DIRECT EXECUTION LISP AND CELL MEMORY

Y. P. Chiang and M. L. Manwaring

Department of Electrical and Computer Engineering
Washington State University, Pullman, WA 99164

Abstract:

Lisp is the second oldest programming language in use today. It was treated as a special language for AI applications until the recent trend towards symbolic processing. The advantages of Lisp in processing symbols sometimes are outweighted by the inefficient implementations. This paper presents a direct execution approach in implementing Lisp. The concept of direct execution is explained. The Lisp execution environment is analyzed. The cell memory architecture is presented as the efficient solution for direct execution Lisp. This memory structure reduces the number of bits required for implementing garbage collection algorithm and the number of memory cycles for Lisp programs. Several examples are given to elaborate the direct execution concept and the advantages of cell memory. This direct execution Lisp machine has built-in functions such as interrupt and I/O which are major functions for real-time application.

1. INTRODUCTION

Lisp is the second oldest programming language in use today. Unlike conventional programming languages, Lisp was designed for computing with symbolic expressions rather than numbers. In the 60's and 70's, computers have been considered as number crunching machines. Some powerful computers emphasizing efficient and parallel computation have been built during that period. These machines typically execute instructions in the range of hundreds of million floating point operations per second. During the same period, Lisp was used primarily by AI researchers. In the 80's, computers have been treated not only as number crunchers but also as machines with intelligence. This new role of computers creates the needs for symbolic computing. With its rich capabilities in expressing information and processing lists, Lisp has emerged as one of the popular programming languages among both AI and non-AI programmers.

It is apparent that conventional computers are not efficient for symbolic processing. Starting from mid 70's, researchers in the area of computer architecture begin their pursuit of computing machines which are ideal for symbol manipulation, in particular, executing Lisp. An experimental Lisp machine had been developed at Kobe University and Hitachi Ltd., Japan [Taki79]. This machine consists of a Lisp processor module and shared main memory module connected to the UNIBUS of DEC LSI-11. The Lisp processor is microprogrammable. There is a high speed stack for list processing. Tag is used in this system to distinguish between three different data types. The NK3 Lisp machine, developed at Kyoto University [Naga79], also uses microcode, hardware stack, and the tagged data structure. The ALPHA machine which had been developed at the Fujitsu Labs [Haya83] uses a highly effective stack which can support a value cache, a virtual stack and high speed garbage collection algorithm for virtual memory. This hardware stack also supports high speed process switching in a multi-process environment.

At MIT several implementations of Lisp had been carried out. The most notable ones are the SCHEME-79 chip [Suss81] and the SCHEME-81 system and chip [Bata82]. SCHEME is a subset of Lisp. It uses lexical scoping and allows tail-recursion. The SCHEME chip implemented on a standard Von Neumann architecture with microcode. It also uses stack and tag. The major impact of SCHEME is that the whole Lisp interpreting environment is implemented on a chip. Several commercial companies adopted this approach and produced Lisp machines. For example, Symbolics 3600 [Moon85] is the modified version of another MIT product, MIT Lisp machine [Bawd79]. The LAMBDA machine from Lisp Machines Inc. uses a Lisp processor running a 20 MHz clock, and a 32-bit word with 24-bit virtual address space. These machines all have similar architectures and implementations to that of SCHEME chip. With the similar implementation method and the advanced IC technology, the current Lisp machines such as the Symbolics 3610, the XEROX 1132, and the Texas Instrument Compact Lisp machine [Matt87], are running faster than ever. Lisp has also been implemented on parallel computers such as Connection machine and Butterfly computer. Most of the Lisp processors were taking a conventional approach in their architecture design. That is, the implementation issues were first fully analyzed, then the proper hardware or software were designed to accommodate those time consuming, critical issues. As a result, a micro-level machine was designed as the core of the system, extra stack and tagged data structure were used

