(7)

EC83−30

高並列推論エンジンＰＩＥの単一化プロセッサと縮退アルゴリズム

湯原雅信・相田　仁
後藤厚宏・田中英彦
元岡　達
（東　大）

１９８３年１０月２７日

社団法人　電子通信学会

高並列推論エンジンＰＩＥの単一化プロセッサと縮退アルゴリズム

Unify Processor and its Reduction Algorithm of the Highly Parallel Inference Engine — PIE

湯原　雅信，相田　仁，後藤　厚宏，田中　英彦，元岡　達

M. Yuhara, H. Aida, A. Goto, H. Tanaka, T. Moto-oka

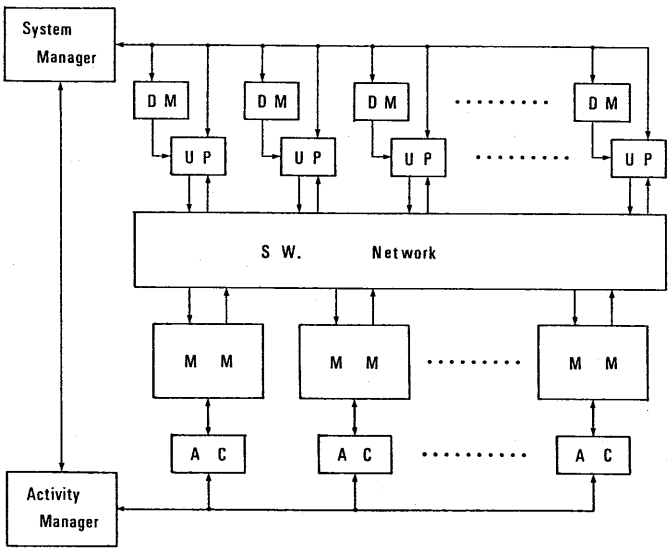
東京大学　工学部

Faculty of Engineering, University of TOKYO

１．はじめに

我々は、Prolog をはじめとする論理型言語を、直接かつOR並列に実行する高並列推論エンジンＰＩＥ（Parallel Inference Engine）の開発を進めている［1，6］。ＰＩＥの基本処理要素である単一化プロセッサは、ＰＩＥの処理能力を上げるために、その機能に適したハードウェアを持たなければならない。

ここでは、はじめに２章で単一化プロセッサの概要について述べ、続く３章・４章では単一化プロセッサが行なう単一化・縮退アルゴリズムについて説明する。５章では、現在設計・試作を進めている単一化プロセッサについて述べる。



UP：Unify Processor
DM：Definition Memory
MM：Memory Module
AC：Activity Controller

図１　ＰＩＥ第１次モデル

２．単一化プロセッサ［4］

２．１　ＰＩＥにおける単一化プロセッサ

ＰＩＥでの実行単位は、中間結果としてのゴール節である。この中間ゴール節は、それまでの単一化（Unification）の結果を含み、互いに独立している。これを、ゴールフレーム（以下ＧＦ）と呼ぶ。一方、定義節にいくつかの情報を加えたものを定義節テンプレート（以下ＤＴ）と言う。ＰＩＥ第１次モデル（図１）の単一化プロセッサUP（Unify Processor）は、MM（Memory Module）からＧＦを一個受けとると、その中の一つのゴールリテラルについて単一化可能性のあるすべてのＤＴをDM（Definition Memory）から取り出し、その各々とＧＦとの間で単一化を行なう。この時DMは、ＵＰからの情報を用いて簡単な照合操作を行ない、単一化の候補となるＤＴを絞り込む。単一化が成功したものについて、ＵＰは縮退操作（Reduction）を行ない不要な情報を取り除いて新しいＧＦを生成する。新ＧＦはMMに送られ、ＡＣ（Activity Controller）による実行制御を受けた後、再びＵＰへ転送される。ＰＩＥでは、論理型言語のもつ非決定性から生じる並列性を、多数のＵＰを置くことにより自然な形で利用して処理を進めることができる。

２．２　単一化プロセッサの機能

単一化プロセッサには次のような機能がある。

（１）単一化

ＧＦとＤＴの単一化を行なう。単一化の進行に伴ない変数が束縛される。

（２）縮退

ＧＦ間のOR並列処理を実現するには、ＧＦの一部を共有して保持する方式と、各ＧＦが別々に持つ方式とがある。ＰＩＥでは後者の方式をとっている。この方式ではＧＦが独立しているため、並列処理上問題になる共有資源へのアクセスがないという大きな利点がある。これに対し、コピー

のオーバーヘッドや環境を冗長に持つことによる記憶能率の悪さ等の欠点もある。

そこで、不要な情報を取り除き、GFの大きさを最小限にして、記憶コストや転送コストを押える操作を導入した。これが縮退である。PIEにおける並列処理では、定義節のすべての選択肢を別々に適用し、後戻りをしない。このため、逐次型ではスタックの上に残しておかなければならない環境（変数の束縛状態等）も、PIEではかなりの部分が不要となり、縮退の効果は大きい。

