ICOT Technical Report: TR-676

TR-676

Exploiting Fine Grain Parallelism in Logic Programming on a Parallel Inference Machine

by
M. Sato, K. Kato, K. Takeda & T. Oohara (Oki)

August, 1991

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Exploiting Fine Grain Parallelism in Logic Programming on a Parallel Inference Machine

Masatoshi SATO, Kenji KATO, Koichi TAKEDA, Teruhiko OOHARA
Oki Electric Industry Co., Ltd.
masatosi@okilab.oki.co.jp

Abstract

Parallel inference machine (PIM) systems are being developed in the Japanese FGCS project. A PIM pilot machine, PIM/i, is now being developed. This paper begins by describing the main characteristics of the target language KL1, which is a concurrent logic programming language. The PIM/i processor architecture which is an amalgamation of RISC, tagged architecture, pipelining and LIW approaches is proposed. This amalgamation is tuned to KL1's main characteristics such as high frequency of branches. By this amalgamation, the PIM/i processor can exploit both temporal and spatial parallelisms from the execution of the target language KL1. A language system and a simulator for this architecture have been implemented and some simulation results are reported in this paper.

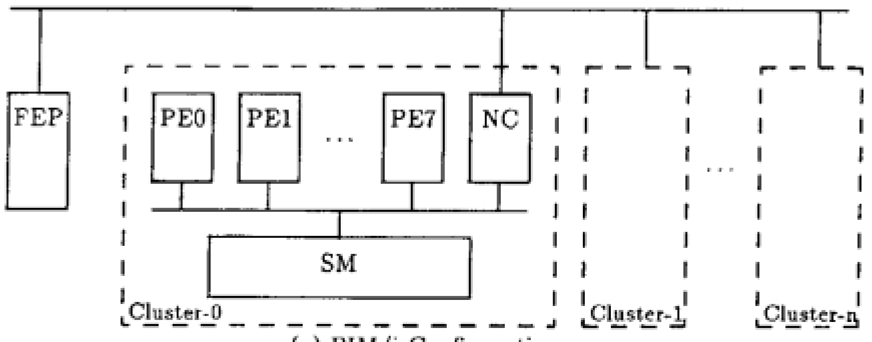
1 Introduction

As part of the FGCS project[7], we are developing parallel inference machine (PIM)[9] systems based on a logic programming framework. The PIM systems consist of the kernel language (KL1)[8] as a target language, the parallel operating system (PIMOS)[8] and the PIM hardware architectures. KL1, which was designed based on GHC[16], is a concurrent logic programming language like Parlog[1] and FCP[15]. The merit of using a concurrent logic programming language is in its implicit concurrency feature. Without explicitly specifying in the program, concurrency of the program is exploited automatically.

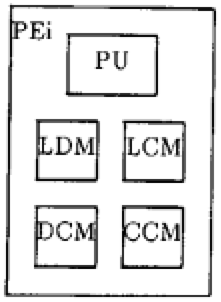
Several hardware architectures are now being developed for the PIM machine. The PIM/i is one of the PIM pilot machines. The PIM/i has a hierarchical structure with a cluster concept[6], as shown in Figure 1. Each cluster consists of eight processing elements (PEs) which communicate through shared memory (SM) over a common bus. The clusters are connected by a network through a network controller (NC). Each PE includes a processing unit (PU), two caches (a Code Cache Memory (CCM) and a Data Cache Memory (DCM)) and two local memories (a Local Code Memory (LCM) and a Local Data Memory (LDM)).

Our PIM/i design goal is to realize very high execution performance for a knowledge information processing system. To achieve this goal, it is essential to extract parallelism both in coarse and fine grain. In coarse grain parallelism, we have studied the KL1 execution model for tightly coupled multiprocessor[6, 14]. The major KL1 implementation issues are scheduling the processes on the available processors at run time, load balancing, synchronization between processors, memory contention when processors share data, and communication delays between processors. Coherent cache[13, 5], which has been optimized for KL1, plays an important role for the extracting of this parallelism.

This paper focuses on the processor architecture which is enhanced to exploit fine grain parallelism. The



(a) PIM/i Configuration



(b) PE Configuration

Figure 1: PIM/i Configuration

PIM/i processor architecture proposes four kinds of approaches for extracting instruction-level parallelism of logic programs. RISC, tagged architecture, pipelining and LIW approaches are amalgamated and balanced in the PIM/i processor. In other logic programming language, the VLIW[4] approach was adopted in parallel execution of Prolog[3] and RISC approach in that of FCP[10].

