計算機アーキテクチャ 74－1
（１９８９．１．２６）

構文木インタプリタＰＡＴＩＥのアーキテクチャ

闞 暁薇　　板野肯三

筑波大学　工学研究科　　筑波大学　電子・情報工学系

対話性の高いプログラミング環境を実現するための一手法として、構文木を内部表現としたインタプリタのハードウェアを実現することを目標に、現在、そのプロトタイプをＰＬ／０用に設計した。構文木インタプリタのアーキテクチャは、機能によって、フローコントロール・ユニット、トラバース・ユニットとデータ・ユニットの３つの部分に分けられている。ハードウェアとしては、これらをパイプライン型に結合して、ハードウェアを設計しやすくすると同時に、実行の高速化を計っている。今回設計した構文木インタプリタでは、ハードウェアを単純にするために、意味解析はソフトウェアで行い、実行速度に実質的に関わる意味実行の部分のみをハードウェアとして抽出することで、ハードウェアの実現性を高めている。
本論文では、このような方針で設計された、構文木インタプリタのハードウェアの構成と動作原理について説明する。

Architecture of a Parse Tree Interpreter PATIE

Xiaowei Kan　　and　　Kozo Itano
Doctoral Program in Engineering, Institute of Information Sciences and Electronics, University of Tsukuba
Tennoudai 1-1-1, Tsukuba-shi, Ibaraki-ken, 305 Japan

Towards the realization of a highly interactive programming environment, a parse tree interpreter is aimed to be implemented as a hardware procssor. Current prototype is designed to evaluate the feasibility of the hardware realization, using a rather simple program language PL/0. The architecture is organized as three functionally indepent units: flow-control unit, traverse unit, and data unit. These units are connected as a pipline to get higher execution speed. To simplify the hardware realization, sematic analysis is excluded from the current hardware design of the parse tree interpreter, and only the execution art is included. In this paper, detailed hardware organization and operation principles of the interpreter is explained.

1．はじめに

高級言語によるプログラミングの大部分は、プログラムの誤りを発見するためのテストやデバッグを行う時の編集と実行の繰り返しで占められている。しかし、プログラマによって行われるこの繰り返しは、無駄な作業や待ち時間がかかり、プログラミングの対話性を低くしている。このため、編集と実行の繰り返しを単一のシステムで一貫して支援することによって、テストやデバッグの過程から不必要な作業や待ち時間を取り去ることができれば、より対話性の高いプログラミング環境を提供することができる[1,6-8]。

編集系と実行系の統合には、様々な方法が考えられるが、その中で、構造エディタと構文木インタプリタを組み合わせる方法は、編集と実行をより密接に組み合わせることができることが知られている[3-5,9-12]。しかし、この方法は、構文木インタプリタを実行に用いるので、コンパイラを用いて実行するのに比べて、実行速度が約百分の一と低い[8]。この欠点を解決するため、構文木インタプリタをハードウェア化することを考えた。

構文木インタプリタをハードウェア化する時に問題となるのは、簡単なハードウェアとして実現できるかどうかということ、コンパイルされたコードを実行するのに比敵する性能が得られるかどうかということである。しかし、CやＰＡＳＣＡＬのような本格的な言語を対象にして、ただちにハードウェアの実現するのは難しいので、準備として、とりあえず簡単な言語であるＰＬ／０が実行できる構文木インタプリタ（PATIE-0: Parse Tree InterperatEr for PL/0）をハードウェアとして設計し、本格的な設計の指針を得ることにした[13]。

本論文では、対象言語とプログラムの表現について簡単に説明してから、今回設計したハードウェアの構造と制御の方式について説明する。

2．　ＰＬ／０の構文木の形式

インタプリタの設計を容易にするために、対象言語としてＰＬ／０を選んだ。ＰＬ／０では、ブロック構造はサポートされているが、標準的なサンプルプログラムであるソートなどがプログラムできないため、アレイや関数や入出力の機能などを導入して、機能を強化した。この機能の強化により、手続き型言語のもつ基本的な枠組は評価できると考えている。

