1989

A finite state machine synthesizer

Jiuling Liu
Portland State University

AN ABSTRACT OF THE THESIS OF Jiuling Liu for the Master of Science in Electrical and Computer Engineering presented April 3, 1989.

Title: A Finite State Machine Synthesizer

APPROVED BY MEMBERS OF THE THESIS COMMITTEE:

[redacted]

Marek A. Perkowski, Chairman

[redacted]

Michael A. Driscoll

[redacted]

Mohammad Ghafarazade

This thesis presents a Finite State Machine (FSM) Synthesizer developed at Portland State University. The synthesizer starts from a high level behavioral description, in which no states are specified, and generates the lower level FSM descriptions for simulation and physical layout generation.

In this thesis the author introduces the concept of the nondisjoint state tables, which has not been defined ever before, as well as the algorithms of state minimization and state assignment with the nondisjoint state tables. He also presents a new algorithm for parallel to sequential conversion, the algorithm to avoid some redundant states assigned to the specified operations and the algorithm to provide different interface

structure between the data path and the control unit. All the above mentioned algorithms have not been applied to the other existing FSM synthesizers.

With the author's contribution, the synthesizer has following distinguished features:

1. By the multiple optimization procedures included in the synthesizer, the hardware implementation of the generated FSM will be greatly simplified.
2. The size of the nondisjoint state table will be greatly reduced compared with the conventional state table, and in consequence the computer memory space required by the state minimization program will be greatly reduced.
3. The designer can define the digital system either in parallel or sequential program statements, which will simplify the ways of the description and speed up the implementation of high level specification.

A FINITE STATE MACHINE SYNTHESIZER

by

JIULING LIU

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1989

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Jiuling Liu presented April 3, 1989.

[redacted]

Marek A. Perkowski, Chairman

[redacted]

Michael A. Driscoll

[redacted]

Mohammad Ghafarzade

APPROVED:

[redacted]

Rolf Schaumann, Chairman, Department of Electrical Engineering

[redacted]

Bernard Ross, Vice Provost for Graduate Studies

ACKNOWLEDGEMENT

I would like to thank my advisor, Marek A. Perkowski, for organizing the synthesis project and providing the framework in which I could work on such interesting research.

I would like to thank Michael A. Driscoll and Mohammad Ghafarazade for their valuable comments and corrections.

I would like to thank Shirley Clark for providing all kinds of help during my entire working period.

I would like to thank all the people in the DIADES research group for their knowledge and technical supports.

I would also like to thank the faculty and staff of the electrical engineering department for their help in knowledge and facilities.

I acknowledge funding for this research under SHARP Microelectronics Technology company grant.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER I

INTRODUCTION

As the technology advances in the area of VLSI design, even more sophisticated digital systems are emerging. This places even higher demands on the quality and sophistication of *control units (CU)* synthesized by the automatic synthesis systems. Newer and more effective ways of synthesizing and optimizing various types of such units must be found. In some automatic synthesis systems [Tsen 86], [Meye 84], [Sout 83], the control unit synthesis begins from the control flow description in which the machine's states are determined. Some systems [Pang 87] begin from the *data flow-graph,* but they can not deal with the flow-graphs of complicated controls. Other systems [Bray 88] consider the hardware performance of the operations in the *data path (DP)* only, when assigning states to those operations, but not the structure of the *control graph* itself. This produces *Finite State Machine* (FSM) realizations of control units with many redundant states. Because of the lack of the state minimization procedures, which is almost the universal case for currently existing systems, FSMs with nonminimum numbers of states are assigned and realized in logic. The consequence is that the area and speed requirements are not satisfactorily met.

This thesis describes one of the two control synthesizers used in DIADES automatic synthesis system [Perk 89] at Portland State University: the FSM Synthesizer. DIADES is a research project sponsored by the SHARP Microelectronics Technology company. It translates a high level behavioral description of a digital system in the program language ADL into the logic or hardware structure description for the lower level simulation or physical layout design. The diagram of DIADES is shown in Figure 1. The

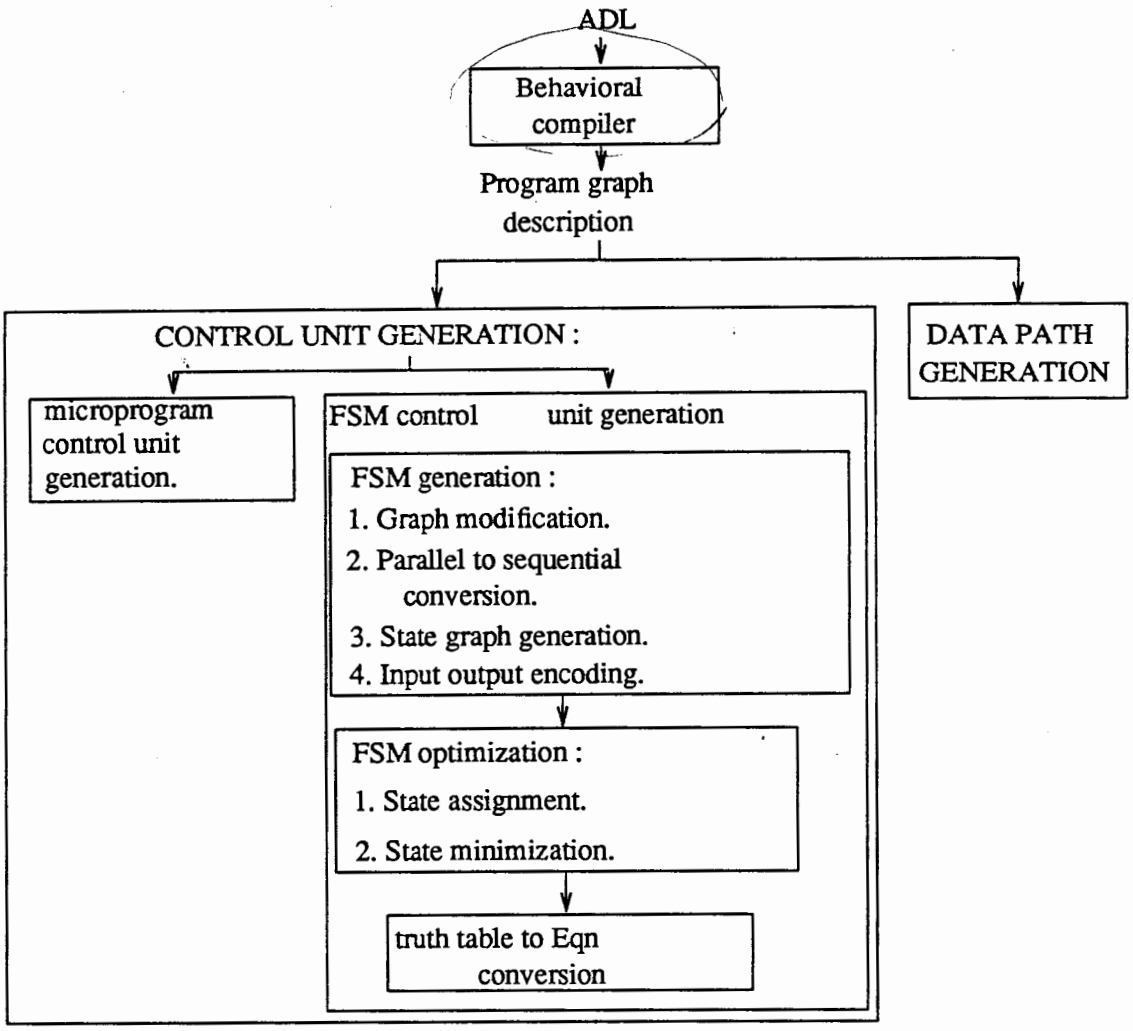


Figure 1. Diagram of DIADES system.

ADL program defines the overall function of the digital system in terms of assignment statements and control flow statements, not necessarily how it is built and what components it will use. DIADES first compiles the ADL program and generates a *program graph (p-graph)* description which includes the *control flow graph (cf-graph)* and some other information not related to this thesis. Afterwards the design task is divided into the control unit generation and the data path generation. The p-graph is the input data for both of the generations. Two choices of the control unit implementation, microprogram control unit and FSM control unit, are provided by the synthesis system. The result of microprogram control unit generation is the microcode for control memory. The result of

FSM control unit generation is the *truth table* or the *Eqn* (a set of Boolean equations) [Walt 85] data formats for the combinational part of the FSM. The Microprogrammed Control Unit Synthesizer, MiCUS, is described in [Yang 89].

The FSM Synthesizer is composed of the FSM generation stage and the FSM optimization stage. The first stage is the process to analyze the cf-graph and assign machine states to the specified operations. It also encodes the input and the output signals for FSM's. In the generation process, the user can select optimal interface structure between the control unit and the data path from different variants of design scheme. The different variants mean either encoding or not encoding the input signals from data path. That can be approached by the program graph modification. For the digital system specified by the parallel description, the corresponding cf-graph includes parallel nodes and paths. In this case a parallel to sequential conversion procedure is needed to find the operations executed in parallel and to assign states to these parallel operations.

The FSM optimization is composed of the state minimization program and the state assignment program. The state minimization reduces both the number of rows and the number of columns of state table, which is a unique feature of DIADES. This program uses state tables without the restriction that input expressions corresponding to the columns of the table should be disjoint, as it is required in most of the textbooks [Koha 82]. The state assignment program encodes machine states to simplify the hardware implementation of the FSM.

The author developed the FSM generation algorithm and program, modified the state minization algorithm and program and fixed the bugs in state assignment program. He also developed the programs to interface different parts in the Synthesizer and to convert different data formats. Totally about 4500 lines of source code were developed by him.

With author's contribution, the synthesizer has the following advantages com-

pared to other synthesis systems.

1) Since the cf-graph does not contain any state transition information, the designer need not be concerned with complicated state transition analysis when defining overall function of the digital system.

2) During the FSM generation stage, some efforts have already been made to reduce the number of the generated states, which will speed up the state minimization procedure.

3) The elimination of the restriction that the FSM input expressions should be disjoint greatly reduce the state table sizes compared with conventional state tables, and in consequence, the computer memory space required by the state minimization program is greatly reduced.

4) The designer can define the digital system either in parallel or sequential program statements, which will simplify the ways of the description and speed up the implementation of high level specification.

Designing such a large software system involves solving two kinds of problems: theoretical models of algorithms and data structures, and practical problems related to the implementation. Only the problems of the first type will be addressed in this thesis. Those interested in the second category problems can referred to the comprehensive annotated and fully commented listings of the programs introduced here as well as to other DIADES documentation.

The details of the above mentioned cf-graph description is introduced in CHAPTER II.

CHAPTER III describes the entire synthesis process of the FSM control unit generation. In § 3.1 the principles of the state partitioning are briefly discussed, which will help to understand the the basic ideas for future development for DIADES.

In § 3.2 the algorithm of state transition generation, which is the transformation from a cf-graph to a state graph description, is analyzed. As the internal data, both the *state transition format, Kiss,* and the *state table format, Stab,* of FSMs are generated.

The input and output expressions included in these formats are encoded according to the rules presented in § 3.3. The outputs of a control unit are the control signals for the register transfer operations, memory read/write operations as well as the multiplexers addressing for multiple-input register loading. The input signals to FSM can be either encoded or not encoded, according to the user's choice. The advantage of providing the two variants is demonstrated in § 3.5.

For the parallel cf-graph, the conversion of the parallel control flow to a sequential control flow is first performed before the *Kiss* format and *Stab* format creation. The theory and the algorithm for this conversion are illustrated in § 3.4.

The state tables are used as the input data for the FSM *state minimization program* [Perk 87][Zhao 89]. To our knowledge, it does not currently exist in any other design automation system. The details of the advantages of using the nondisjoint state tables for state minimization and the modified algorithm for this task are described in § 3.6.

The current state assignment program in DIADES is *FASS* [Perk 89]. The *Kiss* is the input data for this program. This is an input format to the *Kiss* program [DeMi 83a], that can be used instead of *FASS*. The confirmation that the modification of the nondisjoint input expression does not affect the state assignment results is presented in 3.7.

In CHAPTER IV, the reader will obtain a concept of the entire performance of the synthesis, with the overview of the synthesis structure as well as the important procedures involved.

CHAPTER V will help the user to be familiar with all the internal and external data formats in the synthesis process and, consequently, to feel comfortable in developing future synthesis programs for DIADES.

CHAPTER VI, together with APPENDICES A and B, presents the design procedures of two examples which are the Eight-instruction CPU design and the Signal Delay Processor design. The first example illustrates the design with encoded instructions and the second example demonstrates the design with the parallel control flow description.

