TORKU, KOFI EMMANUEL
FAULT TEST GENERATION FOR SEQUENTIAL
CIRCUITS: A SEARCH DIRECTING HEURISTIC.

THE UNIVERSITY OF OKLAHOMA, PH.D., 1979

University
Microfilms
International 300 N. ZEEB ROAD, ANN ARBOR, MI 48106

THE UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

FAULT TEST GENERATION FOR SEQUENTIAL CIRCUITS: A SEARCH DIRECTING HEURISTIC

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

DOCTOR OF PHILOSOPHY

BY

KOFI EMMANUEL TORKU

Norman, Oklahoma

1979

FAULT TEST GENERATION FOR SEQUENTIAL CIRCUITS:

A SEARCH DIRECTING HEURISTIC

APPROVED BY

Ben M. Huey

W. T. [illegible]

Abbas Rafii

DISSERTATION COMMITTEE

ACKNOWLEDGEMENTS

At the end of an effort to produce a dissertation, I owe special credits to many people: more than just another page of acknowledgements.

Dr. B. M. Huey deserves special thanks, as my major advisor and dissertation director. He got me interested in this area of research and gave a tremendous personal and professional support; in fact, this is truly our dissertation. To Drs. W. T. Cronenwett, J. A. Payne, A. Rafii and H. J. Kumin I am grateful for their consideration and encouragement during the preparation of this dissertation.

I am also indebted to Dr. C. R. Haden, former Director of the School of Electrical Engineering for his personal encouragement.

This research is based on the SCIRTSS project begun at the University of Arizona. We are grateful for permission to use their circuits for testing.

The partial financial support I received from Kumasi University is appreciated.

Finally, my deepest gratitude goes to my wife, Regina, for her loving support; and to my daughters, Agatha and Amy for their lost evenings.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ABSTRACT

The use of Petri nets to model the register transfers and change of control states in a sequential machine described in a Computer Hardware Description Language (CHDL) with the aim of guiding state space searches is proposed. Each fault to be detected defines a set of goal nodes for the state space search. These goal nodes together with a CHDL description of the circuit are used to generate a Petri net. Some portions of this process are invariant with respect to the goal states, depending entirely on the CHDL description.

Two guidance mechanisms are derived from the petri net: heuristic cost value and input vector guidance. For each machine state encountered during the state space search, a state vector is derived from the petri net. A heuristic cost value is computed based on the state vector; this cost value being a measure of the effect of reaching one machine state in the state space search on the transitions in the petri net. The petri net also contains information about input vectors that are associated with each control state. The most important of these are selected based on an established criteria. The heuristic cost value and the input vectors are used to guide sensitization searches in the Sequential Circuit Test Search System (SCIRTSS). Four case studies are presented to test the effectiveness of the guidance mechanism. The results show that the developed model is a promising tool that can be used in fault test set generation in complex sequential circuits.

CHAPTER I

INTRODUCTION

The advent of integrated circuits and very large scale integration has made fault detection in digital systems a complex process. The number of states inside integrated circuit chips has grown: one chip may have hundreds of flip flops, hence the state space has become much larger. It is no longer possible to provide additional test points brought to external connections due to packaging limitations. This pin limitation constrains the number of outputs observable and inputs to control the sequential circuit inside. The concepts of _controllability_ and _observability_ are important in understanding this problem. Control refers to the ability to apply a complete set of tests to a subsystem via external inputs, that is, control points. Observation refers to the ability to observe the outputs of a subsystem via external outputs, that is, observation points. If many flip flop outputs were observable and controllable, faults could be easily detected through direct observation of the outputs. However, this is expensive and one has to rely on the limited inputs and outputs of the chip to develop a test generation sequence. Constraining the number of inputs forces the testing to become highly sequential; thus a sequence of inputs that enable the effect of the fault to be observable at the output must be found by the fault detection system. Finding this sequence must be efficient to avoid waste of computer time.

1.1 Previous Results

The earliest fault diagnosis programs were written to exercise machine functions, rather than hardware. Generally, a complex machine instruction like MULTIPLY or DIVIDE was executed and the results were compared with those obtained using an equivalent sequence of simpler instructions, like SHIFT, ADD or SUBTRACT. If there was a discrepancy between the results, then the complex operation was assumed to be defective. The results from such tests might not necessarily be valid due to the limited nature of the test as reported by Estrin (1953) and other investigators.

Eldred (1959) was one of the first investigators to appreciate the importance of diagnostic programs which test machine hardware rather than its functions. This was a major improvement and hardware-oriented diagnostics came into general use and are still being used. Eldred's results were developed for fault detection in combinational circuits of one or two levels. An extension of Eldred's work to circuits having any number of levels followed; the process is labelled one dimensional path sensitization. Although many investigators worked in this area, Armstrong is prominently linked with this method.

The idea is to choose a path from the site of a fault to the output; the inputs to the gates along this path are assigned values so as to propagate any change on the faulty line along the chosen path to the output. This path is called a sensitized path and the process of constructing the path is called the forward-trace phase of the method. After setting up a sensitized path, we trace back from the gates along the sensitized path toward the primary inputs. This is the backward-trace phase of the method.

Although Schneider (1967) has provided a counter example to show that the method is not an algorithm, this method has been very useful in practice. Its defect is the occasional inability to produce a test when one exists.

J. Paul Roth (1966, 1967) formulated an algorithmic method which sensitizes all possible paths from the site of the fault to the output simultaneously. He called this method the d-algorithm. The d-algorithm has proved to be a general solution to the problem of fault detection in combinational circuits.

The formal algorithmic approach to fault detection in sequential circuits has been studied by various investigators including Poage and McCluskey (1964), Hennie (1964) and Kime (1966). These algorithmic approaches are impractical for any but small circuits and small classes of faults. This is due to the following difficulties:

a. For each possible circuit state, a potential test input must be evaluated. The number of states increases as 2^n where n is the number of memory elements in the circuit. This puts a practical limit on the complexity that the circuit can have.

b. A homing sequence (Hennie, 1964) must be found that forces the machine into a known state. It may be very lengthy or one may not exist for some circuits.

c. Some sequential circuits, especially large circuits, can not be easily described by a state table.

The alternative approaches to fault test generation in sequential circuits are non-algorithmic: methods that treat fairly large circuits and are economical of computer time.

The Sequential Analyzer of Seshu and Freeman (1962, 1965) was one of the first non-algorithmic test generation methods. The Analyzer is a digital simulator. An heuristic is presented to the Analyzer with a sequential circuit plus a specified set of faults. The heuristic proposes one or more potential tests which are simulated to determine their performance. Some sort of numerical measure of performance is computed for each test input and the one with the highest figure of merit is used provided its value exceeds a predetermined value. Otherwise the heuristic has failed and another is tried. Four heuristics were developed and are tried. If all four heuristics fail, the system gives up.

All of the heuristics of the Analyzer have proved to be reasonably effective for small circuits. They are, however, impractical for large circuits because of the computer time required to simulate the various candidate tests. Because the heuristics employ local rather than global optimization techniques, they do not guarantee a minimal test sequence..

Other non-algorithmic test generation methods were developed by Kubo (1968), Breuer (1971), Bouricius et. al (1971) and Rutman (1972). All of these essentially transform the sequential circuit into an iterative combinational circuit. The d-algorithm is then applied to generate a candidate test. The iterative model has been quite successful for sequential circuits that are largely combinational in form and where the number of circuit iterations required to model the time frames needed to propagate the fault to the output is small. But when the number of state variables is large, the testing procedure has to be abandoned due to the fact that the computational time becomes exorbitant.

1.2 The Sequential Circuit Test Search System (SCIRTSS)

The Sequential Circuit Test Search System, developed by Hill, Belt (1973), Carter (1973) and Huey (1975) is based on a non-algorithmic method using heuristic graph searching techniques. Two heuristic tree search procedures automatically determine trial input sequences which are used to simulate simultaneously all single faults of the circuit. The sequential circuit is partitioned into its control and data portions as done in most Computer Hardware Design Languages (CHDL's). The control input combinations are applied in every node expansion while only input vectors previously specified by an input vector generating routine are considered. Thus, the search is primarily that of the state graph of the control circuit. The node expansion is computed by simulating a CHDL rather than by circuit simulation, an approach which leads to a great reduction in computation time. The design language used is AHPL (Hill and Peterson, 1978).

We present here a simple description of SCIRTSS. A more detailed description is given in Hill and Huey (1977) and Huey (1978). SCIRTSS incorporates the single permanent fault assumption and assumes that both the faulty and the good network operate in clock mode. Fig. 1.1 shows a block diagram of the test generation system. SCIRTSS has two main search routines: The sensitization search and the propagation search.. For a particular fault to be detected, a sequence of inputs must be found that takes the fault-free circuit from its initial state to a state such that the input sequence generates a sensitized path from the site of the fault to either an output or to a flip-flop. This input sequence is called the fault-sensitization sequence and the process of determining this sequence is the sensitization search shown in block 2. The application of the fault-sensitizing sequence may cause the effect of the fault to appear at the output or to be stored in a flip-flop. In the latter case, a sequence of inputs is needed to cause the register transfers to make the discrepancy between the faulty and fault-free circuits observable at the output. This input sequence is the fault-propagation sequence and the process of propagating the stored fault to the output is the propagation search shown in block 3.

Both the propagation and sensitization searches use guidance mechanisms to reduce the search cost. The searches are

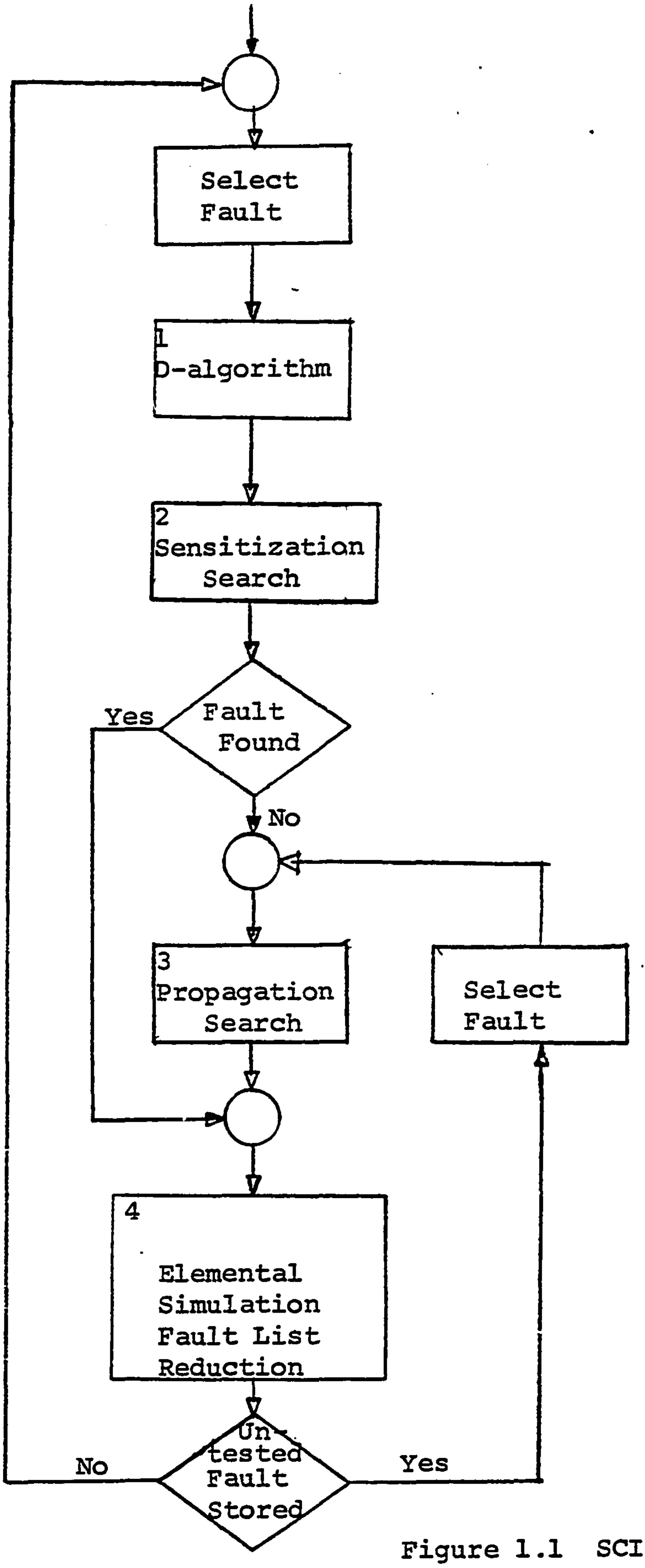


Figure 1.1 SCIRTSS Flow Diagram

conducted over control and user-specified data input only and consider only one fault at a time.

Blocks 1 and 4 in Fig. 1.1 are the d-algorithm and elemental simulator, respectively, that are incorporated into SCIRTSS. If a fault is ever to cause malfunction, there must be some state of the machine for which the outputs are in error or the next states of the good and faulty machines differ. If the set of untested faults and a circuit inter-connection list are given, a modified d-algorithm can find states for which the faults will cause erroneous next states or outputs. This d-algorithm treats the circuit as if it were combinational by considering its behavior for only one clock period. The test vectors returned by the d-algorithm are converted into primary inputs and "present" states and the test generation problem becomes reaching one of these present states. Inputs are selected heuristically and the response of the machine is simulated with the AHPL simulator until the search for a sequence of input vectors to move the machine to one of the goal states is successful. Once this happens, SCIRTSS enters the fault propagation mode to extend the effect of the fault to a primary output. Inputs are again selected heuristically and the machine is simulated in a search of the control state graph until a sequence of input vectors is found which will move the fault to the output.

After finding a test sequence, it must be verified using the elemental simulator of block 4 in Fig. 1.1. This simulator and the AHPL simulator are different in three aspects:

a. The effect of using the faulty gate to perform register transfers in propagating the fault is only approximated in the AHPL simulator.

b. The AHPL simulator uses a single machine and each variable may be 0, 1, D, $\overline{D}$, or X (unknown). The elemental simulator permits each variable to be only 0, 1, or X for a given machine, but simultaneously simulates the good machine M, and for each undetected fault f_i, a faulty machine M_{fi}.

c. The AHPL simulator is about 25 times faster than the elemental simulator. This makes the trial and error searching practical in terms of computer time.

In block 4, Fig. 1.1, all other faults detected by the same input sequence are removed from further consideration. SCIRTSS checks the states of the good and faulty machines remaining for faults stored in flip-flops as a result of the input sequence just applied. If new faults are stored, the program continues in fault propagation mode. There is a point at which the set of untested faults is not empty, but none of the remaining faults have resulted in an error occurring in a register. The register transfer simulator is no longer useful at this stage for propagating faults to the output and SCIRTSS must reenter sensitization mode.

1.3 Guiding Sensitization Search

SCIRTSS has been very effective in generating faults for highly sequential circuits. Huey and Hill (1977) give some statistics to show the usefulness of SCIRTSS. The propagation search has produced consistent results throughout the history of SCIRTSS. Sensitization search, on the other hand, has not been as successful. This is due to the fact that sensitization mode searching occurs after SCIRTSS has run out of faults to propagate and the remaining faults are usually difficult to reach. Essentially, the sensitization search is confronted with the task of moving the machine into a small set of goal states which are inherently difficult to reach. Many highly circuit dependent heuristics were written to cope with this problem. Huey (1975) was the first to attempt to provide a general purpose heuristic function and minimize the manual effort required of a user in forming an input sequence. His proposals also improved the efficiency of the sensitization search.

The question that is answered by the fault sensitization phase of SCIRTSS is "how can the states of the fault-free circuit and faulty circuit be differentiated?" For a given fault f_i, a sensitized path from f_i to a flip-flop FF_i is determined by a combined state and input vector $v_j = (X_j, Y_j)$ where X_j is the input to the combinational logic and Y_j is the flip-flop state. Of course, V_j may not be unique; thus, for f_i we have a set of vectors $V_i = \{V_{i1}, V_{i2}, . . . V_{in}\}$ which determine the sensitized path. After the V_{ij}'s have been found by the modified d-algorithm, a sequence of input vectors must be found to move the machine from its present state to state Y_j. Finding this sequence of inputs is the sensitization search which can be regarded as a graph theoretical problem of finding a path from any of the starting nodes to any of a set of goal nodes, states that provide a sensitized path to an output or a flip-flop. This search, like the propagation search, requires direction to be efficient.

The problem reduction graph approach of Huey (1975) has provided a guidance mechanism for the sensitization search whose effectiveness is independent of the circuit under test. In the problem reduction graph, the problem of reaching a goal state is resolved into subproblems which are in turn iteratively broken into subproblems terminating in simple problems. The nodes in the problem reduction graph are weighted and a heuristic value for each state encountered in the state space search is computed based on the weights of the nodes in the problem graph that are not satisfied by the search state. The problem graph also indicates input vectors associated with each control state. These input vectors are used together with the heuristic function to guide the sensitization search.

1.4 Proposed Work

The problem reduction graph approach has demonstrated its effectiveness in guiding fault-sensitization searches for goal states in complex sequential circuits that are very difficult to reach. This approach is the first to provide a general purpose guiding mechanism for the sensitization search.

Is it the best? Using the idea of analyzing a design language description can a more efficient method be found? This research is an attempt to find out answers to these questions. Also, the problem reduction graph introduces the idea of "τ links" which were not actually used in generating the heuristic function nor in the selection of input vectors to guide the search. It is our intention to study an efficient method of selecting input vectors to guide the sensitization search.

In studying how the heuristic cost function is derived from the problem reduction graph, it becomes apparent that the function is trying to measure the effect of satisfying a node on the overall desired goal of reaching a solution. Petri nets are graph models that have been used in various areas of computer science to study the interconnection properties of systems. It appears then that petri nets are also very suitable for measuring the effect of reaching one state in a state space search on the overall desired goal state.

This research presents the use of petri nets to model the register transfers and change of control states in a sequential machine described in a Computer Hardware Description Language with the aim of guiding fault-sensitization searches.

The next chapter develops the model and the following two chapters develop the guidance mechanisms for the sensitization searches. In Chapter five, four different circuits are used to test the method.

CHAPTER II

PETRI NETS AS AN AID TO FAULT DETECTION

2.1 Introduction

For each search state of the sensitization search, a cost value is computed and external input vectors are provided to the search program to guide the search. This combination of input vectors and heuristic cost values increases the efficiency of the search. Because the heuristic cost value must be computed for each node as it is generated, its computation must not be time-consuming, otherwise it will slow down the search. The external input vectors must be judiciously chosen for each state to minimize "trial and error."

Before the commencement of the sensitization search, goal states are defined by the d-algorithm as explained in section 1.2. These goal nodes can be broken down into subnodes and the subnodes broken down further until an essentially trivial node is reached. These subnodes are concerned with transferring vectors $(a_1,a_2...a_n)$, a: $\varepsilon(0,x,1)$ into given registers or moving the machine into a given control state and/ or applying an input vector $(a_1,a_2...a_n)$ at a given control state. The problem can then be thought of as: "How can we reach the goal node(s) starting from the trivial nodes?" In this chapter we present the technique of modeling the register transfers and change of control states in a given machine by a petri net.

By studying the relationship between the various "transitions" and nodes in the petri net for each machine state, we can derive an heuristic value for the given state. Also, input vectors to be applied at any given control state can be obtained from the petri net.

It is assumed the given circuit is described in a computer hardware description language (CHDL). For our discussion we use AHPL (Hill & Peterson 1978) due to familiarity.

2.2 Background

"Petri nets" are graph models used to study the interconnection properties of concurrent and parallel systems. C.A. Petri (1962) proposed in his dissertation "Communication with Automata" that the basic phenomena of communication, such as the switching logic of totally asynchronous automata, are representable by purely combinatorial-topological means. Thus, he proposed the construction of a net with more practical applicability in the design and programming of information processing machines than does the theory of abstract automata. Holt et al (1968) developed Petri's work to such a state that it is applicable to many areas in computer science.

Our purpose here is to use this modelling device to model the register transfers and state transitions that can occur in a machine, given a set of goal nodes.

2.2.2 Firing Rules

Fig. 2.1(a) is a large-scale distributed system which interconnects many information processing elements or processors. For the purpose of studying the relationship between the interconnection and the overall behavior of the system, each information processing element may be represented by a module of the general form shown in Fig. 2.1(b). The vertical bar is called a "transition" while the circles are referred to as "places" or "locations." A petri net is the interconnections of such modules. Thus we may look upon a petri net as a directed bipartite graph wherein there is allowed a directed arc from a place or location to a transition, or from a transition to a place. In order to simulate the flow of control in a petri net, each place is marked with (that is, may have assigned to it) a non-negative number of tokens. We may think of a token as representing a datum, or denoting the presence of some condition or some control signal associated with its place.

The transition obeys the following rules:

a. A transition is said to be "enabled" or "firable" if each of its input places contain at least one token,

b. The "firing" of an enabled transition consists of removing one token from each of its input places, and adding one token to each of its output places. Fig. 2.2 gives an illustration.

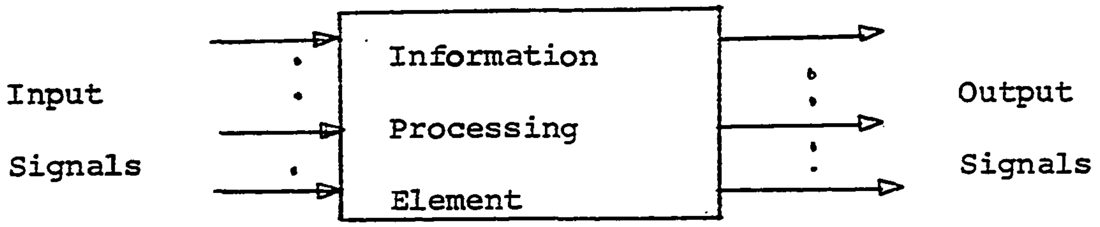


Fig. 2.1(a) Model

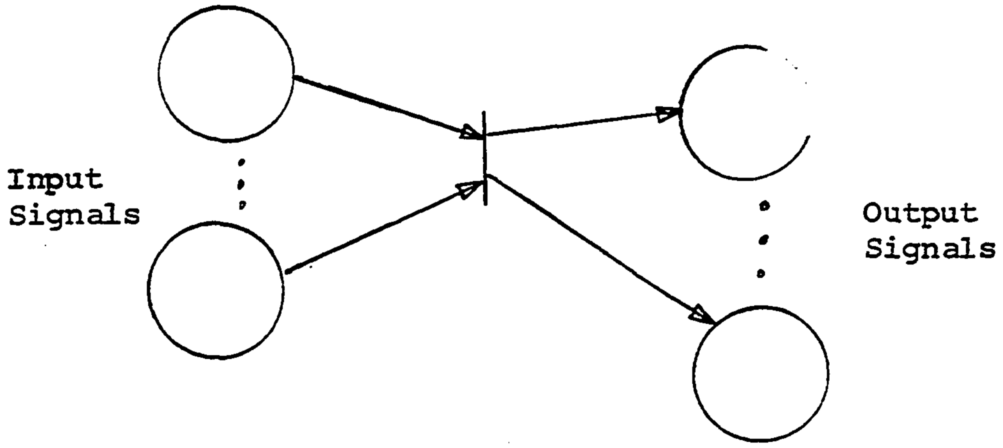
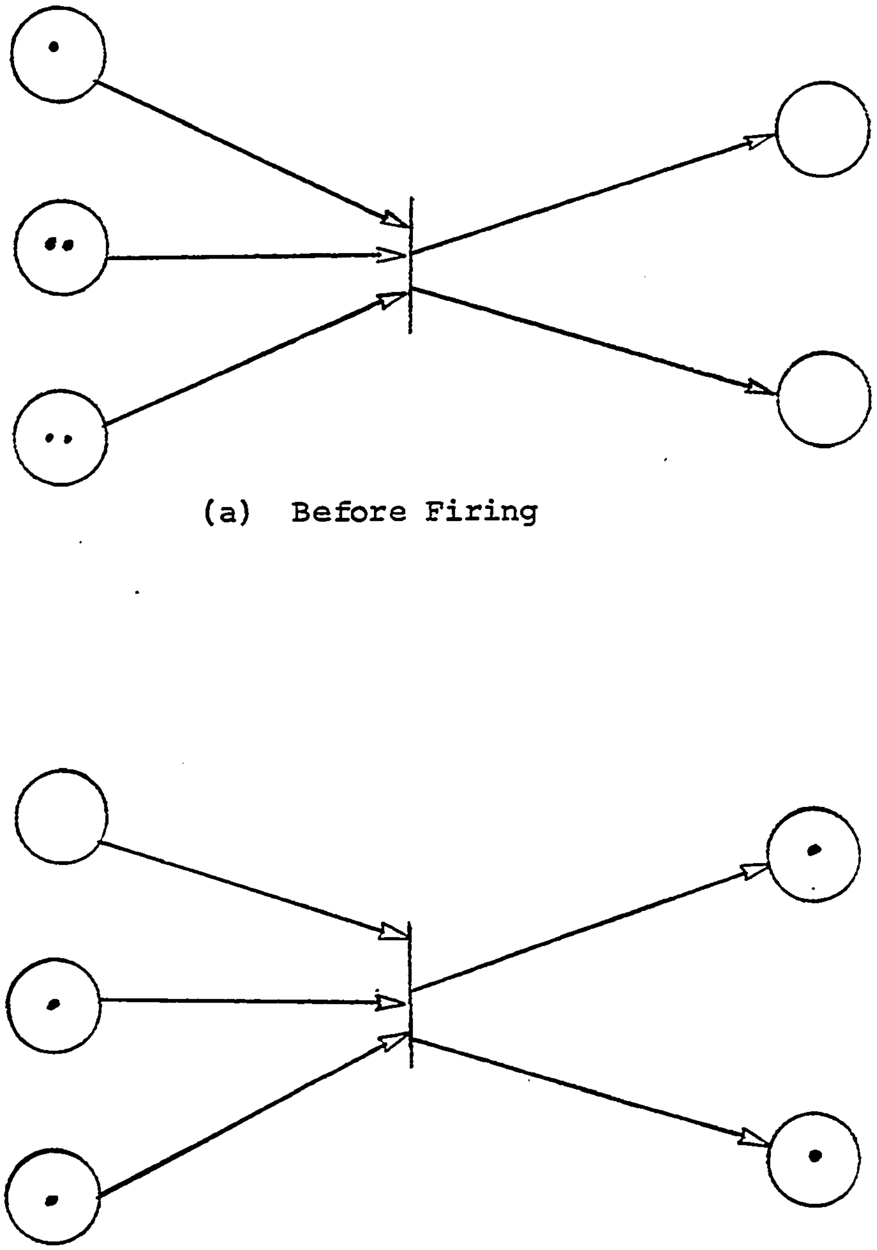


Fig. 2.1(b) Building Block



(a) Before Firing

(b) After Firing

Figure 2.2 Firing of an Enabled Transition

c. Though it may fully be enabled, a transition cannot fire until directed to do so (by some outside control).

In summary, we may think of transition as an event which can fire (i.e., occur) if all places (conditions) input to that transition have tokens (are satisfied).

2.3 Petri Net as an Aid to Guiding Sensitization Search

Given a set of goal nodes G( ), we can reduce them to subnodes, until we obtain a set of trivial or terminal nodes. Thus, for each set of goal nodes we can generate a petri net where the transitions correspond to register transfers or changes of control state. Remember that a transition is simply an event: a register transfer that must be done or a change of state of the machine from one control state to another. For our model we will define five types of transitions:

(1) Register Transfer: This type of transition models the change of state of a register; transferring a vector $(a_1,\ldots,a_n)$, $a \varepsilon (0,X,1)$ from one set of registers or input into a destination register.

(2) Control: This type of transition models the change of control state of the sequential machine.

(3) Simple Transfer: This type of transition models unclocked control states and terminal expressions. The transition that fires to fill the goal place (sec. 2.3.1) belongs to this group.

(4) Count: The count transition models the change of state of a counter.

(5) Shift/Rotate: This transition type models the shifting and/or rotation of a given register.

Generally, all transition types, except type three, have an execution completion time associated with them. This timing requirement is included in our model because we are dealing with clocked sequential circuits: if all the conditions for loading a register are fulfilled at time t_n, the register is loaded with the vector at the next clock period. Similarly, if conditions for change of control state are fulfilled during time t_n, the machine enters the next control state at time t_{n+1}. For slower memories, the reading (or writing) from memory is not completed until some units of time after the process was begun. Hence, this timing provision takes care of all timing requirements; in fact, the simple transfer (type three) is a special case in which the process is completed during the same clock period. This latter case correctly models the "NO DELAY" timing requirement of AHPL (Hill & Peterson, 1978).

For each transition in the petri net, we can associate a time unit t(i) which would indicate the number of time units that separate the transition and the goal. Why should we link the transition time to the goal place? This is done to give a measure of the time units that would elapse before the goal is filled with a token after a particular transition is fired. Remember that filling the goal place is our target and as such every formulation takes into account the question "how easily can the goal place be filled?" In Fig. 2.3, taking P_1 as the goal and assuming all transitions have execution completion time of one unit associated with them, then it can be seen that if t_1 fires then the goal P_1 would be filled; however, if t_4 fires, t_3 and t_1 must fire before the goal can be filled. Thus the transition time of t_4 is $t(4) = 3$ while the transition time of t_1 is $t(1) = 1$. Of course, this assumes that the firing order is $t_4 \rightarrow t_3 \rightarrow t_1$.

Summarizing our discussion of the preceeding paragraphs, for each transition in the petri net generated for a given fault, we can associate two parameters:

s =: the type of transition

t(i) =: transition time; that is defined as the number of time transitions separating the transition and the goal place.

2.3.1. Types of Places

In our model, the places represent conditions or requirements which must be satisfied during a sensitization search. We can define five different types of places:

a. Goal: This is a unique place in the petri net; it represents the condition of sensitizing the fault under consideration.

b. Control: This type of place or location represents the requirement of moving the machine into a given control state.

c. Register: Loading or transferring a predetermined vector $a_1, a_2 \ldots a_n$ $a_i \varepsilon (0,1,x)$ into a register. Counter and shift registers are in this category.

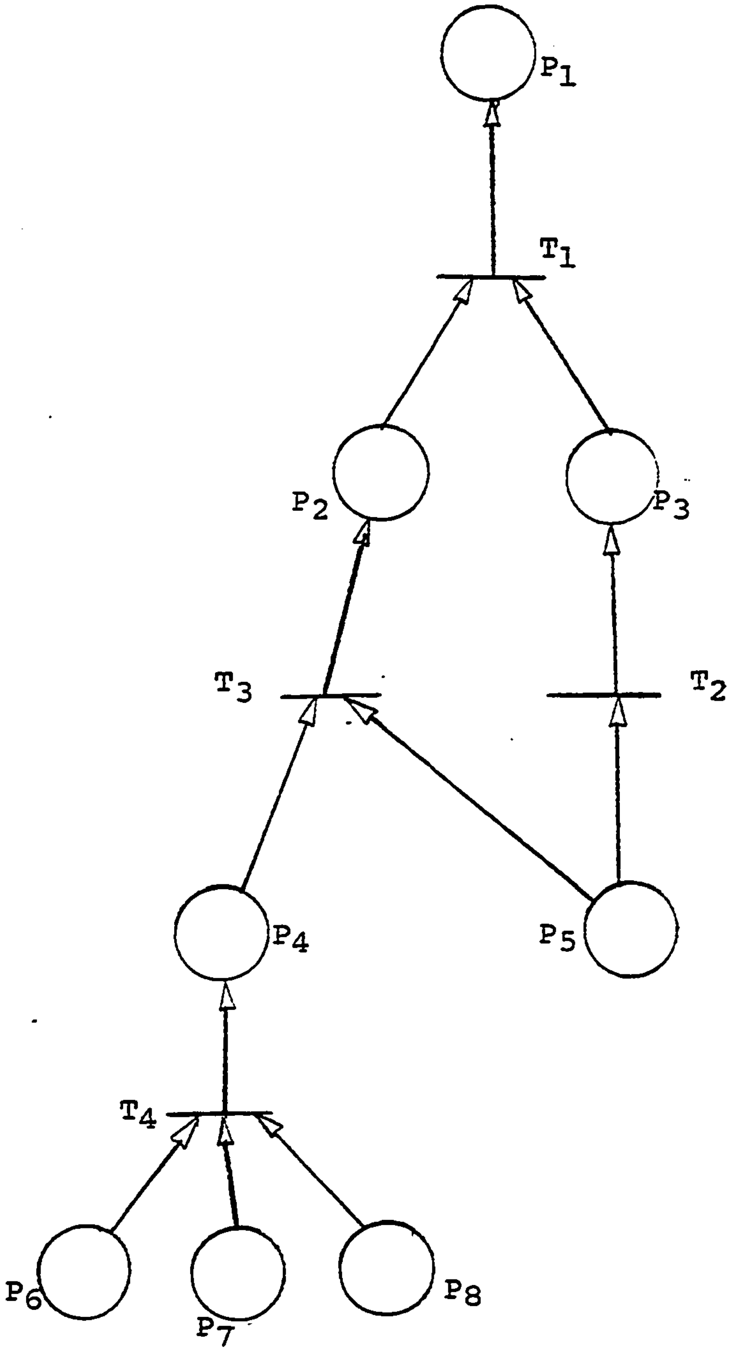


Figure 2.3 Petri Net for Illustrating Transition Time

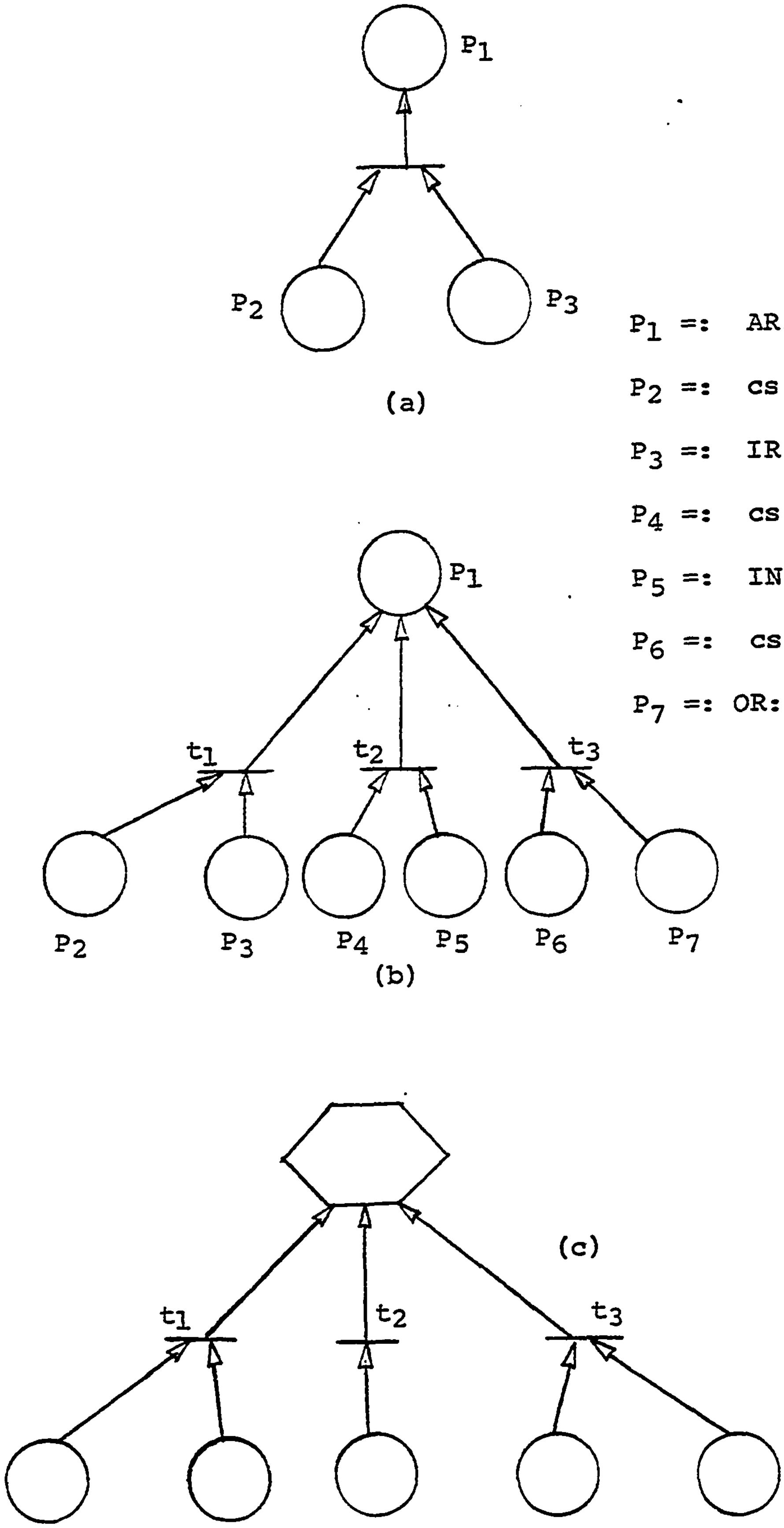
d. Input: Placing a vector $a_1,a_2\ldots a_n$ on an external input.

e. Output: This type of place models the condition of a vector $a_1,a_2..a_n$ appearing at the external output.

These conditions will be shown just outside the circles as in Fig. 2.4(a). Since the machine can only be in one and only one control state at any given time,and a register can only be loaded with one vector in any given control state, the places in the petri net can only have a maximum of one token at any time. When a condition is fulfilled, the appropriate place is filled with one token.