（3）システム述語の実行［5］

ユーザが pure Prolog の範囲で記述できない述語や、効率化の必要な述語については、組込述語としてシステムが提供する。入出力等に関するシステム述語を実行する場合には、System Managerと協力する必要がある。また、実行制御に関するシステム述語の場合は、ACに対してその指示を送る。

（4）優先度スコアの計算［3，7］

a．ゴールフレーム選択用

資源の管理や処理効率等を考慮して、MM中に多数存在するGFから適切なGFをACが選択する。UPは、この選択のための評価値を与える。

b．ゴールリテラル選択用

1つのGFの中で、次にどのリテラルを実行すれば効率的であるかを評価関数に従って計算する。

2．3　単一化プロセッサの構成方法

単一化プロセッサ内での単一化に関する並列性には次のようなものが考えられる。

① リテラル引数間の並列性

リテラル内の複数の引数を、各々並列に単一化することが可能である。本並列性は、単一化がその順序によらず、同じ結果をもたらす性質に基づいている。しかし、引数間の関係がANDであるため、リテラル内で共有されている論理変数の取り扱いが難しい。また、処理の単位が小さいため、制御コストが大きくなる恐れがある。

② 定義節テンプレート間の並列性

選択されたゴールリテラルに対応する定義節が複数存在する場合、各々について独立に単一化・縮退操作を実行できる。相互の関係がORであるため、比較的容易に実現できる。

③ パイプライン処理

入出力バッファを設ければ、GF単位のパイプラインが実現可能である。すなわち、UPが単一化／縮退操作を進めている間に、次のGFを受け取り定義節の絞り込みを行なうことができる。

このようなUP内並列性をうまく抽出し利用することができれば、UP内の処理を高速化できる。しかし、実現にはいくつかの問題点がある。また、PIEにおけるUPは、単一化を行なうのみでなく、新しいGFを再構成する際の縮退操作も行なう。処理の負荷としては、後者の方が大きい。従って、単一化のみを高速化してもシステム全体の性能は、あまり上がらない。そこでPIE第1次モデルのUPとしてはとりあえずGF単位のパイプラインだけを考えることにする。

2．4　PIE第1次モデルの単一化プロセッサ

PIE第1次モデルのUPの内部は図2のようになる。

- Unifier：　Local　Memory 中のGFとDT間の単一化を実行する。
- Reducer：　成功した単一化の結果を用いて、縮退操作を行なう。
- System Predicate　Processor：
  システム述語を処理する。
- Scorer　：　優先度スコアを計算する。
- Local　Memory　：　GF用とDT用とがある。単一化を開始する前に、GFは入力バッファからGF用　Local Memory へコピーされる。DTはDMからDT用Local Memory へコピーされる。UPでの単一化・縮退の実行時には、両Local　Memory 内に、変数の束縛状態を記憶するための変数部（variable area ）が設けられる。
- Input　Buffer　：　入力バッファはMMから取り出したGFを一時的にバッファリングする。
- Output Buffer　：　縮退の結果できる新GFは、出力バッファに書かれる。でき上がったGFはMMに送り返される。

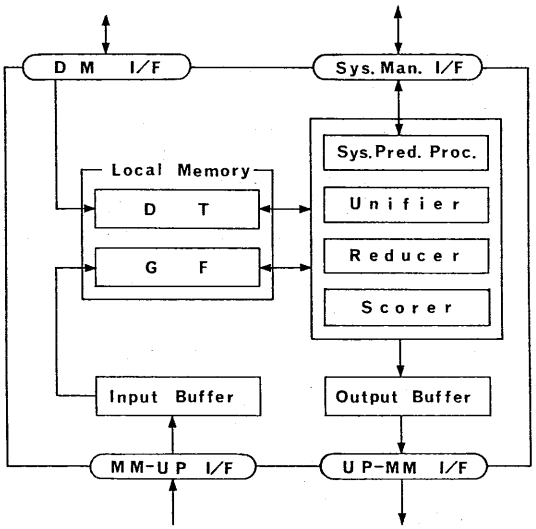
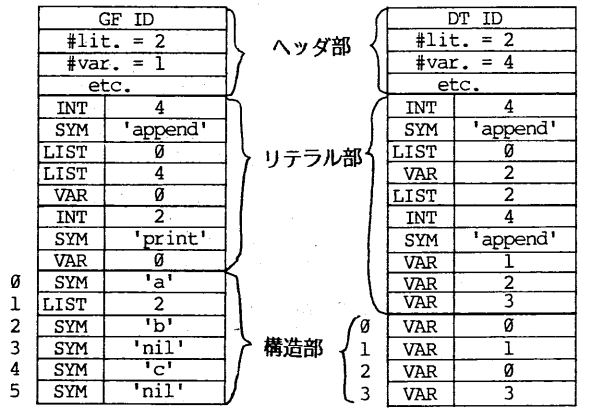


