関数型データフロー計算機システム

Data-Flow Machine Architecture Based on Functional Program Model

伊藤 敏美　斉藤 恒雄　星3 幸男
Toshimi Itoh　Tsuneo Saitoh　Yukio Hoshiko
東北大学　工学部
Faculty of Engineering, Tohoku University

1. まえがき

半導体製造技術の急速な発展に伴ない計算機システムのハードウェアコストが低下する一方ソフトウェアの比重が相対的に増大してきている．このような状況の中で，記述性，生産性，信頼性などに優れたプログラミングシステムとして関数型システムの重要性が認識されるようになってきている．

また，こうした素子技術の進歩を背景に，並列処理や分散制御システムがますます重要なものとなってきており，処理に内在する並列性を十分に引き出し効率的に実行する計算機システムやＶＬＳＩ環境に適する計算機アーキテクチャーの開発が強く要求されている．

これらの要求に対し，次のような一般的な特徴を備えたデータフローシステムは，従来のノイマン型モデルを基礎としない新たな計算機システムの一つとして注目されている．

(1)処理間の実行の順序制御はそれらの間のデータの依存関係によって規定されるので，処理に含まれる並列性を自然な形で表現することができる．処理のレベルとして，ジョブ，プログラム，手続きなどを考えることもできるが[6]命令レベルでのデータフローシステムを扱うことによって，従来の計算機システムでは困難であった高度な並列処理が可能となる．

(2)メモリセルや処理間で共有される広域的な変数の概念はなく，処理間で受け渡されるのは値そのものである(Value Oriented)．

(3)処理は，その処理に必要とされるデータが与えられることによって起動され(Data駆動)，その処理の結果は入力データの値だけに依存し，過去の履歴や周囲の処理の進行などに影響されない(副作用がない)．

従って，データフローにおける処理は一つの関数として考えることができ，関数型言語のもつ概念を直接的に反映させた関数型データフロー計算機を構成することが可能である．

しかし，こうしたデータフローシステムのもつ特長を活かし，一つの実用的な関数型システムとして実現するには多くの未解決の問題が残されている．

その1つはデータフローモデル自体の構成法である．データフローモデルをどのように位置付けるかにもよるが，関数性という立場から見ると，現在までに提案されている幾つかのデータフローモデルにおいては，特殊な制御オペレータの導入やループ構造の導入などによって関数性が徹底されていない．

本稿では，計算の過程をデータフローグラフの縮約(Reduction)の過程として捉えたデータフローモデルを提案し，その具体的実現方法やマシンアーキテクチャーなどについて述べる．

2. データフロープログラムモデル

λ計算モデルに代表されるように，関数型モデルでの計算過程は，与えられた変換規則に従って，式自体が各ステップでの意味的な同値性(identity)を保ちながら縮約されて行く過程として考えることができる．[5]

データフローモデルでは，式はデータフローグラフに対応し，式の評価過程はデータフローグラフの縮約の過程として考えることができる．

また，アークは変数に，オペレータノードは変換規則に対応して考えることができ，関数呼出しオペレータの実行は，このオペレータノードをその関数本体に置換することに対応する．

本稿で述べる関数型データフローモデルとは，データフローグラフの縮約と置換の概念に基づき，かつデータ駆動によってその実行が制御されるデータフローモデルである．

結果として，他のデータフローモデルの幾つかのものとは異なり，特別な制御オペレータや条件分岐構造，ループ構造を導入する必要がなく，統一的な制御規則と単純な機構によってデータフロープログラムを処理することができる．

本モデルについて述べる前に，関数性という観点から従来のデータフローモデルについて検討する．

2.1 従来モデルの問題点

長谷川ら[1]は，同様に，主として制御構造から生じるglobalな状態概念を除去し関数性を徹底するという立場から，Dennis [2] 等のモデルを中心として検討を行なっている．

すなわち，これらのモデルでは

(1) データフローグラフの構成法によってはトークンの衝突やデッドロックが発生する可能性がある．

(2) 特別な発火規則を持つ制御オペレータが存在し，関数性を損なっている．

(3) アーク上を2度以上トークンが流れるため，トークンの衝突回避のために先行チェックを行なう必要がある．

などの問題点を挙げ，ループ構造を禁止し再帰関数を用いること，及び単一代入規則を徹底しアーク上に2度以上トークンを流さないというモデルを提案している．

このモデルでは，Dennisらのモデルに比べより関数性が徹底されているが，条件式の実現のためには Switch オペレータや Merge オペレータが用いられている（図1）．

データフローモデルの関数性として何を考えるかにもよるが，データフローグラフ上のトークンの分布といった制御上の状態に，基本的には関数として捉えられるデータフローオペレータの実行が強くは依存しないということが，関数性の一つの要請であると考えられる．

