Invited Paper

Processor Element Architecture for a Parallel Inference Machine, PIM/p

Atsuhiro Goto*, Tsuyoshi Shinogi**, Takashi Chikayama*, Kouichi Kumon** and Akira Hattori**

This paper describes the design of processor element architecture for the parallel inference machine prototype, PIM/p. Several innovative features are incorporated in the processor architecture to suit concurrent logic programming languages such as KL1. The processor's design is based on tagged architecture. With the variety of tag handling operations, instructions can be executed by a one-cycle pitch pipeline. Macro-call instructions are introduced to allow a lightweight subroutine call function for polymorphic operations required in the execution of high-level languages such as logic programming languages. This makes it easy for system designers to define high-level instructions without losing the benefits of the pipelining mechanism. Dedicated instructions are introduced to support incremental garbage collection. Local coherent cache and optimized memory operations tailored to the memory access characteristics of KL1 can reduce common bus traffic in shared-memory multiprocessors. In this paper, we describe the design decisions related to these architectural features. The LSIs are now being fabricated by means of CMOS standard cell technology.

1. Introduction

The Japanese Fifth Generation Computer Systems (FGCS) project involves the development of parallel inference machine systems based on a logic programming framework [9, 7]. The current interest in parallel programming stems from its declarative semantics, which facilitates the writing and debugging of programs and removes most of the need for explicit uncovering and control of synchronization in concurrent programming.

KL1 [4], the kernel language of the parallel inference machine system, was designed on the basis of GHC [23]. GHC is a concurrent logic programming language with clear and simple semantics, which allows programmers to express important concepts in parallel programming, such as inter-process communication and synchronization.

We hope to realize very high execution performance for logic programming in KL1 to promote research on parallel logic programming applications. However, KL1 has features that make conventional machines unsuitable for efficient execution. Three of these features are as follows: (1) unification is a *polymorphic* operation, usually on dynamically constructed linked data structures; (2) the execution context, though small, is frequently switched because of data flow synchronization; and (3) the single assignment feature demands a high memory bandwidth and an efficient memory management scheme.

A parallel inference machine prototype (PIM/p) tailored to KL1 is now being developed, and is planned to include up to 512 processor elements arranged in a hierarchical structure. Eight processor elements form a cluster, communicating through shared memory over a common bus. The clusters are connected with one another by a multiple hypercube packet switching network. This article presents the processor element design for PIM/p.

Some of the innovative features introduced in the PIM/p processor element architecture are (1) a lightweight subroutine call function using macro-call instructions, which exploits the advantages of both hardwired reduced instruction set computers and microprogrammable high-level instruction set computers; (2) architectural support for incremental garbage collection; and (3) local coherent cache and optimized memory operations tailored to KL1 parallel execution, which can reduce common bus traffic within shared memory multiprocessors, such as a PIM/p cluster.

This paper is organized as follows. The concurrent logic programming language, KL1, is briefly introduced in Section 2, and its influence on the processor architecture described. Section 3 and Section 4 describe the design decisions related to the CPU and cache, and innovative architectural features tailored to KL1 programs. Section 5 presents on overview of the processor element implementation. Finally our, conclusions are given outlined in Section 6.

*Institute for New Generation Computer Technology (ICOT) 4-28, Mita-1, Minato-ku, Tokyo 108, Japan
**Fujitsu Limited 1015, Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

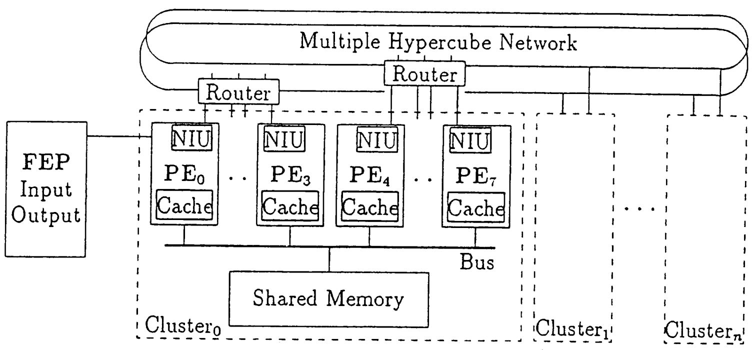

Fig. 1 PIM/p overview.

2. Characteristics of KL1

To understand the underlying motivation of the processor element design presented later in this paper, it is beneficial to review the concurrent logic programming language KL1 and its execution characteristics.

2.1 Brief Introduction to the KL1 Abstract Machine

KL1 was initially specified as flat guarded horn clauses (FGHC) [23], and has been extended into a practical language introducing meta-call and priority scheduling functions. The general-purpose concurrent programming capability of KL1 is shown through the development of a self-contained operating system, PIMOS [4] for Multi-PSI systems [27, 14].

