ICOT Technical Report: TR-265

TR-265

# Hardware Architecture of Sequential Inference Machine : PSI-II

by
H. Nakashima (Mitsubishi), and K. Nakajima

June, 1987

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 ARCHITECTURE OF THE SEQUENTIAL INFERENCE MACHINE : PSI-II

Hiroshi Nakashima

Mitsubishi Electric Corporation

Katsuto Nakajima

Institute for New Generation Computer Technology

## Abstract

The *PSI-II*, a special purpose machine for logic programming language, has been designed as an enhanced version of the Personal Sequential Inference Machine, the PSI-I[1]. Moreover, the PSI-II is also an element processor of a parallel inference machine called *Multi-PSI*, which executes a parallel logic programming language *KL1*. The PSI-II has compact hardware in LSI chips, which reduced the hardware to a quarter of the size of the PSI-I. The PSI-II also has very high performance, which is 400 KLIPS in deterministic "*append*", and will be three times as fast as the PSI-I in most application programs. The performance improvement greatly owes to the modification of the execution mechanism, and specialized hardware components for multiple stack management and tagged data manipulation. This paper describes its architecture and hardware mechanisms for logic program execution.

## 1 Introduction

This paper describes the architectural design of a logic programming language-oriented personal machine, the *PSI-II*.

The PSI-II belongs to a family of sequential inference machines designed as a software tool in the Japanese fifth generation computer systems project. The PSI-I [Taki 84] was the first machine of the family and has the following unique features:

- High level machine language :
  A microprogrammed interpreter executes a Prolog-like logic programming language, *KL0* [Chikayama 83].
- Tagged architecture :
  Each register or memory word has 40 bits, six bits of which are dedicated for data type tags and two bits for garbage collection. Special hardware components are provided to speed up the manipulation of the tag bits.
- Multi-stack-oriented memory architecture :
  Each stack can grow or shrink freely without any collision nor copying as long as the total of their sizes does not exceed the total memory size.
- Large main memory :
  The main memory can be extended to 16 Mwords, enabling memory-consuming AI-like programs to be programmed without too much memory saving efforts.
- Large writable control store :
  16 Kwords of 64-bit writable control store allows efficient micro-programmed execution of Prolog-like programs.
- Flexible man-machine interface :
  I/O devices such as a bit-mapped display and a mouse provide a comfortable highly interactive programming environment.

The PSI-II is a new model of the family which functionally inherits the above features of the PSI-I, but is greatly modified in both the architecture and the hardware design to improve the execution speed and to reduce the amount of the required hardware. Moreover, the hardware is designed to have enough flexibility to construct multi-processor system, *Multi-PSI*.

This paper is organized as follows: First, section 2 describes the objectives of the development of the PSI II machine; section 3 describes the machine architecture; section 4 describes detailed hardware design; section 5 evaluates the performance; and section 6 gives the conclusion.

## 2 Objectives

There are three targets for the hardware design of the PSI-II. The first target is to reduce the hardware amount, the second is to improve the performance and the third is to construct a multiprocessor system.

### 2.1 Reduction of Hardware Amount

The PSI-I was designed as a personal workstation for logic programming, but its size is rather larger than the size associated with the word "personal". In fact, the CPU

[1] In this paper, we call the conventional PSI as PSI-I to distinguish from the PSI-II explicitly.

of the PSI-I is constructed from about 2000 conventional TTL MSI and MOS RAM IC chips, on twelve printed circuit boards.

The major strategy to reduce the hardware amount is to pack hardware logic into LSIs. There are four standard LSIs and nine custom LSIs. The standard LSIs are ALU and multi-port register files. The custom LSIs are sliced from the CMOS gate-array master on which 24,000 transistor pairs are mounted. The drawing precision is 2 microns and the typical propagation delay is 2 ns per gate.

The LSIs and highly-integrated MOS RAM chips have made it possible to pack the CPU logic into three printed circuit boards, each of which is about 310 mm by 290 mm.

### 2.2 Performance Improvement

The PSI-I executes stack oriented machine instructions represented in table-like format [Yokota 84]. The machine instruction is syntactically similar to the Prolog source code. For example, a goal

```
..., g(X,a,1), ...
```

is translated to the sequence of the address of the goal g, the identifier of the variable X, the identifier of the atom a, and the integer 1. This approach simplifies the design of the compiler, but requires complicated microprogrammed interpreter.

Another approach to execute logic programs was proposed in [Warren 83]. In this approach, programs are compiled into the sequence of register oriented instructions for an abstract machine (which is often called Warren Abstract Machine : *WAM*). The goal g, shown above, is translated to the sequence of WAM instructions as follows:

```
put_value      Yn,A1
put_constant   'a',A2
put_constant   1,A3
call           g/3
```

Since those instructions are similar to the "Move" and "Branch-and-link" instructions of conventional machines, it is fairly easy to execute them at a high speed.

The compiler of WAM can often reduce instructions by sophisticated register allocation technique. In the example above, if the compiler finds that the value of X remains in register Xj when g is called, put_value Yn,A1 is replaced by put_value Xj,A1 which transfers the content of register Xj to register A1 and is faster than the original instruction. Moreover, if the value of X remains in the register A1 itself, put_value A1,A1 can be omitted.

