TR-098

Logic Design: Issues in Building Knowledge-Based Design Systems

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

December, 1984

Logic Design: Issues in Building Knowledge-Based Design Systems

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

Nobuaki Kawato, and Takao Uehara

FUJITSU LIMITED

Kawasaki, Japan

ABSTRACT

This paper presents a Prolog-based expert system for hardware logic design and discusses some of the issues involved in building knowledge-based design systems. Issues of the design task have been pointed out in many places (e.g. Hayes-Roth et al. 1983). We raise some problems that we found significant while building the system, and explain how we solved them or what is needed to solve them in the logic programming environment.

The system manages the entire design process from specification to completed CMOS circuits. The specification is a concurrent algorithm described in occam, 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 with 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.

This work is being done as part of the activities of the Fifth Generation Computer Systems (FGCS) Project of Japan.

1. INTRODUCTION

The Fifth Generation Computer Systems (FGCS) Project of Japan is researching applications of knowledge-based systems as well as more fundamental research in areas such as knowledge acquisition. One of the many applications is hardware logic design. Our target was a design system, rather than a diagnosis system. There are a number of successful knowledge-based diagnosis systems, including medical systems, but few knowledge-based design systems are in practical use. We believe that building good design systems would provide a key to a wide variety of future applications.

We are not saying that current computers do nothing for the task of design. Introducing computer-aided design methodology into digital system design has greatly improved design reliability. It is almost impossible to design a large-scale computer without using other computers. However, design quality is another story. It would be difficult for a fully-automated design system to achieve a design as efficient as that of a human designer; the designer's expertise plays a crucial role. That is why we are taking the knowledge-based approach.

Previous work in this area includes the Palladio system, developed at Stanford University (Brown et al. 1983). Palladio was an attempt to create an integrated design environment. Its main concern was to provide compatible design tools, ranging from simulators to layout generators, to permit specification of digital systems from architecture to layout, in compatible languages. It was also to enable explicit representation, construction, and testing of such design tools and languages.

We investigated the logic design process and broke it down into subprocesses according to the flow of designers' task. After studying each subprocess carefully, we determined how closely knowledge is related to each task. For example, the functional design process entirely depends on the designer's experience. We wanted the system to cover the entire design process, even the functional design process.

Another objective is to evaluate the logic programming paradigm. As Bobrow (Bobrow 1984) stated, no single paradigm is appropriate to all problems. What we would like to do is to see whether we can build effective knowledge-based design systems based on the logic programming paradigm, by looking intensively at the application area of logic design.

2. SYSTEM OVERVIEW

The system covers the entire design process, from specifications described in occam to completed CMOS circuits. Occam is a programming language characterized by its treatment of concurrency (Taylor and Wilson 1982). It enables the user to easily specify concurrent algorithms. However, specifications do not have to be 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 generated, a finite-state machine description is generated in DDL, a hardware description language (Dietmeyer 1971). 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. All the subsystems appear in the figure with the exception 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. However, to design circuits, we need information about hardware structure; functional descriptions must be transformed into structural descriptions. Here, the translator subsystem comes into play. It generates design information on data paths and on control circuits.

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 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 automaton 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 performance requirements. The circuit decomposition subsystem takes a logical expression and breaks it down into subexpressions so 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.

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 an appropriate cell. If one is found, it is assigned to the hardware component, possibly with slight modifications. Otherwise, the subsystem either assembles a cell from the basic cells in the library, or it attempts to design one from scratch.

The system provides a facility that optimizes the entire CMOS circuit after the basic and 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 following sections, we describe key points of each of the three design phases and discuss several design issues.

3. FUNCTIONAL DESIGN

Functional design is the phase of design in which it is determined which hardware concepts to apply in implementing the concurrent algorithms and how the hardware components should behave. There are three primary things the functional design subsystem, one of the most knowledge-intensive parts, is supposed to do.

1) Implementing variables described in occam using hardware elements (e.g. registers).
2) Designing hardware control mechanisms for "constructs" of occam (e.g. "SEQ" for sequential processes, "PAR" for parallel processes).
3) Implementing 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.

As an example, the concurrent algorithm of the pattern-matching chip proposed by M. J. Foster and H. T. Kung (Foster and Kung 1979) is shown in Figure 2.

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

Figure 3 shows the conversation between the user and the system for this example.

```
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 ...


Implementing 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 ...
```

Figure 3 Interaction in functional design

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, it asks questions 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 the variable and gets the user's confirmation. The system cannot determine 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 infers 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.

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 increases the performance level of the generated hardware. Figure 4 shows a sample rule, compatible.

```
compatible(Operation1,Operation2):-
   store_operation(Operation1,Var1,Source1,...),
   store_operation(Operation2,Var2,Source2,...),
   followed_by(Operation1,Operation2),
   Var1 \== Var2,
   implementation(Var1,register,...),
   implementation(Var2,register,...),
   not(referred_to(Var1,Source2)).
```

Figure 4 Rule "compatible" (in Prolog)

This rule reads that two operations in a sequence are compatible, or can be executed simultaneously, if: both are store-type operations, such as assignment processes, the variables into which the sources are to be stored are different, both variables are to be implemented as registers, and the variable in the first operation is not referred to in the source of the second operation. Using this rule, an occam sequential process such as

```
SEQ
  r := t
  t := TRUE
```

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 decides to use a single clock, he confirms the system request. The system also asks if the user wants high-speed communication rather than steady communication such as hand shaking. If the user does, the system tries to implement communication with coordinated transmission and reception timing.

