時相論理型言語：Tokioによるアルゴリズム記述とCMOSゲートアレイの自動合成

藤田 昌宏＊　石曽根 誠　中村 宏　田中 英彦　元岡 達
東京大学 工学部
＊：現在、富士通研究所

Abstract

There have been few systems that assist function designs of hardware. Here we present a silicon compiler that accepts an algorithm description as an input and outputs circuits of CMOS gate arraies. Hardware description in algorithm level and state diagram level are written in Tokio: a logic programming language based upon Temporal Logic. Since Temporal Logic can describe temporal relations among variables, we can easily describe concurrency by Tokio, which is difficult to describe in Prolog. Moreover, as Tokio has a well founded mathematical model, we can make formal verification and synthesis much easilier. To date, a simulator for Tokio, which is installed on C-Prolog, is developed.

Descriptions in state diagram level ate automatically synthesized into logic circuits for CMOS gate arraies. The core part of the synthesis program is already developed on C-Prolog. We apply it to design Unification Processor for a parallel inference engine.

We here present a general overview of logic design assistance with Tokio and the details of the synthesis program.

1. はじめに

デバイス及び、実装技術の著しい進歩により、大規模・複雑なハードウェアを作成することができるようになってきている。しかし、従来のCAD技術は、主に実装に重点をおいているため、より高位レベルである、ハードウェア論理設計支援は、十分であるとは言えない。特にシステムレベルから一貫して、論理設計を支援し、実装設計用CADにつながるハードウェア論理設計支援システムいわゆるシリコン・コンパイラが強く望まれる。

このような論理設計支援システムには、以下のような機能が要求される。現在の論理設計支援は、Register Transfer レベルのハードウェア記述言語やゲート回路のシミュレーションによるものがほとんどであり、システムレベルの（ソフトウェアもハードウェアも含めたレベルの）ものを支援するものは非常に少い。従って、①システムレベルから一貫して記述できるような言語が必要となる。

ハードウェアシステムを設計する際、設計者は、まず処理アルゴリズムを考え、それをシミュレーションによって確かめる。従って、②使いよい設計環境（つまりハードウェア記述に対するよいプログラミング環境）が必要である。これは、次の2つに分けられる。②－1高速シミュレータ、②－2知的なデバッグ環境（デバッガ）

また、設計が進んでくると、設計者は単なるシミュレーションのみで、自分の設計に自信がもてるとは限らない。そこで、形式的な検証機能も必要になる。これには、設計記述が、数学的基礎のしっかりしたモデル上になされている必要がある。言いかえると数学的基礎のしっかりした言語で設計が記述されていなければならない。さらに、このような言語で記述することによって、設計記述の意味が明確になり、設計の自動合成もやりやすくなると考えられる。そこで、③数学的基礎のしっかりした言語が必要である。

さらに全体として、使いよい設計支援システムであることが必要であり、そのためには④高度なマンマシンインターフェイスが必要である。このためには、設計記述を言語で行なうことは、必ずしもベストであるとは限らない。しかし、④については、ワークステーション上において現在いろいろ研究されているため、ここでは考えないことにし、①、②、③を中心テーマとする。

まず言語については、数学的基礎をもったものとして、論理型言語を考える。論理型言語の代表として、Prolog があるが、Prolog が基礎とする（古典）述語論理には、時間の概念がなく、並列動作を記述できないため、ハードウェア記述には工夫がいる。これに対し、時間の概念を含んだ論理として、時相論理（Temporal Logic）［1、2］がある。そこでここでは、時相論理に基づくプログラミング言語（時相論理型言語）Tokio［3、4］を考え、ハードウェア記述に用いることにする。そして、もととなる時相論理の理論を基礎として、Tokioで記述された設計記述の検証や合成（つまり、reasoning ）を行うことを考える。

ここでは、1つのシリコン・コンパイラを考え、入力としてTokioによるアルゴリズム記述を受け取り、設計者と対話しながら設計を進めて行き、最終的にCMOSゲートアレイ用のゲート回路を出力するシステムを考える。出力をCMOSゲートアレイとしたのは、比較的集積度が高く、また実装設計用CADシステムが充実しているからである。

2. 時相論理型言語：Tokio

本章では、時相論理型言語Tokioについて簡単に述べ、次にTokioを中心としたハードウェア設計支援について説明する。Tokioの詳細については、文献［3、4］を参照されたい。

2.1 時相論理とTokio

時相論理（Temporal Logic）とは、通常の古典論理に、時間に関する記述ができるように幾つかの時相演算子（temporal operator ）を付け加えたものとして定義される。時相論理は離散時間上で定義され、各時相演算子を用いて各変数の各時刻での値の取り方を指定できる。付け加える演算子によって、Linear Time Temporal Logic（LTTL）［1］、Interval Temporal Logic（ITL）［2］等、色々な時相論理が存在するが、Tokioは、LTTLを拡張し、ITLの記述容易性を取り入れたものであり、LTTLへ容易に変換できる。LTTLは命題論理の範囲では決定可能であり、また論理自体の研究も進んでいる［1］。

従来、筆者らは、LTTLを用いてハードウェア同期部の仕様記述を行い、ゲート回路の検証や状態遷移表の自動合成等のシステムを開発してきた［5］。LTTLを用いれば、ハンドシェイク・シーケンス等の並列性は容易に記述できるが、一方、Pascal で、

```
begin
  P;
  Q
end ;
```

