計算機アーキテクチャ 42－3
（1981. 7. 22）

ハードウェアによるコンパイラの実現：設計とシミュレーション評価

平山正治　工藤司　房岡瑋
（三菱電機株式会社）

1. まえがき

Bashkow's FORTRAN マシンの提案以来，ソフトウェアをハードウェア化しようとする試みは多数提案されている。これらの多くはプログラミング言語をコンパイルせずに直接実行しようとするものであるが，現在，いくつかの商用マシン（Burroughs B5500，SYMBOL）以外は実験的なマシンがほとんどであり，このようなマシンが広く利用される状況には至っていない。従って，多くの計算機システムにおいてコンパイリングは計算機タスク中のかなりの比重を占めている現状である。このような状況においてコンパイル処理をハードウェア化しようという試みは，以下のような観点から，将来の計算機システムにおいて有望なアプローチと考えられる。

(1) 近年のLSIの進歩はめざましく，かなりの複雑さ（数十万ゲート）をもったシステムまで容易にLSI化できる状況にあり，コンパイラのような複雑なソフトウェアでもLSI化し，高速かつ安価に実現する可能性がある。

(2) コンパイル処理は逐次的な記号処理であり，コンパイル・アルゴリズムのパイプライン構造は並列処理プロセッサによるハードウェア化に適した構造と考えられる。

(3) コンパイラを数チップのVLSIで実現し，インテリジェント端末などの中に組込んでしまい，メインフレームからコンパイリングの負荷を取り除いたような機能分散システムはこれからの計算システムとして望ましい方向と考えられる。

本報告では高級言語Pascalのソースプログラムを仮想スタックマシンのマシンコードであるP-codeにコンパイルする処理装置の設計思想，基本構成，およびその評価について述べる。

2. コンパイラマシンの背景

通常はソフトウェアによって処理されているコンパイラをハードウェア化しようという試み（コンパイラマシン）は以下のような背景から導かれたものである。

(1) 高級言語直接実行マシンの限界

1960年代から多くの高級言語マシンが提案されているが，いくつかの商用マシン以外は，現在，ほとんど利用されていない状況にある。高級言語のソースプログラムを直接，インタプリートしようというアイデアは単純で魅力的ではあるが，その実現には本質的に困難な問題を含んでいると考えられる。すなわち，直接実行マシンを複雑かつ非効率的なものにしている原因は，言語上の表現からハードウェア内部のプログラム構造とデータ構造に変換する処理（記号処理）と，実際の演算（データ処理）という2つの異なった機能を実現しなければならない点にあると考えられる。また各言語がサポートしているデータタイプやオペレーションを効率良く処理しようとすると，各言語ごとに異なったアーキテクチャの直接実行マシンを開発しなければならないという問題もある。このような

観点から，高級言語は最適なオブジェクトコードにコンパイルされ，これが高性能の処理装置によって実行される事が必要であり，このためにコンパイラをハードウェア化によって高速化をはかるような試みが考慮されるべきだと考えられる．

(2) LSI技術の進歩

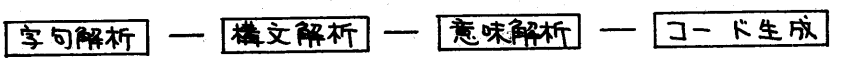
近年のLSI技術の進歩はめざましく，スピードのうえでも実装密度のうえでも大幅に性能が向上している．　従って，コンパイラのような,かなり複雑なソフトウェアでも数チップのVLSIで構成でき，インテリジェント端末やパーソナルコンピュータ等の内部に組込む事が容易に可能な状況になりつつある．　このような状況のもとでは，計算機の機種ごとに多数のプログラミング言語のコンパイラを開発するよりも，特定のコンパイル専用ハードウェアを開発しておき，機種，言語の仕様に応じてROM，PLA等の内容を変更する事でコンパイラを実現する方が安価にできる可能性がある．