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. For example, the DDL description of the pattern-matching chip is generated as shown in Figure 5.

```
<SYSTEM> pm.
  <TIME> clk.
  <ENTRANCE> pin(8), sin(8), xin, lin, rin.
  <TERMINAL> pout(8), sout(8), dout, xout, lout,
             rout, 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, ->state1.
      state1: 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<-dout, ->state1.
      state1: |* l *| r<-t, t<-1
                     ; t<-(t&(x|d))., ->idle.
    <END>.
  <END> acc.
<END> pm.
```

Figure 5 DDL description

4. CIRCUIT DESIGN AND CMOS DESIGN

Circuit design stands between functional design and CMOS design, and provides all the information necessary for designing CMOS functional cells and assigning basic cells. In the CMOS design phase the information, which is technology-independent, is used to generate CMOS cells and their connections.

4.1 Translator and Data Path Design

The translator subsystem transforms the DDL finite-state machine descriptions into circuit design information. It gathers and edits conditions for terminal connection, register transfer, and state transition operations; it then organizes the data in a frame-like structure, as illustrated in Figure 6.

```
register: r
    bits: [0,0]
    automaton: acc
    source: [0, rin, t]
    condition: [acc_init, acc_idle∧send1, acc_state1∧1]
          .
          .
          .
```

Figure 6 Translated result in a frame-like structure

We see that register r has only one bit, 0th bit. Also, register r belongs to automaton "acc" and has three transfer operations; for the first operation, the source is 0 and the condition under which it is executed is acc_init, and so on.

All logical expressions are given unique names to prevent them from arbitrarily duplicated by combinational circuits. The number of occurrences of each logical expression is taken into account to determine which logical expression to implement as a CMOS functional cell.

Once the translator generates the information just described, the data path design subsystem is ready to design circuits around functional components such as registers. The logic diagram around register r in Figure 6 is generated as shown in Figure 7.

4.2 Automaton Design

An automaton is a finite-state machine for controlling data facilities. Each automaton assumes one of its states in each cycle, which is defined as the period between two adjacent clock pulses. The automaton design subsystem designs a control circuit around flip-flops, which implement states, according to information about state transition supplied by the translator. If possible, it reduces the number of states, as shown below.

Figure 8 shows the state diagram of a simple computer, in which the address is set in state ADS, the instruction is fetched in state IFT, the operation code OP is decoded in state DEC, and the next state is determined by the setting of OP. Coding those eight states into three flip-flops would give us a control circuit shown in Figure 9, where the high-order six bits of register IR, the instruction register, constitute the operation code. However, we can build a 4-state 2-flip-flop machine with states ADS, IFT, DEC, and a new state, EXC, instead of the 8-state 3-flip-flop machine. When this 2-flip-flop machine is in the EXC state, the actual state is determined by the setting of the high-order six bits of the instruction register. The state diagram is greatly improved, as is the corresponding control circuit shown in Figure 10.

The following four conditions must be satisfied to enable state reduction:

1) A state must have two or more branches (branching state), and the transition between the state and each of its subsequent states must be indicated by a single register.

2) Each subsequent state must have no other predecessor.

3) When the transition occurs, the register value must not be affected.

4) In each subsequent state, there must be no recursive transition

that would change the contents of the register.
Our knowledge base contains this rule in Prolog.

4.3 Design of Functional Components

There are two extreme automatic allocators for functional components: a distributed allocator and a central allocator (Thomas et al. 1983). The distributed allocator adds a new functional component for each unique reference in the functional description. However, this design is inefficient. The central allocator tries to map all references onto a structure with a single functional component. While such an approach might be adequate for a simple computer, it is not optimal for a large system.

Our system is capable of determining whether to add a new functional component or to map the reference onto a structure with a single functional component on an individual basis, by checking whether a component can be requested for more than one operation at the same time.

4.4 Functional Cell Design

The functional cell design subsystem implements a random logic function on an array of CMOS transistors. Figure 11 shows the basic layout of a CMOS functional cell. AND/OR gates in the logic diagram correspond to series/parallel connections in the circuit diagram, in which the P-MOS side and N-MOS side complement each other. Physically adjacent gates can be connected by a diffusion area. Separation is required when physically adjacent transistors are not supposed to be connected. Since the cell height and the basic grid size depend on the technology used, an optimal layout is obtained by minimizing the number of separations.

The best results are obtained by using the alternative circuit shown in the Figure 12. The circuit is logically equivalent to the one in Figure 11. As may be noticed, separation is related to whether there is an Euler path on the graph model. We adopted a heuristic algorithm (Uehara and vanCleemput 1981) based on the fact that if every gate has an odd number of inputs, there is a pair of Euler paths, one for the P-side and one for the N-side, having the same sequence on the dual graph model. The algorithm specifies to add one "pseudo" input to every gate that has an even number of inputs, and then to change the vertical order of inputs on the logic diagram so that the interlacing of "pseudo" and real inputs is reduced to a minimum, because "pseudo" inputs, except for those at the top or bottom, indicate separation areas. The resulting vertical order of inputs gives an optimal gate sequence layout. We found that complicated as it is, the heuristic algorithm can be succinctly expressed using Prolog.