In Section 2 of this paper we will describe a brief overview of the main characteristics of KL1. The processor architectures for extracting fine grain parallelism of KL1 are discussed in Section 3. This section presents the instruction set and the configuration of PU. The process of translating KL1 programs into PIM/i machine code is outlined in Section 4, and in Section 5 we evaluate the PIM/i processor architecture, based on simulation results. Finally, in Section 6 we discuss our conclusions and the status of the project.

2 KL1 and KL1b

2.1 Target Language : KL1

KL1 is a parallel logic programming language based on GHC. A KL1 program is a finite set of guarded Horn clauses of the following form:

$H : -G1, ..., Gm|B1, ..., Bn.(m \geq 0, n \geq 0)$

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

Reduction of a KL1 program, like that of a Prolog program, proceeds by reducing a given goal clause to the empty clause. This reduction proceeds by attempting unification between a given goal and a clause head. If unification succeeds, reduction of the guard goals are attempted. If the guard succeeds, the reduction "commits" to that clause. After that, body goals are reduced.

The major differences between KL1 and Prolog are: (1) KL1 has no backtrack mechanism; (2) the execution of KL1 is done by the heap base operation not the stack base operation; (3) unification in the passive parts has the synchronization mechanism in KL1, i.e., it is restricted to be input unification only. If output unification is attempted to the caller's variable in passive parts, the call potentially suspends until another process binds that variable.

2.2 Behavior of KL1b

Clauses in KL1 programs are compiled into a sequence KL1b[12] instructions just as Prolog is compiled to the WAM[17]. A KL1b code for a set of candidate clauses includes passive unification instructions for head and guard parts, active unification instructions, argument preparation instructions and goal fork instructions for the body part.

Typical passive unification instructions begin with dereferencing of argument. If it is not instantiated, the test for this clause is abandoned. Then execution proceeds to the next candidate clause. If no clause is selected and there are variables that cause the suspension, the current goal is linked to these variables, setting the tag of the variable by HOOK, to realize a non-busy waiting synchronization mechanism.

If a clause is selected, the body part of that clause is executed. Active unification instructions are performed for four kinds of actions depending on the argument data type: (1) in case of an uninstantiated variable without suspended goals, the value of the argument is assigned into that variable; (2) in case of an uninstantiated variable with suspended goals, these suspended goals are resumed before assignment; (3) in case of an instantiated cell with a target tag, the unification is performed; and (4) otherwise, the unification fails. Several goal fork instructions are provided to push and pop a goal context to and from a ready-goal-stack.

Consequently, most instructions in KL1b include run-time data type checks and branches for the preceding actions.

3 Architecture for KL1

3.1 Design decisions

The characteristics of our target language KL1 can be summarized in that the branch frequency is very high, especially a conditional branch by data type. From these characteristics, the PIM/i processor combines several new architectural techniques into a single design to reduce the branch penalties.

1. RISC Architecture

   RISC approach has many possibilities to optimize the codes in their generations by the following architectural techniques. RISC architecture also takes advantages of VLSI implementations. VLSI is essential for practical and large scale parallel systems. Our RISC approach is based on the following design principles. (1) An instruction is reduced to a simple instruction (2) which can be executed in a single machine cycle at the peak execution rate. (3) Only reads and writes access memory; all others perform register-register operations. (4) High-performance processor requires care in the design of the entire memory hierarchy. We enhance the memory system to support local memories and caches which are split into for instruction and data.

2. Tagged Architecture

   Tagged instructions are useful for implementing KL1, because the data type checks and the value calculations can be executed by each of the operation units in one cycle. 40 bits are assigned for a tagged word. In the data, eight bits are used for a tag and 32 bits for a value. All instructions are 40 bits long corresponding with the data size.

3. Pipelining

   To make convenient use of hardware parallelism, we introduce pipelining. However, the instruction set must be carefully tuned to support pipelining. We adopt a three-stage pipeline not only to control pipeline stalls, but also to reduce pipeline branch penalties at the dynamic behavior of branches. The three stages are:

(a) F (instruction fetch) Stage.

(b) E (execution) Stage,
including instruction decode, register fetch, execution and effective address calculation.

(c) W (write back) Stage.