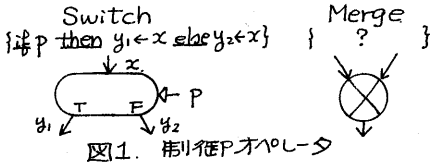


図1．制御型Pオペレータ

この意味において上記の(1),(3)の側面は，関数的なデータフローモデルからは取除くべき性格のものである．

本稿で述べるデータフローモデルの関数性は上記の要請の他に，オペレータやアークなどのデータフローモデルの基本的要素に対して，関数的意味付けが可能であることを要求する．

データフローグラフのアークは変数に対応するが，関数型システムにおける変数は，1つの対象（Object）を表わすものであり，一度その値が確定すればプログラムの実行中にこれが変化することはない．この意味からも，ループ構造を禁止し，単一代入規則を徹底する必要がある．

また，図1に示したif文に対応するSwitchや関数として意味付けの困難なMergeオペレータはこの点において問題となる．

これらの従来モデルに共通する概念は，データフローグラフをその実行中不変のネットワークとして考え，計算過程をこのネットワーク上を流れるトークンの流れを制御し，その出力端にトークンを導く過程として考えていることのように思われる．

2.2 関数型データフローモデル

本モデルにおいては，データフローグラフ自身が，トークンによって駆動され縮約されて行くことによって計算が進行する．

すなわち，入力トークンとオペレータは実行によって出力トークンに置換される．この様子を図2に示す．オペレータノード内のラベル（記号）は，置換の規則を示している．

具体的には，次の規則を順守するデータフローモデルを関数型データフローモデルと呼んでいる．

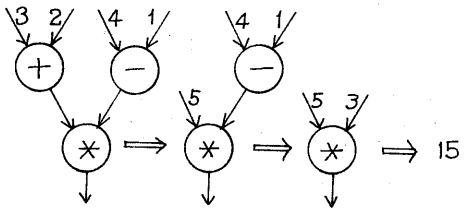


図2. データフローグラフの縮約

(1)［構成規則］

データフローグラフは，オペレータノード及びアークから成る非巡回的な有向2部グラフである．オペレータは少なくとも1つの入力アークを持ち，複数の入力アークを持つ場合には，その接点位置が識別可能なものとする．また，アークの両端に接続するオペレータノードは高々1つであり．アークが分岐したり合流したりしない．

(2)［発火規則］

オペレータノードの全ての入力アーク上にトークンが到着することによって，そのオペレータは発火可能となる．発火可能なオペレータは，発火によってその全ての出力枝上にトークンを出力し，このとき，入力アーク，入力アーク上のトークン及びオペレータノードは消滅する．

(3)［関数定義と関数呼出し］

関数定義は，関数名及びその本体であるデータフローグラフから構成される．この本体において入力端にノードを持たないアークを仮アークと呼び，仮アークは少なくとも一本以上存在し，複数の仮アークがある場合にはその相対位置が順序付けられ互いに識別可能なものとする．

関数呼出しのオペレータ（CALL）が実行されると，呼出された関数本体がコピーされ，CALLオペレータノードがこのコピーされた本体に置換えられる．このとき，CALLオペレータの実アークと本体の仮アーク及びRETオペレータの間は動的にリンクされる．（図3）CALLオペレータの実アーク数と呼出す関数の仮アーク数は一致しなければならない．

ここで述べた構成規則や発火規則は，他のデータフローモデルと比べると強い制限が課せられている．

オペレータが全ての出力枝上にトークンを出力することを要請するのは，オペレータを関数と見なせるようにするためである．

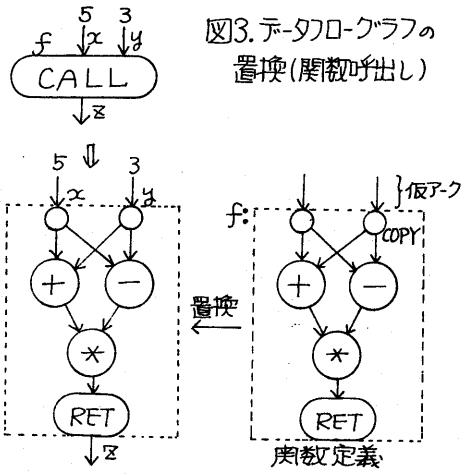


図3. データフローグラフの置換（関数呼出し）

ループ構造が禁止されていること，アークの入力オペレータが高々1つであること，及び関数呼出しの都度コピーが行なわれるので，単一代入規則は守られ，トークンの衝突は生じない．

また同様にループ構造が無いこと，オペレータは全ての出力アーク上にトークンを出力することなどから，関数呼出し時に引数の数が一致している限り，全てのオペレータが唯一度だけ実行される．従って，プログラムの実行中アーク上にトークンが位置付けられるのは高々一度であり，かつ少なくとも一度である．このことから，アーク＝変数＝対象として考えることができる．

