ICOT Technical Report: TR-045

TR-045

The Design and Implementation of a Personal Sequential Inference Machine: PSI

by

Minoru Yokota, Akira Yamamoto, Kazuo Taki,
Hiroshi Nishikawa, and Shunichi Uchida

February 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

The Design and Implementation of a Personal Sequential Inference Machine: PSI

Minoru Yokota, Akira Yamamoto, Kazuo Taki,
Hiroshi Nishikawa, and Shunichi Uchida

ABSTRACT

A Personal Sequential Inference Machine, called PSI, is a personal computer designed as a software development tool for the Fifth Generation Computer Systems(FGCS) project. PSI has a logic based, high-level machine instruction set, called Kernel Language Version 0 (KL0). The machine architecture of PSI is specialized for direct execution of KL0. "Unification" and "backtracking" are the principal operations in Logic Programming, and they are efficiently performed by PSI hardware/firmware. Its estimated execution speed is 20K to 30K LIPS. The machine is also equipped with a large main memory with a maximum of 40 bits x 16M words. This paper presents the key points of its design and the features of its machine architecture.

1. Introduction

The Fifth Generation Computer Systems (FGCS) project[1] has actually started to realize a Knowledge Information Processing System, KIPS. The initial three years of this project is the fundamental research and development stage. During this initial stage, many experimental systems and tools will be developed for the next stage. PSI has been designed as a powerful software development tool to achieve this aim.

The major motivation for the development of PSI is to establish a good programming environment for logic programming. This means that PSI should achieve high execution speed for logic programs and should be equipped with a good man-machine interface. This is why a totally new machine was designed, in spite of the short research and development period of about one and a half years.

To fully specialize PSI for logic program execution, it is not adequate to utilize ready-made micro-processors or ready-made microprogram sequencers. The CPU of PSI is constructed from about 2000 commercial TTL SSIs and MSIs. Since design efforts were concentrated on the CPU and the memory system, the input/output interface is designed as simply as possible, and the standard I/O bus interface, IEEE-796 BUS, is adopted. However, this interface has enough versatility and many commercial I/O devices can be connected to the PSI.

Fig. 1 shows the PSI system configuration. PSI is designed as a personal tool, and is connected to the local area network as one of the work stations.

The intention of this paper is to discuss what functions PSI offers and how they are implemented. The following two sections describe the required architectural features of PSI, Section 4, how PSI machine instructions are designed, Section 5, their implementation, Section 6, internal data representation, and finally, Section 7, the addressing mechanism.

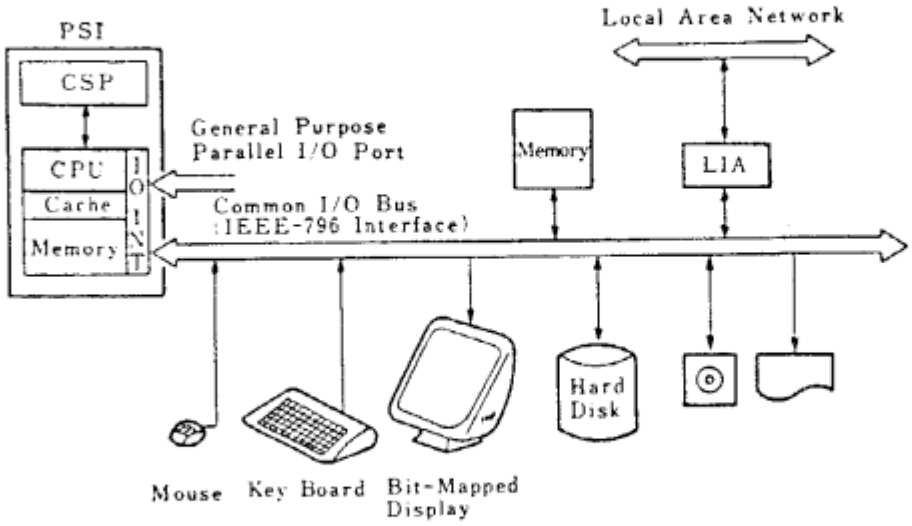


Fig. 1 PSI system configuration.

2. What is a Machine Architecture for Inference?

Since a machine architecture should reflect the computation model on which the target languages or application systems are based, the definition of "inference" must be made clear at first. However, answering this question is one of the main objectives of the FGCS project. At the beginning, logic programming language like Prolog was chosen for the kernel language of this project is considered to which involve a primitive inference mechanism. Therefore, "inference" here means such mechanism that is used to execute a program written in logic programming. Prolog was the starting point of our inference machine.

Two essential operations are required to execute Prolog programs; one is "unification", and the other is "backtracking". These operations bring in the main advantages to logic programming.

2.1 Unification

Unification consists of the following primitive operations.

o To search for the called clause.
o To fetch the arguments of both the caller and callee predicates.
o To examine the equality of the arguments

When a query is issued as shown in Fig. 2, a Prolog system searches for the corresponding clause which has the same predicate name on its head (left part of ":-") as the caller goal predicate. Each variable included in the called clause is independent of those in other clauses, even if they have the same variable name. Furthermore, the variable can have different values at the same time by virtue of its recursive call. Because of these characteristics, they are similar to the local variables in conventional subroutines. Therefore, a dynamic memory allocation mechanism is required in order to create cells in which the value of the variable is stored. This cell is called a "variable cell", and it may be implemented generally by using a stack.

To quickly fetch the value of an argument, fast memory access is required, and a cache memory is generally useful to achieve this. In addition to it, a fast data type checking mechanism is necessary to quickly examine the equality of the contents of both arguments. To satisfy this need, a tagged data form is very effective.