Our three-stage pipeline can be explained in comparison with a five-stage pipeline (IF:instruction fetch, ID:instruction decode and register fetch, EX:execution and effective address calculation, MEM:memory access, WB:write back) which is discussed in [11].

There are two differences between our three-stage pipeline and the five-stage pipeline. First is the MEM stage. In [11], the MEM stage is used for memory access or branch completion step. In PIM/i, memory access is done by the separated instructions, which are the address generation instructions and the memory access instructions, using the same E stage of a different cycle. The branches are realized by another field, which is discussed in the next item. This separation makes clean the data hazard. Another difference is both the ID and the EX stage are merged into an E stage, because the F stage takes much time to access the outside of PU chip compared with the inside operations.

4. LIW Architecture

To make use of the 40 bit instruction field effectively, we introduce an LIW architecture. LIWs use multiple, independent functional units via multiple operations packaged into one instruction. A PIM/i instruction might include a Sequencer (S) operation , a Memory (M) operation and a Processor (P) operation. These separated operations allow to be issued the processor operation and/or the memory operation at branch points. The LIW architecture is not the only way to extract low-level parallelism but is a simple and effective way to design the instruction set compared with the full collection on branch behaviors.

3.2 Instruction Set

1. Instruction Types

An instruction, which is 40 bits wide, has a set of operations, i.e., S, M and P operations. The field size of the S operation, 8, 16, or 40 bits, is altered by its operand size. The field sizes of S and M operations are fixed at 8 bits and 24 bits respectively. Figure 2 shows three kinds of instruction types.

- TYPE1 (S+M+P)

  TYPE1's S operation is an 8 bit branch operation including 6 bits of the PC-relative field. The remainder is assigned to both a M operation and a P operation.

- TYPE2 (S+P)

  TYPE2's S operation occupies 16 bits. This field includes either the condition field (8 bits) or the register field for indirect branch (5 bits). The remaining field can be used by a P operation.

- TYPE3 (S)

  TYPE3's S operation use all 40 bits, including the 40 bit address field (PC-relative or absolute).

2. S Operation

This is the operation class to change the flow of control. As mentioned instruction types, S operations have three kinds of operation length. In addition to this, there are three kinds of methods for dealing

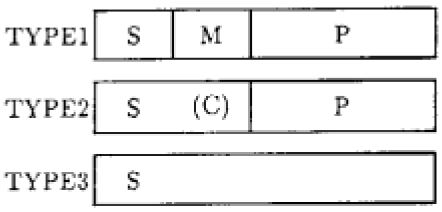


Figure 2: Instruction Types

with the pipeline stalls due to branch delay. Delayed branch instruction performs useful work for both the branch-taken and branch-not-taken paths. Normal branch instructions perform useful work for only the branch-not-taken path. Canceled delayed branch instructions perform useful work for only the branch-taken path.

3. M Operation

   Loading and storing from memory is done with the address generation operations and read/write operations. Read/write operations use the M operation field. Read/write operation might be overlapped with other P and/or S operations.

4. P Operation

   All P operations are register-register operations. The operations include integer arithmetic, logical, tag manipulation, bit manipulation, byte manipulation, priority encode.

Operations are listed in the appendix. In this list, goto or gosub shows PC-relative branch, jmp or jmpsub shows absolute branch.

3.3 Configuration of the PE

Figure 1 (b) shows the configuration of the PE. The PE includes local memories and caches for instructions and data. This configuration provides a fast, efficient memory hierarchy.

The configuration of the PU is showed in figure 3. In this figure, the horizontal axis shows three independent function units, so the PU can issue three operations in every clock cycle as in a LIW architecture. The vertical axis shows three units of pipeline stages. In the P operation of the E stage, the PU can execute the tag part and the value part simultaneously as in a tagged architecture.

4 Code Generation

To transform KL1 programs into PIM/i machine codes, current code generator has four main phases: KL1 compiler, KL1b post-compiler, macro expander, assembler/linker.

1. KL1 compiler

   KL1 programs are compiled into KL1b by the KL1 compiler in [12]. Figure 4 (a) shows a simple KL1 program to wait a atom 'a'. We outline the translating process using this example. Figure 4 (b) is a corresponding sequence of KL1b instructions.