(4)［条件式の扱い］

本モデルでは条件式の機能を実現するために特殊な制御オペレータや条件分岐構造を用いることはしない．

関数型システムにおける条件式の役割は，条件の成否によって評価すべき式自身を選択する，或いは置換すべき式を選択することである．

データフローグラフは式に対応するので，条件に応じてデータフローグラフを選択してその入力アーク上にトークンを位置付ければ良い．すなわち，選択の機能と関数呼出しによって行なうことができる．図4に条件式

if p then f(x) else g(x) ----(1)

に対応するデータフローグラフを示す．

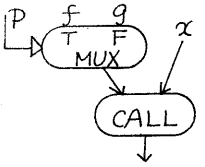


図4. 条件式(1)
if p then f(x) else g(x)

ここで，MUXオペレータは(2)の発火規則を満たすもので，if_then_else 関数の実現である．すなわち，(1)式の処理は，

(if p then f else g)(x)

の形で処理される．

しかし，(1)式の条件式は最も単純なものであり一般の条件式を扱う場合には次のような問題がある．第1に，(1)式の式 f(x)，g(x)は関数呼出しの形をしているが一般には式 E_1，E_2 である．第2に，これらの各式 E_1，E_2 において参照する変数の集合は同じであるとは限らない点である．前者に対しては，コンパイラ等によって式 E_1，E_2 に対応する関数定義が成されていることを仮定する．すなわち，一般的な条件式

if p then E_1 else E_2 -----(2)

に対し，

$f_1(\langle Ref(E_1)\rangle) \equiv E_1$，$f_2(\langle Ref(E_2)\rangle) \equiv E_2$

の形の2つの関数定義が成される．ここで，$Ref(\cdot)$ は式において参照される変数集合を表わし，$\langle\cdot\rangle$ はその順序付けを表わす．

第2の点に関しては，基本的にはこの局所的な関数呼出しの前後で，$Ref(E_1)$，$Ref(E_2)$ をそれぞれデータ構造化し，

(if p then f_1 else f_2)(if p then $\langle Ref(E_1)\rangle$ else $\langle Ref(E_2)\rangle$)

の形で処理できる．しかし，これではデータ構造化とその要素への分解へのオーバーヘッドが大きく，実現時の大きな問題となる．

（条件式の駆動条件） 一般に，データ駆動による処理は，その処理に必要とされる環境が確定することによって起動されると考えることができる．環境は対象の集合であり，単純な式の場合にはその処理に局所的であり，必要十分な環境が確定することによって駆動される．一方，条件式の中の式の場合には，条件式全体に必要となる環境が確定することによって駆動されると考えられる．すなわち，(2)の条件式に対して

$f_1(\langle Ref(E_1) \cup Ref(E_2)\rangle) \equiv E_1$
$f_2(\langle Ref(E_1) \cup Ref(E_2)\rangle) \equiv E_2$ }

の2つの関数定義が成され，図5に示す様なデータフローグラフを構成することによって，データ構造化を行なうことなく一般的な条件式の処理を行なうことができる．この場合，選択した関数では使われないデータが渡されることがあり，このようなトークンの吸収のために depose オペレータが用いられる．

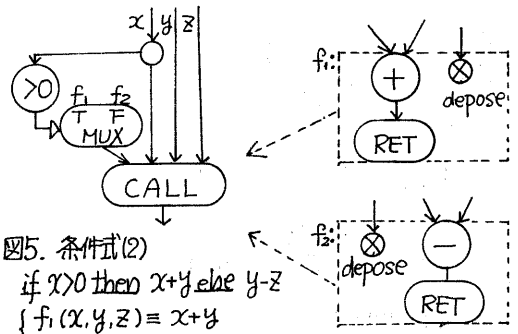


図5. 条件式(2)
if x>0 then x+y else y-z
$\begin{cases} f_1(x,y,z) \equiv x+y \\ f_2(x,y,z) \equiv y-z \end{cases}$

(5)［繰返し処理］

本モデルでは，ループ構造が禁止されていることから明らかなように，繰返しの処理は全て再帰関数とその呼出しによって行なうことを基本としている．

2.3 本モデルの特徴と実現上の問題点

本モデルでの構成条件や発火条件は，従来モデルに比べると強い制限が課せられている．また，従来モデルがデータフローグラフをその実行中不変なものと考えているのに対し，本モデルではデータフローグラフ自体がトークンによって駆動されることによって縮約されて行く．

これらのことから，本モデルの特長として

- 関数型言語の概念を直接的に反映させた関数型マシンを構成することができる．
- 従来モデルの持っていた，特に制御モデルからの状態遷移的な性格を除くことができる．
- 制御構造が極めて単純なものとなり，単純な機構で関数型言語を効率的に実行できる可能性がある．
- トークンの衝突は生じない．また，関数呼出しにおいて引数の数のネ一致がない限りデッドロックは生じない．
- 単一代入規則が徹底されている．

などを挙げることができる．

一方，このモデルを実現しようとする場合，

幾つかの問題点が生ずる.