KL1 execution is modeled as a partially ordered set of reductions wherein the initial user query (a set of goals) is reduced to the empty set. In KL1, as in Prolog, procedures are composed of sets of clauses with the same name and arity, of the form: $H{:}-G_1,\ldots G_m | B_1,B_2,\ldots,B_n$ where H is the head of the clause, G_i are guards, "|" is the commit, and B_j are the body goals. Execution proceeds by attempting to unify a goal (the caller) and a clause head (the callee), followed by guard unification. If these unifications succeed, the procedure call "commits" to that clause (other candidate clauses are dismissed) and the input goal is reduced to the body goals in that clause. To make the KL1 goal reduction efficient, an abstract machine called KL1-B [12, 7] has been developed, which is on a similar level of abstraction to Prolog's WAM [24].

The abstract KL1 architecture can be summarized as follows [12, 17]. A goal is represented by a goal record, as in a Prolog environment [24]. Reducible goal records are stored in a goal pool. A processor fetches a goal from the goal pool and executes the compiled KL1-B code sequence corresponding to the goal, attempting to commit to one of the clauses of its procedure. If a clause is committed to, the body instructions cause body goals to be created and put into the goal pool. If no clause is committed to, but one or more clauses are waiting for some variables to be bound, the goal is *suspended*. When one of these variables is bound at some later time, the resumption routine is executed and restores the suspended goal(s) to the goal pool.

2.2 Conditional Execution Features in KL1

Dereference is required at the beginning of most unification instructions in KL1-B. In dereference, a register is first tested to see whether its content is an indirect pointer or not. If is is, the cell pointed to is fetched into the register and its data type is tested again. Unification is performed according to the data type.

Many instructions in KL1-B include run-time data type checks even after dereferencing. For example, active unification between a KL1 variable X (whose contents are unknown) and a given structure *Str* has one of four kinds of action, selected by the data type check: (1) when X is an unbound variable without suspended goals, *Str* is assigned to the variable cell; (2) when X is an unbound variable with suspended goals, these suspended goals are resumed after the assignment to X; (3) when X matches the data type of *Str*, general unification for elements of both is performed; 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. How to implement these polymorphic operations is one of the key issues in the processor design for concurrent logic programming languages. Therefore, tagged architecture is chosen as the basis of the PIM/p processor element, and tag conditional macro-call instructions are introduced to perform polymorphic operations in KL1-B, which will be discussed in Section 3.

2.3 Incremental Garbage Collection by MRB

KL1 is a concurrent language with no side effects. Destructive memory assignment is not allowed at the KL1 language level. Therefore, naive implementations of KL1 tend to consume memory area very rapidly, so that garbage collection must occur frequently. The locality of memory references is supposed to be very low during garbage collection, because most garbage collection schemes [5] walk around a wide memory area. As a result, cache misses and memory faults often occur. In sequential Prolog [24], this problem is not very serious because of the backtracking feature. However, since concurrent logic programming languages have no backtracking, an efficient garbage collection method with high memory reference locality is important in KL1 implementation.

Incremental garbage collection by multiple reference bit (MRB) [3] is introduced in KL1 architecture. MRB is a one-bit tag in a pointer to show whether the referenced data object may be referenced from other data objects (*on-MRB*) or not (*off-MRB*). When a pointer to a data object has *off-MRB*, the corresponding memory area can be reclaimed after its contents have been read, because there will be no other paths to the data. The reclaimed memory area is usually linked to free-lists for reallocation. As an optimization, the reclaimed memory area can be reused immediately with its contents.

The contents of a data object are read during unification. Therefore, the KL1 compiler detects places where cells may become garbage, and inserts garbage collection instructions at appropriate places. Unification in KL1 may produce a chain of variable cells containing indirect pointers. These indirection cells with *off-MRB* can be reclaimed during dereferencing.

The locality of memory references can be raised by using MRB incremental garbage collection instead of allocating new areas at completely irrelevant addresses, because memory areas that have recently been read are likely to be reclaimed and reused. The MRB is also used to implement constant time stream merging and array updating in KL1 programs. For example, an array element can be destructively updated without destroying the logic programming semantics when the array is referenced by an *off-MRB* pointer. These features are very important in making a general-purpose programming language.

MRB information maintenance and incremental memory management include conditional execution with bit manipulation. This is a costly operation for conventional machines, because MRB information has to be maintained in each unification. Therefore, the MRB scheme in KL1 architecture requires low-level architectural support.

3. CPU Architecture Design

As discussed in Section 2, KL1 has some features that are difficult to implement efficiently on conventional computers. These include polymorphic operations and incremental garbage collection. In this section, the key issues in CPU architecture design tailored to KL1 are discussed.

