AN ADAPTABLE LISP MACHINE BASED ON MICRO PROCESSORS

Masayuki Ida and Koutaro Mano

Dept. of Industrial and Systems Engineering, College of Science and Engineering,
Aoyama Gakuin University
6-16-1 Chitosedai, Setagaya, Tokyo,157 Japan

A micro processor based Lisp machine for practical use is described. The machine named ALPS/I has been working for several years. ALPS/I is the first outcome of our research, in which we seek the high utility and adaptation of Lisp-based systems and its hardware, along with the progression of micro processor technologies.

The memory space of the micro processor in ALPS/I is expanded and hierarchically splitted into two 32k partitions. The lower partition is used as usual byte-accessible memory, but the higher partition is not. The latter is used to access the 64k word(35bits each) bulk memory for holding all the Lisp data. On transferring data between the bulk memory and the lower partition, the address of the former must be arbitrarily specified, but the latter may be limitted for several locations, by the virtue of the Lisp characteristics. Then we designed an interface circuits composed of 16 address mapping registers named AAM. Using our interface, only a few instructions of micro processor are necessary to transfer data between bulk memory and the buffers in lower partition.

INTRODUCTION

Lisp, a symbol manipulation language, is widely used in artificial intelligence fields. As one of the Lisp characteristics, it is well known that large storage capacity is necessary to cope with meaningfull applications. Consequently, most popular Lisp systems are implemented on the large-scale general purpose machines. We had desired the free access Lisp system for many years, and tried to make a Lisp system using micro processors.

In constructing the system, we considered the followings regarding Lisp characteristics and user availability:

1) No overhead to start, and free access.

2) Lower cost and compactness. For extensibility and maintainability, it is desired to use the standard components. Micro processors seem for us to be extensible components to build adaptable computer systems. As a result, lower cost and compact implementation are also realized. And along with the technological improvement of hardware, we expect the growth of our system.

3) Large storage capacity. The size of main memory is cruicial for Lisp applications, except for educational use.

4) Many functions and facilities to assist the users.

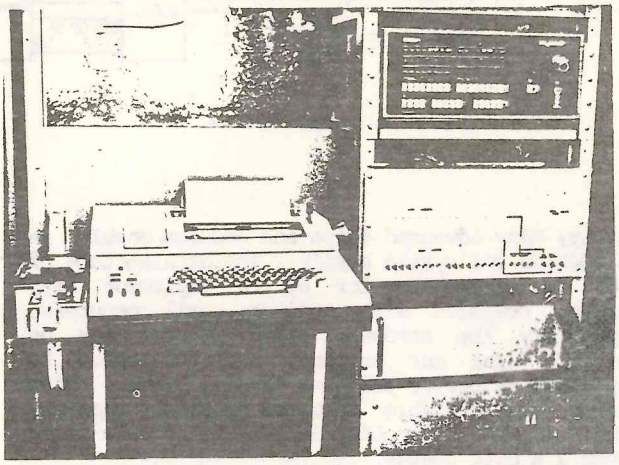

Fig.1 ALPS/I System

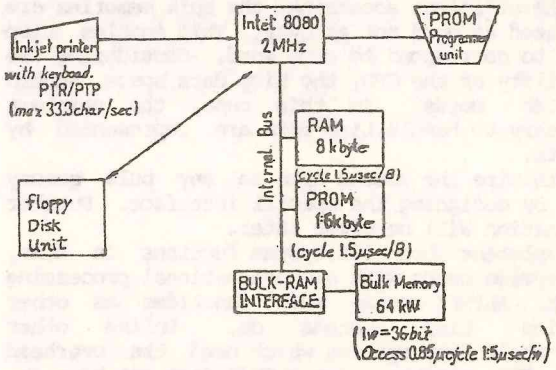

Fig. 2 Hardware configuration.

According to these principles, we designed and implemented a Lisp machine, named ALPS/I (Aoyama List Processing System/I). ALPS/I has been working for several years. And as one application, formula manipulation language REDUCE-2, which is written in Lisp, is running.

HARDWARE ORGANIZATION OF ALPS/I

Design Principles — Employing an 8 bit Micro Processor

According to the general requirments, following basic considerations are fixed:

1) Employ an 8 bit micro processor as a control unit for it's popularity and adaptability.