本モデルでは,関数呼出しの都度その本体がコピーされるが,データフローグラフを縮約するという立場からこのコピーは本質的に必要なものである.しかし,関数のコピーは以下に述べる様に必ずしも重大な効率の低下を招くとは考えてはいない.

すなわち,関数のコピーは実際には関数呼出し時に全体を一度に行なう必要がなく,オペレータ毎にオペレータが実行される直前までに(例えばその入力アーク上に最初のトークンが現われた時点で)コピーすれば良い.この過程は通常の計算機が命令をフェッチする過程と同じである.また,オペレータや入力トークンは実行と同時に消滅するので,ある時点で実質的にコピーして記憶しておく必要があるのは,入力アーク上にトークンを持ちかつ他のトークンを待ち合わせているようなごく一部のデータフローグラフだけで良い.このために要求される記憶量は従来の計算機システムでのスタックや一時変数のために必要となる記憶量に比べ,ほぼ並列度倍だけ増大しているにすぎない.

コピーに関するより重大な問題は,関数のコピーよりもデータのコピーの時に発生する.データフローモデルにおけるトークンは値そのものであるので,値が構造的なものである場合にはコピーは著しいオーバーヘッドを引起こす.本処理システムでは,これを通常の参照カウント方式で処理しているが,これについては5章で述べる.

関数型データフローモデルでは,従来モデルとは異なり条件式や繰返しに対しても関数呼出しを行なうので,関数呼出しがより頻繁に発生する.従って,このような関数呼出しが効率良く処理できることが要求される.

3. データフロープログラムの実行制御

本章では,関数型データフローモデルを実現するための具体的機構として,まず機械語構成を示し,トークンの同期制御,関数呼出しの機構などについて述べる.

関数呼出しは2つのモードを持っており,呼出しプロセスとは別なプロセスとして実行する(CALL)モードと,呼出しプロセス内に埋込まれてその一部として実行される(LINK)モードがある.前者は通常の関数呼出しに対して用いられ,後者は条件式実現のための関数呼出しや繰返し処理実現のための末尾再帰関数の呼出しのために用いられる.

3.1 機械語の構成

図6に機械語及び命令レジスタの構成を示す.機械語と命令レジスタの形式が異なるのは,トークンの値を保持するための領域は実行時においてのみ必要とされるからである.

機械語はデータフローグラフのオペレータノード及びその出力アークに対応する.関数本体を構成するデータフローグラフの全体は,機械語に変換されて一つのセグメントを構成している.各フィールドの意味は以下に説明する.

(1) 命令コードフィールド(OPC):演算機能を指定する.

(2) オペランド数フィールド(N):入力アークの数を指定する.基本命令では1又は2であるが,一般に多くの(15程度まで)オペランドがあっても良い.

(3) 行先フィールド(SD, LD):トークンの送り先を示す.LDフィールドは,更に番地フィールド(LOC)とオペランド番号フィールド(FA)とから構成され,前者はセグメント内での命令語の位置を指定し,後者はそこでのオペランドの位置(入力アークの接点番号)を示す.SDフィールドは,LDフィールドと同様の役割を持つが,LOC部が省略されており次の番地に限定される.但し,値0はSDフィールドを用いないことを示す.

(4) 定数フィールド(LiT):LDフィールドはまた定数のために用いられ,両者は命令コードによって区別される.

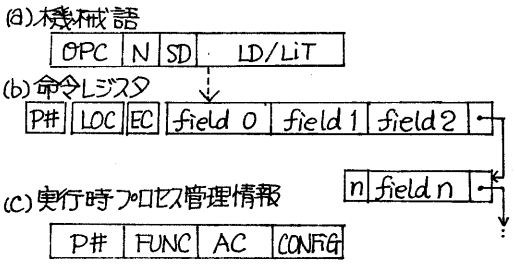


図6. 機械語/命令レジスタの構成

命令レジスタはコピーされたオペレータおよびトークンを保持するもので，従来の計算機とは異なり多数存在する．命令レジスタはプロセス番号P#，命令アドレスLOCによって連想アクセスされるのでCAM(Content Addressable Memory)を用いて実現するのが望ましい．

命令レジスタは，実行制御ユニット(ECU)内に置かれるがこれについては後で述べる．(図8，図11参照)．

基本的な命令の実行は以下に示すようにして実行される．

前のステップの命令の結果が演算ユニットなどから結果パケット

<P#, LOC.FA, data >

として送られてくると，実行制御ユニットは，<P#, LOC>をkeyとして命令レジスタを検索する．このとき，検索が失敗するとこの命令を読出すために実行制御ユニット内のキャッシュメモリを検索し，無ければメモリに対して

