FLATSにおける記号処理命令について

稲田信幸* 鈴木正享** 平木敬** 清水謙多郎** 中田哲雄* 相馬嵩* 後藤英一*,**
（*理化学研究所・情報科学研究室，** 東京大学・理学部・情報科学科）

0. はじめに

FLATSとは昭和54年度より理研で開発中の数式及び記号処理専用計算機システム[1]の総称であり，現在大型汎用計算機上で稼動している数式処理システムREDUCE-2[2]をFLATSに移植し高速に処理することを当面の目標とし，かつ将来はFLATS固有のハードウェアを意識したソフトウェアを整備拡充することにより更に高速化を目指す。これまでは通常の規模の数式の諸導は汎用機で処理されてきたが，数式が複雑かつ超大規模になると処理速度の他に大量の数式データを格納するために汎用機ではサポートされていない程の広いアドレス空間が必要となってくる。FLATSでは2^{29}語のデータ専用空間とcdr codingによる線型リストの圧縮によって，最高2^{25}Lispセルを扱うことが可能となっている。

現在稼動している汎用数式処理システム（REDUCE, MACSYMA, muMATH等）はホストとしてLispが用いられており，数式処理を高速に実行するためには高速なLisp処理系が望まれるのは至極当然である。FLATSでは頻繁に使用される関数及び基本Lisp primitive等は高速なハードウェア命令として直接実行され，かつデータ型も並列的にチェックされるのでソフトウェアによるLispシステムのボトルネックが解消され，処理速度はメモリ・アクセス・タイムで決定されるまで向上が期待される。

数式処理では数値計算とは異って実行時に使用する最大の主記憶量を予測することは一般に困難であるため，自由領域からLispセルをon demandで消費し，不用となった廃品（garbage）はゴミ集め処理（以下GBCと称す）を施すことにより再生使用する。ソフトウェア・Lispシステムの場合，自由領域の大きさが大きくなるとGBCの時間も無視できなくなってきている。FLATSではベクトルやハッシュ表等の連続領域もサポートしているので，従来のcdr chainによるGBCは使用せずに生成順序を保存しかつ領域を詰め替えるGBC[3]をハードウェア命令を備えて高速に行っている。

我々はFLATSを数式及び記号処理を専用に行なうback end processorとして位置付け，入出力・paging等はfront endのSVP（サービス・プロセサ）に任せている。

FLATSの特徴及び高速化手法には，

1). 高速連想処理を可能にする並列ハッシュ・ハードウェアの組込み。
2). Lisp基本操作及びpredicateの高速ハードウェア命令化。
3). 高速アクセスを可能とするため，命令キャシュ・データ・キャシュ別々に設けている。
4). GBC専用命令群（marking, pointer adjust, bit count, find bit等）。
5). 先行制御により GOTO, 条件分岐, CALL, RETURN の各命令実行時間の0化。
6). データ型の実行時並列チェック。
7). 3ポートread可能なGVキャシュを汎用レジスタ用に512組，ローカル・フレーム用に512組用意し，ひとつの関数内ではそれぞれ128組ずつ使用可能。
8). big numberソフトウェアのための命令群。
9) CPU命令の3アドレス指定による高速化。

10). on demand paging をはじめとする仮想記憶システムのハードウェア・サポート及びソフトウェアによる領域分割。
11). CPU・主記憶間のデータ転送巾は64 bit。
等が挙げられる。

1. FLATS データ表現

1.1. データ型

FLATS上で稼動するLispはユタ大学のStandard Lisp [4]に対して上位互換性をもたせてあるので図1のようなデータ型をbuilt inしている。FLATSで拡張されたデータ型には単一表現[5]の各H型とamt及びcatのハッシュ表類である。ソフトウェアによるLispシステムの場合，扱うデータ型が豊富になればなる程処理系の速度が遅くなる傾向にある。