CH1474-6/79/0000-0210$00.75 © 1979 IEEE

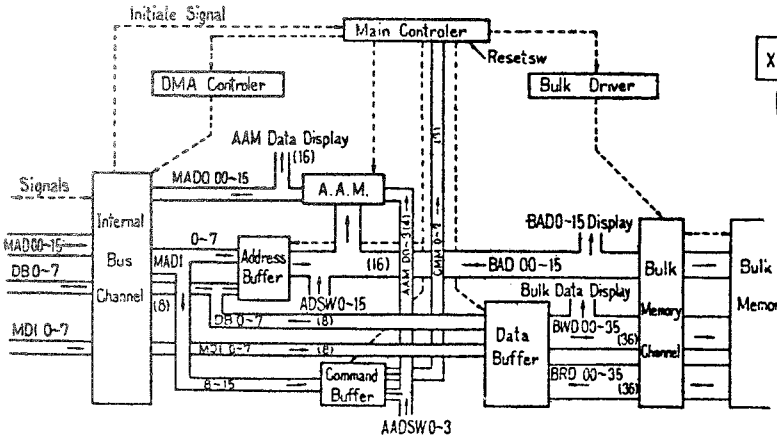

Fig. 3 Bulk-RAM interface.

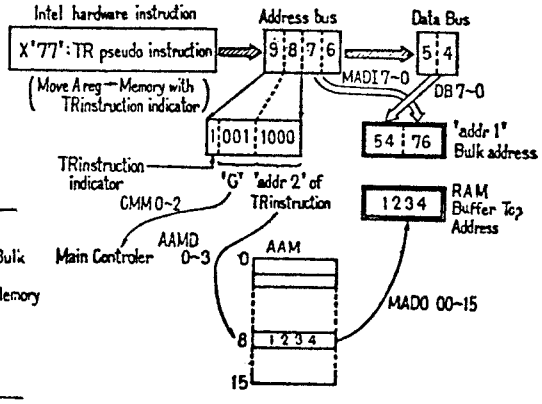

Fig. 4 An example of the formation of bulk and RAM addresses from TR pseudo instruction.

Lately, many advanced chips and modules enable us to upgrade our system easily. The architecture of 8080 is preferable for us to implement Lisp system, compared with ordinary small computers. Especially, the hardware stack pointer may be suitable for our purpose, and the speed to implement recursive programs is also fast. Lisp seems more suitable to micro processor than any other language. The Lisp interpreter module needs about 2 k byte spaces.

2) Employ bulk IC memories to store Lisp data. For the efficient accessing, the bulk memories are organized as word not as byte. This enables Lisp cell to correspond to each word. Considering the capability of the CPU, the Lisp data space should be 64k words. In this case, the pointers necessary to handle Lisp data are represented by 16 bits.

3) Minimize the access time to any bulk memory word by designing the special interface. Further explanation will be given later.

4) Implement about 100 system fuctions in ROMs, and provide users with a conversational processing system. ALPS/I has as much functions as other existing Lisp systems do. Unlike other large-scale Lisp systems which need the overhead under TSS monitor, our ALPS/I does not have it. Thus ALPS/I's response time can be comparable with other Lisp systems.

ALPS/I system consists of an 8 bit micro processor (Intel 8080), EPROM 16 k bytes (initially 1702, currently 2716), RAM 8k bytes, an inkjet printer (with ptp/ptr), a floppy disk unit and 64k word bulk memory (1 word = 35 data bits and one parity bit), (Fig.1,Fig.2). On the stage of designing in 1974, there was no peripheral chips like 825x nor single-board computer. To keep extensibility, we used and modified INTELLEC 8/MOD 80, which was only the system development support tool sold at a market then. We also made many board module by hands.

Expansion of memory space by splitting into a hierarchical organization

In Lisp environment, all the data are identified and processed by pointers. Pointers for numerals and atomic symbols are required to be unique. And the binary tree representation is merely an orderd pair of two pointers.

Then the design of memory space allocation and hierarchy affects the performance and speed of the system. On ALPS/I, the original space of micro processor is splitted into two spaces of 32k each. These two spaces have different bit-width each other. The lower 32k space contains EPROM 16kB and RAM 8kB to hold the system program and Lisp processor (interpreter). The higher partition is not a statically accessed storage. Accessing the higher partition activates the special interface between bulk word containing Lisp data and 5 byte RAM (Fig.4). A bulk memory is accessed by a few instructions as described later.

