ICOT Technical Memorandum: TM-0168

TM-0168

KNOWLEDGE-BASED EXPERT SYSTEM FOR HARDWARE LOGIC DESIGN

真野民男、丸山文宏、林　一司
角田多苗子、川戸信明、上原貴夫
(富士通)

May. 1986

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

論文の題名　「Knowledge-Based Expert System
for Hardware Logic Design」

概要　高水準プログラミング言語Occam で記述したハードウェアの並列動作アルゴリズムを入力すると，CMOS 回路を生成する論理設計エキスパートシステムを開発した。本システムは，第五世代コンピュータ・プロジェクトの一環として，新世代コンピュータ技術開発機構(ICOT)からの受託により開発した。ハードウェアの論理設計では，熟練した設計者により初めて信頼性の高い，効率の良い設計が得られるのであり，これまでのCAD システムに見られる決定的なアルゴリズム処理による設計支援には限界がある。論理型言語の推論機構の上に知識を組み込んだ知識ベースシステムの手法により，この限界を超える論理設計支援を試みた。論理型言語としては，Prolog，推論機構としては，設計の詳細化に前向き推論，条件のチェックには後向き推論を採用し，それらの上に論理設計作業における知識を組み込んだ。本論文では，論理設計に関する知識の具体例を挙げるとともにPrologを使ってそれらをどのように実現したかについて説明する。

発表者名　＊真野　民男　　丸山　文宏　　林　一司　　角田　多苗子
川戸　信明　　上原　貴夫

連絡先　〒211
川崎市中原区上小田中1015番地
㈱　富士通研究所　川崎研究所
情報処理研究部門　システム研究部　第一研究室
真野　民男
℡　044-777-1111 内線　2-6286

以上

KNOWLEDGE-BASED EXPERT SYSTEM FOR HARDWARE LOGIC DESIGN

Tamio Mano, Fumihiro Maruyama, Kazushi Hayashi, Taeko Kakuda,

Nobuaki Kawato, and Takao Uehara

FUJITSU LIMITED

Kawasaki, Japan

ABSTRACT

We have developed a knowledge-based expert system for hardware logic design that designs CMOS circuits from a concurrent algorithms written in the OCCAM high-level programming language. This work was done as part of the activities of the Fifth Generation Computer Systems (FGCS) Project of Japan. Our system aims at supporting the entire design process from a specification to completed CMOS circuits by incorporating the designers' expertise into the computer and utilizing it effectively. Prolog was selected for the implementation/knowledge-representation language. This paper gives examples of the knowledge provided for this system, emphasizing the functional design phase which is heavily dependent on the designers' own expertise. This paper also explains how Prolog is used to express this knowledge, and how the inference engine was created. While constructing this system, we also evaluated Prolog for its effectiveness as an implementation language for a new generation of CAD systems.

1. INTRODUCTION

The FGCS Project has undertaken research on knowledge-based systems. We chose hardware logic design as an application area for the following three reasons. First, the application must be in an area in which human expertise plays a significant role. At present, reliable and efficient hardware logic design can only be done by skilled designers. Therefore hardware logic design is suitable for the investigation of the use of expert knowledge. Secondly, because hardware logic design is synthesis-oriented, which is not similar to any successfully developed analysis-oriented system such as MYCIN, there are many unknown qualities to research. Lastly, since hardware logic design covers a wide range of considerations, many types of knowledge are required.

The implementation language is Prolog, which is also used as the underlying knowledge-representation language. Knowledge representation is an important issue. A single multi-purpose framework for knowledge representation would be simplest. However, as described above, hardware logic design employs various kinds of knowledge, and design data must be represented as well. In addition, human designers switch from one representation to another in the course of their work. For these reasons, we have not adopted any particular existing tool for knowledge representation.

2. SYSTEM OVERVIEW

Our system aims at supporting the entire design process from a specification to completed CMOS circuits by incorporating the designers' expertise into the computer[1][2][3]. This system designs CMOS circuits that conform to the input specification through the interaction with the user as shown in Figure 1. The input specification is a concurrent algorithm described in OCCAM[4] which is a programming language characterized by its treatment of concurrency. The input specification is based on software concepts, such as variables and procedure calls, and does not contain hardware concepts, such as registers and clocks. Therefore, with OCCAM, a designer can create a concurrent algorithm without being familiar with hardware details.

Given a concurrent algorithm, the system performs functional design using expert knowledge and interaction with the user to determine the hardware behavior. The hardware concepts, such as registers and clocks, first emerge in this intermediate design stage. Next, the system designs CMOS circuits to achieve the hardware behavior determined by the functional design process. The system outputs the CMOS basic cells and CMOS functional cells, and their connections. Another program should be used for placement and routing of these cells on a chip.

