ICOT Technical Memorandum: TM-0105

TM-0105

Design of a High-speed Prolog Machine (HPM)

Ryosei Nakazaki, Akihiko Konagaya, Shin'ichi Habata,
Hideo Shimazu, Mamoru Uemura, Masahiro Yamamoto,
(NEC Corp.)
Minoru Yokota and Takashi Chikayama
(ICOT)

April, 1985

ICOT

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

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Design of a High-speed Prolog Machine (HPM)

Ryosei Nakazaki(*), Akihiko Konagaya(*), Shin'ichi Habata(*),
Hideo Shimazu(*), Mamoru Umemura(*), Masahiro Yamamoto(*),
Minoru Yokota(**) and Takashi Chikayama(**)

(*) C&C Systems Research Laboratories, NEC Corporation,
4-1-1 Miyazaki Miyamae-ku, Kawasaki 213, Japan
(**) ICOT Research Center
Institute for New Generation Computer Technology,
Mita Kokusai Bldg. 21F, 4-28 Mita 1-chome, Minato-ku, Tokyo 108, Japan

ABSTRACT

This paper describes a High-speed Prolog Machine (HPM) architecture and its hardware structure, which are developed as a product of Fifth Generation Computer System (FGCS) project in Japan. HPM realizes high performance and provides a practical programming environment. A major HPM feature is a large memory capacity and specialized hardware for unification and stack operations. HPM has a compiler-oriented architecture with high-level stack-control instructions. Furthermore, the HPM architecture provides side-effect operations, nonlocal-exit control and multiprocess support primitives, which are effective to develop system programs. The performance is estimated at 280 KLIPS (Kilo Logical Inferences Per Second) in executing a deterministic "concatenate" Prolog program. The HPM processor is implemented with high-speed CML (Current Mode Logic) chips and with 100 nanosecond machine cycle.

1. Introduction

Recently, the logic-programming language Prolog[1][2][3] has been considered to have possibilities useful in the field of Artificial Intelligence, such as natural-language processing and knowledge-based systems. Prolog is useful in AI research due to its argument matching function, automatic backtracking control and internal database-management facilities.

Realization of these features for Prolog programs requires huge working memory capacity and extensive execution time. Prolog language processors on conventional computers lack the necessary memory capacity and performance. Therefore, Prolog application programs on a conventional computer have been mainly limited to experimental use. In order to make Prolog a practical language, an attractive approach is to develop a high-level language machine[4] with both a high performance Prolog processor and a large real memory. Realizing a Prolog programming environment on the Prolog machine is also very important for developing practical application programs.

A High-speed Prolog Machine (HPM) is designed to satisfy these goals. HPM is developed as a product of Fifth Generation Computer System (FGCS) project in Japan. The FGCS project organizes a sequential inference machine (SIM)[6], which consists of a personal sequential inference machine (PSI)[5][6][7] and HPM. The PSI is suitable for relatively small-scale Prolog program execution. In the SIM, HPM provides high performance and large memory capacity to execute Prolog programs too large and too complex to perform on the PSI.

HPM was designed to be a back-end processor, connected to the PSI. HPM is highly specialized for Prolog program execution. PSI and HPM differ at the machine instruction level. The level of HPM is lower than that of PSI. Performance improvement in HPM is mainly due to compiler optimization techniques and specialized hardware modules, compared with microprogram interpretation techniques in the case of PSI. Although HPM's instruction set is different from that of PSI, most Prolog programs can be executed on both machines.

The following section describes the HPM design approach and system overview. Next, the HPM architecture background is discussed and the architecture is summarized. Then, the HPM hardware organization is described. Finally, HPM performance is estimated on a Prolog program example.

2. System Overview

The HPM goals are as follow.

(1) Realizing a high performance machine that executes Prolog language in conformity with DEC-10 Prolog[8].

(2) Providing a large main memory capacity for a large scale Prolog program.

(3) Providing a practically usable software development support tool.

In order to achieve the first goal, the authors develop a specialized Prolog processor, that is, the HPM processor, and chose the following machine design approach.

- Back-end processor configuration.
- Specialized hardware-module for Prolog-language processing.

- Compiler-oriented architecture with high level stack control instructions.

- Effective compiler with optimization techniques.