Any word on bulk memory consists of 35 data bits and one parity bit. Data bits consist of 32 Lisp data bits and 3 flag bits. By using 8080's standard instructions, these words can not be handled. Any data on bulk memory is transferred to 5 byte RAM area named bulk-buffer, and then processed by CPU. We added pseudo instruction to transfer data. By using this instruction, rapid access has been available. It generates the commands and informations which are necessary to transfer data. It only needs 5 clock to trigger the interface circuit. (Fig.3,Fig.4)

On transferring data, bulk address must be arbitrarily specified. But RAM address may be limitted for several locations, by virtue of the Lisp characteristics, that is:

1) 4 arguments for system built-in functions are required at maximum.

2) To process serial operation for linked list traversing or stack look up, at most 2 buffers for each are enough.

3) Recursive call is not used very often in

system built-in functions (SUBR,FSUBR).

Reckoning with them, we determined that 16 buffers used simultaneously are enough. And by assuming the arguments for SUBRs are on bulk-buffers, we can minimize the load of secondary data transfer to other RAM area.

The interface which has been actually made has following characteristics:

1) It has AAM (Address Association Memory, 16wx16bits) as RAM buffer address translation registers. On transferring time, AAM register number is specified in the instruction as a RAM buffer address.

2) We consider that following TR instruction is necessary.

TR c,addr1,addr2

c represents transferring command. c =1 : (bulk [addr1]) -> (ram[AAM [addr2]], ... ,ram[AAM [addr2] +4]) ; c=2 then inverse transfer (ram -> bulk); c=3: addr1 -> AAM [addr2];

To implement this instruction virtually, memory write operation for upper half partition is used (Fig.4). With this interface, the transferring time between a bulk word (memory cycle time is 0.85 micro second/word) and 5 RAM bytes (1.0 micro second/byte) takes about 20.5 microseconds. It is comparable to the time to ordinary 16bit transfer in RAM.

Interface circuit always monitors the memory-write signal upon higher 32k partition (memory write cycle with MADI15 is one). If the interface circuit detects such case, it latches the data bus (DB07-00) and the address bus (MADI15-00). DB07-00 and MADI07-00 form the bulk address (BAD15-00). MADI14-12 becomes the command(CMM0-2). MADI11-08 becomes the AAM address(AAMD03-00).

3) It has an augumented data buffer. It enables byte-width access for internal bus and 36bit-width access for bulk memory driver. On transferring data between RAM and bulk, parity generation and checking processes occur. The lower 4 bits of fifth byte are always discarded. Interface is dynamically stopped for wrong parity data.

4) It provides it's own maintainance panel.

Bulk read/write instruction examples are on Fig.5. It shows the way of transfer using AAM1 and AAM2. On GBULK1 macro definition, it has one argument named COM. COM should be an immediate value for the concatenation of c and addr2.

x'9n' is read operation with AAM n.
x'An' is write operation with AAM n.
x'Cn' is set AAM operation with AAM n.

On entry of GBULK1 macro, H-L register is assumed to contain addr1 of TR. H-L register is saved on execution.

A Portable Floppy Equipment

An 8 inch floppy unit is connected through the two 8 bit parallel input ports and two 8 bit parallel output ports. The controller consists of 256 byte RAM(128x2), command interpreter circuit, LSI floppy controller chip for IBM 3740-type format. The data on a diskette may be asynchronously transferred between a sector and either two RAM areas on the controller. Each RAM byte may be randomly accessible and may be randomly filled by the data from CPU. There are 14 commands. The interface was built by Mori and reported in [Ida79B].

```
GBULK1  MACRO  COM
        MOV    A,H
        MVI    H,COM
        MOV    M,A ; INITIATE BULK INTERFACE
        MOV    H,A ; RESTORE H REGISTER
        ENDM
        .
        .
        .
        LXI    H,1000 ;  SET AAM1 AND AAM2
        GBULK1 0C1H   ; IN ACTUAL ALPS/I-LISP,
        LXI    H,1500 ; AAM IS SET IN THE INIT
        GBULK1 0C2H   ;   ROUTINE ONLY.
        .             ; 16 BULK-BUFFERS ARE FIXED
        .             ;  AND THEN NAMED BULKBUF,
        .             ;  BULKBUF2,ZZARG1-ZZARG14
        LXI    H,2000
        GBULK1 0A1H ; RAM(1000-1004) TO BULK(2000)
        GBULK1 92H  ; BULK(2000) TO RAM(1500-1504)
        LXI    H,3000
        GBULK1 91H  ; BULK(3000) TO RAM(1000-1004)
        .
        .
        .
```