Our system consists of a functional design phase and a circuit

design phase as shown in Figure 2. Because the system is divided into these two phases, the system is able to flexibly cope with different semiconductor technologies. The functional design phase, the first half of the design process, determines the application of hardware concepts in implementing the concurrent algorithms in OCCAM, and produces the finite-state machine description in DDL[5] using expert knowledge related to the functional design for hardware. Although the hardware behavior is determined at this step, it is independent from the semiconductor technology employed. The circuit design phase, the latter half of the design process, transforms the finite-state description into the desired CMOS circuits. To execute this transformation, the translator subsystem extracts and rearranges information required for the circuit design from DDL. The functional cell design subsystem designs combinational circuits, and the functional block design subsystem allocates cell libraries, such as flip-flops. In the circuit design phase, the technology-dependent knowledge related to circuit synthesis is used.

3. DESIGN EXAMPLE

This section discusses how the system works, taking a pattern matcher proposed by M.J.Foster and H.T.Kung[6], as an example. Figure 3 and Figure 4 show the outline of the pattern matcher. This pattern matcher checks whether a given pattern, which is a fixed length vector of characters, is embedded in a given text string, which is an endless string of characters as shown in Figure 3.

Let us denote the input string stream as $s_0 s_1 s_2 \ldots$, the input finite pattern stream as $p_0 p_1 p_2 \ldots p_k$, and the output result stream as $r_0 r_1 r_2 \ldots$. Characters in the two input streams may be compared for equality, with the wild card character X matching any character in an input stream. The output bit r_i is to be set to 1 if the substring $s_{i-k} s_{i-k+1} \ldots s_i$ matches the pattern, and 0 otherwise. For example, in Figure 4 the pattern AXC matches the substrings $s_0 s_1 s_2$, $s_3 s_4 s_5$, and

$s_4 s_5 s_6$(ABC, AAC, and ACC, respectively).

3.1 Design Specification

The concurrent algorithm of the pattern matcher, which is input to this system, is described in OCCAM, as shown in Figure 6.

The OCCAM program consists of three declaration parts, (A),(B), and (C), and a description of parallel processes, (D). The declaration parts are as follows.

(A) declares channel vectors. A channel vector is a set of channels. Channels are used for communication between concurrent processes. For example, pattern[6] means that there are six channels named pattern, and they are numbered from 0 to 5, as shown in Figure 4.

(B) declares a single comparator process. PROC gives the name comp to this process, and identifies five formal parameters, the internal channels, pin, sin, pout, sout, and dout, as shown in Figure 5. When the named process is substituted for the subsequent process (D), the formal parameters are replaced by the actual parameters. Process comp is a sequential process, which consists of two processes. One is the initialization (B-1), and the other is an endless iterative process, WHILE TRUE. The iterative process contains a sequential process, which consists of two processes. The first two are output and input processes (B-2). The last process compares the characters in the pattern with those in the text string and outputs a Boolean value, TRUE or FALSE (B-3).

(C) declares a single accumulator process. The process named acc contains seven formal parameters, as shown in Figure 5. The variable d stands for the current comparison result of the comparator, xnew and xold for don't care bit, lnew and lold for end of pattern, rnew and rold for final result of matching, and t for temporary result of matching. Process acc is a sequential process, which consists of two processes. One is the initialization of variables (C-1). The other is an iterative process, similar to process comp. The iterative process includes a

sequential process, which consists of two processes. The first includes output and input processes (C-2). The second is a conditional process (C-3). When the end of pattern reaches, an accumulator uses the value t (current temporary result) as the final result, and then resets t to TRUE. Otherwise, it maintains a temporary result t, which is set by the logical expression t := t /\ (xnew \/ d). /\ and \/ stand for boolean AND and OR, respectively. Thus if the current temporary result t is TRUE, and xnew or the current comparison result d is TRUE, then the new temporary result will be set to TRUE.

(D) indicates a 2x5 array of concurrent processes, as shown in Figure 4. The cells at the top are the comparators; the pattern flows from left to right and the string flows from right to left. The bottom cells, accumulators, receive the results of the comparison from above. They maintain partial results, and shift completed results from right to left. Two bits associated with the pattern flow through the accumulators from left to right. One bit is the end of pattern, L . The other is the wild card character, X.

3.2 Functional Design Phase

In the functional design phase, the finite-state machine description in DDL is produced from a given concurrent algorithm. Functional design can be thought of as the design phase that determines which hardware concepts to apply in implementing the concurrent algorithm and describes how the hardware components should behave. This phase is one of the most knowledge-intensive parts, and its function depends heavily on the knowledge which is used. This phase continues with the following processing operations through the interaction with the user as shown in Figure 7.

(1)Analyze the structure of the OCCAM specification. Taking the concurrent descriptions into account, it analyzes the OCCAM construct(WHILE, SEQ, IF, etc.) and creates an outline of the hardware

control mechanism (state transitions).

