ICOT Technical Report: TR-013

TR-013

The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture

by

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

June, 1983

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 Personal Sequential Inference Machine (PSI):

Its Design Philosophy and Machine Architecture

Hiroshi Nishikawa Minoru Yokota
Akira Yamamoto Kazuo Taki Shunichi Uchida

Institute for New Generation Computer Technology
Mita-Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

ABSTRACT

As a software development tool of the Fifth Generation Computer Systems (FGCS) project, a personal sequential inference machine is now being developed. The machine is intended to be a workbench to produce a lot of software indispensable to our project. Its machine architecture is dedicated to effectively execute a logic programming language, named KL0, and is equipped with a large main memory, and devices for man-machine communication. We estimate its execution speed is about 20K to 30K LIPS. This paper presents the design objectives and the architectural features of the personal sequential inference machine.

1. Introduction

The final goal of the Fifth Generation Computer Systems (FGCS) project[1] is to develop the basic technology for a totally new computer system which has the ability to handle knowledge information. Inference is for the key mechanism in that system. That is the basic motivation why our project chose logic programming as the basic programming framework.

As one of the actual programming languages based on logic, there is an on-going active move of using Prolog to build new computer applications, especially in the artificial intelligence area. However, the processing power of existing computers is not sufficient for this purpose. It is quite important for the project to rapidly establish the logic programming environments. To satisfy this aim, the Sequential Inference Machine (SIM) is under the development in ICOT.

SIM is mainly intended to be a software development tool, however, the design of its architecture has many experimental aspects. It seemed to be difficult to design the ideal machine at once. So we have taken the following development steps:

1) designing a new programming language based on logic.

2) designing a personal sequential inference machine which is specialized for that new language.

3) designing a new operating system running on that new machine.

4) designing an advanced sequential inference machine based on the experiences from 1) - 3).

As the first step, the logic programming language, called Kernel Language Version 0 (KL0), was designed to take the place of Prolog. KL0 is mainly used to describe system software, such as the operating system kernel, compilers, and interpreters. Therefore KL0 can be regarded as a conventional assembly language except for logic programming features. And a Personal Sequential Inference machine (called PSI:) which is designed to execute KL0 is now under the development as the second step.

The following sections describe PSI design objectives, its system overview, and its machine architecture.

2. Design Objectives

As PSI is considered as a main computing tool in the initial stage of the FGCS project, the main requirements for its design are the high performance and an easy-to-use man-machine interface. Since PSI must be available as soon as possible, the main efforts are

focused on designing its processing unit and memory unit. However, in another aspect, designing PSI can be considered as an experimental step toward the target inference machine.

2.1 Performance Goal

As a software development tool, an adequate execution speed and a sufficient memory space must be provided in order to execute real-world applications.

On this point, it is suitable to compare them with the DEC-10 Prolog system[2]. Because, it is the most popular one and its compiler generates very fast codes.

However, the DEC-10 Prolog system is limited in its memory size (256K words) for users. It is relativity small for actual Prolog applications. This limitation may cause a serious problem in its use. The lower execution speed might be compensated by longer processing time, however, there is no way to continue the program execution if the system has used up such a memory space as a stack. From our experience in using the DEC-10 Prolog system, we estimated that at least a 10 times larger memory space must be necessary. In this situation, the virtual storage system would be an attractive feature, however, its implementation in the Prolog environment involves several problems to be studied. We have to study such problems more deeply as the swapping ratio between main memory and secondary storage, namely the locality of memory accesses, effective cache control mechanism, and an effective garbage collection algorithm working real-time[3]. Therefore we decided to leave the virtual storage system as a future extension. Instead of it, PSI is equipped with a relatively large real memory, maximum 16M words. About execution speed, PSI is designed to attain 20K to 30K LIPS (Logical Inference Per Second) which is the similar performance to the DEC-10 Prolog compiler version running on DEC-2060.

2.2 Personal Use

PSI is designed as a self-contained, personal machine in order to provide its user with powerful computing facilities and an efficient programming environment.

An easy-to-use, sophisticated man-machine interface is the most important features for software development tools. To provide good man-machine communications, PSI is equipped with a bit-mapped display device and a pointing device (a mouse). And a multi-window system is planned to be implemented on them. The input/output devices for Japanese characters will also be included, and PSI will support a word processing system for Japanese.

2.3 Local Area Network

PSI is planned to be connected to a local area network in order to give its user a more productive environment. Although any kind of peripheral devices can be connected to PSI, an usual PSI system will have a limited number of devices according to its own system characteristics.

Through a local area network, the distributed processing system connecting several PSI's can be built. Furthermore, the user can access other machines from PSI, such as a relational data base machine also being developed in the project, and conventional commercial machines.

2.4 Flexibility

PSI has adopted microprogrammed control for flexibility and extendability.

The project has decided on KL0 as a machine level language, however, its usefulness will be verified after PSI completes. In addition, the research and development of new programming languages, such as concurrent Prolog[4][5], is also one of the important subjects in the project. Therefore PSI must be able to execute those experimental languages as their test bed.

2.5 Evaluation

Using PSI, several items of measurements are planned for evaluation on programs behavior and machine design. One is to evaluate characteristics about the execution profile of logic based programs. Another one is to evaluate the validity about PSI architecture and hardware design. Especially, the measurement of memory access characteristics including cache hit ratio is one of the important items, because memory access is the most frequent operation in inference machines. The next advanced models of SIM will be designed utilizing effectively these evaluation results.

