ICOT Technical Report: TR-389

TR-389

Macro-call Instruction for the Efficient KL1 Implementation on PIM

by
T. Shinogi, K. Kumon, A. Hattori(Fujitsu),
A. Goto, Y. Kimura and T. Chikayama

June, 1988

©1988, ICOT

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

Macro-call Instruction for the Efficient KL1 Implementation on PIM

Tsuyoshi Shinogi Kouichi Kumon Akira Hattori
Fujitsu Limited*
Atsuhiro Goto Yasunori Kimura Takashi Chikayama
Institute for New Generation Computer Technology†

Abstract

In the FGCS project, a parallel inference machine prototype, *PIM*, is now being developed based on the concurrent logic programming language, KL1. KL1-B, the virtual machine instruction designed for KL1, is extended for parallel execution. Run-time type checking is necessary in many instructions of KL1-B. The action of a KL1-B instruction varies very much depending on its argument type. To enable efficient implementation of KL1-B, RISC-like instructions with macro-call functions are incorporated in the processor element architecture of the *PIM*, so that the processor element can use only the best features of RISCs and high-level instruction set computers (HLICs) with micro-programs.

The *PIM* includes clusters connected by a network. Each cluster consists of eight processor elements, communicating through shared memory over a common bus. The processor element has two instruction streams, external and internal. Most of external and internal instructions are common RISC-like instructions, which can be executed every machine cycle using a four-stage pipeline. KL1 programs are compiled into the object programs consisting of the external instructions stored in shared memory. The external instructions include conditional macro-call instructions. They can determine, depending on tag conditions of their register operands, to proceed the execution of instruction streams or to invoke their macro-bodies at small cost. The macro-body is written in the internal instructions and stored in the local memory of each processor element. The internal instructions can handle both the register operands and the immediate value operands of a macro-call instruction by using indirect registers.

The RISC features in the processor element architecture are effective for unconditionally executed portions both in compiled codes and in macro-bodies. The HLIC features by macro-call instructions are suitable for implementing most instructions of KL1-B, whose actions vary depending on their argument type. As a result, system designers can flexibly implement KL1 on the *PIM* fully exploiting the advantages both of RISCs and HLICs.

1 Introduction

In the Japanese FGCS project, the parallel inference machine systems are being developed based on a logic programming framework at ICOT [5, 3]. The kernel language, KL1, has been designed. The parallel operating system is being written in KL1 as a *self-contained* operating system. A parallel inference machine prototype, *PIM*, optimized to KL1, is now being developed, which is planned to include 128 processor elements.

The principal aim of parallel processing is to increase the execution performance, so that users will be able to solve large application problems. Hierarchical structure is introduced in the *PIM* to

*1015, Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan
†4-28, Mita-1, Minato-ku, Tokyo 108, Japan

connect 128 processor elements. Eight processor elements form a cluster, communicating through shared memory over a common bus. Local coherent cache, designed specifically for KL1 parallel execution, is provided to enable quick shared memory access and efficient communication within a cluster. The clusters are connected with other clusters by a packet switching network of multiple hypercube. The development of a high-performance processor element is, of course, the first step to the target parallel inference machine.

KL1 has the features that make conventional machines unsuitable for efficient execution. Three of these features are: (1) unification is a *polymorphic* operation on, usually, dynamically constructed linked data structures; (2) the execution context, though small, is frequently switched during execution because of synchronizing function in KL1; and (3) single assignment feature demands efficient memory management by incremental garbage collection.

The architecture design started from development of KL1-B [6], a virtual machine instruction set for KL1. Experiments using KL1-B emulators found that: most instructions in KL1-B include run-time data type checks; and the action that follows the run-time type check within a KL1-B instruction varies very much depending on the type, even though the run-time data type check selects only a portion of them. Therefore, it is difficult to implement KL1-B efficiently by expanded compiled code of RISC-like instructions.

Focusing on the macro-call function in the processor element architecture of the *PIM*, this article presents how to exploit the advantages both of RISCs and high-level instruction set computers (HLICs); how to implement the macro-call function in the processor element to realize both features; and how efficient the KL1-B instructions can be executed by the macro-call function.

2 KL1-B: Virtual Machine Instruction for KL1

2.1 Brief introduction to KL1

KL1 is developed based on GHC [11, 12]. A GHC program is a finite set of guarded Horn clauses of the form:

$$H :- G_1, \cdots, G_m | B_1, \cdots, B_n. \ (m \geq 0, n \geq 0)$$

where H, G_i, and B_i are called the *clause head*, *guard goals* and *body goals*. The operator, |, is called a commitment operator. The part of the clause preceding | is called the *passive-part* (or *guard*), and that following it is called the *active-part* (or *body*).

When an input goal, H, is given, reduction of H is tried *in parallel*, and one of the clauses whose head unification and guard goal execution succeed is selected. After that, body goals B_js are executed. This means that goal H is reduced to B_js. If unification requires the instantiation of a variable during passive part execution, this unification is suspended.

KL1 was initially specified as flat GHC, taking efficient implementation into consideration. Flat GHC is a subset of GHC, which allows only built in predicates as guard goals. This restriction makes language implementation more efficient while keeping most of GHC's descriptive power. Starting from flat GHC, KL1 has been extended to be a practical language introducing the features required for the parallel operating system design.

2.2 Basic execution mechanism of KL1

KL1-B [6] is a virtual machine language interfacing parallel inference machine hardware and KL1, just as WAM [13] interfaces Prolog and sequential machines. In other words, KL1-B represents the abstract architecture of the parallel inference machine.

