PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

MACRO-CALL INSTRUCTION FOR THE EFFICIENT KL1 IMPLEMENTATION ON PIM

Tsuyoshi Shinogi   Kouichi Kumon   Akira Hattori
Fujitsu Limited
1015, Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

Atsuhiro Goto   Yasunori Kimura   Takashi Chikayama
Institute for New Generation Computer Technology
4-28, Mita-1, Minato-ku, Tokyo 108, Japan

ABSTRACT

Parallel inference machine (PIM) systems are being developed in the Japanese FGCS project. The kernel language, KL1, and its virtual machine instruction set, KL1-B, have also been designed at ICOT.

A PIM pilot machine, PIM/p, is now being developed. Since the action of KL1-B instructions varies a great deal depending on their argument type, it is necessary to check them as fast as possible. So, RISC-like instructions with *macro-call* functions are incorporated in the processor element architecture of the PIM/p.

Macro-call function corresponds to the microprogram in high-level instruction set computers (HLICs). Conditional branch is carried out quickly in macro-call invocation, and the code size of compiled programs remains small by using this. So macro-call instructions are suitable for implementing instructions of KL1-B, whose actions vary depending on their argument type.

The merits and demerits of both RISCs and HLICs in relation with the implementation of KL1 are discussed. Then the macro-call function which is introduced to realize both the features in the processor element is described. Finally how the KL1-B instructions can be executed by the macro-call function is presented along with the hardware specification of PIM/p such as pipeline mechanism.

1 INTRODUCTION

In the Japanese FGCS project, parallel inference machine (PIM) systems are being developed based on a logic programming framework at ICOT (Goto and Uchida 1986, Goto et al. 1987, Goto et al. 1988). A kernel language, KL1, has been designed. A PIM pilot machine, PIM/p, tailored to KL1, is being developed. The principal aim of parallel processing is to improve execution performance, and enables users to solve large application problems. The development of a high-performance processor element is, of course, the first step to the target parallel inference machine. The target processor element performance is 200K to 500K RPS[1], so that 10 to 20M RPS is expected as the total performance for practical applications.

In PIM/p, 128 processor elements are connected in a hierarchical structure, as shown in Figure 1. 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, and a 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.

KL1 has the features for efficient execution that conventional machines lack. Three of these features are: (1) unification is a *polymorphic* operation on, usually, dynamically constructed linked data structures; (2) the execution context, which is usually 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 (Kimura and Chikayama 1987), a virtual machine instruction set for KL1. Experiments using KL1-B emulators found: (1) most instructions in KL1-B include run-time data type checks; (2) the action that follows the run-time data type check within a KL1-B instruction varies widely depending on the type, even though the run-time data type check selects only a portion of the actions. 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/p, this article presents

[1]RPS: KL1 goal reduction per second

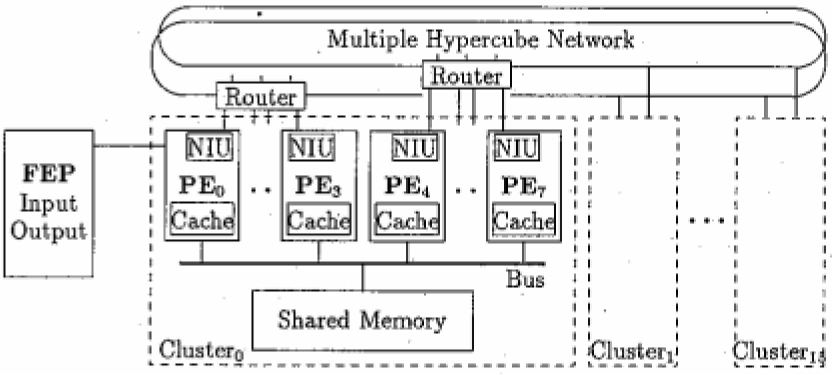


Figure 1: PIM/p overview

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 efficiently 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 based on GHC (Ueda1 1986, Ueda2 1986). 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 succeeded 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 most of GHC's descriptive power is still kept. 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 (Kimura and Chikayama 1987) is a virtual machine language interfacing parallel inference machine hardware and KL1, just as WAM (Warren 1983) interfaces Prolog and sequential machines. In other words, KL1-B represents the abstract architecture of the parallel inference machine.

