TR-093

OCCAM TO CMOS
Experimental Logic Design Support System

Tamio Mano, Fumihiro Maruyama,
Kazushi Hayashi, Taeko Kakuda,
Nobuaki Kawato and Takao Uehara
(Fujitsu Ltd.)

December, 1984

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

OCCAM TO CMOS

Experimental Logic Design Support System

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

Nobuaki Kawato, and Takao Uehara

FUJITSU LIMITED

Kawasaki, Japan

ABSTRACT

Building an expert system to assist in hardware logic design is one of the activities undertaken in the Fifth Generation Computer Systems (FGCS) Project. This paper describes the current status of such an expert system. The system supports the entire design process generating CMOS circuits from a specification in OCCAM.

1. INTRODUCTION

The Fifth Generation Computer Systems(FGCS) Project has undertaken research on knowledge-based systems in some application areas. Among these is hardware logic design.

Logic design involves a variety of aspects ranging from purely algorithmic processing to tasks that are heavily dependent on expert knowledge. This system explores those aspects by incorporating designers' expertise, for instance. A prototype has been implemented in Prolog. In the course of implementation, we evaluated Prolog for its effectiveness as an implementation language for a new generation of CAD system.

Given a concurrent algorithm described in OCCAM, the system designs a CMOS circuit to aid the designer in the logic design process. OCCAM is a programming language characterized by its treatment of concurrency. It enables the user to specify concurrent algorithms with great ease. The result of functional design, the first half of the logic design process, is a finite-state machine description in DDL, a hardware description language. This is the first level at which

the correspondence to hardware concepts emerges. Circuit design, followed by CMOS design, the second half of the logic design process, transforms the finite-state machine description into a CMOS circuit.

Suppose a designer invented a new concurrent algorithm or a parallel processing architecture. At first, he would describe his ideas in OCCAM. Then he could execute the OCCAM as a compiled program on a conventional host computer, and simulate his ideas. He also could make a prototype using many transputers. The prototype allows him to do real time experimentation. If the performance of the prototype is not sufficient, the whole or important parts could be converted into VLSI by our CAD system. The resulting system is physically heterogeneous, but semantically homogeneous since the semantics are OCCAM based.

An experiment transforming an Ada program unit to silicon was reported by a team at the University of Utah(Organick et al. 1984), although the transformation had not yet been fully automated. Our experiment shows that a knowledge-based approach is inevitable to transform a programming language to silicon.

2. OVERVIEW

The system covers the entire design processes from specifications, described in OCCAM (Taylor and Wilson 1983), to complete CMOS circuits. OCCAM is a programming language characterized by its treatment of concurrency. It enables the user to easily specify concurrent algorithms. However, specifications are not necessarily hardware-oriented. In other words, the user is not required to describe specifications based on hardware concepts.

Between the initial stage of OCCAM design specifications and the final stage, in which CMOS circuits are produced, a finite-state machine description in DDL(Dietmeyer 1971) is generated. In this intermediate design stage the correspondence with hardware concepts first emerges. The system's final output is CMOS basic cells, functional cells, and the connections between them.

The system consists of ten subsystems. Figure 1 shows how the subsystems are related to one another. All the subsystems appear in the figure with the ex-

ception of the editor subsystem. The top two subsystems are responsible for functional design. The functional design subsystem determines the application of hardware concepts in implementing the concurrent algorithms, and produces the finite-state machine description in DDL. The state machine optimization subsystem inspects the finite-state machine description and makes modifications to refine it.

The finite-state machine description in DDL is functional, not structural. In order to design circuits, we need information about hardware structure; this means that functional descriptions must be transformed into structural descriptions. Here, the translator subsystem plays its part. It generates two kinds of design information: that concerning data paths and that concerning control circuits.

The control circuit 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 transition supplied by the translator subsystem. The data path design subsystem allocates data paths around functional components, such as registers, memories, adders, and decoders.

Both the data path design subsystem and the control circuit design subsystem generate logical expressions, which are then implemented as combinational circuits using CMOS functional cells. It is not always possible to implement a given combinational circuit using a single functional cell, because large cells fail to meet the high performance requirements. The circuit decomposition subsystem takes a logical expression and breaks it down into subexpressions in such a way that each subexpression can be implemented by a single cell satisfying the performance requirements. These subexpressions are passed to the functional cell design subsystem, which creates a functional cell for each subexpression.

On the other hand, functional components, such as registers, memories, adders, decoders, and I/O pins, are designed by the basic cell assignment subsystem. The subsystem searches the basic cell library for the appropriate cell. If one is found, it is assigned to the hardware component, possibly with slight

modification. Otherwise, the subsystem either assembles a cell using basic cells in the library as components, or it attempts to design one from scratch.

The system provides a facility that optimizes the entire CMOS circuit after the basic cells and the functional cells have been completed. It also provides a user interface facility, which is used throughout the design process under control of the editor subsystem.

In the design process the system explores a variety of design aspects. In the course of this exploration, the techniques applied range from algorithmic approaches to knowledge-based approaches. We describe several of these techniques in the following sections taking the pattern matcher proposed by M. J. Foster and H. T. Kung (Foster and Kung 1979) as an example to show how the system works.

3. DESCRIPTION IN OCCAM

The first thing the user should do is to describe an algorithm in OCCAM.