現在、実現を計画しているプログラミングシステムは大きく分けて、構造エディタ、意味解析器、ハードウェアの構文木インタプリタ（PATIE-0）の三つで構成されており、これらは、編集や実行の対象となっているプログラムの構文木表現を共有する形で結合されている。具体的には、ソースプログラムは、文字列情報をトークン列として表現し、構文情報を構文木の形で表現していて、構造エディタは、この二つの表現上でプログラムの編集を行う。現在、実現している方式では、意味解析は、シンボルテーブルを作成しないで、構文木のノードの中に意味情報を保存するように設計されており、プログラムを実行する前に、一度、意味解析器で前処理をしてから、構文木インタプリタがこの属性付き構文木の実行を行うようになっている。

図１に構文木のノードの形式を示す。今回、対象言語にしたＰＬ／０は、５８種類のノードが定義されている。各ノードは、最初に対応するトークンのノードへポインタと、構文木の親ノードへポインタを持ち、次に、このノードの種類を表すコード、二つの子ノードへポインタ、意味解析の結果を格納する領域（オペランド）を持つ。一般には、各ノードが持つノードの数は不定である。例えば、　s→ if cond then s1 else s2という生成規則に対応する構文木のノードは三つの子ノードを持つ。しかし、構文木インタプリタのハードウェアを単純にするために、今回の設計では、構文木の各ノードは最大二つの子ノードしか持てないように制限した。また、ノードのアドレスはコードを格納するワードを指すように設計してある。これらのことは、高速に構文木を実行することにもつながっている。

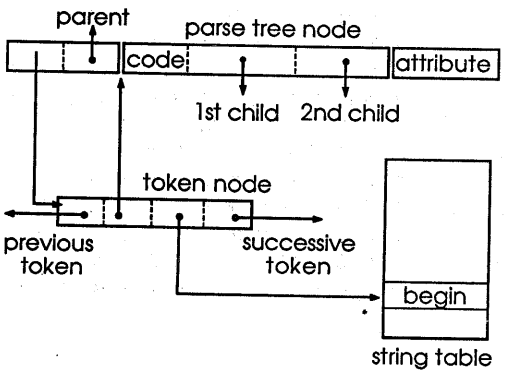

図１　構文木とトークンノード

具体的なノードの形式を図２に示す。ノードの形式は、全部で、四種類ある。type1とtype3では、子ノードが二つ指定されるが、type2とtype4では、ポインタの一部がオペランドになる。type3では、オペランドを指定するのに次のワードを使用する。

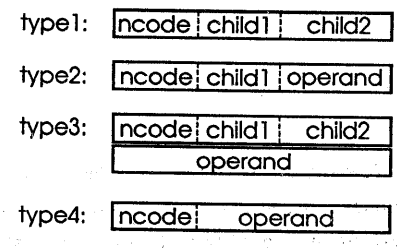

図２　ノードの形式

3．構文木の実行

構文木インタプリタは、前述のような属性付き構文木をプログラムとして解釈しながら実行する。ところで、この

実行は、主に構文木をたどるための実行と各ノードに定義されている意味の実行の二つの部分に分けられる。そこで、この二つに対応して、構文木のトラバースのメカニズムと意味実行のメカニズムを考えて設計した。

3．1　構文木のトラバースのメカニズムの設計

普通の計算機の命令の実行と異なり、構文木の各ノードを頂点とする部分木の実行には、そのノードの子ノードを頂点とする部分木の実行が含まれている。そのため、一つのノードの実行が完全に終わる前に、そのノードの実行を途中で中断し、子ノードを呼び出して子ノードを実行しにいかなければならない。さらに、子ノードの実行が終わってから、またそのノード（親ノード）へ戻ってきて、残っている親ノードの実行をつづけることが必要である。一般に、各ノードの実行は、このように中断と再実行を繰り返しながら行われる。一般には、子ノードの実行が終わったら親ノードへ戻らなければならないが、ある子ノードの実行が終わったら親ノードで実行すべきことがなければ、親ノードへ戻る必要がないので、最適化を行うことができる(tail recursion optimization)。

このような方針に基づいて、構文木をトラバースするために以下の三つの基本的な操作を用意した。

- call：親ノードがこの操作によって、自分の子ノードを呼び出して、この子ノードを頂点したの部分木を実行しに行く。子ノードの実行が終わった時、親ノードへ戻る必要がある。
- goto：親ノードがこの操作によって、自分の最後の子ノードを呼び出して、この子ノードを頂点としての部分木を実行しに行く。子ノードの実行が終わった時、親ノードへ戻らず、より上の呼び出された親ノードへ戻る（tail recursion）。
- return：子ノードがこの操作によって、呼ばれた親ノードへ戻る。