のような順序性の記述は簡単であるとは限らない。しかし、このような場合にも、PやQが実行中である時のみactiveであるような信号を用いれば、容易に記述できることが分かっている［5］。このような信号はインターバルと呼ばれるものと同じであり、ITLはこのインターバルを基礎とした時相論理である［2］。しかし、ITLは一般には決定不能であり、その他にも計算機処理しにくい面があるため、LTTLの拡張としてTokioを定義している［3］。

Prolog が古典論理に基礎をおいた論理型言語であるように、Tokioは、時相論理に基礎をおいた時相論理型言語である。時相論理は古典論理を含むため、必然的にTokioは Prolog を含み、Prolog を並列処理記述用に拡張したものと考えることができる。このため、Tokioを用いれば、システムレベルからゲート回路まで様々なレベルのハードウェア記述を容易に行うことができる。次章では、Tokioを中心としたハードウェア設計支援システムの構成について説明する。

3. 時相論理型言語Tokioを中心としたハードウェア設計支援システム

設計支援システムの構成を図1に示す。設計者はまず、設計対象の処理アルゴリズムの決定を行う。これはシステムレベルの記述として、Tokioを用いて記述され、シミュレータでの実行を通して、確認を行う。

このアルゴリズムの記述は、ハードウェアの物理的イメージとは必ずしも合わないため、設計者は、ハードウェアをより意識したものへと設計記述の改版を重ねていく。この際、処理イメージに合った段階的詳細化が行われ、階層的に設計記述がなされていく。

この設計記述の詳細化では、次の3つの支援がなされる。

① シミュレーション

② 2つのTokioの記述間の同一性の検証
③ Tokioの記述と物理実体（データパス）の情報からより詳しいTokio記述の合成

①については、C－Prolog上に作成されたTokioインタプリタがある［6］。しかし、実行スピートが遅いため、現在C言語を用いて高速なシミュレータを作成中である［4］。 また、②、③については、ハードウェアの同期部のみを扱うものについては、既に作成されているが［5］が、今後演算部も含めたハードウェア全体を対象としたものに拡張する予定である。

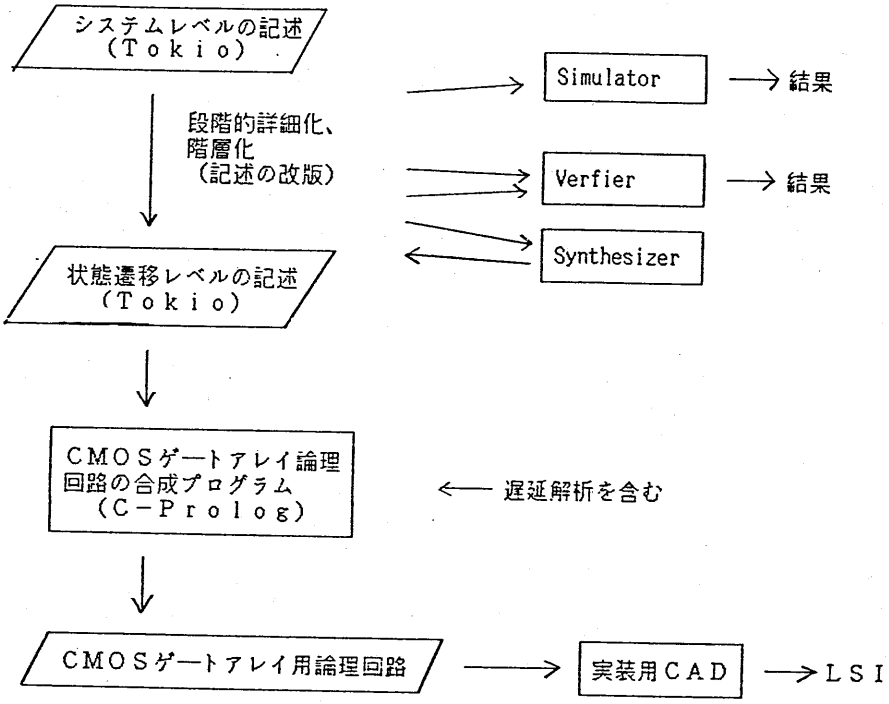


Figure 1 Hardware Design Assistant System with Tokio
- A Temporal Logic Based Language

以上の設計の詳細化は、状態遷移図レベル（例えば、ハードウェア記述言語ＤＤＬ［7］と同じレベル）まで行われる。状態遷移図レベルとなったTokioの記述は、状態遷移表へと展開され、ＣＭＯＳゲートアレイ論理回路の合成プログラムにかけられる。このプログラムは既に大部分完成しており、次章で詳しく説明する。この出力は、ＣＭＯＳゲートアレイ用論理回路であり、実装設計用ＣＡＤに送られ、ゲートアレイとしてＬＳＩになることになる。

以上のうち、現在完成しているのは、C－Prolog によるTokioシミュレータとＣＭＯＳゲートアレイ論理回路合成プログラムである。

4． ＣＭＯＳゲートアレイ論理回路の合成プログラム

図１の構成図では、状態遷移図レベルのTokio記述を入力として受け取ることになっているが、現在Tokioに対する支援は進んでいないので、ここでは、仮にＤＤＬ［7］の記述を入力とする。Tokioは２章でも説明したように、幾つかの連続した時刻列であるインターバルごとに記述されるが、そのインターバルの長さを全て１にしたものがＤＤＬと考えることができる。従って、Tokioの記述を詳細化して、状態遷移レベルとしたものがＤＤＬであると考えてよい。

