FLATS 記号処理命令
（高速並列ハッシュ命令及び多倍長命令）

鈴木正幸　平木敬　清水謙多郎　佐藤三久　稲田信幸
東大・理　電総研　東大・理　東大・理　理研

1. はじめに

本稿では、FLATS[1]のLISP拡張機構である次の2点に絞って報告する。

(1) hash 機構
- hash表を利用したdata型とdata構造
- hash型dataへの演算
- parallel hashing hardware
- hashingのpipeline処理

(2) 多倍長演算機構
- 多倍長整数のdata構造
- 多倍長演算用命令(BIGNUM命令)
- 多倍長演算の並列処理

これらの機構は、記号処理の高速化に大きく貢献することが期待される。

2. FLATS hardware概要

Hardwareの構成を図1.に示す。CPUは3つのpipeline unit (I, V, D)より成り、各unitは独自のCachéによりdataが供給される。Iは命令の先読み、Vはoperandの先読み書込み、Dは命令の実行をおこなう。Pipelineの流れを図2に示す。命令は2clock cycleに1個実行される。Iは先読みと同時にCALL, RET, GOTOの先行制御をおこない、これらの命令の実質実行時間は0になる。CALL, RETのstack操作はCによって処理される。

Cachéは命令用にI(容量8KB, CPUとの巾 8byte)、register(general, frame)用にV(6KB, 24byte (3port))、data用にD(32kB, 32byte-hash, 8byte)