(3) 効率的機能分散

高級言語のプログラムをマシンコードに変換する処理と，このマシンコードを実行する処理との両方が将来の計算機システムにおいても必要であると考えられる．　しかし，実際の演算に対して補足的なタスクであるコンパイリングをメインフレームで実行する必要はなく，またメインフレーム自体，コンパイリングのようなテキスト処理を得意としていない．　従って，コンパイリング，プログラム編集，デバッグ等のマンマシンインターフェイスに属するようなタスクをメインフレームから取除き，インテリジェント端末等に組込んでしまう方が望ましいと考えられる．　この結果，各種言語のプログラム編集，デバッグ，コンパイル等を行なうフロントエンドプロセッサと，各言語の種類に対応した専用の高速データ処理機能をもったバックエンドプロセッサ（たとえばFORTRANに対するアレイプロセッサ，COBOLに対応するファイルプロセッサ，データベースプロセッサ等）を備えた機能分散システムが将来の計算機システムとして実現される．

3. コンパイラマシンの設計方針

Pascalで記述したPascalコンパイラが約4,000ステートメントにおよぶ事からも明らかなように，コンパイラはかなり複雑なソフトウェアのひとつと考えられる．　しかし，コンパイラの処理内容は以下に示す4つの基本的なサブタスクに分解する事ができる．



コンパイラをハードウェア化する1つの方法としては，これらのサブタスクを専門に処理するマイクロプロセッサを結合し，パイプライン的にコンパイル処理を行なう事が考えられる．　しかし，汎用のマイクロプロセッサではコンパイリングのような記号処理には不向きな点もあり，また，単なるソフトウェアの分解，マイクロプログラムへの再コーディングだけでは本来の意味でのハードウェア化とは言い難い．

本報告で述べるコンパイラマシンは，複数の処理モジュール間をデータトークンが移動していく中でコンパイリングが行なわれ，また，各処理モジュールはVLSI化の見地から，ほとんどの演算，制御をPLAで実現し，PLA＋Xモリ（レジスタ）という単純な構成で設計する事をめざしている．

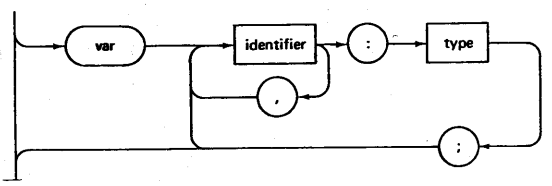
3.1 構文解析

コンパクトで効率の良い構文解析ハードウェアを実現するためには，対象とする言語の文法とその解析方法とに依存するが，Pascalの文法は以下に示すGreibach's normal formと呼ばれる書きかえ規則の形式で，ほとんどを記述する事ができる．このような性質をもつ言語に対してはtop-downに構文解析を行なう事が容易であり，これを実現するためにはスタックつきの有限状態マシンを構成すれば良い．

Greibachの標準形は以下のように表わされる．

$$Z \rightarrow a$$
$$Z \rightarrow a\,X$$
$$Z \rightarrow a\,X\,Y$$

ここで，X，Y，Zは非端末記号であり，aはhandleと呼ばれる端末記号である．このような書きかえ規則によって展開できる言語の構文解析は以下に示すような有限状態マシンによって実現される．すなわち，X，Y，Zを〈状態〉と考え，aを〈入力シンボル〉とし，現在の状態Zにおいてシンボルaを入力した時，次の状態X，Yを生成するようなマシンを考えれば良い．ここで，次の状態X，Yに応じて以下のような動作をすればよい．




```
<block>   --->   var <d.var1>
          SP token op="05

<d.var1>  --->   id. <d.var2>
          SP token op="15, data="8nnn

<d.var2>  --->   , <d.var1>
          --->   : <type> <d.var3>

<d.var3>  --->   ; <d.var4>
          SP token op="16

<d.var4>  --->   id. <d.var2>
          SP token op="15, data="8nnn
          --->   xx <d.var5>   (*)
          SP token op="04

<d.var5>  --->   procedure <d.proc1>
          ...
```

図1. 変数宣言部の構文解析

$$(Z, a) \longrightarrow (X, Y)$$

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 $X \rightarrow Z$, $Y \rightarrow$ push down stack

Pascalにおける変数の宣言を行なう構文における書きかえ規則を例として図1に示す．

しかしながら，このような決定性文法では記述できない，またはできにくい構文がPascalの中に存在しているため，現実のハードウェアではこれを回避するための機構が導入されている．