本プログラムはＤＤＬの記述をＤＤＬトランスレータ［8］にかけ、その出力であるレジスタ転送表等を入力として受け取る。

ＤＤＬトランスレータの出力は、
・ターミナル転送表
・レジスタ（含メモリ）転送表
・状態遷移表
・ダミーターミナル（トランスレータが生成したターミナルで、各種転送・遷移の条件式になっている）の論理式の表
・クロス・リファレンス
からなる。転送表は転送先ターミナル・レジスタ毎に、転送範囲（何ビット目から何ビット目か、及びメモリの場合アドレス）、転送元、転送条件をまとめたもので、遷移表は、現在の状態、次の状態、遷移条件をまとめたものである。転送表の例を図２に示す。

MAR	(0:9)	10 BIT REGISTER
SINK RANGE	SOURCE	TRANSFER CONDITION
(0:9)	IAR(0:9)	ADS
	ADR(0:9)	DEC

Figure 2 Outputs of DDL Translator
(Register-Transfer Table)

4．1　CMOSゲートアレイとその論理設計

ゲートアレイとは、セミカスタムＬＳＩの一種で、シリコン・チップ上にゲートを配列し、ユーザの仕様に合わせて配線し、所定の機能を持つようにしたものである。ゲートアレイにはCMOSゲートアレイの他にもＴＴＬゲートアレイやＥＣＬゲートアレイなどがあるが、低消費電力・高集積度という特徴をもつCMOSゲートアレイが最も一般的であり、またその将来性も大きい。

CMOSゲートアレイでは、『ベーシック・セル』と呼ばれる基本ゲートが、チップ上に規則正しく配列されている。１つのベーシック・セルは数個のn-MOS・p-MOSトランジスタから成る。図３にベーシック・セルの等価回路を示す。この例では、１個のベーシック・セルはn-MOS・p-MOS各２個のトランジスタにより構成される。この他に、入出力用としてチップ周辺部に『Ｉ／Ｏセル』が設けられている。これらの基本ゲートをアルミ配線で接続することにより目的の回路を得る。

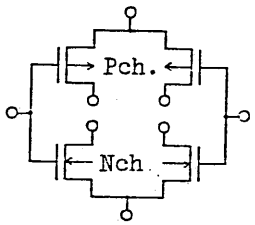


Figure 3 Equivalent Circuits of the Basic Cell

ベーシック・セルは配線してはじめて論理素子としての機能を持つ。そこでメーカでは、ＮＯＴやＮＡＮＤ、それにフリップ・フロップなど、論理回路設計に使われる各種の基本的な論理素子の配線をあらかじめ決めており、ライブラリに登録している。このライブラリに登録された論理素子の集まりをユニットセル・ファミリと呼び、ユーザはこのファミリ中の素子を使用して論理回路を設計する。

ゲートアレイの開発フローを図４に示す。前述したように、ユーザはメーカが提供するユニットセル・ファミリに属する論理素子のみを使用して回路を設計する。この回路について論理設計の検証を行い、設計が正しいことを確認した上でレイアウト設計を行う。ここで遅延時間解析などを行った後、チップを製造する。

このうち、論理設計の検証及びレイアウト設計用には、論理シミュレータ、自動配置配線プログラム、遅延時間解析プログラムなど、多くのＣＡＤシステムが存在し、それらを自動化、または人間の作業を支援している。従って、論理設計が自動化されれば、ハードウェアの仕様記述からチップ製造までのすべての段階が自動または半自動化され、いわゆる「シリコン・コンパイラ」に近づくことができる。

CMOSゲートアレイの論理設計は、前にも述べたように、最終的にはユニットセルを用いた回路を設計することである。その際注意すべき点として、

① ゲート数（ベーシック・セル数）が、使用するゲートアレイのゲート数以下であること。（実際には、配置・配線の都合上、余裕を持たせる）
② CMOSゲートアレイでは特に配線容量、入力容量による遅延が大きいので、遅延時間が、クロックなどにより決定される値を越えないこと。

などが挙げられる。

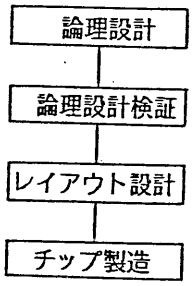


Figure 4 Flow of Designing Gate Arraies

4．2　合成処理の流れ

やるべきことは、ターミナル・レジスタ転送表や状態遷移表などのＤＤＬトランスレータ出力を読み込んで、CMOSゲートアレイ用の、ユニットセル・レベルの論理回路を合成するものであるが、その処理の際、行わなくてはならないことは以下の３つである。

① 回路の簡単化

ＤＤＬトランスレータの出力から得られる各種転送表などは、そのままでは冗長度が大きく、そのままユニットセルの回路に展開することは、ゲート数の増大、遅延時間の増加を招くことになる。また、ＣＭＯＳゲートアレイの論理回路の特徴（ＮＡＮＤ・ＮＯＲ中心）に合わせた回路を合成しないと、やはりゲート数、遅延時間の両面で不利である。従って回路の合成途中での回路の簡単化は不可欠である。

② ファンイン・ファンアウトの考慮

ユニットセルのファンインの大きさには上限があるため、それ以上の入力を持つ素子はいくつかのユニットセルに分割する必要がある。また、ファンアウトの大きさも、メーカで指定しているため、それを越えないような回路を合成しなくてはならない。

③ ゲート数・遅延時間の解析

使用するゲートアレイにより、ゲート数の上限は決定され、またクロックスピードなどにより、遅延の最大値も決定される。従ってこれらの基準が満たされないことがわかった場合には、論理分割・設計変更などを行う必要がある。なお、遅延時間は、正確にはチップ上に配置・配線しなければわからないが、それでも概略値を計算しておくことは、開発効率の面から重要である。