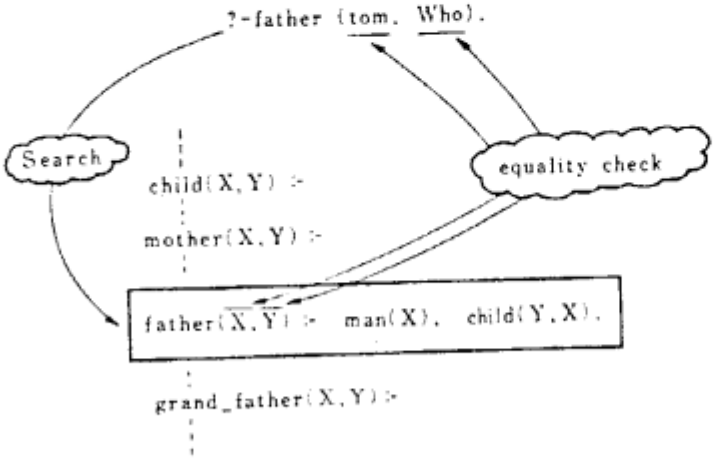


Fig. 2 Predicate call.

2.2 Backtracking

In the Prolog system, program execution is carried out determining whether the predicates are true or false. In other words, the Prolog system sets some value to an argument to make the predicate true. If there are several clauses corresponding to the goal predicate, several possibilities exist to make the predicate call true. Since the sequential machine selects only one of the possibilities at one time, the machine has to try another alternative when the previous selection fails. This retry operation is called "backtracking", and includes undo operations to reset variables to their previous status. To realize backtracking, the machine must obviously keep track of each choice point and the binding environments of the variables. Roughly speaking, the machine should hold the entire execution history. Therefore, this feature requires compiicated execution control and a lot of memory.

A large memory space is necessary for backtracking, and stack mechanism is common for its control. Although it is effective to eliminate meaningless choice points, the establishment of a useful algorithm to intelligently perform backtracking is now being researched.

The following is a summary of features of the inference machine now being considered.

- o The inference mechanism is based on Prolog.
- o Dynamic memory allocation is required for fast creation and deletion of variable cells.
- o Fast memory access is required for fast argument value fetching.
- o The tagged data form is required for fast argument type checking.
- o A large memory space is required to maintain a large execution environment.

3. PSI Design Objectives

In addition to satisfying the minimum requirements described above, there are several other requirements from the user's point of view. PSI has two characteristics, one as a tool and the other as an experimental machine. Their requirements are described below.

3.1 Requirements as a Tool

The main role of PSI is to be a computing tool in the intermediate stage of the FGCS project. To satisfy this aim, PSI hardware and its operating system should be designed as a practical

system and thus, the hardware system should be implemented as soon as possible. The key points in designing PSI are summarized as follows.

- o Practicality
- o Ease of use
- o Personal computing

(1) Practicality

Since PSI is a real tool, it needs to have high cost performance. It is desirable to discuss about the performance on the basis DEC-10 Prolog[2], because it is the most successful and practical logic programming language and its compiler can generate very fast object codes. The execution speed of a compiled Prolog program is estimated at about 30K LIPS (Logical Inference Per Second) on DEC-2060.

On the other hand, the DEC-10 Prolog system provides its user with only limited memory space (256K words), which is usually too small for actual Prolog applications. The lower execution speed might not be fatal defect because it would be compensated by longer processing times. However, there is no way to continue program execution if the system has exhausted its memory space.

Since the limitation of memory space is crucial, the design of PSI is intended to provide a larger memory space as well as fast execution speed. Our experience with the DEC-10 Prolog system has shown that at lest 10 times larger memory space available at present is required. As a result of these considerations, it was decided that PSI should have a large main memory with maximum of 16M words and an execution speed nearly equal to DEC-10 Prolog compiled codes.

In addition to the speed and memory space, reliability is also an important aspect for a tool in that the system must operate continually for a long time without failure. Consequently, PSI is equipped with enough error checking and correction mechanisms.

(2) Ease of Use

As a tool, a good man machine interface is the most important feature. To achieve this aim, both hardware and software facilities should be provided. Accordingly, PSI is equipped with special hardware such as a bit-mapped display and a pointing device (a mouse). In addition, I/O devices for Japanese characters will also be installed in order to communicate with PSI in our tongue.

Software facilities include a PSI operating system with sophisticated programming systems. A multi-window system is one of the important facilities.

(3) Personal Computing

Since computer hardware was expensive, a large and centralized computer system has been common in the past. However, personal computing becomes desirable by decreasing hardware costs and increasing computer utilization. For smooth man-machine interaction, localized data processing will be essential rather than using conventional centralized time sharing systems.

To provide a user with PSI's powerful computing power, PSI was designed as a stand-alone, personal computer connected to the local area network. Through the network, several PSIs can communicate with each other and establish a distributed system. Since PSI can also communicate via the network with different machines, such as a Relational Data Base Machine, it can be regarded as an open-ended system.

3.2 Requirements as an Experimental Machine

Although PSI is intended to be a practical machine, designing its architecture involves a lot of research because no one has built such an inference machine before and there are few evaluation data on Prolog program's behavior.

As the first specialized machine for logic programs, it was considered that the following needs should be satisfied.

- o Support for fast unification
- o Support for multi-processing
- o Support for low level system description
- o Support for evaluation and measurement of logic programming characteristics
- o provision of flexibility in language support

(1) Fast Unification Support

As described in Section 2, unification is one of the most essential operations in an inference machine. If a machine architecture is dedicated to inference, it should support unification at the machine level through its hardware and firmware.

