# 高級言語向きスタックマシンに関する研究 : Forthマシンと Pascalマシンの実現と評価

和田, 耕一

(Degree)
博士（学術）

(Date of Degree)
1984-03-31

(Date of Publication)
2008-11-06

(Resource Type)
doctoral thesis

(Report Number)
甲0465

(URL)
https://hdl.handle.net/20.500.14094/D1000465

# 博　士　論　文

# 高級言語向きスタックマシンに関する研究

—Forth マシンと Pascal マシンの実現と評価—

昭和58年12月

神戸大学大学院自然科学研究科

和　田　耕　一

# 目　　次

# 第1章　緒論

従来の計算機において、高級言語で記述されたプログラムは、コンパイラによってその計算機の機械語に翻訳された後、実行される[1][2][3]。現在までのいわゆるフォン・ノイマン型のコンピュータが直接実行できる機械語は、メモリの参照命令や単純な算術演算、論理演算などの命令セットで成立しており、1946年に開発されたENIAC以来大きな変化はない。一方、高級言語は応用分野の特性に則して言語設計がなされており、多くの場合計算機の構造に対応付けることが困難である。このため、高級言語による一つの文を翻訳した場合、数十から数百の機械語命令に展開され、プロセッサ内およびプロセッサと主記憶間において冗長なデータ転送が行われるなど、実行時の効率が悪いという欠点が生ずる。この原因は、原始言語である高級言語とそれを翻訳することによって生成される目的コードである機械語との意味的な差が大きい点にある[4][5][6]。

その他、フォン・ノイマン型のコンピュータの特徴として、以下の点が挙げられる。①命令とデータに明確な区別がない。命令語をデータとして参照することもでき、逆にデータを命令と解して実行することもできる。②メモリには順序付けられた番地が与えられており、一次元の構成になっている。③デ

ータは意味を備えていない。例えば、文字を表す語と浮動小数点数のような数値を表す語との間には何ら物理的な差異がなく、その識別はプログラムの論理構造で与えられる。

これらに対して高級言語の特徴は、以下の様である。①命令とデータは明確に区別されている。②変数は認別名で参照され、変数が割り当てられる主記憶空間は、変数の離散的な集合で成立している。すなわち、変数が順序立てて並んでいる必要は全くない。相異なる2つの手続き中の名前の異なる変数が同一記憶空間に割り当てられることも想定していない。また、高級言語で取り扱うデータ構造は線形な一次元構造ばかりでなく、リスト構造や多次元配列など多岐にわたる。③高級言語で扱うデータには、全て意味が与えられている。

以上の様に、フォン・ノイマン型コンピュータの構造と高級言語の概念との間には大きな差異が存在する。そこで、高級言語の言語構造および言語処理系の構造を考慮して、それらに計算機の構造を適合させ、かつ機械語のレベルを上げることによって処理効率の改善をはかろうというものが高級言語マシンである[4][5][6][7]。

高級言語マシンの目的と利点は、①実行速度の向上、②ソフトウェアの生産性向上、③言語処理系の高効率化、などが挙げられる。すなわち、機械語の

レベルを高くすることにより、命令の読み出し回数を大幅に減少させることが可能になると共に、言語機能に適したデータ転送機能やメモリ参照機能を計算機に持たせることにより、実行速度の向上が期待できる。また、フォン・ノイマン型コンピュータでは原始プログラムとそれを翻訳して生成された機械語との隔たりが大きいため、実行結果が正しくないなど実行時に誤りが生じた場合、その状況と原始プログラムとの対応がとりにくく、プログラムの検査、修正が非常に困難である。高級言語マシンでは、機械語のレベルが原始言語に近いため前述の対応付けが容易で、未定義の変数や値が設定されていない変数を参照したときの誤りや、配列の添字の範囲が限界を越える誤りなど、実行時の誤り検出の機能を容易に組み込むこともでき、プログラムの生産性や信頼性を向上させることができる。また機械語の機能が対象高級言語の機能を反映しているため、言語処理過程が簡単化される。言語の翻訳過程は、語彙解析、構文解析、コード生成の3つに大別される[1][2]。このうち、構文解析までは言語構造に依存するが、コード生成は計算機構造に依存する。従って、高級言語マシンにおいてはコード生成部が簡単化、もしくは省略できる。この利点は、特に計算機の会話的利用環境において有効で、プログラムの修正、翻訳、実行のサイクルを短縮できる。

この様に高級言語マシンは数々の利点を備え、発展してきたが、その背景にはハードウェア技術、マイクロプログラム制御技術の著しい発達[8][9][10][11][12][19][52]がある。計算機システムの開発コストについては、ハードウェアの開発コストとソフトウェアの開発コストを考慮に入れねばならない。近年、半導体集積回路の集積度が飛躍的な向上を見せたため、半導体素子のゲート当りの単位が急激に低下し、ハードウェアのコストが著しく下がった。また、高級言語マシンの様に、高機能なハードウェアを備えた計算機を設計する場合、従来一般的に用いられてきた布線論理によると、制御回路が非常に複雑になり、設計が困難である。1951年にM.V.Wilkesが提案したマイクロプログラム制御方式は、計算機内の制御回路を規則的なハードウェアで構成することにより、制御回路を単純化しようとするもので、現在では高機能計算機の実現に不可欠な技術になっており、高級言語マシン開発にとって極めて有力な手段となっている。マイクロプログラム制御方式では、順序制御情報を保持する制御記憶に、論理素子の動作速度に見合った高速読み出し可能なメモリが要求されるが、近年の半導体技術の発達を基盤として高速メモリが利用可能となり、マイクロプログラム技術が大きく推進される結果となった。最近では、書き換え可能な高速メモリも開発され、実験機や商用機に幅

広く利用されており．動的にマイクロプログラムを書き換える方式なども提案されている[18]。

高級言語マシンの概念はすでに1960年初頭に提案され、1961年には商用機としてバローズのB5000[14]が発表されている。B5000は．Algol言語を対象とした高級言語マシンで、ハードウェアスタックやデータの属性や手続きなどの制御情報を表現する記述子を導入するなどの特徴を持っており、後年の高級言語マシンに大きな影響を与えている。続いて1967年にIBM 360モデル30にEULER言語を対象にしたEULERマシンがマイクログラムによって開発された。1970年代には、バローズのB1700，IBMの5100，ヒューレットパッカードのHP2100など数々の商用機が発表された。さらに、リスト構造を持つデータを扱う記号処理言語であるLISP言語を対象とした高級言語マシンも数多く発表され、いくつかの成果が見られる[16][17]。

近年、制御分野をはじめ、画像処理．データベース．他の言語プロセッサの開発等の分野でForth言語[20][21][31][36]が注目され．広く使われている。Forthは、プログラムが本質的にモジュール化されていて．各モジュールは単独で実行可能であるため．プログラムの開発が非常に容易である点や、所要メモリが少ない等．数多くの長所を備えた言語である．従来の計算機上でのForth処理系は．算術．論理演算の

中心となって働く演算用スタックとプログラムの実行順序制御用スタックをそれぞれソフトウェアで実現しており、スタック機構を中心とした仮想スタックマシンの構造を持っている。

スタックマシン[23][24][25][26][49][50]は、①多くの命令のオペランドがスタックに限定されていて、アドレスを表す部分が不要であるため、所要メモリが少なくて済み、メモリとのデータ転送量も減少する。②レジスタの管理を行う必要がなく、目的コードの生成が単純である。③関数の帰納的定義に容易に対応できる、などの利点を持っている。また、原始プログラムを翻訳、実行する過程で、機械語よりも機能レベルの高い中間コードを用いる方式を採っている高級言語の処理系では、多くの場合、スタックマシンを想定した中間コードを生成しており、スタックマシンは、高級言語と計算機構造のレベル差を縮める効果的な方法といわれている。しかし、従来の計算機上で、ソフトウェアによってスタックマシンを構成すると、スタックの管理における効率が悪く、実行速度も遅いという欠点が生じる。

Forth言語の処理系は、スタック機能に対する依存性が非常に高く、Forth言語を手掛りとして設計された計算機は、前述の2つのスタックを有するスタックマシンの構造を持ち、他言語に対しても有効に動作し得ると考えられる。

本研究では、高級言語の実行時に非常に重要な役割りを果たすスタック機能に注目して、演算用スタックと実行制御用スタックの2つのスタックを中心とした処理系の構造を持つForth言語を主対象言語に選んだ高級言語マシン（本研究ではHigh Level Language Engineと呼び、以下HLLエンジンと記す。）を開発した。さらに、本マシン上にPascal[27][28]プログラムを高速に実行できるPascalマシンを実現する方法について論じ、Forthマシン、Pascalマシンそれぞれについての性能評価を行うものである。

第2章では、Forth言語の特徴と言語処理系の構造について述べ、HLLエンジンシステムの基本設計について検討する。設計にあたって、まず従来の計算機がForthプログラムの実行に不向きである点について考察する。特にForth言語の重要な機能であるスタックをマイクロコンピュータ上に、ソフトウェアで実現し、ベンチマークプログラムによるテストを行うことによって、Forthプログラムの高速実行における高機能スタックの必要性について考察する。

第3章では、2章で述べた基本方針に従って設計されたHLLエンジンシステムの全体構成と、HLLエンジン内のハードウェアモジュールとその機能について述べる。特に高速化のために備えた特殊な構造を持つハードウェアスタックや変数専用高速メモリ

については、備えた根拠を明らかにし、構造と機能について詳しく述べる。また、HLLエンジンとそれをサポートするホストコンピュータとのインタフェースについて述べ、ホストコンピュータのアドレス割り当てについて述べる。

第4章では、Forthプログラムを高速に効率良く実行できるForthマシンをHLLエンジン上に実現する方法について述べる。また、Forth言語処理系に検討を加え、本システムに適合した処理系の開発について述べる。次に、HLLエンジンの実行中に入出力などの要求が生じたときに必要なホストコンピュータ上の割り込み処理について述べ、Forthプログラム中で利用できるマイクロアセンブラや直接マイクロコードを生成するコンパイラについて述べる。さらに、ベンチマークテストを用いて本Forthマシンの実行時間の測定を行い、ベンチマークプログラムに対するマイクロ目的コードを解析し、本Forthマシンの有効性について評価を加える。

第5章では、Pascal言語の特徴と言語処理系の構造について述べ、HLLエンジンの構造とPascal言語との整合性について考察する。そして、従来の計算機上でPascalプログラムを実行する場合の不都合な点について検討し、HLLエンジン上にPascalマシンを実現する方法について考察する。また、Pascalプログラムを翻訳して得られる中間コードに対して検

討を加え、本エンジンのマイクロ命令の機能を考慮に入れた、より効率の良い機能を持つ新中間コードの設計と、新中間コードを生成するPascalコンパイラの開発について述べる。さらに、Pascalマシンを実現するためのソフトウェアである、マイクロプログラム化インタプリタと、新中間コードからマイクロ目的コードを生成するコンパイラの開発について述べる。最後に、ベンチマークテストを用いたPascalマシンの実行時間の測定を行い、マイクロ目的コードに詳細な検討を加えることにより、計算機構造に適合した新中間コードの機能や本Pascalマシンの有効性について、静的および動的評価を行う。

# 第２章　HLL エンジンシステムの設計

## 2.1　緒言

高級言語マシンを実現する際の、一つの形態としてスタックマシンがある。スタックマシンは、スタック機構を中心とした構造を持つ計算機であり、レジスタの管理が不要で、関数の帰納的定義に容易に対応できるなどの利点を有している。

本HLLエンジンは、強力なスタックマシンとしての機能を持たせるために、Forth言語[35]…[40]を基礎において設計を行った。Forth言語処理系[29][30]は、内部に演算用スタックと実行順序制御用スタックの２種類のスタックを持っており、Forthプログラムの記述も、逆ポーランド形式と呼ばれる演算子後置形式で行う。この様に、Forth言語は、スタック機構をユーザに陽に開放し、スタック機能に対する依存性が非常に高いという点で、極だった特徴を持つ言語であり、近年様々な用途に広く用いられている。

本章では、まずHLLエンジンシステム設計の基礎となるForth言語の特徴と処理系の構造について述べる。次に従来の計算機上でForthプログラムを実行する上で効率が悪い点について考察し、特に重要と考えられるスタック機能については定量的な測定を行うことにより、強力なスタック機能の必要性を

明らかにする。次に以上の点に留意し、HLLエンジンの基本的な設計方針について述べる[32][33][34]。

## 2.2 Forth言語の特徴と処理系の構造

プログラミング言語Forthは、1969年Charles. H. Mooreにより開発された関数型言語で、高い移植性や必要メモリ容量が少ない等の利点を備えている。1台のミニコンピュータによりプロセスコントロールとデータ収集，解析を行う必要から考案された言語であるが、現在ではデータベースをはじめ画像処理や他の言語プロセッサの開発まで広く利用されている。

本節では、HLLエンジン設計の基礎となるForth言語の特徴について述べ、言語処理系の構造について詳述する[20][21][29][30][31]。

### 2.2.1 Forth言語の特徴

Forth言語は、次の様な特徴を持っている。

(1)会話型の関数型言語である。

ユーザはターミナルを介してForthシステムと会話しながらプログラムの作成とデバッグを行うことができる。また、Forthではあるまとまった処理に対して名前付けを行い、それをワードと呼び、ワードの集合をディクショナリと呼んでいる。

作成されたプログラムは、本質的にモジュール化され、かつ階層化されており、ソフトウェア工学で提唱されている構造化設計技法の導入がきわめて容易である。

(2)自己増殖型である。

Forth言語を用いたプログラム開発は、定義済みのワードを参照して新たなワードを定義することの繰り返しで行う。そして、最終的に一つのワードがユーザジョブの全体を担当するまで定義を続ける。それぞれのワードの定義は、多くの場合1～3行程度で、定義されるとただちにコンパイルされ、そのワードは実行可能となる。

(3)プログラムの一部をアセンブリ言語で記述することも可能である。

アセンブリ言語のような低レベルの言語を用いたワードの定義が可能である。すなわち、リアルタイム処理のような実行時間に厳しい制約のつくルーチンや、特殊なデバイスの制御を行うルーチンなどをアセンブリ言語を用いて記述し、ワードとして利用することができる。

(4)所要メモリが少ない。

コンパイル結果のオブジェクトコードが非常にコンパクトであり、アセンブリ言語を用いた場合よりも所要メモリが少なくて済む。この理由としては、スタックの利用による変数領域の減少、プ

ログラムの階層化によるワードの有効利用があげられる。また、Forthシステム自身約7Kバイトと非常にコンパクトで、制御機器組込み用として用いるときは実行中ルーチンのみで済み、その場合約800バイトまで小型化できる。

(5)リターンスタック・パラメータスタックの2種類のスタックを持つ。

Forthシステムは、ワード呼び出しの制御や繰り返し制御文のインデクスとリミットの格納に使われるリターンスタックと、あらゆる演算に対するソースとデスティネーションの格納場所として用いられるパラメータスタックの2種類のスタックを持っている。

### 2.2.2 Forth言語処理系の構造

次に、Forth言語の内部構造の大きな特徴の一つであるディクショナリエントリとディクショナリの構成について述べ、定義されたワードをコンパイルするテキストインタプリタと、実行時の制御を行うアドレスインタプリタについて述べる。

(1)ディクショナリ

新たにワードを定義することにより、そのワードに対してディクショナリエントリと呼ばれる目的コードの一種が生成される。ディクショナリは、ディクショナリ・エントリの連結リストであり、エント

り間はヘッダ部にあるリンク・フィールドによって連結されている。図2.1に、ディクショナリの構成を示す。特定のワードをディクショナリ内から捜し出すときは、最新のものから捜し始めるため、同じ名前をもつワードを再定義したときは、最新のものが有効になる。

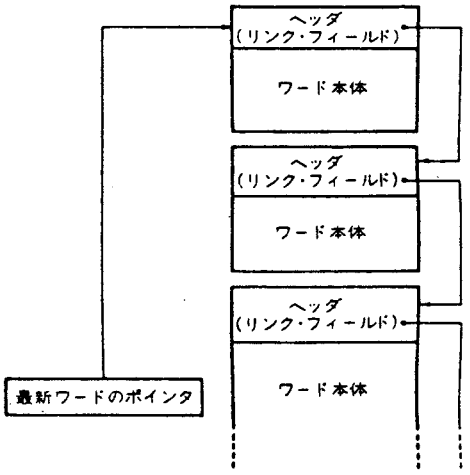



図2．1　ディクショナリの構成

またForthにはボキャブラリという概念があり、単一の連結リストのみでなく、エディタやアセンブラなどをたがいに異なる連結リストとし、このようなディクショナリのサブセットを切り換えることによって、用途に応じたワード群を選択することができる。

Forth処理系では、インタプリタ類やコンパイラなども大部分がワードの形でディクショナリ内に存在しており、そのようなシステム用のワードとユーザが定義したワードは同レベルのものとして取り扱うことができる。したがって、ユーザがコンパイラやOSなどに手を加えることも容易で、このこともForthの大きな特徴になっている。

(2)ディクショナリエントリ

ディクショナリエントリは、大きくわけてヘッダ部とワード本体から成り立っている。ヘッダ部は、識別用のネームフィールドとディクショナリを形成するためのリンクフィールドから成っており、主としてディクショナリの管理に用いられる。

ネームフィールドにあるプレシデンスビットは、コンパイル時であっても、そのワードが参照されたときには強制的に実行されるワードに対してセットされる。実際にはディクショナリエントリ中にある領域を確保する等、主としてコンパイル時に特殊な操作を行うワードを定義する場合に用いられる。リンクフィールドは、最新の定義済みワードへのポインタで、連結リストであるディクショナリを構成するために用いられる。

ワード本体は、そのワードが既に定義済みのワードを参照して新たに定義されたものについては、参照されたワードへのアドレスポインタ列で成立しており、アセンブリ言語を用いて定義したワードであれば、機械語命令列で成立している。

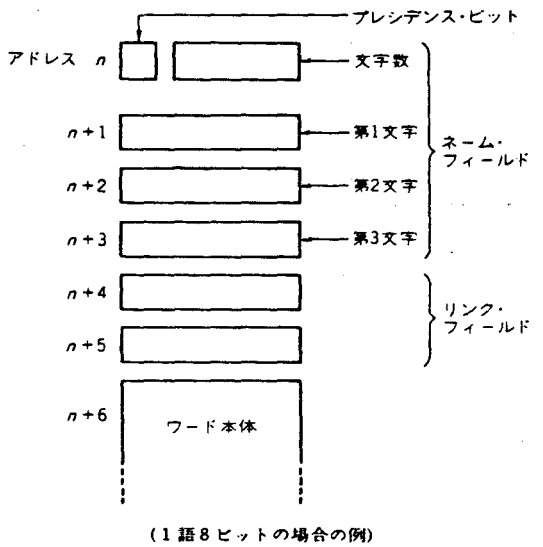



図2．2 ディクショナリエントリのヘッダ部の構成

図2.2にディクショナリエントリのヘッダ部の構成例を示す。

(3)テキストインタプリタ

テキストインタプリタは、アウター・インタプリタとも呼ばれ、Forth処理系の中核をなす部分で、コンソールやディスク・ファイルからの入力ストリングの解析を行なう。テキスト・インタプリタのフローチャートを図2.3に示す。

まず、入力ストリングの中からワードを順にひろいだし、実行モードの場合は実行すべきワードに対

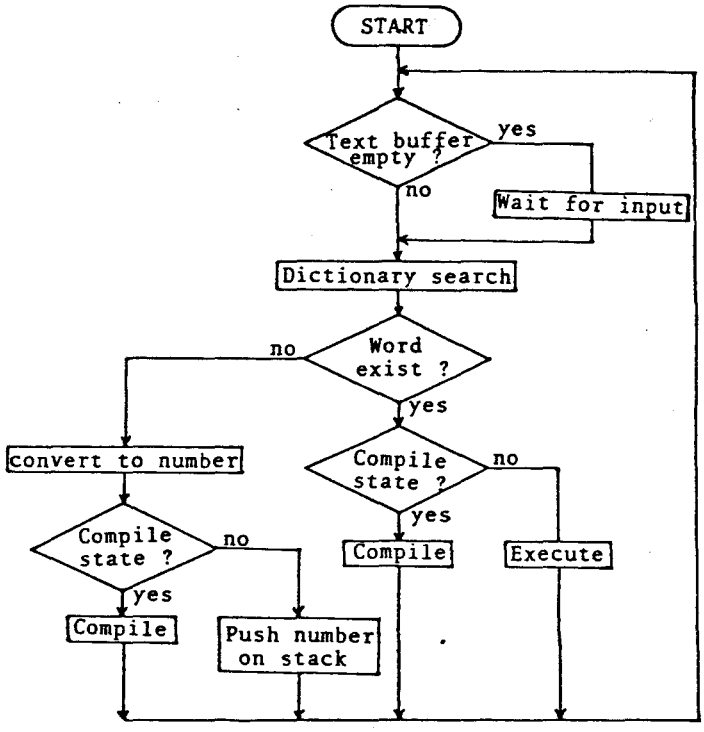



図2．3　テキストインタプリタのフローチャート

するディクショナリ・エントリをディクショナリの中から捜し、そのアドレスとともに制御をアドレス・インタプリタに引きわたす。実行終了後は、再びテキスト・インタプリタの制御にもどり、つぎに拾いだされた入力ワードに対して同様の処理を行なう。

コンパイル・モードにあるときは、同様に参照されたワードに対するディクショナリ・エントリをディクショナリ中から捜し出し、そのアドレスを現在作成中のディクショナリ・エントリのパラメータ・フィールドに積む処理を行なう。いずれのモードにおいても、ディクショナリ中に該当するエントリが見つからない場合には、拾い出されたワードが数値かどうかのチェックが行なわれる。

上に述べたようにForth言語のコンパイルは非常に簡単で、特定のエントリをディクショナリ中から捜し出すのに大半が費やされる。そこで、コンパイルを高速に行なうために、定義名の一部に対してハッシュ関数を用いることによってディクショナリを分割し、複数の連結リストでディクショナリを構成しているForth処理系もある。

(4)アドレスインタプリタ

アドレスインタプリタは、インナー・インタプリタとも呼ばれ、ワードの実行に直接関与するものである。インタプリタといっても、文字列の解釈などを行なうわけではなく、ディクショナリ・エントリ

中のワード本体部分にあるアドレス・ポインタによる間接ジャンプの制御を行なうだけなので、高速実行が可能になっている。

実行時においては、次に実行すべきワードを記憶しておくためにアドレス・ポインタ部を指示するカウンタ（インストラクション・カウンタと呼ぶ場合もある）を持っている。Forthでは一般に多重のワード呼び出しが行なわれるので、このインストラクション・カウンタを退避させる必要がある。退避させたいときに用いられるのがリターン・スタックで、アドレス・インタプリタが間接ジャンプの制御とともにインストラクション・カウンタのプッシュ，ポップを行なう。アセンブリ言語を用いて定義されたワードの実行については、ディクショナリエントリの本体が機械語命令で展開されているため、アドレスインタプリタの制御を離れて実行が行なわれる。

## 2.3 HLLエンジンシステムの基本設計

Forth言語で記述されたプログラムを実行する場合、従来の計算機は次の点で不向きである。

1. パラメータスタックの負担が非常に大きい。
2. ワード呼出し時の間接ジャンプ及びリターンの処理が遅い。

表２．１　フィボナッチ数列の計算に要する
実行時間の測定結果

（単位はｓｅｃ）

| アセンブリ言語プログラム | | Ｆｏｒｔｈプログラム |
|---|---|---|
| レジスタのみでの演算 | スタックを用いた演算 | |
| １４．５２８ | １４３．２１６ | ２７９．２１５ |

これらの裏付けを得るためにZ-80マイクロコンピュータ（クロックは２MHz）上でベンチマークテストを行い、その結果を表2.1に示す。Forth言語はpoly-Forthを用いた。ベンチマークプログラムはフィボナッチ数列の計算で、10,000回のループを100回繰り返し、けたあふれは無視している。表では、まずアセンブリ言語を用いてレジスタのみを有効に用いて計算を行った場合と、スタックを仮定してデータの動きがForthプログラムと同等になるようにアセンブリ言語で記述した場合と、更にForthプログラムによって計算した場合の実行時間をそれぞれ測定し結果を示している。表から、レジスタのみによる演算に対し、スタック上で演算を行うと実行速度が約1/10に低下し、ワード呼出しを伴うForthでは約1/20に低下していることが分かる。

以上のほか、

3. 主記憶中の命令語やデータのアクセスひん発によるボトルネックの発生.
4. 乗算をプログラムで行うことによる低速性.