図2　単一化プロセッサ　UP

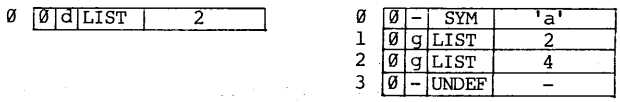
2．5　GF／DTの内部表現

DTや実行中のGF等は、PIEの内部表現で表される。内部表現は、タグ付きの固定長セル（40ビット）の一次元配列で、ヘッダ部（ header area ）、リテラル部（ literal

area）、構造部（ structure area ）から成る。UP外でのGFのすべての変数の値は未定義であることが保証されている（4章参照）。GF、DTの内部表現の例を図3に示す。



（a） GF　　（b） DT



（c） 単一化成功後の変数部

図3 GF・DTの内部表現

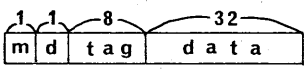


図5 セルのビット構成

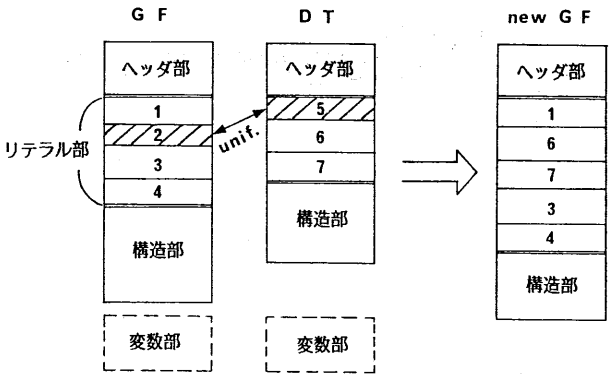


図6 縮退操作

3． 単一化アルゴリズム

Unifier の実行する単一化のハードウェア・アルゴリズムを図4に示す。

Local Memory 内において、GFおよびDTの各セルには、図5のようにdビット（definition bit）とmビット（mark bit）が付加される。dビットは構造体や変数がGF側のものであるかDT側のものであるかを区別するのに用いる。ただし、単一化ではmビットは使用しない。

Local Memory のリテラル部・構造部・変数部へのアクセスは、アドレスとして与えられるセルのタグの内容により、そのいずれかの部分が指される。

本アルゴリズムでは、単一化を行なうGF中のゴールリテラルとDTの頭部リテラル間で、すべての引数について次の①、②の繰り返す。

① セルの fetch

g_addr、 d_addrが指すセルを、それぞれメモリから g_cell, d_cellに取り出す。取り出したセルがVAR型である間は、その変数番号（とdビット）の示すセルを変数部から取り出し続ける。これを“変数のたぐり”と呼ぶ。

② fetch したセルに従った処理

a． fetch した2つのセルのいずれかが未定義変数（UNDEF）であれば相手側セルを変数部に書き込む。

b． いずれのセルもUNDEFでない場合は同一のセルであるかどうかを調べる。両セルが同じ型の構造体であった場合には、単一化を再帰的に行なう。

図3のGF，DTの単一化が成功した後の変数部を同図（c）に示す。

4． 縮退アルゴリズム

4．1 縮退操作

Reducer における縮退操作では、単一化の成功したGFとDTから必要な部分をコピーしながら新しいGFを作り上げて行く（図6）。図6ではGFの2番目のリテラルとDTの頭部リテラルとの単一化が成功し、その結果が変数部に残っている。縮退が行なわれると、単一化の対象になったゴールリテラルの位置にDTのホディー部が埋め込まれる。新GFの構造部はもとのGFとDTの構造部から必要なものだけを取り出し再構成される。新GF中の変数はすべて未定義なので、新GFには変数部を設ける必要がない。

```
typedef {
  BIT mark;  /* mark bit */
  BIT def;   /* definition  bit */
  TAG tag;   /* data type */
  DATA data; /* some value */
} CELL;

struct LOCAL_MEMORY {
  CELL literal[];   /* terminated by END */
  CELL structure[]; /* terminated by END */
  CELL variable[];
} GF_MEM,DT_MEM;

CELL g_addr, g_cell, .....;
CELL d_addr, d_cell, .....;
int  length;
       ......

unification()  {
  /*           Initialization
   * g_addr points the first cell of selected
   * goal literal in GF_MEM.  d_addr points
   * the first cell of definition head literal
   * in DT_MEM.
   * All variables in variable area are UNDEF.
   * Stack is cleared.
   * It is Assumed that predicate name and
   * arity of the DT copied to DT_MEM are the
   * same as those of selected goal literal.
   */

  length := load( g_addr ).data - 1;
  g_addr.data += 2;     /* skip arity and  */
  d_addr.data += 2;     /* predicate name. */

LOOP_UNIF:
  if( length > 0 ) goto FETCH_UNIF;
  if( stack_empty() )  return( SUCCESS );
  else pop_unif()

FETCH_UNIF:
  fetch_unif();
  length--;
  g_addr.data++;
  d_addr.data++;

  if( g_mem == d_mem )  goto LOOP_UNIF;
  case( tag_type_pair( g_cell, d_cell ) ) {
      <UNDEF,UNDEF>:
            store( g_mem, d_mem );;

      <UNDEF,not UNDEF>:
            store( g_mem, d_cell );;

      <not UNDEF,UNDEF>:
            store( d_mem, g_cell );;

      <LIST,LIST>:
            if( length > 0 )
                  push_unif();
            length := 2;
            g_addr := g_cell;
            d_addr := d_cell;
            goto FETCH_UNIF;;

      <VECT,VECT>:
            if( length > 0 )
                  push_unif();
            g_addr := g_cell;
            d_addr := d_cell;
            g_cell := load( g_addr );
            d_cell := load( d_addr );
            if( g_cell.tag != INT ||
                d_cell.tag != INT )
                  return( FAIL );
            if( g_cell.data !=
                        d_cell.data )
                  return( FAIL );
            length := g_cell.data;;
            g_addr.data++;
            d_addr.dara++;

      <ATOM,ATOM>:
            if( g_cell.tag !=
                        d_cell.tag )
                  return( FAIL );
            if( g_cell.data !=
                        d_cell.data )
                  return( FAIL );;

      otherwise:
            return( FAIL );;

  }
  goto LOOP_UNIF;
}

fetch_unif()
  /*
   * Fetches to g_cell and d_cell the cells
   * referred by g_addr and d_addr,
   * respectively. g_mem and d_mem get the
   * last address accessed.
   */

store( addr, src )
  /*  src is written to the address
      pointed by addr */

load( addr )
  /*  returns the cell pointed by addr */

push_unif()
  /* g_addr, length,and d_addr are pushed
     to the stack. */

pop_unif()
  /* g_addr, length,and d_addr are popped
     from the stack. */
```