From the aspect of an end user, the HPM processor works as a back-end processor for Prolog programs, as shown in Fig.1. This system configuration concentrates the designer's efforts realizing the high performance back-end processor, because complicated input/output controls are offloaded from the HPM processor to a host computer, PSI. In this machine configuration, HPM executes large-scale Prolog programs, while PSI controls input/output operation.

HPM provides a compiler-oriented architecture with high-level stack control instructions. The HPM instructions realize argument matching functions, stack control and backtracking control that support non-deterministic control characteristic of the Prolog language. These functions are implemented with functionally specialized hardware modules and microprogrammed control. Tagged data are also incorporated. Furthermore, these instructions are useful for a compiler to optimize deterministic control written in a Prolog program. The optimized instruction sequence can substantially improve HPM processing performance and save required memory capacity, by using tail-recursion optimization, register-access optimization, and clause-indexing techniques[9][10].

In order to have a large main memory, HPM provides a 64 Mega-word(36 bits/word) main memory system, which is realized with 256 Kbit DRAM chips. The memory area is mainly consumed as working [illegible] areas.

In order to be used as a practical software development support tool, programming environments are realized on the HPM back-end processor itself. The HPM processor is designed to execute fundamental operating system (except for input/output operation), a compiler and a debugger as well as Prolog application software.

3. Architecture

This section describes the design background of HPM architecture from performance and software productivity view points, and describes a summary of the basic HPM architecture.

3.1 Performance improvement

One of our major design goals is to realize a high performance Prolog machine. In order to achieve the goal, HPM adopts a compiler oriented architecture, because a compiler can remarkably eliminate run time overhead which occurs in direct Prolog-program interpretation. HPM provides a high-level stack control instruction set proposed by Warren[9]. The features of the instruction set are as follow.

(1) Unification and backtracking control can be compiled to a sequence of register-access instructions and stack control instructions, that are of higher level than conventional instructions.

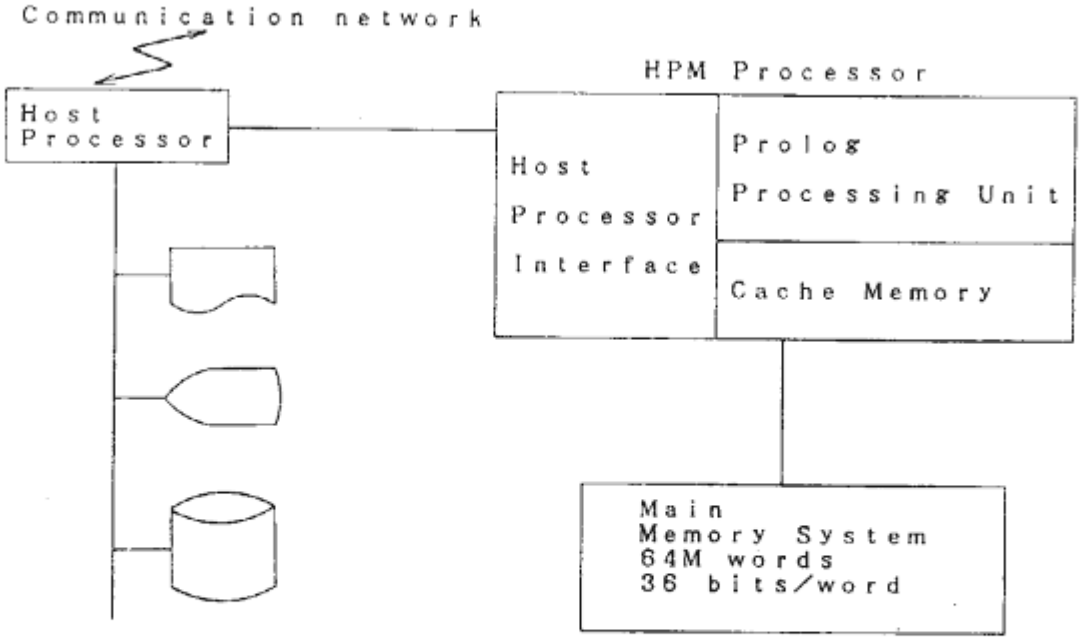

Fig. 1 HPM system configuration

(2) Optimization techniques, such as 'clause-indexing' and 'tail-recursion optimization' [11], are applicable.

