TR-075

HARDWARE DESIGN AND IMPLEMENTATION OF THE PERSONAL SEQUENTIAL INFERENCE MACHINE (PSI)

by
Kazuo Taki, Minoru Yokota, Akira Yamamoto,
Hiroshi Nishikawa, Shunichi Uchida
(ICOT)
Hiroshi Nakashima, Akitoshi Mitsuishi
(Mitsubishi Electric Corp.)

August, 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

HARDWARE DESIGN AND IMPLEMENTATION OF THE PERSONAL SEQUENTIAL INFERENCE MACHINE (PSI)

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

ICOT Research Center
Institute for New Generation Computer Technology
Tokyo, Japan

and

Hiroshi Nakashima, and Akitoshi Mitsuishi

Information Systems and
Electronics Development Laboratory
Mitsubishi Electric Corp.
Kamakura, Japan

ABSTRACT

The Personal Sequential Inference Machine (PSI) is a personal computer designed as a tool for software and hardware development in Japan's Fifth Generation Computer Systems (FGCS) project. This paper describes PSI's hardware systems and the unique features of its data processing and sequence control units.

The PSI system adopts a logic programing language as its primary language. It consists of a large main memory (16 mega words), interactive I/O devices, and operating system support and language support hardware. PSI's machine language is a high-level language based on logic programming, and its description level is very similar to that of Prolog. It is called Kernel Language Version 0 (KL0). Unification and backtracking, the principal operations of KL0, are performed by the KL0 firmware interpreter in cooperation with several dedicated hardware components. These include branch and dispatch facility testing tags, a cache memory designed for stack access, and a high-speed local memory (called a work file) designed for use in tail recursive optimization.

Commercially available high-speed Schottkey TTL ICs are used in the CPU. Printed circuit boards for the CPU, main memory, and I/O controllers are mounted in a single cabinet along with secondary storage devices. A prototype machine has been manufactured and microprogram development is nearly complete.

1 INTRODUCTION

Japan's Fifth Generation Computer Project has started using a new logic-based programming language as its primary language for both software and hardware research and development. However, programming environments for such languages have not been sufficiently developed in conventional computer systems in terms of their processing speed, memory space, language support, and flexibility for experimentation. In order to build a research and development tool fulfilling these requirements, a high-level language machine specialized for logic programming is under development at ICOT and supporting companies [Uchida 83], [Yokota 83], [Nishikawa 83]. The machine is called the Personal Sequential Inference Machine (PSI), reflecting its machine features and functions.

To develop a viable programming environment for logic programming, several targets have been established for PSI, as follows:

(a) Efficient execution of logic programming language KL0 (Kernel Language version 0), which is the machine language of PSI [Chikayama 84-1]

(b) A machine architecture that supports the SIMPOS operating system developed for PSI [Hattori 83]

(c) Memory size and execution speed sufficient for executing large application programs. Specifically, as compared with the compiler version of Dec-10 Prolog [Bowen 81] on the Dec-2060, PSI will have a maximum of 16M words of memory, which is 64 times larger than that of Dec-10 Prolog, and will attain approximately 30K LIPS (logical inferences per second) in processing speed, which is almost equivalent to the Dec-10 Prolog on the Dec-2060.

(d) Highly interactive I/O devices, such as a bit-mapped display, mouse, etc.

(e) A local area network (LAN) system for inter-PSI communication and resource sharing

(f) Reasonable physical size for personal use and practical cost-effectiveness

(g) Reliability as a research and development tool

(h) Early availability

Other target specifications are listed below. These involve the plan to use PSI as a tool for architectural research into efficient execution mechanisms for logic programming.

(a) The adoption of hardware mechanisms resulting from ICOT research that increase unification speed

(b) A flexible microprogrammed sequence controller with large writable control storage

(c) Hardware and firmware evaluation facilities for measuring dynamic characteristics and collecting statistical data

To satisfy these specifications, we have proceeded with the designs for the architecture and the hardware. In this paper, PSI's hardware system and its unique features will mainly be described. The PSI architecture is presented first, then the hardware configuration and the detailed specifications for the specially designed part of the PSI CPU are described. The action and usage of each hardware component of the CPU at the time of program execution are also mentioned.

2 PSI ARCHITECTURE

In this chapter, we summarize the hardware architecture of PSI mainly from the machine-language level, i.e., from the system programmer's point of view.

2.1 Word format

A word consists of 40 bits, as shown in Fig.1. Eight bits are used for a tag and 32 bits for data. The tag contains two mark bits for garbage collection (GC tag) and six bits for a data tag that represents one of the following data types:

undefined, symbolic atom,

integer, floating point number,

stack/heap vector, string, code, built-in code,

local/global variable, local/global reference,

hooked variable, control marks, etc.

2.2 Machine Instructions

KL0, a logic programming language whose specifications are almost equivalent to those of Dec-10 Prolog, is designed to define the functions of the PSI machine instructions. The representation of the machine instruction, shown in Fig.2, is a simple converted form of the KL0 source program. Each instruction code can correspond to each component of the source program. The machine instruction is executed by the firmware interpreter, by which unification and backtracking are also performed. The reason for the adoption of high-level machine instructions is discussed in section 4.1.

The representation of the machine instruction of a KL0 clause contains a clause header, head arguments and some body goals (cf. Fig.2). These body goals include user-defined predicates (which are actually pointers to the instruction representation of the clause and arguments) and built-in predicates. Most built-in predicates have a compact format that contains one operation code and at most three arguments in one word. Each of the arguments has a 3-bit tag and 5-bit data. When the built-in predicate is executed, a corresponding firmware subroutine is called directly according to the operation code. If integers or variable numbers appearing as arguments are small enough to represent in five bits, most built-in predicates can be packed in one word. This representation is quite effective in saving memory space and shortening the execution time.

2.3 KL0

The specification of KL0 is summarized as follows:

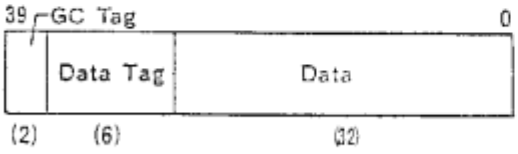


Fig.1 Word Format

(a) It is based on a subset of Dec-10 Prolog.

(b) It has extended control structures.

(c) It has hardware control functions.

"Subset of Dec-10 Prolog" means that KL0 does not include built-in predicates compatible with those of Dec-10 Prolog for internal data base management, such as Assert or Retract, and I/O predicates, such as Read or Write. These predicates are replaced by user-defined predicates using primitive built-in predicates of the hardware control functions.

The hardware control functions correspond to direct hardware operations to handle hardware registers, memory, and the I/O bus.