5. ISSUES IN BUILDING KNOWLEDGE-BASED DESIGN SYSTEMS

The purpose of this section is to discuss some of the issues involved in building knowledge-based design systems, that characterize the task of design.

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. In other words, a good method must be provided for referring to the relations among design objects. The rule, "compatible", in Figure 4 is a good example; it expresses the relation between two operations and Prolog provides a natural means to express this relation. It would be a good idea to represent essential relations, or concepts, more generally, with predicates of the logic programming language and build a design system based on them. The logic programming paradigm constitutes a "relation-oriented" or "concept-oriented" paradigm.

As Brown (Brown et al. 1983) stated, circuit 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. Since forward chaining seems to be able to simulate this process, it is the principal mechanism for the task of design. Forward chaining operates in cycles, from the initial specification until the design is completed; in each cycle, changes are made to the working memory when a rule is fired. Yet some backward chaining is also necessary for checking conditions like "compatible". In backward chaining, provocation of rules does not affect the working memory. Backward chaining is already implemented as the execution mechanism of Prolog. The problem here is that how to implement the working memory for forward chaining. The working memory must be able to record data that may vary with time. Also there must be a good way to structure the working memory. Prolog's assert and retract functions are not sufficient. ESP (Chikayama 1984) is a solution to this problem. ESP, not only as a logic programming language, but also as an object-oriented language, provides us with time-dependent states and a frame-like structure, retaining essential logic programming language features.

Control is also an important issue concerning design decisions. Among other things, which decision to make first is significant, because one decision may make another decision unnecessary or at least make it easier. Figure 13 shows a rule for implementing variables in the functional design phase.

```
implement_variable(Var,Task_list):-
   looks_similar(Var,Another_var),
   not(member(Another_var,Task_list)),
   implement_variable(Another_var,[Var|Task_list]),
   implementation(Another_var,Bit_width,...),
                  .
                  .
                  .
   assert(implementation(Var,Bit_width,...)).
```

Figure 13 Rule for implementing variables (in Prolog)

This is a rule for implementing variables using hardware elements. It deals with a particular case, in which the bit number of a variable is determined based on the result of implementing another variable. The rough idea is that if there are few clues to the number of bits for a variable and there is another "similar" variable, try to determine the number of bits for the similar one first and use the result. There is a twist; in order not to fall into a loop, predicate "implement_variable" takes the second argument, which is the list of all the variables that have been put off. In this case, the regular flow of control is altered by changing the order of tasks locally. Global control should be considered at a higher level.

A useful aspect of a logic programming language is its declarative reading. An example is shown in Figure 14, where setof is a predicate that gives the list of all solutions to a problem, the second argument, with respect to the variable of the first argument.

```
idle_state_candidate(State,Automaton):-
  setof(X,(state_candidate(X,Automaton),
           in_loop(X),passive_state(X)),[State]).
```

Figure 14 Rule for finding the idle state (in Prolog)

This implies that if there is only one passive state in an automaton (Automaton), that is visited repeatedly, the state can be thought of as the idle state of the automaton (Automaton). The idea behind this is that an automaton executes a series of operations according to what it has been told, returns to what is called the idle state, and waits there for a signal from the outside indicating the next operation to be executed. We say the idle state is a passive state, because it takes action after receiving an external signal. The above rule can read declaratively as follows: If the set

$\{x \mid state\_candidate(x,Automaton) \wedge in\_loop(x) \wedge passive\_state(x)\}$ is a singleton, the only element (State) can be thought of as the idle state. Predicates "state_candidate", "in_loop", and "passive_state" are defined somewhere else.

Another issue is how to express spatial or topological characteristics. There are a lot of spatial and topological characteristics involved in circuit design. Human designers can recognize and use them with no difficulty, but computers cannot. Here lies a distinction in performance between human designers and computers. We believe it is essential that those characteristics be represented in explicit terms. In the circuit design phase, for example, the automaton design subsystem must recognize that an automaton has a ring-like state diagram. The following rule is used to reduce the number of states in the automaton design subsystem.

```
branching_state(State):-
   single_predecessor(State),
   not(single_successor(State)).
```

Figure 15 Rule for recognizing a branch (in Prolog)

This rule recognizes a branch as in Figure 8. It is an example of the way how spatial and topological characteristics are represented with other concepts. A network of concepts will be built up in this way.

As we have just seen, the logic programming paradigm provides a "concept-oriented" or "relation-oriented" paradigm. The important thing in building knowledge-based design systems is to prepare all the necessary concepts and to build a system based on those concepts. For this reason, we think it is necessary to research knowledge acquisition, in order to acquire concepts and construct a network of these concepts.

6. CONCLUSION

We have implemented a prototype of a Prolog-based expert system for logic design. It even covers functional design, which has usually been done only by human designers. This experience has revealed to us useful aspects of the logic programming paradigm and also suggested directions for further research and improvement.