などが問題として上げられる.

以上の点に注目し、本研究ではForthプログラムを高速に実行できる構造を持ったHLLエンジンシステムの設計, 試作を行っている.

プロセッサ部の設計に当って、システムに柔軟性を持たせ、比較的短期間で試作、デバッグを完了させるため、次のような方式を採用することにした。

1. マイクロプログラム制御方式とする.
2. ビットスライスのバイポーラマイクロプロセッサを用いる.
3. 市販のマイクロコンピュータシステムにより、HLLエンジンをサポートし、初期のデバッグ効率を上げると共に、実行時の処理負担の分散を図る.

又、前述のボトルネックに対応してHLLエンジン及びシステムに次のような機能とハードウェアモジュールを持たせることにした。

4. ハードウェアによる強力なパラメータスタッ

ク，リターンスタックを設ける．スタックから演算部やメモリ等に効率良くデータを送れるような内部バス構造を持たせる．

5. マイクロルーチンにForthのディクショナリ構造を持たせ，ユーザプログラムをすべてマイクロプログラムにコンパイルする．又，マイクロインストラクションのフェッチにはパイプライン制御を行うことにより命令の先読みを行う．
6. HLLエンジン内に高速メモリを持たせ，変数や配列の取り扱いを高速化する．
7. 乗算用LSIを用いる．
8. HLLエンジン内におけるデータ転送専用のハードウェアを設ける．

特に5.について補足すると，一般に，高級言語をマイクロプログラムにコンパイルして処理する手法は，実行速度の点で有利であるが，コンパイラの負担が極めて大きいという欠点がある．しかし，Forthにおけるコンパイルは非常に単純かつ容易であり，テキストインタプリタは，マイクロプログラムのような低レベルのルーチン群をも強力に管理する機能を持っているため，マイクロルーチンにディクショナリ構造を持たせることにより，この手法を効率良く実現できる可能性を見い出せた．すなわち，本マシンは，従来の意味での主記憶は持っていない．ユ

ーザが定義したワードは、マイクロインストラクションにコンパイルされ、WCS†上のディクショナリにディクショナリエントリとして付加される。

## 2.4 結言

本章では、本HLLエンジンシステムの主対象言語であるプログラミング言語Forthの数々の特徴と、言語処理系の構造について述べた。さらに、従来の計算機上でForthプログラムを実行した場合に生じるボトルネックについて検討した。特にスタック機構に関しては、レジスタのみを用いての演算に対し、スタックを用いることにより実行速度が約1/10に、その上にスタックを利用して関数呼出しを行うと約1/20に低下することをベンチマークテストを行うことにより明らかにした。それらの検討の結果、Forthプログラムを高速に効率良く実行するためにHLLエンジンが備えるべきハードウェアについて以下の様に決定した。

1) 強力な機能を持つパラメータスタック、リターンスタックを備える。及びそれらのスタックを有効に利用できるバス構造を持たせる。
2) 変数や配列専用の高速メモリを備える。

† Writable Control Storage (書込み可能な制御記憶)

3) 乗算用LSIを用いる.

4) データ転送用ハードウェアを持たせる.

また、Forth言語の記述機能を生かして、ユーザプログラムは全てマイクロプログラムにコンパイルする等、システム構築に関する基本的方針について述べた.

# 第3章　HLLエンジンシステムの構成

## 3.1　緒言

前章で述べた方針に従って設計されたHLLエンジンシステムの構成について詳述する。

まず、HLLエンジンとそれをサポートするホストコンピュータを含めたシステム全体の構成について述べる。次にForthプログラムを高速に実行できる様に備えられたHLLエンジン内部の個々のハードウェアモジュールの機能や容量について述べる。さらに、HLLエンジンのマイクロ命令の構成について述べる。最後に、本システムのハードウェアサイズと実装状態について述べる。

## 3.2　HLLエンジンシステムの全体構成

図3,1にHLLエンジンシステムの構成を示す。構成としては、Z-80をCPUとするマイクロコンピュータシステムに、3種類のレジスタとメモリインタフェースを通じてHLLエンジンが接続されている。Z-80側からは、データメモリとWCSがアクセス可能で、32K Byteを1ページとしてページ切換えを行うことにより、主記憶の高位32K Byteにそれぞれが接続される。HLLエンジン側からは、データメモリは1語

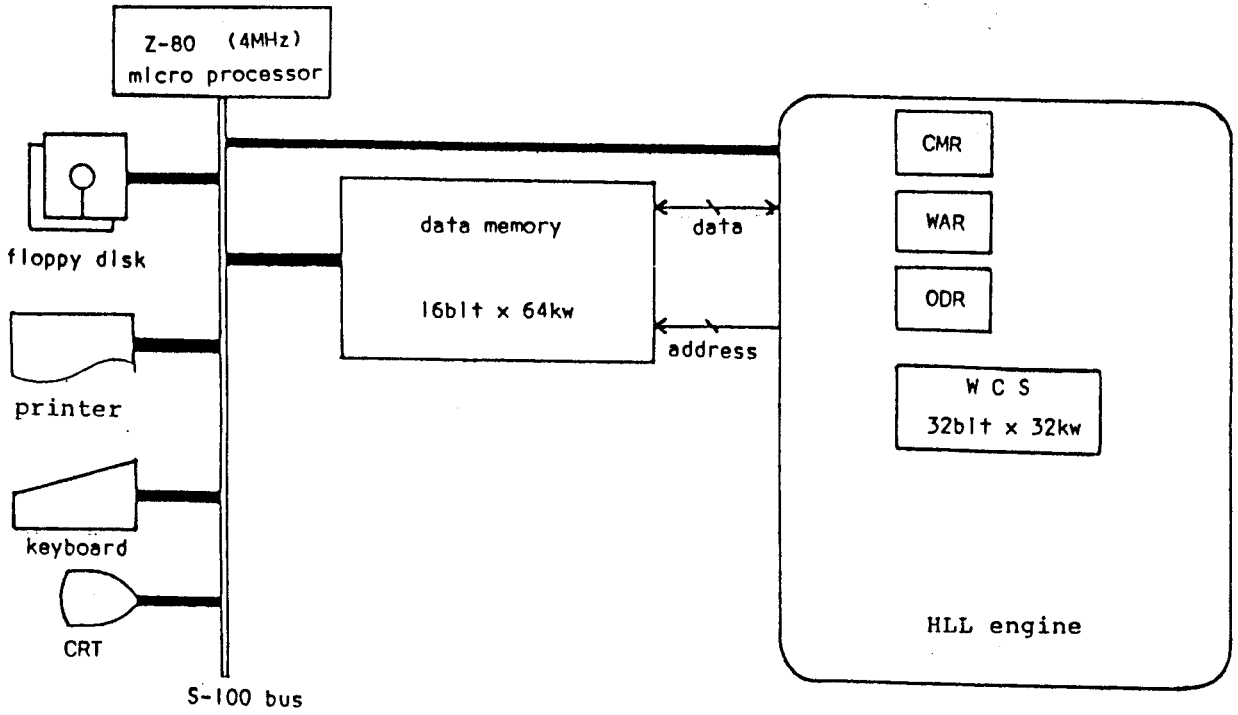



図3．1　ＨＬＬエンジンシステムの全体構成

16 bit, 64K語を占めている。つまり、データメモリは共有メモリの形で存在し、Z-80とHLLエンジンのいずれの側からもアクセス可能である。

システムの起動時は、Z-80がHLLエンジンに対して初期化を行う。テキストインタプリタはZ-80上に作成し、入出力の処理やファイルの管理はZ-80が担当する。Z-80から見ると、HLLエンジンはI/Oデバイスであり、参照できるレジスタは、コマンドレジスタ(CMR)，ワードアドレスレジスタ(WAR)，アウトプットデータレジスタ(ODR)の三つで、それぞれ16 bit 幅である。CMRは、HLLエンジンの制御のためのレジスタで、WARは、実行すべきワードのアドレスの引き渡しに用いられるレジスタである。すなわち、実行は、Z-80が実行すべきワードのアドレスをWARに書き込み、CMRに実行指令を与えるという手順で行われる。そして、処理結果は、ODRを通じてZ-80に渡される。

### 3.2.1 ホストコンピュータとのインタフェース

HLLエンジンは、ホストコンピュータであるZ-80マイクロコンピュータシステムにI/Oポートを介して接続されていることは、既に述べた。ここでは、HLLエンジンの制御用レジスタであるCMRの各ビットの機能について詳しく述べる。CMRの各ビットと、制御命令との対応を表3.1に示す。内容について詳し

表３．１　コマンドレジスタの各ビットの機能

| | | |
|---|---|---|
| Bit 7 | --- | unused |
| Bit 6 | read/write | MAINTENANCE |
| Bit 5 | read/write | Z-80 INTERRUPT MASK |
| Bit 4 | read/write | FORTH MACHINE INTERRUPT MASK |
| Bit 3 | read | RUN/HALT (MACHINE STATE) |
| Bit 2 | read/write | RUN/HALT (COMMAND) |
| Bit 1 | write | EXECUTE (JUMP ZERO) |
| Bit 0 | write | STEP |

く述べると、以下の様である。

Bit 6 MAINTENANCE

このビットは、メンテナンスモードを指定するビットで、１をセットすることによりHLLエンジンは停止し、WCSがZ-80からアクセス可能になる。

Bit 5 Z-80 INTERRUPT MASK

HLLエンジンからZ-80へかける割り込みをマスクするビットで、１をセットすることにより割り込みがマスクされる。

Bit 4 FORTH MACHINE INTERRUPT MASK

Z-80や外部事象、エンジン内のハードウェアエラー（スタックのオーバーフローなど）によるHLLエンジンへの割り込みのマスクビットで、１をセットすることにより割り込みがマスクされる。

Bit 3 RUN/HALT （MACHINE STATE）

HLLエンジンが実際にRUNモードにあるのか、HALTモードにあるのかという状態を知るためのビットで、読み出しのみが可能である。

Bit 2　RUN/HALT（COMMAND）

Forthプロセッサに対して、実行や停止の制御を行うもので、メンテナンスパネルスイッチがRUNに設定されている時のみ有効である。このビットを1とすると実行モード、0とすると停止モードを指定することになる。つまり、このビットを1とし、メンテナンスビットを0、パネルスイッチをRUNとした時のみ実行モードに入り、Bit3が1となる。

Bit 1　EXECUTE

1を書き込むことにより、マイクロアドレスとして0番地が強制的にWCSに与えられる。Forthプログラムのワードなどを実行する時などに用いる。

Bit 0　STEP

HLLエンジンのステップ実行を行うためのビットで、1を書き込むことにより1ステップ実行される。HLLエンジンがHARTモードにあるときのみ有効で、メンテナンスパネルのステップスイッチによってもステップ実行が可能である。

CMRを含めて、Z-80からアクセス可能なレジスタ類のポートアドレスを表3.2に示す。タイマーは、HLLエンジンの評価に用いるカウンタで、HLLエンジンの実行時間をmsecの単位で計測することがで

表3．2　レジスタとポートアドレス

| ポートアドレス | モード | 機　　能 |
|---|---|---|
| D2 | OUT | ページレジスタ |
| C3 | IN | タイマデータ |
| C2 | IN／OUT | タイマデータ　／　タイマストップ |
| C1 | IN／OUT | タイマデータ　／　タイマスタート |
| C0 | IN／OUT | タイマデータ　／　タイマクリア |
| 05 | OUT | HLLエンジン初期化 |
| 04 | IN | ODR　(上位バイト) |
| 03 | IN | ODR　(下位バイト) |
| 02 | IN／OUT | WAR　(上位バイト) |
| 01 | IN／OUT | WAR　(下位バイト) |
| 00 | IN／OUT | CMR |

注）ポートアドレスは、16進数

| | bit 31 | | | bit 0 |
|---|---|---|---|---|
| ADDRESS 0 | 8000 (16) | 8001 (16) | 8002 (16) | 8003 (16) |
| ADDRESS 1 | 8004 (16) | 8005 (16) | 8006 (16) | 8007 (16) |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

図3．2　WCSのアドレス割当て

きる。

### 3.2.2 ホストコンピュータのアドレス割り当て

Z-80は、自身の主記憶としてアドレス0000～7FFF(16)の32Kバイト持っている。アドレス空間の上位32Kバイトは、32Kバイトを1ページとするアドレス多重化領域となっている。Z-80のI/Oポートの1つにページレジスタが接続されており、このレジスタを用いてページを切り換えることにより、データメモリやWCSがアクセスできる。

ページ配分は

0ページ～3ページ　：　データメモリ

4ページ～7ページ　：　WCS

となっている。特に、4ページをページレジスタに書き込んだ場合の、WCS内のZ-80から見たアドレスを図3.2に示す。

## 3.3 HLLエンジンのハードウェア構成

図3.3にHLLエンジンのハードウェア構成を示す。大きく分けて、CCU，インタラプトコントローラ，リターンスタック，演算モジュール，パラメータスタック，バリアブルストレージ，バイパスコントローラから成り立っている。

内部バスは16ビット幅で、ソースバスとして

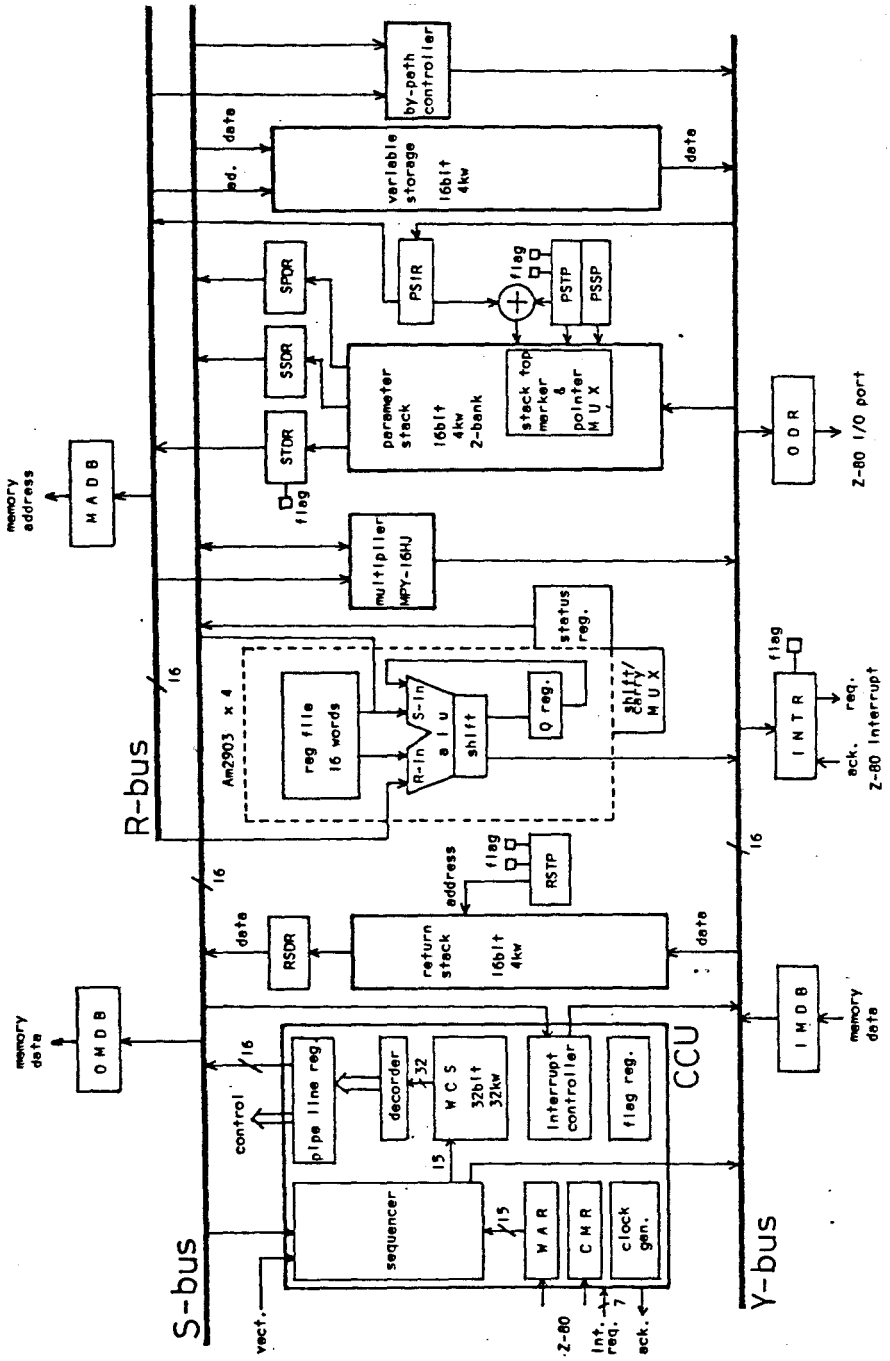



図3．3　ＨＬＬエンジンのハードウェア構成

R-bus と S-bus 、デスティネーションバスとして Y-bus の合計3つのバスがある。各モジュールのバスに対する接続関係は、Forthプログラムを効率良く、高速に処理できる様に定められている。Forthにおけるプログラムの特徴として、パラメータスタック内のデータ操作のためのワードが頻繁に用いられ、その代表的なものとして、

DUP ： パラメータスタックのトップにある値を再度プッシュし、複製を作る。

SWAP ： スタックのトップにある値と、セカンドにある値とを交換する。

OVER ： スタックのセカンドにある値をとり出してプッシュする。

などがある。

またリターンスタックとパラメータスタック間のデータ転送もよく行なわれる処理の一つである。これらの処理をできるだけ少ないステップ数で行える様に、各モジュールとバスとの相互関係を定めてやらねばならない。そこで、図に示した様な構成とすることにより、ほとんどのスタックオペレーションが1マイクロステップで実行できる。

i) CCU

コンピュータコントロールユニット(CCU)は、HLLエンジン内の全てのハードウェアモジュールの

コントロールを行うもので、クロックジェネレータから発生するシステムクロックにより、各モジュールの動作の同期をとっている。CCUは、ワードアドレスタ(WAR)，コマンドレジスタ(CMR)，クロックジェネレータ，マイクロプログラムシーケンサ，WCS，マイクロインストラクションデコーダ，パイプラインレジスタ，インタラプトコントローラなどから成り立っており、本マシンでは、パイプライン制御により、命令の実行と次の命令のフェッチを並行して行なっている。

クロックジェネレータ部は、28.5MHzのオシレータと、アドバンストマイクロデバイス(AMD)社[22]のマイクロサイクルレングスコントローラAm2925から成っている。Am2925は、4相のクロック出力を持ち、8種類のサイクル長を選択できる。また、RUN，HALT，WAIT，STEP，動作のコントロール入力も持っているので、システムクロックのコントロールが非常に簡単に行なえる。

本マシンでは、高速化のため、どのハードウェアモジュールを用いるかによりその時の伝搬遅延時間を考慮に入れてサイクル長を決定し、6種類の長さを適時切り換えて実行する様にしている。この切り換えは、CCU内のデコーダがマイクロインストラクションをデコードし、アクティブとなるハードウェアモジュールを検出して、自動的に行われる。アク

ティブとなるハードウェアモジュールと、対応するマイクロサイクルとその長さの関係を表3.3に示す。

マイクロプログラムシーケンサは、同じくAMD社から数種類でているが、本マシンの32K語の容量のWCSに対応できるアドレススペースを持たないものもあり、他のものもスタックの容量が不足で、スタックの内容を外部に読み出せない、マイクロプログラムカウンタの内容を読み出せない、などの点でForthの処理には不向きである。従って、本マシンでは、ショットキータイプのTTL・ICを中心に構成している。マイクロプログラムシーケンサのブロック図を、図3.4に示す。すなわち、WCSに与えるアド

表3．3　マイクロサイクル

| オペレーション | クロックサイクル | クロック長 (nsec) |
|---|---|---|
| ALU | 7 | 245 |
| ALU WITH (PSIR) | 9 | 305 |
| MPY | 8 | 280 |
| MPY WITH (PSIR) | 10 | 350 |
| BY-PATH | 5 | 175 |
| BY-PATH WITH (PSIR) | 5 | 175 |
| その他のオペレーション | 6 | 210 |

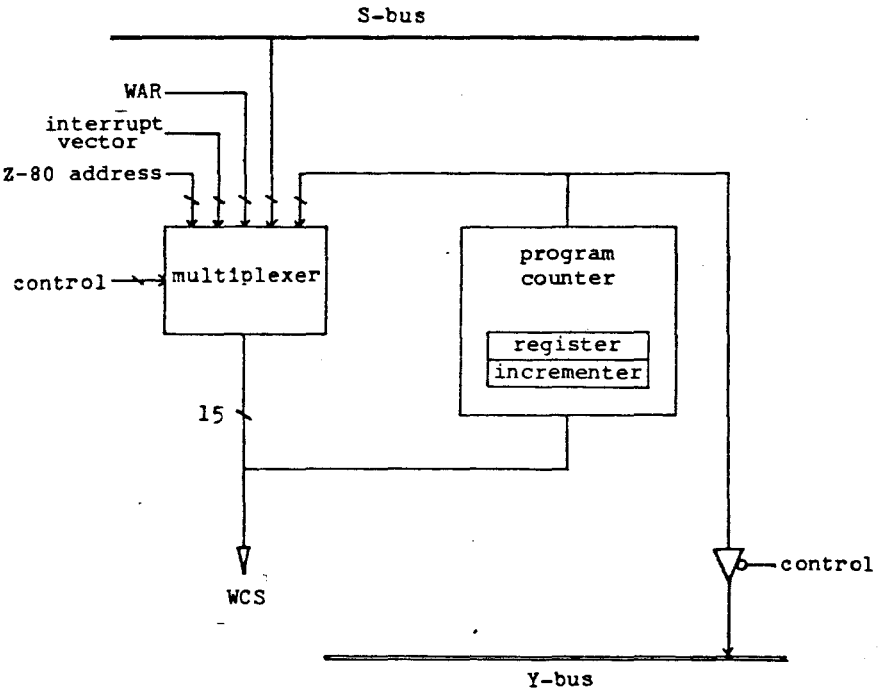



図3．4　マイクロプログラムシーケンサの構成

レスソースとして、実行時には、プログラムカウンタ，S-bus 上のデータ，WAR，インタラプトベクターの4つがあり、マイクロプログラムもしくはインタラプト受け付けにより切り換えられる。メンテナンスモードにある時は、Z-80からのアドレスがWCSに与えられる。アドレス幅は15ビットあり、最大32K語のWCSをアクセスできる。また、プログラムカウンタの値が外部にとり出せる様になっており、サブルーチンジャンプを行う時やインタラプトを受け

付けた時に、Y-busを通じてリターンスタックへプッシュできる。

WCSは、アクセスタイム55msecの4Kビットメモリを用い、1語32ビットで最大32K語まで実装可能である。

マイクロインストラクションデコーダは、WCSからの32ビットの命令を、各ハードウェアのコントロールラインとほぼ1対1に対応する様にデコードするものである。ここでは、32ビットの命令から70本のコントロール出力を得ている。

