解　説

機能メモリのアーキテクチャとその並列計算への応用

7. 人工知能への応用†

樋口哲也††　古谷立美††

1. はじめに

連想メモリの応用といった場合に，一般にまず思い浮かべるのは，仮想アドレスの変換機構の中で使われているようなテーブル検索の高速化であろう．データ数にかかわらず，つねに一定時間で検索操作を実行することができるので，SIMD (Single Instruction Multiple Data stream) 型の並列処理が実現されている．

しかしこのような単純な使い方は連想メモリの用途のごく一部であって，たとえば連想メモリ用数値演算アルゴリズムにみられるように，検索操作の途中結果を連想メモリの各ワード内にいわばレジスタ的に保持し，それに基づいて以後の計算を進めるような方法をとることによって，より多くの応用に連想メモリを用いることができる．

このような使い方は，連想メモリの一語一語をいわば小規模の PE (Processing Element) として用いているわけで，大容量の連想メモリを使うことによって超並列システムを構築できることを示している．ただし，これまでは大容量連想メモリが入手しにくいこともあって，このような連想メモリの利点が一般に十分に理解されているとは言いにくい．

しかしながら近年では大容量の連想メモリ，および機能メモリ（通常のメモリになんらかの特殊な機能を付加したもので，連想メモリもその一部に入る）の研究，商品化が進み，これらのメモリを用いた専用システムの研究開発が行える状況となってきた．そして最近では応用がある程度専用的ではあるが，他の超並列システムと比較しても遜色のない性能を達成できるケースも示され始めている．

† Artificial Intelligence Applications with Content Addressable Memories by Tetsuya HIGUCHI and Tatsumi FURUYA (Electrotechnical Laboratory).

†† 電子技術総合研究所

そこで本稿では，人工知能への種々の応用例を対象に，このようなSIMDデバイスとしての機能メモリの導入効果とその使い方を述べることにする．応用対象としては，リスト処理，部分マッチング，プロダクションシステム，意味ネットワーク，遺伝的アルゴリズムをとりあげる．

超並列デバイスとして機能メモリを用いる際には，第一にSIMD的なデータレベルの並列性が問題において利用可能かどうか，第二にこれらのメモリ上でのデータ表現をどう行うかを考慮することが重要である．

後者はシステムの実現者が直面する中心的な課題であり，また機能メモリの導入効果自体を左右するものである．つまり，機能メモリはメモリの1ビットあたりにそれぞれ論理機能が付加されている，いわば高価な並列処理計算資源であり，実際の応用をみてみると，1ビットでも有効に使おうと，処理すべき情報の効率的な表現方法に実に多くの工夫が払われていることが分かる．したがって本稿で応用事例を示すにあたっては，データ表現技法の実際面を重視する．また機能メモリの導入効果については一部にクレイやコネクションマシンなど他の高性能アーキテクチャとの比較も含める．**2.** では人工知能処理にあたって機能メモリに必要とされる機能の考察と，システム例について述べる．**3.** では人工知能への応用例について述べ，**4.** では今後の課題について考察する．

2. 人工知能処理向き機能メモリアーキテクチャ

本章では応用対象として人工知能の諸問題を考えたときに一般的に要求される機能メモリの要件を考察し，また現在主に人工知能処理向きとして実現されているメモリとシステム例について述べる．

2.1 人工知能応用における機能メモリの要件

人工知能応用といっても実に多岐にわたるが，次のような性質をもつ場合が機能メモリを用いるのに適切ではないかと考えられる．まず第一に処理対象となるデータ構造が配列など単純な構造では表しにくく，リスト，レコード，ネットワークなど，より複雑な構造で表現せざるをえない．第二にそれらのデータに対しパターンマッチや条件検索を繰り返し行うことによって目的とするデータを抽出することが要求されるが，検索条件についての事前の予測がむずかしく，したがってハッシュなどの技法を適用できないような場合である．第三に，処理データレベルにSIMDの並列性が利用可能な場合である．

これらの処理にあたって機能メモリのもつべき主たる機能はいうまでもなく検索機能であるが，これに加えて次のような機能の付加が望ましい．以下に5つの機能をあげるが，以下の(1)と(2)は，上で述べた第一の場合，(3)と(4)は第二の場合に対応するために必要と考えられる．

(1) 複数の語にまたがる検索機能

複雑なデータ構造を扱う場合，それを機能メモリの1ワードで表現できることは少なく，一つのデータオブジェクトを複数のワードで表現せざるをえないことが多い．そのようなオブジェクトを検索する場合，当然1回の検索ではなく，一般的に複数回の検索が必要である．仮にNワードからなるオブジェクトを検索するとすれば，N回の検索を繰り返して行う．ただしその場合，それぞれの検索結果は，その一つ前の検索結果が真の場合のみ有効となる．つまり前回の結果との論理積をとりつつ，複数ワードにまたがる検索を進めていく機能が必要である．

(2) 部分書き込み機能

機能メモリは貴重な計算資源であり，通常のメモリのように生のデータをそのまま保持したりする利用形態は今のところ考えにくい．さまざまな情報圧縮やコード化などを通じ，1ビットでも有効に利用しようとするのが現状である．その場合ワード，あるいはバイト単位でしか書き込みが行えないようだと，余分なビット操作（シフトやマスク）やメモリアクセスが増え，効率的な処理が実現できない．

(3) 結果保持のためのタグビット

ある条件に基づいて検索を行うことは，処理データ全体の中からある部分集合を取り出すことに相当する．通常の計算機システムでの検索は，逐次的に進み，かつ，その検索条件を満たすデータも逐一アクセスされる．そして検索されたデータがリスト，あるいは配列といった形で，もとのデータ構造とは別個な形で集合として保持され，あとの処理に備えられる．

これに対し，機能メモリでは検索は常にオーダ1であることに加え，さまざまな検索結果をそのワードに付属するタグビット，あるいはそのワードの一部に保持することが可能であり，後述の並列書き込み機能があれば，この過程自体もオーダ1で実現することができる．

このようなタグビットは，それ自体が一つの部分集合を示しており，さらにこれらのタグビット自体を検索条件とすることで，より複雑なパターン検索がワード並列で行うことができる．つまり，各ワードにおいて静的なデータ表現部に加えて，応用プログラムにより動的に書き換えられるタグビット部を設けることは，PE におけるレジスタ的な役割を実現する．タグビットは機能メモリの各ワードにおいて1ビットレジスタとして設けられてもよいが，前述の部分書き込み機能があれば，各ワード内の任意のビットをタグビットとして利用することができて使いやすい．

(4) 検索結果に基づく並列書き込み機能

検索結果を機能メモリ内にタグビットの形で保持する場合，検索してヒットしたワードに対してのみ，並列に書き込みを行える機能は処理時間の低減にきわめて効果的である．機能メモリに保持されるデータ数をNとすればオーダNからオーダ1への低減である．したがって大容量データほど，この効果は大きいが，たとえデータ数が数千であっても十分である．この並列書き込み機能のない機能メモリは，人工知能処理用としてはかなり制約が大きく不適当と考えられる．