2. KL1b post-compiler

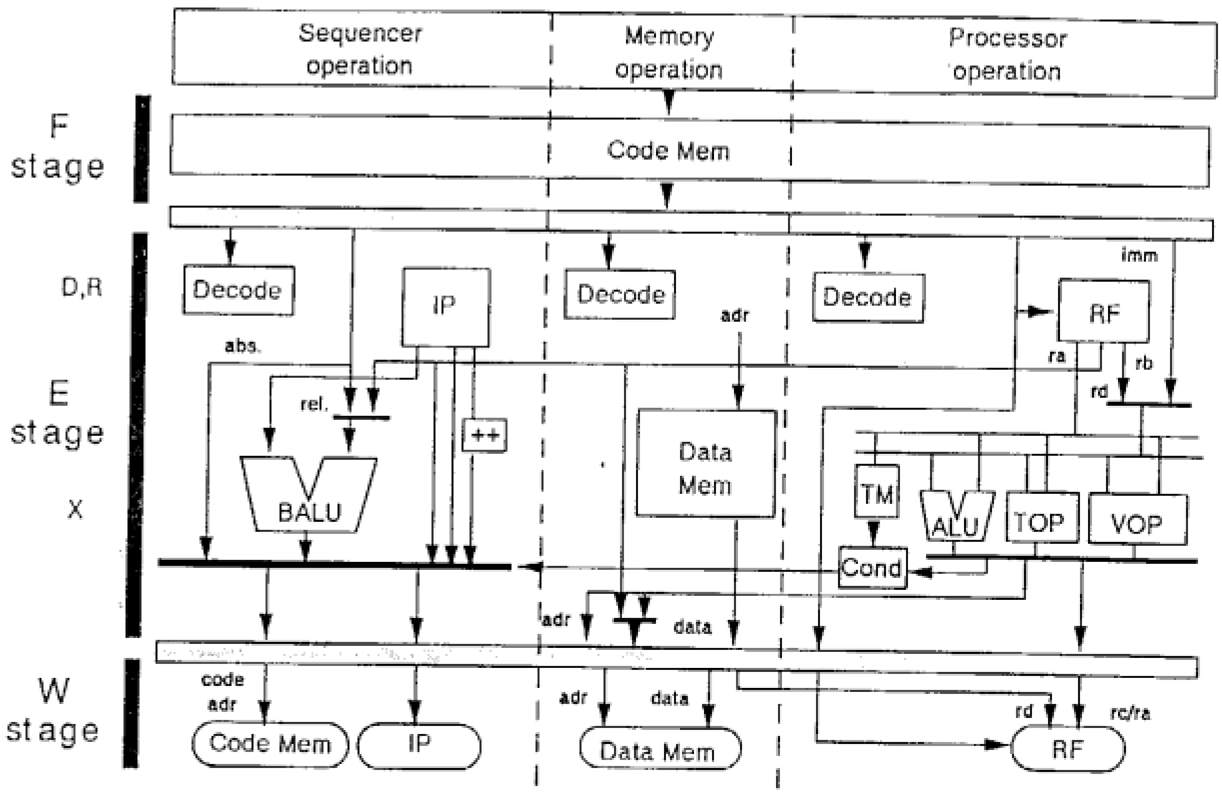


Figure 3: PU Configuration

Because KL1b is an abstract machine instruction, it is necessary to transform each KL1b into the machine dependent format. This format includes macro definition's corresponding to KL1b instructions. This transformation is done by the KL1b post-compiler. In this transformation, the KL1b post-compiler adds the label and control information and optimizes the KL1b code. In the case of Figure 4, **`try_me_else`** is eliminated by adding label information to the **`wait_constant`** code. The **`wait_constant`** is also optimized to a specific data type KL1b instruction, **`wait_atom`**. The transformed sequence is shown in (c). In this sequence, `::` shows the control information for the assembler.

3. Macro expander

The macro expander expands each machine dependent KL1b code into a sequence of PIM/i machine instructions. To use the full set of macro functions, we use cpp, the preprocessor of the C language. Currently each KL1b definition is generated from a template format to make the description of macro definitions easy. In this template, complex transactions are ejected to local memory as run-time libraries, because the naive expansion of KL1b causes an explosion of the static code size. Figure 4 (d) shows an example of template format. The information of external label, which is the entry for run-time libraries, is declared via `%i`, and argument information via `%` for register, `%c` for constant or `%l` for label.

