計算機アーキテクチャ　63－3
（1986・11・20）

プログラム可能論理演算装置を用いた計算機の制御方法について

A Control Method of a Computer Using Programmable Logic Unit

入野祥明[+]　吉岡良雄[++]　中村維男[+]　重井芳治[+]

Yoshiaki IRINO　Yoshio YOSHIOKA　Tadao NAKAMURA　Yoshiharu SHIGEI

[+] 東北大学工学部　[++] 岩手大学工学部

Tohoku Univ.　Iwate Univ.

1．はじめに

従来のノイマン型計算機は、プログラムがＣＰＵの命令列の形で主記憶装置内に格納され、プログラムカウンタによる実行制御の下に各々の命令をＣＰＵに取り込み実行している。即ち、ＣＰＵ内の演算器を時分割的に各々の命令に割り当てて演算を実行している。このため、本来の目的である演算以外の操作、つまりデータ移動やアドレス計算が処理時間の大半を占めてしまい、バスボトルネック等の問題の要因となっている。

これに対しハードウェアマッピング型演算方式は処理する演算を全てハードウェア的に用意する方式である。つまり必要とする演算回路をあらかじめ構築しておき、プログラム実行時には、被演算データのみをその演算回路に与えるだけで、必要な演算結果を得る方法である。この演算方式は様々な専用処理装置(ＦＦＴ等)として実現されており、処理速度の点で非常に有利である。しかしその反面、これらの専用処理装置は専門分野以外の処理に対しては充分な能力を発揮しない場合が多く、その意味で非常に"高価"な装置であると言える。そこで、ハードウェアマッピング型演算方式を実現し、かつソフトウェアによる汎用性も備えた演算装置として、プログラム可能論理演算装置(ＰＬＵ：Programmable Logic Unit)が提案されている[1]。このＰＬＵは多数の演算器をメモリ内部に分散配置し、従来記憶機能しかなかったメモリ装置に演算機能を加えたものである。更にＰＬＵ内部の演算器は、その演算機能を外部から定義することが出来る。従ってＰＬＵの演算機能は可変であり、従来のノイマン型計算機と同様にソフトウェアによって汎用性を実現する。又、ＰＬＵ内部に定義された演算回路内では、データの流れだけで演算が進行するので、高速な演算処理が可能である。

本稿では、このようなハードウェアマッピング型演算を実現する一方式であるＰＬＵの構成について検討し、さらに、ハードウェアおよび言語処理系などのソフトウェアの両面から、ＰＬＵを構成資源として用いた計算機の有効性について検討する。

2．プログラム可能論理演算装置

ＰＬＵは"演算機能を持つ記憶装置"であり、その基本要素(ＰＥ：Processing Element)は、数語の記憶セル、演算器、データ転送の為の相互結合ネットワークおよび外部とのインタフェース機構から構成される。本章ではＰＥの構造とＰＥ間の結合形態について考察する。

2．1　ＰＥの構造

ＰＬＵは"演算機能を持つ記憶装置"であり、充分な数の演算器を記憶セルの近傍に配置したものである。従来のノイマン型計算機は、記憶セル内に格納されたデータをパスを介して中央処理装置(ＣＰＵ)まで移動させてから、実際の演算操作を加えていた。これに対し、ＰＬＵは記憶セルの近傍に直接演算器を用意しておくことで、本来の演算操作とは無関係なデータ転送を減少させ、かつ多数の演算器を同時に動作させることで演算レベルでの並列性を与え、処理の高速化を図ろうとするものである。

ＰＬＵの基本要素ＰＥは数語の記憶セルと数個の演算器から成る。高速化のため、演算器は内部にラッチを含まず、ＡＬＵ程度の演算機能を持つ。更にＰＥ内の記憶セルの一つは演算定義用のセル(ＩＲ：Instruction Register)であり、外部からこのＩＲに演算定義命令を格納することで、ＰＥの演算機能が設定される。もう一つは、演算データ格納用のセル(ＤＲ：Data Register)である。ＰＥの具体的構造は様々なものが考えられるが、図１に我々が試作したＰＥの構造を示す。

このようなＰＬＵと同様に、従来の記憶装置に演算機能を持たせ、処理を細分化することで高速化を図るアーキテクチャとして、コネクションマシン[2]が提案されている。コネクションマシンの基本要素は数Ｋビットのメモリと簡単な演算器を持つプロセサから成る。メモリ内データのアドレスは外部の制御装置から与えられ、各プロセサは自身のメモリから読みだしたデータに対して演算を施し、再びメモリに格納する。プロセサが実行する命令列も制御装置から与えられ、数万台のプロセサが全て同じ命令を実行する。従ってコネクションマシンでは、メモリ内に分散された多量のデータに対して、マシン全体で同期した一括操作を加えることにより、大規模な並列性を得ていると言える。これに対して、ＰＬＵは各ＰＥが非同期に動作することで演算の並列性を引き出している。ＰＬＵでは各ＰＥごとに独立した機能定義部を持つので、多数の異なる命令群を並列に実行可能である。更に複数のＰＥをハードウェア的に接続することで、より高度な演算回路をＰＬＵ内部に構築することが出来る。ＰＬＵは同期用クロックを持たないので、各ＰＥの演算器およびいくつかのＰＥが結合された演算回路(テンプレート)は全て、並列かつ非同期に動作可能であり、ＰＬＵ内部では常に、大規模な並列演算が実行されている。

２．２　相互結合ネットワーク

コネクションマシンでは16台の処理要素が４＊４の格子状に結合され、それらがルータを介してキューブ結合されている。プロセサ間の情報交換はパケット転送で行なわれ、パケットの転送制御はルータが担当する。従って、各処理要素は他の任意の処理要素と通信可能であり、ソフトウェア的に完全結合されていると言える。この方法は柔軟性に秀れ、必要とする任意の相互結合を実現できるという特徴を持つが、パケットの生成や転送のオーバヘッドはデータ転送の高速化に反するものである。

ＰＬＵは高速演算処理を目的とする装置であり、被演算データの転送、移動のオーバヘッドを最少限に抑えなければならないため、ＰＥ間を直接ハードウェア的に結合している。任意の結合を実現する為には、全ＰＥを完全結合しなければならないが現実的ではない。ＰＬＵの相互結合ネットワークに必要な条件として、