3.2 意味解析

コンパイリングにおける複雑な処理のひとつとしてデータタイプの処理やデータの配置など，変数に関連するものがあり，この中には複雑なキーによるテーブルサーチのようにハードウェア化が難かしい処理がある．Pascalの場合，プログラムの実行部で使われる定数，変数，プロシジャ等は，必ずプログラムの宣言部で前もって宣言されていなければならず，これらがプログラムの実行部で参照された時には，コード生成のために必要なコンパイル環境はすでにできあがっている．このようなPascalの構文上の構造を反映して，現在検討中のコンパイラマシンも，プログラムの宣言部の意味を解析してオブジェクトコードの生成に必要なコンパイル環境を作り出す部分と，プログラムの実行部の内容に従ってオブジェクトコードの生成を行なう部分とを独立した処理装置を割当てている．

また，上記のコンパイル環境や，ソースプログラム上のレキシカルな表現と内部コードとの対応表等を格納するための共有記憶装置を設け，これには連想機能のような高度な検索機能によって，複雑な情報検索が容易にかつ高速に行なえる構成になっている．

3.3 システムの構造

以上のような考察から，現在検討中のコンパイラマシンは，図2に示すように，5台の処理装置（Lexical Processor，Declaration Parser，Semantic Processor，Body Parser，Code Generator）と1台の共有メモリ（List Memory）が2本の共通バス（P-BUS，Q-BUS）とで結合された構成になっている．

各処理装置はコンパイル処理の各パイプラインステージに対応し，（構文解析は宣言部と実行部で別の処理装置を割当てている）前のステージから転送されたデータパケット（トークン）を受取り，要求されるオペレーションの後，新しいトークンを生成して次のステージに転送するという単純なサイクルをくりかえす事により，Lexical Processorに入力したPascalのソースプログラムが，Code GeneratorからP-codeにコンパイルされて出力される．

このコンパイラマシンはホスト計算機と通信回線で接続された端末として動作し，コンパイルの対象となるソースコードや，コンパイル結果のオブジェクトコードは通信回線を経由してホスト計算機との間で転送される．

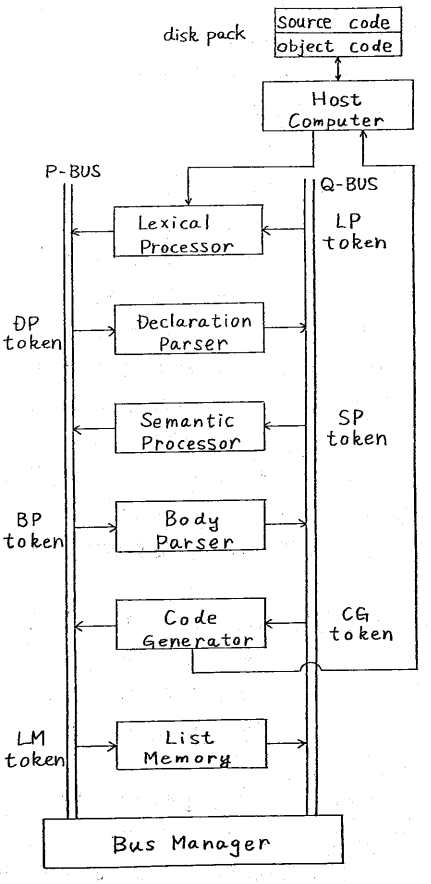


図2．システムの構成

4. コンパイラマシンの内部構成と動作概要

4.1 字句解析

ホスト計算機から転送されてくるPascalのソースプログラムは，Lexical Processorによって ASCII コードによる表現から内部コードによる表現に変換される．この変換過程は大きく２つの処理に分ける事ができる．　最初の処理では，入力された文字列を走査し，特殊記号，文字列，整定数，実定数，文字定数に分解する．　また，この段階で，不用なブランク，改行コード，コメントは取除かれる．次の処理として特殊記号を１B長の内部コードに変換する．　文字列，各定数はList Memoryに転送し既定義か未定義かのチェックを依頼する．　各定数の場合はこれらの値が登録されたテーブルの番地がList Memoryから転送されるので，これに定数のタイプを示すコードを加え，２B長の内部コードを生成する．

