Invited Paper

A 32-bit LISP Processor for the AI Workstation ELIS with a Multiple Programming Paradigm Language, TAO

YASUSHI HIBINO*, KAZUFUMI WATANABE* and IKUO TAKEUCHI**

This paper describes a 32-bit LISP processor chip developed for the AI workstation ELIS with the multiple programming paradigm language TAO. The objective of this microprocessor is to realize an S-expression machine that can match the speed of conventional machines for compiled code execution. Architectural features are a repetitive structure for VLSI implementation of the tagged architecture and a dedicated datapath for list manipulation. All the processor functions are realized on a single VLSI chip that uses a 2-micron CMOS process. ELIS supports not only LISP but also multiple programming paradigms. The ELIS interpreter has a higher performance than that of any other dedicated machine on the market.

1. Introduction

LISP has been recognized as a highly productive programming language, because of both its language features and its language processor construction. In LISP the control structure is based on functional language and both data and program are represented as S-expressions (Symbolic expressions). The language processor usually includes an interpreter. This interpreter-centered language processor gives a substantial interactive programming environment, which is suitable for prototyping, especially in AI system programming. To realize productive AI programming, we have developed a dedicated LISP microprocessor named ELIS (Electrical Communication Laboratories List processor). This machine is intended to increase LISP's already high productivity.

Other microprocessor implementations of architectures dedicated to LISP include the Explore chip and the Ivory chip [1, 2]. The origin of both processor architectures can be traced back to the CONS machine (developed at the Massachusetts Institute of Technology) [3]. Both machine architectures should be categorized as high-level-language machines with microprogrammed control, where the instruction set, defined for execution of LISP primitive functions, is interpreted by the microprogram. The interpreter for the LISP instruction set is stored in the on-chip ROM. The LISP interpreters of both processors are written in these instruction sets. Therefore, the execution performance of these interpreters is not very high. In contrast, ELIS achieves rapid interpretive execution of LISP by employing an interpreter-oriented architecture. ELIS's architecture realizes a compact workstation with an ideal interactive programming environment for AI software development. This paper describes the ELIS processor architecture as well as its microprpcessor implementation.

*Nippon Telegraph and Telephone Corporation Human Interface Laboratories.
**Nippon Telegraph and Telephone Corporation Basic Research Laboratories.

2. Design Concept and Objectives

2.1 S-expression Machine

The syntax of LISP is based on a general representation called S-expressions. These expressions can express arbitrary tree structures, and realized in the computer

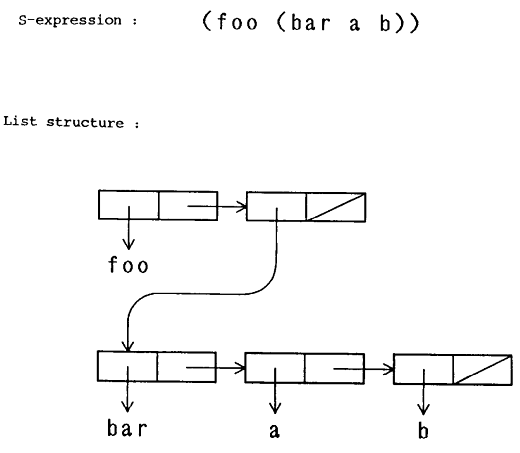


Fig. 1 Examples of S-expression and list structure data.

memory as list structures, that consist of chained memory cells with address pointers (see Fig. 1). These list structures that is, S-expressions, are the basis not only of program representation but also of data representation, and thus they may be applicable to wide information processing areas.

The outstanding feature of LISP is that program and data representations are the same; that is, S-expressions are used for both the syntax and list structures in the computer memory. This feature is identical with one of the principles of the von Neumann Computer, that instructions and data are represented in the same form by memory words.

In the same way as von Neumann constructed an execution mechanism for a computer whose control section could fetch instructions and data from memory, we constructed our execution mechanism for the S-expression machine so that the control section could interpret S-expression programs and manipulate list structure data. The following two requirements are necessary to accomplish this aim:

(1) An effective mechanism for continual access to memory cells, which are elements of the list structure.

(2) An effective mechanism for discriminating between the different types of data memory cell.

The first mechanism is needed to strengthen the memory access mechanism and to augment memory throughput. The second is accomplished by tagging each data memory cell and by providing tags with a discrimination function that can indicate data types.