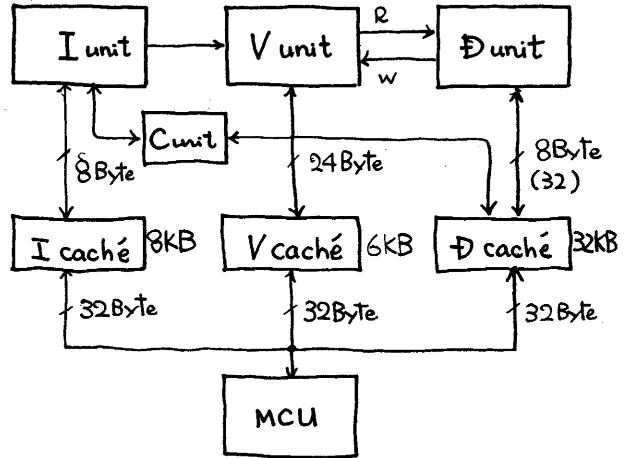


Fig. 1. FLATS 構成図

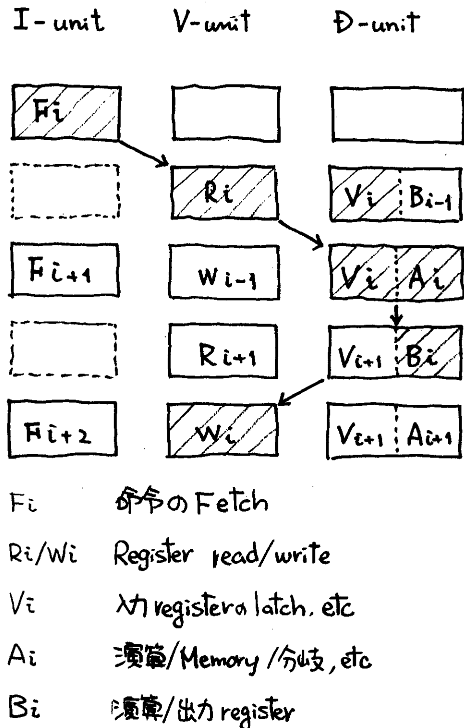


Fig. 2 Pipeline の Flow

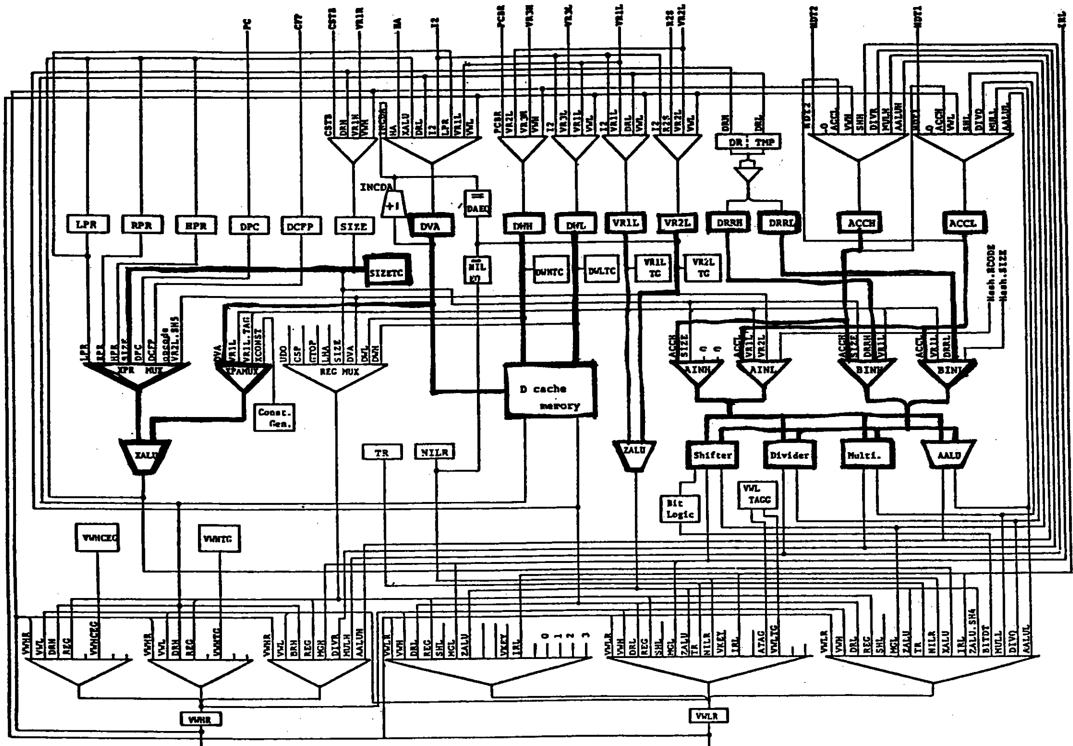


Fig3. CPU D unit

が用意されている。Caché に対するアクセスは論理addvessによっておこなう。

MCU(Memory Control Unit)は Caché と Channel からの memoryへのサービスをおこなう。Address 変換、Page表の管理もあわせておこなう。

Main Memory はアクセスタイム270nのMOS DRAM、16MB 実装の予定。

本稿で述べるhashと多倍長演算の高速化は主にDunitによっておこなわれる。(図3の太黒線部が特に関与)Dの内部は、Memory/List処理部、pointer/index処理部、演算処理部、Hash処理部より成る。Dは約250 bit の水平型micro-program[2]によって制御される。D内の各部はかなり独立に動作できることを利用して、hash命令、BIGNUM命令ではその並列処理により、高速化を実現している。

3。Data Type と Data構造[3]

FLATSでhardwareによりサポートされる(実行時tag check、memory access check)data typeを図4に示す。これらは、Standard LISP が提唱するものにAMT(Associative Mapping Table)、CAT(Content Addressed Table)、Bit Vector、BIG-FLOAT を付加し、それぞれのUnique 表現としてのHtype を導入したものである。[4]

AMT(図5-①)、CAT(5-②)、HTT(5-③)、BIGINT(5-④)にdata構造を示す。HTTはHtypeを生成する際に検索される表である。

可変長のdata(listを除き)はMemoryのaccess範囲を実行時にcheckするため descriptor(〈b,ℓ〉の対、bはbase、ℓはlimit(hash表の時はsize)address)を通してaccessされる。(図5-④)。

AMT/CATを検索するには、Keyと呼ばれるH-typeのdataを使用する。Keyは登録される際に、頭2bitをoccupied/empty/deletedを表わすために用いる。(図5-⑥)

HTTの場合は、頭1bitをそのために用いて、残りをvkey(virtual key)とH type dataへのpointerとして使用される(図5-⑥)。

Hash表を指すdescriptorは、表の状態を表わすfield(enc)をもっている。その意味は、rehash:H typeのrelocationによりhash表が無効となり、rehashをおこなう必要有。frozen:hash表を順検索中であり、書込みを禁止する。trap:独自のhashingによって作られたhash表であるため、softで処理することを示す。