(5) 速 度

機能メモリに要求される速度は応用ごとに異なり，一概に論ずることはむずかしい．しかし単純な検索機能だけを考えると，現在の機能メモリは必ずしもソフトウェアによる検索と比べて圧倒的に有利とは言えない．最近のワークステーション

の進歩は早く，たとえば SUN 4/SS 2 においてC言語のハッシュ関数で検索を実行した場合，キーの長さにもよるがだいたい 10 マイクロ秒以上となる．

これに対し，機能メモリの場合，たとえビット並列の連想メモリでも，筆者の経験では CPU からの種々のオーバヘッドを含めると1桁以上の差をつけることはむずかしい．ただしこれは単純な検索の場合であり，検索条件が複雑になれば機能メモリの優位さは明らかである．また機能メモリは SIMD デバイスであるから，大容量データを扱うほど効果が顕著になる．算術演算ではデータ数が 1000 あれば通常の逐次計算機をしのげる．

また機能メモリはデータの初期設定時以外にも通常のメモリとしてアクセスされることが多い．このため RAM と機能メモリとのアクセス速度に大きな開きがあるのは好ましくない．

2.2 人工知能向き機能メモリの構成例

現在，人工知能処理向き機能メモリとして商品化，およびそれに準ずる段階にあるものとしては，NTT の 20 K bit CAM (Content Addressable Memory) と Coherent Research Inc.（アメリカ）の 9 K bit CAM がある．これらはいずれも前節で述べた機能メモリの要件を満たしている．これらに対し，たとえば AMD の CAM チップである Am 99 C 10 のように主に検索機能しかない機能メモリでは，人工知能処理向きとしては利用範囲がかなり狭まると考えられる．

（1） NTT 20 K bit CAM[11]

512 語 × 40 ビットの構成をもち，現在最大のビット数を有する連想メモリである．各語は32ビットのデータ部と8ビットのタグビット部からなる．27種の命令モードをもつ多機能な連想メモリであり，並列書き込みやマルチワード機能など，2.1 で述べた機能はすべて有する．特徴的なのはガーベージコレクションで，不要なワードの管理をハードウェアでサポートしている．また検索においては，各ワード，あるいは隣接するワード間で過去の検索結果と現在の検索結果の論理和，論理積をとることが可能であり，大小比較など，種々の算術論理演算アルゴリズムの実現に効果的である．詳細については，文献 13) に譲る．

この連想メモリを用いたシステムとしては，64台の連想プロセッサからなる電総研の並列連想プロセッサ IXM 2 がある．各連想プロセッサにおいて連想メモリはトランスピュータのアドレス空間にメモリマップされ，連想メモリの一処理を約400ナノ秒で実行する．IXM 2 全体で 256 KW × 40 ビットを有し，したがって応用によっては最大 26 万の超並列処理を実現できる．SUN 4 をホストとして動作する IXM 2 の外観を図-1 に示す[14]．

また NTT では同種のアーキテクチャをもつ 4 K ビット CAM も製作しており，これを用いたシステムに Prolog マシン ASCA がある[12),13)]．

（2） Coherent Research Inc. 9 K bit 連想メモリ[15]

同社の連想メモリは，シラキュース大学で研究された連想メモリをもとに商品化したものである．連想メモリチップは，3本の 36 ビットレジスタ（マスク，データ，ネバーマッチレジスタ），256 個の PE，そして複数のワードがヒットしたときの裁定回路 (Multiple Response Resolver ; MRR) よりなる．

PE は，実質的には 36 ビットの CAM と，5個の1ビットレジスタを含む論理部で構成される．各 PE の内容は図-2 に示すように，32 ビットのデータ部，4 ビットのタグビット部，三つの汎用

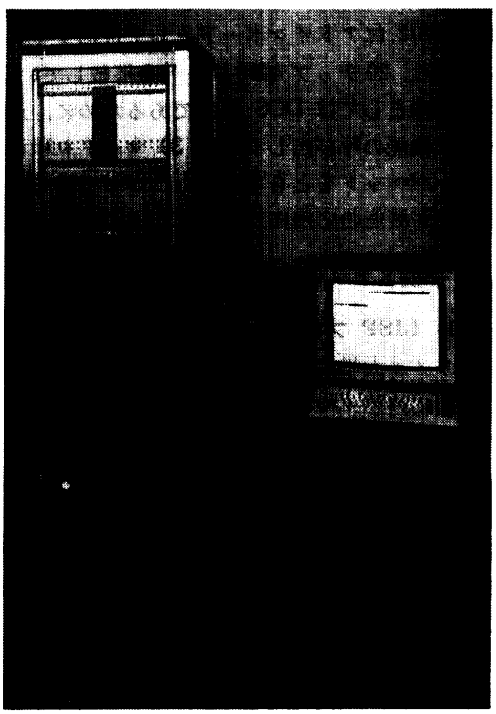


図-1 並列連想プロセッサ IXM 2

1ビットレジスタ (R 1, R 2, R 3), 演算処理部 (General Purpose Logic Block ; GPLB), 各語のヒット結果などを保持する1ビットレジスタ (MR), 列間でのデータ移動などに用いるシフトレジスタ (SR) からなる.

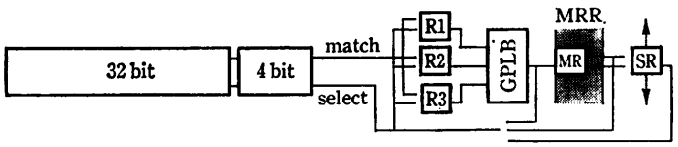


図-2 CRI 9K ビット CAM の1語の構成

汎用レジスタには検索結果を選択的に書き込むことができるとともに, GPLB を用いて, それらの3出力に対しての任意のブーリアン関数を実現することができる. またMRは複数ヒット時の裁定回路 MRR の一部を構成しており, 全体で 256 ビットのベクトルを形成する. MRR は, その 256 ビットのベクトルを対象に, その中のトップ番地にある語を選び出す. 加えて MRR は, 現在マッチがあるのかどうか, さらに一つ以上のマッチがあるのかどうかを示す信号も出力する.

なおこのメモリには, don't care 情報を表現するために quad モードというものがあり, その場合2ワードが連結されて, 論理的な1ワードを成す.

このCAMの動作モードの詳細は公開されていないが, **2.1** の要件はほぼ満たしている. 一動作あたり 50 ナノ秒で, 一回の検索に 150 ナノ秒を要する. この機能メモリを用いたシステムには, IBM PS/2 にアドインボードとして付加するタイプがあり, 20 チップ 4096 ワードを搭載している. 利用形態としては DOS 経由であるために, 連想メモリ機能の呼び出しごとに 30 マイクロ秒程度のオーバヘッドをともない, したがって実時間処理など高機能の応用には適さない.

3. 機能メモリの応用事例

3.1 LISP プログラムの高速化

イリノイ大学の LISP マシンでは, リスト表現の一部に連想メモリを用いることにより, リストアクセスの高速化, リスト表現の簡潔化, および LISP の基本処理操作の高速化を図っている[4]. そのリスト表現の基本アイデアは, (1)リストを論理的な2進木として表現する. (2)次にその論理表現の中で有用な情報だけを連想メモリ上に格納する, というものである.