Using the 'clause-indexing' technique, a nondeterministic program execution can be deterministically executed. Consider a process to find a specific clause in a procedure composed of lots of clauses. Without the 'clause-indexing' technique, an executer must scan the clauses one by one until encounters the objective clause. On the other hand, under the technique providing a hash table for finding the clause, an executer can directly find out the objective clause, and never backtracks for the other clauses. For example, suppose a Prolog program contains the clauses:

```
capital_city('Tokyo','Japan').
capital_city('Washington','USA').
capital_city('Paris','France').
        .......
```

If the first input argument is 'Washington', only the second clause is selected with the hash table and executed.

'Clause-indexing' is also applicable to the following list-processing procedure.

```
concatenate([],X,X).
concatenate([X|Y],Z,[X|YZ])
        :-concatenate(Y,Z,YZ).
```

In this case, 'clause-indexing' is achieved by providing a 'tag-branch' instruction which recognizes whether the first input argument is a variable, a nil or a list. Using the technique, 'concatenate' is executed deterministically as long as the first argument is not a variable.

Figure 2 shows HPM compiled codes for the 'concatenate' program. 'switch_on_term A0' instruction is provided for deterministic execution using clause-indexing technique. This instruction checks the tag type of the first argument within a register (A0), and branches to one of four ways, depending on the tag type; a variable, a list, a structure, or a constant (a nil). After branching, unification instructions are performed. 'get' instructions correspond to top level arguments of the clause header, and 'unify' instructions correspond to list (structure) arguments. The 'get_list' instruction is used for both matching of existing list structure, and creation of a new list structure, depending on the corresponding argument type, a list or a variable.

The codes are executed nondeterministically when the first argument within the register (A0) is a variable. 'try_me_else' instruction is used for saving the backtracking information, and 'trust_me_else_fail' instruction is used for recovering the previous state. In addition, all arguments are saved in registers (Ai, Xi indicate the register number) to achieve tail recursion optimization.

3.2 Programming environment support

The other HPM design goal is to provide a practical programming environment. On designing a Prolog machine architecture, it is an important

definition:

```
concatenate([],X,X).
concatenate([X|Y],Z,[X|YZ])
        :-concatenate(Y,Z,YZ).
```

compiled code:

```
concatenate/3:
 switch_on_term A0
 jump Lr                    variable
 jump Ll                    list
 fail                       structure
 jump Lc                    constant
Lr:
 try_me_else 3,$1                concatenate(
Lc:
 get_nil A0                      [],
 get_temporary_value A2,A1       X,X
 proceed                         ).
$1:
 trust_me_else_fail              concatenate(
Ll:
 get_list A0                     [
 unify_temporary_variable X3     X|
 unify_temporary_variable A0     Y],Z,
 get_list A2                     [
 unify_temporary_value X3        X|
 unify_temporary_variable A2     YZ]:-
 execute-relative                concatenate(Y,Z,YZ).
            concatenate/3
```

Fig. 2 Example of compiled code for concatenate/3

choice whether the programming environment is wholely written in the Prolog language or not. In order to maximize the performance of the Prolog processor, the authors chose to write all the programming environment in Prolog, including operating system functions, such as process management, memory management, input/output control. This choice requires to extend the Prolog language to provide the following system facilities.

(1) Some kinds of side-effect operations.
(2) Nonlocal-exit control for exception handling.
(3) A multiprocess environment support primitives.

A practical operating system requires a lot of state informations which are changed rapidly. Although the state transition may be implemented without the notion of side-effect, this is not a suitable way to construct a real programming environment. In order to realize fast state transition, the language provides a rewritable data structure called 'vector'. The vectors are used for various system information tables such as process control blocks, file control blocks, etc.

The language also provides global data access mechanism called 'system slot' and 'process slot'. The system slots hold data objects identified uniquely in the system, such as the current active process control block, interrupt handler table, etc. The process slots hold data objects identified

uniquely in the process, such as the current input stream and the current output stream.

Nonlocal-exit control is represented by the 'catch and throw'. The 'catch and throw' provides a means to return directly to an ancestor goal beyond ordinary "call/return" controls. This is useful for dealing with errors and interrupts encountered during execution of programs. For example, a Prolog listener may be written as follows.

```
listener_top :- repeat,
                catch(abort_listener,abort_msg),
                start_listener.

abort_msg :- print('execution is aborted'),
             nl,fail.
```

In the above program, if the goal 'throw(abort_listener)' in the descendent goals of 'catch(abort_listener,abort_msg)' is called, all choice-points between the catch goal and the throw goal are pruned. Then a catch-handler 'abort_msg' is called next. If the catch-hander fails, it acts as if the catch goal fails. If the catch-hander succeeds, the next goal of the catch goal, that is, start_listener is called as the continuation goal.