パイプライン制御を行うにあたって、処理効率について考えると、実行に要する時間と命令フェッチに要する時間の関係が問題になる。つまり、実行を終えた時点で命令のフェッチも完了していなければならず、逆もまた満たされていなければならない。すなわち、両者に要する時間が等しい場合が、最も効率が良いことになる。そこで、このインストラクションデコーダをパイプラインレジスタの前段に置くことにより、実行サイクルを短くでき、マイクロインストラクションのフェッチ+デコードのための時間と、実行に要する時間とを近づけることができるため、効率を上げることができる。

また、このデコーダは、インタラプト受け付け時には強制的にマイクロプログラムカウンタの値をリターンスタックにプッシュし、ベクター・アドレスに

ジャンプする様なコントロール出力を発生する機能を持っている。

インタラプトコントローラは、AMD社のAm2914を用いている。このLSIは次の様な特徴を持っている。

1) 8個のインタラプト入力を持つ。
2) マスクレジスタ、ステータスレジスタを持つ。
3) 16の内部オペレーションを持つ。

本エンジンでは、リターンスタックとパラメータスタックのオーバーフロー、アンダーフローが生じた場合、最上位レベルの割り込みをかけている。

ii) 演算モジュール

R-bus, S-bus, レジスタファイルなどのデータを入力として演算を行い、結果をY-busに出力するモジュールである。

ALUは、AMD社のバイポーラビットスライスプロセッサ, Am2903を使用している。特徴は、

1) 16個のレジスタファイルを持つ。
2) 演算後のデータに対するシフト機能を持っている。
3) キャリー, オーバーフロー, ゼロ, ネガティブの4つのステータスフラグを持つ。

4) 演算の両入力を外部から与えることができる。

などである。

本エンジンでは、乗算を高速に行うため、乗算専用のエレメントを備えている。この乗算器は、TRW社のMPY-16HJで、その特徴は、

1) 16ビット×16ビットの乗算を140msecで行うことができる。
2) 符号なしデータの乗算と2の補数データの乗算を行うことができる。

などである。演算結果の32ビットデータは、16ビットずつ2回に分けて取り出す。

両デバイス共、R-busとS-bus上のデータに対して演算を行うことができ、乗算器は、ALUのレジスタファイルの値を演算ソースとすることも可能である。

iii) パラメータスタックとリターンスタック

スタックは、Forthプログラムの実行において、非常に重要な役割りを果たす。従って、本エンジンでは、強力なスタックを持たせることを前提として設計を行っている。以下に、そのスタックの機能と、効率の向上に役立つと考えられる点を明確にする。

パラメータスタックは、あらゆる演算に対するソースとなり、また格納場所にもなる。特に、パラメータスタックのトップとセカンドの2値に対して演算を行い。結果をトップに返す、という処理が非常に多い。そこで、スタックに対する演算に注目して処理効率の向上をはかるならば、次の機能が、スタックに要求されると考えられる。

1) スタックポインタによる間接アクセス後、ポインタの自動増加。同様に、自動減少後、ポインタによる間接アクセスができること。
2) スタックのセカンドにある値も、スタックトップの値と同時にとり出せること。

本エンジンにおいては、この2点を共に満足するパラメータスタックを備えている。この機能により、スタックに対する演算を、すべて1マイクロサイクルで行うことができる。パラメータスタックのブロック図を図3.5に示す。すなわち、スタック本体は、バンクAとバンクBの2バンク構成になっており、ポインタもそれぞれのバンクに対して、ポインタAとポインタBの2つがある。バンクA, B共に、アクセスタイム55msecの4Kビットメモリ16個で構成されており、合わせて8K語の容量がある。

プッシュ動作が行われるたびに、データはバンク

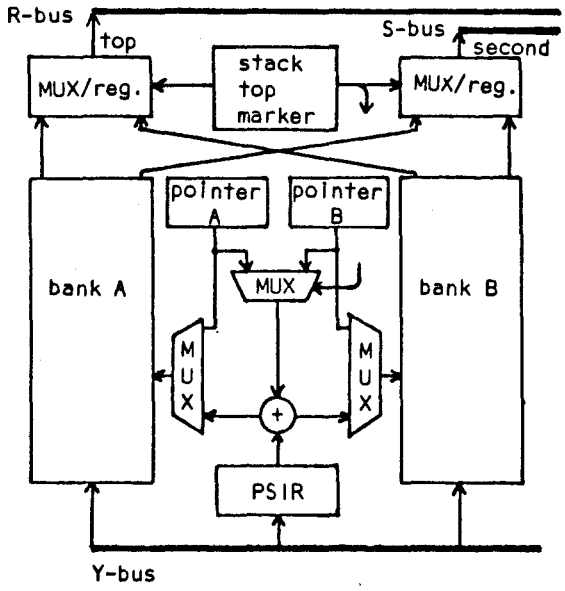



図３．５　パラメータスタックの構成

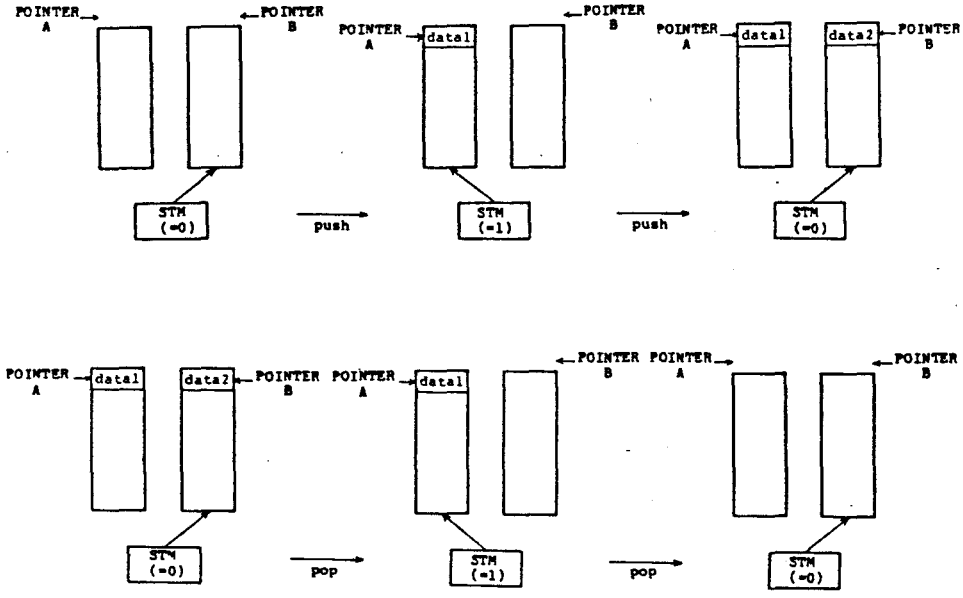



図３．６　パラメータスタックの内部動作

A、B交互にプッシュされ、ポップについても同様である。そして、どちらのバンクがスタックトップのデータを持っているのかということを記憶しておくものが、スタックトップマーカ(STM)である。STMが0ならば、バックBがスタックトップのデータを持っており、1ならば、バンクAが持っている。初期状態のSTMは、0である。

データがプッシュ、ポップされるときの概念図を図3.6に示す。個々のバンクのトップに注目すると、どちらかのバンクにスタックトップが、他方のバンクのトップにスタックのセカンドに相当するデータが入っていることになる。プッシュ，ポップのたびに、それぞれのポインタは、スタックトップポインタとスタックセカンドポインタの役割りを交代する。ポインタの変化、及びSTMの変化は、ハードウェアにより自動的に行われるために、ユーザから見た場合、論理的には通常のスタックと同様に扱うことができる。また、スタックトップの値はR-busに、セカンドの値はS-busに読み出すことができる。

パラメータスタックインデクスレジスタ(PSIR)は、13ビットあり、スタックポインタからあるオフセットを持ったアドレスのデータを、S-bus上に読み出したい時に使うレジスタで、そのオフセット値を書き込むことにより機能する。内部動作としては、スタックトップになっている方のポインタを左に1

ビットシフトし、最下位ビットをSTMとした値と、PSIRの値とが加算され、上位12ビットがバンクA、Bにアドレスとして与えられる。そして、最下位ビットが0ならばバンクBが、1ならばバンクAが読み出される。つまり、2つのバンクを合わせて、0000～1FFF(16)までのアドレスを持つ、8K語の容量のスタックと見立てて、加算を行うことになる。例として、PSIRに2を書き込んで、PSIRによるアクセスを行う時の様子を図示すると、図3.7のようになる。図(a)においては、3つのデータがプッシュされた状態であるから、ポインタAがスタックトップポインタになっている。そして、図(b)に示した様な加算が行なわれ、結果の最下位ビットが1で、スタックアドレスがFFE(16)であるから、バンクAのアドレスFFE(16)の内容が読み出され、DATA1を得ることがで

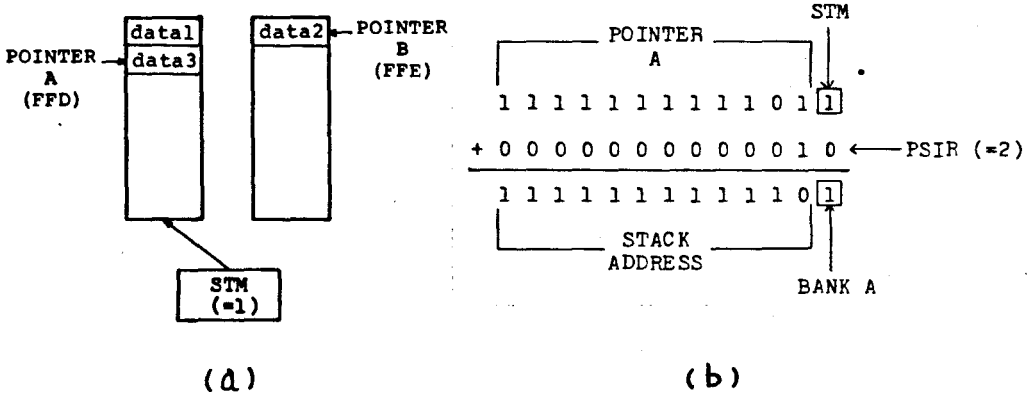



図3．7　ＰＳＩＲを用いたアクセスの概念図

きる。この機能により、様々なスタック操作が可能になり、効率も上がる。PSIRを用いてスタックをアクセスし、読み出した値をスタックトップにプッシュする動作も1マイクロサイクルで行うことができる。

フラグに関しては、アンダーフローフラグ、オーバーフローフラグの他、スタックトップが0の時にセットされる、スタックトップデータゼロフラグ(STZF)も持っている。

リターンスタックは、関数呼び出しの制御やDO-LOOPのインデクスとリミットなどの格納場所として使用される。アクセスタイム55msecの4Kビットメモリ素子により構成され、容量は4K語である。リターンスタックも、間接アクセス後自動増加と、自動減少後間接アクセスの機能を持つ。

フラグは、アンダーフローフラグとオーバーフローフラグを持つ。

iv) バリアブルストレージ

このモジュールは、Forthにおける変数や配列の取り扱いを高速化するために設けたものである。

Forthの場合、一般に、変数名を入力すると、その変数が格納されているメモリー中のアドレスがスタックトップに返される。例を示すと、まず、

```
13  VARIABLE  Q
```

と入力することにより、「Q」という名前を持ったディクショナリエントリが作られ、そのエントリ中に変数領域として16ビット分が確保され、初期値として13が与えられる。次に、

Q @ .

と入力すると、最初の「Q」によって、その変数領域のアドレスがスタックトップに積まれる。「@」は、スタックトップの値をアドレスとしてメモリを参照し、その値をスタックトップに返すワードである。続いて、「.」によって、スタックトップの値をターミナルへ出力し、結果として13が得られる。また、

26 Q !

と入力すると、「!」により、スタックトップの値をアドレスとして、スタックのセカンドにある値をメモリに書き込む。つまり、「Q」の変数領域に26を書き込んだことになる。

一般的なForthでは、この様にディクショナリ内に変数領域をとるが、Forthマシンでは WCS 内にディクショナリを構成しており、また命令の先取りを行っている関係上、WCSへの書き込みは、処理速度の点で不利である。従って、このHLLエンジンでは変数用メモリとしてアクセスタイム55msecの高速メモリを用いて16ビット×4K語用意し、スタックトップの値をアドレスとして、1マイクロサイクルで

変数の読み書きを行っている。

v) バイパスコントローラ

Forthプログラムの実行においては、スタック間のデータ転送の占める割り合いが大きく、これを高速に行えれば、効率向上に非常に有効である。

このモジュールは、データ転送のみに関与するもので、R-busもしくはS-bus上のデータをY-busへ送り出す役目を果す。ALUによってもこの機能を実現できるが、伝般遅延時間が長く、表3.3に示した様に、サイクルタイムとして245～305msec必要であるのに対して、このバイパスコントローラを用いた場合175msecで実行できる。つまり、40～70%程度の速度向上が期待できる。

vi) 各種レジスタ

以上のモジュールの他、データメモリのアクセスのため、メモリアドレスレジスタ(MADR)、インプットメモリデータレジスタ(IMDR)、アウトプットメモリデータレジスタ(OMDR)があり、全て16ビットである。

MADRはR-bus、OMDRはS-bus、IMDRはY-busに接続されていて、バリアブルストレージの場合と同様にスタックトップの値がアドレスとしてデータメモリに与えられ、さらに書き込み時には、スタッ

クのセカンドの値が書き込みデータとして与えられる。

データメモリのアクセスには、WAIT命令と共に用いることにより、CCUに対してWAIT要求が出され、サイクルの第1フェーズにおいてWAIT状態に入ることができる。データメモリのアクセス時間を十分に見込んだ上で、メモリインターフェースからREADY信号を返すことにより、プロセッサはWAIT状態から抜け出す。この機能により、様々なメモリのアクセス時間に対応できる。

前述の様に、データメモリとバリアブルストレージのアクセス手順は同じであるが、アドレス空間は異なり、どちらをアクセスするかはマイクロインストラクションにより指定する。

また、Z-80へのデータ引き渡し用のレジスタとして、16ビットのアウトプットデータレジスタ(ODR)がある。用途は、

1) HLLエンジンにおける演算結果の引き渡し。
2) 比較的短いメッセージ等の文字コードの引き渡し。

などで、ハードウェア、ソフトウェアのデバッグにも有効である。

Z-80への割り込みをかける際に使用するレジスタ

として、8ビットのインタラプトレジスタ(INTR)がある。このレジスタに書き込むことにより、Z-80に割り込みが発生する。そして、Z-80からのアクナリッジ信号(ACK)により、書き込んだデータがベクターとして渡される。またACK信号によってアクナリッジフラグがセットされ、割り込みが受け付けられたことを知ることができる。

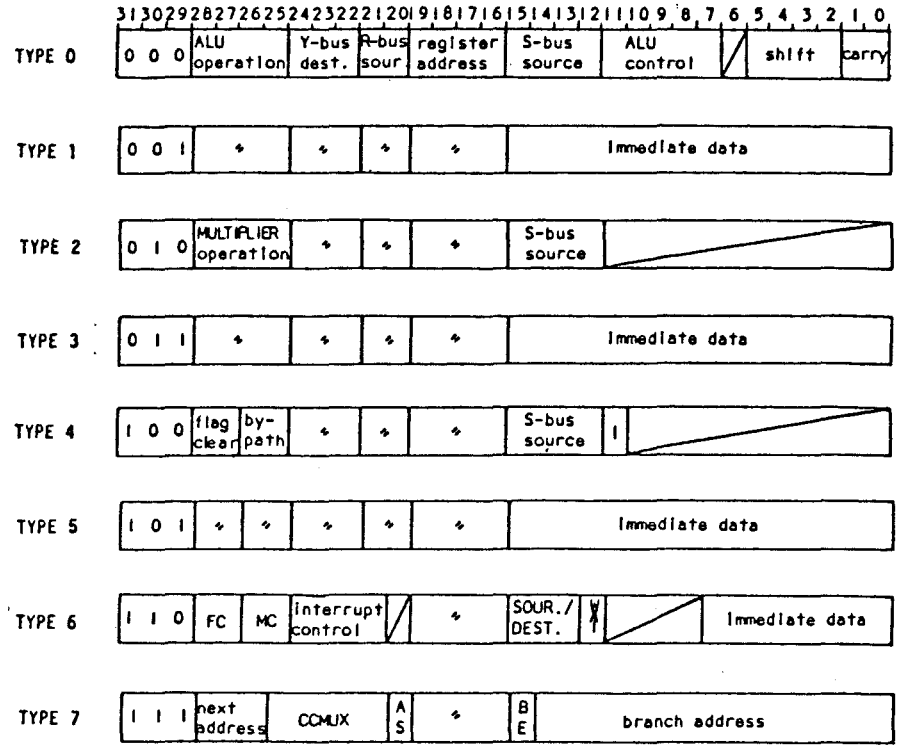
フラグレジスタは、10ビットでALUの演算結果に対応するフラグと、スタック類のオーバーフロー、

表3．4　フラグレジスタのビット構成

| ビット | フラグ&ステータス |
|---|---|
| 15－10 | 未使用　(＝0) |
| 9 | スタック　トップ　データ　ゼロ |
| 8 | インタラプト　アクナリッジ |
| 7 | パラメータ　スタック　オーバー　フロー |
| 6 | パラメータ　スタック　アンダー　フロー |
| 5 | リターン　スタック　オーバー　フロー |
| 4 | リターン　スタック　アンダー　フロー |
| 3 | OVR　(　ALU　ステータス　) |
| 2 | N　(　ALU　ステータス　) |
| 1 | C　(　ALU　ステータス　) |
| 0 | Z　(　ALU　ステータス　) |

アンダーフロー，スタックトップデータゼロ，インタラプトアクナリッジに対するフラグを保つもので，条件ジャンプに利用できる。フラグレジスタのビットのうち分けを表3.4に示す。

## 3.4 HLLエンジンのマイクロ命令




FC: Field Control
MC: Memory Control
CCMUX: Condition Code Multiplexer
AS: Address Source on S-bus
BE: Branch address Enable

図3．8 マイクロインストラクションの構成

本エンジンのマイクロ命令は、32ビット幅で構成されている。図3.8にその構成を示す。つまり、上位3ビットによりマイクロインストラクションのタイプを分けている。タイプ0と1がALUオペレーションであり、タイプ2、3がマルチプライヤオペレーション、タイプ4と5がバイパスコントロールで、タイプ6がメモリオペレーション、タイプ7がブランチオペレーションである。

ハードウェアの面から見ると、より ビット幅を広くとり、水平型のマイクロインストラクションに近づける方が、デコーダの負担も軽減されて扱いやすくなる。

しかし、Forthの場合、ディクショナリエントリの構成から、以前に定義済のワードを呼び出すための命令、もしくはアドレスポインタが大きな部分を占めるので、ビット幅を広くとりすぎるとWCSのビット利用率が格段に悪くなる。また、ビット幅を縮める際には、様々なオペレーションのうち排他的なオペレーションを選び出して、さらにコード化を施す手法をとるのが一般的であるが、命令フェッチ時にデコードを行わねばならず、負担が増すことになる。ここでは、その負担増とビット利用率とを考え合わせて、32ビット幅に決定した。

R-bus, S-busのソースとY-busデスティネーションの各フィールドと、ハードウェアモジュールと

| Y-bus destination | | | |
|---|---|---|---|
| bit2 | bit1 | bit0 | |
| 0 | 0 | 0 | reg. |
| 0 | 0 | 1 | (PSTP) |
| 0 | 1 | 0 | (RSTP) |
| 0 | 1 | 1 | PSIR |
| 1 | 0 | 0 | INTR |
| 1 | 0 | 1 | -(PSTP) |
| 1 | 1 | 0 | -(RSTP) |
| 1 | 1 | 1 | ODR |

| S-bus source | | | | |
|---|---|---|---|---|
| bit3 | bit2 | bit1 | bit0 | |
| 0 | - | - | - | reg. |
| 1 | 0 | 0 | 0 | (PSSP) |
| 1 | 0 | 0 | 1 | SR |
| 1 | 0 | 1 | 0 | (PSIR) |
| 1 | 0 | 1 | 1 | (RSTP) |
| 1 | 1 | 0 | 0 | (PSSP)+ |
| 1 | 1 | 0 | 1 | MULL |
| 1 | 1 | 1 | 0 | (PSIR)+ |
| 1 | 1 | 1 | 1 | (RSTP)+ |

| R-bus source | | |
|---|---|---|
| bit1 | bit0 | |
| 0 | 0 | reg. |
| 0 | 1 | PSIR |
| 1 | 0 | (PSTP) |
| 1 | 1 | (PSTP)+ |

**図3．9　ソースとデスティネーションの真理値表**

の対応関係を図3.9に示す。

## 3.5　HLLエンジンシステムの実装状態

本システムのハードウェアサイズについて述べる。HLLエンジン側は、17cm x 22cmのユニバーサル基板9枚（WCS4K語実装時）から成り、おおよそのうち分けは次の通りである。

i)　WCS　　1枚

ii)　CCU　　1枚

| | | |
|---|---|---|
| iii) | マイクロインストラクション デコーダ | 1枚 |
| iv) | リターンスタック | 1枚 |
| v) | バリアブルストレージ | 1枚 |
| vi) | ALU | 1枚 |
| vii) | 乗算器、バイパスコントローラ | 2枚 |

基板コネクタピン数は100ピンで、内部バスのみで多くを消費し、コントロールライン用が不足するため、不足分は前面のフラットケーブルにより補っている。電源は、5V-60Aの容量である。配線は、2枚がはんだ付け、残りはワイヤラッピングで行っている。

Z-80側は、HLLエンジンとのインタフェース、データメモリとのインタフェース、タイマー制御回路を2枚のS-100バス用ユニバーサル基板上に構成している。

HLLエンジンの実装については、以下の通りである。使用したユニバーサル基板は、1枚につき16ピンのTTLICが77個しか実装できず、パラメータスタックなどは、ポインタ制御部とメモリ部を2枚の基板に分割せざるを得なかった。1つのモジュールを2枚に分割することは、コネクタピンを不必要に消費することにもつながり、得策とはいえない。また、1枚あたりの実装密度が上がっても、電力消費の大

きいICを密に並べると、発熱の問題もあり新たな問題が生じる。従って、本マシンの規模拡大を考えるならば、基板の有効面積と共に実装密度が大きく、コネクタピン数の多いものを選ぶべきであろう。本マシンでは、基板あたりの実装密度を高くして、極力モジュールの分割を避け、発熱の問題に対しては、ファンによる強制空冷で対処している。

また、システム開発の初期におけるハードウェア、ソフトウェアのデバッグ効率を良くするため、リターンスタックとパラメータスタックの基板に7セグメントのLEDを取り付け、基板上でスタックポインタを目視できる様にした。これは、今後のソフト開発においても有効に利用できる。

## 3.6 結言

Z-80から参照できるHLLエンジン内のレジスタとその機能、Z-80のアドレス割り当てについて述べた。また、HLLエンジンの内部構成を示し、各モジュールの接続関係について述べた。Forthプログラムの特徴と対比させ、各モジュールの機能と構造を決定した際の根拠を明らかにした。そして、各モジュールの構造について詳しく述べ、2バンクから成るパラメータスタックやバリアブルストレージなど、本エンジンが、Forthプログラムを高速に処理

できるアーキテクチャを持つことを示した。また、マイクロ命令の構成と機能について述べ、それらがForth言語の基本ワードの多くを1マイクロサイクルで実行できる様に設定されていることを示した。最後に本エンジンのハードウェアサイズについて述べ、実装に関して注意を払った点について述べた。

# 第４章　HLLエンジンによるForthマシンの実現と評価

## 4.1　緒言

本章では、HLLエンジン上にForthプログラムを高速に実行できるForthマシンを実現する方法について述べる。前章までに述べてきた様に、本HLLエンジンはForth言語を基礎に置いて設計が為されているため、Forth言語処理系の構造とHLLエンジンのハードウェアモジュールとの対応は容易にとることが出来る。しかしながら、Forthマシンを実現するためには、HLLエンジンのWCS上のディクショナリを管理できるテキストインタプリタ等、本システムに適合したソフトウェアを開発する必要がある。[41]