The extended control structures [Takagi 83] contain such functions as Bind-hook, On-backtrack, Extended-cut, etc. Bind-hook is a special function for procedure invocation, which calls a previously registered procedure when a specified variable is unified to a value. On-backtrack is a function that invokes a previously registered procedure only when backtracking occurs and control returns to the registration point of the procedure. Extended-cut specifies the level of a predicate's call to cut the or nodes of that level. These extended control structures enhance the descriptive power of the language, but require many run-time supports (operations that can not be determined during compilation).

```
p1(X,Y,test) :- p2(X,Z), add(Z,5,A), p3(A,256,Y)
```

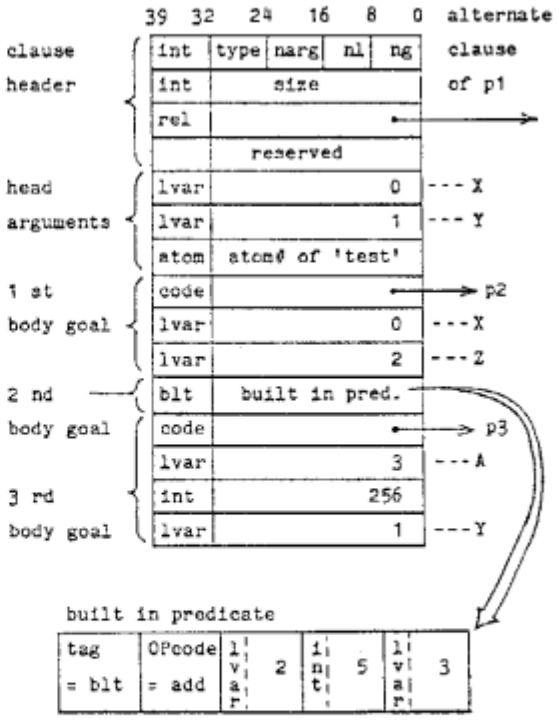


Fig.2 Representation of Machine Instructions

All the system programs and the user programs are written in logic programming language ESP (Extended Self-contained Prolog) [Chikayama 83-2] [Chikayama 84-2], which is the system description language and user language of PSI. These programs are compiled into KL0 for execution.

2.4 Execution environment for KL0

To execute KL0 programs, the interpreter uses four stacks, namely, local, global, control, and trail stacks, and one heap area. The heap area is used to store machine instructions and vectors (vectors include usual structured data such as lists and arrays). For representing structured data, the structure-sharing method [Warren 77] is used. The utilization of the stacks and execution control mechanisms are basically the same as in Dec-10 Prolog [Bowen 81][warren 77]. However, Dec-10 Prolog's local stack is separated into control and local stacks in PSI, because an independent control frame is needed for extended control structures.

Fig.3 shows the execution environment of KL0 during unification. There are machine instructions for the clauses of 'caller' and 'callee' in the heap area, and instruction pointers for each. As in Dec-10 Prolog, a group of variable cells, called a frame, is made corresponding to caller or callee. These frames are placed on the local stack for variables and on the global stack for variables in the structured data. To access a variable cell, the relative distance from the frame base that is pointed to by a frame-base pointer is used. The control stack is used to store frames containing information of the return chain and the backtrack chain, as well as pointers to the environment for continuing execution at the return point. The trail stack is used for storing cell addresses that must be recovered to the initial state on backtracking and is accessed using its stack-top pointer. These pointers, namely, instruction pointers, frame-base pointers, and stack-top pointers, constitute the execution environment of KL0.

2.5 Address representation

To execute KL0 programs, four stacks and a heap area are required. Concurrent execution of multiple processes is necessary for PSI, and sharing of instruction codes and variable spaces among the processes are also required. To satisfy these requirements, the address space is divided into independent logical spaces, called 'areas', and each is identified by an area number. An area can be assigned to one of four stacks of a process, or to a heap area shared among processes for code storage and common variable spaces. Thus, the address representation of PSI, shown in Fig.4, contains an 8-bit area number and a 24-bit inner area address. This means that there can be up to 256 areas, each of which can be assigned physical memory up to 16M words.

2.6 Address translation

PSI can have up to 16M words of physical memory. To allocate and relocate physical memory more efficiently to each area, an address translation mechanism is introduced. Physical memory is managed in 1K-word pages. Pages are allocated to each area on demand, and deallocation is performed by a garbage collector. Fig.5 illustrates the address translation mechanism, which is performed using two tables, one for the page map base and another for the page map.

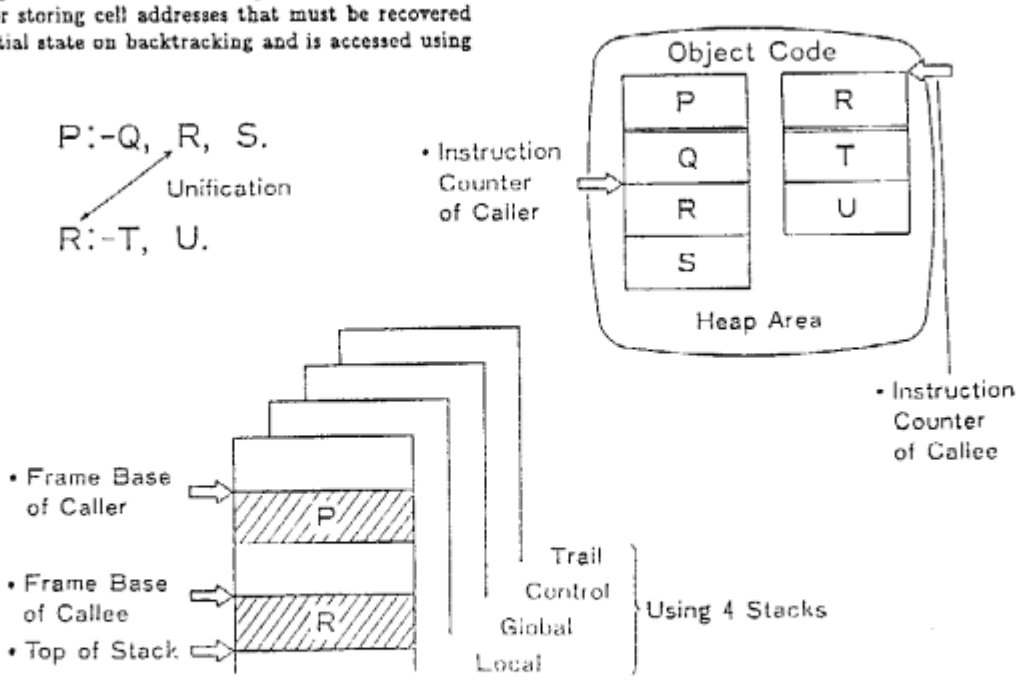
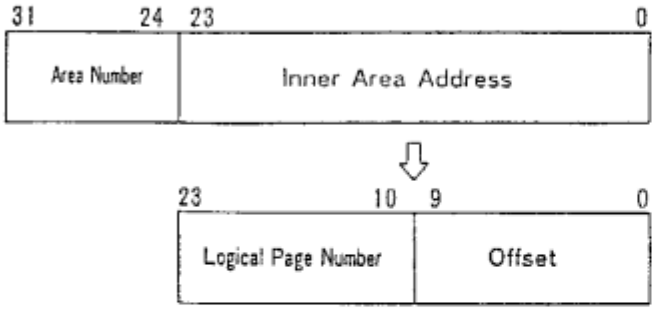