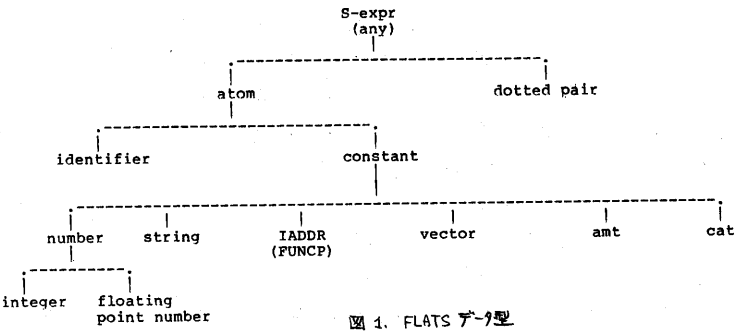


図1. FLATSデータ型

dotted-pairは実体がcdr codingされている外，通常Lispと同一である。identifierは専用の領域に4語ずつ割り当てられ，各語はそれぞれ束縛値，property list，関数定義及びprint nameのplace holderとして使用される。function pointerは命令空間のアドレスでsubr等の入口番地を示している。stringは後述のdescriptorを介して実体の文字列はbinary領域に置かれている。pnameやstringを直接扱う命令はないが，identifierの場合にはLDID, STID, XIDVの専用命令を有している。Standard Lispの特徴の中にアドレス計算により要素へアクセスするvectorが追加された点が挙げられる。FLATSではvector専用命令でレジスタ上に置かれたdescriptorを介してアクセスし，同時にrangeのチェックも行なわれる。短整数間の演算以外は割出し後ソフトウェアによるがデータ型の判別を行なう必要がないので高速処理が行なえる。amt (associative membership table)はKeyから構成されるハッシュ表で，cat (content addressed table)はKeyとvalueのpairから構成されるハッシュ表であり，ともにvector同様メモリ上のdescriptorを介して専用命令でアクセスされる。OBLIST (intern table)もcatを用いて表現することができ，INTERN操作，REMOB操作の高速化が実現できる。amt, catはKeyよりアドレス生成されアクセスされるのが通常の使用方法であるが，どのようなKeyが登録されているか等を調べるためにも有効要素を順次scanするSWEEP命令も備えている。

1.2. 領域分割及びアドレス空間

図2のようにFLATS全体の論理アドレス空間（アドレス変換の対象となるアドレス）として2^{25}語，命令空間2^{24}語及びデータ空間2^{24}語の仮想アドレスを扱える。データ空間の場合格納されるデータ型に応じて領域分割を行っている。この領域分割はハードウェアでは意識する必要はなく，ソフトウェアで領域アドレスよりデータ型を判別するために使用される。命令空間へのアクセス（命令語の読出し，命令語によるread/write）は専用のI-cache (8KB)を介して行なわれる。V領域以外のデータ空間へのアクセスはD-cache (32KB)を介して行なわれる。L及

びH領域は小さなアドレスに向って使用され，他は逆方向に使用される。

1.3. ポインタ表現

図3のようにポインタは大別して3つのフィールドより成り，最初の2bitはcdr coding，Cstack内のデータ認識及び2語長データのために使用される。一般のポインタはタグ部6bit（内頭1bit）はshort float数表現に使用）の内5bitで32種類のタグを扱うことが可能である。情報部24bitはタグに応じてアドレス又は値そのものが入れられている。

blockと呼ばれる実体が連続領域に格納されるデータ型に対してはdescriptor（図4）を介してアクセスしている。このdescriptorはレジスタに

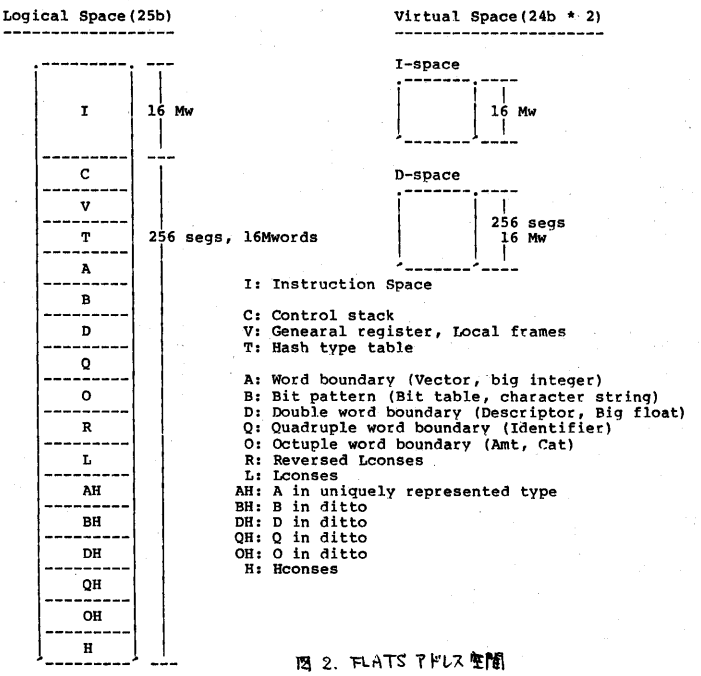
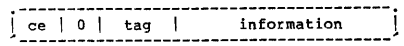