2.6 Specialized Hardware Supports

As a first experimental inference machine, an effort has been made to introduce several specialized hardware supports suitable for executing a logic programming language in PSI. To improve unification speed, PSI has hardware buffers. The role of these buffers is to quickly refer to the binding values of variables. For dynamic data type checking, each word has an 8 bit data tag (tag architecture). To

make memory access operations faster, the connection between the main memory and the processing unit was designed as tightly as possible.

PSI design objectives can be summarized as the combination of high performance of the 32 bit "super mini-computer" with the good man-machine interface of the "super personal computer".

3. System Overview

One of the key factors to determine a machine architecture may be the design of the machine instruction set. PSI is a specialized machine for executing the logic programming language (KL0),however, it must have its own operating system to be a self-contained personal machine. This section briefly summarizes the software system and hardware configuration of PSI.

3.1 Language System

One advantage of a logic programming language is to use its non-determinism effectively. However the non-determinate operation is considered unnecessary for describing low-level system control such as the kernel of operating systems, because it mainly consists of determinate operations and thus non-determinate operations would produce redundancy. In general, if a machine architecture is dedicated to some high- level programming language, it becomes difficult to implement its operating system in that language on the same machine. In this situation, a different programming language could be used for system description, however, this approach would degrade the uniformity of the system. And the machine architecture should support two different types of language processing. To make the entire system uniform, we decided to implement a PSI operating system based on the logic programming concept. KL0 has been designed to make this possible.

Figure 1 shows the language system hierarchy. The system programmer uses KL0 directly to develop a compiler, an interpreter, and operating system kernels. From the user's view point, KL0 can be regarded as a machine language of PSI, however, KL0 is basically a high-level, logic programming language. Its features are summarized as follows:

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

o extended execution control facilities

KL0 includes the normal unification mechanism and clause handling mechanism like usual Prolog. From this view point the users can regard PSI as a complete Prolog machine. On the other hand, the users can also specify the machine level control with its extended facilities in KL0.