to provide efficient execution environment for Lisp. Like all the other programming languages, the execution of Lisp program requires two steps. First, Lisp programs are translated into low level machine code. Second, the machine code is executed by the hardware processor. Although some machine code has the same functions as those defined in Lisp, the whole execution process still requires machine code translation before the program can run. If Lisp is the preferred language because of its abilities in expressing and manipulating information, then any other languages will not be able to do the same things effectively. The extra language translation may not only increases the processing time but also loses the advantages of the Lisp language. This paper presents a direct execution approach for designing Lisp machine. Direct execution means the elimination of the machine code translation process. The notion of direct execution language (DEL) has been presented in [Chu81]. It is understood that DEL is able to obtain direct and effective support from its architecture. However, it is not easy to find such a DEL. We would like to argue that for each given programming language it is possible to design an architecture which directly execute this language. This approach may not viable for general purpose processor, but, for dedicated processor, it becomes very attractive. In the following sections, we first discuss the execution environment of Lisp. The concept of direct execution is explained and its implementation method is presented. The cell memory architecture is presented as the architecture for direct execution Lisp machine. Several examples are used to demonstrate the advantages and the implementations.

2. LISP EXECUTION ENVIRONMENT

The process of executing Lisp program is one of interpretation. This means that a complete Lisp list is composed or assembled and then presented for execution. Its results are then printed, and the process is done again on the next list in the program. This interpretive process is an abstract fetch-execute cycle. Using Lisp itself to describe the process, the following Lisp statement is executed continuously:

(PRINT (EVAL (READ))).

This Lisp cycle has remarkable resemblance to the low level machine instruction cycle except the machine instruction becomes list or list of lists. One may ask, can we use Lisp as our assembly language? The answer is yes. However, because of the recursive nature of Lisp, a novel architecture is inevitable for the core processor.

From the language theory perspective the processor is a sequential machine which recognizes its own machine (assembly) language. The complexity or the power of the processor is manifested through its machine language. If the machine language is of type 0 (regular) language, then a finite state (regular) machine is adequate for recognizing it. Finite state machine is the simplest type of sequential machine. Most of the current general purpose or the micro level Lisp processors are of this type. Since Lisp or other high level programming language is subset of type 1 (context free) language, a pushdown automaton is required for recognizing such language. A finite state machine plus a stack is able to mimic the operations of a pushdown automaton. A typical example is that when a Pascal program is executed, the micro level processor has to allocate stack or heap in order to do it. Therefore, if Lisp is the assembly language, then the processor has to be a pushdown automaton or a finite state machine with built-in stack.

Recognizing or accepting a language is formally called parsing. The parsing process is straight forward for regular languages since the number of states involved may be as little as two states, fetch and execute. The parsing for context free languages is nontrivial at all. There are many parsing methods described in Aho and Ullman's book [Aho72]. The algorithm which is most suitable for programming languages and hardware implementation is called the syntax directed recursive decent algorithm. Detailed description about this algorithm is given in [Back79]. The hardware implementation of this algorithm on a stack based architecture for direct execution BASIC is reported in [Srid83]. Although the same design methodology is used in this paper for the direct execution Lisp processor, the capabilities of manipulating list structure makes this processor a unique one.

If one takes away the "read" and "print" functions which are analogous to the fetch cycle of a machine instruction, then the execute cycle of Lisp is simply the function "eval". As a matter of fact, "eval" is the major function of any Lisp machine. The "read" and "print" functions can be written in Lisp and executed by the eval function. Consequently, the major effort of a Lisp machine design is on designing an eval processor.

3. IMPLEMENTATION ISSUES

By taking the language directed design method, the first task becomes specifying the grammar. The grammar for Lisp has fairly simple syntax. Nevertheless, the language is recursive and that makes it a context free language. A tentative version of the grammar for Lisp is given in the following. It is expressed in BNF.

```
<initial expression> ::= <expr>
<expr> ::= nil | <atom> | <list> |<value> |
  <func>
<atom> ::= <alphanumeric> | <alphanumeric>
  <atom>
<list> ::= (<expr>)
<value> ::= <numeric> | <numeric> <value>
<func> ::= <lambda form> |<defined func>
```