<fetch , FUNC , LOC.0 , P# >

の形のフェッチパケットを送る．FUNCはこのプロセスが実行している関数名である．また，空きレジスタにkeyを登録し，EC (Enabling Count)部を0にクリアしておく．

次に求めた命令レジスタ内のFAに対応するフィールド位置にdataを書込む．この場合，大きなFAに対しては図6のようにオペランドデータ部をリンクする必要があるが，これは関数呼出し命令の時などに行なわれるだけであり，基本命令に対してはこれを行なう必要はない．

オペランドデータの書込みと同時に命令レジスタ内のEC部は次のように変更される．

FAが0のときは，このデータは機械語であることを示しておりそのオペランド数フィールドの値NがECの値に加えられる．FAが0でないときはEC←EC-1とされる．これらの変更の結果，EC部が0となると全てのオペランドデータおよび機械語コードが揃ったことになり，命令は実行可能となる．

この場合には，命令レジスタの全内容を読出して命令パケット

<OPC , data, … , data, LD/LIT, LD2, P#>

を編成し，命令コード部の実行を行なえる機能ユニットに向けてこれを発信する．ここで，LD2はSDフィールが指定されていた場合，その省略値を補った行先アドレスLOC+1.SDである．尚，命令レジスタはこのパケット編成時に解放される．

この命令パケットが演算ユニットなどで処理されて結果パケットを返すことにより，次の命令が起動される．

プロセス管理情報として，関数名FUNC，AC (Activity Count)，このプロセスの使用しているレジスタ数CONFIGなどが使われる．ACは，命令レジスタが待ち状態のプロセスによって占められてしまった場合にこのプロセスの追い出しに用いられる[3]．

3.2 関数呼出しの処理

本節ではまず通常の関数呼出し(CALL)について述べ，ついで条件式や繰返しの処理(LINK)について具体的処理方法を示す．

n入力m出力の関数呼出しの呼出し側および本体側の機械語ブロックを図7に示す．

CALL命令が実行可能となると，通常の命令と同様にCALLパケット

<CALL , f, x_1, … , x_n, k+1.1 , P#>

が編成され，プロセス・スケジューリングを役割とするプロセス管理ユニット(PCU)に送られる．プロセス管理ユニットは，これに対しシステムで一意的なプロセス番号new-P#を生成し，適当な実行制御ユニットにプロセス起動パケット

<INVOKE , f, new-P# >

を発信する．このとき，プロセスの相互関係及結果の戻し先などを記録しておく．プロセス起動パケットの作用は，実行制御ユニットに割当てられたプロセスおよびその関数名を通知することである．引続いて，各実引数x_iに対し結果パケット

<new-P#, i.1 , x_{i+1} >, i=0, n-1

を作り，先の実行制御ユニットに発信すればプロセスが起動する．

関数本体の最初のn個の命令をリンケージブロックと呼ぶが，実引数はこの各オペランドとして送られる．ENTER命令は単にデータを受渡すだけの命令で，一般にこの部分は任意の命令であっても良い．

呼出されたプロセスでRET(urn)命令が実行

SEG	LOC	OPC	N	SD	LD/LIT	
	k	CALL	n	1	f	呼出し側
	k+1	ENTER	1	0	y_1	
	:	:	:	:	:	
	k+m	ENTER	1	0	y_m	
	:	:	:	:	:	
f	0	ENTER	1	0	x_1	関数本体
f	:	:	:	:	:	
f	n-1	ENTER	1	0	x_n	
f	:	:	:	:		
f		RET	m			

図7. 関数呼出し手続きブロック

されると，プロセス管理ユニットにRETパケット　<RET, y_1, … , y_m, new-p#> が送られ，呼出されたプロセスは終了する．

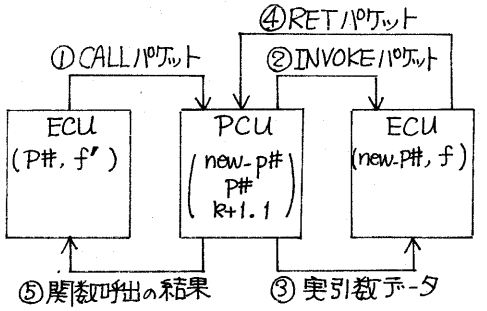
プロセス管理ユニットは，呼出しプロセスに対し，結果のパケット

< p#, k+i.1, y_i >　　$i=1, m$

を送ることによって関数リンケージが行なわれる．CALL命令に引続くm個の命令群もリンケージブロックを構成し，結果はこの各オペランドとして戻される．

これらの関数呼出しの様子を図8に示す．

図8. 関数リンケージの手順



次にもう一つの関数呼出しの機構（LiNKモード）について説明する．

2.2において条件式は2つの関数定義とその呼出しによって処理できることを述べた．しかし，これらの関数定義は一般の関数定義とは異なる性質を持っており，これを利用することにより効率的な処理が可能となる．すなわち，通常の関数定義が任意の場所から呼出される可能性があるのに対し，条件式から生じる関数定義は呼出される場所が唯一であるという点である．