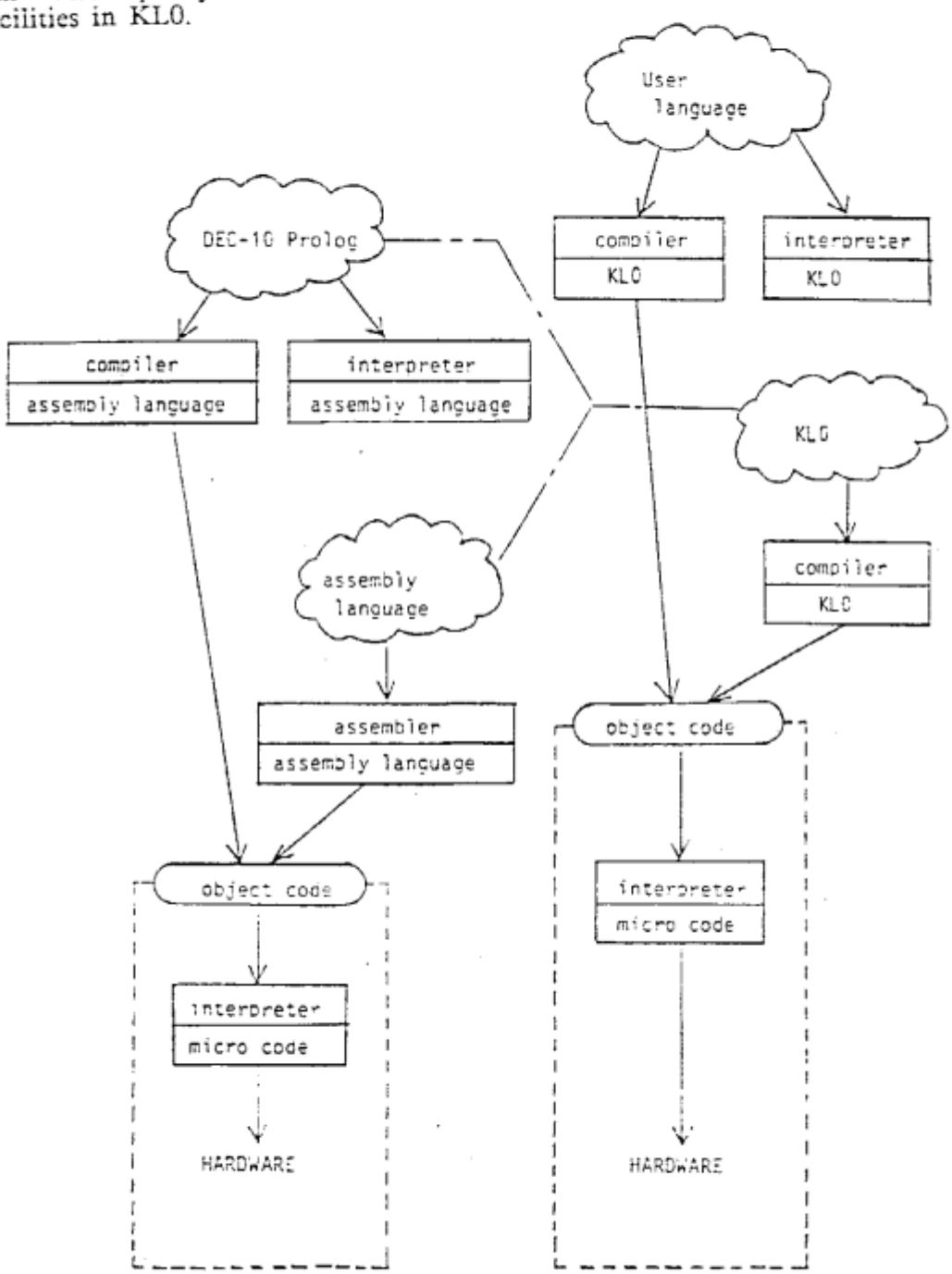

Figure 1. language hierarchy

3.2 Operating System Support

The kernel parts of the PSI operating system are written in KL0. These are transformed into internal machine forms by the compiler which is also written in KL0 itself. Then PSI hardware/firmware directly executes those internal forms. Furthermore, time crucial parts of the operating system kernel, such as the garbage collector or process switcher, are executed directly with firmware. The applications programmers will use a higher programming language than KL0. This language is executed with the interpreter or is compiled by the compiler written in KL0 into internal machine forms.

From the software side, it can be said that the PSI operating system is written in completely a logic programming style. From the hardware side, it can also be said that PSI architecture supports the primitive kernel operating system functions.

3.3 System Configuration

Figure 2 shows the PSI system configuration. CPU has a microprogram sequencer. The capacity of its writable control storage is 16K words. The micro instruction is 64 bit long and is executed in less than 200 nsec.

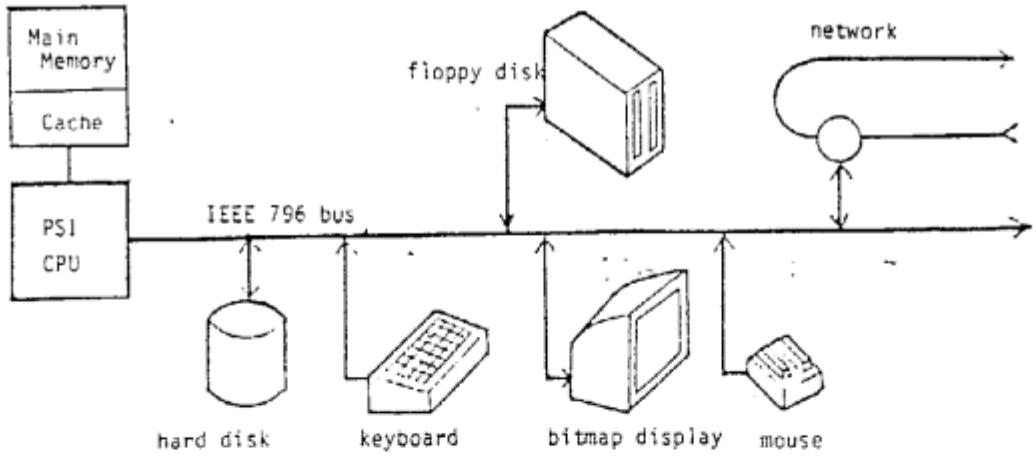

Figure 2. system configuration

CPU interprets internal object forms of KL0 with its micro-coded interpreter. Its hardware mechanism is mainly dedicated for the fast unification. It includes several discrete registers, register files, and an arithmetic operation unit.

The memory unit has a relatively large main memory instead of being equipped with a virtual memory system. A maximum of 16M 40 bit words can be installed. To shorten memory access time, PSI is equipped with a cache memory. It consists of 2 sets of 4K word memory, and a write-back strategy is adopted. Since several stack areas are required for interpretation of KL0 and each stack area will arbitrary grow during program execution, PSI introduces logical memory addressing. Therefore the roles of the memory control unit are address translation and cache control. If the required data exist in the cache memory, PSI can fetch that data within one micro instruction cycle.

A general purpose input/output bus is provided to PSI. To keep design simplicity and generality, IEEE-796 standard bus(MULTIBUS) is adopted. As a minimum configuration, PSI supports a fixed head disk, a floppy disk, a key board, a bit-mapped display, a mouse, a printer, and a local area network interface. Since PSI is planned to be connected to a local area network, the peripheral devices may be selected according to their own characteristics.

PSI also has an additional parallel interface port, in order to satisfy the requirement for connecting special I/O devices directly. For example,this parallel interface will be used to connect the relational data base machine or the voice recognition device, and etc.

4. Machine Architecture

The architecture of PSI was decided based on various considerations. A KL0 program is compiled into the internal object forms of PSI. But the level of the object code has been decided to be higher than that of ordinary machine instructions. So PSI is regarded as a high level language machine. In order to attain high performance, PSI adopted a tag architecture. Furthermore, a cache memory and special purpose registers are provided to improve the unification speed. PSI always refers to memory with logical address, and also has hardware supports for multi-processing.

4.1 How to Design the Machine Instruction Set

KL0 is a logic programming language, however, it is mainly used for system description. Therefore, performance in its execution is crucial. To take advantage of the source program information as much as possible, we decided to employ a compiler and thus PSI executes compiled codes instead of interpreting source codes directly. Even though, after compiling KL0, there still remains many operations to be performed only in execution time, such as unification, because of a

dynamic feature of the logic programming language. Then several levels of machine instructions can be considered.

The lowest one may be the conventional machine instruction level, and a KL0 program would be compiled into small pieces of those primitive machine instructions. The highest one is the internal form which is translated one by one from a source statement of KL0. The desirable machine instruction level depends on the characteristics of the language.

Originally, KL0 contains two different groups of elements. The first group is user-defined clauses to be executed within the logic programming framework. Namely, it is executed based on unification and backtracking. The execution of them is slightly simple and dominated with memory access operations. Therefore, it is undesirable that such execution is broken into many small machine instructions, because many instruction fetches are needed. In addition to this, there is less room for macro optimization on the hardware side because of low level machine instructions. This results in increased redundancies in both execution time and memory usage. We considered that PSI should have these unification and backtrack control facilities by itself.