- リスト表現の方法

((a (b c) d) e (f (g)) h) というリストを例に, 上述の論理的な2進木への変換, つづいて連想メモリ上への格納について説明する. このリストから論理的な木への変換は, 各ノードのアドレス, つまりノード番号の生成規則を次のように定めることにより行われる. つまり, あるノード n からの部分木のノード番号の決め方は, 左部分木が L $(n)=2n$, 右部分木が R $(n)=2n+1$ とする. リストのベースとなるノード番号は1から始まるので, LとRオペレータを用いて, 上記のリストは図-3 に示す論理的2進木へ変換される. たとえばルートのノード番号は1であり, したがって左の部分木のノードは2, 右は3となる. このようなアドレス表現の方法はすでに提案されておりそのメリットとして, アドレス生成が簡単になること (ノード番号 n が与えられれば, それをシフトして, さらに1を足すかどうかだけで部分木のアドレスが求まる), そしてメモリへのアクセス回数を減らせる (いちいち CAR 部のポインタをたどる間接的な参照がなくなる), などがあげられる.

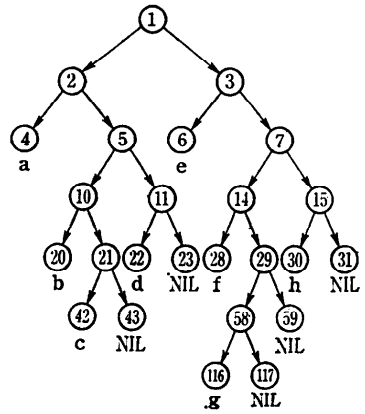


図-3 リストの2進木表現

しかし, もしこれをそのまま通常のリニアなメモリ空間に写像したのでは, 図-3 でのノード番号のふり方をみて分かるようにノード番号値はスパースであり, したがってメモリの利用効率が非常に悪くなる (このアプローチを提案した Berkling は最終的には断念した).

そこで図-3 の表現はあくまでも論理的なもの

とし，これから，実際にリストを構成するのに必要な情報のみを物理的メモリ（連想メモリ）に写像する方式をとっている．

つまり図-3 から分かるように，リストのアトムはその木の葉としてすべて現れており，それらはリストを構成するときに重要である．これに対し，ノード番号 5 や 7 といった中間ノードはポインタを保持するだけである．もちろん中間ノードはリストの構造を決定するのに不可欠ではあるものの，その情報についてはL，Rオペレータの定義によって補うことが可能であり，したがって中間ノードについての情報をメモリに保持しておく必要はなくなる．つまりアトムである葉 (leaf) の値とノード番号の二つの情報をもっていればこのリストの再構成（図-3 の論理表現）はつねに可能である．

このような葉のノード，あるいは他のリストからもさされているノードなどを例外ノードと呼ぶが，これら例外ノードの情報を含む表を例外テーブル (exception table ; ET) として連想メモリ上に置く．たとえば図-4 は図-3 の木に対する ET である．ET の各要素は {論理ノード名，例外の種類，値} の三つ組である．ET はリストごとに用意され，したがって一つのプログラムに複数の ET がある．LISP プログラムが扱うのはあくまでもS表現や図-3 のような論理表現であるが，実際の操作はより低位のプログラムが ET を操作することにより実現される．

Logical Node Name	Exception_nature	Value
4	ATOM node	a
20	ATOM node	b
42	ATOM node	c
43	NIL node	—
22	ATOM node	d
23	NIL node	—
6	ATOM node	e
28	ATOM node	f
116	ATOM node	g
117	NIL node	—
59	NIL node	—
30	ATOM node	h
31	NIL node	—

図-4 例外テーブル

- ET を用いたリスト処理

LISP プロセッサの生成するアドレスは，{木の論理番号，論理ノード番号} の組からなる．前者は，プログラム内に複数存在する木のどれか，つまり結局は ET を指定する番号である．後者はそのリスト内での論理ノード番号である．たとえば図-3 の木の番号が 7 とすると，a を表すノードは，{7, 4} となる．

この表現と ET を用いて，LISP の基本操作は次のように実行できる．たとえば atom 関数の実行結果は，そのアーギュメントとなるノードが，7 番の ET の中に登録されていて，かつ例外の種類が ATOM_node, NIL_node のときに真となる．この際の，ET に登録されているかどうかのチェックは連想メモリ上に ET が置かれているので効率良く行える．

car, cdr 操作は，解として {木の論理番号，論理ノード番号} の形でリストへのポインタを返す．たとえば図-3 で car (7, 5) の答えは (7, 10) となる．このときの操作では，(7, 5) について ET の登録があるかどうかを連想メモリ検索により調べている．もし登録があるのであれば (7, 5) は例外ノードであり，アトムがNILである可能性が高いので，そういうものに対して car 操作はそもそも成り立たない．この場合 (7, 5) についての ET の登録がないので，(7, 5) の 5 に対してR オペレータを実行して car 部のノード番号を得，(7, 10) が答となる．同様に cdr (7, 5) は (7, 11) となる．

これらの例から分かるように，このイリノイ大の方式の第一の利点はリストアクセスの高速化である．つまり部分木のノード番号が現在のノード番号にL，Rオペレータを施すことで簡単に計算でき，しかも，このとき通常のリストセルの実現のときのようなポインタによるメモリへの間接参照をやらなくて済む．第二の利点はリスト表現の簡潔さである．メモリ使用量でいえば，従来提案されている cdr コード化が通常方式の 2 分の 1 になるところ，この方式では 4 分の 1 になるとしている．

機能メモリを用いた LISP の高速化研究としては，ほかにマサチューセッツ大学の研究がある．リアルタイム処理を LISP プログラムで実現しようとする場合，ガーベージコレクションの発生が

ネックとなる．Bonar の提案はガーベージコレクションの高速化を目的としたもので，各リストセルを連想メモリのワードに割り当て，各ワードごとにガーベージビットを設けるというものである．新たにセルが必要になったときには，ガーベージビットの立っているワードを検索し，その中から選べばいいので不要なワードを別個に登録し管理しなくても済む．ただし高速化というメリットはあるものの，循環リストを扱えない，仮想メモリシステムに向かない，などの欠点がある．なお，この方式に基づくパイロットシステムが320×256 バイトの連想メモリを用いて試作されている[8]．

3.2 部分マッチング

連想メモリは機能的に完全なマッチングを基本とするが，応用によっては部分マッチングが必要なことも多い．たとえば，検索対象としておのおのが複数のフィールドからなるレコードの集合があるとする．このときレコード全体ではなく，いくつかのフィールドの値のみを検索データとして指定することで，それらの値を一部に含むようなレコードを見つけるのは部分マッチングである．この処理はもちろん通常の連想メモリを用いてもできるが，何通りかの組合せで検索データを用意し検索を繰り返さなくてはならないため，効率が悪い．このため連想メモリ上でのデータ表現を工夫する必要が出てくる．本節では部分マッチングの有効な方法として，スーパインポーズド・コード (Superimposed Code) をとり上げる[2]．

スーパインポーズド・コードはハッシュを拡張したもので，一つのレコードの各フィールド値をそれぞれハッシュして固定長のビットストリングに変換後，そのレコード内のすべてのフィールド値に対応するビットストリングの論理的 OR をとったものをスーパインポーズド・コード語 (Superimposed Code Word；以下 SCW) として生成する．