Fig.3 Execution Environment for KL0



An Area: A Logical Address Space of Maximum 16 M Words

Whole Address Space (32 Bits): Consisting of Independent 256 Areas

Fig.4 Address Representation

2.7 Multiple processes

Many programs, such as the editor, compiler, device handlers, and user programs, are executed as different processes in PSI. Each of these processes has a process status that includes KL0 execution environment and hardware control information, such as processor priority for interrupt processing. This environment and information is collected in a table, called a process control block (PCB).

The PCB of an inactive process is stored in a local memory in the CPU, whereas the PCB of an active process (current PCB) is distributed in CPU registers. The contents of the current PCB are swapped by the firmware when process switching occurs. Process switching is initiated by an interrupt or various built-in predicates. The maximum number of processes is 63 due to the limitation on the number of areas. However, this is sufficient for the operating system and most user programs.

2.8 Interruption

A vectored interrupt system is adopted in PSI. An interrupt vector is prepared for each interrupt source (e.g., an I/O device) and a registered process identifier is assigned to each vector. When an interrupt occurs, a process is switched to the corresponding registered process by the firmware. There are eight interrupt levels, two for external and six for internal interrupts. PSI also has a non-maskable trap system to deal with errors that occur during program execution.

Garbage collection (GC) is performed as an independent process in PSI and is invoked by a GC trap. However, some interrupts, such as hardware errors and urgent interrupts from I/O devices, may take priority over garbage collection. These urgent interrupts are handled by special processes, called supra-GC processes, that use some special areas for stacks and a heap. These areas, called GC-less areas, are not subject to garbage collection.

3 SYSTEM CONFIGURATION

3.1 Configuration of the total system

Fig.6 shows the system configuration of PSI. The PSI CPU contains a sequence control unit, a data processing unit, a memory module, which includes a cache and an address translation unit, and an I/O bus interface unit. These are connected to each other by internal buses. The PSI I/O system contains a IEEE-796 standard bus and several I/O devices. A console processor is connected to the CPU for maintenance, initialization, and debugging support. A mini-computer (PDP11/23plus) can also be connected to the CPU instead of the console processor as a more powerful debugging aid.

3.2 I/O devices

PSI has the following I/O devices:

a bit-mapped display (1200 x 900 pixels),

an optical mouse, a keyboard,

hard disk drives (37M bytes x 2),

floppy disk drives (1M bytes x 2),

a local area network (LAN),

and a serial printer

Some commercially available devices that have IEEE-796 standard interfaces can also be connected. There is a 512K-byte buffer memory on the I/O bus, which is used for data transfer to secondary storage devices and the LAN. The buffer memory is also managed as a disk cache by the software. The bit-mapped display controller has raster operation functions and independent image memory. The image memory can store more than ten full screen images and character fonts. Window images are normally stored here to decrease the load on the I/O bus.

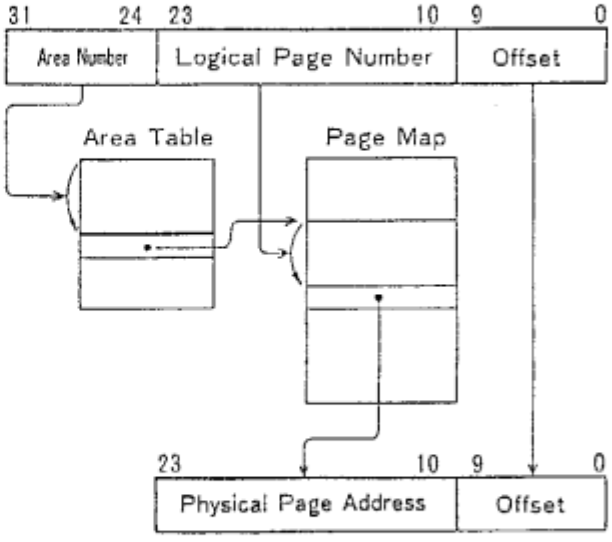


Fig.5 Address Translation Mechanism

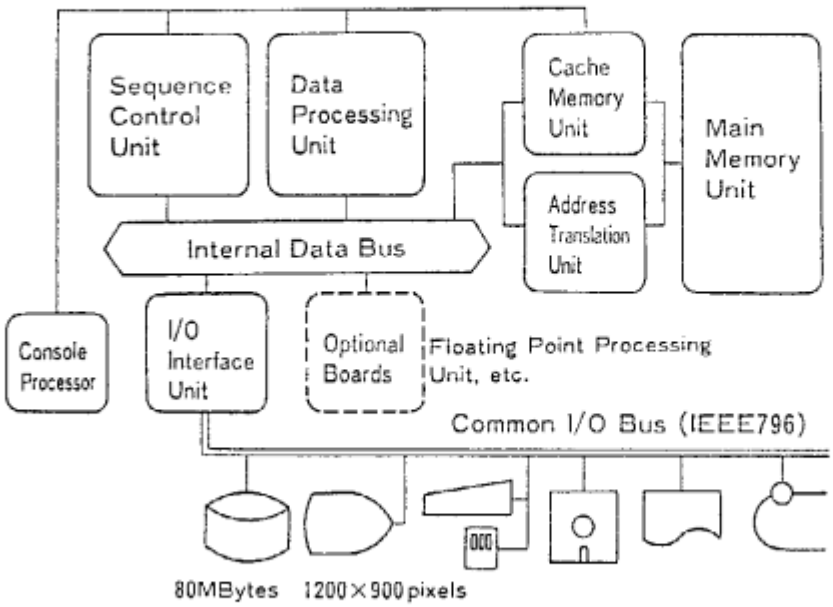


Fig.6 System Configuration

4 HARDWARE DESIGN

4.1 Basic design concepts

In the PSI hardware design, priority was given to sufficient execution speed and large memory space, while keeping reasonable physical size and early availability. To satisfy these requirements, the design philosophy called for avoiding hardware complexity and for utilizing microprogram techniques. However, the PSI CPU has adopted some specialized hardware mechanisms concentrating on speedup of the KL0 firmware interpreter, especially of unification and execution controls. To satisfy the requirement for fast development, it was decided to utilize commercially-available LSIs and time-tested implementation techniques. In this section, the basic design concepts that determine PSI hardware architecture are discussed.