The second group is built-in predicates for such operations as arithmetic operations and input/output operations[6]. Since the execution of them is performed determinately, their object codes can be represented in compact forms like conventional machine instructions. These built-in predicates are introduced not only to enhance the efficiency of frequently used operations but also to be able to include such primitive operations as register handling and direct memory manipulations used in the operating system.

Consequently the PSI machine instruction set has two types in its internal object forms. The first one corresponds to user-defined clauses. Actually, such a clause is compiled into the sequence of several internal object forms according to source clause definition. PSI interprets that sequence as a machine instruction on the whole. The others correspond to built-in predicates. Basically, a built-in predicate is compiled into one internal machine form.

4.2 Internal Object Forms

A KL0 program is translated into the corresponding internal object form described above. How to represent data and clause is shown in this section.

4.2.1 Internal Data Representation

Through examining PSI machine architecture, providing it with enough ability for increasing requirements from application areas is considered. At least 32 bits are necessary for representing sufficient magnitude of numbers and addressing space. In result, PSI employs a 40 bit word representation as shown in Figure 3. The upper 8 bits represent tag bits (tag part), and the remaining 32 bits represent the data itself (data part). 2 bits of the tag part are used by the garbage collector and the remaining 6 bits indicate the type of data included in the data part.

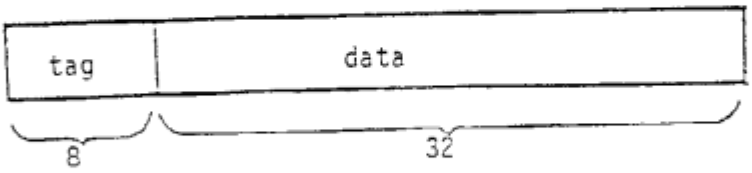

Figure 3. word format

PSI has several internal data types corresponding to ones in KL0. The visible data types for user are listed below:

- o symbol
- o integer
- o real
- o vector
- o string
- o local variable
- o global variable
- o void variable

(a) Symbol

This indicates the identifier of an atom. In the data part, the symbol number corresponding to an atom is stored. The printing image of an atom is managed by the operating system. So there is no direct relation between the symbol number and its printing characters.

(b) Integer, Real.

These are numerical data on PSI. The value of them is stored in the data part.

(c) Vector

A vector is a block of continuous memory slots, and is used to represent various structured data such as binary trees. As shown in Figure 4, a vector is usually accessed by way of its descriptor. However, this representation always needs an extra memory access whenever a vector is accessed. Since it is supposed that the vector which has a few elements is frequently used in programs, the direct vector type is introduced in order to effectively access such vectors. The conventional list structure is an example, and its representation

is shown in Figure 5. Comparing the performance of the structure sharing[7] with that of the copying strategy on structured data handling, PSI employs the structure sharing method similar to the DEC-10 Prolog. Therefore the structured data are manipulated as the pair of a structure (representing in a vector) and its values (located in the global stack). This address pair is called a molecule.

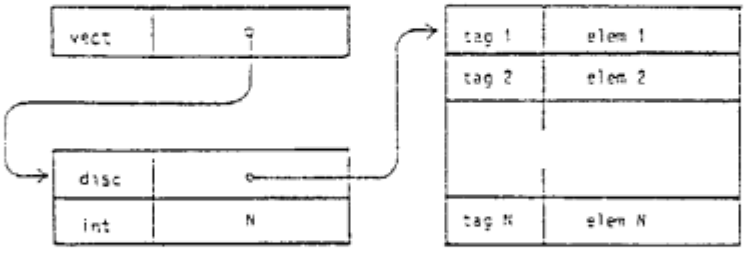

Figure 4. vector representation

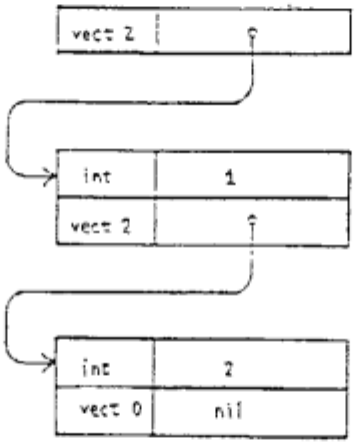

Figure 5. list representation

(d) String

A string data type is introduced for manipulating a byte (CHARACTER), double bytes (KANJI), and a bit (FIGURE) string data. Like the vector representation, string data is also accessed by way of its descriptor.

(e) Local/Global Variable

This data type indicates a local/global variable included in a clause. In the data part, the variable number is stored. The instance of a local variable is created in the local stack. The instance of a global variable is created in the global stack. Roughly speaking,the difference between the two is that the instances of local variables are cleared when the clause including them are executed

determinately, however, those of global variables are not cleared.

(f) Void Variable

This type means that the variable can have an arbitrary value, namely can be unified to any type of data.

4.2.2 Clause Representation

The definition of a clause in KL0 is the same as the one in Prolog. It consists of a head predicate and several goal predicates. The compiler translates a clause into a corresponding internal object form. As shown in Figure 6, each clause is represented as continuous memory slots, called code, in PSI. A code is a similar data type to a vector, and consists of a clause header, head arguments, goal predicate name and its arguments.