For example, the algorithm of the pattern matcher is described in OCCAM, as shown in Figure 2. This pattern matcher checks whether a given pattern, which is a fixed length of characters, is embedded in a given text string, which is an endless string of characters as shown in Figure 3. The pattern matcher produces a stream of bits, each of which corresponds to one of the characters in the text string.

Let's denote the input string stream as $s_0 s_1 s_2 \cdots$, the finite pattern stream as $p_0 p_1 p_2 \cdots p_k$, and the output result stream as $r_0 r_1 r_2 \cdots$. Characters in the two input streams may be compared for equality, with the wild card character x considered to match any character in input streams. The output bit r_i is to be set to 1 if the substring $s_{i-k} s_{i-k+1} \cdots s_i$ matches the pattern, and 0 otherwise. For example, in Figure 3, 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).

The OCCAM program consists of declaration parts, (A)(B)(C), and description of parallel processes (D).

Declaration parts are as follows. (A) declares vectors of channels. A channel vector is a set of channels. Channels are used to communicate 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 in the subsequent process, the formal parameters are replaced by the actual parameters. Process comp consists of two sequential processes. One is the initialization of variables p and s, which stand for pattern and string, respectively,(B-1). The other is an endless repetitive process, WHILE TRUE. This repetitive process contains three sequential processes. The first and second processes are output and input processes, (B-2). The last process compares p and s, and outputs the truth 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 variables d, x, l, r, and t stand for the current comparison result of the comparator, don't care bit, end of pattern, final result of matching, and temporary result of matching, respectively. Process acc consists of two sequential processes. One is the initialization of variables, (C-1). The other is a repetitive process, similar to process comp. This repetitive process includes three sequential processes. The first and second processes are output and input processes, (C-2). The last is a conditional process, (C-3). If the value of l is TRUE, that is at the end of pattern, an accumulator uses the value t (concurrent 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 /\ (x \/ d). /\, \/ stands for boolean AND, and OR, respectively. It means that if the current temporary result t is TRUE, and x or the current comparison result d is TRUE, then the new temporary result will be set to TRUE.

(D) means 2*5 array of concurrent process, 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 is the end of pattern, λ. The other is the wild card character, x.

4. FUNCTIONAL DESIGN

Functional design can be thought of as the phase of design that determines which hardware concepts to apply in implementing the concurrent algorithms and describes how the hardware components should behave. The three primary things the functional design subsystem, one of the most knowledge-intensive parts, is supposed to do are as follows:

1) Implement variables described in OCCAM using hardware elements (registers, etc.).
2) Design hardware control mechanisms for "constructs" of OCCAM ("SEQ" for sequential processes, "PAR" for parallel processes, etc.).
3) Implement communication between processes described in OCCAM ("?" for inputting a value from a channel, "!" for outputting a value to a channel).

The end result of the functional design subsystem is a finite-state machine description, which is further refined by the state machine optimization subsystem.

Figure 6 shows how the interaction goes between the user and the system for the previous example. First, the system analyzes the structure of the OCCAM specification, looking at OCCAM's constructs. Then, it determines how to implement OCCAM variables. At this point, questions are asked about the number of bits each variable should have. Since there is an assignment operation that sets variable l to "FALSE", the system tries to allot one bit for it and gets the user's consent. The system cannot figure out how many bits variable s should have, so it asks the user. The user wants the variable to have 8 bits, so he enters 8. The system does not have any direct evidence about variable p, but suspects that variable p should have as many bits as s. In fact, there is an operation in the OCCAM specification that compares s and p. The number of bits

for a variable is determined to be just one if all its sources turn out to be truth values ("TRUE", "FALSE", or a logical expression).

Next, operation sequences are compressed in order to fully utilize the inherent parallelism of hardware. Some sequential processes can be transformed into hardware operations executed in parallel, which gives better performance of the generated hardware.

Here, the rule, a Prolog clause, shown in Figure 7, is used. It reads that two operations in a sequence are compatible, or can be executed at the same time, if : both are store-type operations, such as an assignment process, the variables into which the sources are to be stored are different, the variable in the first operation is not referred to in the source of the second operation. Using this rule, the OCCAM sequential process in (C-3) in Figure 2 is compressed into the following two DDL register transfer operations, which are executed in parallel: r <- t, t <- 1.

The system then implements OCCAM's inter-process communication. It asks whether it can take advantage of overall synchronism by using a single clock. If the user sees no problem in using a single clock, he replies in the affirmative. The system also asks if the user wants high-speed communication rather than steady communication such as hand shaking. If the user wants this, the system tries to implement communication with the transmission and reception timing of communication coordinated.

With inter-process communication implemented, the hardware control mechanisms, including automata and their states, are completely determined. 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.

5. CIRCUIT DESIGN

The translator subsystem transforms the DDL finite-state machine descriptions into design information for the circuit design process. It gathers and edits conditions for terminal connection, register transfer and state transition operations. Then it organizes this data in a frame-like structure, classified

into eleven categories; data about systems, clocks, automata, input signals, output signals, terminals, memories, registers, states, arithmetic expressions, and logical expressions.

All logical expressions are given unique names and the occurrences of each logical expression are counted, to prevent their arbitrary duplication by combinational circuits.

For example, the DDL code shown in Figure 8 includes three register transfer operations of register "r" in automaton "acc". These operations suggest respectively the following information:

1) ¦* acc_init *¦ r <- 0.
2) ¦* acc_idle & send1 *¦ r <- rin.
3) ¦* acc_state1 & 1 *¦ r <- t.

where, the states whose identifier is modified by "acc_" belong to the automaton "acc". "¦* *¦ ." stands for an if-clause.