BIGINTは、2's complement表現

$$A = -S\cdot(2^{24})^n + \sum A_i\cdot(2^{24})^i$$

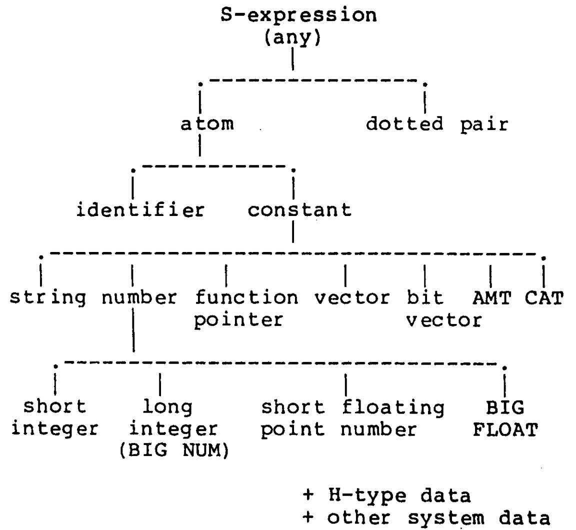
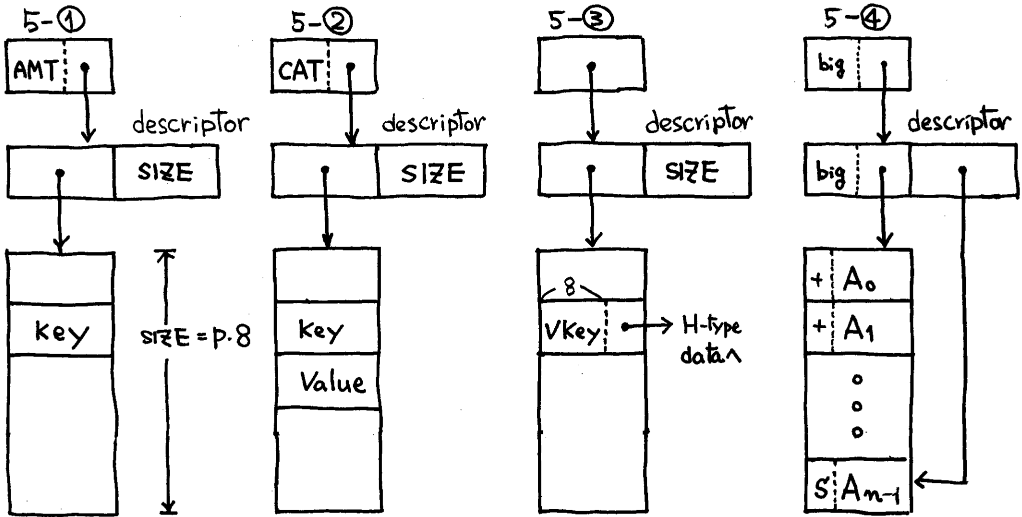
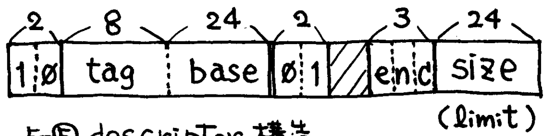


Fig. 4 Data Types

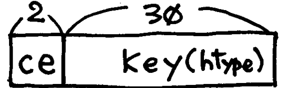


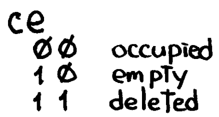
①AMT data 構造　②CAT data 構造　③HTT data 構造　④BIGINT data 構造 (Vector)



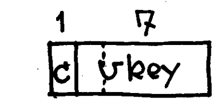
enc
1 × × frozen
× 1 × rehash
× × 1 trap

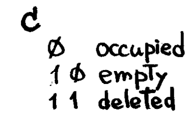
5-⑤ descriptor 構造





5-⑥ AMT/CATのKey 構造





5-⑦ HTTのVkey

		GET	PUT	REM	LISP
AMT	Key ∈ AMT	T	NIL (分岐)	NIL (分岐)	集合演算 (flag)
	Key ∉ AMT	NIL (分岐)	T	T	
CAT	Key ∈ CAT	value	NIL (分岐)	NIL (分岐)	p-list演算
	Key ∉ CAT	NIL (分岐)	T	T	