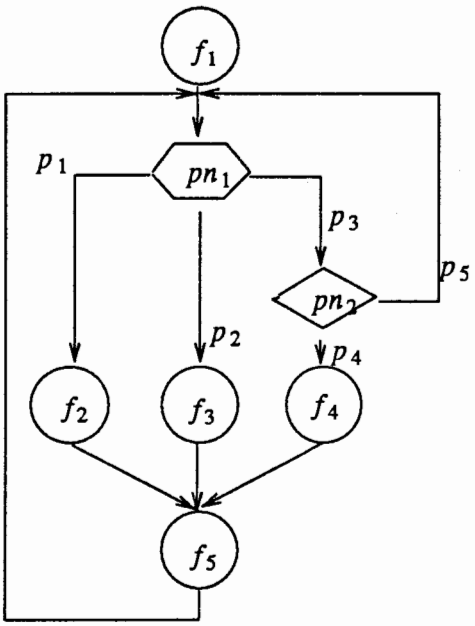
The statistics of twelve design examples are presented in CHAPTER VII. The suggestions of future developments are also proposed in this chapter.

CHAPTER II

CONTROL FLOW GRAPH

§ 2.1 CONTROL FLOW GRAPH DEFINITION

Control flow graph (cf-graph) is an execution flow graph of the circuit under design. It describes the sequence of operations, as well as the relations between the operations. One example of this kind graph is shown in Figure 2. In DIADES such graphs are represented as components of the description in language *GRAPH88* [Yang 89], [Perk 82]. The graph is composed of the following items:



f_1 : a = 0, b = 0, c = 0;
f_2 : a = a + 1;
f_3 : b = b + 1;
f_4 : c = c + 1;
f_5 : flag = input;

pn_1 : p_1 - flag = 1;
p_2 - flag = 2;
p_3 - flag = 0;
pn_2 : p_4 - stop;
p_5 - (not stop).

Figure 2. Example of a program graph.

- *A set of operational nodes* - $F = \{f_1, f_2, \cdots, f_n\}$. Each operational node

specifies a set of operations to be performed in this node within one machine cycle. In Figure 2, $F = \{f_1, f_2, f_3, f_4, f_5\}$. f_1 specifies the operations $a := 0$, $b := 0$ and $c := 0$, f_2, f_3, f_4 and f_5 specify the operations $a := a + 1$, $b := b + 1$, $c := c + 1$ and *flag* := *input* respectively.

A set of predicate nodes - $PN = \{pn_1, pn_2, \cdots, pn_m\}$. Each predicate node specifies a set of *predicates,* denoted by $P_i = \{p_{i1}, p_{i2}, ..., p_{ik}\}$. A predicate can be a relation or its complement, a variable or its complement, or a Boolean expression of the relations and variables. In Figure 2, PN = $\{pn_1, pn_2\}$. The predicate node pn_1 specifies predicates p_1, p_2 and p_3 which represent the relations *flag* = 1, *flag* = 2 and *flag* = 3, while the node pn_2 implies predicates p_4 and p_5 which represent the logic variable *stop* and its complement.

A set of arcs - $A = \{a_1, a_2, ..., a_n\}$. The arcs direct the control flow of the cf-graph. An arc is a triple of the form (p_i, psn, nxn) . Here p_i represents a predicate which determines the existence of the arc, *psn* is the head node that the arc points from and *nxn* is the tail node that the arc points to. Both *psn* and *nxn* can be either an operational node or a predicate node. The actual meaning of this representation is *"if p_i, go from psn to nxn."* For the graph in Figure 2, the arc from operational node f_1 to predicate node pn_1 is represented by $(1, f_1, pn_1)$, the arc from predicate node pn_2 to pn_1 is $((not\ stop), pn_2, pn_1)$.

The *immediate successor* of an operational node is one of its successor nodes which is linked by an arc or a set of arcs and predicates. In Figure 2, the nodes f_2, f_3 and f_4 are immediate successors of node f_1. The node f_1 is also an immediate successor of itself. The control flows from an operational node to its immediate successor when all the predicates along the path between them are satisfied. It is, therefore, the Boolean *AND* of all the predicates on the transition leading path. This Boolean expression is defined as the *transition predicate* which is denoted by tp_k. In Figure 2, the transition

predicate leading the transition from operational node f_1 to the node f_4 is $p_3 p_4$.

Two predicates are *disjoint* when they can not be satisfied at the same time. It can be defined as follows:

Definition 1 : Two predicates are *disjoint*, denoted by $\mathrm{Dis}(p_i, p_j) = 1$, if $p_i \cap p_j = 0$.

In a program graph, all the predicates included in one predicate node are disjoint and the Boolean *OR* of them has a logic value 1, which guarantees correct and sufficient control flow information. For instance, in Figure 2, predicates p_1, p_2 and p_3 are disjoint since they are specified by the same predicate node pn_1. From an operational node of the graph, there is only one arc pointing to another node.

The *sequential cf-graph* is the graph in which the control flow, starting from a single *initial node*, will keep transferring from one node to another single node. The *transition function* is defined as the control flow from one operational node to its immediate successor under certain transition predicate. For the sequential graph it can be defined as

$$f_{sj} = Q(f_{pri}, tp_k)$$

where $f_{sj} \in F$, $f_{pri} \in F$ and f_{pri} represents present operational node while f_{sj} represents one of its immediate successors. For example, the transition function defining the transition from node f_1 to node f_4 in Figure 2 is

$$f_4 = Q(f_1, p_3 p_4)$$

One of the important features of DIADES is the existence of *parallel behavioral specifications*. To represent this kind of a p-graph (cf-graph), called a *parallel p-graph (cf-graph)* some special control nodes are required.

The first control node is *FORK* which means that the control from the current node is concurrently transferred to the several nodes pointed by all the arcs starting from it. It can be illustrated by the example from Figure 3. In this example, after node *FORK*

the predicates *a* and *b* are parallelly evaluated. With the different combination of predicates the different combinations of operational nodes will be approached. In this example four different combinations are involved and each combination is composed of two operational nodes. The distinguishing feature of the *parallel branch* in this *parallel forking (forking)* is that the predicate linking the node *FORK* to one of the nodes f_1 and f_2 is *nondisjoint* to the other predicates linking the *FORK* to any one of the nodes f_3 and f_4.

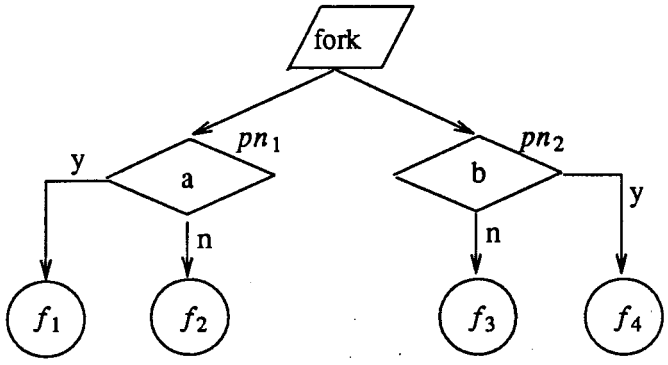


Figure 3. Parallel branches starting from *FORK*.

The second control node is *DAND* which is an end of parallel branches *(coincidence join)*. Only when all the branches coming to *DAND* have been arrived, this node is passed. An example to illustrate this is presented in Figure 4 (a). In this figure, the nodes pn_1 and pn_2 are parallel. f_3 will be executed only when both pn_1 and pn_2 are satisfied. Its equivalent serial description is in Figure 4 (b). In comparison, it can be seen that the graph in Figure 4 (a) is a more compact description than the one in Figure 4 (b). With *DAND* node, the deadlock, which is the endless loop, might happen in certain situation. The current Synthesizer can not check out this problem so that the user have to do some analysis to avoid it.

The third control node is *DEXOR* which is a dual case to *DAND*. This node is passed whenever any one of its inputs, instead of all the coming branches, is reached by the control. In this case all other controls that were parallel to this control under the

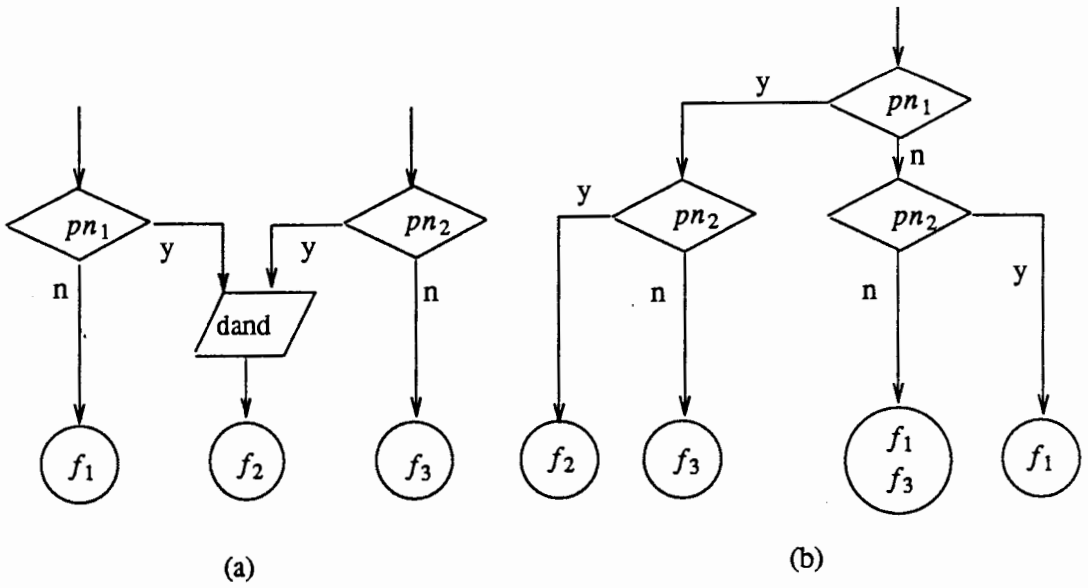


Figure 4. Parallel branches merged with *DAND* and its equivalent sequential graph.

same *FORK* are terminated. The example of the graph with *DEXOR* node and its equivalent serial graph is shown in Figure 5.

The next control node, *DROP*, locally terminates one control path of a graph. For instance, in one of three parallel branches of a *FORK*, where only two of them are joined with a *DAND*, the another one might be terminated by a *DROP*.

The final control node is *STOPADL* which globally terminates the execution of the graph, removing all existing controls, when it is reached.

The transition function for parallel cf-graph is

$FS_n = Q\,(FP_m,\ tp_k)$

where $FS_n \subseteq F$, $FP_m \subseteq F$ and FP_m is the set of present operational nodes while FS_n represents the set of immediate successors. This equation can also be used for serial program graph since sequential graph is a special case in which both FS_n and FP_m are composed of only one element.

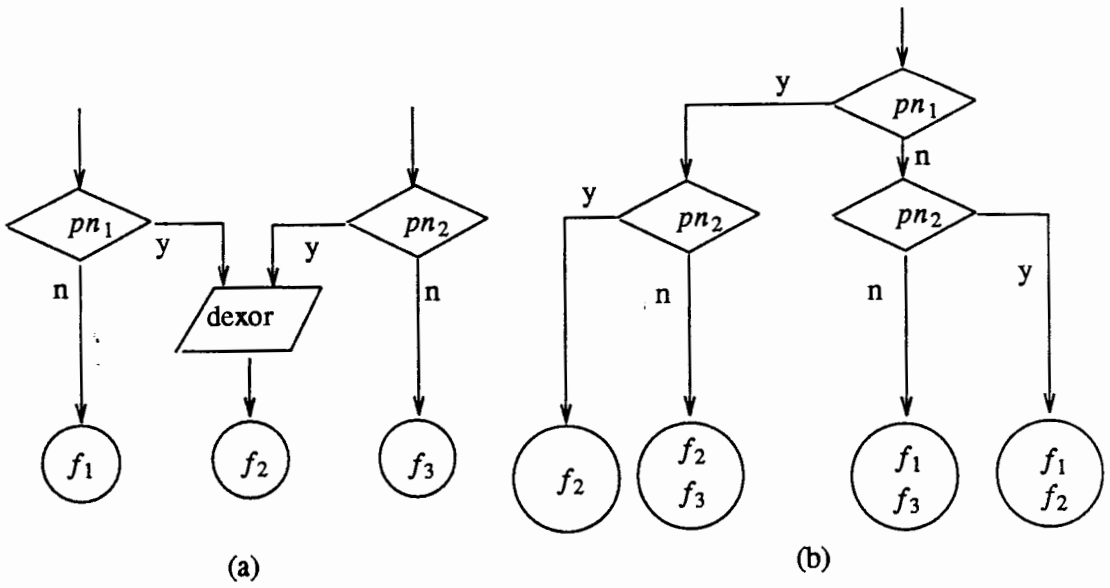


Figure 5. Parallel branches merged with *DEXOR* and its equivalent sequential graph.