3.1 Alternatives to KL1-B implementation

Unifications include polymorphic operations for a variable cell whose type is not known until run-time. In addition, the incremental garbage collection by MRB is embedded in dereferencing. Therefore, tagged architecture is vital to the efficient implementation of KL1.

Most Prolog machines, such as PSI [15], have been implemented as high-level instruction set computers with microprograms, that is, with WAM [24] interpretation by microprograms. However, KL1-B interpretation by microprograms has the following disadvantages. First, it is difficult to make full use of micro-instruction fields, because the actions of each KL1-B instruction are determined by run-time data type checks, as explained in Section 2.2. Next, the data type check often proceeds to the next instruction without any operations or with just a simple operation. Therefore, when each KL1-B instruction is interpreted by a microprogram, the cost of dispatching to a microprogram from a fetched instruction will be relatively large.

Tagged architecture has recently been incorporated into reduced instruction set computers (RISC) [11], taking advantage of compile-time optimization and low cost in hardware design. However, this architecture has the following disadvantages in KL1 implementation.

When KL1-B instructions are expanded by low-level RISC instructions, the static code size of compiled programs will be very large. In addition, these compiled programs may include many branch instructions. This is because most KL1-B instructions involve polymorphic operations. As a result, instruction cache misses often occur and common bus traffic may increase in tightly-coupled multiprocessors with local coherent cache [2], such as a PIM/p cluster (see Section 4). Software simulation by Matsumoto et al. [13] found that the compiled code of the original KL1-B code, when expanded two times and four times, caused 15% and 70% increases in the common bus traffic of a PIM/p cluster, so that the total performance of a cluster will be degraded by 5% and 30% [10]. Certainly, high-level instruction set computers with microprograms are advantageous for reducing common bus traffic.

The compiled programs in RISC-like instructions can, of course, be kept small by using small conditional subroutine calls. However, subroutine calls have a high cost in conventional methods. Therefore, to use only the best features of both RISC and high-level instruction computers, we aimed to design a processor that facilitates an efficient conditional subroutine call function on data tag, accompanied by a RISC-like instruction set. As a result, the PIM/p processor element instruction set includes RISC-like instructions and an efficient one-level subroutine call function by tag condition. These are presented in the following subsections.

3.2 Basic Instructions

The processor element of PIM/p has two kinds of instructions, external and internal. *External instructions* are mainly used to represent the compiled codes of user programs, while *internal instructions* are used to define high-level instructions, as described later in Section 3.4. Most of these instructions are RISC-like instructions, in

Table 1 Form of Basic Instructions.

ALU instructions			
ALU-op		Rs1, Rs2, Rd	Rd←Rs1 *op* Rs2;
ALU-op		Rs1, imm, Rd	Rd←Rs1 *op* imm;
Memory access instructions			
read	*sub-op*	Rd, Ra, ofst	Rd←M[Ra+ofst];
write	*sub-op*	Rs, Ra, ofst	Rs←M[Ra+ofst];
Branch instructions			
jump (delayed jump)	*cond*	Rt, mask8, (imm8,) ofst	if condition is true, PC←PC+ofst
jump_and_link (delayed jump_and_link)		Ra, Rd, retofst, ofst	Rd←PC+retofst, PC←Ra+ofst;
skip	*cond*	Rt, mask8, (imm8)	if condition is true, skip next

the sense that they can be executed by a one-cycle pipeline. However, there are almost 100 varieties of instructions, which is more than in other RISC processors. This is because instructions for tag handling and dedicated instructions for KL1 are added, as described later.

Table 1 shows the form of basic instructions. Basically, ALU instructions have three register operands (one of the source operands may be a short immediate value). In memory access instructions, the memory location is specified by a register and an immediate offset. The sub-opcode *sub-op* can specify the transferred data width, which can be 8, 16, 32 bits, 32 bits with an 8-bit tag, or 64 bits. 64-bit data is loaded to (or stored from) two neighboring registers. As will be shown in Section 5.3, branching costs three additional cycles. Thus one-cycle delayed branch instructions and conditional skip instructions are provided.

3.3 Tagged Architecture

Taking practical KL1 implementation into consideration, 40-bit (8-bit tag+32-bit data) registers and tag branch instructions are provided in the CPU. The MRB is assigned in one of the 8-bit tags.

As discussed in Section 2.2, most unification includes a multi-way branch based on the KL1 data type. Some Prolog machines, such as the PSI [21], have a hardware-supported multi-way branch function. The processor element of PIM/p 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 have equal chances of being chosen. The PIM/p instruction set has only a two-way tag condition in macro-call instructions and in tag branch instructions, but various tag conditions can be specified in the instructions, as follows.

The tag conditions can be specified as bit-wise logical operations between the tag of a register Rt and the 8-bit tag value imm8 in the instruction, as in Table 2. The (Not-)XOR checks whether the tag of Rt matches imm8. In addition to these exact matching conditions, tag conditions are provided to examine only specified bits in the tag of a register. The mask8 value is used to

Table 2 Tag conditions in the PIM/p processor instructions.