Some places in the petri net will have more than one incoming arc. This means that the condition represented by the place can be fulfilled or satisfied from any of several transitions. For example, for a register AR to be loaded with a vector $a_1a_2...a_n$, the machine must be in either control states 1 , 3 or 5. In each control state, when some condition is fulfilled, then AR is loaded. In control state 1, this occurs when register IR contains $a_1,a_2...a_n$. In control state 5, an input vector $a_1a_2...a_n$ is applied. Fig. 2.4(b) illustrates this condition. Of course, t_1, t_2, or t_3 can only fire when their respective places are filled.

To differentiate the goal place from all other places in the petri net, we use the visual representation shown in Fig. 2.4(c). The input places to t_1, t_2 and t_3 are the test vectors generated by the d-algorithm any of which would cause an erroneous next state to result from the presence of the



P_1 =: AR:$a_1a_2...a_n$

P_2 =: cs.1

P_3 =: IR:$a_1a_2...a_n$

P_4 =: cs.5

P_5 =: IN:$a_1a_2...a_n$

P_6 =: cs.3

P_7 =: OR:$a_1a_2...a_n$

Figure 2.4 Petri Nets Illustrating Different Types of Places

fault. If any of the transitions t_1, t_2 or t_3 are fired, then the goal place is filled which implies the fault is sensitized.

2.3.2 Formal Definition of Petri Net for Guiding Sensitization Search

Although petri nets have many properties like reachability, liveness, and safeness (or boundaries), most of the work reported on the properties of petri nets are concerned with subclasses of petri nets (such as "marked" graphs). We do not intend to investigate any of these properties in our model; rather our aim is to develop a means of guiding the sensitization search from the petri net generated for a given fault, given the CHDL description.

Before giving a formal definition of our model, we define the state of a petri net: A token distribution in a petri net is called a marking or state. Initially, each place has a status (full or empty) referred to collectively as marking M.

We have now presented all the material needed for a formal definition of a petri net as an aid to computing heuristic values for guiding sensitization search:

For a given fault we define a petri net as a quintuple:

$$P = \{G, T, P_N, P_T, M_n\}$$

where G = set of test vectors returned by the d-algorithm, each of which will cause an erroneous next state to result from the presence of the fault.

T = the set of transitions.

P_N = the set of non-terminal places or locations from which further subnets can be generated.

P_T = the set of terminal places.

M_n = marking or state of the petri net. Usually one will be interested in the state just after the machine has been driven into a search state Si.

Generally $P_N \cap P_T = \emptyset$, the null place and we will often denote $P_N \cup P_T$ by P, the set of all places in the petri net.

2.4 Generating the Petri Net from AHPL

The petri net generation process starts with the goal states returned by the d-algorithm. The way(s) in which these goal places can be filled is generated using the knowledge of the hardware for the control states, registers, inputs and memories which are available in the AHPL description statements. For example, if the d-algorithm returns (AC: XIOX, IR: IXX) and (AC: 10XX, lR: lXX), we would have the net shown in Fig. 2.5(a). t_1 and t_2 are the transitions which fire to fill the goal place with a token. By our model, it needs either t_1 or t_2 to fire to have the goal place P_1 filled. We must now generate the remaining portions of the petri net from P_2, P_3 and P_4.

There are three different types of expressions in the AHPL description from which the remaining portions of the petri net must be generated:

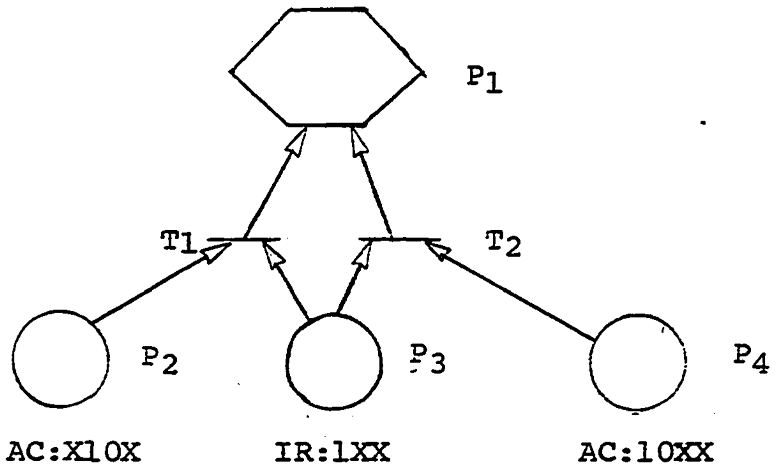


Figure 2.5(a) Setting Up the Goal Places

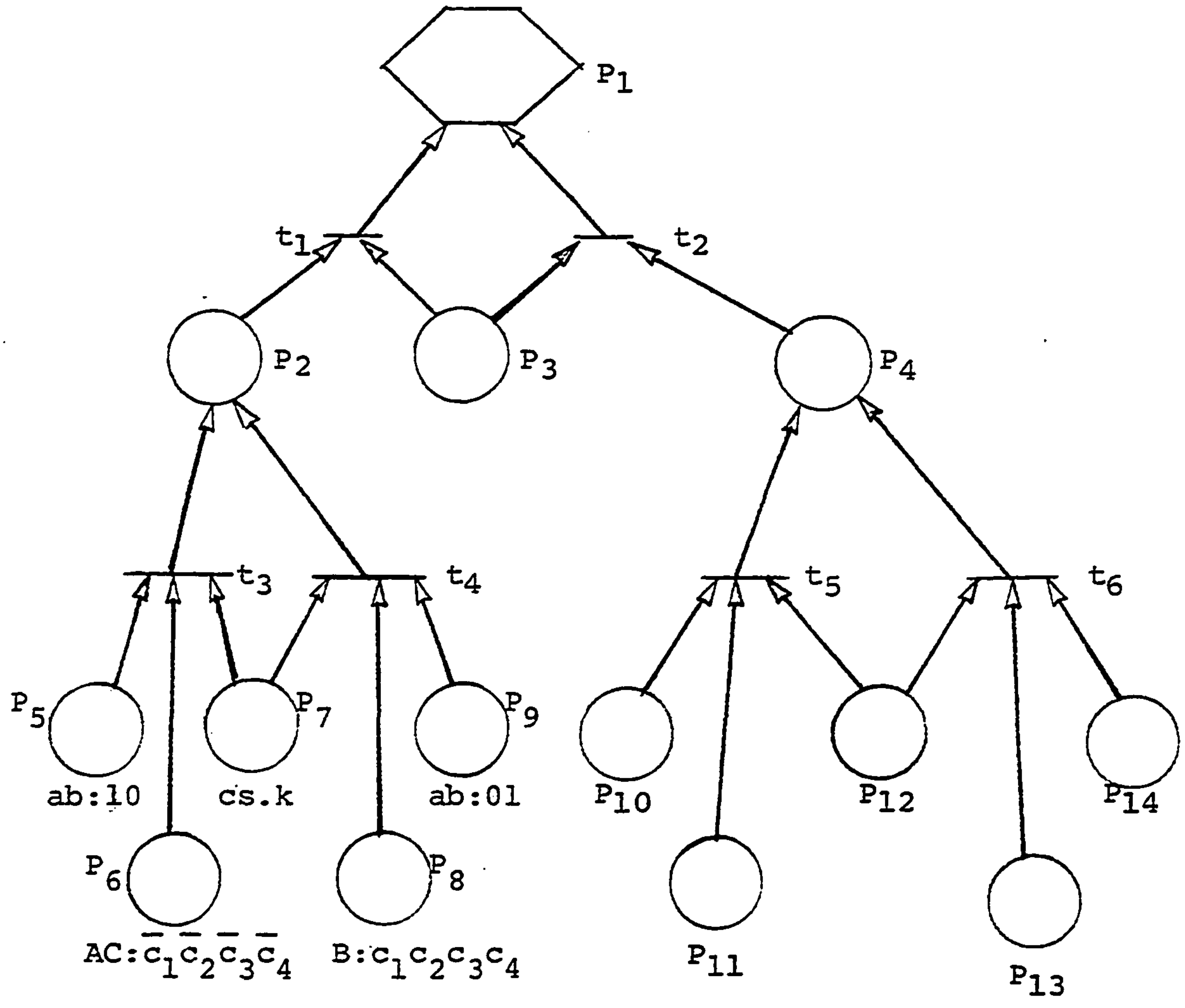


Figure 2.5(b) Expanding the Goal Places

1. Condition expressions
2. Register Transfer expressions
3. Control Branch expressions

A good discussion of how these expressions are handled is given in SCIRTSS (Huey, 1975, pp. 24-35) and will not be given here in detail.

Equation 2.1 contains condition expressions which must be satisfied before a register transfer can take place.

$$K. \quad AC \leftarrow (\overline{IR}_1 \wedge IR_2 \wedge \overline{AC}) \vee (IR_1 \wedge \overline{IR}_2 \wedge B) \qquad (2.1)$$

It simply means that if IR_1 is zero and IR_2 is one, then transfer the complement of the contents of register AC into register AC; however, if IR_1 is one and IR_2 is zero, the contents of register B are transferred into register AC. Of course, this can only be done at control K. Assuming IR is a 3-bit register, then the condition $\overline{IR}_1 \wedge IR_2$ is translated into the condition IR:01X; similarly $IR_1 \wedge \overline{IR}_2$ becomes the condition IR:10X. Notice that these conditions are not dependent upon the values in any of the specified registers; that is, the conditions are invariant with respect to the goal places. We will return to this point a little later.

For the sake of generality, equation 2.1 is rewritten as in equation 2.2 where IR_1 is replaced by a and IR_2 by b; a and b are in effect control variables.

$$K. \quad AC \leftarrow (\bar{a} \wedge b \wedge \overline{AC}) \vee (a \wedge \bar{b} \wedge B) \qquad (2.2)$$

Given equation 2.2 which is a register transfer expression and the goal nodes of Fig. 2.5(a), our task is to find transitions and their input places such that if the transitions are fired, places P_2 and P_4 would be filled with tokens; that is, register AC would be loaded with the vector X10X or 10XX. From the register transfer expression of equation 2.2, we see that AC can be loaded with a required vector $C_1C_2C_3C_4$ in one of two ways:

1. If the machine is in control state K and AC contains $\overline{C}_1\overline{C}_2\overline{C}_3\overline{C}_4$

or 2. If the machine is in control state K and register B contains $C_1C_2C_3C_4$. For the first case the condition ab:10 must be satisfied while ab must be 01 in the second case. We thus need two transitions t_3 and t_4 to expand the goal place AC:X10X. Each one of these transitions has input places as shown in Fig. 2.5(b). The place AC:10XX is expanded in a similar fashion. Transitions t_3,t_4,t_5 and t_6 are transitions of type one since they all model the register transfer which takes place if all the conditions are fulfilled one time period earlier.

Places P_5, P_6 and P_7 are all input places to the same transition and they will be called brothers. Transition t_3 is a descendant of transition t_1 since if t_3 is fired and place P_3 is filled with a token, then t_1 can fire assuming all other conditions are fulfilled. More generally, a transition $t_j(t_i)$ is said to be a descendant (ancestor) of a transition $t_i(t_j)$ if t_i can be ultimately fired after the firing of t_j (after progressing through some further firing, if necessary). In particular, $t_j(t_i)$ is an immediate descendant (ancestor) of $t_i(t_j)$ if an output place of t_j is an input place to t_i.

After the goal places have been expanded as in Fig. 2.5(b), the new register places generated are also expanded by the same reasoning.

The third type of expression in AHPL is the control branch expression. There are two ways control can pass from one control state K to another.

a. Unconditionally:

$K. \rightarrow (i)$

b. Conditionally:

$K. \rightarrow (a,b,c,\ldots)/(i_1,i_2,i_3,\ldots)$

In the first case, control passes from control state K to control state i without any condition. This often happens after a register transfer or some other action takes place in control state K. The machine is then sent to control state i to initiate some other action. The second case of transfer of control occurs only when a given condition is fulfilled. In the example given above, control passes from control state K to control state i_1, i_2, or i_3 depending on whether the condition a, b or c is fulfilled.

In order to represent these two types of control branch expressions in the petri net, we classify two types of control transitions:

1. Type 2a: Conditional control transition
2. Type 2b: Uncondtional control transition.

Where we have conditional change of control state, each member of the set {i_1, i_2, i_3,...} becomes an output place of a transition t_{c1}, t_{c2}, tc_3, respectively, whose input places are control state K and the respective conditions: a,b,c,....

Control can pass from more than one control state to control state i unconditionally in a given circuit. To follow the rules of transition firing, this has to be modelled as shown in Fig. 2.6(a) so that if any of transitions t_{c1}, t_{c2}, t_{c3} fires the place CS·K is filled. This accurately models the hardware behavior but can lead to a proliferation of transitions. For this reason, we choose the representation of Fig. 2.6(b) which violates the general firing rule. For this type of transition (type 2b), if <u>any</u> of the input places is filled, the transition becomes firable. This is justifiable since a transition is modelling a change of control state and we are interested only in the firing of the transition.

Notice that the control state expansion is completely invariant with respect to the goal places; that is, it is not dependent on where the fault is located in the machine.

After discussing how subnets are generated from control branch and register transfer expressions, one may ask "how are the firings of transitions derived from these expressions handled?" In section 2.2, we gave the firing rules of a transition: firing an enabled transition consists of removing one token from each of its inputs and adding one token to its output

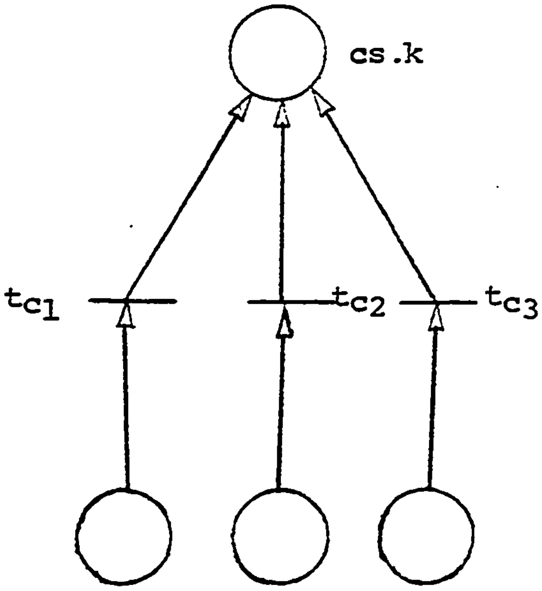


Figure 2.6(a)

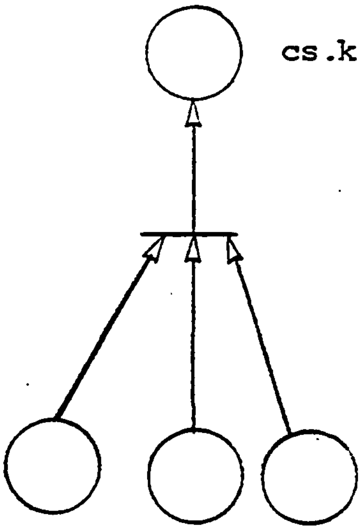


Figure 2.6(b) Unconditional Control Transition

place(s). Since register transfer in AHPL is non-destructive and the conditions for a control state transition remain after the change of control state, there seems to be a problem with our model! As will be explained in Chapter three, we are mainly interested in transitions that have fired during each sensitization state. Hence, we care only about the output places of transitions that are fired. In section 3.3 we introduce the notion of implied transition firing; the discussion in that section will give a good understanding of why we do not take pains to model the non-destructiveness of register transfers nor restore the condition tokens for control state transition.

2.5 Complete Petri Net

We use the AHPL described circuit of Fig. 2.7 to illustrate the generation of a full petri net from the circuit description.

The set of goals which would sensitize the fault is:

10111XX11XX
XX0XXXX01XX

From the AHPL declaration syntax, AC:11XX, MDR: 11XX and IR: 101 are input places to transition t_1 while IR:XX0 and AC:01XX are input to transition t_2. Any of t_1 and t_2 firing fills the goal place with a token and the fault is sensitized. There is no time delay involved, hence t_1 and t_2 are simple transfers of type 3. The first stage of our net is shown in Fig. 2.8(a).

```
MODULE:  SP
   MEMORY:  IR[3]; MDR[4]; AC[4]
    INPUT:  INP[4]
   OUTPUT:  MOR

1.     $IR \leftarrow \alpha^3/INP$
       $\rightarrow (INP_4, \overline{INP}_4)/(1,2)$

2.     $\rightarrow (IR_3, \overline{IR}_3)/(3,7)$

3.     $\rightarrow ( (\overline{IR}_1 \wedge \overline{IR}_2), (\overline{IR}_1 \wedge IR_2), (IR_1 \wedge \overline{IR}_2) )/(4,5,6)$

4.     $MDR \leftarrow INP;$  $MOR \leftarrow AC$
       $\rightarrow 1$

5.     $AC \leftarrow INP$
       $\rightarrow 1$

6.     $AC \leftarrow AC \wedge MDR$
       $MOR \leftarrow AC$
       $\rightarrow 1$

7.     $AC \leftarrow \uparrow AC$
       $MOR \leftarrow AC$
       $\rightarrow 1$
```

Figure 2.7 AHPL Description of Example

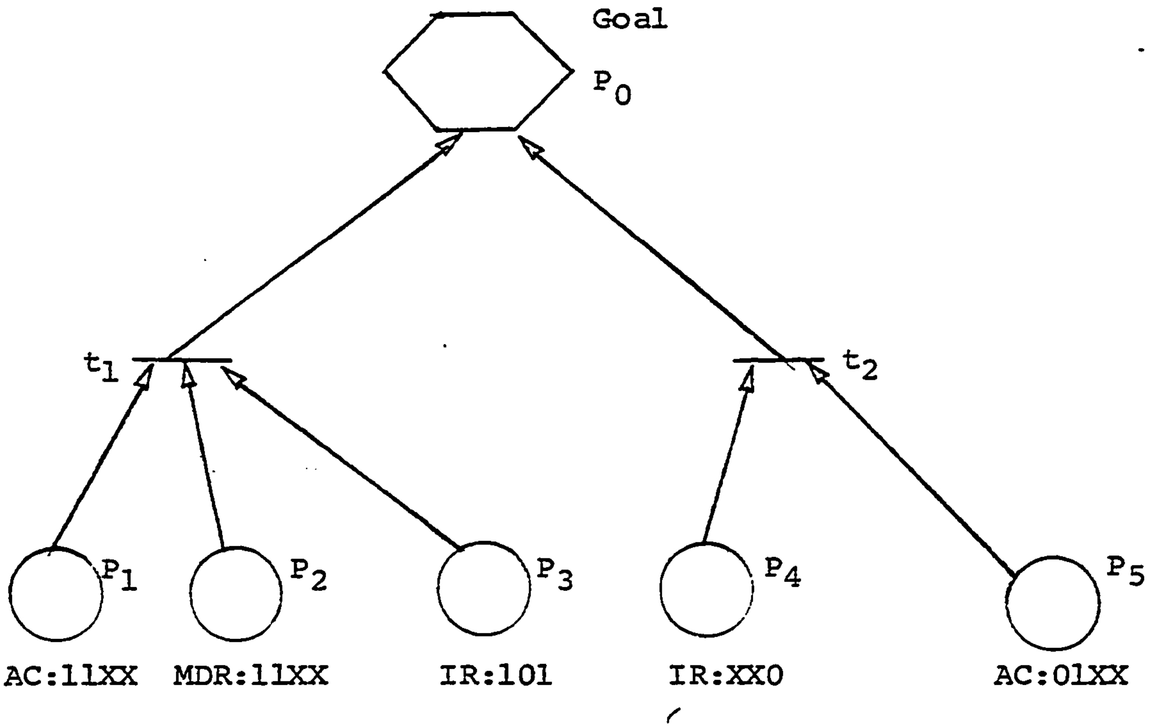


Figure 2.8(a) First Stage of the Petri Net Generation

At the second stage, we start by searching for a solution to the question "how can place P_1 be filled with a token?" P_1 filled with a token means that the register AC has been loaded with the vector 11XX. There are two alternate ways of doing this:

a. In the first case, if a transition t_3 with input places INP:11XX and CS·5 is fired, then at the next clock period, the contents of AC will be 11XX. Hence t_3 has transition type 1; that is register transfer and its transition time is one.

b. Alternatively, if a transition t_4 with input places CS·6, AC:11XX and MDR:11XX is fired. Transition t_4 also has transition time one and is of type one.

Notice how, for example, the input places MDR:X1XX and AC:01XX are obtained from the "and" operation of control state 6 since the output place is AC:01XX and we have logical AND of registers AC and MDR, we specify the vector $a_1a_2a_3a_4$ to correspond to the desired goal and then determine what the contents of MDR must be to give the correct result.

With this reasoning, we obtain the second stage of the petri net as shown in Fig. 2.8(b).

Now that we have encountered control states as places, we shall explain how these are treated before going on with the complete net generation. As discussed in the section 2.3, control branch expressions are completely invariant with respect

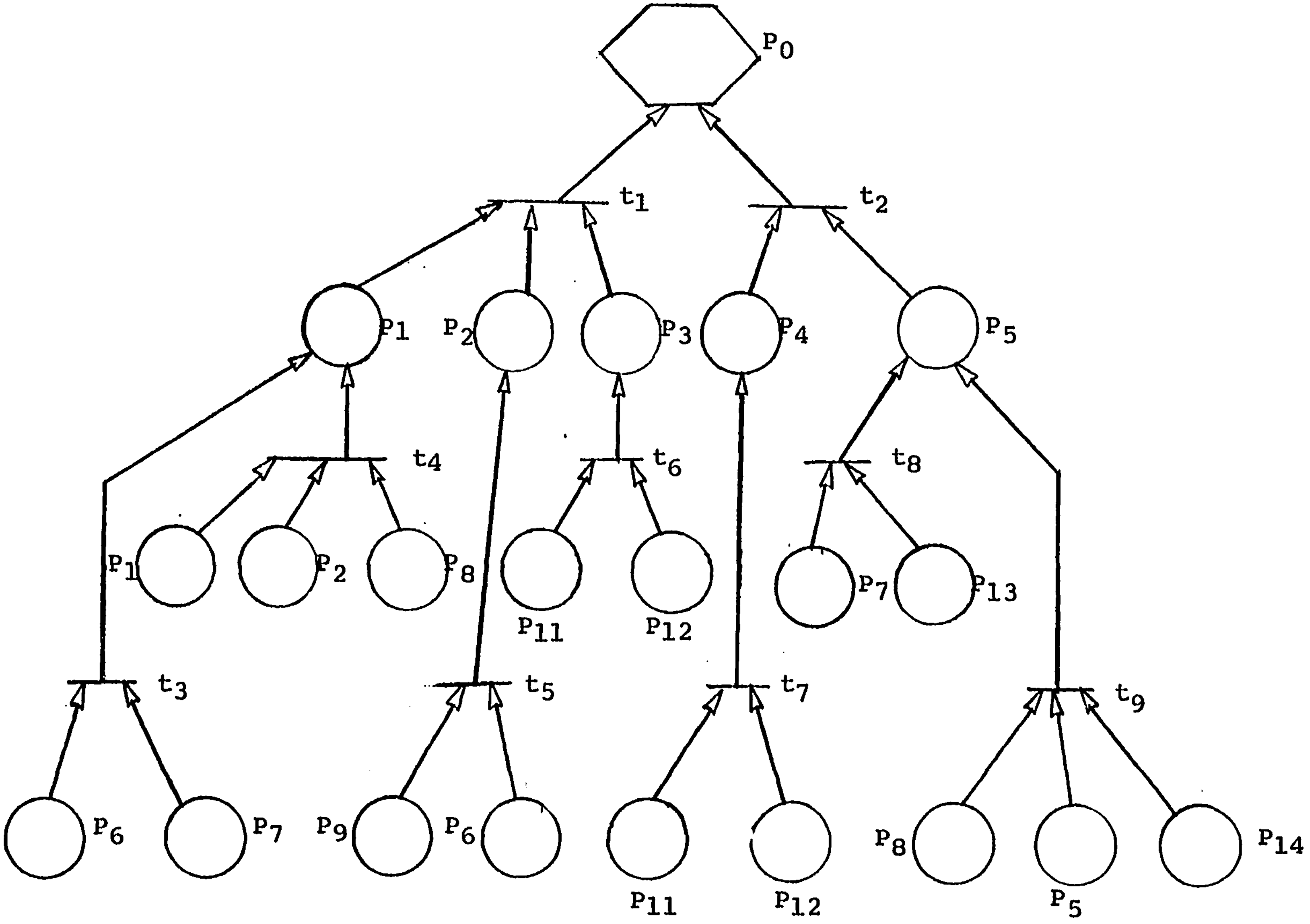


Figure 2.8(b) Second Stage of Generation of Example Petri Net

to the goals. These control subnets are generated before the generation of the full petri net. To generate the subnet corresponding to a control state, we use the same reasoning: how can I get to control state K? In the example circuit under discussion, this is almost trival. Fig. 2.9 shows the subnet for control states six and one. It is appropriate to show here one subnet from one of the example circuits discussed in Chapter Five. This is the control state nineteen subnet of case four, the four-bit microprocessor. This subnet in Fig. 2.10 is complex compared to our example circuit of Fig. 2.7.

When the decision to add the subnet of a control state to a main petri net is made, the linking step consists of adding the transition time of the output of the control state place to the transition time of the transition to which the control state is output. In this case, if we are linking CS·6 in Fig. 2.9 to transition t_4 in Fig. 2.8b, we would add the transition time of t_4 to the transition time of CS.6.

To complete the petri net generation, we would link the subnets for control states 1,4,5 and 6 to t_3,t_4,t_5,t_6,t_7, t_8 and t_9 in Fig. 2.8(b). After expanding the goal places we have the complete petri net shown in Table 2.1. The name of each place is given in Table 2.2.

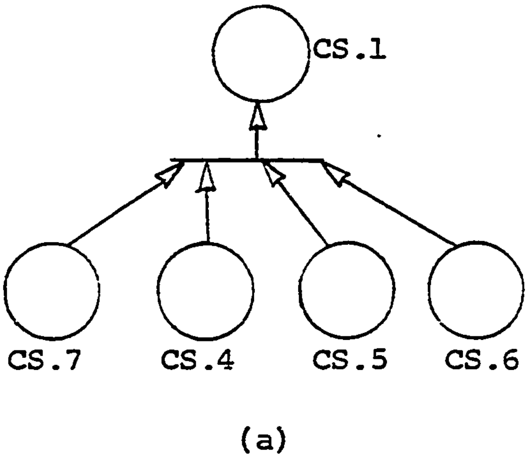


Figure 2.9 Subnets for Control States One & Six

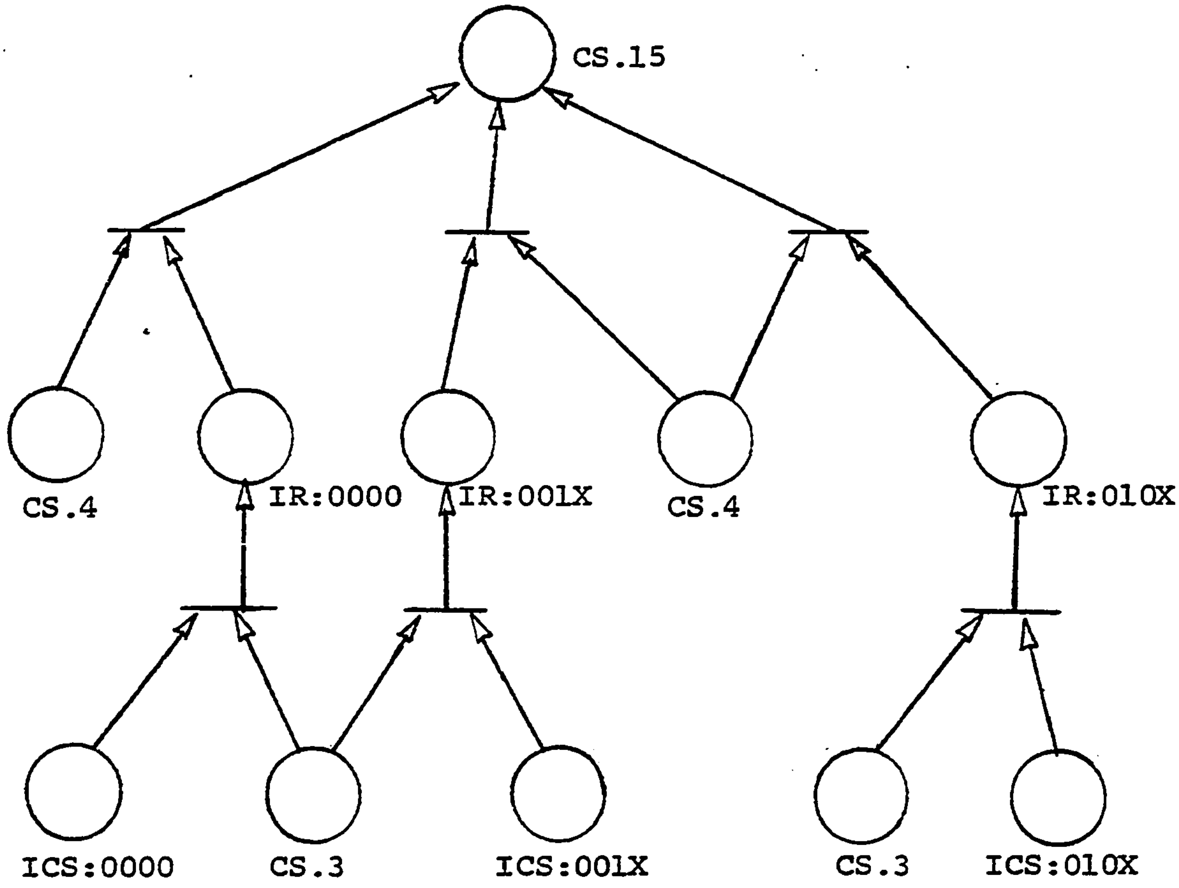


Figure 2.10 A Complex Control State Subnet

Transition	Output Place	Input Places .	Transition Time
1	P_0	P_1,P_2,P_3	0
2	P_0	P_4,P_5	0
3	P_1	P_6,P_7	1
4	P_1	P_1,P_2,P_8	1
5	P_2	P_9,P_6	1
6	P_3	P_{11},P_{12}	1
7	P_4	P_{11},P_{22}	1
8	P_5	P_7,P_{13}	1
9	P_5	P_5,P_8,P_{14}	1
10	P_7	P_{10},P_{15}	2
11	P_{15}	P_{11},P_{13}	3
12	P_8	P_{10},P_{16}	2
13	P_{10}	P_{17},P_{23}	3
14	P_{23}	P_{11},P_{18}	4
15	P_{16}	P_{11},P_{19}	3
16	P_9	P_{10},P_{20}	2
17	P_{20}	P_{11},P_{21}	3
t_{18}	P_1	P_{23},P_{25}	1
t_{19}	P_5	P_{23},P_{24}	1

Table 2.1 The Petri Net Listing for Figure 2.7

Place	Name	Place	Name
P_0	Goal	P_{13}	INP:01XX
P_1	AC:11XX	P_{14}	MDR:01XX
P_2	MDR:11XX	P_{15}	IR:01X
P_3	IR:101	P_{16}	IR:10X
P_4	IR:XX0	P_{17}	CS.2
P_5	AC:01XX	P_{18}	INP:XX1X
P_6	INP:11XX	P_{19}	INP:10XX
P_7	CS.5	P_{20}	IR:00X
P_8	CS.6	P_{21}	INP:00XX
P_9	CS.4	P_{22}	INP:XX0X
P_{10}	CS.3	P_{23}	CS.7
P_{11}	CS.1	P_{24}	AC:X01X
P_{12}	INP:101X	P_{25}	AC:X11X

Table 2.2 Place Listing for Table 2.1 and Figure 2.8(b)

2.6 Summary

In section 2.3 we defined a petri net as an aid to guiding sensitization searches and followed this up with an example in section 2.4. As noted in the background information of section 2.2, petri nets have been used to model various systems and can thus be used to model the machine of Fig. 2.7. It is not the subject of this work to show how a petri net can be used to model a machine itself, given the CHDL description; however, we would point out that such a model would be very different from the model of section 2.3. The latter is based on the notion of a goal place and is an attempt to model the change of control states and register transfers that must take place to fill the goal place with a token. It is thus dependent on the particular fault under consideration. Most of the places are dependent on the goal state; the only exception being the places that are responsible for control state branching and conditional register transfer.

CHAPTER III

HEURISTIC FUNCTION DEVELOPMENT

3.1 Introduction

In SCIRTSS, both the processes of fault-propagation and fault sensitization are accomplished by an heuristic graph search. The use of heuristic evaluation functions to direct the search of state-space graphs has been studied by many authors (see, for example, Hart et al, 1968 and Michie and Ross, 1970). Nilson gives a good treatment of the different ideas on which these evaluation functions are based. SCIRTSS assigns a weight to each node as it is reached. This weight is given by

$$W = G + wH$$

> "where G is the minimum number of transitions from the initial node state to the node, H is some heuristically determined value, and w is a constant indicating the relative importance of H in computing the total weight" (Carter, 1973).

In this chapter we present the development of a heuristic function from the Petri net to guide the sensitization search. First, we present the tools that are needed in developing the heuristic function; then the several ideas considered are presented.

3.2 State Equation of a Petri Net

Although the mathematical properties of petri nets have not been well exploited, we have found the state equations a useful tool in developing a heuristic function for guiding the sensitization search.

Throughout this chapter, the reader is reminded that we have the "natural" functioning of petri nets presented, followed by our application.

Let p and t denote the numbers of places and transitions in a petri net, respectively.

Defn. 3.1: A marking or state vector, M_K, is a p x 1 column vector of non-negative integers. The jth entry of M_K , m_j denotes the number of tokens on place j immediately prior to the Kth firing.

In the natural functioning of the petri net, it is customary to progress through a series of firing sequences; thus, we can speak of the "Kth firing." M_o denotes the initial marking or state.

Defn. 3.2: The Kth "firing" or "control" vector, V_K, is a t x 1 column vector of 1's and 0's. The i^{th} entry of V_K is one only if transition i is to be fired at the K^{th} firing opportunity.

Let $A^- = [a_{ij}]$ be a t x p matrix having $a_{ij}^- = 1$ if place j is an input place for transition i; otherwise $a_{ij}^- = 0$.

A^+_{ij} is similarly defined with $a^+_{ij} = 1$ only if place j is an output place of transition i.

Defn. 3.3: The matrix $A = A^+ - A^-$ represents the token changes in each of the p places when transition i fires once.

The state equation:

$$M_{K+1} = M_K + A^T V_K, \quad K = 0,1,2,\ldots \quad (1)$$

gives the marking M_{K+1} resulting from marking M_K by the K^{th} firing vector, V_K. T implies matrix transpose operation.

$M_K + A^T V_K \geq 0$ for each K.

An example will make these definitions clearer.

Example 1: For the petri net of Fig. 3.1 the A^- and A^+ matrices are:

$$A^- = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

$$A^+ = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

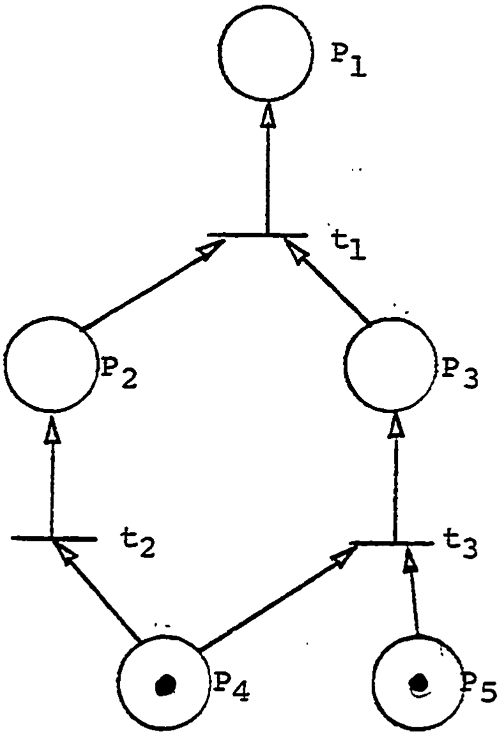


Figure 3.1 Petri Net for Example 3.1

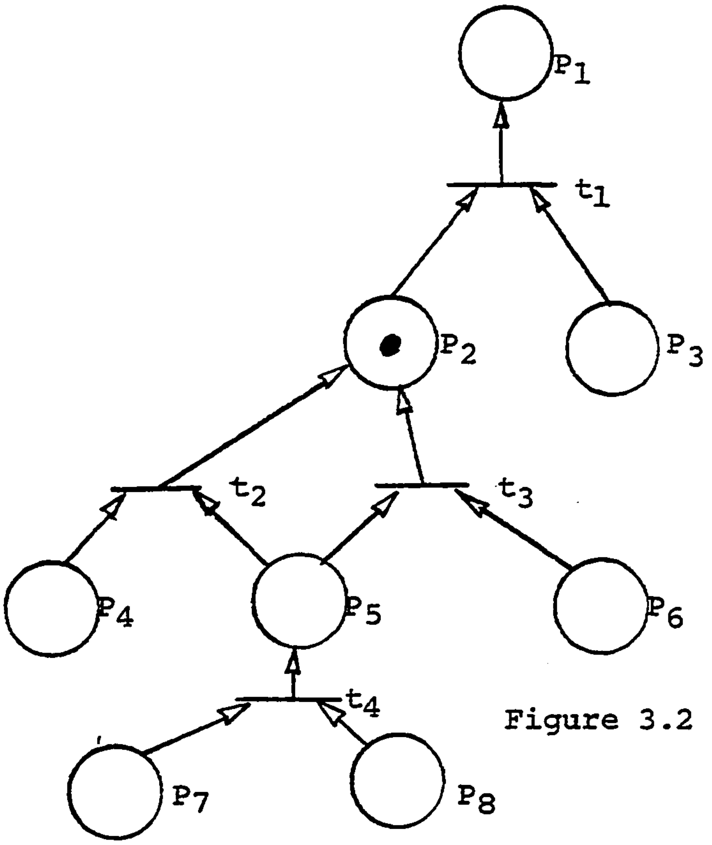


Figure 3.2 Example Petri Net for Illustrating Transition Firing

The matrix A is

$$A = A^{+} - A^{-} = \begin{bmatrix} 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & -1 \end{bmatrix}$$

The initial marking $M_o = [0\ 0\ 0\ 1\ 1]^T$. The marking M_1 resulting from firing t_2 and t_3 is:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & -1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

For the petri net generated for a given fault, we are interested in transitions that have been fired after driving the machine into a search state S. Thus, we shall let M_S denote the marking or state vector after reaching state S. Then M_{S+} denotes the marking vector after all firable transitions have been fired.