To compare the PSI-I and WAM approaches, a WAM-like microprogrammed interpreter for KL0 was imple -mented in the PSI-I and its performance is evaluated [Nakajima 85]. The Result was about twice as fast as that of the original KL0 interpreter for both simple benchmark programs and rather complicated application programs. From this result, it was decided to adopt WAM approach for the execution mechanism on the PSI-II.

Hardware facilities, such as instruction pre-fetch registers and the operand field extractor are also introduced to support the execution of WAM-like instructions. As will be described in section 5, a very high performance is achieved, three to ten times faster than the PSI-I.

### 2.3 Multi-Processor System

The PSI-II is not only a logic programming workstation, but also an element processor of a parallel inference machine called the *Multi-PSI* [Taki 86]. The Multi-PSI is constructed from up to 64 CPUs of the PSI-II connected by a two-dimensional mesh network. Each processor element has 40 bits by 16 Mwords (local) main memory and a network communication controller. Since micro-operation codes and an internal data bus port are reserved for the network communication controller, the controller can be easily attached to the PSI-II CPU.

The Multi-PSI executes a parallel logic programming language, *KL1*, which is based on *GHC* [Ueda 86]. KL1 is enables the parallel execution of AND-connected goals synchronizing and communicating through shared logical variables. Another machine instruction set which has parallelism is required for KL1 execution. The PSI-II CPU is designed to have enough flexibility, allowing implementation of various abstract machine instruction sets.

## 3 Architecture

This section describes the architecture of the PSI-II as a KL0 machine.

### 3.1 Implementing KL0

The execution mechanism of the PSI-II is based on that of WAM, but is extended in order to implement KL0. The main extended features are as follows.

#### (1) Built-in Predicates

KL0 has various built-in predicates, such as those for type checking, structure manipulation, arithmetic and logical operation, comparison, execution control, and operating system support. Since they are frequently called in practical programs, their execution speed greatly affects the system performance.

Each built-in predicate is implemented as a machine instruction and is directly executed by microprogram. It requires no environments and creates no choice points, unlike programmer-defined ordinary predicates. Thus, from the point of view of the variable classification, the invocation of a built-in predicate can be treated as part of the following ordinary goal. That is, a variable that occurred only in a sequence of built-in goals and the ordinary goal following it can be classified into temporary variables.

There are two ways to pass arguments to built-in predicates. Input arguments, such as the addend and the augend of add(X,Y,Z), are passed by *put* instructions before the predicate invocation. Output arguments, such as the sum of the add(X,Y,Z), are passed by *get* instructions following the invocation.

The arguments are passed through *any* argument registers, because the machine instructions for built-in predicates have operands, each of which designates an argument register. This mechanism greatly reduces the restriction of the optimal register allocation by the compiler. For example, the clause

```
nth_element(I,[X|L],E):-
                        I1 is I-1,
                        nth_element(I1,L,E).
```

is compiled to;

```
get_list        A2
unify_void      1
unify_variable  A2
put_constant    1,X4
subtract        A1,X4,A1
execute         nth_element/3
```

## (2) Cut

The cut operation is frequently used in practical programs to eliminate redundant alternatives in order to improve execution speed and save memory space. Especially, *neck* cut operations, which precede all non-built-in goals, are the large part of them.

The clause indexing mechanism often makes a neck cut redundant. For example, a version of the deterministic "*append*" is written as;

```
append([X|L1],L2,[X|L3]):-
                        !, append(L1,L2,L3).
append([],L,L).
```

When the cut of the first clause is executed, the alternative is already eliminated by the switch_on_term instruction. Many programmers, however, will not delete the cut mark, because some of them don't know the clause indexing mechanism and the others want to make sure the deterministic execution. The compiler also can not delete it because the first argument may be an unbound variable (KL0 doesn't have the *mode declaration*).

For the fast execution of the non-operating neck cut, we introduced the flag called DET, which indicates whether the current clause has alternatives or not. The DET flag is turned on by the instructions call , execute, trust(_me_else) and the cut operations, and is turned off by try(_me_else) and retry(_me_else). The neck cut is implemented as a special instruction called cut_me to distinguish it from cut marks in other parts. The cut_me instruction examines the DET flag and quickly finishes the execution if the flag indicates that the clause is deterministic. The execution time of the redundant neck cut, in fact, is minimized to time taken by single microprogram step. The effective neck cut is also faster than others, because the choice point to be discarded is found easily, that is, it is the last choice point.

Furthermore, it is planned to optimize the execution of the set of clauses, all of which have neck cut. (Strictly speaking, the last clause of the set may not have the neck cut). The optimization stands on the fact that the choice point of the clauses should not be nested, that is, it should be discarded before any goal invocations. This characteristic of the *simple* choice point greatly reduces the amount of the information to be saved. For example, argument registers don't need to be saved if the compiler generates codes to avoid their destruction until the neck cut occurs. In fact, it is necessary to save only three pointers which refer to the alternative clause and the backtrack points of the global stack and the trail stack. The reduction of the information must accelerate the shallow backtracking.

A selective trailing technique also accelerates the neck cut operation. In the execution of those clauses, the binding of the variables which are not beyond the last non-simple choice point is trailed to the special stack instead of the trail stack, until the neck cut is encountered. The neck cut simply discards the contents of the special stack without complicated selective discarding of the entries on the trail stack. The special stack can be placed on the top of the local stack, because the local stack cannot grow before the cut.