<lambda form> ::= ((lambda (<parms>) <func list>)
  <arg list>)
<parms> ::= <atom> | <atom> <parms>
<func list> ::= <func> | <func> <func list>
<arg list> ::= <exp> | <exp> <arg list>
<defined func> ::= (<atom> <arg list>)

In BNF, everything within the square brackets is nonterminal symbol. Otherwise, it is a terminal symbol which the machine can recognize. The set of terminal symbols is not shown here. They include the alphanumeric, the numeric, and the built-in functions. There are not too many rules in this grammar. But, it is highly recursive. For instance, the <atom> is used to define itself. The nonterminal <func> includes the built-in functions and user supplied functions. With the language directed approach, it is very easy to add built-in functions such as interrupt or I/O operations. These functions are very difficult to implement with the conventional design methodology. A microcoded finite state machine and a stack on RAM memory were used as the pushdown automaton to recognize the language generated by this grammar. After the successful recognition, the semantics associated with that function will be carried out by the hardware data path. In order to increase the performance the components of the data path should accommodate the semantics of Lisp.

The major operations of Lisp are list handling processes. Since list is conveniently represented by linked list data structure, a large portion of Lisp functions simply manipulate pointers. This implies that a register file is needed to facilitate this type of operations. From the above grammar one can see that the basic item used in a Lisp program is the expression, or s-expression. The implementation of expression has deciding factor on the performance of the processor. Most of the existing Lisp systems use the word-oriented memory structure. That is, a word plus tag bits are fetched at a time, then the type of the expression will be determined by the value in the tag. This arrangement creates extra memory access when fetching a list. A list can be represented by a CONS cell which has CAR and CDR pointers. Fetching these two pointers takes two memory reference cycles. In fact, accessing CONS cells is the primary operation in Lisp. If the processor can accomplish the same task with less memory cycle, then it definitely will have faster speed. The cell memory structure is our solution to reduce memory cycle.

The other issue in all Lisp implementations is the memory management, or sometimes referred to as the garbage collection (GC). In the word oriented memory system, most of the GC algorithms require certain number of bits allocated for each word to store information such as colors, mark flag, or reference count. In the cell memory system, the GC information is associated with each cell which has several words instead of with each word. This means the amount of bits required to support a given garbage collection algorithm is less in the cell memory system. Also the number of garbage collection cycles becomes less when collecting the same number of words. It is believed that the cell memory is better suited than the word-oriented memory system for Lisp execution environment.

4. CELL MEMORY ARCHITECTURE

The expression as defined in the grammar could be an atom, a list, a function, a numeric value, or nil. The cell structure should be able to express all of them. In our design, each cell has the same width of 120 bits. There are four different type of cells which can be distinguished by two bits in the cell. The four cell types and their functions are outlined below:

1. ROOT cell
   This cell is used in conjunction with the ATOM cell to make up an atom. This cell holds a pointer to a corresponding ATOM cell, along with pointers point to lists which are used by the garbage collector.

2. ATOM cell
   A complete atom is formed by using this cell along with the root cell. This cell also holds pointers to the atom's function, value, property list, and print name. The reason for using two cells to represent one atom is because of the size limitation of the cell. With 120 bits, there are not enough space to store pointers to the four attributes of the atom and the lists for garbage collector.

3. CONS cell
   This is the basic Lisp cell which holds the CAR and CDR pointers. It also has pointers for the garbage collector.

4. VALUE cell
   This cell is pointed to by the print name of an atom. It is designed to hold an integer, floating point number, or a string of 10 characters along with pointers for the garbage collector.

Each cell type consists of two 32-bit words, two 24-bit words, two 2- bit tags, and one 4-bit field. The first four words are used as address pointers. The other 8 bits are used to distinguish cell type, the color of the cell, and the states of the garbage collector.

The four words within each cell are address fields. They are pointers labeled as follows:

1. ROOT cell: CAR, CDR, SP, I_USE.
2. ATOM cell: NAME, VALUE, FUNC, PROP.
3. CONS cell: CAR, CDR, SP, I_USE.
4. VALUE cell: LSW, MSW, EXW, I_USE.