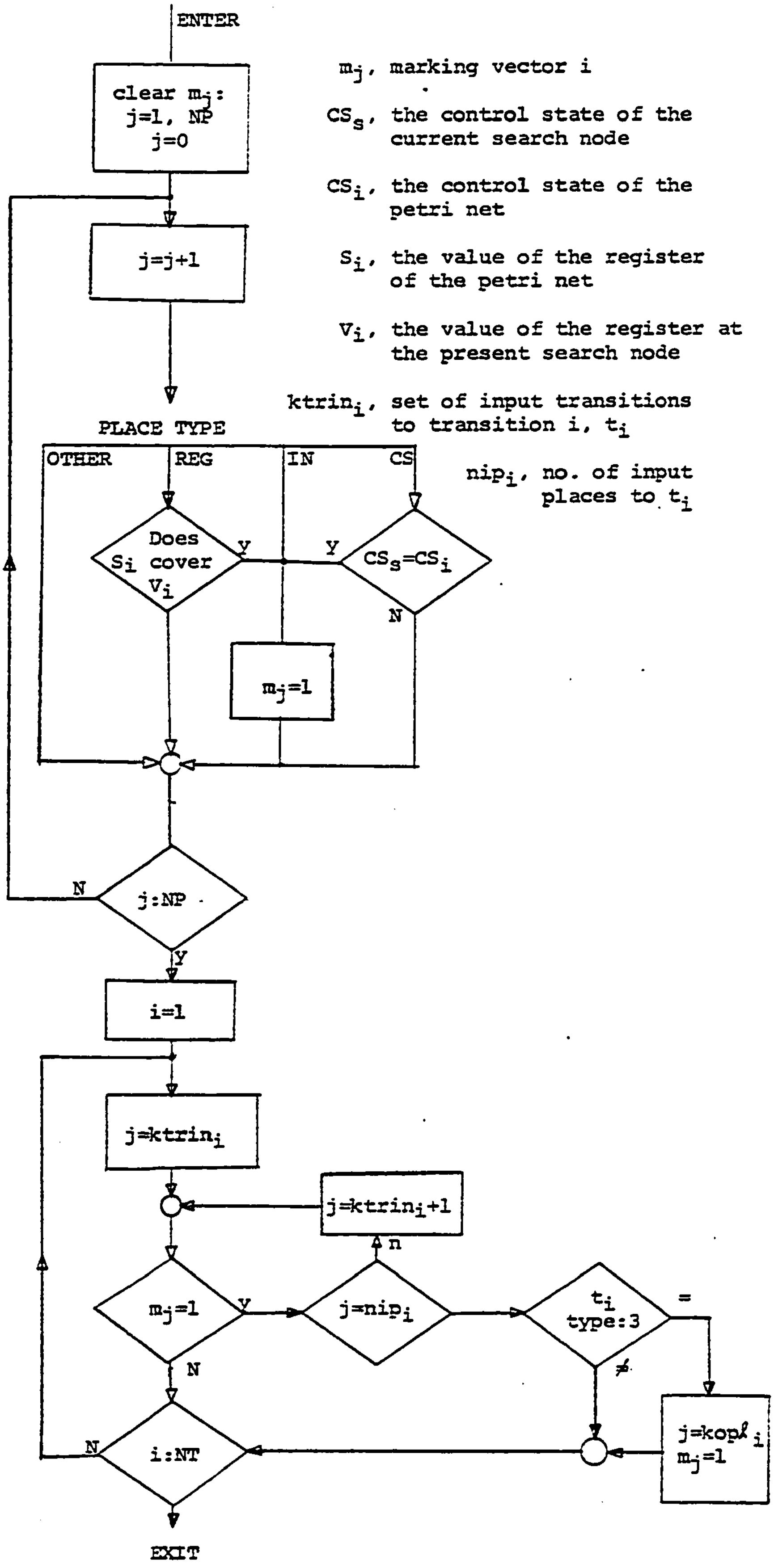
The state equation of a petri net for a given fault is now written as:

$$M_{S+} = M_S + A^T V_S \qquad (2)$$

where A is the matrix defined in Defn. 3.3 and V_S is the firing or control vector, at search state S, that defines which transitions are to be fired.

During the sensitization search, starting from the initial state, each unique state is numbered and called a node. Hence, the marking vector M_S can also be written as M_i where i is the node number that is associated with search state S. M_S gives the conditions fulfilled at search state S. In our model, there is a transition time associated with all but the type 3 transition (section 2.3). For this type of transition, if all the input places are filled with tokens, it is fired. This is not the case with all other types of transitions; they require time. For example, if conditions for loading a register are fulfilled at search state S, the register will be loaded during the next clock period. Thus, M_{S+} in equation (2) will add the outputs of those transitions that have no time associated with them to the state vector M_S.

To compute M_{S+}, we have two choices: either use the arithmetic and matrix operation of equation (2) or use the data structure of the petri net together with the information in the search state S to derive M_{S+}. In the former case, we have to deal with large sparse matrices A^+, A^- and A. When the algorithm was written, it was apparent that there would be a waste of computer memory. Hence, we chose the second alternative: relying on the data structure of the petri net and the search state to derive M_S and M_{S+}. The algorithm for doing this is shown in Fig. 3.3. The first section of the algorithm compares the present machine state and register contents with the machine state and register contents needed to place a token



m_j, marking vector i

CS_s, the control state of the current search node

CS_i, the control state of the petri net

S_i, the value of the register of the petri net

V_i, the value of the register at the present search node

ktrin$_i$, set of input transitions to transition i, t_i

nip$_i$, no. of input places to t_i

Figure 3.3 Derivation of Marking Vector

in each place in the petri net. This portion of the algorithm is, of course, similar in the problem reduction graph (page 75, Huey, 1975). The second section tests if all the input places of a transition are filled with tokens. If so, the output place of the transition is filled with a token (the transition fires) if the transition is of type 3.

3.3 Implied Transition Firing

In the application of petri nets to fault detection, we are interested in the goal place being filled with a token. Thus, if any place, say P_2 of the net in Fig. 3.2 is filled, we have to be concerned with which transitions were fired or can be inferred to be fired for that particular place to be filled with a token. For P_2, either t_2 or t_3 or both might have been fired at the K^{th} firing for it to be filled. After P_2 is filled, only P_3 must be filled for t_1 to be fired, filling the goal place with a token. Hence, after the K^{th} firing, once P_2 is filled, we will consider all transitions that are descendants of P_2 to be fired since they are of no interest. A transition descendant of a place is a transition that fires to have the place filled with a token. In Fig. 3.2, t_2 and t_3 are both descendants of P_2. By similar reasoning, transition t_4 is an (immediate) descendant of t_2 and t_3.

The marking vector M_{K+} after the K^{th} firing for Fig. 3.2 is

$$M_{K+} = [0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^T.$$

Recall that this gives the places that have been filled with tokens after the K^{th} firing. From M_{K+}, we can find all transitions in the petri net that were fired or can be inferred to be fired. We define a new vector R_K:

<u>Defn. 3.3</u>: The K^{th} transition status vector $R_K = \{r_i\}$ is a $1 \times t$ vector having entries r_i where

$$r_i = \begin{cases} 1 & \text{if transition i was fired or can be inferred} \\ & \text{to be fired during the } K^{th} \text{ firing.} \\ 0 & \text{otherwise.} \end{cases}$$

The notion of implied transition firing is actually related to the heuristic function development which is presented in section 3.4. When a set of places is filled at a search state S, then we attempt to identify the set of transitions which need no longer be considered as being necessary to fire before filling the goal place with a token. Put in another way, if say P_2 of Fig. 3.2 is filled with a token, then we pose the question: "Starting from the terminal nodes and transitions, which transition firing sequence might have caused P_2 to be filled with a token?" In this case it must have been the sequence $t_4 \rightarrow t_3$ or $t_4 \rightarrow t_2$. The notion of implied transition firing is not found in the natural functioning of the petri net.

What happens if there are loops in the petri net? This is simply dealt with during the construction of R_K, the transition status vector. R_K is derived from M_{S+};

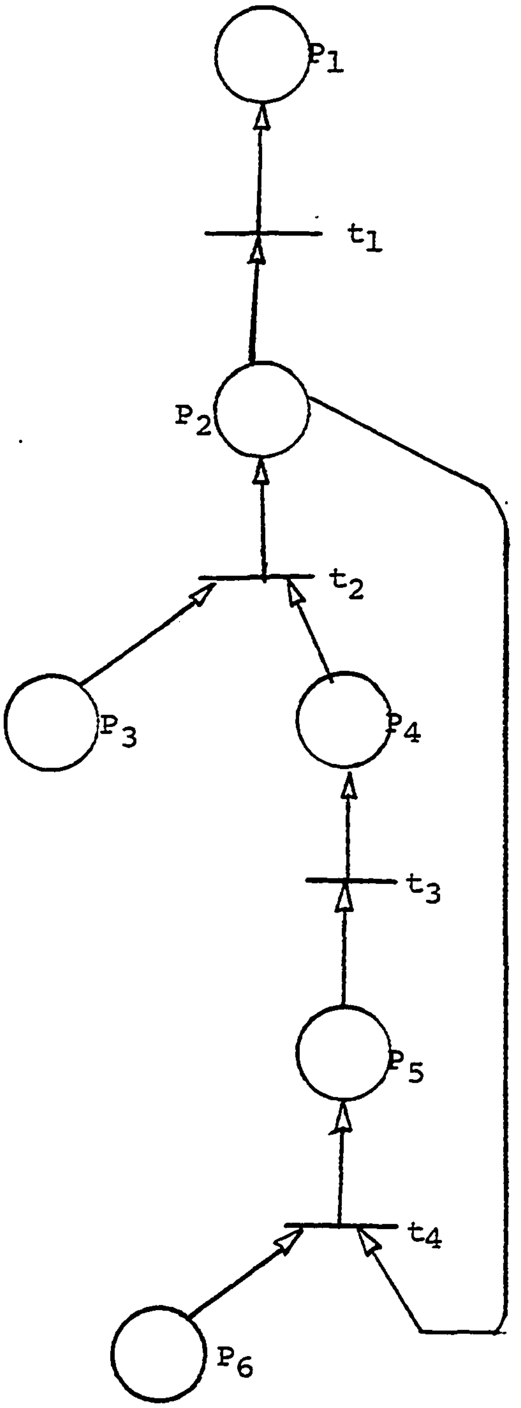
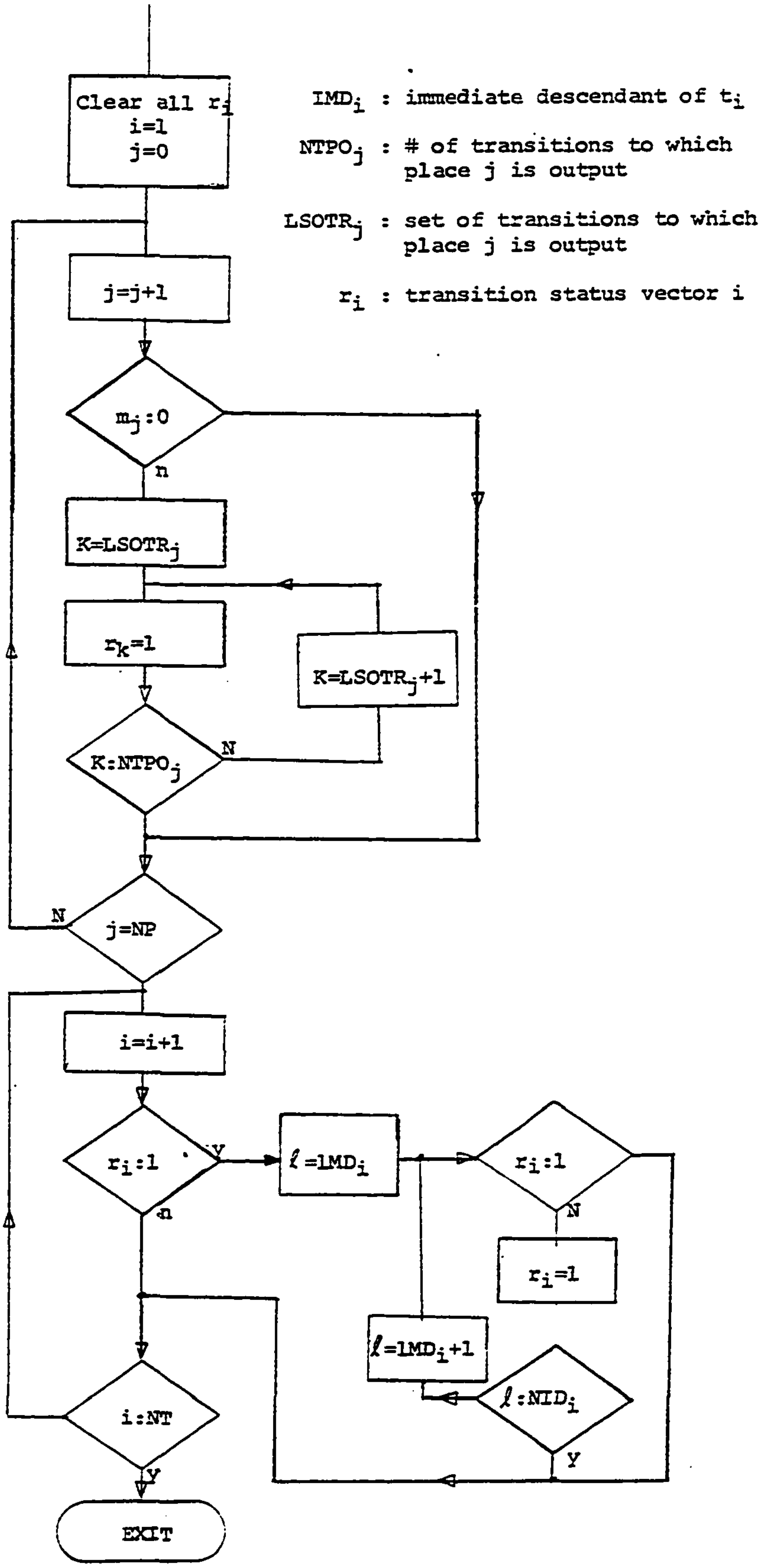


Figure 3.4 Dealing with Loops in the Petri Net



IMD_i : immediate descendant of t_i

$NTPO_j$: # of transitions to which place j is output

$LSOTR_j$: set of transitions to which place j is output

r_i : transition status vector i

Figure 3.5 Derivation of R_K

during the derivation process we enumerate descendant transitions of a place that is filled with a token. During the enumeration process if any transition is already marked in R_K the whole process is terminated. In Fig. 3.4, if P_2 is filled, the algorithm of Fig. 3.5 which derives R_K would detect a loop between the transitions t_2 and t_4. Since t_2 is the transition descendant of P_2, it is marked first in R_K. The immediate descendant of t_2 is t_3 while t_4 is the immediate descendant of t_3. In attempting to mark the immediate descendant of t_4 (which is t_2), it is discovered that t_2 is already marked and the process is terminated.

3.4 The Heuristic Function

For each node that is reached during the sensitization search, we would like to compute a heuristic cost value based on information from the petri net. Our aim is to indicate which node is most likely to be useful in finding the goal node. For sensitization search state S, we seek to minimize the heuristic cost function H(S); then for all nodes that are candidates for expansion, we choose that which has the minimum cost value H(S) as the most promising.

For each search state S, our main concern is: how can the machine be moved nearer the goal from state S. This question must be answered from the petri net. Three options seem appealing, either:

a. use the places that have been filled in the petri net at search state S,

b. use the transitions that have been fired at state S, or

c. use a combination of both the places and transitions

as an indicator of nearness to the goal. The background discussion of section 2.2 on petri nets will be helpful in understanding the present discussion. Remember that we use petri nets to model "conditions" represented by places and "events" represented by transitions.

To use both the places filled and transitions fired as our indicator of nearness to the goal, i.e., to compute H(S) would be superfluous since the module of Fig. 2.1 represents an information processing element.

When a place is filled with a token, it indicates a condition has been fulfilled. Hence, it is possible to use the places (conditions) filled with tokens (fulfilled) to indicate how near we are to the goal. However, to be dealing with the places instead of the transitions, we have to spend more time detecting loops between places and this can slow down the search. Also, in the petri net, it is more natural to be concerned with the firing of transitions and transition firing sequence.

The firing of a transition indicates an "activity" has taken place--in our model there has been, say, a change of control state, for example. Our interest is to indicate how this affects the overall behavior of the machine, for that matter, how near we are to the goal. From these considerations, we choose as our basic measure, the number of fired transitions in the petri net.

The transition status vector, R_K defined in Defn. 3.3 actually constitutes a mask on the transitions in the petri net that are no longer of interest to us; we might think of these transitions as having been fired already. Hence, for each search state S, we can compute the heuristic cost function as:

$$H(S) = N_t - \sum_{i=1}^{Nt} r_i \qquad (3)$$

where N_t is the total number of transitions in the petri net

$R_K = \{r_i\}$ is the transition status vector.

Consider Fig. 3.6. If for state A, P_2 is filled, then the marking vector M_{A+} is

$$M_{A+} = [0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^T.$$

The transition status vector would be derived as explained in the previous section to be

$$R_K = [0 \quad 1 \quad 1 \quad 1 \quad 0]$$

then $\qquad H(A) = 5 - 3 = 2.$

If, on the other hand, state B has P_5 filled with a token we would have:

$$M_{B+} = [0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0]^T$$

and $R_K = [0 \quad 0 \quad 0 \quad 1 \quad 0]$.

H(B) would be computed as:

H(B) = 5 - 1 = 4, indicating the importance of state A over state B.

The simple expression of equation (3) is not satisfactory when a terminal place of a transition is filled with a token but the transition itself is not fired. Specifically, in Fig. 3.6, if for state A, P_4 and P_6 are filled with tokens, then

$$M_{A+} = [0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0]^T$$

and $R_K = [0 \quad 0 \quad 0 \quad 0 \quad 0]$ since no transition was fired.

Now, if for state B, no place of the petri net is filled with a token, then

$$M_{B+} = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$$

and $R_K = [0 \quad 0 \quad 0 \quad 0 \quad 0]$.

For both states A and B, the heuristic function computed from equation (3) would be:

H(A) = H(B) = 5.

Intuitively, state A should be nearer to our desired goal than state B.

This suggests that terminal places must be given special treatment in the computation of H. The modified expression for H now becomes:

$$H(S) = N_t - [\sum_{i=1}^{Nt} r_i + \sum_{j \varepsilon P_T} \frac{\delta_j}{n_j} m_j^{+}] \qquad (4)$$

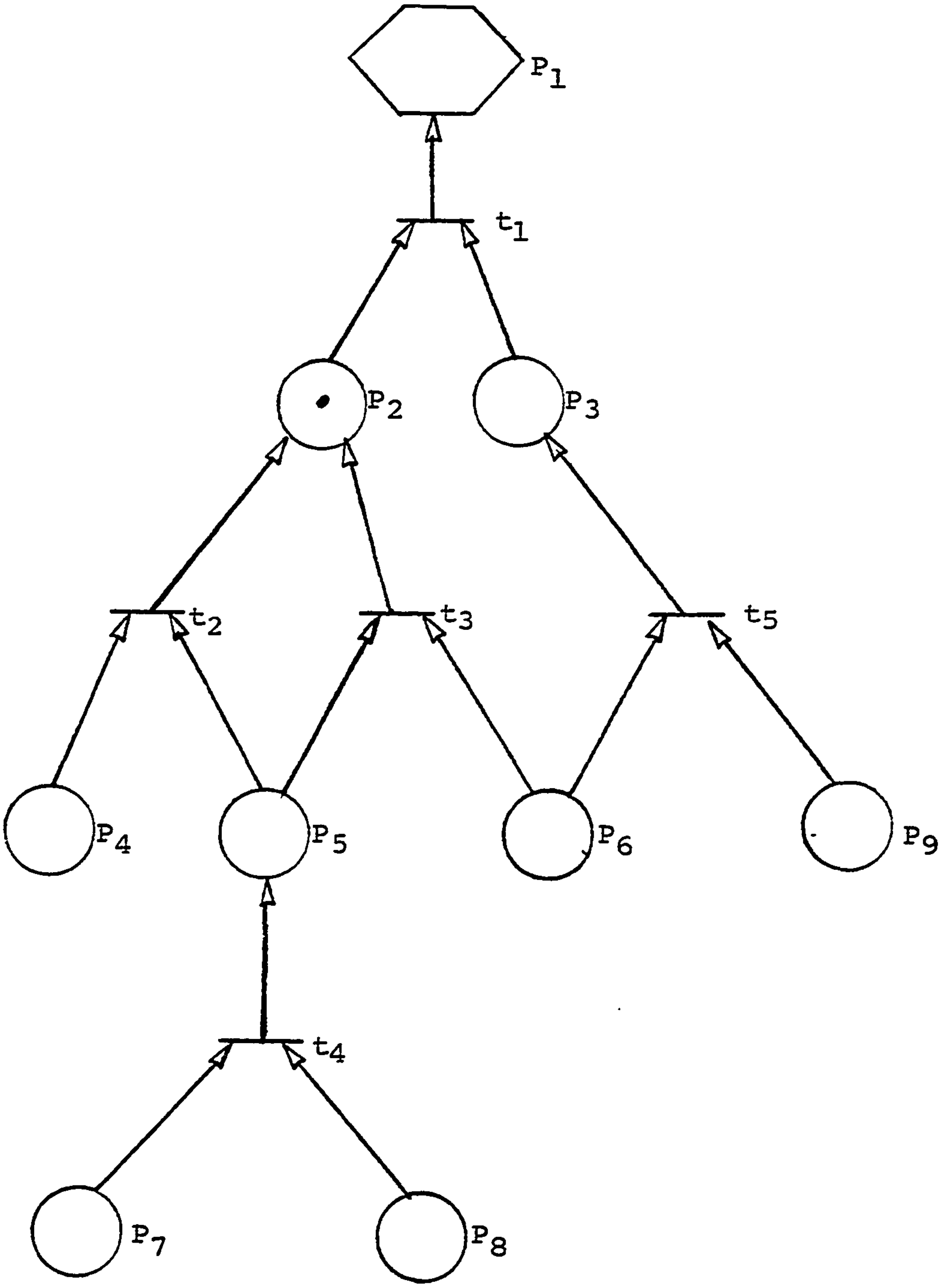


Figure 3.6 Petri Net for Illustrating Heuristic Function Computation

where

$M_S^+ = \{m_j^+\}$ the marking vector after all transitions have been fired in state S

n_j = number of brothers of P_j

$$\delta = \begin{cases} 0 & \text{if } P_j \text{ is an input to any member of } R_s \\ 1 & \text{otherwise.} \end{cases}$$

The last term of equation (4) is the one that computes contributions from terminal places that are filled but their associated transitions have not been fired. The δ factor takes care of this situation. It is assumed that for a transition to fire, each filled place contributes a fraction $1/n_i$ where n_i is the number of places input to that transition.

Applying equation (4) to the two states A and B mentioned in the previous paragraph, with:

$$M_A{}^+ = [0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0]^T$$

$$R_K{}^+ = [0 \quad 0 \quad 0 \quad 0 \quad 0]$$

and

$$M_B{+} = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$$

$$R_K{+} = [0 \quad 0 \quad 0 \quad 0 \quad 0].$$

Using equation (4) we have

$$H(A) = 5 - [0 + \tfrac{1}{2} + \tfrac{1}{2} + \tfrac{1}{2}] = 3.5$$

and

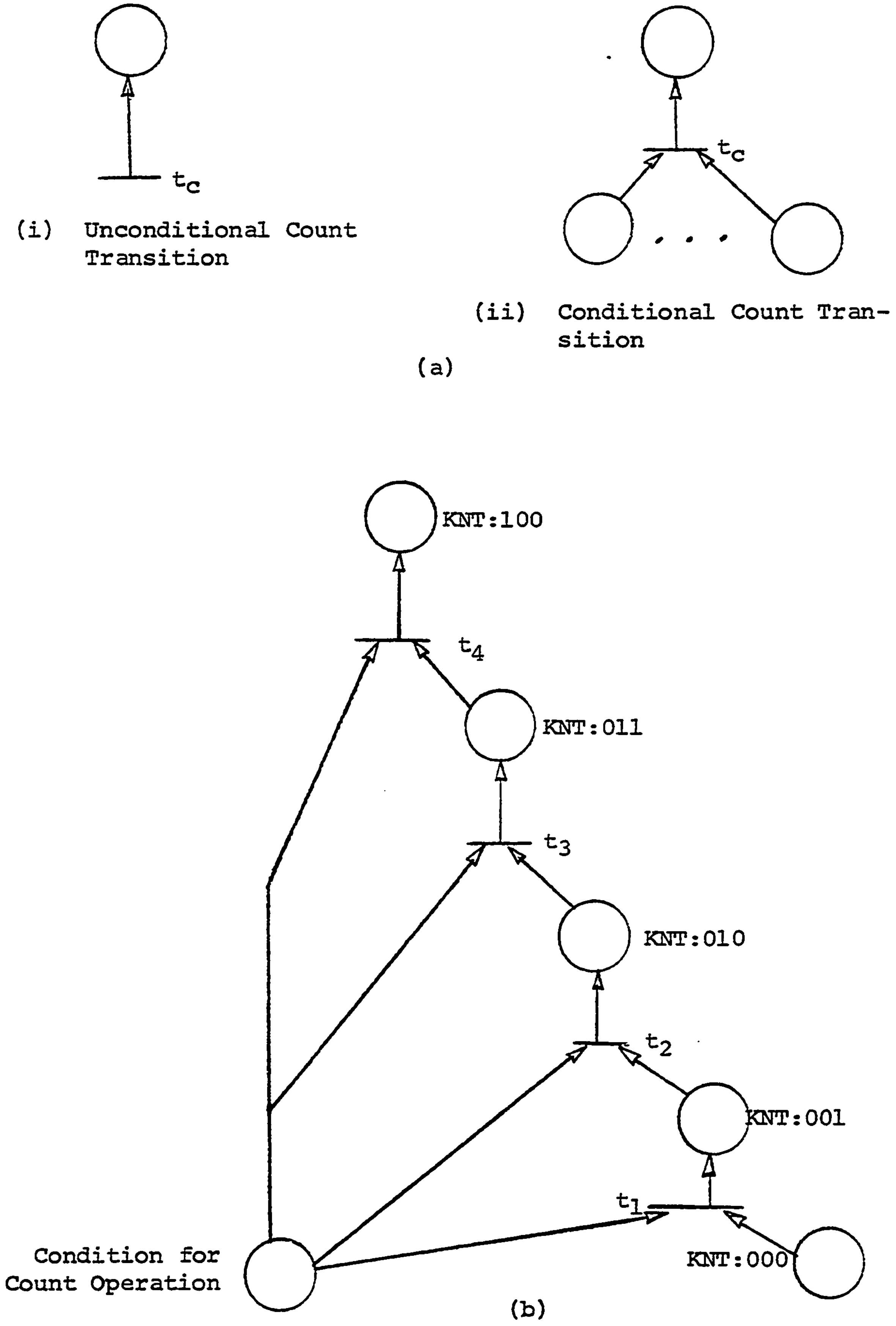
$$H(B) = 5 - [0] = 5$$

which indicates correctly the importance of state A over state B. Note also how the function treats the importance of terminal place P_6 which is an input place to more than one transition.

3.4.1 Counter and Shift/Rotate Transitions

In chapter two we introduced the count transition; this is a transition that models the change of state of a counter. Similarly, the shift/rotate transition models the change of state of a shift register. The model of the counter transition shown in Fig. 3.7(a) is actually a compression of what would be a series of transitions and places. Consider a counter that counts from 0 to 4. There is a change of state of the counter, that is, a transition whenever the required conditions are fulfilled (i.e., the condition place is filled with a token). Thus, if the counter is in state 0, and if the condition place is filled, the transition t_1 will "fire" during the next clock period. This must be repeated four times before the desired state KNT:100 can be reached (Fig. 3.7 b). This suggests that when a counter transition is included in the petri net, we increase the number of transitions in the petri net by the number of times the counter must count before reaching its goal state. For Fig. 3.7(b), we might have to increase the number of transitions by 4.

This approach would not give an accurate guidance to our search routine. If, for example, a counter is enabled and loaded with 0110 (binary six), counts to 1010 (ten), and then is disabled, and we then add 15 = 16 - 1 to the total number of transitions, we have not given an accurate representation of the counter operation. Thus, the heuristic function computed on this basis would be misleading.



(i) Unconditional Count Transition

(ii) Conditional Count Transition

(a)

(b)

Figure 3.7 Modelling Count Transition

An alternate and more accurate approach is to consider the output place of the count transition as a goal and then compare the state of the counter during the search with the goal. Let bd(i,s) be the arithmetic binary difference between the value in the counter at the present search state S and the goal state of the counter. Then

$$bd(i,s) \;=\; \perp(g_1 \ldots . g_n) \;-\; \perp(V_1 \ldots . V_n)$$

where $g_1 \ldots . g_n$ = goal state of the counter

$V_1 \ldots . V_n$ = value in counter i after the current search state S.

Obviously, if bd(i,S) is zero, then our goal is reached and t_c fires, hence the contribution from the counter transition is one. However, if bd_i is not zero, then we are a distance of bd(i,S) from the goal and the contribution from the count transition is

$$[1 - \; bd(i,S)/\perp\,(g_1, \ldots . g_n)]. \qquad (5)$$

Note that although we have chosen the distance between the goal state of the counter and the value in the counter at state S to measure our nearness to the goal, we are in essence answering the question: "how far is the count/shift transition from firing?" Hence, it is the transition firing that is actually our measure of nearness to the goal. Accordingly, expression (5) takes on values ranging from zero to one.

For shift registers, we define bd(i,S) as the number of times the shift register must be shifted (left or right) from the present state S so as to satisfy the desired goal.

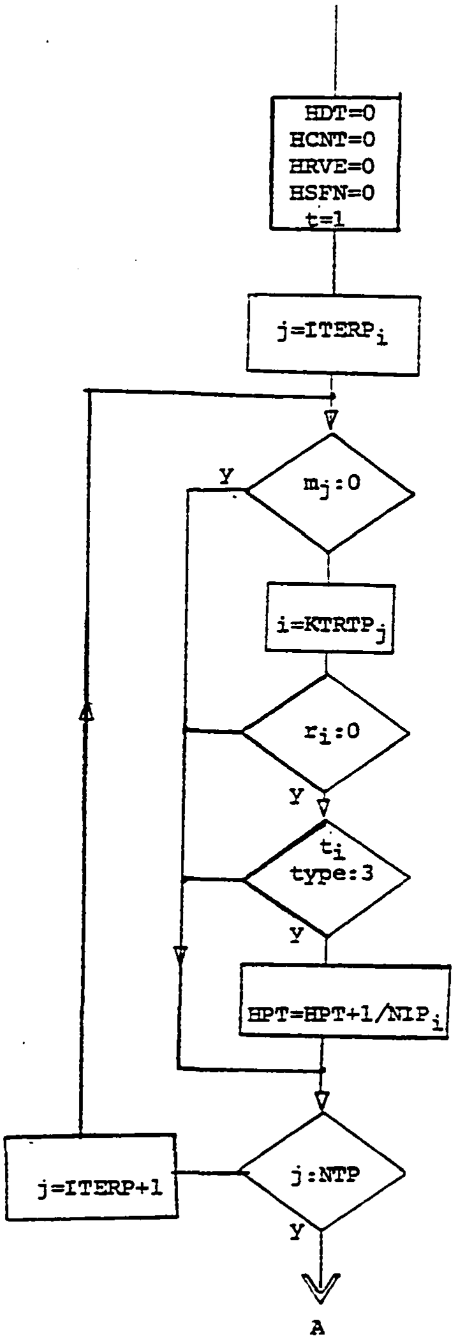
Shift registers and counters are very highly sequential circuits that are very troublesome in fault test generation, especially when they are buried. Expression (5) enables us to provide proper guidance at each state of the search; for ordinary registers we give guidance based on whether the register contains the correct vector or not; guidance for shift registers/counters goes further than that. If the shift register or counter does not contain the correct value, we compute how far it is from reaching the required value. Needless to say, it is impossible or very difficult to use the same criteria for ordinary registers.

Adding the contribution from the counter and shift/rotate transitions, the heuristic cost function of equation (4) now becomes

$$H(S) = N_t - [\sum_{i=1, i \notin c}^{Nt} r_i + \sum_{j \in p_T} \frac{\delta}{nj} \quad m_j^+$$

$$+ \sum_{I \in C} (1 - bd(i,S)/ \perp(g_1 g_2 \ldots g_n)]$$

where C = the set of count and shift/rotate transitions. The flow chart for computing the heuristic cost function is shown in Fig. 3.8.



HRVE, contribution form R vector

HCNT, contribution from counters

HPT, contribution from terminal placer

HSFN, Heuristic Cost Value

t_i, transition i

ITERP$_i$, set of terminal places

KTRTP$_j$, set of transitions to which place j is input

NIP$_i$, no. of i/P places to t_i

NTP, no. of terminal places

Figure 3.8 Heuristic Cost Function Computation

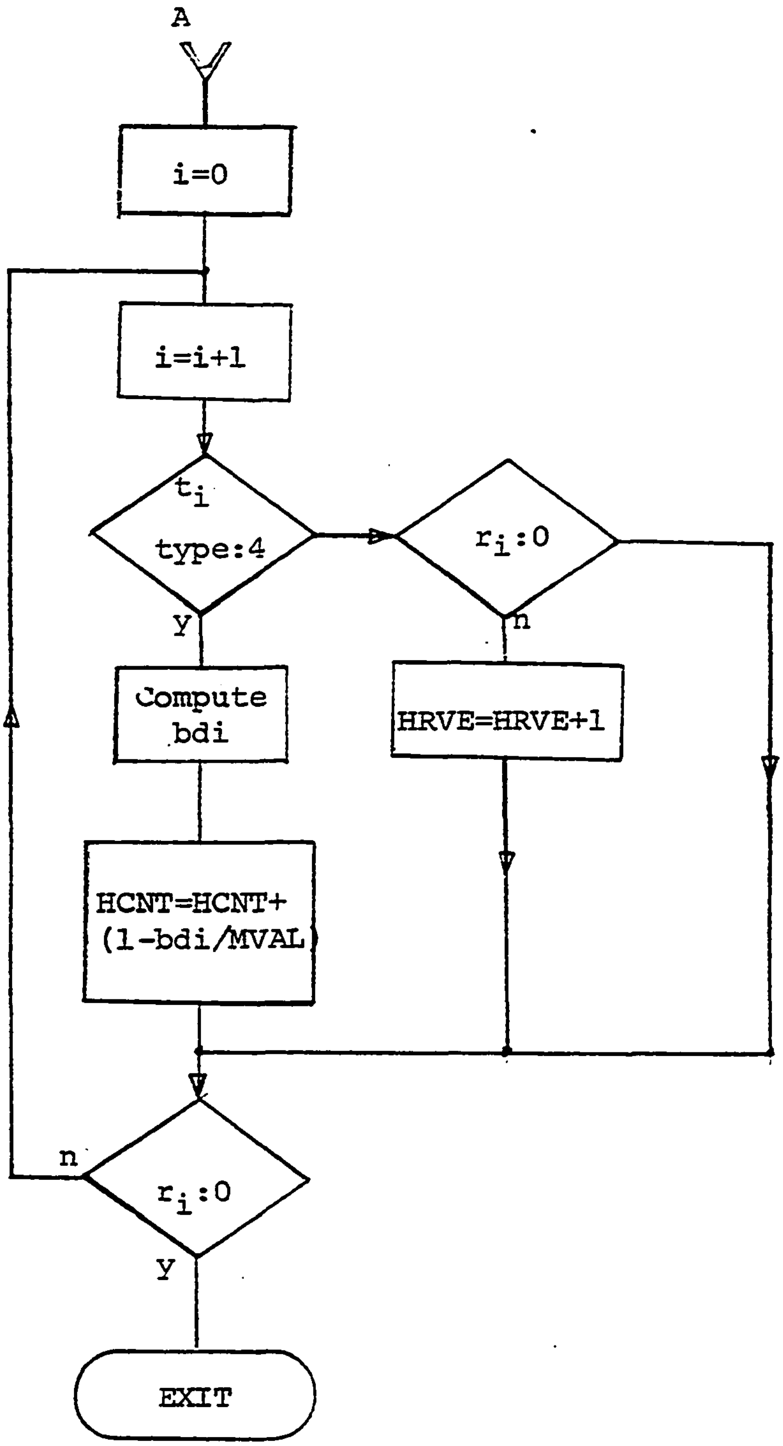


Figure 3.8 cont'd

CHAPTER IV

INPUT VECTOR GUIDANCE

4.1 Introduction

In the last chapter we developed a guiding mechanism for a search by finding a heuristic value that enables us to indicate the nodes most likely to lead to the goal. The second guidance mechanism used in SCIRTSS is the reliance on user supplied input vector tables (Carter, 1973). SCIRTSS III attempts the selection of these input vectors automatically (Huey 1975, p. 81).