## (3) Dynamic Predicate Call

KL0 has special predicate call mechanisms. One of them is called bind_hook, which is the same as the *freeze* function of Prolog II. Another is called exception_hook, which is very convenient for the exceptional case handling in practical programs. Each built-in predicate requires restricted conditions on its arguments, such as data types and the range of numeric values. Each such condition can be associated with an *exception_hook* predicate which is called at its violation.

Those dynamic call mechanisms bring a troublesome problem to the optimizing compilation. That is, predicates may be called by the binding of hooked variables or the violation of the condition on the argument of built-in predicates, *before* the first ordinary goal in the source program is called. This causes the ambiguity of the words *first* goal, on which the variable classification and the neck cut optimization greatly depend.

We solved this problem by introducing a special environment, which is created when a predicate is called dynamically. The special environment is similar to ordinary one except that it contains the contents of all argument registers instead of permanent variables. An ordinary choice point is also created if the current clause is executed in the *simple* choice point mode. The contents are loaded to the registers and the environment is discarded when the predicate terminates successfully. This mechanism assures that the execution of the clause can be continued as if no predicates were called.

(4) ESP support

*ESP* is the system description language for *SIMPOS*, the programming and operating system of the PSI-I and PSI-II [Chikayama 84]. ESP is not only a logic programming language but also an object-oriented language. An object is represented by a set of *methods* and *slots* in ESP.

Calls of the same method may have different semantics if the values of the first argument, which is an ESP object, are different. The slots are time-dependent state variables, and thier values are examined and altered using the built-in method `get_slot` and `set_slot`. The first arguments of them also represent an object containing the slot to be manipulated.

Since the value of the first argument generally cannot be determined at the compilation time, the address of the slot and the codes for the method are determined at the execution time. To improve the execution speed, there is a set of instructions and built-in predicates to accelerate the method call and the slot manipulation. They require arguments which represent the object and the name of the method or the slot. The address is obtained by looking up a hash table connected to the object, using the name as a key.

## 3.2 Data Representation

Data is represented by a 40-bit word separated into 8-bit tag field and 32-bit value field, as shown in Figure 1. The top two bits of the tag field are used as mark bits for the garbage collection. The lowest six bits represent data type, such as:

- unbound variable
- reference (to variable)
- atom
- integer
- floating point number
- list
- stack vector
- heap vector
- string

A Prolog compound term is represented by a stack vector with the first element being the principal functor.

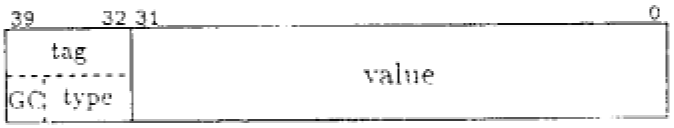

Figure 1: Data Representation

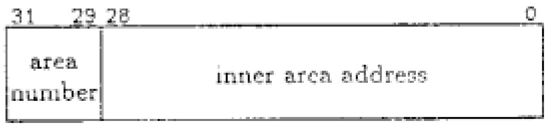

Figure 2: Logical Address Representation

An element of the heap vector can be altered by destructive assignment (not by unification), and the result is not lost on backtracking. A string represents a packed 8-bit ASCII character string or 16-bit Kanji character string.

The value field for a constant contains its own value, that is, an identifier for an atom, or a numeric value for a number. For other types, the value field contains the 32-bit logical address of a variable or a data structure.

## 3.3 Address Space

The PSI-II can execute multiple processes concurrently. Each process has 4 Gwords of logical address space represented by a 32-bit address shown in Figure 2. The logical address space is divided into eight distinct sub-spaces called *area*, which are identified by the top 3-bit field of the logical address. Three areas correspond to the stacks, a local stack, a global stack, and a trail stack, which are local to each process. Two areas, heap and system area, are shared by all processes. The remaining three areas are reserved for future extension.

The stacks is similar to those of WAM. The heap contains codes, ESP objects and data objects whose values can be altered by the assignment operation. The system area contains system control information, such as process control blocks, an interrupt vector and memory management tables.

## 3.4 Instructions

Figure 3 shows the instruction format of the PSI-II. An instruction is represented in one or more 40-bit words. The 2-bit classification field divides the instruction set into four classes. The first class consists of basic Prolog instructions which correspond to unification, predicate call and return, and backtracking control. The second comprises built-in predicates. The third and fourth are reserved for future extension, such as parallel programming language implementation. The 8-bit instruction code field can represent up to 256 instructions for each class.

The 24-bit operand field of the first word of an instruction contains one of the following; three 8-bit operands; one 8-bit operand and one 16-bit operand; or a single 24-bit operand. An 8-bit operand represents an argument register number, an offset of a local stack frame, or a (signed or unsigned) short integer value. A 16-bit operand is used for a relative branch address. A 24-bit operand is used as an atom identifier. The second and consequent words are optional. They usually represent 40-bit (atomic) operands with tag.

The basic Prolog instructions are divided into the groups, *get*, *put*, *unify*, *procedural*, *indexing*, *method* and *cut* instructions. The first five groups are similar to those of WAM. The *method* instructions call ESP methods. The *cut* instructions discard the alternatives of the current predicate and the goals preceding the cut operator.

