数理解析研究所講究録
第 482 巻 1983 年 41-48

Design of a Lisp Machine - FLATS

E. Goto*,**, T. Soma*, N. Inada*, T. Ida*, M. Idesawa*

K. Hiraki**, M. Suzuki**, K. Shimizu**, B. Philipov**

* The Institute of Physical and Chemical Research
Wako-shi, Saitama, 351 Japan

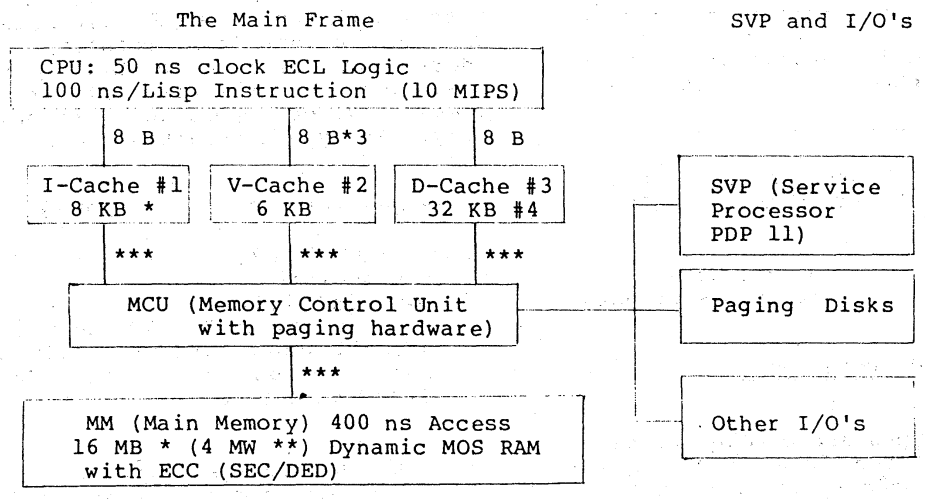
** Dept. of Information Science, University of Tokyo
Bunkyo-ku, Tokyo, 113 Japan

ABSTRACT

Design of a 10 MIPS Lisp machine used for symbolic algebra is presented. Besides incorporating the hardware mechanisms which greatly speed up primitive Lisp operations, the machine is equipped with parallel hashing hardware for content addressed associative tabulation and a very fast multiplier for speeding up both arithmetic operations and fast hash address generation.

1. Introduction

The FLATS machine is configured as shown in Fig. 1. A more detailed diagram is shown in appendix 3.



* 1 B = 1 Byte = 8 bits
** 1 W = 1 Word = 4 Bytes = 32 bits
*** 32 B Parallel Block Transfer
#1 64 set, 4 way — Set associative, 10 ns R/W Access
#2 64 set, 1 way — Time, ECL Bipolar RAM.
#3 256 set, 4 way — 32 B write buffer, non store through.
#4 D-cache is equipped with an 8 word (32 B) parallel match logic for speeding up searching.

Fig. 1 Configuration of the FLATS Machine

2. Basic Data Format

2.1 The standard word format consists of 4 bytes (32 bits): 1 B (tag) + 3 B (24 bits, used for addresses or short integers). Non-standard formats (32 bit, bit pattern and 8 B formats) are described later.

2.2 The 8 bits in the tag byte are used as: 2 bits (used for cdr coding) + 1 bit (short float word tag bit) + 5 bits (used for identifying 32 data types)