図2. FLATS アドレス空間



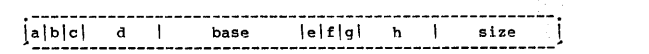
tag: a field to denote data types
insignificant when ce is invisible
information: a value or an address

図3. ポインタ表現

載せられても2語長で意味をもつものなので，ceフィールドを使用して一般ポインタと区別し，通常のデータとしてdescriptorにアクセスするとエラーとなる。

1.4. タグ及びタグセット

Lispの場合実行時にデータ型を示す情報も命令や関数に渡さなくてはならない。ソフトウェアによるLispシステムの場合これまでにアドレス域によってデータ型と対応をつけたり，読み出した語のおにタグをつけていたりしたが，語長がアドレス空間に対して充分大きい場合には語内タグ法がソフトウェアの場合でも有用である。FLATSでは



	in memory	on register
a,e	always 1 (descriptor bit)	
b,f	undefined (0) 0:even 1:odd physical address	
c	if c=0 then d is a tag, otherwise bit address.	
d	tag or bit address	
base	base word address	
g	unused bit	
h	if c=1 then bit address, otherwise encoded value	
size	size or limit address	

creation	MKDES
loading	LDDES b,f are set
store	DEPDES
modify	RPLADES (special subroutine name)

h (encoded value)

....X	frozen bit, 1 during sweep operation
...X.	relocated rehash required bit
XXX..	unused (must be filled with zeros)

kind	c	g	d	base	h	size
SDES	0	1	STR	base(B)	00000	length in byte
HSDES	0	1	HSTR	base(BH)	00000	length in byte
PDES	0	1	PBINT	base(A)	00000	# of words - 1
HPDES	0	1	HPBINT	base(AH)	00000	# of words - 1
NDES	0	1	NBINT	base(A)	00000	# of words - 1
HNDES	0	1	HNBINT	base(AH)	00000	# of words - 1
VDES	0	1	VECT	base(A)	00000	upbv
HVDES	0	1	HVECT	base(AH)	00000	upbv
ADES	0	1	AMT	base(O)	000XX	table size - 1
HADES	0	1	HAMT	base(OH)	000XX	table size - 1
CDES	0	1	CAT	base(O)	000XX	table size - 1
HCDES	0	1	HCAT	base(OH)	000XX	table size - 1
TDES	0	1	HTT	base(T)	00000	table size - 1
BDES	1	1	bit.adr	word.base	bit.adr	word.limit

図4. デスクリプタ表現

単一表現データも含めすべてタグを5bitのフィールドに収めている（図5参照）。データ型はタグ及びタグの集合で表現されるので，FLATSではタグの値によって分岐するTAGGO命令，タグの同値性テスト命令及びタグ集合のメンバーをテストする命令が備っているので，タグに応じた処理への高速分岐及びLispのpredicate等の高速化を図っている。

1.5. Cstack内データ表現

Cstackを利用する命令は局限されており，直接自由領域のようには扱わないので32bitの頭2bitを利用し，linkage，binding等の特殊目的及び一般データに分けて使用している。

1). linkage

CALL命令の実行により復帰すべきアドレス24bitとCFP (Current Frame Pointer) の増分値7bitを入れている。頭1bitはonにされ，他のデータと区別し，RETURN命令によってチェックされる。通常のpop命令でlinkageデータが指定されたらエラーとなる。

bits	name	meaning
00000	UNDEF	undefined values
00001	UNDEF	
00010	UNDEF	
00011	UNDEF	
00100	UNDEF	
00101	RPAIR	dotted pair in R area
00110	LPAIR	dotted pair in L area
00111	HPAIR	dotted pair in H area
01000	ID	identifier in Q area
01001	HID	H type identifier in QH area
01010	HSSTR	H type short string (pointer itself)
01011	BIN	binary data (des. in D area)
01100	IADDR	function pointer (pointer itself)
01101	HADDR	D space address (pointer itself)
01110	PZINT	non-negative short integer (pointer itself)
01111	NSINT	negative short integer (pointer itself)
10000	PBINT	positive big integer
10001	NBINT	negative big integer
10010	HPBINT	H type positive big integer
10011	HNBINT	H type negative big integer
10100	PBFLT	non-negative big floating point number
10101	NBFLT	negative big floating point number
10110	HPBFLT	H type non-negetive big floating point number
10111	HNBFLT	H type negative big floating point number
11000	STR	string
11001	HSTR	H type string (its length exceeds 3)
11010	VECT	vector
11011	HVECT	H type vector
11100	AMT	associative membership table
11101	HAMT	H type associative membership table
11110	CAT	content addressed table
11111	HCAT	H type content addressed table