従って，その結果を戻す場所も唯一であり，コンパイル時にこれを埋込むことによってRETオペレータを除くことが可能となる．また，条件式の内部の処理は元のプロセスの処理に含まれると考える方が自然であり，オーバーヘッドも少ない．

LiNKオペレータの作用は，動的にデータフローグラフを結合することである．図9に先に例として用いた条件式（図5）に対応する機械語を示す．（機械語の持つ定数フィールドは一つであるので，定数発生命令CONSTによってf_2を生成している）

LiNK命令の一般形は <LiNK, f, x_1, …, x_n> であり，fは局所関数の入口を示す番地である．LiNK命令は，その各引数データに対し

< p#, f+i.1, x_{i+1} >，$i=0, n-1$

の結果パケットを発生する．但し，p# はこのLiNK命令を発したプロセス番号である．

次に繰返し処理について述べる．

繰返しの処理は，基本的には末尾再帰関数として扱えば良い．しかし，この場合にも末尾再帰に特有の性質：この関数が再度自分自身を呼出すのは，その本体処理の最も最後の命令であること，従ってこの呼出しを行なうときにはこの本体の他のオペレータノードは全て消滅していることを利用することができる．図10にLiNKオペレータを用いた繰返し処理の様子を示す．

LOC	OPC	N	SD	LD/LIT
	COPY	1	1	$LiNK.2
	>0	1	1	$MUX.3
	CONST	1	0	$MUX.2
$MUX	MUX	3	1	f_1
$LiNK	LiNK	4	0	—
f_1	ADD	2	0	result
	ENTER	1	0	f_1.2
	DEPOSE	1	0	—
f_2	DEPOSE	1	0	—
$SUB	SUB	2	0	result
	ENTER	1	0	$SUB.2

図9. 図5の条件式の機械語表現

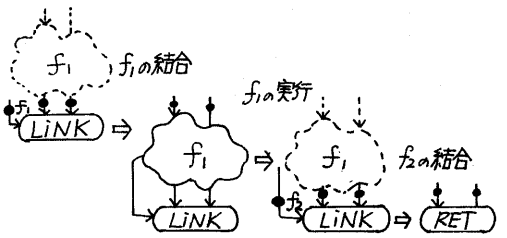


図10. 繰返し処理

4. データフロー計算機の構成

図11に基本的なデータフロー計算機の構成を示す．図12は，小中規模のマルチプロセッサシステムの構成を示している．

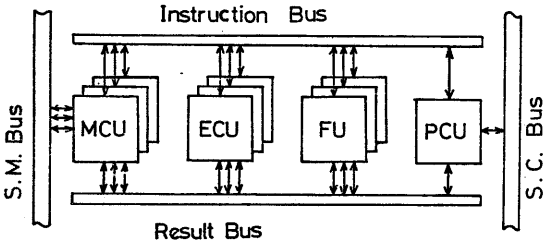
図11のデータフロー計算機は4種類の機能ユニットと2つのバスシステムとから構成されている．各ユニットの役割について以下に説明する．

(1) 実行制御ユニット（ECU）

前章で述べたトークンの同期制御，命令のフェッチの機能を行なうもので，一般に複数の実行制御ユニットが1つのデータフロー計算機の中に含まれる．また，1つの実行制御ユニットの中では，同時に複数のプロセスが実行可能である．但し，1つのプロセスは一時には1つの実行制御ユニット内でのみ処理される．

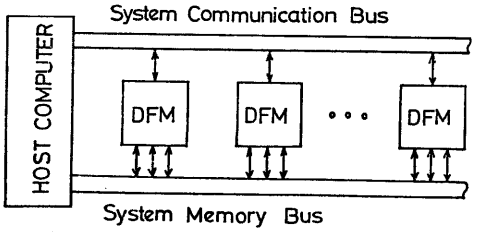
結果パケットは全て結果バス(Result Bus)から入力して，実行制御ユニットは命令パケットを命令バス（Instruction Bus）に出力する．

図13にその内部構成を示す．その中心を成しているのは命令レジスタ，プロセス管理表及び命令フェッチによるメモリ競合を減らすための



Data Flow Machine Architecture

図11. データフロー計算機の構成．



Multi-Processer System

図12. マルチプロセッサシステムの構成．

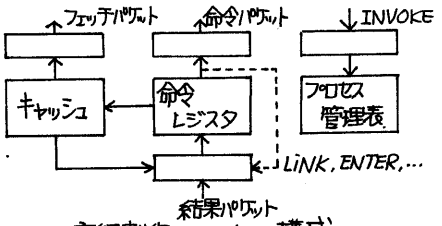


図13. 実行制御ユニットの構成．

キャッシュであり，これは全て連想アクセス機能を必要とする．キャッシュは過去においてフェッチした機械語の幾つかを記憶し，繰返し処理や再帰関数の処理効率を高めるものである．