XOR, Not-XOR	tag(Rt)=imm8, or not
OR, Not-OR	tag(Rt) | mask8=all 1, or not
AND, Not-AND	tag(Rt) & mask8=all 0, or not
XORmask, Not-XORmask	(tag(Rt) & mask 8)=imm8 or not

specify the bit field in the tag of Rt. The (Not-)OR condition examines whether the specified bits are all one, while the (Not-)AND examines whether they are all zero. The (Not-)XORmask examines whether the specified bits match imm8. By these tag conditions, various groups of data types, as well as the combination with MRB, can be specified in two-way branch instructions, such as jump, skip, and macro-call instructions.

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 tags of dedicated registers. Therefore, these flags can also be examined in the same way as KL1 data types.

3.4 Macro-call Function

A macro-call instruction can be regarded as a *lightweight* subroutine call with tag conditions, whose form is

MCall cond, R1, R2/imm8, R3/imm8, i-Addr

where i-Addr is the entry address of the internal instruction memory, and R1, R2, and R3 are the register numbers. R2 and R3 can be 8-bit immediate values (imm8). The macro-call instruction first tests the data type of a register, given as its operand R1, then invokes or does not invoke its macro-body in the internal instruction memory (IIM), depending on the result of the test. The contents of these registers, as well as the immediate values, can be accessed through *indirect access registers* and *indirect value registers* in the macro-body, as described later.

The macro-bodies stored in the internal instruction memory are written in *internal instructions* by system designers. Here, most external and internal instructions are held in common. Therefore, system designers can easily specify a high-level instruction, using one kind of RISC-like instructions instead of the complicated micro-instructions used in conventional computers. In view of the difficulty of making full use of long micro-instructions, this scheme is advantageous to system designers. In addition, the specification of a high-level instruction is very flexible, because a macro-body can include subroutine calls in external instructions stored in main (shared) memory, as well as subroutine calls in the internal instruction memory.

One of the overheads in usual subroutine calls is the branching cost both for call and return. As described in Section 5.3, the tag condition for macro-call is tested at the second stage of a four-stage pipeline. When the condition is true, the program counter for external instructions is frozen, and the execution stream is switched to the internal instructions by putting entry address (i-Addr) in the internal program counter. Therefore, the cost of invoking macro-body is only one additional cycle, while usual jump instructions cost three additional cycles to take the branch. The cost of returning from the macro-body is also minimized, as follows. 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. When the internal instruction with *eoi* is put into the pipeline, the external instruction follows without branching costs, melting the external program counter.

Another overhead is the cost of the arguments passing to and from the subroutine bodies. To avoid these costs, two kinds of virtual registers are provided. *Indirect value registers* are used to get the operand of the macro-call instruction as an immediate value, and *indirect access registers* are used to access the contents of the register that is specified in the macro-call operand. Each of these virtual registers corresponds to the operand position of the macro-call instruction. Therefore, the arguments of macro-call can be efficiently passed to and from its macro-body.

3.5 Support for Dereferencing and MRB Garbage Collection

As explained in Section 2.3, garbage collection support is one of the most important issues in parallel inference machines. The PIM/p instruction set includes several instructions tailored to MRB garbage collection.

In MRB incremental garbage collection, each variable cell or structure is allocated from a free list. When reclaimed, its memory area is linked to a free list. To support these free list operations, the Push and Pop instructions listed in Table 3 are provided. Push links a cell to the free list, and Pop allocates it from the free list, in one machine cycle. PushTag and PopTag put a new tag into the register. For example, allocation of a list cell referenced by "LIST" tag can be done by one instruction:

```
PopTag Rd, Ra, ofst, LIST
```

The MRB of each pointer and data object has to be maintained correctly in all unification instructions. Here, the most primitive operation is MRB maintenance during dereferencing. In dereferencing the MRB of the dereferenced result should be *off* if and only if MRBs of both the pointer and the cell are *off*. In this case, the indirect word cell can be reclaimed immediately, because the indirect word cell has no other reference paths to it. Two dedicated instructions, ReadOrMRB and Deref, support this operation. ReadOrMRB accumulates both the address register's MRB and the destination register's MRB, then sets the result in the destination register. Deref performs MRB accumulation along with the Pop operation. This means that