文字列の場合は，予約語，システムであらかじめ決められている標準名，およびユーザ定義の識別名の３種の可能性があり，List Memoryからの応答に従って，それぞれ，１B長，３B長，２B長の内部コードを生成する．（図３に内部コードの一覧表を示す．）このようにして変換された１～３B長の内部コードはDPトークンとしてDeclaration Parserに転送される．

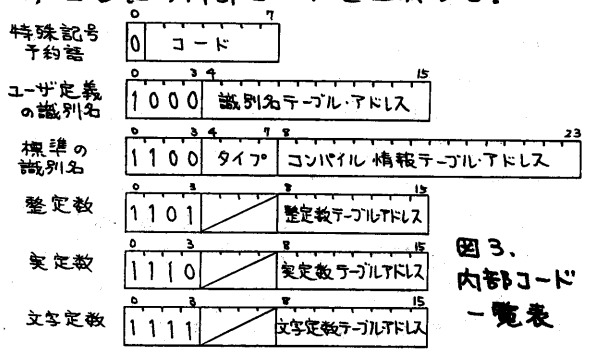


図３．内部コード一覧表

現在検討中のコンパイラマシンではLexical Processor，Semantic Processor，Code Generatorの３種の処理装置を同一のハードウェアで実現する事を検討している．（この処理装置の内部構成を図４に示す．）

この処理装置は，図からも明らかなように，マイクロプログラム制御のプロセッサであるが，

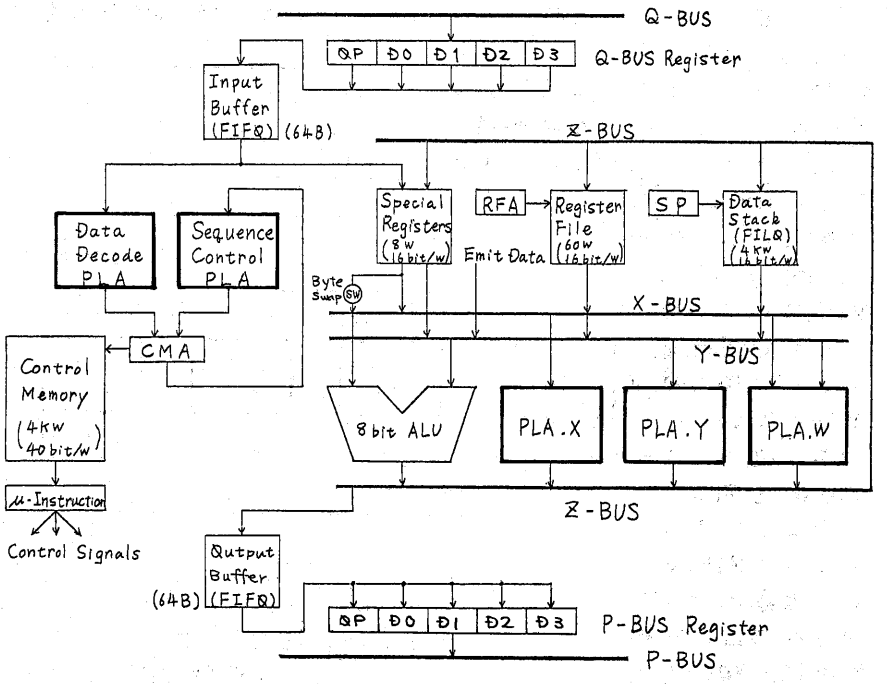


図４．Lexical Processorの内部構成

演算装置として3箇所，さらに命令のシーケンス制御，入力データのデコードの各所にPLAを使用し，これらの具体的な機能はこの処理装置を使う目的に応じて設定すればよく，この動作の区別はマイクロ命令のフィールドで指定すればよい．　従って，特殊な演算，任意のデータデコード，自由な条件つき分岐等がハードウェアを変更する事なく設定でき，PLAと制御記憶のマイクロプログラムを変更する事で広範な応用に適応できるような柔軟な構造をもった処理装置である．

4.2 構文解析

前述のようにPascalの構文ではプログラムの宣言部と実行部とが明確に分離され，コンパイラにとっても処理が容易な構造になっている．　コンパイラマシンでもこの利点を反映して，構文解析を2つの独立した処理装置Declaration Parser，Body Parserの2つに割当て，それぞれプログラムの宣言部，実行部の構文解析を受けもっている．