The primitives to support a multiprocess environment are:

(1) Process context-exchange operation.
(2) Asynchronous firmware/software trap management
(3) Interrupt/trap disable or enable control.

These primitives are provided as built_in predicates to construct whole software system in Prolog. The multiprocess environment supported by HPM is based on the message passing paradigm such as 'send/receive' and 'asynchronous-signal'. The multiprocess environment provides asynchronous input/output data transfer, break-interrupt from keyboard and foreground/background job control.

3.3 Basic HPM architecture

Basic HPM architecture features are summarized in the following.

3.3.1 Data areas

HPM provides six data areas for holding runtime status. The areas are system area, code area, heap, global stack, local stack and trail stack.

The system area contains the information required for memory management and process control. This area is also used for micro program working spaces. The code area contains compiled codes. The heap contains data structures that create side effects when updated.

The global stack contains structures and lists instantiated in the unification process (using the 'structure copying' technique). The local stack contains frames for variable binding and execution control. The trail stack contains references to the variables to be unbound when backtrack occurs.

3.3.2 Registers and frames

HPM provides the following hardware registers for defining the current status of a Prolog computation. The main control registers are:

```
LE :    last environment-frame pointer
LC :    last choice-point-frame pointer
CLC:    call's last choice point frame pointer
LTOP:   local stack-top pointer
GTOP:   global stack-top pointer
TTOP:   trail stack-top pointer
HTOP:   heap-top pointer
GBACK:  backtrack pointer for global stack
SP:     structure pointer
PC:     program counter
CP:     next goal address
LEV:    procedure-level counter
CDT:    current data-table pointer
```

In addition, HPM provides 32 argument registers (A0, A1, ... A31) for passing the arguments to a procedure. The argument registers are also used for holding the values of temporary variables that are variables not referred to in the rest of goals in a clause.

In order to deal with procedure call and backtrack control, the local stack contains three kinds of frames: an environment-frame, a choice-point-frame and a catch-frame. The environment-frame contains values of (permanent) variables and information needed to continue program execution (CP, CDT, LE, CLC and LEV). The choice-point-frame contains an address (continuation) of an alternative clause and runtime status required for backtracking (LC, CLC, PC, CP, LEV, CDT, LE, TTOP, GTOP, n and A0,....,An-1, where n is the number of arguments). The catch-frame is provided for a nonlocal-exit using 'catch and throw'. The catch-frame has the same information as the choice-point frame, but it has a label and a continuation to a catch-handler in place of the alternative clause.

3.3.3 Instruction set

About 180 machine instructions (including around 100 built-in predicates) are available in HPM. The instruction set is classified into 6 categories, i.e., 'get', 'put', 'unify', 'control', 'index' and built-in predicates.

The 'get', 'put' and 'unify' instructions are provided for unification. They correspond to the arguments of goals in a clause. The 'control' instructions are provided for procedure invocation and environment-frame allocation. The 'indexing' instructions are provided for backtracking control and for 'clause-indexing'. Built-in predicates are provided for primitive operations, such as arithmetic, input/output functions etc.

4. Hardware

The HPM architecture can be implemented on conventional computers. However, conventional computers have limitations in execution speed and main memory capacity, necessary for practical Prolog programs. The HPM attains the following objectives from the hardware design aspect:

- Implementing fast argument matching and stack control mechanisms.

- Providing a large main memory capacity for a single user.

An HPM processor is realized by using microprogramming technology and CML device technology. An HPM memory system provides a 64 Mega word memory (256 Mega bytes, 36bit/word).

4.1 System Configuration

The HPM system is a high-level language machine for a single user. It consists of an HPM processor, a host computer and a main memory system as shown in Fig.3.

The PSI is provided as a host computer, which controls input/output devices, an HPM processor interface and other external system interfaces. PSI provides an excellent man-machine interface and a local area network interface. In this configuration, a large Prolog program is offloaded from the PSI and executed in the HPM.

The HPM processor consists of a Prolog processing unit, a cache memory and a host processor interface unit. The HPM processor directly interprets internal object data and codes by using its microprogram and hardware, whose cycle time is 100 nano-seconds.