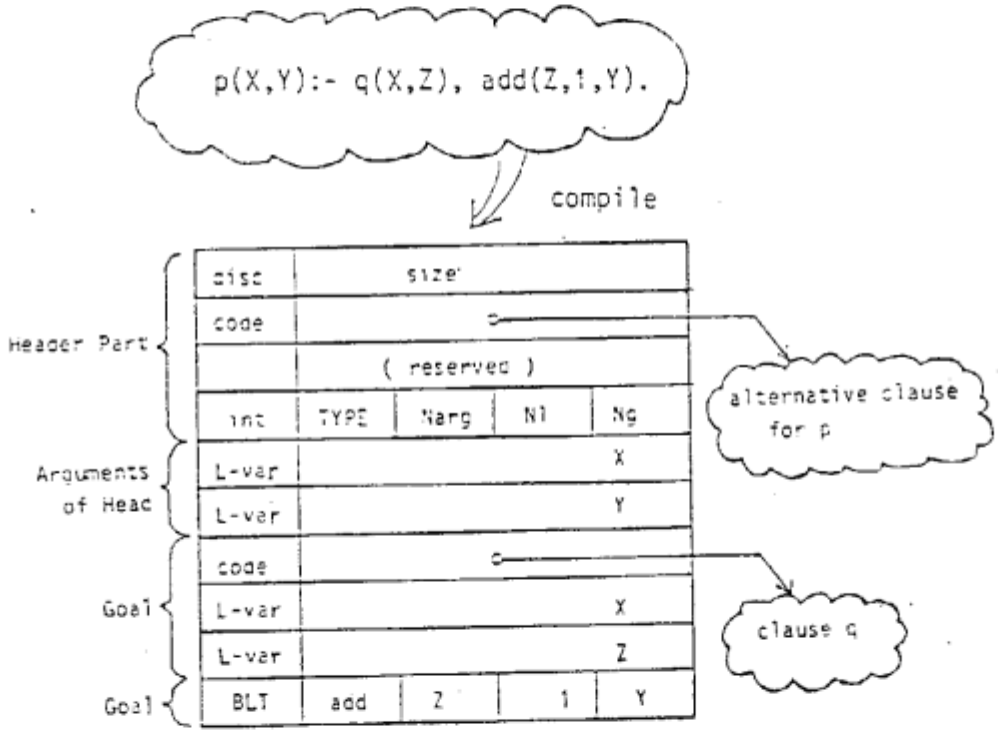

Figure 6. clause representation

A clause header consists of four words. The first word indicates the size of the code. The second word has an address to the code representing the next alternative clause. The third word is a reserved word. It might be used by the garbage collector. The last word indicates attributes of the clause. TYPE shows the clause type. For example, it is a unit clause, or having alternative clauses etc.

Narg shows the number of arguments included in the head predicate. Nl/Ng shows the number of local/global variables included in this clause.

Following a clause header, the head predicate arguments are located. Each argument is represented in the data types described in 4.2.1.

The remaining codes show the internal form of goals. There are two types of goal representation according as the called goal predicate is a built-in predicate or not.

(a) User-Defined Predicate Call

A goal predicate name is compiled into the pointer to the code representing the called clause. This pointer is stored in the data part and the tag of this pointer is set to a code type. The goal arguments are arranged continuously, following this pointer.

(b) Built-in Predicate Call

In the data part, a compact representation of machine instructions is stored and it consists of an 8 bit operation code and three 8 bit operands. The role of built-in predicates is to create objects, test the attributes of objects, and manipulate objects etc. A built-in predicates is compiled into one word object code basically, so that it can be executed efficiently on PSI.

Each goal is compiled into the pointer of the corresponding clause and its arguments. There are three connection types of goals, which PSI can directly interpret with its firmware interpreter.

(a) AND Connection

AND connection shows that the goals are combined as an AND node in the AND-OR search tree. Each goal is continuously located as shown in Figure 7-(a). AND connection means that each goal is executed sequentially and if a goal is failed, then backtracking occurs.

(b) OR Connection

This type is used to represent an OR connection included within a clause. OR connection shows that each goal is combined as an OR node in the AND-OR search tree. This connection is realized by an OR instruction as shown in Figure 7-(b). At first execution, the first goal is tried. When they fail, then the second goal is tried. Each branch of the OR connection can be composed of several goals. Therefore each branch of an OR connection is the same as an ordinary alternative clause except that they are included in only one clause and require no unification process.