(2) Multi-Processing in Logic Programming

A Prolog program is composed of single layered relations. They can be referred from any portion of the program. That is, a program is seemed a large, single module. This feature of the Prolog program is not suitable for the programmer, and the lack of program modularization makes it difficult to build large, complex programs.

However the introduction of modularity and concurrency into Prolog is under research, our original motivation was to establish conventional multi-processing environment in logic programming. It was imagined that, for example, a user of PSI will use an editor, compiler, and debugger at the same time through a multi-window system.

All these software products are written in a Prolog-like logic programming language. This situation requires a few relatively large processes.

On the other hand, the description of parallel processing in logic programming is becoming attractive research theme and is one of the important research targets in the FGCS project. In this point of view, a lot of small sized processes such as in Concurrent Prolog[4] should be efficiently supported. In this situation, the size of process is very small but the number of processes is over a thousand. To support this type of multi-processing, the machine architecture may become different from PSI. Although PSI can not support a large number of processes, it can be used to experiment such a multi-process environment.

(3) Low Level System Description

Since PSI is designed to be a stand-alone, personal machine, it must be furnished with its own operating system. In general, the operating system does not fit high-level languages because it deals with physical resources though the latter deals with logical ones. Logic programming language is also regarded as being undesirable for writing operating systems. One of the advantages of using a logic programming language is its non-determinate operation. It makes programmers to be able to quit specifying the entire perfect procedures. In spite of this, the machine finds the answer by examining the possible alternatives using fragmentary rules. On the other hand, the procedures in the operating system are usually deterministic, and efficiency is the most important factor in its execution.

Further, variables in a logic programming language are logical variables and not objects of an assignment operation. That is, rewriting the value of a variable is basically not allowed in logic programming. However, the checking and rewriting entries of common tables are frequent operations in an operating system.

Even though there are such difficulties, the uniformity of the system is very important especially in a personal machine[5]. If a different language is introduced for system description, the user must be familiar with both the low-level system language and high-level user languages. This situation makes it difficult for the user to understand the operating system and to maintain its system software. As for the machine architecture, this situation leads to incomplete optimization of both the system and user language, because each language requires different characteristics. Therefore, it is desirable to unify both the operating system and user applications in the same language concept.

Because of this consideration, it was decided to make the operating system in logic programming. To achieve this aim, a low-level system description capability was introduced into Prolog.

Furthermore the concept of object oriented programming was also introduced for description of the PSI operating system.

(4) Evaluation and Measurement

To evaluate PSI machine architecture design, the measurements of the detailed characteristics of machine's behavior is necessary. And its evaluation results will become quite important to improve the PSI architecture and to develop the next advanced inference machine. It will also be quite useful to measure the profile of the logic programs on PSI. PSI has the evaluation support mechanism both at the hardware and firmware levels, and detailed information can be gained without measuring overhead.

(5) Flexibility

Since the authors do not have enough experience in logic programming, the PSI target language may be revised in the future. Further, to design better logic programming languages is one of major subjects of our project. Therefore, the machine architecture should support the execution of several languages, one for a tool and the other for experiments. The microprogrammed implementation of a language interpreter makes this possible.

4. Machine Language

One of the key factors in determining machine architecture is the level of its machine language. PSI is designed for effectively executing logic programming language, and the FGCS project chose Prolog as a starting point. After that, the several extensions of Prolog were required as described in section 3, such as a low-level system description capability. Therefore, FGCS Kernel Language Version 0, KL0[6], was designed as a uniform programming language to satisfy the requirements from low to higher application levels.

4.1 KL0

KL0 has syntax similar to DEC-10 Prolog except for internal database operations such as assert and retract operations. KL0 is used to describe the PSI operating system in order to achieve a uniform, self-contained personal system. Therefore, KL0 includes many built-in predicates which can directly manipulate the hardware resources, refer to arbitrary memory locations, and perform system control functions. The features of KL0 are summarized as follows.

- o a subset of DEC-10 Prolog
- o a reinforced execution control ability
- o an extended ability for hardware resource handling
- o an extended ability for interrupt handling and process control

4.2 Machine Instruction Level

In designing PSI, two possible implementations were considered. One is to design its machine instructions as low as possible and to generate optimized object codes by compiler. The other is to design them as high as possible and to execute them directly by PSI hardware and firmware. If the target language has static features and if the compiler can generate as effective object codes as in FORTRAN, the former approach is better. However, Prolog has many dynamic features, including unification and backtracking. Since the complex operation can be optimized by hardware/firmware, it is appropriate that they be condensed into one machine instruction as much as possible.

Furthermore, the main part of unification and backtracking is composed of simple memory access operations. It is undesirable to divide their operations into small codes because the machine must fetch those codes from the memory simultaneously with operand data fetching. To effectively fetch both small pieces of code and operand data at the same time, a sophisticated instruction prefetch unit is necessary.

Because of the above considerations, it was decided that PSI machine language level would be almost equal to KL0. In other words, PSI was designed to directly execute KL0.

Fig. 3 shows the language hierarchy of PSI. For end-users, a high-level, end-user language is provided and its compiler written in KL0. At the system designer level, a macro assembler language of KL0, called ESP[7], is used and its compiler is also written in KL0. All these programming languages are compiled into internal object form and then executed directly by PSI firmware system.

PSI firmware system consists of two parts. One is a KL0 language interpreter, and the other is system control routines for operating system support. Process switching, interrupt handling, and exception handling are involved in the latter group.