FIG.5 EXAMPLES OF THE BULK COMMANDS

Lisp Interpreter on ALPS/I

Its characteristics and Programming Conventions

On programming the Lisp interpreter, following conventions are used:

1) Use D-E register of 8080 as a function value register. All other registers serve as working registers except for some nucleus functions. H-L register is most useful and should be a working register. D-E register is the next usefull and is exchanged easily with H-L by XCHG instruction.

2) Do not save and restore registers on every subprocedure entry and return. Use PUSH and POP instructions only in the routines, which calls another routine and keeps local value in registers. So, the overhead of procedure calling is minimized.

3) Design modularly as much as possible. System subprocedure and functions (SUBR, FSUBR) consist of about 200 modules, which can be independently assembled. Use stacks instead of explicitly reserved working storages to minimize cpu-time and storages for work variables. ALPS/I has only 60 global work variables for about 12k byte instructions.

4) Use machine instruction effectively. The PUSH, POP, XCHG, XTHL instructions are used as much as possisble. By doing so, CALL instruction enables us to make almost all the programs be recursive. We can easily update them to re-entrant modules. The INX and DCX instructions are used to 16bit-width counting. Zero test is achieved in a few steps such as, MOV A,H;ORA L; JZ jzero; which mean if H-L register is zero then

jump to jzero. The RST(restart) instruction is used instead of CALL instruction for the frequently used system subprocedures to minimize cpu-time and spaces. Console output routine, value save routine, EVAL function (Lisp interpreter nucleus), and console interrupt routine are resident in RST-area. In any module, values are kept in register as long as possible.

5) Make the canonical order of Lisp data correspond to the bulk storage assignments. The attribute of each Lisp data is distinguished rapidly in a few steps. For example, consider ATOM function, one of the Lisp basic predicates. It has one argument. It returns true if the argument is atomic else false. Assuming atomic or non-atomic boundary is specified by BOUND, the following steps check the contents of H-L register; determine it's attribute; give the true/ false(nil in Lisp) into D-E register as value.

```
ATOM definition
        LHLD   ZZADR1 ; LOAD 1ST ARGUMENT
        MOV    A,H    ;   INTO H-L
        CPI    BOUND/256 ;BOUND assumed that
        JC     SETTRUE   ;256 byte boundary
        JMP    SETNIL
        .
        .
        .
SETTRUE LXI    D,TRUE
        RET
SETNIL  LXI    D,NIL
        RET
```

In this procedure, TRUE is the pointer to the location containing the T atom. NIL is the pointer to the NIL atom. Each value is determined at the stage of system generation, because they are always loaded to the same location in the initialization routine according to the hashing fuction.

6) Use bulk-buffers as fixed argument registers. Arguments for system built-in functions (SUBR,FSUBR) may be any data on bulk memory and they are specified by the locations, i.e. pointers. According to our bulk-RAM interface design, 16 buffers can be recognized simultaneously by the interface (arguments for FSUBR are grouped as one). We left two buffers for system works. Then 14 arguments may be applied to system functions.

7) Separate a save area for the data to be guarded, from 8080's stack. If we use 8080's stack as a save area for return address and temporary data, the processing speed would be maximized. But, temporary Lisp data should be guarded against the garbage collector. If they are in 8080's stack, much time would be necessary to distinguish them from other saved data. Then, we provide a stack on bulk to store pointers to temporary value which is necessary to be guarded. SUBR's argments are automatically stacked by EVAL or APPLY routine. So, in system built-in functions, only list-constructing functions need to save the temporary value explicitly in the area. When garbage collection occurs, garbage collector is activated by the cons-nucleus. It scans and marks the current cons item, variable binding stack, the value field of atomic element and this stack area. Then unmarked words are reclaimed.