(2)Implement variables described in OCCAM, using hardware elements(registers, terminals, etc.). Because the OCCAM specification does not provide information about the number of required bits, the system queries the user. However, the system automatically infers that the number of bits for a variable is one if all of its sources turn out to be Boolean values ("TRUE","FALSE", or the evaluation of a logical expression).

(3)Compress the operation sequences. This is an attempt to transform some OCCAM sequential processes into DDL register transfer operations executed in parallel, which improves the performance of the generated hardware.

(4)Determine what hardware resources are to be used for inter-processing communication in OCCAM. Basically the communication is implemented as hand shaking. Signal lines through which data is transferred and signal lines used for synchronous signals are provided.

After implementing OCCAM primitive operations as DDL hardware operations and generating partial DDL descriptions, the system puts these partial descriptions together to complete the final DDL description, as shown in Figure 8.

3.3 Circuit Design Phase

The translator subsystem transforms the DDL finite-state machine description into information for the circuit design process. It gathers and edits conditions for terminal connection, register transfer, and state transition operations. Then the translator subsystem organizes the data in frame-like structures, classified into twelve categories: data about systems, clocks, automata, input signals, output signals, terminals, memories, registers, states, decoders, arithmetic expressions, and logical expressions. Each logical expression is given a unique name, and the occurrences of each logical expression are counted, to prevent their arbitrary duplication by the combinational circuit.

The DDL code shown in Figure 8 contains three register transfer operations of register "r" in automaton "acc". These operations provide the following information:

1) ¦* acc_init *¦ r <- 0.

2) ¦* acc_idle & send1 *¦ r <- rin.

3) ¦* acc_state1 & 1 *¦ r <- t.

where the states whose identifiers are modified by "acc" belong to the automaton "acc". "¦* *¦ ..." stands for an if-clause. Based on the above information, the input circuit of register "r" is designed, as shown in Figure 9.

The automaton design subsystem implements automata having the appropriate states using flip-flops. It designs a control circuit around these flip-flops according to information on state transitions provided by the translator subsystem. Figure 10 shows the resulting circuit design for a pattern matcher that consists of a pair of the comparator and the accumulator.

The size of a combinational circuit that can be created using a single CMOS cell is limited by CMOS technology. Signal delay depends mainly on the number of FETs inserted in series between the power supply line and output line. Therefore, the circuit decomposition subsystem decomposes the combinational circuits so that the number of FETs in series does not exceed the limit imposed by signal delay considerations. The combinational circuit around register "r" is decomposed into four parts as shown in Figure 9.

The functional cell design subsystem implements the decomposed combinational circuits as functional CMOS cells. The optimal layouts of the functional CMOS cells are obtained by a heuristic algorithm[7]. However, if the designer is required to use NAND or NOR gates, then the partial combinational circuit (iv) in Figure 9 results in mediocrity as shown in Figure 11(a). In this case, we take advantage of the property of the CMOS functional cell that physically adjacent gates can be con-

nected by a diffusion area. As a result, the optimal layout is obtained as shown in Figure 11(b). Note that the optimal array is smaller than the basic conventional array by almost 50%.

Components such as registers, memories, decoders, adders, and I/O pins are assembled from a cell library of basic cells. Since these functional cells and library cells are of the same height and have the same power connections and standardized connection points, they can readily be incorporated into existing automated layout systems.

4. INFERENCE MECHANISM AND KNOWLEDGE

The functional design phase, which applies hardware concepts to a given concurrent algorithm, is heavily dependent on the designers' own expertise. It is difficult to automate the functional design by conventional CAD techniques. We automated the functional design by in corporating the designers' expertise into the computer.

This section discusses how the inference mechanisms were created and gives examples of the expert knowledge provided for the functional design phase. We also evaluate Prolog for its effectiveness as an implementation/knowledge-representation language for a knowledge-based logic design system.

4.1 Characteristics of Functional Design

We must take a few points into account in performing functional design by the knowledge-based approach. Functional design usually involves some design revisions before the development of a satisfactory design. In other words, functional design does not go straight from beginning to end. Instead, the designer gradually refines the design while checking the overall specifications and clarifying the entire circuit. At each design step, the designer evaluates the situation and makes appropriate decisions. Therefore, it is extremely important in design to always have a good picture of the relations among design objects. The

knowledge-representation framework should reflect this fact. A good method must be provided for referring to the relations among design objects.

Design situations are frequently changed. Some changes caused by design decisions affect the other design parts. The order in which the decisions are made may significantly affect the design cost and/or the design quality. Therefore, controlling such decision orders, which requires complicated operations, should be expressed without any difficulty.

4.2 Inference Mechanism

4.2.1 Forward chaining

Forward chaining is used as the basic inference mechanism. Logic design is a process of incremental refinement. Incremental refinement comes from successive design decisions; every time a decision is made, the current design development is changed. The process that the designer gradually refines the design and clarifies the entire circuit while referring to the overall specifications can be obtained by causing side effects and changing the design environment on the working memory.