本章では、ホストコンピュータ上に作成されたテキストインタプリタと、そのディクショナリ管理方法について述べ、メモリ効率と実行効率を考慮して決定されたディクショナリエントリの構成について述べる。さらに、HLLエンジンから入出力要求等が生じたときに必要なホストコンピュータ上の割り込み処理について述べる。また、Forthプログラム中でマイクロ命令を利用する為のマイクロアセンブラと、Forthプログラムの実行をより高速化するために開発した直接マイクロコード生成型コンパイラに

ついて述べる．最後に，ベンチマークテストを用いて本マシン上でのForthプログラムの実行時間を測定し，さらに実行ステップ数を解析することにより，HLLエンジンが有するハードウェアの機能に対して評価を加え，HLLエンジンの構成の，Forthプログラムの高速実行に関する有効性について検討する．[43]

## 4.2 Forthマシンの実現

### 4.2.1 テキストインタプリタとそのディクショナリ管理

ホストコンピュータであるZ-80は，オペレーティングシステムとしてデジタルリサーチ社のモニタコントロールプログラムCP/M を使用しており，Forthシステムの中核をなすテキストインタプリタは，Z-80上でアセンブリ言語を用いて開発を行った．テキストインタプリタでの入出力やファイルの管理には，CP/M[53]の機能を利用している．

ここで本システムでは，Z-80の主記憶中のディクショナリとWCS上に作成されるディクショナリの2つがあり，ディクショナリサーチにおいて，どちらのディクショナリをサーチするべきかという問題が生じる．本システムでは，この問題に対し，Z-80モードとForthマシンモードの，2つのモードを設けることにより対処している．また，それぞれのモー

ドについて、コンパイルモードと実行モードの2つのモードが存在する。すなわち、Z-80モードであれば、Z-80のディクショナリに対してディクショナリサーチが行われる。コンパイルモードであれば、Z-80のディクショナリにコンパイルされ、実行モードであればZ-80のワードが実行される。Forthマシンモードであれば、まずHLLエンジンのWCS上のディクショナリに対してディクショナリサーチが行われる。コンパイルモードであれば、WCS上のディクショナリにコンパイルされ、実行モードならHLLエンジンのCMRに実行命令が与えられる。この場合、WCS上のディクショナリにワードが見つからなかったときは、引き続きZ-80のディクショナリサーチが行われ、その時点でワードが見つかれば、実行モードのときのみZ-80の実行が行われる。

Z-80モードとForthマシンモードの切り換えは、Z-80のディクショナリ内に定義されているワードにより行われる。システム起動後は、通常、Forthマシンモードにあるため、この様なディクショナリ管理方法をとることにより、Z-80のディクショナリにあるワードも有効に利用でき、4.2.4で述べるマイクロアセンブラも容易に実現できる。

また、Forthでは1024バイト毎のブロックとしてディスクファイルを取り扱う仮想メモリ方式をとっている。本システムのテキストインタプリタも、

Z-80の主記憶中に2ブロック分のバッファを備え、コンソールキーボードからの入力テキストの他、ブロック中のテキストを入力とすることも可能である。ブロックには、それぞれブロックナンバーが付いており、本システムでは、ブロックナンバー00～EF(16)までの240ブロックを扱うことができる。そのうち分けは、ブロックナンバー00～7F(16)までをZ-80のワード用とし、80(16)～EE(16)までをForthマシンのワード用とレている。ブロックナンバーEF(16)は、ブロック間のコピーなどに使う、ワーキング用ブロックである。

システムの起動時は、Z-80モードにセットされておりブロック00をロードすることにより、システム用のワードがZ-80のディクショナリに作成される。

次に、ブロック80(16)をロードすることにより、Z-80のディクショナリにForthマシンのサポート用ワードが作成され、HLLエンジンのWCSに、実行時に必要なワード呼び出しルーテンや、HLLエンジンのインタラプトルーテンなどの書き込みが行われる。続いてForthマシンモードに切り換わり、WCSにディクショナリが作成される。

### 4.2.2 ディクショナリエントリの構成

Z-80モードで定義されたワードは、図4.1に示すようなディクショナリエントリとなって、主記憶中の

ディクショナリに付加される。図4.1(a)は、アセンブリ言語によってワードの定義を行ったときに生成されるもので、識別用のネームフィールドやリンクフィールド等から成るヘッダ部と、マシンコードから成るコードセクションで構成されている。図4.1(b)は、既定義のワードを参照して新しいワードを定義したときに生成されるディクショナリエントリで、

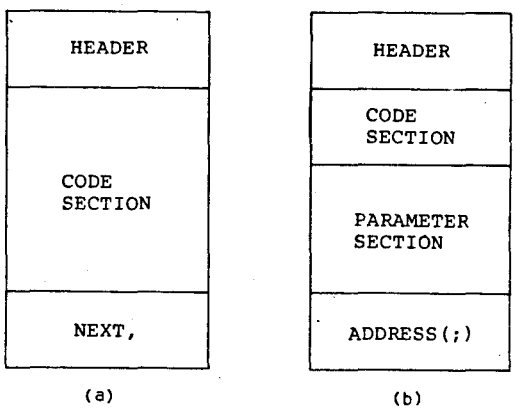
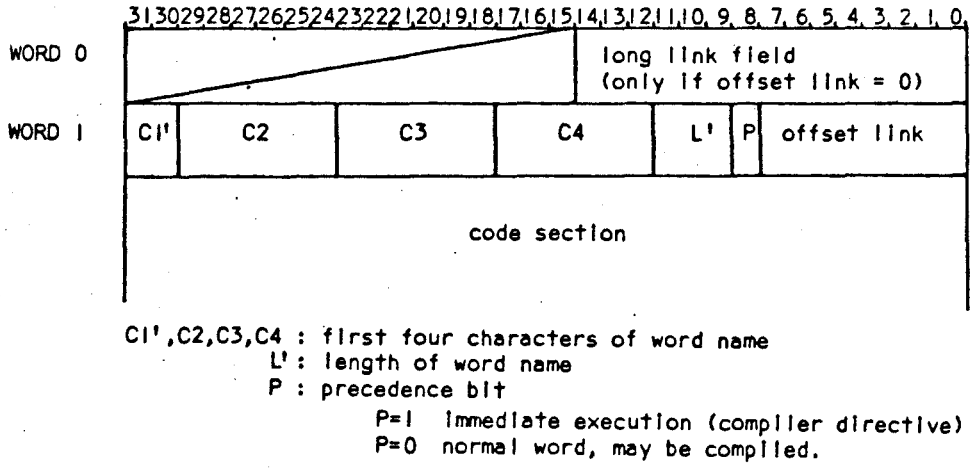



図4．1　ディクショナリエントリの構成




C1',C2,C3,C4 : first four characters of word name
L' : length of word name
P : precedence bit
P=1 immediate execution (compiler directive)
P=0 normal word, may be compiled.

図4．2　ディクショナリエントリのヘッダ部の構成

ヘッダ部とパラメータセクションから成っている。パラメータセクションは定義の際に参照したワードへのアドレスポインタのみで構成しており、所要メモリの節約をはかっている。実行時には、このアドレスポインタにより間接的にワードの呼び出しを行う。

図4.2に、ヘッダ部の構成を示す。ヘッダ部は1語の場合と2語から成る場合の2種類が存在する。通常は1語であるが、ディクショナリ内のリンクすべきワードが8ビットのオフセットリンクでは届かないとき、ロングリンクフィールドを付加し2語構成とする。ネームフィールドは、最初の4文字と11文字までの長さによって、ワードを識別できる様になっている。

HLLエンジンのWCS上に生成されるディクショナリエントリの場合、パラメータセクションは存在せず、アドレスポインタに相当するものとしてサブルーチンコール命令が埋め込まれる。この理由は、WCSが1語32ビット構成であるため、アドレスポインタのみを置いてもメモリの節約にはつながらないことと、ワードの呼び出しを高速化するためである。言い換えると、本マシンのWCS上のディクショナリは、サブルーチンパッケージの集まりとみなすことができる。

### 4.2.3　ホストコンピュータの割り込み処理

本システムでは、Forthマシンモードにおいて、HLLエンジンの実行終了や入出力の要求などが生じた場合、Z-80へ割り込みをかけ、Z-80側で処理される。

HLLエンジンからの割り込みは、Z-80のマスク可能な割り込み入力端子に接続されている。この端子への割り込みに対するZ-80の応答は、3つのモードが存在し、ソフトウェアによりモードの選択を行う。それぞれのモードに対する応答は、次の様である。

モード0　：　次の1命令に関してデータバスから命令コードを読むが、プログラムカウンタの値は変化しない。

モード1　：　自動的に、0038(16)番地をサブルーチンコールする。

モード2　：　Iレジスタと呼ばれる8ビットのレジスタを上位バイトとし、データバスから読み込まれる8ビットデータを下位とする16ビットアドレスを、自動的に間接サブルーチンコールする。

以上のモードのうち、本システムではモード2を用いている。このときの間接サブルーチンコールに

使われる16ビットアドレスの下位8ビットは、HLLエンジンのINTRから読み込まれる。従って、HLLエンジンは、INTRに書き込む8ビットデータにより、Z-80に様々な異なる処理を要求することができる。表4.1にINTRに書き込まれる値と、対応するZ-80側の処理を示す。説明を加えると、INT-1は、HLLエンジンがODRに実行結果などの数値を書き込んでから使われる割り込みである。Z-80はODRの内容を読み込んで、ターミナルに表示する。INT-2は、HLLエンジンの実行終了をZ-80に伝えるために使われる。INT-3からINT-6は、HLLエンジンのハードウェアエラーを知るために用いられる。また、INTRに10(16)を書き込むと、Z-80は何の処理も行わず、再び割り込み受け付け可能状態へ戻る。これは、HLLエンジンが、Z-80にコンソール出力要求の割り込みをかけた後、INTR=10(16)の割り込みをかけ、受け付けられたことによってZ-80の処理が完了したことを知るために使われる。つまり、

表4．1　インタラプトレジスタへの書き込みデータとZ－80の割込み処理

| | 書込みデータ(16) | Z-80の割込み処理 |
|---|---|---|
| INT-1 | 04 | 数値出力 |
| INT-2 | 06 | 実行の終了 |
| INT-3 | 08 | パラメータスタックアンダフローエラーメッセージ出力 |
| INT-4 | 0A | パラメータスタックオーバフローエラーメッセージ出力 |
| INT-5 | 0C | リターンスタックアンダフローエラーメッセージ出力 |
| INT-6 | 0E | リターンスタックオーバフローエラーメッセージ出力 |
| INT-7 | 10 | 割込み待ち |
| INT-8 | 12 | 文字出力 |
| INT-9 | 14 | Z-80ワードの実行 |

HLLエンジンとZ-80の同期をとるための割り込みである。

### 4.2.4 マイクロアセンブラ

低レベルの記述から高レベルの記述によるワードの定義が可能であるというForth言語の大きな特徴を最大限に生かすためには、本マシンの最下位レベルでの記述、すなわちマイクロインストラクションのニーモニックコードでの記述によるワードの定義が可能でなければならないと考えられる。これを可能にするために、本システムのマイクロアセンブラは、Z-80上のワードによって開発されている。すなわち、Z-80の主記憶中のディクショナリに、クロスマイクロアセンブラが作成されていることになる。このマイクロアセンブラ用のワード群は、Forthマシンモードにあるときに有効に利用される。

### 4.2.5 直接マイクロコード生成型コンパイラ

Forthマシンモードにおいて、以前に定義済のワードを参照して新たなワードを定義した場合、通常は、コンパイルされたワードの本体には、参照されたワードを呼び出すマイクロコードが組み込まれる。しかしながら、本マシンでは、スタック操作やデータ転送などで代表されるForthの基本的なワードは、ほとんどのものが1マイクロサイクルで実行される。

それらの基本的なワードの呼出しを行うと、1マイクロサイクルの処理のために、呼出しとリターンに必要な2マイクロサイクルのオーバヘッドが生じることになる。そこで、本システムでは、1~3マイクロサイクルで完了するような特定の基本ワードと、DO-LOOPやIF-ELSE-THENなどの制御用ワードが参照された場合、その処理に対応する直接マイクロコードが生成され、ワードの本体に埋め込まれるようになっている。

## 4.3 Forthマシンの性能評価

実際のベンチマークプログラムを実行したときの本システムの実行時間と、Z-80, LSI-11上のpoly-FORTH[29][38], FACOM230-38, ACOS-700S, Z-80上のFORTRAN, 及びZ-80のアセンブリ言語によってそれぞれ作成されたベンチマークプログラムの実行時間の測定結果を示す。(付録参照)

ベンチマークプログラムは、次の3種類を用いた。

(i) フィボナッチ数列の計算: $f_n=f_{n-2}+f_{n-1}$, $f_1=1$, $f_2=1$で表される数列で、Forthプログラムでは2回のスタック操作と1回の加算で記述されている。ここでは、10,000回のループを100回繰り返し、計算途中で生じるけたあふれは無視している。

(ii) 2次関数値の計算：$f(n)=n^2-60n+100$, $n=1,2,\cdots 200$ で表される関数で、それぞれのnに対する計算結果は配列として記憶している。以上の計算を100回繰り返している。なお、本システムにおいては、配列をバリアブルストレージ上にとった場合とデータメモリ上にとった場合との2種類について実行し、所要時間の測定を行っている。

(iii) ソーティング：配列内のあるデータとそれ以降のデータとの比較を行い、後者の方が大きい場合は、その都度前者とデータの交換を行う方法をとっている。実際には、1,2,…1.000のデータを配列として記憶させ、第1回目は1.000から1へと逆順に並べ変えを行う。第2回目は、1回目を実行した後、配列をそのままにして同じプログラムを実行させる。つまり、2回目は並べ変えが行われず、配列内の値の大小比較のみが行われる。第1回目のソーティングをSORT1, 第2回目をSORT2とする。

以上のベンチマークプログラムによる実行時間の測定結果を表4.2に示す。表より、Z-80のpolyFORTHと比較すると127倍から522倍，LSI-11上のpoly-FORTHと比較すると60倍から244倍の速度比で本マシンが実行していることが分かる。又、ACOS-700SのFORTRANプログラムとほぼ同等の実行時間で、本

## 表4．2　ベンチマークプログラムの実行時間の測定結果

（単位はs）

| | FORTH マシン | poly FORTH (Z-80) | poly FORTH (LSI-11) | FORTRAN (FACOM 230-38) | FORTRAN (ACOS-700S) | FORTRAN (Z-80) | アセンブリ言語 (Z-80) |
|---|---|---|---|---|---|---|---|
| フィボナッチ数列 | 2.201 | 279.215 | 132.4 | 35.313 | 3.3 | 107.500 | 14.528 |
| 2次関数 | *0.066<br>**0.074 | 34.465 | 16.1 | 0.793 | 0.063 | 12.342 | 12.013 |
| SORT 1 | 2.705 | 496.689 | 237.7 | 23.171 | 2.792 | 201.611 | 36.260 |
| SORT 2 | 1.803 | 292.134 | 135.6 | 14.195 | 1.089 | 102.955 | 14.292 |
| SORT1+SORT2 | 4.508 | 788.823 | 373.3 | 37.366 | 3.881 | 304.566 | 50.552 |

*：配列をバリアブルストージ上にとった場合
**：配列をデータメモリ上にとった場合

注）Z-80のCPUクロックは2MHz

## 表4．3　総ステップ数と平均サイクルタイム

| | フィボナッチ | 2次関数値 | SORT 1 | SORT 2 |
|---|---|---|---|---|
| 総ステップ数 | 11,001,407 | 321,107 | 13,500,498 | 9,004,998 |
| 平均サイクルタイム（ns） | 200 | 206*<br>230** | 200 | 200 |

*：配列をバリアブルストレージ上にとった場合
**：配列をデータメモリ上にとった場合

## 表4．4　マイクロ命令のタイプ別利用率

| マイクロ命令のタイプ | 利用率 (%) | | | |
|---|---|---|---|---|
| | フィボナッチ数列 | 2次関数 | SORT 1 | SORT 2 |
| タイプ 0 | 18.2 | 18.7 | 7.4 | 11.1 |
| タイプ 1 | 9.1 | 6.3 | 7.4 | 11.1 |
| タイプ 2 | 0.0 | 6.2 | 0.0 | 0.0 |
| タイプ 3 | 0.0 | 0.0 | 0.0 | 0.0 |
| タイプ 4 | 54.5 | 37.5 | 40.8 | 33.4 |
| タイプ 5 | 0.0 | 12.5 | 3.7 | 5.6 |
| タイプ 6 | 0.0 | 6.2 | 11.1 | 5.6 |
| タイプ 7 | 18.2 | 33.3 | 29.6 | 12.6 |

## 表4．5　オペレーション別利用率

| オペレーション | | | 利用率 (%) | | | |
|---|---|---|---|---|---|---|
| | | | フィボナッチ数列 | 2次関数 | SORT 1 | SORT 2 |
| パラメータスタック | トップ | ソース | 45.5 | 50.0 | 37.1 | 38.9 |
| | | デスティネーション | 54.5 | 62.4 | 44.5 | 38.9 |
| | | ( )+ | 27.3 | 31.2 | 25.9 | 27.8 |
| | | -( ) | 36.4 | 56.1 | 37.1 | 33.4 |
| | セカンド | ソース | 27.3 | 25.0 | 22.2 | 16.7 |
| | | ( )+ | 9.1 | 24.9 | 11.1 | 5.6 |
| リターンスタック | | ソース | 27.3 | 25.0 | 22.2 | 22.2 |
| | | デスティネーション | 18.2 | 6.3 | 7.4 | 5.6 |
| | | ( )+ | 18.2 | 6.3 | 11.1 | 11.1 |
| | | -( ) | 18.2 | 6.3 | 11.1 | 11.1 |
| (PSIR) | | | 0.0 | 0.0 | 3.7 | 0.0 |
| メモリ | | 読出し | 0.0 | 0.0 | 3.7 | 5.6 |
| | | 書込み | 0.0 | 6.2 | 7.4 | 0.0 |
| ジャンプ | | 無条件 | 9.1 | 6.3 | 18.5 | 16.6 |
| | | 条件 | 9.1 | 6.3 | 11.1 | 16.7 |

( )+：ポインタの自動増加
-( )：ポインタの自動減少
(PSIR)：パラメータスタックインデクスレジスタを用いたスタックのアクセス

マシンの処理が行われていることが示されている。

次に、実行時間の測定結果をもとに、各ハードウェアモジュールの利用率を求め、評価を行う。利用率とは、ある特定のハードウェアモジュールを用いたオペレーション回数の、総ステップ数に対する比率をいう。

まず、表4.3はベンチマークプログラムごとの総ステップ数と、一命令当りの平均サイクルタイムを示している。表4.4は、マイクロインストラクションのタイプ別の利用率を表す。表4.5は、各ハードウェアモジュールとその機能の利用率を表している。これらの表をもとにして、以下にハードウェアモジュール別の評価を行う。

(1) 演算モジュール

これらのベンチマークプログラムにおいて、ALUは、主に加減算とDO-LOOPのインデクスの増加や、IF-ELSE-THENにおける大小比較に用いられている。乗算器は、2次関数値を求めるプログラムのみで用いられている。利用率は、総ステップ数の6.2%にすぎないが、このプログラムにおいて他機種との速度比が最も大きい。すなわち、高速乗算器を備えたことと、その機能を十分に生かせるバス構造が妥当であったといえる。

(2) パラメータスタック

表4.5より、パラメータスタックの利用率は、全プ

ログラムを通じて非常に高い。このことから、パラメータスタックの強力な機能が、Forthプログラムの高速実行に大きく貢献していることが確認できた。スタックのセカンドを読み出せる機能は、およそ4~7ステップに1回の割合で利用されており、スタックトップの読出し回数の50%にものぼるひん度で用いられている。もしもこの機能がなければ、同じ処理について1~3ステップ余分に必要であり、ALUのレジスタファイルも併用しなければならない場合も生じる。従って、スタックのセカンドの値を読み出せる機能は、高速実行および処理効率の向上に十分役立っていると考えられる。パラメータスタックトップポインタ(PSTP)の自動増加、自動減少の機能は、スタックオペレーションのうち60~80%で併用されており、この機能も不可欠であると考えられる。パラメータスタックセカンドポインタ(PSSP)における自動増加機能の利用率は、PSTPの場合と比較すると低い値を示している。これは、データ転送のみの処理時の利用率が低いためであると考えられる。しかしながら、スタック上での演算やメモリの書込みを1マイクロサイクルで完了させるためにはやはり不可欠な機能で、このことは、演算と配列への書込みがひん繁に行われる2次関数値を求めるプログラムでの利用率が、非常に高いことからもうなずける。

(3) リターンスタックとマイクロプログラムシー

ケンサ

リターンスタックは、全プログラムを通じて4~6ステップに1回の割合で読み出されている。デスティネーションとしてよりもソースとして用いられる利用率の方がはるかに高いのは、DO-LOOPのインデクスを、リターンスタックからポインタを変化させずに読み出して演算を行う回数が多いためである。ベンチマークテストに用いたプログラムでは、ワード呼出しのレベルが浅いが、実際の応用プログラムにおいては多重のワード呼出しが行われるため、リターンスタックの利用率が増加すると考えられる。以上のことから、マイクロプログラムレベルで多重サブルーチンコールを可能としたマイクロプログラムシーケンサ，リターンスタック，及びそのバスとの接続関係は有効であったと判断できる。又、リターンスタックの、テンポラリレジスタとしての有用性も確認できた。

(4) バリアブルストレージ

2次関数値を求めるプログラムに注目すると、表4.5より、配列へのアクセスは6.2%にすぎないが、表7に示されているように、配列をバリアブルストレージ上にとった場合、データメモリ上にとったときよりも実行時間に12%の改善が見られる。従って、ひん繁に配列へのアクセスを行うようなプログラムに対して、バリアブルストレージは非常に有効であ

ると考えられる。

次にソーティングを行うプログラムについて考察する。表7において、SORT1に要する時間とSORT2に要する時間との差をとると、配列内のデータの入れ換えのみに必要な時間が得られる。この時間とSORT1，SORT2に要する時間について、本マシンの実行時間を1として表に示すと表4.6のようになる。この表から、全機種を通じて、SORT1，SORT2に要する時間の比よりも、差をとった時間の比の方が大きいことが分かる。すなわち、本マシンにおいて、配列内のデータの入れ換えが非常に高速に行われていることを示している。この理由は次のように考えられる。Forthでは、演算結果は常にパラメータスタック上にある。本マシンは、その結果の値をメモリ書込み用レジスタなどへ転送することなく、そのままの状態でバリアブルストレージやデータメモリへ1マイクロサイクルで書き込むことができる。つ

表4．6　ソーティングに関する実行時間の比

| | FORTH マシン | poly FORTH (Z-80) | poly FORTH (LSI-11) | FORTRAN (FACOM 230-38) | FORTRAN (ACOS-700S) | FORTRAN (Z-80) | アセンブリ言語 (Z-80) |
|---|---|---|---|---|---|---|---|
| SORT1 | 1.000 | 183.6 | 87.87 | 8.566 | 1.032 | 74.53 | 13.40 |
| SORT2 | 1.000 | 162.0 | 75.21 | 7.873 | 0.604 | 57.10 | 7.927 |
| SORT1 -SORT2 | 1.000 | 226.8 | 113.2 | 9.950 | 1.889 | 109.4 | 24.35 |

まり、演算後の配列への書込みや配列からの読出し後の演算などの処理が、効率良く行われていると考えられる。従って、バリアブルストレージと内部バスとの接続関係は、Forth プログラムの処理に都合の良い形だといえる。

(5) バイパスコントローラ

表4.4において、タイプ4とタイプ5の利用率の和がバイパスコントローラの利用率を表している。すなわち、全プログラムを通じて、ほぼ2ステップに1回の割合で利用されていることになる。このことから、データ転送用としてバイパスコントローラを設けたことが有効であったと考えられる。