Built-in predicates are divided into the following groups.

(a) type checking

e.g. `atom(+X), unbound(+X)`

(b) structure manipulation

e.g. `vector_element(+Vect,+Pos,-Elem),`
`new_heap_vector(-Vect,+Len)`

(c) arithmetic and logical operation

e.g. `add(+X,+Y,-Z),`
`shift_left(+X,+Count,-Y)`

(d) comparison

e.g. `equal(+X,+Y), less_than(+X,+Y)`

(e) execution control

e.g. `bind_hook(+X,+Handler)`

(f) operating system support

e.g. `change_process(+Process)`

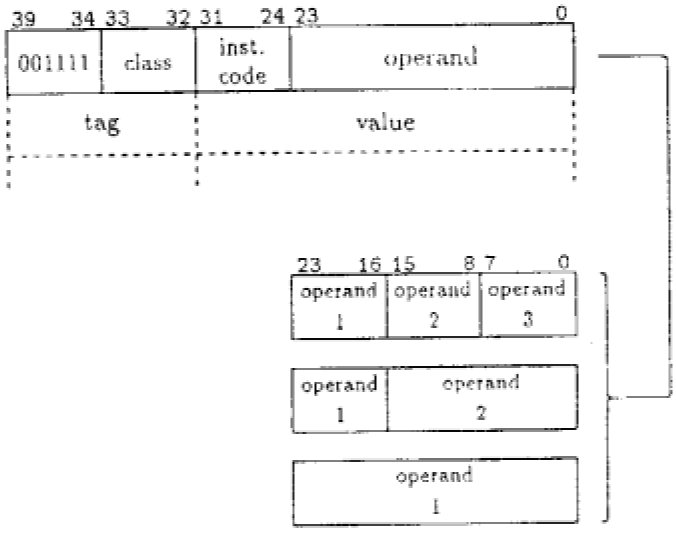

Figure 3: Instruction Format

## 4 Hardware Design

This section describes the hardware design of the PSI-II. Figure 4 shows the system configuration of the PSI-II. The machine cicle of the PSI-II is 166.7 ns (24 MHz), which is 20 % faster than that of PSI-I, 200 ns.

### 4.1 Main Memory

The PSI-II has physical main memory of 40 bits by up to 64 Mwords constructed from 1 Mbit dynamic RAMs. This large scale memory enables implementation of most of the memory consuming applications without virtual storage. For each word, seven ECC bits are added to correct single bit errors and detect double bit errors.

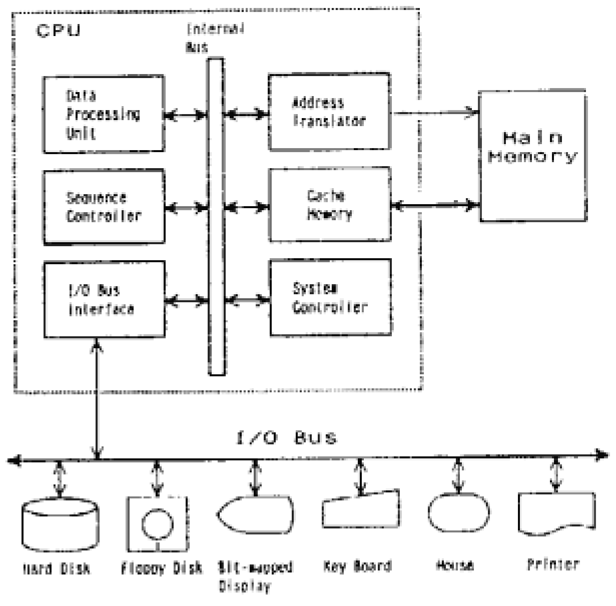

Figure 4: System Configuration

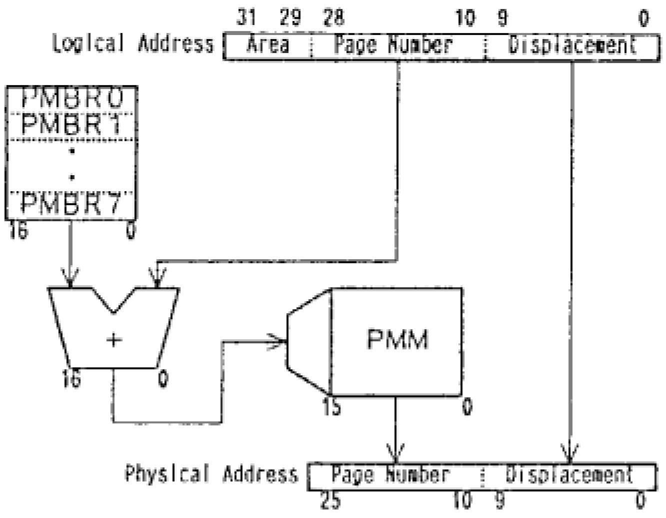

Figure 5: Address Translation

## 4.2 Address Translator

Figure 5 shows the address translation mechanism. A 32-bit logical address is translated into a 26-bit physical address by the two level table look up. The top 3-bit field of the logical address, identifying an area, selects one of the eight *Page Map Base Registers (PMBR)*. Each PMBR contains the base address of a set of entries of the *Page Map Memory (PMM)*. The entries are continuously set in the PMM. Each contains a 16-bit physical memory page number allocated to the area. The middle 19-bit field of the logical address, called the logical page number, is added to the PMBR to obtain a physical page number from the corresponding entry of the PMM. The resulting physical page number is used as the top 16-bit field of the physical memory address. The lowest 10-bit field of the logical address is used as the lowest 10-bit field of the physical memory address directly.