2.3 Hardware-tagged data types are shown in Fig. 2.

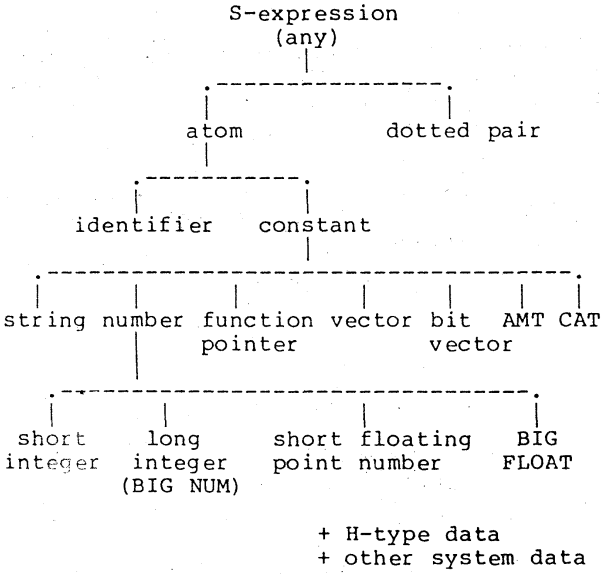


Fig. 2 Data Types

These data types are checked by hardware. "BIG NUM" and "BIG FLOAT" argument(s) in arithmetic operations causes a trap to extended arithmetic routines.

Most data types except CAT and AMT are similar to those of other Lisps. CAT, AMT and H-type data are associative (hashed) data types and are explained later.

3. Address (Pointer) Space

Word addressing is employed, except for bit vectors with bit addressing capability. The virtual addressing space is divided into two sub-spaces, the I-space and the D-space, with 2^{24} word capacity each. The I-space (I for Instruction) is used for storing compiled codes, and pointers into this space are tagged as a function pointer (cf. Fig. 2). All other data types are stored in the D-space (D for Data).

4. Basic Instructions

4.0 Instructions are word addressed

Most instructions are 1 W (4 Bytes) in length and the first byte is the op. code: (op,...).

4.1 High Speed Registers

The 128 global registers (G-reg.) and 127 local stack frame registers (F-reg.) are provided, and the "V-cache" (Fig. 1) is used to realize these registers. Three identical copies of each register are provided in order to realize 3 parallel read ports. Use of both F- and G- registers would speed up the execution time of some programs (cf. recursive APPEND in appendix 1).

4.2 R^3, the 3 Register Address, Type Instructions

R^3 instructions consist of 4 bytes (OP, R1, R2, R3). Each of R1, R2 and R3 is an 8 bit register address. While a register address 0 through 127 specifies a G-reg., 128 through 254 specifies the offset address of an F-reg. relative to CFP (the Current Frame Pointer). Register address 255 is used to specify an immediate constant (32 bits) at the next address. Typical operations performed by a single R^3 type instruction are:

```
r3 := cons[r1, r2]          100 ns.

r3:= add[r1,r2]
r3 := subtract[r1, r2]
r3:= multiply[r1, r2]
        100 ns, if R1, R2 and the results
    are all short integers. BIG-NUM (big
    number) argument(s) causes a trap to
    BIG-NUM routines.
```

4.3 R j R Type Instructions

This format consists of 4 bytes: (OP, R1, "j", R3). The meaning of the 3 bytes op, R1 and R3 is the same as in R^3 (4.2). "j" stands for a conditional short jump to a relative address j, $-128 \leq j \leq +127$. Typical operations of this type are:

```
r3 := car[r1,"j"]
r3 := cdr[r1,"j"]
    100 ns if the invisible pointer of cdr
    coding is not involved. Makes a short
    jump in 100 ns if car or cdr of R1
    cannot be taken.

eqj[r1,"j",r3]
eqnj[r1,"j",r3]
    Always 100 ns. Short jump or non-jump
    to "j" on the truth of (EQ R1 R3).

r1 :=rplaca[r1,"j",r3]
r1 := rplacd[r1,"j",r3]
```

100 ns if the invisible pointer of cdr coding is not involved. Short jump to "j" in 100 ns on bad argument(s).

atomj[r1,"j",-]
atomnj[r1,"j",-]
Always 100 ns. Short jump or non-jump to "j" on atomic R1.

neqj[r1,"j",r3]
neqnj[r1,"j",r3]
Short jump or non-jump to "j" on numerical equality of R1 and R3. 100 ns if R1 and R3 are short integers. BIGNUM argument(s) causes a trap to BIGNUM routines, and non-number argument(s) to an error handler.