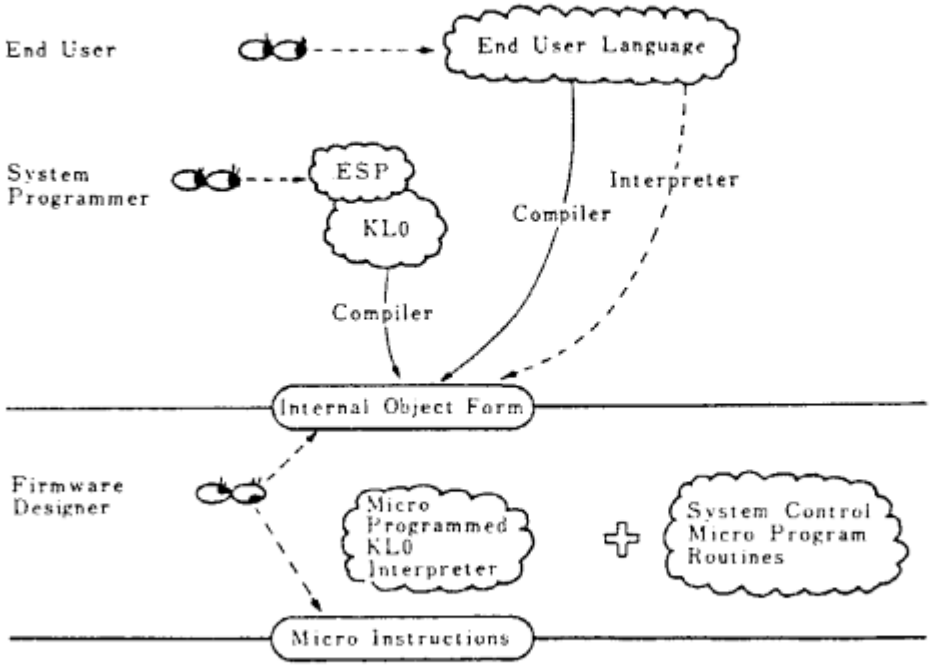


Fig. 3 PSI language hierarchy.

5. Interpretation of KL0

Since KL0 is similar to DEC-10 Prolog, the interpretation mechanism is also similar[8]. Essential operations required for executing Prolog are clause invocation, unification, and backtracking. These operations could be implemented in various mechanism. However, if their effective execution is emphasized, a stack mechanism may be most practical in the current state of the art. Since PSI is intended to be a practical machine, it was decided that it would be designed to be as simple and effective as possible.

5.1 Stack Oriented Interpretation

As a result, the interpretation of KL0 is performed by using 4 independent stack areas; control, local, global, and trail stacks.

The control stack is used for keeping track of execution environment, and various control information is saved into this stack in order to control the execution sequence of clauses and to reset the environment during backtracking.

The local/global stacks are used for locating variable cells dynamically. The scope of each variable is localized within the clause, and its variable cell is allocated in the local/global stack areas when that clause is called from some other clause.

There are two types of variables, local and global. A global variable is one which is an argument of structured data. Since structured data is shared among clauses, this type of variable can not be deleted until on execution fails or garbage collection. This is the reason for isolating local and global stacks.

The trail stack is used for saving the cell addresses each of which the value is bound to. When backtracking occurs, the values of the variables are reset to unbound status using these saved cell addresses.

Fig. 4 shows the KL0 execution environment. The object codes are loaded into a heap area. The PSI microprogrammed interpreter fetches both the caller and callee codes and executes unification between them by creating environments in 4 stacks.

In this interpretation, several stack frame base addresses should be kept in working registers. They are local stack frame base, global stack frame base, object code address, and so on.

The detailed usage of stacks is similar to DEC-10 Prolog. However, since KL0 has more complex execution control features, such as remote cut, stack control is more complicated than in DEC-10 Prolog. For this reason, control stack is separated from local stack in PSI.

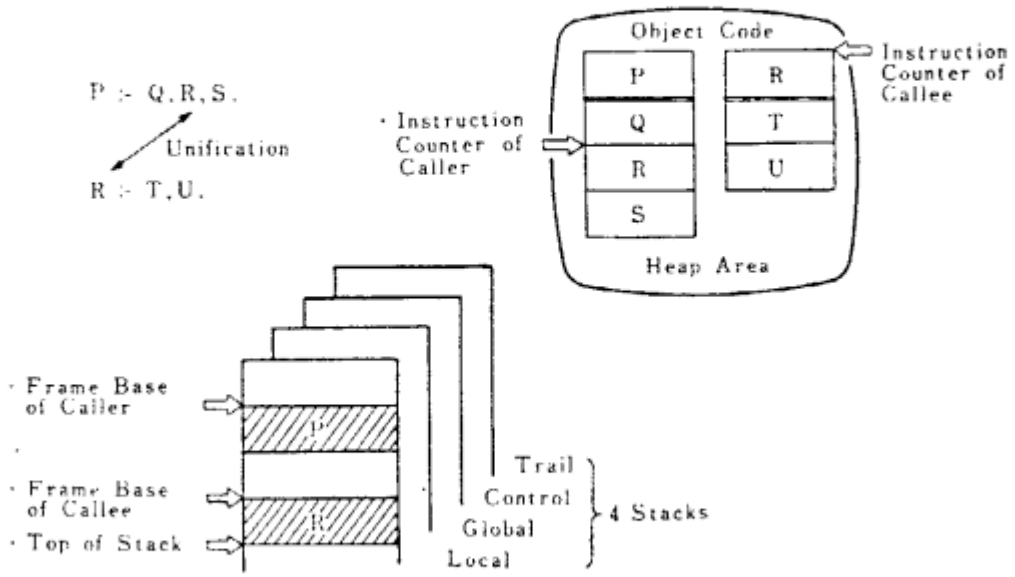


Fig. 4 Execution environment of KL0.

