＳＬ－ＦＸ入力部

合成対象のＨＳＬ－ＦＸ記述データを入力し、構文・意味解析、編集を行い、動作表現データとして論理合成用データベースに格納する。

本処理部は、入力言語の文法構造定義、出力生成規則の定義、及び変換処理をProlog系の文法記述形式ＤＣＧを用いて定義することで実現している。図5にＤＣＧによる文法定義の例をＢＮＦ(Backus Naur Form)の定義例とともに示す。

ＤＣＧを用いた理由は、以下の特徴による。

ｉ）文法の追加や修正が容易で、入力言語の文法変化に柔軟に対応可能である。

ii）Prologの記述が容易に盛り込め、さらにPrologで定義した手続きの利用が容易に行える等、文法定義から変換処理、出力生成規則まで簡単に記述可能である。

ただし、文法定義の読解性を優先したＤＣＧ記述による入力言語構文解析は、冗長な処理が多くなり、その処理速度が低下するという問題がある。そのため、読解性は若干損なわれるが、Prolog処理系の動作を考慮したＤＣＧ記述を行うことで処理速度の向上を図った。

```
<STATEMENT> ::= <COMPOSITE.STATEMENT>
              | <IF.STATEMENT>
              | <CASE.STATEMENT>
              | <CASEOF.STATEMENT>
              | <AT.STATEMENT>
              | <WHILE.STATEMENT>

<IF.STATEMENT> ::= <IF.COND.ST> THEN <STATEMENT.RFR>
                   <ELSE.STATE> <OFF.STATE> END - IF
<IF.COND.ST> ::= <IF.KEY> <SIMPLE.EXP>
<IF.KEY> ::= IF
<ELSE.STATE> ::=
               | ELSE <STATEMENT.RFR>
<OFF.STATE>  ::=
               | OFFSTATE <Z01X>
<Z01X> ::= 0|1|Z|X
```

（ａ）ＢＮＦ例

```
statement(X) --> composite_statement(X).
statement(X) --> if_statement(X).
statement(X) --> case_statement(X).
statement(X) --> caseof_statement(X).
statement(X) --> at_statement(X).
statement(X) --> while_statement(X).

if_statement(X) -->
    ["IF"], simple_exp(X1),
    ["THEN"], statement_rfr(X2),
    if_statement_s1(X3),
    {X=["IF ",X1,[10],"THEN",[10],X2,[10],X3]}.
if_statement_s1(X) -->
    ["END"], ["-"], ["IF"], {X="END-IF"}.
if_statement_s1(X) -->
    ["ELSE"], statement_rfr(X1),
      (["END"], ["-"], ["IF"],
          {X=["ELSE",[10],X1,[10],"END-IF"]};
       ["OFFSTATE"], off_state_value(X2),
       ["END"], ["-"], ["IF"],
          {X=["ELSE",[10],X1,[10],
              "OFFSTATE",[10],X2,[10],"END-IF"]}).
if_statement_s1(X) -->
    ["OFFSTATE"], off_state_value(X2),
    ["END"], ["-"], ["IF"],
    {X=["OFFSTATE",[10],X2,[10],"END-IF"]}.
off_state_value(X) --> bzx(1,X).
```

｛ ｝内：変換処理、出力生成規則

（ｂ）ＤＣＧ例

図5 文法定義例

4．2 機能ブロック割付部

論理合成用データベース中の動作表現データを入力し、機能ブロック間の接続関係を表す構造表現に変換する。図6に処理フローを示す。

機能ブロックとして、、通常のランダム回路用のものに加えて、データパス用のもの、ＦＳＭ（有限状態機械）用のもの（ＨＳＬ－ＦＸのオートマトン記述に対する割付対象）を設け、動作表現データより抽出した機能セットを構造表現に反映できる様にした。割付時には、同一タイミングで動作する資源を併合し、データパス部を組立てた後、ランダム回路部、データパス部、ＦＳＭ部の分類、割付ブロックの決定と転送関係の抽出を行う。さらに、それらの接続関係をデータ信号と制御信号に分類して出力する。図7（a）のＨＳＬ－ＦＸ記述に対する割付例を図7（b）に示す。

