プログラミング言語解析用 VLSI チップの設計と評価†

平山 正治††

本稿では，プログラミング言語の解析処理を実行する VLSI チップの実現方式について述べる．この場合，コンパイラ等のソフトウェアで用いられている解析モデルをそのまま適用するのではなく，ハードウェア規模，解析処理の実行速度，複数のプログラミング言語への適応性等の観点から，VLSI チップとして最適な言語解析アーキテクチャの検討を行う必要がある．我々は，プログラミング言語 Pascal を対象とする字句解析チップと構文解析チップにおける言語解析アーキテクチャを設計するとともに，これらのアーキテクチャに関して，ソフトウェアによる実現方式と比べたハードウェア規模や実行性能等の評価を行った．この結果，プログラミング言語の解析処理に適するように拡張された PLA と記憶要素を主体とする簡潔な VLSI アーキテクチャによって，言語の構文を直接入力するだけで言語解析用 VLSI チップが容易に実現できることが示された．また，このチップは，現在の技術水準でも十分に1チップ化できるハードウェア規模であり，実行性能の面でもソフトウェアよりも2桁以上高速に解析処理を実現できることが示された．

1. まえがき

プログラミング言語を対象とする字句解析と構文解析の解析法は，オートマトンや言語理論との関連で理論的に良く研究されてきた分野であり，その成果は実際のコンパイラやインタプリタの開発等に有効に生かされてきた[1),2)]．一方，近年の VLSI デバイス技術と CAD 技術の急速な発展によって，プログラミング言語の解析処理のような大規模ソフトウェアを専用の VLSI チップによって実現することの可能性が高まってきている[3)]．

実際，このような言語解析処理をハードウェア化，VLSI 化する研究として，正規言語を解析する VLSI チップに関する研究[4)]，複数の μ プロセッサを用いたコンパイラの実現方式に関する研究[5)]，高級言語の直接実行マシンにおける字句解析プロセッサの研究[6)]等が報告されている．また，我々も，プログラミング言語 Pascal を対象として，複数の VLSI チップでコンパイル処理をパイプライン的に実行するコンパイラ・マシンについての検討[7)]や，これに用いられる構文解析用 VLSI チップの設計と評価[8)]を行ってきた．これらの研究でも指摘されているように，コンパイラ等のソフトウェアで用いられている解析モデルをそのまま適用して，言語解析処理を VLSI 化するのではなく，

(1) ハードウェア規模，
(2) 解析処理の実行速度，
(3) 複数のプログラミング言語への適応性

等の観点から，VLSI チップとして最適な言語解析アーキテクチャの検討を行う必要がある．

本稿では，Pascal を対象とする字句解析チップと構文解析チップの言語解析アーキテクチャについて述べるとともに，これらのアーキテクチャに関して，ソフトウェアによる実現方式と比べたハードウェア規模や実行性能等の評価結果について報告する．

2. 言語解析アーキテクチャ

プログラミング言語の解析処理の具体的な例として，Pascal コンパイラ[9),10)]における字句解析部と構文解析部を想定する．すなわち，字句解析部は，ASCII コードの列として表される Pascal のソース・プログラムを入力し，これを Pascal の文法で定義されている予約語 (36 語)，演算子，句切り記号等の特殊記号 (21 種)，および，ユーザ定義の名前と定数 (整数，実数，文字列の3種) からなるプログラムの言語要素を識別し，これらの言語要素を1または2バイトのトークンに変換して構文解析部に出力する．このトークンの形式を**図1**に示す．

構文解析部は，字句解析部で生成されたトークンを入力し，Pascal の文法に適合しているかどうかのチェックを行い，正しいプログラムの場合にはその構文に応じた処理を促すための1から3バイトのコマンド (意味解析コマンド) を出力する．すなわち，プログラムの宣言部に対しては，定数，データタイプ，変数，関数，手続き等のプログラム・オブジェクトを各テーブルに登録するためのコマンド，また，プログラムの実行部に対しては，for，repeat 等のプログラム制御構造や数式の構造に対応したオブジェクト・コードを

† Design and Evaluation of VLSI Chips for Programming Language Analysis by MASAHARU HIRAYAMA (Central Research Laboratory, Mitsubishi Electric Corporation).
†† 三菱電機(株)中央研究所

生成するためのコマンドを出力する．

以下では，これらの解析処理を2個の専用 VLSI チップ，すなわち，字句解析チップと構文解析チップによってパイプライン的に実行する方式を想定し，これら VLSI チップのアーキテクチャについて述べる．

2.1 字句解析アーキテクチャ

（1） 基本方式

プログラミング言語の解析手法に関しては，従来からの研究成果として，言語の性質に応じた各種の解析モデルが作られている．ここで対象とする Pascal の字句解析においては，言語要素が正規文法によって記述されるため，有限オートマトン・モデルが適用できる．すなわち，Pascal ソース・プログラムの文字コードを1文字ずつ入力し，この文字コードと状態レジスタの内容に応じて，適切なトークンと次の状態を生成する1個の PLA (Programmable Logic Array) によって，字句解析ハードウェアを容易に実現できる．

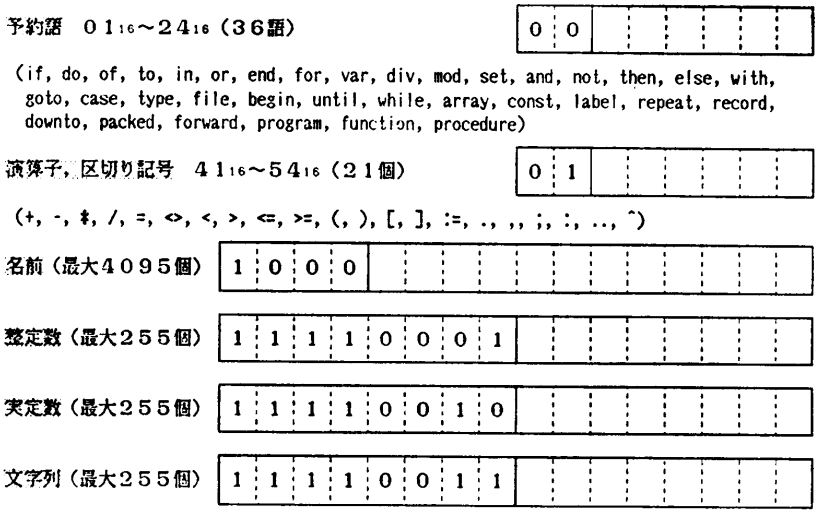

図1 トークンの形式
Fig. 1 Formats of tokens.