5.2 Structure Sharing

Another key point affecting the machine architecture is the manipulation of structured data. There are two methods of representing structured data, one is structure sharing[9] and the other is structure copying. Fig. 5 shows typical data structures for each data representation. The characteristics of those two methods are opposite as shown in Table 1. Superiority of one method to the other depends on the characteristics of the program. That is, sharing is suitable for programs that frequently use large structured data and do not frequently access their elements. However, if programs use relatively small structures, such as lists, and frequently access their elements, the copying method is suitable. Although the copying method is simple, the sharing method has been chosen for PSI because of the following reasons.

- o Since structured data can be easily manipulated in Prolog, the usage of large structures will be expected high.
- o The average copying overhead might be larger than access overhead of shared structure elements.
- o For large structured data, the sharing method is very efficient in terms of memory utilization and execution speed.

Using structure sharing, a structured data is represented by actual values and its structure. Therefore the variable cell should have two pointers, one points to the value area in the global stack and the other to the structure representation. In DEC-10 Prolog, each variable cell has these two pointers. In this implementation, the size of each pointer is limited to only a half word, 16 bits in PSI. As it was intended to make full use of the large main memory with maximum of 16M words, the packing of these two pointers into one variable cell was abandoned. Therefore, this pair of pointers is allocated in the global stack as shown in Fig. 6 whenever the variable binds to the structured data.

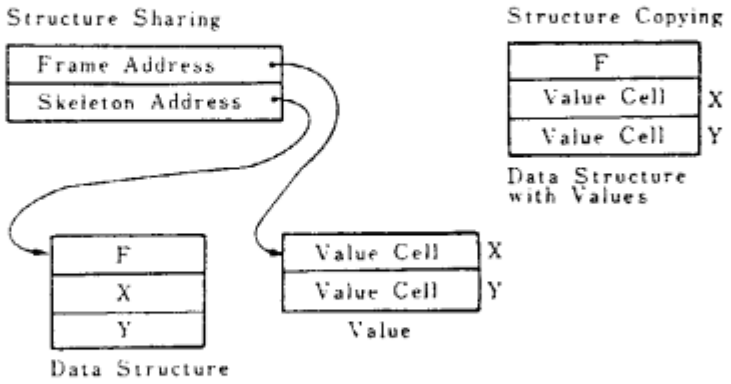


Fig. 5 Implementing structured data F(X,Y).

Table 1 Structure sharing v.s. structure copying.

	Sharing	Copying
Unify to unbound variable	No copy operation* (Share the same structure)	Copy structure
Unify to the same structure	Access element through skeleton (indirect access)	Direct access to the element*

* This aspect is regarded as an advantage.

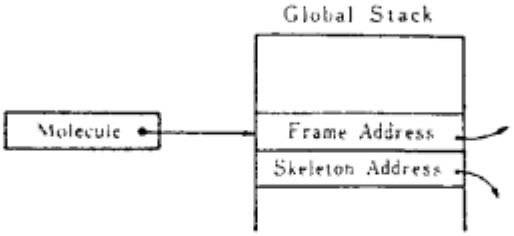


Fig. 6 Structure sharing in PSI.

6. Internal Data Representation

This section summarizes the internal representations of data and clauses in PSI.

6.1 Data Representation

As described in Section 2, Tag bits is adequate for interpreting logic programming language. Although the word size is usually 32 or 36 bits in conventional computers, it is short to include tag bits. Including the tag bits for garbage collection, 8 bits is required for a tag part. If the machine is designed as a 32 bit machine, this only leaves 24 bits for the data part, which is insufficient for representing numbers and addresses. Therefore, PSI adopts 40 bits for word size, 8 bits for the tag and 32 bits as the data part. Although the word size is incompatible with conventional machines, this causes no problems because only the 32 bit data part is usually transferred to/from the outside of PSI.

Fig. 7 shows PSI word format. The tag part consists of two parts, GC tag and Data tag. GC tag is used during garbage collection, and data tag is used to identify the contents of the data part. For fast data type checking, PSI is equipped with special hardware to dispatch micro program control according to the data tag.

PSI could have a maximum of 64 internal data types which are directly manipulated by hardware/firmware.

One group represents data types defined in KL0. They are listed in Fig. 8. For practical requirements, PSI can manipulate floating point numbers and several types of string data. Especially, double-byte string data is prepared for representing Kanji characters.

A vector data type is used to represent almost all structured data including list structures, and its length is shown in a extra descriptor. Since small structured data may be frequently used in programs, it can be represented without the descriptor. In that case, 3 bits of the data tag are used to indicate the vector length as shown in Fig. 9.

Another group is used for representing the object form of clauses as described next. The other group is prepared to control interpretation of KL0. Some examples of them are an invisible pointer or a frame base pointer.

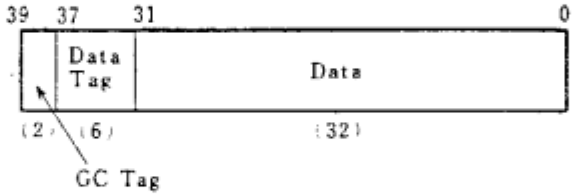


Fig. 7 Word format.

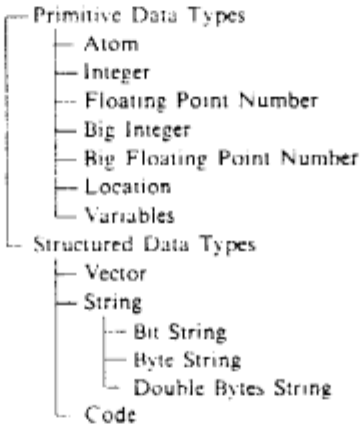


Fig. 8 Data types in KL0.