このような複雑な処理を行うため、直接ユニットセル・レベルの回路を合成するのは効率的ではない。そこで「マクロユニット」という仮想的なユニットを導入し、まずマクロユニット・レベルの回路を合成し、そこで各種の処理を行った後、ユニットセルの回路に展開するという方法をとる。この、マクロユニットという仮想的な論理素子は、次のような特徴を持つ。

① ファンイン無制限
② ファンアウト無限大
③ Ｎビット入力－Ｎビット出力

これは図５に示すように、１個のユニットで複数ビット入出力を可能にするものである。

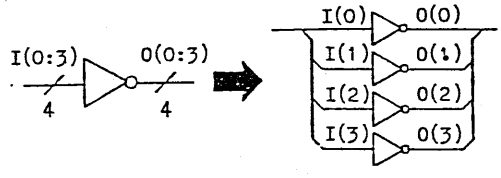


Figure 5 N-bit Input and N-bit Output

④ ユニットセルより機能が高度（より抽象的）

例えばＡＮＤ・ＯＲといった基本演算に加えて加減算、大小比較、レジスタといったより高度な機能を持つユニットがある。

このような特徴を持つマクロユニット・レベルの回路で簡単化やゲート数・遅延時間解析を行うことは、ユニットセル・レベルで行う時に比べ、次のような利点がある。

① ユニットセル・レベルの回路に比べ、データ量が少ないため扱いやすく、処理の高速化を図ることができる。
② 機能の抽象度が高いため、ある程度ではあるがテクノロジー独立となる。このことはつまり、マクロユニット・レベル以降の処理さえ変更すれば、ＣＭＯＳゲートアレイ以外にもこのシステムが適用できるということである。

従って処理の方針は、ＤＤＬトランスレータの出力を読み込み、それをマクロユニットに展開し、種々の処理を施した後ユニットセルに展開するということである。簡単な例で、レジスタＲへの転送部分がどのように展開されるかを図６に示す。

４．３ 処理の概要

前節で述べた方針に従い、処理段階を６つのフェーズに分割した（表１）。

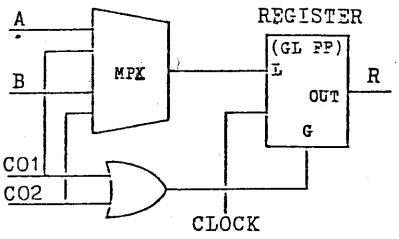
Ｐhase１はＤＤＬトランスレータの出力である、ターミナル・レジスタ転送表、状態遷移表、遷移・転送条件の論理式をまとめた表を読み取り、Ｐrolog の記述に変換する。

```
        .
        .
¦* CO1 *¦ R <- A.,
¦* CO2 *¦ R <- B.,
        .
        .
```

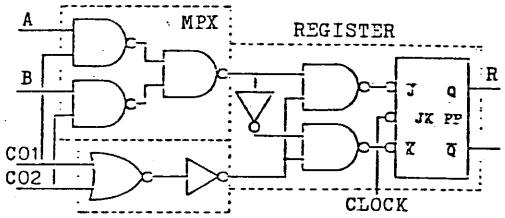
(a) Descriptions in DDL

R (0:0) 1 BIT REGISTER		
SINK-RANGE	SOURCE	TRANSFER CONDITION
(0:0)	A	CO1
	B	CO2

(b) Outputs of Translator



(c) Expansion to Macro Units



(d) Expansion to Unit Cells

Figure 6 Expansion of Register R

Phase2は、Phase1で得られたデータから、マクロユニットを用いた回路への展開を行う。同時に、第1次の簡単化を行う。そしてクロス・リファレンスを作成する。簡単化については次節でまとめて述べることにする。この段階では、展開結果はAND・OR・REGISTERなどの一般的な機能を使っており、特定のテクノロジーには依存していない。

Phase3はマクロユニット・レベルでの簡単化である。詳細は後に述べるが、ここでCMOSゲートアレイ向きの回路に変換される。

Phase4はゲート数解析及び遅延時間の解析を行なう。マクロユニットは前述の通りファンアウト無限大として扱ってきたが、実際にユニットセルに展開される時のことを考えて、ここではファンアウト解析と（必要ならば）バッファの挿入も同時に行う。なお、遅延時間はチップ上にレイアウトしてみなければ正確な値がわからないこと、またこの後のPhase5でも簡単化を行ない、ゲート数の変動があることから、ここで求めた値はともに概略値である。

Phase5ではマクロユニットをユニットセルに展開する。このとき第3次簡単化を合わせて行う。

Phase6は、得られたユニットセルの回路のデータをシミュレータなどの各種ツール（厳密な遅延解析など）に入力するためのデータ変換である。

Phase1　：DDLトランスレータの出力結果をProlog の記述に変換
Phase2　：マクロユニットへの展開、第1次簡単化、クロス・リファレンス作成
Phase3　：第2次簡単化
Phase4　：ゲート数、遅延時間解析と、それにともなう設計変更
Phase5　：ユニットセルへの展開、第3次簡単化
Phase6　：展開結果のシミュレータなどへのデータ変換