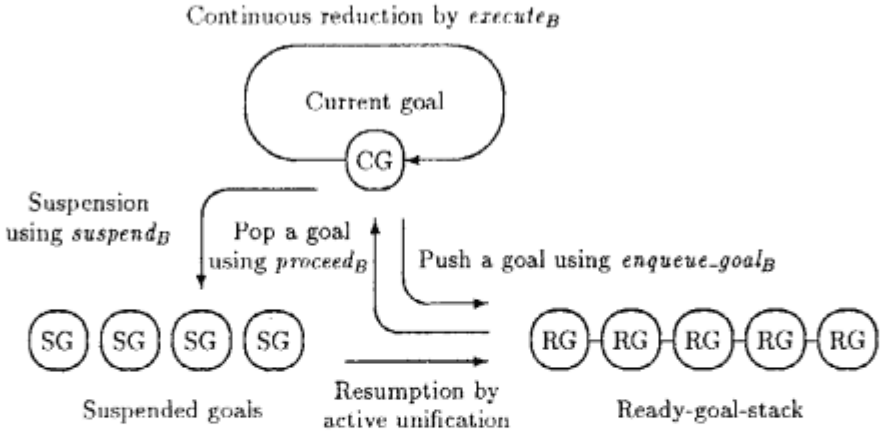

Figure 1: Goal state transition

To build an efficient parallel inference machine, execution on each processor element must be as efficient as possible. Therefore, KL1-B was designed first based on sequential execution [6]. Then, it was extended for parallel execution. The KL1 parallel execution mechanism is summarized in the following subsections[1].

2.2.1 Data and control structures and execution control

Data structures or variables shared among goals are stored in a heap. A data structure called a *goal-record* is used for representing a goal. A goal-record consists of its argument list, a pointer to the compiled code corresponding to its predicate name, and some control information. The argument list includes atomic values or pointers to variables or structure bodies in the heap.

A goal can be a *ready goal* (RG), a *suspended goal* (SG) or a *current goal* (CG), as shown in Figure 1. The *ready* goal-records are linked into a list forming a *ready-goal-stack*. A KL1-B instruction, $proceed_B$ [2], pops up a goal as a current goal, then reduction of the goal is initiated.

2.2.2 Execution of the passive-part

For the current goal, candidate clauses are tested sequentially by head unification and guard execution to choose one clause whose body goals will be executed. If the instantiation of a variable is required during the execution of the passive part, the test for this clause is abandoned and execution proceeds to the next candidate clause. Figure 2 shows a typical KL1-B instruction for passive part execution. $Wait\_constant_B$ corresponds to a passive unification between a current goal argument, *Ai*, and a constant value, *C*. *Label* is a branch address when the unification is suspended or failed. When a unification is suspended, the variable which caused the suspension is saved in a suspension stack.

If no clause is selected, the current goal becomes a suspended goal by a $suspend_B$ instruction. First, variables which cause the suspension are popped up from the suspension stack. Then, the current goal is linked to these variables to realize non-busy waiting synchronization mechanism between KL1 goals.

[1] An explanation of each KL1-B instruction can be found in [2, 6].

[2] In this article, each KL1-B instruction is written with postfix B, e.g. $proceed_B$.

wait_constant$_B$ *C, Ai, Label:*
put the dereference result of *Ai* to *Ai*
check the equality between *Ai* and *C*
if they are equal **then** proceed to the next code
else if *Ai* is uninstantiated **then** push *Ai* to the suspension stack
jump to *Label*

Figure 2: A KL1-B instruction: *wait_constant*$_B$

get_list_value$_B$ *Xj, Ai:*
put the dereference result of *Ai* to *Ai*
if *Ai* is uninstantiated
then if *Ai* is linked by suspended goals **then** resume suspended goals
Ai := *Xj* and proceed to the next code
else if *Ai* is list **then** do general unification between *Xj* and *Ai*
else *Failure*

Figure 3: A KL1-B instruction: *get_list_value*$_B$

2.2.3 Execution of the active-part

If a clause is selected, the body part of that clause is executed. Execution of the body part consists of two kinds of operations, *active unification* and *body goal fork*. Figure 3 shows a typical KL1-B instruction for active unification. *Get_list_value*$_B$ unifies a variable, *Ai*, with a List pointed by *Xj*. Active unification is executed on the spot. Here, suspended goals may be resumed by this active unification, by moving the goal-records linked from the variable to the ready-goal-stack again. (See Figure 1.)

The body goal fork is done by argument preparation instructions, *set_XXX*$_B$, *put_XXX*$_B$, followed by an instruction for linking the goal record to the ready-goal-stack, *enqueue_goal*$_B$. New variable cells or structures may be allocated by these instructions. One body goal can be executed by *execute*$_B$ without pushing it back to the ready-goal-stack. (See Figure 1.) Depth-first scheduling is adopted, here.

2.3 Incremental garbage collection by MRB

KL1 is a concurrent language with no side effects. Destructive memory assignment is, in principle, not allowed. Therefore, naive implementations of KL1 consume memory area very rapidly, so that garbage collection would occur frequently. The locality of memory references is supposed to be very low during garbage collection by widely used schemes, so that cache misses and memory faults would occur often. In sequential Prolog [13], this problem is not very serious because of the backtracking feature. However, as committed choice languages have no backtracking, an efficient incremental garbage collection method is important in KL1 implementation.

Incremental garbage collection by multiple reference bit (MRB) [2] is introduced in KL1-B architecture. MRB is one-bit information in a pointer to show whether the pointed data object is possibly referred to by other data objects (*on-MRB*) or not (*off-MRB*). When a data object is pointed by a pointer with *off-MRB*, the corresponding memory area can be reclaimed after reading its contents. Therefore, the locality of memory references can be raised using MRB incremental

```
collect_list_B Ai:
    if MRB of Ai is off
      then reclaim the cons cell pointed by Ai
      else proceed to the next instruction.
```

Figure 4: A KL1-B instruction: *collect_list*$_B$

garbage collection. The MRB is also used to implement the efficient stream merge and array operations in KL1 programs. For example, an array element can be destructively updated when the array is referenced by an *off-MRB* pointer.