§ 2.2 THE DATA FORMATS REPRESENTING CF-GRAPH IN DIADES

In DIADES the cf-graph is represented by a number of lists, specified description of which is as follows:

*A list of arcs (contained in *coplisset*)*

This list contains all of the arcs in the graph. For example, the element (x 3 13) in the list means that an arc exists between the operational node 3 and node 13; (e 3 13) also specifies the arc between node 3 and node 13 but node 3 is a control node or the node performing some combinational logic functions which dosen't need one machine cycle to delay; (6 3 4) indicates that node 3 is a predicate node and if the predicate 6 specified by node 3 is fulfilled the arc will point to node 4, while ((not 6) 3 5) corresponds to the failure of the predicate.

*A list of operational nodes (contained in *nalisset*)*

This list specifies all the operational nodes as well as the functions specified by each node. For example, (5 (:= y a)) means that node 5 specifies the register

transfer operation $y := a$, (6 (:= a (a + 1))) indicates that in node 6 a register increment operation will be executed, while (7 (5 6)) means the operations specified by both node 5 and 6 are contained in node 7.

*A list of descriptions of node properties (contained in *nolisset*)*

The property of each node is described in this list. Each element in this list has the format as either (cond *number* nil) or (*number number* nil). For example, (cond 8 nil) means that node 8 is a predicate node; (5 5 nil) and (5 12 nil) indicates that both node 5 and 12 are operational nodes and specify the same operations as in node 5. The meaning of the last element (nil) is irrelevant at this point.

*A list of predicates (contained in *plisset*)*

In this list each element represents a predicate number as well as the predicate that it specifies. For two complement predicates, only one is specified in this list. Whenever referring another one, the complement of the predicate number is used. For example, (10 (lessp x 20)) means that the number 10 predicate implies the relation "$x < 20$". The predicate "$x \geq 20$" is referred by the expression (not 10).

*A list of node groups (contained in *anlisset*)*

This list groups all the nodes of the same property. All the predicate nodes are considered to have the same property and those nodes of the same operation are included in one group. For example, (cond (3 4 6 8 10)) includes all of the predicate nodes of the graph; (5 (5 12)) groups two nodes of the same operation specified by node 5.

*A list of memory variables (contained in *lzmset*)*

All the memory variables, either the register or memory types of the graph are put in this list. For example ((1 (y j))) means that in system 1 there are two register variables y and j.

Each of *coplisset*, *nalisset*, *plisset*, *anlisset* and *lzmset* contains a set of lists which describes a set of the cf-graphs. The graphs are identified by the graph number. Each of the sets are of the format as

((1 (*list of description*))(2 (*list of description*)) ···).

The number before the list of description is the graph number.

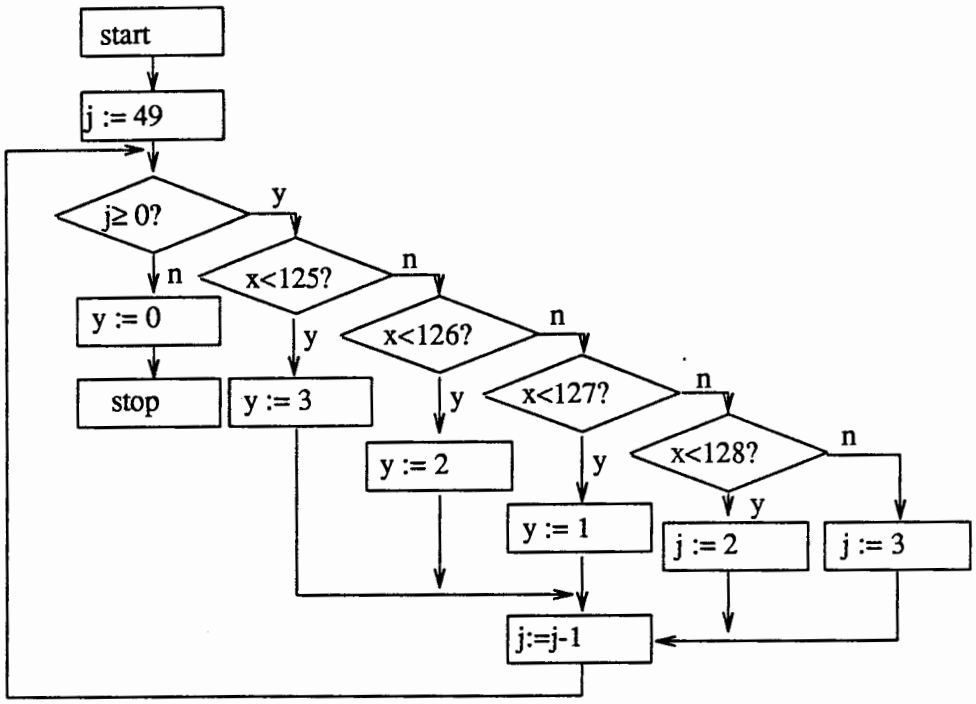


Figure 6. Flowchart example.

For the flowchart example shown in Figure 6, the corresponding program graph, which is shown in Figure 7, is described by the following LISP statements which specify a number of lists. Instead of the operation and control statements in the flowchart, the program graph contains the operational nodes represented by circles and predicate nodes represented by the diamonds. The actual meaning of each node is specified by the lists stored in the lists under the names *nalisset* and *plisset*. For example, the node 2 in Figure 7 corresponds to the statement *"j := 49"* in the flowchart, which is specified by the element *(2 (:= j 49))* from the list stored in *nalisset*. The node 3 in Figure 7

corresponds to the control statement *"j ≥ 0 ?"* in flowchart, which can be referred by the element *(3 (lessp 0 j))* from the list stored in *plisset*. The difference is that the later expression is specified by the LISP statement. So is the operation statement *(13 (:= j (plus j (minus 1))))* which specifies the operation *"j := j - 1"*. The control flow in the program graph is specified by the arc list stored in *coplisset*. The control flow from the predicate node 3 is either to the operational node 14, specified by *((not 3) 3 14)* , or to the predicate node 4, specified by *(3 3 4)*. Since this program graph corresponds to only one flowchart, each list specified by the LISP statement contains only a single list with the graph number 1.

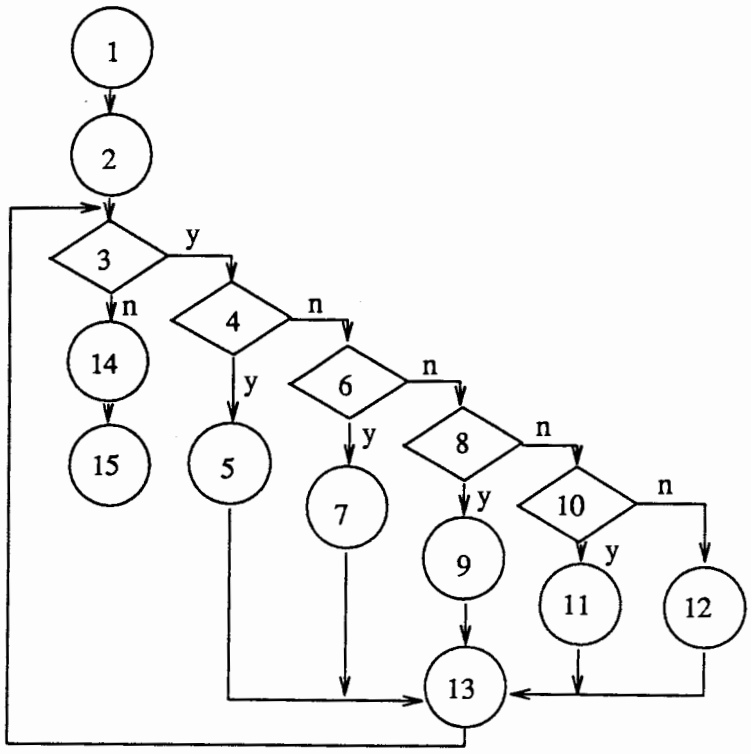


Figure 7. Program graph description for the flowchart in Figure 6.

```
  (setq *coplisset* '
((1
  ((x 14 15)
   ((not 3) 3 14)
   (x 13 3)
```

```
  (x 5 13)
  (x 7 13)
  (x 9 13)
  (x 11 13)
  (x 12 13)
  ((not 10) 10 12)
  (10 10 11)
  ((not 8) 8 10)
  (8 8 9)
  ((not 6) 6 8)
  (6 6 7)
  ((not 4) 4 6)
  (4 4 5)
  (3 3 4)
  (x 2 3)
  (x 1 2)))) )

  (setq *nolisset* '
((1
 ((stopadl 15 nil)
  (14 14 nil)
  (13 13 nil)
  (5 12 nil)
  (7 11 nil)
  (cond 10 nil)
  (9 9 nil)
  (cond 8 nil)
  (7 7 nil)
  (cond 6 nil)
  (5 5 nil)
  (cond 4 nil)
  (cond 3 nil)
  (2 2 nil)
  (start 1 nil)))) )

  (setq *nalisset* '
((1
 ((14 (:= y 0))
  (13 (:= j (plus j (minus 1))))
  (9 (:= y 1))
  (7 (:= y 2))
  (5 (:= y 3))
  (2 (:= j 49))))) )

  (setq *plisset* '
((1
 ((10 (lessp x 128))
  (8 (lessp x 127))
  (6 (lessp x 126))
  (4 (lessp x 125))
  (3 (lessp 0 j))))) )

  (setq *anlisset* '
```

```
((1
  ((stopadl (15))
   (14 (14))
   (13 (13))
   (5 (5 12))
   (7 (7 11))
   (cond (3 4 6 8 10))
   (9 (9))
   (2 (2))
   (start (1))))) )

   (setq *lzmset* '
((1 (y j))) )

   (setq *symlis* '
((c 1)) )
```

CHAPTER III

FSM DESCRIPTION GENERATION

Control synthesis is a process to generate a control unit (CU) description according to the control and operational flow description. Control synthesis consists of *operation partitioning (control allocation), state synthesis* and *logic synthesis.* In DIADES, the state minimization is also the part of the state synthesis. In this chapter we will briefly discuss the basic idea of operation partitioning since DIADES currently has no program to perform this task. The state synthesis will be discussed comparatively in detail. The details of the logic synthesis can be referred to the related materials [Wu 88][Hell 88][Cies 89].

§ 3.1 OPERATION PARTITIONING

Operation partitioning assigns (partition, decompose) algorithm operations into machine cycles. In the high level description, operations are usually presented in algorithms as assignment statements such as $a := b + c * d$. According to different structures of data paths and length of machine cycles the operations described by the algorithms have to be partitioned in order to enable each group of operations to be performed in a single machine cycle.

In the design process, different choices of the data path structures have to be made due to the area, speed and power consumption *constraints*. The choices are sometimes made between parallel and serial structures of DP for operation performance [Pang 87]. The parallel structure needs comparatively more area but less execution time and the serial one needs less area but more execution time. The choices are the trade-offs

between the area and the speed. For example, the assignment $a = b + c + d + e$ can be either carried out by one adder or by three adders as shown in Figure 8 (a) and (b). Considering the one adder case, it takes four machine cycles to perform this assignment, while three adders need only a single machine cycle (assuming that the addition can be executed in 1/3 machine cycle). The speed is gained at the expense of more silicon area.

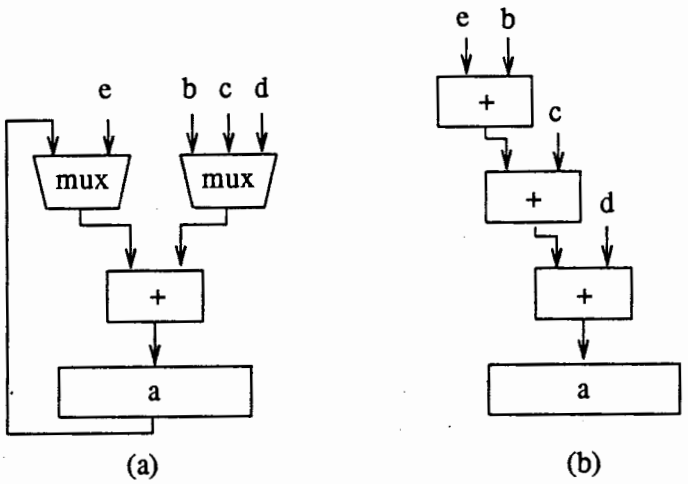


Figure 8. Example of data paths of different structures.