4. Assembler and linker

The assembler/linker produce the object code file which will be allocated in the shared memory. Run-time libraries, which are allocated in local memory, are linked with this object. Currently run-time libraries are programmed separately by PIM/i assemble language.

```
go(a) :- true | true.
```

(a) KL1 program

```
procedure((go),(1)).
label((go)/(1)).
  try_me_else((go)/(1)/(1)).
  wait_constant((a),(1),(go)/(1)/(1)).
  proceed.
label((go)/(1)/(1)).
  suspend((go)/(1)).
```

(b) KL1b code sequence

```
:: listing [predicate_id,go,1].
 :: label(go_1).
:: listing [(listing),(wait_atom)].
    wait_atom(a00,a,'Instr!Label!0',go_1_1)
 :: label('Instr!Label!0').
:: listing [(listing),(proceed)].
    proceed('Instr!Label!1',none)
 :: label('Instr!Label!1').
 :: label(go_1_1).
:: listing [(listing),(suspend)].
    suspend(go_1,1,'Instr!Label!2',none)
 :: label('Instr!Label!2').
:: listing [end_procedure].
```

(c) macro format sequence

```
/*** wait_atom Ai atom ***/
%i Sub4_psv_deref_if_unbound_then_push
%  Ai            a00
%c C_ATOM        a
%{ wait_atom Ai, Atom
  gr00 := Ai,if(notREF( Ai )) goto $start.
  jmpsub Sub4_psv_deref_if_unbound_then_push.
$start:
  Ai := gr00,if(notATOM(gr00)) goto $not_atom.
  SetAtom(gr00, C_ATOM).
  gr00 := -gr00+Ai,if(zero) goto Next_inst.
$not_atom:
  goto Alter.
%} end of wait_atom
```

(d) template format

Figure 4: Example codes

Table 1: Characteristics of the Benchmarks

	qk8	qu8	prm	bup	han	av.
size(Kw)	2.9	2.9	1.1	13.5	1.0	4.1
ex.(Kcyc)	3080	6092	2518	4599	3414	3908
code $	99.99	99.99	99.99	99.92	99.99	99.66
data $	97.81	93.40	93.39	95.71	88.48	93.56

Table 2: Branch Behavior in Benchmarks (%)

	qk8	qu8	prm	bup	han	av.
br/inst	39.4	29.6	41.3	38.3	32.5	36.2
Uncond	8.6	6.3	8.8	11.3	12.6	9.5
Cond	11.6	11.9	14.7	9.5	15.5	12.7
Tag_cond	19.2	11.4	17.8	17.5	4.5	14.1

5 Evaluation

To evaluate the PIM/i architecture, a register transfer level simulator has been constructed. The simulator computes run-time statistics such as instruction frequency, pipeline utilization, branch-delay slot utilization, frequency of each operation type, and others.

The simulator simulates a single processor with code and date cache whose size is 32K words, respectively. Five benchmark measurements (two 8-queen programs (qk8 and qu8), prime number generator program (prm), bottom up parser (bup) and towers of Hanoi program (han)) are analyzed in this paper. High-level characteristics of the benchmarks are given Table in 1. Static assemble code size, execution cycles, cache hits ratio of code and data are given. The size of the run-time libraries is 4K words, and libraries are allocated in the local memory.

5.1 Branch Behavior in Benchmarks

Since branch behavior is the starting point of our architecture, we should summarize the branch behavior in benchmarks. Table 2 shows the percentage of branch instructions per executed instructions (br/inst) and the breakdown data: that of unconditional branches (Uncond), conditional branches (Cond) and tagged conditional branches (Tag_cond).

In RISC type machines, branch frequencies tend to be lower. However, Table 2 shows 36% of the executed instructions are used for branch instructions, higher than 26% for VAX (CISC type) and 14% for DLX (RISC type) in [11]. The breakdown data shows tagged conditional branches are used very frequently. Half of branches are tagged conditional branches and they are more than that of VAX and DLX.

5.2 Reducing Pipeline Branch Penalties

Branches consume a large fraction of time because they cause pipeline stalls and pipeline flushes. Scheduling of the branch-delay slot can avoid situations where cycles are wasted on decoding instructions that are never executed. Table 3 shows percentage of the branch-delay slots in execution time (DS/T) and the wasted slots in execution time (Wasted_DS/T), the utilization of branch-delay slot. The utilizations are divided into two classes: Uncond shows that of unconditional branches, Cond shows that of conditional branches.