Forward chaining seems to be able to simulate the incremental refinement design process. When the conditions set forth by the premise section of a forward chaining rule are satisfied, the conclusion section is activated and the working memory is updated. Then another rule is activated referring to the updated working memory. Processing continues in this manner. We use the assert(addition of fact) and retract(deletion of fact) functions of Prolog for updating the working memory. The working memory consists of design information, which is represented by about 40 types of Prolog facts. As the design process continues, the working memory is gradually filled with design information.

Figure 12 shows the general format of the forward chaining rule. When the conditions 1, ..., n are satisfied in the premise section, ac-

tions 1, ..., m are activated in the conclusion section. The working memory is updated while causing side effects by the assert and/or retract functions.

4.2.2 Backward chaining

Backward chaining is used for various conditional checks. In checking the conditions set forth in the premise section of a forward chaining rule, only the reference to the working memory is required in some cases. In the other cases, several inference steps are required. When a specified condition is checked, the backward chaining rule whose conclusion section matches to the specified condition is fired. Then, the various conditions set forth by the premise section of the fired rule are checked. In this way, checking can be efficiently performed by selecting only those rules which are related to the conditional check. Backward chaining is already implemented as the execution mechanism of Prolog.

Figure 13 shows the general format of the backward chaining rule. The head of the Prolog clause contains the conclusion, while the body contains the premise section. When conditions 1, ..., n are satisfied in the premise section, condition 0 is concluded. In backward chaining, provocation of rules does not affect the working memory.

4.3 Knowledge Representation

In this section, specific knowledge examples of the functional design phase are presented to show how Prolog is used to express expert knowledge of designers'. Functional design can be regarded as the process that determines hardware concepts according to software specifications in OCCAM. Thus, the knowledge used in this process primarily associates software specifications with hardware concepts.

4.3.1 Knowledge for implementing a variable with hardware elements

Figure 14 shows the knowledge used to implement a variable in OCCAM

with hardware elements. This rule is a forward chaining one for implementing a OCCAM variable Var and states that :

IF:

(1)(2) All sources input from the external processes through the channel are all Boolean values,

(3)(4) and all sources assigned to the Var are all Boolean values,

(5) and other conditions are satisfied,

THEN:

(6) Store in the working memory the information that the variable Var is to be implemented as a 1 bit register.

When a predicate implement_variable is called with the OCCAM variable as the first argument, the clause shown in Figure 14 is activated. If all conditions(1)-(5) are satisfied, the determination(6) that Var must be implemented as a 1 bit register is written into the working memory. This rule is used to implement, for example, variable t in the specification of the pattern matcher.

input_source(Var, Input_sources) is a procedure call. what it means according to the declarative reading of Prolog clauses, becomes clearer; it reads "input source of Var is Input_sources".

4.3.2 Knowledge for compressing sequential operations

Figure 15 shows the knowledge used to check whether the software sequential operations can be transformed into hardware operations executed in parallel, which would increase the performance level of the generated hardware. This rule is a backward chaining one and states that :

IF:

(1)(2) Both are store-type operations such as assignment processes,

(3) and they are processed one after anothers,

(4) and the variables into which the sources are to be stored are different,

(5)(6) and both variables are going to be implemented as registers,

(7) and the variable in the first operation is not referred to in the source of the second operation,

THEN :

two operations in a sequence are compatible, or can be executed simultaneously.

When a predicate compatible is called with two operations as the arguments, the clause shown in Figure 15 is activated. Backward chaining is performed as follows. The conditional check store_operation and implementation are only resolved by referring to the working memory, but the referred_to check activates other rules. When all conditions (1)-(7) are satisfied, the compatibility check is successful.

Using this rule, an OCCAM sequential process within the dashed box in Figure 6 is compressed into DDL register transfer operations, which are executed in parallel, shown within the dashed box in Figure 8. Here, the variables rold and rnew are implemented as a single register r, using the knowledge about merging two registers into one.

The rule "compatible" expresses the relation between two operations and Prolog provides a natural means of expressing such relations. It is convenient to represent essential relations, or hardware concepts, with predicates of Prolog.

4.3.3 Knowledge involving local control

Figure 16 illustrates another forward chaining rule for implementing an OCCAM variable using hardware elements. In particular, this rule changes the order of inference locally. The idea of the predicate implement_variable is as follows; when no clues are provided regarding the number of bits for a variable, and a "similar" variable exists, try to determine the number of bits for the similar one first and use the result. To prevent falling into a loop while a decision about a specific variable is postponed, the list of all postponed variables is stored as the second argument of the predicate, implement_variable. This rule

states that :

IF:

(1) There exists a variable which looks similar to Var,

(2) and the implementation of Another_var has not been postponed,

(3) and Var is added to the list of postponed variables and Another_var is implemented first,

(4) and the number of bits, Bit_width is confirmed,

(5) and other conditions are satisfied,

THEN :

(6) Store in the working memory the information that the variable Var is to be implemented as a Bit_width bit register.