Operation partitioning follows the data flow restrictions [Bray 88]. Each operation as well as each data transition between operations takes a certain delay time. Therefore, in one clock cycle only a given number of operations can be performed. The number of operations grouped in one state depends on the total delay time of the operations. Whenever a delay time exceeds the time limit for one machine state, an additional operation node has to be assigned. For example, if the time delay for the sequence of three adders in Figure 8 exceeds one machine cycle, another state cycle has to be involved to carry out this operation.

The partitioning should also consider the performance difference between operations of the different speeds. For example, the input of the operation can not be changed before the operation is done; while the outputs of some fast operators are fed into slower

operators, they have to be stored in memory units so that additional states should be added. This can be illustrated by the example shown in Figure 9. The circuit in Figure 9 (a) should realize the assignment $d = a * b + c$. The multiplication operation takes more time to perform than the addition operation. Before the multiplication is finished, any changes of b or c will cause a wrong result. With another register added, the circuit is modified as in Figure 9 (b). For this structure, an extra state cycle has to be introduced to load the result of the $b + c$ to the register d. During the state cycle of multiplication, the changes of b or c have no effects on the assignment result.

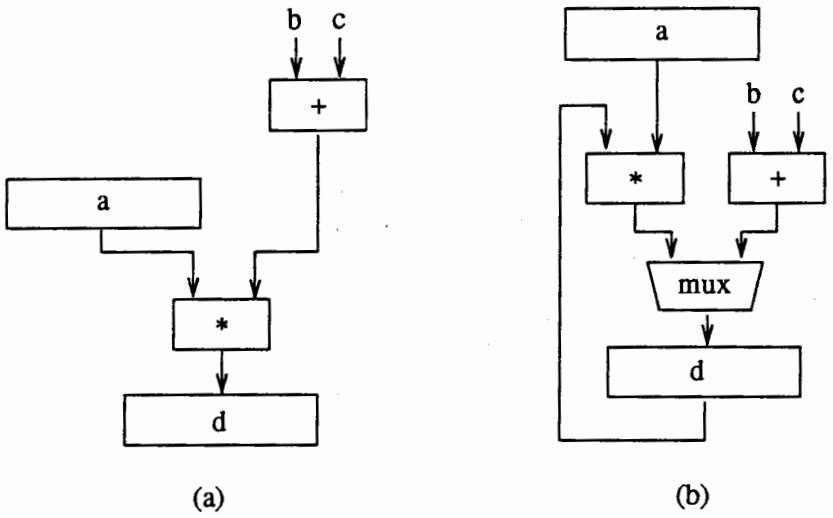


Figure 9. Connections of operations of different speed.

In DIADES the initial sequence of operations is specified by the designer. Each algorithm needs one machine cycle to be executed. The system first generates an initial version of the data path and control unit descriptions. Later on, system will make the transformation of the DP architecture to satisfy the specified constraints. This approach permits the designer to specify, for the implementation of the data path, the components as macros that are not the reserved operators acquainted by the compiler.

The different choices of the data path structures are reflected on the program graphs, which are the results of the transformations from one cf-graph to another. The

time delay of each operational node on each of the different program graphs is the same. The transformation consists of the regrouping, decomposing and merging of the operational nodes.

§ 3.2 STATE GRAPH GENERATION

In DIADES the state transition description is represented by *Kiss* [DeMi 83a] format and *state-table* format [Koha 82]. These two formats represent the *state transition graph*. The state transition graph specifies a set of the *present states*, a set of the *next states*, a set of the *input expressions* and a set of the *output expressions*. The items in these four sets are one-to-one correspondents. The four corresponding items from the four sets specify the state transition from the given present state to the given next state if the specified input expression is satisfied. During the next state the FSM will give the output control signals specified by the output expression. The generation of the state transition description is actually the generation of the state transition graph (state graph) which is transformed from the program graph (either original or transformed).

In different design automation systems, the state synthesis starts from different types of graphs. *Yorktown* silicon compiler [Bray 88] starts from a graph similar to the graph introduced in CHAPTER II, except there is no restriction of one machine cycle for each operational node. In the transformation of this graph to the state graph, each loop on the graph is initially assigned one state and then additional states are added according to the restrictions introduced in operation partitioning. Unfortunately, some obvious redundant states are produced by this approach, which will be indicated later on in this chapter. The system proposed by [Pang 87] starts with a flow-graph shown in Figure 10. In state synthesis, this graph is sliced into different levels and each level corresponds to one machine state. For the complicated controls, however, the states can not be divided into levels. For example, if there are two operations in the same level. Under different condi-

tions only one of them will be executed and after execution it will transfer to the different next operation. In this case these two operation can not have the same state eventhough they are in same level.

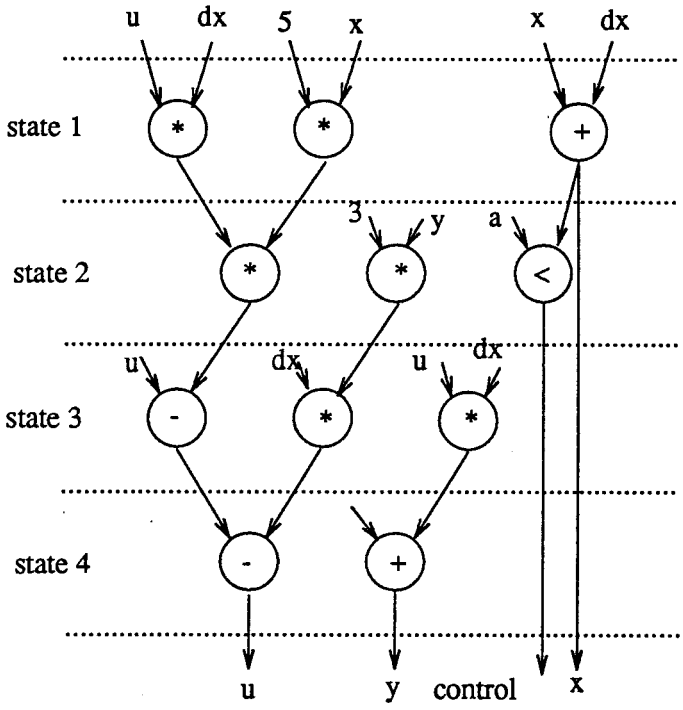


Figure 10. Example of operations slice.

Since in the cf-graphs of DIADES, each operational node needs one machine cycle to execute, the cf-graph to state graph transformation can be directly done by mapping each operational node to one state. The *input of the state transition* of the state graph is the *transition predicate* along the path from one operational node to another one of its immediate successors [see CHAPTER II]. The output is the control signal needed to enable the operations implied by the immediate successor. This kind of conversion does not make any possible reduction of the state graph states and leaves all the work of the state minimization to a future state minimization program. In DIADES there is a program called FMINI [Zhao88] which does both row and column minimization for state table description of incompletely specified FSM. However, some state reductions will be

done comparatively faster on the p-graph level than on the state table level where more comparisons would be needed.

For a FSM description of m states and n input expressions, one state reduction in the state graph generation stage will save (m - 1) row comparisons in the state table level minimization. Each row comparison involves the comparisons of n pairs of next states as well as n pairs of outputs.

Two operational nodes on the cf-graph can be assigned the same states when they come to the same predicate node or operational node, as shown in Figure 11. In this figure, predicate node sn_k and operational node f_k are called *merging nodes* which is the node pointed by more than one operational nodes. The state assigned to f_i and f_j are called the *previous state* of the merging node, which is the state assigned to the operational nodes pointing to a common merging node. In the *Yorktown* silicon compiler, these two nodes are always assigned different states.

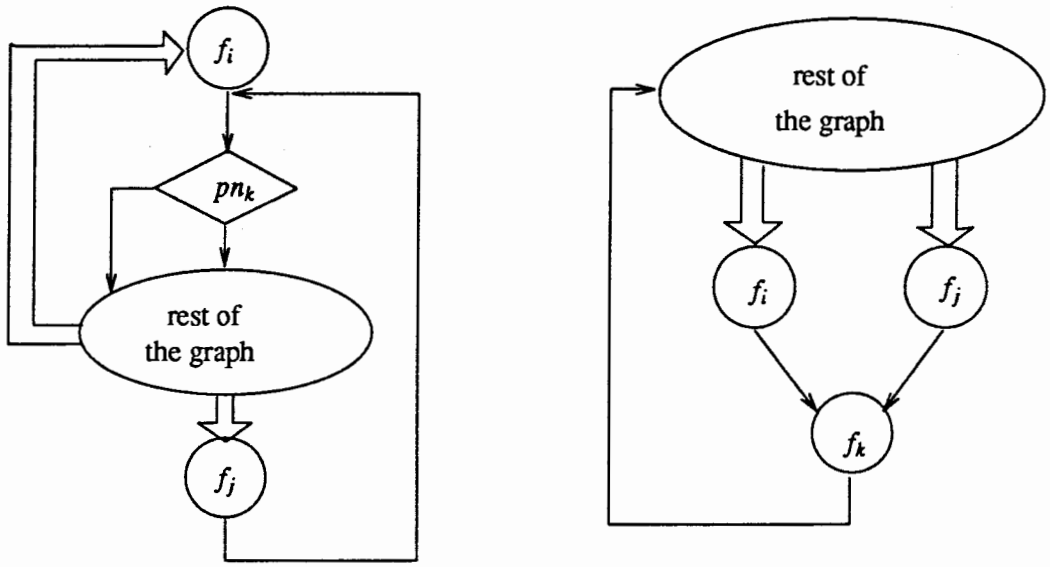


Figure 11. Structure of the two nodes that can be of the same states.

The state generation process is to create two lists which contain the information about the state transitions. One of the lists is a so called *transition-list*.

transition-list = ($transition_1$,$transition_2$,...,$transition_n$)

where

$transition_i$ = (present-state-number, next-state-number, transition-number)

The predicates that determine the transition and the operations to be enabled during the next state for each state transition are generated in another so called *relation-list.*

relation-list = ($relation_1$,$relation_2$,...,$relation_n$)

where

$relation_i$ = (list-of-predicate-numbers, list-of-operation-numbers, transition-number)

Each number in the list-of-predicate-numbers represents a predicate. The Boolean *AND* of all the predicates determines the state transition. Each number in list-of-operation-numbers represents an operation. All the operations corresponding to the numbers in the list should be enabled during the next state.

The generation process keeps the searching from one operational node, which is assigned a state called the present state, to its immediate successors (another operational nodes linked by arcs and predicate nodes), which are to be assigned the states called the next states. Each time an immediate successor is encountered in tracing, the algorithm first checks to see if this node is already assigned a state to avoid the conflict state assignments to a single operational node. If not, it will trace one step further to see if the next node of the *encountered immediate successor node* is a merging node. If it is, the encountered immediate successor node would be assigned the previous state of the merging node.