In sensitization searches a branch is made to each possible next control state from the current control state of the search node being expanded, if an input vector exists which satisfies the conditions of the control branch. Although the search routine itself generates part of the input vector needed to satisfy the control branch condition, we can derive information from the petri net to make this process more efficient. Also, the sensitization search can use a great deal of guidance where data inputs are concerned.

In a Computer Hardware Description Language like AHPL where each input-to-register transfer is associated with a control state, the input vector suggestions can be grouped by the control state at which each is to be applied. In this way, at any given control state an input vector can be readily available for use to increase the search efficiency. This has been the approach in SCIRTSS.

In the petri net generated we have places or locations that represent the condition of placing an input vector (a_1, a_2,...a_n) a_i ε(0, 1, X) on an external input. The control states at which these places are filled with tokens can be readily obtained from the petri net. Most of these places or locations are invariant with respect to the goal or the fault under consideration as they are generated from control expressions. Hence, the control state associated with these places can be obtained in the form of subnets before the commencement of any sensitization search.

4.2 Selecting Input Vectors

In some circuits, many input vectors may be associated with a given control state. In SCIRTSS, when attempting to expand a node, the search routine applies each suggested input vector to all control branches. Thus, for a node that has m input vectors and n successor control states, the search expands m x n next states. Hence, these input vectors must be judiciously selected to avoid misleading the search.

However, this selection process is not a trivial issue. In fact, each input place that appears in the petri net is important, for if a given input place is never filled, it may be impossible to reach the goal!

In our input vector selection process, we classify the input vectors according to the two main types of expressions in AHPL:

Control Branch Expression

and Register Transfer Expression.

4.2.1 Control Branch Input Selection Procedure

Generally, a conditional control branch is made in AHPL depending upon

a) some input signal, such as ready, link, etc. We call such a signal control signal.

and/or b) the bit combinations in some register(s), many of which are loaded directly from the external inputs at some control state(s).

Both the control signals and registers that are responsible for branching from one control state to another appear in the petri net. In the case of control signals, we can select the values of these signals so as to prevent the generation of unnecessary nodes during the state space search. In particular, consider the examples

(i) k $\quad \mathrm{OUT} \leftarrow \mathrm{A}$

$\rightarrow (\overline{\mathrm{ready}}, \mathrm{ready})/(\mathrm{k},\mathrm{j})$

(ii) k $\quad \mathrm{A} \leftarrow \mathrm{INC(A)}$

$\rightarrow (\overline{\wedge/\mathrm{A}} \wedge \mathrm{ready}, \wedge/\mathrm{A})/(\mathrm{k},\mathrm{j})$

In both examples, the machine waits in control state k until some condition is fulfilled. During the search, should this behavior be simulated? Not necessarily; for if we examine the register transfer at cs.k of example (i), it would be a waste of time to continue looping to control state k to be performing the same register transfer. On the other hand, it is essential to repeat the counter operation of example (ii). Hence the search must branch to cs.k whenever it enters control state k.

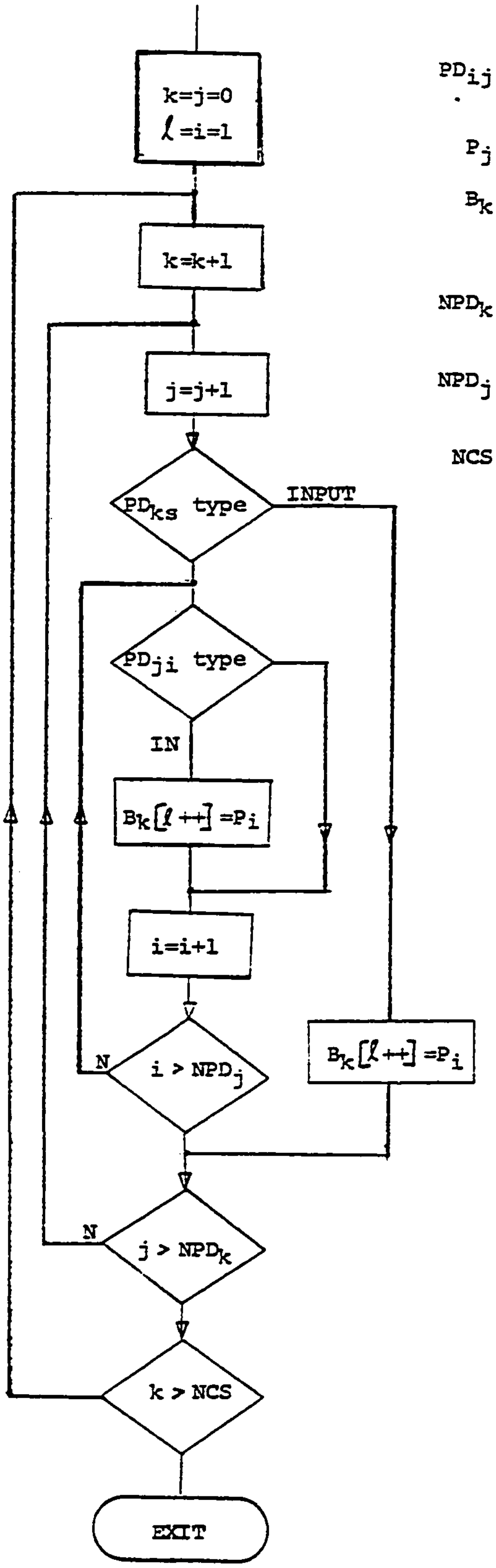
The examples of the preceeding paragraph indicate that we can control the control states that the machine branches to during the search in order to improve the search efficiency. This is done during the branching function generation. The petri net contains information on the control signals that cause branching from one control state to another. If the value of a control signal causes the machine to wait in any particular control state and neither a count nor shift operation takes place in that control state, then the machine is not allowed to loop in that control state during the search. Because we have the count and shift transitions in the petri net, these conditions are easily detected.

The more common method in which a conditional control branch is made in AHPL depends on the contents of some register, for example an instruction register in a computer. Many of these registers are loaded directly from the external inputs at some control states. Frequently there are too many input vectors to be applied at the respective control states and we have to choose only about two or three of these input vectors. The importance of properly selecting these input vectors is not hard to see: by leaving out some input vectors it may not be possible to visit some control states(s) and the penalty can be very high.

Ideally, we should select the input vectors such that it would be possible to visit all control states in the petri net. This philosophy is not without danger, however. In some cases, there may be more than one goal. Thus if <u>any</u> of the control states say, i_1, i_2, i_3 is reached and the correct register is loaded then the search is successful. In this case, although many control states may appear in the petri net, it seems appealing to select a subset of these control states and aim at reaching only members of this subset. This approach would nullify the advantage of having more than one goal and of course it is very difficult and time consuming to select a subset of control states. We aim at selecting the input vectors such that it would be possible to visit all the important control states in the petri net.

Defn 4.1: The Control State Branch Vector, $B_{jk} = \{s_1, s_2, ..s_n\}$ is a set of input vectors s_i, that can cause a transfer from control state j to control state k. The set of all Control State Branch Vectors is denoted by $B_k = \{B_{jk}, .., B_{mk}\}$.

The vector B_k should not be confused with the input vectors that can be applied at control state k. In the latter case, we have input vectors which, if applied when the machine is in control state k, causes some register transfer. The Control State Branch Vector, B_k, on the other hand consists of input vectors which are actually responsible for the machine ever branching to cs.k. B_k is derived from the control state subnets as shown in Fig. 4.1. In this figure, we mark all register places that have been encountered for easy identification when we are selecting input vectors in section 4.2.3. In the petri net, an immediate predecessor of a place i or location



PD_{ij} : j^{th} immediate predecessor of place i

P_j : place j

B_k : set of input vectors that cause branch to CS.k

NPD_k : no. immediate predecessors of CS.k

NPD_j : no. immediate predecessors of place j

NCS : no. of CS

Figure 4.1 Construction of CS Branching Function

is a place j that is input to a transition that fires to fill place i with a token.

The vector B_k is independent of all faults and thus can be constructed once for all sensitization searches.

The next step in the Control Branch Input selection is the formation of a Common Transfer Vector.

Some input vectors in each B_k cover other input vectors in some B_j. Hence we define a common transfer vector,

$$F_L(k_1,k_2,\ldots) = \left\{s_1,s_2,\ldots s_3\right\} \quad L = 1,2,3\ldots$$

as the set of input vectors s_i that are common to control states $k_1,k_2,\ldots$ The vector F_L is easily derived by checking if each input vector s_i in B_{k_1} covers any other input vector in B_{k_2}.

We give an illustration at this point. In the next chapter, we will present a 4-bit microprocessor as a case study. The Control Branch Vectors for this circuit are:

$$B_5 = \{4,\text{ics:X001, 1XXX}\}$$

$$B_6 = \{5,\text{ics:XX01, X01X}\}$$

$$B_8 = \{5,\text{ics:10XX}\}$$

$$B_{10} = \{5,\text{ics:X1X0; 6,ics:X1XX}\}$$

$$B_{12} = \{10,\text{ics:XX1X}\}$$

B_{14} = {12, ics:XXX0; 15, ics:X01X, ics:X100; 10, ics:XXX0X}

B_{15} = {4, ics:X01X, ics:X000, ics:110X}

B_{16} = {4, ics:0110}

B_{18} = {4, ics:0111}

B_{19} = {15, ics:X100, ics:X101}

B_{20} = {15, ics:X000}

There is a 4-bit input line, ics, that is used to load the Index Register. The machine has twenty control states. Those control states that are not directly controlled by input vectors do not, of course, appear in the Branch Vectors. The common transfer vectors derived from the B_k's are:

F_1(4,5,6,8,14) = {ics:1001}

F_2(4,5,10,12,14) = {ics:1110}

F_3(4,6,10,5,15,19) = {ics:1101}

F_4(4,5,10,14) = {ics:1100}

F_5(4,14,5,6,8) = {ics:101X}

F_7(4,15) = {ics:X000}

F_8(4,16) = {ics:0110}

F_9(4,18) = {ics:0111}

F_{10}(4,15,19) = {ics:1100; ics:1101}

F_{11}(4,15,20) = {ics:0000}

Comparing the transfer vectors F_7 and F_3, we observe that the input vector cs:X000 will cause a branch to control state 15 only while cs:1101 will cause a branch to control state 15 and control state 14. Hence the input vector cs:1101 can replace cs:X00 and we say that F_3 has overridden F_7. To test those transfer vectors that have been overridden, we use the expression:

$$F_i(k_1,k_2...) - F_i(k_1,k_2...) \cap F_j(k_1,k_2,k_3,...) \qquad (4.1)$$

for i = 1,2,...n; j = 1,2,...n; i ≠ j where n is the number of common transfer vectors. If expression 4.1 is empty, then F_i is overridden by F_j and F_i is deleted together with its corresponding input vector.

The danger with the test of 4.1 is that the input vector picked would let the machine wander from one control state to another. For example,ics:X000 of F_7 would branch from control state 4 to control state 15. However, since F_3 overrides F_7, ics:1101 replaces ics:X000. In this case to reach control state 15, the machine might have to visit control state 14 before reaching control state 15! This is our dilemma: on the one hand trying to limit the number of inputs and on the other hand the "best" selected inputs periodically wandering from one control state to another. However, it is better to be able to visit many control states with a few input vectors than being unable to do so at all.

Applying the test of expression 4.1 to the Common Transfer Vector of our example, we have F_1, F_4, F_7 and F_{10} overridden. The Reduced Common Transfer Vectors are:

$$F_2(4,5,10,12,14) = \{\text{ics:1110}\}$$

$$F_3(4,5,6,10,15,19) = \{\text{ics:1101}\}$$

$$F_5(4,5,6,8,14) = \{\text{ics:101X, ics:1001}\}$$

$$F_8(4,16) = \{\text{ics:0110}\}$$

$$F_9(4,18) = \{\text{ics:0111}\}$$

$$F_{11}(4,15,20) = \{\text{ics:0000}\}$$

Hence from the initial 20 Control Branch Input vectors, we have six vectors in the Common Transfer Vectors. Which one of these should be selected?

Our final reduction process calls for the removal of any control state that is common to all the Reduced Transfer Vectors. The resulting vector is called a "G Common Transfer Vector." For our example, we have

$$G_1(5,10,12,14) = \{\text{ics:1110}\}$$

$$G_2(5,6,10,15,19) = \{\text{ics:1101}\}$$

$$G_3(5,6,8,14) = \{\text{ics:101X, ics:1001}\}$$

$$G_4(16) = \{\text{ics:0110}\}$$

$G_5(18) = \{ics:0111\}$

$G_5(15,20) = \{ics:0000\}$

In this example, we have six input vectors that determine the control states that the machine can branch into. We must only select two or three of these input vectors for application at the required control state. Several factors need be taken into account when selecting input vectors from the G Common Transfer Vectors.

(i) The input selection procedure outlined in the preceeding paragraphs is completely independent of the fault being sensitized; the G Common Transfer Vectors are derived once for all sensitization searches. Hence when the decision is made to select input vectors, the input vectors in the G Common Transfer Vectors are selected based on the control states in the Common Transfer Vectors and the petri net for the fault. The control states that do not appear in the petri net for the particular fault are dropped from the Common Transfer Vector for this fault. If any of the G_i becomes empty then it is dropped from consideration.

(ii) Any of the G_i which contains a control state that is one of the goals for the sensitization search should certainly be included.

(iii) For those G_i that have only one control state and the control state is not one of our goals, we have to check if any register transfer takes place in the particular control state. If not, the input vector associated with the control state can be dropped from the list. On the other hand, if a register transfer takes place and the only way that transfer can take place is for the machine to be in that particular control state, then the control state may be important.

We formalize the discussion above by computing a factor of importance, q, for each Common Transfer Vector that is left after all control states not appearing in the petri net have been dropped. For each G_i, we have:

$$q_i = mnp \tag{4.2}$$

where m = the number of control states in G_i; generally, n = p = 1. However, if any member of G_i is a goal control state, then n = M, where M is the largest value of m. The constant, p, takes care of those G_i that have only one control state as a member. If a count or shift operation occurs in that control state or the only way a register transfer takes place in the machine is when the machine is in that particular control state, then p is made equal to 2 to reflect that importance.

After computing the factor, q_i, for all G_i, the two (or three) input vectors that have the highest factor of importance are selected; these input vectors would control the states the machine visits during the search.

4.2.2 Register Transfer Input Selection

The second type of expression from which input vectors must be generated is the register transfer expression. This usually consists of loading a register with an external input at any given control state.

After selecting the input vectors for branching from one control state to another, we must select another two (or three) input vectors which would determine the vectors that are loaded into the various registers. We could approach this selection process in much the same way that we approached the Control State Input selection. However, we can efficiently make use of the transition time of the transition to which the vector is an input place in the petri net and obtain quite an accurate result. We give an illustration of this process.

The places P_4 and P_5 in Fig. 4.2 represent the condition of placing the vectors X100 and 100X respectively on the external input, IN. Both places are associated with the same control state, cs.k. P_4 is an input place to transition t_1 which has transition time $t = 1$ while P_5 is an input to transition t_2 which has transition time $t = 3$. If the machine reaches

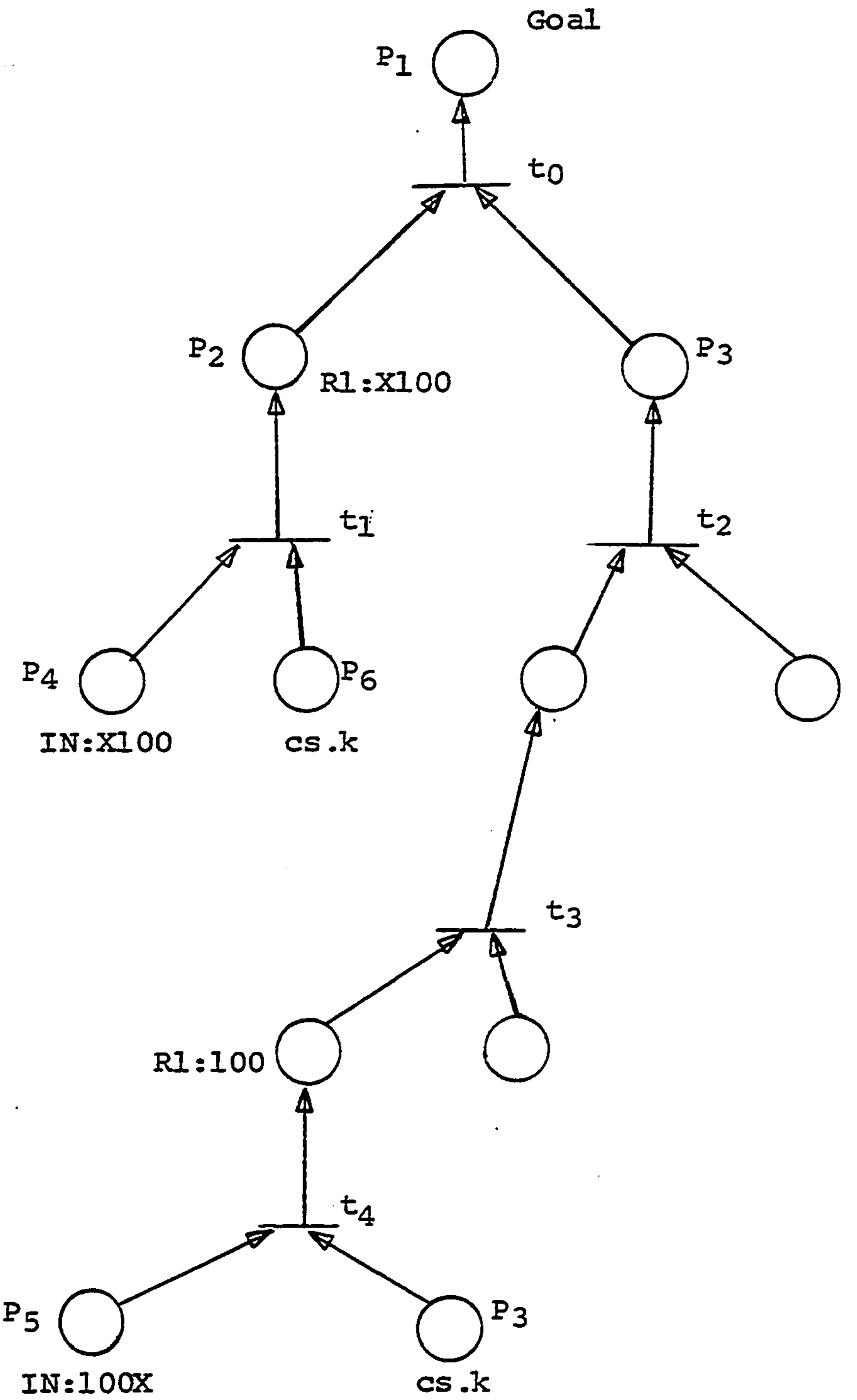


Fig. 4.2 Petri Net Example for Input Vector Weighting

control state K and the input vector {IN:X100} is applied then transition t_2 becomes firable and the goal place would be filled during the next clock period. However, if the input vector IN:100X is selected, transition t_2 would have to fire, followed by transition t_4 and finally t_3 before the goal can be filled. Obviously, input vector IN:X100 is a better choice than IN:100X for our aim is to reach the goal with the least number of input sequences.

This example demonstrates that the information from the transition time of the transitions in the petri net can be helpful in selecting input vectors to be loaded into registers. For an input place P_i in the petri net, we can compute the "weight," W of an input vector from:

$$W(P_i) = q(Q - \tau(i)) \qquad (4.3)$$

where Q = maximum transition time in the petri net.

$\tau(i)$ = transition time of the transition to which P_i is input.

q = a factor indicating how critical the register transfer may be.

The factor q is computed as:

$$q = (c - n)$$

where n is the number of ways the register can be loaded and c is an arbitrary constant selected such that no q_i is zero. The factor q is quite important; if a register A can be loaded in three ways and another register can only be loaded in one way, then the input vector that is used to load B must be more critical than the one used to load A.

For the example of Fig. 4.2, taking c = 2, we have

$$q = 2 - 1 = 1$$

$$Q = 3$$

and

$$W(P_4) = 1 \times (3 - 1) = 2$$

$$W(P_5) = 1 \times (3 - 3) = 0$$

Hence we see the importance of P_2 over P_4.

One may argue that in Fig. 4.2, if the input vector of P_5 is not selected, P_3 may never be filled with a token and as such it would be impossible to reach the goal! This may be true and in fact, the same argument may arise in connection with all the input vectors. The idea is to select the most "promising" input vectors and leave the less critical ones to be generated randomly.

Before weighting the input vectors, we check if these input vectors are covered by any of the input vectors selected by the control branch selection procedure. If so, the particular input vector is not taken into consideration again.

In attempting to select input vectors from register transfer expressions, special attention must be devoted to counters and shift registers. This class of registers represents complex sequential circuits that are troublesome in test set generation. If an input vector must be loaded into any of these special registers it may be critical. In a given machine only a few input vectors may be loaded into a counter or shift register. For these reasons, any input places associated with count or shift/rotate transitions in the petri net are included in the list of input vectors to guide the search.

In section 4.2.1 we discussed how to handle conditional control state branching expressions. One may wonder whether condition expressions which control register transfers require any special treatment. In the AHPL expression:

$$k\cdot \quad A \leftarrow B*cb$$

if the input cb is high, then register B is loaded into register A. In this example there will be only one input vector cb:1 associated with control state cs.k; this input vector will naturally be used to guide the search. However, if there are many input vectors to be applied at cs.k, the input selection procedure of this section will have to be evoked and the input vectors that control conditional register transfers are treated like the other register transfer input vectors.

4.3 Using the Input Vector Selection Procedure

The input vector selection procedure treated in the preceding sections is applied to a given circuit only if the number of input vectors to be applied at a given control state exceeds a user specified number. The optimum number is not known although five (5) has been used for previous SCIRTSS tests and is used for testing in the next chapter. The input vector selection is done once for each sensitization search; the Common Transfer Vectors of section 4.2.2 are constructed only once for each machine while the input vectors from the Common Transfer Vectors are selected after the construction of the petri net.

Naturally, when the number of input vectors per control state is less than the user specified number for all control states in a given machine, the input vector selection procedure is not needed. In this case all the input vectors appearing in the petri net are used to guide the search.

Finally, the input vector guidance mechanism, like the heuristic cost value guiding mechanism, is intended to be machine invariant. However, it may be easy to find test sequences for some machines without using input guidance. It will be very difficult to detect this "easy" condition using the artificial intelligence method proposed in this work. An experienced user, on the other hand can recognize such types of machines. For this reason, a user may have the option of indicating to SCIRTSS whether he wants input vector guidance or not. In the next chapter we present case studies to show some machines that do not need input vector selection procedure.

4.4 Terminating the Petri Net

The question of terminating the petri net generation has been deferred till now because we want to explain how the heuristic function is calculated and how inputs are selected for guiding the search. Since we are concerned about the number of transitions that have been fired in the net for a given search state, it is essential that we include enough transitions in the petri net. For smaller circuits, then, given a set of goal nodes, the petri net must be expanded until the initial control state, usually control state one, is reached and the input places are all external places.

However, for more complex circuits, for example, the microprocessor circuit described in the following chapter, it is necessary to terminate the petri net generation to prevent having too many places and transitions. In SCIRTSS III, the problem graph generation is terminated based on the ease with which a node was satisfied in past searches. We use the same decision rules for terminating the generation of the petri net, with the following added:

1. Every control state at which a register transfer or conditional branch occurs must be expanded at least once. This implies that it its not necessary to expand a control state in the AHPL description in which only an unconditional branch to another control state occurs. This rule is due to the fact that if a particular control state, say cs.5 is used as a terminal place, then if the machine is in cs.4, cs.3, or cs.2, no transitions in the petri net can be inferred to have been fired. Hence the weighting function would not be able to differentiate between control state five and control state three, for example, and this is misleading to the search routine.

2. Where a register, RE, is loaded with primary inputs at a given control state and the place $\{RE:a_1a_2...a_n\}$ is associated with a transition whose transition time is 2 or less, this particular place must be expanded at least once. Since in weighting input vectors to be selected we gave a high priority with places whose transitions have small transition time, a register associated with such a transition should not be left to be randomly loaded!

When the two rules above are followed and still there are many more places to be expanded, all places associated with transition times bigger than a user specified value are marked terminal and not expanded.

CHAPTER V

CASE STUDIES AND RESULTS

The guidance mechanisms described in the previous chapters are supposed to be independent of any circuit description. A user would only have to prepare the parameters of his particular circuit and submit it as data to the routine. To test these concepts, four markedly different circuits with varying degrees of complexity were submitted to the test generation program. Faults that were considered difficult for SCIRTSS sensitization searches to reach were selected for detection.

For each fault, the d-algorithm routine found a set of goal nodes. The petri net was generated manually and submitted as data to routine GNPT whose listing appears in Appendix A. This routine sets up pointers to the various places and transitions in the petri net. The routine HEUSUB computes the heuristic value at each step of the sensitization search. This routine is also listed in Appendix A. The only cases where there were more than five input vectors were cases III and IV. For these cases, the input vectors were selected based on the criteria in chapter four and submitted as data to the main search routine. All four circuits have been previously used as test cases in the full SCIRTSS run at the University of Arizona. In the early tests, special guidance routines had to be written for each case (Ng (1974), Van Helsland (1974)). Huey (1975) used these circuits to test his general purpose guidance mechanism and we shall frequently refer to the results obtained using the petri net and those obtained using the problem reduction graph which was used in SCIRTSS III.

For each sensitization search, the goal node(s) and starting node are submitted to the main search routine as data. The routine expands each node and computes the heuristic value for the node. This heuristic value is compared with other nodes that are candidates for expansion. The node with the minimum heuristic value is picked as most promising and expanded. The search is either successful in which case it returns "SEARCH SUCCESSFUL" message together with the goal reached, or fails. In the latter case, there are two ways it can fail:

a) When the search routine runs out of nodes to expand, i.e., all nodes have been expanded without any new node being generated, it returns "MINIMUM HEURISTIC SEARCH FAILS."

b) If the search continues for more than a user specified limit (NSIM call limits) without finding a successful input sequence, it is terminated as an unsuccessful sensitization search. A limit of 1000 was set for the test run.

5.1 Case I: The Narrow Window Circuit

The first circuit to be used to test the guidance mechanism is a "narrow window" circuit where certain control states are hard to reach due to control branching conditions which are hard to satisfy. The only searches to fail detection in earlier SCIRTSS testing were those where reaching a goal node involved reaching a control state in one set when the initial state for the search was in the other. The AHPL description of this circuit is shown in Fig. 5.1 while Fig. 5.2 shows the control state diagram. There are two sets of control states: GA and GB. The fault requiring the most difficult sensitization search possible is the one associated with the branch logic from cs.11 to cs.1 if the machine is initially in cs.1. The fault to be sensitized is at the output of the logic which implements the branch condition:

$$\text{cs.11} \longrightarrow (\wedge/A)/(1)$$

```
MODULE:  NARROW WINDOW CIRCUIT
   MEMORY:  A[3];  B[3];  CNT[4]; Y[1]
   INPUTS:  X[3];  I1, I2
  OUTPUTS:  Z, B1, C

 1.    $A \leftarrow X;\ \ Y \leftarrow I1;\ \ CNT \leftarrow INC(CNT)$
       $\rightarrow (\overline{I2},\ I2)/(2,5)$

 2.    $B \leftarrow \omega^{3}/ADD(A,B);\ C \leftarrow \alpha^{1}/ADD(A,B)$
       $\rightarrow (\overline{Y},Y)/(3,4)$

 3.    $B \leftarrow \omega^{3}/ADD(A,B);\ C \leftarrow \alpha^{1}/ADD(A,B)$

 4.    $CNT \leftarrow INC(CNT)*I1$
       $\rightarrow (\ (CNT_1 \wedge CNT_2),\ (\overline{CNT_1 \wedge CNT_2})\ )/(8,1)$

 5.    $A \leftarrow \uparrow(A \wedge B)$
       $\rightarrow (\overline{I2},\ I2)/(6,7)$

 6.    $B \leftarrow \overline{\epsilon(3)};\ Z \leftarrow 1;$

 7.    $CNT \leftarrow INC(CNT)*Y$
       $\rightarrow (\ (CNT_1 \wedge CNT_2),(\overline{CNT_1 \wedge CNT_2})\ )\ /\ (8,1)$

 8.    $A \leftarrow B*I2;\ B \leftarrow A*I2$
       $\rightarrow (\overline{I1},I2)/(9,11)$

 9.    $A_3 \leftarrow \wedge/(A,B)*I1$

10.    $B \leftarrow \uparrow B;\ A \leftarrow \uparrow A$

11.    $B,A \leftarrow B_2,B_3,A,X_3$
       $\rightarrow (\ (\vee/A),\ (\overline{\vee/A})\ )/(8,1)$
```

Figure 5.1 AHPL Description of Case Study I

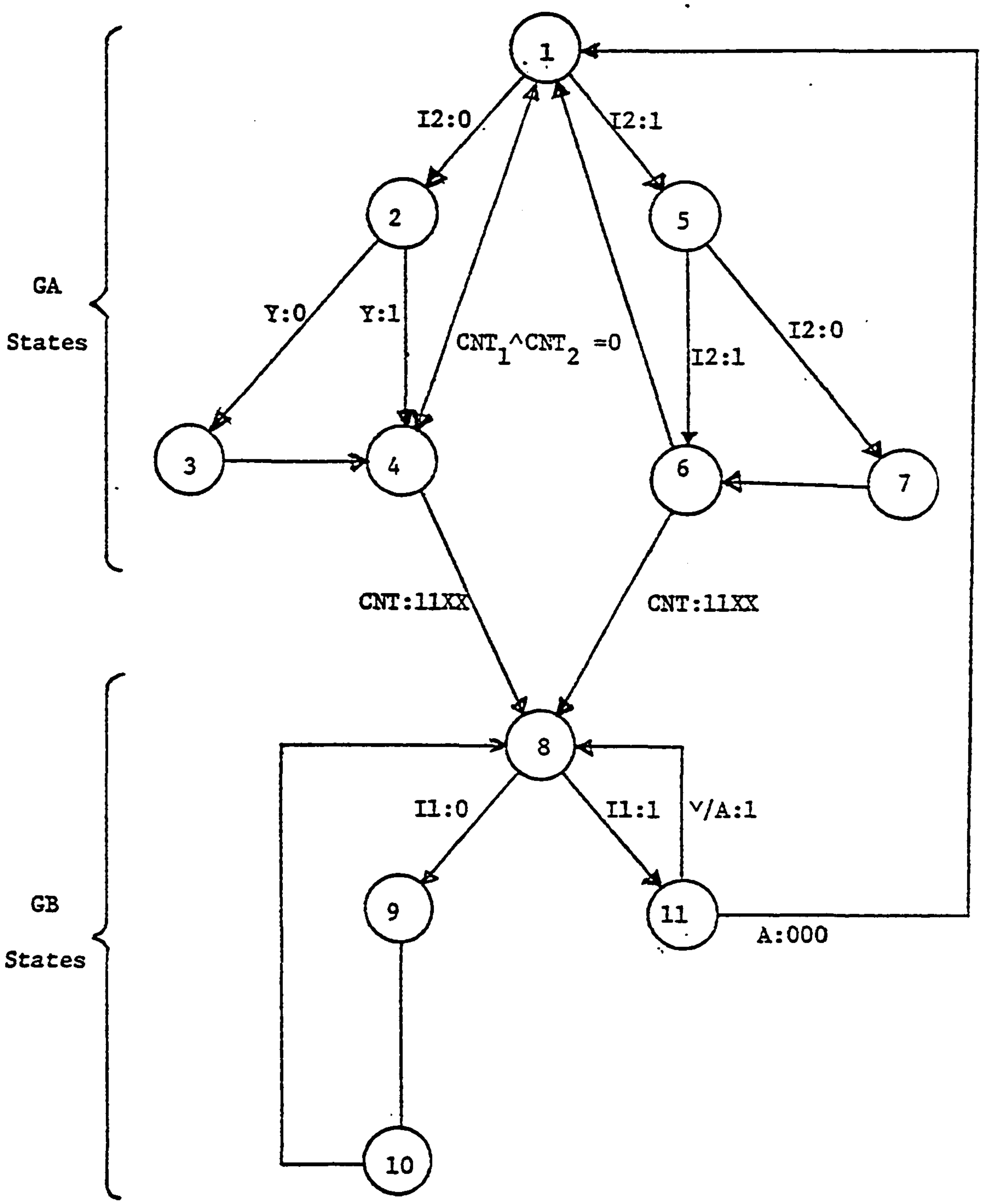


Figure 5.2 Control State Diagram of Narrow Window Circuit

This fault is the output of OR gate #90 stuck-at-one. To sensitize this fault, starting from control state one, the narrow window conditions for going from GA to GB must first be satisfied $\{$CNT:11XX$\}$, then the condition A:000 must be satisfied.

The d-algorithm returns one test vector which indicates that place A:000 and cs.11 must be satisfied for the fault to be sensitized. The petri net generated is shown in Table 5.1 and the place listing in Table 5.2. From the petri net we have only one input vector X:000 which was submitted to help in guiding the search. This input vector alone provided enough guidance to find an input sequence that is comparable to those found using heuristic function guidance. Two sets of tests were run on this circuit:

1) When the machine is in reset state, i.e., c.s.1 and state vector is 0000 0000 000, and

2) When the machine is in control state 1 and registers A and B contain the vectors $\{$111$\}$. In both cases the goal node is the same.

When the search starts from the reset state, the goal is not very difficult to reach although it is not trivial. The combined heuristic value-input vector guidance expands about 50% less nodes than using input vectors only. It may

Table 5.1 Petri Net Listing for Case I

Transition	Type	Output Place	Input Places	Immediate Descdt
T1	3	P1	2,3	2,3,4,5,6,7
T2	1	P2	7,8	8,9,10,14,24
T4	1	P2	8,9	8,9,10,24,11,12
T5	1	P2	5,6	
T6	1	P2	10	13
T7	2	P3	9	11,12
T8	1	P8	10	13
T9	1	P8	10	13
T10	1	P8	14	21
T11	2	P9	12,13	17,18,19
T12	2	P9	11,13	16,17
T13	2	P10	16	15
T14	1	P8	3,17,18	7
T15	2	P16	9	11,12
T16	6	P11	14,19	21
T17	4	P13		
T18	2	P12	7	22
T19	2	P12	15	20
T20	2	P15	7	22
T21	2	P14	19	23

Table 5.1 cont'd

Transition	Type	Output Place	Input Places	Immediate Descdt
T22	2	P7	6	
T23	2	P19	6	
T24	1	P8	15	20

Table 5.2 Place Listing for Case I

P	1	GOAL	P	11	CS.4
P	2	A:000	P	12	CS.7
P	3	CS.11	P	13	KNT:11XX
P	4	A:X00	P	14	CS.3
P	5	IX:000	P	15	CS.6
P	6	CS.1	P	16	CS.9
P	7	CS.5	P	17	B:X00
P	8	B:000	P	18	A:0XX
P	9	CS.10	P	19	CS.2
P	10	CS.10			

be startling at first to observe from Table 5.3(a) that the heuristic cost value alone expanded the same number of nodes as the combination of heuristic value and input vector guidance. This is expected since the register A contains {000} to start with and this is the same vector that is loaded by the input vector.

In the second test run, with registers A and B both containing the vector {111}, more nodes are expanded before reaching a goal. The input vector only guidance found a sequence whose length is three more than the combined input and heuristic cost value guidance. The results of this test are summarized in Table 5.3(b). With the heuristic value only, only 60 nodes were expanded and the length of the sequence found is 27. In this particular result, register A was loaded with {000} on the first expansion thus leading to the expansion of very few nodes. However, the length of sequence found is suboptimal. Case Study 1(b) is a good illustration of the fact that both heuristic cost value and input guidance are required to give an optimal sequence. SCIRTSS III ran the same test and expanded about 60% more nodes than the results reported here. In both cases, the length of the sequence found is about the same.

Table 5.3. Test Runs for Case I

	A	B	CNT	Y
(a) starting node:	000	000	0000	0

Type of Guidance	Length of Sequence Found	Total Nodes Searched
No guidance	none found	1000
Input vector only	33	170
Heuristic vector only (w = 75)	26	94
Heuristic value (w=75) and input vector	26	94

	A	B	CNT	Y
(b) starting node:	111	111	0000	0

Type of Guidance	Length of Sequence Found	Total Nodes Searched
No guidance	none found	1000
Input vector only	26	252
Heuristic value only (w = 75)	27	60
Heuristic value (w=50) and input vector	23	277
Heuristic value (w=200) and input vector	24	238

5.2 Case II: The Anti-Random Circuit

The second case study is an anti-random circuit. This circuit has the special feature of a chain of control states where each control state either loops to itself unless a counter has counted up to seven, or resets to an initial state if the control input RS is 1. The AHPL description is given in Fig. 5.3. The heuristic function computation will in this case be very much dependent on the count transition.

The fault to be sensitized is in the logic that implements the register transfer

$$Z \longleftarrow \wedge/B$$

in control state five. This fault is at the output of AND gate #66, stuck-at-zero. The d-algorithm returns a vector which indicates the machine must be driven into cs.5 and load register B with 11111111 to sensitize the fault. The petri net generated for this case is shown in Table 5.4. The inputs to transition t_1 which fires to fill the goal place are cs.5 and B:11111111. Needless to say, to load all one's into a register is not likely to happen by chance.