図4　単一化アルゴリズム

```
reduction(){
  /*    Initialization
   * g_addr points the first address of literal
   * area of GF. o_lit, o_str become the first
   * address of literal and structure area of
   * new GF, respectively. o_varx becomes marked
   * VAR-type cell whose data field is zero.
   */
  goto NEXT_LITERAL;

CHECK_LENGTH_LITERAL:
  o_lit.data++;
  if( length > 0 )  goto FETCH_RED_LITERAL;

NEXT_LITERAL:
  g_cell := load( g_addr );
  g_addr.data++;
  if( tag_type( g_cell ) == END )  {
  if( stack_empty() )  return( END_REDUCTION );
      else {
          pop_red();
          goto LOAD_ARITY;
      }
  } else if( g_cell.mark == MARK ) {
         /* Unified goal literal is skipped */
      g_addr.data += g_cell.data;
      g_cell := load( g_addr );
      push_red();
      g_addr := /* pointer to the first cell
                * of the head literal of DT */
      g_cell := load( g_addr );
      g_addr.data += g_cell.data + 1;
      goto LOAD_ARITY;
  } goto SAVE_ARITY;

LOAD_ARITY:
  g_cell := load( g_addr );
  g_addr.data++;
SAVE_ARITY:
  store( o_lit, g_cell );
  length := g_cell.data;
  o_lit.data++;

FETCH_RED_LITERAL:
  fetch_red();
CASE_JUMP_LITERAL:
  g_addr.data++;
  length--;
  case( tag_type( g_cell ) ) {
      <ATOM> or <VAR>:
        store( o_lit, g_cell );;

      <VECT>:   t_cell := load( g_cell );
        if( t_cell.mark == MARK ) {
            store( o_lit, t_cell );
        } else {
            push_red();
            g_addr := g_cell;
            g_cell.data := o_strx.data;
            g_cell.mark := MARK;
            store( g_addr, g_cell );
            store( o_lit,  g_cell );
RED_VECT:
            length := t_cell.data;
            o_str := o_strx;
            store( o_str, t_cell );
            g_addr.data++;
            o_str.data++;
            o_strx.data += length + 1;
            goto FETCH_STRCTURE;
        };;

      <LIST>:   /* omitted */

      <UNDEF>:g_cell := o_varx;
        o_varx.data++;
        store( g_mem, g_cell );
        store( o_lit, g_cell );;
  }
  goto CHECK_LENGTH_LITERAL;


CHECK_LENGTH_STRUCTURE:
  if( length == 0 ) {
      pop_red();
      if( addr_area( g_addr ) == LITERAL )
          goto CHECK_LENGTH_LITERAL;
  }
  o_str.data++;
FETCH_STRCTURE:
  fetch_red();
CASE_JUMP_STRUCTURE:
  g_addr.data++;
  length--;
  case( tag_type( g_cell ) ) {
      /*
       *  Almost the same as CASE_JUMP_LITERAL,
       *  except that o_lit is changed to o_str,
       *  and push_red is executed only if
       *  length > 0.
       */
  }
  goto CHECK_LENGTH_STRUCTURE;
}

fetch_red()
  /*
   * Fetches to g_cell referred by g_addr.
   * Fetch ends when g_cell.tag gets something
   * other than UNDEF or g_cell.mark is marked.
   * G_mem will be the last fetched address.
   * By fetch_red, unnecessary variable-links
   * and instantiated variables are removed.
   */

push_red()
  /* g_addr, length, and o_str are
     pushed to the stack. */

pop_red()
  /* g_addr, length, and o_str are
     popped from the stack. */
```