According to this information, the input circuit of register "r" is produced, as shown in Figure 9 .

The logic diagram of the pattern matcher is obtained, as shown in Figure 10. The top half represents the diagram for the accumulator and the bottom half represents the diagram for the comparator.

The circuit decomposition subsystem breaks down a logical expression into subexpressions in such a way that each subexpression can be implemented by a single cell satisfying the performance requirements.

6. CMOS DESIGN

A CMOS functional cell is produced based on the logical expression which represents the combinational circuit. A functional cell is a strategy for a large-scale accumulation of VLSI. This section discusses the implementation of a random logic function on an array of CMOS transistors. A heuristic algorithm that minimizes the array size is presented.

The basic layout of a functional cell is illustrated in Figure 11, starting

from the AND/OR (sum of products) logical expression. AND/OR gates in the logic diagram correspond to the series/parallel connections in the circuit diagram. It is clear that for every AND/OR expression of a Boolean function, one can obtain a series/parallel implementation in CMOS technology, in which the p-MOS side and n-MOS side are each other's dual.

Physically adjacent gates can be connected by a diffusion area. The layout can be further improved by judicious pairing of sources and drains. Separation is required when there is no connection between physically adjacent transistors. However, the best results are obtained using the alternative circuit shown in Figure 12(b). This circuit is logically equivalent to the one shown in Figure 11(b). Since both the cell height and the basic grid size are functions of the technology employed, an optimal layout is obtained by minimizing the number of separations. Finally, the layout of the functional cell can be optimized as shown in Figure 12(d). This array is smaller than the basic layout by almost 50%.

In order to reduce the array size, it is necessary to find a pair of Euler paths (an Euler path is an edge chain that contains all the edges of the graph model) having the same sequence of labels on the dual graph model, because p-type and n-type gates corresponding to the same input signal have the same horizontal position in the CMOS array. Since the graph-theoretical algorithm to obtain the best solution is exhaustive, the following heuristic algorithm has been proposed (Uehara and vanCleemput 1981):

Step 1) To every gate with an even number of inputs add a "pseudo" input.

Step 2) Add this new input to the gate in such a way that the planar representation of the logic diagram shows a minimal interlace of "pseudo" and real inputs. The vertical order of inputs on the planar logic diagram produces an optimal gate sequence layout. "Pseudo" inputs, except for those at the top or bottom, correspond to separation areas.

The separation areas can be minimized using a logic diagram, which clearly shows the structure of the series/parallel graph.

An algorithm for constructing a minimal interlace is implemented in Prolog.

as outlined in Figure 13. An example of applying the minimal interlace algorithm to the logical expression [or,[and,[or,1,2],[or,3,4]],5] is shown in Figure 14. Figure 14(b) is a conceptual model of the logic diagram shown in Figure 14(a). The black and white triangles correspond to real and "pseudo" inputs, respectively.

Triangles 1, 2 and p1 in subtree T1 are rearranged by the algorithm. The result is represented by a single triangle with a white top and black bottom("white-black"), because the color of the top triangle, p1, is white and the color of the bottom triangle, 2, is black. T2 is similarly represented by a new triangle. A new model is then obtained as illustrated in Figure 14(d). The arrangement of subtree T3 is shown in Figure 14(e). (Note that T3 is represented by a white triangle because the top triangle, p3, is white and so is the bottom of triangle T2.) The final rearrangement of the tree is represented in Figure 14(f). In the end, we obtain a logic diagram with an input sequence characterized by minimal interlacing, as shown in Figure 14(g) [[1,2,4,3],[5]]. This sequence shows the separation between the two sublists. This sequence of transistors gives the optimal layout of the CMOS functional cell.

Let's produce CMOS functional cells from the decomposed combinational circuits around register "r", as shown in Figure 9. Because, (i)(ii) and (iii) parts are typical combinational circuits, they will be prepared in the library of conventional design automation systems although they are generated from scratch by the previous algorithm, as shown in Figure 15. As the (iv) part is not a regular circuit, its CMOS cell is not contained in the conventional library. The implementation of this circuit is shown in Figure 16. (a) is the implementation of the circuit using cells in the conventional library. (b) is the optimal layout using the previous algorithm. The combinational circuit around register "r" is implemented by routing between these four CMOS functional cells.

We have presented a systematic method of implementing random logic functions using functional cells. Components such as registers, memories, decoders, adders, and I/O pins are assembled from a library of basic cells. Since these cells are of the same height, and have the same power connections and standardized connection points, they can be readily incorporated into existing automated

layout systems. The pattern matching chip will be implemented as shown in Figure 17.

7. CONCLUSION

We have developed a CAD system which covers the entire design process from specifications to complete CMOS cells. The implementing language Prolog appears to have the ability both to succinctly express algorithms and to effectively represent knowledge, for the logic design system.

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 of ICOT (Institute for New Generation Computer Technology) for his encouragement and support.

REFERENCES

Organick, E. I. et al. Transforming an Ada Program Unit to Silicon and Verifying Its Behavior In an Ada Environment: A FIRST EXPERIMENT, pp31-49, IEEE Software, Vol. 1, No. 1, 1984.