The SP and I_USE pointers keep track of the stack and the in-use list. One of the advantages of the language directed design methodology is the creation of a real-time environment by introducing interrupt and I/O functions into the language. However, the garbage collection algorithm has to be real-time in order to maintain this advantage. With garbage collector running in parallel it is possible to provide a transparent GC. But, this approach greatly increases the hardware complexity. We choose an incremental sequential GC which is a mark and sweep algorithm similar to the one proposed by steel [Stee75]. The SP and I_USE fields on each cell were added in order to implement the incremental GC. The detailed operations of the garbage collector on the cell memory are reported by Hoover [Hoov87]. The CAR and CDR pointers are used as defined in Lisp. The value cell allows the storage of non-address data. It only has one I_USE pointer for garbage collector. The first two 32-bit words are the least and the most significant 32-bit words. EXW represents the extended 24-bit word. The first three words together can be used to hold a standard floating point number. The main purpose of the value cell is for I/O operations and data type representations such as fixnum, flonum, and string.

Figure 1 shows the representation of an atom which is both a variable and a function. The atom is stored in the OBLIST which is a linked list. It takes a pair of ROOT cell and ATOM cell to represent one atom. In this example, the print name of this atom is X, so the NAME pointer of the ATOM cell is pointing to a CONS cell whose CAR pointer points to a VALUE cell with character X. Similarly, the VALUE pointer of the ATOM cell points to a CONS cell which provides a list (5 10.5) as the value. VALUE cells have been used to hold an integer 5 and a floating point number 10.5. The FUNC pointer of the ATOM cell also points to a CONS cell whose CAR pointer points to a lambda form of the function. The lambda form basicly involves similar structure as an atom.

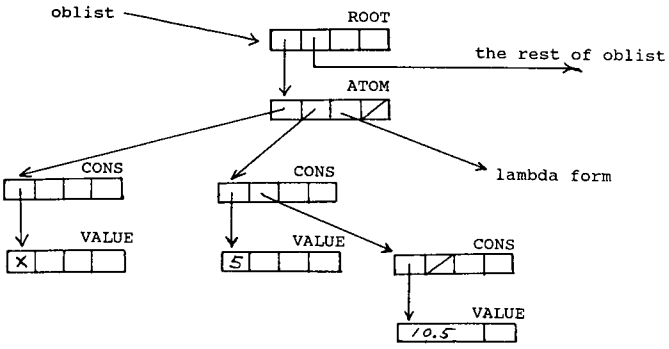

Figure 1. Representation of an ATOM.

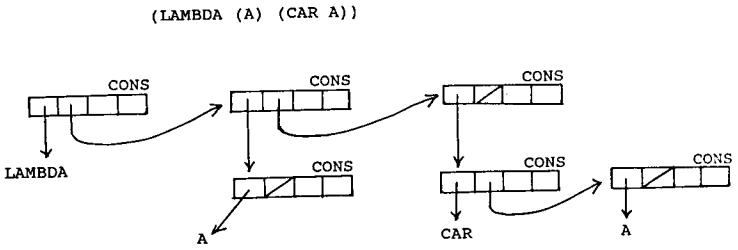

Figure 2. LAMBDA form representation.

An example representation of the function "head" is demonstrated in Figure 2. "head" is defined as the CAR of a list. Lambda form is used as the internal representation of a function. The lambda form of function "head" is (LAMBDA (A) (CAR A)). This form is represented by three linked CONS cells. The CAR pointer of the first CONS cell points to an atom structure which is not shown here. This atom structure contains the build-in function LAMBDA. The CAR pointer of the second CONS cell points to an atom which has the print name A. The last CONS cell has its CAR points to another CONS cell which contains the list (CAR A). Its CDR pointer becomes nil to signify the end of the list. The CDR pointers of the first two CONS cells are used to complete the linked list.

When a Lisp statement such as (SETQ FLG X) is entered into the system, the responses from the cell memory is described in Figure 3. First, the Lisp statement is represented by three CONS cells on the top of the Figure. The CAR pointers of these three cells point to three atoms, SETQ, FLG, and X. SETQ is a build-in function while the other two atoms have their atom structures which include ROOT and ATOM cells. Before executing the statement, the VALUE pointer of the ATOM cell for FLG may not be used or points to other value. After the execution, this pointer points to the value of X. If X is known as in Figure 1, this pointer will be pointing to the CONS cell which has a list (5 10.5) as its value.