ところで、子ノードを呼び出す時には、中断された親ノードの実行の中間状態を保存しなければならない。このために、PATIE-0ではスタックを用いてこの状態を保存することにした。callを行う時に、中断する親ノードの状態をスタックにプッシュして保存し、returnを行う時に、スタックから中断した親ノードの状態を回復してから、親ノードの実行をつづける。gotoを行う時には、親ノードの状態を保存する必要がないので、スタックにプッシュしないまま、子ノードの実行を行いにいく。

3．2　制御構造の実行

ここで使用しているＰＬ／０では、制御構造に関連するconstructとしてif文、while文、関数の呼び出し文、return文の四つがある。これらを実行するメカニズムを設計するときに、構文木をトラバースするメカニズムやオペランドをフェッチするメカニズムなどと組み合わせて設計することにした。具体的には、先に説明した三つの基本操作と組み合わせ次のような八つのプリミティブを設計した。これらのプリミティブは次に実行すべきノードのアドレスを保存するPC、オペランドを保存するoperand、構文木をトラバースする途中で現れるすぐに実行できないノードのアドレスと保存するスタックであるPC_STACKを用いて定義される。

- call left (PC←child1; PCSTACK←child2)
  一番目の子ノードをcallする。child1を次に実行するべきノードのアドレスとして、PCにセットし、child2をあとで実行するべきノードのアドレスとして、PC_STACKにプッシュして保存する。
- call right (PC←child2; PC_STACK←child1)
  二番目の子ノードをcallする。child2を次に実行すべきノードのアドレスとして、PCにセットし、child1をあとで実行すべきノードのアドレスとして、PC_STACKにプッシュして保存する。
- goto right (PC←child2)
  二番目の子ノードへgotoする。child2を次に実行すべきノードのアドレスとしてPCにセットする。
- goto left with operand
  (PC←child1;operand←next word)
  一番目の子ノードへgotoし、同時に拡張したワードからオペランドを読み込んでくる。child1を次に実行すべきノードのアドレスとしてPCにセットする。
- goto right with operand
  (PC←child2;operand←next word)
  二番目の子ノードへgotoし、同時に拡張したワードからオペランドを読み込んでくる。child2を次に実行すべきノードのアドレスとしてPCにセットする。
- return with operand(PC←pop; operand←value)
  意味解析によって生成され、葉ノードに格納されている変数のアドレスや定数などを読みだしてくる。読みだされたデータはoperandに保存する。次の枝へ実行に行くために、PC_STACKをポップし、次に実行すべきノードのアドレスとして、PCにセットする。
- conditional branch
  (if cond then PC←child1; else PC←child2)
  if文による選択分岐のために用いる。各ノードが最大二つ子ノードしか持てないので、if文は、二つのノードに分けれて表現されていて、実行も二段階に分けて行う（図３）。まず、最初のノードifでは、call leftを実行して、条件式の評価のための木の制御を移す。この条件式の部分の評価が終わると結果がcondに格納されて返って来る。この段階で、二番目のノードif2で、このプリミティブconditional branchを実行することによってchild1またはchild2の一方が選択さ

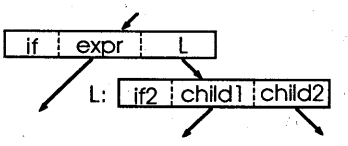

図３．　if文の実行

れ、分岐が起こる。つまり、condがtrueだったらchild1が、また、falseだったらchild2が次に実行されるアドレスとしてPCにセットされる。

・loop(while cond do PC←child1;)

while文による繰り返しを行うのに用いる。ifと同じようにwhile文も2段階にわけて実行する（図4）。L1、L2はノードwhile、while2に付けているラベルである）。まず、最初のノードwhileでは、プリミティブcall leftを実行して、条件式の評価のための木に制御を移す。そして、この条件式の評価が終わると結果がcondに格納されて返ってくる。そして、この段階で、このプリミティブloopを実行することによって、ループ体を実行するかどうかを判断する。condがfalseの時は、何にもせずに、親ノードへ戻る。trueの時は、実行に入る。ノードwhile2で、単にプリミティブcall leftを実行する。ここで、child2はノードwhileのアドレスであるので、自然にノードwhileに戻ってもう一回ループする。

