ICOT Technical Memorandum: TM-0426

TM-0426

Co-operative High Performance Sequential Inference Machine:CHI-II

by
S. Habata, R. Nakazaki, A. Konagaya,
A. Atarashi & M. Umemura

December, 1987

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

Co-operative High Performance Sequential Inference Machine: CHI-II

By

Shin-ichi HABATA,* Ryosei NAKAZAKI,* Akihiko KONAGAYA,*
Atsushi ATARASHI* and Mamoru UMEMURA*

Reprinted from the *NEC RESEARCH & DEVELOPMENT*
No. 88, pp. 92–101, January 1988

UDC 681.3.01/.06:800.92:658.512.2

Co-operative High Performance Sequential Inference Machine: CHI-II

By Shin-ichi HABATA,* Ryosei NAKAZAKI,* Akihiko KONAGAYA,* Atsushi ATARASHI* and Mamoru UMEMURA*

ABSTRACT A Co-operative High performance sequential Inference (CHI) machine has been developed as part of the Fifth Generation Computer Systems (FGCS) project in Japan. In this project, a logic-programming language is used as a kernel language, and the CHI was designed as a logic-programming language processor. This paper describes the CHI system features and its performance, especially the CHI-II (advanced CHI version) design approach and its characteristics. CHI-II was designed for personal use and as a desk-side computer. Generally, the CHI systems have four features: tag architecture, back-end machine, large capacity main memory and logic-programming language oriented instruction set. CHI-II performance is evaluated to be at 500 KLIPS for a deterministic append program execution in 6 MHz machine clock.

KEYWORDS High level language machine, Machine architecture, Hardware, Performance, Compiler

1. INTRODUCTION

Logic-programming language, such as Prolog, has been considered to provide useful capabilities in Artificial Intelligence (AI) [1][2], and is used as a kernel language in the Fifth Generation Computer Systems (FGCS) project in Japan. It excels in knowledge representation, and provides a backtracking function for resolution search. However, on conventional computer systems, the logic-programming language requires a large amount of CPU power, and occupies much memory space. Therefore, it is difficult to handle a large practical application system on conventional computer systems, and it is necessary to develop specialized computer systems, which efficiently execute logic-programming language programs.

The Co-operative High performance sequential Inference (CHI) machine has been developed as part of the FGCS project in Japan. CHI was planned as an AI researcher's tool for AI application systems development, and was designed as a logic-programming language processor. During the project, two CHI versions were developed: CHI I (prototype CHI) [3], and CHI-II (advanced version).

CHI I was completed in 1985, and has attained 285 Kilo Logical Inferences Per Second (KLIPS) executing a deterministic append program. CHI-II was planned to improve CHI-I hardware, and to attain a more powerful performance than was available with CHI-I. As a result of hardware and architecture improvements, CHI-II performance is evaluated to be at 500 KLIPS in the append program execution.

This paper describes CHI system features and performance, especially CHI-II design approach and its characteristics.

2. CHI SYSTEM FEATURES

The CHI architecture is based on the Warren Abstract Machine (WAM) [5]. The CHI-I was an experimental machine constructed to prove the CHI architecture, and to attain high performance. The CHI-I is implemented with Current Mode Logic (CML) ICs, which are used on conventional large or medium sized computer systems. Then, it is not desirable to use the CHI I systems in office circumstances. However, the CHI-II was planned as a practical machine for personal use, and designed as a desk side machine for use in an office. Specifications for CHI I and CHI II are described in Table I. Generally, the CHI system features are as follows:

- Tag architecture
- Back end machine
- Large capacity main memory
- Logic-programming language oriented instruction set

2.1 Tag Architecture

The tag architecture is adopted into the CHI systems, because the logic-programming language requires frequent dynamic checking of the data type during program execution, and the tag architecture is suitable for such data type checking. CHI data consists of a tag field and a value field. The tag field is used

*C&C Systems Research Laboratories

for checking the data type, and the value field holds the data value. The CHI machines can execute both tag checking and value computation in parallel.

2.2 Back-end Machine

The CHI machine is designed as a back-end machine. A host machine is used for controlling human machine interface and peripheral devices. The CHI system provides an excellent programming environment, in which multi window systems and pointing devices, such as a mouse, can be used by making use of the programming environment on the host machine. Moreover, any application systems executed on the CHI machine can manipulate multi-window systems and any peripheral devices on the host machine. This configuration makes it easy to save development costs and to concentrate on a logic programming language processor design.