Another point to be considered is that a LISP program consists of nested function calls. The execution process of such a program needs a push-down stack. Therefore, it is natural to adopt a stack architecture. The control section that executes the S-expression program must be able to handle complex state transitions and control very long sequences in order to manipulate list structures in the computer memory. It is advantageous to introduce a microprogram control into the control section to allow it to do so. Construction of an S-expression machine means that the whole LISP interpreter is written in microcode. Therefore, this means that a universal function EVAL, many special forms, and fundamental LISP functions are written in microcode, like this interpreter. An interpreter written in this manner can execute S-expression programs rapidly [4].

The objective of the microprocessor described in this paper is to realize an S-expression machine that can match the speed of conventional machines for compiled code execution [5, 6].

2.3 Multiple Paradigms in S-expression Language

AI programming has proved to be so complex and multi-faced that no single programming paradigm seems to be sufficient. Powerful AI languages have to involve many programming paradigms. The representative power of S-expressions can fuse many programming paradigms into one language syntax. For example, an S-expression is written in the form:

(S S... S)

where S's are symbols. If this expression is a LISP paradigm, the first item of the expression is interpreted as a function symbol, so the expression is evaluated as a function call. However, a different method of interpretation can be applied to the expression; it is described in detail in Section 5.

2.4 Common Hardware for Repetitive Structure

The circuit structure should be repetitive for LSI implementation of the circuit. Structural repetition affects the logic design by repetitively defining the module or the declaration copy, and helps shorten the design period. It also affects the placement design, which involves a simple pattern-copying method as well as reduction of the space and suppression of scattering of wire lengths. For this reason, irregular circuits should be avoided in the design of dedicated circuits for special-processors. One of our design principles is that circuits purpose that perform essentially the same operations, regardless of their purpose, should be realized with the same type of logic circuit. The concept of memory-purpose general registers is based on this principle.

From the beginning of the ELIS project, we have kept to this design principle. In fact, a prototype system implemented with TTL IC's in the early years had entirely the same architecture as the microprocessor described in this paper.

3. Architectural Features

3.1 Memory-General Registers

Memory-general registers, MGRs, are multi-purpose registers. Each MGR can be utilized as an operand register for ALU operation and also as memory address and memory data registers for memory access operations. There are four MGRs, each eight bytes wide. The MGR number and the byte position of an MGR is indirectly specified by an index register called a Source Destination Counter (SDC). These MGRs and SDCs provide for effective performance of the following functions:

(1) One memory cell with a 64-bit width can be fetched in a single memory operation. We call an MGR used for this function a CAR-CDR pair register.

(2) A pointer address is accessed immediately after a list cell is fetched. Two MGRs dynamically change between address register and data register. This function is effective for tracing list data, because it eliminates the address data transfer necessary in conventional architecture.

(3) A combination of MGRs and SDCs facilitates the manipulation of variable-length bytes. Bytes or half-word data in the MGRs can be extracted or in-

serted at any position specified by SDCs. Each SDC is a 5-bit-wide counter register and has an automatic count-up capability. Of the fives bits, the upper two select one of the four MCRs, and others specify the byte position in the selected MGR. SDC increments are controlled so that it operates as a modulo 8, 16 or 32 counter. By iteratively executing a microinstruction that performs an SDC increment operation and accesses the byte or half-word data in MGRs specified by the SDC, the data can be sequentially accessed, as if the MGRs were connected and used as a byte-buffer register (see Fig. 2).

(4) In the case of compiled code execution, an MGR serves as an instruction prefetch buffer for eight instructions. We used an instruction format called byte-code. The combination of half of an MGR with an SDC forms a program counter. The lower three bits of the SDC specify the instruction position, and the half MGR contains the upper bits of the program counter.

As mentioned above, the MGRs are used as multi-purpose registers related to all the memory access operations such as CAR/CDR register, character buffer, instruction buffer, and program counter. The four MGRs are constructed with the same circuit design.

3.2 Tagged Architecture and Address Space

High-level language machines mostly introduce tagged architecture in order to discriminate between data types. LISP particularly needs some form of dynamic data type discrimination, because LISP data types have little declaration. Although in recent years there have been efforts to improve compiled code execution by introducing a type of declaration into the language specification, as Common LISP, dynamic data type discrimination is still necessary in an interpreter-centered language processor intended for an S-expression machine. When introducing a tag system, the word-length problem has to be considered. Word length is the basis for the format of the data or address that represents a word. In general, word length is a multiple of 8, usually either 8, 16, 32, or 64. This format conforms to the customary standard for data transfer and storage. Almost all the peripheral LSI specifications adopt this format.