先に述べたENTER，LINK，及びコピー命令の一部は実行制御ユニット内で処理される．

(2) 演算ユニット（FU）

算術／論理演算を行なうユニットである．

(3) プロセス管理ユニット（PCU）

関数呼出し（CALL）及び結果のリンク(RET)処理を行ないプロセスの広域的スケジューリングを行なうためのユニットである．一般にプロセスをどの実行制御ユニットに割付けるかは任意であるが，キャッシュの効率を高めバス及びメモリにおける競合を減らすためには，同じ関数の呼出しに対しては同じ実行制御ユニットに割付ける方が望ましい．反面，同じ関数に対する呼出しが多発する場合には別な実行制御ユニットに負荷を分散させた方が効率が良くなることも予想される．

また，プロセスの実行をシステム通信バスを通して他のデータフロー計算機に行なわせることが可能であるが，この場合割付け先のマシンに関数のコピーがない場合には，システムメモリバスを通して本体をコピー（DMA転送）する必要があり，プロセスのスケジューリング方法については今後の検討課題の1つとして残されている．

再帰呼出しを繰返して行く場合，プロセスはいつかは結果待ちとなって待ち状態になる．このプロセスによって占められている命令レジスタ数は多くはないと予想されるが，限られた命令レジスタを有効に使用し大規模な計算を行なうにはプロセスの追出しが必要となる．先に述べたAC（Activity Count）はこのような目的でプロセスの状態をモニタするのに使われる．プロセス管理ユニットはプロセス相互間の呼出し

関係（プロセス木）を保持しており，追出しは根に近い待ちプロセスから行なう．結果の到着によってこのプロセスを再開する場合，割当てる実行制御ユニットは元のユニットと同一である必要はない．

(4) メモリ制御ユニット（MCU）

データ構造の記憶，管理及びそのデータ構造に対する各種の操作を行なう機能ユニットで，物理メモリの構造に束縛されない仮想化されたデータ構造をプログラムに提供する．これについては次章で再び述べる．

(5) 命令及び結果バス（Instruction/Result Bus）

各ユニット間でのパケット転送のためのバスシステムで，高速非同期バスとして実現される．

5. データの表現とデータ構造の処理

本処理システムにおけるトークンの内部表現は，命令によってその意味の定まる解釈自由なビット列としてではなく，そのデータタイプを示すタグを付けて表現される．

この結果，データの表現に要するビット数は幾分増加するが，オペレータの機能が一般化する，不正なデータタイプ間の演算を実行時に検出できるなどの利点があり，特に動的データ構造を扱う場合には必須のものと思われる．

データタイプとしては，整数型，実数型などの基本的な型の他に，リスト構造データを表わすポインタ型，配列などを表わす構造記述子型などが用意されている．

データフローモデルでのトークンは値そのものであり，これは値が構造的な場合も同じである．しかし，データ構造自体をトークンとして用いるのは実用的でないので，そのデータ構造へのポインタ（前述のポインタ型，構造記述子型）がトークンを表わすのに使われる．また，データ構造のコピーはこのポインタのコピーによって行なうことにより，コピーのオーバーヘッドを無くすることができる．反面，ガーベッジコレクションを行なう必要が出てくる．

また，構造的な値に対する演算は通常その表現形式としてのデータ構造の一部だけを変更することで済むので，新たにデータ構造を生成するのではなく既存のデータ構造を用いる方が効率的な処理が可能となる．反面，前述のコピーによってこのデータ構造が共有されている場合は副作用を発生する．

こうした副作用を除き，データ構造の共有制御，ガーベッジコレクションを行なう方式として参照カウント方式[3]が知られているが，本システムもこの方法を採っている．

本処理システムで扱う基本的なデータ構造はリスト構造であるが，一般にプログラムの要求するリスト構造は，各ノードにおける次数，要素の型，従って必要なビット数などは様々である．本処理システムでのデータ構造処理に対する基本的な考え方は，プログラムの要求する多様なデータ構造の中で多用されるものに関しては，メモリ側の機能に含めることであり，また極力物理的な表現形式に捕われない論理データ構造をプログラムに提供するということである．

演算ユニットが算術／論理演算を行なうのに対し，メモリ制御ユニットは全てのデータ構造に対する操作を行なう機能ユニットである．

図14はそれぞれ(a)メモリ制御ユニットの提供する論理データ構造（ソフトウェアレベル），(b)メモリ制御ユニットの扱うデータ構造（ファームウェアレベル），(c)物理メモリでの表現形式（ハードウェアレベル）を示している．特殊なデータ型dmyはメモリ制御ユニット内でのみ用いられるポインタを示している．