2.3 Large Capacity Main Memory

Logic programming language requires much memory space during execution for backtracking and predicate call operations. The backtracking and predicate calls are fundamental operations. These operations construct frames on a stack, required for logic-programming language execution. The CHI system was intended to provide an environment for executing large practical application systems. Thus, the CHI-I machine is equipped with massive main memory units, which are used on conventional large scale computer systems. The CHI I system provides 64 megaword main memory capacity. In the CHI-II machine, 18 cards are used for the main memory system, while 4 cards are used for the processor. The CHI-II system provides 128 megaword main memory capacity.

Table I Specifications for CHI I and CHI II

Item	CHI-I	CHI-II
Device	CML 256K bit DRAM	TTL, CMOS 1M bit DRAM
Machine clock	10 MHz	6 MHz
Data length	36 bits 4 or 7 bit tag 32 or 29 bit value	40 bits 8 bit tag 32 bit value
Main memory	64 MW (256 MB)	128 MW (640 MB)
Cache memory	8 KW (32 KB)	32 KW (160 KB)
Micro-instruction	80 bits	78 bits
WCS	11 KW (110 KB)	16 KW (160 KB)
Performance on append	285 KLIPS	500 KLIPS

2.4 Logic programming Language Oriented Instruction Set

The CHI system makes use of instruction sequence optimization by a compiler. The compiler can reduce many redundant operations, such as duplicate data write operations to registers, data transfers in vain from one register to another and unify operations indicated as failed. In the CHI system, the logic programming language oriented instruction set is provided for the compiler to efficiently optimize instruction sequence. Optimization by the compiler is one of the important factors which must be implemented to attain a high level performance on the CHI systems.

3. CHI II GOALS

The followings are the improvement features for CHI-II, inheriting fundamental concept of CHI I.

- Desk-side machine
- Performance improvement
- Excellent programming environment
- Flexible system development

3.1 Desk-side Machine

The CHI-II was planned as a practical machine, which can be used in an office. To minimize the size, three hardware technologies were used. First, about 20,000 gates are integrated on two CMOS gate array LSI chips. Second, 1 megabit dynamic memory chips are used for increasing the CHI-II main memory capacity and decreasing the number of memory ICs. Third, high-density packaging technology is applied to implement the main memory card with 8 megaword, or 40 megabyte capacity.

3.2 Performance Improvement

In the CHI-II system, hardware and architecture were improved to attain more efficient performance. The data length was extended from 36 bits to 40 bits, to introduce new data types, and expand the logical memory space. The cache memory capacity was increased from 8 kilowords to 32 kilowords, and other hardware functions were improved to decrease the number of micro-steps required during program execution.

3.3 Excellent Programming Environment

Excellent programming environment is essential to developing large practical applications. To make a better programming environment, the communication mechanism between the host machine and the CHI-II machine is improved, and an asynchronous bidirectional communication unit is applied in the CHI-II system. Therefore, any applications handled on the CHI-II

machine can efficiently manipulate multi-window systems, and utilize the data from a mouse device on the host machine.

3.4 Flexible System Development

A general purpose instruction set was introduced for describing most built-in predicates. Thus, it is easy to add, improve, and modify the built in predicates. The built-in predicates are used on the CHI-II operating system and other utility programs, such as an editor, an interpreter, and a compiler, which are important parts of the CHI II system. Therefore, it is necessary to add, improve, and modify the built-in predicates, in order to improve the CHI II performance.

4. CHI-II HARDWARE TECHNOLOGIES

The following hardware technologies were used to implement a desk-side size, 128 megaword main memory capacity and more efficient performance.

- CMOS gate array chips
- High-density packaging technology

4.1 CMOS Gate-array Chips

Two CMOS gate-array LSI chips have been developed: one for cache memory and instruction prefetch control, and the other for microprogram sequence control. These chips are constructed using a 1.5 micron rule CMOS process with two metal layer levels for interconnection. About 20,000 gates are integrated on the two chips. Each chip is housed in a pin-grid-array package with 208 pins.

4.2 High-density Packaging Technology

High-density packaging technology is applied to decrease the number of memory cards. One megabit dynamic memory chips are used for main memory. 384 memory ICs are surface-mounted on both sides of a memory card, which provides an 8 megaword, or 40 megabyte memory capacity. Then, 16 memory cards are required to provide 128 megaword, or 640 megabyte main memory capacity.