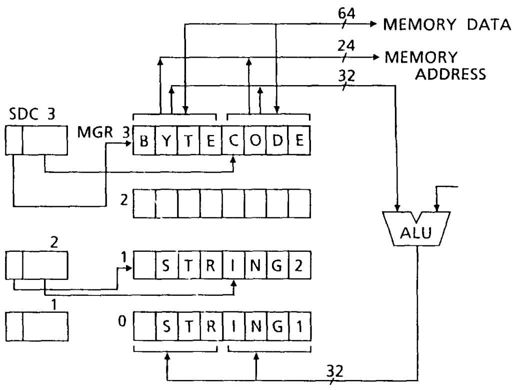


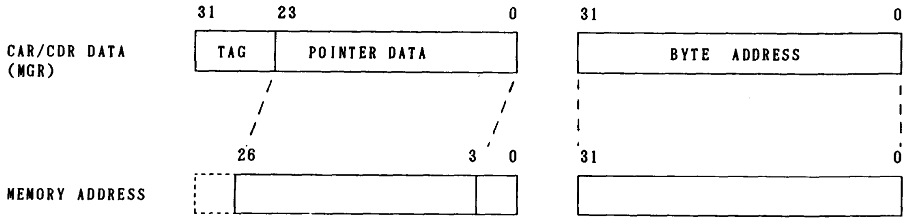
Fig. 2 Byte manipulation using MGR-SDC pair.

We decided to make the word length 32 bits wide and to include an 8-bit tag field in each word. This decision means that the maximum address space is 2**24 unit words. This may seem rather small, but a unit word is an 8-byte cell, so the address space has 2**27 bytes (128 MB). Furthermore, a byte address access mode is provided. In this mode, all 32 bits are effective byte address bits, so the maximum address space is 2**32 (4 GB) (see Fig. 3).

3.3 TAG Bit Assignment and Memory Operation Controlled by TAG

A list cell format is shown in Fig. 4. A cell consists of two consecutive memory words. Both the CAR part and the CDR part consist of an 8-bit tag field followed by a 24-bit pointer field. Of these tag bits, the lower six bits (tag5-tag0) are used for data type encoding. These bits stand for the type of data pointed to. The sixth one (tag6) is for tag extension, and the seventh one (tag7) is a marker bit for garbage collection.

The fifth tag bit (tag5) is a special one that can directly control a memory access operation. Tag5 indicates whether the content of the following 24-bit field means a pointer or not. The memory operation for pointer access is initiated by using the special memory



(2^{27}bytes)
(1) Cell Addressing Mode.

(2^{32}bytes)
(2) Byte Addressing Mode.

Fig. 3 Addressing Mode.

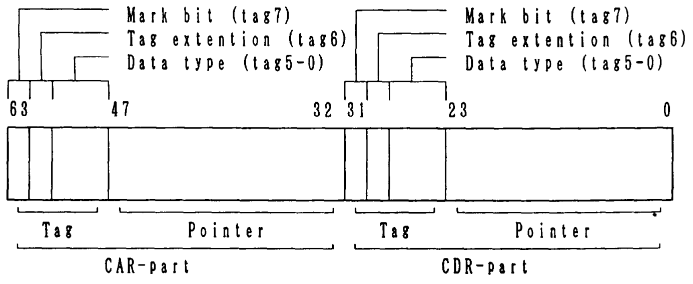


Fig. 4 List cell format.

operation, READ-BOTH-WORDS-WITH-CANCEL, if tag5 is set; otherwise, it is canceled. This tag5 bit function is very effective in tracing list structure. For microcoding, it is possible to issue a memory operation before type checking; that is, consecutive memory operations can be performed without previously checking the list tail marker by nil or atom. This saves dynamic micro-steps for list tracing.

Data type decoding is achieved by a branch microinstruction for tag bits in a single microcycle. Tag bits are also used to speed up the interpretation of a multiple paradigm language.

3.4 Hardware Stack

The LISP language is basically constructed as a function and LISP programs are defined as nested function calls. When these programs are executed, function calls are frequently issued, and an execution environment for function invocation must be maintained. For nested function calls, the execution environment is maintained and saved in a push-down stack. The performance of function invocation is an important factor in achieving rapid list processing. Therefore, we employed a hardware stack in our processor chip.

Usually, the execution environment is maintained in the form of a stack frame that stores the control information and the operand data. To allow this information to be accessed, it is convenient to provide dedicated hardware registers, such as a top frame register and a bottom frame register, which keep the top and bottom of a stack frame; a working pointer register, which gives access to the information at an arbitrary position in a stack frame; and a normal stack top register, which has push and pop operation capability.