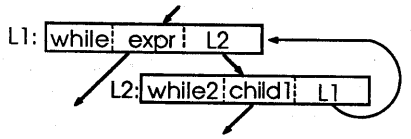

図4　while文の実行

3．2　データ構造の実行

構文木インタプリタは、先の構文木のトラバースのアルゴリズムを用いて構文木をたどりながら、各ノードに定義されている意味を実行することによって、構文木で表現されたプログラムを実行する。ここでは、言語ごとに定義される意味のうち、データの転送や演算を効率良くハードウェアで実行するための実行モデルについて説明する。

（1）フレーム管理

フレームの管理はディスプレイを用いる方法を採用した。ブロックの深さは0から6まで許すこととし、ディスプレイは7つのフレームポインタの組(FP0,FP1,....FP6)として、実現することとした。ここでは、FPiがブロックのレベルiに対応する。フレームの退避と回復は以下の二つプリミティブを用いて行う。また、return文などでブロックから脱出を行う時、現在のフレームだけではなく、ハードウェアの内部にあるスタックに記録している実行状態も要らなくなるので、そのため、block_in_flag_setとblock_out_flag_popのプリミティブを設計しており、脱出時のスタック処理を行う。

・make_new_frame：　次の3つの操作によって、新しいフレームを生成する。

```
MEM[SP] ← FPi
FPi     ← SP
SP      ← SP + val
```

ここで、FPiはフレームのベース・アドレスを指してるフレーム・ポインタである（0<=i<=6、ブロックのレベルである）。SPはフレームのトップ・ポインタである。valは意味解析によって作成し、ノードの拡張ワードに格納しているフレームのサイズである。

・recover_old_frame：次の2つの操作によって、旧フレームを元に戻す。

```
SP  ← FPi
FPi ← MEM[SP]
```

・block_in_flag_set：　新しいブロックに入る時、PC_STACKやV_STACKの最初にプッシュしよう要素にはフラグ1を付けてプッシュする。その後の要素に、全部フラグ0を付けてプッシュする。

・block_out_flag_pop：　脱出の時に、PC_STACKやV_STACKのフラグ1を付いている要素まで、まとめてポップする。

（2）名前の参照

変数のディスプレイスメントなどの名前を参照するために必要な属性は、あらかじめ、意味解析によって生成され、変数宣言文に対応する部分木の葉ノードに格納されている。そして、実行時に、名前を参照する時には、記号表を使わず、直接対応する宣言文の葉ノードまで行って、その中に保存されているこの変数の現フレーム中でのディスプレイスメントを読み込んでくる。さらに、それを用いて、次の2つプリミティブで、名前の物理的アドレスを計算し、参照する。

・MEM[FPi + displacment]　　：変数の参照
・MEM[FPi + displacment + v]：アレイの要素の参照

ここで、アレイの参照にvは配列のインデックス値をもっているものとしている。

（3）データ演算

PL／0を定義されるのに使われているデータの演算のプリミティブは次の11個である。演算は、基本的には、vとスタックのトップのデータとの間で定義されている。

```
・v ← -v
・v ← pop == v
・v ← pop != v
・v ← pop <= v
・v ← pop >= v
・v ← pop <  v
・v ← pop >  v
・v ← pop +  v
・v ← pop -  v
・v ← pop *  v
・v ← pop /  v
```

ここで、vは実行結果の値であり、popはスタックV_STACKをポップして取り出された値である。

（４）スタックの操作

各ノードの意味を実行するための基本的な属性は広域変数として設計した。この広域変数は、実行の流れの中で保存回復を行うことが必要であるが、PATIE-0では、これを自動的には行わず、明示的に指定されたところで行うようにしている。この明示的な広域変数の保存回復のプリミティブとして次のようなものがある。

- push_V_STACK(v)
- push_V_STACK(a)
- push_V_STACK(t)
- t←pop_V_STACK
- a←pop_V_STACK

ここで、push_V_STACK(x)は値xをスタックにプッシュすることを示す。x←pop_V_STACKはスタックのトップに格納されている値をxにポップすることを示す。vは実行結果の値である。aは名前を参照する時にメモリアドレスを計算するのに使用する。tは関数の引数の数を計算するのに使用する。、引数は関数の値であるかもしれないので、この値を保存する必要がある。

４．　ハードウェアの構成

Patie-0のアーキテクチャ設計にあたっては、ユニットレベルのパイプラインを実現することと、各ユニットの機能独立性を高めることを基本的な方針とした。以下に、具体的に設計したハードウェアの構成について示す。