The memory system consists of two main memory units, a system control unit and a service processor with a maintenance console. Internal object code and data are stored in the 36-bit word main memory units.

4.2 Prolog processor

The main consideration in designing the Prolog processor is the performance. In order to realize a high performance machine, the HPM processor can simultaneously execute several functional modules, instruction fetching, instruction decoding, argument matching between a goal and a head of a Prolog program, stack pointer manipulation, floating-point arithmetic operation and fixed-point arithmetic operation. These functional modules are connected to a 36-bit long data bus, the same as a cache memory and a host processor interface. Figure 3 shows the Prolog processor configuration.

The Prolog processor executes instructions sequentially in a pipeline mode with three stages; instruction fetching, decoding and execution. The fetching stage and the decoding stage are preprocessed before an execution stage begins. At the instruction fetching stage, an instruction register receives a one-word instruction from a cache memory. At the instruction decoding stage, the instruction is decoded and the decoded results are latched in a Parent-Address Register (PAR) and a Current-Address Register (CAR). These registers point to respective registers in a two-port 32-word Data Register File (DRF). The register file contains argument registers and temporary registers, described in section 3.3.

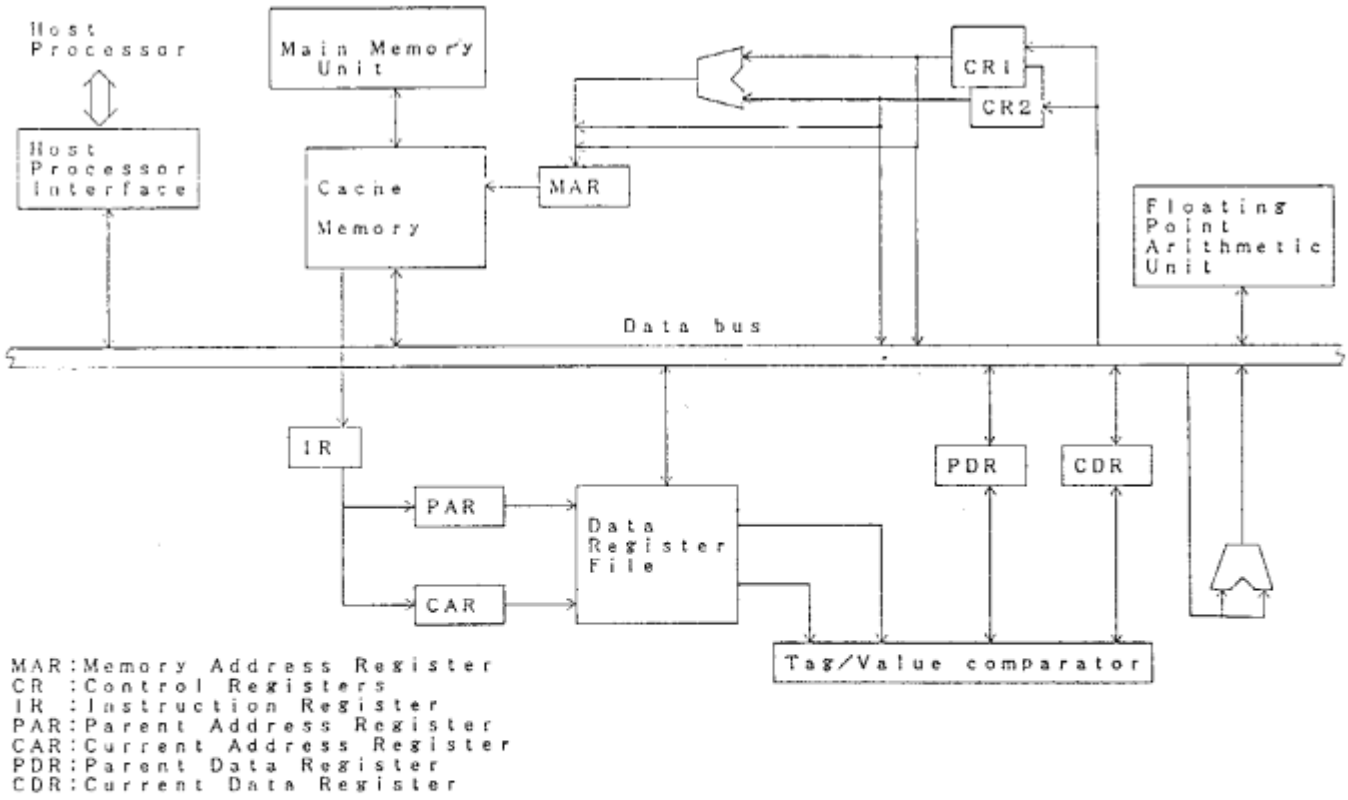

Fig. 3 HPM Processor hardware configuration

At the execution stage of an HPM instruction, an argument-matching module and a stack-pointer manipulation module are mainly used in parallel. In regard to the argument-matching module, argument data and temporary data in the data register file are compared using a tag/value comparator. The DRF enable simultaneous access to both argument registers and temporary registers. In regard to the stack pointer manipulation module, the HPM processor can update stack pointers in the register files using an ALU and a tag generator. Two-way read/write access to the control register files is possible by dividing the files into two register files. Dividing into two register files provides a way to compare and modify various pointers in the control register files simultaneously. The HPM architecture requires many control registers. The 48 control registers are assigned to stack pointers and other registers specified in HPM architecture. These registers are also assigned to working registers used by microprograms. The usage of these registers and the checking function of Tag/Value comparator are shown as micro-program operations in Fig.4.

The microprogram memory structure is an 80-bit word and 11 K word memory size. A microinstruction is divided into [illegible] fields whose [illegible] can be executed in parallel. The instruction decoder and the tag/value comparator are composed of random-access memories, instead of random logic circuits.