(6) 直接マイクロコード生成形コンパイラ

関数呼出しの命令を生成する本システムの通常の

表4．7 直接マイクロコード生成形コンパイラを用いた場合と通常のコンパイラを用いた場合の実行時間

| | 直接マイクロコード生成形コンパイラを用いた場合 | 通常のコンパイラを用いた場合 |
|---|---|---|
| フィボナッチ数列 | 2.201 | 3.511 |
| 2 次 関 数 | 0.066*<br>0.074** | 0.109*<br>0.109** |
| SORT 1 | 2.705 | 4.419 |
| SORT 2 | 1.803 | 2.711 |
| SORT 1<br>-SORT 2 | 0.902 | 1.708 |

*：配列をバリアブルストレージ上にとった場合
**：配列をデータメモリ上にとった場合

コンパイラを用いてプログラムをコンパイルし、実行した場合と、直接マイクロコード生成形コンパイラを用いて実行した場合の所要時間を表4.7に示す。表より、直接マイクロコードを生成した方がおよそ1/1.9～1/1.5程度実行時間が短いことが分かる。この結果から、直接マイクロコード生成形コンパイラは、非常に有効な手段であったといえる。

本システムでは、ワードの本体が3ステップ以内のものについてこの手法を適用しているが、各基本ワードの使用ひん度から考えて、本体が1ステップのものだけに限定しても、ほぼ同様の結果が得られると考えられる。

## 4.4 結言

HLLエンジンを利用してForthマシンを実現する方法について述べた。HLLエンジンをサポートするホストコンピュータ上に作成されたテキストインタプリタは、ホストコンピュータの主記憶上のディクショナリとHLLエンジンのWCS上のディクショナリを効率良く管理できる様な工夫がなされていることについて述べた。その結果マイクロアセンブラや直接マイクロコード生成形コンパイラが容易に開発できた。また、ディクショナリエントリの構成について述べ、WCS上のエントリ本体部はアドレスポイン

タから成るインダイレクトスレッデッドコードを用いず、サブルーチンの集合で構成していることを述べた。また、HLLエンジンからZ-80への割り込みコードとそのコードに対する割り込み処理について述べた。最後にベンチマークテストを用いて本マシンの評価を行い、以下の結果を得た。

1) Z-80マイクロコンピュータのForthプログラムと比較すると最高522倍、またACOS-700S上のFORTRANプログラムとほぼ同等の実行速度が得られた。

2) 実行ステップ数をもとに算出したハードウェアモジュールの利用率から、2バンク構造を有する強力なパラメータスタックが、Forthプログラムの高速実行に有効であることを確認した。特に、スタックのセカンドにある値を読み出せる機能は不可欠であることがわかった。

3) 本エンジンのリターンスタックとマイクロプログラムシーケンサ、及びそのバスとの接続関係が有効であることを確認した。

4) 変数専用高速メモリであるバリアブルストレージと内部バスとの接続関係は、演算後の配列への書き込みや配列からの読み出し後の演算等の処理に有効である。

5) データ転送用ハードウェアであるバイパスコントローラは、2ステップに1回の割合で利用

されており、プロセッサ内のデータ転送がForthプログラムの高速実行に重要であることがわかった。

6) 直接マイクロコード生成形コンパイラを用いることにより最高約2倍実行速度を向上させることができた。

# 第5章　HLLエンジンによるPascalマシンの実現と評価

## 5.1　緒言

Pascal言語は、簡潔な言語構造・厳格なデータ型のチェックを行うことによる高信頼性等の特徴を有するプログラミング言語で、1968年にN.Wirthによって作成された。その用途は、研究や教育用をはじめとして、最近では実用面においても広く用いられる様になってきている。

Pascalプログラムは多くの場合、Pマシン(Pseudo-machine)と呼ばれる仮想計算機の機械語であるPコード(Pseudo-code)にコンパイルされ、そのPコードを解釈、実行することにより処理が行われる。Pマシンは、内部にいくつかのレジスタやポインタ、評価用スタック等を持つスタックコンピュータである。従って、従来の計算機上でPコードを実行する場合、スタック機構をソフトウェアで実現しなければならないため実行効率が極めて悪くなる。

ここで、HLLエンジンとPマシンの構造上の整合性について検討したところ、両者共基本的にスタックコンピュータであることから、HLLエンジンはPマシン内の仮想ハードウェアに容易に対応できる構造を持っていることがわかった。

本章では、Pascal言語の特徴と言語処理系、すなわちPコードとそれを実行するPマシンの構造について詳しく述べる。次に、従来の計算機を用いてPマシンを実現した時に実行速度、実行効率を低下させる原因について考察する。そしてそれらを解決するために、実際にHLLエンジンの有するハードウェアとPマシンの仮想ハードウェアとをどの様に対応させるかについて述べる。ここで、Pコードの構成と機能について考えると、従来のPコードには当然のことながらHLLエンジンのハードウェアを直接制御できる命令がないため、冗長な実行ステップを要する場合がある。そこで、従来のPコードの機能とHLLエンジンの構成やマイクロ命令の機能とを照らし合わせ、よりHLLエンジンの機能に適合した中間コードを設定した。続いて、Pascalマシンを実現するためのソフトウェアについて述べる。Pascalマシンは2通りの手法で実現した。すなわち、①Pコードインタプリタをマイクロプログラム化する手法と②Pコードをマイクロ目的コードにコンパイルし、さらに目的コード生成に関して若干の最適化を施す方法である。また、新しい中間コードを生成するPascalコンパイラの開発について詳述する。最後に、本Pascalマシンの静的特性の評価とベンチマークテストを用いた動的特性の評価を行う。[44][45][46]

なお、基礎となるPascal処理系の種類については、

現在ミニ・マイクロコンピュータ上で広範に使われているUCSD-Pascalを選んだ.

## 5.2 Pascal言語の特徴と処理系の構造

### 5.2.1 Pascal言語の特徴

Pascal言語は、1968年頃チューリヒ連邦工科大学(ETH)のN. Wirthによって開発された。プログラミングを教育するために適した言語を使用可能にすることと、高い信頼性を有する処理系を作成することなど、教育，研究用として設計されたが、現在では応用プログラム開発用にも広く使われている。処理系が要する記憶容量も小さくて済むため、ミニ・マイクロコンピュータ上で稼動するPascalシステムもいくつか存在する[3][8]。

Pascalは次の様な特徴を持つ言語である.

1) 簡潔な言語設計

設計当初の目標として、処理系が複雑にならず規模が大きくならないことが掲げられている。諸機能の一貫した方針に基づく定義や例外規定を極力少なくしたことにより、豊富な言語機能を持っているにもかかわらず非常に簡潔な言語仕様になっている。

2) 高信頼性

Pascalではすべてのデータに型を与え、変数などは参照前に宣言することにより、コンパイル時にチェックするなど、信頼性を保証している。従来の言語のようなデータ型の自動変換などは行わず、変数の型の明確化を強いている点も信頼性の向上に役立っている。

3) 洗練された制御構造

繰り返し文や条件分岐に関する文など、ソフトウェア工学で提唱されている構造化プログラミングに適合した制御機能を持っている。

### 5.2.2 Pascal言語処理系の構造

Pマシン(Pseudo-machine)は、Pascalプログラムの中間言語であるPコード(Pseudo-code)を解釈・実行する仮想計算機であり、内部にいくつかのレジスタやポインタ、評価用スタック等を持ち、処理のビット幅が16ビットで、スタック上で演算を行うスタックコンピュータである。

Pascalプログラムをコンパイルすると、目的Pコードと制御情報等を含むPコードファイルが生成される。そして、実行時においてはPコードファイルが読み込まれ、図5.1に示すようなコードセグメント, データセグメント, MSCW(Mark Stack Control Word)が主記憶中に作成される。それぞれについて以下に

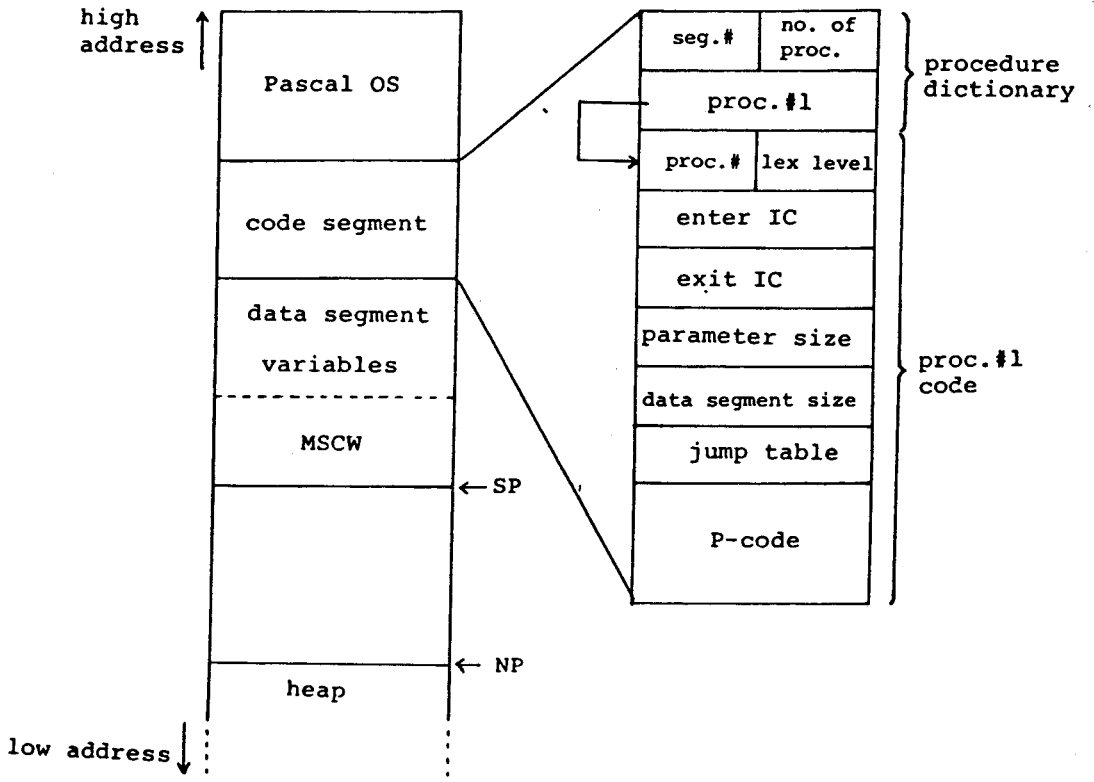



図5．1　Ｐａｓｃａｌプログラム実行時のメモリ割り当て

説明する．

1)　コードセグメント

コンパイラによって作成されたPコードファイル中のプロシージャ・ディクショナリや目的Pコード，さらにプロシージャナンバ，レキシカルレベル等で構成されている．レキシカル・レベルとは、入れ子構造を持つPascalプログラムのプロシージャの深さを表す．一番外側は０であり、順に内へ入るごとに１ずつ増える．

2) データセグメント

呼ばれたプロシージャのためのローカル変数が確保される。

3) MSCW

呼ばれたプロシージャの制御状態を示す以下の6語の情報から成り立っている。

i) MSSP (Mark Stack Stack Pointer)

呼んだプロシージャの評価用スタックの先頭を示す制御ポインタの値を保存する。

ii) MSIPC (Mark Stack Interpreter Program Counter)

呼んだプロシージャのコール命令の次の番地を保存する。プロシージャリターン時に使用する。

iii) MSSEG (Mark Stack SEGment pointer)

呼んだプロシージャの属するセグメントのプロシージャディクショナリへのポインタを保存する。

iv) MSJTAB (Mark Stack Jump TABle)

呼んだプロシージャの属する属性テーブルへのポインタを保存する。

v) MSDYN (Mark Stack DYNamic link)

MSCWの作られた順番(呼ばれた順番)を示すチェーンで、1つ前のMSCWのMSSTATを指す。このプロシージャの実行完了時、そのプロシージャを呼んだ状態に戻す時に用いる。

vi) MSSTAT (Mark Stack STATic link)
呼ばれたプロシージャよりレキシカル・レベルが1つ低いプロシージャのMSCWのMSSTATを指す.

MSCWの下位は評価用スタック領域として用いられ、Pコードの実行はこの領域を用いて行われる。また、実行時に動的にデータが生成されるリスト構造などを取り扱う場合に対して、ヒープ領域が備えられている。

また、Pマシンの内部には、次に示すようなレジスタが想定されている。

1) SP (Stack Pointer)
メモリの上位から下位へ伸びるスタックのトップを示すポインタ.

2) MP (Mark stack Pointer)
現在実行中のプロシージャのMSCWへのポインタ。ローカル変数をアクセスする時に用いられる.

3) BASE (BASE procedure pointer)
最新のベースプロシージャのMSCWへのポインタ。グローバル変数をアクセスする時に用いられる.

4) NP (New Pointer)

メモリの下位から上位の方へ伸びるヒープ領域の先頭を示すポインタ。

5) JTAB (Jump TABle pointer)
現在実行中のプロシージャの属性テーブルへのポインタ。

6) SEG (SEGment pointer)
現在実行中のプロシージャが属するセグメントのプロシージャディクショナリへのポインタ。

以上のポインタが、Pascal実行の制御を行っている。

Pコードは、8ビットの命令コード部と最大3個までのパラメータ部で構成されている。大きく分けると、次の6種類である。

1) 定数コード
命令コード部が00～7F(16)のもので、パラメータ部を持たない。1バイトコードである。

2) メモリ操作用コード
命令コード部と最大2個までのパラメータの持つ。コード長は1バイトから4バイトである。

3) 比較用コード
パラメータ部を持たない1バイトコードである。

4) 算述論理演算用コード
パラメータ部を持たない1バイトコードである。

5) 分岐用コード

1～3個のパラメータを持つ．コード長は，2バイトから7バイトである．

6) プロシージャ呼出し及び関数呼出し用コード
プロシージャナンバを表すパラメータを1個持つ．コード長は，2バイトである．

## 5.3 Pascalマシンの設計

前節で述べたような構造・機能を持つPマシンを既存の計算機上でソフトウェアによって実現し，Pascalプログラムを実行する場合，以下の様な欠点が生じる．

1) ソフトウェアによるインタプリタを用いているため，Pコードのフェッチ等に時間を要する．
2) 様々な用途に用いられるスタックをソフトウェアにより実現し，スタック領域を主記憶上にとっているためスタックポインタの操作，スタックを用いた演算・メモリ参照に関する実行効率が悪い。
3) その他の多くの仮想レジスタ，仮想ポインタの管理を行う必要があり，実行時の負担が大きい．

以上の点に注目し，本研究ではPascalプログラム

を効率良く処理できる構造を持ったPascalマシンシステムをHLLエンジンシステム上に実現した.

システムの設計に当って、Pascalプログラムの高速実行に重点を置き、かつ比較的短期間で柔軟性のあるシステムの開発・デバッグを完了させるため、基本的に次の様な方針をとることにした.

1) 中間コードインタプリタをマイクロプログラム化する.
2) 中間コードをマイクロ目的コードにコンパイルする. さらに、目的コード生成に関して若干の最適化を施す.

また、本システムの開発・デバッグを効率良く行い、開発後のシステムをより使い易いものにするためのサポートプログラムを開発した.

### 5.3.1 HLLエンジンとPマシンの構造上の対応

次に、Pマシン内の仮想レジスタやポインタ、スタック等を、HLLエンジン内の実際のハードウェアにどのように対応させるかについて、具体的に述べる. Pascalプログラムの実行時は、スタックの参照が頻繁に行われ、変数参照のためにMP、BASEなどのポインタの読み出しも高い頻度で実行される. これらの実行時の状況を考慮し、本システムでは以下

の様に対応させた。

1) データメモリを主記憶として用いる。すなわち、インタプリタの場合、データメモリ上のPコードを解釈し、実行することになる。
2) 実行時に非常に重要な役割りを果たすスタックにはパラメータスタックを対応させた。そうすることにより、主記憶とスタックとを異なる記憶領域に分離でき、主記憶の参照回数の軽減が可能である。
3) MPやBASE等のポインタは、ALUのレジスタファイルの一部を利用する。
4) インタプリタ方式の場合、インストラクションカウンタは、ALUのレジスタを用いる。

### 5.3.2 中間言語の検討

なお、より一層の高速処理を目指すため、中間コード生成時の大域的最適化に重点を置き、中間コードを生成するPascalコンパイラの検討を行った。また、マイクロ命令の機能を考慮して、個々の中間コードの有する機能にも検討を加えた。

前節で述べた様な構成の既存のPコードに合わせてHLLエンジン上でPascalプログラムを効率良く実行できるシステムを開発する場合、局所的最適化は可能であるが、大域的最適化が困難であり、冗長な

マイクロ命令を実行する部分が生じる。これは既存のPコード自体が、HLLエンジンのハードウェアを有効に利用する構成・機能を持ち合わせていない点にある。

以下に、従来のPコードをHLLエンジン上で実行する際の不都合な点について列挙する。

1) パラメータの値が1バイト又は2バイトの可変長データであり、8ビットマシン向きであること。
2) ジャンプテーブルを参照してから分岐アドレスを決定する分岐方式を用いていること。
3) プロシージャ呼出し時に、プロシージャナンバをパラメータとしてプロシージャディクショナリを参照して制御情報を得ること。
4) 配列要素のデータへの代入。
5) 直接にハードウェアを扱うPコードがない。

1)について説明を加えると、データが可変長であることより、パラメータフェッチ時に条件判断機能を必要とする。また、Pコードのフェッチ及びPコード処理において、メモリ参照が頻繁に生ずる等、インタプリタの処理効率が非常に悪くなる。2)は、Pコードの分岐命令は、オフセット値を含んでおり、そのオフセットを用いてジャンプテーブルを参照し

て分岐アドレスを決定していることを表している。この方法では、実行時の負担が大きく、繰り返し制御文の多いプログラムで効率が悪い。4)について詳述すると、配列要素へのデータへの代入においてコンパイラのPコード生成は、まず代入される変数のアドレスを規定するコード列を生成し、次に代入するデータ側の式の評価を行い、最後にメモリストア用コードを生成する。すなわち、評価用スタックのトップに書き込むべきデータ、セカンドにメモリアドレスがプッシュされた状態から代入が実行されるが、HLLエンジンは、データメモリへの書き込みに際して、スタックトップにアドレスが置かれていることを前提として設計されている。従って、HLLエンジンでは、パラメータスタックのトップとセカンドの値を交換してから書き込み動作を行う必要がある。5)について補足すると、特にfor-do文などの繰り返し制御において、リミット値の格納、ループインデックスの参照などメモリ参照が頻繁に生ずる。これは、リターンスタックやレジスタ等を利用することによって、メモリ参照回数を減少させることができる。

以上の欠点に注目し、Pコードを若干修正し、本HLLエンジンのハードウェア構成を考慮した新しい中間コードを設定した。以後この新中間コードをP'コードと呼ぶ。さらに、そのP'コードを生成する

Pascalコンパイラを開発し、前述の2つの処理系、すなわちマイクロプログラム化インタプリタと直接マイクロコード生成型コンパイラをP⁺コードに対しても試みる。

## 5.4 Pascal マシンの実現

本システムのソフトウェア体系をまとめると、次のようになる

1) 中間コード(P⁺コード)生成型コンパイラ
2) マイクロプログラム化インタプリタ
3) 直接マイクロコード生成型コンパイラ
4) サポートプログラム

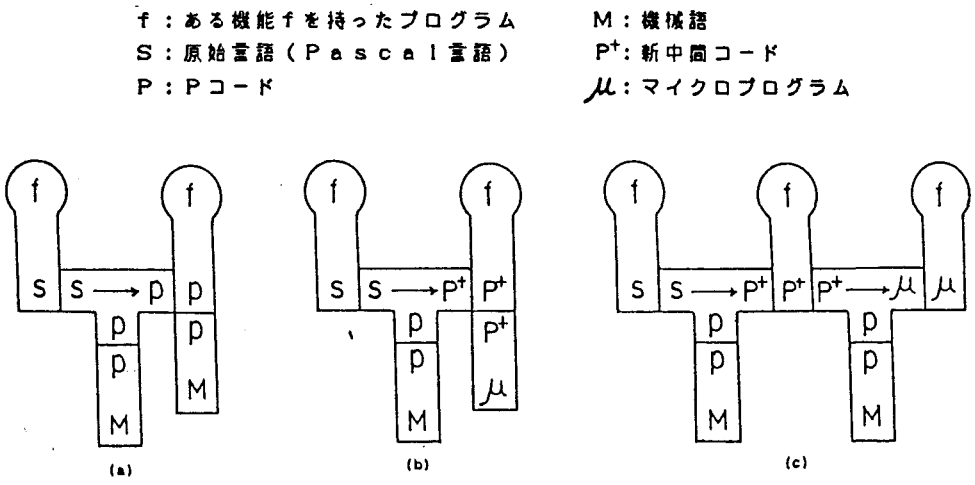



図5．2 処理系の模式図

2), 3)については、Pコードと$P^+$コードの各々について処理系の開発を行った。

処理系の模式図を図5.2に示す[3]。(a)は、従来のUCSD-Pascalにおける処理系を表しており、(b)にマイクロプログラム化インタプリタ、(c)に直接マイクロコード生成型コンパイラのそれぞれに対する処理系を図示している。(b)の中央のT字型の部分が、$P^+$コードを生成する新たに開発したコンパイラを表している。

上記のソフトウェアシステムは、UCSD-Pascal上で処理可能なPascal言語で記述されており、それが翻訳されて作られたPコードを実行することにより処理される。従って、本システムはUCSD-Pascalシステムによりサポートされる形態になっている。テキストファイルのエディタ，入出力処理，ファイル管理などは、UCSD-Pascalが担当する。

本マシンシステムでは、ユーザと上記ソフトウェアシステムとのインタフェースとしてサポートプログラムを開発した。サポートプログラムを用いることにより効率良いプログラム開発が可能である。

プログラムの処理手順としては、まずUCSD-Pascalシステムを起動させ、画面エディタによりソースプログラムを作成・編集しファイラによりソーステキストファイルをフロッピーディスク上に作成する。続いて、本システムを起動させると、システムの初期

化が行われてモニタが呼ばれ各コマンド処理へと移る。通常プログラムを実行する場合、WCSにマイクロプログラムの実行に必要な初期ルーテンをロードする。コンパイラによりソーステキストファイルからプログラムを読み込み、解析して中間コードを生成し、P'コード列をファイルとして格納する。さらに、ここでインタプリタを用いるときは、マイクロプログラム化されたインタプリタをWCS上に格納し、レジスタなどの初期化を行い、インタプリタを起動する。ソースプログラムをマイクロコードにまでコンパイルする場合は、中間コードデータをマイクロコードに展開し、マイクロコードファイルを作成する。実行コマンドによりマイクロコードファイルよりマイクロコードをWCS上に格納し、マイクロプログラムを起動する。実行すべきマイクロプログラムの開始アドレスをHLLエンジンのWARに書き込み、CMRを通じてエンジンに起動をかけ、プログラムを実行する。

本システムでもForthシステム同様、実行終了や入出力の要求などが生じた場合、UCSD-Pascal(Z-80)への割り込みをかけ、Z-80側でこれらの処理が行われる。

ユーティリティは、本Pascalマシンシステムの Pascal実行の支援及びハードウェア状態を監視するのに都合が良い構成・機能を持っている。また、

Forth言語で記述されマイクロプログラムにまでコンパイルされたWCS上のプログラムをマイクロコードファイルとしてフロッピーディスク上に格納することができる。つまり、問題に応じて記述しやすい方の言語でプログラムを作成しコンパイルしておき、目的コードファイルとして格納できる。