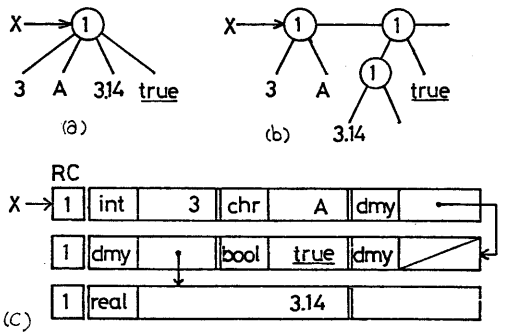


図14．リスト構造

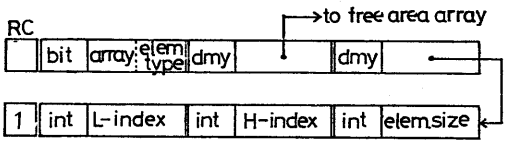


図15．配列構造記述子

図15は配列を表わす構造記述子を示している．実際の配列データはメモリ制御ユニット内の別な領域に置かれ，この領域へのポインタはソフトウェアレベルには渡されない．また，配列の添字計算などもメモリ側の役割として位置付けている．

このようにデータ構造の操作と管理（参照カウントによる共有制御，ガーベッジコレクション，メモリスペース管理など）をメモリ側の機能とすることは，マシン内のトラヒックを減少させる，処理の分散が成される，強力でかつ効率的な処理機能が提供されるという点で有効な方法と考えられる．しかし，抽象データ型の概念にも見られるようにデータ構造とその処理法は無数に存在し，その中のどのデータ構造に対してファームウェア化するか，或いはソフトウェアで実現するかはマシン上に実現される言語を含めて決定する必要があり，今後の検討課題として残されている．

6. 実験システム

関数型データフローシステムに対する評価を行なう目的で，図11のサブセットの部分から成るデータフローマシンを開発している．

各ユニットは，図16に示すような汎用の構成を採っており，各ユニットの機能を行なうよう制御プログラムを書換えることでデータフローマシンの機能をエミュレートするものである．

現在，4ユニットがハードウェア的にほぼ完成しており，ファームウェア及びソフトウェアの開発段階に入っている．

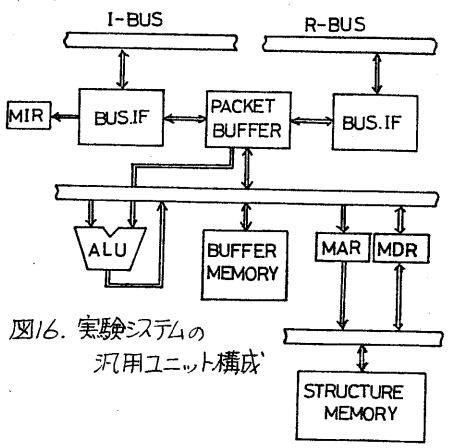


図16. 実験システムの汎用ユニット構成

7. あとがき

本稿ではまず置換と縮約という関数型言語のもつ2の基本的概念を素直に反映させた関数型データフローモデルについて述べ，結果として単純で統一的な機構でデータフロープログラムを実行できることを示し，またその具体的実現方法やデータフロー計算機の構成について述べた．今後の課題としては，関数型言語を含めた実験システムの完成，プロセススケジューリングアルゴリズムの検討など先にふれた問題の他に，副作用の導入の問題がある．

現在のデータフローモデルでは全ての副作用が禁止されているので，実質的にデータベース，ファイル，入出力などの機能は従来型計算機の補助を必要としているが，1つの計算機システムとして確立するためには何らかの形である種の副作用を導入する必要があると思われる．これをどのような形で，データフローモデルの全体のわく組の中に組込むかが，今後の大きな課題として残されている．

参考文献

(1) Dennis, J.B. et.al, "A Preliminary Architecture for a Basic Data Flow Processor", MIT Proj. MAC Comput. Struct. Group Memo 102, 1974

(2) Dennis, J.B. "First Version of A Data Flow Procedure Language", MIT Proj. MAC TM-61, 1975

(3) Rumbaugh, J. "A Data Flow Multi-Processor", IEEE C-26, 2, 1977.

(4) Weng, K.S., "Stream Oriented Computation in Recursive Data Flow Schemas," MIT Proj. MAC TM-68 1975.

(5) Stoy, J. "Denotational Semantics".

(6) P.C. Treleaven, "Exploiting Program Concurrency in Computing System", IEEE Computer, 1, 1979

(7) 長谷川他，"連想記憶を用いたデータフローマシンの一構成法"，信学会計算機アーキテクチャー研資，EC-79-55, 1980