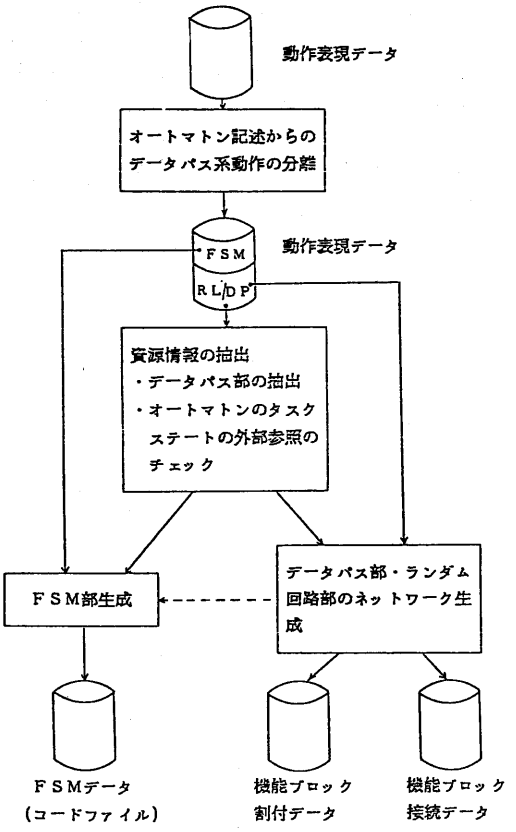
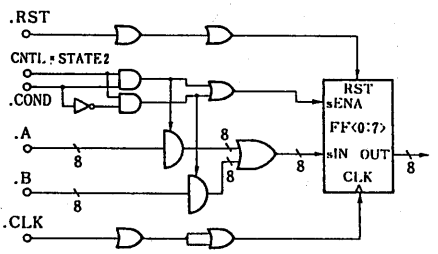


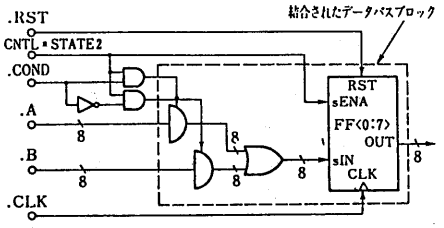
図6 機能ブロック割付部の処理フロー

```
        ⋮
REGISTER : FF〈0:7〉 ;
BEHAVIOR-SECTION ;
  BEGIN
    IF .RST THEN RESET ( FF ) END-IF ;
    IF CNTL*STATE2 THEN
      AT .CLK DO
        IF .COND THEN FF := .A ;
                 ELSE FF := .B ; END-IF ;
      END-DO ;
    END-IF ;
        ⋮
```

(a) HSL-FX記述



(b) 機能ブロック割付後



(c) 機能ブロック最適化後

図7 機能ブロック合成過程

4．3 機能ブロック最適化部

機能ブロック割付済のデータパス部、ＦＳＭ部、ランダム回路部に対し、デザインルールに依存しない最適化処理を行う。また、データパス部に関しては、各データパスに割付けられた機能ブロックのビット幅、接続関係よりそれらを結合して行き、一連のデータ信号を扱うデータパスモジュールを生成する。図7（ｂ）の割付結果に対する最適化例を図7（ｃ）に示す。

最適化処理は、ＦＳＭ部とデータパス部、ランダム回路部により異なる。ＦＳＭ部は、主に、積項数の低減と外部で参照されていないステート、タスクをエンコードしてレジスタ数の削減を行う。データパス部、ランダム回路部は、使用機能ブロック数、総ファンイン数の削減を目的として、冗長ブロック、固定値入力ブロック、未使用ブロック（入力信号数＝０／出力信号数

＝０）、スルーブロック（入力値＝出力値）の削除、ブロックの共有化等、これまでに提案されている最適化と同様の処理を行う。ただし、データパス部に関しては、可能なかぎり規則性を保持することで、レイアウトへの悪影響を防止している。

４．４　論理ブロック割付部

論理ブロック割付部では、合成対象のデータパス部、ランダム回路部の機能ブロック、接続データ、及びユーザ指定の論理ブロックライブラリを入力し、それらを論理ブロックとそれらの接続関係に変換する。

論理ブロックへの割付処理においては、図８に示すような、信号極性の調整、多入力ＡＮＤゲート等の展開、Ｆ／Ｆの分類、演算ブロックの展開等を行う。何通りかの割付が可能な場合は、レイアウトの高集積・高性能化を実現するため、より大きな機能を有する論理ブロックから優先的に割付けている。

４．５　論理ブロック最適化部

論理ブロックの最適化処理は、４．３節で述べた機能ブロックの最適化処理とほぼ同一であるが、デザインルールに依存した面積、遅延、消費電力等を考慮した最適化を行う。例えば、論理ブロックの駆動能力によるファンイン、ファンアウト数の調節、バッファの挿入、反転端子の利用、AND-OR系回路とNAND-NOR系回路間の相互変換等である。

また、データパスモジュール等の機能セット間にまたがる最適化処理を行わないことで、処理効率の向上を図った。

４．６　フロアプラン出力部

フロアプラン出力部では、割付済の論理ブロックとそれらの接続データ、及び論理ブロックライブラリを入力し、各データパスモジュール、ＦＳＭブロックの面積の見積りを行った後、データパスモジュール内の論理ブロック配置、合成対象全体の概略配置を行う。