However, this hardware stack should be implemented in such a way as to allow multi-programming and to provide flexibility in the stack frame design. For multi-programming it is insufficient to simply consider the effectiveness of the stack operation for a single process. For flexibility of the stack frame design, it is necessary to avoid falling into a situation in which the hardware stack restricts the stack frame design to a set pattern. This results in ineffective utilization of the hardware resources.

In line with the above considerations, we designed three pairs of a stack pointer register and a stack top cache register on the processor chip. These registers, with SRAM chips outside the chip, construct three sets of push-down stacks. The three stack functions are the same, and the way they are used is left up to the programmers. We assume that one will be used as a stack top register and the others as working pointer registers.

The capacity of the stack is 32 Kwords (32 bits/word), and it is divided into sixteen 2-Kword areas. Any continuous 2-Kword areas can be assigned to any of the three stacks, and the overflow and underflow of each stack are automatically checked.

This large stack plays an important role in multi-process execution, and we designed it to support up to 128 processes. A stack area may be allocated for each process that consists of several 2-Kword areas. If two processes occupy their own stack areas, process switching takes only 40 microseconds. Even if the stack area is insufficient, it takes 1.6 milliseconds to swap a single stack area. Since stack swapping does not occur very often, process switching is sufficiently rapid.

A stack operation can normally be executed in a single microcycle, except when specific stack access sequences are required. Stack pointer registers can execute both the auto-decrement-indirect mode (push operation) and the indirect-auto-increment mode (pop operation) in a machine cycle. In the former, the stack pointer is decreased in the current cycle and the new stack top data is written into the stack memory array in the next cycle. In the latter, the stack pointer is increased and the stack memory array is accessed by an increased address in the current cycle, at the end of which new stack top data is latched to the stack top cache register. If the push operation is followed by the pop operation in two consecutive machine cycles, a conflict of access to the stack memory is automatically detected and the machine cycle is stretched.

3.5 Bus Structure

The whole datapath structure as summarized above is shown in Fig. 5. It has three 32-bit buses: two for operand sources and one for destination. The arithmetic logic unit (ALU) with a one-bit shifter

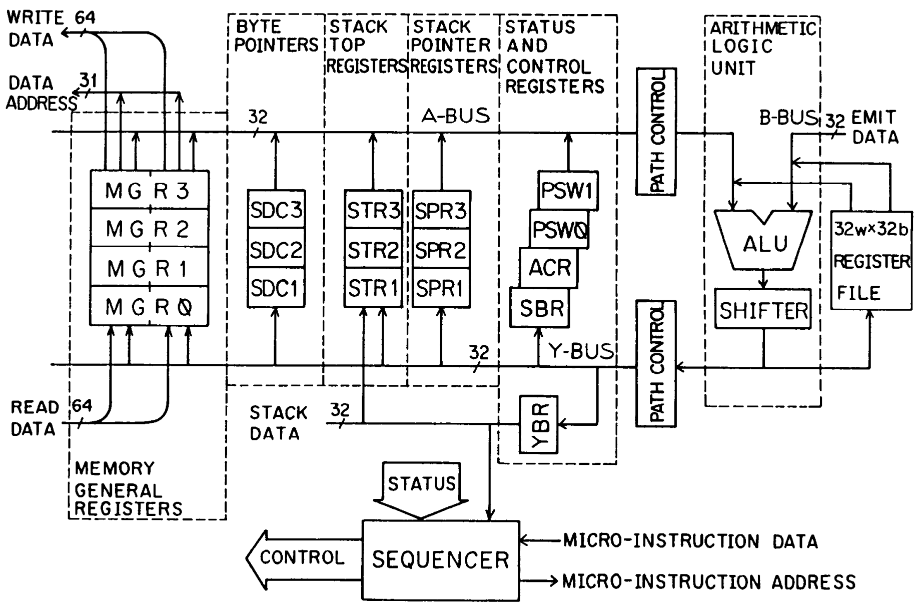


Fig. 5 Datapath structure.