図7　縮退アルゴリズム

縮退操作をまとめると次のようになる。

① 参照されなくなったセルの除去
構造部中のセルは、新しいGFのリテラル部からアクセス可能なものだけ順次コピーして行く。重複したコピーを避けるために、すでにコピーした構造体については先頭セルのmビット（mark bit）を立て、かつそのセルの内容がコピー先を指すように書き換える。
② 値の定まった変数の除去
変数はたぐってから処理することにより、中継変数が除かれる。たぐった値がinstantiate されていれば、その値について縮退操作を行なう。
③ 変数番号の付け直し
変数をたぐった結果がUNDEFならば新しい変数番号を割り当てる。UNDEFだった変数部内のセルのmビットを立て、新しい変数を指すように内容を変更する。
①の部分は、圧縮移動型ガーベジコレクション［9］に類似している。②と③の部分はこのアルゴリズムに固有のもので、次節で述べるように①のアルゴリズムと融合できる。

4．2 縮退のアルゴリズム

縮退アルゴリズムの特徴を以下に示す。
① 新GFを結合網上にまとめて送り出せるように、GFのリテラル部、構造部がそれぞれ連続した領域に構成される。
② 1パスである。
③ 縮退の手間は、おおよそ新GFのセル数に比例する。
④ 不必要な push を行なわないので、再帰用に必要なスタック容量はLIST型ならばリストの入れ子の深さに比例する。（ tail recursion の最適化）
⑤ cyclicな構造体の縮退も正常に停止する。

縮退のアルゴリズムを図7に示す。図中で o_strx は次に割り当てる構造部の位置を、 o_varx は次に割り当てる変数番号を表わしている。 g_addr の指すセルを縮退して、 o_lit （または o_str ）の指す新GF中のセルに出力する。以下で、リテラル部の1つのセルに関する縮退（図7におけるFETCH_RED_LITERAL以降）がどのように行なわれるかを、例を挙げながら説明する。

a． アトムに束縛された変数の縮退
図8で、 fetch_red を始めると、まず注目セル（A）がマークされていない変数なので、たぐってみるとセルBに達する。しかし、セルBもマークされていない変数なので、さらにたぐってセルCを得る。今度はVAR型ではないので fetch_red が終了する。セルCがアトムなので、そのままコピーするだけでよい。

b． 未定義変数の縮退
図9で、セルA、セルBが同じ変数を指している。セルAからたぐって行くとUNDEFである。そこで、 o_varx を新しいセルの内容とし、同じ内容のセルにマークビットを立てたものをUNDEFだったセルに書き込んでおく。 o_varxの変数番号は1増やす。
その後、セルBの縮退をする時に、同じ変数を見に行くと、今度はマークされているので、セルの内容をそのままコピーする。

c． VECT型の縮退
図10のように、fetch したセルがVECT型の場合には、ベクタの本体の先頭セル（B）がマークされているかを調べる。マークされていれば、そのセルをそのままコピーする。図の場合は、マークされていないので、新GFの構造部に必要なセルを確保し、そこを指すVECT型のセルをつくる。セルBには、コピー先を残してマークしておく。

5． 単一化プロセッサの試作［8］

5．1 試作UPの全体構成

Unifier, Reducer に重点を置いてUPの試作を進めている。ハードウェアでサポートするUNIREDと呼ぶ部分は、図2のうち、Unifier／Reducer、Local Memory 、I／O Buffer である。他の部分は、SVP（Service Processor）がソフトウェアでエミュレートする。
SVPは 68000をCPUとしたマイクロプロセッサシステムでUPの一部のエミュレータとして次のような働きを担う。
① UNIREDへのGF、DTの転送
② UNIREDの動作開始の指示
③ 組込述語の実行
④ 単一化の結果（成功／失敗）を受けとる。
⑤ 単一化が成功した場合に新GFを取り出す。
その他、マイクロプログラムのUNIREDへのロードや、UNIREDの動作に関する統計情報収集も行なう。さらに、次のようなデバッグ機能を持つ。
① UNIRED内のメモリ、レジスタの内容の読み書き。
② UNIREDのマイクロステップ動作。

5．2 試作UPの特徴

① タグ・アーキテクチャ
データ型の区別をハードウェアで行なうことのできるタグ・アーキテクチャとした。
② ゴール側と定義側の独立した内部バス
GF用Local Memory とDT用Local Memory に同時にアクセスできるように内部バスを分けた。
③ 垂直／水平折衷型マイクロプログラム制御
開発期間を短縮するためにマイクロプログラム制御を採用した。