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

Dietmeyer, D. L. Logic Design of Digital Systems, Allyn and Bacon, 1971.

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

Thomas, D. E. et al. Automatic Data Path Synthesis, COMPUTER, Vol.16, No.12, 1983.

Uehara, T. and vanCleemput, W. M. Optimal Layout of CMOS Functional Arrays, IEEE Trans. Vol.C-30, No.5, 1981.

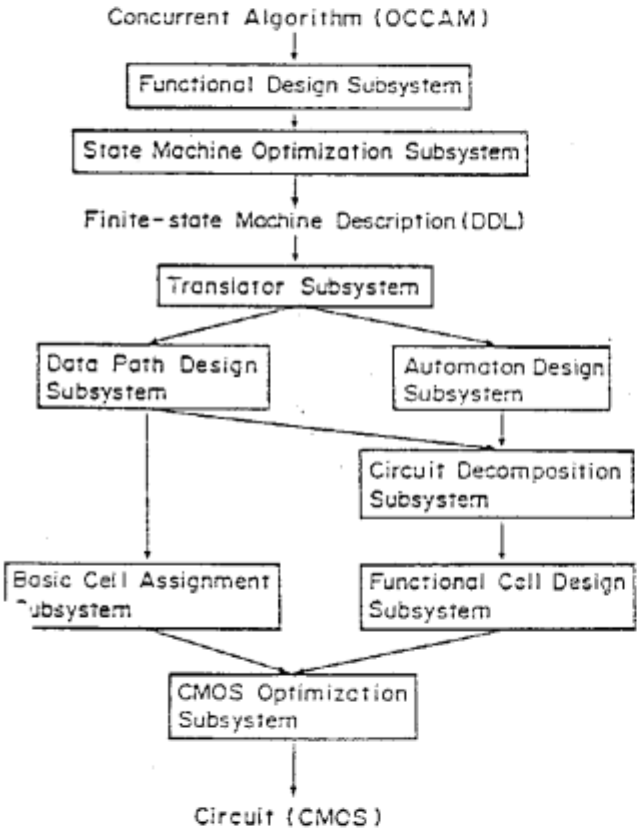


Figure 1. System configuration

```
CHAN  pattern[6]:                                                    }
CHAN  string[6]:                                                     }
CHAN  data[5]:                                                       } (A)
CHAN  end[6]:                                                        }
CHAN  wild[6]:                                                       }
CHAN  result[6]:                                                     }
PROC  comp( CHAN pin, sin, pout, sout, dout ) =                      }
  VAR  p, s:                                                         }
  SEQ                                                                }
    PAR                                                              }
      p := 0                                                         } (B-1)
      s := 0                                                         }
    WHILE  TRUE                                                      }
      SEQ                                                            }
        PAR                                                          }
          pout ! p                                                   }
          sout ! s                                                   } (B-2)
        PAR                                                          }
          pin ? p                                                    }
          sin ? s                                                    }
        dout ! p = s:                                                } (B-3)
PROC  acc( CHAN xin, lin, rin, din, xout, lout, rout ) =             }
  VAR  d, x, l, r, t:                                                }
  SEQ                                                                }
    PAR                                                              }
      x := FALSE                                                     }
      l := FALSE                                                     } (C-1)
      r := FALSE                                                     }
      t := TRUE                                                      }
    WHILE  TRUE                                                      }
      SEQ                                                            }
        PAR                                                          }
          xout ! x                                                   }
          lout ! l                                                   }
          rout ! r                                                   } (C)
        PAR                                                          }
          din ? d                                                    } (C-2)
          xin ? x                                                    }
          lin ? l                                                    }
          rin ? r                                                    }
        IF                                                           }
          l = TRUE                                                   }
            SEQ                                                      }
              r := t                                                 } (C-3)
              t := TRUE                                              }
          l = FALSE                                                  }
            t := t /\ ( x \/ d ):                                    }
PAR  i = [ 1 FOR 5 ]                                                 }
  PAR                                                                }
    comp( pattern[i-1], string[5-i],                                 } (D)
          pattern[i], string[6-i], data[i-1] )                       }
    acc( wild[i-1], end[i-1], result[5-i],                           }
         data[i-1], wild[i], end[i], result[6-i] )                   }
```