Since stacks are local to each process, the stack areas of different processes must be mapped to different physical memory space. For this mapping mechanism, PMBRs associated to stack areas are stored in the process control block and restored during process switching. PMBRs for shared areas, heap and system area, are common to all processes.

Physical memory pages are managed using the demand page allocation technique. Unallocated physical memory pages are pooled in a free page list. If a stack (or heap) grows up and more physical pages are required, pages are allocated to the stack area from the free page list. The new page allocation causes new PMM entry allocation to set the page numbers of the obtained pages. This page allocation is performed quickly by a microprogrammed memory manager.

If no free pages remain pooled or the desired PMM entry is already occupied by another area, the microprogrammed memory manager raises a trap to the operating system. A memory manager in the operating system tries to release pages allocated to shrunk stacks and/or relocate PMM entries. To avoid frequent relocation of PMM, the memory manager tries to distribute the set of entries as sparse as possible. This distribution technique is encouraged by the fact that PMM has 96 K entries, more than maximum number of physical memory pages, 64 K pages.

The demand page allocation requires that any stack growth should check whether new stack top is within already allocated page. An naive implementation of stack growth brings some overhead. This problem was solved by introducing a sophisticated technique using buffer pages, called *gray page*. Figure 6 shows the concept of the gray page.

A PMM entry contains a flag bit, called the *gray bit*, indicating whether the corresponding page is *gray*. The gray page is allocated as the last page of each stack area. If the gray page is accessed, a new page requirement trap is raised. The trap transfers microprogram control to the microprogrammed memory manager at the end of the instruction which grows a stack in the gray page. The stack growth is completed normally, because the gray page is associated with a real physical memory page.

Different stack areas have different gray pages, because some instructions may grow multiple stacks. The same stack area of different processes, however, can share a single gray page, if it is certain that no process switch is performed leaving the stack top pointer in the gray page. All processes, in fact, share a single physical memory page as a gray page for each stack to minimize redundant memory space. The microprogrammed memory manager gets a free page and copies the contents of the common physical gray page to the obtained page. Since most of the instructions grow a stack less than ten words, valid data in the gray page can be copied quickly. The PMM entry associated with the gray page is re-associated with the obtained page by resetting the gray bit, and a new PMM entry is allocated for the gray page.

## 4.3 Cache Memory

The cache memory has 40 bits by 4 Kwords smaller than 8 Kwords of the PSI-I. The capacity reduction required to reduce the hardware amount will decrease the hit-ratio. However, we have estimated, by simulation, that degradation of the hit ratio is not much; about 2 percent.

The write-swap method is used for write operations. In this method, data is written only to the cache memory when a write order is directed. After a cache miss-hit, the contents of a certain cache memory block are written back to the main memory, if the contents are modified after they were loaded to the cache memory. It was estimated that the write-swap cache memory has about 10

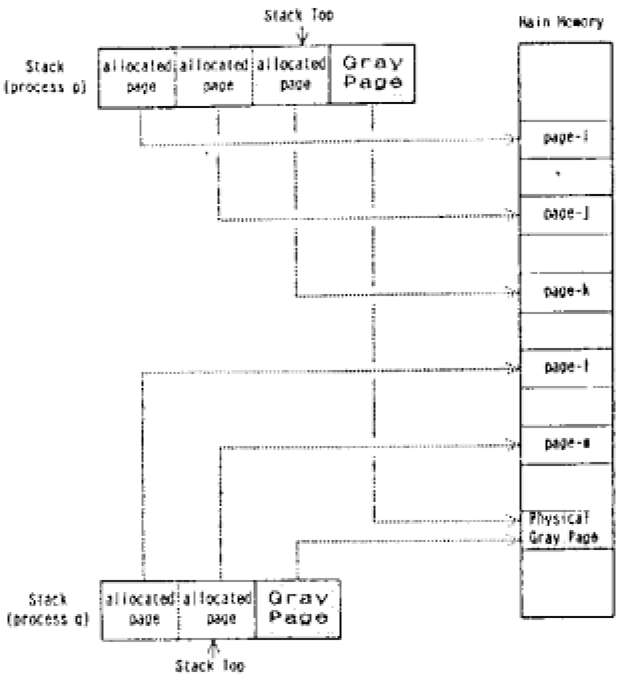

Figure 6: Gray Page

percent better performance than the write-through cache memory.

For growth of stacks, a sophisticated write operation, called *write stack*, is introduced. Usually, when a write operation misses the cache, a memory block is loaded from the main memory to validate other words of the block. For a write operation which pushes a stack, however, validation for the words beyond the stack top is not necessary. The write-stack operation for the first word of a cache memory block does not load a block even if it misses the cache. If the write back operation is not necessary, the miss-hitting write-stack operation is as fast as a hitting write operation.