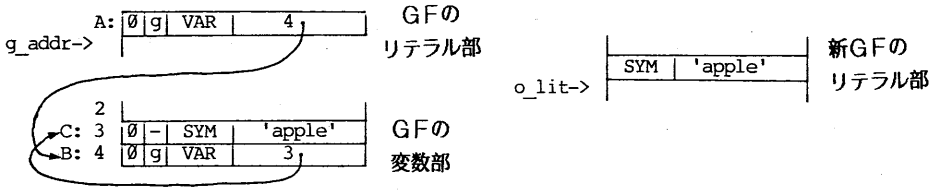
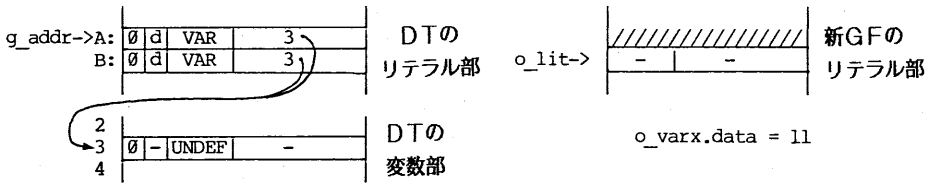
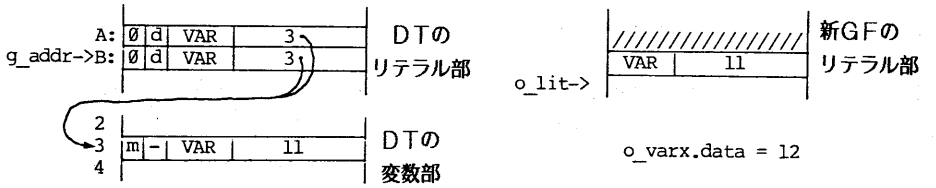


図8　アトムに束縛された変数の縮退

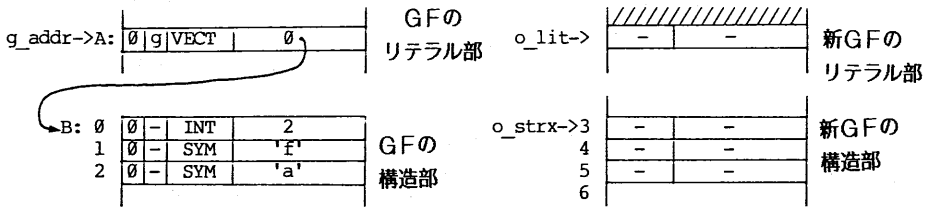


（1）　セルAの縮退前

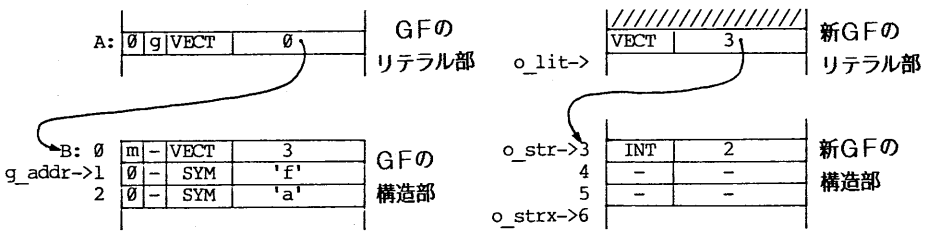


（2）　セルAの縮退後

図9　未定義変数の縮退



（1）　縮退前



（2）　縮退後

図10　VECT型の縮退

④ ハードウェアによる“変数のたぐり”

fetch を高速に実行するために変数のたぐりはハードウェアで行う。fetch はひとつのマイクロ命令になる。

⑤ インクリメント等の可能なレジスタ

3、4章のアルゴリズムでわかるように、ポインタの＋1や引数の残りの数の－1がしばしば行なわれる。そこで、6個のレジスタをカウンタで構成し、マイクロ命令の実行と並行して複数のインクリメント／デクリメントを行なえるようにした。

⑥ 2個のレジスタの内容による多重分岐

単一化や縮退時のタグの値による case 分岐を1マイクロサイクルで行なえるようにした。特に、単一化でfetch したセルがともにアトムの場合には、単一化の成功失敗も同時に判断する。

⑦ レジスタに付属し、同時にpush／pop されるハードウェアスタック3個

⑧ 高速なローカルメモリ

試作では、Local Memory のリテラル部・構造部・変数部は、一つのメモリを区分して実現する。単一化や縮退の主要部分において、Local Memory 、I／O Buffer へのアクセスが実行命令の半分近くを占めるので、これらに高速のメモリを使用し、1マイクロサイクルでアクセスできるようにした。

5．3 試作UPのハードウェア構成

試作UP（UNIRED部分）はデータ処理部とシーケンサおよびSVPとのインターフェイスからなる。

（1） データ処理部

データ処理部のブロックダイヤグラムを図11に示す。

① セルバス

セルバスは、レジスタ相互間やレジスタ～Local Memory 間のセルの転送に用いる。

ゴール側と定義側のセルバスをそれぞれ

GC（Goal Cell ）バス

DC（Definition Cell ）バス

と呼んでいる。

GC・DCバスのビット構成を図12に示す。

GCバスとDCバスの間にはゲートがあり、GC→DC又はDC→GCにセルバスを結合させることができる。このゲートにより、あるレジスタの内容を、GCバス上のレジスタとDCバス上のレジスタに同時に書いたり、GF用 Local Memory とOutput Buffer に同時に書いたりすることが可能となる。