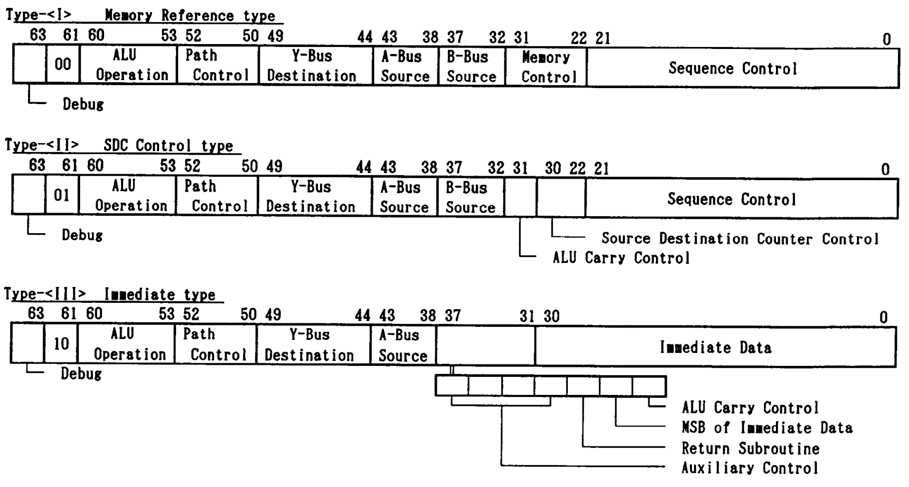


Fig. 6 Microinstruction format.

(SHIFTER) has two operating modes: one for 24-bit data and the other for 32-bit data. The path control extracts or inserts byte or half-word data. The register file is a 32 x 32-bit two-port RAM. The status and control registers contain a processor status word (PSW), a stack boundary register (SBR), an ALU result register (YBR), and an I/O register (ACR). The stack pointer registers (SPRs) are capable of automatic up/down counting. The SPRs and the stack top cache registers (STRs) are used to construct three sets of push-down stacks attached to high-speed SRAM chips outside the processor chip. The memory-general registers (MGRs) are 64 bits wide and are connected to the three types of datapaths. Each MGR is physically connected to a 64-bit memory data bus, a 32-bit address bus, a 32-bit ALU-source bus (A-BUS), and a 32-bit ALU-destination bus (Y-BUS). The byte pointer registers (SDCs) are used to specify a byte location in the MGRs. The 64-bit memory data bus, which is twice as wide as a conventional 32-bit processor, allows access to a list cell in a single memory operation and reduces the memory access frequency.

3.6 Microinstruction Format

The microinstruction formats are shown in Fig. 6. An ALU operation can be executed concurrently with a memory operation by Type-<I>, or with an SDC increment operation by type-⟨II⟩. The ALU operates in a three-address mode: two addresses for operand sources, and one for the destination. All the ELIS-CPU's registers, described in Section 3.5, can be A-Bus sources and Y-Bus destinations. The B-Bus source field is used to specify the register file, or small integer data (-16 to 15). The Type-⟨III⟩ microinstruction generates 32-bit immediate data. A stack pop or push operation is executed when the stack is specified as a source or a destination. The memory control field has two subfields: one for read/write-control, and one for specifying the address register and data register. The Source Destination Counter Control field is used to control the increment of each SDC.

Both Type-⟨I⟩ and Type-⟨II⟩ have a 14-bit wide address field in the sequence control field, which is used for generating the next microinstruction address. During branching, the next address is formed from the address field with some lower bits modified according to the status flag bits. These flag bits are ALU results, tag bits, the content of YBR, and so on. Since Type-⟨III⟩ does not have an address field, the next address is generated by adding "1" to its address.

The clock control needed for executing microinstructions sequentially is supported entirely by hardware, so microprogrammers are not concerned with timing constraints. Futhermore, the function of microinstructions is close to that of machine instructions. Because of these two features, microprogrammers can write in the same way as for assembly-language programs.

4. Chip Design

4.1 Processor Cycle Time and Memory Access Time

The relationship between processor cycle time and memory access time is an important consideration in the design of processors. In list processing, since memory is frequently accessed in following or creating pointer-chaining list data, not only the processor cycle time but also the memory access time dominates the performance. To obtain a short processor cycle time, Emitter Coupled Logic (ECL) devices are desirable. However, they have problems as regards cooling and hardware size in the typical workstation environment. Use of a TTL or CMOS device provides the normal cycle time of 150 to 250 nanoseconds, and the memory access time with MOS DRAM chips is two to three times that of the processor cycle time. For faster access, cache memory is required. If data is not found in cache, however, significant overhead delays occur. To avoid memory access limitations on performance, a special function allows the processor to execute microinstructions while accessing memory.

When this function is used for microprograms the memory access time does not significantly influence the performance. The processor cycle time is 180 nanoseconds. While the processor is waiting for completion of a memory access, three microinstructions that are unrelated to the last memory operation can be executed.