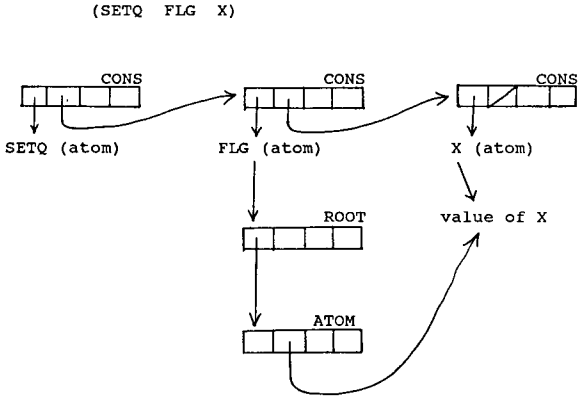

Figure 3. Example of function execution.

5. SYSTEM ARCHITECTURE

The block diagrams of the direct execution Lisp processor and the cell memory is presented in Figure 4. The Lisp processor in this figure includes a microcode storage and a specially designed chip which has 16 internal registers for fast pointer operations. The schematic diagram of the data path unit is shown in Figure 5. Besides the microprogrammable controller and the ALU unit, the 16 32-bit registers occupied most of the chip. The cell memory consists of four modules for the pointer/data fields, and 8-bit field for tag, color, and the state of the garbage collector. Each cell is referenced by one address. There are other control bits from the Lisp processor to select the modules within the cell.

The operation of the system is better understood with an execution example. In the following is the list of microcode, in its mnemonics form, for the build-in function, CAR.

```
carcmd:
    readcdr(currptr, currptr); /*get the
                        argument list*/
    ifnotnil(currptr);
        jump(carcmd1);
        jump(reperr);
carcmd1:
    readcar(currptr,currptr); /*get
                        argument*/
    eval; /*evaluate the argument*/
    ifnotnil(currptr);
        jump(carcmd2);
        jump(exiteval);
carcmd2:
    latchaddr(currptr);
    ontag;
        tjump(jaerror); /* atom */
        tjump(carcmd3); /* root */
        tjump(carcmd4); /* cons */
```

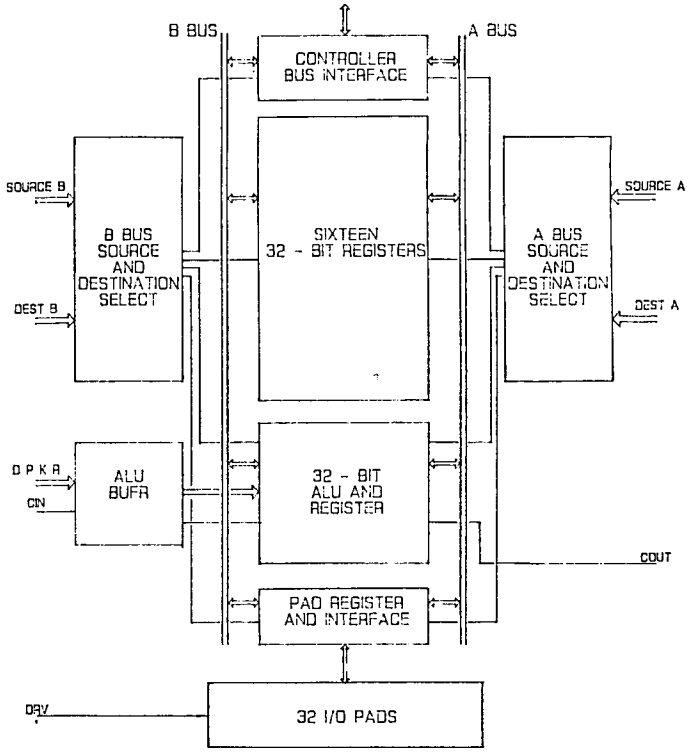

Figure 5. The schematic of the data path unit

```
carcmd3:                    /* value */
    finishread;
    setnil(currptr);
    jump(reperr);
carcmd4:
    finishread;
    readcar(currptr,currptr);
    jump(exiteval);
```