(1) Machine instruction level and CPU architecture

Two different design methods were considered concerning the machine instruction and CPU architecture designs. The first method takes high-level machine instructions, whose level is nearly the same as the source language level, like those of PSI. The representation of the machine instructions can closely correspond to the source program; thus, the size of the instruction code is held down. In this method, machine instructions are executed by the firmware interpreter. For interpretive execution, it is useless to adopt such heavy hardware as an instruction pre-fetch unit or a pipelined execution unit because microprogram branch occurs very frequently and it breakes the execution pipes. The second method is to choose low-level machine instructions. In this method, source programs are compiled into machine instructions, fetched by an instruction pre-fetch unit and executed less interpretively (determinately) by a pipelined execution unit (for example, [Tick 84]). In this method, the determinate instruction execution mechanism makes the hardware easy to optimize; thus, it is more suitable for high-speed execution. However, the complexity and the amount of the hardware will increase.

The extended control structures of KL0 [Takagi 83] require several run-time supports (operations that can not be determined during compilation). These run-time supports are easily realized by the interpretive execution method which doesn't require complex hardware. And translation cost between source programs and machine instructions is very low for the method. For these reason, PSI has chosen the interpretive execution method. Since the method doesn't require frequent memory access to the instruction codes, because of the small instruction code size, the instruction pre-fetch unit or the instruction cache memory can be omitted. The interface between the CPU and the main memory is then simplified to a single connection between the CPU and one cache memory. As a result, PSI has adopted a simple hardware architecture. However, several hardware components are specially designed to enhance the performance of the KL0 firmware interpreter.

(b) Speeding up stack access

In executing a language like Dec-10 Prolog, information for backtracking is often pushed and left on the stack, and thus the frame of the caller clause is often buried deep in the stack. Because of this, the stack accesses scatter both to the top and to the inner part of the stack. Hence, a stack cache that has only a copy of the stack-top data in high-speed memory doesn't work efficiently. An independent hardware stack is also unsuitable because it is not large enough to be used for the global stack KL0 requires. Accordingly, a cache memory that is a more general hardware facility has been chosen to speed up PSI and a few functions suited to stack access have been adopted for the cache memory.

(c) Specialized hardware

Data paths and the basic CPU control timing have been kept as simple as possible. However, branch mechanisms for micro instructions, such as conditional branch and dispatch, which are often used in the firmware interpreter, and a register file used for the tail recursion optimization [Warren 80] are specially designed for the efficient execution of KL0. These are described in detail in following sections.

4.2 Micro Instructions

4.2.1 Control features

A very simple pipelined control is used to fetch the next micro instruction in parallel with execution of the current micro instruction. The branch control circuit is designed so that the execution result of a micro instruction, such as an ALU flag or a register value, can be used in the immediately subsequent micro instruction as a jump condition or as dispatch source data. These simplify microprogram coding and increase the execution speed of unification, which uses many branches and dispatch operations.

4.2.2 Micro instruction format

As shown in Fig.7, a micro instruction has a 64-bit word length and has a field assignment that enables effective parallel control of hardware resources. There are three micro instruction types. They have common fields between bit 63 to 22. These fields mainly specify data operations. Bits 21 to 0 have different meanings in each instruction type and mainly specify branch controls and ALU operations.

Type 1 instructions specify various conditional branches and dispatch operations. Relative addresses (up to $\pm$ 256) are used for conditional branches. Arithmetic operations are available in type 1. Type 2 instructions specify absolute jump, logical operations, and bit rotation of the barrel shifter. Type 3 instructions specify various operations, such as arithmetic and logical operations, bit rotation, tag replacement with immediate data, I/O bus controls, etc. However, jump operations are limited to indirect jumps using the jump register.

The three-operand operation is specified by the data operation fields. Namely, two operands specified by the SC1F and SC2F fields are processed by the ALU and stored in a register specified by DSTF field in one micro instruction cycle. One of frequently used registers can also be specified as an destination register by the multi-destination field. Data specified by the SC2F field can be shifted and masked by the barrel shifter and field extractor before the ALU operation. Memory access control is specified by the cache control field (CCF) independent of the data operations mentioned above. DRF and LARF fields specify the selections of the data register and the logical address register used in memory access from PDR and CDR, and PLAR and CLAR. LAIF specifies the automatic increment of the logical address register.

4.3 Data processing unit

4.3.1 Configuration of the data processing unit

Fig.8 shows the configuration of the data processing unit. A register file, called a work file, the ALU for 32-bit operation with barrel shifter and field extractor at its entrance, address and data registers for memory interface, and tag circuits are connected to each other by internal buses. There are three such buses; two are source data busses and one is a destination bus. Each is 40-bits wide; 8 bits are for tag transfer and 32 bits are for data transfer. These internal buses also connect other units that are shown in Fig.6. The work file (which has many addressing modes), pairs of memory interface registers, and tag operation circuits are special hardware for KL0 execution.

4.3.2 Treatment of tags

Tag processing (which doesn't contain branch and dispatch using tags) rarely appears in usual microprograms with the exception of tag replacement and tag comparison.