Here, a similar variable is defined as a variable that stores the same type of data. The knowledge that if data of the same type are stored in different variables, it is possible to implement these variables with the same number of bits is precisely expressed in the previous rule. For example, this rule is applicable to variables pnew and snew which appear in PROC comp in Figure 6. As shown in Figure 7, the system attempts to implement the variable pnew first, but the decision is postponed because there are no clues to the number of bits. Next, the system tries to implement the similar variable snew. In this case, there are also no clues to the number of bits for the variable snew. Therefore, the system asks the user. After the variable snew is implemented, the system infers that pnew should have the same number of bits as snew, and requests the user's consent to implement variable pnew. In this way, the regular flow of control is altered by changing the order of tasks locally.

4.4 Effectiveness of Prolog

We evaluated the effectiveness of Prolog as an implementation/knowledge-representation language for a new generation of CAD systems.

While constructing this system, we confirmed that the following

characteristics of Prolog are useful for the construction of a knowledge-based expert system for logic design.

(1) The "predicate represents inter-object relationship" characteristic of Prolog is very efficient for expressing hardware concepts, software concepts, and other high-level concepts.

(2) The characteristic that Prolog code can be declaratively read is suitable for representing the simulation of human inference in design process.

(3) Controlling design orders, which requires complex operations for a conventional programming language, is facilitated by using the Prolog execution mechanism.

5. SYSTEM IMPLEMENTATION AND PERFORMANCE

This system was developed using C-Prolog and runs on the VAX-11/780. The C-Prolog interpreter of the VAX-11/780 runs at approximately 1 kLIPS. Table 1 lists the program sizes of various subsystems and the execution time of the two hardware design examples (CPU time of VAX-11/780). Design example 1 applies to the pattern matcher. Design example 2 applies to the simple microprocessor. The input OCCAM specifications were about 40 and 50 lines, respectively. The functional design phase took 37 seconds of CPU time to generate the DDL description of the pattern matcher and took 13 seconds for the microprocessor. The circuit design phase created the CMOS circuit of the pattern matcher in 11.6 minutes and that of the microprocessor in 13.3 minutes. These circuits correspond to approximately 350 gates for the pattern matcher and 1000 gates for the microprocessor.

6. CONCLUSIONS

We discussed the knowledge-based expert system for hardware logic design implemented in Prolog, with an emphasis on the inference mechanism and knowledge representation of the functional design phase.

Areas for further research are as follows.

First, the working memory needs to be appropriately structured. The working memory currently used in this system is a flat-type that consists only of Prolog facts. All operating procedures for the working memory (Prolog's assert and retract functions are used to update the data) are contained in rules. As a result, it is complicated to manipulate the data in the working memory that may vary with time. The use of ESP is a solution to the problem of structuring the working memory. ESP[8] provides us with time-dependent states and a frame-like structure while retaining essential logic programming language features.

Secondly, the current system does not have a rule-interpreter on Prolog. As previously mentioned, the local control flow can be modified by changing the local task sequence. Global control, however, is much more difficult. Global inference cannot be controlled without developing a Meta-rule interpreter.

Lastly, expert system capability largely depends on the amount of stored knowledge. To improve it, expert knowledge must be repeatedly extracted and restored in the knowledge-base. For this reason, we believe it is necessary to research on knowledge acquisition.

ACKNOWLEDGEMENTS

This work is based on the results of the R & D activities of the Fifth Generation Computer Systems Project. The authors would like to thank Dr. K. Furukawa and Mr. Y. Iwashita of ICOT (Institute for New Generation Computer Technology) for their encouragement and support.

REFERENCES

[1] Maruyama,F.,Mano,T.,Hayashi,K.,Kakuda,T.,Kawato,N. and Uehara,T., "Prolog-Based Expert System for Logic Design," International Conference on Fifth Generation computer Systems 1984(FGCS'84), pp.563-571, Nov. 1984.

[2] Mano,T.,Maruyama,F.,Hayashi,K.,Kakuda,T.,Kawato,N. and Uehara,T., "OCCAM to CMOS Experimental Logic Design Support System," 7th Computer Hardware Description Languages and their Applications (CHDL 85), pp.381-390, Aug. 1985.

[3] Maruyama,F.,Mano,T.,Hayashi,K.,Kakuda,T.,Kawato,N. and Uehara,T., "Logic Design: Issues in Building Knowledge-Based Design Systems," Expert Systems(to appear).

[4] Taylor,R. and Wilson,P.,"OCCAM: Process-oriented language meets demands of distributed processing," Electronics, Nov. 30, 1983.