4.4 GOTOs

The "GOTO J" instruction has a one word format: (1 byte op code) + (a 24 bit I-space address). The time for GOTO is made practically zero by parallelism as described later. On the other hand, the instruction for "computed GOTO on an integer R1 to one of n = R3 places" has a special n+1 word format and takes 250 ns to execute.

4.5 CALL, RETURN - C-stack Instructions

A hardware stack, called the C-stack (C for Control) different from the local stack frame (cf. 4.1), is provided for stacking a return address and an incremental value, DELTA-CFP of the CFP (Current Stack Pointer cf. 4.1). The "CALL" instruction is always followed by a "GOTO J" instruction. The first byte of the "CALL" instruction is the op. code, the second byte is the immediate value of DELTA-CFP and the last 2 bytes have no significance. In the "RETURN" instruction only the first op. code byte is significant. "CALL" increments the CFP by DELTA-CFP, pushes a linkage word, the return address and DELTA-CFP onto the C-stack, and then goes to J. "RETURN" pops the linkage word from the C-stack, restores the old CFP by subtracting DELTA-CFP from the CFP and returns. The times for "CALL" and "RETURN" are also made practically zero by built-in parallelism.

5. The Architecture for Basic Lisp Operations

5.1 Cdr Coding and RCONS

Besides implementing cdr coding [3] by hardware as in other Lisp machines, RCONS, (Reverse CONS) is also hardware supported. The RCONS instruction (RCONS, R1, R2, R3) can be defined operationally as a statement:

r3 := cdr[rplacd[r2;cons[r1; NIL]]].

RCONS was recognized by Risch [4] as a type of recursion removal pattern, which is typical in list copying part of APPEND and UNION. Recursion can be removed from these functions by using RCONS, which constructs a list from head to tail while CONS constructs a list from tail to head. In the cdr coding system, however, the use of RPLACD would generate a non-linear structure occupying 2 word per list cell in excess of a linear structure. RCONS is hardware implemented so as to construct a compact linear list structure from the right of the free list area while CONS does the same from the left [10]. A programming example with RCONS is given in appendix 1 (cf. APPEND (Iterative)).

5.2 Pipeline and Advanced Control

Three pipeline stages I, V and D are employed: "I" for "Instruction" fetching and prefetching, "V" for reading and writing the "Values" of high speed registers (G and F reg. cf. 4.1), and "D" for instruction execution with memory accesses through the "D-cache". Besides these 3 pipelined stage units, the C-unit, provided for controlling the C-stack, runs concurrently. The C-unit makes use of the D-cache on a cycle steal basis. The I-cache is separated from the D-cache to improve the performance of instruction prefetching. Up to 6 instructions can be prefetched within the I-stage unit. The time needed for branching by short jumps (4.3) is made practically zero by prefetching both instructions in the branching and non-branching sides in parallel with the evaluation of the branching conditional predicate. GOTO, CALL and RETURN instructions are executed in parallel with the execution of other instructions by means of the I-stage unit and the C-unit. Thereby, the time needed to execute these instructions is also made practically zero.

Since conditional branching, GOTO, CALL and RETURN instructions occupy about 50% of the compiled codes in typical Lisp programs, the speeding up of these instructions by parallelism is considered very effective. Some examples are given in appendix 1. Wherein, examples of ASSOCQ and APPEND show that the speeding up of CALL and RETURN is almost effective as recursion elimination.

A new pipeline recurrence relation formulated by Shimizu was used in the design of the pipeline logic [9]. A logic simulator system DDL* written by Shimizu [9] has been used throughout the design of the FLATS. The DDL* system had to be written in Fortran (about 14,000 lines) because all Lisp systems accessible to our group were considered too slow. The world would have been different if FLATS were available!

5.3 Vectors

The operational specification of vector instructions is the same as MKVECT, GETV and PUTV in the Utah standard Lisp [5]. A vector is internally represented by a "vector descriptor" which consists of a pair of pointers (L, U) occupying two words (8 B format data). L and U give the lower and upper bounds of the memory space allocated for the vector. The instruction (MKVECT, R1, -, R3) places a pointer (tagged as a vector) to a new vector descriptor (L, U) in R3, where U = L + R1, provided that R1 is an integer representing the size of the vector. Vector range violation is always checked by hardware in vector access instructions, GETV and PUTV.