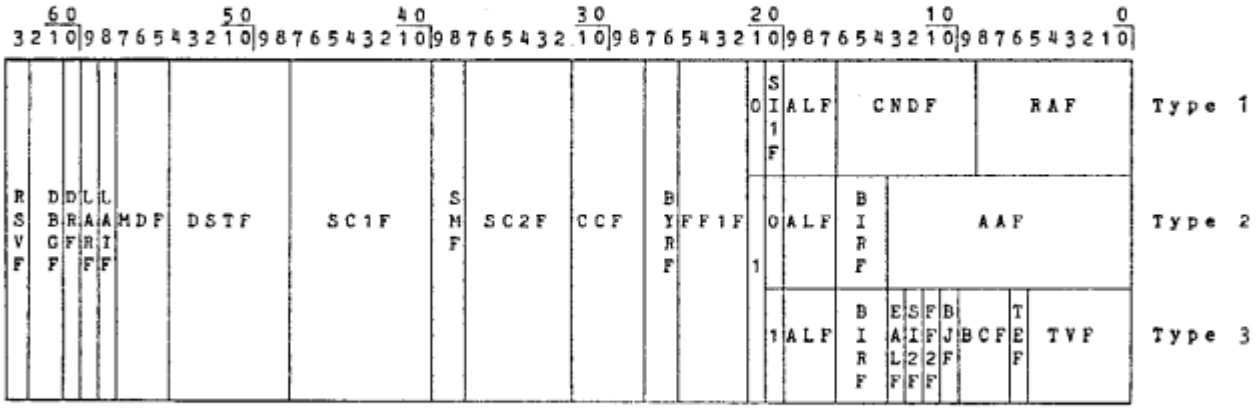


Fig.7 Micro Instruction Format

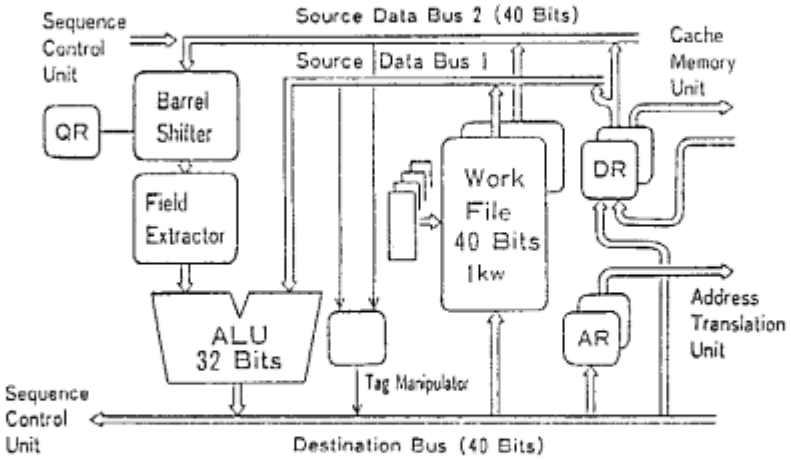


Fig.8 Configuration of the Data Processing Unit

When data processing or data transfer is performed, tag data from one source bus is transferred directly to the destination bus or completely replaced by immediate tag data and transferred. Tags on two source buses are compared in parallel with the data processing and an equality flag is set. This can be tested in a conditional branch instruction. Only the garbage collection microprogram requires tag processing. Bit operation of the GC tag is performed by the same ALU for the data processing. The tag usage in the microprogram sequence control is described in section 4.4. Only the data registers of the memory interface and the work file have tags.

4.3.3 Barrel shifter and field extractor

The barrel shifter can perform up to 32 bits of rotating shift. It can also perform left/right shift of 1 bit combined with the Q register, which is used for multiplication and division as specified by the ALU control field.

The field extractor is a masking circuit that has three different types of masking operations, namely, the low most 5 bit through, the lowest byte through and the lower double byte through operations. These masking operations are often used for extracting operands of built-in predicates, string data, and packed information of machine instructions in combination with the barrel shifter.

4.3.4 ALU and a swap circuit

The ALU is constructed from commercially available ALU LSIs. The ALU control field of the micro instruction has an encoded format that controls only the required functions of the ALU. Arithmetic operations include addition, subtraction, those with carry or borrow, multiplication and division combined with the shift and Q-register operations, and the 24-bit operations for inner area address calculation mentioned in section 2.5. Some flags, such as carry, overflow, and zero, are set for use in conditional branches when flag setting is enabled by micro instruction FF1F field. Logical operations include THROUGH, AND, OR, EXCLUSIVE-OR, AND with SWAP, and OR with SWAP. 'With SWAP' means that bytes are exchanged between byte 0 and 3, and between 1 and 2. The swap circuit is positioned at the exit of the ALU. It is used for re-directing numerical byte data and byte string data; these have the opposite byte order. This circuit is also used to re-direct bytes in the I/O bus access.

4.3.5 Address registers and data registers

There are pairs of memory interface address registers and data registers called PLAR, CLAR, PDR and CDR. LAR means logical address register; P and C mean parent and current of predicate call respectively. When unification is performed, machine instructions and data of both the parent clause (caller) and the current clause (callee) must be fetched from memory. Registers prefixed P and C are used for memory access for the parent clause and the current clause respectively. Tags of PDR and CDR are used for tag dispatch and the least-significant 5 bits of PDR and CDR are used for addressing the work file, as described in later sections. PLAR and CLAR are automatically incremented when contiguous data is being read or written.

4.3.6 Work File

The work file (WF) is a multi-purpose register file most frequently used in the data processing unit. The work file has a 40-bit x 1K-word capacity and has many addressing modes. The WF can be read from and written to different arbitrary addresses in a single micro cycle. That is, data read from the WF is sent to the ALU and the result is rewritten to different WF addresses in one micro cycle. The first 16 words of the WF are designed as dual-port registers for use as general registers. Fig.9 shows the following WF addressing modes.

(a) Direct addressing

The first and last 64 words of the WF constitute an area directly addressable by micro instructions. The first 16 words are used as general registers and the subsequent 48 words are mainly used as logical registers containing information of the current KL0 execution environment. The last 64 words are called the constant area because

the mask patterns and constants used by the firmware interpreter are stored there.

(b) Indirect addressing

WFAR1 and WFAR2 are address registers of WF. The WF can be indirectly accessed by any address using these registers. These registers have auto-increment and auto-decrement, and boundary detection functions. The latter means that flags are set when the contents of the register points to the 32-word or 256-word boundary of WF. These functions enable a part of the WF to be used as a stack area. In practice, they are used to access the local frame buffer and trail buffer, described later.

(c) Indirect addressing using PDR and CDR

In this addressing mode, a WF address is generated by concatenating the content of WFBR and the least-significant 5 bits of PDR or CDR (whichever is specified by the DRF field). This is used to access a local variable cell on the local frame buffer (LFB). WFBR points to the base of LFB, and PDR or CDR holds the cell number of a local variable that is a part of the machine instruction code fetched from memory.

(d) Direct addressing using a base register

In this addressing mode, a WF address is generated by concatenating the contents of WFCBR and a 5-bit direct address specified by the micro instruction. WFCBR is used to point to the base of the extended constant area or work area.

(e) Local frame buffer

The local frame buffer (LFB) is a temporary local frame for a current clause (corresponds to a current frame on a local stack, as shown in Fig.3) created in the WF, not on the local stack in main memory. In the unification using LFB, the local variables of the parent clause that are required for unification are first copied to LFB, then unified with the arguments of the clause head of the current clause. The LFB then temporarily becomes a current local frame. When the execution proceeds to the first body goal of the current clause, if it is a user-defined clause, LFB is pushed onto the local stack and a new temporary local frame for the new current clause is created in the WF. However, if the first body goal is a built-in predicate and the following body goals are also built-in predicates, LFB continues to be used as the current local frame, and is not pushed onto the stack until a user-defined predicate appears. When a user-defined predicate is called, if it is the last body goal and it has no alternative clause, LFB is over-written to new local variables used in the new current clause instead of being pushed onto the stack. This means that the last body goal that has no alternative clause is not called, but invoked through jumping. This corresponds to tail recursive optimization [Warren 80]. This method of using the LFB often leaves local variable cells in the WF and decreases stack access in the main memory.