There are two input vectors in the petri net which were used as heuristic input vectors. These two input vectors were very effective in guiding the search. However, the heuristic function-input vector guidance provided an efficient

```
MODULE:  ANTI-RANDOM CIRCUIT
   MEMORY:  KNT[3]; B[8]
   INPUTS:  X[8]; RS
  OUTPUTS:  OUT[8]; Z

1.     KNT ← $\overline{\epsilon(3)}$
       → ($\overline{RS}$,RS)/(1,6)

2.     KNT ← INC(CNT)
       → ( ($\wedge$/KNT),($\overline{\wedge/KNT}\wedge\overline{RS}$),($\overline{\wedge/KNT}\wedge RS$) )/(3,1,2)

3.     KNT ← $\overline{\epsilon(3)}$; B ← X

4.     KNT ← INC(KNT)
       → ( ($\wedge$/KNT),($\overline{\wedge/KNT}\wedge\overline{RS}$),($\overline{\wedge/KNT}\wedge RS$) )/(5,1,4)

5.     Z ← $\wedge$/B; OUT ← $\overline{\epsilon(8)}$
       → 1

6.     KNT ← INC(KNT)
       → ( ($\wedge$/KNT),($\overline{\wedge/KNT}\wedge RS$),($\overline{\wedge/KNT}\wedge\overline{RS}$) )/(7,1,6)

7.     KNT ← $\overline{\epsilon(3)}$; B ← $\overline{X}$

8.     KNT ← INC(KNT)
       → ( ($\wedge$/KNT),($\wedge$/KNT$\wedge$RS),($\overline{\wedge/KNT}\wedge\overline{RS}$) )/(9,1,8)

9.     Z ← 0; OUT ← B
       → 1
```

Figure 5.3 AHPL Description of Case Study II

Table 5.4 Listing of Petri Net for Case II

Transition	Type	Output Place	Input Places	Immediate Descdt
T1	3	P1	2,3	2,3,4
T2	2	P2	4,5	5,6,7
T3	1	P3	6,7	9
T4	1	P3	8,9	8
T5	2	P4	6	9
T6	4	P5		
T7	4	P5		
T8	2	P9	5,10	6,7,10
T9	2	P6	5,11	6,7,11
T10	2	P10	12	
T11	2	P11	12	

Table 5.5 Place Listing for Case II

P	1	GOAL	P	7	IX1111111
P	2	CS.5	P	8	IX0000000
P	3	B11111111	P	9	CS.7
P	4	CS.4	P	10	CS.6
P	5	KNT:111	P	11	CS.2
P	6	CS.3	P	12	CS.1

guidance: only fifty nodes were expanded and the length of the sequence found is two less than when only input vector guidance is used. It is interesting to note that heuristic value only could not find any sequence. This is because of the nature of the goal: to randomly generate all 1's is not very easy. However, the combination of the two types of guidance expands 50% fewer nodes than the input vector only! A summary of the test run is given in Table 5.6.

Comparing the problem reduction graph method of SCIRTSS III, we note that 122 nodes were expanded by the heuristic-value input vector guidance to find a sequence of length 29.

Table 5.6. Summary of Test Runs for Case II

Type of Guidance	Length of Input Sequence	Total Nodes Searched
No guidance	none found	1000
Input vectors only	20	113
Heuristic value only (w = 75)	none found	1000
Heuristic value (w=50) and input vectors	18	50
Heuristic value only (w=100) and input vectors	18	53

5.3 Case III: Search-Sort Processor

Another circuit used to test SCIRTSS is a search-sort processor which includes a random access memory. The data word is only two bits in width since the width of data word does not present any problem in test generation. The instruction register is externally loaded when the machine is in control state 1. Fig. 5.4 shows the AHPL description.

The petri net generated for this case is shown in Table 5.7. From the petri net one can notice that places that are of register transfer type dominate the control states 3:1. Hence the heuristic function computed would be very much controlled by register contents.

A careful look at the machine description shows that all control branch conditions are determined by the contents of the instruction register which is in turn dependent upon the input vectors applied at control state one. This makes the input vector selection very crucial and in fact, is the first test to the selection procedure of Chapter Four.

The fault being sensitized is at the output of the memory cell M_1^4 stuck-at-zero. The d-algorithm returns one test vector which signifies that the machine must be moved into control state four, and registers AR, IR loaded with vectors 100, XX1 respectively while X1 must be written into the fourth memory location to sensitize the fault.

```
MODULE:  SEARCH-SORT PROCESSOR
   MEMORY:  M[8;3]; AR[3]; IR[3]; MD[3]; AC[3]
   INPUTS:  A[3]; IN[3]; X[3]
  OUTPUTS:  AC[3]; out; accept; input

1.    AR ← A; IR ← IN; accept ← 1

2.    → $(IR_0, (\overline{IR}_0 \wedge \overline{IR_1}), (\overline{IR}_0 \wedge IR_1 \wedge \overline{IR_2}), (\overline{IR}_0 \wedge IR_1 \wedge IR_2)\ )/(3,4,5,6)$

3.    AC ← $\overline{AC} * (IR_1 \wedge IR_2) \vee (AC \wedge MD) * (IR_1 \wedge \overline{IR}_2) \vee MD * (\overline{IR}_1 \wedge IR_2) \vee X * (\overline{IR}_1 \wedge \overline{IR}_2)$
      MD ← $AC * (\overline{IR}_1 \wedge IR_2)$; input ← $\overline{IR} \wedge \overline{IR}_2$
      → 1

4.    MD ← $\uparrow MD * \overline{IR}_2 \vee IR_2 * BUSFN(M; DCD(AR)\ )$
      → 1

5.    out ← 1
      → 1

6.    M*DCD(MA) ← MD
      → 1
```

Figure 5.4 AHPL Description of Case Study III

Table 5.7 Petri Net Listing for Case III

Transition	Type	Output Place	Input Places	Immediate Descdt
T1	3	P1	2,3,4,5	2,3,4,15
T2	2	P2	6,7	5,6
T3	1	P3	8,9	
T4	1	P4	10,11	7,9
T5	1	P14	13,19	10
T6	1	P7	8,22	
T7	2	P11	6,12	8
T8	1	P12	8,16	
T9	1	P10	13,14,15	5,10,11,14
T10	2	P13	6,17	13
T11	1	P14	13,20,21	10,12
T12	1	P21	8,18	
T13	1	P17	8,18	
T14	1	P15	8,22	
T15	1	P5	8,23	

Table 5.8 Place Listing for Case III

P	1	GOAL
P	2	CS.4
P	3	AR:100
P	4	M4:X1
P	5	RIR:XX1
P	6	CS.2
P	7	RIR:00X
P	8	CS.1
P	9	IA:100
P	10	MD:X1
P	11	CS.6
P	12	RIR:01X
P	13	CS.3
P	14	AC:X1
P	15	RIR:X01
P	16	IN:011
P	17	RIR:1XX
P	18	IN:200
P	19	AC:X0
P	20	IX:X1
P	21	RIR:X00
P	22	IN:001
P	23	IN:XX1

The Control Branch Vectors obtained for this circuit are:

B_3 = IN:1XX

B_4 = IN:00X

B_5 = IN:010

B_6 = IN:011

Only control state five does not appear in the petri net and hence the G Common Transfer Vectors are:

$G_1(3)$ = IN:1XX

$G_2(4)$ = IN:00X

$G_3(6)$ = IN:011

For the input vectors associated with register transfers, we have:

A:100

IN:XX1

IN:001

Both A:100 and IN:XX1 received high weighting values and were selected according to our rules. Hence we have the vectors:

IN:1XX; IN:00X; IN:011

selected from the Control Branch Input Selection procedure and

A:100; IN:XX1

selected from the Register Transfer Input selection procedure. The input vector IN:XX1 from the second selection process covers the vector IN:00X and so IN:00X is dropped from the list. Hence for application at control state one we have the four vectors:

A:100; IN:XX1; IN:011 and IN:1XX

selected to guide the search.

To make the search as difficult as possible the starting node was chosen as:

CS	AR	IR	AC	MD	M^4
1	001	110	00	01	00

This circuit responds very well to guidance: only 36 nodes were expanded to reach the goal! The whole state space search is shown in Table 5.9 while the various runs are summarized in Table 5.10.

Comparing our results with SCIRTSS III, we note that in SCIRTSS III, 69 nodes were expanded to reach the goal and the length of the sequence found is 17. The input vector only

Table 5.9 State Space Search for Case III

	ITER.	NODE	LEVEL	COST VALUE	C.S.	PRED.NODE	INPUT VECTOR	STATE VECTOR
	0	1	0	106	1	0	22222222001	11001000000000000000000
	1	2	1	81	2	1	10000100100	00101000000000000000000
	2	3	1	94	2	1	01001100010	01101000000000000000000
	3	4	1	107	2	1	10110010101	10001000000000000000000
	4	5	2	68	4	2	01001111100	00101000000000000000000
	5	6	3	176	1	5	00100000100	00100000000000000000000
	6	7	2	82	6	3	10111000010	01101000000000000000000
	7	8	3	96	1	7	01100011010	01101000000010000000000
	8	9	4	84	2	8	10000101100	00101000000010000000000
	9	10	4	97	2	8	11001101110	01101000000010000000000
	10	11	4	110	2	8	00110001001	10001000000010000000000
	11	12	5	71	4	9	11110000100	00101000000010000000000
	12	13	6	179	1	12	01011101100	00100000000010000000000
	13	14	5	85	6	10	10101111110	01101000000010000000000
	14	15	6	99	1	14	10110101110	01101000000010000000100
	15	16	7	87	2	15	10001110100	01101000000010000000100
	16	17	7	103	2	15	10101101101	01101000000010000000100
	17	18	7	113	2	15	10110011101	10001000000010000000100
	18	19	8	74	6	16	10001100100	01101000000010000000100
	19	20	9	89	1	19	10101010100	01101000000010001000100
	20	21	10	90	2	20	10001110100	01101000000010001000100
	21	22	10	103	2	20	11101111111	01101000000010001000100
	22	23	10	116	2	20	01110011011	10001000000010001000100
	23	24	11	77	6	21	00100110100	01101000000010001000100
DLET	24	25	12	92	1	24	11001000100	01101000000010001000100
	25	25	8	88	6	17	00100011101	01101000000010000000100
	26	26	9	102	1	25	00000101101	01101000000010000010100
	27	27	10	90	2	26	10001100100	01101000000010000010100
	28	28	10	103	2	26	00001111000	01101000000010000010100
	29	29	10	116	2	26	11110001111	10001000000010000010100
	30	30	11	77	6	27	11100101100	01101000000010000010100
	31	31	12	92	1	30	01010010100	01101000000010001010100
	32	32	13	93	2	31	10000111100	00101000000010001010100
	33	33	13	106	2	31	01101110011	01101000000010001010100
	34	34	13	119	2	31	00010010000	10001000000010001010100
	35	35	14	80	4	32	00100010100	00101000000010001010100

```
                    SEARCH SUCCESSFUL


        GOAL REACHED      10022122222222222221222222
        FINAL NODE        10000101000000010001010100


     36 NSIM CALLS
```

guidance in both cases expanded about the same number of nodes but our results show a sequence of length 15 while SCIRTSS III found a sequence of length 629.

This case study has again demonstrated that both heuristic cost value and input vector guidance are necessary to produce an efficient search and obtain a sequence of reasonable length.

Table 5.10. Summary of Test Runs for Case III

Type of Guidance	Length of Sequence Found	Total Nodes Searched
No guidance	none found	search failed
Input vectors only	15	577
Heuristic value only (w = 200)	none found	search failed
Input vectors and heuristic value (w = 200)	14	36

5.4 Case IV: Four-Bit Expandable Microprocessor

The last case study is a four-bit microprocessor slice. As far as automatic test generation is concerned, the data word poses no difficulty: thus there would be very little difference if, say an eight-bit microprocessor were being tested. The preceeding three cases were designed with the aim of stalling the test generation and guidance principle. This case is intended to test the usefulness of the guidance principle on a real world problem. Besides, the control description is far more complex than the previous cases. The arithmetic unit includes a full adder and other more sophisticated combinational logic functions. Figure 5.5 gives the AHPL description and the block diagram appears in Figure 5.6.

The fault selected for the sensitization search was at the carry out bit of the program counter slice (OR gate #172) stuck-at-zero. The d-algorithm returned three vectors that could sensitize the fault:

{CS.10, PC:1111, IR:XX1X}
{CS.14, PC:1111}
{CS.19, PC:1111}

The petri net generated is shown in Table 5.11. As expected, the petri net is large, having fifty-seven transitions and 57 places. Almost all places that were of register transfer type were expanded except IR:1XXX and IR:X001 which were left unexpanded because of the ease with which they have been satisfied in the past.

The main source of difficulty in performing sensitization searches on this circuit is that often many instructions

```
MODULE:  FOUR-BIT MICROPROCESSOR
   MEMORY:  UR[4]; AC[4]; IR[4]; PC[4]
   INPUTS:  DN[4];ICS[4]; linki, slave, ready
  OUTPUTS:  DO[4]; IOSR; linko

1.    UR ← PC

2.    DO ← UR; IOSR = 1,0,1
      → ($\overline{\text{ready}}$, ready)/(2,3)

3.    IR ←ICS; UR ← DN; IOSR ← 0,1,0
      → ($\overline{\text{ready}}$, ready)/(3,4)

4.    → ( ($IR_1 V(\overline{IR}_2 \Lambda \overline{IR}_3 \Lambda IR_4)$ ),($\overline{IR}_1 \Lambda IR_2 \Lambda IR_3 \Lambda \overline{IR}_4$),($IR_1 \Lambda IR_2 \Lambda IR_3 \Lambda IR_4$),
      ( $\overline{(\overline{IR}_1 \Lambda \overline{IR}_2 \Lambda \overline{IR}_3 \Lambda IR_4)} \Lambda \overline{(IR_2 \Lambda IR_3)}$ )/(5,16,18,15)

5.    DO ← UR*slave; IOSR ← (1,0,1)*slave
      → ($\overline{\text{ready}}$, (ready $\Lambda$( ($\overline{IR}_3 \Lambda IR_4$)V($\overline{IR}_2 \Lambda IR_3$) ) ),
      (ready$\Lambda \overline{IR}_2 \Lambda \overline{IR}_3 \Lambda \overline{IR}_4$),(ready$\Lambda IR_2 \Lambda \overline{IR}_4$),
      (ready$\Lambda IR_2 \Lambda IR_3 \Lambda IR_4$) )/(5,6,8,10,12)

6.    UR ← DN; IOSR ← 0,1,0
      → (ready, ready)/(6,7)

7.    DO ← UR; IOSR ← 1,0,1
      → ($\overline{\text{ready}}$, (ready$\Lambda IR_1 \Lambda \overline{IR}_2$),(ready$\Lambda \overline{IR}_1 \Lambda \overline{IR}_2$), $IR_2$)/(7,8,12,10)

8.    UR ← DN; IOSR ← 0,1,0
      → ($\overline{\text{ready}}$, ready)/(8,9)
```

Figure 5.5 AHPL Description of Case Study IV

```
9.   $AC \leftarrow (UR * \overline{IR_3}) \vee (ADD(AC,UR) * (IR_3 \wedge \overline{IR_4}))$
     $\vee (NAND(AC,UR) * (IR_3 \wedge IR_4))$
     $lnko \leftarrow (CO \wedge slave)*(IR_3 \wedge \overline{IR_4})$; CO=Carryout of ADD(AC,UR)
     $\rightarrow (14)$

10.  $UR \leftarrow (AC * \overline{IR_3}) \vee (INC(PC) * IR_3)$
     $lnko \leftarrow (\overline{slave} \vee CPO) * IR_3$; CPO=Carryout of INC(PC)
     $\rightarrow (11)$

11.  $DO \leftarrow UR$;   $IOSR \leftarrow 1,1,0$
     $\rightarrow (\overline{ready}, (ready \wedge \overline{IR_3}), (ready \wedge \overline{IR_3}))/(11,14,12)$

12.  $UR \leftarrow DN$;   $IOSR \leftarrow 0,0,1$
     $\rightarrow (\overline{ready}, ready)/(12,13)$

13.  $PC \leftarrow UR$
     $\rightarrow (\overline{IR_4}, IR_4)/(14,1)$

14.  $PC \leftarrow INC(PC)$;   $lnko \leftarrow \overline{slave}\ CPO$; CPO=Carryout of INC(PC)
     $\rightarrow (1)$

15.  $HALT \leftarrow 1$;   $IOSR \leftarrow (1,0,0)*(\overline{IR_2} \wedge \overline{IR_3} \wedge \overline{IR_4})$
     $AC \leftarrow \overline{\varepsilon(4)}*(\overline{IR_2} \wedge \overline{IR_3} \wedge \overline{IR_4})$
     $AC \leftarrow \uparrow(AC,lnki)*(\overline{IR_2} \wedge IR_3 \wedge IR_4)$
     $lnko \leftarrow AC *(\overline{IR_2} \wedge IR_3 \wedge IR_4)$
     $lnko \leftarrow ((AC \wedge \overline{slave}) \vee (lnki \wedge slave))*(IR_2 \wedge \overline{IR_3} \wedge \overline{IR_4})$
     $lnko \leftarrow (v(AC) \vee lnki) * (IR_2 \wedge \overline{IR_3} \wedge IR_4)$
```

(Continued)

```
15.   (Continued)
      $\rightarrow ( (\overline{IR}_2 \wedge \overline{IR}_3 \wedge \overline{IR}_4), (IR_2 \wedge \overline{IR}_3 \wedge \overline{IR}_4 \wedge \text{lnki}),$
        $(\overline{IR}_2 \wedge IR_3), (IR_2 \wedge \overline{IR}_3 \wedge \overline{IR}_4 \wedge (\text{lnki}),$
        $(IR_2 \wedge \overline{IR}_3 \wedge IR_4 \wedge (\overline{V/ACV\ \text{lnki})}\ ),$
        $(IR_2 \wedge \overline{IR}_3 \wedge IR_4 \wedge (V/ACV\ \text{lnki})\ )/(20,19,14,14,19,14).$

16.   $UR \leftarrow DN;\ IOSR \leftarrow 0,1,0$
      $\rightarrow (\overline{\text{ready}}, \text{ready})/(16,17)$

17.   $AC \leftarrow UR$
      $\rightarrow (14)$

18.   $DO \leftarrow UR;\ IOSR \leftarrow 1,1,1$
      $\rightarrow (\overline{\text{ready}},\ \text{ready})/(18,14)$

19.   $PC \leftarrow INC(PC);\ \text{lnko} \leftarrow \overline{\text{slave}} \wedge CPO;$   $CPO$=Carryout of $INC(PC)$
      $\rightarrow (14)$

20.   $\rightarrow (\text{lnki}, \overline{\text{lnki}})/(3,20)$
```

Figure 5.5 cont'd

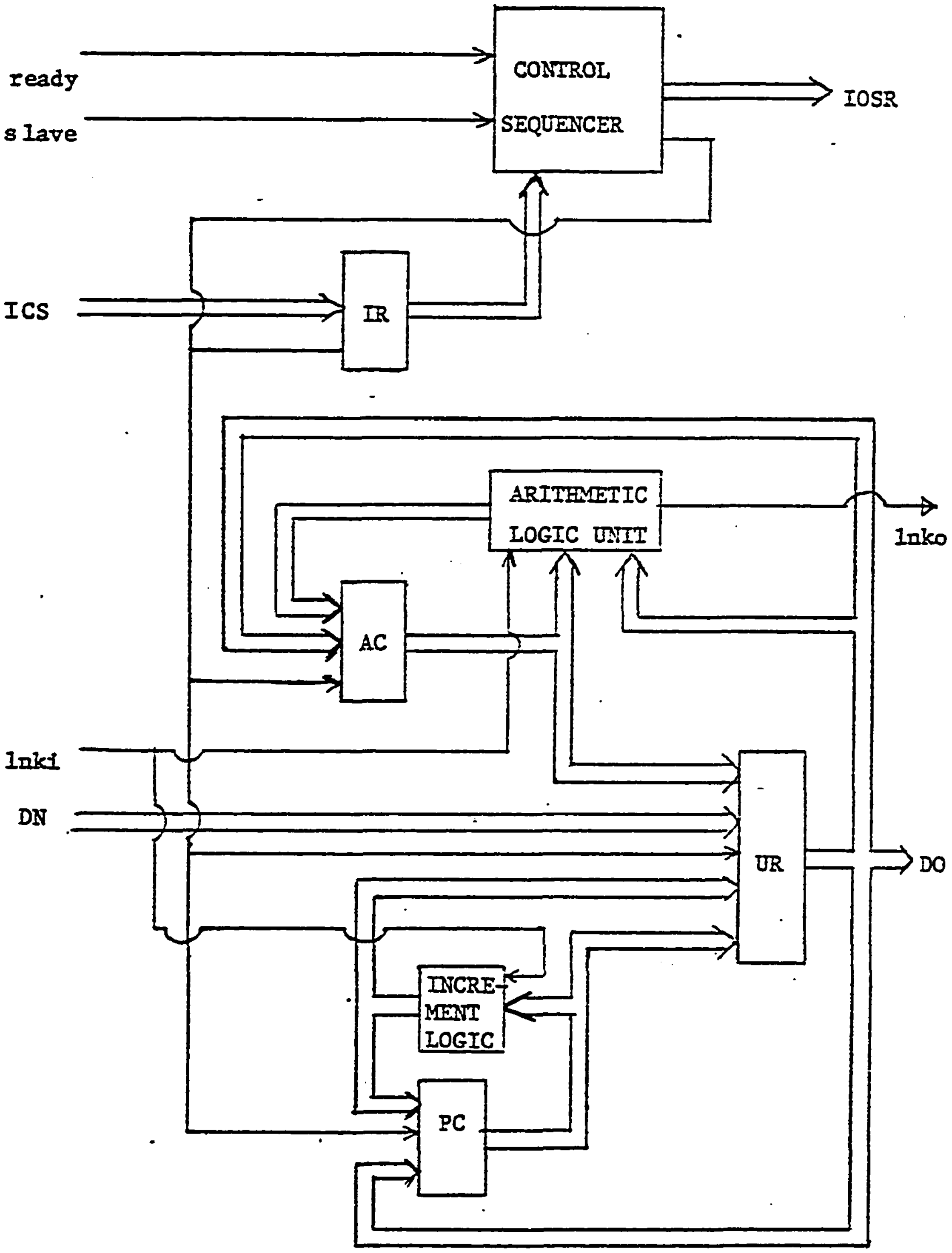


Figure 5.6 Block Diagram of Four-bit Microprocessor

Table 5.11 Petri Net Listing for Case IV

Transition	Type	Output Place	Input Places	Immediate Descdt
T1	3	P1	2,3,4	21,22,4,6
T2	3	P1	3,5	4,6,7,8,9,10
T3	3	P1	3,6	4,6,12,13
T4	4	P3	5,6	7,8,9,10,12,13
T5	4	P22	5,6	7,8,9,10,12,13
T6	1	P3	7,8	39,40,41,42,43,44,45
T7	2	P5	11	26
T8	2	P5	12,13	37,53
T9	2	P5	7,13	37,45
T10	6	P5	6,9,10	12,13,32,34
T11	2	P12	2	21,22
T12	2	P6	15,16	14,15,16,17
T13	2	P6	16,17,18	15,16,17,18
T14	1	P15	19,20	56
T15	2	P16	24,25	19,55
T16	2	P16	24,26	20,55
T17	2	P16	24,27	23,55
T18	1	P17	20,52	56
T19	1	P25	20,28	56
T20	1	P26	20,29	56

Table 5.11 cont'd

Transition	Type	Output Place	Input Places	Immediate Descdt
T21	2	P2	15,30	14,53,54
T22	2	P2	15,32	14,24
T23	1	P27	20,21	56
T24	2	P32	34	27,28
T25	1	P14	20,33	56
T26	2	P11	41	38,52
T27	2	P34	30,35	29,53,54
T28	2	P34	30,36	30,53,54
T29	1	P35	20,37	
T30	1	P36	20,21	56
T31	1	P4	20,38	56
T32	2	P10	24,39	33
T33	1	P39	20,50	56
T34	2	P9	44	35,56
T35	2	P44	24,42	36,55
T36	1	P42	20,46	56
T37	1	P13	20,40	56
T38	2	P41	30,14	25,53,54
T39	1	P8	34,51	56
T40	1	P8	2,22,45	5,21,22
T41	1	P8	47,51	46,47,49
T42	1	P8	41,51	38,52

Table 5.11 cont'd

Transition	Type	Output Place	Input Places	Immediate Descdt
T43	1	P8	44,51	35
T44	1	P8	2,13,43	21,22,37
T45	2	P7	47,0	46,47,49
T46	2	P47	30,49	50,53,54
T47	2	P47	32,48	48,24
T48	1	P48	20,31	56
T49	2	P47	12,49	11,50
T50	1	P49	20,23	56
T51	1	P45	20,38	56
T52	2	P41	14,32	24,25
T53	2	P30	24,55	55
T54	2	P30	24,56	55
T55	2	P24	20	56
T56	6	P20	53,54	57
T57	2	P53	57	

Table 5.11 cont'd. Place Listing for Case IV

P	1	GOAL
P	2	CS.10
P	3	KPC:1111
P	4	RIR:2212
P	5	CS.14
P	6	CS.19
P	7	CS.13
P	8	UR:1111
P	9	CS.17
P	10	CS.18
P	11	CS.9
P	12	CS.11
P	13	RIR:XX00
P	14	RIR:1000
P	15	RIR:X100
P	16	CS.15
P	17	RIR:X101
P	18	AC:0000
P	19	ICS:X100
P	20	CS.3
P	21	ICS:0101
P	22	KPC:1110
P	23	ICS:X111
P	24	CS.4
P	25	RIR:0000
P	26	RIR:001X
P	27	RIR:010X
P	28	ICS:0000
P	29	ICS:001X
P	30	CS.5
P	31	ICS:00XX
P	32	CS.7
P	33	ICS:1000
P	34	CS.6
P	35	RIR:X01X
P	36	RIR:XX01
P	37	ICS:X01X
P	38	ICS:XX1X
P	39	RIR:0111
P	40	ICS:XX00
P	41	CS.8
P	42	RIR:0110
P	43	AC:1111
P	44	CS.16

Table 5.11 cont'd

P	45	RIR:XX1X
P	46	ICS:0110
P	47	CS.12
P	48	RIR:00XX
P	49	RIR:X111
P	50	ICS:0111
P	51	DN:1111
P	52	ICS:X101
P	53	CS.2
P	54	CS.20
P	55	RIR:1XXX
P	56	RIR:X001
P	57	CS.1

must be executed to cause the necessary data vector manipulations for reaching a goal node. If the instruction register, IR, is not loaded properly at control state three, the next 5 to 10 control steps do not offer any opportunity to modify the contents of IR nor does the machine return to CS.3. Also, a great many of the control branches depend entirely on the contents of IR.

To reach our goal, place {PC:1111} must be filled with a token while our initial state is PC:0000. To avoid many cycles of incrementing the program counter, a jump type instruction must be executed. This is either the instruction {IR:1110} or {IR:0110}.

Needless to say, the input vector selection is very crucial in reaching our goal. The instruction register IR is loaded with external input at CS.3. There are over sixteen input vectors associated with CS.3 from which four or five must be selected.

The Control Branch Input Vectors of this microprocessor were used as an example in Chapter Four, Section 4.2.1. From that procedure three vectors were selected:

ics:101X; ics:1100; ics:0110

It should be noted that the Control Branch·Input Vector selection process is invariant with respect to the fault; for that matter, there was no way of knowing that ics:1110 & ics:0110 are critical! However, with the basic philosophy of selecting the input vectors such that as far as it is possible all important control states are visited during the search, we have been able to select the "best" input vectors. From the petri net for this circuit, it was detected that the machine does not have to "wait" in control states 3, 6, 7, 8, 12, 16, 18 and 20 during the search. As discussed in Section 4.2.2 the register transfers that take place in these control states do not need to be repeated when the control signal "ready" is low.

The results obtained from this machine are fantastic! With the information on the control signals that can control branching from one control state to another, we performed two separate tests:

a) the control branching derived directly from the AHPL description

b) control branching selected according to the information from the petri net.

The results are listed in Tables 5.12(a) and 5.12(b). In SCIRTSS III, the control branching functions were derived directly from the AHPL description. This corresponds to the results of Table 5.12(a). Even in this case, 508 nodes were expanded to obtain a sequence of length 15. The best result reported in Table 5.12(a) expanded 50% less nodes to obtain a sequence of length 18.

We have included the results of Table 5.12(a) only for the sake of meaningful comparison with the results of SCIRTSS III. The results of Table 5.12(b) indicate the effectiveness of the petri net in guiding the search in complex circuits with very difficult goals. Using input vectors only, a sequence of length 16 was found and only 91 nodes were expanded! However, both the heuristic cost value and input vectors produced a sequence of length 20 and expanded only 47 nodes! Although the input sequence is suboptimal, the search is almost 50% more efficient than using only input vectors and thus is a very good result.

The visual representation of Figure 5.8 is given to show the disparity between the results of the search in all four circuits when no guidance is used and when there is guidance. The best results of SCIRTSS III, the problem reduction graph model and the petri net model are also indicated.

As a natural consequence of the comparison of the results obtained in this work with the results of the problem reduction graph approach, one may ask, "which is faster?"

TEST RUNS FOR CASE IV

Table 5.12(a)

Type of Guidance	Length of Sequence Found	No. of Nodes Expanded
No guidance	none found	1000
Input vectors only	none found	1000
Heuristic value only	none found	1000
Heuristic value (w = 75) and input vectors	18	287
Heuristic value (w = 100) and input vectors	18	230

Table 5.12(b)

Type of Guidance	Length of Sequence Found	No. of Nodes Expanded
No guidance	none found	1000
Input vector only	16	91
Heuristic value only	none found	search failed
Heuristic value (w = 100) and input vectors	20	47

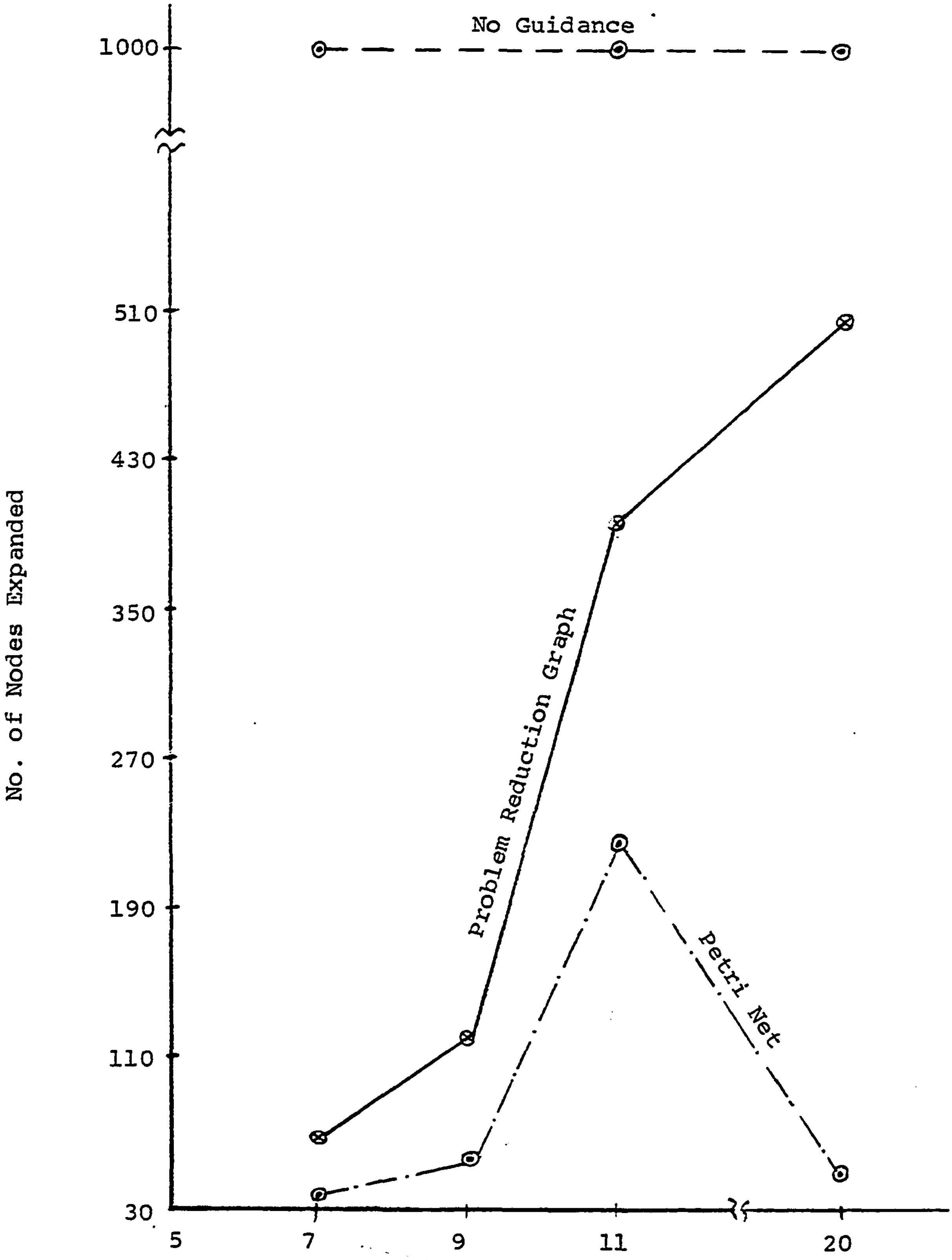


Fig. 5.8 Circuit Complexity (No. of Control States)

This is a difficult question to answer since SCIRTSS III does not report the amount of computer time taken to perform the search. The number of nodes expanded is by far a better comparison; however, we can refer to an informal computer output of SCIRTSS III that expands 300 nodes in 48 seconds. Using the same circuit, our program expanded 202 nodes in 29 seconds. On the other hand, the input vector selection process of Chapter Four is much more complex than that of the Problem Reduction Graph. Again since this is done only once for each search (if needed) the far fewer number of nodes expanded more than offsets the complexity of the selection procedure.

CHAPTER VI

SUMMARY AND CONCLUSION

6.1 Summary

Guidance for sensitization searches enable these searches to reach their desired goals after expanding relatively fewer nodes than if the searches are not guided at all; in most cases these unguided searches terminate abruptly.

A petri net model is presented that models the register transfers and change of control states in a sequential machine described in a Computer Hardware Description Language (CHDL). For each sensitization search, a new petri net is generated based on the goal node(s) and the CHDL circuit description. Portions of this process are completely independent of the fault.

Each set of goal nodes forms input places to a transition which if fired implies the fault is sensitized and the search is successful. Only one control state appears as input to each of these transitions. The remaining portions of the petri net are generated from these input places.

For each machine state encountered during the state space search, a marking or state vector is derived from the petri net, using the general state equations of a petri net. Based on the marking vector, a heuristic cost value is computed which measures essentially the effect of reaching one machine state on the transitions in the petri net. The direction of the search is determined by these heuristic cost values: a node with the minimum heuristic cost value is selected as most promising and is expanded in the state space search.

The petri net also contains information about input vectors that are associated with each control state. The most important of these input vectors are selected for inclusion in the input vector table that guides the search. For the purpose of selection, the input vectors are classified into two categories: those input vectors that are responsible for control state branching and those that are used only for register transfer. The input vectors that cause the sequential machine to branch to the most number of control states are selected from the first category for inclusion in the table while the register transfer input vectors are weighted, using information from the petri net. The input vectors receiving the highest weight from this weighting process are selected and, together with those selected from the control branch category, form the set of input vectors that provide input vector guidance. Again, portions of the input vector selection process are independent of the fault.

Although AHPL was used in this research, the results are applicable to any Computer Hardware Description Language that has the same structure as AHPL. That is, the expressions in that particular CHDL must be classifiable as:

1. Conditional Register Transfer expression
2. Unconditional Register Transfer expression
3. Conditional Control Branch Expression

or 4. Unconditional Control Branch Expression.

Four very difficult circuits were used as case studies to test the proposed guidance mechanism. The sensitization search goal for each of these circuits was selected to be as difficult as possible. In each case, the proposed guidance mechanism provides an improved performance in the sensitization search when compared with the guidance methods of the problem reduction graph of SCIRTSS III.

6.2 Limitations and Further Work

The effort to provide an automatic test sequence generation sequence has resulted in a complicated test generation system. For each sensitization search a new petri net has to be generated; although portions of this process simply involve linking subnets yet this can be time consuming. Further, where there are many input vectors, an input selection procedure must be evoked.

Although Computer Hardware Description Languages are becoming increasingly popular as design tools, many machine designs do not use them. This limits the scope of application of this work.

The generation of a new petri net for each fault can be avoided if we can have a petri net model of the machine itself such that each fault has a "unique impact" on the petri net. From this "unique impact" we can derive a heuristic cost value and perhaps be able to choose input vectors to guide the search. This research has not been able to produce such a petri net model of the machine; we had to generate the petri net starting from the goal nodes. This area can be investigated further.

The Problem Reduction Graph relies heavily on past SCIRTSS runs to obtain statistical information both for computing the heuristic value and terminating the graph generation. The petri net relies on statistical data from previous runs only for terminating the petri net generation. It would be desirable to get rid of relying on statistical data from previous runs completely. Maybe if we can produce the "universal" petri net mentioned in the previous paragraph then the problem would disappear. If not, a user specified optimum maximum transition time must be given for terminating the petri net generation. This optimum number is not yet known.