Table 3 Instructions for dereferencing and MRB garbage collectic

Intruction	*Operands*	*Comment*
Push	Rs, Ra, ofst	M[Ra+ofst]←Rs, Rs←Ra;
PushTag	Rs, Ra, ofst, imm8	M[Ra+ofst]←Rs, data(Rs)←data(Ra), tag(Ra)←imm8;
Pop	Rd, Ra, ofst	Rd←Ra, Ra←M[Ra+ofst];
PopTag	Rd, Ra, ofst, imm8	Rd←Ra, tag(Rd)←imm8, Ra←M[Ra+ofst];
ReadOrMRB	Rd, Ra, ofst	Rd←M[Ra+ofst], mrb(Rd)←mrb(Ra)|mrb(old Rd);
Deref	Rd, Ra, ofst	Rd←Ra, Ra←M[Ra+ofst], mrb(Ra)←mrb(Ra)|mrb(old Ra);

```
Deref Reg, Ptr
```

saves the pointer Reg to the dereferenced cell to anoth register, Ptr, then reads the contents into Reg wi MRB accumulation. Therefore, succeeding instructio can reclaim the cell referenced by Ptr by examining t MRB of Reg.

These instructions can minimize the costs of free-l operations and dereferencing with MRB manageme in PIM/p.

4. Cache Architecture Design

PIM/p has a hierarchical structure, as shown in Fi 1. Eight processor elements (PEs) form a cluste communicating through shared memory (SM) over common bus. Processor elements within each clust share one address space, so that they can quickly cor municate by reading or writing shared memor However, KL1 programs require high memory ban width, because data structure manipulations domina whole computation rather than arithmetic comput tion. Therefore, we optimized local coherent cache f the memory access characteristics of KL1.

4.1 Motivations for Cache Design

As explained in Section 2.1, a processor executes go reductions of a relatively small granularity (compare with those in procedural languages). Thus, focusing c the parallel processing within a PIM/p cluster, the are significant differences in KL1 memory referencir characteristics from the characteristics of convention multiprocessor systems such as Symmetry [18].

First, the frequency of memory write is higher than i conventional languages. Memory access characteristi of KL1 benchmarks, gathered by simulation, indica that the data write frequency is 36% [8].

Next, the processors communicate more often wi each other, through the logical variables, than the usu parallel processing on the Symmetry system, becau parallel goals share logical variables. In addition, con munication is necessary for scheduling KL1 goals. Th it is important for a locally parallel cache to have an efficient cache-to-cache data transfer mechanism as well as to work as a shared global memory cache.

Finally, there are many exclusive accesses to communicate through shared logical variables. The frequency of locking shared data in the KL1 execution is relatively high. However, we can expect that exclusive memory accesses seldom conflict with each other [17].

4.2 Cache Protocol

Copyback cache protocols have been proved effective for reducing common bus traffic in shared-memory multiprocessors for procedural languages, as shown by Goodman [6] and Archibald [1], among others. Thus the basis for the PIM/p cache is a copyback protocol. Local coherent cache protocols [e.g. 1, 2, 16, 19] use both invalidation and broadcast to ensure that all caches are consistent. Invalidation reduces common bus traffic when the frequency of shared block write accesses is low, while broadcast is better when many processors frequently write data to the same shared blocks [1]. In view of the single-assignment feature of KL1, most logical variables are shared by only two KL1 goals. Thus broadcasting is not necessary for most programs, and invalidation suffices.

4.3 Local Coherent Cache Optimized for KL1

The PIM/p cache protocol is similar to the Illinois protocol [16], but has several memory operations optimized for KL1 as listed in Table 4.

In normal write operations, a fetch-on-write strategy is used. However, it is not necessary to fetch the contents of shared global memory when a new cache block is allocated for a new data structure. For example, in KL1, new data structures are created dynamically on the top of the heap area when the free lists for those structures are empty. To accomplish this, the direct-write instruction is introduced to avoid useless swap-in from shared memory. The *direct-write* instruction can also be useful for stack pushing operations in WAM-based architectures.

In KL1 parallel architectures, interprocessor communication (such as for goal distribution) uses a shared message buffer. In this case, swap-in and swap-out of meaningless data can be avoided by invalidating the sender's cache block after a cache-to-cache transfer and by purging the receiver's cache block after the receiver finishes reading. To accomplish this, the exclusive-read, read-invalidate, and read-purge instructions are introduced.

These new memory access instructions can reduce common bus traffic by avoiding useless swap-in and swap-out operations. Cache simulations [8] indicate that these optimizations reduce bus traffic by 40–50% with respect to an unoptimized system. Direct write affords a 35–45% reduction and other optimizations only a 5% reduction. From the evaluation in Tick [22], we believe these optimizations will prove effective on other parallel logic programming architectures as well.

Table 4 Optimized memory access instructions.

Instruction	*Operation*
read_invalidate	After cache misses, the source cache block is invalidated. Otherwise, the same as Read.
read_purge	After CPU reads, the cache block is purged. The shared blocks in other caches are also purged.
exclusive_read	For the last word in a cache block, same as Read_Purge. Otherwise, the same as Read_Invalidate.
direct_write	If cache misses at block boundary, write data into cache without fetching from memory. Otherwise, ordinary memory write.
lock_read	Lock a memory word, then read the content.
write_unlock	Memory write, followed by unlock.
unlock	Unlock a memory word.