5. CHI-II CONFIGURATION

The CHI II system provides an excellent logic programming environment for a single user, including multiprocessing and multi-window environment. It consists of a host machine and the CHI-II machine (as shown in Fig. 1).

The host machine controls peripheral devices, and provides excellent human-machine interface functions, file management functions and CHI-II hardware monitoring functions. An engineering workstation with UNIX* system V is used as the host machine. The CHI-II machine is connected with the host machine by SCSI, which is a standard interface for small computer systems.

The CHI-II machine is composed of a high-performance processor (CHI-II processor), a large capacity main memory (CHI-II memory system) and a host interface unit (HIU) (as shown in Fig. 2). The HIU controls asynchronous bidirectional communications between the host machine and the CHI-II machine. It also converts CHI-II 40-bit data into 32 bit data used in the host machine, and vice versa. Therefore, the host machine is considered an excellent I/O processor for the CHI-II processor.

The CHI-II memory system provides a 128 megaword space with error correction, and includes a mapping unit, which translates logical address to physical address. In the CHI-II machine, eight independent logical spaces are required to execute logic-programming language program (as shown in Fig. 3).

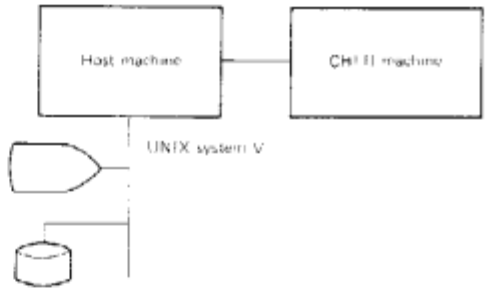


Fig. 1 CHI-II system configuration.

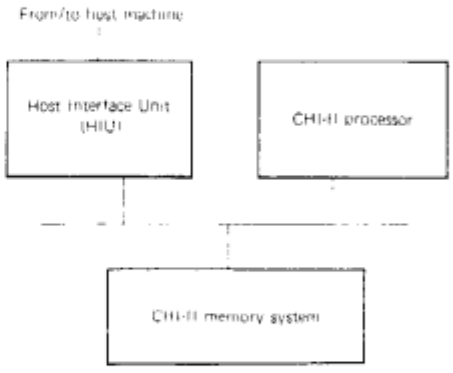


Fig. 2 CHI-II machine configuration

*UNIX is an operating system developed by AT&T Bell Laboratories of the United States.

These spaces are called:

- System Control Area: For system information and micro work area
- Code Area: For executable machine code
- Common Data Area: For shared data
- System Object Area: For system resource
- Local Stack Area: For program control information
- Global Stack Area: For list and structure data
- Trail Stack Area: For undo control
- Local Heap Area: For process information

All processes handled on the CHI-II machine share four areas; System Control area, Code area, Common Data area, and System Object area. The CHI-II logical address consists of three fields (as shown in Fig. 4). One is the Process ID field, which is 12 bits in length, and defines a corresponding process. Another is the Area ID field, 3 bits in length, and indicates an area identity number. The other is Internal Address field, which provides a 29 bit internal address in the area specified by the Area ID field.

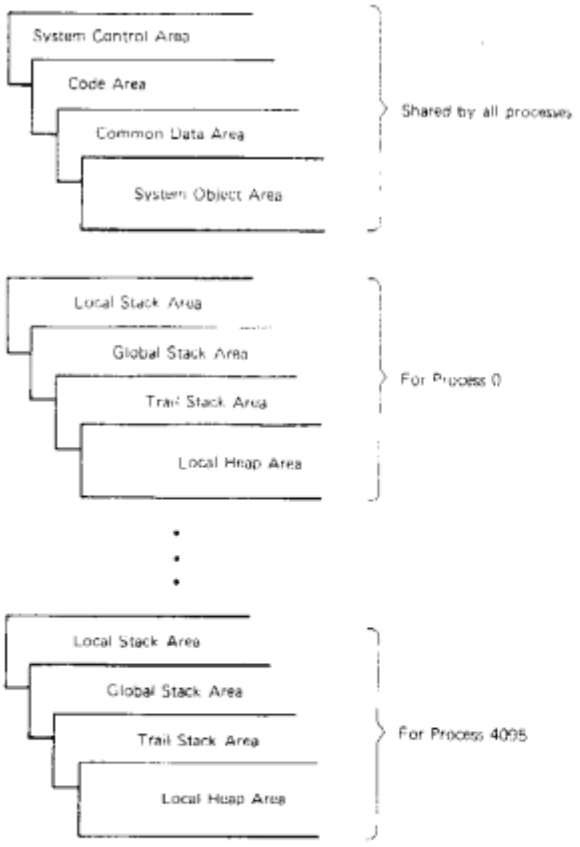


Fig. 3 Logical memory space.