8) Use macros as much as possible. 20 macros are used.

Development processes

To write system programs, we implemented cross softwares such as assembler, simulator, converter and editor on IBM 370 and a mini computer. All the programs(about 12k steps) are debugged on s370. The construction of ALPS/I was completed and the first version of ALPS/I-LISP was released on Jan.1975, and worked well immediately. To implement REDUCE-2, which needs about 50k words to hold, we also used the simulator for initial loading. After loading it into a floppy diskette, conversational facilities of ALPS/I-LISP and a floppy-base editing system has been used to improve the implementation.

The Structure of the Lisp interpreter

The definition of the interpreter may be written in Lisp, which was already shown in [Ida79A]. It's major characteristics are as follows:

1) Top level function is evalquote.

2) The informations which need the uniqueness are gathered into the same group, and have the same format. They are stored in the area named H-area. External representation of the data on H-area is converted to the internal representation by the hashing method. The structure of the data in H-area (H-element) is shown in Fig.6.

H-elements are splitted into two sub-groups, shown in Table 1. In our format, atomic symbols cannot have property-list. But associators, the more convinient facilities, are equipped. We can find the value of an associator by the cost of O(1). Examples are shown in Fig.6 D-F. There are three types of associators: Harray element is for property-list and sparse array, assoccomp associator acts as short-term memories for function values (assoccomp idea was derived from [Got74]), prog label associator is for fast label processing by prog-interpreter. Hashing collision is resolved by rehashing. We examined several rehashing algorithms. Then, we found that quadratic rehash on power of 2 table is suitable. Thus, H-area is organized as the power of 2 size, since folding the value into power of 2 size can be performmed by masking operation, but for prime number table size, 16 bit address comparison and some calculation or division would be necessary.

3) For variable binding, a stack is used.

4) System function 'evlis' is defined as follows: evlis[m] = [null[m] -> *last := NIL; null[cdr[m]] -> *last := cons[eval[car[m]]; NIL]; T -> cons [eval[car[m]]; evlis [cdr[m]]]] ; the *last is used to indicate the last car-element's node, else NIL. By appending *last to the tail of the free-list, the created list for actual arguments for SUBR is immediately collected after apply-ed. Therefore, garbage collector time decreases. Some examples shows that about 20% of used cells are

	ATOMIC SYMBOLS								ASSOCIATORS		
ATTRIBUTE NAME	-	SUBR	FSUBR	EXPR	FEXPR	APVAL	HEXPR	HARRAY	HARRAY ELEMENT	ASSOCCOMP ASSOCIATOR	PROGLABEL ASSOCIATOR
ATTRIBUTE VALUE	0	1	2	3	4	5	6	7	8	9	10
CREATED BY	READ ETC.	-	-	DEFINE DEFLIST	DEFLIST	CSET CSETQ	DEFLIST ASSOCCOMP	ARRAY	SETA	HEXPR FUNC. REFER	PROG LABEL
DELETED BY	GGC	-	-	-	-	-	-	DEARRAY	DELETE DEARRAY	GGC	GGC

TABLE 1 H-ELEMENT PROPERTIES

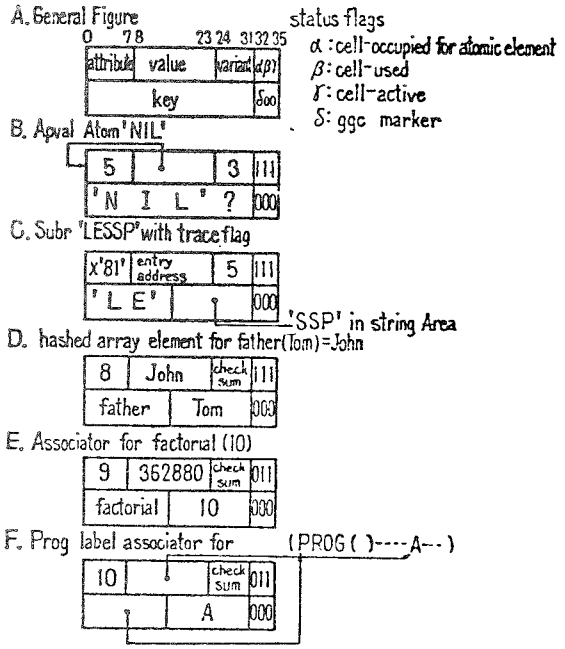

Fig. 6 Data structue of H-molecule and its examples.

reclaimed by it. The basic idea was also derived from [Got74]. Evlis coding is shown as a sample in Fig.7.