Lock operations are essential in shared global memory architectures. The KL1 language processor uses lock operations for heap and communication area accesses [17]. The frequency of locking and unlocking shared data is high. The simulation result in Goto et al. [8] shows more than 5% of all memory accesses. However, actual lock conflicts seldom occur [17]. Therefore, it is effective to introduce a hardware lock mechanism that has a lower overhead when there are no lock conflicts.

The PIM/p cache allows a *lightweight* lock and unlock operation by using the cache block status, lock address registers, and busy-wait locking scheme. When the CPU issues a lock command to its cache to attempt a lock-read instruction, the cache checks the corresponding address tag and status tag. If the address hits and its status is *exclusive*, the address can be locked without using the common bus. The locked address is held in a lock address register. When another processor attempts to access the locked address, the access itself is automatically postponed until the address is unlocked. This lock protocol is effective for reducing the bus traffic of lock/unlock operations: for KL1, no bus cycles are needed for the high percentage of lock reads hitting in exclusive blocks and unlocks to non-waiting locks.

5. Processor Element Implementation

A PIM/p processor element will be implement on a single board, which includes the CPU, internal instruction memory (IIM), cache system, and two co-processors: a network interface unit (NIU) and a floating point processor unit (FPU), as shown in Figure 2. The target of the basic machine cycle is 50 nanoseconds. The LSIs are now being fabricated by CMOS standard cell technology that can include up to 80K gates.

PIM/p has a 4G-byte global virtual address space on each cluster. KL1 data is represented by a 40-bit word (an 8-bit tag and 32-bit data). Normal KL1 data is placed by 40-bit KL1 tagged data in aligned 64-bit words in

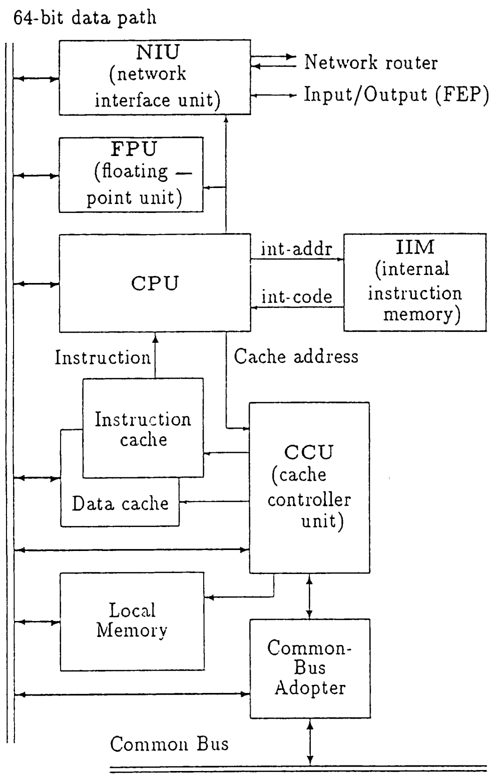

Fig. 2 PIM/p Processor Element Configuration.

the PIM/p memory system, while instructions and some data structures, such as strings or floating point numbers, are placed on a byte boundary.

5.1 Cache System

The processor element includes two caches: an instruction cache and a data cache. The instruction cache supplies the instruction buffer in the CPU with an external instruction stream in parallel with data accesses by the CPU. The contents of both cache memories are identical, so that, in a branch instruction, the CPU can fetch a branch target instruction from the data cache as shown in Section 5.3.

The cache controller unit (CCU) manages the instruction cache and the data cache. The cache address array would be updated by both commands from both the CPU and a common bus. To avoid the access conflict, the common bus adopter has a copy of the cache address array with cache block status.

In general, a larger cache is necessary to maintain a high hit-ratio. The simulation in Goto et al. [8] shows that a capacity of at least 64K bytes is necessary for KL1. However, it is preferable not to attempt to form a large cache by enlarging the cache block size, since the simulation results [8] also show that a cache block larger than four tagged words causes an increase in shared blocks between caches in parallel execution of KL1, so that mutual cache invalidation may increase. On the other hand, the results show that it is difficult to provide an address array for 64K bytes of 32-byte blocks (four tagged words), because the size of the cache address array is restricted by the LSI capacity of the cache controller unit (CCU). In view of these observations, we designed the following cache system. The capacity of both the instruction and data caches is 64K bytes. The CCU has a block status tag for each 32-byte block, and an address tag for each two blocks, that is, every 64 bytes. Our simulation result also shows that this scheme does not decrease the performance so much as a full 32-byte block cache of the same capacity.