In this way, memory access can proceed simultaneously with the execution of necessary microinstructions. In the implementation of the LISP interpreter, microinstruction sequences utilize this procedure as much as possible.

4.2 Pin Multiplexing

Because of the adoption of interpreter-oriented architecture, it is difficult to integrate all the microcode of the LISP interpreter onto a single processor chip with the current technology. The LISP interpreter requires a total microcode of 64 Kwords. To store this microcode, a writable control store (WSC), constructed with a SRAM chip array, is attached outside the processor chip, as well as an SRAM chip array for the stack.

Consequently, the chip requires a large memory data transfer throughput. The throughput from the writable control store to the chip is 44 megabytes/second, between the stack memory and the chip it is 22 megabytes/second, and between the main memory and the chip it is 14 megabytes/second. The total throughput is 80 megabytes/second. This figure is equivalent to that between the CPU and the cache memory of mainframe computers with a speed of several MIPS.

To allow such a large data transfer throughput to the chip with 208 I/O pins, 112 of the pins are highly multiplexed. Of these, 64 are used for three individual data lines—one for 64 WCS-data input lines, one for 64 main-memory-data input/output lines, and the other for 32 main-memory address lines—32 are for stack-data input/output lines, and 16 are for 16-bit input/output port lines.

4.3 Design Method for Repetitive Structure

The layout design reflects the architectural design of this chip. Our goal for the layout design was to shorten the design period as much as possible and obtain the target machine cycle time by using 2-micron CMOS standard cell technology and an automatic layout system. One controversial result of applying the automatic layout system to standard cell design is that the line delay time varies widely, because cell placement cannot be easily controlled.

With our architectural principles, since almost all parts of the datapath have regular structures with repeated circuit patterns, cell placement is easily done by making copies of the original placement, which can be done by hand. These regular, repetitive structured portions occupy more than 50 percent of the total chip area, including four MGRs, three SPRs, and three STRs.

In addition, the ALU is constructed by copying the four-bit slice units eight times.

This placement method controls variances in line

Table 1 Chip characteristics.

Process	Double-metal-layer CMOs
Design Rule	2 um
Transistors	79,438
Chip Size	15.0 mm × 15.0 mm
Clock Rate	16.67 MHz
Microcycle	180 nsec (variable)
Power Dissipation	1.0 W
Power Supply	+5 V
Package	208-pin PGA

delay time so that the delay time simulation is sufficient, on the assumption that the delay time is proportional to the number of fanouts.

4.4 Chip Design Summary

Chip characteristics are summarized in Table 1. The chip, implemented with a 2-micron CMOS double-metal-layer process, operates at a 16.67-MHz clock speed and has a cycle time of 180 nanoseconds [7]. This chip contains about 80,000 transistors in a 15.0 mm × 15.0 mm die and is assembled in a 208-pin pin-grid-array package. A photo of the chip is shown in Fig. 7. The chip contains a clock generator, a microprogram sequencer, and control logic circuits for the datapath mentioned in Section 3.5.

A complete LISP processor function can be realized on a single CPU board using this chip. The actual CPU board is shown in Fig. 8. It includes a 64 K × 64-bit WCS, a 32 K × 32-bit stack memory, a 64 K × 20-bit memory for dynamic address translation (DAT), and supplemental logic on a 30 cm × 42 cm printed circuit board.

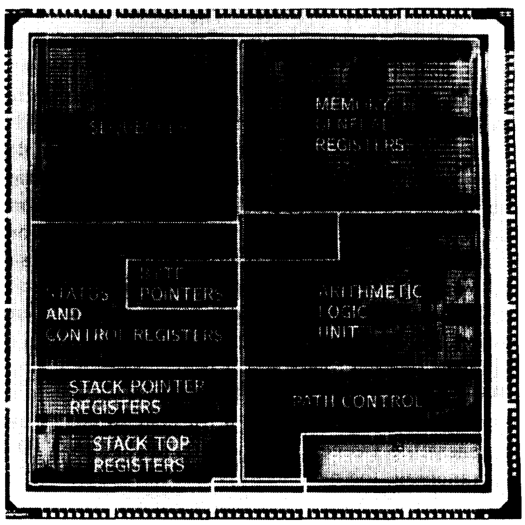


Fig. 7 Microphotograph of chip.