LFB has fixed size of 32 words. Physically, two LFBs are used alternately by firmware control. In unification, the information to be pushed onto the trail stack is also temporarily stored in a WF area, called the trail buffer.

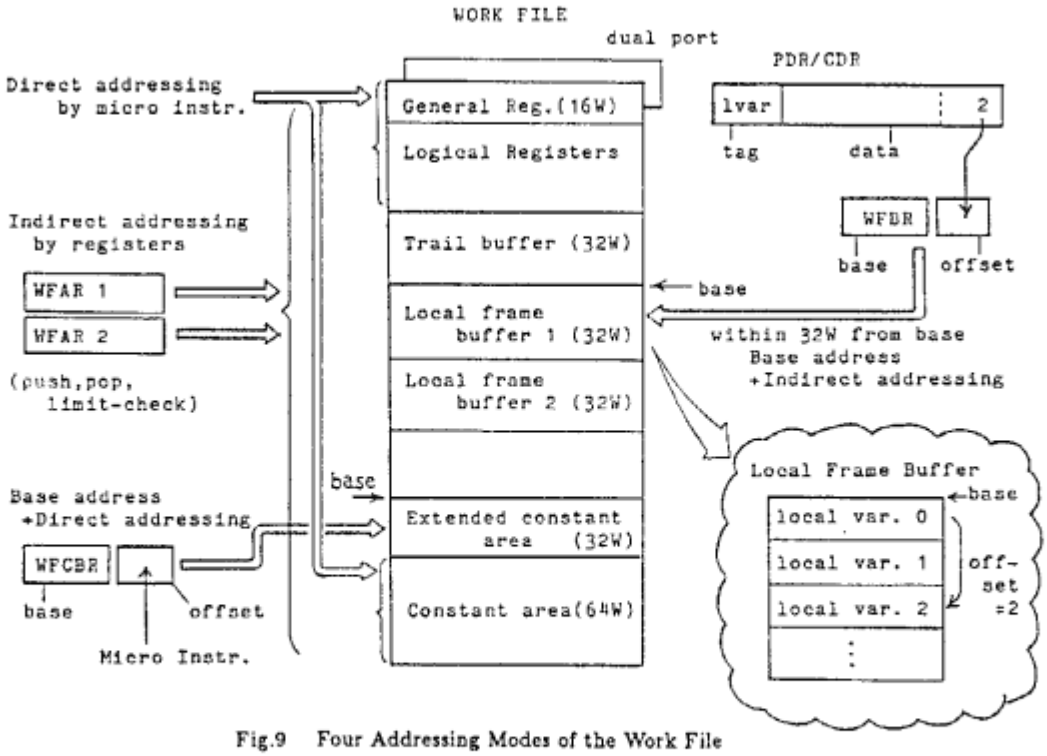


Fig.9 Four Addressing Modes of the Work File

4.3.7 WCS as a register file

The writable control storage (WCS) is used for fetching micro instruction in the latter half of the micro instruction cycle. WCS is designed to be accessible from the internal bus in the first half of the micro cycle. This enables read and write access to WCS under micro program control. Using this function, the last 1K-word area of WCS is assigned as a save area for the process control block. This increases the speed of process switching.

4.4 Sequence control unit

4.4.1 Configuration of the sequence control unit

Fig.10 shows the configuration of the sequence control unit. A micro program control system is used in PSI. A 64-bit x 16K-word WCS is implemented. The first half of the micro instruction cycle generates the address of the next micro instruction, and the second half fetches it. There are several address generation methods, such as absolute branch, relative branch, continuation, OP code dispatch, tag dispatch, multi-way branch using operand tags, conditional branch using a relative address, micro subroutine call, subroutine return, and indirect branch through a jump register. The specialized features of this machine are tag dispatch, multi-way branch and variations on the branch conditions.

4.4.2 OP code dispatch

The instruction code for a built-in predicate contains an operation code (OP code), as shown in Fig.2. This instruction code is transferred to the instruction register (IR) and the OP code is extracted and fed to the dispatch memory. The dispatch memory translates the OP code into the start address of the firmware subroutine corresponding to the operation in a half micro cycle, and this is used for fetching the next micro instruction. The dispatch memory has 256 x 14-bit entries that can be used for up to 256 built-in predicates.

4.4.3 Tag dispatch

The operation for testing the tag is frequently required in the firmware interpreter. The tag dispatch circuit is introduced to increase the speed of tag testing and branch address generation. Address generation is performed in a half micro cycle using the tag of the data read into PDR or CDR from memory. In contrast to OP code dispatch, tag dispatch is a multi-way branch using a base address specified by a micro instruction and an offset generated by the dispatch memory.

There are 64 types of tags in PSI. However, only up to 16 branch targets for multi-way branches are required in practical firmware coding. One of the firmware routine needs a five-way branch, an other needs an eight-way branch, etc. A PDR or CDR tag is translated into a code of three or four bits by the dispatch memory. The code is then concatenated with the base address and used for the next micro instruction address. Twelve translation patterns, from the tag to the code, can be stored in the dispatch memory. Nine are used. The pattern to be used is specified by the micro instruction. As one to several steps of the operation must be executed at the branch destination, the translation pattern is designed to generate multi-way branch addresses in one-, two-, four- or eight-word intervals, according to the request.