図5. タグの種類

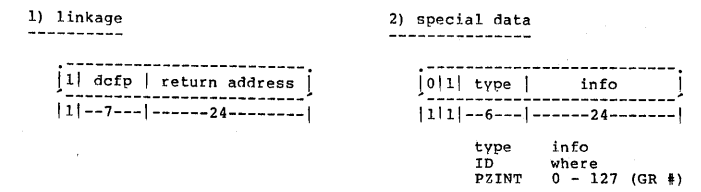


図6. Cstack内データ

2) binding

FLATSでは変数の他に汎用レジスタもbindingが可能であり，できるだけフレーム又は汎用レジスタを使用した方が命令数も減少し処理の高速化が期待できる。REDUCE等のプログラムを走らせる場合には，予めグローバル変数等は汎用レジスタに割り当てたりしてoptimizeすることにしている。binding/unbindingのための専用命令は用意されていないが変数のbindingとunbindingは，

```
XIDV  new, old, id                    CEPOP  id
CPUSH old          あるいは            CPOP   old
CEPUSH id                             STID   id, old
```

のそれぞれ3命令で実行が可能であり，汎用レジスタの場合には，

```
CPUSH  old                            CEPOP  dummy
CEPUSH GR#         あるいは            CPOP   old
```

で行なう事ができる。汎用レジスタの場合正常sequenceではCEPUSH/CEPOPは不要であるが，errorsetで自動的にunbindingが行なわれるためには必要である。

2. Cdr coding 及び list 処理命令

2.1. cdr coding

複雑かつ大規模の数式処理に於いては，計算の進行中に大量の Lisp セルを消費している。これまでの Lisp システムでは処理速度との関係で car 部と cdr 部の bit 長が同一で表現されてきたが，Xerox の Lisp マシン [6] においては car 部と cdr 部の bit 長が異なり car 部の方が大きい。また MIT の CONS/CADR マシン [7] においては cdr 部は car 部の中に取り入れられている。このように cdr coding はハードウェア化とともに実用に至った。その結果線型化リストならば 50% のメモリ使用量で済むため見かけ上アドレス空間が同じでも最高倍までのデータの格納が可能となった。この cdr coding ソフトウェアでは処理速度が格段におちてしまう。語長が足りない場合等は利用できないが，主記憶容量の少ないシステムではソフトウェアでも有用であると思われる。FLATS ソフトウェア・エミュレータにより実測すると通常の recursion の多いプログラムでは 50% 近いメモリの節約が期待できるし，文献 [6] によれば Lisp セルが 62 bit から 45 bit に減少した効果があるとされている。

FLATS では主記憶より読出された語の頭 2 bit は特別に検出できる等のハードウェア・ロジックで cdr coding をサポートしている。この cdr coding によってデータの圧縮の他に，キャシュのミス・ヒット及び page fault の減少という効果も期待される。頭の 2 bit は MIT の Lisp マシンと同じく，

1) cdr は NIL である (cdrnil)。
2) 次の番地が cdr である (linear)。
3) 通常の 2 語セルである (normal)。
4) 2 語セルへの間接番地である (invisible)。