しかし，このような有限状態マシン・モデルによる字句解析ハードウェアでは，通常，入力1文字ごとに1回の状態遷移が行われるため，後述するように状態数，遷移数が多くなり，結果としてハードウェア規模を大きくしてしまうことになる．また，これに続く構文解析では基本的な動作として1言語要素ごとに1回の状態遷移を行うため，1文字に1回ずつ状態遷移する字句解析との間で処理時間の格差が生じる可能性がある．以上のような問題の解決策として，本字句解析チップでは最大4文字の文字コードを一度に入力して解析するアーキテクチャを採用している．この4文字一括入力機能の具体的な実現方式や評価については後述する．

一方，字句解析処理として，ユーザ定義の名前や各種定数（整数，実数，文字列）を識別するだけでなく，同じ名前に対しては同一の識別コードを生成する必要がある．ここではプログラム要素に関する各種テーブル類を保持する共通メモリ装置（“テーブル・メモリ”と呼ぶ）を想定し，字句解析チップはこれに抽出された文字列を転送し，識別コードへの登録と検索を依頼する方式を採っている．整定数に関しても，名前と同様の方式で識別コードへの変換が行われるが，実定数と文字列定数の場合はプログラムで使われるごとに新しい識別コードが割り当てられる．

（2） 4文字一括入力機能

処理速度とハードウェア規模の両方の要求から，本字句解析チップでは最大4文字の文字コードを一度に入力して解析するアーキテクチャを採用している．4文字の入力を一度に解析することに伴って，ある条件が他の条件に含まれる場合にどちらかの一致出力を抑止する機能や，特定の条件が満足しない場合を検知する機能（otherwise 機能）を実現する必要がある．本字句解析チップでは，AND 平面と OR 平面との接続部分に簡単なゲート回路を付加した拡張型 PLA によって，これらの抑止機能と otherwise 機能を実現している．拡張型 PLA の構造を図2に示す．

例えば，以下の入力条件を調べる場合，上の条件は下の条件に含まれており，文字列“A B C D”が入力された場合はすべての条件が満たされるが，解析結果としては一番上の条件だけが検知され，下の条件の検知を抑止する必要がある．

入力文字列＝“A B C D”，または，
“A B C ?”，または，
“A B ? ?”，または，
“A ? ? ?”

（ここで，“?”は任意の文字を許すことを示す．）

抑止機能が有効な例としては，連続するブランクのスキップや，“:=”と“:”の区別等がある．このよう

な抑止機能は，図2の上部に示すように，上の一致出力の否定信号によって，すぐ下の一致出力をゲートする回路で実現される．

一方，otherwise 機能は，以下の例に示すように，幾つかの条件のいずれも満足しなかった場合を検知しようとするものであり，ブール式としてこの条件を定義しようとすると多数の項を必要とする．

入力文字列
="A B C ?"，または，
"D E F G"，または，
"H I ? ?"，または，
上記以外の場合

この機能は，図2の下部に示すように，常に真となる一致出力（入力文字列="? ? ? ?"）が，すべての条件が満たされないときだけ真のままOR平面に入力される回路によって実現される．

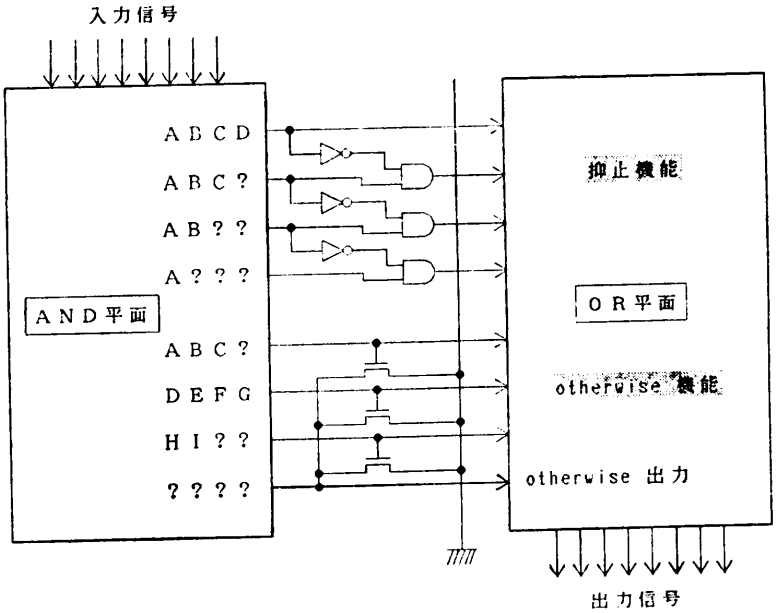

図 2 拡張型 PLA の構造
Fig. 2 Internal structure of extended PLA.