４．１　パイプラインの構成

構文木の各ノードの実行は、ノードのフェッチ、デコード、意味実行という三つの処理に分けられるので、これらの処理に対応して、フローコントロール・ユニット、トラバース・ユニット、データ・ユニットという三つをパイプライン結合して、PATIE-0のハードウェアを構成し（図５）、この三つのユニットは機能のみに独立しており、各ユニット間はメッセージで結合されている。これらのユニットのうち、フローコントロール・ユニットは、単に、実行すべきノードのプリフェッチを行うというだけでなく、ifやwhileといったフローコントロールのプリミティブを実行する機能も持っている。トラバース・ユニットとデータ・ユニットは、１レベルパイプライン型のマイクロプログラムのメカニズムとみなすこともできるが、むしろ、ここでは、トラバース・ユニットは、構文木のノードを意味的にトラバースするための状態マシンであると考える。また、データ・ユニットはメッセージによって、トラバース・ユニットと結ばれているので、物理的なコントロールの面からみると、独立していると考えられる。

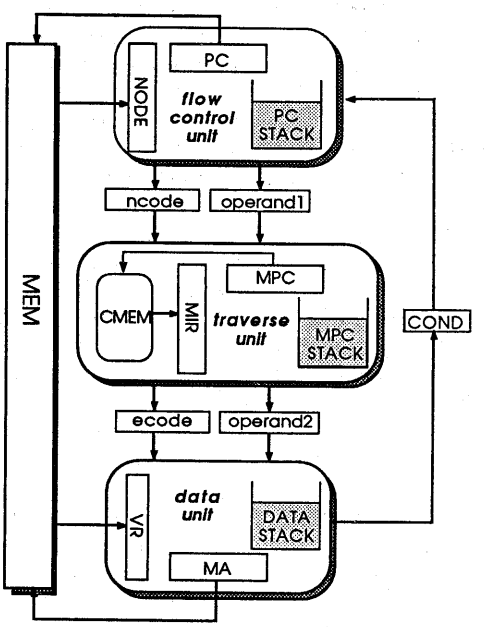

図５　Patie-0のパイプライン構成

４．２　フローコントロール・ユニット

フローコントロール・ユニットは、主にメモリMEMに格納されている構文木をたどりながら、構文木のノードを一つずつフェッチしてくるユニットである。このユニットでは、単にノードのフェッチを行うだけでなく、if文やwhile文などの制御の流れに関する実行も行う。このために、ノードのアドレスに関する情報は、すべてこのユニットの中に管理されている。

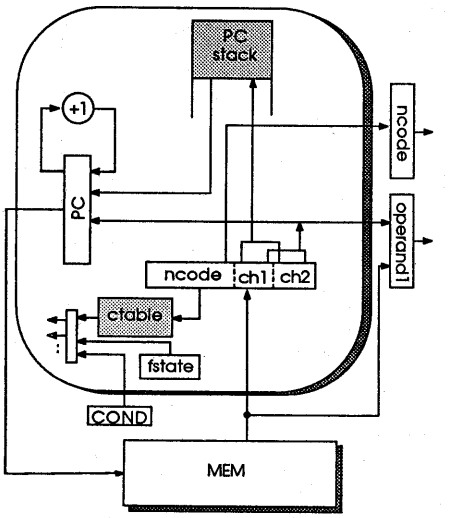

図６　フローコントロール・ユニットの構成

このような機能を実現するために、フローコントロール・ユニットは図６に示すように構成されている。内部には、フェッチすべきノードのアドレスを保持するレジスタPC、フェッチしてきたノードを保存するレジスタnode、構文木をトラバースするのに使用されるスタックPC_STACK、状態制御用レジスタfstateを持っている。このユニットには、構文木を格納しているメモリMEMが接続されており、デコード・ユニットとの間のインタフェースとして使用されるレジスタncodeとoperand1を持っている。

実行がこのユニットの中で完結してしまうノードの時は、次のトラバース・ユニットに何にも送られないが、ノードのトラバースの制御で必要がある場合や意味の実行を伴う場合は、フェッチしたノードの中の必要な情報を次のトラバース・ユニットに送る。この時に、インタフェースレジスタのncodeとoperand1が使われる。