To build an efficient parallel inference machine, execution on each processor element must be as efficient as possible. Therefore, KL1-B was first designed based on sequential execution (Kimura and Chikayama 1987). Then, it was extended for parallel execution. The basic execution mechanism of KL1 from the implementation point of view is summarized in the following subsections[2].

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 arguments' slots, a pointer to the compiled code corresponding to its predicate name, and some control information. The arguments' slots include atomic values or pointers to variables or structure bodies in the heap.

The state of a goal can be a *ready goal*, a *suspended goal* or a *current goal.* The *ready* goal-records are linked into a list forming a *ready-goal-stack.* A KL1-B instruction, *proceed*$_B$[3], 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

[2]An explanation of each KL1-B instruction can be found in Chikayama and Kimura(1987) and Kimura and Chikayama(1987).

[3]In this article, each KL1-B instruction is written with postfix B, e.g. *proceed*$_B$.

```
get_list_value_B Xj, Ai:
    put the dereferenced 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 2: A KL1-B instruction: *get_list_value*$_B$

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 the next candidate clause is tried.

If no clause is selected, the current goal becomes a suspended goal by a *suspend*$_B$ instruction. That is, the current goal is linked to these variables, which caused the suspension, to realize a non-busy waiting synchronization mechanism between KL1 goals.

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 2 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.

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. Depth-first scheduling is adopted.

2.3 Incremental garbage collection by MRB

KL1 is a concurrent language with no side effects, so 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. Since the locality of memory references is supposed to be very low during garbage collection by widely used schemes such as 'mark and sweep' method, cache misses and memory faults would occur often.

In sequential Prolog (Warren 1983), this problem is

```
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 3: A KL1-B instruction: *collect_list*$_B$

not very serious because of the backtracking feature. However in KL1 implementation, an efficient incremental garbage collection method is required because committed choice languages such as KL1 have no backtracking feature.

Incremental garbage collection by multiple reference bit (MRB) (Chikayama and Kimura 1987) 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.

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 3 is a typical KL1-B instruction which reclaims memory area by checking the MRB at run-time. Memory area can be also reclaimed during dereferencing. 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 dereferencing, 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 dereferencing:

Unification instructions in KL1-B are classified as passive unification, active unification or argument preparation (Kimura and Chikayama 1987). Dereferencing is required at the beginning of passive and active unification instructions. In dereferencing, an argument 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 dereferencing:

Variable cells pointed by an indirect pointer with *off-MRB* can be reclaimed. These cells are reclaimed in dereferencing. Therefore, each dereferencing 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 2 executes 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/p:

- expansion of compiled code by RISC-like instruction set,
- KL1-B interpretation by microprogram.

The merits and demerits of these approaches are summarized as follows.

RISC-like instruction set:

A RISC or RISC-like instruction set can be executed using a 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 multiprocessors, such as a PIM/p cluster.

Here, reducing common bus traffic is a more important design issue than reducing the cache miss ratio (Matsumoto et al. 1987), so the deficiency of the expanded compiled code is fatal for such systems. Our software simulation found that the expanded compiled code causes an increase in the common bus traffic, and that the total performance of a cluster will seriously be degraded as a result (Matsumoto et al. 1987).

Microprogram:

The static code size can be small in a high-level instruction set computer (HLIC) with a microprogram, such as the PSI (Nakashima and Nakajima 1987). However, the KL1-B interpretation by micro-instructions has the following disadvantages regarding design of a high-performance processor element for the PIM/p.

Firstly, rather long micro-instructions may be incorporated to use the low level parallelism specified in each micro-instruction field. Then, skilled designers would strive to write the microprogram for KL1-B, but they would find it difficult to make full use of micro-instruction fields because the actions of each KL1-B instruction are often determined by run-time data type checks as described in section 2.4.

Secondly, the data type check often selects to proceed to the next KL1-B instruction without performing 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 because of useless micro-instruction dispatching.