## 4.4 Data Processing Unit

Figure 7 shows the configuration of the data processing unit. An ALU, a register file and several discrete registers are connected with three 40-bit wide data buses, called *Source-1 bus*, *Source-2 bus* and *Destination bus*.

The ALU performs addition , subtraction, and logical operations including multiple bit shift and bit-field manipulation. An ALU operation is performed to the 32-bit value parts of the Source-1 and Source-2 buses, and the result is output to the 32-bit value part of the Destination bus. The 8-bit tag part of the Source-1 bus is passed to the tag part of the Destination bus. The tag parts of the Source-1 and Source-2 buses are also used to control the microprogram sequence (see the next section).

Various discrete registers are installed for special operations. The main registers are as follows.

- MAR: Memory address register, containing a 32-bit logical address of the main memory.
- IAR: Instruction address register, containing a 32-bit logical address of the machine instruction. It is the physical implementation of the abstract register P of WAM.
- MDR: Memory data register, containing a 40-bit data which is read from or written to the main memory.
- IR: Instruction register, containing the 40-bit machine instruction being executed.
- IBR: Instruction buffer register, containing the 40-bit pre-fetched machine instruction.
- IFR: Instruction fetch register, containing the 40-bit doubly pre-fetched machine instruction, or 40-bit data of the main memory.
- GR: Global top register, which points the top of the global stack.
- SR: Structure pointer, which refers an element of a structure to be unified.
- LC: Loop counter, which is used for microprogrammed DO loops.

Output ports of these registers are connected to the Source-1 bus. The MDR, IR, IBR and IFR also have an output port to the Source-2 bus via a byte field extractor. The byte field extractor, especially for the IR, is used to extract 8-bit, 16-bit, or 24-bit operand field with or without sign bit extension. The MAR, IAR, GR, SR and LC has auto-increment/decrement function without using ALU.

The IR, IBR and IFR are used for pipeline registers to pre-fetch machine instructions. At the beginning of execution of a machine instruction, the IR contains the instruction to be executed, the IBR contains the next instruction, and the IAR refers the instruction after the next one. The last micro-instruction for a machine instruction performs the following operations:

(a) Dispatching to the micro-routine for the next instruction by decoding the instruction code field of the IBR.

(b) Transferring the IBR to the the IR.

(c) Fetching the instruction pointed by the IAR into the IBR, incrementing the IAR after that.

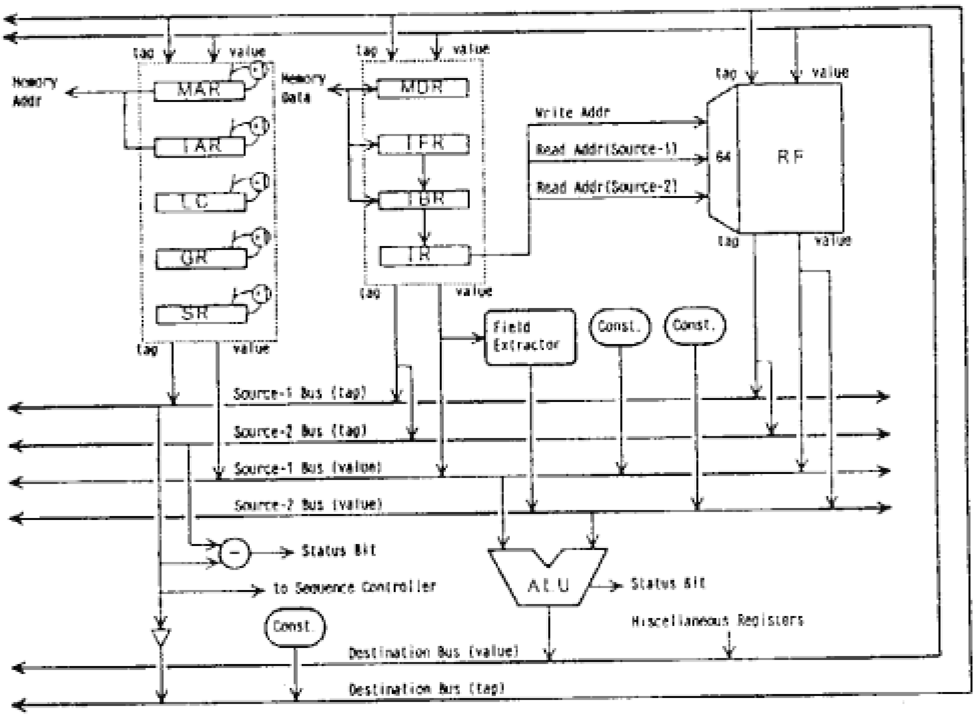

Figure 7: Data Processing Element

These operations can be performed concurrently with other micro-operations, except for the main memory access. If memory needs to be accessed (written) in the last micro-instruction, the IFR is used as a pre-fetch buffer. That is, in the non-last micro-instruction, the instruction pointed by the IAR is pre-fetched into IFR. In the last micro-instruction, the IFR is transferred to the IBR instead of fetching a new instruction. This instruction fetch mechanism is fairly simple, and works well in most cases.

The register file (*RF*) has a capacity of 40 bits by 64 words. The first half of the RF is used for argument/ temporary registers. The entry address for this area is usually supplied from the operand fields of the IR. Another supplier of the address is the *RF address register (RFAR)*, which can be incremented automatically. The RFAR is used for multiple loading or storing of the argument registers with auto-incrementing.