Table 3: Utilization of Branch-delay Slot (%)

	qk8	qu8	prm	bup	han	av.
DS/T	31.8	22.0	33.4	31.3	22.7	28.2
Wasted_DS/T	13.8	8.5	12.2	9.4	4.5	9.6
Uncond	70.1	82.3	63.7	59.4	75.7	70.2
Cond	52.7	55.9	63.3	74.3	81.2	65.5

Table 4: Utilization of instruction fields (%)

	qk8	qu8	prm	bup	han	av.
`_ _ P`	28.8	37.6	36.9	29.0	38.3	34.1
`_ M _`	11.7	10.1	7.6	9.7	5.9	9.0
`S _ _`	0.0	0.0	0.0	0.0	0.0	0.0
`_ M P`	11.5	9.7	4.4	11.2	11.8	9.7
`S _ P`	0.0	0.0	0.0	0.0	0.0	0.0
`S M _`	0.7	0.3	0.8	1.1	1.5	0.9
`S M P`	0.0	0.0	0.0	0.0	0.0	0.0
TYPE 1	52.6	57.7	49.7	50.9	57.4	53.7
`S S _`	10.8	4.1	11.8	10.7	6.4	8.8
`S S P`	28.7	33.5	28.4	29.3	27.8	29.5
TYPE 2	39.5	37.6	40.2	40.0	34.2	38.3
`S S S`	7.9	4.7	10.1	9.1	8.4	8.0
TYPE 3	7.9	4.7	10.1	9.1	8.4	8.0

This shows that only about one-third of the branch-delay slots are wasted and that they are about 10% of the execution time. We assume one null CPU cycle occurs on both a branch-taken and a branch-not-taken by relative comparison, as a worst case[2]. By the utilizations of branch-delay slot, the execution time is reduced about 16% from the worst assumption. The difference between the utilizations of branch-delay slot in unconditional branch instructions and that in conditional branch instructions is small.

5.3 Multiple issue of operations

Utilization of instruction fields shows the extracted parallelism in current KL1 system. Table 4 shows utilization of instruction fields for each type. In this table, the combination with S, M, P and _ shows the percentage of each instruction contained in each set of operations.

This table shows the most instructions, 54%, are used as in TYPE 1. The most frequently used combination is that of S and P, branch and ALU operation, which is a `S S P` in type 2. By multiple issue of operations, the execution time is reduced by about 24% from the case in which each operation is issued in each cycle.

6 Conclusion

The processor element architecture for the PIM pilot machine, PIM/i, has been presented. The PIM/i processor architecture is an amalgamation of four new architectural techniques. With the pipelining approach, it can exploit temporal fine grain parallelism. With tagged architecture and LIW approach, it can exploit spatial fine grain parallelism. With RISC approach, these approaches are naturally amalgamated into optimized codes. We have also presented the first simulation results. Our results indicate that the execution time could be reduced by about 16% from the worst assumption by utilization of branch-delay slot and about 24% from the sequential executions of each operation.

Meanwhile, a PIM/i processor is being implemented in 1.2μm CMOS VLSI. The target of the basic machine cycle is 100 nanoseconds. The design of the processor chip is completed. The PE is implemented on a single printed board using two VLSIs such as processor and cache controller. In a cluster, eight PEs and a shared memory are connected via an 80-bit wide common bus. The PIM/i has a 1T-byte global virtual address space on each cluster. The current implementation of the cluster includes 320M bytes of shared memory. We plan to precisely measure and to evaluate the PIM/i processor and system during program execution.

Acknowledgment

The authors would like to thank all of the PIM research members both at OKI Electric Industry and at the Institute for New Generation Computer Technology (ICOT). Especially we wish to thank ICOT researcher, Dr. K. Hirata, for his useful comments. Finally, we would like to thank Dr. K. Fuchi, the director of ICOT and Dr. S. Uchida, the manager of ICOT's Research Department, Dr. K. Taki, the chief of the first research section in ICOT, Mr. Y. Haga, the director of the Systems Laboratory at OKI Electric Industry, Mr. A. Yamamoto, the chief of the Parallel Processing research section at OKI Electric Industry, for giving us the opportunity to participate in this research.

References

[1] K.L. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *ACM Tran. on Programming Languages*, 8(1), 1986.

[2] J. A. DeRosa and H. M. Levy. An Evaluation of Branch Architectures. In *14th International Symposium on Computer Arch.*, Jun. 1987.