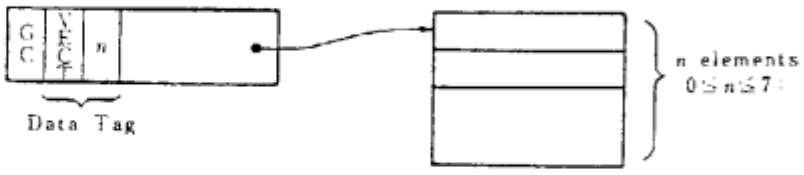


Fig. 9 Representation of a small vector.

6.2 Clause Representation

The general form of a clause in KL0 is the same as the Horn Clauses in First Order Logic as shown below.

p(X1,X2,...) :- q(Y1,Y2,...), r(Z1,Z2,...),...

There is at most one predicate appearing at the left part of ":-", which is called a head predicate. The right part of ":-" is called body goals which consists of an arbitrary number of predicates, including zero.

These KL0 clauses are translated into internal object forms by the compiler written in KL0. The basic representation of an object form is shown in Fig. 10. Each clause is packed into a block of contiguous memory cells. A "procedure" is defined as a collection of clauses whose head predicate names are the same, and the object form is generated in the unit of a procedure. Therefore the PSI object code consists of a procedure header and the number of blocks of clauses. The procedure header represents the number of its arguments and the size of the object code.

Each clause is represented by a clause header, head arguments, and combined body goals. The clause header represents the clause type and the number of the variables included in the clause. Each argument in the head predicate is compiled into corresponding one word internal form. In the case of such constant argument as an atom or an integer, it is represented in the immediate value.

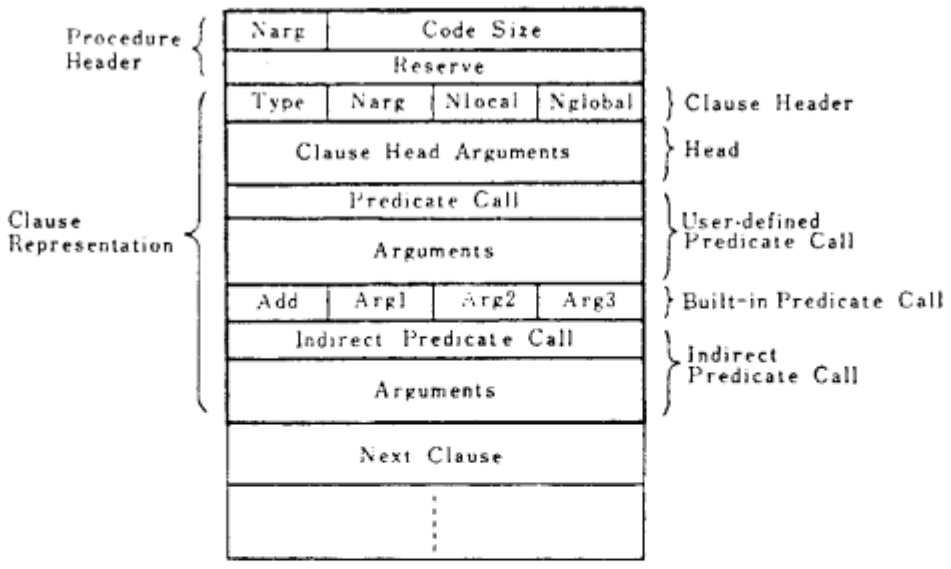


Fig. 10 Representation of a procedure.

The representations of body goals are placed following head arguments. They are used to further predicate calls. There are three types of predicate calls.

(a) User-Defined Predicate Call

This predicate call invokes a user-defined clause. A goal predicate name is compiled into the pointer to the code corresponding the called procedure and goal arguments are placed continuously after this pointer.

(b) Built-in Predicate Call

This predicate call invokes a built-in predicate. The function of built-in predicates is to efficiently perform primitive and frequently used operations such as arithmetic and logic operations, string data manipulation, system control operations, and so on. Most built-in predicates are packed into one word to make the representation compact. It consists of an 8 bit operation code and maximum of three 8 bit operands.

(c) Indirect Predicate Call

This predicate call invokes some user-defined clause through an indirect pointer. Since this indirect pointer can be modified by the program, it is possible to dynamically change the procedure to be called.

Each body goal of a clause can be regarded as the condition which makes the head predicate true. They are connected in tree different types; AND, OR, and CASE connections.

(a) AND Connection

The AND connection indicates that the head predicate is true if and only if all body goals are true. Each goal is continuously placed as shown in Fig. 11-(a). The AND connection means that each goal is executed sequentially and if a goal is failed, backtracking occurs.