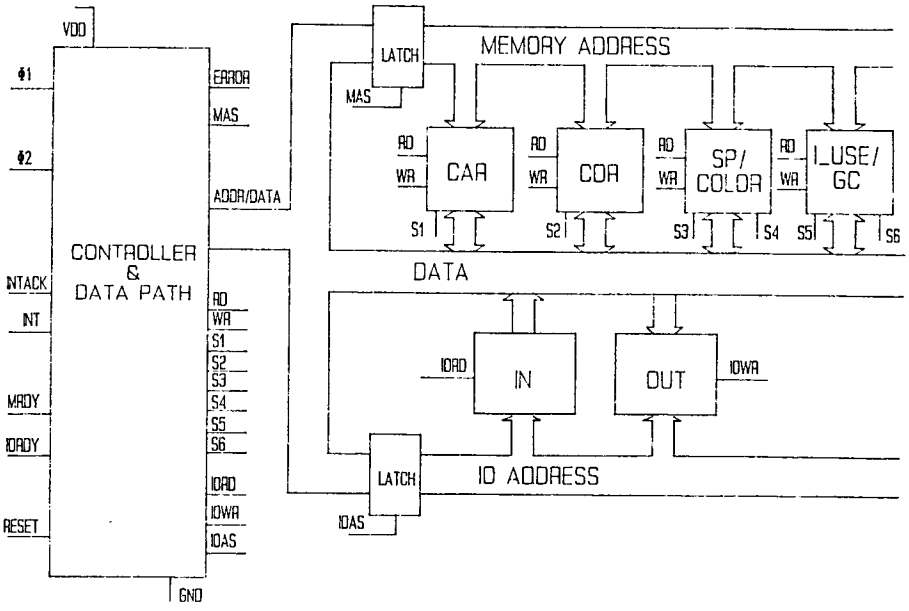

Figure 4. The Lisp system with cell memory.

This microcode sequence is part of the recursive EVAL cycle since the main function of the processor is to do eval. At the beginning of EVAL cycle, the address of a list structure is put in a register called currptr. At the end of the cycle, this register will contain the address of the resulting value. The above micro sequence will be invoked once the processor recognizes the first element in the current list is the function CAR. So the first statement is to fetch the argument list by reading in, from the same memory cell, the CDR pointer of the current list and puts it into the register currptr. Several tests are conducted following the first statement. They include the testing for empty argument list, testing for the result nil, and the testing for the proper cell type that the argument list should have. Under the carcmd1 section, the function eval is called in order to evaluate the argument. Different type of errors will be reported as the carcmd2 indicates. At the completion of the CAR operation, the CAR pointer of the argument list is returned into the register currptr. The operation, exiteval, involves the internal stack operation, POP. What POP does is to restore the previous level of the nested EVAL cycle. This means a lot of information has to be retrieved. In our current implementation, POP operation has 17 memory references to pointers. With the cell memory structure, we only need to set up the address 5 times. The other 12 memory references are to the different modules within the same cell. Similar speed up occurs in the internal PUSH operation.

Since the I/O and the interrupt are built into the language, the Lisp processor is able to communicate with I/O devices and even a math coprocessor through its address and data buses. This unique feature allows this Lisp processor to be applied to a real-time control environment.