Figure 2. Algorithm for the pattern matcher in OCCAM

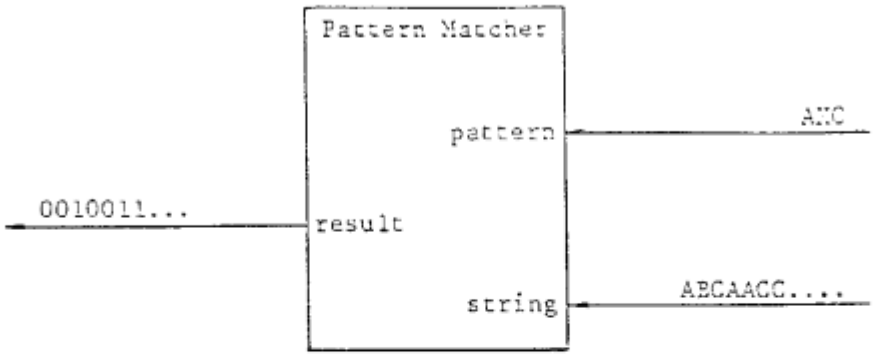


Figure 3. Pattern Matcher

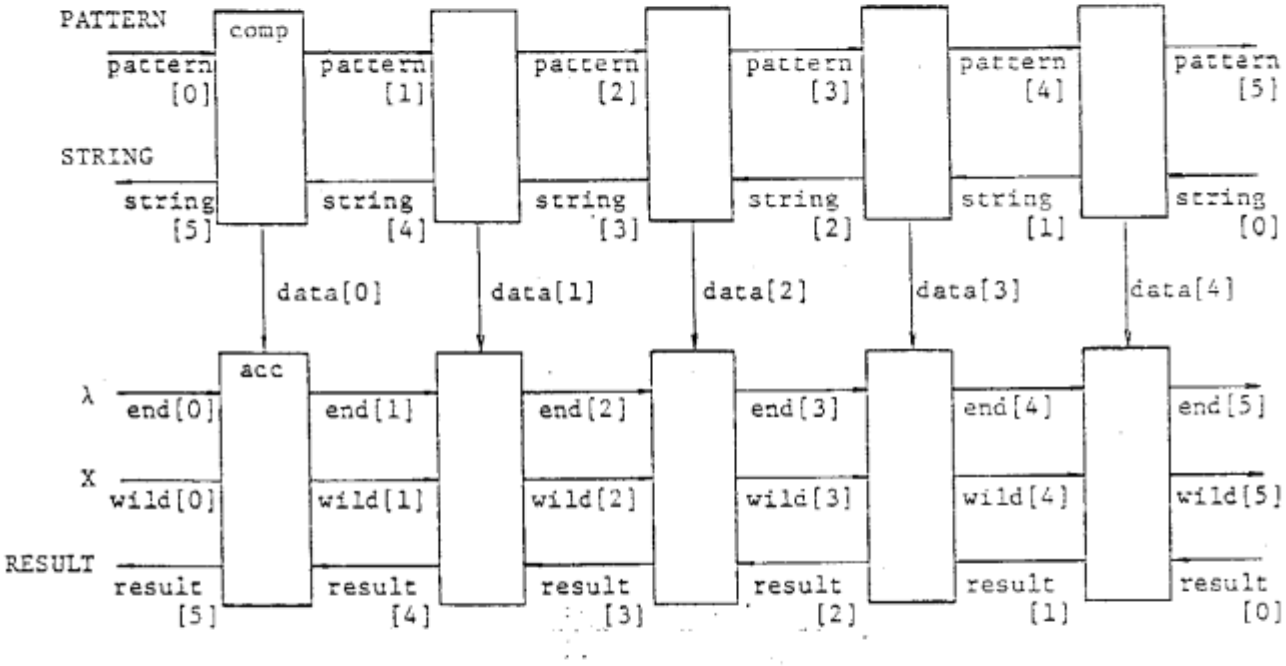


Figure 4. Dataflow of the pattern matcher

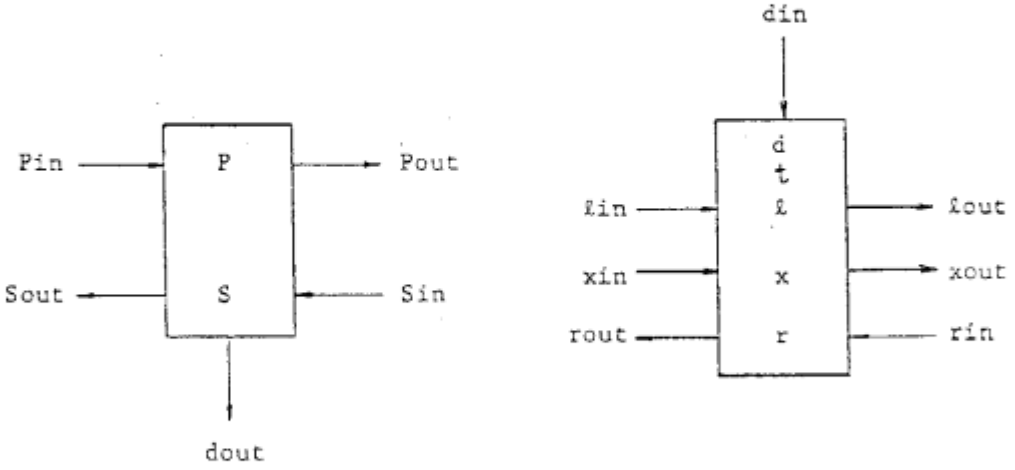


Figure 5. Formal parameters of processes

(a) Comparator (b) Accumulator

```
yes
| ?- functional_design.

Parsing your specifications in OCCAM ...


Implementing OCCAM variables ...


Should variable l have only one bit? y/n
|: y

How many bits should variable s have?
|: 8

Should variable p have as many bits as variable s? y/n
|: y

Should variable r have only one bit? y/n
|: y

Should variable x have only one bit? y/n
|: y

Compressing a sequence of operations ...


mplementing inter-process communication ...


Can the entire system be controlled by a single clock? y/n
|: y

Would you prefer faster overall communication? y/n
|: y

Generating partial DDL descriptions ...


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


yes
| ?-
```

Figure 6. Interaction with the user

```
compatible (Operation_1, Operation_2):-
  store_typed_operation (Operation_1, Variable_1, Source_1, ...),
  store_typed_operation (Operation_2, Variable_2, Source_2, ...),
  Variable_1 \== Variable_2,
  implementation (Variable_1, resister, .....),
  implementation (Variable_2, register, .....),
  not (appear (Variable_1, Source_2)).
```