たとえば，あるレコードRが，二つのフィールド値 (john, lawyer) からなっており，これらのフィールド値をコード化して 10 ビットの SCW を生成する場合を考える．またビットストリング長 k を $k=10$，各フィールドに対するハッシュ回数 i を $i=4$ とし，得られるハッシュ値は 0 から $k-1$，つまり 0 から 9 の間であるとする．

まず john についてハッシュを 4 回行うが，得られるハッシュ値が {1, 4, 7, 8} とする．これらの値に対応するビットに**図-5** (a)のように 1 をセットする．次に lawyer に対するハッシュを 4 回行い，{1, 3, 6, 8} が得られたとすると，これらのビット位置に 1 をセットするとともに，前回の john に対して得られたビット表現との論理的 OR をとる．その結果，**図-5** (b)のビット表現が得られ，これがこのレコードRについての SCW となる．

このような SCW が複数あるとして，部分マッチングは次のように行われる．(1)特定のいくつかのキーを含むレコードを見つけ出したいとするとまず部分マッチの問い合わせの中で指定されるそれらのキーに対して上と同一のハッシュを行い，得られるビット表現の論理的 OR をとる．これを問い合わせコード語 (Query Code Word, QCW) と呼ぶ．たとえばいま問い合わせのキーは一つだけで，それが john だとすると，ハッシュの結果得られる QCW は図-5 (a)と同一のものとなる．(2)次に QCW の中で 1 となっているビットと，それらのビットに対応するビットがすべて 1 であるような SCW を見つける．それらが部分マッチングの解となる．たとえば，この QCW (図-5 (a)) の中で 1 が立つビットはすべて図-5 (b)の SCW と一致するので，レコードRが部分マッチングの解の一つとなる．

このように問い合わせに対していったん QCW を作り，そしてすべての SCW が連想メモリ上に展開されているとすれば，あとは連想メモリに対する 1 回の検索操作で部分マッチングを実現することができる．

ただし，SCW の問題点は偽のマッチングが起こり得ることである．もし tom という語が問い合わせのキーで，これに対するハッシュにより {1, 4, 6, 8} という QCW が得られたとすると，これは図-5 (b)の SCW を満たすので，この問い合わ

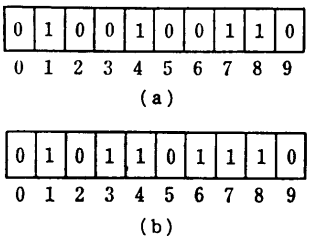


図-5 スーパインポーズド・コード

せは真となる．ところがレコードRにはtomは含まれていないので，これは間違いである．しかしながら偽マッチはSCWへのコード化の方法をうまく選ぶことで十分に小さくできる．

SCWの応用としてまずあげられるのがデータベースに対する問い合わせの高速化であり，ベル研究所では通常のRAMをもとに連想メモリモジュールを構成し，その上にSCWを置くシステムを試作している[3]．しかしSCWはデータベースに限らず，他の応用に対しても部分マッチングによる情報検索技法として有効である．その例としてプロダクションシステムを次節で述べる．

3.3 プロダクションシステム

エキスパートシステム構築用の基本メカニズムとして広く用いられるプロダクションシステムは，ルールを格納するプロダクションメモリ，ルールによって参照・更新されるデータを格納するワーキングメモリ，およびこれらを制御するインタプリタからなる．ルールは，手続き型言語における IF 文に似た“前提条件→動作”の形をとる．またワーキングメモリの各データはワーキングメモリエレメント (Working Memory Element; WME) と呼ばれる．

プロダクションシステムの実行は 1. 照合 (matching), 2. 競合解消 (conflict resolution), 3. 実行の三つのサイクルからなる．最初の照合サイクルでは，ルールの前提条件部を満たすWMEを見つけるが，このルールとWMEの組合せを見つける操作を，すべてのルールと WME を対象に行う．ここで多数の組合せが見つかる可能性があり，そのうちの一つを競合解消サイクルで選択し，そのルールに指定された動作を実行サイクルで処理する．

照合サイクルにおいて，すべてのWMEとルールのマッチングを毎回調べるのは時間がかかる．もし1000個のプロダクションと100個の WME があれば，100,000 回のマッチングが各照合サイクルごとに必要になる．このようなことをさけるため，OPS システムなど逐次型計算機上でのプロダクションシステムでは Rete と呼ばれる高速照合アルゴリズムを採用している．

照合サイクルでの処理は機能メモリに適した応用の一つであり，ここでは IBM の Kogge によるフィルタリングと呼ぶ手法，そして Rete アルゴリズム自体の連想メモリによる高速化について述べる[7]．

3.3.1 フィルタリング

照合サイクルですべてのルールと WME のマッチングを調べるのではなく，あるルールの実行で一つの WME が生成されたら，その時点ですぐに，その WME を要素としているルール (群) に対して，その WME がルールを満たすかどうかをチェックする．この方法をここではフィルタリングと呼ぶ．

例としてモンキーバナナの問題，つまり猿が天井にぶらさがっているバナナをとるために，はしごを動かしたりする問題をとりあげる．

フィルタリングの実現のために，2種類の連想メモリを用意する．一つはワーキングメモリ用，もう一つはコード化したルールを格納するプロダクションメモリ用である．

まずプロダクションメモリの構成を説明する．**図-6** は，プロダクションメモリの一部とそこにコード化されるルール (ルール番号 11) を示したものである．ルール記述においてコロンの付く文字列はフィールド名，そのあとに続く文字列は定数，〈 〉付きの文字列 (たとえば〈place〉) は変数を表す．

このルールでは前提条件が4つの条件要素の AND として示されているが，その各条件要素ご

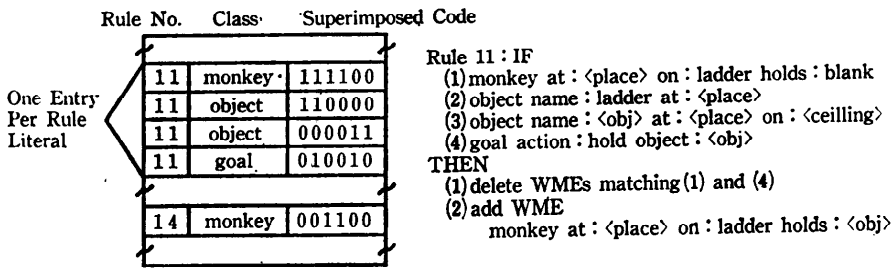


図-6 プロダクション規則とそのコード化表現

とにコード化して連想メモリの一語を割り当てる．したがってこのルール 11 は，プロダクションメモリにおいて4語を占めている．

その一語の内容は，1. ルール番号，2. 照合テストを行うべきルールのクラス，そして 3. そのクラスの WME が他のフィールドにもつべき値（たとえば一番目の条件要素の中にある，on フィールドの ladder という値）を **3.2** で述べたスーパインポーズド・コードにより，一つのフィールドにコード化したもの，からなる．