6.3 Conclusion

The four case studies of Chapter Five are complex sequential circuits with diverse characteristics and were selected with the aim of examining potential weaknesses in the approach. Moreover, the initial states were selected to be a maximum distance from the goals. Based on these four tests, we conclude that for sequential machines with very different characteristics, the guidance provided for sensitization searches using the petri net derived from the Computer Hardware Design Language description of the machine, is significantly more efficient. Sensitization searching in SCIRTSS is thus less likely to encounter node limit termination. Petri nets have been used in various areas of computer science to study the interconnection properties of systems. In our approach we have diverged from the normal use of petri nets when applicable; the idea of using these nets to analyze a Computer Hardware Description Language with the aim of guiding a state space search is novel and has proven to be remarkably effective.

APPENDIX A.1

SCIRTSS SEARCH ROUTINES

```
CT
C  PROGRAM TO PERFORM SCIRTSS IV SEARCH
C  K. E. TORKU  DECEMBER  7TH 1978
C
.C
C  CONTROL STATE BRANCHING FUNCTION TABLES.
C     NRSCS(I)        NUMBER OF SUCCESSOR C.S. FOR C.S. I.
C     KSSC(I,J)       J-TH SUCCESSOR C.S. TO C.S. I.
C     NRTRMS(I,J)     NUMBER OF TERMS IN BRANCHING FN. FROM C.S. I TO
C         THE J-TH SUCCESSOR.
C    VECTORS TO REPRESENT THE TERMS OF THE CONTROL STATE BRANCHING FNS.
C    EACH VECTOR (I,J,K) IS THE I-TH TERM OF BR. FN. FROM C.S. J TO
C          THE K-TH SUCCESSOR.
C    BIT I =0 FOR DON'T CARE OR ZERO ON INPUT OR FF I.
C          =1 FOR VALUE REQUIRED OR 1 FOR INPUT OR FF I.
C     MFIP(I,J,K)     VECTOR OF REQUIRED INPUTS
C     MFIV(I,J,K)     VALUES REQUIRED FOR INPUTS GIVEN BY MFIP
C     MFSP(I,J,K)     VECTOR OF REQUIRED FF'S FOR TERM.
C     MFSV(I,J,K)     VALUES REQUIRED FOR FF'S INDICATED BY MFSP.
C  PETRI NET ARRAYS
C  KTYP(N)   NTH TRASITION TYPE
C  KPNAM(I)  ITH PLACE  NAME
C  LNK(I)    INDEX TO VECTOR OF PLACES
C  M(J)      MARKING VECTOR
C  KR(I)     VECTOR OF FIRED TRANSITIONS
C  NIPL(N)   NO OF INPUT PLACES TO TRANSITION  N
C  IPLAD(N)  POINTER TO TO SET OF INPUT PLACES TO  TRANSITION N
C  KTRIN(N)  LIST OF INPUT PLACES TO TRANSITION N
C  KOPL(N)   OUTPUT PLACE OF TRANSITION N
C  KTRT(NN   TRANSITION TIME OF TRANSITION N
C  IDESP(N)  POINTER TO IMMEDIATE DESCENDANTS OF TRANSITION N
C  IMDCT(N)  SET OF IMMEDIATE DESENDANTS OF TR. N
C  KOTR(I)   POINTER TO SET OF TRANSITIONS TO WHICH
C            PLACE I IS OUTPUT
C  LOSTR(I)  LIST OF TRANSITIONS TO WHICH  OUTPUT
C            PLACES POINT
C  NP        NO OF PLACES IN THE PETRI NET
C  NT        NO OF TRANSITIONS IN THE NET
C  NTPL      NO OF TERMINAL PLACSE
C     MGRP(I)         I-TH FF VECTOR (VALUES PRESENT) FOR GRAPH.
C     MGRV(I)         I-TH FF VECTOR (VALUES WHERE REQUIRED) FOR GRAPH.
C     NCSIV(I)        NUMBER OF INPUT VECTORS TO BE APPLIED AT C.S. I.
C      MCSIP(I,J)     J-TH INPUT VECTOR TO BE APPLIED AT C.S. I.
C      MCSIV(I,J)
C  SEARCH SPACE.  I-TH ELEMENT OF EACH ARRAY IS FOR I-TH NODE.
C     MGOALP(I)       I-TH GOAL NODE. FF 'S WHERE VALUES ARE REQUIRED.
C     MGOALV(I)       I-TH GOAL NODE. REQUIRED FF VALUES.
C     NODECS(I)       MACHINE CONTROL STAE AT NODE I.
C     NODIP(I)        FULL INPUT VECTOR APPLIED TO PREDECESSOR TO REACH I
C     NSTATE(I)       FULL FLIPFLOP VALUE VECTOR FOR CURRENT (I-TH) NODE.
C     NPRED(I)        INDEX TO THE PREDECESSOR OF NODE I.
C     NODLEV(I)       LEVEL OF THE SEARCH GRAPH FOR NODE I.
```

```
            C      NODEH(I)         HEURISTIC WEIGHT COMPUTED FOR NODE I.
            C  COMMUNICATION VECTORS FOR PSDNSM ROUTINE.
            C      KTRMI(I)         0=0, 1=1, 2=X REQUIRED VALUE FOR INPUT I.
            C      KTRMF(I)         0=0, 1=1, 2=X REQUIRED VALUE FOR FLIPFLOP I.
            C      MTRMF(I)         VECTOR OF NEXT STATE VALUES.
            C  GENERAL AUXILIARY USE VECTOR.
            C      KTEMP(I)         GENERAL INDEX STORAGE USE ARRAY.
            C
            C
 0001              INTEGER   NRSCS(25),KSSC(25,5),NRTRMS(25,5)
 0002              DIMENSION NCSIV(25)
 0003              INTEGER   NODECS(1000),NODIP(1000),NPRED(1000),NODEH(1000)
 0004              INTEGER KTEMP(5),NODLEV(1000)
 0005              INTEGER   KGOAL(5)
 0006              INTEGER   MFIP(25,5,5),MFIV(25,5,5),MFSP(25,5,5),MFSV(25,5,5)
 0007              INTEGER   MCSIP(25,15),MCSIV(25,15)
 0008              INTEGER MGOALP(5),MGOALV(5),NSTATE(1000)
 0009              LOGICAL FST, COVER,DEBUG
 0010              LOGICAL PCSBF,PGRNDS,PGRWT,PHINV
 0011              COMMON/PAK/NVEC
 0012              COMMON/NSIM/KTRMI(16),KTRMF(36),MTRMF(36),NRFF
 0013              COMMON/HEUSB/NE,IMEGA,KPNAM(100),LNK(100),KOPL(100),KTRT(100),
                  *IPLAD(100),NIPL(100),KOTR(100),IMOCT(480),IDESP(120),LSOTR(240),
                  *KTRIN(240),KTYP(100),ITERP(45),KTRTP(85),KTPTA(50),KNIPL(10,3),
                  *MGRP(40),MGRV(40),NT,NP,NTPL
 0014              COMMON/PRINT/PCSBF,PGRNDS,PGRWT,PHINV
 0015              EQUIVALENCE (KNTRL,KSNODE)
 0016              INTEGER*2 SETA(1000)
 0017              DATA BLANK,XNEW,DEL/'     ','NEW ','DLET'/
 0018              DATA KS/'CS  '/
 0019              DATA KGR/'GR. '/
 0020              DATA DEBUG/.FALSE./
 0021              IRXA=65549
            C  DATA INPUT SECTION ***************************************************
            C
            C  READ GENERAL PARAMETERS.
 0022              READ 1, NRIN,NRFF,NCS
 0023              READ 1,NLIM,IMEGA
 0024              READ 7777,PCSBF,PGRNDS,PGRWT,PHINV
 0025         7777 FORMAT (5L1)
 0026              PRINT 801,NRIN,NRFF,NCS,NLIM,IMEGA
 0027          801 FORMAT(1H1/20X,'NUMBER OF INPUTS',T45,I10,/
                C   20X,'NUMBER OF FLIPFLOPS',T45,I10,/
                C   20X,'NUMBER OF CONTROL STATES',T45,I10/
                C   /20X,'SEARCH LOOP LIMIT',T50,I10/,20X,'OMEGA',T50,I10//)
            C  READ IN CONTROL STATE BRANCH FUNCTIONS.
 0028              IF (PCSBF) PRINT 802
 0029          802 FORMAT(15X,'CONTROL STATE BRANCH FUNCTIONS.',/)
 0030              DO 12 I=1,NCS
 0031              READ 1,NSCS
 0032              NRSCS(I)=NSCS
 0033              IF (PCSBF) PRINT 503,I,NSCS
 0034          803  FORMAT(20X,'CONTROL STATE ',I3,10X,I3,' SUCCESSORS')
 0035               DO 12 J=1,NSCS
 0036               READ 1, KSSC(I,J), NTRMS
 0037               NRTRMS(I,J)=NTRMS
 0038               IF (PCSBF) PRINT 804,KSSC(I,J),NTRMS
 0039          804  FORMAT(20X,'   C.S.',I3,10X,'FUNCTION HAS',I3,' TERMS.'/
                  C  30X,'INPUTS',10X,'FLIPFLOPS')
 0040               DO 12 K=1,NTRMS
 0041               READ 2, KTRMI,KTRMF
 0042               IF(PCSBF) PRINT 805,(KTRMI(IX),IX=1,NRIN)
 0043               IF(PCSBF) PRINT 806,(KTRMF(IX),IX=1,NRFF)
 0044          805  FORMAT(30X,16I1)
 0045          806  FORMAT(1H+,50X,36I1)
 0046               NVEC=NRIN
 0047               CALL PACK(KTRMI,MFIP(I,J,K),MFIV(I,J,K))
 0048               NVEC=NRFF
 0049               CALL PACK(KTRMF,MFSP(I,J,K),MFSV(I,J,K))
 0050           12  CONTINUE
 0051            1  FORMAT(3I5)
 0052            2  FORMAT(80I1)
                C  READ IN STARTING NODE AND SET UP AS FIRST NODE.
 0053               READ 4,KNTRL,MTRMF
 0054               PRINT 810,KNTRL,(MTRMF(IX),IX=1,NRFF)
 0055          810  FORMAT (//15X,'STARTING NODE',/30X,'C.S.',I3,5X,'STATE   ',36I1)
 0056               CALL PACK(MTRMF,X,NSTATE(1))
                C  READ IN GOAL NODES.
 0057               READ 1, NGOALS
 0058               PRINT 820,NGOALS
 0059          820  FORMAT(///15X,'GOAL NODES',10X,I3,' NODES.')
 0060               DO 10 I=1,NGOALS
 0061               READ 4,KGOAL(I),(KTRMF(IX),IX=1,NRFF)
 0062            4  FORMAT (I5,5X,36I1)
 0063               PRINT 822,KGOAL(I),(KTRMF(IX),IX=1,NRFF)
 0064          822  FORMAT(30X,'C.S.',I3,5X,'STATE   ',36I1)
 0065           10  CALL PACK(KTRMF,MGOALP(I),MGOALV(I))
 0066               CALL   GPTNT
                C  READ IN CONTROL STATE INPUT VECTORS.
 0067               IF (PHINV) PRINT 840
 0068          840  FORMAT(////20X,'HEURISTIC INPUT VECTORS BY CONTROL STATE.'/)
 0069               DO 17 I=1,NCS
 0070               READ 1, N
 0071               IF (PHINV) PRINT 841,I,N
 0072          841  FORMAT(20X,'CONTROL STATE',I3,10X,I3,' VECTORS.')
 0073               NCSIV(I)=N
 0074               NVEC=NRIN
 0075               DO 17 J=1,N
 0076               READ 2, KTRMI
 0077               IF (PHINV) PRINT 821,(KTRMI(IX),IX=1,NRIN)
 0078          821  FORMAT(10X,36I1)
 0079           17  CALL PACK(KTRMI,MCSIP(I,J),MCSIV(I,J))
 0080               DO 888 N=1,1000
 0081               SETA(N)=0
```

```
0082          888  CONTINUE
             C
             C  SET UP STARTING NODE POINTERS.
0083               PRINT 850
0084           850  FORMAT(1H1,///45X,'STATE SPACE SEARCH'///
                  C  10X,'ITER.',6X,'NODE',4X,'LEVEL',4X,'COST VALUE',3X,'C.S.',4X,
                  C 'PRED.NODE',3X,'INPUT VECTOR',5X,'STATE VECTOR'//)
0085               NCALLS=0
0086               NRNDS=1
0087               NODE=0
0088               NODECS(1)=KNTRL
0089               NXSTT=NSTATE(1)
0090               NODIP(1)=0
0091               NE=0
0092               NODLEV(1)=0
0093               FST=.TRUE.
0094               GO TO 500
             C
             C  SEARCH EXPANSION LOOP
             C
             C  SCAN FOR NEXT NODE TO EXPAND
             C  NEXT NODE TO BE EXPANDED WILL BE THE ONE WITH THE SMALLEST HEURISTIC
             C  VALUE WHICH NEVER BEFORE BEEN EXPANDED.
0095             9  FST=.FALSE.
0096               MINH=3000
0097               I=NRNDS
0098           109 I = I-1
0099               IF (I.LE.0) GO TO 104
0100               IF (NODEH(I).GE.MINH) GO TO 109
0101               IF(SETA(I).EQ.1) GO TO 109
0102               MINH=NODEH(I)
0103               NODE=I
0104               GO TO 109
             C  GENERATE NEXT INPUT FROM INPUT VECTOR SET.
             C  SEARCH INPUT APPLICATION LOOP * * * * * * * * * * * * * * * * * * * * * * * * * * *
0105           104  IF (MINH.LT.1500) GO TO 114
0106               PRINT 902
0107           902  FORMAT(///45X,'MINIMUM HEURISTIC NODE SEARCH FAILS.',/
                  C  45X,'SEARCH TERMINATED.')
0108               STOP
0109           114  KSNODE=NODECS(NODE)
0110               NRI=NCSIV(KSNODE)
0111               NSCS=NRSCS(KSNODE)
0112               SETA(NODE) = 1
             C  PLACE NODE ON SETA , ALREADY EXPANDED
0113               NE=NODLEV(NODE)+1
0114               DO 200 IIN=1,NRI
             C  CONTROL STATE APPLICATION LOOP.
0115               DO 300 ICS=1,NSCS
0116               NVEC=NRIN
0117               CALL UNPACK(KTRMI,MCSIP(KSNODE,IIN),MCSIV(KSNODE,IIN))
0118               IF (DEBUG) PRINT 1501,(KTRMI(IX),IX=1,II)
```

```
 0119          1501  FORMAT(10X,'INPUT APPLICATION LOOP',  16I1)
 0120                JJ=0
 0121                  NTRM=NRTRMS(KSNODE,ICS)
 0122                DO 301 I=1,NTRM
               C  CHECK IF TERM IS SATISFIED BY STATE OF PREDECESSOR NODE.
 0123                NVEC=NRFF
 0124                IF (.NOT.COVER(MFSP(KSNODE,ICS,I),MFSV(KSNODE,ICS,I),NSTATE(NODE))
                   C  ) GO TO 301
               C  TERM WAS SATISFIED.  FORM TABLE OF TERMS SATISFIED.
 0125                JJ=JJ+1
 0126                KTEMP(JJ)=I
               C CHECK IF  INPUT SATISFIES TERM
 0127                NVEC = NRIN
 0128                CALL UNPACK(KTRMF,MFIP(KSNODE,ICS,I),MFIV(KSNODE,ICS,I))
 0129                 IF(DEBUG) PRINT 1500,(KTRMF(IX),IX=1,11)
 0130          1500  FORMAT(10X,"CS BRANCHING TERM      ',16I1)
 0131                DO 309 J=1,NRIN
 0132                IF (KTRMI(J).GT.1) GO TO 309
 0133                IF (KTRMF(J).GT.1) GO TO 309
 0134                IF (KTRMI(J).NE.KTRMF(J)) GO TO 301
 0135           309  CONTINUE
 0136                GO TO 350
 0137           301  CONTINUE
 0138                IF (JJ.EQ.0) GO TO 300
               C  OVERRIDE INPUT VECTOR.
 0139                CALL RANDU(IRXA,IRXB,RANF)
 0140                IRXA=IRXB
 0141                J=RANF*NTRM+1
 0142                CALL UNPACK(KTRMF,MFIP(KSNODE,ICS,J),MFIV(KSNODE,ICS,J))
 0143                IF (DEBUG) PRINT 1505,(KTRMF(IX),IX=1,11)
 0144          1505  FORMAT(10X,'OVERRIDE INPUT VECTOR.',16I1)
               C  MERGE INPUT VECTOR AND REQUIRED VECTOR
 0145           350  DO 311 J=1,NRIN
 0146                IF (KTRMF(J).LT.2) KTRMI(J)=KTRMF(J)
 0147           311  CONTINUE
 0148                IF(DEBUG) PRINT 1503,(KTRMI(IX),IX=1,11)
 0149          1503  FORMAT(10X,'MERGE INPUT VECTOR     ',16I1)
               C  RANDOM FILL
 0150                DO 201 I=1,NRIN
 0151                IF (KTRMI(I).LE.1) GO TO 201
 0152                CALL RANDU(IRXA,IRXB,RANF)
 0153                IRXA=IRXB
 0154                KTRMI(I)=RANF+0.5
 0155           201  CONTINUE
 0156                IF (DEBUG) PRINT 1502,(KTRMI(IX),IX=1,11)
 0157          1502  FORMAT(10X,'RANDOM FILL RESULT     ',16I1)
               C
               C  PSEUDO-NSIM ROUTINE TO SIMULATE CIRCUIT BEHAVIOR.
               C
 0158                NCALLS=NCALLS+1
 0159                NVEC=NRFF
 0160                CALL UNPACK(KTRMF,0,NSTATE(NODE))
```

```
      CALL PSDNSM(KNTRL)
      NVEC=NRIN
      CALL PACK(KTRMI,X,NODIP(NRNDS))
      NVEC=NRFF
      CALL PACK(MTRMF,X,NXSTT)
      NSTATE(NRNDS)=NXSTT
      NODECS(NRNDS)=KSSC(KSNODE,ICS)
      NODLEV(NRNDS)=NE
C
C  COMPUTE HEURISTIC VALUE FOR VECTOR.
  500 CALL HEUSUB(NODECS(NRNDS),NXSTT,IVAL)
      IF (IVAL.LE.1023) GO TO 1010
      PRINT 100, IVAL
  100 FORMAT (' HEURISTIC VALUE EXCEEDS 1023 AT',I10)
      IVAL=1023
C  CHECK FOR REDUNDANT NODE. ASSIGN MINIMUM HEURISTIC LINKAGE.
 1010 N=NRNDS
      NODEH(NRNDS)=IVAL
      NTEMP=NRNDS
      STAT=BLANK
C  CHECK FOR REALISED GOAL.
  400 DO 102 I=1,NGOALS
      IF (.NOT.COVER(MGOALP(I),MGOALV(I),NSTATE(NRNDS))) GO TO 102
      IF (KGOAL(I).LT.0) GO TO 900
      IF (KNTRL.EQ.KGOAL(I)) GO TO 900
  102 CONTINUE
C CHECK IF NODE WAS ALREDY EXPANDED. IF SO DELETE
C IF ALREDY EXPANDED BUT HAS LOWER COST VALUE
C REMOVE FROM SET A AS A CANDIDATE FOR EXPANSION
  103 N=N-1
      IF (N.LE.0) GO TO 101
      IF (NSTATE(N).NE.NSTATE(NRNDS)) GO TO 103
      IF (NODECS(N).NE.NODECS(NRNDS)) GO TO 103
      IF (NODEH(N).LE.NODEH(NRNDS)) GO TO 106
      IF(SETA(N).EQ.0) GO TO 1111
      STAT=XNEW
      SETA(N) = 0
C REMOVE FROM SETA
      GO TO 105
 1111 CONTINUE
      NODIP(N)=NODIP(NRNDS)
      NPRED(N)=NPRED(NRNDS)
      NODEH(N)=NODEH(NRNDS)
      NODLEV(N)=NODLEV(NRNDS)
      GO TO 105
  101 NTEMP=NTEMP+1
      IF (NRNDS.LT.1000) GO TO 105
      PRINT 108
  108 FORMAT('0 ARRAY DIMENSIONING EXCEEDED.  SEARCH HALTS.')
      STOP
  106 STAT=DEL
  105 PRINT 800, STAT,NCALLS,NRNDS,NODLEV(NRNDS),IVAL,NODECS(NRNDS),NODE
```

```
      C ,(KTRMI(IX),IX=1,NRIN),(MTRMF(IX),IX=1,NRFF)
860   FORMAT(1X,A4,T8,6(I5,5X),4X, 11I1,T90,36I1)
      NRNDS=NTEMP
      IF (FST) GO TO 9
      IF (STAT.EQ.XNEW) PRINT 862, N
862   FORMAT(1H+,'NEW',I4)
      IF (NCALLS.LE.NLIM) GO TO 300
      PRINT 107
107   FORMAT (' NSIM CALL LIMIT EXCEEDED.  SEARCH HALTS.')
      STOP
300   CONTINUE
200   CONTINUE
      GO TO 9
909   CALL UNPACK(KTRMF,MGOALP(I),MGOALV(I))
      PRINT 904,(KTRMF(IX),IX=1,NRFF)
904   FORMAT(///45X,'SEARCH SUCCESSFUL'///30X,'GOAL REACHED',5X,36I1)
      PRINT 905,(MTRMF(IX),IX=1,NRFF)
905   FORMAT (30X,'FINAL NODE  ',5X,36I1)
      PRINT 875,NCALLS
875   FORMAT(//20X,I8,' NSIM CALLS')
      STOP
      END
```

```
FORTRAN IV G LEVEL  21                MAIN                DATE = 78341          13/34/28              PAGE 0001

            CT
                     SUBROUTINE UNPACK(K,KP,KV)
                     INTEGER K(36)
                     COMMON/PAK/N
                     KQ=KP
                     KW=KV
                     I=N
                 1   KQQ=KQ/2
                     KWW=KW/2
                     K(I)=2
                     IF (KQQ*2.GE.KQ) K(I)=0
                     IF (KWW*2.LT.KW) K(I)=1
                     KQ=KQQ
                     KW=KWW
                     I=I-1
                     IF (I.GT.0) GO TO 1
                     RETURN
                     END
```

```
FORTRAN IV G LEVEL  21                MAIN               DATE = 78341          13/34/28             PAGE 0001

              C/4
              CT
 0001               LOGICAL FUNCTION COVER(MP,MV,KV)
 0002               INTEGER K(36),M(36)
 0003               COMMON/PAK/N
              C  DETERMINE IF THE VALUES IN THE K-VECTOR SATISFY THE REQUIREMENTS
              C  PRESENTED BY THE M-VECTOR.
 0004               COVER=.FALSE.
 0005               CALL UNPACK(K,9,KV)
 0006               CALL UNPACK(M,MP,MV)
 0007               DO 1 I=1,N
 0008               IF (M(I).GT.1) GO TO 1
 0009               IF (M(I).NE.K(I)) RETURN
 0010             1 CONTINUE
 0011               COVER=.TRUE.
 0012               RETURN
 0013               END
```

```
                C
 0001                 SUBROUTINE PACK(K,KP,KV)
                C  REDUCES AN ARRAY TO TWO INTEGER WORDS
 0002                 INTEGER K(36)
 0003                 COMMON/PAK/N
 0004                 KP=0
 0005                 KV=0
 0006                 DO 1 I=1,N
 0007                 KP=KP+KP
 0008                 KV=KV+KV
 0009                 IF (K(I).GT.1) KP=KP+1
 0010                 IF (K(I).EQ.1) KV=KV+1
 0011             1   CONTINUE
 0012                 RETURN
 0013                 END
```

APPENDIX A.2

PETRI NET AND HEURISTIC COST VALUE ROUTINES

```
              C
              CT
 0001               SUBROUTINE GPTNT
              C ROUTINE THAT GENERATES THE PETRI NET
              C  READ IN PLACE ARRAYS AND SET POINTERS
 0002               COMMON/HEUSB/NE,IMEGA,KPNAM(100),LNK(100),KOPL(100),KTRT(100),
                   *IPLAD(100),NIPL(100),KOTR(100),IMDCT(480),IDESP(120),LSOTR(240),
                   *KTRIN(240),KTYP(100),ITERP(45),KTRTP(85),KTPTA(50),KNIPL(10,3),
                   *MGRP(40),MGRV(40),NT,NP,NTPL
 0003               DIMENSION INPUT(9,100),KTEM(20),ITEM(20),KTRMF(36)
 0004               LOGICAL DEBUG
 0005               DATA DEBUG/.FALSE./
 0006               DATA  REG,KONTR,IXIN,LAST,SP /'R','C','I','F',' ' /
 0007               M = 0
 0008               N = 0
 0009               IND = 0
              C READ IN PLACE TOKENS
              C  NEXT CARD
 0010          120  N = N+1
 0011               READ 10,(INPUT(J,N),J=1,9),KQ,(KTEM(J),J = 1,8)
 0012            10 FORMAT (9A1,1X,I2,1X,8I2)
 0013               IF (INPUT(1,N).EQ.LAST) GO TO 27
 0014               LNK(N) = 0
 0015               KPNAM(N) = INPUT (1,N)
 0016               IF (KPNAM(N).EQ.KONTR) LNK(N) = KQ
 0017               IF (KQ.GE.0)  GO TO 16
 0018               READ 72,KTRMF
 0019            72 FORMAT(80I1)
 0020               M = M+1
 0021               CALL PACK(KTRMF,MGRP(M),MGRV(M))
 0022               LNK(N) =M
 0023            16 CONTINUE
              C SET POINTER TO SET OF TRANSITIONS TO WHICH PLACE IS  OUTPUT
 0024               IND = IND +1
 0025               KOTR(N) = IND
 0026               DO 18 I = 1,8
 0027               LSOTR(IND) = KTEM(I)
 0028               IF (KTEM(I).EQ.0) GO TO 20
 0029               IND=IND +1
 0030            18 CONTINUE
 0031            20 GO TO 120
              C GET TOTAL # OF  PLACES
 0032            27 NP = N  - 1
              C NOW READ IN TRANSITION ARRAY
 0033               N = 0
 0034               IND = 0
 0035               M =0
 0036           220 N = N+1
 0037               READ 200,KTYP(N),NIPL(N),(KTEM(J),J =1,8),KOPL(N),
                  CKTRT(N),(ITEM(J),J =1,10)
 0038           200 FORMAT(I1,1X,I2,1X,8I2,1X,I2,1X,I2,1X,10I2)
 0039               IF (KTYP(N).GT.6) GO TO 300
```

```
C SET POINTER TO INPUT PLACES OF CURRENT TRANSITIONS
          IND =IND+1
          IPLAD(N) =IND
          DO 25 I=1,8
           IF (KTEM(I).EQ.0)) GO TO 28
          KTRIN(IND) = KTEM(I)
          IND =IND+1
       25 CONTINUE
       28 M = M+1
          IDESP(N) =M
C THE ADDRESS
          DO 29 I=1,10
          IMDCT(M) =ITEM(I)
          IF (ITEM(I).EQ.0) GO TO 30
          M = M+1
       29 CONTINUE
       30 GO TO 220
      300 NT =N  - 1
C TOTAL NO OF TRANSITIONS
C PRINT THE PETRI NET
          PRINT 400
      400 FORMAT(1H1,/////,15X,'LISTING OF PETRI NET')
          PRINT 401
      401 FORMAT(///,9X,'TRANSITION',5X,'TYPE',5X,'OUTPUT PLACE',
         C5X,'INPUT PLACES',2X,'IMMEDIATE DESCDT ')
          DO 470 N=1,NT
          PRINT 450,N,KTYP(N),KOPL(N)
      450 FORMAT(9X,'T',I2,10X,I2,10X,'P',I2,15X,'P',I2)
          NINPP =NIPL(N)
          IF (NINPP.EQ.0) GO TO 453
          M =IPLAD(N)
          DO 452 I=1,NINPP
          KK =KTRIN(M)
          IF( M.EQ.0) GO TO 470
          PRINT 460,KTRIN(M)
      460 FORMAT(50X,I2)
          M=M+1
      452 CONTINUE
      453 CONTINUE
          M =IDESP(N)
          DO 451 I=1,8
          KK = IMDCT(M)
          IF (KK.EQ.0) GO TO 470
          PRINT 411,KK
      411 FORMAT(62X,I2)
          M =M+1
      451 CONTINUE
      470 CONTINUE
          PRINT 480
          DO 500 N=1,NP
      480 FORMAT(1H1,//////,20X,'PLACE LISTING ',///)
          PRINT 490,N,(INPUT(J,N), J=1,9)
```

```
0087          500 CONTINUE
0088          490 FORMAT(20X,'P',I4,1X,9A1)
            C READ IN COUNTER PLACE NUMBERS
0089              N =0
0090          600 N =N+1
0091              READ 611,KNTPL(N,1),KNTPL(N,2),KNTPL(N,3)
0092          611 FORMAT(3(I2,1X))
0093              IF (KNTPL(N,1).EQ.0) GO TO  687
0094              IF(DEBUG) PRINT 622, KNTPL(N,1),KNTPL(N,2),KNTPL(N,3)
0095          622 FORMAT(5X,'COUNTER PLACE ',I2,5X,I2,5X,I2)
0096              GO TO 600
            C READ IN TERMINAL PLACE INFO
0097          687 N=0
0098              IND =0
0099          633 N =N+1
0100              READ 644,ITERP(N),(KTEM(J),J=1,15)
0101              IF (ITERP(N).EQ.0) GO TO 666
0102          644 FORMAT (I2,1X,15I2)
0103              IND =IND+1
0104              KTPTA(N) =IND
0105              DO 655 I=1,15
0106              KTRTP(IND) =KTEM(I)
0107              IF (KTRTP(IND).EQ.0) GO TO 633
0108              IND =IND+1
0109          655 CONTINUE
0110          666 NTPL=N-1
0111              IF(.NOT.DEBUG) RETURN
0112              DO 700 N=1,NTPL
0113              PRINT 677,ITERP(N)
0114          677 FORMAT(5X,'TERM PLACE ',I2,'INPUT TO',I2)
0115               M =KTPTA(N)
0116              DO 688 I=1,15
0117              KK = KTRTP(M)
0118              IF (KK.EQ.0) GO TO  700
0119              PRINT 777,KK
0120          777 FORMAT(12X,I2)
0121              M =M+1
0122          688 CONTINUE
0123          700 CONTINUE
0124              RETURN
0125              END
```

```
               CT
 0001               SUBROUTINE HEUSUB(KNTRL,NXSTT,IVAL)
               C  THIS ROUTINE DERIVES THE STATE VECTOR OF THE PETRI NENT
               C  BEFORE & AFTER FIRING. THE VECTOR KR IS COMPUTED AND
               C  THE FINAL HEURISTIC FUNCTION
 0002               COMMON/HEUSB/NE,IMEGA,KPNAM(100),LNK(100),KOPL(100),KTRT(100),
                   *IPLAD(100),NIPL(100),KOTR(100),IMOCT(480),IDESP(120),LSOTR(240),
                   *KTRIN(240),KTYP(100),ITERP(45),KTRTP(85),KTPTA(50),KNTPL(10,3),
                   *MGRP(40),MGRV(40),NT,NP,NTPL
 0003               INTEGER M(100),KR(100),KFIRB(100)
 0004               INTEGER KTRMF(36),MTRMF(36),IXXZ
 0005               INTEGER GRPH,XINC,IXIN,CS
 0006               COMMON/PAK/NV
 0007               COMMON/NSIM/IXXZ(88),NRFF
 0008               LOGICAL DEBUG,COVER
 0009               LOGICAL GT,LT
 0010               DATA DEBUG /.FALSE./
 0011               DATA GRPH,CS,IXIN,XINC /'G', 'C','I','K'/
 0012               NV=NRFF
               C CLEAR M VECTOR TO ZERO
 0013               DO 1 I =1,NP
 0014             1 M(I) =0
 0015               DO  6 I =1,NP
               C C GOAL OR COUNTER PLACE
 0016               IF(KPNAM(I).EQ.GRPH.OR.KPNAM(I).EQ.XINC) GO TO 6
 0017               IF (KPNAM(I).EQ.CS) GO TO 62
 0018               IF (KPNAM(I).EQ.IXIN) GO TO 80
               C     REGISTER VECTOR NODE -- CHECK FOR COVER.
 0019            60  LIK=LNK(I)
 0020               IF (COVER(MGRP(LIK),MGRV(LIK),NXSTT)) GO TO 80
 0021               GO TO 6
 0022            62  IF (LNK(I).NE.KNTRL) GO TO 6
 0023            80 M(I) =1
 0024             6 CONTINUE
 0025               IF(DEBUG) PRINT 77,(M(J),J=1,NP)
 0026            77 FORMAT(///,5X,'STATE VECTOR BEFORE FIRING ',/,5X,80I2)
               C NOW DERIVE THE STATE VECTOR AFTER THE K TH FIRIG
 0027               DO 199 I =1,NT
 0028               KFIRB(I) =0
 0029               NIP =NIPL(I)
 0030               IND =IPLAD(I)
               C PICK THE INPUT  PLACE T O TRANSITION I
 0031           111 J = KTRIN(IND)
 0032               IF (M(J).EQ.0) GO TO 199
 0033               NIP = NIP-1
 0034               IF (KTYP(I).EQ.6) GO TO 125
               C UNCONDITIONAL CONTROL STATE TRANSITION
 0035               IF (NIP.EQ.0) GO TO 120
 0036               IND =IND+1
 0037               GO TO 111
 0038          120  IF (KTYP(I).NE.3) GO TO 125
 0039               J = KOPL(I)
 0040                  M(J) =1
 0041                  GO TO 199
 0042              125 KFIRB(I) =1
              C KFIFB IS THE LIST OF FIRABLE TANSITIONS I.E.
              C ALL CONDITIONS FOR FIRING ARE FULFILLED ONLLY THEY HAVE NOT BEEN
              C  COMMANDED TO FIRE
 0043              199 CONTINUE
 0044                  IF (DEBUG) PRINT 200,(M(J),J=1,NP)
 0045              200 FORMAT(5X,'STATE VECTOR AFTER FIRING',///,1X,80I2)
 0046                  IF (DEBUG) PRINT 2222,(KFIRB(J),J=1,NT)
 0047             2222 FORMAT(5X,'FIRABLE TRANSITIONS',/,1X,80I2)
              C DERIVE VECTOR KR. THE SET OF ALL FIRED TRANSITIONS AND THOSE THAT CAN
              C BE INFERRED TO BE FIRED
              C CLEAR VECTOER KR
 0048                  DO 201 I =1,NT
 0049              201 KR(I) =0
 0050                  DO 249 J =1,NP
 0051                  IF (M(J).EQ.0) GO TO 249
 0052                  IND = KOTR(J)
 0053                  IF(IND.EQ.0) GO TO 249
 0054              211 I = LSOTR(IND)
              C PICK TRANSITION TO WHICH THIS PLACE IS OUTPUT
 0055                  IF(I.EQ.0) GO TO 249
 0056                  KR(I) =1
 0057                  IND = IND+1
 0058                  GO TO 211
 0059              249 CONTINUE
 0060                  IF (DEBUG) PRINT 250,(KR(J),J=1,NT)
 0061              250 FORMAT(//,1X,'FIRED TRANSITIONS ',//,1X,80I2)
 0062                  DO 299 I =1,NT
 0063                  IF (KR(I).EQ.0) GO TO 299
 0064                  IND = IDESP(I)
 0065             252  IMS =IMDCT(IND)
              C PICK IMMEDIATE DESECNDANT OF TRANSITION I
 0066                  IF (IMS.EQ.0) GO TO 299
 0067                  KR(IMS) =1
 0068                  IND = IND+1
 0069                  GO TO 252
 0070              299 CONTINUE
 0071                  IF (DEBUG) PRINT 300,(KR(J),J=1,NT)
 0072              300 FORMAT(//,5X,' KR AFTER ALL INFERRED TRAS. HAVE BEEN ADDED',
                     C//,80(I2))
              C NOW COMPUTE THE HEURISTIC FUNCTION.
 0073                  HPT =0.
 0074                  HCNT =0.
 0075                  HRVE = 0.
 0076                  HSFN =0.
              C ITERP IS THE SET OF YE   TERMINAL PLACES. KTPTA IS THE POINTER TO
              C TRANSITIONS TO WHICH  TERMINAL PLACE I POINTS
              C T  KTRTP IS THE LIST  OF TRANSITIONS WHICH HAVE TERMINAL PLACES AS INPUT
 0077                  DO 349 L=1,NTPL
 0078                   J =ITERP(L)
```

```
              C PICK TERMINAL PLACE
 0079               IF (M(J).EQ. 0) GO TO 349
              C CHECK ABOVE
 0080               N =KTPTA(L)
              C ADDRESS OF TRANSITION TO WHICH  J IS  INPUT
 0081           305 I = KTRIP(N)
              C PICK TRANS
 0082               IF (I.EQ.0) GO TO 349
 0083               IF (KR(I).EQ.1) GO TO 310
 0084               NIP =NIPL(I)
 0085               IF (KTYP(I).EQ.3)  HPT = HPT + 1./NIP
 0086           310 N =N+1
 0087               GO TO 305
 0088           349 CONTINUE
 0089               IF(DEBUG) PRINT 350, HPT
 0090           350 FORMAT(5X,'COTRIBU. FROM TERM PLACES ',F10.4)
 0091               DO 449 I=1,NT
 0092               IF(KTYP(I).NE.4) GO TO 400
              C    INC HEURISTIC COMPUTATION
 0093               J =KOPL(I)
              C OUTUT PLACE
              C KNTPL IS THE SET OF COUNT PLACES
              C KNTPL(I,1) CONTAINS THE # PLACE #,KNTPL(I,2) IS
              C THE NO OF FLIPFLOPS IN THE COUNTER
              C KNTPL(I,3) IS THE FIRST FF NO OF COUNTER I
 0094               DO 351 N=1,10
 0095               IF (KNTPL(N,1).EQ.J) GO TO 356
 0096               IF (N.EQ.10) PRINT 355,J
 0097           351 CONTINUE
 0098           355 FORMAT(5X,'KNT PLACE ',I2,'NOT FOUND IN KNTPL TABLE')
 0099           356 KK = KNTPL(N,2)
 0100               JJ = KNTPL(N,3)
 0101               JK = JJ + KK - 1
 0102           64  LIK=LNK(J)
              C  INC HEURISTIC COMPUTATION.
 0103               KVAL=0
 0104               MVAL=0
 0105               LT=.FALSE.
 0106               GT=.FALSE.
 0107               CALL UNPACK(MTRMF,MGRP(LIK),MGRV(LIK))
 0108               CALL UNPACK(KTRMF,0,NXSTT)
              C  CHECK FOR GOAL GREATER THAN, LESS THAN, AND EQUAL TO NEXT STATE.
 0109               DO 65 J=JJ,JK
 0110               IF (MTRMF(J).EQ.0.AND.KTRMF(J).EQ.1) LT=.TRUE.
 0111               IF (MTRMF(J).EQ.1.AND.KTRMF(J).EQ.0) GT=.TRUE.
 0112               IF (GT.OR.LT) GO TO 66
 0113           65  CONTINUE
              C  SET MTRMF INDETERMINATES TO GIVE MINIMUM DIFFERENCE FROM NXSTT
 0114           66  DO 67 K=JJ,JK
 0115               IF (MTRMF(K).LT.2) GO TO 66
 0116               MTRMF(K)=1
 0117               IF (GT) MTRMF(K)=KTRMF(K)
```

```
               C  COMPUTE VALUES IN EACH COUNTER.
 0118            68  KVAL=KVAL*2+KTRMF(K)
 0119            67  MVAL=MVAL*2+MTRMF(K)
 0120                IF (LT) MVAL = MVAL + 2**KK
 0121            70  KD = (MVAL - KVAL)
 0122                XVAL = MVAL
 0123                HCNT = HCNT + (1. -KD/XVAL)
 0124                IF (DEBUG) PRINT 357,KD,HCNT,(KTRMF(N),N=1,NRFF),
                    *(MTRMF(N),N=1,NRFF)
 0125            357 FORMAT(5X,'KD ',I3,' HCNT ',F10.4,/,5X,' KTRMF   ',5O12,
                    */,5X,' MTRMF    ',5O12)
 0126                GO TO 449
 0127            400 IF(KR(I).EQ.0) GO TO 449
 0128                HRVE  =HRVE + 1.
 0129            449 CONTINUE
 0130                HSFN =NT- (HPT + HCNT + HRVE)
 0131                IVAL = NE + IMEGA*HSFN/NT
 0132                IF(DEBUG) PRINT 455,HRVE,HSFN,IVAL
 0133            455 FORMAT(5X,'R VEC CONTR.  ',F10.4,' HFN  ',F10.4,I3)
 0134                RETURN
 0135                END
```