The MRB is maintained in each KL1-B instruction. In addition, several garbage collection instructions are introduced to KL1-B. The compiler detects candidate places where garbage cells can possibly be collected, and inserts garbage collection instructions at appropriate places. *Collect_list*$_B$ in Figure 4 is a typical KL1-B instruction which reclaims memory area by checking the MRB at run-time. Memory area can be also reclaimed during dereference. Unification in KL1 produces a chain of variable cells pointed by indirect pointers. When a variable cell pointed by an indirect pointer with *off-MRB* is found in dereference, the memory area for the variable cell can be reclaimed.

2.4 Summary of KL1-B instruction features

The characteristics of the KL1-B instruction features can be summarized as follows.

Conditional dereference: Unification instructions in KL1-B are classified as passive unification, active unification or argument preparation [6]. Dereference is required at the beginning of passive and active unification instructions. In dereference, a register is first tested whether its content is an indirect pointer or not. When it is an indirect pointer, the pointed cell is fetched into the register, then the data type is tested again. Otherwise, unification is performed depending on the data type.

Embedded incremental garbage collection in dereference: Variable cells pointed by an indirect pointer with *off-MRB* can be reclaimed. These cells are reclaimed in dereference. Therefore, each dereference operation includes the MRB test and, possibly, reclamation operation.

Polymorphic instructions: Many instructions in KL1-B include run-time data type checks even after dereferencing. For example, the active unification instruction, *get_list_value*$_B$, in Figure 3 has one of four kinds of actions, selected by the data type check: (1) when *Ai* is a list, general unification is performed; (2) when *Ai* is an uninstantiated variable without suspended goals, the *Xi* is assigned into the variable cell; (3) when *Ai* is an uninstantiated variable with suspended goals, these suspended goals are resumed with the instantiation of *Ai*; and (4) otherwise, the unification fails.

Consequently, most instructions in KL1-B include run-time data type checks. The actions that follow the run-time type check are very different.

3 Efficient KL1 Implementation by Macro-call

3.1 Alternatives for KL1-B implementation

The following alternatives can be candidates of the KL1-B implementation on the *PIM*:

- expanded compiled code by RISC-like instruction set
- KL1-B interpretation by micro-program.

A RISC or RISC-like instruction set can be executed using short pipeline and has advantages in hardware design cost. However, considering the naive expansion of KL1-B using low-level RISC instructions, the static code size of compiled programs will be very large. This may cause instruction cache misses or may increase common bus traffic in tightly-coupled multiprocessor, such as a *PIM* cluster.

Local coherent cache [1] in each processor element is introduced in *PIM* architecture. The cache mechanism increases the efficiency of local execution on each processor element. It also enables high-speed communication within a cluster. Here, reducing common bus traffic is a more important design issue than reducing cache miss ratio [7]. The deficiency of the expanded compiled code is fatal for such systems. Our software simulation found the expanded compiled code causes the increase of the common bus traffic, so that the total performance of a cluster will seriously degrade [7].

On the other hand, the static code size can be small in a high-level instruction set computer (HLIC) with micro-program, such as the PSI [8]. However, the KL1-B interpretation by micro-instructions has the following disadvantages to design a high-performance processor element for the *PIM*.

First, short vertical micro-instructions are not advantageous in their performance. Then, rather long micro-instructions may be incorporated, so that skilled designers would strive to write the micro-program for KL1-B. Here, they would find it difficult to make full use of micro-instruction fields. This is because the actions of each KL1-B instruction are determined by run-time data type checks as in section 2.4.

The data type check often selects to proceed the next KL1-B instruction without any operations. In addition, KL1-B includes simple instructions, such as register-to-register move instructions. Therefore, when every KL1-B instruction is interpreted by micro-instructions, HLIC may suffer from the cost of useless micro-instruction dispatching.

Here, if a processor has very efficient conditional subroutine call function on data tag, accompanied by a RISC-like instruction set, the processor can use only the best features of both RISC and HLIC. The efficient one-level subroutine call function is implemented on the processor element of *PIM* by introducing macro-call instructions and internal instructions.

3.2 Macro-call function in the processor element of *PIM*

The processor element of *PIM* has two kinds of instructions, external and internal. *External instructions* are mainly used to represent compiled codes of user programs. The external instruction set includes macro-call instructions. The macro-call instruction first test the data type of a register given as its operand, then it will or will not invoke its macro-body in the internal instruction memory (IIM) depending on the result of the test. The macro-bodies stored in the IIM are written in *internal instructions* by system designers, just as microprogram of HLIC processors.

Here, most of both external and internal instructions are common RISC-like instructions, including KL1 specific instructions. Therefore, system designers can flexibly specify the machine level language, KL1-B, using one kind of RISC-like instructions instead of complicated micro-instructions in conventional computers. Considering the difficulty to make full use of long micro-instructions, this scheme is advantageous to system designers.

3.3 Macro-call instructions and their conditions

A macro-call instruction can be regarded as a *light-weight* subroutine call or as a high-level instruction realized by microprogram. Macro-call instructions are introduced to implement high-level KL1-B instructions.