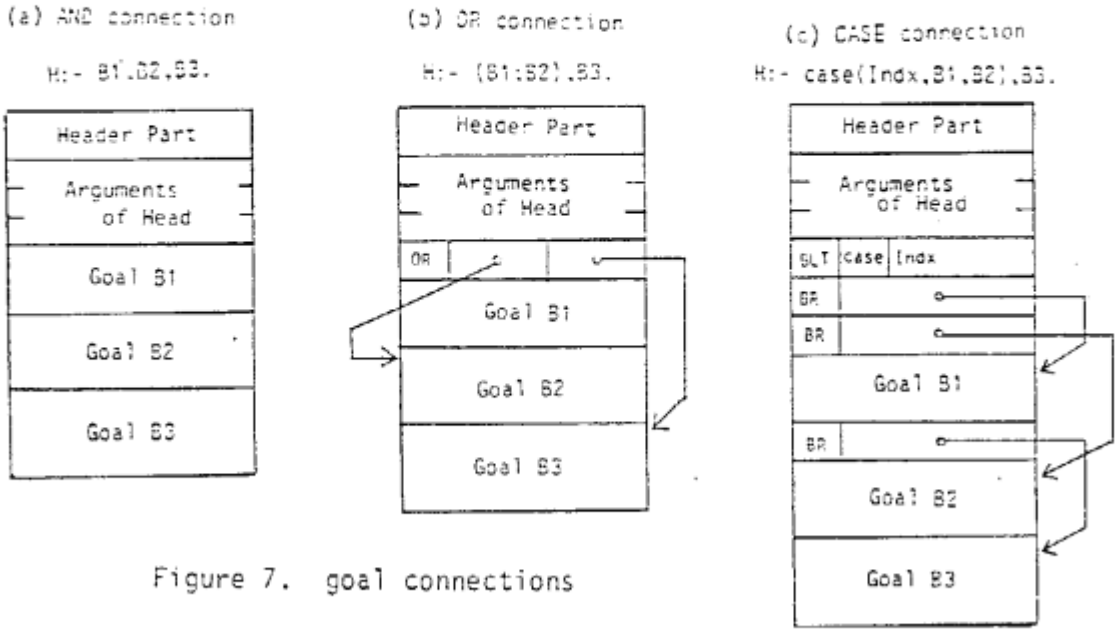

Figure 7. goal connections

(c) CASE Connection

CASE connection can be regarded as the arrangement of indexed goals. Figure 7-(c) shows the internal format of CASE connection. One of the goals is selected by the result of CASE instruction and if it is successively executed then the goal following the case block is executed next. Even if backtracking occurs, unlike OR connection, the remaining indexed goals are not executed.

4.3 Execution of PSI Internal Object Forms

For interpretation of the KL0 program, the following four stacks are needed:

- o local stack
- o global stack
- o trail stack
- o control stack

The use of these stacks is similar to those of DEC-10 Prolog, however, the control stack is separated from the local stack in order to efficiently execute the extra control primitives of KL0.

The local stack is an instance region for local variables. Preceding the unification process, PSI allocates the stack entries according to the number of local variables included in a clause. These stack entries are popped up when the evaluation of the clause including them is determinately terminated, or unification fails. They are also cleared when it is pruned by a "cut" operation.

The global stack is an instance region for global variables. Similar to the local stack, PSI allocates stack entries according to the number of global variables. These entries are only popped and cleared when unification fails. In addition, a molecule generated during unification and some control information are also allocated in this stack.

The trail stack is used for undoing variables when backtracking occurs. In this stack, binding information (i.e. the cell address where a value is stored during unification and whose content must be changed to 'undefined' when unificatoin fails) is stored. When the instance value of a variable is modified, its old contents are also stored in the trail stack in addition to its cell address.

In the control stack, various book-keeping information required for the execution control is stored. All of them are pointers which represent the execution environment of corresponding clauses. They are used to return to the calling clause, or to the backtrack point when unification fails.

There are some data types dynamically generated during program execution. Some of them are described below.

(a) Reference

It indicates a pointer generated during unification.

(b) Molecule

PSI adopts the structure sharing method to represent structured data described before. Since a molecule consists of two words in PSI, it can not be located into a variable cell. Therefore, a molecule itself is allocated in the global stack, and the reference to it is located in the variable cell.

4.4 Address space

PSI has a 32 bit logical address space. It is composed of 256 logically independent areas. The size of each area is 16M words, and managed by pages of 1K words. The reason why the concept of area is introduced is as follows:

(a) Since PSI supports multi-processing, it is desirable for the

operating system to assign completely independent areas to each process.

(b) Since PSI firmware interpreter uses four stacks described in section 4.3, it is desirable to be able to expand each stack area independently. If these stacks are allocated to the same space, a collision between stack areas will occur. At that time, one of them must be moved to another space. This situation causes serious overhead time.

Since four stacks are required for interpretation of a KL0 program, it means that each process needs at least four areas for its execution environment. On the other hand, code areas might be shared among many processes. If four areas are assumed to be used for code areas, namely heap areas, a maximum of 63 processes can be created on PSI from 256 areas.

Each area is divided into 16K pages. A page consists of 1K words. An area is managed by PSI operating system in page units. PSI allocates one page when a process needs more memory. On the other hand PSI disallocates some pages when a process release memory.

In result, the memory address field is divided into an 8 bit area number, 14 bit page number, and 10 bit offset as shown in Figure 8.

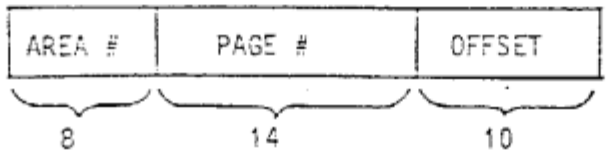

Figure 8. address format

The address translation mechanism is shown in Figure 9. The translation from a logical address to a physical address is performed with an area table and a page table. Each area table entry shows the base address of a page table located in the page map table corresponding to an area. And each page table entry shows the physical page address corresponding to a logical address page. As a first step to generate a physical address from a logical address, the area table is accessed using the area number, and a page table base address is obtained. Then the page map table is accessed using the sum of that page table base and a the page number. Finally concatenating the output of the page map table and the page offset, a 24 bit physical address is obtained.