[3] K Ebcioğlu and M Kumar. A Wide Instruction Word Architecture for Parallel Execution of Logic Programs coded in BSL. In *FGCS 1988*, pages 931 – 942. ICOT, Nov. 1988.

[4] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, 1986.

[5] A. Goto et al. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *16th International Symposium on Computer Arch.*, May. 1989.

[6] M. Sato et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *The Fourth International Conference on Logic Programming*, pages 338 – 355, 1987.

[7] S. Uchida et al. Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. In *FGCS 1988*, pages 16 – 36. ICOT, Nov. 1988.

[8] T. Chikayama et al. Overview of the Parallel Inference Machine Operating System (PIMOS). In *FGCS 1988*, pages 230 – 251. ICOT, Nov. 1988.

[9] A. Goto and M. Sato et al. Overview of the Parallel Inference Machine Architecture (PIM). In *FGCS 1988*, pages 208 – 229. ICOT, Nov. 1988.

[10] A. Harsat and R. Ginosar. CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent Prolog. In *FGCS 1988*, pages 964 – 969. ICOT, Nov. 1988.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, chapter 6 Pipelining. Morgan Kaufmann Publishers Inc. San mateo, CA, 1990.

[12] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Int. Symp. on Logic Programming*, pages 468 – 477, Aug 1987.

[13] A. Matsumoto and M. Sato et al. Locally Parallel Cache Designed Based on KL1 Memory Access Characteristics. Technical Report TR - 327, ICOT, 1987.

[14] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305 – 318. NOTTH-HOLLAND, Apr. 1988.

[15] E. Shapiro. Concurrent Prolog: a Progress report. *IEEE Computer*, pages 44 - 58, Aug 1986.

[16] K. Ueda. Guarded Horn Clauses. Technical Report TR - 103, ICOT, 1985.

[17] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report TR - 309, SRI International, 1983.

Instruction Set

Sequencer Operation

S(39:32)	return_Delay3	-
	goto_Delay3	jr6 (jr6: 6 bit branch offset)
S(39:24)	goto_Delay2	rd
	jmp_Delay2	rd
	gosub_Delay2	rd
	jmpsub_Delay2	rd
	return_Delay3_Cond	-
	goto_Delay3_Cond	jr6
	merge_tag	rx, imm8 (imm8: 8 bit immediate)
S(39:0)	goto_Delay2	jr30 (jr30: 30 bit branch offset)
	jmp_Delay2	ja30 (ja30: 30 bit absolute jump address)
	gosub_Delay2	jr30
	jmpsub_Delay2	ja30
NOTE	Delay3: normal, delayed, canceled Delay2: normal, delayed Cond: tag_eq, not_tag_eq, fwd_eq(fwd:tagged full word), not_fwd_eq, eq, not_eq, not_ovf, ovf, high_same, low, high, low_same, gt, le, test_tag	

Memory Operation

M(31:24)	MemOp	rd
	LockOp	rd
	StrResOp	-
NOTE	MemOp: read, write LockOp: read_lock, write_unlock StrResOp: write, write_unlock, unlock	

Processor Operation

P(23:0)	TagOp	tag(k,l)!ra, (n,m)!rb *or* tag(k,l)!ra, imm5 ((a,b): bit field from a to b)
	ValOp	(k,l)!ra, (n,m)!rb *or* (k,l)!ra, imm5
	PriOp	ra, rb
	AOp	ra, rb, rc *or* ra, imm5, rc *or* ra, rb ≪ ix, rc *or* ra, imm5 ≪ ix, rc
	LOp	ra, rb, rc
	AdrOp	ra, rb ≪ ix , (mar,rc) | mar
		or ra, imm5 ≪ ix, (mar,rc) | mar
	AdrDbyteImmOp	ra, imm16
	DbyteImmOp	ra, imm16
NOTE	TagOp: deposit_tag, merge_tag, extract_tag, merge_tag_imm ValOp: deposit, merge, extract, merge_imm, merge_dynamic_imm, deposit_dynamic, merge_dynamic, extract_dynamic PriOp: find_first_zero, find_first_one, reverse_find_first_zero, reverse_find_first_one AOp: *arithmetic operation* LOp: *logical operation* AdrOp: set_adr, set_adr_post_modify, set_adr_pre_modify AdrDbyteImmOp: set_adr_imm DbyteImmOp: add_signed_imm,load_low_imm,merge_high_imm	