と encode される。

2.2. list 処理命令のアルゴリズム

図 7 に示した list 処理命令は cdr coding のチェック を cache の read と並列に実行している。CAR, CDR, RPLACA, RPLACD の各命令はオペランドの R1 が pair でなければ分岐する。RPLACA, RPLACD は Lisp 関数のように値は返さない (値は R1 と同じ)。実際の主記憶との転送中は 64 bit あるので CAR, CDR 部とも 1 回の read で読み出し可能である。CONS や RPLACD の結果，page fault になって必要ならば GBC を起動する。CONS は R2 の値に応じて 1 語又は 2 語 Lisp セルを消費し，タグには pair を入れて返す。これらの命令は多くの場合，ハードウェアの基本サイクルで終了するので高速に実行されることになる。LPR は L 領域の利用可能なアドレスを保持している。

```
CAR(R1,EJ)
 if tag of R1 is not pair then error.jump(EJ);
 temp := read.d.cache(R1);
 if ce of temp is invisible then temp := read.d.cache(temp);
 result in R3 is taginfo of temp;

CDR(R1,EJ)
 if tag of R1 is not pair then error.jump(EJ);
 temp := read.d.cache(R1);
 result in R3 is
  if ce of temp is linear then R1+1
  else if ce of temp is cdrnil then NIL
  else if ce of temp is normal then taginfo of read.d.cache(R1+1)
  else taginfo of read.d.cache(temp+1);

RPLACA(R1,EJ,R3)
 if tag of R1 is not pair then error.jump(EJ);
 temp := read.d.cache(R1);
 if ce of temp is invisible then write.d.cache(temp,R3)
 else write.d.cache(R1,ce of temp | taginfo of R3);

RPLACD(R1,EJ,R3)
 if tag of R1 is not pair then error.jump(EJ);
 temp := read.d.cache(R1);
 if ce of temp is normal then write.d.cache(R1+1,R3)
 else if ce of temp is invisible then write.d.cache(temp+1,R3)
 else { temp1 := make.normal(temp,R3);
        write.d.cache(R1,invisible | temp1) };

CONS(R1,R2)
 result in R3 is
  if R2 is NIL then make.cdrnil(R1)
  else if info of R2 is equal to LPR+1 then make.linear(R1)
  else make.normal(R1,R2);
```

図 7. list 処理命令アルゴリズム

3. FLATS GBC アルゴリズム

FLATSに於ける廃品回収（GBC）は，ある領域の使用可能なメモリが設定値以下になった時又はpage faultを契機に起動される。FLATSには単一表現型データ領域も存在するのでいつ，どのように回収するという戦略は沢山あり得る。一般的なGBC処理の流れは図8のようになる。リストを線型にする処理は大変コストが高いのとハードウェアのGBC専用命令が有効に活かされないので，いつ行うかの決定にはアプリケーションの性格等を考えて動的に対処しなくてはならない。FLATSの仮想アドレス空間が広いのでpage faultをなるべく生じさせない工夫が必要で，relocation tableは変換アドレスとbit表のpairにし，ミスヒットやpage faultを減少させている。FLATSでは各領域毎にmark bit表を用意し，各領域の先頭番地及び終端番地，各mark bit表のdescriptor及び各relocation table a descriptorは予め定ったレジスタに載せられている。

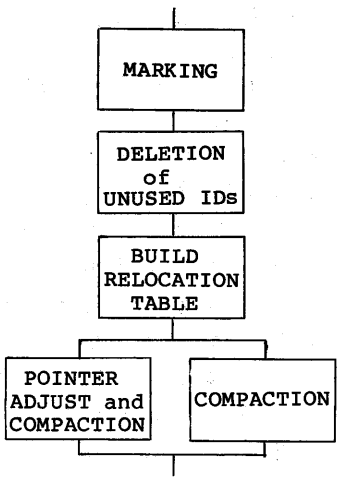


図8. GBC処理

(1) marking

markすべきポインタが与えられると，TAGGO命令で分岐しポインタの属している領域の先頭番地（LPMIN≡base.pair.area）を引き領域内相対位置を得て，TSTMRKB (Test & Mark Bit)命令でmarkingすると同時にmarkされていたかどうかで分岐する。markされていなかった場合には更に先をmarkしに行く。当然ポインタでないデータはその先をmarkしに行かない。図9にidentifierとpairのmarkingアルゴリズムを示しておく。このMARKはサブルーチンであるがFLATSの場合CALL/RETURNは0ns命令のため実質的には数命令で1つのセルのmarkingは完了する。

```
MARK(pointer) : subroutine
loop.back:
  go on tag of pointer {TAGGO
tag.of.id:
  for i=pointer,pointer+3 {
    j := i - base.id.area;
    if unmarked(id.bit.table, j)
       then MARK(read.d.cache(pointer)) };
  return;
tag.of.pair:
  j := pointer - base.pair.area;
  if unmarked(pair.bit.table, j)
     then MARK(read.d.cache(pointer));
  switch on ce of pointer into {
  case of linear:
      pointer := pointer + 1;
      go to loop.back;
  case of cdrnil:  return;
  case of normal:
      j := j + 1;
      if unmarked(pari.bit.table, j)
         then MARK(read.d.cache(pointer+1));
      return;
  case of invisible:
      j := pointer - base.pair.area;
      if marked(pair.bit.table, j) then return;
      pointer := read.d.cache(pointer);
      go to tag.of.pair;
  };
.
. /* the other cases are omitted due to space
.  consideration. */
.
}TAGGO
```