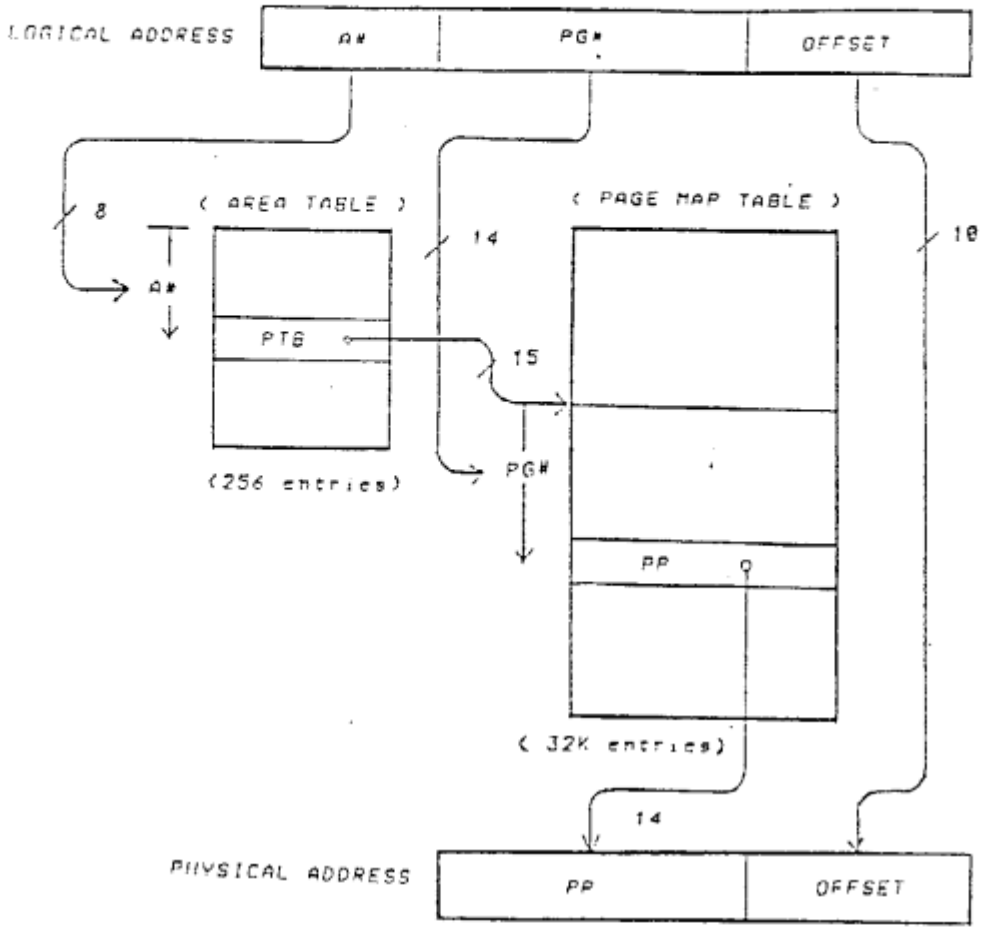

Figure 9. address translation

To achieve address translation, it is common to use a Translation Lookaside Buffer(TLB). Each TLB entry contains a logical page address and corresponding physical page address. TLB is a sort of cache memory, and if the address pair corresponding to a logical address is stored in it, there is no reference to the translation table existing in the main memory. PSI does not adopt this method. Instead, the area table and the page map table are located in special fast memory. In result, the address translation process is performed within a micro instruction cycle. The reasons why TLB is not adopted are shown below:

(a) If the address pair is not in TLB, the translation table in main memory must be accessed to generate a physical memory address.

(b) Since the garbage collector must search all memory space, it is supposed that the memory access locality during garbage collection is not so high. Therefore, TLB might not work well in that situation.

(c) PSI does not adopt virtual memory. The total amount of page map table entries can not exceed the number of physical pages. Since the maximum size of main memory is 16M words, it is sufficient to have 16K entries in the page map table.

The size of an area can extend from one page to 16K pages. Before program execution, the maximum number of pages used in a area cannot be predicted. Furthermore, a process needs at least four areas, however, the utilization of each area is different among processes. Accordingly, as the number of page table entries increases during execution, a page table may collide with another page table within the page map table. To avoid that case when possible, it is desirable to locate each page table corresponding to an area as dispersively as possible. Also, the page map memory size should be larger than the number of physical pages. To satisfy this condition, PSI has a page map memory of 32K entries. Since the standard physical memory size is 4M words, the size of a page map memory is eight times larger than that of physical pages.

If a page table collision occurs in page map memory, a trap occurs and page table relocation must be done. There are many algorithms to be considered. It is a future research theme to examine which algorithm is better.

4.5 Hardware Supports for Fast Unification

Unification plays an important role in executing a KL0 program. To efficiently execute the unification process, the hardware support mechanism is indispensable. The major part of the unification process is memory access and data type checking. The facilities employed in PSI are as follows:

- o Cache memory
- o Tag bits
- o Frame buffer

(a) Cache Memory

The merit of using cache memory is to reduce the cost of all memory accesses besides stack access. PSI adopts the write back-strategy for cache control, not the write-through strategy. A cache memory manages logical addresses. Therefore, if the accessed data exists in cache memory, no address translation is needed. The address translation is required only if a cache miss-hit occurs. PSI memory controller performs address translation during the cache memory access in parallel. This mechanism creates no overhead time for translation when the cache memory miss-hit occurs.

(b) Tag Bits

A tag is essential to effectively interpret a data type. To realize fast unification depends on how rapidly the data types can be examined. For this aim, tag bits are attached to all data, and they specify the type of the data. A special hardware mechanism, which decodes tag bit pattern efficiently, is provided in PSI.