（3） 内 部 構 成

字句解析チップは，図3に示すように，字句解析のための状態遷移を行う拡張型 PLA を中心として，4文字の入力文字を保持するレジスタ，状態レジスタ，および，PLA から出力されるトークンを保持する

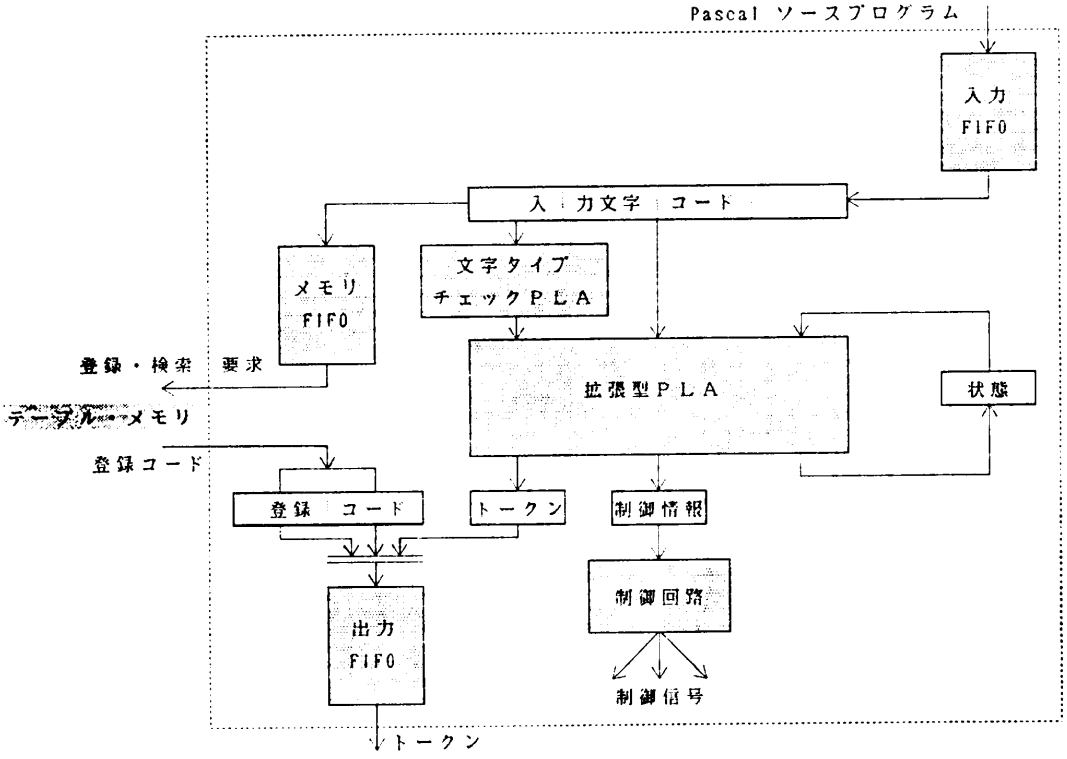

図 3 字句解析チップの構成
Fig. 3 Configuration of lexical analysis chip.

トークン・レジスタ，チップ全体の制御情報を保持するレジスタからなる簡単な回路によって実現される．ただし，この拡張型 PLA の項数を減らすために，最初の入力文字のタイプ（英字，数字，記号等）を判定する PLA がこれの前段に付加されている．また，テーブル・メモリへの文字列を格納しておく FIFO (First In First Out) メモリと，これから送られる登録コードを受け取るためのレジスタ，および，ソース・プログラムの入力とトークンの出力をスムーズに行うための2個の FIFO メモリが用いられている．

2.2 構文解析アーキテクチャ

（1） 基 本 方 式

Pascal の文法は，LL(1) と呼ばれる文法の性質をほぼ完全に満たしているため，以下に示す Greibach の標準形による書換え規則によって記述できる．

[Greibach の標準形][11]

$Z \rightarrow tXY$ t：端末記号

X, Y, Z：非端末記号

このような性質をもつ言語に対しては，プッシュダウン・オートマトンのモデルに従って，top-down に構文解析を行うことが可能である．すなわち，X, Y, Z を〈状態〉，t を〈入力トークン〉と考え，現在の状態 Z においてトークン t を入力したとき，次の状態 X, Y を生成するマシンを考える．このとき，次の状態 X, Y の有無に応じて以下の状態遷移とスタック操作を行う必要がある．

if $X=\phi$ and $Y=\phi$ then pop up stack$\rightarrow Z$

if $X\neq\phi$ and $Y=\phi$ then $X\rightarrow Z$

if $X\neq\phi$ and $Y\neq\phi$ then $X\rightarrow Z$ and $Y\rightarrow$push down stack

(ϕ：状態が生成されなかったことを示す．)

このような処理は，1個の言語要素に対応するトークン (t) と現在の状態 (Z) を入力し，次の2個の状態 (X, Y) を生成する1個の PLA とスタックによって実現される．

ところが，Pascal のような大きな言語仕様をもつプログラミング言語を完全な LL(1) の書換え規則で記述すると，膨大な状態数や状態遷移数をとることや，再帰的な文法構造から解析が難しくなる場合がある．この問題を解消するため，本構文解析チップでは，入力トークンのホールド機能と名前タイプの問合せ機能という2つの機能を状態遷移回路に付加して，構文解析ハードウェアを実現している．