Table 2 Macro Units

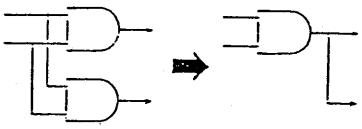
・　簡単化

前節で簡単化には第1次・第2次・第3次と3種類あることを説明したが、これらはそれぞれ論理式レベル、マクロユニット・レベル、ユニットセル・レベルでの簡単化に対応している。各簡単化の内容は、

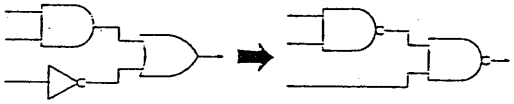
第1次簡単化　：　似たような論理式から共通部分を取り出してきてまとめる（図7（a））
第2次簡単化　：　ここがメインの簡単化であり、重複するユニットを削除（図7（b））、その後ルールによる置換を行なって最適化する。（図7（c））。ここで、CMOSゲートアレイの特徴に従った簡単化、つまり基本ゲートをNAND／NORにする、等の処理を行う。
第3次簡単化　：　定数とデータとの演算器の最適化（図7（d））。
である。

```
T1 = A & B & C & D          COM = A & B & C
T2 = A & B & C & E           T1 = COM & D
                             T2 = COM & E
```

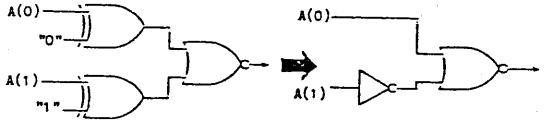
(a) Sharing Terms



(b) Eliminations of Duplications



(c) Simplification by Rules



(d) Optimization of Constants

Figure 7 Simplification

4．4　Prolog による実装

このシステムは、Edinburgh大学で開発された、UNIX上のC－Prolog［9］により記述されている。以下、その実装について上で述べた処理のフ

ェーズに従って解説する。

・ Phase1

Phase1はDDLトランスレータの出力をPrologの記述に変換するのだが、その変換形式をまず決定しておく必要がある。まずネット名であるが、これは転送条件等のように1ビットのものやデータ・バスのように複数ビットのものがある。そこで

［ネット名］ ：1ビットのネット。

例）［COND1］

［ネット名，［ビット］］，

［ネット名，［ビット1，ビット2］］ ：複数ビットのネット。ビット1，2は範囲を示す。

例）［DATA，［0，7］］はネットDATAの第0ビットから第7ビットまでの8ビットを表す。DDLの表記では、DATA（0：7）に相当する。

また、論理式については、S式を採用することにして、

A AND （B OR C） は、

［and，［A］，［or，［B］，［C］］］

と表す。

これらを使用して記述するが、変換すべきデータは、

・ターミナルの転送表

・レジスタの転送表

・状態遷移表

・転送・遷移その他の条件（ダミーターミナルと呼ばれる）の論理式

である。これらを述語terminal_trans，register_trans，state_trans，dummytermにそれぞれ格納している。例えば、terminal_transは、

```
terminal_trans(Terminal_net,
       [[Sink_range1,[Source1,Transfer_condition1],
                     [Source2,Transfer_condetion2],...],
        [Sink_range2,Source_and_condition_list1,
                     Source_and_condition_list2,...],...
```

という形式で、例えば図8のようなターミナルの転送表を変換して、

```
terminal_trans(['TERM',[0,15]],
       [[[0,15],[['ADR',[0,15]],['COND1']],
                [['MAR',[0,15]],['COND2']]],
        [[0,7], [['DAT',[0,7]],['COND3']],
                [[add,['DAT',[0,7]],[b,1]],['COND4']]]])
```

というデータを作成し、ダミーターミナルは

```
dummyterm(Dummyterminal_net,Expression).
```

という形式で図9の表は、

```
dummyterm(['CO0001'],[and,['MAIN'],
                ['CO0002'],['ADR',[23]]]).
```

と変換される。

実は、すべての情報がDDLトランスレータの出力に含まれているわけではない。欠けている情報とは、

・STORAGE宣言されている、メモリの情報。これは、トランスレータ出力では普通のレジスタと見なされる。

・OPERATOR宣言されている、マクロ演算子。トランスレータ出力ではこのマクロは展開されていない。

従って、この2つについては、もとのDDLの記述を参照して、人間がデータを作成することにする。ただし、人間が作成するのは必要最小限のデータでよく、残りはそれから自動生成するので、この部分の自動化も可能である。

TERM	(0:15)	16 BIT TERMINAL
SINK RANGE	SOURCE	TRANSFER CONDITION
(0:15)	ADR(0:15)	COND1
	MAR(0:15)	COND2
(0:7)	DAT(0:7)	COND3
	DAT(0:7) + 1	COND4

Figure 8 Terminal-Transfer Table

CO0001	1 BIT DUMMY TERMINAL
MAIN & CO0002 & ADR(23)	

Figure 9 Dummy Terminal Table

・ Phase2

マクロユニットの記述形式は、

macro （［種別 情報］，ID番号，個数，入出力リスト）．

であり、

種別 ： ユニットの機能、例えばand，or，reg （register）等を表す。

情報 ： 種別に付随する情報、例えば入力数や、加減算器等ではデータ幅等を表す。

ID番号 ： ユニット固有の番号で、ユニットはこの番号により区別される。

個数 ： Nビット入力－Nビット出力を許すた

め、1つのユニットのように扱っても、実際は、複数個あることがある。その個数。

入出力リスト　：　ユニットの入出力のネット名。普通は、

［入力，出力］　または　［入力リスト，出力］

だが、レジスタのように、

［入力，出力，ゲート，クロック］

という形を持つものもある。

・仮想

through　：入力と出力をそのままつなぐ

cat　：連結（例えば８ビット幅のネットと４ビット幅のネットを連結して１２ビットのネットにするなど）

・論理

and，or，nand，nor，eor（exclusive-or），enor（exclusive-nor），inv（not）

・算術　add，sub

・比較　eq，ne，ge，gt，le，lt

・外部入出力

inbuf　：外部入力

outbuf　：外部出力

clkbuf　：クロック用外部入力

・その他

reg　：レジスタ

memory　：外部メモリ

dff　：Ｄ－フリップ・フロップ

mpx　：マルチプレクサ

Table 1 Processing Phases

表２に、今回使用したマクロユニットのリストを示す。このうち、memoryは、本来外部メモリなのだが、テクノロジー独立、機能の抽象度を上げるという方針にしたがって、マクロユニットとして扱うことにした。