[5] Dietmeyer,D.L.,"Logic Design of Digital Systems," Allyn and Bacon, 1971.

[6] Foster,M.J. and Kung,H.T., "Design of Special-Purpose VLSI Chips: Example and Opinions," CMU-CS-79-147, 1979.

[7] Uehara,T. and vanCleemput,W.M., "Optimal Layout of CMOS Functional Arrays," IEEE Transactions on Computers, vol.C-30, no.5, May 1981.

[8] Chikayama,T.,"Unique Features of ESP", International Conference on Fifth Generation Computer Systems 1984(FGCS'84), pp.292-298, Nov. 1984.

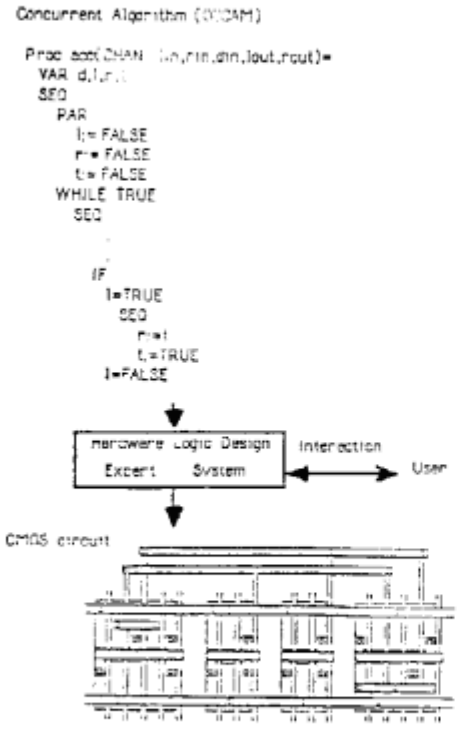

Figure 1 Input and output of the system

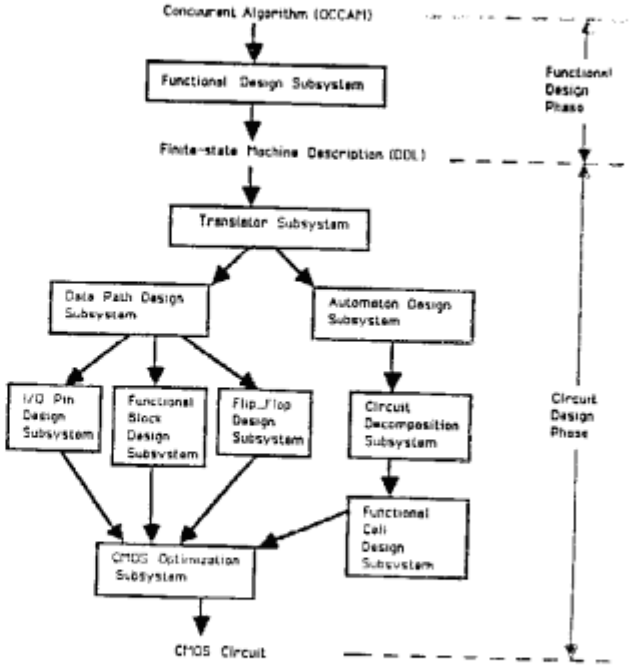

Figure 2. System configuration

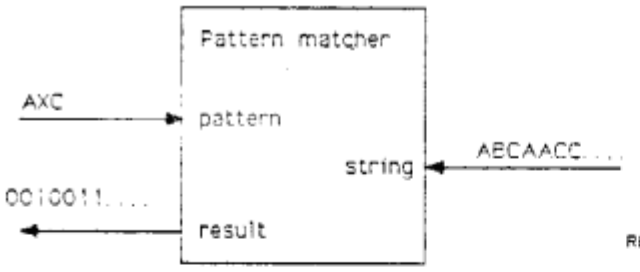

Figure 3. Data to and from the pattern matcher

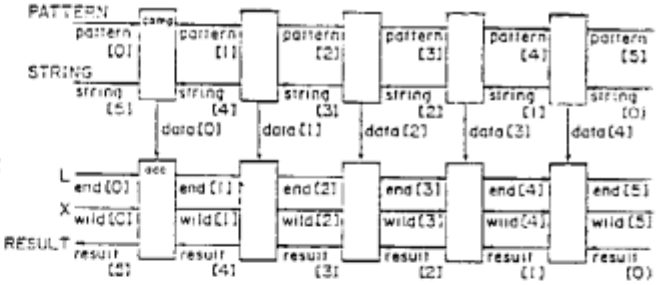

Figure 4. Pattern matcher

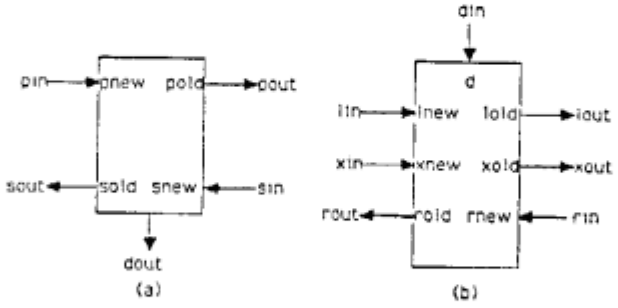

Figure 5. Formal parameters and variables
(a) Comparator (b) Accumulator

```
CHAN pattern[6]:
CHAN string[6]:
CHAN data[5]:
CHAN end[6]:
CHAN write[6]:
CHAN result[6]:                                              (A)

PROC comc(CHAN pin, sin, pout, sout, dout) =
  VAR pnew, pold, snew, sold:
  SEQ
    PAR
      pold := 0                                              (B-1)
      sold := 0
    WHILE TRUE                                               (B)
      SEQ
        PAR
          pin  ? pnew
          pout ! pold                                        (B-2)
          sin  ? snew
          sout ! sold
        PAR
          pold := pnew
          sold := snew                                       (B-3)
          dout ! pnew = snew

PROC acct(CHAN xin, lin, rin, din, xout, lout, rout) =
  VAR d, xnew, xold, lnew, lold, rnew, rold, t:
  SEQ
    PAR
      xold := FALSE
      lold := FALSE                                          (C-1)
      rold := FALSE
      t    := TRUE
    WHILE TRUE                                               (C)
      SEQ
        PAR
          xin  ? xnew
          xout ! xold
          lin  ? lnew                                        (C-2)
          lout ! lold
          rin  ? rnew
          rout ! rold
          din  ? d
        PAR