5.2 Registers

The CPU in the processor element includes 32 general-purpose registers, several dedicated registers, indirect value registers, and indirect access registers (see Section 3.4). These registers are specified by a 6-bit register specifier in most instructions. Each general-purpose register has an 8-bit tag and 32-bit data.

The dedicated registers include a condition code register and a slit-check register (see Section 5.4). Most flags, such as the condition code, are placed in the tag part of the dedicated registers, and can be tested by the tag-branch instruction.

In addition to the above registers, NIU and FPU have several co-processor registers, which are handled only by co-processor interface instructions.

5.3 CPU Execution Pipeline

The CPU has two instruction streams, one from the instruction cache, and the other from the internal instruction memory (IIM). The CPU uses an instruction buffer and a fourstage pipeline, DATB, to attempt to issue and complete an external instruction every cycle. External instructions are either four or six bytes long and therefore the instruction buffer has a hardware aligner. Each internal instruction requires two additional stages before stage D, to set the internal instruction address (stage S) and to fetch the instruction (stage C).

Table 5 shows the pipeline stages and their corresponding operations. General-purpose registers are updated only in the last 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

Table 5 Pipeline Stages and Their Operations.

ALU operation	*Memory access*	*Branch*
D Decode	Decode/ register read (address)	Decode/ register read (address)
A —	Operand address calculation	Branch address calculation
T Register read	Cache address access	Cache address access
B ALU operation/ register write	Cache data access/ (register write)	Cache data access/ condition test

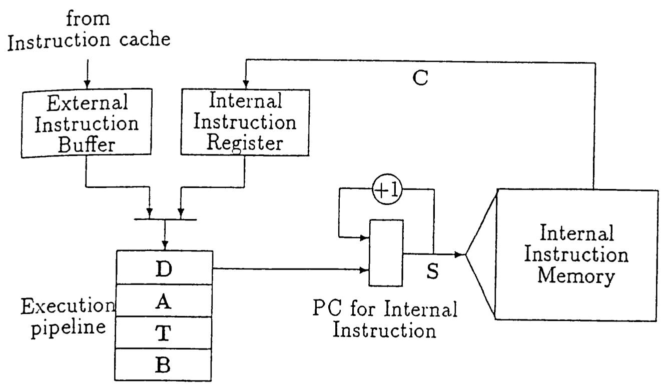
from
Instruction cache
External Instruction Buffer
Internal Instruction Register
C
+1
D
A
T
B
S
Execution pipeline
PC for Internal Instruction
Internal Instruction Memory

Fig. 3 Macro-call Instruction Mechanism.

not yet been written to the general registers.

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 one of the three cycles by executing an effective instruction.

Most tag branch instructions test their condition at stage B. However, macro-call instructions and internal branch instructions test their condition at stage A. Figure 3 shows the invocation mechanism of the macro-call instruction, and Figure 4 shows their pipelining features. A macro-call instruction puts the entry address in the program counter for internal instructions and initiates the internal instruction fetch (stage S) at stage D, then tests its condition at stage A. When the condition is true, the program counter for external instructions is frozen at this point, cancelling the next external instruction. Therefore, it costs only one additional cycle for a macro-call instruction to invoke a subroutine in the internal instruction memory. In addition, delayed macro-call instructions are provided to avoid the penalty. The return from macro-call, that is, the return from internal instructions to external instructions, can be indicated by a one-bit flag, *eoi*, in each internal instruction except for branch instructions. When an internal instruction with *eoi* is put into the pipeline, the instruction stream is switched back to external at stage D, and the external instruction frozen by the previous macro-call instruction follows without waiting cycles (see Fig. 4).

```
When the condition is true:
D  A  (condition test at A)  : macro-call instruction
   D  (cancelled)            : next external instruction
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  (condition test at A)  : macro-call instruction
   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 eoi
   S  C  (cancelled)         : internal instruction
      S  (cancelled)         : internal instruction
         D  A  T  B          : next external instruction
```

Fig. 4 Pipelining Features of Macro-call Instruction.

5.4 Slit-check and Interrupt

Various events may arise asynchronously during KL1 execution; for example, other processors may require a garbage collection of shared memory. However, the actions corresponding to these events are delayed until a current goal reduction finishes, even if the event occurred during a goal reduction. This is because garbage collection is difficult to start during a goal reduction. Therefore, the actions may be delayed until after the goal reduction is finished. The detection of these events at the end of goal reduction is called *slit-checking*.

The processor element of PIM/p incorporates a hardware mechanism for *slit-checking*, as well as ordinary interrupts for debugging and error detections. A hardware interrupt, in general, causes the program status to be saved automatically, whereas slit-checking does not. Each processor element has flag registers, each of which can keep an individual event, such as a signal from another processor or a network packet arrival. The slit-checking mechanism has an additional flag to show whether any event has happened or not, this can be tested by one conditional branch instruction. Therefore, the KL1 language processor can detect normal but asynchronous events by itself at an appropriate point. On general-purpose computers, the slit-checking might be implemented by using normal interrupt mask/unmask operations and a cumbersome interrupt handler. It would cost to much for the KL1 system. By incorporating the hardware slit-checking mechanism, the processor element can avoid frequent mask/unmask operations and interrupt handling overhead.