### 5.4.1 Pascalコンパイラの開発

コンパイラの作成にあたっては、次の点を考慮した。

1) 効率の良い中間言語プログラムを出力すること。
2) 簡単なコンパイラであること。
3) 変更・拡張が容易であること。
4) 簡潔なコード化。

処理の対象とする高級言語は、Pascal言語のサブセットである。この言語によるプログラムは、標準Pascalのコンパイラでも処理可能である。

コンパイラの構文解析部は、Pascal-S[17]の構造をもとにUCSD-Pascalの機能を盛り込んだ構造を持つが、ファイル関係の処理を除いた上記のPascalサブセットを扱っている。構文解析ルーチンは、再帰的下向き構文解析法を採用しており、機能別にモジュ

ール化されている。以下に主な処理について述べる。

1) 宣言の処理

各変数に対して型の属性を与える。型チェック及び変数・定数等のコンパイルに用いるようなシンボル表を作成する。

2) 文の処理

i) ソースプログラムの実行文中に現われる変数や定数は、コンパイラが宣伝部を処理して作成した各種の表の対応するエントリへのポインタに変換される。

ii) 式の中に現われる各種演算子は、同じ演算子記号のものでもオペランドのデータ型により意味が異なる場合があり、これらのものについてはあらかじめコンパイラで異なる中間コードに変更しておき、インタプリタ実行時のオーバーヘッドも減少させている。

iii) 変数の中間コード化は、レキシカル・レベルなどにより3種類に分類している。

iv) <式>に対応する中間コード列は、逆ポーランド記法で表わす。インタプリタ実行時のオーバヘッドが少ないなどの利点をもつ。

v) 変数への代入文は、既に述べた様に、スタックへプッシュする順序を考慮し、常にまず代入するデータ側、最後に代入される変数のアドレスの順序でコード生成を行う。

(1) 中間言語の設計

本システムでは、Pascalプログラムを一度仮想的な計算機のための中間言語に変換し、前述した2つの処理系で処理・実行する。従って、2つの処理系を作成しやすい中間言語をどのように設定するかが問題になる[5][7]。

中間言語(P'コード)は、次のような点を考慮して設計した。

1) スタックコンピュータのための命令であること。
2) インタプリタを作成しやすいこと。
3) 直接マイクロコード生成型コンパイラの処理を簡単にするため、マイクロ命令の機能を考慮に入れること。
4) ハードウェアモジュールを効率良く利用できる目的プログラムを出力できること。
5) デバッグが容易となる簡単な構成。

特に、3), 5)について補足すると、コードの値によってパラメータの数を以下のように規定した。

| コード値 | : | 0～199, | 200～229, | 230以上 |
|---|---|---|---|---|
| パラメータ数 | : | 0, | 1, | 2 |

以下に、中間言語命令の仕様について述べる。

本マシンシステムの中間言語命令であるP'コードは、8ビットの命令コード部と最大2個までのパラメータ部で構成され、約100種類備えられている。また、次に示すデータ型を備えている。

- Boolean（1ビットの論理値を示す。）
- Integer（16ビット長の2進数）
- Scalar（部分範囲型も含む。）
- Character（8ビットの文字データ）
- Real
- Array, Record（構造型データ）
- String（全文字列数最大600まで可能）

各中間言語命令とその主な特徴について、以下に述べる。

a) 分岐形式（分岐用コード）

UCSD-PascalのPコードでは、ジャンプテーブル中に作られた分岐アドレスの値を用いる方式であったが、本マシンシステムでは命令中で分岐アドレスが規定される。従って、インタプリタ時の分岐処理のステップ数を減少できる。

b) 定数の割り付け方法（定数コード）

UCSD-PascalのPコード同様、リテラル値が7F(16)以下の場合、命令コード中に作成する。こ

れも、インタプリタ時の高速処理化に貢献するものと考える。

c) パラメータの構成（全コード）

Pコードの場合は、8ビット又は16ビットの可変長データであった。P'コードは、16ビットマシンであるHLLエンジンを意識した構成をとっている。つまり、パラメータの値は全て16ビット長のデータとする。これにより、マイクロコード生成型コンパイラにおけるパラメータの取り扱いが簡単になり、汎用性も増す。また、インタプリタのデータ読み出し時間が大幅に短縮できる。

d) 変数のアクセス（メモリ操作用コード）

3種類の変数（local, global, intermediate）に対して、3つのタイプのアクセス形式を備え、コンパイラは適宜使い分ける。9種類の読み出し命令と1つの書き込み命令が用意されている。バロース社のB6500で用いられているディスプレイ・レジスタ機構[10]は採用せず、MPDO, BASEDOの2つの制御ポインタのみでアクセスを行う。その理由としては、ローカル変数・グローバル変数が主に使われ、それ以外の中間レキシカルレベル中の変数の使用頻度が比較的少ないと考えられるからである。

e) プロシージャ及び関数の呼出しとリターン

呼出し命令として4種類が備えられており、呼

ばれるプロシージャの場所に応じて使い分けられる。リターン命令としては、3種類が備えられている。実行順序制御においては、バロース社のB5500[10]などで採用されているMSCWを用いているが、2語の情報で成り立っていてB5500やUCSD-Pascalのものより簡単である。

f) 入出力の処理

入出力命令は、10個用意されている。write文における演算結果や文字コードは、HLLエンジンのODRを通してZ-80側に引き渡され処理される。この際、Z-80に割り込みをかけて出力要求を与える。

g) ハードウェアモジュールの直接制御

これは、従来のPコードに見られない機能であり、P⁺コードの最も特徴のある部分である。ループインデックスの参照に利用されるリターンスタックを制御する命令が備えられている。

(2) データアクセス機構

Pascalプログラムをコンパイルした場合、ロード・ストアなどの簡単なデータ操作命令が頻繁に用いられる。従って、Pascalプログラムを実行する際、データをどれだけ高速にアクセスするかが大きな問題となる。そこで、本システムでは、データへのアクセスを効率良く行う機構を備えている。

アクセス形式は、以下の4つに大別される。

i) リテラル値　　：直接その値が中間コードそのもの又はパラメータとなる。
ii) アドレス参照型　：データのあるアドレスをアクセス。
iii) ダイレクト型　　：アドレスをパラメータとして与え、直接その内容（データ）をアクセス。
iv) インダイレクト型：アドレスをパラメータとして与え、間接的にデータをアクセス。

後者3つに対しては、仮想アドレス機構[1]を採用していて、固有のベースレジスタとしてMPDOとBASEDOの2つのポインタを持たせ、変数の種類によって使い分けている。UCSD-Pascalのpコードシステムでは、iv)の型を備えていない。

コンパイル時に区分される変数の種類と変数アクセスの方法を以下に示す。図5.3に各々のデータアクセス機構を示す。

1) グローバル変数

プロシージャの入れ子構造で一番外側、つまりレキシカル・レベルが0で宣言された変数はどのレベ

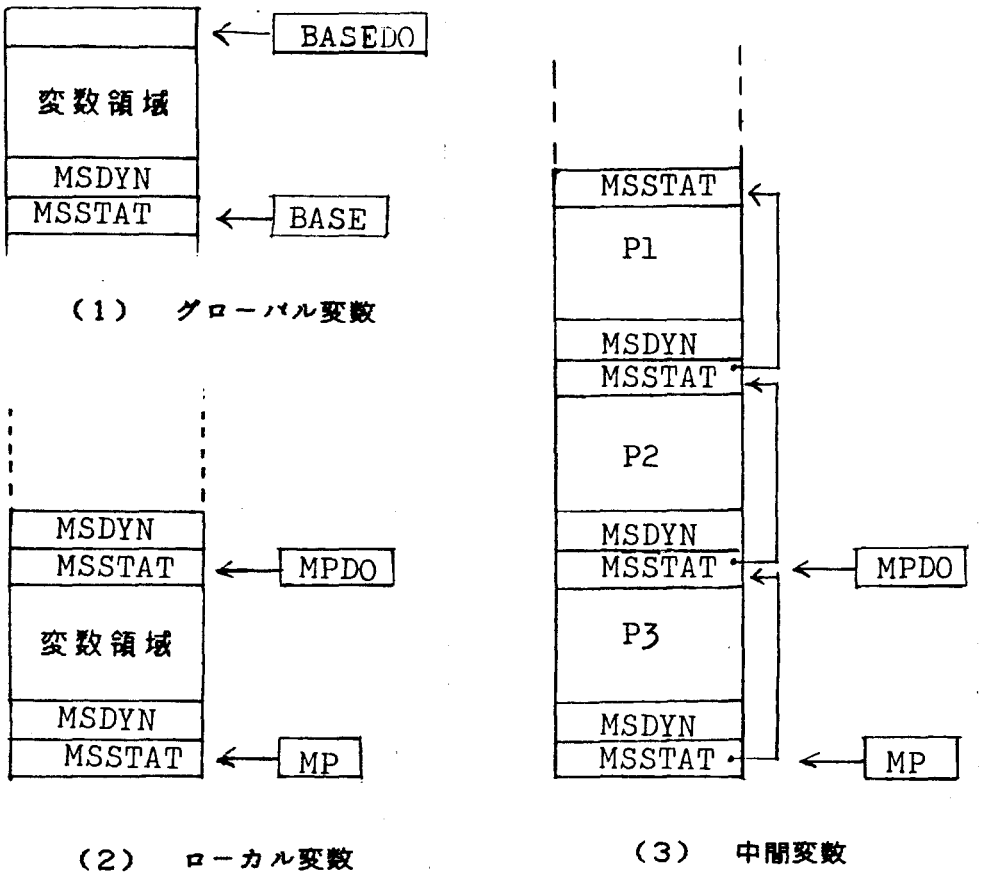



図5．3 データアクセス機構

ルからもアクセス可能である。図5.3の(1)に示される制御ポインタBASEDOを基準にオフセット値をパラメータとして各変数をアクセスする。

2) ローカル変数

現在実行中のプロシージャで宣言された変数。制御ポインタMPDOを基準にオフセット値をパラメータとして変数をアクセスする。図5.3の(2)に示す。

3) 中間変数

スタティックリンクをたどってアクセスするような1), 2)以外の変数。図5.3の(3)でP3実行中にP1の変数をアクセスする際は、制御ポインタMPDOが指すスタティックリンク部よりレキシカル・レベル差(この場合2回)だけさかのぼった場所を基準としてオフセット値をパラメータとしてアクセスする。

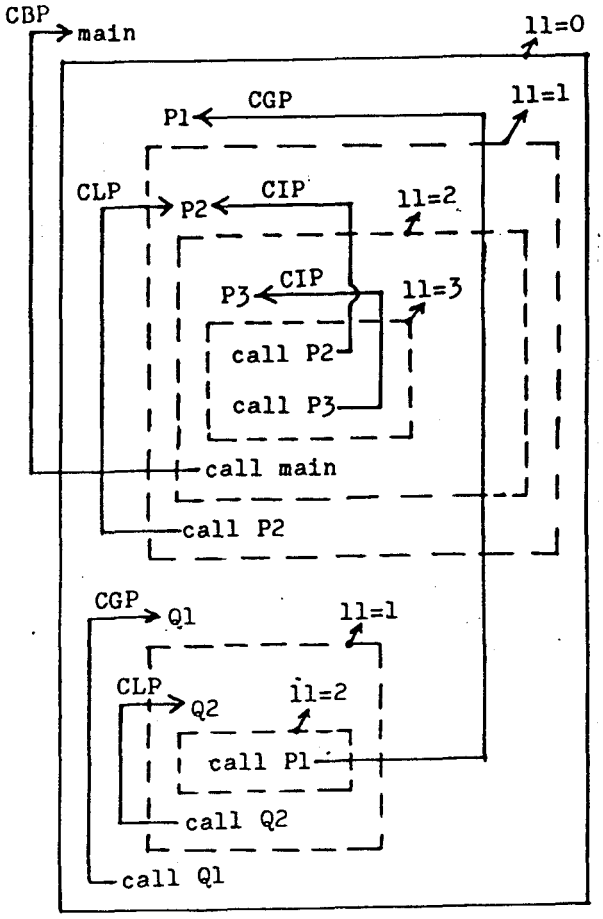



図5．4 プロシージャコール命令の使用例

(3) 実行順序制御機構

各プロシージャコールとリターンによるMSCWの構造及び各制御ポインタの動作を図5.4の例に従って以下に説明する。

プロシージャコールとリターンには、以下に示す6種類の命令がある[28]。

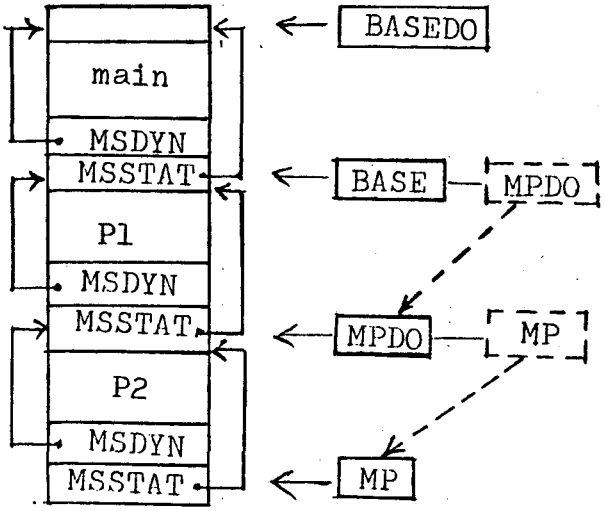
1) CLP (Call Local Procedure)
2) CGP (Call Global Procedure)
3) CBP (Call Base Procedure)
4) CIP (Call Intermediate Procedure)
5) RNP (Return Normal Procedure)
6) RBP (Return Base Procedure)

1) CLP：現在実行中のプロシージャのすぐ子供のプロシージャを呼ぶときに用いられる。

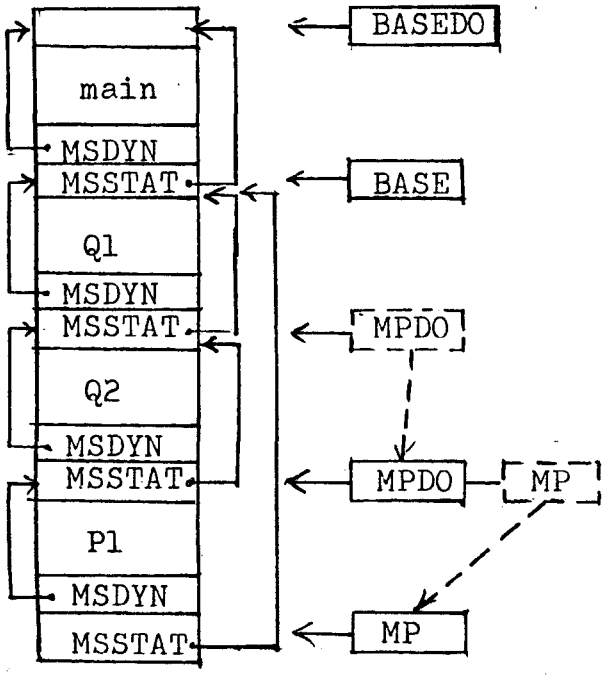
例) P1におけるP2のコール

図5.5の(1)のようにMSCWが構成される。すなわち、P2のダイナミックリンク，スタティックリンクは現在実行中のP1のMSCWを指す。また、局所変数アクセスに必要なMPDOとMPは点線に示すように更新される。

2) CGP：レキシカル・レベルが1 ($ll = 1$)であるプロシージャに用いられる。どのレキシカル・レベルからでも呼ぶことが可能で

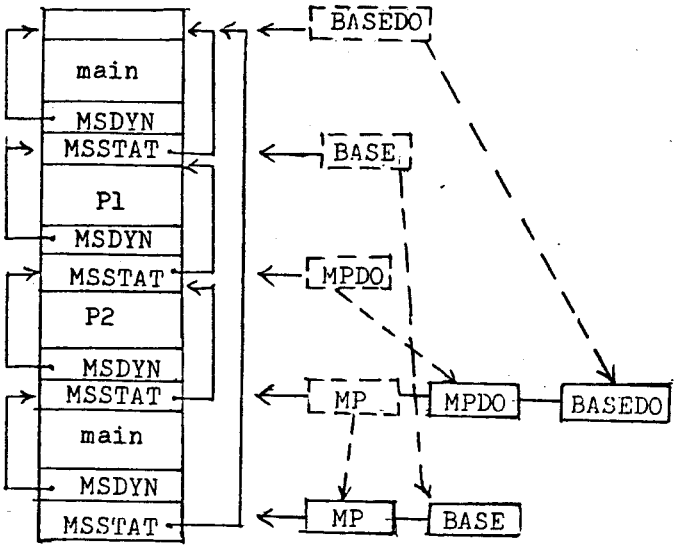



(1) CLP

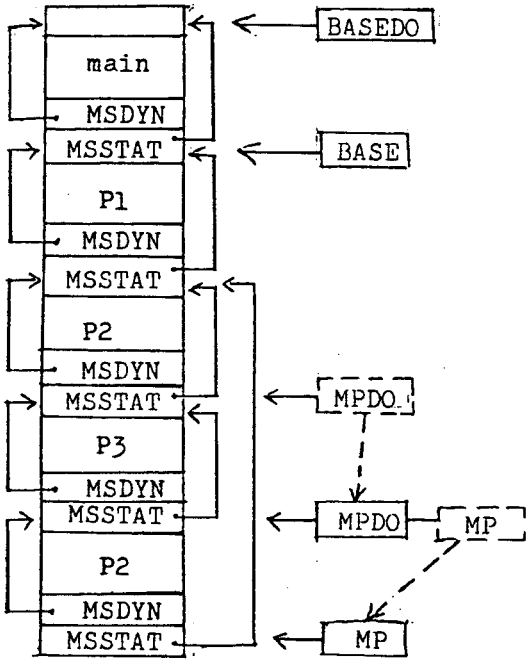



(2) CGP

図5．5 実行順序制御機構

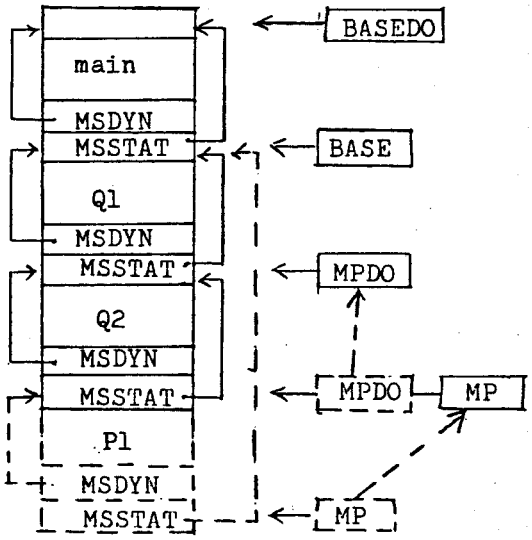



(3)　CBP

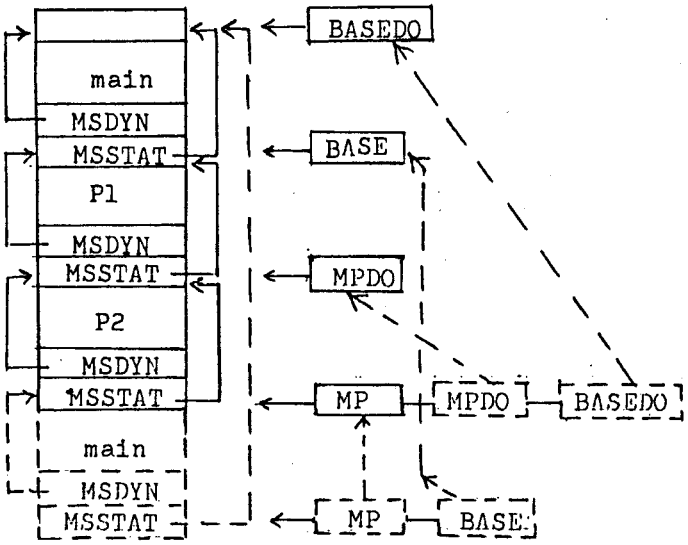



(4)　CIP

図5．5　実行順序制御機構




(5) RNP




(6) RBP

図5．5 実行順序制御機構

ある．

例）Q2におけるP1のコール

図5.5の(2)のようにMSCWが構成される．すなわち，P1のダイナミックリンクは現在実行中のQ2のMSCWを指す．一方，スタティックリンクは制御ポインタBASEと同一の場所を指す．つまり，P1で使用可能な変数は，P1内の局所変数とmainで宣言された大域変数であることを示している．また，局所変数アクセスに必要なMPDOとMPは点線に示すように更新される．

3) CBP：レキシカル・レベルが0 (ll=0)であるベースプロシージャを呼ぶときに用いられる．どのレキシカル・レベルからでも呼ぶことが可能である．セグメントプロシージャに対応する．

例）P2におけるmainへのコール

図5.5の(3)のようにMSCWが構成される．すなわち，mainのダイナミックリンクは現在実行中のプロシージャP2のMSCWを指す．スタティックリンクは，制御ポインタBASEが指しているスタティックリンク部が指す場所を指す．これによって新たにmainの実行を始める際に，P1・P2の局所変数を参照できない仕組みになっている．制御ポインタの更新は点線で示す．

4) CIP：レキシカル・レベルが2以上の途中のプ

ロシージャを内部プロシージャから呼ぶときに用いられる。

例) P3におけるP2のコール

図5.5の(4)のようにMSCWが構成される。すなわち、P2のダイナミックリンクは現在実行中のP3のMSCWを指す。一方、スタティックリンクは、更新前の制御ポインタMPDOが指す場所から、call側とcaller側のレキシカルレベル差だけリンクをたどった場所を指す。この場合、P3とP2のレキシカル・レベル差は1であるので1回だけリンクをさかのぼった場所を指す。これによって新たにP2の実行を始める際にP3の局所変数を参照・使用できない仕組みになっている。

5) RNP：レキシカルレベルが0でないプロシージャからのリターン時に用いられる。

例) P1からのリターン

図5.5の(5)のようにMSCWが捨てられる。すなわち、Q2におけるP1のコール前の状態に戻る。ダイナミックリンクをたどることによって点線で示すように制御ポインタを更新する。

6) RBP：レキシカル・レベルが0であるベースプロシージャからのリターン時に用いられる。

例) mainからのリターン

図5.5の(6)のようにMSCWが捨てられる。すなわ

ち、P2におけるmainのコール前の状態に戻る。
ダイナミックリンクとスタティックリンクをたどることにより、て点線に示すようにポインタを更新する。

以上をまとめると、MSCWが2語の情報のみで構成されていて、UCSD-Pascalの6語より簡単になっている。つまり、MSCWの生成及び除去の際、MSCWと制御ポインタの管理が非常に容易であり、レジスタの使用により、メモリ参照回数が少なくて済む。

また、変数アクセス専用の制御ポインタBASEDO, MPDOの導入により変数の高速アクセスが可能である。評価用スタックをメモリ上で用いていないため、MPDOが自動的に前のMSCWを指すことになる。さらに、プロシージャディクショナリの参照なしに制御情報が得られる利点がある。従来のMSIPCつまりリターン時のリターンアドレスは、リターンスタックを利用しておりMSCWに情報は不要である。

### 5.4.2 マイクロプログラム化インタプリタ

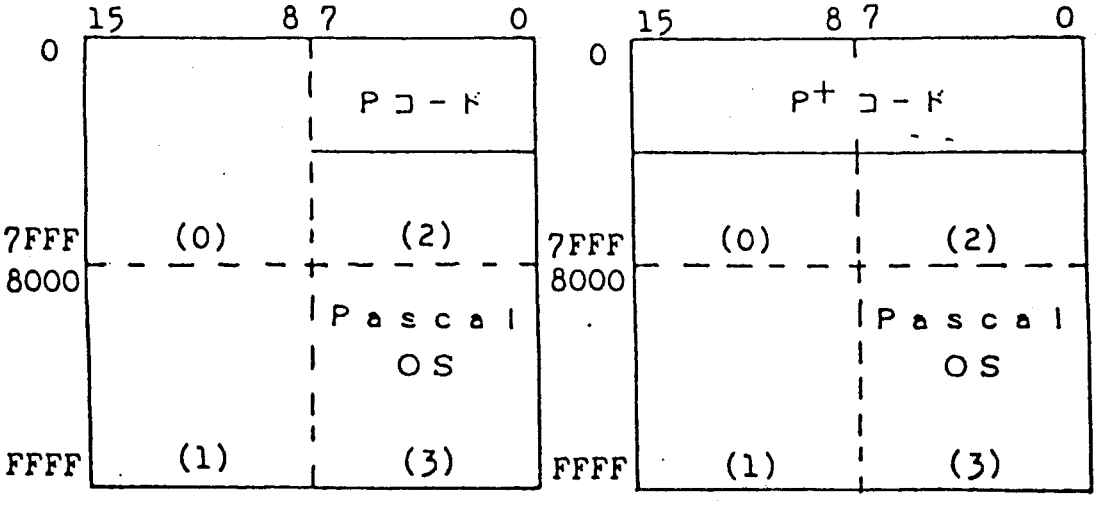
本処理系は、データメモリ上に置かれたPコードまたはP'コードをマイクロプログラムによるインタプリタで解釈・実行することにより高速化をはかるものである。本処理系は、大きく分けるとコード転送部とインタプリタ部から成り立っている。