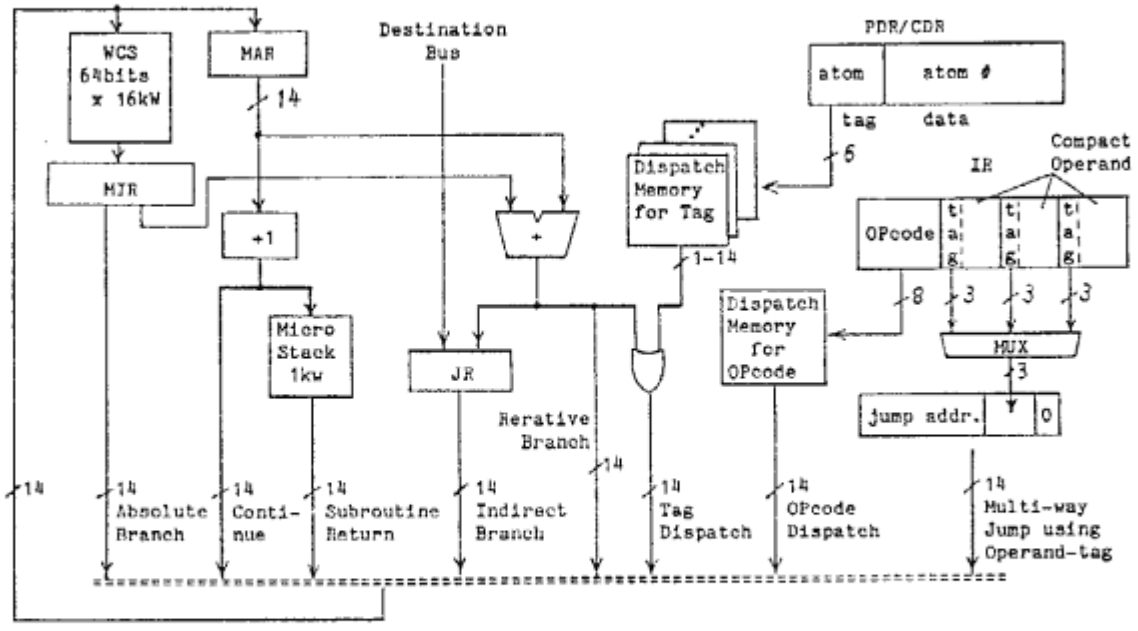


Fig.10 Configuration of the Sequence Control Unit

The dispatch memory consists of 14 bits x 1K words of RAM and is shared for OP code dispatch and tag dispatch.

4.4.4 Multi-way branch using operand tag

The built-in predicate takes up to three compact operands, as shown in Fig.2. Each operand has three bits of compact tag. The instruction register extracts these tags. They are then shifted to the left by one bit and concatenated with the base address of the multi-way jump. Thus an 8-way branch with a two-word address interval is achieved.

4.4.5 Conditional branch

The wide variety of branch conditions is one of the special features of the system. True and the false branches can be specified for each of 64 branch conditions. A conditional branch testing the equality between a register tag (in PDR, CDR, or WF) and an immediate tag data is also available. The major flags used for branch conditions are listed below.

(a) 10 types of ALU flags
(b) Universal flags that can be set and reset independently through FF1F of micro instructions.
(c) Each bit of a register tag of PDR, CDR, or WF
(d) Interrupt request flags
(e) A flag indicating that jump register is equal to zero
(f) I/O bus condition flags

Among these conditions, (b) is frequently used for the interface between firmware modules. These 'switches', which can be easily set and reset, are a valuable asset for a system with so many firmware modules.

4.4.6 Subroutine call and return

Microprogram subroutine call and return are available. The return address is automatically pushed onto and popped from the micro address stack (MSTK). MSTK can be also read from, and written to by the internal bus, so it can be saved and restored when process switching occurs. The MSTK is 1K-word deep but it uses less than 16 words in current coding.

4.4.7 Indirect branch using jump register

The branch address can be set to the jump register (JR) in two ways: from the destination bus, and from the calculated result of the relative address specified by RAF of the micro instruction. An indirect JR branch is available in type 1 and type 3 micro instructions.

JR is also used as a loop counter. Decrementing is specified by the MDF field and zero testing is performed by the function of (e) described in section 4.4.5.

4.5 Memory module

4.5.1 Configuration of the memory module

Fig.11 shows the configuration of the memory module. It contains the cache unit, address translation unit, main memory, and cache control unit. All memory access is performed using the cache order given in the micro instruction. Only when the cache misses an actual memory access is initiated by the cache control unit. The cache memory is accessed through a logical address. The logical-to-physical address translation is performed in parallel with the cache access and the result of the translation is used only when the cache misses. The cache control unit has an independent sequence controller and controls the address translation unit, memory access timing, and memory refresh. Once the cache order is executed, the cache unit works independently from the main sequence control unit. The CPU works in parallel with the cache until completion of the cache order.

4.5.2 Cache unit

The cache memory has a 40-bit x 8K-word capacity, it is constructed from two sets of 4K words. The access time is equivalent to one micro cycle for hit and four micro cycles for miss-hit. The set-associative method is used for cache management and the LRU method is used for the replacement algorithm. The block size is four words and the contents of a block are replaced when the cache misses.

The write-swap method is used in write operations in which write data is only written to the cache instead of to main memory when a write order is executed. When the cache misses, the old data in a cache block is actually written back to main memory. Although the method necessitates writing back old data and reading in required data when the cache misses, it enhances performance when data must be frequently pushed onto and popped from stacks, because there is less overhead for write access to the cache. The write-swap cache is easier to design if the memory has no DMA paths with the I/O devices, as in PSI.

4.5.3 Address translation unit

The address translation mechanism is shown in Fig.5. The page map uses a valid bit that is set during page allocation and tested during address translation. This unit has another memory, called page map size memory, which holds the page size allocated to each area.

4.5.4 Main memory

The word length of the main memory is 40 bits. Up to 16M words of main memory can be installed in PSI. There is an error detection and correction circuit in the cache unit, that can correct single-bit errors and detect double-bit errors. A four-word block transfer is used to transfer data between the main memory and the cache memory. This is performed using the nibble-mode of a dynamic RAM to increase the transfer speed.