6. CHI-II PROCESSOR DESIGN APPROACH

The CHI-II is implemented with CMOS gate array LSIs and off-the-shelf TTL ICs. Their gate propagation speed is slower than that for CML ICs used on CHI-I. To overcome the slow gate propagation speed problem, the following four points were considered:

- Memory access speed
- Micro-operation execution in parallel
- Micro-branch condition integration
- Compiler optimization

6.1 Memory Access Speed

The CHI-II machine clock cycle was fixed to 6 MHz based on cache memory cycle, in order to carry out a memory access operation in a micro-step when the data exists on the cache. In the logic-programming language, memory accesses occur very frequently, and most fundamental operations require sequential data accesses. Then, it is necessary to immediately handle any memory access operation required as the result of the memory access executed in the previous micro step.

There are two characteristic memory access operations on the logic-programming language. The first is to construct two kinds of frames on the local stack: the environment frame, which stores information for predicate call, and the choice point frame for backtracking (as shown in Fig. 5). The predicate calls and backtracking are fundamental operations in the logic programming execution. Then, in the logic-programming language, memory accesses occur very frequently, and a great amount of memory space is consumed.

The second is to sequentially trace a reference chain. The reference pointers are used for unifying two variables, and it is necessary to trace a reference chain for accessing a variable. The second characteristic makes it difficult to execute memory accesses in parallel or pipeline. Then, the CHI-II machine was designed to execute a memory access operation in a micro step, whenever the data existed in the cache memory.

6.2 Micro-operations in Parallel

The CHI-II processor was designed to eliminate micro-steps, which do not contain any memory access

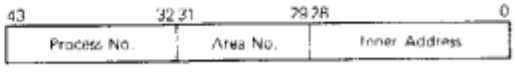
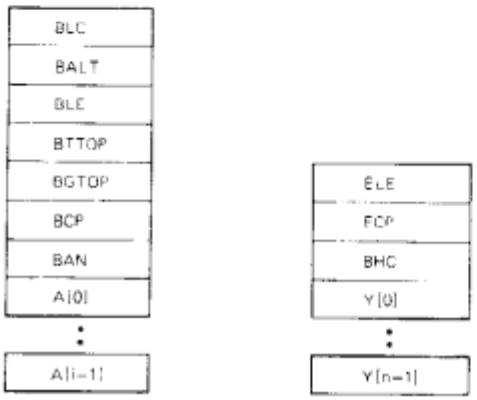


Fig. 4 Logical address format.

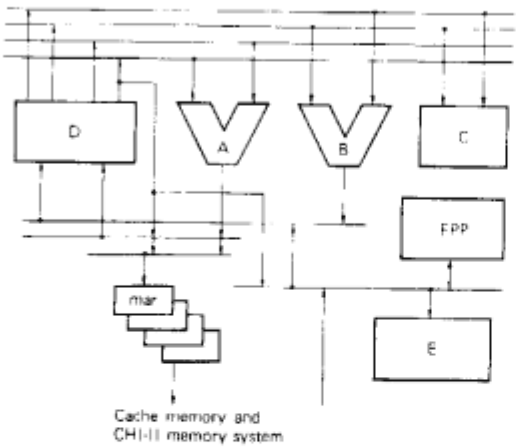
operation. Therefore, the number of data paths has been increased, and two ALUs are provided, one for address calculation, and the other for data computation (as shown in Fig. 6).

The CHI-II processor also provides an argument comparator for data type checking, and a multi-port register file, which has four read ports and two write ports. Then, the CHI-II processor can execute address calculation, value comparison, and tag checking in a micro-step. It can also increment or decrement a memory address, and calculate the instruction prefetch address in the same micro-step. Most instructions can be executed in two or three micro-steps in the CHI-II processor.



(a) Choice point frame. (b) Environment frame.

Fig. 5 Stack frames.