5.4 Bit Vector for Garbage Collection

A bit pattern handling hardware [6] is implemented for speeding up the marking of active cells, pointer adjustments and relocation in compactifying garbage collection. Bit vectors (32 bit word) with bit addressing hardware are used for this purpose.

6. P-list vs. CAT, AMT

6.1 P-list (Property-list)

P-list is an important programming concept introduced in Lisp 1.5 [1]. However, it often causes global name clash problems because P-list is usually associated with a global name (atom). This problem can be resolved by using a "gensym" mechanism as shown in 6.2. P-list is usually implemented literally as a "list structure", which results in a rather slow O(n) operation time when n items are placed on P-list.

6.2 AMT and CAT

Two data types, AMT (Associative Membership Table) and CAT (Content Addressed Table), which may be regarded as nameless P-lists, are provided. Operationally, each AMT or CAT instruction corresponds, line by line, to a P-list operation as in:

P-list	AMT, CAT
p := gensym[];	p := mkcat[];
put[p; A; 1];	putcat[p; A; 1];
x := get[p; A];	x := getcat[p; A];
a := gensym[];	a := mkamt[];
flag[a; A];	putamt[a; A];
y := flagp[a; A];	y := getamt[a; A];

The values of x and y are 1 and T respectively in each program. The speed up is realized in AMT, CAT instructions by skipping the gensym mechanism and by using hardware supported hash retrieval so as to realize O(1) operation times.

7. Hardware Hashing and H-Type Data

In the D-cache (cf. Fig. 1), 8 words are compared in parallel to speed up the searching by a hashing hardware [7]. Besides speeding up of AMT and CAT operations (6.2), hashing is employed to construct uniquely represented data types, called the H-type data.

McCarthy [2] once noted about (HCONS X Y), which is like (CONS X Y) but only one copy of the consed object is to be made by searching through the storage to check whether the same structure has been made before.

Searching is to be made by hashing for the sake of speed. HCONS is hardware implemented in our machine. Equality checking of two tree structures, say, a and b, can be made in O(1) time by the pointer comparing primitive eq[a; b] when they are constructed by HCONS. McCarthy remarked that the problem of speeding up the equality checking of large mathematical expressions would be resolved by using an HCONS scheme. However, this is not sufficient. The expression A + B + C may be expressed in many different lists (ordered n-tuple) (A, B, C), (B, A, C), ... owing to the commutative nature of the addition. Unique representation of sets (unordered n-tuple) would resolve this problem [8], since the equivalence of a set is defined as: {A, B, C} ≡ {B, A, C}, ... Hashing hardware for uniquely defining sets is also implemented in our machine. Starting from <ATOM> which is a uniquely defined object in any Lisp, H-type data <H> is defined as nested lists (ordered tuples) and sets (unordered tuples) : <H> ::= <ATOM>|(<H>,...,<H>)|{<H>,...,<H>} in BNF.

Equality checking of any two H-type data can be made in 100 ns by the EQJ or EQNJ instruction (cf. 4.3). Since H-type data are unique like any literal atoms, they can be used as indicators and flags in P-lists, AMTs and CATs. Thus, the H-type data operations are believed to provide a powerful associative computation scheme.

ACKNOWLEDGMENTS

The authors would like to acknowledge members of the FLATS group of Applied Electronics Department, Computer Systems Headquarters, Mitsui Engineering and Shipbuilding Co., Ltd. for the construction of FLATS system, and Computer Systems Group, Fujitsu Ltd. and Fujitsu Laboratories for valuable comments on design methods for ECL logic.