1)　高速性

ＰＥの演算機能に対して充分なバンド幅をもつこと

2)　結合数

任意のＰＥ間に、常に十分な数の転送路が確保できること

3)　簡単な構造

ＰＥはプロセサではないので、パケット生成やメッセージ転送、経路制御等複雑な制御を行なう能力は無い。

4)　再構成可能性

ＰＬＵ内の任意の場所に任意の演算回路を構築できること。

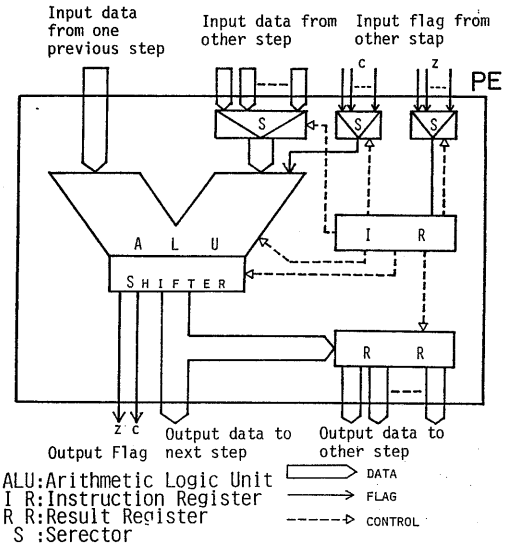


図１　試作ＰＥの構造

5)　構造の一様性

最終的にはVLSIによる実現を目指している。

等が挙げられる。現在ＰＥの結合形態として、直結型、分割バス結合型[3]および時間分割バス結合型[4]が提案されている。

直結型ＰＬＵでは、直前の演算結果を参照することが最も多い、と考えられることから、各ＰＥの演算器入力の一方を前段のＰＥの演算出力と直結し、もう一方の入力はセレクタを介して数段前のＰＥと結合している。従って演算の途中結果をすみやかに次段に転送できることから、数式の演算処理に有利である。

分割バス結合型ＰＬＵはＰＬＵ内に複数のバスを用意し、各ＰＥはセレクタを介してバスから被演算データを入力し、演算結果を再びバスに出力する形態である。これは複数のＰＥで演算データを共有できることや、バス長の許す限り任意のＰＥ間でデータ転送可能であり、非常に柔軟性に富むことが特徴である。

時間分割バス結合型ＰＬＵはＰＥ間のデータ転送に同期クロックを併用し、ＰＥへのデータ入出力を同期させることで、ＰＬＵ内でパイプライン的処理を行ない、システムのスループットを向上させる方式である。

各結合形態とも一長一短があり、必要とする処理内容に応じて最適な結合を選ばなければならない。直結型はセレクタを介して、最大何段前のＰＥと結合出来るかという、結合数が重要な問題である。結合数が充分でない場合、外部の制御装置がソフトウェアでＰＥ間を結合する操作が必要である。同様に分割バス結合型ではバス本数が、さらに時間分割バス結合型では同期クロック周期も、ＰＬＵの性能に大きく影響する。

２．３　格子結合型ＰＬＵの提案

直結型ＰＬＵは、バス長とバス本数に制限がある分割バス結合型と考えることも出来る。ここではＰＥ間の結合にさらに特殊性を与えた格子結合型ＰＬＵを提案する。

格子結合型ＰＬＵは図２に示すように、ＰＥを格子状に配置しＰＥ間を単方向二重バスで結合した形態である。各ＰＥは常に４近傍ＰＥから被演算データを受け取り、演算結果を常に４近傍ＰＥに出力している。ＰＥの構造を図３に示す。ＤＲおよびＩＲとなる記憶セルには他型式のＰＬＵと同様に、一連のアドレスが割り当てられており、外部の制御装置はＰＬＵにアドレスを入力すれば、任意の時点で任意のＰＥのＤＲおよびＩＲの内容を参照することが出来る。ＰＥ内の演算器は二入力に限定されず、四入力まで拡張可能である。また、ＰＥ間の結合が４近傍に限られるため、ＰＥの構造は他型式のＰＬＵより単純で済む。更にＰＥ単位の規則性が高く、VLSIチップへのインプリメントも容易である。

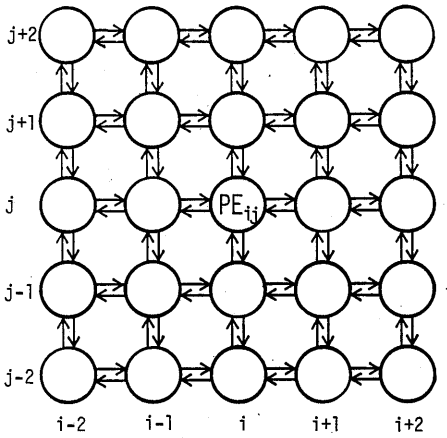


図２　格子結合型ＰＬＵ

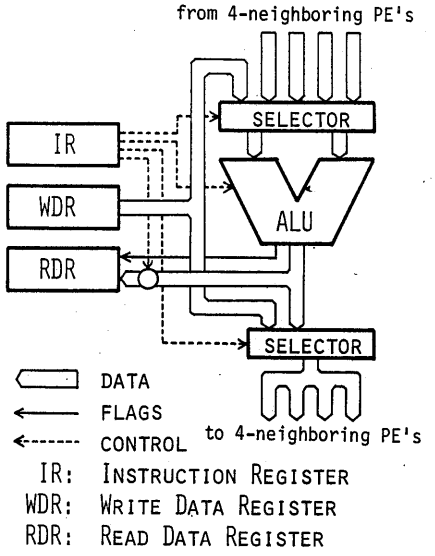


図３　ＰＥの構造

２．４　各ＰＬＵの特徴

直結型およびバス結合型ＰＬＵを用いた演算処理に関して、次のような特徴が挙げられる。

A)　ゲート遅延時間レベルの高速演算

複数のＰＥを接続して構築(マッピング)された演算回路は内部にラッチ等を含まないので、演算は全て"垂れ流し的"データフローにより、ゲート遅延時間レベルで実行される。直結型は途中結果を直接次段の演算器に渡せるため、各ＰＥごとにバスを分割するためのゲートを通る必要がある分割バス結合型と比べて、演算遅延時間が少なく高速演算に有利である。格子結合型はＰＥ間の結合距離が短かく、更に直接結合されているＰＥが４個ある。従って直結型と比較してさらに高速演算の可能性がある。