The run-time test of the type tag is a primitive operation used very often in KL1 implementation. As discussed in section 2.4, most unification includes a multi-way branch based on the goal argument type. Some Prolog machines, such as the PSI [10], have a hardware-supported multi-way branch function. The processor element of *PIM* does not have such hardware. This is because: (1) it is costly to adopt a hardware-supported multi-way branch to a pipeline processor; and (2) branches taken in run-time are biased; not all possibilities are chosen by equal chances. The *PIM* instruction set has only two-way tag condition in macro-call instructions and in tag branch instructions, but various tag conditions can be specified in them.

The macro-call instruction has the form:

`MacroCall` **if** *cond*, *Address* **with** $\texttt{reg}_0$, $\texttt{reg}_1/\texttt{immed}_1$, $\texttt{reg}_2/\texttt{immed}_2$, ..., $\texttt{reg}_n/\texttt{immed}_n$

where:

Address :	Entry address of the internal instruction memory
$\texttt{reg}_i/\texttt{immed}_i$:	register number for the argument of macro-call or short immediate constant
cond :	`And`, `NotAnd`, `Or`, `NotOr`, `Xor`, `NotXor`, `XorMask`, `NotXorMask` condition for the macro-body invocation.

A tag condition, *cond*, can be specified as a logical operation between a register tag, $\texttt{reg}_0$ and a register tag, $\texttt{reg}_1$, or an immediate tag, `immed`. In addition, a tag-mask register can be used to mask logical operation (see `XorMask`, `NotXorMask`). To avoid frequent update of the tag mask register, some macro-call instructions have an immediate tag mask in their operand.

In the processor element of *PIM*, various hardware flags, such as the condition code of ALU operation or an interrupt flag, can be accessed as the tag of dedicated registers. Therefore, these flags can also be used as conditions of macro-call.

3.4 Indirect registers for internal instructions

The macro-body is specified by internal instructions stored in the IIM. The internal instruction can specify virtual registers, called indirect registers, as its register operands. Through the indirect registers, internal instructions can handle the operands of a macro-call instruction which has invoked the macro.

There are two kinds of indirect registers. One is used to get the operand of the macro-call instruction as an immediate value. The other is used to access the contents of the register that is specified in the macro-call operand. Each indirect register corresponds to the operand position of the macro-call instruction. Therefore, the operands of a macro-call can be efficiently passed to its macro-body. In addition, a macro-body can be used flexibly by changing the argument of the macro-call instruction.

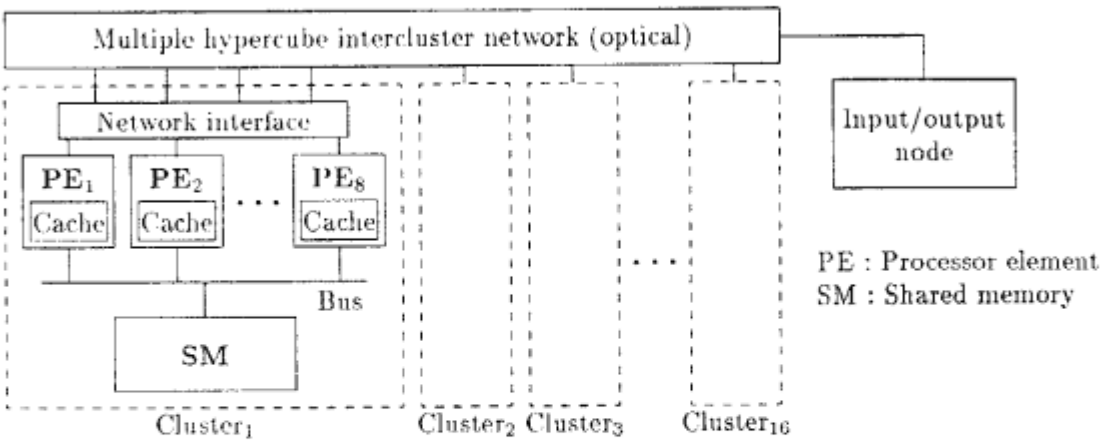

Figure 5: PIM overview

Each internal instruction has an additional bit, called *eoi*, to specify the exiting point from macro-body, so that the execution of macro-body can finish at any non-branch instruction.

4 PIM Architecture Overview

4.1 PIM configuration

The parallel inference machine prototype, *PIM*, is planned to include 128 processor elements. The target processor element performance is 200K to 500K RPS[3], so that 10 to 20M RPS is expected as the total performance for practical applications. The *PIM* has a hierarchical structure, as shown in Figure 5. Eight processor elements (PEs) form a cluster, communicating through shared memory (SM) over a common bus. Processor elements within each cluster share one address space, so that they can quickly communicate by reading or writing shared memory. The *PIM* will consist of 16 clusters. It will be possible to increase the number of clusters. Processor elements are connected with processor elements in other clusters by a multiple hypercube network. Because each cluster has its own address space, inter-cluster parallel processing is performed by sending and receiving message packets with address transformation. Each processor element has a network communication port to send and receive messages between clusters.

4.2 Processor element configuration

Figure 6 shows the processor element configuration. The CPU has two instruction streams, one is from the instruction cache, and the other is from the internal instruction memory (IIM). Hopefully, the CPU will execute an instruction at every machine cycle using a four-stage pipeline in most cases. The IIM is similar to a writable microprogram store. The IIM can store about 8K internal instructions, which are preloaded by special instructions. The CPU has two co-processors: a network interface unit (NIU) and a floating point processor unit (FPU). The CPU has a common protocol to use both co-processors.