ACKNOWLEDGEMENTS

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

REFERENCES

Hayes-Roth, F., Waterman, D., and Lenat, D. Building Expert Systems, Addison-Wesley, 1983.

Brown, H., Tong, C., and Foyster, G. Palladio: An Exploratory Environment for Circuit Design, COMPUTER, Vol.16, No.12, 1983.

Bobrow, D. If Prolog Is the Answer, What Is the Question?, pp.138-145, FGCS'84, 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.

Chikayama, T. Unique Features of ESP, pp.292-298, FGCS'84, 1984.

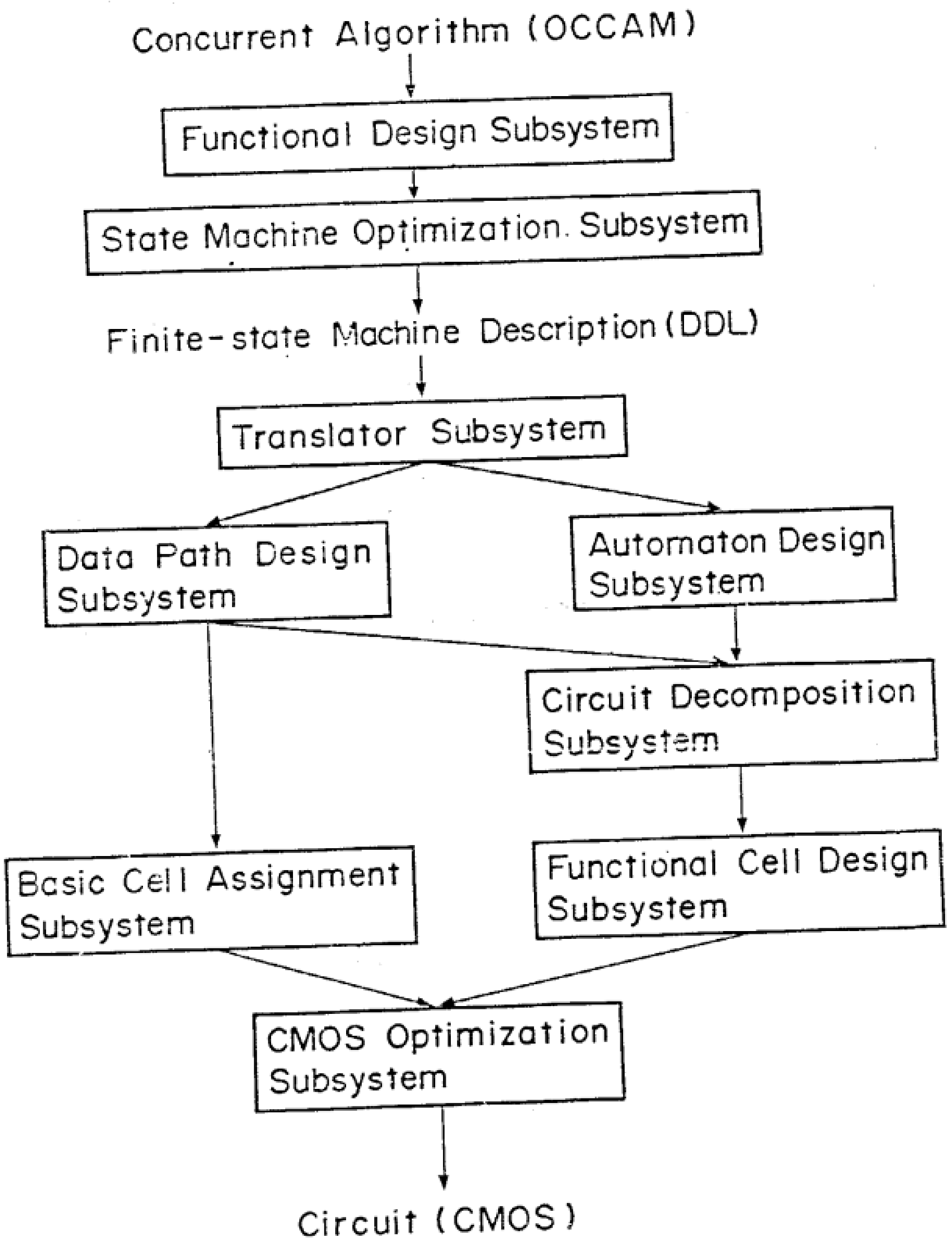

Concurrent Algorithm (OCCAM)
Functional Design Subsystem
State Machine Optimization Subsystem
Finite-state Machine Description (DDL)
Translator Subsystem
Data Path Design Subsystem
Automaton Design Subsystem
Circuit Decomposition Subsystem
Basic Cell Assignment Subsystem
Functional Cell Design Subsystem
CMOS Optimization Subsystem
Circuit (CMOS)