A: ALU for address calculation
B: ALU for data computation
C: argument comparator
D: multiport register file
E: instruction prefetcher
FPP: floating point processor
mar: memory address registers

Fig. 6 CHI-II processor configuration.

6.3 Micro-branch Condition Integration

Some micro-branch conditions are integrated to reduce the number of micro-steps. Micro branch operations are immediately executed in the micro-step, in which the condition is checked. In the CHI-II processor, the result of tag checking determines what kind of micro branch should be used. As an example of unification, which is a fundamental operation matching two arguments in logic-programming language, the result of tag checking directly indicates whether both arguments have the same data type or not. For example, in the case when both data types are "atom"s, the result of value comparison, "equal" or "not equal," determines the next micro-routine, "success" or "fail." However, if their data types are "variable," the result of value comparison, "greater" or "less," is used for determining the next micro routine. In the case of different data types, the result of tag checking is sufficient to determine the next micro routine, "fail" or "unify a variable with the other argument." Examples of micro-branch operations for unify operation are shown in Table II.

6.4 Compiler Optimization

The CHI-II processor can execute plural micro operations in a micro-step. However, the original WAM instruction set does not fully make use of this CHI-II processor capability. Therefore, new instructions are introduced for indexing and list processing. The effect of the newly introduced instructions will be explained, using the deterministic append program.

The append program written on the WAM instruction set requires 8 instructions to process each element of a list (as shown in Fig. 7(b)). In this case, 20 micro-steps are required.

Table II Examples of micro-branch operation.

Results of tag checking	Result of value comparison	Micro branch operation
atom:atom	Va = Vb	call micro routine for next instruction
	Va =/= Vb	jump "fail" routine
var:var	Va > Vb	"Va → Vb" routine
	Va < Vb	"Vb → Va" routine
atom:list	—	"fail" routine

```
append([ ],X,X)
append([X | A],B[X | C]):-append(A,B,C).
```

(a) append program written on Prolog.

```
●  ○  $ap:  switch_on_term        $nl,$ls,$fail
      $v0:  try_me_else           $v1
      $nl:  get_nil               A0
            get_t_value           A1,A2
            proceed
      $v1:  trust_me_else_fail
●  ○  $ls:  get_list              A0
   ○        unify_t_variable      A3
   ○        unify_t_variable      A0
   ○        get_list              A2
   ○        unify_t_value         A3
   ○        unify_t_variable      A2
   ○        execute               $ap
```

(b) Compiled code, using the original WAM instructions.

```
   ○  $ap:  switch_on_list        $nl,$ls,$fail
      $v0:  try_me_else           $2
            get_nil               A0
      $nl:  get_t_value           A1,A2
            proceed
      $v1:  trust_me_else_fail
            get_list              A0
   ○  $ls:  unify_t_variable      A3
   ○        unify_t_variable      A0
●  ○        get_list              A2
●  ○        unify_t_value         A3
●  ○        unify_t_variable      A2
   ○        execute               $ap
```

(c) Compiled code, using switch_on_list.

```
●  ○  $ap:  switch_on_list        $nl,$ls,$fail
      $v0:  try_me_else           $v1
            get_nil               A0
      $nl:  get_t_value           A1,A2
            proceed
      $v1:  trust_me_else_fail
            get_list              A0
●  ○  $ls:  unify_t_variable      A3
   ○        unify_t_variable      A0
   ○        get_list_t_val_t_var  A2,A3,A2
   ○        execute               $ap
```

(d) Compiled code, using get_list_t_val_t_var.

```
   ○  $ap:  switch_on_list_t_var  A3,$nl,$ls,$fail
      $v0:  try_me_else           $v1
            get_nil               A0
      $nl:  get_t_value           A1,A2
            proceed
      $v1:  trust_me_else_fail
            get_list              A0
            unify_t_variable      A3
   ○  $ls:  unify_t_variable      A0
   ○        get_list_t_val_t_var  A2,A3,A2
●  ○        execute               $ap
```

(e) Compiled code, using switch_on_list_t_var.

```
(○) $ap:  switch_on_list_t_var  A3,$nl,$ls,$fail
    $v0:  try_me_else           $v1
          get_nil               A0
    $nl:  get_t_value           A1,A2
          proceed
    $v1:  trust_me_else_fail
          get_list_t_variable   A0,A3
 ○  $ls:  unify_t_variable      A0
 ○        get_list_t_val_t_var  A2,A3,A2
 ○        switch_on_list_t_var  A3,$nl,$ls,$fail
          goto                  $v0
```

(f) Compiled code, exchanging execute for switch_on_list_t_var.