5) Lisp 1.6-type prog interpreter is implemented with label associator and pseudo compiler. Prog interpreter consists of two phases. The first is code decomposition into the intermediate code area. It creates declaration statement for prog variables. Labels are registered on H-area. This phase occurs on the first execution only. The second is the interpretation of the codes in that area. Go-function which has an atomic element i.e. unconditional jump, is substituted to direct transferring function, **go, in the intermediate area. The definitions of go and **go are shown in Fig.8.

6) Loading facility of machine language subroutine is implemented. (Note that all the software is written in ROM.)

Other characteristics were written in [Ida79A].

```
;
;    ******* EVLIS *******
;
EVLIS   EQU     $
        LHLD    ZZADR1
        HIFNIL  EL001
        TVSAVE
        CALL    GETCELL
        TVSAVE    ; SAVE TOP ADDR. FOR VALUE
        PUSH    H
EL000   EQU     $
        LHLD    ZZARG1+2
        HIFNIL  EL002
        PUSH    H
        CALL    GETCELL
        PUSH    H
        LHLD    ZZARG1
        SHLD    ZZADR1
        GBULK1  91H
        EVALE     ; EVAL[CAR[M]]
        XCHG
        SHLD    BULKBUF2
        POP     H
        POP     D
        SHLD    BULKBUF2+2
        XRA     A
        STA     BULKBUF2+4
        XTHL      ; GET ADDR. AND SAVE CDR
        GBULK1  0AFH ; CONS
        XCHG
        GBULK1  91H
        JMP     EL000
EL002   EQU     $ ;  CDR[M] IS NIL
        LHLD    ZZARG1
        SHLD    ZZADR1
        GBULK1  91H
        EVALE     ; EVAL[CAR[M]]
        XCHG
        SHLD    BULKBUF2
        LXI     H,NIL
        SHLD    BULKBUF2+2
        XRA     A
        STA     BULKBUF2+4
        POP     H
        GBULK1  0AFH ; CONS[EVAL[...],NIL]
        SHLD    ELLAST ; FOR I.G.C.
        TVRESTOR ; TOP ADDR. FOR VALUE
        XCHG     ; SET IT TO VALUE REGISTER
        TVRESTOR
        SHLD    ZZADR1
        GBULK1  91H
        RET
EL001   EQU     $  ; NULL[M]->*LAST:=NIL
        SHLD    ELLAST
        XCHG
        RET
        .
        .
ELLAST  DS      2 ; MAY LINK FREELIST
ZZADR1  DS      2 ; THE ADDR. OF ARG1
ZZARG1  DS      5 ; THE VALUE OF ARG1
        .
        .
        .
```

FIG.7 EVLIS MODULE

CONCLUSIONS AND THE MEASUREMENTS

By virtue of the adaptability of this system, the components of ALPS/I hardware have been modified and replaced easily along with the technological progressions. For example, the 1702 PROM module was replaced by the 2716 module, the cassette tape unit by the floppy unit. System programs are also updated. The current version is about 2 times faster than the initial version.

Table 2 shows the processing time for some sample programs, derived from Chang and Lee's theorm prover[2] with some other Lisp of large machines. Execution time on ALPS/I in Table 2 is nearly the same as the response time for users. We may run REDUCE-2 on ALPS/I, and it shows ALPS/I-LISP'S compatibility and utility. A record of conversational use of REDUCE-2 on ALPS/I is shown in Fig.9. Loading time of Reduce is about 10 minutes. It is automatically loaded from the floppy unit with the help of bootstrapping paper tape. To solve the problem in Fig.9, over several hours seem to be required by hand, but about 10 minutes is necessary on ALPS/I. The user level compatibility of our Lisp language to others is shown by the fact that REDUCE-2 is running on ALPS/I.

```
go[x]=[atom[x]→[addr=hassoc [x; pform]→
                progn [change-statement [**go; addr];
                       seq-counter=addr];
               T→label-error[]];
       addr=hassoc [eval [x]; pform]→addr;
       T→labe-error []
**go [x]=setq [seq-counter; x]
```

pform is an atom, whose value is a current prog-body top address, seq-counter is an atom to indicate a next prog statement to be eval-ed.