このように構成されたプロダクションメモリがあり，いま新しく WME が生成されたとすると，照合サイクルは次のように三つのステップで実行される．

第一に，その新しい WME のすべてのフィールドの値をそれぞれハッシュしてからそれらの論理 OR をとって，スーパインポーズド・コード語 (SCW) を作る．第二に，これを満たすルール条件要素があるかどうかをみるために，プロダクションメモリに対して検索を行い，もしヒットするルールがあればそれらのルール番号を記録する．たとえばいま次のような WME が生成されたとすると，

monkey at : T 7-7 on : ladder holds : blank

これに対して SCW が作られ，さらにこれを用いて図-6 のプロダクションメモリを検索する．仮にルール 11 と 14 がヒットしたとすれば，番号 11 と 14 を記録する．第三に，選ばれた各ルールについて，それぞれの条件要素を満たす WME があるかどうかを，連想メモリ上のワーキングメモリに対して検索を行って調べる．もしすべての条件要素が満たされればそのルールは実行可能となる．

たとえば，ルール 11 の場合を考える．上記の monkey についての新しい WME がルール 11 の最初の条件要素である，'monkey at : ⟨place⟩ on : ladder holds : blank' にマッチしたため，その中の ⟨place⟩ 変数に T7-7 をバインドする．次に2番目の条件要素が満たされるかどうかのテストを行う．つまり ⟨place⟩ 変数を T7-7 に置き換えた，'object name : ladder at : T7-7' という WME がワーキングメモリにあるかどうかを検索する．もしなければルール 11 は失敗である．もしあれば，3 番目の条件要素，さらに4番目の条件要素のテストへと進む．この方式の効果についての予測では，連想メモリの使用により約 100 倍の高速化が可能としている．

3.3.2 Rete アルゴリズムの高速化

OPS システム[16]では，プロダクションの前提条件部をコンパイルして Rete ネット[17]と呼ぶネットワークデータ構造を作り，そこに照合サイクルの中間結果を保持することにより，照合サイクルでのマッチング処理量の軽減をはかっている．Rete アルゴリズムは並列処理に向いており，マルチプロセッサによる実現の報告もあるが，ここでは連想メモリによる実現について述べる．

Rete ネットでは，実行サイクルの結果，新しい WME が追加されると，その WME が Rete ネットワークに入力され，ネットワークの更新が行われる．Rete ネットワークは2種類のノード，アルファノードとベータノードから構成される．アルファノードでは，入ってきた WME のもつ属性値についてのテストを行い，テストが成功したときのみ，その WME を次のノードへ送る．ベータノードでは，入ってきた WME とすでにベータノード内にある WME との属性値のテスト（2入力テストと呼ぶ）を行い，成功した WME の組みのみを次のノードへ送る．いくつかのテストを経て Rete ネットワークの終端に達すると，その WME の組は終端に対応したプロダクションとともにコンフリクトセットに追加される．

このような Rete ネットワークの動作の様子を図-6 を例に示す．**図-7** は，図-6 のモンキーバナナ問題のルール 11 に対する Rete ネットワーク表

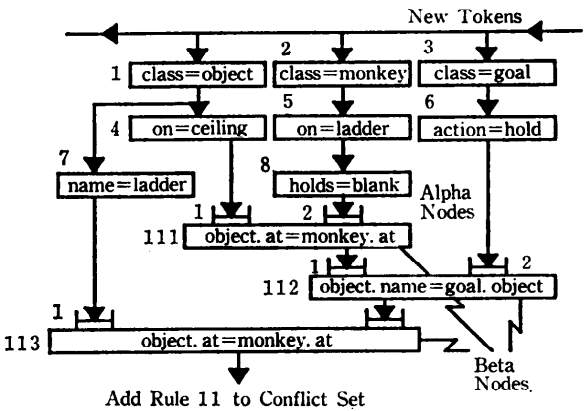


図-7 Rete ネットワーク

現である.

いまクラスが monkey である次のような WME が新たに生成され,

monkey at : T 7-7 on : ladder holds : blank

が, Rete ネットに入ったとする. まずクラスが monkey であるかどうかのテストがアルファノードの2番で行われ, もしこのテストに通れば次のノードのアルファノード5番に送られ, on フィールドが ladder であるかどうかのテストが行われる. もしこれに通れば8番のアルファノードで holds フィールドのテストとなる. さらにこれに通れば次はベータノード2番でのテストである. ベータノードは2入力でそれぞれキューをもつ. ここでは, いま入った monkey の WME が, もう一方の入力から入ってキューにたまっている, クラスが object である WME の at フィールドの値と比較される. この比較の回数は, そのときにもう一方の入力キューにいくつ WME がたまっているかによって決まる. 文献18) によれば, ベータノードに到着した WME は平均的に 20 から 25 の WME (そのベータノードのもう一方のキューにたまっている) と比較され, また一つの WME は約 35 のベータノードでのテストを経るという. したがって一つのWMEに対して約700回以上の比較操作が行われることになる.

連想メモリの導入は, このような比較操作回数を低減できるとともに, WME を Rete ネットワークから消去する操作においてきわめて効果的である. 以下に示す Kogge の提案はまだ初歩的な段階にあり稼働するには至っていないものの, 上の二つのメリットを実現するアイデアとして面白い. そこでは, アルファノード用とベータノード用の2種類の連想メモリを用意する.

アルファノード用連想メモリの各エントリは, アルファノードにおいてテストされる WME のクラス, そのテストで使われるフィールドの値, および他のフィールド用の don't care よりなる. また各エントリには, そのアルファノードでのテストをパスしたあとに行くべきベータノードのキュー番号が付く. たとえば図-8 は図-7 の Rete ネットに対応するアルファノード用連想メモリとベータノード用連想メモリの内容の一部を示したものである. アルファノード用メモリの最初のエントリである, "1111 object X ceiling X" は, 図-7 のアルファノード1, 4番に対応しており, まず WME のクラスが object でなくてはならないこと, またそこでのテストで使われる値が ceiling であること, そしてこのテストが成功したら, ベータノード111番の1番目 (エントリには 1111 とある) の入力キューに行くべきことが示されている. このような表現をとることにより複数回のアルファノードのテストが一回の連想メモリ検索で済み, 比較操作の回数が低減されることが分かる. 新しい WME が Rete ネットに入ってきたら, このアルファノード用連想メモリを検索すればよく, 送るべきベータノードの番号も同時に分かる.