・　展開の方法

展開には、細かく分類すると、以下の様なものがある。

・レジスタの展開

・メモリの展開

・ターミナル転送部分の展開

・状態遷移部分の展開

・ダミーターミナル（各種条件の論理式）の展開

・メモリのＲ／Ｗ（読出し・書込み）及びアドレス生成部分の展開

これらを順次行うことにより展開が完了する。

レジスタ・メモリの展開時には、同時にネット名を新しく付けなくてはならない。ターミナルなどは、ターミナル名をそのままネット名にすればよいが、レジスタなどは、入力・出力にそれぞれ異なるネット名を付けなくてはならないからである。そして、そのデータを格納しておき、他の展開時にレジスタを参照するときには、そのネット名を使うようにする。

また、メモリをアクセスする場合には、そのアドレスと、書込みか読出しかの情報を格納しておき、最後のＲ／Ｗ、アドレス生成部分の展開で、まとめて展開する。

実際に展開を行う述語は、

develop（論理式，出力ネット）

というもので、例えば、

```
develop（［and，［inv，［net1］］，
          ［net2］］，［out］）
```

を実行すると下のような展開結果が出力される。

```
macro([inv,1],1,1,[[net1],[n1]]).
macro([and,2],2,1,[[[n1],[net2]],[out]]).
```

・　第１次簡単化

第１次簡単化は、論理式レベルの簡単化で、似ている論理式の共通部分をまとめる、というものである。そのアルゴリズムは、

・論理式を１つ読む

・別に格納しておいた、１つ前の式までの共通部分を取り出し、今読んだ式との間で新たな共通部分を取る

if　新たな共通部分が無いか少ない　then

　・ストアしてあった共通部分及び式をマクロユニットへ展開する

　・今読んだ式そのものを共通部分の新たな初期値として、格納する

else

　・新しい共通部分、及びその式自身をデータとして格納する

・上記を式がなくなるまで繰返し、なくなった時点で、

if　共通部分及び式がまだ展開されずに格納されている　then

　・それらをマクロユニットへ展開する

というものである。

・　クロス・リファレンス

マクロユニット同士の接続関係を表すため、クロス・リファレンスを作成する。その形式は、

xref（ネット名，そのネットに入力が接続されているユニットのＩＤ番号のリスト）

である。

・　Phase3

第２次簡単化は３段階ある簡単化の中で、メインとなるものである。それは、さらにいくつかの処理に分けることができる。それを以下に示す。

①　同じ機能及び入力を持ち、出力のネットのみが異なるユニットの統一（図１０）
②　全く同じ入力が複数ある、ＡＮＤ・ＯＲゲートなどの簡単化（図１１）
③　ソースは同じだが条件の異なる組があるマルチプレクサゲートの簡単化（図１２）
④　置換ルールによる簡単化

④については後で詳しく述べる。

また、簡単化の他に、実際にはユニットセルに展開されないマクロユニットであるthrough 、cat の消去もこのPhase3で行う。

・　マクロユニットの消去・変更・追加

簡単化では、当然マクロユニットの消去・変更・追加といった処理が必要になる。この処理は、当然クロス・リファレンスの変更をも意味する。そこで、modify_macro という述語を作り、ユニットの消去・変更・追加は、全てこの述語を呼ぶことにより、誤処理を防止し、ユニットのデータの格納方式の変更等が起きたときの修正も最小限で押えられるようにした。この機能は、