3.2 Design decisions made for PIM/p

From the above consideration, a new RISC processor with the efficient one-level subroutine call function, called *macro-call*, and with a pipeline mechanism, is designed as a processor element of PIM/p.

By introducing the macro-call function to the RISC processor, the static code size of compiled programs will remain small. And thus, the common bus traffic may not be increased.

Skilled designers will be released from writing hairy microprograms by writing the macro body routine with the RISC instructions and by executing them with the pipeline mechanism.

Conditional branch mechanism of macro-call function described in the next subsection will eliminate the cost of useless dispatching which HLICs may have. And *indirect registers* will enable the fast passing of operands in macro-call instructions to their body routines.

3.3 Macro-call function

The run-time data type check 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 (Taki et al. 1984), have a hardware-supported multi-way branch function. The processor element of PIM/p does not have such hardware. This is because it is costly to adopt a hardware-supported multi-way branch mechanism to a pipeline processor, and because branches taken in run-time are often biased, so not all possibilities are chosen by equal chances. The PIM/p 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.

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

The macro-call instruction has the form:

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

where:

cond : `And, NotAnd, Or, NotOr, Xor, NotXor, XorMask, NotXorMask`
Condition for the macro-body invocation.

$\texttt{reg}_i\texttt{/immed}_i$:
Register number for the argument of macro-call or short immediate constant.

Address :
Entry address of the internal instruction memory.

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/p, 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 Execution mechanism of Macro-call function

The processor element of PIM/p shown in Figure 4, 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.

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

3.5 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. The indirect registers are used only by internal instructions. Through the indirect registers, internal instructions can handle the operands of a macro-call instruction which has invoked the macro.

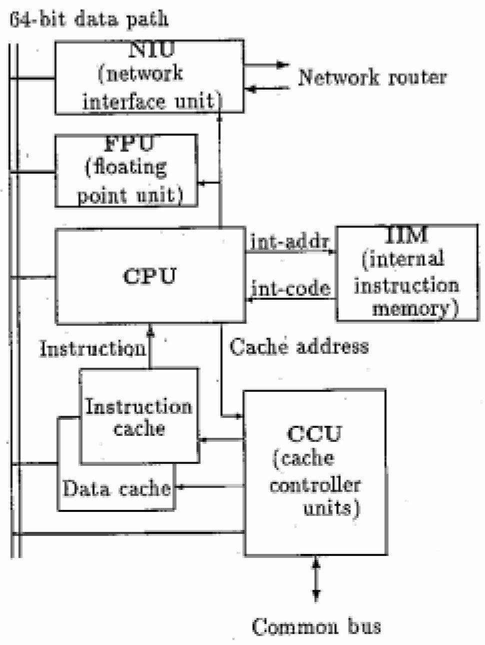
There are two kinds of indirect registers. One is used to get the operand of the macro-call instruction as an immediate value. The other is used to access the contents of the register that is specified in the macro-call operand. Each indirect register corresponds to the operand position of the macro-call instruction, so 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.

4 PIM/p PROCESSOR ELEMENT

4.1 Processor element configuration

Figure 4 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). We hope that 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 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 in



64-bit data path
NIU (network interface unit)
Network router
FPU (floating point unit)
CPU
int-addr
int-code
IIM (internal instruction memory)
Instruction
Cache address
Instruction cache
Data cache
CCU (cache controller units)
Common bus

Figure 4: PIM/p processor element configuration

PIM/p is investigated to be suited for the KL1 execution (Matsumoto et al. 1987).

Lock operations are essential for implementing KL1 in the shared memory multiprocessor. The PIM/p cache enables a *light-weight* lock and unlock operations by using the cache block status, lock address registers, and busy-wait locking scheme.

4.2 Registers in CPU

The processor element includes 32 general-purpose registers and several dedicated registers. Each general-purpose register has an 8-bit tag and 32-bit data. The dedicated registers include a condition code register, an interrupt request register, 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.

The PIM/p 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.