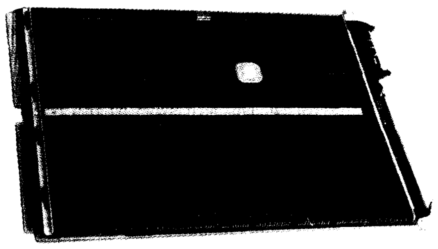


Fig. 8 ELIS CPU board.

5. Multiple Paradigm Language TAO

Powerful AI languages involve many programming paradigms, because AI programming is so complicated that no single programming paradigm is sufficient. For a productive AI programming environment, a combination of multiple programming paradigms is effective, and the difference in the execution speeds of paradigms should be small. TAO, the language implemented on ELIS, is a new multiple programming paradigm language. TAO assimilates the essence of object-oriented programming and logic programming into the LISP world. The user can write programs mixing these three paradigms at the expression level, not at the program module level [8,9]. The basic idea of fusing the object-oriented programming paradigm and the logic programming paradigm to the LISP S-expression is as follows. In general, an S-expression is written in the form

$$(S\ S \ldots S) \qquad (1)$$

where S's are symbols.

The expression is interpreted as shown in Fig. 9. If the first item of expression (1) is a function symbol, the expression is interpreted as the customary form of a function call. If the first item of expression (1) is not a function symbol but a predicate symbol of logic programming, the expression is interpreted as a logic operation in TAO. In the following example, &append is a predicate symbol and means that the two list items L1 and L2 are appended to L3.

(&append L1 L2 L3)

If the first item of expression (1) is neither a function symbol nor a predicate symbol, the expression is interpreted as a message-passing form of object-oriented programming and the first item is treated as a receiver object. In the following example, a-window is a receiver object of the subsequent message pattern move.

(a-window move x y)

As shown in the example mentioned above, an S-expression can be extended to convey more meanings than is possible in conventional LISP.

The execution speed of either object-oriented programming or logic programming is one-half that of

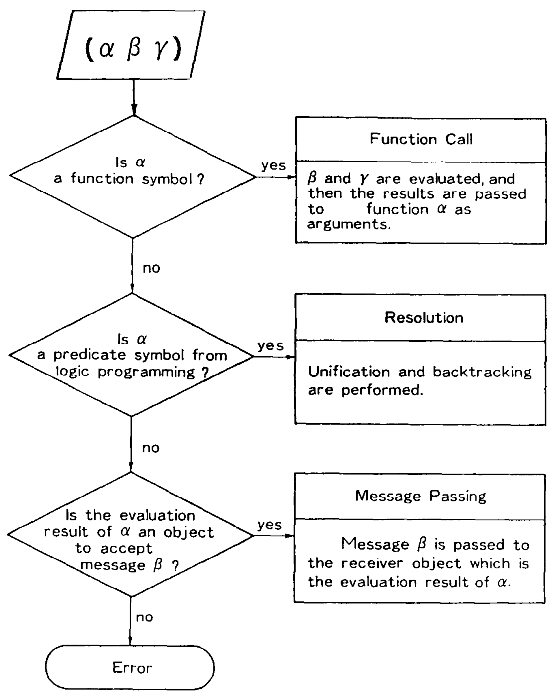

(α β γ)
Is α a function symbol ?
yes
Function Call
β and γ are evaluated, and then the results are passed to function α as arguments.
no
Is α a predicate symbol from logic programming ?
yes
Resolution
Unification and backtracking are performed.
no
Is the evaluation result of α an object to accept message β ?
yes
Message Passing
Message β is passed to the receiver object which is the evaluation result of α.
no
Error

Fig. 9 Interpretation flow of the S-expression (α β γ).

LISP. The reason for this relatively small difference is that microcoding speeds up the interpretation of message forms for object-oriented programming, and unification and backtracking for logic programming.

6. Performance Evaluation

Performance measurements based on the LISP benchmark programs are show in Table. 2. The TARAI benchmarks measure stack performance for function calls. The SREV benchmarks measure the performance of list manipulation. The TPU benchmarks (Theorem Prover by Unit binary resolution—a 400-line LISP program; see reference [10] for details) are typical examples of practical programs. The compiled programs were executed three to five times faster than the LISP source programs.

The TAO interpretation speed is extremely high in comparison with the compiled code execution on a DEC-2060 for the TPU program. In particular, for the larger programs shown in the bottom three rows of Table 2, the TAO interpreter reveals its real power. This confirms the advantages of the ELIS architecture.

7. Summary

We have achieved a high-performance LISP processor chip by choosing an appropriate architecture. This means that we have realized an S-expression machine by an interpreter-oriented architecture approach, which has a higher level of performance than any other machine achieved for compiled code execution.