Figure 7. Compression rule in Prolog

```
<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: |* l *|  r <- t,  t <- l
                     ;  t <- ( t & ( x | d )).,  -> idle.
    <end>.
  <end> acc.
<end> pm.
```

Figure 8. Final DDL description

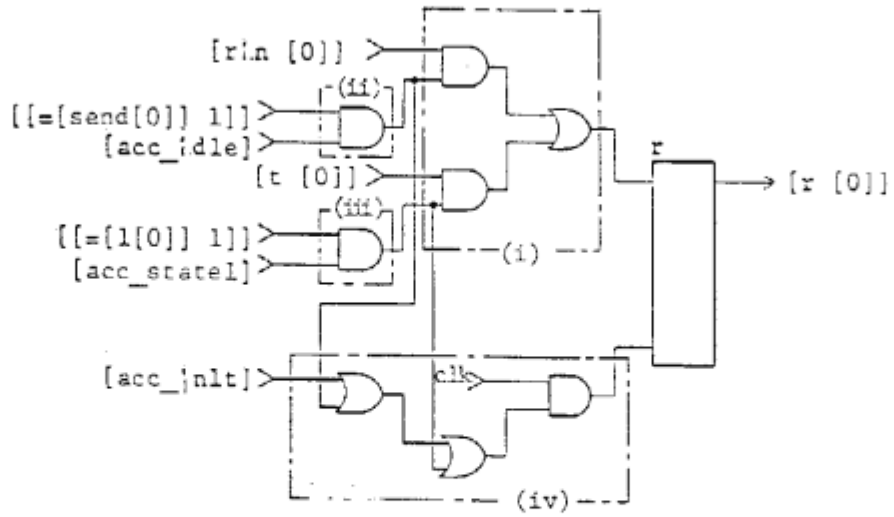


Figure 9. Input circuit of register r

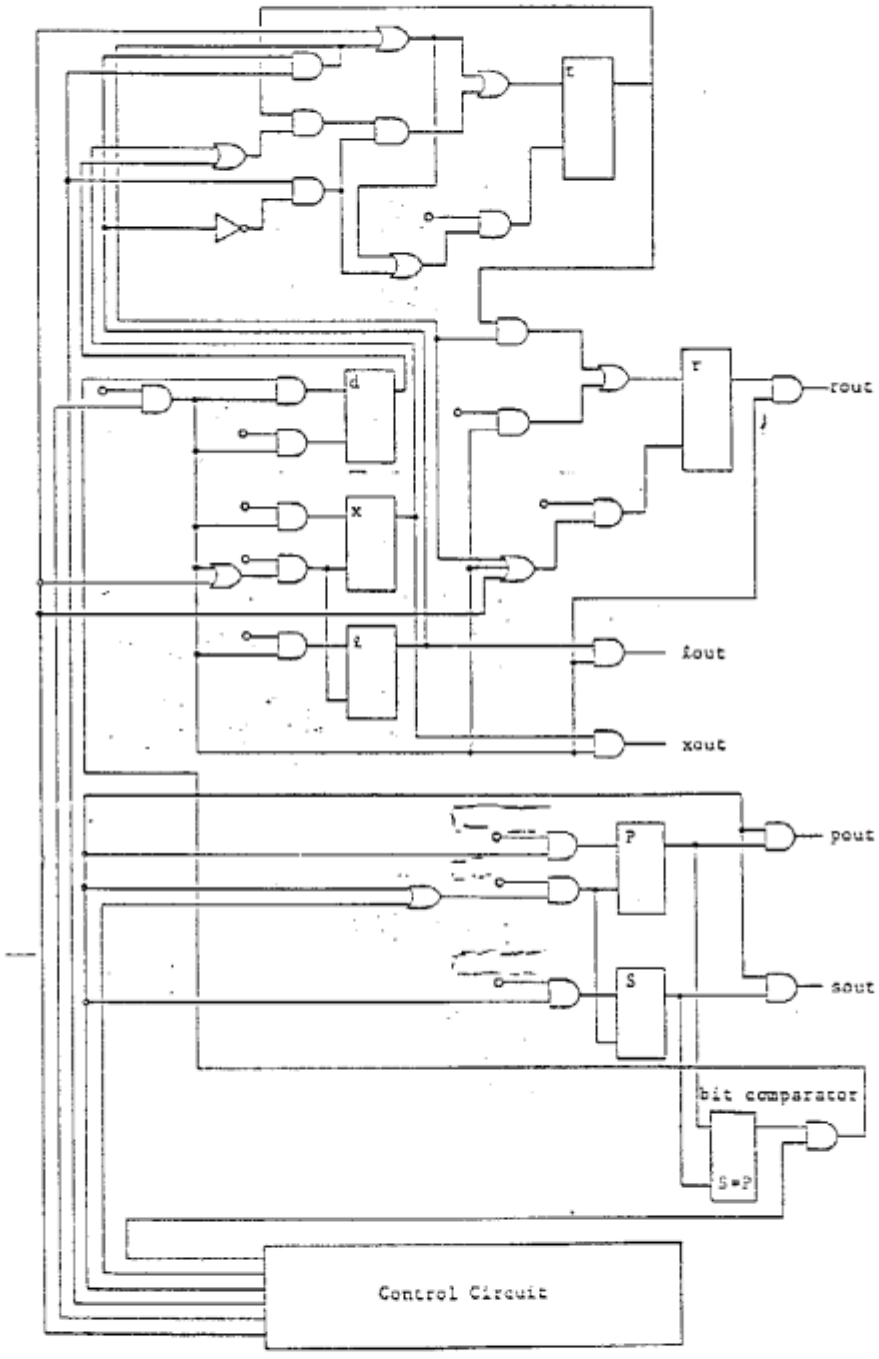


Figure 10. The logic diagram of the pattern matcher

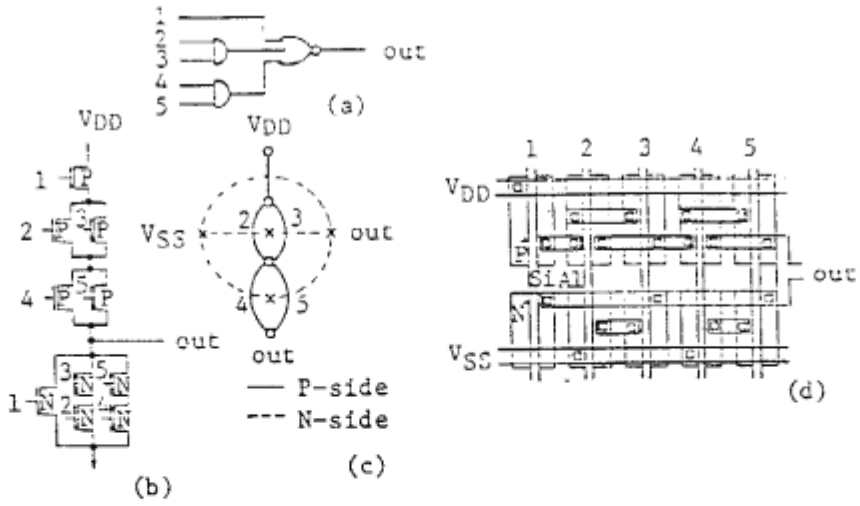


Figure 11. Basic layout of the functional cell
(a) Logic diagram (b) Circuit (c) Graph model (d) Layout

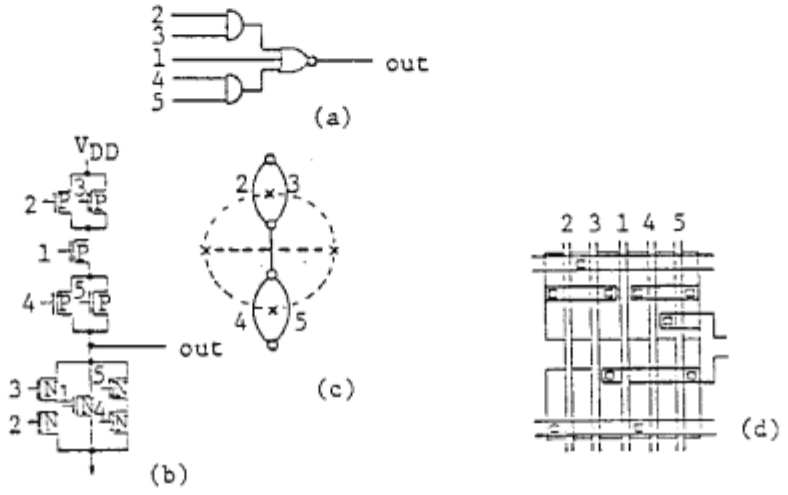


Figure 12. An alternative circuit and optimal layout
(a) Logic diagram (b) Circuit (c) Graph model (d) Layout

minimal:-

white_select:-
Select every white triangle and put it in the list

one_black & white_select:-
Select only one black & white triangle and put it in the list with the white part on top.

black_select:-
Select every black triangle and put it in the list.

rest_black & white_select:-
Add the remaining triangles to the list in such a way that their top parts alternate (black, white, black, white, etc.).

color_set:-
Determine the list color from ① W-LIST, ② WB-LIST, ③ B-LIST and ④ LOG-EXP2.