Fig. 7 Examples of compiler optimization.

Table III Performance evaluation in append program execution.

Program	Instruction count	Micro-step count	Execution speed
(b)	8 instructions/LI	20 micro-steps/LI	300 KLIPS
(c)	7 instructions/LI	19 micro-steps/LI	315 KLIPS
(d)	5 instructions/LI	17 micro-steps/LI	352 KLIPS
(e)	4 instructions/LI	15 micro-steps/LI	400 KLIPS
(f)	3 instructions/LI	12 micro-steps/LI	500 KLIPS

Table IV Newly introduced instructions.

```
switch_on_list L0,L1,L2
switch_on_list_t_variable Xi,L0,L1,L2
switch_on_list_t_value Xi,L0,L1,L2
switch_on_list_p_variable Yi,L0,L1,L2
switch_on_list_p_value Yi,L0,L1,L2
get_list_t_var_t_var Ai,Xj,Xk
get_list_t_var_t_val Ai,Xj,Xk
get_list_t_val_t_val Ai,Xj,Xk
get_list_t_val_t_var Ai,Xj,Xk
get_list_t_var_p_var Ai,Xj,Yk
get_list_p_var_t_var Ai,Yj,Xk
get_list_t_val_p_var Ai,Xj,Yk
get_list_p_var_t_val Ai,Yj,Xk
get_list_t_var_p_val Ai,Xj,Yk
get_list_p_val_t_var Ai,Yj,Xk
get_list_t_val_p_val Ai,Xj,Yk
get_list_p_val_t_val Ai,Yj,Xk
put_list_t_var_t_var Ai,Xj,Xk
put_list_t_var_t_val Ai,Xj,Xk
put_list_t_val_t_val Ai,Xj,Xk
put_list_t_val_t_var Ai,Xj,Xk
put_list_t_var_p_var Ai,Xj,Yk
put_list_p_var_t_var Ai,Yj,Xk
put_list_t_val_p_var Ai,Xj,Yk
put_list_p_var_t_val Ai,Yj,Xk
put_list_t_var_p_val Ai,Xj,Yk
put_list_p_val_t_var Ai,Yj,Xk
put_list_t_val_p_val Ai,Xj,Yk
put_list_p_val_t_val Ai,Yj,Xk
unify_list_t_var_t_var Xi,Xj,Xk
unify_list_t_var_t_val Xi,Xj,Xk
unify_list_t_val_t_val Xi,Xj,Xk
unify_list_t_val_t_var Xi,Xj,Xk
unify_list_t_var_p_var Xi,Xj,Yk
unify_list_p_var_t_var Xi,Yj,Xk
unify_list_t_val_p_var Xi,Xj,Yk
unify_list_p_var_t_val Xi,Yj,Xk
unify_list_t_var_p_val Xi,Xj,Yk
unify_list_p_val_t_var Xi,Yj,Xk
unify_list_t_val_p_val Xi,Xj,Yk
unify_list_p_val_t_val Xi,Yj,Xk
```

One newly introduced instruction
switch_on_list
includes both
switch_on_term
and
get_list

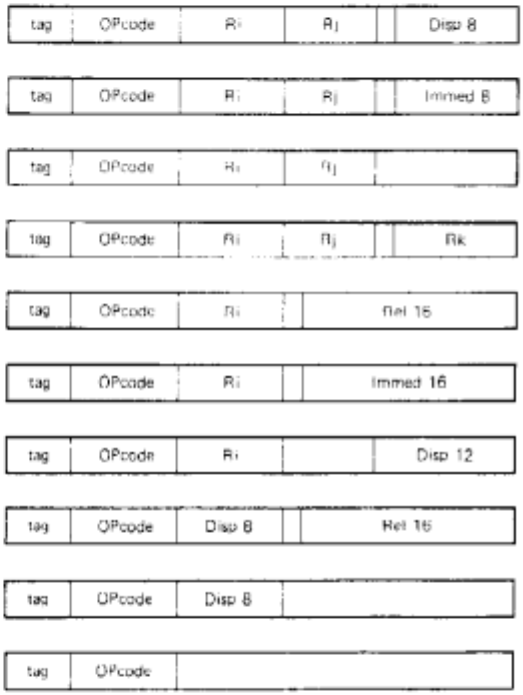


Fig. 8 Instruction code format.

functions of the WAM. Then, during the execution of the append program using the newly introduced instruction, switch_on_list, 7 instructions and 19 micro-steps are required in the same operation (as shown in Fig. 7(c)).