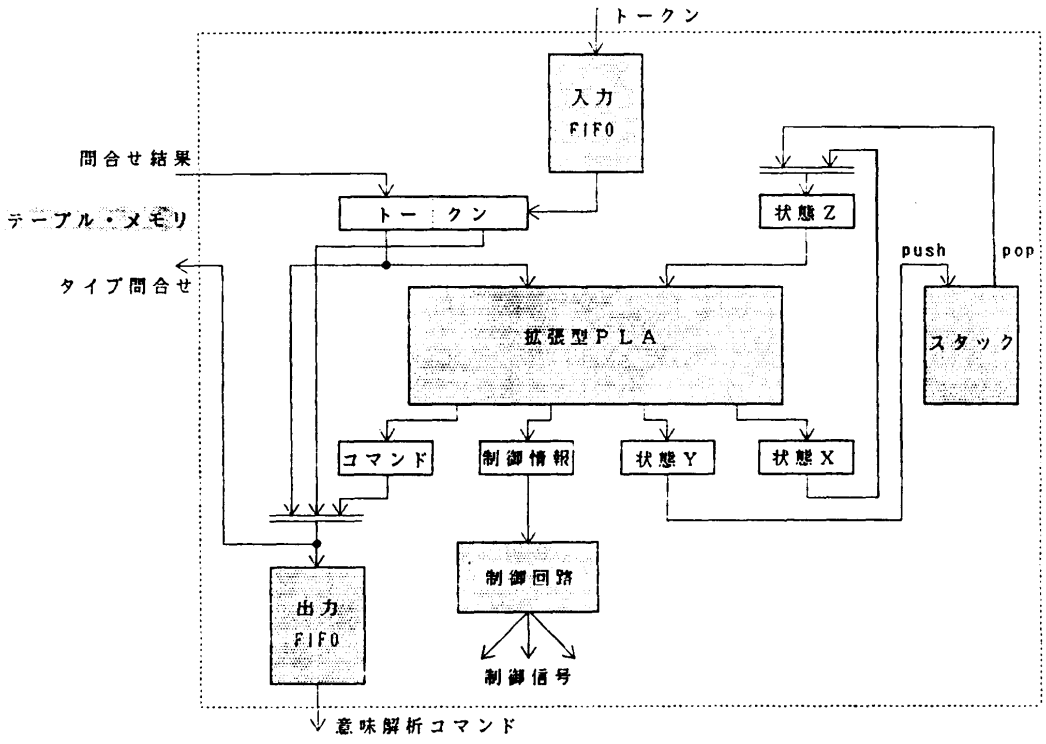

図 4 構文解析チップの構成
Fig. 4 Configuration of syntactic analysis chip.

(2) 入力トークンのホールド機能

この機能は，1回の状態遷移ごとにかならず1個のトークンを消費するのではなく，ある状態での遷移が決定されないときはそのトークンをホールドし，次の解析にも再度利用する機能である．例えば，以下に示すように，状態〈block〉においてトークン "label" が入力されればラベル宣言のはじまりとして状態〈d.label 1〉に遷移するが，それ以外の時はこの入力トークンをホールドして状態〈block 1〉に遷移し，ここで再度，同じトークンが解析される．

〈block〉 "label"→〈d. label 1〉

〈block〉 ? →〈block 1〉, hold トークン

(?："label" 以外の任意のトークン)

この機能によって，状態遷移の動作回数は増加するが，解析に必要な状態遷移のルール数を減らすことができる．この機能は，字句解析チップで述べた拡張型 PLA の otherwise 機能と PLA へのトークンの入力を制御する回路で実現される．

(3) 名前タイプの問合せ機能

字句解析チップから出力されるユーザ定義の名前を表すトークンには，その名前のタイプに関する情報（変数名，定数名，関数名などの区別）が含まれていない．しかし，Pascal における幾つかの構文規則では，このタイプの情報によって特定の構文を決定できる場合がある．このために，本構文解析チップでは，いったん，その名前をテーブル・メモリに転送し，そのタイプを通知してもらってから本来の状態遷移を行う方式を採っている．

例えば，状態〈variable〉においてユーザ定義の名前を表すトークン "name" が入力された場合，次の状態を同じく〈variable〉としたまま，テーブル・メモリにこの "name" のタイプを問い合わせ，次のサイクルでこの結果を受けて，本来の状態遷移を行う．

〈variable〉 "name"→〈variable〉, send トークン

〈variable〉 "var. id"→〈var 2〉

〈variable〉 "field. id"→〈var 2〉

この機能は，通常の PLA を用いた状態遷移によって実現できるが，特定の状態遷移時の動作として，テーブル・メモリへの問合せを行い，その返答を受け取るまでの間，字句解析チップの動作を停止する機能が必要となる．

表1 シミュレーション結果
Table 1 Simulation results.

プログラム名		EX1	EX2	EX3	EX4
プログラム・サイズ（文字数）		900	1,687	2,587	14,193
字句解析チップ	状態遷移回数 P	809	1,355	2,000	11,481
	出力トークン数	280	409	635	3,428
	（バイト数）	390	572	873	4,798
	文字数／トークン数	3.21	4.12	4.07	4.14
	TM 呼出し回数 T	110	163	238	1,370
	データ転送量（バイト） D	453	786	1,121	7,450
	実行時間 (μs)T_1* $(2P+10T+D)$	317	513	750	4,411
	性能（文字数/μs）	2.84	3.29	3.45	3.22
構文解析チップ	状態遷移回数 P	859	1,150	1,812	11,130
	入力ホールド回数	579	741	1,177	7,702
	スタック R/W 回数	584	716	1,168	7,696
	スタックの最大深さ	15	12	21	23
	意味解析コマンド数	497	641	1,045	6,505
	（バイト数）	695	956	1,490	8,969
	TM 呼出し回数 T	76	126	169	952
	データ転送量（バイト） D	304	504	676	3,808
	実行時間 (μs)T_2* $(2P+10T+D)$	278	406	599	3,559
	性能（トークン数/μs）	1.01	1.01	1.06	0.96
全実行時間 (μs)T_1+T_2		595	919	1,349	7,970
実行時間の比 T_1/T_2		1.14	1.26	1.25	1.24
VAX11/780 での解析時間 (s)T_3		0.4	0.5	0.6	2.7
性能比 $T_3/(T_1+T_2)$		670	540	440	340

* 1クロック＝100 ns として計算した． TM: テーブル・メモリ

(4) 内部構成

構文解析チップは，図4に示すように，構文解析のための状態遷移を行う拡張型 PLA とスタックを中心として，トークン・レジスタ，現在の状態 Z を保持するレジスタ，PLA から出力される状態 X, Y を保持する2個の状態レジスタ，PLA で生成された意味解析コマンドを保持するレジスタ，および，チップ全体の制御情報を保持するレジスタからなる簡単な回路によって実現される．また，テーブル・メモリへの問い合わせ行う回路，トークンの入力と意味解析コマンドの出力をスムーズに行うための2個の FIFO メモリが用いられている．

3. アーキテクチャの評価

前章で示した VLSI チップの言語解析アーキテクチャを評価するために，以下の検討と実験を行った．

(1) フルセットの Pascal を対象として，字句解析チップと構文解析チップにおける状態遷移動作を解

析し，これらのハードウェア規模を推定した．

（2） ハードウェア仕様記述言語 ISPS (Instruction Set Processor Specifications)[12] によって両 VLSI チップの構造を記述し，これに以下に示す4個の Pascal プログラム[13]を入力してシミュレーション評価を行った．