modify_macro （delete, ＩＤ）

ＩＤ番号ＩＤを持つユニットの除去とクロス・リファレンス修正

modify_macro （append, Func , ＩNOUT）

機能（ＡＮＤ・ＯＲなど）Func 及び入出力リストＩＮＯＵＴを持つユニットの追加とクロス・リファレンス修正

modify_macro （subst , ＩＤ, NewFunc ）

ＩＤ番号ＩＤを持つユニットの機能のみをNewFunc に変更

である。

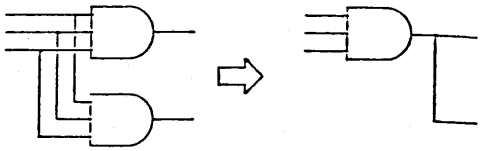


Figure 10 Simplification of Gates Having Same Inputs

Figure 11 Simplification of Gates Having Duplicate Inputs

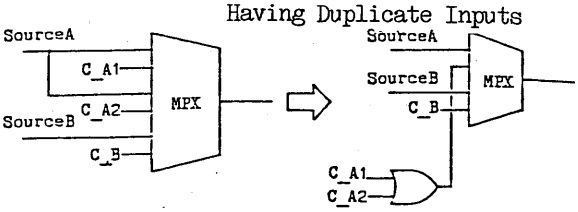


Figure 12 Simplification of Ports of Multiplexer

・　置換ルールによる簡単化

置換ルールによる簡単化とは、ルールが適用できる回路パターンを見付け、ルールを適用し、ユニットを置き換えることによって簡単化を行うものである。これにより、

・多入力のＡＮＤ・ＯＲゲートなどの分解
・ＮＯＴ－ＮＯＴ、ＡＮＤ－ＯＲゲートなどの最適化

を行う。

この処理は、大きく次の２つに分けられる。

・ルールに適合する回路パターンを探す
・ルールを適用して置換を行う

これをインプリメントした述語optimizeの定義を下に示す。

```
optimize(macro(F,ID,B,IO)):-
        match(N,STAT,ID),
        substitution(N,ID,Info),
        (
                STAT == opt,
                count_info(Info,M),
                M < 0
        ;
                STAT \== opt
        ),
        exec_info(Info),
        !.
```

使われている述語について説明を加えると、

match （N，STAT，ID）

ＩＤ番号ＩＤを持つユニットがルールのパターンにマッチすると、ルールの番号Ｎとその種類ＳＴＡＴを返す

substitution（N，ＩＤ，Ｉnfo ）

ＩＤ番号ＩＤを持つユニットにＮ番のルールを適用し、置換用のデータＩnfo を返す。（実際に置換するわけではない）

count_info（Ｉnfo ，Ｍ）

データＩnfo を実行したときのゲート数変化量Ｍを返す。

exec_info（Ｉnfo ）

データＩnfo に従って実際に置換を実行する。

従ってこの述語の動作は、「あるユニットに対し、パターンがマッチするようなルールがあり、それが最適化用のものでない（ＳＴＡＴ＝＝　opt ）ならばそのまま置換を実行、最適化用のものなら、ケート数が減少（Ｍ＜０）する時のみ置換を実行する」というものである。

この動作を具体的な例で示す。図１３が例題の回路とそのＰrolog 表現である。

ここで、

```
optimize(macro([or,4],1,1,[[[n1],[n2],
                           [n3],[n4]],[o1]])).
```

を実行すると、あるパターンにマッチする（パターンの詳細は省略）と、次にsubstitutionが実行され、Ｉnfo として、データ

```
[delete(1),
append(nand,[[[n10],[n11],[c1],[n5]],[o1]]),
append(inv,[[n1],[n10]]),delete(3),
append(nand,[[[b1],[b2]],[n11]]),delete(4)]
```

が返ってくる。このルールは最適化用のもので、ｃount_infoによりゲート数の変化が計算され、この場合Ｍ＝－２となり、exec_infoが実行される。その実行結果と、対応する回路を図１４に示す。

・　Phase4

Phase4では、まず最初にファンアウト解析を行う。これは、

・ユニットセルのファンアウトには、最大値があるため、それを越えたマクロユニットは、ユニットセルに展開できない。

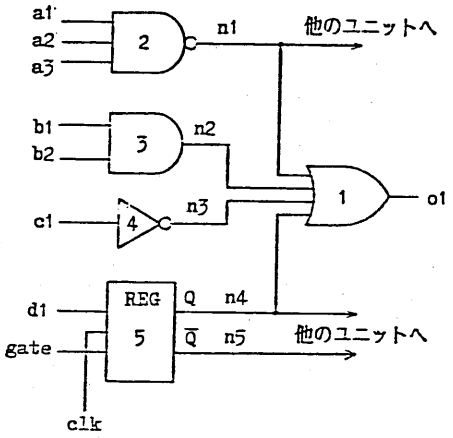
・ＣＭＯＳゲートアレイの遅延は、ファンアウトに大きく依存している。従ってファンアウトが最大値以下であっても、（ゲート数の増大があっても）バッファを挿入してファンアウトを押えたほうが、遅延時間の面から有利なことがある。

という２つの理由による。

処理方法は、

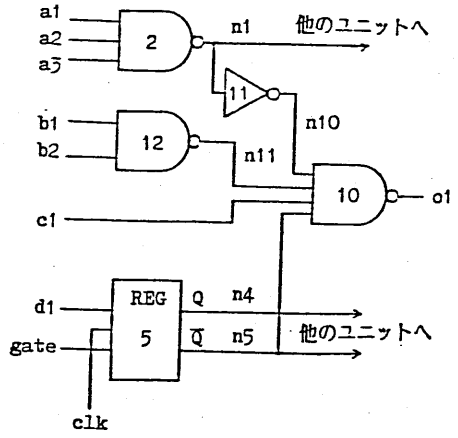
・各ユニットについて、出力につながるユニットを調べ、ファンアウトがおおきすぎる場合には、出力につながるユニットを、その入力容量がほぼ等しいような適当な数のグループに分け、それぞれに対しバッファを挿入する。

というものである。




```
macro([or,4],1,1,[[[n1],[n2],[n3],[n4]],[o1]]).
macro([nand,3],2,1,[[[a1],[a2],[a3]],[n1]]).
macro([and,2],3,1,[[[b1],[b2]],[n2]]).
macro([inv,1],4,1,[[c1],[n3]]).
macro([reg,1],5,1,[[d1],[[n4],[n5]],[gate],[clk]]).
```

Figure 13 An Example




```
macro([nand,3],2,1,[[[a1],[a2],[a3]],[n1]]).
macro([reg,1],5,1,[[d1],[[n4],[n5]],[gate],[clk]]).
macro([nand,4],10,1,[[[n10],[n11],[c1],[n5]],[o1]]).
macro([inv,1],11,1,[[n1],[n10]]).
macro([nand,2],12,1,[[[b1],[b2]],[n11]]).
```