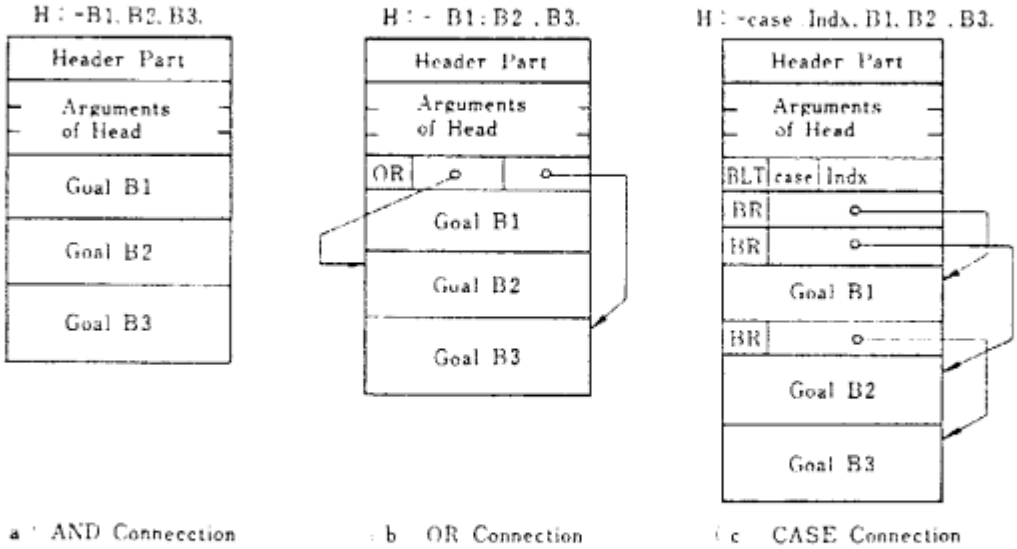
(b) OR Connection

The OR connection indicates that the head predicate is true if and only if one of the body goals is true. This connection is represented by an OR instruction as shown in Fig. 11-(b). At the first execution, the first goal is tried. If it fails, the second goal is tried next. Each branch of the OR connection may be composed of several goals. Therefore this branch is the same as an ordinary alternative clause except that it is included within a clause and sharing the same environment with it.

(c) CASE Connection

The CASE connection can be regarded as a case branch operation. Fig. 11-(c) shows the internal form of a CASE connection. It is different from the OR connection, in that if one of the goals consisting of a case connection is selected, the remaining goals are never tried.

As described above, each clause in KL0 is directly translated into a sequence of words which include all the information of the original clause. There are no explicit machine instructions to perform unification and backtracking. All of them are implicitly executed by the PSI microprogrammed interpreter.



(a) AND Connection (b) OR Connection (c) CASE Connection

Fig. 11 Body goal connection.

7. Addressing Mechanism

For the machine architecture specialized in logic programming language, it is desirable to logically provide an infinite memory space, and from this point of view, a virtual memory system is attractive. To introduce it into the inference machine, however, it is necessary to resolve the problems related to garbage collection within the virtual memory space[10]. Since it is impossible to collect garbage in the infinite memory space, some new algorithm might be required to localize the operation of a garbage collector. In addition, the substantial memory capacity required for practical applications is not well known. In consideration of present research status, it was decided that PSI would be equipped with a large main memory instead of a virtual memory system. A 16M word main memory was chosen, being based on experience with DEC-10 Prolog.

7.1 Area Based Logical Addressing

Although PSI incorporates a real main memory, it has a 32 bit logical address space. This logical addressing is introduced for the following reasons.

o Since PSI microprogrammed interpreter uses the four stacks described in section 5., it is desirable to be able to expand each stack area independently. If the same space is allocated to each of these stacks, a collision between stack areas will occur. At that time, one of them must be moved to another location. This causes serious overhead.

o Since PSI supports multi-processing, areas as for each process should be separated from those of other processes.

PSI logical address space is divided into 256 independent areas. The concept of "area" is the same as in segmentation. The size of each area can be logically expanded to a maximum of 16M words. Obviously, the total area size is physically limited to a maximum of 16M words.

Area usage depends on the memory management system. However, it may become a heap area or one of four stacks.

The area is managed by the unit of a page, and page size is 1024 words. Therefore, each area can have a maximum of 16K pages. If the page size was smaller than this, the memory space could be more effectively used. However, the page allocation would occur more frequently, and it would increase memory management overhead.

As a result, the memory cell in PSI is accessed by an area number, a page number, and inner page offset as shown in Fig. 12.

As the interpretation of KL0 requires four stacks, each process consumes at least four areas for its execution environment. Since PSI has 256 areas and processes share heap areas, a maximum of 63 processes can exist in PSI.

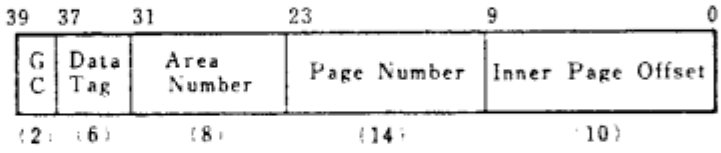


Fig. 12 Area based logical address format.

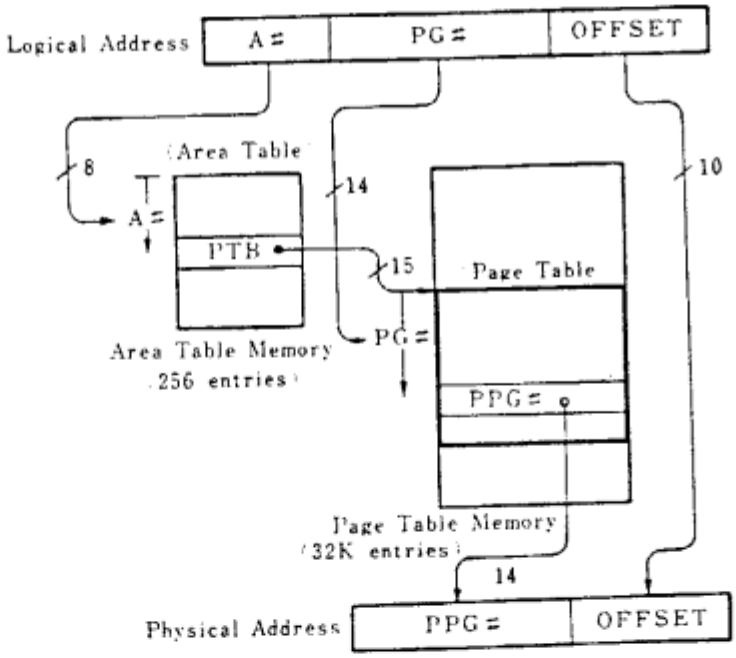


Fig. 13 Address translation mechanism.