EX 1: 8 queens

EX 2: Topological Sort

EX 3: Reference Table Generator

EX 4: PL/0 Compiler

この時のシミュレーション結果を表1に示す．

（3） （1）と同一の文法規則を字句解析ジェネレータ (Lex[14]) と構文解析ジェネレータ (Yacc[15]) に入力し，これから生成される認識マシンのハードウェア規模と(1)の結果との比較を行った．

（4） これらのチップで用いられる機能モジュールを，レイアウト記述言語 DPL[16] で設計し，チップ全体としての大きさを推定した．

以下に，これらの評価結果について述べる．

3.1 字句解析アーキテクチャの評価

（1） ハードウェア規模の評価

4文字一括入力の PLA を用いた Pascal の字句解析チップの状態遷移表を表2に示す．（この表において，SHIFT, CLEAR, PUSH, TERM は，入力文字のシフト数，メモリ FIFO のクリヤ，メモリ FIFO への格納，トークンの認識完了を示す制御ビットである．）この PLA の諸元を以下に示す．

入力信号数：40（状態：6，文字タイプ：2，文字コード：32），

積項数：192（抑止積項数：11，otherwise 積項数：73 を含む）

出力信号数：20（状態：6，トークン：8，制御ビット：6）

一方，上と同一の Pascal 字句解析規則を入力として，字句解析ジェネレータ Lex を実行したところ，状態数 176，状態遷移

表2 字句解析における状態遷移
Table 2 State transitions in lexical analysis.

	PLA入力			PLA出力						拡張機能	
	状態	タイプ	文字列	状態	コード	SHIFT	CLEAR	PUSH	TERM	抑止	otherwise
基本状態	1		□□□□	1		4				○	
	1		□□□?	1	(16進数)	3	(□：ブランク)			○	
	1		□□??	1	↓	2				○	
	1		□???	1		1				⊥	
	1		:=??	1	4F	2			1		○
	1		:???	1	53	1			1		⊥
	1		. .??	1	53	2			1		○
	1		.???	1	50	1			1		⊥
	1		<=??	1	49	2			1		○
	1		<>??	1	46	2			1		○
	1		<???	1	47	1			1		⊥
	1		>=??	1	4A	2			1		○
	1		>???	1	48	1			1		⊥
	1		(*??	2		2					○
	1		(???	1	4B	1			1		⊥
	1		'???	3		1	1				
	1		+???	1	41	1			1	⊥は抑止・otherwise の対象を示す	
	※この他の1文字の記号			10個 {−*/=) [] , ; ^} も同様							
	1	数字		10		1	1	1			
	1		IF??	21		2	1	1			○
	※この他の4文字以下の			予約語21語も同様							
	1		BEGI	50		4	1	1			○
	※この他の5文字以上の			予約語13語も同様							
	1	英字		20		1	1	1			⊥
コメント	2		*) ??	1		2				○	○
	2		??*)	1		4				⊥	○
	2		?*) ?	1		3				○	○
	2		???*	2		3				⊥	○
	2		????	2		4					⊥
文字列	3		'' ??	4		1				○	○
	3		'???	1	F3	1			1	⊥	○
	3		????	3		1		1			⊥
	4		????	3		1		1			
数字	10		E+??	13		2		1		○	○
	10		E−??	13		2		1		○	○
	10		E???	13		1		1		⊥	○
	10		. .??	1	F1	0			1	○	○
	10		.???	11		1		1		⊥	○
	10	数字	????	10		1		1			○
	10		????	1	F1	0			1		⊥
小数	11	数字		12		1		1			
	12		E+??	13		2		1		○	○
	12		E−??	13		2		1		○	○
	12		E???	13		1		1		⊥	○
	12	数字		12		1		1			○
	12		????	1	F2	0			1		⊥
指数	13	数字		14		1		1			
	14	数字		14		1		1			○
	14	?		1	F2	0			1		⊥
名前	20	記号		1	80	0			1		
	20	英数字		20		1		1			
予約語	21	記号		1	01	0			1		
	21	英数字		20		1		1			
	※この他の4文字以下の			予約語21語も同様							
予約語	50		N???	51		1		1			○
	50		????	20		0					⊥
	51	記号		1	22	0			1		
	51	英数字		20		1		1			
	※この他の5文字以上の			予約語13語も同様							
状態数：60 遷移数：192 抑止項数：11 otherwise 項数：73											

数 753 の有限状態マシンが生成された．したがって，これを実現する PLA は以下の諸元をもつと考えられる．

入力信号数：16（状態：8，文字コード：8）
積項数：753
出力信号数：19（状態：8，トークン：8，制御ビット：3）

VLSI チップ上でのこれら PLA モジュールの占有面積を推定するために，入出力信号数や積項数をパラメータとした通常の PLA，および，これに抑止機能と otherwise 機能を備えた拡張型 PLA をレイアウト記述言語 DPL を用いて設計した．**表 3** に示す PLA モジュールの設計諸元から，上記の 2 個の PLA は以下の大きさをしていると推定される．

本字句解析チップの拡張型 PLA：1.59 Mλ^2
Lex の生成結果を実現する PLA：2.73 Mλ^2
（λ：マスクパターンの基本距離単位）

したがって，本字句解析チップは，4 文字の入力データ用シフトレジスタや幾分複雑な制御回路を必要とはするものの，その本体となる PLA の面積として，58% に小型化でき，拡張型 PLA がハードウェアの小型化に大きく貢献していることがわかる．

（2） 実行速度の評価

本字句解析チップや Lex で生成される有限状態マシンにおける言語解析の処理速度は，これらのマシンでの状態遷移回数で調べることができる．Lex による

表 3 PLA モジュールの設計諸元
Table 3 Design specification of PLA modules.