Declaration ParserはLexical Processorから転送されてくる内部コード（OPトークン）を受取り，このうちプログラム宣言部の構文をチェックし，構文に従って変数，データタイプ，プロシジャの登録等，コンパイル環境を作り出すための指示をSPトークンとしてSemantic Processorに送出する．（図1参照）

この時の構文解析は前章で示したように，有限状態マシンのモデルをレジスタ，スタック，およびPLAでインプリメントしている．　Parserの具体的な内部構造は，図5に示すように，状態レジスタX，Y，Z，入力コードを格納するCレジスタ，状態スタック，および2個のPLA（X-State PLA，Y-State PLA）とで有限状態マシンを構成している．　これら構文解析回路の他にSemantic Processorに送るSPトークンの命令コードを生成するOP-code PLA，状態に応じた制御情報を出力するControl PLA，および Parser 全体の制御を行なうMain Control PLAとで構成されている．

この処理装置もPLA主体の構成となっており，以下に述べるBody Parserも同じハードウェアを用い，PLAの内容を変更する事によって，プログラムの実行部の構文解析を行なっている．

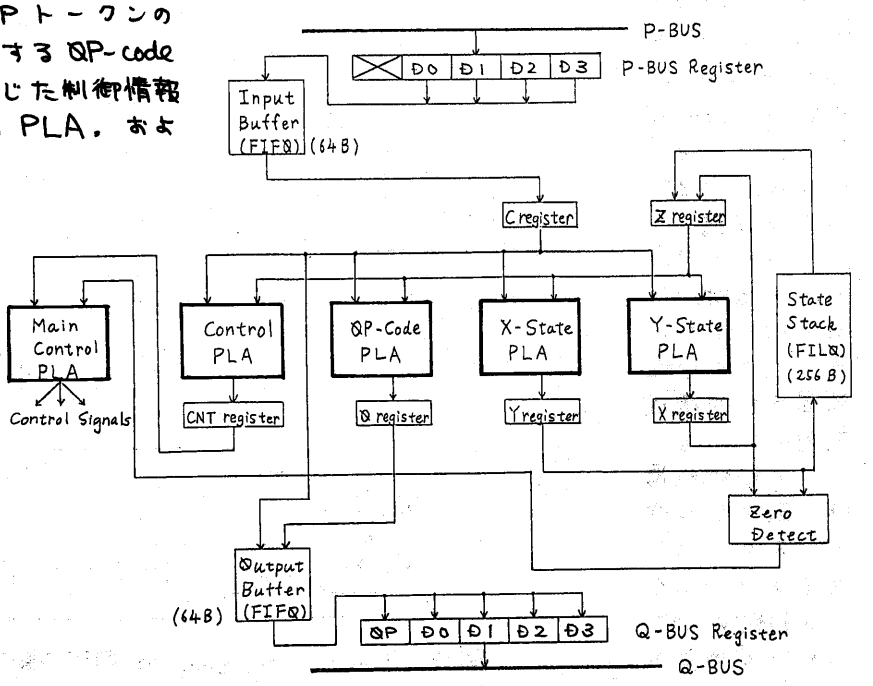


図5．　Parserの内部構成

Declaration Parserがプログラム実行部の内部コードを入力した時は構文解析を行なわず，その内部コードをそのままSemantic Processorに送る．　Semantic Processorでは，ユーザ定義の識別名，定数に対してこれらのコンパイル環境が格納されている領域がアクセスしやすいように内部コードのアドレス部を変更するだけで，その他の内部コードに対しては何も操作せずにBody Parserに送出する．

Body Parserはこれらの内部コードを受取り，Declaration Parserと同等の方法でプログラム実行部の構文をチェックし，その構文に従ってコード生成のための指示をCGトークンとしてCode Generatorに送出する．（図6参照）

4.3 意味解析・コード生成

前述したように，本コンパイラマシンでは意味解析を行なうSemantic Processorとコード生成を担当するCode Generatorの2個の独立した処理装置によって，オブジェクトコードの生成を実現している．　しかし，いずれの場合も，前のステージにおいてDeclaration Parser，またはBody Parserによって構文解析が完了しているため，これら2台の処理装置の機能は比較的に単純である．すなわち，Semantic ProcessorはSPトークンの命令コードに従って，List Memory中にコンパイル環境を生成，Code GeneratorはCGトークンの命令コードに従ってオブジェクトコードを生成すれば良い．