For example, Figure 12 (a) is part of a cf-graph. The state graph generation for this partial cf-graph starts from the operational node f_1 which is assigned a state, say, m. By tracing this cf-graph, f_1 is assumed to be approached first as the immediate successor of itself. Since it has been already assigned state m, a state transition from the present

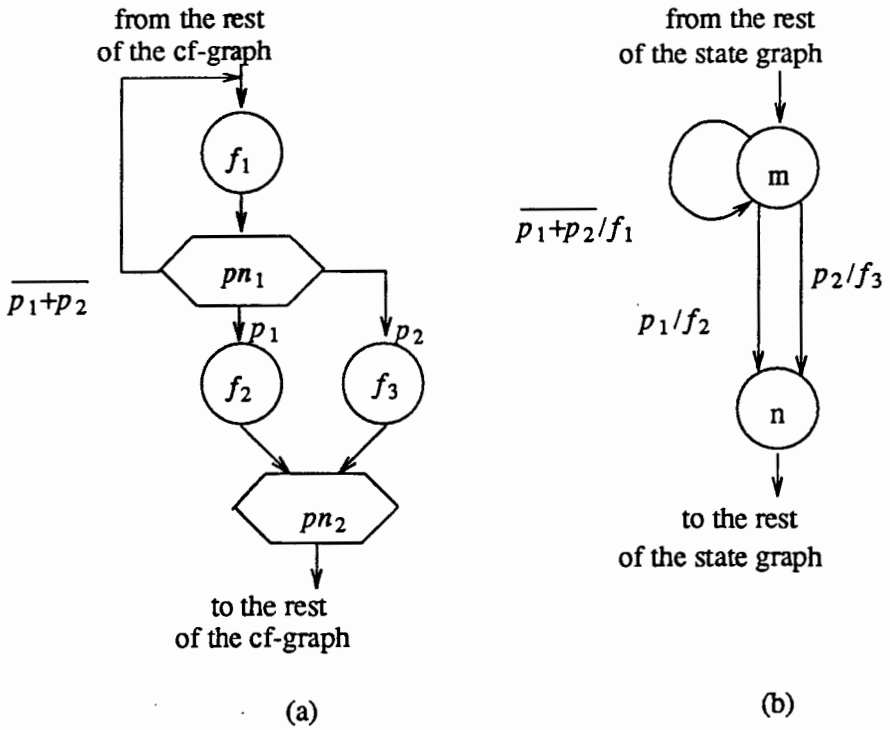


Figure 12. Example of state graph generation.

state m to the next state m is generated. Node f_2 is next found to be another immediate successor which has not been assigned a state. By tracing one step further, it is found that its next node is an unencountered predicate node. Therefore a new state, say n, is assigned to f_2 and a state transition from state m to n is generated. At this stage, the predicate node sn_2 is not realized as a merging node, but it is stored in the list of encountered nodes. Only after the third immediate successor f_3 is approached and one step further tracing encounts sn_2 again, it is found that sn_2 is a merging node and f_3 is to be assigned the same state as f_2. The state graph generated for the cf-graph shown in Figure 12(a) is presented in Figure 12 (b).

The algorithm of the state graph generation is as follows.

Algorithm 3.1

1) Initialization :

start from an initial-node on the cf-graph and set it to be the

current–node;

find the successor node of the initial-node and set it to be the *next–node*;

present–state = 1;

global–next–state = 2;

transition–number = 1;

closed–list = {(*initial–node* . *present–state*)}.

2) While (*next–node* is a predicate node)

{

current–node = *next–node*;

set–of–predicates–along–the–path

= *set–of–predicates–along–the–path* $\cup$ {*current–node*};

waiting–list = *waiting–list* $\cup$ {*all the successor nodes of the current–node*};

next–node = *one of the successor nodes of the current–node*;

delete *next–node* from *waiting–list*;

}

3) By one step further tracing find the successor node of the *next–node*, which will be called the *further–next–node* (FNN).

4) if (*next–node* is closed [comment: has been assigned a state])

{

next–state = *closed–state* [comment: the state assigned to the *next–node*];

closed–merging–list = *closed–merging–list* $\cup$ {(*FNN* . *next–state*)};

}

else

{

if (FNN is included in *closed–merging–list* [has been encounted])

next−state = *previous−state* of FNN;

else

{

next−state = *global−next−state*;

global−next−state = *global−next−state* + 1;

closed−merging−list = *closed−merging−list* $\cup$ {(*FNN* . *next−state*)};

}

closed−list = *closed−list* $\cup$ {(*next−node* . *next−state*)};

}

5) *relation−list* = *relation−list* $\cup$ {(*transition−number*,

{*predicates along the path*},{*operations specified by the next−node*})};

transition−list

=*transition−list* $\cup$ {(*present−state*, *next−state*, *transition−number*)};

relation-number = *relation-number* + *1;*

present−state = *next−state*;

{*set of predicates along the path*} = $\varnothing$;

6) If (the *next−node* is closed)

{

if (the *waiting−list* is empty)

terminate the process;

else

{

current−node = one element in the *waiting−list*;

delete *current−node* from the *waiting−list*;

}

}

else

current-node = next-node;

next-node = the successor node of the current-node;

repeat step 2 to step 6;

end of generation.

§ 3.3 INPUT AND OUTPUT SIGNAL GENERATION

The state graph description generated by the state graph generation process is specified by the transition-list and relation-list (see § 3.2). For FSM hardware realization, each state transition is determined by the input code and the present state code. The FSM outputs are also the specified binary codes. The principles of encoding the input and output signals for FSM with generated state graph description will be introduced in this section. The state encoding problem will be discussed in § 3.6.

§3.3.1. Input codes generation

The input signals of FSM consist of the *internal inputs,* which represent the predicates from the data path, and the *external inputs,* which are the inputs from the outside of the digital processor under design.

Each input signal represents a predicate. As mentioned in CHAPTER II , there are three types of predicates: the variables, the relations and the Boolean function of the variables or relations. For the predicate to be a specified variable, an input signal is needed to represent the logic value of this variable. For instance, the statement *"if a then ..."* is a predicate with the variable *a*. Therefore, one bit input representing the logic value of *a* has to be involved. When the predicate is a relation, an input signal to indicate the logic value of this relation is also needed. For example, the statement *"if (m > n) then ..."* specifies a predicate with the relation $(m > n)$. One bit input signal is needed to identify the logic value of the relation $(m > n)$. The data path has to contain the hardware to

perform the comparison and to provide one bit signal to indicate the result of this comparison.

In the case that the predicate is a Boolean function of the relations or variables, the input signals can either be the signals representing the logic values of the relations or variables, or the signals corresponding to the outputs of this Boolean function. For example, to implement the statement *"if* $(a \wedge (m > n))$ *then ..."* either two bits signals, which correspond to the variable a and relation $(m > n)$ respectively, or one bit of the signal, which is the result of the Boolean *AND* of the variable a and the relation $(m > n)$, are needed. The input corresponding to the second approach is called a *pre–encoded* input and the additional combinational logic gates have to be included to realize this Boolean function. If the arguments in the Boolean function, a and $(m > n)$ in above example, have already been used as the inputs in realization, then the second approach would cause an extra input signal as well as the hardware to generate this signal. In the current FSM Synthesizer, either of the above two approaches can be selected by the user.

§3.3.2. Output codes generation

Output codes of the FSM are the control signals to enable certain operations in the data path. For different objects under control, different types of control codes are needed. For example, the bus oriented system needs the signals to control buffers for the register transfer operations, while the multiplexer oriented system needs the control signals to address multiplexers. The current version of DIADES can generate only multiplexer oriented systems. The data path operations to be controlled are the data transfers among registers and memories.

1) Control signals for register transfer :

A. Special control: Special controls are defined for control registers only. They include the control for register *clear*, *increment* and *decrement* operations. Each operation needs a unique signal to enable it. For example, the assignments for register a,

(:= *a* 0), (:= *a* (*plus a* 1)) and (:= *a* (*plus a* (*minus* 1)), are three forms of operation that need special control.

B. Control for register loading : If the input of the register comes from a single source, the control of the *load* operation needs only one bit signal. Whenever the input comes from more than one source, we refer to the case of *similar assignments*. In such case the control output provides the signals for multiplexing of the multiple inputs. The input source is the output of the other register or combinational logic. Assuming that there are similar assignments orderred from number 1 to number n, which are the different assignments to a certain destination register, the *multiplexer addressing* is shown below:

0...00 - No operation

0...01 - assignment 1

0...10 - assignment 2

.

.

1...11 - assignment n

The right most bit of the above addressing codes, as well as of all the codes introduced afterwards, is the *least significant bit* while the left most bit is the *most significant bit*. The bits from most significant to least significant are represented as $b_m, b_{m+1}, \ldots, b_{m+k}$.

In cf-graph description, each assignment has a corresponding *assignment number*. In the above addressing code, assignment 1 refers to the lowest number of assignment specified by the cf-graph. Assignment n refers to the highest number of assignment specified by the cf-graph.

The *don't care conditions* occupy the upper end of the address range. These addresses are never selected. Here don't-care conditions mean that the respective codes

are not used.

The multiple inputs can be identified by similar assignments specified in the cf-graph. For example, statements (2 (:= *x a*)), (5 (:= *x b*)), and (8 (:= *x c*)) specified in the cf-graph are the similar assignments. The first number in each parentheses is the assignment number. If the assignments for x consist only of these three forms, then the multiplexing addressing codes for them would be

00 - No operation.

01 - Control of (:= *x a*)

10 - Control of (:= *x b*)

11 - Control of (:= *x c*)

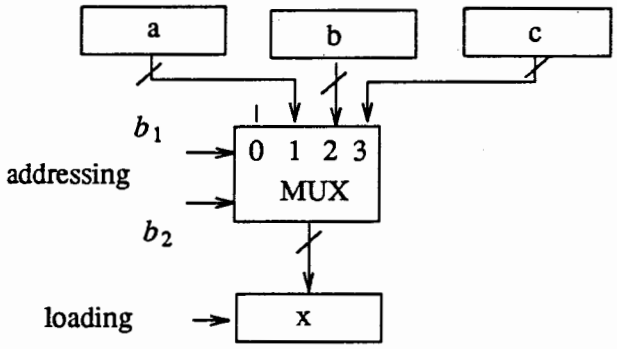


Figure 13. Block diagram of the similar assignment example.

The block diagram of the hardware implementation to realize these assignments is shown in Figure 13. The registers used in DIADES system are the ones which can be loaded only the load input is active high.

Special control does not belong to similar assignments but concatenate does. For example, the statements (:= *x* ((*b* [0 % 3]) @ (*c* [0 % 3]))) (symbol @ means concatenate) and (:= *x* ((*a* [0 % 3]) @ (*c* [0 % 3]))) are two similar assignments. There are two hardware choices to realize these assignments. Their block diagrams are presented in Figure 14. For either implementation, the addressing signals to select the inputs for

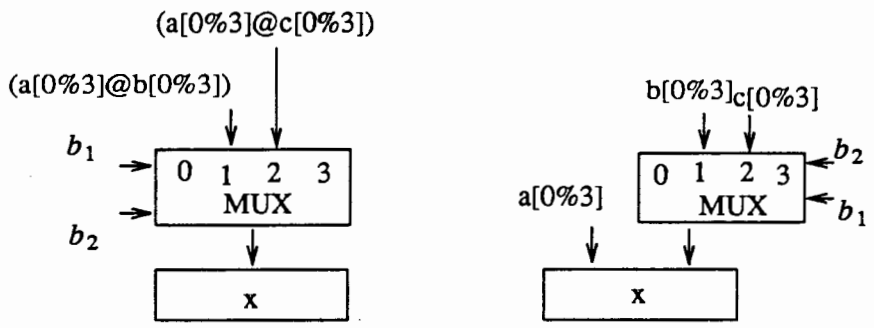


Figure 14. Block diagram of the concatenate assignment example.

the multiplexer have to be generated. The control unit, however, generates the same control signals in both cases. The assignments (:= (*x* @ *y*) *d*) and (:= *x a*) also contain similar part, since the statement (:= *x* (*d* [0 % *m*])), which assigns the first m bits to register x, and (:= *x a*) are similar.

2) Control signal for memory

For the memory reference data transfers, two bits of read-write control signals are needed. The control implied by these two bits is shown below:

00 - No operation

01 - Read

10 - Write

11 - Never used

If the memory is addressed possibly by more than one source, similar to multiple inputs register assignments, the extra signals for multiplexer addressing, which select the memory address from different sources, should be generated. All the assignments which refer to the memory unit are considered to be similar assignments. The difference of the read and the write assignments is identified by the read-write control signal. For instance, if three different memory referring assignments specified by program graph are (2 (:= (*m* [*R* 2]) *a*)), (5 (:= *b* (*m* [*R* 1]))) and (8 (:= (*m* [*R* 3]) *c*)), the memory control

signals for them should be as below:

0110 - Control for assignment (:= (*m* [*R* 2]) *a*)

1001 - Control for assignment (:= *b* (*m* [*R* 1]))

1110 - Control for assignment (:= (*m* [*R* 3]) *c*)

The first two bits of the above signals select the memory address from different registers. The last two bits perform the read-write control. For instance, the first two bits of the control code 0110 are 01 which address the multiplexer to select the memory address from the register *R* 2. The last two bits 10 control memory read operation.

It should be pointed out that for the memory read assignments, the control signals for the destination register loading need to be generated as well, which was, however, not shown in the above example. For example, two memory read statements (5 (:= b (m [R1]))), (7 (:= d (m [R2]))) specify two memory read operations to transfer data from the memory to different registers b and d. The addresses of the memory are provided by the registers R1 and R2. The control signals not only select the memory addresses for the operations, but also select the registers to load the data.

The address selection codes are ordered by the assignment numbers specified in the cf-graph description. It is the same method as the one used for multiplexer addressing for similar assignments in the register loading control.