モジュール名	パラメータ	幅 (λ)	長さ (λ)	トランジスタ数
PLA	I：入力数，P：積項数 O：出力数 T：書込みテーブル	$16I+80+36$	$8P+123$	$5I+P+60$ $+I*P+O*P$
拡張型 PLA	（同上） S：抑止積項数 W：otherwise 積項数	$16I+80+77$ ただし，$S=0$ ならば $16I+80+51$	$8(P-S-W)+123$ $+15S+9W$	$5I+P+60$ $+I*P+O*P$ $+5S+W$

（λ：マスクパターンの基本距離単位）

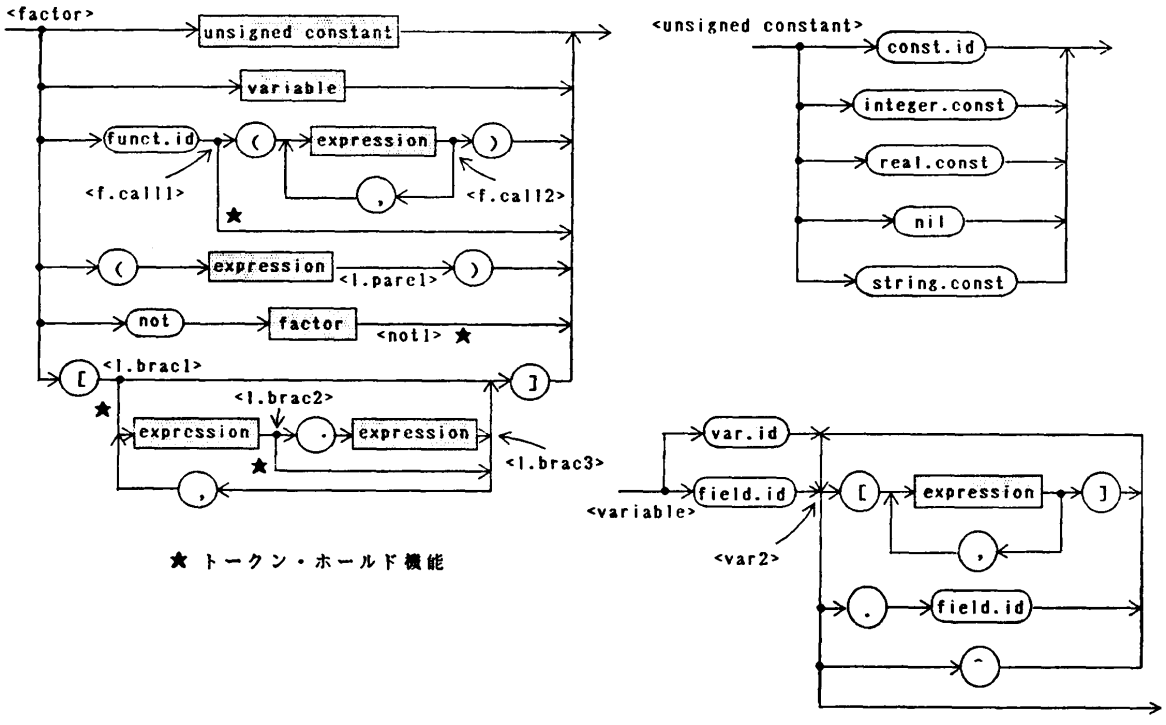

図 5 <factor> の構文規則
Fig. 5 Syntactic rules of <factor>.

マシンでは1文字を入力するごとに1回ずつ状態遷移すると考えられるため，本字句解析チップのシミュレーション結果における状態遷移回数と各 Pascal プログラムの文字数を比較することにより，本字句解析チップは Lex によるものより約 20% 程度高速化されていることがわかる．1個のトークンに対応するソースプログラムの文字数を計算してみると，表1に示すように約4文字／トークンであるが，余分なブランク等を除けばこの値は3文字／トークン以下に低下する．したがって，通常の Pascal プログラムにおいて，4文字を一度に解析しても，トークンを生成できる場合はそれほど多くなく，解析対象の文字数を増やしても，実行速度の面ではそれほど大きな性能改善を望めないことがわかる．

一方，本字句解析チップのアーキテクチャでは，名前や定数の登録と検索のために，テーブル・メモリの呼び出しが頻繁に行われ，約2.5個のトークンに対して1回の割合で発生している．また，これに伴うテーブル・メモリとのデータ転送量も多く，テーブル・メモリの内部処理やデータ転送等の名前と定数の処理がチップ全体の性能に大きく影響すると考えられる．

3.2 構文解析アーキテクチャの評価

Pascal の全構文規則を前章に示した形式の書換え規則で書き下した結果，状態数 124，状態遷移数 248 のプッシュダウン・オートマトンによって実現できることがわかる．（例として，〈factor〉の構文規則とこの解析を行う状態遷移表を**図5**と**表4**に示す．）したがって，この構文解析チップは以下の諸元をもつ1個の PLA によって実現できる．

入力信号数：16（入力トークン：8，状態コード：8）
積項数：248（otherwise 積項数：69 を含む）
出力信号数：29（コマンド：8，状態コード：16，制御コード：5）

表 4 構文解析における状態遷移（<factor> の解析）
Table 4 State transitions in syntactic analysis of <factor>.

PLA 入力		PLA 出力				拡張機能
状態 Z	トークン	状態 X	状態 Y	HOLD	CHECK	otherwise
<factor>	name	<factor>		1	1	
<factor>	const. id					
<factor>	integer. const					
<factor>	real. const					
<factor>	nil					
<factor>	string. const					
<factor>	var. id	<var. 2>				
<factor>	field. id	<var. 2>				
<factor>	funct. id	<f. call 1>				
<factor>	(	<expression>	<l. pare 1>			⊥は otherwise
<factor>	not	<factor>	<not 1>			の対象を表す
<factor>	[	<l. brac 1>				
<f. call 1>	(	<expression>	<f. call 2>			○
<f. call 1>	?			1		⊥
<f. call 2>	,	<expression>	<f. call 2>			
<f. call 2>	)					
<l. pare 1>	)					
<not 1>	?			1		
<l. brac 1>	]					○
<l. brac 1>	?	<expression>	<l. brac 2>	1		⊥
<l. brac 2>	..	<expression>	<l. brac 3>			○
<l. brac 2>	?	<l. brac 3>		1		⊥
<l. brac 3>	,	<expression>	<l. brac 2>			
<l. brac 3>	]					