たとえば，図1に示した変数宣言の例であれば，SPトークン"05によって変数宣言の初期設定，"15によって同時に送られてくる識別子の変数としての登録，"16によって変数のデータタイプの登録，"04によって変数宣言の終了処理をそれぞれ行なえば良い．　また，図6に示すif文におけるコード生成の例であれば，CGトークン"11によって前の<exp>の値のロード，および条件つき分岐命令（FJP）の生成，"12によってlabel2への無条件分岐命令とlabel1の生成，"13によってlabel1の生成，"14によってlabel2の生成をそれぞれ行なえば良い．

以上のように，処理の内容は非常に簡単なものもあるが，多くはList Memoryの参照やループ構造を必要とす

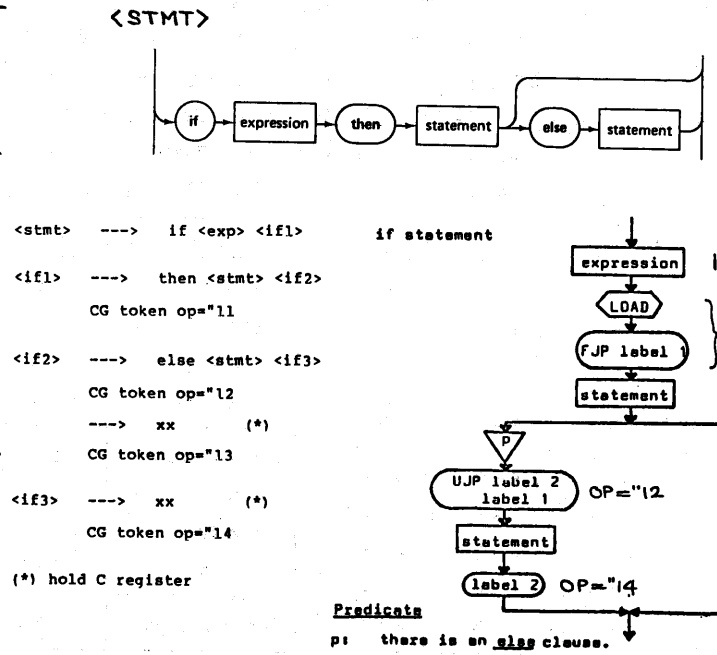


図6．if文のコード生成

る処理もあり，簡単なハードウェアによる実現はむずかしい．　現在は，前述したようにLexical Processor と同じハードウェアを用い，PLAのプログラミングとマイクロプログラムによる実現を検討している．

5．性能評価

効率の良いハードウェアを開発するためには事前のシミュレーションによる性能評価を十分に行なう必要があるが，我々が現在，設計中のコンパイラマシンであれば，以下のような項目について調べる事が重要である．

- 各処理装置でのトークンの処理時間，およびトークンの発生頻度と分布
- 2本の共通バスの使用状況
- List Memoryへのアクセス頻度と応答時間
- トークンの形式，データ転送の形式
- 処理装置の内部回路のみなおし，メモリ容量，PLA容量の推定

以上のような項目に関して解析する事を目的として，現在，ハードウェア記述言語 ISPSを用してシミュレーション実験を行なっている．　以下では，Lexical Processor と Declaration Parser のシミュレーション結果について述べる．

(1) Lexical Processor の評価

図4に示すマイクロプログラム制御のプロセッサによってLexical Processorを実現したものをハードウェア記述言語ISPS で記述し，実際に3個のサンプルプログラム（EX1：quick sort，EX2：8 queens，EX3：topological sort）を実行してみた時の結果を表1に示す．

この表からも明らかなように，入力文字1文字あたり平均12クロック（1クロックを100 nsecとすれば1.2 μsec）を要している．　また，List Memoryへのアクセス頻度はそれほど高くなく，40マイクロ命令に1回程度の割合になっている．　この処理を実現するために要したマイクロ命令数は約170語，また，使用したPLAの項数は，PLA.Xが50項，PLA.Yが18項，PLA.Wが0項，Sequential Control PLAが111項，Decode PLAが20項という状況であった．　このようにPLAはかなり有効に使用されているが，反面，スタック