Fig.8 go and **go function.

```
REDUCE 2 (MAR-21-79) ....

OFF ALLFAC;
ARRAY H(20);
H(0):=1$
H(1):=2*X$
FOR N:=1:10 DO BEGIN H(N+1):=2*X*H(N)-2*N*H(N-1);
 WRITE "H(",N,") = ",H(N) END;
H(1) = 2*X

                  2
H(2) = 4*X  - 2

                  3
H(3) = 8*X  - 12*X

                  4       2
H(4) = 16*X  - 48*X  + 12

                  5       3
H(5) = 32*X  - 160*X  + 120*X

                  6       4       2
H(6) = 64*X  - 480*X  + 720*X  - 120

                  7       5       3
H(7) = 128*X  - 1344*X  + 3360*X  - 1680*X

                  8       6       4       2
H(8) = 256*X  - 3584*X  + 13440*X  - 13440*X  + 1680

                  9       7       5       3
H(9) = 512*X  - 9216*X  + 48384*X  - 80640*X  + 30240*X

                  10      8       6       4       2
H(10) = 1024*X  - 23040*X  + 161280*X  - 403200*X  + 302400*X  - 30240
```

Fig.9 Hermite Polynomials from 1st order to 10th order by the use of the REDUCE-2 on ALPS/I-LISP

Acknowledgement

The autors would like to express our gratitude to many people who helped and encouraged us, especially to Professor Goto of Tokyo university for his helpful suggestions; to Professor Mori of University of Tsukuba for his kind guidance to prepare the paper. We also thank students and fellows who assisted us to develop and maintain the ALPS/I system.

REFERENCES

1. [Cha70] C.L.CHANG; The Unit proof anf the Input proof in Theorem Proving, J.ACM, Vol 17, No.4, pp698-707, Oct. 1970
2. [Cha73] C.L.CHANG and R.C.T.LEE: Symbolic logic and Mechanical Theorem Proving, 1973, Academic Press
3. [Got74] E.Goto: Monocopy and Associative algorithms in an extended Lisp, Technical Report No 74-03,Tokyo University, may 1974
4. [Hea73] A.C.Hearn: Reduce-2 User's Manual; Univ. of Utah, March 1973
5. [Hop72] F.Hopgood and J.Davenport:The quadratic hash method when the table size is a power of 2; Computer Journal, Vol.15, No.4, pp314-315, 1972
6. [Ida79A] M.Ida and K.Mano:A Lisp machine based on a micro processor, J.of IPSJ, Vol 20, No 2, pp113-121, March 1979. (in Japanese)
7. [Ida79B] M.Ida,Y.Mori,S.Kawai and T.Tomine : Recent Topics on ALPS Project, preprint of WGSYM, 7-1, IPSJ, March 1979. (in Japanese)
8. [Qua70] L.H.Quam and W.Diffie ; Stanford LISP 1.6 manual, Stanford AI Labs. Operating Note, No 28.6,1970
9. [Tak78] I.Takeuchi:The report of a Lisp contest, preprint of WGSYM 5-3 IPSJ, August 1978

CPU	ALPS/I 8080	Ll.6* PDP-10	HLISP** H-8800	UTLISP4.1*** CDC6600
TPU-1	87.0	1.55	4.79	24.64 SEC.
TPU-2	390.0	7.51	13.41	70.71
TPU-3	157.0	3.42	5.68	29.43
TPU-4	194.0	2.65	8.03	41.52
TPU-5	28.0	0.67	0.87	4.38
TPU-6	820.0	2.30	22.85	123.2
TPU-7	150.0	3.56	5.08	26.84
TPU-8	142.0	5.18	3.46	18.68
TPU-9	133.0	4.05	2.66	13.78

* ORIGINAL PROGRAM'S DATA[1]
** TOKYO UNIVERSITY'S LISP[3]
*** UNIVERSITY OF TEXAS'S LISP[9]
DATA FROM [TAK78] EXCEPT Ll.6

TABLE 2 EXECUTION TIME OF SAMPLE PROGRAMS