Another new instruction
get_list_t_val_t_var
replaces three WAM instructions.
get_list
unify_temporary_value
unify_temporary_variable
(as shown in Fig. 7(d)). Thus, 5 instructions and 17 micro steps are enough to process each element in a list.

The new instruction

switch_on_list_t_var

adds

unify_temporary_variable function

to

switch_on_list instruction

(as shown in Fig. 7(e)). In this case, 4 instructions and 15 micro-steps are required.

A compiler can exchange

execute instruction

for

switch_on_list_t_var

(as shown in Fig. 7(f)). Then, 3 instructions and 12 micro-steps are executed to process each element in a list.

Execution speeds in the append program are summarized in Table III.

The CHI-II performance is improved from 300 KLIPS to 500 KLIPS, applying the compiler optimization. The newly introduced instructions are listed in Table IV. To append the new instructions, operation code length is expanded from 8 bits to 10 bits. There are 10 kinds of instructions formats in the CHI-II system (as shown in Fig. 8).

7. PERFORMANCE EVALUATION

CHI-II performance is estimated based on micro-program execution for 18 typical machine instructions and the append program, on the assumption that the cache hit ratio is 100%. The number of micro-steps and memory access operations in each machine instruction was counted to compare CHI-I and CHI-II performances (as shown in Table V).

In most machine instructions, the number of micro-steps in CHI-II is smaller than that required in CHI-I, and the total micro-step count for typical 18 machine instructions in CHI-II is approximately half as

Table V Memory access ratio in typical 18 machine instructions.

CHI machine instruction	Micro-step count		Memory access count	Memory access ratio	
	CHI-I	CHI-II		CHI-I	CHI-II
get_permanent_variable	2	2	2	1.000	1.000
get_temporary_variable	2	1	1	0.500	1.000
get_permanent_value	5	3	2	0.600	0.667
get_temporary_value	3	1	1	0.667	1.000
get_list	2	1	1	0.500	1.000
get_structure	6	3	2	0.500	0.667
put_permanent_variable	3	2	2	0.667	1.000
put_temporary_variable	3	2	2	0.667	1.000
put_permanent_value	4	3	2	0.500	0.667
put_temporary_value	3	1	1	0.333	1.000
put_list	3	1	1	0.333	1.000
put_structure	5	2	2	0.400	1.000
unify_permanent_variable	4	3	3	0.750	1.000
unify_temporary_variable	4	2	2	0.500	1.000
unify_permanent_value	5	3	3	0.600	1.000
unify_temporary_value	5	2	2	0.400	1.000
unify_list	2	1	1	0.500	1.000
unify_structure	6	3	2	0.333	0.667
Total micro-step count	67	36	32	—	—
Average micro-step count	3.7	2.0	1.8	0.478	0.889
Average execution time	370.0	333.4	—	—	—

Table VI Performance estimation in append program execution.

	Machine instruction count/LI	Micro-step count/LI	Memory access ratio	Performance (KLIPS)
CHI-I	9	35	0.457	285
CHI-II without compiler optimization	8	20	0.850	294
CHI-II with compiler optimization	3	12	0.917	500

many as that in CHI-I.

The memory access ratio, between memory access count and micro-steps, was measured to evaluate the hardware and architecture improvement effect. The average memory access ratio, in executing 18 typical machine instructions, is 0.889 in the CHI-II processor, while it is 0.478 in the CHI-I. These values indicate that most of the micro-steps, required in CHI-II contain memory access operations as a result of the hardware improvement. When executing the 18 typical machine instructions, 36 micro-steps are required, and 32 micro-steps contain memory access operation.

The average execution time is 333.4 nanoseconds in CHI-II, and 370.0 nanoseconds in CHI-I. CHI-II is superior to CHI-I in the typical 18 machine instructions execution.

In the append program execution, 8 instructions and 20 micro steps are required to process each element in a list in CHI-II, while 9 instructions and 35 micro-steps are required in CHI-I (as shown in Table VI).