状態数：124　遷移数：248　抑止項数：0　otherwise 項数：69

(1) 入力トークンのホールド機能

構文解析チップの状態遷移のうち，入力トークンのホールド機能を 44 か所の遷移で利用している．この機能の使われ方としては，以下の3形態がある．

① 詳細な解析を保留し，別の状態に移って解析する場合．(〈factor〉における〈l. brac 1〉や〈l. brac 2〉のような20か所がある．)

② 特定の構文が解析完了し，1段上位の状態での解析を行う場合．(〈factor〉における〈f. call 1〉のような11か所がある．)

③ 解析構文が終了したときの処理を挿入するために，本来，不要な状態遷移を行っている場合．(〈factor〉における〈not 1〉のような 13 か所がある．)

①の場合は解析に必要な PLA の項数を減らす効果があるが，状態遷移の動作回数は多くなるため，実行時間は増大する．②の場合，現在の構文がどのような構文から参照されているかが特定できないため，多くの構文から参照される〈statement〉や〈variable〉等のような構文をこの機能無しで実現することはほとんど困難と思われる．

表1に示すシミュレーション結果では，入力トークンの数の約3倍の状態遷移を行っており，1個のトークンに対して，平均2回ずつホールド機能が働いていることがわかる．上記の②，③の利用形態でのホールド機能は必須なものであるが，①の利用形態に関しては実行速度とハードウェア量のトレードオフについて，詳細な検討を行う必要がある．

(2) 名前タイプの問合せ機能

名前のタイプが指定されている構文を正確に解析するために，本機能は 13 か所の状態遷移で使われている．この中には関数の定義時のデータタイプ名のように検査を省力して，構文解析を進めることができる場合もあるが（9か所），例で示した〈factor〉の場合や，〈statement〉，〈variable〉，〈simple type〉の場合のように複数の名前タイプを許し，このタイプに応じて解析方法が変わるもの（4か所）に関しては，この機能が有効である．この機能を用いないとすると，トークンのホールド機能の②の場合と同じ状況が起こり，構文解析が困難になると思われる．

シミュレーションにおけるこの機能の発生状況は，トークン数に対して 26～30% ほどであり，字句解析チップと同様にテーブル類の参照，検索処理の高速化が要求される．ただし，上で述べたように，構文解析チップとしてはこの問合せを省力できる場合もあり，ある程度の高速化は容易に可能である．

(3) ハードウェア規模の評価

本構文解析チップと同じ形式の書換え規則を入力として構文解析ジェネレータ Yacc を実行したところ，状態数 453，shift 動作数 1,085，reduce 動作数 273，accept 動作数 1，error 動作数 180，goto 動作数 264，default goto 動作数 125 のパーサが生成された．Yacc で用いられているマシン・モデルは，**図6**に示すように，shift 動作，reduce 動作，goto 動作を合計

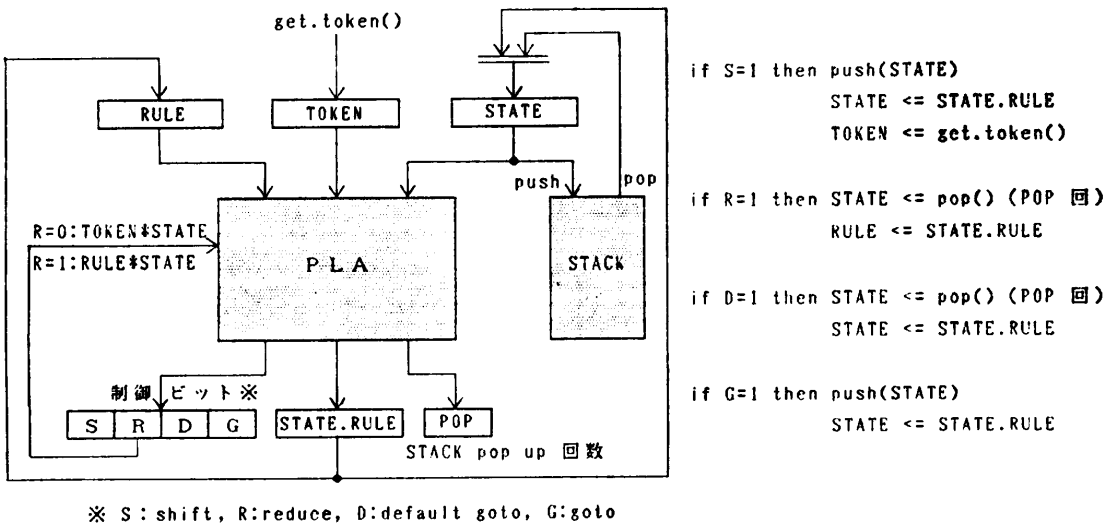

図 6 Yacc モデルのハードウェア化
Fig. 6 A hardware implementation of Yacc model.

した数の項数をもつ PLA ベースのハードウェアによって実現できる．この PLA の諸元を以下に示す．

入力信号数：25（状態：8，トークン：8，規則番号：8，制御：1）

積項数：1,622 (shift：1,085, reduce：273, goto：264)

出力信号数：13（状態：8，pop up 数：2，制御ビット：3）

この PLA と本構文解析チップで用いた拡張型 PLA の大きさを比べると以下のようになる．

本構文解析チップの拡張型 PLA：1.17 $M\lambda^2$

Yacc の生成結果を実現する PLA：7.91 $M\lambda^2$

したがって，Yacc によって生成される LR(1) 文法を基本とするパーサは，本構文解析チップにおける PLA より 6.8 倍も大きな面積を必要としている．この差は主として言語の性質によるものと考えられ，Pascal のように top-down に解析できる言語の場合には一般の LR パーサよりもはるかに簡単なハードウェアによって実現できることを示している．

3.3 言語解析アーキテクチャ全体の評価

字句解析チップと構文解析チップで用いられる機能モジュールをレイアウト記述言語 DPL で設計し，これらの機能モジュールを配置・配線して各チップの全体イメージを構成してみた．この結果，現在の MOS デバイス技術から，製造可能な最小寸法である 2λ を 1.5 μm と仮定すれば，これらの言語解析チップの大きさは，字句解析チップが，3.3 mm×5.1 mm (16.8 mm²)，構文解析チップが，3.3 mm×5.2 mm (17.2 mm²) と見積られる．したがって，これらのチップは両方合わせても 1 個のパッケージに入る程度の大きさであり，Pascal 解析用 1 チップ VLSI を実現することも十分に可能である．