The processor element includes two caches: an instruction cache and a data cache. The contents of both cache memories are identical. They are provided to enable the CPU to fetch both data and instructions every machine cycle. The cache consistency protocol used in *PIM* is based on the

[3]RPS: KL1 goal reduction per second

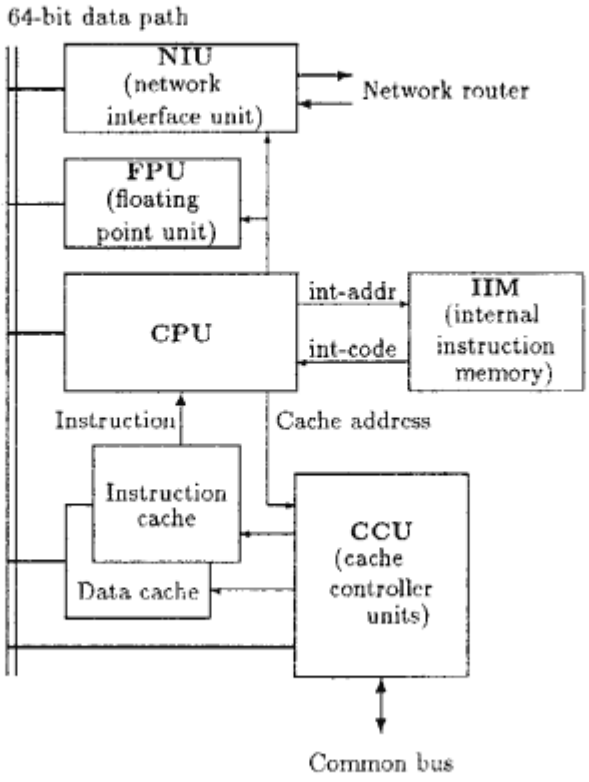

Figure 6: *PIM* processor element configuration

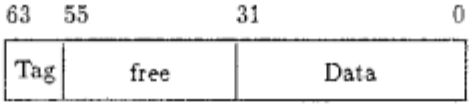

Figure 7: KL1 tagged data representation

protocol described in [9] (called the *Illinois* protocol) with some functional extensions [7]. Lock operations are essential for implementing KL1 in the shared memory multiprocessor. The *PIM* cache enables a *light-weight* lock and unlock operation by using the cache block status, lock address registers, and busy-wait locking scheme.

Taking practical KL1 implementation into consideration, 40-bit KL1 data (an 8-bit tag and 32-bit data) is necessary. Normal KL1 data is placed by 40-bit KL1 tagged data in aligned 64-bit words [4] in the *PIM* memory system, as shown in Figure 7. The MRB is assigned in its tag part. Instructions and some data structures, such as strings or floating point numbers, utilize the full capacity of 64 bits.

4.3 Registers in CPU

The processor element includes 32 general-purpose registers and 6 dedicated registers. Each general-purpose register has an 8 bit tag and 32-bit data. The dedicated registers are implemented as discrete registers in CPU LSI. They include a condition code register, an interrupt request register,

[4] As the alternative of MRB garbage collection, LRC(lazy reference counting) [4] is now being examined. In LRC method, the free three bytes in each tagged data will be used as reference count field.

and a tag mask register. Most flags, such as the condition code of ALU, are placed in their tag part. These registers are specified by a 6-bit register specifier in most instructions. In addition to the above registers, the processor element has special registers and co-processor registers.

The *PIM* instruction set can use 16 indirect registers in its register operand. Through indirect registers, internal instructions can handle the operands of a macro-call instruction which has just invoked the internal program code. It can represent either the immediate value or the contents of a register specified in the operand of macro-call instruction.

4.4 Instruction set

The *PIM* instruction set can be classified as follows. (Appendix shows the list of principal instructions.)

4.4.1 Branch instructions

The instruction set includes three kinds of branch instructions: external branch, internal branch and macro-call. (Table 3.) An external branch instruction can be used not only as an external instruction but also as an internal instruction. In both cases, its branch target is an external instruction whose address is specified by a register, or the instruction pointer, with address offset. Internal branch instructions are used to branch within internal instructions, whose branch address is specified by the absolute address of the IIM. Macro-call instructions invoke macro-bodies in the IIM as in section 3.2.

4.4.2 ALU instructions

ALU instructions have two source registers and one destination register. (Table 4.) These instructions can be classified into three kinds: 32-bit data computation, 8-bit tag computation, and 40-bit computation. Most ALU instructions can be used both as external and internal instructions. Although logical operations are available for both the tag and data, arithmetic operations and shift operations are limited to the data part.

4.4.3 Memory access instructions

Memory access instructions include the instructions for dereference and MRB, as well as the instructions that access shared memory with coherent cache control. (Table 5.) Each memory access instruction reads data to or writes data from a register from or to a memory location whose address is specified by a register and immediate address offset. The transferred data width can be 8, 16, 32 bits; 32 bits with an 8-bit tag; or 64 bits. A new tag can be given in memory access instruction `WritewTag`. Instructions to move the tag part of a register to the data part of another register, and its reverse, are provided as *register move* instructions.

In MRB incremental garbage collection, each variable cell or structure is allocated from a free list and, when reclaimed, its memory area is linked to a free list. For efficient free list operations, the `PUSH` and `POP` instructions are used. `PUSH` can link a variable cell or a structure to the free list, and `POP` can allocate it from the free list, in one machine cycle.