7.2 Address Translation

The address translation mechanism is shown in Fig. 13. Translation from a logical to a physical address is performed with two tables; an area table and a page table. Each entry in the area table specifies the base address of a corresponding page table allocated within the page table memory and each entry in the page table represents a physical page address corresponding to the logical page. As a first step in address translation, the area table is accessed using the area number, and a page table base address is obtained. Then the page table is accessed using the sum of the obtained page table base and the page number. Finally, by concatenating the output of the page table and the page offset, a 24 bit physical address is obtained.

Since PSI did not use a virtual memory system and the capacity of its main memory is physically limited to a maximum of 16K pages, the sum of whole page table entries is also limited to 16K entries. For this reason, PSI holds whole page tables in a fast memory, called a page table memory in addition to an area table memory for the area table. Using their hardware, the address translation described above is performed in only one machine cycle.

The total amount of page table entries is limited, however, the size of each page table is dynamically changed. That is, it is impossible to predict how many pages are required for each area. This means that the system can not determinately allocate each page table in the page table memory. To resolve this problem, the PSI memory manager accomplishes housekeeping of the page table memory. Initially, each page table is allocated scatteringly in the page table memory. When a new physical page is allocated to an area, its page table will be extended by one entry. At this time, if the page table has been adjacent to the next page table and there is no room for adding the new table entry, the memory manager performs rearrangement of page tables. To reduce the occurrence of this page table collision, an arrangement algorithm must be considered and the size of the page table memory should be as large as possible. PSI is equipped with a 32K entry page table memory which is two times larger than the required capacity.

To manage a lot of page tables, it is common to use a Translation Lookaside Buffer (TLB) in conventional machines. Although above page table memory management is a little complex, PSI did not adopt TLB for the following reasons.

o Since TLB is regarded as a sort of cache memory, the translation table in the main memory must be accessed when the required logical-physical address pair does not exist in TLB.

o Since the garbage collector must search all memory space, the locality of memory access is expected to be not so high. Therefore, TLB might not work well during garbage collection.

o Implementation of TLB is complex in itself, and requires a complex interface mechanism between itself and the main memory.

8. Conclusion

Compared with conventional machine architectures, the main features of PSI is summed up in its machine instruction set based on a Prolog-like logic programming language, KL0, which is executed directly by a microprogrammed interpreter. Many architectural features are arranged around to effectively support direct execution of KL0. This paper focuses mainly on this aspect.

Another feature is that the PSI operating system is to be entirely written in KL0, and PSI itself is equipped with support facilities for it at the machine architecture level.

The detail hardware design is almost finished and design of its microprogrammed interpreter has already begun. To evaluate of PSI machine architecture, several experiments and measurements are planned. Since many software products will be made on PSI, it will be possible to gather program profiles of the logic programming. This evaluation data will be useful for designing the next advanced models.

ACKNOWLEDGMENTS

The authors express their grateful thanks to Dr. Takashi Chikayama for his valuable advice, to Mr. Kazuhiro Fuchi, Director of the ICOT Research Center, and Dr. Toshio Yokoi, Chief of the Third Research Laboratory for their continuous encouragement, and to other members of ICOT for their useful comments and discussions.

REFERENCE

[1] "Outline of Research and Developments for Fifth Generation Computer Systems"
ICOT Research Center, April (1983)

[2] Pereira, L.M., F.C.N. Pereira, and D.H.D. Warren
"User's Guide to DECsystem-10 PROLOG"
Department of Artificial Intelligence, Univ. of Edinburgh (1978)

[3] Murakami, K., T. Kakuta, T. Miyazaki, N. Shibayama and H. Yokota
"A Relational Database Machine: First Step to Knowledge Base Machine"
Proc. of 10th International Symposium on Computer Architecture (1983)

[4] Shapiro, E.Y.
"A Subset of Concurrent Prolog and Its Interpreter"
ICOT Technical Report TR-003 (1983)

[5] Ingalls D.H.H.
"Design Principles Behind Smalltalk"
Byte Vol. 6-8, August (1981)

[6] Chikayama, T., M. Yokota, and T. Hattori
"Fifth Generation Kernel Language"
Proc. of the Logic Programming Conference '83 (1983)

[7] Chikayama, T.
"ESP - Extended Self-contained Prolog - as a Preliminary Kernel Language of Fifth Generation Computers"
New Generation Computing, Vol. 1 No. 1 (1983)

[8] Warren, D.H.D.
"Implementing PROLOG - compiling predicate logic program"
Vol.1-2, D.A.I Research Report No.39-40,
Department of Artificial Intelligence, Univ. of Edinburgh (1977)

[9] Boyer, R.S and J.S.Moore.
"The Sharing of Structure in Theorem Proving Programs"
Machine Intelligence Vol.1-7, Edinburgh Up (1972)

[10] Cohen, J.
"Garbage Collection of Linked List Data Structures"
Computing Surveys, 13-3 (1981)

[11] Uchida, S., M. Yokota, A. Yamamoto, K. Taki and H. Nishikawa
"Outline of the Personal Sequential Inference Machine: PSI"
New Generation Computing, Vol.1 No.1 (1983)

[12] Nishikawa, H., M. Yokota, A. Yamamoto, K. Taki and S. Uchida
"The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture"
Proc. of Logic Programming Workshop '83 (Portugal 1983)