CHI-II performance is evaluated to be at 300 KLIPS in the append program execution, while it is evaluated as 285 KLIPS in CHI-I. Moreover, by applying the compiler optimization technique, the CHI-II performance is improved at 500 KLIPS.

8. CONCLUSION

CHI-II has been designed and implemented as a practical desk-side machine. It has improved architecture and hardware functions for realizing a machine superior to CHI-I. Moreover, as a result of improvement in the memory access ratio, applying compiler optimization, CHI-II achieves high performance. In the append program execution, the CHI-II performance is evaluated to be at 500 KLIPS.

The append program can be executed in 8 or 9 micro-steps, if the CHI-II processor can be improved. Then, the authors consider that, if a few megabit SRAM and 100,000 gates could be integrated on a chip, the ideal CHI processor could be realized, resulting in a practical logic-programming language processor with several mega-LIPS.

ACKNOWLEDGMENTS

The authors would like to express their thanks to Shunichi Uchida, Takashi Chikayama and Kazuo Taki (ICOT) for their support to progress this project, and to Minoru Yokota (NEC) for his valuable technical discussions and to all of project members who are engaged in implementing research effort. Thanks are also to Masahiro Yamamoto, Tatsuo Ishiguro and Yasuo Kato (NEC) for their continuous encouragement and valuable advices and support.

REFERENCES

[1] R. Kowalski, "Logic for Problem Solving," North Holland Publishing Co., New York, 1979.

[2] A. Konagaya, et al., "Knowledge Information Processing Language: ShapeUp," *New Generation Computing*, 2, 2, pp. 195-201, Pressed by Ohmsha, Springer Verlag, 1984.

[3] R. Nakazaki, et al., "Design of a High-speed Prolog Machine (HPM)," *Proc. of the 12th Internat. Symp. on Computer Architecture*, June 1985.

[4] S. Habata, et al., "Co-operative High Performance Sequential Inference Machine: CHI," *Proc. of 1987 IEEE Internat. Conf. on Computer Design*, Oct. 1987.

[5] D. H. D. Warren, "An Abstract Prolog Instruction Set," *Tech. Report*, 309, *Artificial Intelligence Center*, SRI International, 1983.

Received November 16, 1987

* * * * * * * * * * * * * * * *

Shin-ichi HABATA received the B.S. and M.S. degrees in electrical engineering from Hokkaido University in 1980 and 1982, respectively. He joined NEC Corporation in 1982, and is now Supervisor of the Computer System Research Laboratory, C&C Systems Research Laboratories. He is engaged in the research and development of computer architecture for knowledge information processing systems.

Mr. Habata is a member of the Information Processing Society of Japan.

* * *

Ryosei NAKAZAKI received the B.S. degree in electronic engineering from Tohoku University in 1973. He joined NEC Corporation in 1973, and is now Supervisor of the Computer System Research Laboratory, C&C Systems Research Laboratories. He is engaged in the research and development of VLSI oriented computer architecture and knowledge information processing systems.

Mr. Nakazaki is a member of the Institute of Electronics, Information and Communication Engineers, and of the Information Processing Society of Japan.

* * *

Akihiko KONAGAYA received the M.S. degree in information science from Tokyo Institute of Technology in 1980. He joined NEC Corporation in 1980, and is now Supervisor of the Computer System Research Laboratory, C&C Systems Research Laboratories. He is engaged in the research and development of advanced computer architecture, next generation programming languages and knowledge information processing systems.

Mr. Konagaya is a member of the Institute of Electronics, Information and Communication Engineers, the Information Processing Society, and the Japan Society for Software Science and Technology.

Atsushi ATARASHI received the B.S. and M.S. degrees in computer science from Tokyo Institute of Technology in 1983 and 1985, respectively. He joined NEC Corporation in 1985. He is engaged in the research and development of advanced computer architecture, next generation programming languages and advanced software development system.

Mr. Atarashi is a member of the Information Processing Society of Japan.

* * *

Mamoru UMEMURA received the B.S. and M.S. degrees in electrical engineering from Keio University in 1969 and 1971, respectively. He joined NEC Corporation in 1971, and is now Research Manager of the Computer System Research Laboratory, C&C Systems Research Laboratories. He is engaged in the research and development of computer architecture for knowledge information processing systems.

Mr. Umemura is a member of the Institute of Electronics, Information and Communication Engineers, the Information Processing Society of Japan, and of the AAAI.

* * * * * * * * * * * * * * * *