The special purpose register, *RF validity flag register (RVFR)*, is also installed to support the garbage collector. Each bit of the RVFR corresponds to an entry of the first half of the RF, that is, the least significant bit is to the first entry and the most significant bit is to the 32nd entry. Each bit of the RVFR is turned on when the corresponding entry of the RF is modified. All bits of the RVFR are cleared explicitly when the contents of all the argument registers become useless (for example, by backtracking). Thus, each bit of the the RVFR indicating the validity of the corresponding argument register tells the garbage collector that the argument register must be a root of marking.

The rest of the RF is used as the control registers of WAM, except for program pointer, global top register, and structure pointer (which are allocated to the discrete registers). Part of this area also is used for the working space of microprogram. The entry address for this area is usually supplied directly from a micro-instruction.

## 4.5 Sequence Controller

Figure 8 shows the configuration of the sequence controller. The sequence controller generates micro-instruction addresses and fetches micro-instructions from the writable control store (*WCS*), which has a capacity of 53-bit by 16 Kwords.

Most of the microprogram branch operations performed by the sequence controller are two-way. 64 type branch conditions are defined, and they can be used for both positive and negative conditions. Branch conditions of checking data type by the tag, called the *tag conditions*, are one of the special features of the PSI-II.

One of the frequently used tag conditions is the comparison of the tag part of the Source-1 bus with an immediate value in a micro-instruction. This condition is used to examine whether data has a particular type. It is also possible to compare the tag masking lower bits to examine whether data is a member of a group of data types. For example, assuming that *atomic* data consists of atoms and integers, and that the tag representation for them are same except for the least significant bit, atomic

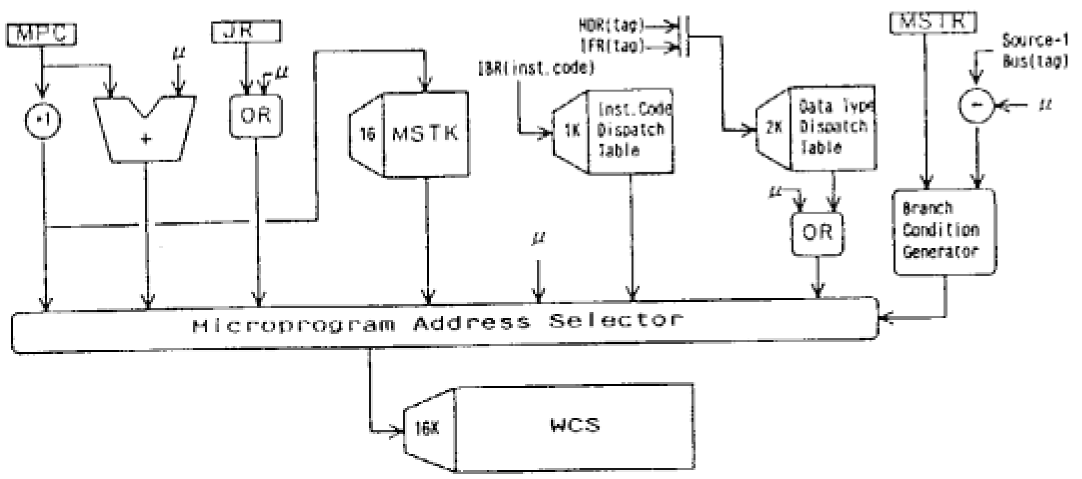
MPC
JR
HDR(tag)
IFR(tag)
IBR(inst. code)
MSTR
Source-1 Bus(tag)
+1
+
OR
16 MSTK
1K Inst.Code Dispatch Table
2K Data Type Dispatch Table
OR
Branch Condition Generator
Microprogram Address Selector
16K WCS

Figure 8: Sequence Controller

data can be checked by a single comparison masking the least significant bit.

Another important tag condition is the comparison result of the tag part of the Source-1 and Source-2 buses, which is set to a bit in the micro-status register (*MSTR*). This condition is used to examine whether two data have the same type. It is also possible to combine the tag equality condition with the value equality condition, which is obtained by subtracting two data.

Branch operations performed by the sequence controller are as follows.

(1) Direct branch with relative or absolute addressing.

(2) Microprogram subroutine call and return using the microprogram address stack (em MSTK), 16 in its depth.

(3) Indirect branch using an indirect jump register (*JR*).

(4) Instruction code dispatching using a 1 K-entry RAM table.

(5) Data type dispatching using a 2 K-entry RAM table.

The RAM table for the data type dispatching is partitioned into 32 blocks. Each block consists of 64 entries each of which corresponds to a data type and contains an offset. On dispatching, the content of the tag part of the MDR or IFR is concatenated with the block number specified in a micro-instruction to generate a table entry address. The content of the entry is ORed with the base address specified in the micro-instruction to obtain the next micro-instruction address. This base-offset allows the same dispatching pattern to be used in different parts of the microprogram. It is also possible that different entries of a block have same offset to perform the same operation for data of several different types.

## 4.6 I/O Devices

Various I/O devices are connected via an I/O bus. Standard devices are a 150 Mbyte hard disk, a 5 inch floppy disk, a Kanji printer, a key board, an optical mouse and a bit-mapped display.