( )…命令での実現

Fig. 6 AMT/CAT 演算

を採用している。BIGINTへのaccessはregister上のdescriptorによっておこなわれる(hash表はrehashがあるために、registerにあげることはできない。)。

4。Hash命令とHashing

4-1。AMT/CAT演算 5)

全てのkey∈H-type, value∈ANYに対して図5の様に演算が定義される。命令としては、T/NILの換りに分岐が用いられている。LISPではAMTがflagにCATがp-listに対応する。但しflag/p-listが1つのsymbolic atomに対して1つしか定義できない(名前がつけられる)ため、keyの衝突、副作用、回収などの問題がある。AMT/CATは値であるため(名前との切離し)にこれらの問題解決と、hash表によるimplementによる高速化がなされている。

4-2。Hashing hardware

Hashingを速くおこなうためhardwareのサポートとして、parallel hashingとD unit内のpipeline(CPU全体とは区別される)がある。

4-2-1。Parallel Hashing 6)

Hash表の8つのentryを同時に読み出し、与えられたKeyと比較をおこなうことで効率を上げるものである。図7に示す様に、D Cachéが8つのbankに分

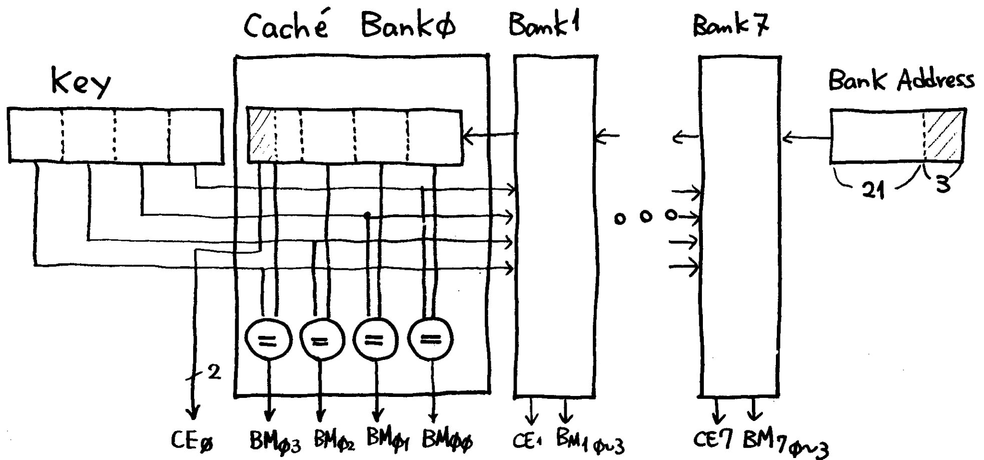


Fig. 7 Parallel Hashing Hardware

かれており、各bankにCE部(頭2bit)と与えられたkeyとのbyte毎の比較回路を備えている。Cachéより出力されるCEφ~7とByteMφφ~7,3がHash Unitにより解析され、Match/Empty情報とそのaddressが出力される。
Parallel hashingによる効率向上については文献6)を参照されたい。

4-2-2。Hashingのⓓunit内Pipeline処理
Hashingのⓓunit内の処理を図8に示す。Hashingは次の3つの処理
1) Next address生成(Δh+hi) --- AALU
2) Parallel search --- Cache
3) Match/Empty address生成 --- Hash
より成る。それぞれを処理するunitの前後に4つのpipe line registerがある。このpipelineは真中のⓓcachéのreadに同期してすすめられる。ⓓcachéがmiss hitしない限り、毎cycle parallel hash searchをおこなう。

4-2-3。Hcodeの生成とHφ
与えられたkeyより、いかに良い(hash表に一様に分布する)Hφをつくるかは、load factor(α)に対するsearch回数に大きく寄与する。(特にαの大きなところで)