This processor features a tagged architecture, a hardware stack, and multi-purpose memory registers (MGRs). These features facilitate the primitive operations essential to list manipulation, especially in the interpreter execution mode. With these features as a basis, we have designed and fabricated a 32-bit processor chip with a VLSI-oriented structure that fully utilizes 2-micron CMOS standard cell technology. A 180-nanosecond machine cycle time led to a performance of about one million LISP operations per second performance in the interpreter mode.

Using this chip, complete processor functions are implemented on a single CPU board. The AI workstation ELIS is based on this CPU board, and supports a multiple-paradigm language that fuses an object-oriented programming paradigm and a logic programming paradigm into a single S-expression syntax.

Acknowledgments

These are the results of the last decade of our research and development activities. There have been

Table 2 CPU-times in seconds for Lisp benchmarks.

Bench mark Programs	ELIS (TAO)		Commercial-I (ZetaLisp)		Commercial-II (Interlisp-D)		Dec-2060 (MacLisp)	
	I	C	I	C	I	C	I	C
Tarai-5	17.1	4.18	608	3.24	1856	26.9	126	4.39
Tarai-6	628	153	22366	119	67357	988	5241	157
Srev-5	0.028	0.010	1.07	0.025	3.98	0.084	0.236	0.008
Srev-6	0.140	0.041	4.34	0.055	16.10	0.335	0.951	0.032
TPU-3	0.878	0.041	17.8	2.77	75.7	9.05	6.10	1.22
TPU-6	4.48	2.26	98.9	11.7	421	53.8	31.5	4.89
TPU-9	0.550	0.232	12.2	1.43	53.9	4.62	3.60	0.549

I: Interpreter
C: Compiled code execution

many contributors to this project. Mr. Atsushi Ishikawa worked as a member of the ELIS chip design group. Mr. Hiroshi Okuno and Mr. Nobuyasu Osato implemented the TAO interpreter. Mr. Minoru Kamio implenented the TAO compiler. All their efforts are greatly appreciated. We would also like to express our thanks to Mr. Atsushi Kawai and Mr. Masami Mori of OKI Electric Industry Ltd. for their contribution to the design and fabrication of the ELIS chip. We also wish to thank Dr. Ryoichi Matsuda, Mr. Shinichi Yamazaki, Mr. Kazuaki Komori, Mr. Takashi Sakai and Mr. Yasuhiro Yamada for their helpful suggestions and encouragement.

References

1. Bosshart, P. W., Hewes, C. R. et al. A 553K-Transistor LISP Chip, *IEEE ISSCC '87 DIGEST* (February 1987), 202-203.
2. Baker, C., Chan, D. et al. The Symbolics Ivory Microprocessor,. A 40-Bit Tagged Architecture Lisp Microprocessor, ICCD (1987), 512-515.
3. Knight, T. CONS, MIT AI lab. *Working Paper*, **80**, 1974.
4. Okuno, H. G., Ohsato, N. and Takeuchi, I. Firmware Approach to Fast LISP Interpreter, *Proc. of the Twentieth Annual Workshop on Microprogramming* (Micro-20), ACM (December 1987).
5. Watanabe, K., Ishikawa, A., Yamada, Y. and Hibino, Y. The ELIS Interpreter-Oriented LISP-Based Workstation, *Proc. of the 2nd IEEE Conference on Computer Workstations* (March 1988), 70-79.
6. Watanabe, K., Ishikawa, A., Yamada Y. and Hibino, Y. Design and Implementation of the ELIS AI Workstation, Review of the E.C.L., **37**, 1 (1989), 71-76.
7. Watanabe, K., Ishikawa, A., Yamada, Y. and Hibino, Y. A **32b** LISP Processor, *IEEE ISSCC* '87 DIGEST (February 1987), 200-201.
8. Takeuchi, I., Okuno, H. G. and Ohasata, N. A LISP Processing Language, TAO with Multiple Programming Paradigms, New Generation Computing, **4** (1986), 401-444.
9. Sugimura, T., Sugiyama, H., Amagai, Y., Murakami, K. and Takeuchi, I. TAO/ELIS, An AI Software Development Environment Based on a Multiple Programming Paradigm Language, Review of the E.C.L., **37**, 1 (1987), 77-78.
10. Chang, L. and Lee, R. Symbolic Logic and Mechanical Theorem Proving, Academic Press (1973).

(Received August 26, 1989)