ベータノード用連想メモリでは, 全ベータノードのキューにたまっているすべての WME を登録する. もし新しい WME がアルファノードでのテストをパスしたらそのコピーがとられ, キュー番号とともにベータノード用メモリへ格納さ

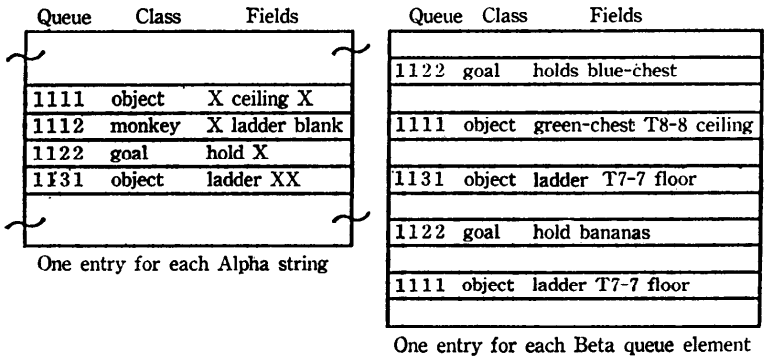


図-8 アルファ, ベータメモリ用連想メモリの構成

れる．そしてそのキュー番号に対応するベータノードの2入力テストが行われる．すなわち，新しいWMEのフィールド値と，そのベータノードのもう一つの入力キューにあるWMEのしかるべきフィールド値とが比較される．

たとえば次のWMEはアルファノード2，5，8番目のテストを経て，

monkey at: T 7-7 on: ladder holds: blank

が，ベータノード111番の2番目のキューに送られるので，これのコピーがとられベータノード用メモリに新たに登録される．図-8のベータノード用メモリの状態はこの直前の状態である．また111番のベータノードのもう一つのキューには，"1111 object greenchest T 8-8 ceiling" と "1111 object ladder T 7-7 floor" の二つがすでにあることが分かる．そしてこれら二つのWMEのatフィールド値と，新しいWME（上記のクラスmonkey）のatフィールド値が一致するかが連想メモリに対する検索処理によりテストされる．この場合は111番ノードの1番目のキューの二つのWMEのうち後者のWMEしかマッチしないので，これと新しいWMEとの組がつくられ，次のベータノード112番へと送られる．

上であげた平均的な一つのWMEあたりの比較回数のデータについて言えば，このような2種類の連想メモリを設けることにより，ベータノードでは比較が25から1回に減ることになる．

また比較操作回数の低減に加えて，実行サイクルにともなって発生するWMEの除去処理が速くなるメリットも大きい．つまり，逐次計算機上でのReteアルゴリズムではこの除去処理に全体の約半分の処理時間がかかるといわれており，連想メモリの使用により，かかる時間がほぼコンスタントにまで低減できる．この二つの速度の向上化要因により，逐次計算機に比べ約50倍の向上が見込めるとしている．

3.4 意味ネットワーク

意味ネットワークは図式的な知識表現の形式であり，知識ベース記述，自然言語処理などに広く用いられている．並列連想プロセッサIXM 2は意味ネットワークの処理を主目的に開発されたマシンであり，連想メモリのもつ超並列性を集合演算や，1対多の同時データ転送に生かしている[14]．ここではそこでの意味ネットワークの連想メモリ上での表現とその効果について述べる．

- 連想メモリ上でのデータ表現

意味ネットワークはノードとリンクからなるネットワーク構造をとる．図-9は5つのノード（ノードAからE）からなる意味ネットワークをIXM 2上で表現した例である．IXM 2での情報の表現はノード単位であり，一つのノードについての情報は連想メモリと通常のメモリ上に展開している．まず連想メモリにおく情報は，マーカビット（28ビット），接続リンク（8ビット），リテラル（16ビット），並列マーカ伝搬情報（22ビット）の4つの領域からなる．これら1ノードに関する情報を表現するのに連想メモリの4ワード（1ワード40ビット）を用いている．次に通常のメモリ上には，そのノードがつながる相手先のノード名や並列マーカ伝搬情報が置かれている．

たとえばノードCは次のように図-9に表現されている．このノードはノードAとisaリンクでつながっているため，ノードCについての連想メモリ上の接続リンク情報中のisaリンクの存在を示すビットに1が立っている．また通常のメモリ上にはノードCの接続先情報としてisaの欄にAと書かれている．

このような表現形式に基づき，IXM 2では最大

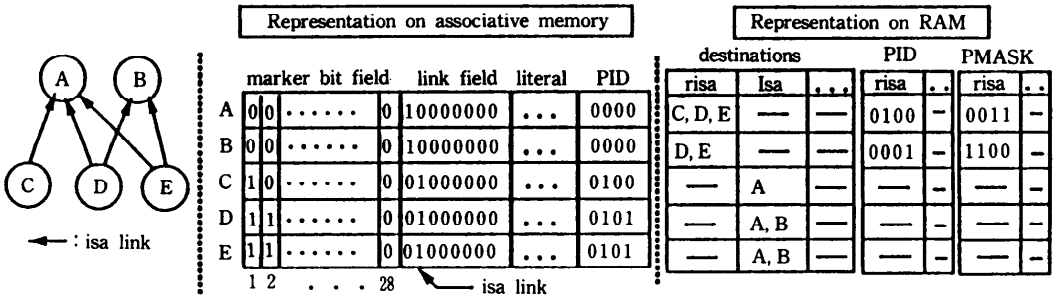


図-9　意味ネットワークのデータ表現

6万4千ノードを並列処理することができる．またネットワークのノードやリンクの種類はユーザが自由に定義でき，表現上の制約はない．

連想メモリ上のマーカビットとは1ビットのフラグであり，そのノードについての中間処理結果などを保持する．またマーカ伝搬とは主に集合の要素を求める際に行う処理で，簡単にいえばノード間でのメッセージ交信である．たとえば図-9でノードAにノードC，D，Eが集合の要素として属するとしたら，ノードAからC，D，Eにそれぞれ特定のマーカビット番号を伝搬し，各ノードにそのマーカビットを立てさせる．この操作をマーカ伝搬と呼び，これによりノードAの集合要素がそのマーカ番号により識別できる．

- 連想メモリ上での並列マーカ伝搬と集合演算

意味ネットワークの処理では，連想，集合演算，マーカ伝搬が基本的で，かつ使用頻度が高い．そしてこれらの処理に連想メモリは効果的で，大規模ネットワークになるほど顕著となる．以下では，マーカ伝搬と集合演算について，連想メモリ上での並列処理とその効果について述べる．

マーカ伝搬では，ある一つの基点ノードから，それにつながる複数のノード（子孫ノードと呼ぶことにする）に同時にメッセージ（マーカビット）を送れる場合が多い．このようなケースを特に並列マーカ伝搬と呼ぶ．その際，逐次計算機，あるいはコネクションマシンによる実現ではそのノード数だけマーカ伝搬によるメッセージ交信を繰り返さねばならない．しかし連想メモリを用いることで，1対多数の同時メッセージ転送が可能である．連想メモリ上に置かれる上述の並列マーカ伝搬情報（PID）がその際に使われる．

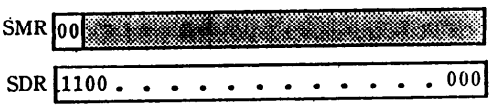
並列マーカ伝搬の基本的なアイデアは，(1)基点ノードに自分の子孫ノードを識別する情報(PID)を与え，また子孫ノードにもその PID を与えておいて，基点ノードがその PID を使って子孫ノードを検索する，(2)子孫ノードが複数ヒットしたところで並列書き込み操作を行い，特定のマーカ番号を子孫ノードに一気に書き込む，というものである．これにより，子孫ノードの数によらず一定期間で並列マーカ伝搬が行える．たとえば図-9でノードAが基点ノードとなって子孫のC，D，Eに並列マーカ伝搬を行おうという場合，ノードAは通常のメモリ上におかれている PID の '0100' と，検索時のマスクデータである PMASK '0011' を使って検索を行い，まずC，D，Eノードをヒットさせる．次にそれらに並列書き込みを行ってマーカ1番を一度にセットする．