Figure 14 Results

・　ゲート数解析

ゲート数解析は、個々のマクロユニットごとに、

そのユニットのユニットセルへの展開が何ゲートに相当するかを調べ、その和を求める。

・　遅延時間解析

遅延時間は、ユニットそのものの遅延と、配線及び入力容量による遅延の和であるため、正確な遅延時間はレイアウト設計時にしかわからないが、概略値を知るために、メーカが近似式を発表しているので、ここではその式を使用して遅延時間の解析を行っている。

・　Phase5、6

Phase5、6は現在実装されていない。しかし、これらの処理は単なる展開であるため、単純である。また、Phase4までの段階でもゲート数、遅延の概略値は分かる。

4．5　合成例：ユニファイ・プロセッサ（UP）

本合成システムの例題として、現在開発中の高並列推論エンジンPIE[10]の、ユニファイ・プロセッサ（UP）を使用し、これをCMOSゲートアレイに回路に変換する試みを行った。

現在PIEに実装されているUPは、人手で設計されたものであり、その回路規模はTTL・IC約500個程度のものである。内部レジスタを17個（計 356ビット）持ち、マイクロプログラム制御を用いている。例題では、このUPの動作をDDLで記述し、これをDDLトランスレータで変換したデータを用いた。このDDLで記述されたUPのソースは、約1000行である。

・　合成結果

○　処理時間

表3に示すのは、各Phaseの処理時間である。なお、この処理時間は、VAX－11／ 730上で測定したCPU時間である。

Phase1……5時間30分
Phase2
　展開……40分
　クロス・リファレンス……11時間30分
Phase3……74時間30分
Phase4
　ファンアウト解析……3時間30分
　ゲート数解析……10分
　遅延時間解析……2時間
合計……92時間

表3　処理時間

○　マクロユニットへの展開結果

表4は、Phase2終了後の、マクロユニットへの展開結果である。

	マクロユニット数	ベーシック・セル数
状態遷移部	43	124
レジスタ部	19	4752
メモリ部	8	0
メモリ・アドレス部	23	1380
レジスタ転送部	344	7646
ターミナル転送部	122	5041
ダミーターミナル部	1796	7600
合計	2355	26543

Table 4 Expansion Results to Macro Units

○　簡単化の効果

・第1次簡単化

第1次簡単化は、論理式の共通部分をまとめる、というもので、この簡単化によりゲート数で1646、全体の約 6%が減少した。

・第2次簡単化

第2次簡単化はさらに細かく

Simp 1　同じ機能・入力を持ち、出力のネットのみ異なるユニットの統一

Simp 2　同じ入力が複数あるAND・ORゲートなどの簡単化及びソースが同じで条件の異なる組のあるマルチプレクサの簡単化

Simp 3　置き換えルールによる簡単化

に分けることができる。表5はこれによるゲート数の変化を示したものである。

	ゲート数	変化	所要時間
最初	26543		
Simp 1	19666	－6877	6時間30分
Simp 2	17467	－2199	1時間
Simp 3	16604	－863	17時間40分
Simp 3	16405	－199	8時間

Table 5 Effects of Simplifications

合計で 10000ゲート以上減少した。これは全体の約40%である。なお、Simp 3を2回実行しているのは、最初の簡単化で変化した所に対し、新たにルールが適用できる可能性があるためである。2回で終了したのは、ゲート数の変化が飽和してきたことと実行時間の増加を押えるためである。

5. おわりに

時相論理型言語Tokioを中心としたハードウェア論理設計支援とCMOSゲートアレイの自動合成について述べた。Tokioの支援については、C-Prolog によるインタプリタしかないが、現在C言語等を用いて高速の処理系を作成中である。また、検証・合成についても検討を進めている。

DDLからのCMOSゲートアレイの自動合成については、基本的なところはC-Prolog を用いて完成している。 20000ゲート程度のものが合成できることが分かっており、また、処理スピードも大型計算機を用いることで、数時間で済む。

今後はTokioの支援を強化し、シリコン・コンパイラとして完成させていきたい。

6. 参考文献

[1] Z. Manna, A. Pnueli : Verification of concurret programs part1 : the temporal framework, Stanford univ. rep.STAN-CS-81- 836, 1981

[2] B. Moszkowski : Reasoning about digital circuits , Stanford univ. rep.STAN-CS-83- 970, 1983

[3] 青柳、藤田、河野、元岡：Tokioをかける言語、Logic Programming Conference '85

[4] 河野、藤田、青柳、田中：時相論理型言語Tokioの実装、i.b.i.d.

[5] 藤田：Logic design assistance with temporal logic、東京大学学位論文、1984

[6] 藤田、河野、田中、元岡：時相論理に基づくプログラミング言語Tempuraのインタプリタ、電子通信学会、電子計算機研究会資料、EC84-63, 1985

[7] J. R. Juley, D. L. Dietmeyer : A digital system design language (DDL), IEEE trans.computer , vol.C-17, no.9, 1968

[8] J. R. Duley, D. L. Dietmeyer : Translation of a DDL digital system specification to boolean equation , IEEE trans.computer , vol.C-18, no.4, 1969

[9] F. Pereira : C-Prolog users manual version 1.5, Ed CAD Edinburgh univ., 1984

[10] 後藤、相田、丸山、湯原、田中、元岡：高並列推論エンジンPIEについて、Logic Programming Conference '83