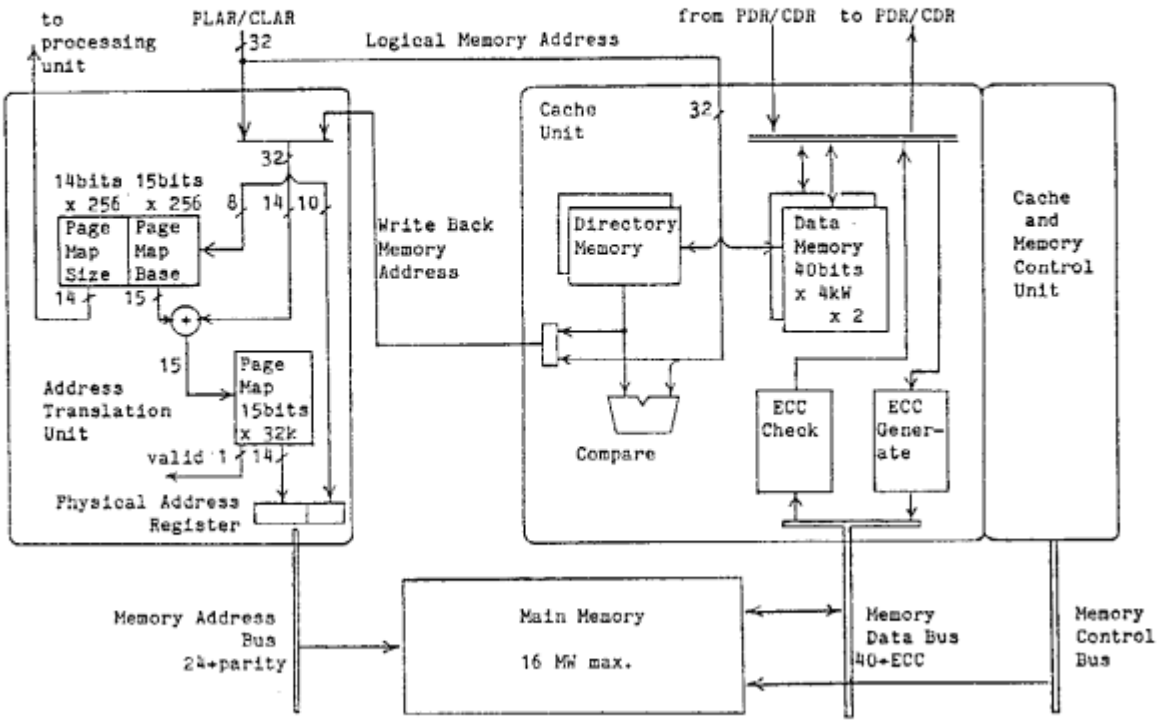


Fig.11 Configuration of the Memory Module

4.6 Hardware implementation

High-speed Schottkey TTL ICs and highly-integrated MOS RAM chips are mainly used in the hardware implementation, because of their commercial availability and small size. The machine cycle time is 200 nano seconds. The CPU is constructed from 12 printed circuit boards, each of which contains about 160 ICs; the main memory is constructed from 16 boards of the same size when 16M words are installed. The CPU boards, the memory boards, 10 or more controller boards for I/O devices, Winchester hard disk drives, and floppy disk drives are all installed in one cabinet.

5 FIRMWARE DEVELOPEMENT

The firmware contains three groups of micro programs, such as the interpreter kernel, built-in predicates, and OS supports such as interrupt handling. Total code size of the firmware is approximately 12K steps.

The interpreter kernel contains routines for basic execution control and unification; both have about same code size, that is, 1.6K steps in total. There are approximately 160 built-in predicates, each of which have from 50 to 100 steps of code; the total code for the built-in predicates is 9K steps. The OS support micro program contains routines for interrupt handling, process switching and memory management, for which the total code size is 1.5K steps.

6 CONCLUSION

In this paper, we presented the machine architecture and hardware design of the personal sequential inference machine, PSI. We also described the system configuration and the basic hardware design philosophy: to design basically simple but partially specialized hardware using microprogram techniques to enhance the efficiency of interpretive execution of high-level machine instructions. We also described the detailed specifications of the hardware components, particularly the register file, which has special functions for tail recursive optimization, and microprogram dispatch facilities using tags.

The experimental hardware development of PSI is already complete, as is testing of the basic firmware modules. An operating system and a programming system for PSI are being developed; tests and debugging are underway on a real machine.

We plan to precisely measure PSI's processing speed using some bench mark programs, and to evaluate the design of hardware components and the firmware interpreter by measuring the dynamic action of the hardware system during program execution. We also plan to compare the architecture of PSI with a machine having determinate instruction execution (not interpretive) and pipelined execution mechanisms (for example, [Tick 84]), and to analyze the strengths and weaknesses of the architecture.

ACKNOWLEDGMENTS

We would like to thank to Mr.Kazuhiro Fuchi, Director of the ICOT Research Center, and Dr.Toshio Yokoi, Chief of the Third Research Laboratory for their continuous encouragement. Thanks are also due to Dr.Takashi Chikayama for his valuable advice, and to other members of ICOT for useful suggestions and discussion. We would also like to extend our thanks to Dr.David Warren for his advice on the tail recursive optimization method.

REFERENCES

[Bowen 81] D.L.Bowen : DEC system-10 PROLOG USER'S MANUAL, Dec.15 1981 Department of Artificial Intelligence, University of Edinburgh

[Boyer 72] R.S.Boyer, and J.S.Moor : The Sharing of Structure in Theorem Proving Programs, Machine Intelligence Vol.1-7, Edinburgh Up (1972)

[Chikayama 83-1] T.Chikayama : Fifth Generation Kernel Language, Proc.of the logic Programming Conference '83, in Tokyo, March 22-24 1983 pp.7.1 1-10

[Chikayama 83-2] T.Chikayama : ESP—Extended Self Contained Prolog—as a Preliminary Kernel Language of Fifth Generation Computers, New Generation Computing Vol.1 No.1 1983, Ohmsha,Ltd.

[Chikayama 84-1] T.Chikayama : KL0 Reference Manual, ICOT Technical Report (to appear)

[Chikayama 84-2] T.Chikayama : ESP Reference Manual, ICOT Technical Report TR-044, Feb. 3 1984

[Cohen 81] J.Cohen : Garbage Collection of Linked Data Structures, ACM Computing Serveys, Vol.13 No.3 1981

[Hattori 83] T.Hattori, and T.Yokoi : Basic Constructs of the SIM Operating System, New Generation Computing Vol.1 No.1 1983, Ohmsha,Ltd.

[Morris 79] F.L.Morris : A Time- and Space-Efficient Garbage Compaction Algorithm, CACM Vol.22 No.10 1979

[Nishikawa 83] H.Nishikawa, M.Yokota, A.Yamamoto, K. Taki, and S.Uchida : The Personal Sequential Inference Machine (PSI)—Its Design and Machine Architecture, Proc.of Logic Programming Workshop, Algrave / PORTUGAL, June 1983, pp.53-73

[Takagi 83] S.Takagi, T.Chikayama, M.Yokota, and T. Hattori : Introducing Extended Control Structures into Prolog, Proc.of 26th Inter- Domestic Conference of IPSJ, March 1983, No.4D-11, in Japanese

[Tick 84] E.Tick, and D.H.D.Warren : Towards a Pipelined PROLOG Processor, Proc.of the International Symposium on Logic Programming, in USA, Feb. 6-9 1984, pp.29-40

[Uchida 83] S.Uchida, 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, Ohmsha,Ltd.

[Warren 77] D.H.D.Warren : Implementing Prolog---compiling predicate logic programs, Vol.1,2, D.A.I Research Report No.39,40, Univ. of Edinburgh, 1977

[Warren 80] D.H.D.Warren : An Improved Prolog Implementation which optimizes Tail Recursion, Proc.of the Logic Programming Workshop, in Hungary, July 1980

[Yokota 83] M.Yokota, A.Yamamoto, K.Taki, H.Nishikawa, and S.Uchida : The Design and Implementation of a Personal Sequential Inference Machine : PSI, New Generation Computing, Vol.1 No.2 1983, Ohmsha,Ltd.

[Yokota 84] M.Yokota, A.Yamamoto, K.Taki, H.Nishikawa, S.Uchida, K.Nakajima, and M.Mitsui : A Microprogram Interpreter of the Personal Sequential Inference Machine, Proc.of International Conference on FGCS '84, Nov. 6-9 1984, in Tokyo