(c) Frame Buffer

Frame buffer is the set of special registers provided for the top of the stack frame. In this buffer, the arguments of a clause and the cells of local variables are stored. Most of the unification is done using this buffer. This reduces the number of memory accesses, and faster unification will be realized. Furthermore, using this buffer, Tail Recursion Optimization (TRO)[8] can be realized efficiently.

4.6 OS Support

Since PSI is designed as a self-contained system, it requires own operating system. This operating system consists of an end-user interface (command interpreter), a programming system (editor, debugger), a file system, and so on. To provide its users with a sophisticated programming environment, that operating system must be an easy-to-use system, and provide good man-machine communications. Considering that these systems are specified by KL0,it is desirable that PSI must have operating system support functions.

To attain this objective, PSI has various hardware and firmware supports. For example, such primitive operations as a memory allocation or a garbage collection included in the memory management system is directly performed by firmware. The process switching of the process management system is also performed by firmware. Furthermore, PSI holds the process information in fast CPU memory in order to reduce process switching overheads. This is an essential hardware support in PSI, because KL0 requires larger execution environment than ordinary programming languages, and without that hardware support the contents of many base registers must be saved into the main memory at process switching.

In addition to higher level operating system support, there are several KL0 built-in predicates which perform low level system control, such as hardware resource handling, direct memory manipulation, and input/output control. These built-in predicates are effectively executed by firmware.

Besides this support described, garbage free regions is introduced to support the operating system kernels. In this region no garbage collection is done. This means that a program running in this region can be executed even while a garbage collection process is being executed. Those special processes unconcerned with the garbage collection are called supra GC processes. The aim of introducing this

GC-less process is to maintain good man-machine interface even when the garbage collector is working.

Summarizing those, the hierarchy from the end-user interface language to the hardware on PSI is shown in Figure 10.

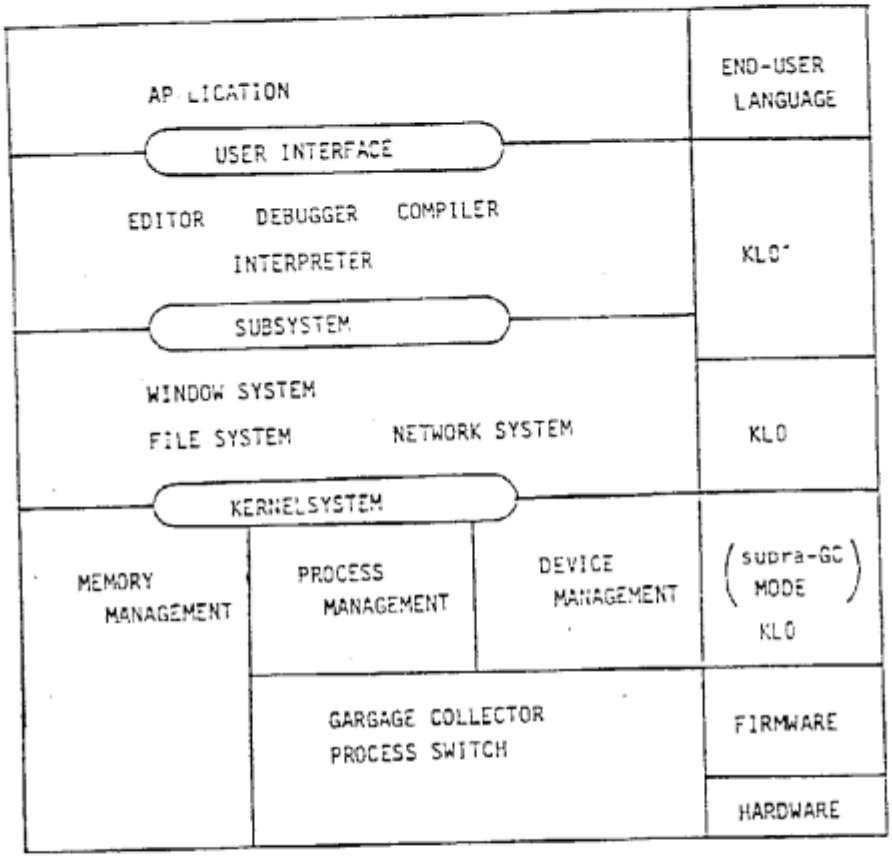

Figure 10. operationg system hierarchy

5. Conclusion

In this paper, we described the design objectives and the machine architecture of a Personal Sequential Inference machine, PSI. Its detail hardware design has almost been completed and the microprogrammed KL0 interpreter is now under the design. The rough estimation of PSI execution speed is comparable to the compiled codes of DEC-10 Prolog system on DEC-2060.

PSI is a first step toward the target inference machine which will be attained in ten years. For designing next advanced SIM, we are planning several evaluations on PSI. Many software products will also be made on PSI. We believe that PSI will be a powerful and useful workbench for our project.

ACKNOWLEDGMENTS

The authors express their grateful thanks to Dr. Takashi Chikayama for his valuable advice, and to Mr. Kazuhiro Fuchi, Director of ICOT Research Center and to Dr. Kunio Murakami, Chief of First 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] 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)

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

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

[5] Takeuchi, A., et al. Interprocess Communication in Concurrent Prolog. Logic Programming Workshop, '83 (1983)

[6] Chikayama, T., et al. Fifth Generation Kernel Language. Proc. of the Logic Programming Conference '83 (1983)

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

[8] Warren, D.H.D. An Improved PROLOG Implementation Which Optimizes Tail Recursion. D.A.I Research Report No.141, Department of Artificial Intelligence, Univ. of Edinburgh (1980)