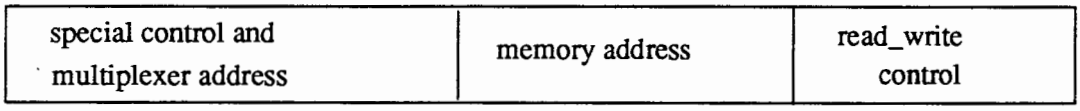


Figure 15. Fields in output control code.

When the control signal is represented by a binary code expression, the total number of bits in it is equal to the total number of signals for the special controls, the register transfer controls, and the memory reference assignment controls. The output

control code is divided into three fields, as shown in Figure 15. In each field the least significant bit is the right most bit and the most significant bit is the left most bit. The first field provides the special control signals and the multiplexer addressing signals for all the distenation registers. The order of the control bits in this field is allocated according to the order of the assignment numbers specified in the cf-graph description. Let us assume that the different assignments specified by the cf-graph are:

(15 (:= *f* (*mem* [*R* 1])))

(13 (:= *b* 0))

(10 (:= *f* (*plus f* (*minus* 1))))

(7 (:= *f* (*and b* (*mem* [*R* 2]))))

(4 (:= (*b* @ *f*) *t*))

(3 (:= (*mem* [*R* 3]) (*or f b*)))

and the block diagram for the hardware implementation to perform these assignments is shown in Figure 16.

The control codes assigned to enable these assignments are as follows.

The special controls (two bits - b_1, b_2):

00 - No operation

01 - (:= *b* 0)

10 - (:= *f* (*plus f* (*minus* 1)))

The controls for register loading (three bits):

loading for register b - one bit (b_3).

0 - No operation

1 - (:= (*b* @ *f*) *t*)

Select input for register f - two bits (b_4, b_5).

00 - No operation

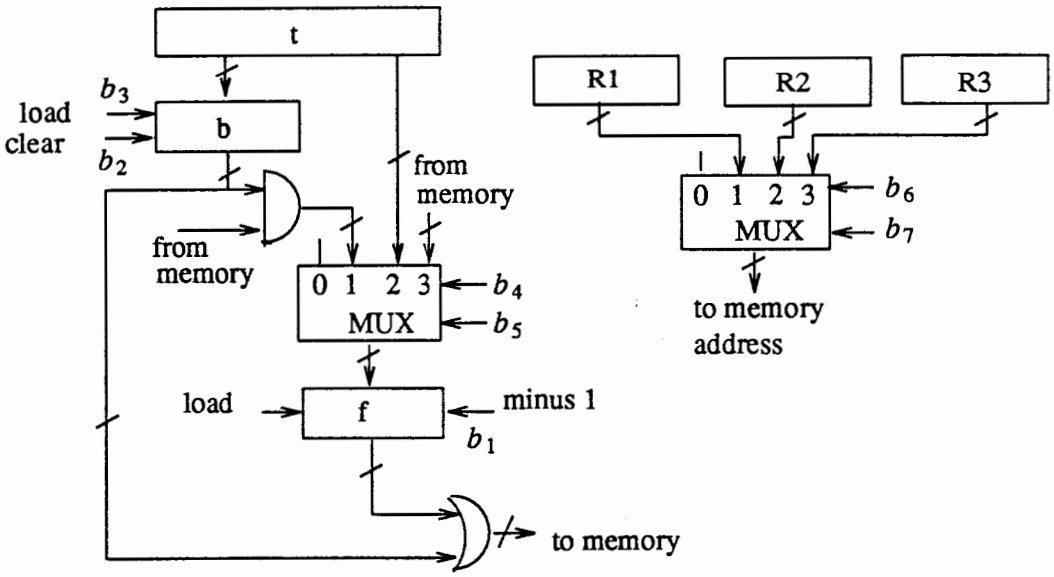


Figure 16. Block diagram of combinational assignments.

01 - (:= (*b* @ *f*) *t*)

10 - (:= *f* (*and b* (*mem* [*R* 2])))

11 - (:= *f* (*mem* [*R* 1]))

The controls for memory reference assignments (four bits - b_6, b_7, b_8, b_9) which include the selection of the memory address as well as the read-write controls:

Memory read assignments :

0000 - No operation

0110 - (:= *f* (*and b* (*mem* [*R* 2])))

1010 - (:= *f* (*mem* [*R* 1]))

Memory write assignment:

0000 - No operation

1101 - (:= (*mem* [*R* 3]) (*or f b*))

The total number of bits in the control output codes needed in this case is nine. If in a certain state transition, the assignments (15 (:= *f* (*mem* [*R* 1]))) and (13 (:= *b* 0))

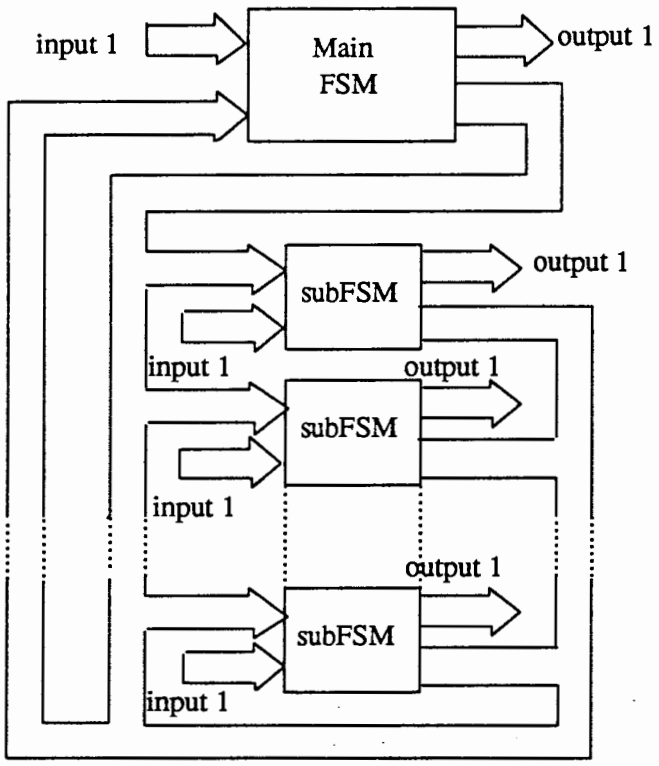
need to be executed during the next state period, then the output control codes are represented by the binary number 010111010. In this expression, the first bit from the left is the control signal for the special control of register *b*, the third bit (loading for register b) has no operation, the next two bits are the control for multiplexer addressing which selects input for register f, the fourth and the fifth bits are the memory address selection and the last two bits are the read-write control.

§ 3.4 PARALLEL CONTROL FLOW TO SEQUENTIAL CONTROL FLOW CONVERSION

For the circuits described using a parallel program graph, there are two kinds of approaches to FSM control unit realization. In the first approach the machine is composed of one main FSM, which controls global operations, as well as several pieces of subFSMs, each of them controlling the operations for one of the parallel branches. Some communications have to be involved between the main FSM and subFSMs as well as the subFSM themselves to trigger each other. The signals of communication are the selected inputs and outputs of each FSM. The block diagram of this approach is shown in Figure 17. Since each piece of parallel branch can be treated as a sequential graph, the state description generation for a sequential program graph can be utilized several times by a designer for a parallel program graph to generate the main FSM and the subFSMs.

In case that the multi-FSMs control will increase the complicity of the control unit, another control unit realization is also a choice which is composed of just a single FSM. For this kind of realization the conversion from the parallel cf-graph to the sequential graph has to be introduced to make it possible to use the remaining programs of the FSM Synthesizer for the parallel graph as well.

The cf-graph shown in Figure 18(a) is a part of a parallel graph. From the given parallel cf-graph description, the following information can be extracted before the



input 1 - from external or data path.
output 1 - to external or data path.

Figure 17. Block diagram of multi-FSM control.

conversion.

An *open–set* $OS = \{fp_1, fp_2, ..., fp_m\}$, which is the set of operational nodes that are simultaneously executed in one machine cycle. In Figure 18, set $\{f_1, f_2\}$ is an open-set.

A *leading–set* $LS = \{TP_1, TP_2, ..., TP_m\}$, in which each TP_i is a set of disjoint transition predicates, denoted by $\{tp_{i1}, tp_{i2}, \ldots, tp_{ik}\}$, which lead the control-flow from fp_i to its *immediate successors*. As mentioned in CHAPTER II, all the tp_i's are disjoint and the Boolean *OR* of them is a logic 1. In Figure 18, the set $\{\{a, \bar{a}\},\{b, \bar{b}\}\}$ is such a leading-set.

An *end–set* $ES = \{ISP_1, ISP_2, ..., ISP_m\}$, in which each ISP_i is a set of immedi-

ate successors of fp_i, denoted by $\{fs_{i1}, fs_{i2}, ..., fs_{ik}\}$. In Figure 18, the end-set is $\{\{f_3,f_4\}\{f_5,f_6\}\}$.

the transition functions : $\{fs_{ij} = Q(fp_i, tp_{ij}) \mid i=1,2,...,m, j=1,2,...,i\}$.

where Q is the transition function defined in CHAPTER II.

The tp_{ix}'s included in a certain TP_i are disjoint, but the tp_{ix} and tp_{jy} are not necessarily so if i ≠ j. When they are nondisjoint, under the predicate tp_k, which is $(tp_{ix} \wedge tp_{jy})$, both the transition functions $fs_{ix}=Q(fp_i,tp_k)$ and $fs_{jy}=Q(fp_j,tp_k)$ are satisfied. It means that the fs_{ix} and fs_{jk} will possibly be approached in parallel. With the currently known information, the set of new parallel operational nodes can be, therefore, derived.

The conversion process is to create a behaviorally equivalent sequential graph. In this sequential graph all the operations executed in parallel are specified by a single operational node, and the transition predicates leading the transitions from one operational node to its immediate successors are disjoint. The following tasks should be, therefore, carried out with the information provided.

Create a new operational node *fn* which is the combination of all the nodes included in *OS* so that all the parallel operations specified by fp_i's in *OS* are specified by the current *fn*.

Generate the set of *new transition predicates* which leads the transitions from *fn* to its immediate successors. The set is denoted by $TP = \{tp_1, tp_2, ..., tp_n\}$

Find out the set of all the new immediate successors of *fn*. All the possible combinations of derived parallel operational nodes will be included in it. This set is denoted by FSN = $\{FS_1, FS_2, ..., FS_n\}$.

The FS_k is defined as the set of elements which is

$$FS_k = \{pf_i \mid pf_i \in ISP_i, pf_i = Q(fp_i, p_i), p_i \in TP_i, tp_k \cap p_i \neq 0\} \quad (1)$$

the above expression means that the newly generated immediate successor FS_k is the combination of the immediate successors of the nodes in OS in the parallel graph. For each element in the combination, there is a corresponding transition specified by the parallel graph. The transition starts from one of the elements in OS, and determined by the transitions predicate nodisjoint to the new predicate tp_k.

The converted control flow is a sequential control flow only if

1) the transition predicates are disjointed,

2) the Boolean *OR* of the transition predicates is a logic 1.

For a set of specified predicates

$$P = \{p_i \mid p_i \in TP_i\} \tag{2}$$

the control flow from a set of parallel operational nodes in OS in the parallel graph will transfer to a set of *parallel immediate successors*

$$PF = \{pf_i \mid pf_i \in ISP_i, pf_i = Q(fp_i, p_i)\} \tag{3}$$

In Figure 18 (a), for a set of specified predicates $P = \{a, b\}$, the control flow from the parallel operational node set $OS = \{f_1, f_2\}$ will go to the parallel immediate successor set $PF = \{f_3, f_5\}$.

The relation between the elements in set P is the logic *AND* relation. If the new transition predicate is defined as the logic *AND* of the elements in set P, formula (1) will be satisfied when the p_i in the formula is the same one in set P. Comparing formula (1) and formula (3), it can been seen that the transition defined by formula (1) is equivalent to the transition specified by the parallel graph.

For example, the new transition predicate of the logic *AND* of the elements in set $P = \{a, b\}$ is ab. It is nondisjoint to both predicates a and b. By formula (1), the transition will go to the immediate successor which performs the operations specified by the operational node f_3 and f_5. They are equivalent to what is specified by the parallel

graph.