Figure 1. System configuration

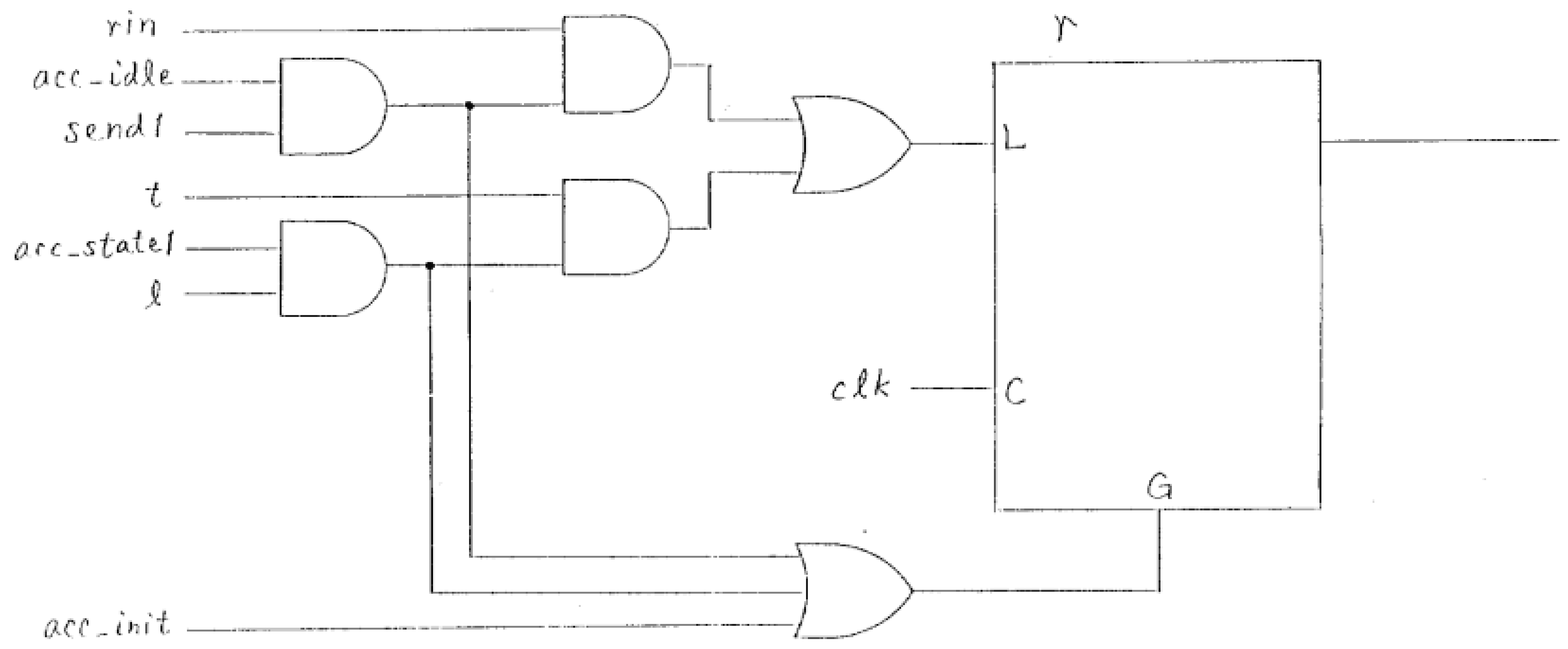


Figure 7 Logic diagram around register r

State diagram

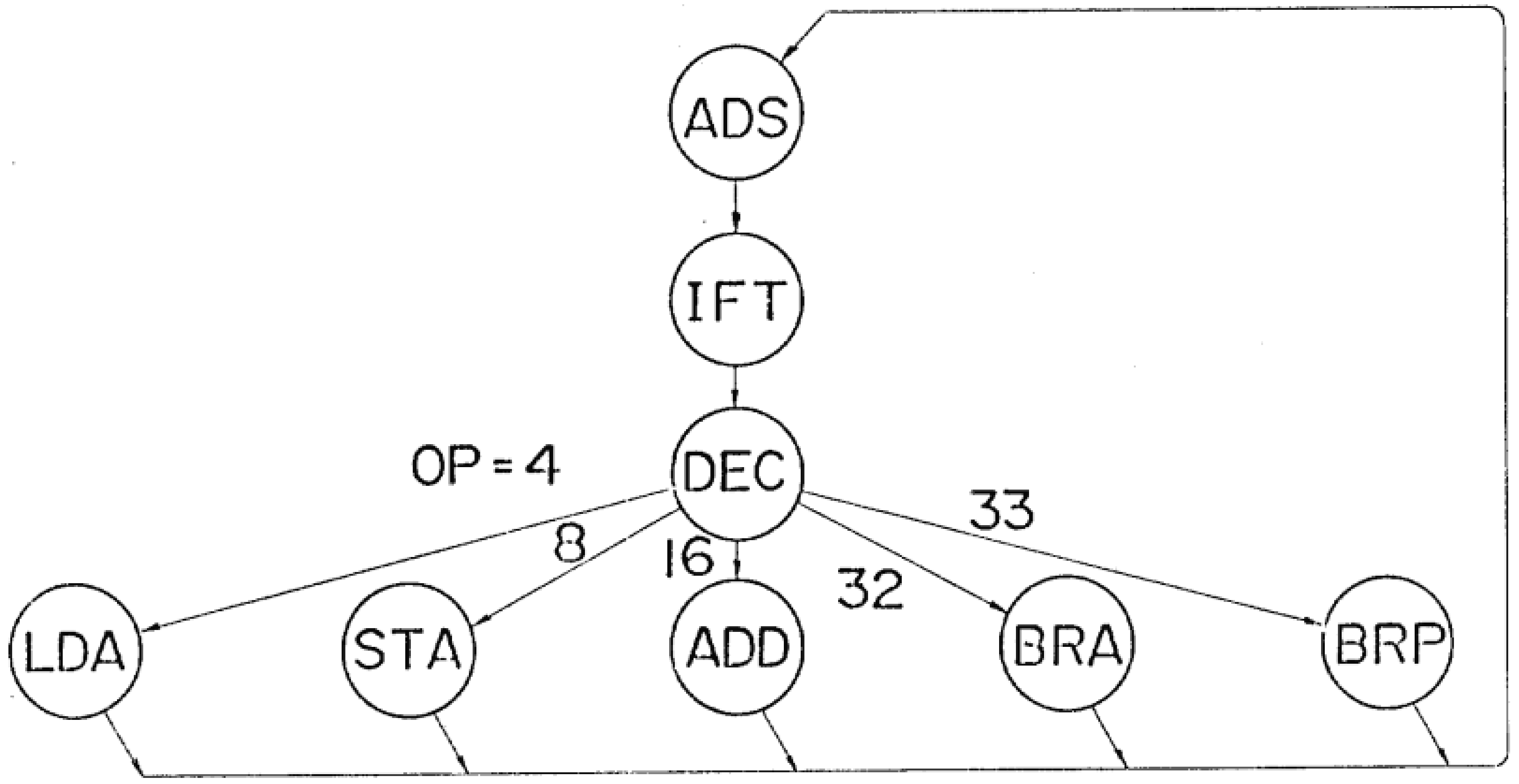