REFERENCES

[1] J.McCarthy et al., "Lisp 1.5, P.M.", MIT Press (1962)
[2] J.McCarthy, in "Symbol Manipulation Languages," D.G.Bobrow ed., North Holland, Amsterdam (1967)

[3] D.G.Bobrow and D.W.Clark, "Compact Encodings of List Structure," ACM TOPLAS, Vol. 1, No. 2, (1979)
[4] T.Risch, "A Program for Automatic Recursion Removal in Lisp," DATALOG Laboratory Report DLU-73-24, Uppsala Univ. (Nov. 1973)
[5] J.B.Marti, A.C.Hearn, M.L.Griss and C.Griss, "The Standard Lisp Report," Univ. of Utah. (1979)
[6] M.Terashima and E.Goto, "Genetic Order and Compactifying Garbage Collectors," Information Processing Letters, Vol. 7, No. 1, (1978)
[7] E.Goto, T.Ida and T.Gunji, "Parallel Hashing Algorithms," Information Processing Letters, Vol. 6, No. 1, (1977)
T.Ida and E.Goto, "Performance of A Parallel Hashing Hardware with Key Deletion," Proc. IFIP Congress 77, North-Holland, (1977)
[8] E.Goto and Y.Kanada, "Hashing Lemmas on Time Complexities with Applications to Formula Manipulation," Proc. ACM SYMSAC 76, (1976)
M.Sassa and E.Goto, "A hashing method for fast set operation," Information Processing Letter, Vol. 5, No. 2, (1976)
E.Goto, M.Sassa and Y.Kanada, "Studies on Hashing PART-2: Algorithms and Programming with CAMs," J. Info. Proc., Vol. 3, No. 1, (1980)
[9] K.Shimizu, "Design and CAD Implementation of Formula Manipulation Machine, FLATS," master thesis, Dept. of Information Science, Univ. of Tokyo, (1982)
[10] M.Suzuki, K.Ono and E.Goto, "A Primitive for Non-recursive Lisp Processing," Journal of Info. Processing, Vol.4, No.4, (1981)
[11] R.Greenblatt, "The Lisp Machine," Working Paper 79, MIT Artificial Intelligence Lab., Camb., Mass., (1974)
[12] R.R.Burton et al., "Overview and Status of Dorado Lisp," Conf. Record of the 1980 Lisp Conference, (1980)

Appendix 1. Execution time of Lisp Functions

The following lists show the definitions of Lisp functions APPEND, EQUAL, and ASSOCQ described in FLATS Lisp assembly language, and their execution time. These are a part of the test programs used for the simulation of the FLATS CPU. The list may be thought of as the object code compiled from the function definitions in Lisp.

APPEND (Recursive)

```
((SUBR  APPEND  2)
      (MOV  FR1  GR127)
((SUBR  APPEND_A  1)
      (CDR  FR0  A1  FR1)
      (CAR  FR0  EJ  FR0)
      (CALL  APPEND_A  1)
      (CONS  FR0  FR1  FR0)
      (RETURN)
A1    (MOV  GR127  FR0)
      (RETURN)
EJ    (CALL  FATAL_ERROR  0)))
```

APPEND (Iterative)

```
((SUBR  APPEND  2)
      (CAR  FR0  A1  FR2)
      (CONS  FR2  NILR  FR2)
      (MOV  FR2  FR3)
LOOP  (CDR  FR0  EXIT  FR0)
      (CAR  FR0  EJ  FR4)
      (RCONS  FR2  FR4  FR2)
      (GOTO  LOOP)
EXIT  (RPLACD  FR2  EJ  FR1)
      (MOV  FR3  FR0)
      (RETURN)
A1    (MOV  FR1  FR0)
      (RETURN)
EJ    (CALL  FATAL_ERROR  0) )
```

Note that in the recursive definition of APPEND:

```
append[x;y]
   == [null[x]->y;
       T->cons[car[x];append[cdr[x];y]]],
```

where the second argument y is simply copied, i.e., y is passed from outer "append" to inner "append" without any change. Such "argument copying" can be removed by storing the value of y in a G (global) register (actually GR127 in APPEND). A Lisp compiler for automatically removing such "argument copying" is now being designed.