IXM2の場合，並列マーカ伝搬は子孫ノードの数によらず常に 35 マイクロ秒である．これに対し，SUN 4/330 では一回のマーカ伝搬（あるノードから別のノードへの1本のリンク上の転送）につき 20 マイクロ秒かかる．子孫ノードの数が N なら $20*N$ マイクロ秒かかる．またコネクションマシン CM-2 では1ノード1PE の割り付け時で，やはり1リンクあたりで約1ミリ秒かかる．CM-2 がこのように遅いのは PE 間がシリアルリンクによって転送されるためである．

子孫ノードの数が多いほど並列マーカ伝搬は効果的である．たとえば現在 IXM2 で行っている実時間機械翻訳の実験で扱う音素ネットワークでは一つのノードから平均で約 40 の子孫ノードがあり，特にその効果が大きい[10]．なお PID などの並列マーカ伝搬情報は，処理に先立って意味ネットワークアロケータが，基点ノードと子孫ノードのペアを認識し生成する．

集合演算では二つの集合の交差 (intersection) を求めることが多い．たとえば図-9 の意味ネットで，集合 A と集合 B の交差はノード D と E である．このような二つの集合の交差を求める場合，逐次計算機では片方の集合（データ数 N）をハッシュすることでオーダ N の処理時間がかかるが，マーカビット表現を連想メモリ上で利用することにより，オーダ1で済む．

たとえば図-9 のマーカビット領域で，マーカビット1が立っているのは集合 A の要素 (C, D, E)，マーカビット2番が立っているのは集合 B の要素 (D, E) である．したがって交差を求めるにはマーカビット1番と2番が共に立っているノードを探せばよく，これは検索マスクレジスタ，検索データレジスタを**図-10** のようにセットして一



SMR：検索マスクレジスタ
SDR：検索データレジスタ

図-10 検索データ／マスクレジスタの表現（交差演算）

回検索を行うだけで済む．その後ヒットしたノードにおける特定のマーカビット（交差を示す）に対し並列書き込みを行い，これも一回で済む．

表-1に示すように連想メモリだけでなくSIMDマシンであれば集合サイズにかかわらずオーダ1の処理が可能であるが，逐次計算機では集合サイズの増加により実行時間も増える．特に集合が64K程度になると，スーパコンピュータでもIXM2と3桁の速度差が生じてくることが示されている．交差の処理は知識ベース処理をはじめAIではよく用いられる処理の一つであるが，大容量連想メモリの効果を示すいい例である．また同じSIMDでもIXM2とCM-2の間に差があるのは，ビット並列と直列の違いからである．

なお算術論理演算ではたとえば32ビットデータに対する大小比較演算時間として36マイクロ秒という数値が得られている．これはビット直列アルゴリズムによるため一見遅い．しかし機能メモリ内の全ワードに対して並列に行えるので，1ワードあたりの処理時間は0.56ナノ秒ときわめて高速となる（データ数256K）．

3.5 遺伝的アルゴリズムに基づくルール学習システム

遺伝的アルゴリズム（genetic algorithm；以下GA）は，自然界の適応メカニズム（淘汰と遺伝）にヒントを得た検索アルゴリズムであり，たとえば10^{30}といった巨大な検索空間をもつ問題に対しても比較的短時間で満足できる解を見つけだせるものとして近年注目を急速に集めつつある[15]．

GAではまず解の候補を0や1からなるビットストリングに変え，これを複数集めたものを初期集団として設定する（このおのおののストリングが染色体，各ストリング内の0や1がそれぞれ遺伝子に対比される）．各ストリングにはそれぞれ目的関数による評価値（fitness）が付随し，評価値の高いストリング同士をペアとし，これらからクロスオーバと呼ぶ操作により，さらに評価値の高いストリングをつくり出す．

たとえば二つのストリング，'01000' と '11001' があるとき，ランダムに選んだ位置（たとえば3ビット目と4ビット目の間）を境界として，その前半部同士をそっくり入れ換えて，'11000'，'01001' という新しいストリングをつくる操作である．このような操作を何世代か繰り返して評価値のより高いストリングをつくり，また評価値の低いストリングを消去していくうちに初期集団が次第に評価値の高いストリング，つまり満足できる解を多く含むように変化していく．

GAは種々の組合せ最適化問題に対して適用されているが，最近ではルール型学習システムへの応用が，機械による学習（machine learning）の新しいパラダイムとして注目されている．このルール学習システムはプロダクションシステムに近いものであるが，違いはプロダクションルール集合自体がGAにより進化，つまりより性能の高いルール集合へと学習していく点にある．つまり評価値のよいルール同士が組み合わされてクロスオーバにより，さらに性能の高いルールへと変化する．逆に役に立っていないルールは淘汰され，別のルールに置き換えられる．

GAに基づくルール学習システムには，ミシガンアプローチとピッツアプローチの二つがある．前者はGAの操作対象となるストリングが，一つのIf-thenルールに対応するものである．後者は，ストリングが複数のルールの集合に対応するもので，一つのストリングが一つのプログラムに相当する．

ミシガンアプローチに基づくルール型学習システムはクラシファイアシステム（classifier system）と呼ばれ，その実現例の一つにBOOLEというブーリアン関数の学習システムがある[6]．

ブーリアン関数として，たとえば6マルチプレクサと呼ばれる問題では，関数が次の形をとる．

$$F = a_0'a_1'd_0 + a_0a_1'd_1 + a_0'a_1d_2 + a_0a_1d_3$$

つまり4つのデータ入力ビット（d_0からd_3）があり，そのうちの一つが二つのアドレス入力指定（a_0とa_1）で選ばれてFとして出力する．BOOLEではGAを使ってルール集合を進化させることにより，この関数の機能をルール集合に学習させるのが目的である．つまり入力値のいろいろなパターン（データ，アドレスの計6ビット）を繰り

表-1　交差演算の実行時間

（単位：マイクロ秒）

set size	1000	10000	64000
IXM2	18	18	18
CM2	103	103	103
Cray X-MP	1829	7372	39823
SUN-4/330	28998	44332	142827

返し提示し，そのたびごとに真偽を判定し，それをフィードバックさせることにより，その関数の挙動を再現できるように学習させる．

具体的にはまず各ルールを，条件，行動，強さの三つ組で表す．6マルチプレクサでは条件はアドレスとデータの6ビット，行動はその条件に対する関数値（F），強さはそのルールの有効度を示す．そして図-11のように初期集団として複数のルールを集めたものをルールベースとして用意する（この段階では，この中に関数Fを正しく反映しないルールも含まれている）．

次に入力パターン（上記の6ビット）を繰り返し提示し，それにマッチする条件をもつルールが正しい行動（関数値）をもつ場合は，そのルールの強さを増加させ，もし行動の値が偽であるときはその強さを減じる．この場合，強いルールほど有効なルールとみなせる．これらを何回か繰り返した時点で，今度は強いルール同士からGAを用いて新しいルールを作り出し，有効でないルールと置き換える．