AMT/CATに対するhashingでは、最初にdescriptorの読み出しをおこなうため、この間に以下の手順でHφをつくり出す。

- key(29bit)よりbit shuffle回路、adderにより24bitの2つ数を作る。
- この2つの数を掛けあわせ、中の28bitをhcodeとする。28bitのうち上位7bitがukey, 21bitが小数としてみて、normalizeされたindex値とする。
- hash表のsizeと上述のindexを掛けて最初のhash address(Hφ)を求める。

hcode生成algorithmは、keyからhcodeへの各bit依存性、bit間の相関による検査で良い結果が得られている。

4-3。Algorithm
4-1, 4-2で説明したhardwareによるhashingのalgorithmのflowを図9に示す。Wの2つのstateはCachéのmiss hitの解消を待つstateである。HC stateでhcode生成とdescriptorのread, HA stateでdesc.のenc-fieldのcheckをおこない、いずれかのbitが立っていれば異なるTrapを起す。REQ stateでNextの準備とParallel read要求を出す。HASH stateではNext準備と

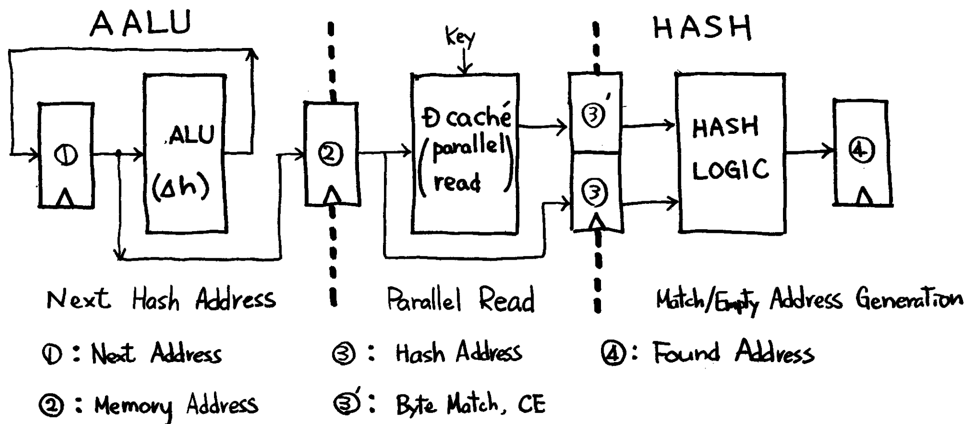


Fig.8 Hashing の Pipeline化

Parallel read と Hash logicをおこなう。Hashingの終了する条件は、Match, Emptyの他に、hash回数のoverflowがある。回数による分岐は、命令を一定時間内に終了またはtrapさせるためと、Hash表の詰まり具合を擬似的に検出し、表を大きくする(rehash)きっかけに利用するためである。このflowの中のHash stateに対するmicro programをAppendix1に示す。

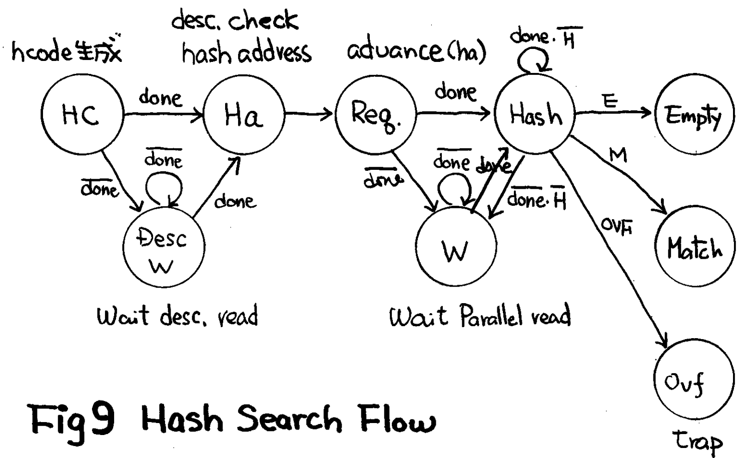


Fig9 Hash Search Flow

4-4. HTT(H-Type Table)

HTTはあるdataが既にunique表現として存在するかを示すhash表である。例えば(A.B)をHtypeにしたければ、(今AとBはHtypeとする) AとBよりh-codeをつくり、HTTをsearchする。この際にkeyとしてはukeyが使われる。Matchがなければ、(A.B)に対するHtypeのdataをつくり、ukeyとともに登録する。Matchあれば、equalでdata構造を比較する。この際に使用されるhashingもukeyによる比較を除き、AMT/CATと同じに実現されている。