The key problem for the conversion is now focussed on how to generate all the possible set of P specified by the formula (2). This is approached by the generation of the Cartesian product of those items included in LS. By the definition, their Cartesian product is expressed as

$$TP_1 \times TP_2 \times \cdots \times TP_m = \{(tp_{1x_1}, tp_{2x_2}, \ldots, tp_{mx_m}) \mid tp_{ix_i} \in TP_i\}$$

The set of generated transition predicates is defined as

$$TP = \{tp_x \mid tp_x = tp_{1x_1} \wedge tp_{2x_2} \wedge \ldots \wedge tp_{mx_m}\}$$

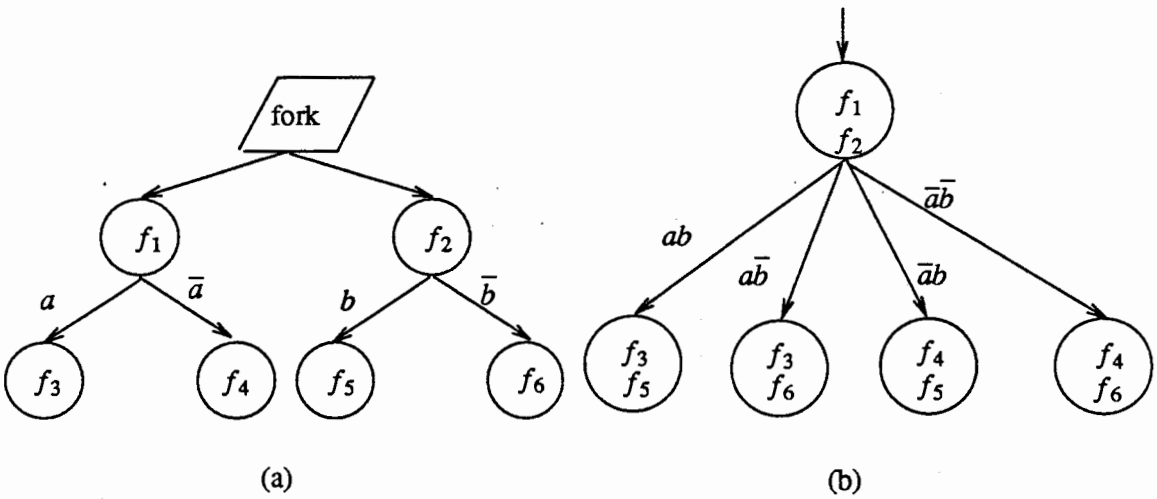


Figure 18. Example of parallel to sequential conversion.

The idea of this conversion will be explained by the next example.

Example 3.1 :

A part of a parallel graph is presented in Figure 18 (a). Two branches after node *FORK* are executed in parallel. A new operational node for the sequential graph is first created, which is the combination of f_1 and f_2. The set of the components of this combination is an *open-set* (OS) which is put to a so called *open-list*. At this stage, the list has only one open-set which is $\{f_1, f_2\}$. The next conversion starts from one of the open-sets in the open-list. For the open-set $\{f_1, f_2\}$ the

leading-set (LS) is $\{\{a, \bar{a}\},\{ b, \bar{b}\}\}$, which determines the parallel control flow from f_1 and f_2 to their immediate successors f_3 or f_4 and f_5 or f_6 respectively. The Cartesian product of the subset in the *leading-set* is a set of new predicates. These predicates determine the transition for the new operational node to its immediate successors. The Cartesian product of the subsets in leading-set $\{\{a, \bar{a}\},\{ b, \bar{b}\}\}$ is

$$TP_1 \times TP_2 = \{(a,b),(a,\bar{b}),(\bar{a},b)(\bar{a},\bar{b})\}$$

The set of the new transition predicates is

$$TP = \{ab\ ,\ a\bar{b}\ ,\ \bar{a}b\ ,\ \bar{a}\bar{b}\}.$$

For each new predicate, a set of the parallel immediate successors of the nodes in the open-set is found. This set of the parallel immediate successors is called a *parallel successor set.* The combination of these parallel successors forms a new immediate successor in the sequential graph. In consequence, the set of the new immediate successors in the sequential graph is generated by using all the elements from the leading-set. The new immediate successors determined by leading-set $\{ab\ ,\ a\bar{b}\ ,\ \bar{a}b\ ,\ \bar{a}\bar{b}\}$ are four operational nodes which are the combinations of f_3 and f_5, f_3 and f_6, f_4 and f_5 as well as f_5 and f_6 respectively. Each combination will then be put into the open-list. The equivalent sequential branch for Figure 18 (a) is created by the above process in Figure 18 (b).

The generation of the Cartesian product can be mathematically approached by the formula

$$(tp_{11}+tp_{12}+...+tp_{1n_1})(tp_{21}+tp_{22}+...+tp_{2n_2})...(tp_{m1}+tp_{m2}+...+tp_{mn_m})$$

By expanding this formula, a sum of products form expression is achieved. Each product form in this expanded expression corresponds to a tp_i in TP.

In Example 3.1, the Cartesian product of the leading set $\{\{a, \bar{a}\}\{b, \bar{b}\}\}$ can be

achieved by the formula

$$(a+\bar{a})(b+\bar{b})=ab+a\bar{b}+\bar{a}b+\bar{a}\bar{b}$$

The generated new predicates are the product terms from the above sum of products form.

In this approach, the generated transition predicates are disjoint, because the different tp_x and tp_y contain different tp_{ix_i} and tp_{iy_i} included in the same TP_i as their Boolean *AND* terms. From the given information, the items included in the same TP_i are disjoint so that the results of the above Boolean *AND* are disjoint.

Any two of the generated predicates in the above example contain disjoint predicates a and $\bar{a}$ or b and $\bar{b}$. They are, therefore, disjoint.

By the definition, the Boolean *OR* of all the generated transition predicates can be expressed as

$$tp_1+tp_2+\cdots+tp_k = (tp_{11}+tp_{12}+...+tp_{1n_1})(tp_{21}+tp_{22}+...+tp_{2n_2})...(tp_{m1}+tp_{m2}+...+tp_{mn_m})$$

From the definition of the leading-set given above, the results of all the *OR* forms at the right side of the above expression have logic values 1. So the result of the *AND* of all the *OR* forms also has a logic value 1. In the above example, the Boolean *OR* of all the generated predicates is

$$ab+a\bar{b}+\bar{a}b+\bar{a}\bar{b} = (a+\bar{a})(b+\bar{b}) = 1$$

Now the fact that the converted graph is a sequential has been verified.

By the above analysis it can be concluded that the converted control flow description is a sequential control flow description and is equivalent to the provided parallel description. When the generated immediate successor is a combination of a set of operational nodes, the new open-set is generated. The next conversion step will start from any one of the open-sets and the information needed for the conversion can be extracted from the parallel cf-graph description.

The conversion algorithm must take, however, some particular cases into account. For instance, the result of the Cartesian product should exclude the null terms. According to the properties of the different parallel control nodes introduced in CHAPTER II, some special procedures should be also included in the conversion process. The open-set should exclude a *DROP* node since such a node terminates the transition of the branch in which it is located. The passing *DAND (DEXOR)* transition is the transition to the node pointed by the arrow from *DAND (DEXOR)* node. The destination node of the passing *DAND (DEXOR)* is called a *passing DAND (DEXOR) node*. If the parallel successor set only partially passes a certain *DAND* node (not all nodes in this set have passed the *DAND* node), then only the *unpassing nodes* are kept in the open-set, since any transition that passes *DAND* node is allowed only when all the nodes in the open-set have passed *DAND* node. Only the *passing DEXOR nodes* are kept in the open-set when the parallel successor set partially passes a *DEXOR* node, since the transition which passes *DEXOR* is performed whenever any one of the nodes in open-set is a passing *DEXOR* node. The transitions starting from the unpassing *DEXOR* nodes will be terminated. The conversion process for passing *DAND* case can be illustrated by the next example.

Example 3.2 :

Figure 19(a) is the part of the parallel cf-graph. For the conversion of this graph to the part of the sequential cf-graph, the current *open-set is* $\{f_1, f_2\}$. The current operational node for the sequential graph is the combination of nodes in the open-set. The set of the new transition predicates is $\{ab, a\bar{b}, \bar{a}b, \bar{a}\bar{b}\}$. For the transition predicate $\bar{a}\bar{b}$, the parallel successor set is the open-set itself and the new immediate successor in the sequential cf-graph is the current operational node. Under the transition predicate $\bar{a}b$, the parallel successor set is $\{f_1, f_3\}$, in which f_3 is an passing *DAND* node. Since the transition from f_1 to itself has not passed the *DAND* node, the f_3 can not be approached. The immediate successor generated for the sequential cf-graph is f_1. The successor transition should start from

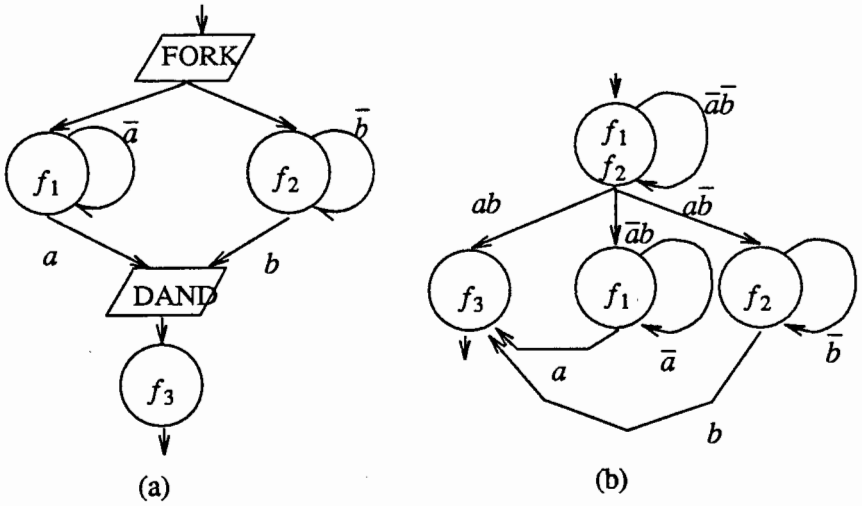


Figure 19. Example of parallel to sequential conversion.

the new open-set $\{f_1\}$ and its immediate successors are f_1 and f_3 led by the predicates $\bar{a}$ and a respectively. With a similar analysis, under the predicate $a\bar{b}$, the generated immediate successor is f_2 and the immediate successors of f_2 are f_2 and f_3. Only under the predicate ab, the *DAND* node is passed and the generated immediate successor is f_3, since both the transitions from f_1 and f_2 have passed the *DAND* node. The converted sequential graph is presented in Figure 19(b).

The parallel to sequential conversion is performed with the parallel control-flow description in terms of the following data formats.

1) *A set of the transition descriptions :* each element in it contains the lists of three elements: *the transition-number, the present operational node* and *the immediate successor.*

2) *A set of the relation descriptions :* each element in it contains the lists of the transition-number, the transition predicates for the transition and *the operations specified by the immediate successor.*

3) *A set of the DROP nodes.*

4) *A set of the passing DAND transitions :* each element in it is a pair of an operational node and its immediate successor which has passed the *DAND*.

5) *A set of the passing DEXOR transitions :* each element in it is of the same form as that in 4).

6) *A set of the passing FORK transitions :* each element in it contains the items of the present operational node and a list of its immediate successors determined by the leading-set.

The data format for above descriptions will be presented in CHAPTER V.

For the example presented in Figure 19, if the node numbers which correspond to f_1, f_2 and f_3 are 1, 2 and 3 respectively; the predicate numbers correspond to predicates a and b is 4 and 5; the node number before fork, which is not shown in Figure 19, is 0. Also, it is assumed that the operation specified by each operational node has the same number as the node number. Then the set of the transition descriptions is

{(1 (0 1))(2 (0 2))(3 (1 3))(4 (2 3))(5 (1 1))(6 (2 2))}.

The set of the relation descriptions is

{(1 nil (1))(2 nil (2))(3 (4) 3)(4 (5) 3)(5 ((not 4)) 1)(6 ((not 5)) 2)}.

The "nil" specification in above set corresponds to logic value 1. It means that the transition from the current node to the node specified in the set of transition descriptions will always happen in any case.

The set of passing *DAND* transitions is {(1 3)(2 3)}. The set of passing *FORK* transitions is {(0 1)(0 2)}.

The algorithm for the conversion can be described as follows.

Algorithm 3.2

1) Start from an initial node specified by the parallel control description. Create an

open-list of the only open-set of this initial node.

2) Pick up one open-set from the open-list and put it to a so called *closed-list.*

3) Find out the leading-set, which corresponds to the current open-set by checking the set of transition descriptions, the relation descriptions and the set of the passing *FORK* transitions. For the sequential transition, this leading-set is composed of only one element which is the set of disjoint transition predicates.