B)　逆ポーランド記法に従った数式の実行

一般に任意の数式は２進木で表現可能である。２進木で表現された数式を一次元のＰＬＵ上にマッピングする方法は、２進木をポストオーダで辿り、その順序でＰＥに演算を割り当てることで実現できる。特に分割バス結合型の場合、２進木の途中結果をバスへ割り当てる操作は、スタックマシンにおいて演算実行中に動的に変化するスタックの内容を、静的にＰＬＵ内に展開することとみなせる。格子結合型は二次元の結合を持つが、トポロジー的に直結型と同様に考えることも出来る。従って比較的単純な操作で、２進木を直接マッピング可能である。さらに一次元の結合に比べてマッピングの自由度が大きく、マッピング方法が一通りに限られないという利点がある。

C)　数式の展開による高速処理

ＰＬＵ内部の演算回路は、その演算が有効であるなしにかかわらず常に演算実行状態にあるため、被演算データが確定したＰＥから順に"有効な"演算が開始される。このことはプリミティブな演算レベルでの並列性を持つことを示すが、一般にＰＥでの演算遅延に比べ、被演算データの確定が遅いため、数式内の並列性を自然に引き出すためには個々の演算器の機能を上げるか、またはより長い数式をテンプレートとして展開する必要がある。

マッピングする数式が完全２進木であると仮定し、直接マッピング可能な最大の数式を木の深さｈで表わせば、

$$h = \begin{cases} \lceil \log_2(L-1) \rceil + 1 & \text{直結型} \\ B & \text{分割バス結合型} \\ 4 & \text{格子結合型} \end{cases}$$

となり、同じハードウェア量ならば分割バス結合型ＰＬＵが最も大きな数式を展開出来る。ただしＬは直結型の接続段数、Ｂは分割バス結合のバス本数である。

例としてN＊Nの配列加減算および、N次ベクトルの内積計算を行なう場合に最低限必要な、ＰＥ数、結合数(バス数)、結合距離および、演算データの最大移動距離と総移動量を表１に示す。表１から格子結合型が最も効率良く、これらの演算を実行出来ることがわかる。

ＰＬＵは以上のような特徴をもつが、このようなＰＬＵを用いて主にどのような演算処理を行なうか、また現在の技術でどの程度のハードウェアを用意できるかによって、最適な形式のＰＬＵを選択しなければならない。処理時間を短縮することに重点を置いた場合は、内部のデータ転送時間が短かい直結型が有利であるが、数式だけではない様々な処理への応用を考える場合には、結合の自由度が大きい分割バス結合型が有利である。格子結合型は結合の局所性を生かせば、ベクトル演算や配列演算等に有効である。更にＰＥ内の演算器を複数にし、多入力演算を行なうようにすれば、二次元データを扱う応用で、近傍のデータ参照のみで実行可能な演算処理(画像処理の分野でエッジの検出や重みづけ、画像の移動、比較、重ね合せ等)もマッピング可能である。

(a) Ｎ＊Ｎ配列加減算の場合

	直結型	バス型	格子型
ＰＥ数	$3N^2$	$3N^2$	$2N^2$
結合数	1	2	4
結合距離	2	2	1
最大移動距離	2	2	1
総移動量	$3N^2$	$3N^2$	N^2

(b) Ｎ次ベクトルの内積計算の場合

	直結型	バス型	格子型
ＰＥ数	4N-1	4N-1	3N
結合数	4	3	4
結合距離	4	4	1
最大移動距離	4	4	1
総移動量	8N-5	8N-5	3N-1

表１　ＰＥ結合形態の比較

３．　ＰＬＵを用いた計算機の制御方式

一般に、演算プログラムは"被演算データ(DATA)"と本来の目的である"演算操作(OPERATION)"および、演算操作の流れを変えるための"演算制御操作(FLOW CONTROL)"から成っている。従来の計算機が"演算操作"と"演算制御操作"を、共にＣＰＵ用コードの形で"被演算データ"と同様に記憶装置に格納し、実行時にＣＰＵがそれらを逐次読み出して実行していたのに比べ、ＰＬＵを用いた計算機では、"演算操作"がテンプレートの形でＰＬＵ内に格納され、"被演算データ"がそのテンプレート内を伝搬していくことで演算が実行される。従ってＰＬＵの制御装置は"演算制御操作"、つまり計算のフロー制御に専念することが出来る。本章ではＰＬＵの制御方式と演算のマッピングについて述べ、ＰＬＵを用いた計算機の仮想モデルを与える。その後、仮想計算機の試作言語処理系の概要を述べる。

３．１　制御方式

ＰＬＵは常に外部と非同期に、更にＰＬＵ内部では全ＰＥが常に非同期に動作しているので、演算自体の制御は必要としない。従ってＰＬＵの制御とは専ら、ＰＬＵと外部とのデータ入出力操作を意味する。ＰＬＵのインタフェース機構は、各ＰＥのＩＲおよびＤＲのみであり、外部からのアドレスによって、内部の演算とは無関係に任意の時点でこれらの内容を参照できる。従って外部から見た場合、ＰＬＵは従来の記憶装置と同様に扱えるので、制御装置はアドレス生成機能とデータ移動機能を備えている必要がある。

一般に、"演算制御操作"は計算機内の状態の変化を伴うので、自分で自身の状態を変えることができる"能動的"な構成資源がこれを実行する。条件判断やループに伴う演算もＰＬＵが実行するが、ＰＬＵは能動的な資源ではないので、実際の判断操作は制御装置が実行しなければならない。以上から、ＰＬＵの外部制御装置として、従来のノイマン型計算機と同様の逐次型制御装置(MCPU：Main CPU)を用いる。逐次型制御を採用することにより、従来の計算機との互換性や、システムの実現性が高くなるであろう。