4.3 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. Each internal instruction requires two additional stages, preceding the 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. General-purpose registers are updated only at 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 5 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** [4]. 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, that is, return from a macro-body to the external instruction just next to the macro-call instruction, can be indicated by a one-bit flag: *eoi*. The internal instruction memory has an *eoi* field for each instruction, so the execution of the macro body will finish with no overhead (except for branch instructions). (See Figure 5.)

4.4 PIM/p Instruction set

The PIM/p instruction set can be classified as follows.

4.4.1 Branch instructions

The instruction set includes three kinds of branch instructions: external branch, internal branch and macro-call. 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

[4]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.

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

```
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 eoi
   S  C                 : canceled internal instruction
      S                 : canceled internal instruction
         D  A  T  B     : next external instruction
```

Figure 5: Macro-call instruction pipeline

absolute address of the IIM. Macro-call instructions invoke macro-bodies in the IIM as in section 3.3.

4.4.2 ALU instructions

ALU instructions have two source registers and one destination register. 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 the incremental garbage collection by the MRB scheme, as well as the instructions that access shared memory with coherent cache control. Each memory access instruction reads or writes data in 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.

```
wait_list_B Ai, Label :
  if tag(Ai) is LIST then proceed to the next code
    elseif tag(Ai) is REF
      then put the dereferenced 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 6: A KL1-B instruction: *wait_list*$_B$

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 6 first tests the data type of a given argument. If the data type is the expected LIST, this instruction finishes. Otherwise, the following operation in Figure 6 will be selected by the data type.

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

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

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 7: Macro-body for *Wait_list*$_B$

for *Label*.

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

The data type tag of register ai is tested first[5]. If the register ai has a value with the LIST 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 7 shows the portion 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 dereferencing. Otherwise, it jumps out to the external instruction specified by @r2, namely acp.

The DEREF instruction is used for dereferencing the operand @r0, and putting the result in @r0 and the pointer to it in ptr. The PUSH and POP instructions are used for efficient free list operations. 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.

The macro-body in Figure 7 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 (if) NotXorMask
          (with) ai, VECT4, acp,
          wait_type;
```

[5]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.

5.2 Compiled code

Figure 8 shows a part of a sample compiled code: machine instructions and KL1-B instructions for *append*. Here, KL1-B instructions that include dereferencing 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 the other three only proceed to the next instruction. The performance of the processor element 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 PIM pilot machine, PIM/p, was presented. Most PIM/p, instructions are RISC-like instructions which can be executed in one machine cycle using four-stage pipeline. The instruction set includes the tagged architecture and the 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 instructions of the macro-body can use indirect registers to access registers or immediate value in the macro-call instruction's operands. As a result, the processor element of PIM/p, has the advantages of a high-level instruction set computer as well as those of a RISC-like computer.

The instruction set has been specified. The detailed design of the CPU, the CCUs and the NIU has been 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 at ICOT and at 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 of Fujitsu Limited for their help in developing the LSIs and for 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

```
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 8: A sample compiled code: Append

of Information Processing Division in Fujitsu Laboratories Ltd, Mr. J. Tanahashi, and the manager of Artificial Intelligence Laboratory in Fujitsu Laboratories Ltd, Mr. H. Hayashi, for their valuable suggestions and guidance.

REFERENCES

(Bitar and Despain 1986) 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.

(Chikayama and Kimura 1987) 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.

(Goto et al. 1987) A. Goto. Parallel Inference Machine Research in FGCS Project. In *US-Japan AI Symposium 87*, pages 21–36, Nov. 1987.

(Goto et al. 1988) A. Goto et al. Overview of the Parallel Inference Machine Architecture (PIM) In *the Proc. of the Fifth Generation Computer Systems 1988*, Nov. 1988.

(Goto and Uchida 1986) 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.

(Kimura and Chikayama 1987) 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.

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

(Nakashima and Nakajima 1987) 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.

(Papamarcos and Patel 1984) 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.

(Taki et al. 1984) 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.

(Ueda1 1986) K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. TR 208, ICOT, 1986.

(Ueda2 1986) K. Ueda. Introduction to Guarded Horn Clauses. TR 209, ICOT, 1986.

(Warren 1983) D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.