図9. markingアルゴリズム

(2) 不用idの消去

marking が終了したら未参照（markされていない）でかつprint nameだけ（但，関数とも未定義かつproperty listなし）のidは消去可能である。Compiler等で生成されたgensymed id等は完全に消去されている。OBLIST（intern table）のscanにはSWEEP命令が使用される。SWEEP命令はemptyまたはdeletedの要素は自動的にスキップする。

```
sweep.index := 0;
sweep intern.table from sweep.index{
  sweep.index := sweep.index + 1;
  id := read.d.cache(intern.table+sweep.index);
  if valuep(id) | getd(id) | plistp(id)
     then MARK(id);
};
sweep.index := 0;
sweep intern.table from sweep.index{
  id := read.d.cache(intern.table+sweep.index);
  x := read.d.cache(id) - base.id.area;
  if unmarked(id.bit.table, x)
     then write.d.cache(intern.table+sweep.index, deleted);
  sweep.index := sweep.index + 1;
};
```

図10. 不用id消去のアルゴリズム

(3) relocation tableの作成

ハードウェアのCPU-MCU (Memory Control Unit) 間の転送巾が64 bitであること，page faultの回数を減らすために(1)で作成したbit表とrelocationの

めのアドレス変換表をひとつにまとめる。この時データ領域のcompactionも一緒に行なうことができる。領域によっては次のポインタ修正時に行う方がよい場合もあり，どちらを採用するか等の判断はアプリケーションにも依存すると思われるので実際には結果のよかった方におちつくとかの柔軟な対処が望まれる。図11にはrelocation対だけを作成するアルゴリズムを載せる。forを含むループ内命令数は5～10である。領域毎に行なう必要があるので処理時間は全領域の和に比例する。

```
l := LPMAX;
for i=(LPMAX-LPMIN)/32,0,-1 {
  x := read.d.cache(pair.bit.table, i);
  c := bitcount.in.word(x);
  l := l - c;
  build.reloc.pair(l,x,2*i,reloc.pair.base);
};
LPR := l;
```

図11. relocation table作成アルゴリズム(L領域)

(4) pointer修正及びcompaction

実体にポインタを含まないbinary等のデータはcompactionのみ行えばよい。FLATSではビット表をscanするための命令FBOU/D (Find Bit One Upward/Downward) が用意され，ポインタ修正のためにPADJ命令（図13にアルゴリズム）も用意されている。図12にL領域のポインタ修正及び領域のcompactionのためのアルゴリズムを掲げる。

```
  m := LPMAX;
  l := (LPMAX-LPMIN)*2+1;
B:
  l := find.bit.one.downward(reloc.pair.base, l);
  if encountered word end then go to WEND;
  x := read.d.cache(base.pair.area, l);
  go on tag of x {TAGGO
tag.of.pair:
  new := pointer.adjust(reloc.pair.base, LPMIN, x);
  write.d.cache(m, new);
  m := m - 1;
  go to B;
.
. /* the other cases are omitted due to space
.    consideration.  */
.
}TAGGO
WEND: if l is equal to 0 then end;
      l := l - 32;
      go to B;
```

図12. pointer adjust & compaction アルゴリズム(L領域)

```
PADJ(R1,R2,R3)
 word.index := info of (R3 - R2);
 bit.posn := word.index & 31;
 bit.index := (word.index >>5) * 2;
 (x,y) := read.d.cache.dooble.word(R1, bit.index);
 cnt := bit.counter(y, bit.posn);
 result in R3 is cetag of R3 | (x + cnt);
```

図13. PADJ命令のアルゴリズム

ポインタ修正を行なうべきポインタ（図12のx）の属する領域の先頭番地（図12のtag.of.pair:の次の行のLPMIN）とその領域に対応したrelocation tableの先頭番地（図12の同一行のreloc.pair.base）がポインタ修正毎に必要であるが，FLATSではTAGGO命令で分岐し，タグ毎にPADJ命令を使用している。PADJ命令のオペランドであるR1,R2及びR3にはそれぞれrelocation tableの先頭番地，ポインタの属する領域の先頭番地及び修正されるポインタが入れられていて，結果としてR3に修正済みのポインタが返される。