このようにしてGAを繰り返し使っていく過程で，初期のルール集合から次第にFを反映する正しいルール集合に変化させることができ，これが学習に対応する．

BOOLEは逐次計算機上で実現され，ニューラルネットによる学習よりもブーリアン関数の学習が速いことが示されている．シラキュース大学ではさらに高速な処理を目指して連想メモリを用いたクラシファイアシステムACSを構築し，その上でBOOLEの実験を行っている[1]．

if-then ルール

条件		行動	強さ
0 1 # 0 # #	/	0	7655
0 1 # 1 # #	/	1	7541
0 0 0 # # #	/	0	7056
1 0 # # 0 #	/	0	7095
0 0 1 # # #	/	1	6665
1 # # # # 0	/	0	212
1 0 # # 0 1	/	1	56
0 1 # # # #	/	1	116

図-11　BOOLEにおけるルール例（6マルチプレクサ）

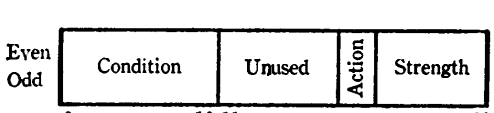


図-12　ACSでのルール表現

ACSでは図-12のようにルールを連想メモリ上で表現する．論理的には1ルール32ビットであるが，一つのビットが条件部においてdon't careを含むため，2.で述べた連想メモリチップのquadモードを用い，1ルールの表現に2ワードを費やしている．

連想メモリに対する主な操作は次の二つである．第一に入力パターンにマッチするルールを検索する．これに要する時間は当然ルール数の影響を受けず一定である．第二に入力マッチで選ばれたルールの中から，強さをもとにクロスオーバのための二つのルール（親）を選ぶために，ルールに付随する強さの最大値，最小値を求める．強さのフィールドのビット幅をnとすると，この操作はオーダnであり，これもまたルール数に影響されない．このようにACSでは検索だけでなく，数値演算にも連想メモリを用いているのが特徴である．

逐次計算機上での処理では全体の処理時間がルールの増加に比例するが，ACSでは連想メモリの使用によりルール数が10倍に増えても，実行時間の増加は25%程度に留まることが示されている．

なおACSはミシガンアプローチであるが，ピッツアプローチのルール学習を機能メモリで行うシステムとしてIXM2を用いたGA-1システムがあり，マッチング速度ではCM-2より1桁以上速いことが示されている[9]．

4. おわりに

機能メモリの人工知能への応用としては前章で述べたもののほかに論理プログラムの高速化があり，NTT，シラキュース大など活発に研究が行われているが，本特集では文献13)で述べられているため，本稿では割愛した．

機能メモリはアーキテクチャ的に分類するとCM-2やMasParなどの超並列SIMDマシンに近いところにあるとみることができる．ただし，機能メモリをシステム的にみた場合，データ交信とプログラミングに若干問題がある．つまり，機能メモリの各ワード間の並列な情報転送の方法としては基本的に隣接するワード間に限られるため，データ交信の並列性や柔軟性という点で制限があるのは否めない．また，機能メモリの機能自

体にも制限があるため，プログラミングの自由度も限られることがある．

しかしながら，応用対象の性質を見極め，そしてそこでのデータ表現手法に十分注意を払えば，3. の応用事例で示したように，ずっと安価に，かつコンパクトに超並列システムを構築することができる．

超並列デバイスとしての機能メモリは，少数の命令セットを有する一種の RISC チップともみなせる．今後の課題としては，このようなチップに対してどのような機能と構成を与えるべきかを設計者の勘だけではなく，種々の応用事例を通して“定量的に”検討し明らかにすることであると考えられる．現状ではまだその方向での蓄積が十分ではない．またその検討にあたっては，機能メモリの使用形態が今後もなんらかの CPU の制御下にあると想定すれば，機能メモリだけではなく，CPU を含めた視点が不可欠である．つまり現在 CPU で行っている処理の機能メモリへの負荷分散や，機能メモリの制御に適した CPU の新たな命令セットの提案などが必要になると考えられる．

参 考 文 献

1) Twardowski, K. E.: Implementation of a Genetic Algorithm Based Associative Classifier System, IEEE Intl. Conf. on Tools for AI (Nov. 1990).
2) Roberts, C. S.: Partial-match Retrieval via the Method of Superimposed Codes, Proc. of the IEEE, Vol. 67, No. 12 (Dec. 1979).
3) Ahuja, S. R. and Roberts, C. S.: An Associative/parallel Processor for Partial Match Retrieval Using Superimposed Codes, Intl. Symp. on Computer Architecture (1980).
4) Sohi, G. S., Davidson, E. S. and Patel, J. H.: An Efficient Lisp-execution Architecture with a New Representation for List Structures, Intl. Symp. on Computer Architecture (1985).
5) Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley (1989).
6) Wilson, S. W.: Classifier Systems and the Animat Problem, Machine Learning, Vol. 2, pp. 199-228 (1987).
7) Kogge, P. M. and Oldfield, J. V.: VLSI and Rule-based Systems, CASE Center Tech. Rep., No. 8812, Syracuse University (1988).
8) Bonar, J. G. and Levitan, S. P.: Real-time Lisp Using Content Addressable Memory, Intl. Conf. on Parallel Processing (1981).
9) Kitano, H., Smith, S. F. and Higuchi, T.: GA-1: A Parallel Associative Processor for Rule Learning with Genetic Algorithms, Intl. Conf. on Genetic Algorithms (July 1991).
10) Kitano, H. and Higuchi, T.: High Performance Memory-based Translation on IXM 2 Massively Parallel Associative Memory Processor, AAAI-91 (July 1991).
11) Ogura, T. et al.: A 20 Kb CMOS Associative Memory LSI for Artificial Intelligence Machines, IEEE J. Solid-state Circuits, Vol. SC-24, No. 4, pp. 1014-1020.
12) Naganuma, J. et al.: High-speed CAM Based Architecture for a Prolog Machine (ASCA), IEEE Trans. Comput., Vol. 37, No. 11, pp. 1375-1383 (Nov. 1988).
13) 長沼次郎，小倉 武：機能メモリを用いた計算機アーキテクチャ，本特集．
14) Higuchi, T., Furuya, T., Handa, K., Takahashi, N., Nishiyama, H. and Kokubu, A.: IXM 2: A Parallel Associative Processor, Intl. Symp. on Computer Architecture (1991).
15) Coherent Research Inc.: Associative Processing Products (1990).
16) Forgy, C. L.: OPS5 user's manual, CMU Tech. Report, CMU-CS-81-135 (1981).
17) Forgy, C. L.: Rete: A Fast Algorithm for Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence 19(1) (1982).
18) Gupta, A. and Forgy, C. L.: Measurements on Production Systems, CMU Tech. Report, CMU-CS-83-167 (1983).

(平成3年7月15日受付)

樋口 哲也（正会員）

昭和30年生．1982年慶應義塾大学大学院工学研究科博士課程修了．電子技術総合研究所入所後，人工知能処理向き並列計算機システムの研究・開発に従事．電子情報通信学会，AAAI 各会員．

古谷 立美（正会員）

昭和22年生．昭和48年成蹊大学大学院修士課程修了．同年電子技術総合研究所入所．現在同所計算機構研究室長．工学博士．IEEE 会員．