Figure 8 State diagram of a simple computer

Control circuit

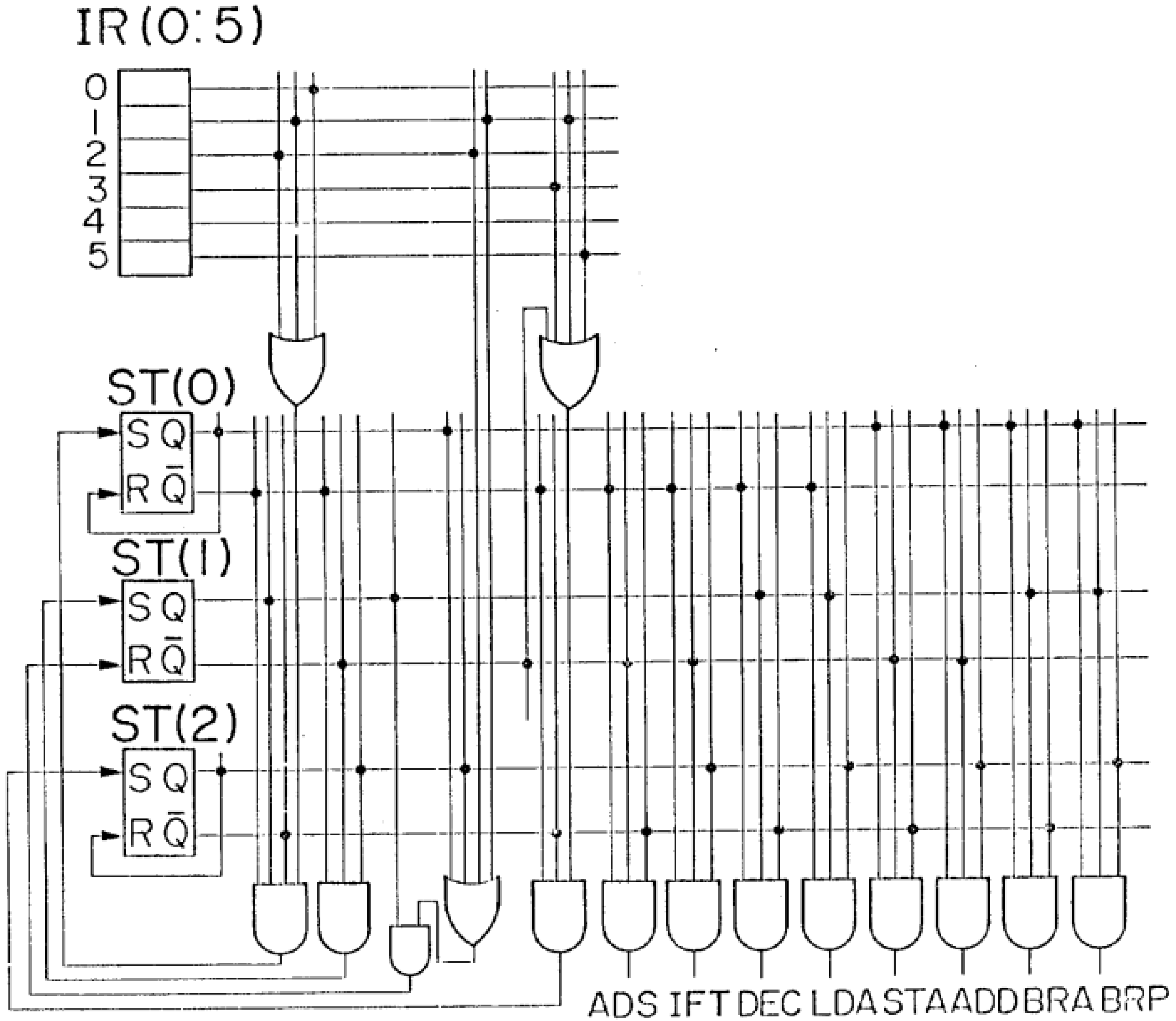
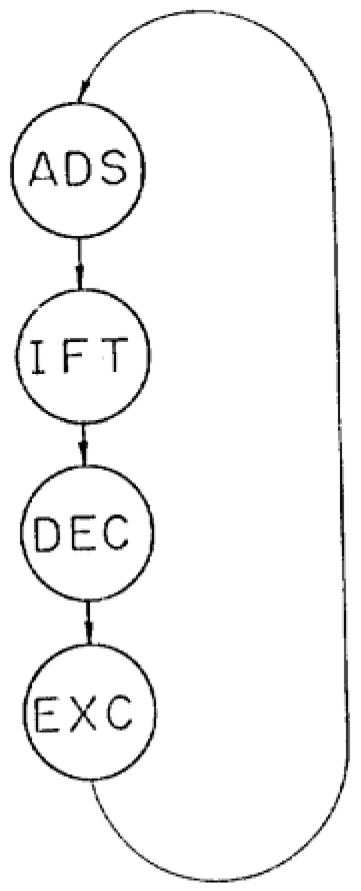