以上述べたGBCのアルゴリズムはFLATSソフトウェア・シミュレータで確認されている。

4. おわりに

FLATSのlist処理命令及びGBC命令を中心に述べてきたが，多倍長のアルゴリズム，割出し及割込み処理，並列ハッシュ命令，入出力等のシステム命令については稿を別にして発表することとしたい。FLATSでは通常の計算機のようなタグなし32bitを扱う命令も用意されていて主にシステムが使用する。

参照文献

[1] Goto, E. et al., FLATS, a Machine for Numerical, Symbolic and Associative Computing, Proceedings of the 6th Annual Symposium on Computer Architecture, (Apr. 1979).

[2] Hearn, A.C., REDUCE 2 User's Manual, UCP-19, Univ. of Utah, (1973).

[3] Terashima, M. and Goto, E., Genetic Order and Compactifying Garbage Collectors, Information Processing Letters, Vol.7, No.1, (Jan. 1978).

[4] Marti, J. et al., Standard LISP Report, UCP-60, Univ. of Utah, (Jan. 1978)

[5] Goto, E., Monocopy and Associative Algorithms in an Extented LISP, Information Science Lab., Tech. Rep. 74-03, Univ. of Tokyo, (1974).

[6] Bobrow, D.G. and Clark, D.W., Compact Encodings of List Structures, ACM Transactions on Programming Languages and Systems, Vol.1, No.2, (1979).

[7] Greenblatt, R., The LISP Machine, Working Paper 79, M.I.T. Artificial Intellegence Lab., Cambridge, Mass., (Nov. 1974).

附録 FLATS命令表(機能別分類)

List processing
CAR, CDR, RPLACA, RPLACD, CONS, RCONS, HCONS
CADR, CDDR, LIST2

Vector handling
UPBV, GETVX, PUTVX

Garbage collector
FBOU, FBOD, PADJ, MARKB, RESETB, TSTB, TSTMRKB, ZCNTB

Table look-ups using hashing
GETAMT, PUTAMT, REMAMT, GETCAT, PUTCAT, REMCAT
GETHTT, GETNHTT, STHTT, REMHTT

Fixed point arithmetic
M, MAD, D, DA, DN, MINT, ZADD, ZSUB
AKN, AKNP, AKNZ, AKP, AKPZ, AKZ, AKC, ACKC
SKN, SKNP, SKNZ, SKP, SKPZ, SKZ, SKB, SBKB

Arbitrary precision arithmetic
ADD0, ADD1, ADD2, ADD3, SUB0, SUB1, SUB2, SUB3
MUL0, MUL1, MUL2, MUL3, DIV0, DIV1, DIV2, DIV3
DIVA0, DIVA1, DIVA2, DIVA3, DIVN0, DIVN1, DIVN2, DIVN3

Transfer control
GOTO, GOTOR, TAGGO, CGO, CALL, CALLR, RET, RTT, RTTJ, RTTN
BTE, BTSM, BNTE, BNTSM, BEQN, BNEQN, BGE, BGT, BEQ, BNEQ
ZEQ, ZNEQ, ZGT, ZGE

Logical instructions
AND, OR, XOR, NOT, ZAND, ZOR, ZXOR, ZNOT
EQ, NEQ, EQN, NEQN, GE, GT, TE, TSM, NTE, NTSM

Push and pop
CPUSH, CEPUSH, ZPUSH, CPOP, CEPOP, ZPOP, LDCT
PSHACCL, PSHACCH, POPACCL, POPACCH

Load and store
ZLDI, ZLDD, ZSTI, ZSTD, STID, LDID, XIDV
MOV, MOVW, ZMOV, LDDES, LDDESN, DEPDES
LDACCL, LDACCH, STACCL, STACCH
LDSPR, STSPR, ZLDMCR, ZSTMCR

Shift
SLL, SRL, ZSLL, ZSRL, SLA, SRA, SLAD, SRAD

Purge cache
PURI, PURSI, PURG, PURSG, PURV, PURSV, PURD, PURSD

Miscellaneous
SWEEP, DI, EI, ZMOVARG, ZTSTR, ZTSTX, ZTSETR, ZTSETX