逐次制御に基づくＰＬＵを用いた計算機の構成を図４に示す。ＰＬＵへのデータ入出力はＭＭ－ＰＬＵ間のデータ転送の形で行なわれる。このデータ転送操作および演算制御操作は、MCPU用コードの形でＭＭ内に格納され、MCPUが逐次実行する。このようなMCPUの構造を図５に示す。このMCPUは演算器および演算用レジスタを持たず、命令レジスタ、プログラムカウンタ、アドレスレジスタ、スタックポインタを持つ。フラグレジスタはＰＥのフラグの参照時や、ＰＥ間のデータ転送時に使用する。

このようなMCPUの基本命令の例として、今回の仮想計算機では以下の７命令を定義している。

1)　データ転送命令

```
MOV    (address),(address)
```

2)　演算制御命令

```
Jcc    (address),address      cc=Z,NZ,S,NS
JMP    address
```

3)　状態制御命令

```
CALL   address
RET
WAIT
HLT
```

Jcc命令では、MCPUが第一オペランドで指定されたＰＥのＤＲの内容をフラグレジスタにコピーし、データのフラグ部の内容により、条件判定を行なう。WAIT命令を実行すると、MCPUはあらかじめ設定されたクロック数だけ待ち状態になる。MCPUがＰＬＵの演算結果の確定を待つ必要がある場合に、このWAIT命令を用いる[5]。

MCPUがアドレスを用いてデータを参照することから、バスボトルネックの問題は依然として残されている。ＰＬＵを含めた記憶装置は"能動的"でなく、被演算データは変更可能な形で格納せざるを得ない以上、この問題は避けられないが、ＭＭ－ＰＬＵ間とMCPU－ＰＬＵ間には演算データのみが流れることや、MCPUの構造と機能が単純で演算命令も持たないので、MCPU用コードもコンパクトになることから、バスの負荷が分散されると思われる。

簡単な実行例としてＮ個の演算子を持つ数式の実行を考える。演算に必要なＮ＋１個の被演算データはＭＭに格納されているものとし、ＰＬＵ内にはプログラムのロード時に、必要な演算回路が構築済みであると仮定する。ＰＬＵを用いた計算機の場合、処理に必要な時間はＮ＋１個の被演算データをＭＭからＰＬＵへ転送する命令の(フェッチ＋実行)時間と演算結果を転送する命令の(フェッチ＋実行)時間およびＰＬＵでの演算遅延時間である。ＭＭのアクセス時間をA_{MM}、ＰＬＵのアクセス時間をA_{PLU}とすると、計算時間T_{PLU}は

$$T_{PLU}=(N+1)(3A_{MM}+A_{MM}+A_{PLU})+(3A_{MM}+A_{PLU}+A_{MM})+OPt_{OP}$$

となる。ただし、OPは最後の入力データに対する演算子の数であり、最大で$\lceil \log_2(N+1) \rceil$個、最小で１個である。また、$t_{OP}$はＰＥ一段での演算遅延時間である。

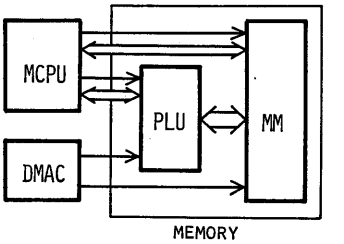
次に比較のため、ＣＰＵ内に十分な数の演算用レジスタと一個の演算器を持つ計算機で実行する場合を考える。レジスタ間の演算命令はＯアドレス、メモリーレジスタ間の転送命令は１アドレス形式であると仮定すれば次の関係が成立する。

$$T_{REG}=(N+1)(2A_{MM}+A_{MM})+(2A_{MM}+A_{MM})+N(A_{MM}+t_{OPR})$$

ただし演算時間をt_{OPR}とし、命令のデコード時間はアドレス情報のフェッチに隠れるものとする。以上二つの式で

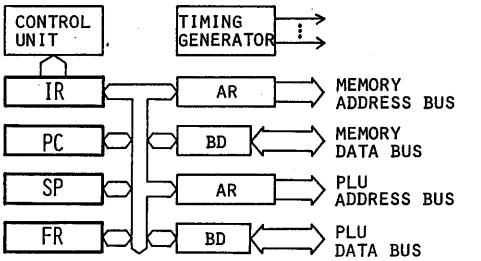
$$t_{OP}=t_{OPR}=t,\ A_{PLU}=A_{MM}=A$$

とした場合の、T_{REG}に対するT_{PLU}のスピードアップ率を$r=t/A$の関数として図６に示す。図６から、データ転送時間(アクセス時間)に対して、演算遅延時間が長く、さらに処理する数式が長くなる程、ＰＬＵの高速性が発揮されることがわかる。ただし一般に、加減算や論理演算等のプリミティブな演算では$r=0.5$程度であるため、ＰＬＵの高速性を生かすためには、演算器の機能を上げることや、データ転送を高速に行なうことが必要である。特に大量のデータをＰＬＵに転送する必要がある場合には、MCPUを介した逐次転送では充分なバンド幅が得られないので、DMACによるブロック転送やＭＭのインタリーブ等の手法により、高い入出力バンド幅を確保する必要がある。



MCPU:Main Central Processing Unit
DMAC:Direct Memory Access Controller
PLU :Programmable Logic Unit
MM :Main Memory

図４　ＰＬＵを用いた計算機



IR:Instruction Register
PC:Program Counter
SP:Stack Pointer
FR:Flag Register
AR:ADDRESS REGISTER
BD:BUS DRIVER