Figure 9 Control circuit

Improved state diagram and control circuit



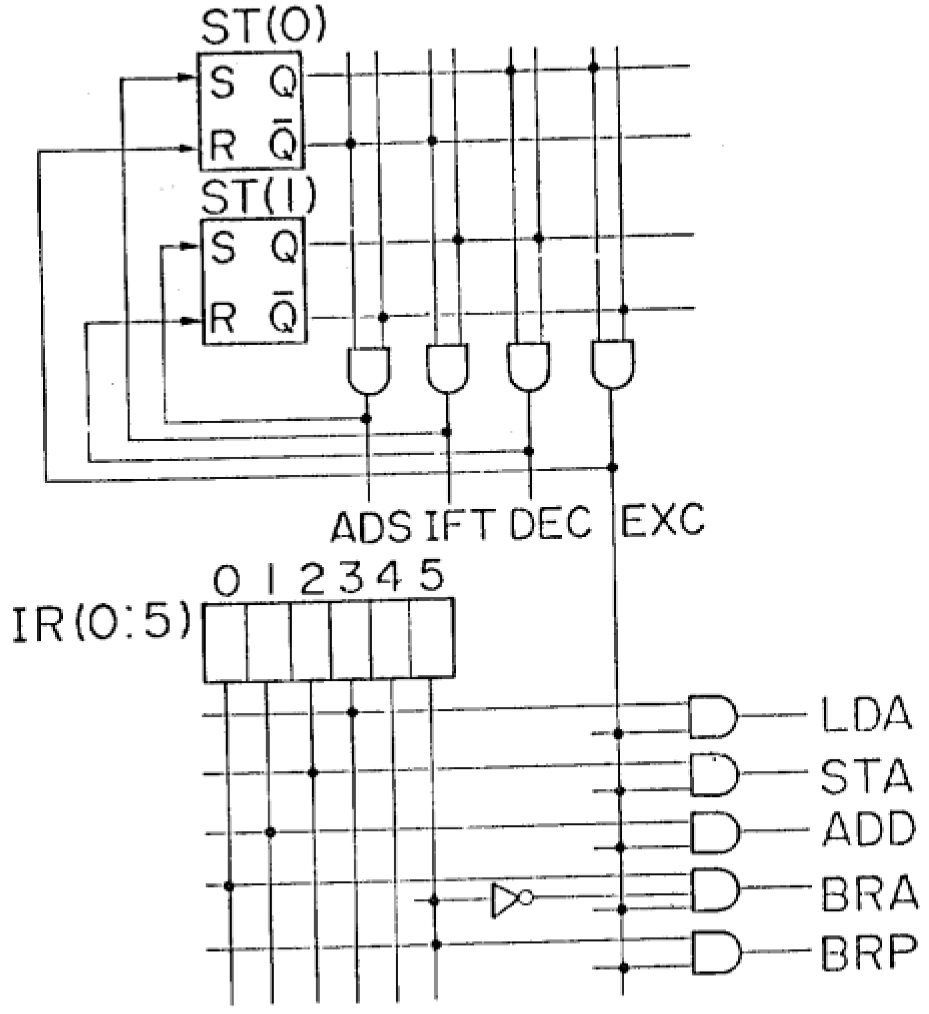
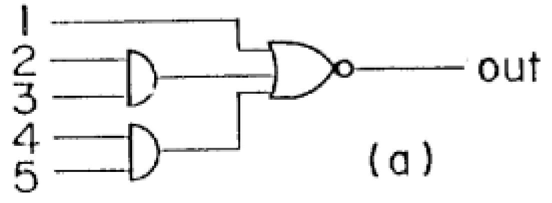
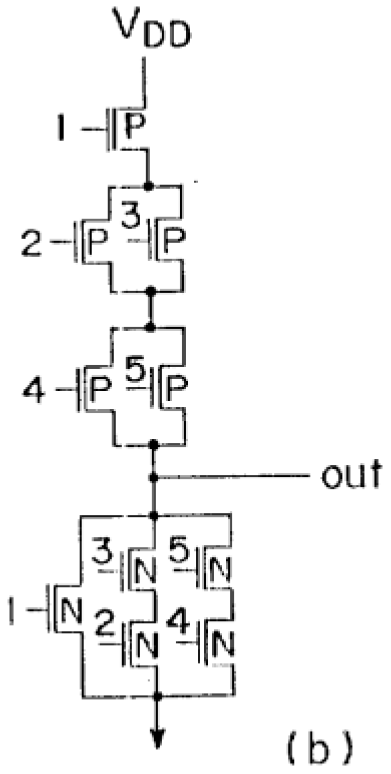
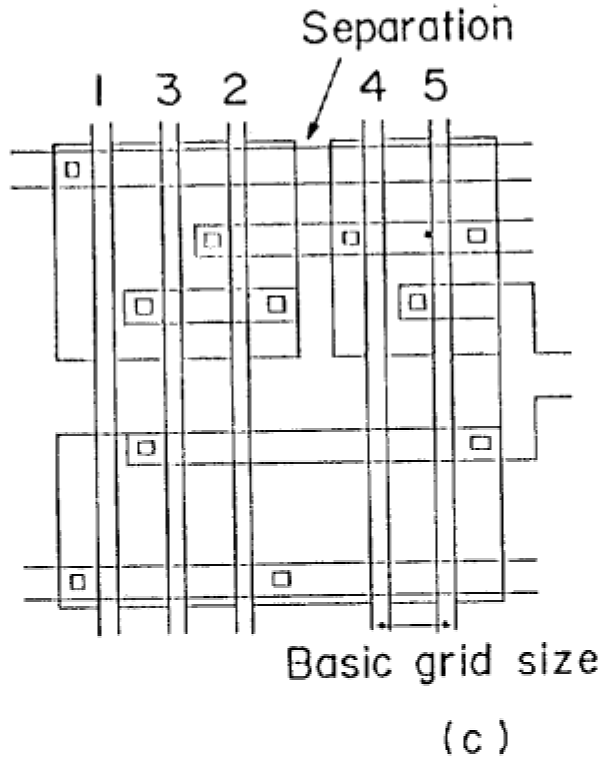