データパスモジュール内の配置では、データ信号の流れに添って左側から右側へ、各論理ブロック間の総接続区間数（論理ブロック１つの通過が１区間）が最小となるよう配置する。

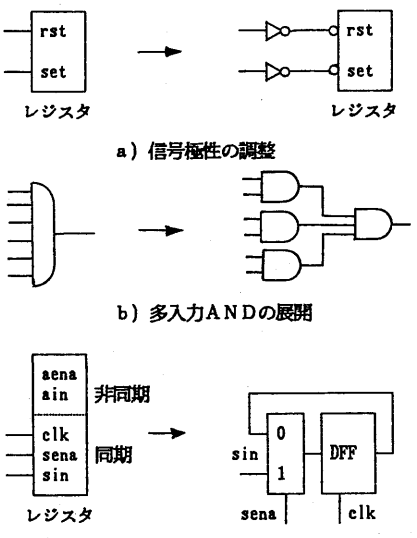


図８　論理ブロック割付部の割付例

合成対象全体の概略配置では、合成過程で識別・抽出されたブロックの種類、信号の流れ／種別より、制御信号は、上側から下側へ、データ信号は、左側から右側へと流れるような相対的な位置関係をもとめる。その後、その相対位置関係をもとに全体の形状が矩形となるように最終的な配置を決定する。

５．適用結果

本章では、種々の合成対象に対するＣＬＳの適用結果について述べる。

表１に各処理部の処理時間と合成した回路の諸元を、図９に回路の規模、種類と全処理時間の関係を示す。この結果より、数Ｋゲート規模の回路に対して、妥当な処理時間で合成可能であることが明らかになった。

回路規模に関しては、システム制約上は、数10Ｋゲート程度の回路も合成可能であることが確認された。

処理時間については、機能ブロック最適化処理の全処理時間に占める割合が高くなっている。これは当処理部で、データパスブロックのモジュール化、デザインルールに依存しない論理最小化処理等、合成対象全体に対する大局

表1　ＣＬＳの適用結果

コード		A	B	C	D	E	F	G	H	I
ＨＳＬ－ＦＸ記述量（行）		17	48	97	69	298	568	298	360	568
処理時間[1]（分）	機能ブロック割付	0.3	0.7	6.3	2.8	16.6	52.3	14.0	20.1	52.7
	機能ブロック最適化	0.3	2.7	10.4	2.5	32.5	132.4	50.0	36.2	274.9
	論理ブロック割付	0.2	0.3	0.5	0.2	1.4	4.3	0.7	1.2	2.7
	論理ブロック最適化	0.5	0.6	5.9	0.2	1.9	9.1	2.1	4.0	15.0
	フロアプラン出力	0.6	1.0	4.1	2.5	7.1	86.8	8.5	17.1	56.8
	計	1.9	5.3	27.2	8.2	59.5	284.9	75.3	78.6	402.1
合成結果[2]（ゲート数）	ランダム回路部	0	9	9	80	158	436	62	131	227
	データパス部	384	656	2646	4	0	0	1810	2392	5866
	ＦＳＭ部	12	83	186	66	290	646	290	357	646
	計	396	748	2841	150	448	1082	2162	2880	6739
回路の種類		データパス(16bits)系			制御系			データパス系＋制御系		

1) 1MIPS相当の計算機を使用
2) 2入力ＮＡＮＤ換算

的な処理を行っているためと考えられる。

一方、論理ブロック割付、同最適化の処理時間の割合は、低くなっている。この理由としては、

1) 本適用結果で使用した論理ライブラリには、複雑な複合ゲートのセルが存在しない。従って、割付、最適化を行う論理ライブラリ候補の競合が少なかった。

2) このことから、論理ブロック割付、最適化の処理では、AND-OR系回路とNAND-NOR系回路の相互変換等、否定論理の利用に対する処理が主体となり処理負担が軽くなった。

3) データパス部に対しては、モジュール毎に独立に処理を行うため、一度に取り扱うデータ量が少なくてすむ等であると考えられる。

さらに、機能ブロックにデータパス用のものを用意する等、束線表現をそのまま処理可能としているため、データパス系回路の処理時間が、同一回路規模の制御系回路の処理時間に比べ大幅に短縮された。