図５　MCPUの構造

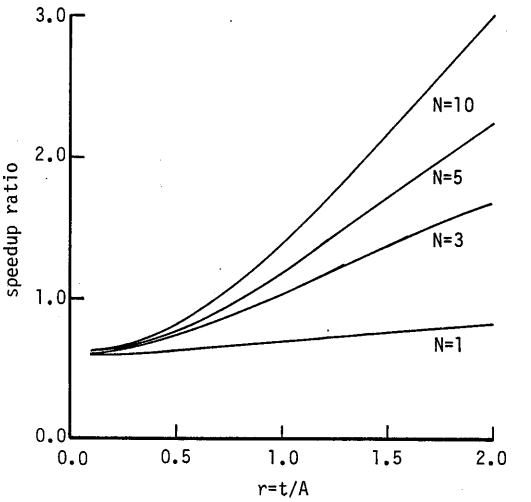


図６　時分割型演算とハードウェアマッピング型演算の比較

３．２　演算のマッピング

直結型ＰＬＵおよび分割バス結合型ＰＬＵへの数式のマッピングは、逆ポーランド記法に従った式への変換を基本としていた。格子結合型ＰＬＵも同様に、直接逆ポーランド式をマッピング可能であるが、直結型と分割バス型が２進木のノードをリーフから辿りながらマッピングするのに対し、格子型では２進木をルートからマッピングして行く。これは、変数を定義してから演算位置を決定するか、演算位置を決めてからその周辺に使用する変数を定義するかの違いである。ＰＥ結合の自由度が大きい場合には前者が有利であるが、結合に制約が多い時には後者の方法を選択せざるを得ない。格子結合型の場合、変数が定義されるＰＥのアドレスと演算が定義されるＰＥのアドレスに規則性が無くなってしまうが、コンパイル時にテンプレートのアドレスを静的に決定してしまうので、演算実行には影響しない。

次に、２進木で表現された式を格子結合型ＰＬＵへマッピングするアルゴリズムを表３に示す。このアルゴリズムではルートノードを持つＰＥから、最も遠い隣接ＰＥに子ノードを割り付ける。格子結合型ＰＬＵにマッピング可能な、最大の完全２進木はｈ＝４であるから、ｈ＝４以上の完全２進木を部分木に持つ式をマッピングする場合は、MCPUが介入して演算の途中結果をＰＥからＰＥへ転送する必要がある。このように演算の途中で外部とのデータ入出力が行なわれると、その時点で同期がとられ、演算の流れが一時中断してしまうのでＰＬＵの高速性が生かされない。従ってそのような演算が多い応用に対しては、より長い数式をマッピング可能な分割バス結合型を用いるべきである。

３．３　言語処理系

前述の仮想計算機に対して今回試作した、言語処理系のターゲット言語にはＣ言語を選んだ。Ｃ言語は比較的単純な構造を持ち、処理系の作成が容易である。基本的には逆ポーランド式で表現可能な記述は、全てテンプレートとしてＰＬＵ内にマッピングされる。

Ｃ言語のプログラムの構造単位は関数である。この処理系は通常、一つの関数記述に対して、一つのテンプレートを構築する。ＰＬＵには関数呼出しの機能はなく、関数は実行時にMCPUのCALL-RETURN 命令により生成、消滅するが、ＰＬＵ内のテンプレートは常に関数を実行している。テンプレート内の仮引数は、MCPUが引数データを転送した時点で実引数となり、有効な演算が開始される。従って関数間の同期はMCPUのデータ転送に従い、同一プログラム内で並列な関数呼出しは行なわない。更にループの制御変数や条件分岐に伴う演算、および配列要素のアドレス計算等も全てＰＬＵが実行する。ＰＬＵをループ状に結合すると、ＰＥでの遅延時間により発振を起こすので、プログラム中のループやi＝i+1等の記述は、MCPUのデータ転送命令により実現されるため、処理速度低下の要因となる。

このように条件判断等で、MCPUがＰＬＵ内の演算結果やＰＥのフラグを参照する時点では、そのテンプレートが定常状態でなければならない。このような場合のMCPUとＰＬＵとの同期はソフトウェアによる。すなわちコンパイル時に、必要と思われる箇所にWAIT命令を挿入し、MCPUは充分長い時間待ち状態になった後、次の処理に移る方法である。

この言語処理系でいくつかの応用プログラムをコンパイルした場合の、MCPUコードサイズと命令数を表２に示す。比較に用いたのは、MC68000MPU(以下68K)用のコンパイル出力である。68Ｋの場合アドレス計算には内部レジスタを、数値計算および関数呼出しにはスタックを主に用いている。MCPUコードの設計にもよるが、ＰＬＵの場合オペランドを全てアドレスで指定するために、１命令のコードサイズが汎用レジスタ方式より大きくなり、全体では68Ｋのオブジェクトと比較して、約1.5倍の大きさになっている。しかし命令数で見ればほぼ同じ量であり、さらに定数を最初からＰＬＵ内に定義しておく方法をとれば、定数代入のデータ移動命令を省くことが出来る。

現在の仮想計算機ではＰＬＵの制御装置としてMCPUを用い、逐次制御を行なっているため、言語処理系の設計が容易であり、最適化等のソフトウェア技法も適用可能である。しかしその反面、演算制御が多いプログラムの場合はMCPUの動作で演算の同期が取られてしまい、演算の並列性が制限され、ＰＬＵの高速性が生かされない。従ってＰＬＵを有効に利用できるプログラムとは、プログラム全体の実行時間に占める制御時間の割合が少なく、テンプレートを大きく取れるようなプログラムである。

プログラム	階乗関数		フィボナッチ		ベクトル内積		配列加算		配列乗算		多項式計算	
	PLU	68K	PLU	68K	PLU	68K	PLU	68K	PLU	68K	PLU	68K
コードサイズ(byte)	144	100	224	138	136	82	256	132	422	244	242	332
データ移動命令数	9	11	15	16	11	8	19	15	33	23	22	33
演算制御命令数	5	2	8	7	6	2	12	4	21	8	3	2
状態制御命令数	4	6	6	6	1	3	1	3	1	3	1	3
演算命令数	---	5	---	6	---	11	---	17	---	32	---	51

表２　コンパイル結果の比較　（ベクトルは16次、配列は16＊16、多項式計算は表４参照）

4．　多重処理

ＰＬＵの高速性を有効に利用するためには、ＰＬＵと外部とのデータ入出力を極力少なくし、演算制御が少なく、MCPUが介入しなくても演算処理が進むようなテンプレートをＰＬＵ内に構築する必要がある。

このようなプログラムの例として、多項式計算のプログラムを表４に示す。このプログラムではfor文の制御ループ中のWAIT命令により、MCPUの制御操作とＰＬＵでの演算操作との同期が取られている。WAIT命令によるMCPUの同期待ち時間は、ＰＬＵ内のテンプレートが大きくなる程長くなるので、高速処理可能なプログラムを実行する場合は、MCPUの稼働率が低いことになる。そこで、MCPUの待ち時間を他の計算に割当てることでMCPUの稼働率を上げ、システム全体のスループットを向上させる方法として、多重化の手法が提案されている[6]。本章では、仮想計算機のシミュレータでのプログラム実行結果から、多重化の効果について検討する。