４．３　トラバース・ユニット

各ノードの実行は幾つかのより細かいステップに分けられる。トラバース・ユニットでは、それぞれの１ステップのマイクロ命令を対応させ、この命令の列を制御メモリCMEMより読み出す。データの演算や転送といった意味処理が必要ならば、意味処理に関する情報をインタフェースを通してデータ・ユニットへ渡す。

このような機能を実現するために、トラバース・ユニットは図７に示すように構成した。内部には、実行すべきマイクロ命令のアドレスを保持するレジスタMPC、読み出したマイクロ命令を保存するレジスタMIR、マイクロ命令のシークエンシングに使用されるスタックMPC_STACK、制御用の状態レジスタdstateを持っている。周辺には、マイクロプログラムを格納している専用メモリCMEM、フローコントロール・ユニットとの間のインターフェイスとして使用されるレジスタncodeとoperand1、実行ユニットの間のインターフェイスであるレジスタecodeとoperand2を持っている。

このユニットでは、フローコントロール・ユニットから送られてきたコードを（ncodeに保存している）をデコードして、ノードごとに定義されたマイクロ命令の列を読み出す。構文木の構造の認識とトラバースに関する制御は、このユニットですべて行われる。データの演算や転送といった意味処理が必要な時のみ、実行の指示を行うコードやオペランドなどをインタフェースであるecodeとoperand2を通して送り、データユニットを制御する。

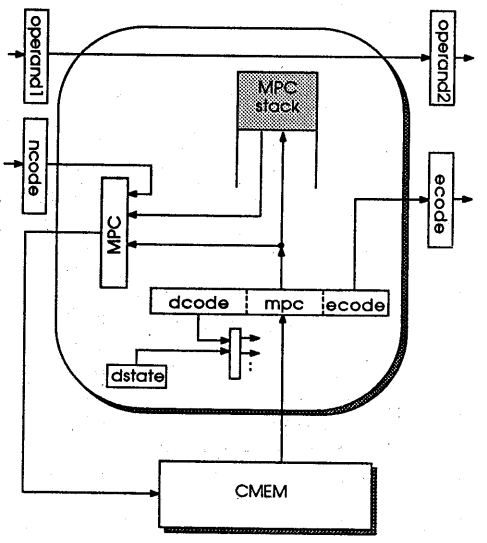

図７　トラバース・ユニットの構成

４．４　データ・ユニット

データ・ユニットは、単にトラバース・ユニットから送ってくれたコードを受け取って、実際にデータの演算や転送といった処理実行を行う。

データ・ユニットは図８に示すように構成する。内部には、データメモリMEMをアクセスするためのメモリアドレスレジスタMA、演算用のスタックdata_stack、レジスタファイル、データやアドレスなどを伝送する用の三本の伝送バス（dst、src1とsrc2）を持っている。レジスタファイルには、データレジスタ（VR）、アドレス計算用レジスタ（AR）、フレームポインタ（FPi）、スタックポインタ（SP）などが置かれる。周辺には、変数やアレイや一時的なデータなどを格納しているデータメモリMEM、トラバース・ユニットとの間のインターフェイスとして使用されているレジスタecodeとoperand2、フローコントロール・ユニットとの間のインタフェースであるレジスタcondを持っている。

データユニットはインターフェイスecodeより、実行するべきコードを取り入れる。それをデコードしながら演算やデータの伝送などを行う。実行中にオペランドが必要な

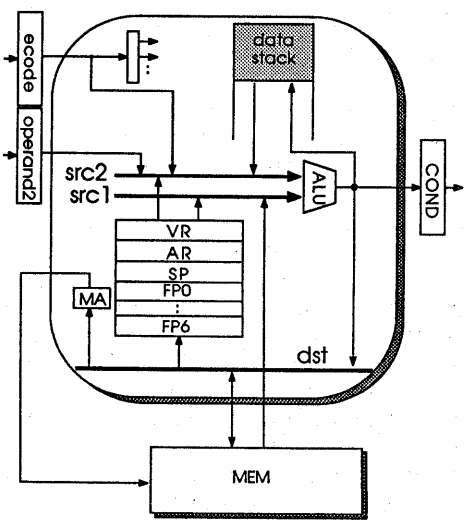

図８　データ・ユニットの構成

らば、インターフェイスレジスタoperand2から取り入れる。ところで、if文とwhile文など分岐を含んでいるノードを実行する時に、フェッチ先行制御のために、条件評価の結果はインターフェイスレジスタcondを通して、フローコントロール・ユニットへ渡される。