図１０に合成例（表１中のコードＣのレイアウト結果）を示す。

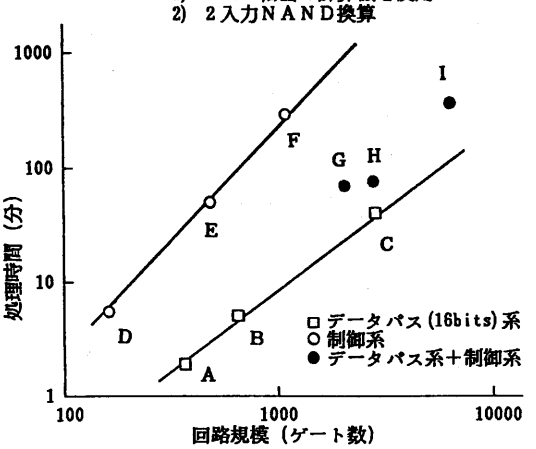


図９　　合成対象の回路規模と処理時間

６．おわりに

以上、ＣＬＳの開発方針、システム構成、合成処理概要、及びその適用結果について報告した。本システムは、チップ全体を機能記述レベルから、制御回路部、データパス部、ランダム回路部に分離し、各々の機能セットを中心に、専用の合成処理を行っている。また、合成過程における割付処理、最適化処理に、人手設計作業の特徴を取り入れることで、最適化・フロアプランの処理負担の軽減、及びチップ面積最小化を図っている。

なお、本システムは、現在、性能評価中であり、今後、その評価結果をもとに改良を重ねる必要がある。特に、ＶＬＳＩ設計で問題となる遅延に対する処理の改良が重要な課題である。これらの課題は、論理合成システム単独では解決困難であり、別途開発中のレイアウト自動生成システムと組み合わせて、作業を進めていく予定である。

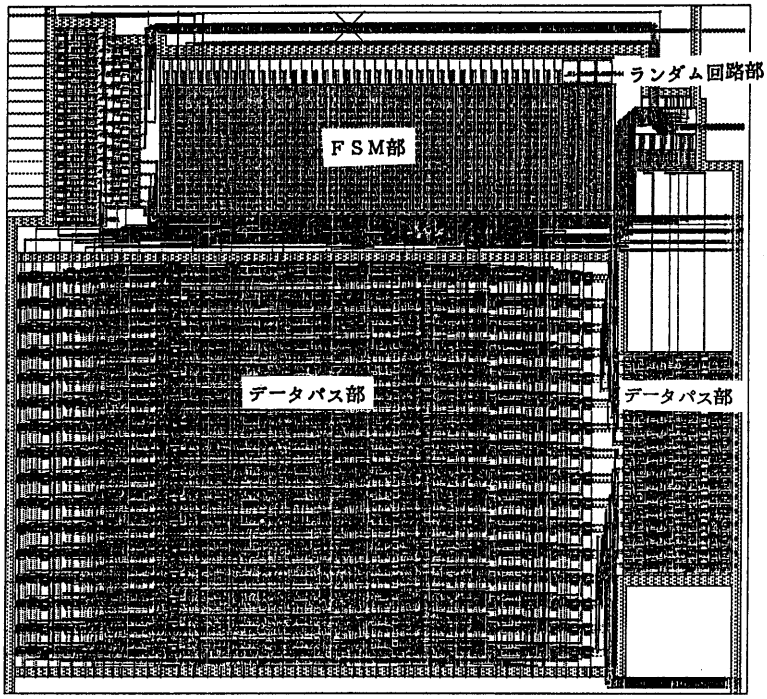


図１０　合成例

【参考文献】

[1] 今村他: 論理合成エキスパートシステム: ＣＬＳ　＝システム構成と入力データ処理部について＝, 情報処理学会第３６回全国大会予稿集, 1988.

[2] 北村他: 論理合成エキスパートシステム: ＣＬＳ ＝合成過程＝ , 情報処理学会第３６回全国大会予稿集, 1988.

[3] ＮＴＴ厚木電気通信研究所集積応用研究室: 拡張階層仕様記述言語ＨＳＬ－ＦＸ仕様書, 1986.

[4] Gregory, D., et al. :SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic. Proc. of 23rd DAC. 1986.

[5] Deman, H. et al. :Cathdral-II: A Silicon Compiler for Digital Signal Procesing, IEEE Design and Test of Comput., 1986.

[6] Jerraya, A., et al. :Principles of the SYCO Compiler, Proc. of 23rd DAC. 1986.

[7] Karatsu, O., et al.: An Automatic VLSI Synthesizer, Proc. of ISCAS 85. 1985.

[8] Darringer, J. A., et al.: LSS: A System for Production Logic Synthesis. IBM J. of Res. and Develop., Vol. 28. No. 5. 1984.

[9] Shahdad, M., et al. : VHSIC Hardware Description Language. IEEE Computer, Vol. 18. No. 2. 1985.

[10] Pereira, F. C. N. and Warren, D. H. D: Definite Clause Grammers for Language Analysis A Survey of the Formalism and Comparison with Augumented Transition Networks. Artificial Intelligence. Vol. 13. 1980.