The bit-mapped display has a microprocessor as a controller with a bit plane of up to 8 Mbytes of monochrome or color. The microprocessor has manages the bit plane, transfers a rectangle, tracks mouse movement, and draws graphic objects. These functions contribute to fast display speed under the complicated multi-window environment.

Optional I/O devices are large capacity hard disks, magnetic tape drives, a laser printer, and an Ethernet LAN interface.

## 5 Performance Evaluation

Table 1 shows measured performance of the PSI-II in simple benchmark programs. The first column of the table shows benchmark programs which are provided by the First Prolog Contest [Okuno 85], except for *append30*. They include two lists concatenation (*append30*), naive reverse of 30 elements (*nrev30*), quicksort of 50 elements (*qsort50*), tree traversing of 1000 elements (*trav1000*). 8 queens problem (*8queens*), bidirectional computation calculating factorial of inverse of fibonacci (*fibfact*), and slow reverse of five elements (*srev5*). The executable codes of these programs are generated by a cross compiler on the PSI-I.

The second and the third columns show the performance of the PSI-II represented by execution time (msec) and*KLIPS*. The forth and fifth columns show the execution time of the PSI-I and the WAM emulator on the PSI-I [Nakajima 85]. The sixth and seventh columns show the execution time ratio of the PSI-II to the PSI-I and

| benchmark programs | PSI-II | | PSI-I | WAM emulator on PSI-I | (2)/(1) | (3)/(1) |
|---|---|---|---|---|---|---|
| | (1) | | (2) | (3) | | |
| append30 | 0.075 msec | 400.0 KLIPS | 0.80 msec | 0.29 msec | 10.67 | 3.82 |
| nrev30 | 1.53 | 324.8 | 13.6 | 4.35 | 8.88 | 2.84 |
| qsort50 | 3.86 | 158.0 | 15.2 | 6.73 | 3.94 | 2.56 |
| trav1000 | 17.7 | 120.2 | 51.7 | 29.0 | 2.92 | 1.63 |
| 8queens | 29.6 | 194.2 | 96.9 | 48.1 | 3.27 | 1.62 |
| fibfact | 15.4 | 84.4 | 38.2 | 22.3 | 2.48 | 1.45 |
| srev5 | 6.60 | 129.1 | 24.8 | 10.3 | 3.76 | 1.56 |

Table 1: Performance of PSI-II

WAM emulator.

The table shows that the PSI-II has very high performance, 300 to 400 KLIPS, and is nine to ten times as fast as the PSI-I, in deterministic programs such as *append30* and *nrev30*. These results are mainly due to the clause indexing technique which eliminates alternative clauses before the head unification. In fact, no backtrack points are made in executing these programs.

In other programs, the PSI-II is approximately three times as fast as the PSI-I. These programs make many backtrack points and cause backtracking frequently. Since the operations to save and load information of backtrack points are essentially the same with PSI-I, the performance improvement of those programs is rather small. The neck cut optimization described in 3.1, however, will improve performance of *qsort50* and *8queens*.

The PSI-II is also approximately one and a half times faster than the WAM emulator on the PSI-I. This improvement owes to the newly introduced hardware facilities which accelerate the Prolog instruction execution, such as the gray page, pre-fetch buffer, and the operand extractor.

## 6 Conclusion

A special purpose machine, the PSI-II, was designed for logic programming languages. The hardware is very compact because of high integration CMOS gate-array LSIs. A very high performance of 400 KLIPS is also achieved by improvement of the architecture and hardware design.

The hardware has been manufactured and the microprogram implemented. Software installation is scheduled to completed by the end of the third quarter of the year. Hardware design and manufacturing of the Multi-PSI will also be finished at the end of the third quarter, and multiprocessor system with a microprogrammed interpreter of a parallel programming language and operating system kernel will begin its operation in the first quarter of next year.

## References

[Taki 84] K. Taki, et al., Hardware Design and Implementation of the Personal Sequential Inference Machine, *Proc. of the International Conf. on FGCS '84*, 1984.

[Chikayama 83] T. Chikayama, et al., Fifth Generation Kernel Language, *Proc. of the Logic Programming Conf.*, 1983.

[Yokota 84] M. Yokota, et al., A Microprogrammed Interpreter for the Personal Sequential Machine, *Proc. of International Conf. on FGCS '84*, 1984.

[Warren 83] D. H. D. Warren, An Abstract Prolog Instruction Set, *Technical Note 309, AI Center, SRI International*, 1983.

[Nakajima 85] K. Nakajima, et al., Evaluation of PSI Micro-Interpreter, *Proc. of Compcon Spring 85*, 1985.

[Taki 86] K. Taki, The Parallel Software Research and Development Tool: Multi-PSI System, *Proc of the France-Japan AI and Computer Science Symp. 86, Institute for New Generation Computer Technology*, 1986.

[Ueda 85] K. Ueda, Guarded Horn Clauses, *Proc. of the Logic Programming Conf.*, 1985.

[Chikayama 84] T. Chikayama, Unique Features of ESP, *Proc. of the International Conf. on FGCS '84*, 1984.

[Okuno 85] H. Okuno, The Report of the Third Lisp Contest and the First Prolog Contest, *Report of WGSYM, No. 33-4, IPSJ*, 1985.