5. 制御の方式

本節では、前述のようなパイプライン型のハードウェア構成を持つPATIE-0の制御方式について具体的に説明する。

5.1 フローコントロール・ユニット

フローコントロール・ユニットの基本的な機能は、メモリから読み出されてきたノードのうち、フィールドncode（図1、2）をデコードして、次にフェッチすべきノードのアドレスを決定することである。これを具体的に行うために、図9に示すようにctable (control table) によって、ハードウェアの制御用の情報（fcode、scodeとbcode）を生成する。

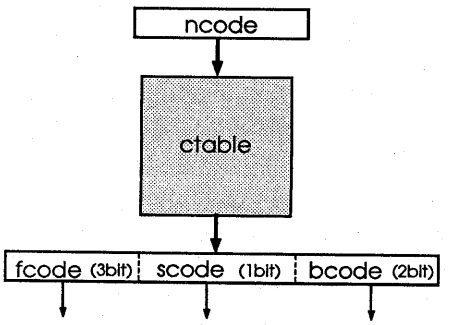

図9 制御用メカニズム

（1） fcodeフィールド
フィールドfcodeは、3ビットのコードであり、制御構造の意味実行に関するその八つのプリミティブ（3.2節参照）に対応する。したがって、フローコントロール・ユニットは、このfcodeをデコードすることによって次のPCを決める。

（2） scodeフィールド
フィールドscode（1ビット）は、ifやwhileなどのようなトラバース・ユニット以後のユニットで実行される必要がないノードを指定するために、存在する。

0： トラバース・ユニットに転送しない。
1： トラバース・ユニットに転送する。

このような判断することで、フローコントロール・ユニットの独立性が高められ、また、全体の実行が最適化されている。

（3） bcodeフィールド
フィールドbcode（2ビット）は、フレーム管理のために用意された。

0： フレームの管理に関係しない一般のノード。
1： 新フレームを作る。
2： 旧フレームを元に戻す。

フローコントロール・ユニットは、このフィールドをチエックすることによって、スタックPC_STACKのフラグに1をプッシュすべきか、あるいは0をプッシュすべきかの値を決める。

5.2 トラバース・ユニット

木のトラバースのアルゴリズムを実現するため、トラバース・ユニットが制御メモリCMEMより読み込んできたマイクロ命令の形式を図10に示す。

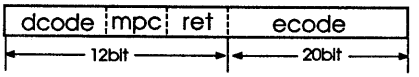

図10 マイクロ命令の形式

このユニットで実行される三つのフィールドを含む前半の20ビットと、データユニットにそのまま送られる12ビットのフィールドの二つの部分からなる。前半の三つのフィールドのうちmpcはCMEM中でのアドレスを指すのに使われ、dcodeとretによってアクションの種類が指定される。

（1） dcodeフィールド
フィールドdcode（2ビット）は、主に、前述のトラバース用のプリミティブに対応しており、以下の様な4種類のマイクロ命令を示す。

seq： ecodeに定義している実行コードを実行ユニットに渡し、フィールドmpcを次に実行するべきマイクロ命令のアドレスとしてMPCにセットする。
call： トラバースのトレースを記録するために、フィールドmpcを次に実行するべきマイクロ命令のアドレスとして、MPC_STACKにプッシュする。
goto： トラバースのトレースを記録する必要がないので、何にもしない。
return： ノードの実行が完了して親ノードへ戻るために、MPC_STACKのトップを次に実行すべき親ノードのマイクロ命令のアドレスとして、MPCにセットする。

（2） retフィールド
フィールドret（2ビット）は、フレーム管理のために使用した。

0： フレームの管理に関係しないマイクロ命令
1： 新フレームを作るためのマイクロ命令
2： 旧フレームを元に戻すためのマイクロ命令

トラバース・ユニットは、このフィールドをチエックする

ことによって、スタックMPC_STACKのフラグに1をプッシュすべきか、あるいは0をプッシュすべきかを決める。

5．3　データ・ユニット

データ・ユニットは、トラバース・ユニットからecodeを一つずつ受けとって、これを実行していく。このecodeの形式は、図11に示すように、二つの形式がある。

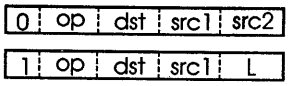

図11　ecodeの形式

演算は、基本的に3オペランド型であり、

dst ← src1 op src2

の形式を持っていて、ecodeの四つのフィールドで、この三つのオペランドと演算の種類を指定する。op、dst、src1とsrc2の四つのフィールド指定される。演算の種類とオペランドの種類を表1に示す。