EQUAL (Recursive)

```
     ((SUBR  EQUAL  2)
     EQUAL  (BEQ  FR0  FR1  A2)
            (CAR  FR0  A1  FR2)
            (CAR  FR1  A1  FR3)
            (CALL  EQUAL  2)
            (BNEQ  FR2  TR  A1)
            (CDR  FR0  EJ  FR0)
            (CDR  FR1  EJ  FR1)
            (GOTO  EQUAL)
     A1     (MOV  NILR  FR0)
            (RETURN)
     A2     (MOV  TR  FR0)
            (RETURN)
     EJ     (CALL  FATAL_ERROR  0) )
```

ASSOCQ (Recursive)

```
     ((SUBR  ASSOCQ  2)
            (CAR  FR1  A1  FR2)
            (CAR  FR2  EJ  FR3)
            (BEQ  FR0  FR3  A3)
            (CDR  FR1  EJ  FR1)
            (CALL  ASSOCQ  0)
     A3     (MOV  FR2  FR0)
     A1     (RETURN)
     EJ     (CALL  FATAL_ERROR  0) )
```

ASSOCQ (Iterative)

```
((SUBR  ASSOCQ  2)
ASSOCQ  (CAR   FR1   A1   FR2)
        (CAR   FR2   EJ   FR3)
        (BEQ   FR0   FR3   A3)
        (CDR   FR1   EJ   FR1)
        (GOTO  ASSOCQ)
A3      (MOV   FR2   FR0)
A1      (RETURN)
EJ      (CALL  FATAL_ERROR  0) )
```

Execution time of some Lisp functions

Function name	Method	Exec.time(cycles) push down	pop up	total	Approx. speed ratio
APPEND	A	10	6	16	1
	B	6	6	12	1.3
	C	4	5/2	13/2	2.5
	D	-	-	6	2.7
EQUAL*	A	34	4	38	1
	B	22	4	26	1.5
	C	16	2	18	2.1
	D	-	-	-	-
ASSOCQ	A	16	6	22	1
	B	10	6	16	1.4
	C	8	5/2	21/2	2.1
	D	-	-	8	2.8

* When EQUAL returns T.

The execution time listed in the above table indicates a time for processing a single element in the argument lists.

1. Method A

All the instructions are executed in the D-unit. As for a branch instruction, the target instruction is fetched after the execution of the previous instruction is finished. It takes 4 machine cycles to execute a conditional branch instruction.

2. Method B

GOTO is executed in parallel with the other pipeline operations. As for the conditional branch, the alternative target instruction is fetched concurrently with the conditional test.

3. Method C

GOTO, CALL, and RETURN are executed in parallel with the other pipeline operations.

4. Method D

Recursion eliminations are made in addition to method C. For recursion elimination RCONS is used in APPEND and tail recursion removal is done in ASSOCQ. No good iterative method is known for EQUAL.

Appendix 2. Comparison with Other Lisp Machines

	Machine Name	CDR Coding	Logic	Cell Space	Cache Memory	Micro Cycle
1	CADR	2 bits	TTL	16 M	None	180 ns
2	Dolphin	8 bits	TTL	16 M	None	200 ns
3	Dorado	8 bits	ECL	16 M	120 ns	60 ns
4	3600	2 bits	TTL	64 M	200 ns	200 ns
5	ELIS	None	TTL	16 M	None	180 ns
6	EVLIS	None	TTL	64 K	None	100 ns
7	ALPS2	None	TTL	.5 M	None	300 ns
8	Kobe	None	TTL	64 K	None	300 ns
9	FLATS	2 bits	ECL	32 M	50 ns	50 ns

(1) MIT CADR [11]

(2),(3) Xerox Dolphin and Dorado [12]

(4) Symbolics Inc.
21150 Califa Street, Woodland Hills,
CA 91367

The project leaders of the following Japanese machines are:

(5) Ikuo Takeuchi (software) or
Yasushi Hibino (hardware),
Musashino Electric Communication Lab.,
Nippon Telegram and Telephone
Public Corporation,
Midori-cho, Musashino-shi, Tokyo 180

(6) Prof. Hiroshi Yasui,
Faculty of Engineering,
Osaka University,
Yamadaue, Suita-shi, Osaka 565