6. Conclusion

This paper described the design of the processor element architecture for parallel inference machine prototype, PIM/p. The execution features of the concurrent logic programming language, KL1,were observed, and its architectural issues were discussed. The innovative processor architecture for KL1 was presented, together with the decisions related to its design. The processor is designed on the basis of tagged architecture. With the variety of tag handling operations, instructions can be executed by a one-cycle pipeline. Macro-call instructions are introduced to allow a lightweight subroutine call function for polymorphic operations in unification, so that system designers can easily define high-level instructions. Dedicated instructions are introduced to support incremental garbage collection embedded in KL1 unifications. The design includes local coherent cache and optimized memory operations tailored to the memory access characteristics of KL1, which can reduce the common bus traffic within shared memory multiprocessors. These features incorporated in the processor architecture can be expected to suit other concurrent logic programming languages. The LSIs are now being fabricated by CMOS technology.

Acknowledgement

We wish to thank all the PIM researchers at both ICOT and Fujitsu Limited. In particular, we thank Mr. S. Arai and Mr. A.Asato of Fujitsu for their useful comments. We also wish to thank Mr. H. Murano of Fujitsu for his help in developing the LSIs and for his useful comments. Finally, we would like to thank the 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. Hahashi, for their valuable suggestions and guidance.

References

1. Archibald, J. and Baser, J. Cache coherence protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transaction of Computer Systems*, **4** (4) (1986), 273-298.
2. Bitar, P. and Despain, A. M. Multiprocessor Cache Synchronization. In *Proc. of the 13th Annual International Symposium on Computer Architecture*, (June 1986), 424-433.
3. Chikayama, A. and Kimura, Y. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming* (1987), 276-293.
4. Chikayama, T., Sato, H. and Miyazaki, T. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the International Conference on Fifth Generation Computer Systems* 1988,Tokyo, November (1988).
5. Cohen, J. Garbage Collection of Linked Data Structures. *ACM Computing Surveys,* **13** (3) (Sept. 1981), 341-367.
6. Goodman, J. R. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc. of the 10th Annual International Symposium on Computer Architecture* (1983), 124-131.
7. Goto, A. et al. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. of the International Conference On Fifth Generation Computer Systems* 1988, Tokyo, Japan, November (1988), 298-229.
8. Goto, A., Matsumoto, A. and Tick, E. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel (May 1989), 25-33.
9. Goto, A. and Uchida, S. Toward a High Performance Parallel Inference Machime—the Intermediate Stage Plan of PIM. In *Future Parallel Computers*, 299-320. LNCS 272, Springer-Verlag, Pisa, Italy, (1986).
10. Hattori, A., Shinogi, T., Kumon, K. and Goto, A. Architecture of Parallel Inference Machine: PIM/p. In *JSPP '89,* IPSJ, Feb. (1989), 107-114. (in Japanese).
11. Hill, M. et al. Design decisions in SPUR. *IEEE Computer*, **19** (11) (November 1986), 8-24.
12. Kimura, Y. and Chikayama, T. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming* (1987), 468-477.
13. Matsumoto, A. et al. Locally Parallel Cache Designed Based on KL1 Memory Access Characterestics. TR 327, ICOT, 1987.
14. Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Chikayama, T. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, Lisboa (June 1989), 436-451.
15. Nakashima, H. and Nakajima, K. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, San Francisco (1987), 104-113.
16. Paramarcos, M. S. and Patel, J. H. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* (1984), 348-354.
17. Sato, M. et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceedings of the Fourth International Conference on Logic Programming* (1987), 338-355.
18. Inc. Sequent Computer Systems. *Sequent Guide to Parallel Programming* (1987).
19. Stewart, L. C. et al. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, **37** (8), August (1988).
20. Takeda, Y. et al. A Load Balancing Mechanism for Large-Scale Multiprocessor Systems and Its Implementation. In *Proc. of the International Conference On Fifth Generation Computer Systems 1988*, Tokyo, November (1988).
21. Taki, K. et al. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo (1984), 398-409.
22. Tick, E. Performance of Parallel Logic Programming Architectures. TR **421**, ICOT (1988).
23. Ueda, K. Guarded Horn Clauses. In E. Y. Shapiro, (ed.), *Concurrent Prolog-Collected Papers*, MIT Press (1987), 140-156.
24. Warren, D. H. D. An Abstract Prolog Instruction Set. Technical Note **309**, Artificial Intelligence Center, SRI (1983).

(Received September 4, 1989)