	dst	src1	op	src2
0	0	0	0	val
1	t	t	==	t
2	v	v	!=	v
3	a	a	<=	a
4	SP	SP	>=	SP
5	FP0	FP0	<	FP0
6	FP1	FP1	>	FP1
7	FP2	FP2	+	FP2
8	FP3	FP3	−	FP3
9	FP4	FP4	*	FP4
10	FP5	FP5	/	FP5
11	FP6	FP6		FP6
12	MEM(ma)	MEM(ma)		L
13	DS	DS		
14	cond			
15	ma			

表1　演算とオペランドの種類

ここで、tからFP6までは、レジスタ・ファイルによって実現することにした。tは引き数の数で、vは値で、aはアドレスで、SPはフレームのトップへのポインタで、FP0・・・FP6はレベル毎のフレーム・ポインタである。MEM[ma]はアドレスmaによりデータメモリへのアクセスである。DSはDATA_STACKに関する操作である（プッシュする、あるいはポップする）。condは条件の評価の結果をフローコントロール・ユニットに送るためのレジスタである。Lは定数である。ここで、s2に関してはaからFP6まですべては必ずしも必要ではないが、s2、dstと対称にすることで、ハードウェアの構造を簡単にしている。

6．　おわりに

PATIE-0のアーキテクチャをC言語によってレジスタ伝送レベルで記述し、Sun3/50の上でシミュレーションを行った。PATIE-0を含むプログラミングシステムはまだ完成していないので、今回は、とりあえず構文木は手で作成した[12]。

今後の課題は、性能の評価を行うことと今回のプロトタイプの設計で得られた指針に基づいて、実用的な言語であるCを対象にしたインタプリタPatie-Cを開発することである。最終的には、インタプリタのハードウェアは、一つだけ作っておいて、ハードウェアの制御のアルゴリズムを言語ごとに入れ換えるようにすることとを考えている。すなわち、言語に依らない部分だけをハードウェア化し、生成系によって、残りの言語に依存する部分を言語の定義から自動的に作り出せるようにすることを考えている。

参考文献

[1]佐藤豊、板野肯三:動的複合実行方式－直接実行系と翻訳実行系を統合した対話型実行方式,コンピュータソフトウェア,2巻,4号,1985,pp.19-29.

[2]板野肯三、佐藤豊、中村敦司:クロススタックキャッシュを用いたブロック構造のためのアドレッシング機構,情報処理学会論文誌,27巻,9号,1986,pp.916-920.

[3]佐藤豊、板野肯三:構造エディタとソースコードインタプリタの統合的記述とその生成系,コンピュータソフトウェア,4巻,2号,1987,pp.135-146.

[4]佐藤豊、板野肯三:構造エディタにおける下降型パーサのための構文木の圧縮法,情報処理学会論文誌,28巻,3号,1987,pp.310-313.

[5]佐藤豊、板野肯三:構造エディタのためのインクリメンタルLLパーサの一構成法,情報処理学会論文誌,28巻,6号,1987,pp.668-672.

[6] 佐藤豊、板野肯三:COSMOS：対話型統合的プログラミング・システム，コンピュータシステムシンポジウム,1985，pp.115-124.

[7]佐藤豊、板野肯三：動的複合実行方式に基づくプログラミング環境,情報処理学会第30回全国大会,5T-7,1985,pp.791-792.

[8]佐藤豊、板野肯三：COSMOSにおける構造エディタおよびソースコード・インタプリタの実現法,情報処理学会第31回全国大会,1F-7,1985,pp.447-448.

[9]佐藤豊、板野肯三:COSMOSの構造エディタ：SED,情報処理学会第32回全国大会,3H-5,1986,pp.653-654.

[10]佐藤豊、板野肯三：COSMOSプログラミングシステムの構文指向画面エディタ：SSE、構造エディタに関するワークショップ（ソフトウェア科学会），1986，pp.1-15.

[11]佐藤豊、板野肯三：C言語指向構造エディタSSE,共立出版,1987,pp.59-70.

[12] Kozo Itano and Xiaowei Kan: "PATIE: A Hardware Parse Tree Interpreater",Technical Note HLLA-E-18,筑波大学電子・情報工学系,1988.