(7) Prof. Koutaro Mano,
College of Science and Engineering,
Aoyama Gakuin University,
Chitosedai, Setagaya-ku, Tokyo 157

(8) Prof. Yukio Kaneda
Faculty of Engineering,
Kobe University,
Rokkodai-cho, Nada-ku, Kobe-shi 657

Appendix 3.1. The Block Diagram of FLATS

Appendix 3.1. The Block Diagram of FLATS

Appendix 3.2. Characteristics of Sub-units in the Block Diagram

Acronym:

CSTB — Control Stack Top Buffer (32 bits) with CSP (C-stack Pointer, 24 bits)

CFP — Current Frame Pointer (24 bits)

GTOP — Top Address of General Registers (24 bits)

FAA 0-2 — Frame Address Arithmetic 0-2 24 bit + 7 bit adder (6 ns)

ACC — ACCumulator (48 bits + sign)

TC 1-3 — Tag Checker 1-3 (each 8 bit hardwired logic)

ARITHMETIC

Combinatorial hardwired logic	
48 bit ALU (+,- etc.)	20 ns
24 bit by 24 bit multiplier	30 ns
48 bit parallel shifter	12 ns
48 bit over 24 bit divider	200 ns

BIT handling unit

32 bit population counter of both 0 and 1 in a 32 bit word. Used for compactifying garbage collection (cf. 5.4),
48 bit bidirectional priority encoder of both 0 and 1 in a masked 32 bit word. Used for compactifying garbage collection and normalization in floating point arithmetic.

HASHing unit

32 byte (256 bit) parallel search on D-cache in 50 ns,
Commutative and noncommutative hash code generation of 21 bit hash address and 7 bit virtual key with 30 bit actual key fully randomized in 50 ns.

BIG NUM pipeline unit

parallel execution of (1) number arithmetic (24 bits or 48 bits), (2) address arithmetic, and (3) memory access.

TAG-GEN — Tag Generator (8 bits)

LPR — L area Pointer Register for CONS

RPR — R area Pointer Register for RCONS

HPR — H area Pointer Register for HCONS

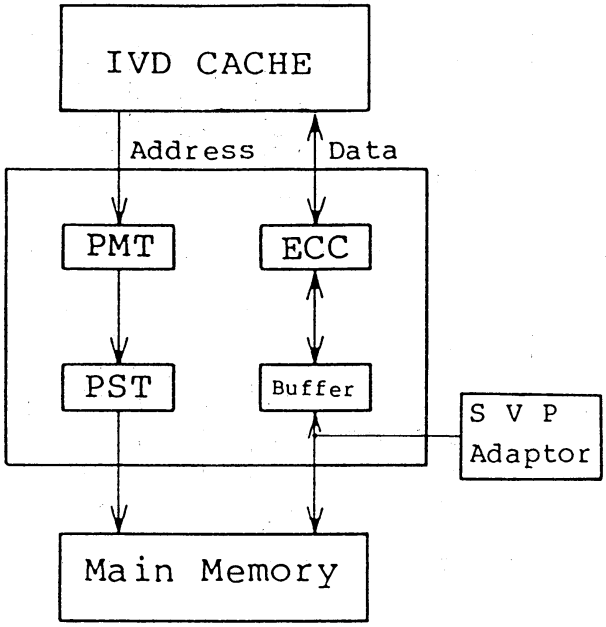
LIST processor

executes CAR, CDR, CONS, RCONS, HCONS, RPLACA, RPLACD, LIST2, CADR, and CDDR.

WCS (Writable Control Storage)

The size and width are 1024 by 150 bits and 256 by 50 bits. The access time is 50 ns per micro cycle.

Appendix 3.3. The Block Diagram of MCU



PMT (Page Mapping Table)

Consists of 2560 entries of 16 bit virtual memory address as a key and 15 bit physical address as a mapped value. A 10 bank parallel hashing hardware within 50 ns is used for searching.

PST (Page Status Table)

Consists of 2048 entries of physical page status (15 bits, 9 bits for on-cache block counter and others for status flags such as resident, modified and valid).