4．1　シミュレーション技法とモデル

ＰＬＵを用いた計算機システムでは、MCPUがMMから命令列を取り出し、解釈、実行している間、ＰＬＵ内では演算データが有効であるか否かにかかわらず、常に外部と無関係に演算実行状態にある。基本的にMCPUとＰＬＵは非同期独立であり、さらに各ＰＥも非同期に演算を実行している。このように複雑な事象を効率良く解析するために、ハードウェアシミュレータを試作した。これは三台のプロセサを、三台の共通メモリでリング状に接続したもので、一台のプロセサがMCPUコードを実行し、他の二台のプロセサは、MCPUとの共通メモリ上にＰＬＵイメージを作り出す、一種のバックグランドプロセサとして動作する。従ってMCPU側から見た時は、共通メモリがＰＬＵ(シミュレータ)として動作している。ＰＬＵシミュレータの演算特性は、ＰＬＵ試作機の特性を参考に設定しているが、試作機が全て市販のＴＴＬで作られていることから、最終目標であるVLSIチップのＰＬＵが実現された時にはより高速になると考えられる。現在のシミュレータは試作機に比べて10^3～10^4程度遅いため、シミュレーションクロックを設定して基準時間を定めている。

仮想計算機の性能を大きく左右する要因の一つは、記憶装置の速度(アクセス時間)である。シミュレーションではMMとＰＬＵのアクセス時間を等しく設定し、主な実行パラメータとして、MMアクセス時間を選んだ。MCPUの命令デコード時間はフェッチ時間に含まれるものとし、MCPUの動作速度はアクセス時間に比例すると仮定する。タスク切換処理はWAIT命令フェッチ後に開始される。MCPUは内部のＰＣとＳＰを、MMのあらかじめ定められた領域に待避し、次の領域からＰＣとＳＰの値を復帰させる。この一連の動作は全てマイクロプログラム化され、タスク切換に必要なMMアクセス回数は(保存レジスタ数＋１)＊２回であるとする。

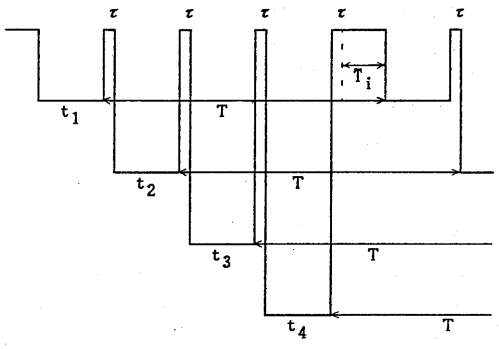


図７　ＰＬＵの多重処理

図７に示すように、MCPUの処理時間をｔ、その確率分布をｆ(t)、MCPUがＰＬＵに演算データを格納してから、結果データを取り出すまでの時間をＴ、タスク切り換えのオーバヘッド時間をτとする。Ｔはシングルタスクの場合の、WAIT命令による待ち時間に相当する。このときMCPUの待ち時間の平均T_iは、多重度をＮとして次式で表わされる。

$$T_i = \int_0^{T-N\cdot\tau} (T-N\cdot\tau-t)\cdot f_{N-1}(t)\,dt \qquad (1)$$

ただし、ｆ(t)は各タスクにおいて全て等しいと仮定し、$f_N(t)$はｆ(t)のＮ回の畳み込みとする。ｆ(t)を平均$1/\lambda$の指数分布と仮定する。MCPUでの処理時間はアクセス回数に比例するので、ｎを一回の処理当りのMCPUのアクセス回数とすれば、

$$1/\lambda = nA$$

また、タスク切り換え時の内部保存レジスタ数をＲとすれば、タスク環境の退避および復帰のオーバヘッド時間は

$$\tau = 2*(R+1)A$$

となる。そこでアクセス時間に無関係な値として

$$1/\lambda = k\tau \qquad k：プログラムに依存する比例定数$$

を単位時間に選び、待ち時間Ｔを$1/\lambda$との比で表わす。

４．２　シミュレーション結果

表４の多項式計算プログラムを、シミュレータにタスクとして投入し、アクセス時間およびシステム内のタスク数を変化させて、MCPUのタスク処理時間、待ち時間およびタスクのターンアラウンド時間を測定した。表４のプログラムの場合、MCPUの制御ループ内でのアクセスは16回、保存レジスタは２個であるから、

$$k = n/2(R+1) = 16/6 = 2.67$$

である。タスクのスケジューリングは静的に行ない、実行に先立って、必要なテンプレートとプログラムは、ＰＬＵおよびMM内に格納しておく。１WAIT命令当りのMCPU待ち時間は最小になるよう調整し、ソフトウェア同期を用いた場合の最大性能を求めている。

図８に１WAIT命令当りのMCPU待ち時間を示す。多重度を上げることにより、見かけ上の演算遅延時間が短かくなり、MCPUの待ち時間が短かくて済むことがわかる。図９に示すように、MCPUの待ち時間を1/λで正規化したものと、式(1)の結果とを比較すると、式(1)の結果よりも実際の待ち時間は少ない。これは仮想計算機では、各WAIT命令毎にタスク切り換えが起動され、更に(必要な場合に)待ち状態に移行するので、WAIT命令の実行時間がタスク切り換えのオーバヘッドに加わるためである。図10に全タスクの実行を終了するまでの、MCPU稼働率の平均値を示す。MCPU稼働率の上限は、タスク切り換えのオーバヘッドτで抑えられてしまうが、τが小さい(従って待ち時間Ｔが大きい)場合には、より多くのタスクをシステムに投入することが可能である。表４のプログラムで、式(1)が成り立つのは制御ループの部分であり、定数代入のデータ転送命令実行時は、演算の並列性が生じないのでMCPUの逐次処理だけが進む。MCPUの稼働率が式(1)の結果よりも高いのは、実際のプログラムには逐次的にしか処理出来ない部分がある為である。従って逐次的な処理が多いプログラムに対しては、多重化処理は有効とは言えない。図11にシステムのスループットを示す。待ち時間Ｔはプログラムに依存するが、Ｔが大きい場合には多重度を上げることにより、大幅な性能向上が期待できる。