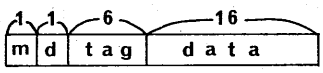


図12 GC・DC バス

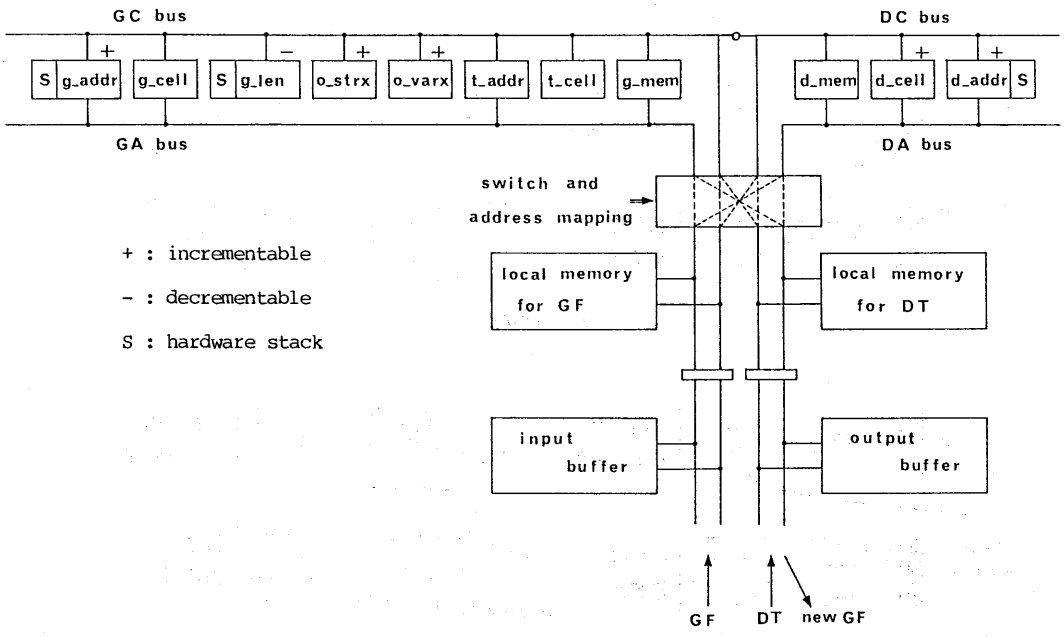


図11 試作UP（UNIRED データ処理部）のハードウェア構成

② アドレスバス

アドレスバスは、Local Memory 又は、入力バッファにアクセスする際の論理アドレスを与えるのに使われる。アドレスバスも

GA (Goal Address) バス

DA (Definition Address) バス

に分かれている。

アドレスバスのビット構成は、セルバスの<データ>を12ビットにしたものである。アドレスバスに接続されていないレジスタもある。論理アドレスは、<タグ>で定まる領域の先頭アドレスに、<データ>を加えることにより、物理アドレスとなる (図13) 。

アドレスバスのdビットは、ゴール側／定義側のどちらであるかを指示し、mビットはローカルメモリ／入出力バッファのどちらであるかを指示する。

アドレスバス上のタグ	指される領域
VECT／LIST	構造部
VAR	変数部
その他	リテラル部

図13 アドレス・マッピング

③ レジスタ

レジスタはセルバスと同じ24ビット幅 (図12) で、ゴール側に 8個、定義側に 3個ある。レジスタは、3・4章のアルゴリズムで使用した同名の変数に相当する。但し、 length は g_len の<データ>で表わし、 o_str , o _lit の役目は d_addr , d_cell に兼ねさせている。

各レジスタの機能は次のとおりである。

ゴール側レジスタ

g_addr : ・インクリメント可能
・スタック付属
・レジスタの<データ>にGCバスの<データ>を加算可能
・<タグ>がリテラル部を指していることによる条件分岐がある。

g_cell : ・mビットをセットしつつ<データ>をGCバスから取り込むことができる。
・マイクロ命令の多重分岐の条件となる。
・mビットによる条件分岐がある。

g_len : ・<データ>部分のみのレジスタ
・GCバスの<タグ>下位 4ビットを零拡張してレジスタの<データ>に取り込むことができる。
・デクリメント可能
・スタック付属
・<データ>部がゼロであることによる条件分岐がある。

g_mem : ・変数のたぐりの際、GAバスから最後にアクセスした時の (論理) アドレスが自動的にこのレジスタに入る。

o_strx : ・インクリメント可能
・レジスタの<データ>にGCバスの<データ>を加算可能

o_varx : ・インクリメント可能

t_addr : ・ (特殊機能はない)

t_cell : ・mビットが条件分岐の入力になっている。

定義側レジスタ

d_addr : ・インクリメント可能
・スタック付属
・多重分岐の入力となる。

d_cell : ・インクリメント可能

d_mem : ・ g_mem と同様

上記のスタックの容量が足りなくなった時は、スタックの中身をSVPが一時的に吸い上げる。

④ Local Memory

Local Memory は、GF用のものとDT用のものと2つあり両者に同時にアクセスすることができる。これまでのソフトウェアシミュレーション結果 [7, 8] から、Local Memory としてUNIREDの評価に十分な容量を次のように用意する。