          xold := xnew
          lold := lnew
          IF
            lnew = TRUE
              SEQ                                            (C-3)
                rold := t
                t := TRUE
            rnew = FALSE
              PAR
                rold := rnew
                t := t /\ (xnew \/ d)

PAR i = [ 1 FOR 5]                                           (D)
  PAR
    comc(pattern[i-1], string[5-i], pattern[i],
                       string[6-i], data[i-1])
    acct(write[i-1],end[i-1],result[5-i],data[i-1],
                       write[i],end[i],result[6-i])
```

Figure 6. Algorithm for the pattern matcher in OCCAM

```
yes
| ?-functional_design

Parsing your specifications in OCCAM ...

Implementing OCCAM variables.....

Should variable rold have only one bit? y/n
|: y
How many bits should variable snew have ?
|: 8
Should variable pnew have as many bits as variable snew? y/n
|: y
Should variable pold  have as many bits as variable pnew? y/n
|: y
Should variable sold  have as many bits as variable snew? y/n
|: y

Compressing a sequence of operations. . .

Implementing inter-process communication.....

Optimizing.....

Enter the name of the register for rnew and rold
|: r

Generating partial ODL descriptions.....

Constructing the final ODL code from partial ODL descriptions...

yes
| ?-
```

Figure 7 Interaction with the user

```
(system) pm,
 (time)   clk,
 (entrance) pin(8), sin(8), xin, lin, rin, din,
 (exit) pout(8), sout(8), dout, xout, lout, rout,
 (terminal) send1,
 (automaton) comp, clk:
  (register) p(8), s(8),
  (states)
    init : p ← 0, s ← 0, → idle,
    idle : pout = p, sout = s,
           p ← pin, s ← sin, → state2,
    state2 : send1 = 1, dout = (p : = s), → idle,
  (end),
 (end) comp,
 (automaton) acc: clk:
  (register) d, x, l, r, t,
  (states)
    init : x ← 0, l ← 0, r ← 0, t ← 1, → idle,
    idle : send1 : xout = x, lout = l, rout = r,
                   x ← xin, l ← lin, r ← rin,
                   d ← din, → state1,
    state1 : [x = d] [t ← t, t ← 1]
                   [] t ← (1 @ (x [] d)), → idle,
  (end),
 (end) acc,
(end) pm.
```