Sample list

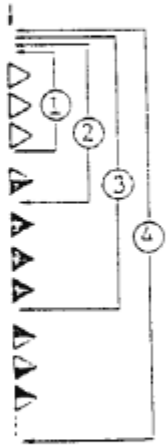


Figure 13. Minimal interlace algorithm

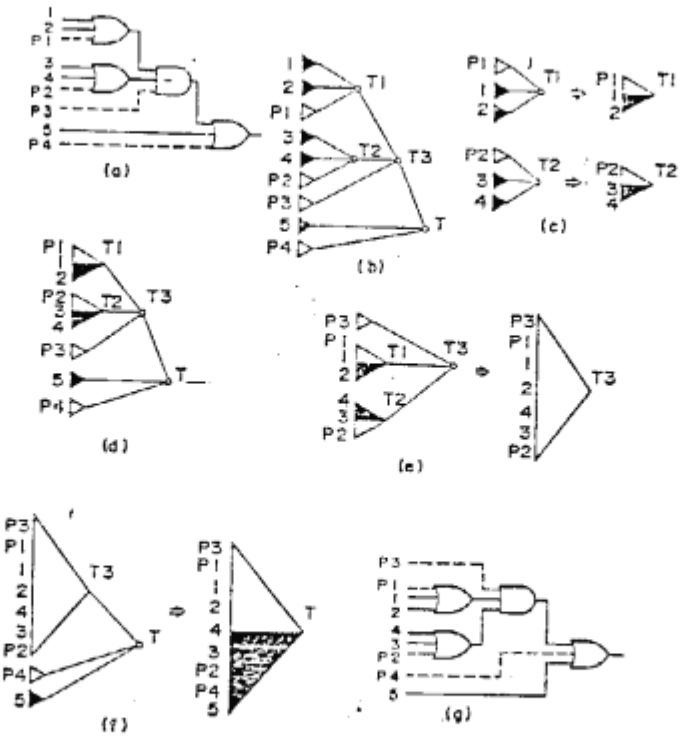


Figure 14. Example of applying the minimal interlace algorithm

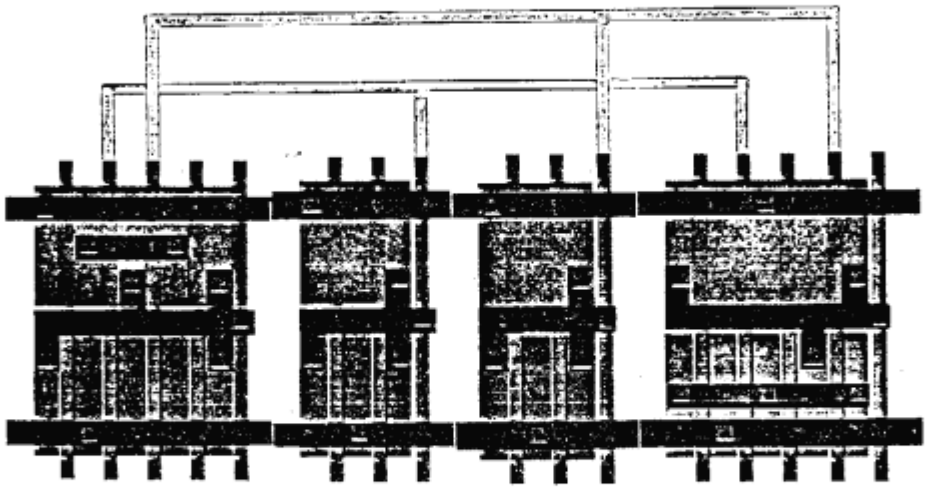
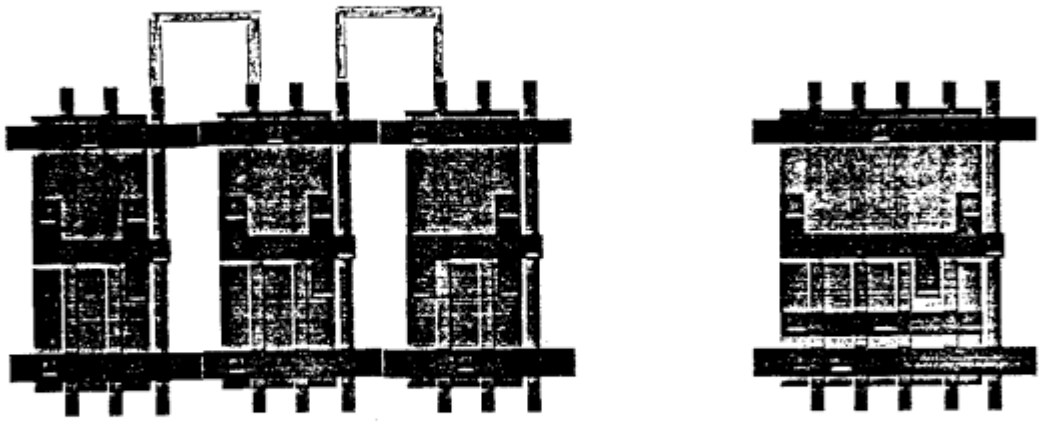


Figure 15. Implementation of the circuit in Figure 9.



(a) (b)

Figure 16. Implementation of the circuit (iv)

(a)NAND gates (b)Functional cell

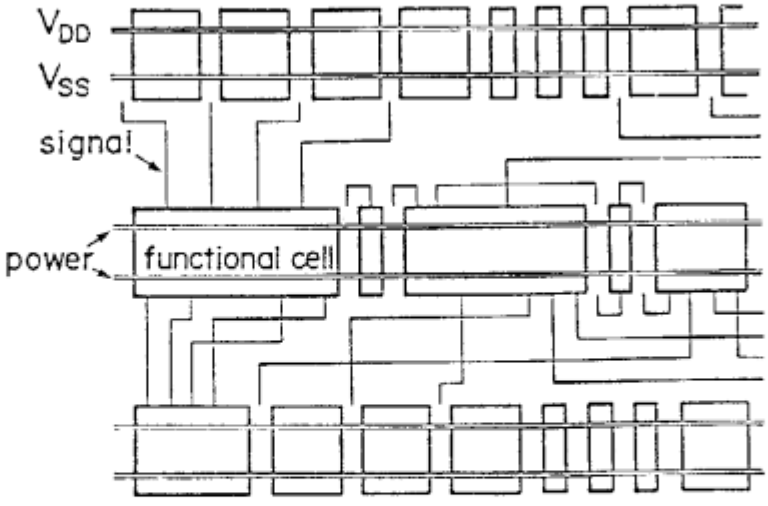


Figure 17. Example of a row-based layout scheme