ＰＬＵ内部のテンプレートが、常に演算実行状態にあることや、MCPUの内部レジスタが少なく環境の退避、復帰が簡単なことから、ＰＬＵを用いた計算機は比較的容易に多重処理を実現できる。仮想計算機では逐次制御を用いているために、プログラム内の関数ごとの並列呼出しは出来ないが、互いに呼出し関係の無いプログラム単位ならば、多重処理により並列に実行可能である。従って逐次処理が少なくMCPUの待ち時間が長いプログラムを実行する場合には、多重化によりシステムのスループットを大幅に向上させることが出来る。

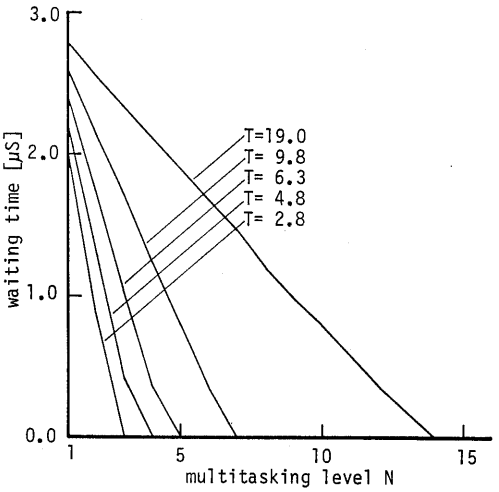


図８　多重度Ｎに対する１WAIT命令の待ち時間

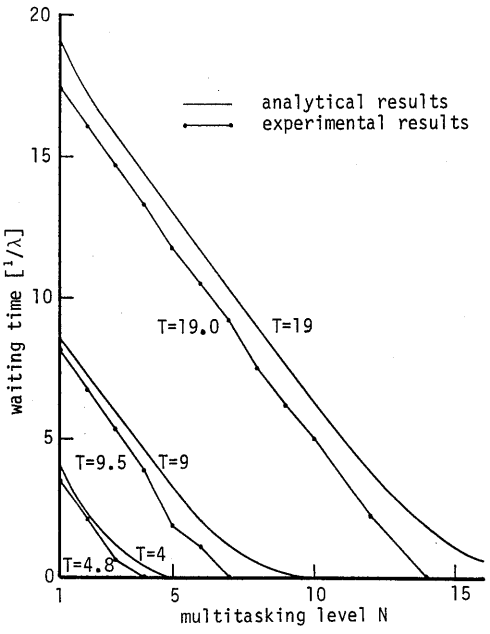


図９　多重度Ｎに対するMCPUの待ち時間

5. むすび

プログラム実行時の制御の流れに注目すれば、ＰＬＵを構成資源として用いた計算機は、従来のノイマン型計算機と同様逐次制御を採用しているため、単一のプログラムカウンタによって演算が規定される。しかし、演算の実行に注目すると、従来の計算機のＣＰＵに相当する制御部には演算機能が無く、全ての演算操作はメモリ装置に見えるＰＬＵ内で実行される。これは専用処理装置を持つ計算機が、処理装置へのデータ入出力のみで演算結果が得られることと同様であり、しかもその処理装置の機能をプログラマブルにしたものと考えられる。さらに演算処理の流れに注目すれば、ＰＬＵ内では必要な演算データが揃った時点から有効な演算が開始され、各演算データがあらかじめ定義されたパスを流れることにより、ゲート遅延時間レベルで演算処理が進む。この時、パス上を流れるのは被演算データそのものであり、通常のデータフロー計算機等で必要なアドレス情報等を必要としない。また、従来の計算機はその演算機能の制約から、大きなデータフローグラフを単項演算／二項演算に分解し、それらを制御の流れによってリンクすることで、データフローグラフを時分割で表現していた。これに対して、ＰＬＵではデータフローグラフそのものを、テンプレートの形で自然に表現可能であり、演算操作はより純粋なデータフロー演算である。従って演算処理の流れから見れば、本計算機はスタティックデータフロー計算機であると見なせる。

以上からＰＬＵを用いた計算機は、ノイマン型計算機とデータフロー計算機および汎用計算機と専用計算機の中間に位置する計算機であり、処理の汎用性と高速化の両方を実現する適当な妥協点を与えるものである。本稿ではＰＬＵの基本要素ＰＥの構造と結合形態について概説し、結合に制限がある場合の例として、格子結合型ＰＬＵを提案しその構成を示した。次にＰＬＵの制御方式および言語処理系について概説した。逐次制御を用いることで、ＰＬＵはノイマン型と同様な汎用性も提供する。さらに逐次制御に基づくＰＬＵを有効に稼働させる手法として提案されている多重化について、シミュレーションによってその有効性を確認した。この結果逐次制御に基づくＰＬＵでは、性能向上の手段として多重処理が非常に有効であることがわかった。他の計算機および演算方式の、モデル化と比較が今後の課題である。。