表1. Lexical Processorの性能評価

プログラム	EX 1	EX 2	EX 3
ステートメント数	49	36	68
入力文字数	1,992	1,559	2,750
出力したトークンの数	280	250	388
出力したコードの量(B)	445	402	602
実行マイクロ命令数	6,311	5,457	9,018
実行クロック数	23,075	19,951	32,992
List Memoryアクセス数	165	153	214
P-BUS使用回数	214	196	280
Q-BUS 〃	165	153	214

表2. Declaration Parserの性能評価

プログラム	EX 1	EX 2	EX 3
入力したトークン数	280	250	388
入力した内部コード(B)	445	402	602
（宣言部	136	111	181
実行部）	309	291	421
出力したトークン数	278	265	396
出力したコードのサイズ(B)	634	589	933
（宣言部	141	112	153
実行部）	493	477	780
実行クロック数	1,692	1,544	2,391

やレジスタファイルはほとんど使われておらず，この処理装置の記憶リソースはLexical Processorとして不用なものであった．

(2) Declaration Parserの評価

図5に示した有限状態マシンの処理装置でDeclaration Parserを実現したものをISPSで記述し，(1)のLexical Processorの出力（内部コード）の構文解析を行なった結果を表2に示す．

この結果，Declaration Parserは入力したトークンの数とほぼ同数のトークンを出力しているが，1トークンあたりの所用時間は平均6クロックであり，Lexical Processorが1トークンを出力するのに平均83クロックを要している事と比較して大きなスピードのアンバランスが生じている．また，Declaration Parserは何のオペレーションもしないプログラム実行部のトークンに対し，各トークンごとに命令コードを付加しているため，データ転送量が2倍近くに増加しているという現象があらわれている．

Declaration Parserで使用したPLAの項数は，X-State PLAが176項，Y-State PLAが28項，OP-Code PLAが117項，Control PLAが119項とかなりの項数を要している事がわかる．

6. あとがき

本報告ではPascalのソースプログラムをP-codeにコンパイルする過程をハードウェアで実現しようとする試みについて述べた．現在，ハードウェア記述言語によるシミュレーション実験を行なっているが，複数の処理装置に固定したタスクを割当てているため，処理時間のアンバランスが生じ，最大性能がfullに発揮されない等の問題が判明してきている．さらに，全処理装置が同時に動作した時の共通バス，List Memoryのアクセス競合の問題もあるかと思われ，今後，各処理装置の方式，データトークンの形式等に十分考慮した設計を行なう必要がある．また，VLSI化を指向した方式としてPLAを全面に使用したアーキテクチャを採っているが，現実のMOSプロセスとの適合も考慮して，再度，詳細に検討を進めていきたいと考えている．

《 参考文献 》

(1) N. Wirth, "Algorithms + Data Structures = Programs", Prentics-Hall, 1976

(2) D.R. Ditzel and D.A. Patterson, "Retrospective on Higher-Level Language Computer Architecture", Proc. of 7th Annual Sympo. on Computer Architecture, 1980

(3) T. Ito and A. Fusaoka, "A Compiler-compiler Machine", Digital Process, Vol. 1, 1977

(4) F. Andre, et. al., "KENSUR: An Architecture oriented towards Programming Languages Translation", Proc. of 7th Annual Sympo. on Computer Architecture, 1980

(5) K.V. Nori, et. al., "The PASCAL <P> Compiler: Implementation Notes", Tech. Report of Institut fur Informatik, ETH, July 1976

(6) A.J. Korenjack, J.E. Hopcroft, "Simple deterministic languages", Proc. IEEE 7th Annu. Symp. of Switching and Automata Theory, pp. 36-46, 1966

(7) J.P. Gray, "Introdaction to silicon compilation", 16th Design Automation Conf. Proceedings, pp. 305-306, June, 1979

(8) R.W. Floyd and J.D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits", Stanford Computer Science Dep. Report No. STAN-CS-80-798, April, 1980

(9) M.R. Barbacci, "The ISPS Computer description language', Carnegii-Mellon Univ. Technical Report CMU-CS-79-137, Aug. 1977