The cache memory, which holds HPM instructions and data, has an access time of 100 nano-seconds. The cache memory adopts a set-associative strategy with four compartments and a write-through memory interface. The cache memory capacity is 8 K words. Another possible cache memory configuration considered provided two cache memories, an instruction cache and a data cache. This configuration is effective if the HPM processor frequently requires simultaneous access to instructions and data. But, most instruction fetching cycles are hidden during execution, which mainly accesses registers. In Fig.4, instruction fetching operations are described in only time 3 and time 6. The other instruction fetching operations are hidden in main micro-program operations.

time	instruction	main micro-program operation
1	switch_on_term A0	load the first word(A0) of DRF into CDR, check the A0 tag : list
2		calculate 'jump Ls' instruction address
3		read 'jump Ls' instruction into IR --- pipeline control
4		load operand syllable of 'jump Ls' into PAR --- pipeline control
5	jump Ls	calculate 'get list A0' instruction address, addressed relatively
6		read 'get_list A0' instruction into IR --- pipeline control
7		load an operand syllable of 'get_list A0' into PAR --- pipeline control
8	get_list A0	check the first word(A0) tag : list
9		load list pointer in A0 into Structure Pointer(SP) in CR, set 'read flag'
10	unify_temp_var X3	load SP into MAR, check 'read flag'
11		read the header element of a list into PDR
12		SP←SP+1, check the PDR tag : constant
13		load PDR into the forth word(X3) of DRF
14	unify_temp_var A0	load SP into MAR, check 'read flag' --- structure read mode
15		read the tail element of a list into PDR
16		SP←SP+1, check the PDR tag : constant
17		load PDR into the first word(A0) of DRF
18	get_list A2	check the third word(A2) tag : reference
19		load the third word(A2) of DRF into MAR
20		read referenced word into CDR
21		load global stack top pointer(GTOP) in CR into SP, check CDR tag : unbound
22		update GTOP
23		write list tag and SP value into the unbound-cell pointed by MAR
24		check the unbound-cell address using CR1, CR2 and ALU
25		load operand syllable of next instruction --- pipeline control
26	unify_temp_val X3	load SP into MAR, check 'read flag' ---structure write mode
27		SP←SP+1, check the forth word(X3) tag in DRF : constant
28		write X3 into the header element of a list pointed by MAR
29	unify_temp_var A2	load SP into MAR, check 'read flag' --- structure write mode
30		load tail list element address in SP into the third word(A2) of DRF
31		write unbound data into tail element
32	execute relative	load tail list element address in SP into the third word(A2) of DRF
33	concatenate/3	check local stack boundary with local stack-top pointer(LTOP)
34		check global stack boundary with global stack-top pointer(GTOP)
35		load operand syllable of next instruction --- pipeline control

Fig.4 Determinate Concatenate Timing and Execution Process

The authors finished debugging the HPM processor hardware. As a result, it has become possible to estimate its hardware size. The Prolog processor is composed of 75 boards, including a clock board. One board contains 64 CML chips. The logic density of a CML chip is roughly twice that of an off-the-shelf TTL IC, on the average. The CML gate delay is around 0.7 nano-seconds.