6. CONCLUDING REMARKS

Direct execution of a programming language eliminates the process of intermediate code translation. This design methodology was examplified through the design of the Lisp processor. The print- eval-read loop of the Lisp language makes it a suitable candidate for direct execution. A microcoded finite state machine and the internal stack provide the capabilities to recognize Lisp. The precised syntax of Lisp has been defined, and a simplified version of the grammar was reported in this paper. Cell memory architecture was presented as an efficient memory structure for the direct execution Lisp. It reduces the extra bits overhead for the garbage collection and the overall memory cycles required for Lisp processing. The uniform size of the cell memory allows the memory manager not to worry about the difficulties caused by cells of various sizes. The Lisp processor chip which performs recursive decent interpretation algorithm has been designed and fabricated. A prototype system has been assembled. An IBM PC was used as the testing bed. The microcode which covers a subset of functions from Common Lisp has been designed and is currently under testing. Although it is not easy to address the performance issue between different Lisp machines, we believe our Lisp machine should have better performance. As a rough estimate, the average eval operation may take 800 to 900 clock cycles. With a 2 MHz clock, this translates into about 2200 to 2500 eval cycles per second. This is the performance with non-optimized microcode. With the technique such as late binding we can greatly improve the number of clock cycles required for each eval operation. Besides having the faster processing speed, this Lisp machine has built-in interrupt and I/O functions which open the gate for real-time application.

7. ACKNOWLEDGEMENT

This project was supported by a contract from the Boeing High Tech Center and the Boeing Military Airplane Company. The authors would like to express their appreciation towards the group of graduate students involved in this project.

REFERENCE:

[Aho72] Aho, A.V., and J.D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Back79] Backhouse, R.C., Syntax of Programming Languages, Theory and Practice, Prentice-Hall, International, 1979.

[Bata82] Batali J., E. Goodhue, C. Hanson, H. Shrobe, R.M. Stallman, and G.J. Sussman, "The SCHEME-81 Architecture -- System and Chip by the Designers" Proc. 1982 MIT Conference on Advanced Research in VLSI, Jan. 1982, PP. 69-77.

[Bawd79] Bawden A., "The LISP Machine", Artificial Intelligence: An MIT Perspective, Vol 2., by Winston P.H. and R.H. Brown, MIT Press, 1979, PP. 343-373.

[Chu81] Chu Y. and M. Abrams, "Programming Languages and Direct-Execution Computer Architecture", Computer, July 1981, PP. 22-32.

[Haya83] Hayashi, H., A. Hattori, and H. Akimoto, "ALPHA: A High Performance LISP Machine Equipped with a new Stack Structure and Garbage Collection System", Proc. the 10th International Symposium on Computer Architecture, 1983, PP. 342-348.

[Hoov87] Hoover, R. P., "An Indirect Execution Lisp Machine with a Real-Time Interrupt Response", Master Thesis, Dept. of Elec. and Comp. Engr., Washington State University, Aug. 1987.

Direct Execution Lisp and Cell Memory (Addendum)

Y.P. Chiang and M.L. Manwaring

The CAN editors inadvertently omitted the last page of references of the above paper in the September 1987 issue of the CAN (Vol. 15, No. 4, pp.52–57). It is reproduced below:

[Matt87] Matthews, G., R. Hewes, and S. Krueger, " Single-chip processor Runs Lisp Environments", Computer Design, May 1987, PP. 69-76.

[Moon85] Moon, D.A., "Architecture of the Symbolics 3600", Proc. the 12th International Symposium on Computer Architecture, 1985, PP. 76-83.

[Naga79] Nagao, M., J. Tsujii, K. Nakajima, K. Mitamura, and H. Ito, "LISP Machine NK3 and Measurement of its Performance", Proc. International Joint Conference on Artificial Intelligence, 1979, PP. 625-662.

[Stee75] Steel, Guy L. Jr., "Multiprocessing Compactifying Garbage Collection", Communications of the ACM, Vol. 18 No. 9, Sept. 1975, PP. 495-508.

[Srid83] Sridhar, R. "A Direct Execution Architecture for BASIC", Master Thesis, Dept. of Elec. and Comp. Engr., Washington State University, Feb. 1983.

[Suss81] Sussman, G.J., J. Holloway, G.L. Steel, Jr., and A. Bell, "SCHEME-79 LISP on a Chip", Computer, Vol. 14, July 1981, PP. 10-21.

[Taki79] Taki, K., Y. Kaneda, and S. Maekawa, "The Experimental LISP Machine", Proc. International Joint Conference on Artificial Intelligence, 1979, PP. 865-867.