APPENDIX A.3

PSEUDO-NSIM SUBROUTINES FOR CASES I-IV

```
              SUBROUTINE PSDNSM(KNTRL)
C   REGISTER TRANSFER SIMULATOR (PSEUDO-NSIM) FOR CASE 1
              COMMON/NSIM/KTRMI(16),KTRMF(36),MTRMI (36),NRFF
              DO 1 I=1,11
          1   MTRMF(I)=KTRMF(I)
              GO TO (10,20,20,40,50,60,70,80,90,100,110),KNTRL
         10   DO 11 I=1,3
              MTRMF(I)=KTRMI(I)
         11   CONTINUE
              MTRMF(11)=KTRMI(4)
              GO TO 71
         30   DO 31 I=7,10
         31   MTRMF(I)=0
         20   IVAL=0
              DO 21 I=1,3
         21   IVAL=IVAL+IVAL+KTRMF(I)+KTRMF(3+I)
              DO 22 J=1,3
              K=IVAL/2
              MTRMF(7-J)=0
              IF (K+K.LT.IVAL) MTRMF(7-J)=1
         22   IVAL=K
              RETURN
         40   IF (KTRMI(4).EQ.1) GO TO 71
              RETURN
         50   DO 51 I=1,3
              IX=I-1
              IF (IX.EQ.0) IX=3
              MTRMF(IX)=0
              IF (KTRMF(I).EQ.1.AND.KTRMF(3+I).EQ.1) MTRMF(IX)=1
         51   CONTINUE
              RETURN
         60   DO 61 I=4,6
         61   MTRMF(I)=0
              RETURN
         70   IF (KTRMI(4).EQ.0) RETURN
C   INCREMENT THE COUNTER
         71   IVAL=0
              DO 72 I=7,10
         72   IVAL=IVAL+IVAL+KTRMF(I)
              IVAL=IVAL+1
              DO 73 I=1,4
              K=IVAL/2
              MTRMF(11-I)=0
              IF (K+K.LT.IVAL) MTRMF(11-I)=1
         73   IVAL=K
              RETURN
         80   IF (KTRMI(5).EQ.0) RETURN
              DO 81 I=1,3
              MTRMF(I)=KTRMF(3+I)
         81   MTRMF(3+I)=KTRMF(I)
              RETURN
        100   DO 101 I=1,3
```

```
FORTRAN IV G LEVEL  21                 PSDRSM              DATE = 78341          13/34/28            PAGE 0002

                 IX=I-1
                 IF (IX.EQ.0) IX=3
                 MTRMF(IX)=KTRMF(I)
           101   MTRMF(IX+3)=KTRMF(I+3)
                 RETURN
            90   IF (KTRMI(4).EQ.0) RETURN
                 MTRMF(3)=0
                 DO 91 I=1,6
                 IF (KTRMF(I).EQ.0) RETURN
            91   CONTINUE
                 MTRMF(3)=1
                 RETURN
           110   MTRMF(1)=KTRMF(2)
                 MTRMF(2)=KTRMF(3)
                 MTRMF(3)=KTRMI(3)
                 MTRMF(4)=KTRMF(5)
                 MTRMF(5)=KTRMF(6)
                 MTRMF(6)=KTRMF(1)
                 RETURN
                 END
```

```
              CT
 0001                SUBROUTINE PSDNSM(KNTR)
              C  REGISTER TRANSFER SIMULATOR (PSEUDO-NSIM) FOR CASE II
 0002                COMMON/NSIM/KTRMI(16),KTRMF(36),MTRMF(36),NRFF
 0003                GO TO (1,2,3,2,9,2,7,2,9),KNTR
 0004              9 RETURN
              C  INCREMENT THE COUNTER (CONTROL STATES 2,4,6,8)
 0005              2 J=0
 0006                DO 101 I=1,3
 0007            101 J=J+J+KTRMF(I)
 0008                J=J+1
 0009                DO 102 I=1,3
 0010                K=J/2
 0011                MTRMF(4-I)=0
 0012                IF (J.NE.K+K) MTRMF(4-I)=1
 0013            102 J=K
 0014                RETURN
              C  CONTROL STATE 3.  INPUT TO ACCUMULATOR
 0015              3 DO 103 I=1,8
 0016            103 MTRMF(3+I)=KTRMI(I)
 0017                GO TO 1
              C  CONTROL STATE 7. COMPLEMENT OF INPUT TO BUFFER.
 0018              7 DO 104 I=1,8
 0019                MTRMF(3+I)=1-KTRMI(I)
 0020                IF (MTRMF(3+I).LT.0) MTRMF(3+I)=2
 0021            104 CONTINUE
              C  CLEAR THE COUNTER (CONTROL STATES 1,3,7,9)
 0022              1 DO 100 I=1,3
 0023            100 MTRMF(I)=0
 0024                RETURN
 0025                END
```

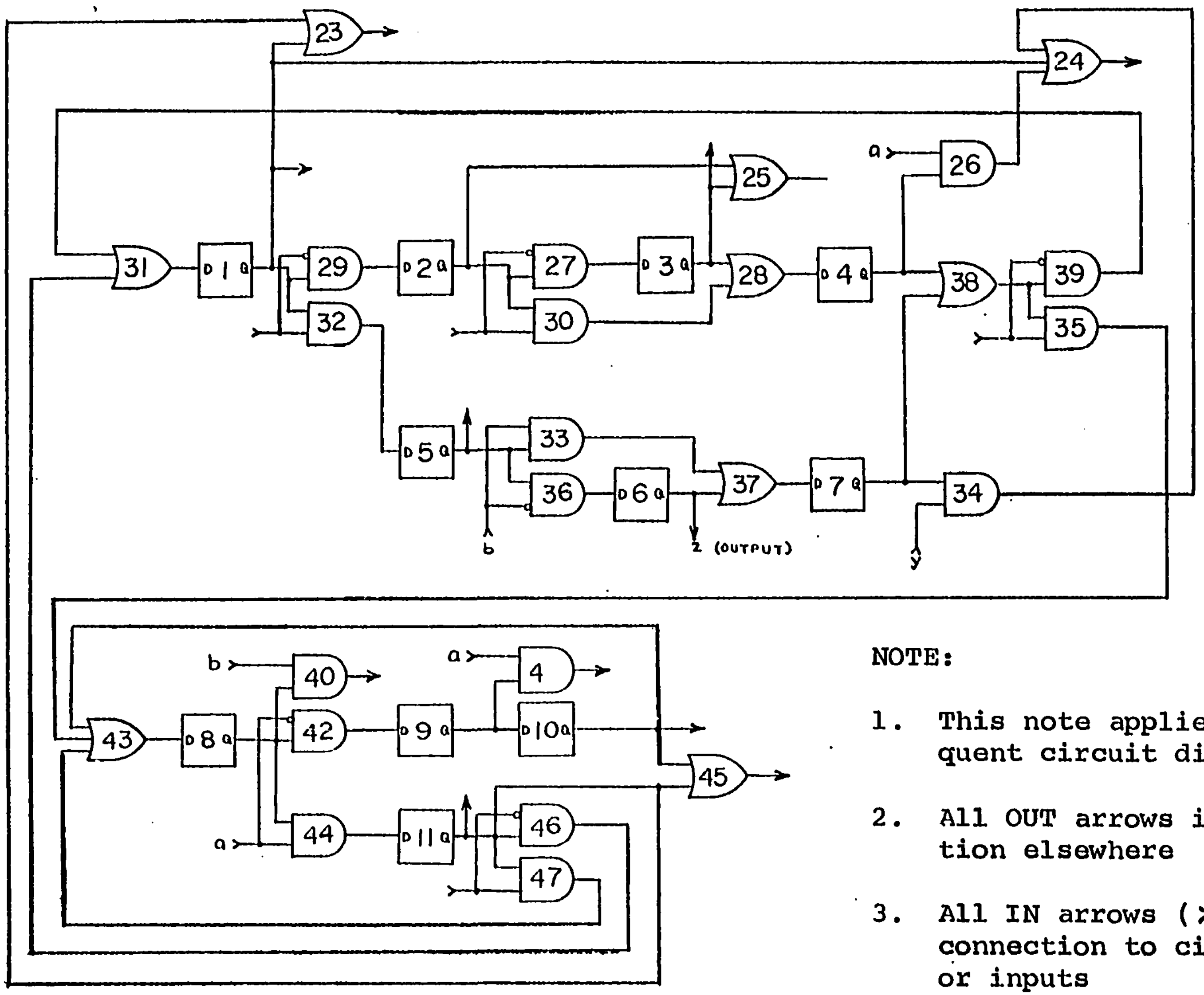
```
      SUBROUTINE PSDNSM(KNTRL)
C  REGISTER TRANSFER SIMULATOR (PSEUDO-NSIM) FOR CASE III
C  AR0=FF1,  IR0=FF4,  MD0=FF7,  AC0=FF9
      COMMON/NSIM/KTRMI(16),KTRMF(36),MTRMF(36),NRFF
      DO 1 I=1,NRFF
    1 MTRMF(I)=KTRMF(I)
      GO TO (10,20,30,40,50,41),KNTRL
C  CONTROL STATE 1     AR=IN A,      IR=IN IR
   10 DO 11 I=1,6
   11 MTRMF(I)=KTRMI(I)
C  CONTROL STATE 2.  BRANCHING ONLY.
   20 RETURN
C  CONTROL STATE 3.  CONDITIONAL TRANSFERS BASED ON IR.
   30 IF (KTRMF(5).EQ.1) GO TO 34
      IF (KTRMF(6).EQ.1) GO TO 32
C  IR=X00       AC=X
      DO 31 I=1,2
   31 MTRMF(8+I)=KTRMI(6+I)
      RETURN
C  IR=X01       AC=MD       MD=AC
   32 DO 33 I=1,2
      MTRMF(8+I)=KTRMF(6+I)
   33 MTRMF(6+I)=KTRMF(8+I)
      RETURN
   34 IF (KTRMF(6).EQ.1) GO TO 36
C  IR=X10       AC=AC.AND.MD
      DO 35 I=1,2
      MTRMF(8+I)=0
      IF (KTRMF(8+I).EQ.1.AND.KTRMF(6+I).EQ.1) MTRMF(8+I)=1
   35 CONTINUE
C  IR=X11       AC=-AC
   36 DO 37 I=9,10
      MTRMF(I)=0
      IF (KTRMF(I).EQ.0) MTRMF(I)=1
   37 CONTINUE
      RETURN
C  CONTROL STATE 4.
   40 IF (KTRMF(6).EQ.1) GO TO 41
C  ROTATE MD LEFT.
      MTRMF(7)=KTRMF(8)
      MTRMF(8)=KTRMF(7)
      RETURN
C  COMPUTE THE BASE FROM THE ADDRESS IN REGISTER AR.
   41 J=0
      DO 42 I=1,3
      J=J+J
      IF (KTRMF(I).EQ.1) J=J+1
   42 CONTINUE
      J=J*2+10
      IF (KNTRL.EQ.6) GO TO 60
C  READ FROM MEMORY TO MD
      DO 43 I=1,2
```

```
 0040          43  MTRMF(6+I)=KTRMF(I+J)
             C  CONTROL STATE 5.  OUTPUT ONLY.
 0041          50  RETURN
             C  CONTROL STATE 6.  WRITE MD TO MEMORY.
 0042          60  DO 61 I=1,2
 0043          61  MTRMF(I+J)=KTRMF(6+I)
 0044              RETURN
 0045              END
```

```
                SUBROUTINE PSDNSM(KNTRL)
          C  REGISTER TRANSFER SIMULATOR (PSEUDO-NSIM) FOR CASE IV
                COMMON/NSIM/KTRMI(16),KTRMF(36),MTRMF(36),NRFF
                INTEGER   JC(5),JB(4), KTRMI,KTRMF,MTRMF,JX,KX
                DO 200 I=1,NRFF
           200  MTRMF(I)=KTRMF(I)
                GO TO  (1,100,30,100,100,3,100,3,9,10,100,3,13,20,15,3,17,
               *  100,20,100,100),KNTRL
             1  DO 201 I=1,4
           201  MTRMF(I)=KTRMF(8+I)
                RETURN
            30  DO 202 I=1,4
           202  MTRMF(12+I)=KTRMI(4+I)
             3  DO 203 I=1,4
           203  MTRMF(I)=KTRMI(I)
                RETURN
             9  IF (KTRMF(15).NE.0) GO TO 205
                DO 204 I=1,4
           204  MTRMF(4+I)=KTRMF(I)
                RETURN
           205  IF (KTRMF(15).EQ.0.OR.KTRMF(16).EQ.1) GO TO 208
                PRINT 806
           806  FORMAT(5X,' ADD(AC,OR)')
                I=4
                JX=KTRMI(9)
           206  KX=LX(KTRMF(I),KTRMF(I+4))
                MTRMF(I+4)=LX(KX,JX)
                KX=LO(KTRMF(I),KTRMF(I+4))
                JX=LA(KX,JX)
                KX=LA(KTRMF(I),KTRMF(I+4))
                JX=LO(JX,KX)
                I=I-1
                IF (I.GT.0) GO TO 206
                RETURN
           208  DO 209 I=1,4
           209  MTRMF(4+I)=1-LA(KTRMF(I),KTRMF(4+I))
                RETURN
            10  IF (KTRMF(15).NE.0) GO TO 20
                DO 210 I=1,4
           210  MTRMF(I)=KTRMF(4+I)
                RETURN
            20  JC(5)=KTRMI(9)
                I=4
           211  JC(I)=LA(KTRMF(8+I),JC(I+1))
                JB(I)=LX(KTRMF(8+I),JC(I+1))
                I=I-1
                IF (I.GT.0) GO TO 211
                IF (KNTRL.NE.10) GO TO 22
            21  IF (KTRMF(15).NE.1) GO TO 100
                DO 212 I=1,4
           212  MTRMF(I)=JB(I)
                RETURN
```

APPENDIX B

CIRCUIT SCHEMATICS FOR CASES I-IV



NOTE:

1. This note applies to all subsequent circuit diagrams

2. All OUT arrows indicate connection elsewhere

3. All IN arrows (≻) indicate connection to circuit elements or inputs

4. Clock and reset lines are not shown

Fig. B-1 Case I: Control Circuit

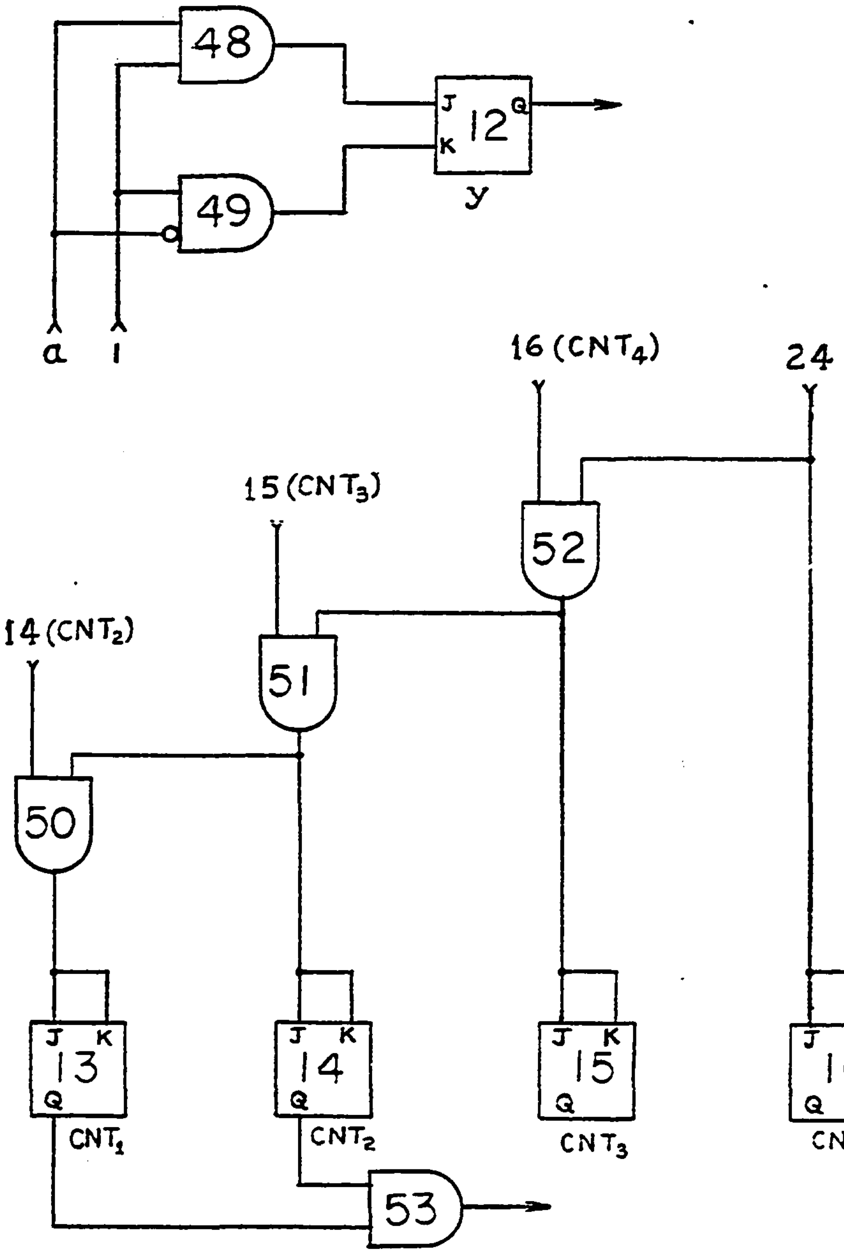


Fig. B.2 Case I: Counter Circuit

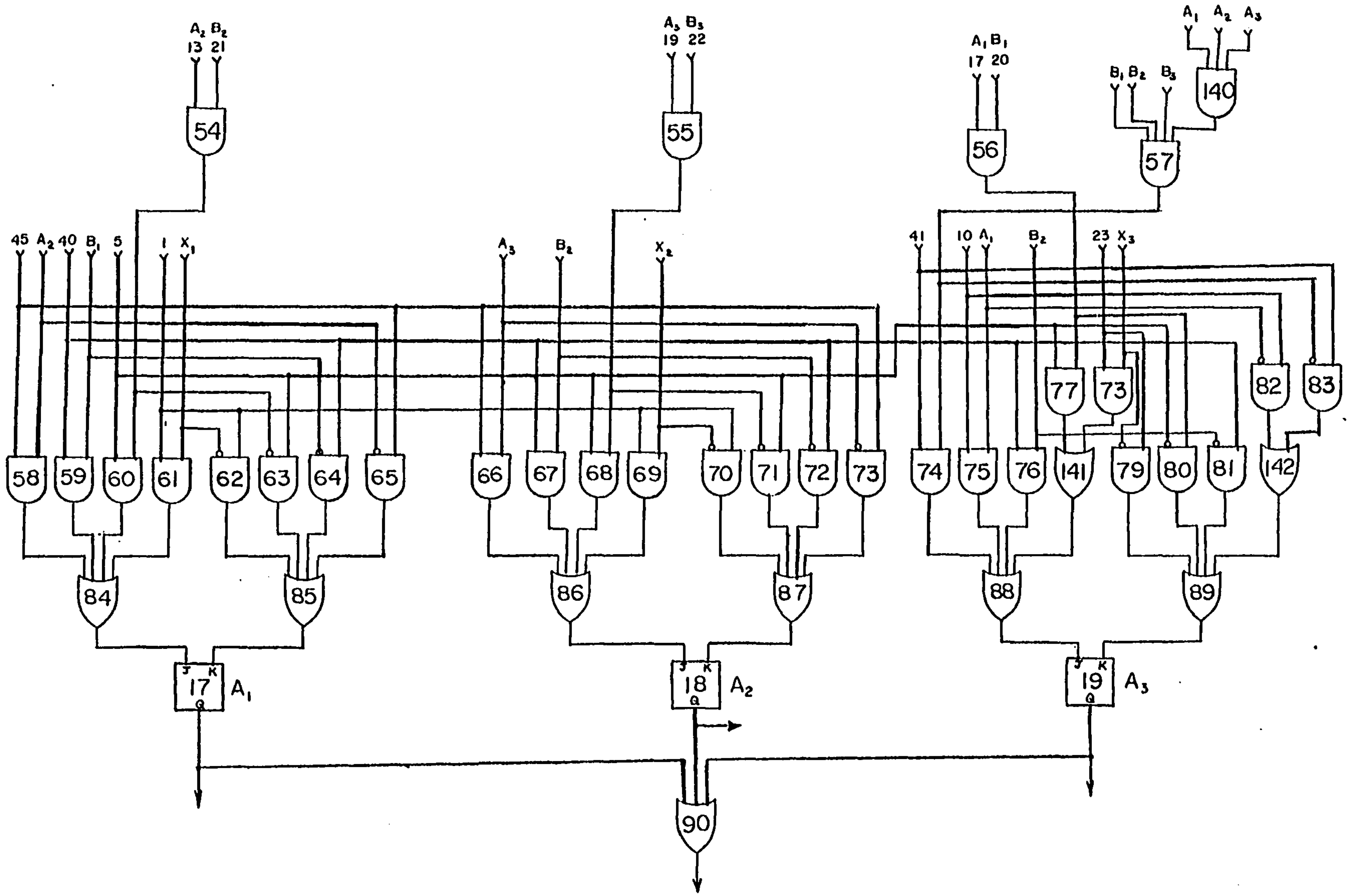


Fig. B.3 Case I: Register A Circuit

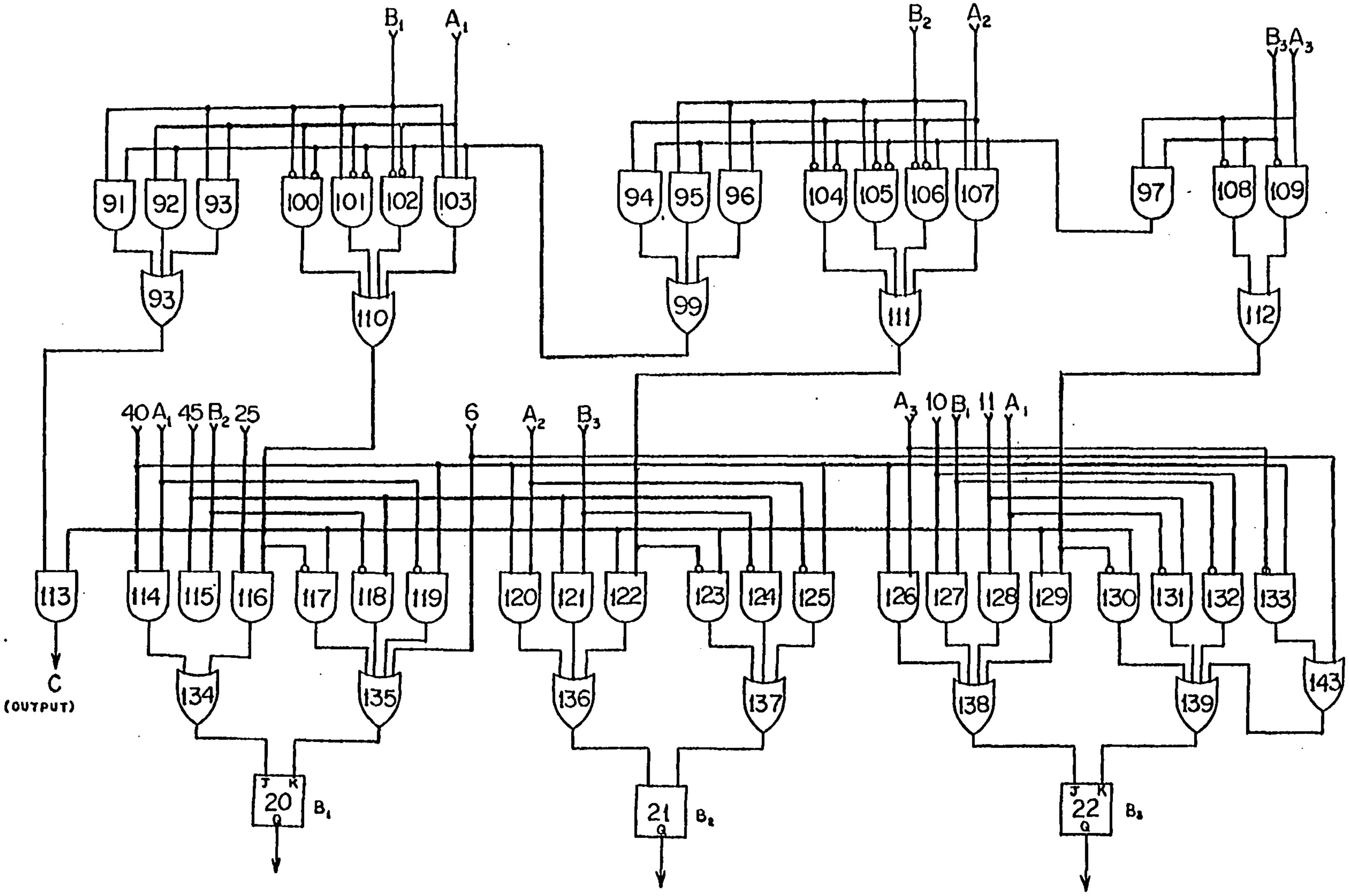


Fig. B.4 Case I: Register B Circuit

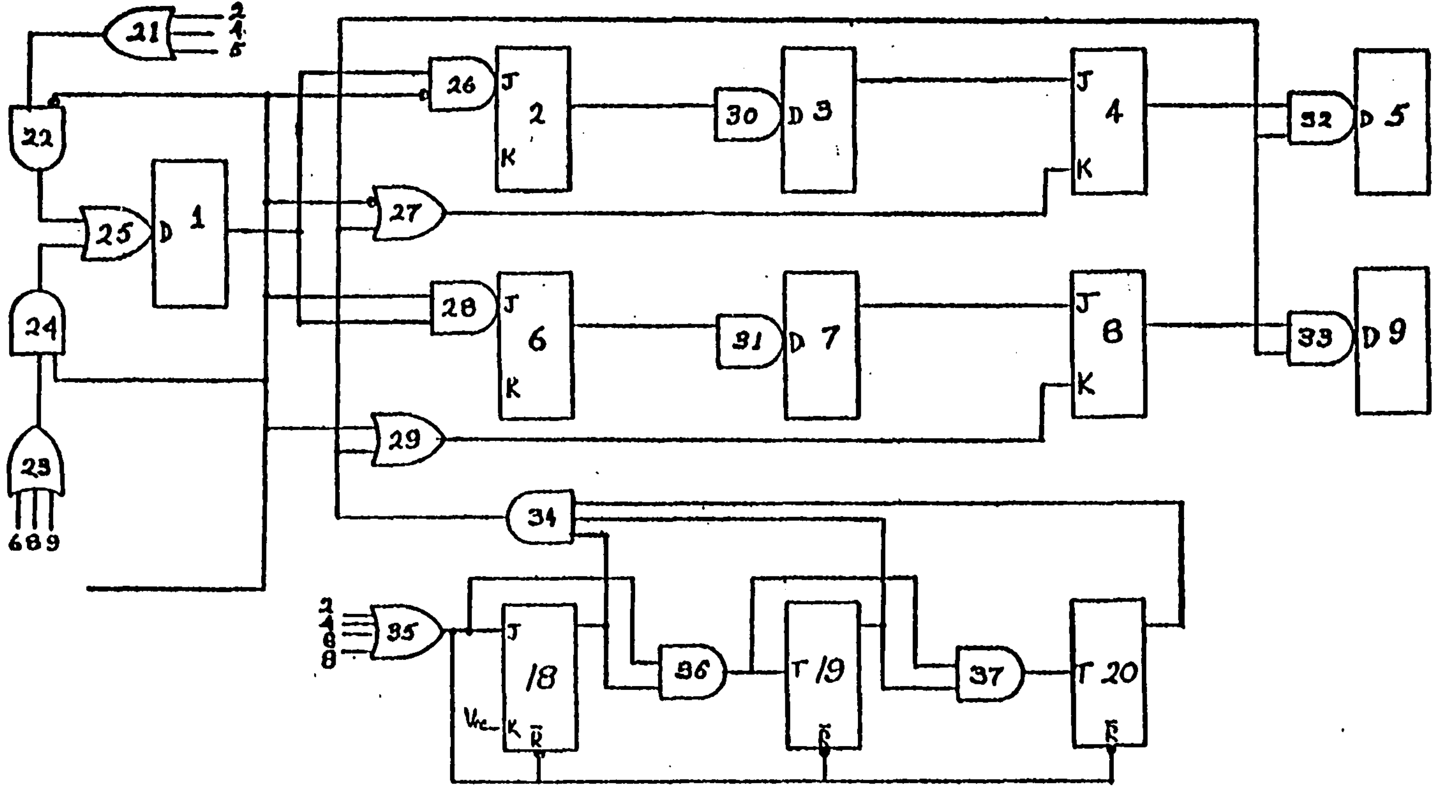


Fig. B.5 Case II: Control Circuit

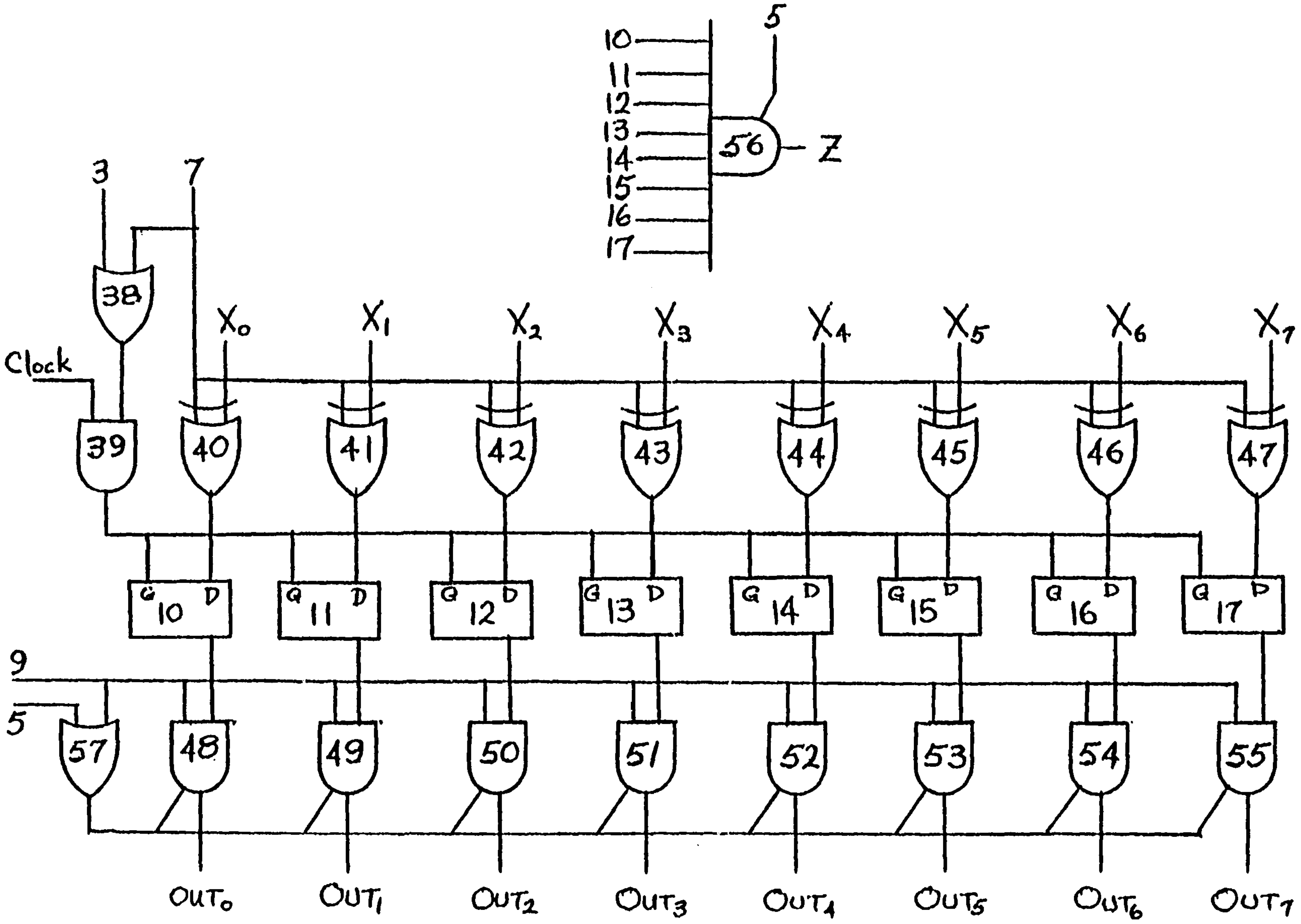


Fig B.6 Case II : Register Circuits

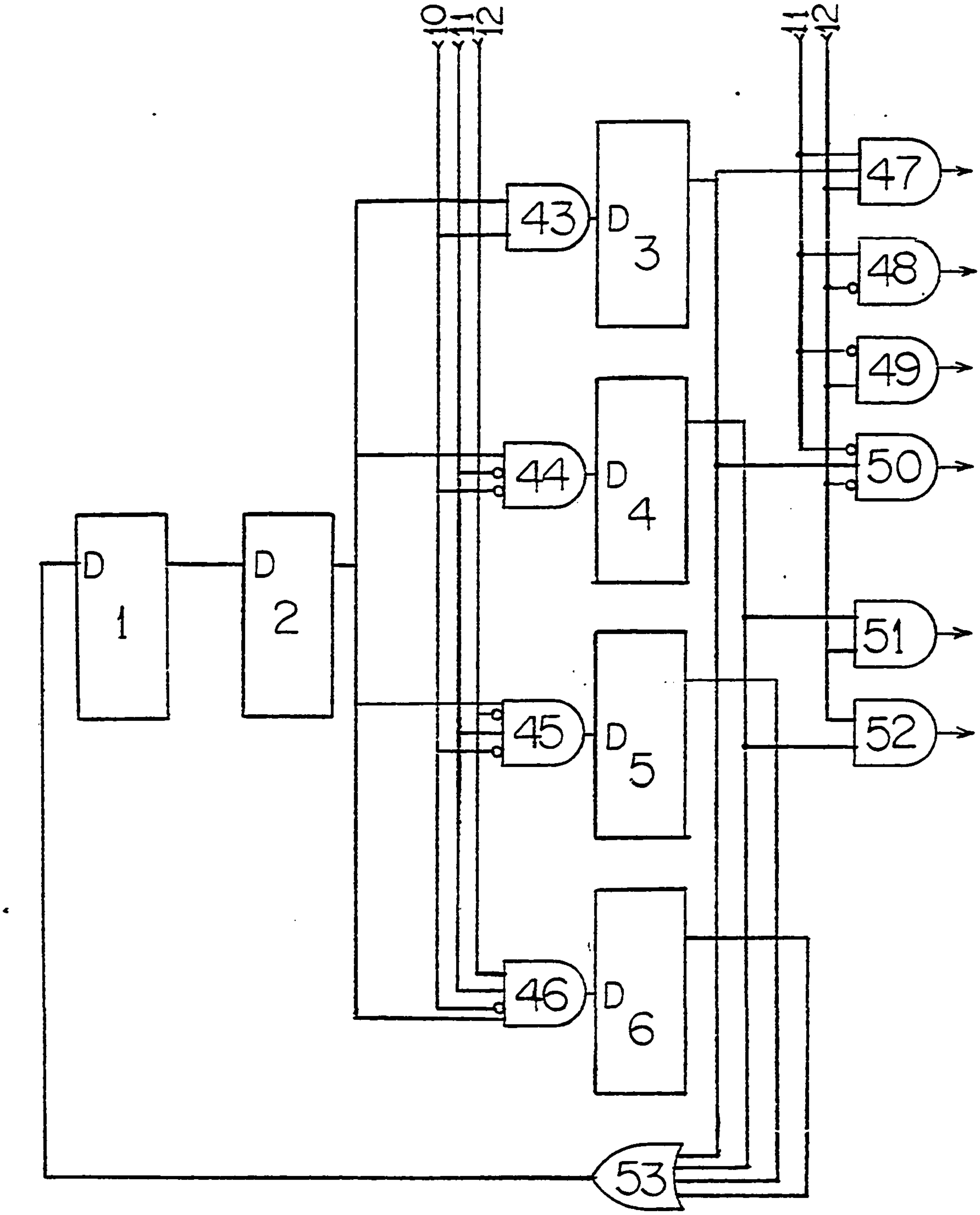


Fig. B.7 Case III: Control Circuit

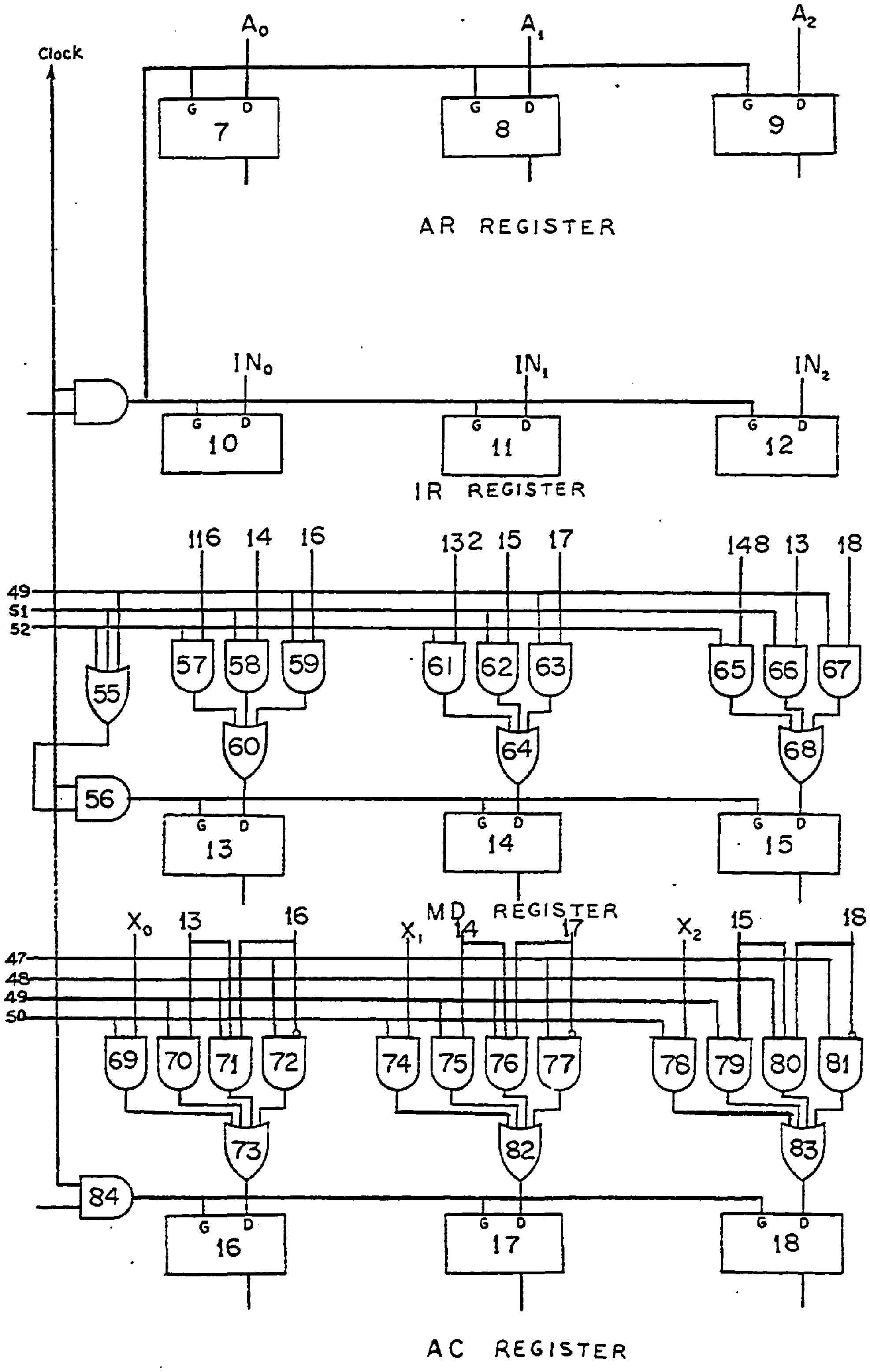


Fig. B.8 Case III: Register Circuits

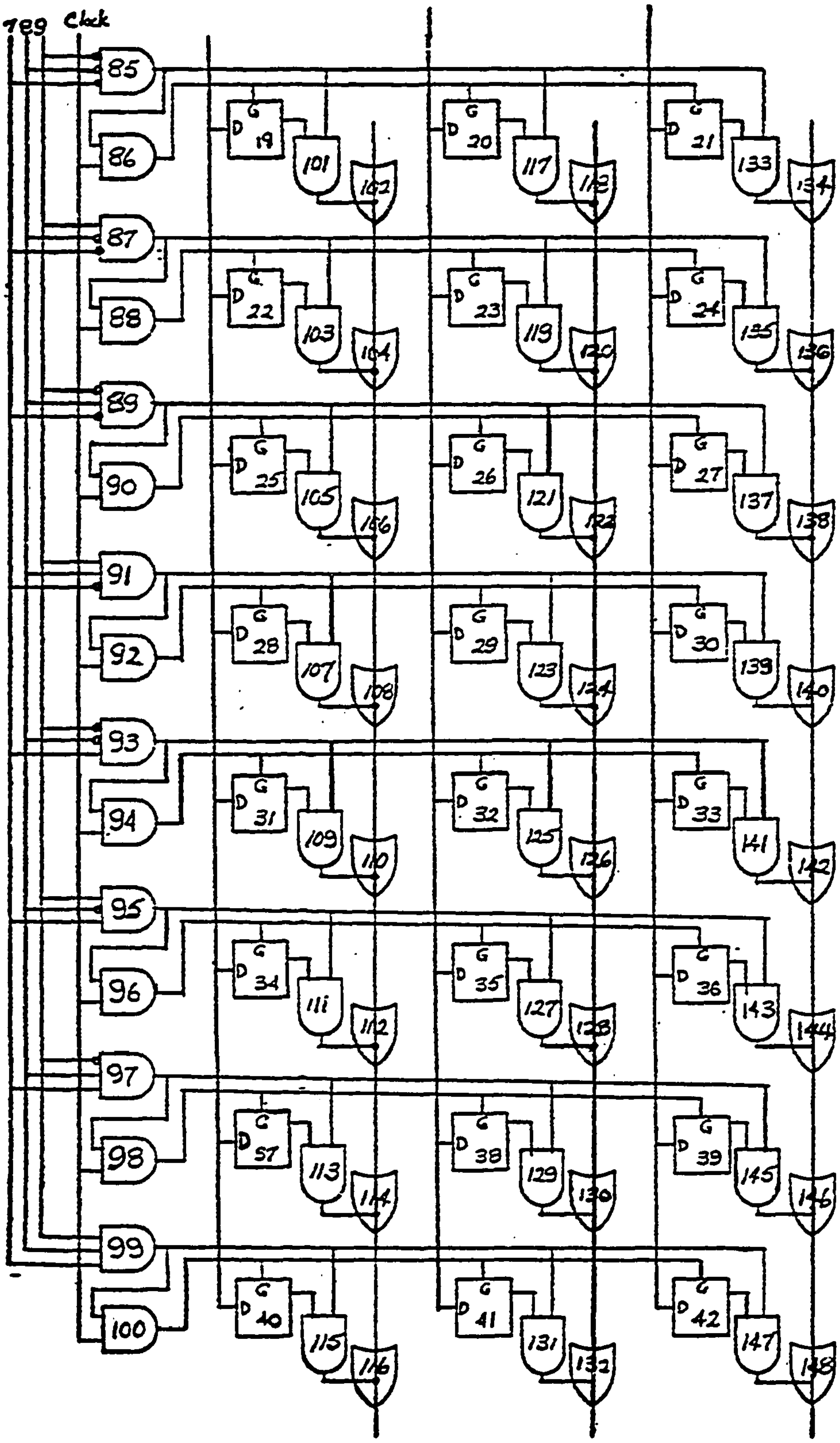


Fig. B.9 Case III: Memory Circuit

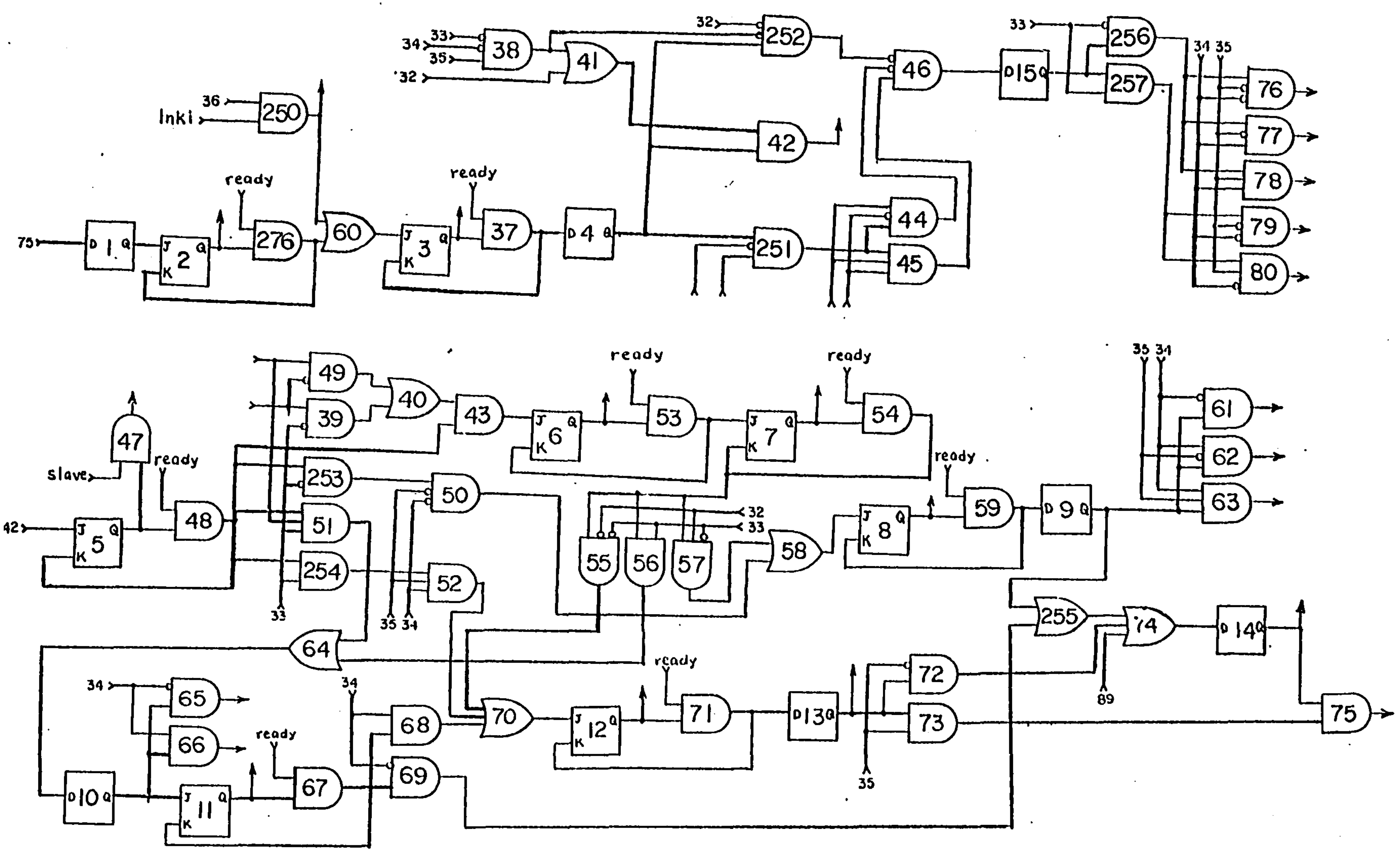


Fig. B.10 Case IV: Control Circuit

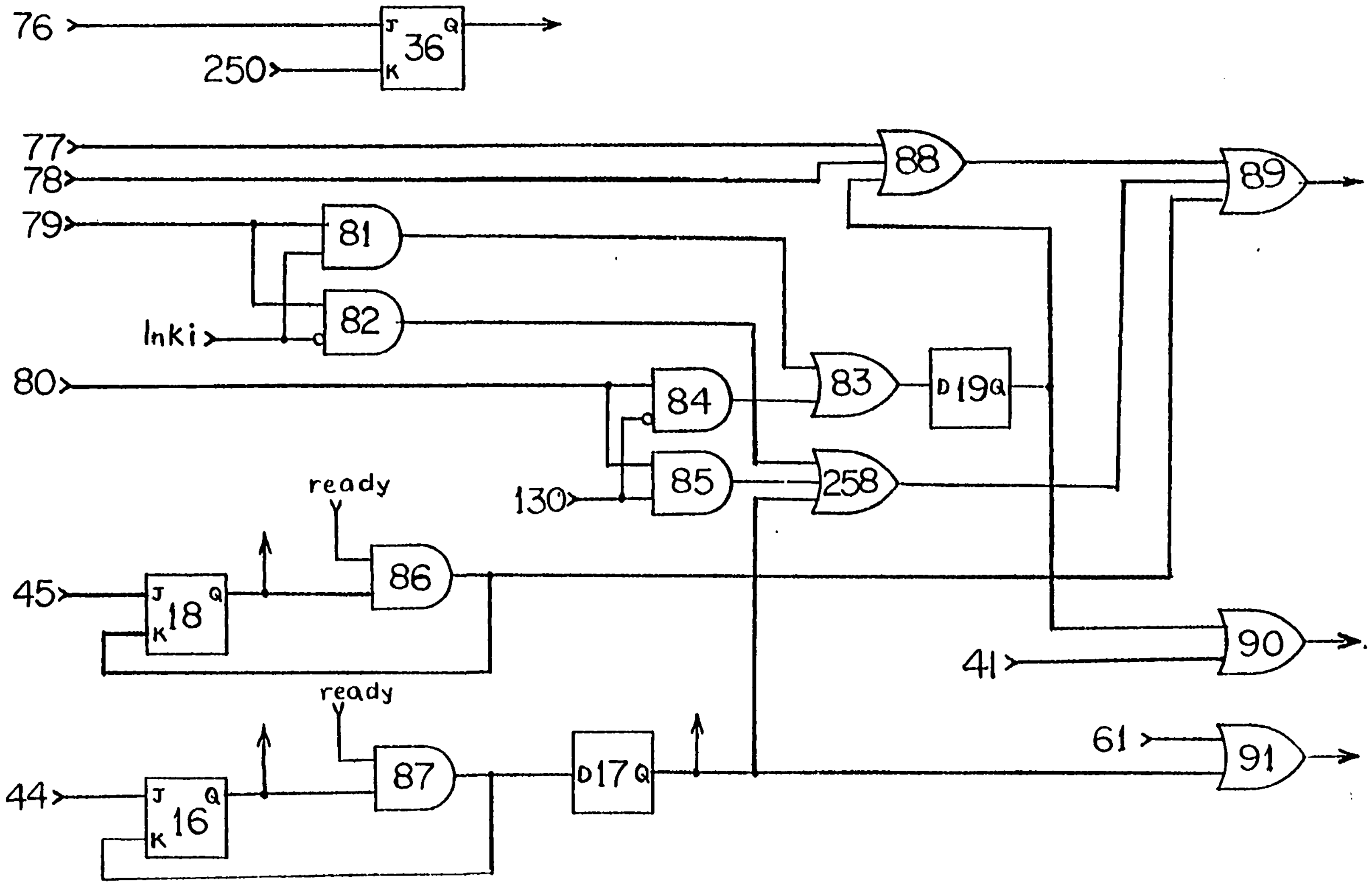


Fig. B.11 Case IV: SKIP Control Circuit

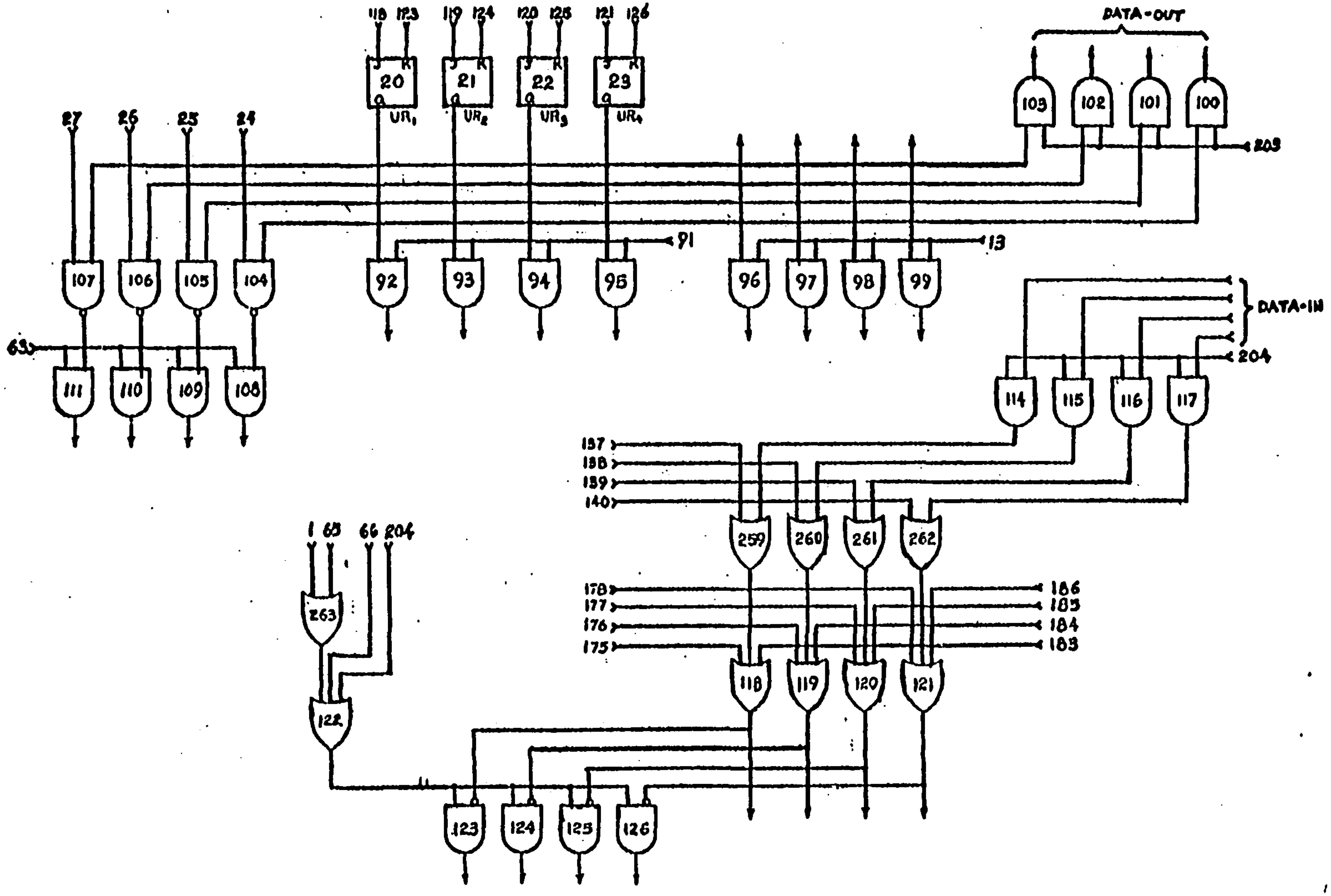


Fig. B.12 Case IV: UR Register Circuit

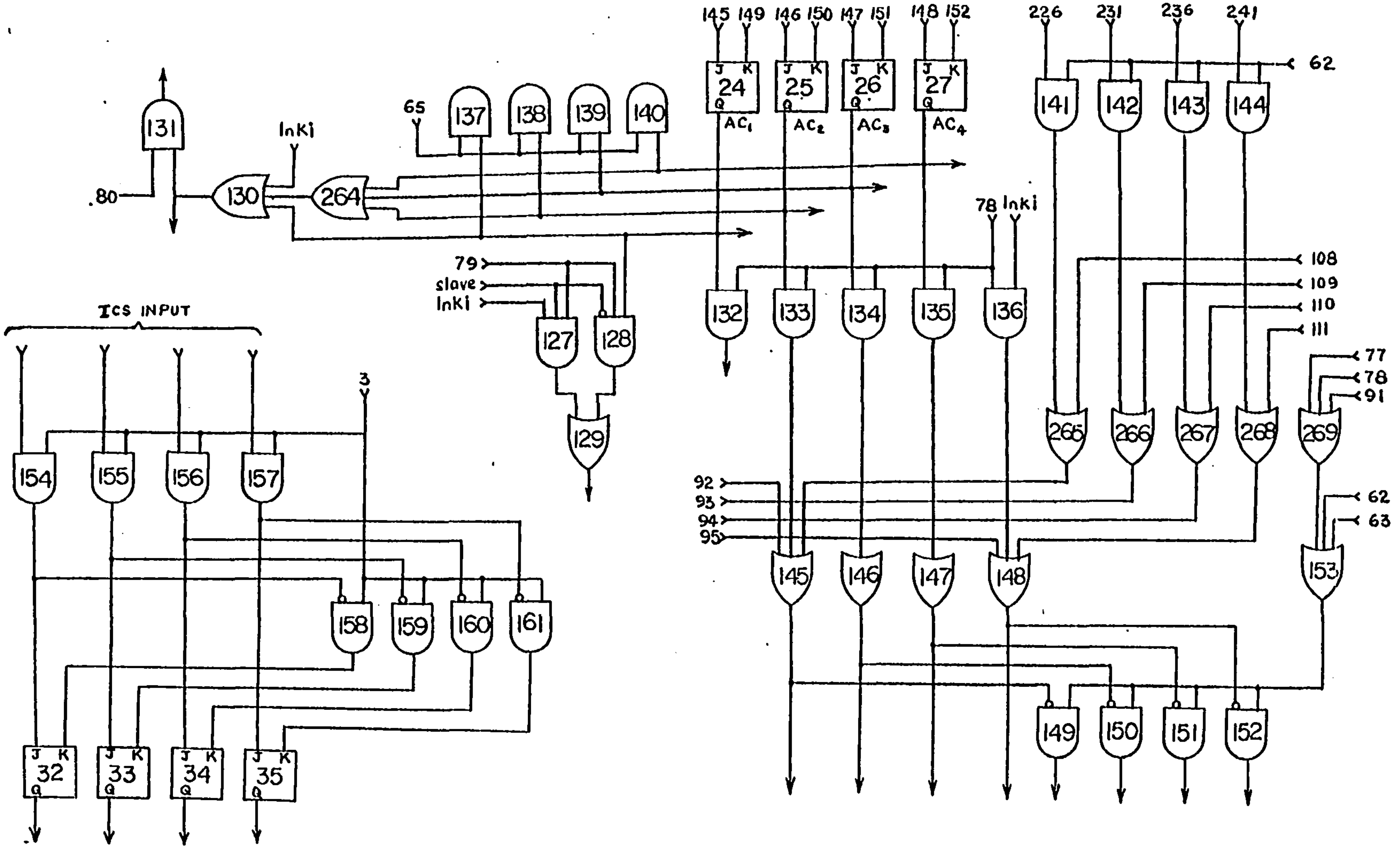


Fig. B.13 Case IV: AC and IR Registers

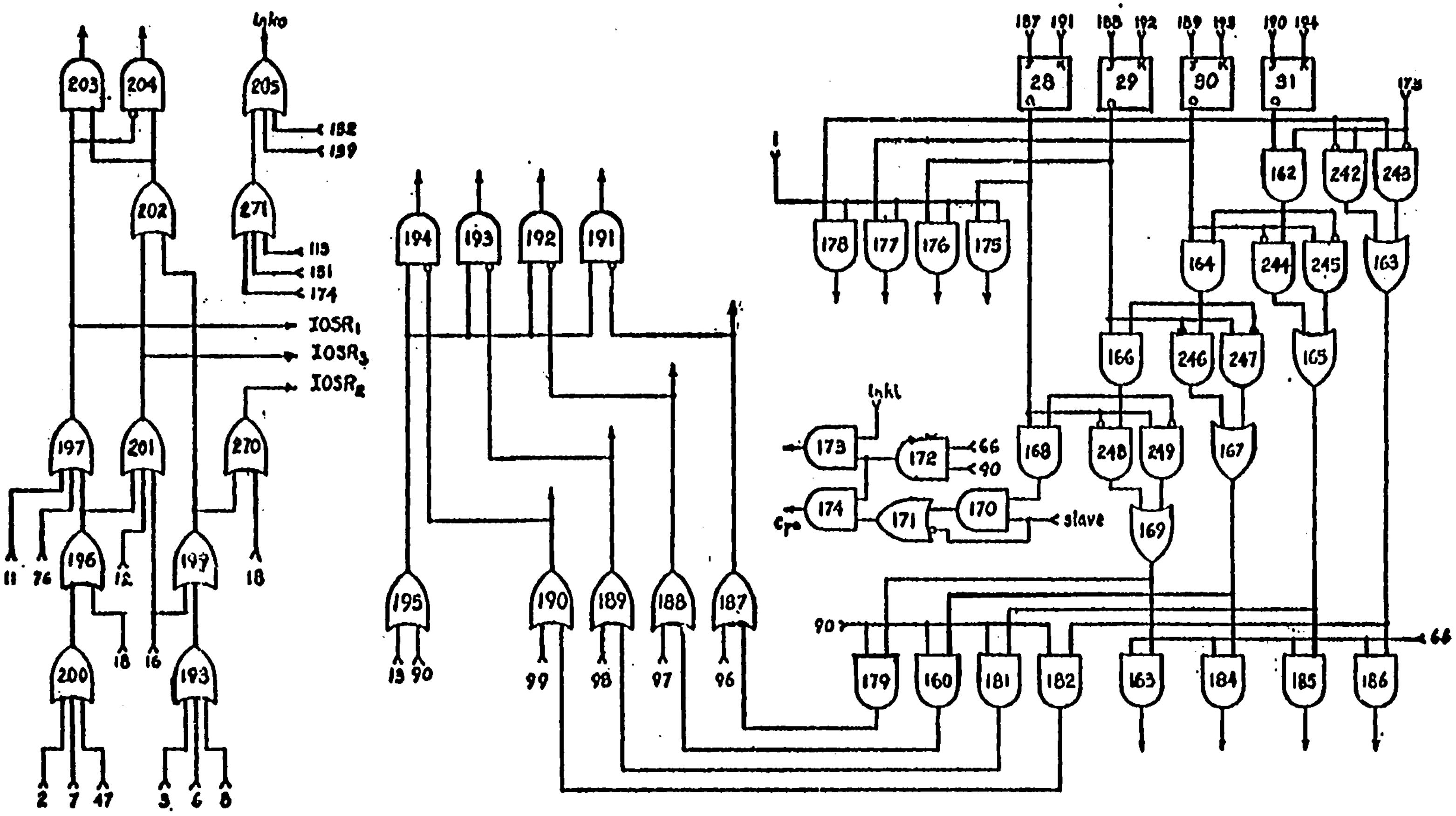


Fig. B.14 Case IV: PC Register and IOSR Circuits

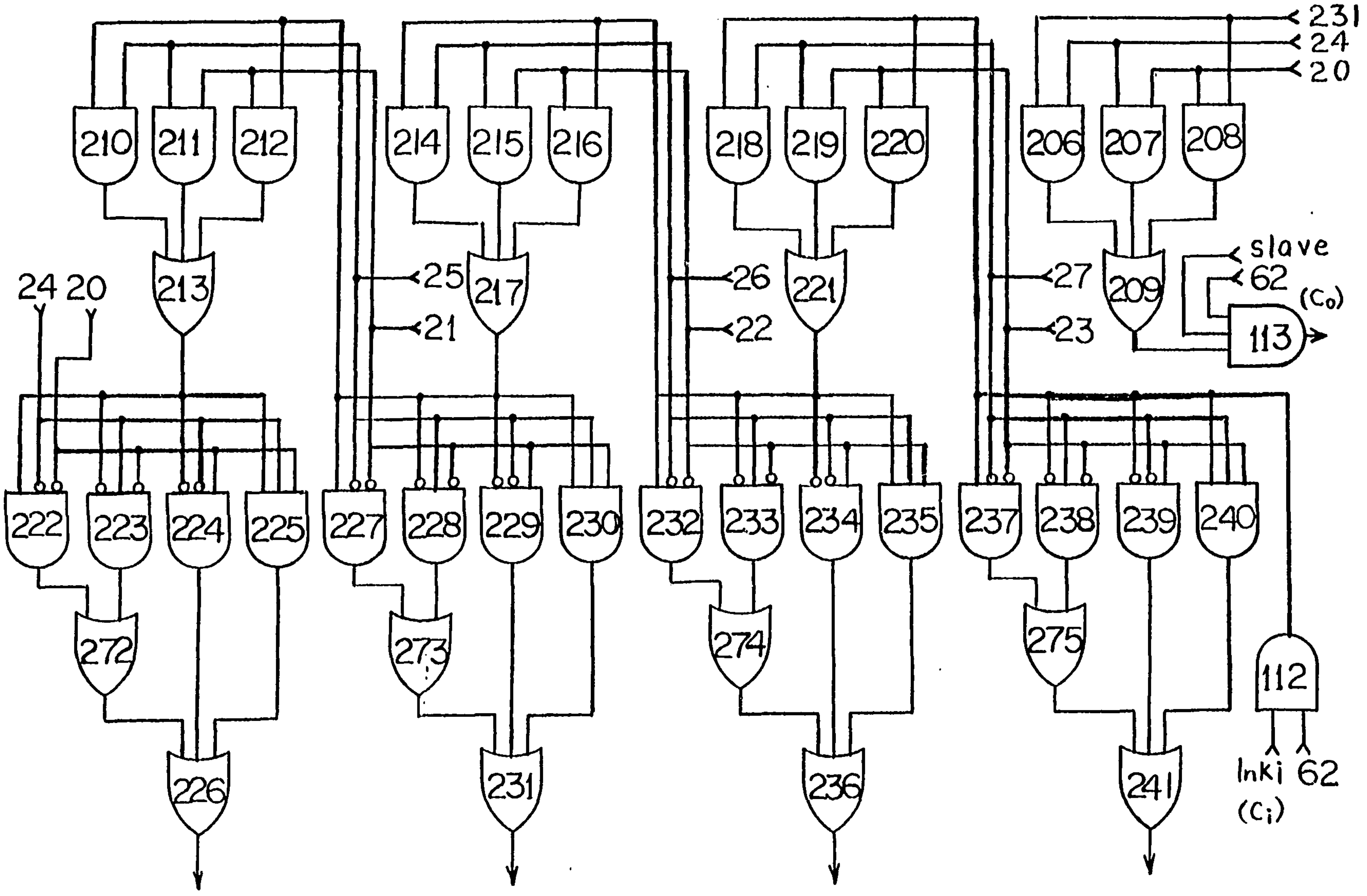


Fig. B.15 Case IV: Full Adder Circuit

LIST OF REFERENCES

Armstrong, D. B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," IEEE Transactions on Electronic Computers, EC-15 (1966), pp. 66-73.

Belt, J. E., A Heuristic Search Approach to Test Sequence Generation for AHPL Described Synchronous Sequential Circuits, Ph.D. Dissertation, Department of Electrical Engineering, University of Arizona, 1973.

Bouricius, W. G., E. P. Hsieh, G. R. Putzolu, J. P. Roth, P. R. Schneider, and C. J. Tan, "Algorithms for Detection of Faults in Logic Circuits," IEEE Transactions on Computers, C-20(11), 1971, pp. 1258-1264.

Breuer, M. A., "A Random and an Algorithmic Technique for Fault Detection Test Generation for Sequential Circuits," IEEE Transactions on Computers, C-20(11), 1971, pp. 1364-1370.

Carter, E. A., Fault Test Generation for Sequential Circuits Described in AHPL, Ph.D. Dissertation, Department of Electrical Engineering, University of Arizona, 1973.

Eldred, R. D., "Test Routines Based on Symbolic Logic Statements," Journal of the ACM, 6(1), 1959, pp. 33-36.

Estrin, G., "Diagnosis and Prediction of Malfunctions in the Computing Machine at the Institute of Advanced Study," IRE International Convention Record, Pt. 7 (1953), pp. 59-61.

Hart, P., N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Transactions on Systems Science and Cybernetics, SCC-4(2) (1968), pp. 100-107.

Hennie, F. C., "Fault Detection Experiments for Sequential Circuits," Proceedings of the 5th Annual Symposium on Switching Theory and Logical Design, 1964, pp. 95-110.

Hill, F. J., and B. M. Huey, "SCIRTSS: A Search System for Sequential Circuit Test Sequences," IEEE Transactions on Computers, Vol. C-26, May 1977, pp. 490-502.

Hill, F. J., and G. R. Peterson, Digital Systems: Hardware Organization and Design, Wiley, New York, 1978.

Hill, F. J. and B. M. Huey, "A Design Language Based Approach to Test Sequence Generation," Computer, Vol. 10, Number 6, June 1977, pp. 28-33.

Holt, A. W., et al., "Final Report of the Information System Theory Project," Tech. Report RADC-TR-68-305, Rome Air Development Center, 1968.

Huey, B. M., Search Directing Heuristic for the Sequential Circuit Test Search System, Ph.D. Dissertation, University of Arizona, 1975.

Huey, B. M., "Guiding Sensitization Searches Using Problem Reduction Graphs," Proceedings of the 14th Annual Design Automation Conference, p. 274-291.

Kime, C. R., "An Organization for Checking Experiments on Sequential Circuits," IEEE Transactions on Electronic Computers, EC-17(4), 1966, pp. 352-366.

Kubo, H., "A Procedure for Generating Test Sequences to Detect Sequential Circuit Failures," NEC Journal of Research and Development, 12, 1968, pp. 69-78.

Michie, D., and R. Ross, "Experiments with the Adaptive Graph Transverser," Machine Intelligence 5, B. Meltzer and D. Michie (eds.), American Elsevier Publishing Company, Inc., New York, 1970, pp. 301-318.

Ng, W. W., Evaluation of a LSI Fault Detection Program Using a Four-Bit Microcomputer Processor Circuit, M.S. Thesis, Department of Electrical Engineering, University of Arizona, 1974.

Nilsson, Nils J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

Petri, C. A., "Kommunikation mit Automaten," Univ. of Bonn 1962; translation by C. F. Green, Jr. "Communication with Automata," Supplement to Tech. Doc. Rep. #1, Rome Air Development Center, Contract # AF30(602)-3324, 1965.

Poage, J. F., and E. J. McCluskey, "Derivation of Optimal Test Sequences for Sequential Machines," Proceedings of the 5th Annual Symposium on Switching Theory and Logical Design, 1964.

Roth, J. P., "Diagnosis of Automata Failures: A Calculus and a Method," IBM Journal of Research and Development, 10, 1966, pp. 278-291.

Roth, J. P., W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests and Distinguish Between Failures in Logic Circuits," IEEE Transactions on Electronic Computers, EC-16(5) (1967), pp. 567-579.

Rutman, R. A., "Fault Detection Test Generation for Sequential Logic by Heuristic Tree Search," IEEE Repository Paper R-72-187, Sept.-Oct. 1972.

Schneider, P. R., "On the Necessity to Examine D-Chains in Diagnostic Test Generation - An Example," IBM Journal of Research and Development, 11(1), 1967, p. 114.

Seshu, S., and D. N. Freeman, "The Diagnosis of Asynchronous Sequential Switching Systems," IRE Transactions on Electronic Computers, EC-11(4), 1962, pp. 459-465.

Seshu, S., "On an Improved Diagnosis Program," IEEE Transactions on Electronic Computers, EC-14(1), 1965, pp. 76-79.

Van Helsland, M., Evaluation of SCIRTSS Performance on Sequential Circuits Biased Against Random Sequences, M.S. Thesis, Department of Electrical Engineering, University of Arizona, 1974.