5. BIGNUM命令

BIGNUM命令は、主として多倍長数を速く処理するために使用される。Dunitにおいて、Memory access, 演算, index演算が同時に動ける事を利用し、macro命令レベルでpipelineをおこなえる様に設計されている。Dの各unitの働きを図10に示す。

後に説明する様にAALUの制御をmacro命令よりmicroのpatternを供給することで実現している。

5-1. Descriptorに対する演算

まずdescriptorを(b, l)で表わす。bはbase, lはlimitである。BIGNUMに対するaccessはこの(b, l)のbかlのいずれかに制限する。(通常のvector命

b: descriptor base
S: descriptor limit

Unit	CACHÉ	Z ALU	X ALU	A ALU
	Memory op	Desc. op	Branch. op.	+, -, *, /, ≫, ≪.
	Nop	Nop	Nop	op(({Acc, DRR, VR1/2}^{2}))
	Get(M(^b_l))	Cut(b'←b+1)		
	Put(M(^b_l))	Trim(l'←l+1)	Branch(b>l)	

Fig.10 BIGNUM命令時のUnit動作

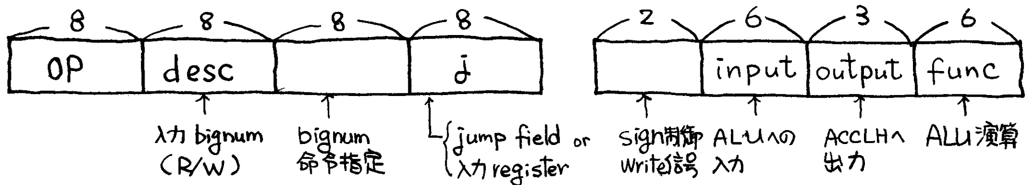


Fig.11 BIGNUM命令 format

命では indexによって access するがこれでは address計算に1cycle 費やす)。access と同時に(b≤ℓ)である事を checkする。(b,ℓ)に対して次の2演算を定義する。

$$\begin{cases} \text{CUT}: (b',\ell) \leftarrow (b+1,\ell), & \text{Mem}(b)\text{へaccess} \\ \text{TRIM}: (b,\ell') \leftarrow (b,\ell-1), & \text{Mem}(\ell)\text{へaccess} \end{cases}$$

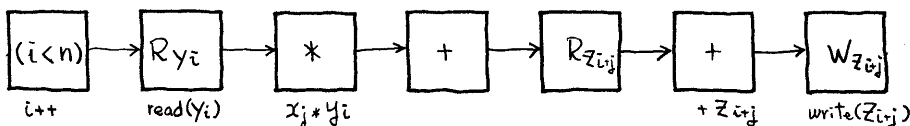
5-2。BIGNUM 命令

BIGNUM 命令は図11に示す様に2語で表わされる。第1語は通常の命令で OP部では {NOP, GET, PUT}⊗{NOP, CUT, TRIM} を与える。desc 部が 入/出力 2 語 register。次の 8bit は BIGNUM命令 であることを示し、最後で jump か 入力 register を与える。第2語目が AALU を制御する CSの pattern であり、BIGNUM命令の実行中は AALUに関する CSの操りにこの語が使用される。最初の 2bit で演算入出力の sign と pipe line register の書込みを指定し、input 部で AALUにつながる A,B bus の選択指定、output部で pipe line register への書き込みdataの選択、func部で AALU に対する演算を指定する。

命令に要する clock数は GET系で2、PUT系で4になっている。

5-3。BIGNUM命令による多倍長計算.