Figure 8. Final ODL description

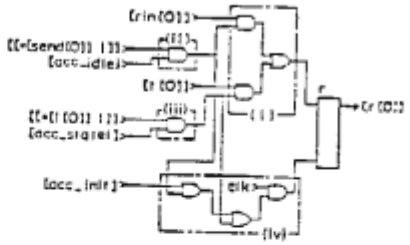

Figure 9. Input circuit of the register r

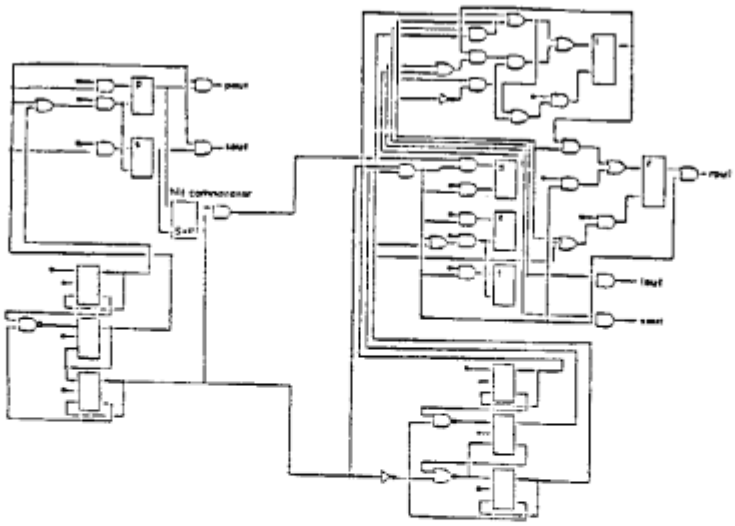

Figure 10. The logic diagram of the pattern matcher

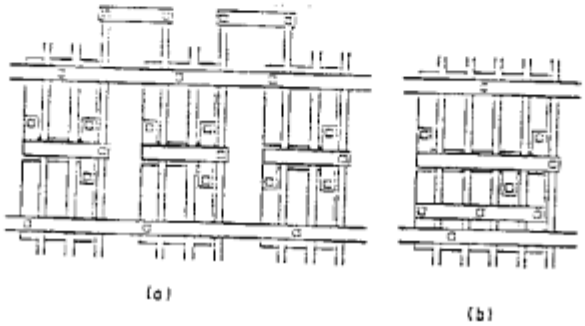

Figure 11. Implementation of the circuit(iv) in Figure 9.
(a) NAND gates (b) Functional cell

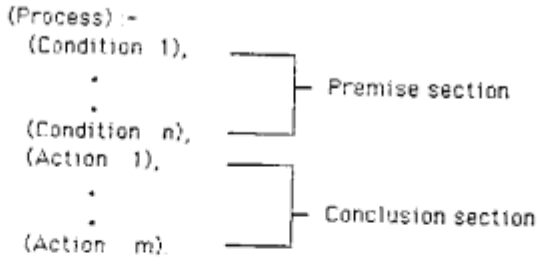

Figure 12. General format of the
Forward chaining rule

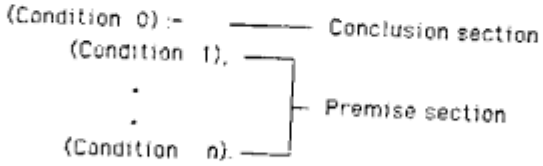

Figure 13. General format of the
Backward chaining rule

```
implement_variable(Var, _):-
      input_source(Var, Input_sources),                    ----(1)
      truth_value(input_sources),                          ----(2)
      assigned_source(Var,Assigned_sources),               ----(3)
      truth_value(Assigned_sources),                       ----(4)
                  .
                  .                                        ----(5)
                  .
      assert(implementation(Var,1,register,...)).          ----(6)
```

Figure 14. Knowledge for implementing a variable with hardware elements (Forward chaining rule)

```
compatible(Operation1, Operation2):-
   store_operation(Operation1,Var1,assign,Source1,....)    ----(1)
   store_operation(Operation2,Var2,assign,Source2,....)    ----(2)
   followed_by(Operation1,Operation2),                     ----(3)
   Var1  \== Var2 ,                                        ----(4)
   implementation(Var1, _, register, ....),                ----(5)
   implementation(Var2, _, register, ....),                ----(6)
   not(refered_to(Var1,Source2)).                          ----(7)
```

Figure 15. Knowledge for compressing sequential operations (Backward chaining rule)

```
implement_variable(Var, Task_list):-
     looks_similar(Var,  Another_var),                     -----(1)
     not(member(Another_var, Task_list)),                  -----(2)
     implement_variable(Another_var, [Var | Task_list]),   -----(3)
     implementation(Another_var, Bit_width, ....),         -----(4)
                  .
                  .                                        -----(5)
                  .
     assert(implementation(Var, Bit_width, register, ..)). -----(6)
```

Figure 16. Knowlede involving local control (Forward chaining rule)

Table 1. Program size and execution time

	Program size (K step)	Execution time(sec)	
		Example1	Example2
Functional Design	1.9	37.0	13.0
Circuit Design			
Translator	0.5	4.0	3.4
Automaton Design	2.1	7.4	10.9
Data Path Design	1.4	9.1	11.1
I/O Pin Design	0.1	4.5	0.2
Functional Block Design	0.2	0.1	1.0
Flip_Flop Design	0.2	5.4	15.0
Circuit Decomposition	1.2	570.6	611.7
Functional Cell Design	0.8	96.2	142.9
CMOS Optimization	0.1	1.6	0.8
Total	8.5	735.9	810.0