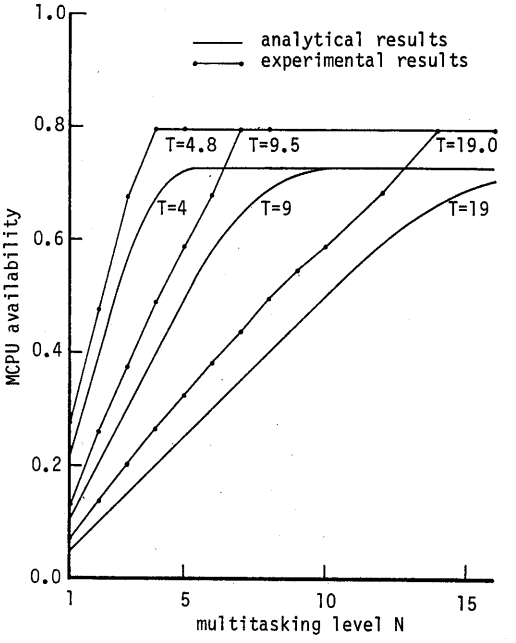


図10　多重度Ｎに対するMCPUの稼働率

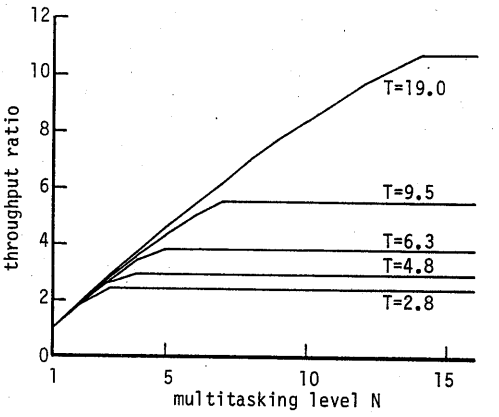


図11　多重度Ｎに対するスループット

```
INPUT:Binary tree with node Ni
OUTPUT:Assignment PE(i,j) for each node Ni

Initialize all PE(i,j), the logical number of PE, to UNUSED.

Perform the following recursive assignment starting from the root.
If assign(root,0,0) returns without encountering a FAILURE,
an assignment exists and corresponds to the PE's.

assign( node, i, j )
{
        PE(i,j) <- node;

        if( node is binary ) {
                if( search(i,j) = OK ) {
                        (nexti,nextj) <- direct(i,j);
                        assign( node left child, nexti, nextj );
                        if( search(i,j) = OK ) {
                                (nexti,nextj) <- direct(i,j);
                                assign( node right child, nexti, nextj );
                        }
                        else FAILURE
                }
                else FAILURE
        }
        else if( node is unary ) {
                if( search(i,j) = OK ) {
                        (nexti,nextj) <- direct(i,j);
                        assign( node child, nexti, nextj );
                }
                else FAILURE
        }
        else return;
}

search( i, j )
{
        if(( PE(i+1,j) = UNUSED )
         or( PE(i,j+1) = UNUSED )
         or( PE(i-1,j) = UNUSED )
         or( PE(i,j-1) = UNUSED )) return( OK );
        else return( NOT OK );
}

direct( i, j )
{
        (nexti,nextj) <- first( i, j );
        if( PE(nexti,nextj) = UNUSED )  return(nexti,nextj);
        else {
                (nexti,nextj) <- second( i, j );
                if( PE(nexti,nextj) = UNUSED )  return(nexti,nextj);
                else {
                        (nexti,nextj) <- third( i, j );
                        if( PE(nexti,nextj) = UNUSED )  return(nexti,nextj);
                        else {
                                (nexti,nextj) <- last( i, j );
                                return(nexti,nextj);
                        }
                }
        }
}

first( i, j )
{
        return( max(max((i+1,j),(i,j+1)),max((i-1,j),(i,j-1))) );
}

second( i, j )
{
        return( max(
                min(max((i+1,j),(i,j+1)),max((i-1,j),(i,j-1))),
                max(min((i+1,j),(i,j+1)),min((i-1,j),(i,j-1)))
                    )
              );
}

third( i, j )
{
        return( min(
                min(max((i+1,j),(i,j+1)),max((i-1,j),(i,j-1))),
                max(min((i+1,j),(i,j+1)),min((i-1,j),(i,j-1)))
                    )
              );
}

last( i, j )
{
        return( min(min((i+1,j),(i,j+1)),min((i-1,j),(i,j-1))) );
}

max( i1, j1, i2, j2 )
{
        if( (abs(i1)+abs(j1)) < (abs(i2)+abs(j2)) ) return( i2, j2 );
        else return( i1, j1 );
}

min( i1, j1, i2, j2 )
{
        if( (abs(i1)+abs(j1)) < (abs(i2)+abs(j2)) ) return( i1, j1 );
        else return( i2, j2 );
}
```

表3　マッピングアルゴリズム

```
main()
{
int     ans, x;
ans=0;

   for( x = 0; x < 16; x++ ) {
      ans = ans + ((((((((((((((((
                1*x+2)
                 *x+3)
                 *x+4)
                 *x+5)
                 *x+6)
                 *x+7)
                 *x+8)
                 *x+9)
                 *x+10)
                 *x+11)
                 *x+12)
                 *x+13)
                 *x+14)
                 *x+15)
                 *x+16)
                 *x+17);
   }
}
```

(a)ソースプログラム

```
;
;       This is test program
;
start:  MOVI    16,pe2
        MOVI    1,pe6
        MOVI    2,pe9
        MOVI    3,pe12
        MOVI    4,pe15
        MOVI    5,pe18
        MOVI    6,pe21
        MOVI    7,pe24
        MOVI    8,pe27
        MOVI    9,pe30
        MOVI    10,pe33
        MOVI    11,pe36
        MOVI    12,pe39
        MOVI    13,pe42
        MOVI    14,pe45
        MOVI    15,pe48
        MOVI    16,pe51
        MOVI    17,pe54
;
;       main    size=58 entry=0
;
main:   MOVI    0,pe0
        MOVI    0,pe1
AAA:    WAIT
        JNS     pe4,AAB
        MOVI    pe56,pe0
        MOVI    pe5,pe1
        JMP     AAA
AAB:    HLT
```

(b)MCPUコード

表4　多項式計算プログラム

文献

[1] 吉岡良雄:"メモリステップにプログラム可能な演算機能を持つメモリ装置の提案", 信学論(D), J66-D, 10, pp.1153~1160 (昭58-10)

[2] W.D.Hillis:"The Connection Machine", The MIT press (1985)

[3] 山田,村山,吉岡,中村,重井:"プログラム可能論理演算装置の動作特性", 信学論(D), J69-D, 9, pp.1264~1274 (昭61-9)

[4] 吉田,吉岡:"Prolog用プログラム可能論理演算装置について", 東北支部連大, 2J26, 昭61

[5] 吉岡,山田,村山,中村,重井:"プログラム可能論理演算装置を用いた計算機の実現性について", 信学論(D), J69-D, 10, pp.1246~1255 (昭61-9)

[6] 村山,山田,吉岡,中村,重井:"プログラム可能論理演算装置を用いた計算機の性能評価", 信学論(D), J69-D, 9, pp.1256~1263 (昭61-9)