Their actions are specified as follows.

```
PUSH  Rs, Ra, offset: M[Ra+offset]←Rs; Rs←Ra
POP   Rd, Ra, offset: Rd←Ra;           Ra←M[Ra+offset]
```

Here, imagine `ft` to be the free list top pointer register, where

Table 1: Pipeline stages of ALU and memory access instructions

	ALU operation	*Memory access*
D	Decode	Decode / register read (address)
A	–	Address calculation
T	Register read	Cache access (address)
B	ALU / register write	Cache access (data) / register write

```
POP r1, ft, -
```

allocates a cell to `r1` from the free list pointed by `ft`, and

```
PUSH ft, r1, -
```

links a cell to the free list pointed by `ft` from `r1`. The `POPwTag` instruction is used to give a new tag in a pop operation. Two dedicated instructions, `MRBorRead` and `DEREF`, are provided to support MRB maintenance during dereference. `MRBorRead` reads data from memory, then sets the MRB *on* in the destination register, if one or both MRBs of the address register and the fetched data are *on-MRB*. `DEREF` is a merged instruction of both `MRBorRead` and `POP`.

The instruction set includes memory access instructions corresponding to each cache function. Exclusive memory access instructions, `LockRead` and `WriteUnlock`, are also provided. These instructions have restrictions on interruptions or traps, as well as they may cause fatal errors in incorrect usages. Therefore, the use of these instructions will be limited to internal instructions.

4.5 Execution pipeline

The processor element uses an instruction buffer and a four-stage pipeline, **D A T B**, to attempt to issue and complete an external instruction every cycle. The target of the basic machine cycle is 50 nanoseconds. External instructions are either four, six or eight bytes long, so that the instruction buffer has a hardware aligner. Each internal instruction requires two additional stages, preceding stage **D**, to set the internal instruction address (stage **S**) and to fetch the instruction (stage **C**). Then they are executed using the same pipeline stages, **D A T B**.

Table 1 shows the pipeline stages in both ALU and memory access instructions. All instructions modify general-purpose registers in the last **B** stage, thereby avoiding write conflicts. Internal forwarding is done by hardware so that the result of a register-to-register instruction can be used by the next instruction even though that result has not yet been written to the register file.

In a branch instruction to an external instruction, the branch target instruction is fetched at stage **B** in the same way as memory read instructions. Therefore, ordinary branch instructions may cost three additional cycles to branch. Delayed branch instructions can avoid wasting the three cycles by executing other effective instructions.

Although most tag branch instructions test their condition at stage **B**, macro-call instructions and some internal branch instructions test their condition at stage **A**. Figure 8 shows the macro-call instruction pipeline. A macro-call instruction initiates the internal instruction fetch (stage **S**) at its stage **D**, then tests its condition at stage **A** [5]. Therefore, even if the branch condition is taken, a macro-call instruction costs only one additional cycle to invoke its macro-body in the IIM. In addition, delayed macro-call instructions are provided to avoid the penalty. Return from macro-call, i.e., return from a macro-body to the external instruction just next to the macro-call

[5] When the register for tag condition is set by a memory read instruction just before the macro-call instruction, the stage **A** of the macro-call instruction is stretched.

When the condition is true:							
D	A						: macro-call instruction (condition test at **A**)
	D						: next external instruction (canceled)
S	C	D	A	T	B		: first internal instruction
	S	C	D	A	T	B	: second internal instruction
When the condition is false:							
D	A						: macro-call instruction (condition test at **A**)
	D	A	T	B			: next external instruction
		D	A	T	B		: external instruction
End of macro body:							
S	C	D	A	T	B		: Internal instruction with *coi*
	S	C					: canceled internal instruction
		S					: canceled internal instruction
			D	A	T	B	: next external instruction

Figure 8: Macro-call instruction pipeline

wait_list$_B$ *Ai, Label :*
if tag(*Ai*) is `LIST` **then** proceed to the next code
elseif tag(*Ai*) is `REF`
then put the dereference result of *Ai* to *Ai*
if tag(*Ai*) is `LIST` **then** proceed to the next code
elseif *Ai* is uninstantiated
then push *Ai* to the suspension stack and jump to *Label*
else jump to *Label*
else jump to *Label*

Figure 9: A KL1-B instruction: *wait_list*$_B$

instruction, can be indicated by a one-bit flag: *coi*. The internal instruction memory has an *coi* field for each instruction. Therefore, the execution of macro body will finish with no overhead (except for branch instructions.) (See Figure 8.)

5 Example of KL1-B Implementation

5.1 High-level instructions using macro-call

Macro-call instructions are used to implement high-level KL1-B instructions. For example, the KL1-B instruction, *wait_list*$_B$, in Figure 9 first tests the data type of a given argument. If the data type is the expected `LIST`, this instruction finishes. Otherwise, the data type selects the following operation in Figure 9.

A macro-call instruction has a condition to invoke its macro-body in the IIM. In the above example, a macro-call instruction corresponding to *wait_list*$_B$ is written as follows, where `LIST` is an immediate tag value and `acp` is an alternative clause pointer register for *Label*.

`MacroCall` *wait_type* (if) `NotXorMask` (with) `ai, LIST, acp;`

The data type tag of register `ai` is tested first[6]. If the register `ai` has a value with the `LIST`

[6]Assume that MRB is assigned in 8-bit tag field, and that the tag mask register holds a value to mask the MRB. The operator, `NotXorMask`, is used to test `LIST` type masking its MRB.

```
MacroCall wait_type NotXorMask ai, LIST, acp;

wait_type:  JumpNotXor @r0, REF, @r2;
            DEREF ptr, @r0;
            MJumpNotAnd @r0, UNB, case_unbound;
            MJumpNotAnd @r0, MRP, case_mrp;
            PUSH fr1, ptr;
            MJumpNotXorMask @r0, @d1, wait_type;
            Nop (eoi);
            ..........
```

Figure 10: Macro-body for $Wait\_list_B$

type, this macro-call instruction simply finishes. Otherwise, this macro-call instruction invokes an internal routine whose entry address is specified as *wait_type*. Figure 10 shows the macro-body in internal instructions at *wait_type*. Here, `@r0` and `@r2` are indirect registers corresponding to the arguments, `ai` and `acp`, in the macro-call instruction. `@d1` is also an indirect register to show the immediate value in the second argument of the macro-call, namely, immediate tag `LIST`. The first internal instruction, `JumpNotXor`, tests the tag of `@r0`, namely `ai`. When the tag is `REF`, proceeds to the next instruction for dereference. Otherwise, it jumps out to the external instruction specified by `@r2`, namely `acp`.