リテラル部： 1Kセル (3KB)
構 造 部： 2Kセル (6KB)
変 数 部： 1Kセル (3KB)

レジスタ～Local Memory 間の転送は1マイクロサイクルで行なえる。

⑤ Input／Output Buffer

Input Buffer 及びOutput Buffer もLocal Memory と同じ容量を持つ。但し、変数部は使用されない。レジスタ・I／O Buffer 間の転送も1マイクロサイクルで行なえる。I／O Buffer は、UNIREDからアクセスされない状態にあればUNIREDから分離されているので、SVPから読み／書きすることができる。

（2） シーケンサ

分岐命令以外のマイクロ命令の実行時には、実行と並行して、マイクロ命令レジスタ（ＭＩＲ）の次命令アドレス（ＮＭＩＡ）フィールドで指定される次命令を、制御記憶（ＷＣＳ）からから読み出しておく。次命令は当マイクロ命令サイクルの最後でＭＩＲにセットされる。

分岐命令には、ある条件を満たすか否かで分岐する単純条件分岐とレジスタの内容に基づく多重分岐がある。

マイクロ命令は基本的には１マイクロサイクルで完了する。但し、Local Memory 又はＩ／Ｏ Buffer に、ＧＦ側・ＤＴ側の両レジスタから同時アクセスする時に、同じ側の Local Memory 又はＩ／Ｏ Buffer が要求された場合には、２マイクロサイクル必要となる。さらに、“変数のたぐり”をハードウェアで実現するために、たぐり終るまで次のマイクロ命令に移らないような制御も行なわれる。

（3） ＳＶＰインターフェイス

Local Memory 、WCS (Writable Control Store) Ｉ／Ｏ Buffer 、レジスタ群は、ＳＶＰのマルチバスのアドレス空間にマップされていて、ＳＶＰからアクセスすることができる。また、Status Register を介してＵＮＩＲＥＤの状態をＳＶＰに知らせることができる。

５．４ マイクロプログラム

ＵＮＩＲＥＤの制御にはマイクロプログラム方式を用いている。マイクロ命令は、基本的に垂直型であるが、すべてのマイクロ命令にＮＭＩＡフィールドやレジスタのインクリメント／デクリメントのビットが含まれている（図14）。

図３の例で、単一化（図４）・縮退（図７）にかかるマイクロサイクル数を見積ったところ、ＧＦとＤＴのコピーが終ってから、単一化が終了するまで、約３５マイクロサイクルで、縮退は約１１０マイクロサイクルであった。

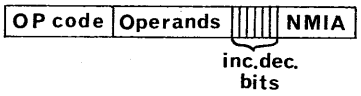


図１４ マイクロ命令形式

７． おわりに

本報告では、ＰＩＥ第１次モデルにおけるゴールフレームの内部表現、単一化・縮退方式について述べた。試作単一化プロセッサは、単一化・縮退に必要な機能をハードウェアでサポートする。

現在、試作単一化プロセッサの回路設計を進めており、今後、試作・評価を行なってゆく。

《参考文献》

［１］ 後藤 他，“推論向き高並列計算機システムの基本アーキテクチャ”，信学技報ＥＣ８２－４３（1982）

［２］ 後藤 他，“推論向き高並列計算機システムのアーキテクチャ”，第２６回情処全大，４Ｎ－４（1983）

［３］ 丸山 他，“推論向き高並列計算機システムのアクティビティ制御機構”，第２６回情処全大，４Ｎ－５（1983）

［４］ 湯原 他，“推論向き高並列計算機システムのユニフィケーション機構”，第２６回情処全大，４Ｎ－６（1983）

［５］ 相田 他，“推論向き高並列計算機システムの基本言語機能”，第２６回情処全大，４Ｎ－７（1983）

［６］ 後藤 他，“高並列推論エンジンＰＩＥについて”，The Logic Programming Conference ’８３，ＩＣＯＴ（1983）

［７］ 後藤 他，“高並列推論エンジンＰＩＥにおける並列処理の効率化手法について”，信学技報ＥＣ８３－９（1983）

［８］ 湯原 他，“高並列推論エンジンＰＩＥ ～単一化プロセッサの構成”，第２７回情処全大，４Ｐ－１２（1983）

［９］ Jacques Cohen, “Garbage Collection of Linked Data Structures”, Computing Surveys, Vol. １３, No.３, ACM (1981)

（盛文社）