また，現在の技術水準から 10 MHz (100 ns) の速度でこれらの VLSI チップが動作できると仮定すれば，字句解析チップは 1 秒間に約 3 百万文字の Pascal ソース・プログラムを解析でき，また，構文解析チップは 1 秒間に約百万トークンを処理する能力を持っていると考えられる．したがって，これら言語解析チップを用いて約 14,000 文字からなる PL/0 Compiler の字句解析と構文解析を連続的に実行した場合，約 8 m 秒かかることがシミュレーション結果からわかる．一方，同じプログラムを VAX 11/780 の Pascal コンパイラで処理した場合，同様の解析に約 2.7 秒を要することから，本言語解析チップは，VAX 11/780 のソフトウェア・コンパイラに比べて，300 倍以上の言語解析能力を有していると推定される．また，字句解析チップと構文解析チップの実行時間を比べると，14〜26 % 程度の差しかなく，これらチップ間のデータの流れは比較的スムーズであると予想される．

4. むすび

本稿では，プログラミング言語解析用 VLSI チップの具体例として，Pascal を対象とする字句解析チップと構文解析チップのアーキテクチャについて述べた．また，これらのチップで用いた言語解析アーキテクチャに関して，ソフトウェアによる実現方式と比べたハードウェア規模や実行性能等の評価結果について報告した．

これらの言語解析用 VLSI チップは，プログラミング言語の解析に適するように拡張した PLA と記憶要素を主体とする比較的簡潔なアーキテクチャによって，言語の構文規則を直接 PLA の書込みパターンとして与えるだけで実現可能であり，多くの言語に対する適応性は高いと考えられる．また，これらのチップのハードウェア規模は全体でも 1 個のパッケージに格納できる程度の大きさであり，実行速度の面でも，シミュレーションの結果から，汎用計算機のソフトウェアによる言語解析に比べて 2 桁以上高速化できることが示された．

今後の課題として，ここで述べた言語解析用 VLSI チップの他のプログラミング言語に対する適応性を評価するとともに，言語処理システム全体の VLSI 化のために，コンパイラにおけるコード生成部やテーブル類の処理部，あるいは，インタプリタにおける実行処理部等の VLSI 化を検討していきたい．また，大規模ソフトウェアを VLSI 化する場合のソフトウェアとハードウェアの協調性，エラー処理のような例外処理への対応，および，マンマシン・インタフェイス機能等に関しても，今後，検討していきたいと考えている．

謝辞 本研究を行うにあたって，日頃より御指導，御討論いただいている三菱電機株式会社中央研究所システム研究部房岡グループ・マネージャ，VLSI 設計システムの開発と機能モジュールのレイアウト設計に御協力いただいた同部杉本，瀬尾両氏に深謝いたします．

参考文献

1) Aho, A. V. and Ullman, J. D.: *The Theory of Parsing, Translation, and Compiling, Vol.*

1: Parsing, Prentice-Hall, N. J. (1972).
2) Pyster, A. B. (松尾訳): コンパイラの設計と構築, 近代科学社, 東京 (1983).
3) Hartmann, A. C.: Software or Silicon? The Designer's Option, *Proc. IEEE*, Vol. 74, No. 6, pp. 861-874 (1986).
4) Floyd, R. W. and Ullman, J. D.: The Compilation of Regular Expressions into Integrated Circuits, *J. ACM*, Vol. 29, No. 3, pp. 603-622 (1982).
5) Andre, F. et al.: KENSUR: An Architecture Oriented Towards Programming Language Translation, *Proc. 7th Annual Symposium on Computer Architecture*, pp. 17-22 (1980).
6) 板野ほか: パイプライン型字句解析プロセッサの設計と実現, 情報処理学会論文誌, Vol. 28, No. 1, pp. 82-90 (1987).
7) Fusaoka, A. and Hirayama, M.: Compiler Chip: A Hardware Implementation of Compiler, *Proc. Symposium on Architectural Support for Programming Language and Operating Systems*, pp. 92-95 (1982).
8) Seo, K. et al.: Design and Evaluation of Parsing Chip, *Proc. VLSI '83*, pp. 317-326 (1983).
9) Nori, K. V. et al.: *The PASCAL ⟨p⟩ Compiler: Implementation Notes*, Wiley, Chichester (1981).
10) Pemberton, S. and Daniels, M. C.: *Pascal Implementation: The P4 Compiler*, Ellis Horwood Limited, Chichester (1982).
11) Greibach, S. A.: Formal Parsing Systems, *C. ACM*, Vol. 7, No. 8, pp. 499-504 (1964).
12) Barbacci, M. R. et al.: The ISPS Computer Description Language, Carnegie-Mellon Univ. Tech. Report No. CMU-CS-79-137 (1977).
13) Wirth, N.: *Algorithms+Data Structures=Programs*, Prentice-Hall, N. J. (1976).
14) Lesk, M. E. and Schmidt, E.: Lex—A Lexical Analyzer Generator, Bell Lab. Computing Science Tech. Report No. 39 (1975).
15) Johnson, S. C.: Yacc: Yet Another Compiler Compiler, Bell Lab. Computing Science Tech. Report No. 32 (1975).
16) Batali, J. and Hartheimer, A.: The Design Procedure Language Manual, MIT A. I. Memo. No. 598 (1980).

(昭和 62 年 4 月 3 日受付)
(昭和 63 年 2 月 10 日採録)

平山 正治 (正会員)

昭和 25 年生. 昭和 48 年北海道大学工学部電気工学科卒業, 昭和 50 年同大学院工学研究科情報工学専攻修士課程修了. 同年三菱電機(株)入社. 以来, 中央研究所にて, VLSI 方式技術, VLSI 設計支援技術に関する研究開発に従事し, 現在に至る.