The macro-body in Figure 10 can be used for other KL1-B instructions. Assume that the data type of a four-element vector is represented by a tag, `VECT4`, and that a KL1-B instruction, $wait\_vect4_B$, unifies a goal argument with a four-element vector. The macro-call instruction corresponding to $wait\_vect4_B$ can be:

```
MacroCall wait_type (if) NotXorMask (with) ai, VECT4, acp;
```

5.2 Compiled code

Figure 11 shows a part of a sample compiled code: machine instructions and KL1-B instructions for *append*. Here, KL1-B instructions that include dereference and unification are represented by macro-call instructions. Ten KL1-B instructions in this example are represented by six macro-call instructions and eight RISC-like instructions. In ordinary execution, three of the macro-call instructions actually invoke their macro-bodies, and other three only proceed to the next instruction. The processor element performance estimated from the compiled code is over 600K RPS for the append program. Note that the estimated performance includes the incremental garbage collection cost using MRB.

6 Conclusion

The macro-call function for the efficient KL1-B implementation was discussed. The processor element architecture for the parallel inference machine prototype, *PIM*, was presented. Most *PIM* instructions are RISC-like instructions which can be executed in one machine cycle using four-stage pipeline. The instruction set includes tagged architecture and MRB incremental garbage collection support. The macro-call instructions are introduced to invoke their macro-body efficiently. The condition of the macro-call instruction can be specified as register tag computation. The internal

```
append([H|X],Y,Z) :- true | Z = [H|ZZ], append(X,Y,ZZ).
```

```
app/2/1:  MacroCallNotXorMask a1, LIST, acp, waittype;   % wait_list_B a1
          Read a1, a4, -;                                % read_variable_B a4
          Read a1, a5, 8;                                % read_variable_B a5
          MacroCallXorMask a4, REF, a1, 0, readvar;      %
          MacroCallXorMask a5, REF, a1, 8, readvar;      %
          MacroCallXorMask a1, MRBLIST,-, alloclist;     % reuse_list_B a1
          Write a1, a4;                                  % write_value_B a4
          MacroCallXorMask fr1, NIL, genvar;             % write_variable_B a6
          WritewTag a1, fr1, 8, REF;                     %
          POPwTag a6, fr1, REF;                          %
          MacroCall a1, a3, getlistvalue;                % get_list_value_B a1,a3
          DelayJumpNotAnd cc, SLIT, app/2/1;             % execute_B append
          Move a5, a1;                                   % put_value_B a5,a1
          Move a6, a3;                                   % put_value_B a6,a3
```

Figure 11: A sample compiled code: Append

instructions of the macro-body can use indirect registers to access registers or immediate value in the macro-call instruction's operands. As the result, the processor element of *PIM* has the advantages of high-level instruction set computer as well as that of RISC-like computer.

The instruction set has been specified. The detailed design of the CPU, the CCUs and the NIU is completed. The target of the basic machine cycle is 50 nanoseconds. The LSI implementations of these chips, as well as the design of the processor element board, are now in progress.

Acknowledgement

We wish to thank all of the PIM research members both in ICOT and Fujitsu Limited. Especially we thank ICOT researchers: Dr. K. Taki, Mr. K. Nakajima, Mr. A. Matsumoto, and Mr. T. Nakagawa, and Fujitsu researchers: Mr. S. Arai and Mr. A. Asato, for their useful comments. We also wish to thank Mr. H. Murano and Mr. H. Tamura in Fujitsu Limited for their help in developing the LSIs and their useful comments. Finally, we would like to thank ICOT Director, Dr. K. Fuchi, the chief of the fourth research section, Dr. S. Uchida, the general manager of Information Processing Division in Fujitsu Laboratories, Mr. J. Tanahashi, and the manager of Artificial Intelligence Laboratory in Fujitsu Laboratories, Mr. H. Hayashi, for their valuable suggestions and guidance.

References

[1] P. Bitar and A. M. Despain. Multiprocessor cache synchronization. In *Proc. of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[2] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 276–293, 1987.

[3] A. Goto. Parallel Inference Machine Research in FGCS Project. In *US-Japan AI Symposium 87*, pages 21–36, Nov. 1987.

[4] A. Goto et al. Lazy Reference Counting – An Incremental Garbage Collection Method for Parallel Inference Machines –. TR 338, ICOT, 1988. (To appear in the Proc. of the Joint Fifth International Logic Programming Conference and Fifth Logic Programming Symposium).

[5] A. Goto and S. Uchida. Toward a High Performance Parallel Inference Machine –the Intermediate Stage Plan of PIM–. In *Future Parallel Computers*, pages 299–320. LNCS 272, Springer-Verlag, 1986.

[6] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 468–477, 1987.

[7] A. Matsumoto et al. Locally parallel cache designed based on KL1 memory access characterestics. TR 327, ICOT, 1987.

[8] K. Nakashima and H. Nakajima. Hardware architecture of the sequential inference machine: PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, pages 104–113, San Francisco, 1987.

[9] M.S. Papamarcos and J.H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, 1984.

[10] K. Taki et al. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 398–409, Tokyo, 1984.

[11] K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. TR 208, ICOT, 1986. (also to appear in Programming of Future Generation Computers, North-Holland, Amsterdam, 1987.).