1) コード転送部

Pascalコンパイラによって、フロッピーディスク上に作成されたPコードファイルをオープンし、ファイル中にあるコードサイズ，目的Pコード，ジャンプテーブル，レキシカル・レベル，プロシージャナンバ，パラメータサイズ，データセグメントサイズなど、実行時に必要なデータをForthマシンシステムのデータメモリの下位バイトに転送する。

ただし、P⁺コードに対しては、ファイル中にある実行開始IPCと目的P⁺コードのみを16ビットデータとしてデータメモリ上に転送する。

図5.6に、データメモリのメモリマップを示す。ペ




(a) Pコード処理系　　(b) P⁺コード処理系

( ) 内の数値は、ページナンバーを表す。

図5．6 データメモリのメモリマップ

ージ3はUCSD-Pascalのos に使用している。P'コードの場合データ転送はP'コードファイルの16ビットデータを取り出し、上位と下位とにデータを分割してページを切り換えながら行う。HLLエンジン側からは16ビット幅でデータの読み出しを行うため、1マイクロステップで参照可能となる。

2) インタプリタ部

本インタプリタは、上に述べたデータメモリ上の中間コード情報を用いて中間コードを解釈・実行するもので、WCS上にマイクロプログラムとして存在する。構成としては、フェッチ部，エミュレート部[]，ジャンプテーブルより成り立っている。フェッチ部，エミュレート部で用いられるMPやBASE等の制御ポインタ，フェッチされたPコードの一時記憶，Pコードファイルのサイズの記憶，インストラクショ

表5．1　レジスタの割り付け

| レジスタ番号 | 役　　割 |
|---|---|
| 1 | Pコードセーブ用 |
| 2 | Pコード情報のサイズ |
| 11 | ローカル変数領域のベースポインタ |
| 12 | グローバル変数領域のベースポインタ |
| 13 | 現在実行中のプロシージャのMSCW部 |
| 14 | ベースプロシージャのMSCW部 |
| 15 | インタプリタのプログラムカウンタ |

ンカウンタには、ALU内のレジスタファイルの一部を利用している。表5.1にレジスタファイルの割り付けを示す。

インタプリタのフローチャートを図5.7に示す。すなわち、インストラクションカウンタが示すアドレスによってデータメモリから1バイト中間コードをフェッチする。そして、その中間コードの値が7F(16)以下であれば定数を表すコードなので直ちにパラメータスタックへプッシュする。80(16)以上であれば、その値に対応するジャンプテーブルのうちの1つへジャンプする。ジャンプテーブル

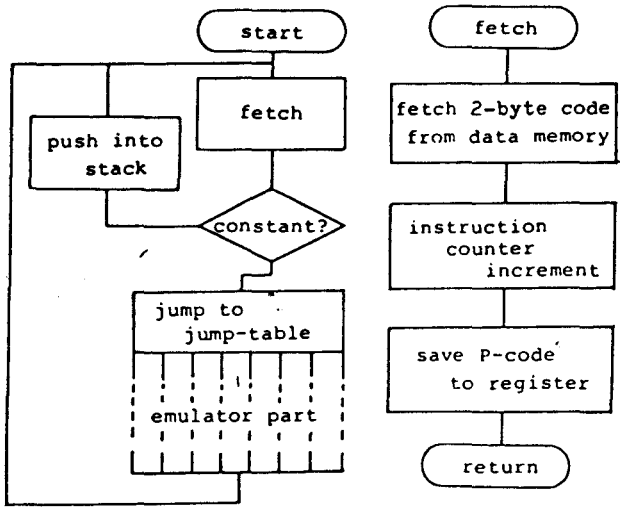



図5.7 マイクロプログラム化インタプリタのフローチャート

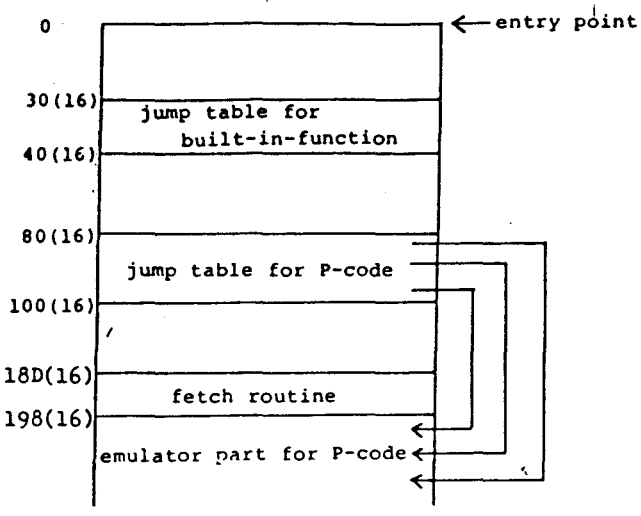



図5.8 WCSのメモリマップ

は中間コードの80(16)からFF(16)に対応して、WCSのアドレス80(16)からFF(16)番地に構成されており、各中間コードの行うべき処理ルーチン(エミュレート・ルーチン)への分岐マイクロ命令で成り立っている。また、組み込み関数用のジャンプテーブルは、30(16)から3F(16)番地にエミュレートルーチンは、198(16)番地以降に構成している。

WCSのメモリマップを図5.8に示す。

### 5.4.3 直接マイクロコード生成型コンパイラ

前節で述べたインタプリタ方式による中間コード実行においては、中間コード及びパラメータのフェッチなど実際にPascalプログラムを実行するのに直接関係のない冗長な操作が必要となる。そこでより一層の高速化を目指すため、入力されたプログラムをマイクロプログラムにまでコンパイルするのが本処理系の手法であり、大幅に実行ステップ数の減少化を表現することができる。

本コンパイラは、UCSD-Pascal コンパイラによって得られるPコードと本研究で開発した中間コード生成型コンパイラによって得られるP'コードそれぞれの中間コードをもとにしてマイクロ目的コードを生成するもので、各中間コードに対応した2種類のコンパイラを備えている。

両中間コードとも一命令当り、平均2～4マイク

ロ命令に展開され、それらを連結することにより、マイクロ目的コードを得ることができる。

基本的には、以上の手順でコンパイルされるが、Pコードに対しては、さらに一層の高速化を達成するために若干の最適化を行ったものを含め、次に示す様な5つのコンパイラを開発した。

1) バージョンA

UCSD-Pascalコンパイラによって得られた目的Pコードをそのまま全てマイクロコードに展開するもの。

2) バージョンB

配列の添字の範囲のチェックを行うPコードである「CHK」を除いてコンパイルするもの。

3) バージョンC

プログラム中の繰り返し制御において、リターンスタックを有効に利用し、ループインデックスの参照においてもリターンスタックを用いる様にコンパイルするもの。

4) バージョンD

バージョンCをさらに拡張し、二重構造から成る繰り返し文の制御において、外側ループのループインデックスの参照にもリターンスタックを用いるもの。

5) バージョンE

バージョンDに加えて、while-do文における繰り返し制御にリターンスタックを利用したもの。

P'コードに対するマイクロ目的コードの生成については特に最適化は行っていないが、バージョンDを用いてPコードをコンパイルした場合と同等、またはそれ以上の効率を有するマイクロ目的コードを得ることができる。

本コンパイラは、2パス方式を採用し、最初のパスで各中間コードに対応したマイクロコードを生成し、次のパスでユーザによって任意に与えられたマイクロコード格納開始アドレスをもとにして、分岐命令の分岐アドレスの更新を行う。また、以上のコンパイラで得られたマイクロ目的コードは、WCS上の実行開始アドレス、ロード開始アドレスと共にマイクロ目的コードファイルとしてフロッピーディスク上に格納される。従って、WCSの任意の領域にマイクロ目的コードをロードすることができる。これは、セグメント管理、分割コンパイルなどシステムの拡張を考えた場合、重要な機能である。

## 5.5 Pascalマシンの性能評価

本節では、Pコードと P'コードを一命令実行するために必要なマイクロステップ数から本システムの

静的特性について述べる。また、マイクロプログラム化インタプリタと直接マイクロコード生成型コンパイラのそれぞれの処理系に対してベンチマークテストを行い、その結果をもとにして動的特性の評価を行う。

### 5.5.1 静的特性の評価

表5.2と表5.3に、2種類の中間コード(Pコードと$P^+$コード)に対応するマイクロ命令のステップ数を示す。表では、インタプリタ方式と直接マイクロコード生成型コンパイラ方式におけるステップ数とそれぞれの比を表している。

表より、Pコードに対するインタプリタ方式の場合、9ステップから55ステップ, $P^+$コードに対するインタプリタ方式の場合、7ステップから19ステップ, コンパイラ方式では、1ステップから9ステップ必要としていることがわかる。また、ステップ数の比は、Pコードにおいては2.2から55であり、$P^+$コードにおいては2.8から17である。さらに、Pコードと$P^+$コードの比較を行うと、20%から70%の実行ステップの減少化が見られる。

中間コードをコンパイルしたときの効果について考察すると、Pコードの「UJP」において、最も大きい比較が示されている。これは、インタプリタ方式では実行時にジャンプテーブルの参照と実効アド

表５．２　Ｐコードに対応するマイクロステップ数

| 種　類 | Ｐコードの例 | インタプリタ | コンパイラ | 比 |
|---|---|---|---|---|
| 定 | ＳＬＤＣ　１ | ９ | １ | ９ |
| 数 | ＬＤＣＩ　１０００ | ３１ | １ | ３１ |
| ロ | ＬＤＯ　２００ | ３８ | ４ | ９．５ |
| ー | ＬＡＯ　３ | ２６ | ３ | ８．７ |
| ド | ＳＩＮＤ　０ | １４ | １ | １４ |
| ス | ＳＴＬ　２００ | ３８ | ４ | ９．５ |
| ト | ＳＲＯ　２０ | ２７ | ４ | ６．８ |
| ア | ＳＴＯ | １７ | ４ | ４．３ |
| 演 | ＡＤＩ | １４ | １ | １４ |
|  | ＳＢＩ | １４ | １ | １４ |
| 算 | ＭＰＩ | １５ | ２ | ７．５ |
| 比 | ＬＥＱＩ　真 | １８ | ５ | ３．６ |
| 較 | ＧＥＱＩ　偽 | １９ | ６ | ３．２ |
| ジ | ＦＪＰ　真 | １６ | ２ | ８ |
| ャ | ＦＪＰ　偽 | ２８ | ２ | １４ |
| ン | ＵＪＰ | ５５ | １ | ５５ |
| プ |  |  |  |  |
|  | ＣＨＫ | ２０ | ９ | ２．２ |
|  | ＣＳＰ　２ | ２３ | １ | ２３ |

表5．3　P+コードに対応するマイクロステップ数

| 種　類 | P+ コードの例 | インタプリタ | コンパイラ | 比 |
|---|---|---|---|---|
| 定数 | SLDC　1 | 7 | 1 | 7 |
| | LDCI　1000 | 16 | 1 | 16 |
| ロード | LGD　200 | 19 | 4 | 4．8 |
| | LGA　3 | 18 | 3 | 6 |
| | SIND　0 | 12 | 1 | 12 |
| ストア | STO | 12 | 1 | 12 |
| 演算 | ADI | 12 | 1 | 12 |
| | SBI | 12 | 1 | 12 |
| | MPI | 13 | 2 | 6．5 |
| 比較 | LEQI　真 | 16 | 5 | 3．2 |
| | GEQI　偽 | 17 | 6 | 2．8 |
| ジャンプ | FJP　真 | 14 | 2 | 7 |
| | FJP　偽 | 19 | 2 | 9．5 |
| | UJP | 17 | 1 | 17 |
| 特殊ループ | RST1 | 12 | 1 | 12 |
| | RST2 | 16 | 5 | 3．2 |
| | FOR1UP | 15 | 4 | 3．8 |
| | FOR2UP | 17 | 6 | 2．8 |

レスの計算が伴うのに対し、コンパイラ方式では実アドレスへのジャンプ命令を直接生成しているためである。また、その次に大きい31という比率を示しているのは、2バイト定数のスタックへのプッシュ命令である。これらのPコードは、多くの場合プログラム中に頻繁に現われる。従って、Pコードをマイクロ命令にコンパイルすることは、マイクロ命令の実行ステップの減少に非常に有効であることがわかる。

また、コンパイラ方式の場合では、多くの中間コードが1から4マイクロステップで実行できることが示されており、仮想計算機であるPマシンとForthマシンとの整合性が高いと考えて良い。

一方、インタプリタ方式におけるPコードとP⁺コードの比較を行い検討する。これも「UJP」で最も効果が大きい。つまり、Pascalコンパイラに分岐アドレスの直接生成機能を持たせたことが非常に効果があったことを表している。また、前述同様2バイト定数プッシュ命令でも効果が大きい。これは、P⁺コードを16ビットマシンを意識したパラメータの構成にしたことが良かったことを示している。

特に、繰り返し制御文であるfor-do文においては、専用の中間コードが生成され非常に大きな効果がある。

繰り返しに関する機能別に、両コードのステップ

表5．4　for－do文における中間コードに
対応するマイクロステップ数

| 機能 | Pコード | P⁺コード | 比 |
|---|---|---|---|
| リミット値の格納 | 116 | 38 | 3．1 |
| インデックスの比較 | | | |
| とインクリメント | 264 | 17 | 15．5 |
| ループからの脱出 | 123 | 23 | 5．3 |
| インデックスの参照 | 38 | 12 | 3．2 |

数とその比を表5.4に示す。表より、P⁺コードを用いると、実行ステップ数が60％から30％に減少している。また、P⁺コードでは、リターンスタックを用いた繰り返し制御を行っているため、メモリ参照回数を減少させることが可能になっている。

以上のことより、ハードウェア機能を考慮して設計したP⁺コードを用いることにより、冗長な実行ステップ数を削減できることが確認できた。

### 5.5.2　動的特性の評価

ベンチマークプログラムを実行したときの実行時間の測定結果を表5.5に示す。表では、本システム上での2種類のマイクロプログラム化インタプリタ（μI-Pascal）とPコードに対する同コンパイラの5種類の最適化バージョン，Z-80マイクロプロセッサ上のUCSD-Pascal，他の数種のPascalシステム，さらにForth，C，FORTRAN言語で同等のプログラムを

表5．5　ベンチマークプログラムの実行時間の測定結果

(sec.)

| | フィボナッチ | 2次関数 | SORT1 | SORT2 | 素　数 |
|---|---|---|---|---|---|
| マイクロインタプリタ (P+コード) | 43.4 | 0.9 | 51.3 | 19.7 | 12.9 |
| マイクロインタプリタ (Pコード) | 77.622 | 2.724 | 165.302 | 75.437 | 26.269 |
| F-Pascal (P+コード) | 7.45 | 0.103 | 8.6 | 3.4 | 2.3 |
| F-Pascal バージョンA | 11.125 | 0.2696 | 19.037 | 7.924 | 3.339 |
| F-Pascal バージョンB | -- | 0.2329 | 13.529 | 6.087 | 2.914 |
| F-Pascal バージョンC | 7.345 | 0.1176 | 12.565 | 5.765 | 2.331 |
| F-Pascal バージョンD | -- | -- | 10.766 | 3.902 | -- |
| F-Pascal バージョンE | -- | -- | -- | -- | 1.598 |
| UCSD-Pascal (Z-80) | 1687.8 | 60.2 | 3411.2 | 1501.0 | 467.8 |
| †UCSD-Pascal マイクロエンジン | -- | -- | -- | -- | 63.0 |
| †NBC Pascal PDP 11/70 | -- | -- | -- | -- | 2.6 |
| Z-80 (アセンブリ言語) | 14.258 | 12.013 | 36.260 | 14.292 | 6.80 |
| FORTH (FORTHマシン) | 2.201 | 0.066 | 2.705 | 1.803 | 1.497 |
| BDS C | 258.1 | 9.76 | 251.6 | 106.1 | 50.2 |
| †Whitesmith C. Z80 | -- | -- | -- | -- | 15.6 |
| FORTRAN (ACOS 900) | 1.729 | 0.046 | 3.201 | 1.187 | 0.737 |

†文献 [47] より。

記述し、実行したときの実行時間の測定結果を示している。

ベンチマークプログラムは、次の4種類を用いた。

1) フィボナッチ数列の計算

2) 2次関数値の計算

3) ソーティング

以上の3種類については、Forthマシンの評価に用いたものと同じである。さらに、次のプログラムを用いた。

4) 素数の計算：0から8190までの間にある素数の個数を求めるプログラムである。方法は、0から8190までの数に対応するサイズ8191の配列を確保し、各要素にその数が素数か否かの真偽値を持たせる。まず、全要素を真にセットしておき、3以上の奇数に注目し、その数に関連する素数でない数のフラグを全要素にわたって偽にセットしながら素数の個数を数える。実際のプログラムを付録に示す。

測定の結果、Z-80 UCSD-Pascalと比較して、インタプリタをマイクロプログラム化することにより、Pコードに対して、17.8～22.1倍，P+コードに対して、36.2～76.2倍，直接マイクロコードまでコンパイルすることにより、Pコードで140.2～223.4倍，P+コードで203.4～584.4倍の実行速度が得られた。また、PDP-11/70のPascalプログラムとほぼ同様の実行時

間で本マシンシステムの処理が行われていることが示されている。

次に、Pコード及びP⁺コードに対する2つの処理系による測定結果をもとに、各ハードウェアモジュールの利用率を求めシステムの評価を行う。

表5.6に、総ステップ数とコンパイル方式の場合の比を示す。表5.7に、フェッチ処理におけるステップ数と比を示す。いずれも比は、P⁺コードのPコードに対する比率を示している。表5.8にマイクロ命令のタイプ別の利用率、表5.9にハードウェアモジュールの機能別の利用率を示す。

次に、以下の3点について詳しく考察する。

1) P⁺コード導入に対する評価
2) コンパイラ方式に対する評価
3) 最適化手法に対する評価

1)については、表5.5よりインタプリタ方式で約1.8～3.8倍，コンパイラ方式で約1.5～2.6倍の高速化が見られる。表5.6からもわかるようにμI-Pascal では、プログラムに関係なくPコードで約89%，P⁺コードで約84%がフェッチ処理で占められている。つまり、インタプリタ方式ではフェッチ処理の効率を上げることが大きな問題となっていることがわかる。

表5.7からは、P⁺コードを導入することによりフェ

表５．６　各処理系における総ステップ数

| | I-Pascal | | F-Pascal | | 比率 |
|---|---|---|---|---|---|
| | Pコード | P+コード | Pコード | P+コード | (%) |
| フィボナッチ | 281,979,836 | 200,967,226 | 52,995,034 | 34,993,824 | 66.0 |
| 2次関数 | 12,330,530 | 3,346,452 | 1,206,644 | 521,312 | 43.2 |
| SORT1 | 758,871,650 | 237,408,415 | 84,975,963 | 39,975,993 | 47.0 |
| SORT2 | 347,783,150 | 91,054,915 | 37,523,463 | 15,500,493 | 41.3 |
| 集数 | 116,775,539 | 56,671,092 | 12,268,004 | 8,698,712 | 70.9 |

表５．７　中間コードのフェッチに要するマイクロステップ数

| | Pコード | P+コード | 比率(%) |
|---|---|---|---|
| フィボナッチ | 228,984,802 | 165,973,402 | 72.5 |
| 2次関数 | 11,123,886 | 2,825,140 | 25.4 |
| SORT1 | 673,895,687 | 197,432,422 | 29.3 |
| SORT2 | 310,259,687 | 75,554,422 | 24.4 |
| 集数 | 104,507,535 | 47,972,380 | 45.9 |

表５．８　マイクロ命令のタイプ別利用率

（単位は%）

| | SORT1 | | 集数 | |
|---|---|---|---|---|
| | Pコード | P+コード | Pコード | P+コード |
| タイプ0 | 22.9 | 28.7 | 23.6 | 30.2 |
| タイプ1 | 1.2 | 1.2 | 3.4 | 1.9 |
| タイプ2 | 3.5 | 0.0 | 1.9 | 0.0 |
| タイプ3 | 0.0 | 0.0 | 0.0 | 0.0 |
| タイプ4 | 21.2 | 35.1 | 21.1 | 19.8 |
| タイプ5 | 30.0 | 18.8 | 25.9 | 24.1 |
| タイプ6 | 10.6 | 10.0 | 12.2 | 13.5 |
| タイプ7 | 10.6 | 6.2 | 11.9 | 10.5 |

## 表5．9　オペレーション別利用率

(%)

| オペレーション | | SORT1 | | 素数 | |
|---|---|---|---|---|---|
| | | Pコード | P+コード | Pコード | P+コード |
| パラメータスタック | トップ | ソース | 39.4 | 41.3 | 44.9 | 47.4 |
| | | デスティネーション | 69.4 | 70.0 | 73.9 | 75.8 |
| | | ( )+ | 39.4 | 40.0 | 42.9 | 45.5 |
| | | −( ) | 68.2 | 70.0 | 72.0 | 75.8 |
| | セカンド | ソース | 33.6 | 30.0 | 32.2 | 30.3 |
| | | ( )+ | 28.9 | 30.0 | 29.1 | 30.3 |
| リターンスタック | | ソース | 1.2 | 17.5 | 3.2 | 6.3 |
| | | デスティネーション | 1.2 | 8.8 | 1.9 | 1.9 |
| | | ( )+ | 1.2 | 8.8 | 1.9 | 1.9 |
| | | −( ) | 1.2 | 8.8 | 1.9 | 1.9 |
| PSIR | | ソース | 3.5 | 0.0 | 1.9 | 0.0 |
| | | デスティネーション | 3.5 | 0.0 | 1.9 | 0.0 |
| メモリ | | 読出し | 8.2 | 6.2 | 9.2 | 8.5 |
| | | 書込み | 2.4 | 3.8 | 4.2 | 5.0 |
| ジャンプ | | 無条件 | 1.2 | 1.2 | 2.7 | 5.3 |
| | | 条件 | 9.4 | 3.8 | 9.2 | 6.7 |
| レジスタ | | R13 | 0.6 | 0.0 | 0.7 | 0.0 |
| | | R14 | 6.5 | 8.8 | 12.1 | 13.5 |

ッ手処理のステップ数が、フィボナッチを除いて約25～46％になることがわかる。特に、ソーティングと2次関数においてその効果が大きい。つまり、P'コードで16ビットマシンを意識したパラメータの構成にしたことが効果的であったことを表している。

また、表5.6のF-Pascalでは各コード自体の特性がわかる。これも、表5.7同様ソーティングと2次関数においてその効果が大きく、ステップ数が約41～47％になり大幅な減少化が図られている。これは、P'コードで配列要素への代入，リターンスタックの利用，ループインデックスの参照などを考慮したことが良かったことを裏付けている。

表5.9よりパラメータスタックの利用率が減り、その分だけリターンスタックの利用率が増している事がわかる。

以上の結果より、P'コードの設計・開発は、Pascalプログラムの高速処理に大きな効果をもたらすことが確認できた。

2)については、マイクロ目的コードにまでコンパイルすることにより約6～10倍の高速化が行われている。また、表5.6より、F-Pascalでは、大幅なステップ数の減少が見られ、マイクロ目的コードを生成する手法が有効であることが示されている。

3)については、配列添字の範囲のチェックを取り除くことによって、1.2～1.4倍の高速化が見られ、特

に配列の添字のチェックの回数が多いソーティングで効果が大きい。また、バージョンCにおいては、リターンスタックの有効利用によりメモリアクセスを削減し、1.4～2.3倍の高速化が示されている。インデックス参照だけを考えても4ステップが1ステップに減少した効果である。さらに、二重ループまで可能なバージョンDでは、1.8～2.0倍の高速化が示されている。また、バージョンEでは、2倍の高速化が実現されている。

以上より、マイクロ目的コードに対する最適化の効果が大きいことがわかった。また、リターンスタックが高速化に不可欠であることが確認できた。

次に、各ハードウェアモジュール別に、その有効性について考察する。

(1) 演算モジュール

表5.8より、タイプ0と1は、ALUの制御命令で演算に用いられるほか、レジスタファイルの読み出しにも利用されている。およそ3，4ステップに1回の割り合いで利用されており、比較的低い利用率を示しているが、これも前述同様にプロシージャコールを多く含むプログラムでは不可欠である。今後、直接レジスタをアクセスする中間コードが必要であると考える。

タイプ2と3の乗算器制御命令は、低い利用率を示しているが、配列のインデックスの計算に用

いられるので、多次元配列の操作を多く含むプログラム等では一層有効に利用されると考えられる。

(2)パラメータスタック

表5.9より、パラメータスタックの利用率は非常に高く、スタックトップの読み出しが約40～47%，書き込みが約70～76%，スタックセカンドの読み出しが約30～34%の割り合いである。本マシンのパラメータスタックの機能が有効に利用されていることを表している。スタックのセカンドを読み出せる機能も不可欠であると考えられる。

スタックポインタの読み出し後の自動増加と自動減少後の書き込みの機能もスタック上のデータへのアクセスと同程度の利用率を示しており、不可欠であると考える。なお、スタックセカンドの自動増加機能の利用率は、トップと比較すると低い値を示している。

パラメータスタックインデクスレジスタ(PSIR)は、配列の添字の範囲のチェックを行うときに用いられるのみである。利用率も低く、不要である。但し、この機能を前提にして、P⁺コードの拡張，コンパイラの修正を行うことにより高効率化に利用できる可能性はある。

(3)リターンスタック

リターンスタックは、比較的低い利用率を示しているが、マイクロ目的コードの最適化を行った

場合に効果的に利用されているのは前述の通りである。また、Pコードにおいても効果的に利用されている。プロシージャコールを含むプログラムにおいては、実行順序制御において不可欠である。

(4)バリアブルストレージとデータメモリ

配列として用いられる場合が多い。また、変数用領域などとしても用いられている。表5.9より、メモリ読出し及び書き込みの利用率は、6.2%～9.2%，2.4%～5.0%となっているが、1マイクロサイクルでメモリ書き込みができる機能を持っており、高速化に役立っていると考える。従って、ソーティングなどのような配列データの入れ替えを行う場合には、非常に有効である。

(5)バイパスコントローラ.