Figure 10 Improved state diagram and control circuit

Basic layout of the functional cell



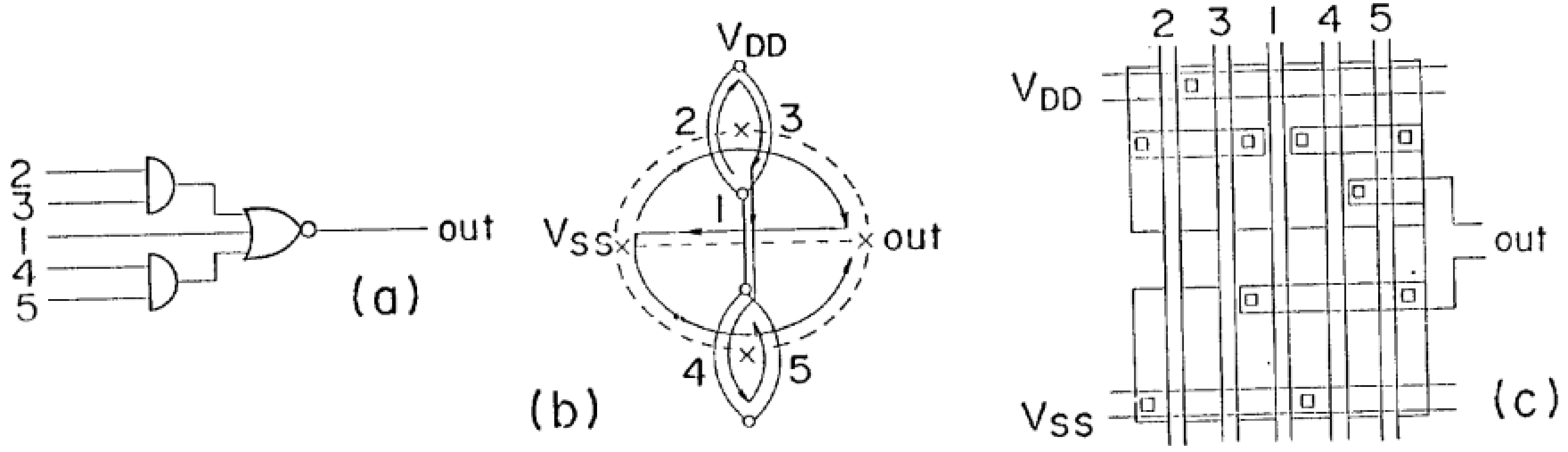




(a) Logic diagram (b) Circuit. (c) Layout.

Figure 11 Basic layout of the functional cell

An alternative circuit and optimal layout



(a) Logic diagram (b) Graph model (c) Layout

Figure 12 An alternative circuit and optimal layout