[12] K. Ueda. Introduction to guarded horn clauses. TR 209, ICOT, 1986.

[13] D.H.D. Warren. An abstract prolog instruction set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.

APPENDIX: PIM Instruction Set

The *PIM* instruction set is listed in the following. Table 2 lists the notation for instruction operands.

Table 2: Notation for instruction operands

Six-bit register specifier		*Immediate value*	
`Rs,Rs1,Rs2`	source registers	`imm(8/32/40)`	immediate constant
`Rd`	destination register	`imtg(8)`	eight-bit immediate tag
`Ra`	base address register	`ofst(8/16/24/32)`	immediate address offset
`Rt1,Rt2`	register for testing tag	`retofst(8)`	offset for return address
`R,R1,R2,..R5`	argument for macro-call	`iaddr(16)`	internal memory address

[a] Branch instructions

Table 3: Branch instructions

Instruction	*Operands*	*Comment*
External branch		
`Jump`*Cond*		
`Delay-Jump`*Cond*`Imm Mask`	`Rt1, Rt2/imtg, ofst`	Tag jump (delay)
`Delay-Jump`*Cond*`40`	`Rt1, Rt2/imtg, imtg, ofst`	Tag jump under immediate mask (delay)
`Delay-`	`Rt1, Rt2, ofst`	40-bit compare jump (delay)
`sKip`*Cond*	`Rt1, Rt2/imtg, imm`	Conditional skip
`Jump/DelayJump`	`Ra, ofst(32)`	Jump (delay)
`JAL/DelayJAL`	`Ra, ofst(24), retofst`	Jump and link (delay)
Internal branch		
`MJump`*Cond*		
`Delay-MJump`*Cond*`A`	`Rt1, Rt2/imtg, iaddr`	Tag jump (delay)
`Delay-MJump`*Cond*`ImmMask`	`Rt1, Rt2/imtg, iaddr`	Tag jump at A-stage (delay)
`Delay-MJump`*Cond*`40`	`Rt1, Rt2/imtg, imtg, iaddr`	Tag jump under immediate mask (delay)
`Delay-MJump`*Cond*`40A`	`Rt1, Rt2, iaddr`	40-bit compare jump (delay)
`Delay-`	`R1, R2, iaddr`	40-bit compare jump at A-stage (delay)
`MsKip`*Cond*	`Rt1, Rt2/imtg`	Conditional skip
`MJAL/DelayMJAL`	`R, iaddr`	Jump and link (delay)
`MJump/DelayMJump`	`iaddr`	Jump (delay)
Conditional macro call		
`MacroCall`*Cond*	`Rt1,Rt2/imtg,[R3,R4,R5] iaddr`	Macro call
`DelayMacroCall`*Cond*	`Rt1,Rt2/imtg,[R3,R4,R5] iaddr`	Delayed
`MacroCall`	`R1,[R2,R3,R4,R5,] iaddr`	Unconditional macro call
`DelayMacroCall`	`R1,[R2,R3,R4,R5,] iaddr`	Delayed

Note: *Cond* : `And, NotAnd, Or, NotOr, Xor, NotXor, XorMask, NotXorMask`

[b] ALU instructions

Table 4: ALU Instructions

Instruction	*Operands*	*Comment*
Dop	Rs1, Rs2/imm, Rd	Normal ALU operation
*Dop*40	Rs1, Rs2/imm, Rd	40-bit ALU operation
Shift	Rs, R/imm, Rd	Shift operation
AddwTag/SubwTag	Rs1, Rs2/imm, Rd, imtg	ALU operation giving a new tag
AddImm/LoadImm	Rsd, imm(32)	Add or load long immediate constant
SextB/HW	Rs, Rd	Sign extension
Top	Rs1, Rs2/imm, Rd	Tag computation
PEC	Rs, Rd	Priority encode
Move	Rs, Rd	Tag and data transfer
MoveTD	Rs, Rd	Move tag to data transfer
MoveDT	Rs, Rd	Move data to tag transfer

Note: *Dop*: Add, AddCarry, Subtract, SubtractCarry, AND, Or, Xor, NOT
*Dop*40: AND40, Or40, Xor40, XorMask40
Shift: ShiftLeft, ShiftRight, ShiftLeftDouble, ShiftRightDouble
Top : TagAnd, TagOr, TagXor, TagXorMask

[c] Memory access and tag handling

Table 5: Memory access instructions

Instruction	*Operands*	*Comment*
Read	Rd, Ra, ofst	Read tag and 4-byte data
ReadB/HW/W/DW	Rd, Ra, ofst	Read 1, 2, 4, 8-byte data
Write	Rs, Ra, ofst	Write tag and 4-byte data
WriteB/HW/W/DW	Rs, Ra, ofst	Write 1, 2, 4, 8-byte data
WritewTag	Rs, Ra, ofst, imtg	Write 4-byte data giving a new tag
PUSH	Rs, Ra, ofst	Push data into a free list
POP	Rd, Ra, ofst	Pop up data from a free list
POPwTag	Rd, Ra, ofst, imtg	Pop up data giving a new tag
MRBorRead	Rd, Ra, ofst	Read data with mrb OR
DEREF	Rd, Ra, ofst	Pop up data with mrb OR
DirectWrite/B/HW/W/DW	Rs, Ra, ofst	Write data in Direct_Write cache mode
ReadPurge	Rd, Ra, ofst	Read data, followed by cache purge
ReadInvalidate	Rd, Ra, ofst	Read data, invalidating other cache
LockRead	Rd, Ra, ofst	Lock address and read data
WriteUnlock	Rs, Ra, ofst	Write data and unlock address
Unlock	Ra, ofst	unlock address