まず多倍長乗算について、通常 index型と BIGNUM 命令によっておこなった場



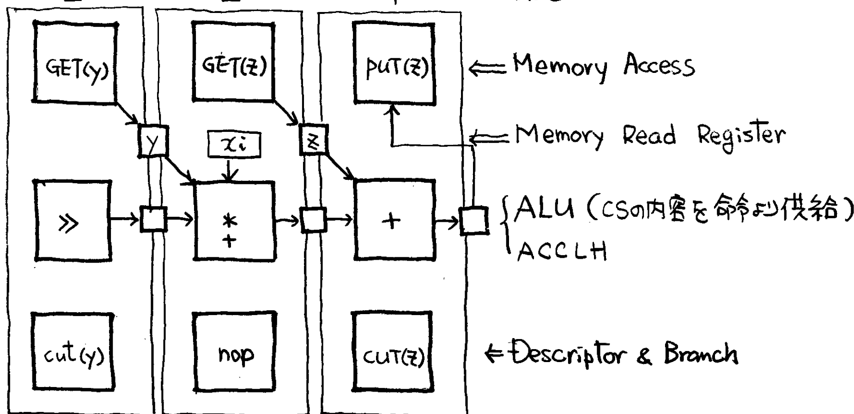


Fig 12. BIGNUM命令(下)と NORMAL命令(上)による多倍長乗算比較

合を図12に示す。箱の上の数字は各演算に要するclock数である。BIGNUM命令によるものの時間が約1/2になっている。BIGNUM命令はstep(n)で(n+1)に必要となるoperandのreadをやると同時に(n-1)で用意されたoperandによって演算をすることが可能であるため、命令間に渡るpipelineを可能としている。

命令間に渡ってpipelineをおこなうため、命令から見えるpipeline registerを必要とする。これにはACCとDRR(memory read register)をあてている。

6。まとめ

FLATSのhashing機構とBIGNUM命令について報告した。

hashingについては、softでおこなうと煩雑なhcode等の計算がhardでは速く、効率よく求められる。またhardによる並列性がうまくおこなえることを示した。

BIGNUM命令の導入によって多倍長計算が速くなることを示した。BIGNUM命令は多倍長計算だけでなく、広い応用が期待できる。

```
GAMT.MEU0=             { ACCH   hash address (hi)
GAMT.MEU1:             { AccL   search count
; Advance
       DVAWR   = DONEWRT                     ; if DONE, then advance
       DASEL   = HA
       ACCLSEL = AALU                        ; ACCL <- ACCL+1 ;count
       ACCLWR  = WRT
       ACCHSEL = AALUH                       ; ACCH <- ACCH-8 ; Hi
       ACCHWR  = WRT
       HADDRWR = WRT

       MEUWR   = WRT        }
       LHASEL  = MATCH      } <- Hash Logic Unitの指定
       HTABLEMODE = AMT     }
       LHMCT   = PUT        }
; Count overflow condition.
       NX0SEL  = HASHOVF   <- overflow条件の設定

; VWL <- address found. Not need for GETAMT (Found Address latch)
       VWLTAGG = TADDR
       VWLTSEL = VWLTAGG      VWLCSEL = SINGLE       VWLISEL = REG
       VWLRWR  = WRT
       REGSEL  = LHA

; Parallel read request
       DC      = HASH
       MRQ     = SA        DDTSWP  = NORMAL        DBSWEN  = ENB

; Next hashing preparation. (hi+Δh)
       HASHMODE= HASH                        ; Hi--
       AINHSEL = ACCH         AINHENB = ENB
       FUNC    = '5                          ; count--
       AINLSEL = ACCL         AINLENB = ENB
       BINLENB = DIS

; Wait
       BRCT    = BRWT   <- 分岐制御
       DWAITCT = BW     <- wait制御

$
       NXBR    = HSR  <- μ分岐Type   <- 次state
;               !(DMEV)      V&!(DME)  D&!(MEV)  (VD)&!(ME) M        E
       NXST    = GAMT.WSA2W, GAMT.OV0, GAMT.MEU1, GAMT.OV1, GAMT.M, GAMT.E
$
```

Appendix 1. GETAMT micro.例.
(1 stateのみ)

Reference.

[1] E. Goto, T. Soma, N. Inada, T. Ida, M. Idesawa, K. Hiraki, M. Suzuki, K. Shimizu, B. Philipov : "Design of a Lisp Machine - FLATS, Proc of LISP Conf. 1982.

[2] K. Hiraki, FLATS Manual, FLATS PROJECT REPORT 1981.

[3] N. Inada, Specifications of the data types and the data structures for FLATS, FLATS PROJECT REPORT 81-02.

[4] E. Goto, M. SASSA, Y. Kanada, "Algorithms and Programming with CAMs." Tech. Reprt. Univ. of Tokyo. 78-05

[5] M. Suzuki, N. Inada, E. Goto, "Associative Data Structure and their Applications, 数理解析研講究録 363, 1979

[6] T. Ida, E. Goto, "Performance of a Parallel Hashing Hardware with Key Deletion, Proc. IFIP Congress 77, North-Holland 1977.