表5.8より、タイプ4とタイプ5の利用率の和がバイパスコントローラの利用率で約44%～54%である。このことから、データ転送用としてバイパスコントローラを設けたことが有効であったと考えられる。およそ2ステップに1回の割り合いで利用されている。これは、データ転送の占める割り合いが大きい事を示しており、高速データ転送用のバイパスコントローラが有効に利用されていることを表している。

## 5.6 結言

本章では、従来の計算機上でPascalプログラムを実行する際の欠点を洗い出し、それらを考慮して、HLLエンジン上にPascalマシンを実現する手法と、その静的、動的特性の評価について述べた。Pascalマシンを実現するために開発したマイクロプログラム化インタプリタと直接マイクロコード生成型コンパイラの2つの方法について述べた。そして、それらの方法を従来のPコードについてと、HLLエンジンの機能に適合する様修正を加えられた新中間コードであるP'コードとについて適用した。本マシンの静的特性の評価の結果、Pコードに対応するマイクロ命令の実行ステップ数は1ステップから9ステップで平均約3ステップであり、PマシンとHLLエンジンの整合性が高いことが確認できた。インタプリタの場合も含めると、P'コードを用いることにより、さらに20%から70%ステップ数を減少させることができ、ハードウェア構造に適した中間コードの効果が明らかになった。

ベンチマークプログラムの実行時間の測定結果より、本マシンはZ-80マイクロコンピュータ上のUCSD-Pascalと比較して最高約584倍の実行速度を有することを示した。また、HLLエンジン内の各ハードウェアモジュールの利用率を算出し、本エンジンのハードウェアモジュールがPascalプログラムの高速実行に効果的に動作していることを確認した。

その結果、強力な演算用スタックの機能は非常に有効で、ループ制御用のスタックの利用も極めて効果的であることがわかった。

# 第6章　結論

高級言語マシンの一形態として、スタック機能を中心とする計算機構造を持つスタックマシンがある。

本論文では、High Level Language (HLL) エンジンと呼ぶスタックマシンについて論じた。設計にあたっては、スタックに対する依存性が極めて高いForth言語を基礎に置いた。本エンジン上に、①Forth言語を対象としたForthマシンと、②Pascal言語を対象としたPascalマシン、を実現し、それぞれの性能評価を行った。

第2章では、Forth言語の特徴と言語処理系の構造について述べ、HLLエンジンの基本設計について論じた。また、マイクロコンピュータ上で、ソフトウェアによる仮想スタック機構を用いると、レジスタのみを用いた場合に比べて約1/20に実行速度が低下することを、ベンチマークプログラムによるテストを行うことによって明らかにした。その結果より、Forthプログラムの高速実行における高機能スタックの必要性を確認した。また設計に関しては、次の様に方針を定めた。

① マイクロプログラム制御方式を採用する。

② Z-80マイクロコンピュータをホストコンピュータとする複合計算機システムの形態をとる。

③ ハードウェアによる強力なスタックを備える。

④ Forth言語の機能を有効に利用して、ユーザプログラムは全てマイクロプログラムに翻訳する。

⑤ 変数専用高速メモリを備える。

⑥ データ転送専用のハードウェアを設ける。

第3章では、前章で論じた基本方針に従って設計されたHLLエンジンシステムの全体構成を示した。また、HLLエンジンの有するハードウェアモジュールの、個々の機能や構成について述べた。特に、スタックのトップとセカンドにある値を同時に読み出せる機能を持たせた演算用スタックであるパラメータスタックについては、内部動作も含めて詳述した。また、マイクロ命令の構成について述べ、本エンジンのマイクロ命令は、Forth言語の基本命令の多くを1マイクロ命令で実行できる機能を持つことを示した。さらに、本システムのハードウェアサイズを示し、実装に際して注意を払った点を述べた。

第4章では、本HLLエンジン上に、Forthプログラムを高速に実行できるForthマシンを実現する方法について論じた。まず、ホストコンピュータ上に開発した、Forth処理系の中核を為すテキストインタプリタについて述べた。Forthプログラムを翻訳した結果、ディクショナリエントリと呼ばれる、一種の目的コードが生成される。本システムにおいて、ディクショナリエントリは、ホストコンピュータ上

とHLLエンジンの制御記憶上に存在する。そこで、双方を効率良く管理できるテキストインタプリタの構成と機能について考察し、それに従って開発を行った。また、Forthプログラム中で、マイクロ命令を利用できるマイクロアセンブラの開発について述べた。さらに、プログラム中でForth言語の基本関数を利用したとき、関数呼び出し命令でなく、直接その関数に対応するマイクロ命令を生成する、直接マイクロコード生成型コンパイラを開発し、それについて述べた。続いて、ベンチマークテキストを用いて、本Forthマシンの性能評価を行い、以下の結果を得た。

① Z-80マイクロコンピュータ上でのForthプログラムと比較して、最高522倍の実行速度が得られた。

② 強力なパラメータスタックの機能は総実行ステップの40～60%で利用されており、Forthプログラムの高速実行に有効であることを確認した。特に、スタックのセカンドにある値を読み出せる機能は、高速実行に不可欠であることが明らかになった。

③ 変数専用高速メモリと、その内部バスとの接続関係は、配列操作に有効であることがわかった。

④ データ転送専用ハードウェアは、約50%の比

率で利用されており、効果的に働いていることが確認できた。

第5章では、HLLエンジン上にPascal言語を対象としたPascalマシンを実現する方法について論じた。Pascalプログラムは、多くの場合、Pマシン(Pseudo-machine)と呼ばれる仮想スタックマシンの機械語であるPコード(Pseudo code)に翻訳され、Pコードを解釈することにより実行される。そこで、まず従来の計算機上でPマシンを実現する際に生じる欠点について考察した。そして、Pascalマシン実現の方針について論じた。さらに、Pコードについても考察し、HLLエンジンの構造に適合するように、機能を修正・追加した$P^+$コードを新たに提案した。また、$P^+$コードを生成するコンパイラを開発した。本Pascalマシンは、①P($P^+$)コードインタプリタをマイクロプログラム化する方法と、②P($P^+$)コードをマイクロ目的コードに翻訳し、さらに若干の最適化を施す方法とを用いて実現した。

続いて、本Pascalマシンの静的特性評価と、ベンチマークプログラムを用いた動的特性評価を行った。そして、以下に示す結果を得た。

① マイクロコンピュータ上のPascalと比較して、最高584倍の実行速度が得られた。

② $P^+$コードを用いたとき、従来のPコードの場合の最高約4倍の実行速度向上が得られ、計算

機構造に適合した中間コードの効果が明らかになった。

③ パラメータスタックは、総実行ステップ数の30～75％で利用されており、Pascalプログラムの高速実行に重要な役割りを果たしていることがわかった。

その他、各ハードウェアモジュール毎の利用比率から、HLLエンジンは、Pascalマシンとしても有効に動作していることを明らかにした。

以上、本HLLエンジンシステムは、複数の言語に対する高級言語マシンを実現し得る構造と機能を持っていることを示した。

Pascalマシンの場合の結果より、Pascal言語に類似した言語構造を持つC言語を対象にした高級言語マシンも、容易に実現できると考える。また、高速の変数代入操作が要求されるProlog言語等に対しては、変数専用高速メモリが有効に働くと考えられ、興味深い。これら他言語に対する適用は、今後の課題としたい。

# 参　考　文　献


[1] David Gries:"Compiler Construction for Digital Computers", John Wiley & Sons, Inc.(1971).

[2] Alfred V. Aho, Jeffrey D. Ullman:"Principles of Compiler Design", Addison-Wesley Publishing Company(1979).

[3] 中田：〝コンパイラ〟，産業図書（１９８１－０９）．

[4] Yaohan Chu, Abram Marc:"Programming Languages and Direct-Execution Computer Architecture", Computer, pp.22-32(1981-Ø7).

[5] Yaohan Chu:"Architecture of a Hardware Dara Interpreter", IEEE Trans. on Computers, Vol.C-28, No.2(1979-Ø2).

[6] L. W. Hoevel:"'Ideal' Directly Executed Languages: An Analytical Argument for Emulation", IEEE Trans. on Computers, Vol.C-23, No.8, pp.759-767(1974).

[7] Michael J. Flynn, Lee W. Hoevel:"Execution Architecture: The DELtran Experiment", IEEE Trans. on Computers, Vol.C-32, No.2(1983-2).

[8] Stanley Habib:"Editor's Overview--Special Section on Microprogramming", ACM Computing Surveys, Vol.12, No.3(198Ø-Ø9).

[9] J. A. Fisher:"Trace Scheduling:A Technique for Global Microcode Compaction", IEEE Trans. on Computers, Vol.C-30, No.7(1981-07).

[10] David Lasdskov, Scott Davidson, Bruce Shriver, Patric W. Mallett:"Local Microcode Compaction Techniques", ACM Computing Surveys, Vol.12, No.3(1980-09).

[11] Subrata Dasgupta:"Some Aspects of High Level Microprogramming", ACM Computng Surveys, Vol.12 No.3(1980-09).

[12] S. Davidson, D. Landskov, B. D. Shriver, P. W. Mallett:"Some Experiments in Local Microcode Compaction Machines", IEEE Trans. on Computers, Vol.C-30, No.7(1981-07).

[13] S. S. Reddi, Edward A. Feustel:"A Restructurable Computer System", IEEE Trans. on Computers, Vol.C-27, No.1(1978-01).

[14] ELLIOTT I. ORGANICK. 土居 訳:"計算機システムの構造－バロース大型計算機シリーズ－". 共立出版. (1978).

[15] O. -J. Dahl, E. W. Dijkstra, C. A. R. Hoare:"Structured Programming", Academic Press(1972).

[16] 滝，金田，前川："ＬＩＳＰマシンの試作 － アーキテクチャと ＬＩＳＰ言語の仕様 －"，情報処理，２０，６，pp．４８１－４８６（昭５４－０９）

[17] 服部，林，秋元："ＬＩＳＰマシンＡＬＰＨＡの概要"，第２６回情報処理学会全国大会，５Ｐ－７，（昭５８）．

[18] Tomlinson G. Rauscher, Ashok K. Agrawala:"Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming", IEEE Trans. on Computers, Vol.C-27, No.11(1978-11).

[19] Tomlinson G. Rauscher, Phillip M. Adams:"Microprogramming:A Tutorial and Survey of Recent Developments", IEEE Trans. on Computers , Vol.C-29, No.1(1980-01).

[20] "Using FORTH", FORTH, Inc.(1979).

[21] John S. James:"FORTH", Dr.DObb's Journal of Computer Calisthenics & Orthodontia(1978-05).

[22] "The Am2900 Family Data Book", Advanced Micro Devices, Inc.

[23] David M. Bulman:"Stack Computers", COMPUTER, Vol.10, No.5, pp.14-28(1977-05).

[24] Russell P. Blake:"Exploring a Stack Architecture", COMPUTER, Vol.10, No.5, pp.30-38(1977-05).

[25] F.G.Duncan:"Stack Machine Development: Australia, Great Britain, and Europe", COMPUTER, Vol.10, No.5, pp.50-52(1977-05).

[26] D. G. Bobrow, B. Wegbreit:"A Model and Sstack Implementation of Multiple Enviroments", CACM, Vol.16, No.10(1973-10).

[27] D. W. Barron:"Pascal-The Language and its Implementation", John Wiley & Sons Ltd.(1981).

[28] K. A. Shillington, G. M. Ackland:"UCSD Pascal Version I.5", Institute for Information Systems(1978-09).

[29] Martin S. Ewing:"THE CALTECH FORTH MANUAL", Second Edition, A Technical Report of the OWENS-VALLEY RADIO OBSERVATORY(1978-06).

[30] "LABFORTH", Laboratory Software Systems, Inc.(1978).

[31] 金田，和田：“ＦＯＲＴＨとは何か”，インターフェース，ＣＱ出版（１９８１－０８）．

[32] 和田，金田，前川：“ＦＯＲＴＨマシンの試作”，信学会情報・システム部門全国大会，Ｓ４－５（昭５６）．

[33] 和田，金田，前川："ＦＯＲＴＨマシンシステムのシステム設計とハードウェア構成"，信学論，Ｖｏｌ．Ｊ６５－Ｄ，Ｎｏ．３，ｐｐ．３３８－３４５（１９８２－３）．

[34] 和田，金田，前川："ＦＯＲＴＨマシンの試作"，信学技報，ＥＣ８０－３５（１９８０－０９）．

[35] "FIG-FORTH INSTALLATION MANUAL", FORTH Interest Group(1979-05).

[36] "FORTH DIMENSIONS", FORTH Interest Group, Vol.2, No.1.

[37] "FORTH REFERENCE MANUAL", FORTH, Inc.(1980-04).

[38] "DEC PDP-11 and LSI-11 User's Supplement to the poly FORTH Reference Manual", FORTH, Inc.(1980-04).

[39] "CPU-SPECIFIC DOCUMENTATION FOR polyFORTH ON THE INTEL MDS-800", FORTH, Inc.(1979-11).

[40] Charles H. Moore, Elizabeth D. Rather:"The FORTH Program for Special Line Observing", Proc. IEEE, Vol.61, No.9, pp.1346-1349(1973-09).

[41] 和田，金田，前川："ＦＯＲＴＨマシンシステムの評価"，信学論，Ｖｏｌ．Ｊ６５－Ｄ，Ｎｏ．３，ｐｐ．３４６－３５３（１９８２－３）．

[42] 和田，金田，前川："ＦＯＲＴＨマシンシステムの評価"，電気関係学会関西支部連合大会，Ｇ６－１８（１９８１）．

[43] Yukio Kaneda, Koichi Wada, Sadao Maekawa:"High-Speed Execution of Forth and Pascal Programs on the High-Level Language Machine", ninth EUROMICRO symposium on microprocessing and microprogramming(1983-09).

[44] 和田，仲辻、金田，前川："スタック構造を持つＰａｓｃａｌマシンシステムの開発"，第２６回情報処理学会全国大会，４Ｎ－１２（昭５８）．

[45] 和田，仲辻，金田，前川："高級言語向きスタックマシン上でのＰａｓｃａｌマシンの実現と評価"，信学技報，ＥＣ８２－５８（１９８２－１２）．

[46] 和田，仲辻，金田，前川："高級言語向きスタックマシン上でのＰａｓｃａｌマシンの実現と評価"，信学論，Ｖｏｌ．Ｊ６６－Ｄ，Ｎｏ．４，ｐｐ．３６９－３７６（１９８３－４）．

[47] Jim Gilbreath:"A High-Level Language Benchmark", BYTE Publications Inc.(1981-09).

[48] Michael J. Flynn:"Directions and Issues in Architecture and Language", Computer, pp.5-22(1980-10).

[49] 中田：“スタック・マシンのための最適コード生成のアルゴリズム”，情報処理，２２，５（１９８１）．

[50] John Couch:"Semantic Structures for Efficient Code Generation on a Stack Machine", COMPUTER, Vol.10, No.5, pp.42-48(1977-05).

[51] M. D. Prycker:"A Performance Analysis of the Implementation of Addressing Methods in Block-Structured Languages", IEEE, Vol.C-31, No.2(1982).

[52] J. A. Stankovic:"The Types and Interactions of Vertical Migrations of Functions in a Multilevel Interpretive System", IEEE Trans. on Computers, Vol.C-30, No.7(1981-07).

[53] "AN INTRODUCTION TO CP/M FEATURES AND FACILITIES", DIGITAL RESEARCH, Inc.(1978-01).

[54] Alfred C. Hartmann:"A Concurrent Pascal Compiler for Minicomputers", Springer-Verlag(1977).

付　　録

ベンチマークプログラム

1）　フィボナッチ数列の計算

（a）　Ｆｏｒｔｈ言語によるプログラム

```
( FIBONACCI SEQUENCE )
DECI
: 1FIBO 0 DO SWAP OVER + LOOP ;
: FIBONACCI 100 0 DO
                  1 1 10000 1FIBO
                  2DROP
                  LOOP;
```

（b）　Ｐａｓｃａｌ言語によるプログラム

```
program FOBONACCI;

const kaisu = 10000;
var i,j,old,last,new : integer;

begin
   old := 1; last := 1;
   for i := 1 to 100 do
   begin
      for j := 3 to kaisu do
      begin
         new := old + last; old := last; last := new
      end;
    end;
end.
```

2)　2次関数値の計算

(a)　Ｆｏｒｔｈ言語によるプログラム

```
( QUADRATIC FUNCTION )
DECI
: QUAV 201 1 DO
                  I DUP 60 - * 1100 + I !V
               LOOP;
: VQUA 100 0 DO QUAV LOOP ;
```

(b)　Ｐａｓｃａｌ言語によるプログラム

```
program QUADRATIC;

type  list = array [1..200] of integer;
var   y : list;
      i,j : integer;

begin
   for j := 1 to 100 do
      for i := 1 to 200 do
        y[i] := i*(i-60) + 1100
end.
```

3) ソーティング

(a) Ｆｏｒｔｈ言語によるプログラム

```
( SORTING )
DECI
: PUTDATA 1001 1 DO I DUP !V LOOP ;
: SORT  2 PICK DROP
  1000 1 DO
         I DUP @V
         1001 I 1+ DO
                  I @V OVER OVER
                  < IF SWAP I !V DUP (PICK) !V
                    ELSE DROP
                    THEN
                  LOOP
                  2DROP
         LOOP;
```

(b) Ｐａｓｃａｌ言語によるプログラム

```
program SORTING;

type list = array|1..1000| of integer;
var  idata : list;
     i,j,k,l,m,work : integer;
begin
   for i := 1 to 1000 do idata|i| := i;
   for m := 1 to 999 do
   begin
      k := m + 1;
      for j := k to 1000 do
      if idata|m| < idata|j| then
         begin
            work := idata|m|;
            idata|m| := idata|j|;
            idata|j| := work
         end
   end
end.
```

## 4） 素数の計算

### （a） Ｆｏｒｔｈ言語によるプログラム

```
( ERATOSTHENES SIEVE PRIME NUMBER )
DECI
: SIZE 8191 ;  : FLAGS 32768 ;
: FILL ROT ROT OVER + SWAP DO DUP I !D LOOP DROP ;
: DO-PRIME FLAGS SIZE 1 FILL
0 SIZE 0 DO
         FLAGS I + @D
         IF I DUP + 3 + DUP I +
         BEGIN
            DUP SIZE <
            IF 0 OVER FLAGS +
             !D OVER + 0 ELSE 1 THEN
         END
         2DROP 1+
         THEN
         LOOP;
```

### （b） Ｐａｓｃａｌ言語によるプログラム

```
program PRIME;

const size = 8190;
var   flags : array[0..size] of boolian;
i,prime,k,count,iter : integer;

begin
   writeln('10 iteration');
   for iter := 1 to 10 do
   begin
      count := 0;
      fillchar(flags,sizeof(flags),true);
      for i := 0 to size do
      if flags[i] then
      begin
         prime := i + i + 3;
         k := i + prime;
         while k <= size do
         begin
            flags[k] := false;
            k := k + prime
         end;
         count := count + 1
      end;
   end;
writeln(count,' primes');
end.
```

# 謝　　辞

本研究の全過程を通じて、直接理解ある御指導、御鞭撻を賜った神戸大学院自然科学研究科システム科学専攻　前川禎男教授（神戸大学工学部システム工学科）に心から感謝の意を表する。また、熱心な御指導、御鞭撻を賜ったシステム科学専攻　金田悠紀夫助教授（システム工学科）に深く感謝する。本研究の実施にあたって、終始、親切な御指導、御助言をいただいた、システム科学専攻　平井一正教授（システム工学科）、松本治弥教授（計測工学科）に厚く御礼申し上げる。

大学院博士過程において、筆者の履習指導委員として御指導、御教示いただいた、システム科学専攻瀬口靖幸教授（システム工学科）、三好旦六教授（電子工学科）に心から御礼申し上げる。三好旦六教授には、筆者が、神戸大学工学部電気工学科に在学中より、御指導ならびに激励をいただいた。重ねて、厚く御礼申し上げる。

本研究を行うにあたって、有益な御助言、御討論をいただいた新世代コンピュータ技術開発機構研究所の滝和男氏に深く感謝する。また、本研究において多大なる御協力をいただいた元神戸大学工学部システム工学科第4講座　大西誠氏、西脇智仁氏、元神戸大学大学院工学研究科システム工学専攻　仲辻

俊之氏に深く感謝する。システム開発に御協力いただいた自然科学研究科システム科学専攻の田村直之氏、小畑正貴氏に御礼申し上げる。また、種々の面でお世話になったシステム工学科第4講座の小畑美保子技官、ならびに第4講座諸氏に深く感謝する。また、本論文作成に多大なる御尽力をいただいた高橋未佳氏に心より御礼申し上げる。