5. Performance Estimation

The HPM performance estimate is based on a microprogram execution for a deterministic execution of the "concatenate" procedure explained in the architecture section. When "?-concatenate([a,b,c,d,e],[f,g], X)", where X is a variable, is executed, the machine instructions and micro-operations in Fig.4 are repeatedly executed corresponding to the second "concatenate" clause.

The 9 machine instructions are executed in 35 machine cycles, when the hit ratio in the cache memory is 100%. In this case, HPM executes the deterministic concatenate clause at 280 KLIPS. The HPM performance is much better than the performance on conventional machines, which are typically around 30 KLIPS[10]. The PSI performance is also around 30 KLIPS[6]. Moreover, the HPM performance can be improved by introducing a more effective instruction prefetching control and more intensive micro-branch operation.

6. Conclusion

The authors have been developing the HPM hardware and software system, which can execute large Prolog programs for practical applications. HPM provides a high level performance with a specialized processor and a large capacity in the main memory, as well as a practical programming environment with compiler, interpreter, debugger and other system programs. HPM hardware will be available as the fastest Prolog machine in the world by this late summer of 1985. The HPM basic software system will also be available. As a result, HPM enhances the possibility of making the Prolog language a practically usable language from a prototyping language. HPM provides a compiler oriented architecture with high level stack control instructions. In order to realize a practical programming environment on HPM, its architecture provides side-effect operations, nonlocal-exit control and multiprocess support primitives. The Prolog machine was developed by using MSI function level CML chips.

The HPM can execute a Prolog list concatenating program at 280 KLIPS. From the hardware configuration viewpoint, it has a possibility to improve performance by adopting intensive pipeline processing and specialized hardware for unification. Further evaluation is necessary on practical Prolog programs, including system functions overhead and application programs productivity.

A single-user computer can provide a better programming environment for knowledge information processing and other software products. HPM should be more compact to be used as a single-user computer. In the future, it is necessary to research the machine architecture and hardware configuration, considering LSI technology. Development of optimization techniques is also necessary to improve the HPM performance.

Acknowledgments

The authors would like to express their grateful thanks to Toshio Yokoi and Shun'ichi Uchida (ICOT), and Yasuo Kato (NEC) for their continuous encouragement and to Katsuya Hakozaki (NEC) for his valuable advice and support. Thanks are also due to Kazuo Taki, Akira Yamamoto, Hiroshi Nishikawa (ICOT), and Kazunori Ueda (NEC) as well as to Noriko Kijima (NEC) for their fruitful discussions and machine implementation researching efforts.

References

1. Kowalski, R., "Logic for problem Solving", North Holland Publishing Co., New York, 1979.
2. Warren, D.H., et al., "PROLOG---the language and its implementation compared with LISP", SIGART/SIGPLAN Notices, pp. 109-115, Aug. 1977.
3. Konagaya, A. and Umemura, M.,"Knowledge Information Processing Language: ShapeUp", New Generation Computing, Vol.2, No.2, Pressed by Ohmsha, Springer Verlag 1984, pp.195-201.
4. Chu, Y., "High-Level Language Computer Architecture", Academic Press,1975.
5. Uchida, S., "Inference Machine: From Sequential to Parallel" Proc. of 10 the International Symposium on Computer Architecture, June 1983, pp. 410-416.
6. Uchida, S. and Yokoi, T., "Sequential Inference Machine: SIM Progress Report", Proc. -of FGCS '84, Tokyo, Nov, 1984.
7. Nishikawa, H., et al., "The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture, "Logic Programming Workshop '83, Universidade Nova de Lisboa, June 1983, pp. 53-73.
8. Bowen, D.L., " DECsystem-10 PROLOG USER'S MANUAL",Department of Artificial Intelligence Univ. Of Edinburgh, Dec. 1981.
9. Warren, D.H., "An Abstract Prolog Instruction Set", Tech. report 309, Artificial Intelligence Center, SRI International, 1983.
10. Tick, E. and Warren, D.H., "Towards a Pipelined Prolog Processor", 1984 International Symposium on Logic Programming, IEEE Computer Society, February 1984.
11. Warren,D.H.,"An Improved Prolog Implementation which Optimises Tail Recursion",Research Paper 156, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1980.