4) Generate the Cartesian product of all subsets of the leading-set generated in step 3, which will become the set of new predicates.

5) Find the set of new immediate successors of the current open-set, which are determined by the new predicates.

6) For the set generated in step 5, delete *DROP* nodes from all the subsets by checking the set of the *DROP* nodes provided; delete the nodes passing the *DAND* nodes from the partially passing *DAND* subsets by checking the set of the passing *DAND* transitions; delete the nodes unpassing the *DEXOR* nodes from the partially passing *DEXOR* subsets by checking the set of the passing *DEXOR* transitions.

7) Check each subset of the set processed by step 6 to see if it has been already included in the closed-list. If no, put it into the open-list.

8) Create converted sequential control-flow description in terms of the new set of transition descriptions and the new set of relation descriptions. Check to see if the open-list is empty. If not, repeat steps 2 to 8. If yes, terminate the conversion.

For the example in Figure 19, the conversion starts from a initial node, say 0. The open-list at this stage is composed of one open-set {0}. This open-set is then put to a closed-list.

The leading-set for the current open-set is {(nil,nil)}. Here the "nil" specification

has the same meaning as the one specified in the set relation descriptions. The Cartesian product of this leading-set is also "nil".

With the above leading-set, the new immediate successor is (f_1,f_2). This combination is then put into the open-list as an new open-set as well as into the closed-list as a closed-set. The open-list now still contains only one element $\{f_1,f_2\}$. The next conversion will start from this open-set.

After putting this open-set to the closed list, the closed list is $\{\{0\},\{f_1,f_2\}\}$. The leading-set of the current open-set is $\{\{a,\bar{a}\},\{b,\bar{b}\}\}$. The Cartesian product of this leading set is $\{ab,a\bar{b},\bar{a}b,\bar{a}\bar{b}\}$. Each product is a new transition predicate.

The parallel successor set is $\{(f_3),(f_2,f_3),(f_1,f_3),(f_1,f_2)\}$. Each element in it will be a new immediate successor of the node corresponding to the open-set. The transitions from the open-set to the new immediate successors are led by the corresponding new transition predicates.

Since $(f_2,f_3),(f_1,f_3)$ are partially passing *DAND*, the passing *DAND* node f_3 should be deleted from them. After the deletion, the parallel successor set becomes $\{(f_3),(f_2),(f_1),(f_1,f_2)\}$. This is the set of new immediate successors in sequential graph. Since the set $\{f_1,f_2\}$ is already in the closed-list, only other three elements in the above set are put into the open-list as the new open-sets. The current open-list is $\{\{f_1\},\{f_2\},\{f_3\}\}$.

Suppose the following transition starts from the open-set $\{f_1\}$, the closed-list will include this set in it. The leading-set for this open-set is $\{\{a,\bar{a}\}\}$. The Cartesian product of this set is itself. The new immediate successors of the open-set are (f_1), (f_3). The first one is included in the closed-list and the second one is already in open-list.

The conversion will be continued until no element is left in open-list.

§ 3.5 GRAPH MODIFICATION

The graph modification process consists of alternating the predicate specifications in cf-graph and, consequently, achieving a variant number of input bits for FSM.

It has been already mentioned that a predicate can be the Boolean expression of relations or logic variables such as $p_1p_2+\overline{p_1p_4}$, or $\overline{(p_1+\overline{p}_4)(\overline{p}_1+p_2)}$, etc. Here p_i is a logic variable or a relation. For an FSM based control unit implementation, the logic value of each predicate is an input signal to the FSM. If the predicate is a Boolean expression, the input signal would be the output of the combinational logic which realizes this Boolean expression. By this approach, some redundant input bits are generated.

Amoung a set of predicates

$$P = \{p_k \mid k=1,2,...,n\}$$

a redundant predicate p_i is defined if there exist

$$p_i = B(p_{s1},p_{s2},...,p_{sm})$$

and

$$\{p_{sj} \mid j=1,2,...,m,\ sj \neq i\} \subset P$$

where $B(p_{s1},p_{s2},...,p_{sm})$ represents the Boolean expression of the arguments in parentheses. The input signal corresponding to this redundant predicate is an extra bit input, since this predicate can be evaluated with the logic values of other predicates which have been already utilized as the input signals. For example, any one of the predicates $p_1=(a>b)$, $p_2=(a<b)$ and $p_3=(a=b)$ would be a redundant predicate if the other two predicates have been already utilized as the input signals, since $p_1=\overline{p}_2\overline{p}_3$, $p_2=\overline{p}_1\overline{p}_3$ and $p_3=\overline{p}_1\overline{p}_2$. If the FSM is realized by the PLA and memory devices, the increment of each input bit corresponds to the increment of one PLA column. In the case of long and narrow PLA's, the column increment causes comparatively much area waste.

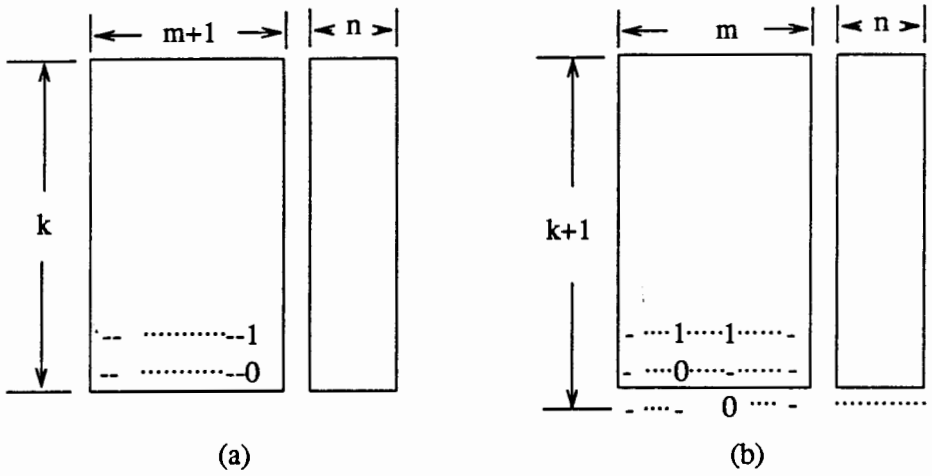


Figure 20. Example of different PLAs for a FSM.

For example, let us assume that a truth table, which is the description of the combinational part of a FSM, has (m + 1) bits of input. Among these (m + 1) bits, m bits represent m unique predicates or their complements, and another single bit is the predicate of a Boolean expression having two predicates as its arguments. If this expression is of the form (P_i * P_j) and can be treated as an unique input signal to the FSM, the PLA description for this FSM is as in Figuer 20(a), where k is the number of rows, and n is the number of output bits. The total area needed by this PLA is proportional to k * (m + 1 + n). If, instead of the predicate of the Boolean function, the predicates of the two arguments of the function are taken as the input signals, then, instead of a single row corresponding to the predicate of the Boolean function, the two rows corresponding to the predicates of the two function arguments should be specified in the truth table. Therefore, only one row will be added to the truth table. Assuming P_i and P_j are the two predicates included in those m bits, then one column would be saved with the second approach.

The corresponding truth-table is presented in Figure 20(b). The size of this PLA would be proportional to (k + 1)(m + n). The size difference between the two PLA's

mentioned is proportional to k - (m + n). If k >> (m + n), the second approach will save a certain amount of silicon area besides the area needed by the combinational logic to realize this function.

Not knowing the relation between k and (m + n), or even the logic realization technology, a design variant will provide the chance to select optimal result from different design approaches. This variant approach can be performed at either the cf-graph level or the FSM description level. For both levels, the purpose of the approach is to modify the predicates which are of the Boolean expression forms.

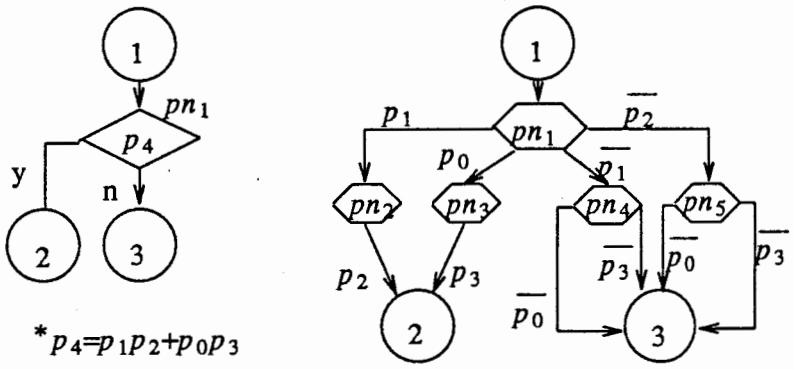


Figure 21. Example of graph modification.

The fact that the Cartesian product of the transition predicates is needed in the parallel to sequential conversion process is considered while making a decision at the modification level. If the transition predicates are only of the product form, then the algorithm to generate their products and to find the empty products which should be excluded will be speeded up as the result of such modification. The product of the product forms is also a product form. The empty product can be checked out whenever two complemented variables are included in the product. Therefore, the modification of the predicates on the cf-graph level will not only generate a variant PLA for FSM but will also benefit the parallel to serial conversion process.

The modification modifies only the predicate nodes of the graph. The idea of

modification will be illustrated in the next example.

Figure 21 (a) presents part of a graph description. The predicates specified by the predicate node pn_1 are p_4 and $\overline{p}_4$. Suppose that p_4 is a Boolean expression of the variables of other four predicates, which is of the sum of products form as $p_1p_2+p_0p_3$. The sum of products form of the $\overline{p}_4$ is

$$\overline{p_1p_2+p_0p_3}=\overline{p_1p_2}+\overline{p_0p_3}=(\overline{p}_1+\overline{p}_2)(\overline{p}_0+\overline{p}_3)=\overline{p}_1\overline{p}_0+\overline{p}_2\overline{p}_0+\overline{p}_1\overline{p}_3+\overline{p}_2\overline{p}_3.$$

The predicates of p_4 and $\overline{p_4}$ are called *original predicates*. The operational nodes 2 and 3 are called the *original next nodes* determined by the original predicates. The modified cf-graph is presented in Figure 21 (b).

The modification creates a predicate node first, which corresponds to a *case* control statement. It specifies a set of predicates which are the different first literals of the product terms in both the sum of products forms of p_4 and $\overline{p_4}$. At this stage, the first literals p_1p_2 and p_0p_3 are $\{p_1\}$ and $\{p_0\}$. in each product term are called the *current literals*. The predicates specified by the first created predicate node for the given example are $p_1, p_0, \overline{p_1}$ and $\overline{p_2}$.

After the first generation, the first literals are put into the so called *predecessor literal sets* of the product terms. At the current stage the predecessor literal sets for product terms are $\{p_1\}$, $\{p_0\}$, $\{\overline{p}_1\}$,$\{\overline{p}_1\}$, $\{\overline{p}_2\}$, $\{\overline{p}_2\}$. Then the product terms that have the same predecessor literal set are grouped, and the second literals in the product terms are set to be the current literals. For the example shown, four groups are found. They are (p_1p_2), (p_0p_3), $(\overline{p}_1\overline{p}_0,\overline{p}_1\overline{p}_3)$ and $(\overline{p}_2\overline{p}_0,\overline{p}_2\overline{p}_3)$. The second literals in four groups are (p_2), (p_3), $(\overline{p}_0,\overline{p}_3)$ and $(\overline{p}_0,\overline{p}_3)$ respectively. These second literals are also the new current literals. For each group, a predicate node which specifies the predicates corresponding to the current literals in the group is created. Each transition from the previous node to the new node is determined by the predicate corresponding to the last predecessor literal of the product terms. For the given example, four new predicate

nodes are created. Two of them specify the predicates p_2 and p_3, respectively. Both of the other two nodes specify the predicates $\overline{p}_0$ and $\overline{p}_3$.

The product terms that have the current literals as their last literals are deleted from the sum of products forms. The transition determined by the predicate corresponding to the last literal in the product term leads to the original next node. The original next node is chosen according to the original predicate whose sum of product form includes the products term and which leads the transition to the original next node. The included product term has the last literal that corresponds to the specified predicate.

For the given example, all the current literals are the last literals in the product terms. Therefore, the transitions determined by their corresponding predicates lead to the original next nodes 2 and 3, respectively. The choice of nodes 2 or 3 for each predicate is done according to which sum of products form the product term is included in. For example, the transition determined by the predicate p_2 leads to the original next node 2, since the corresponding literal of the predicate p_2 is the last literal of the product term p_1p_2. This product term is included in the sum of products form of p_4. In the orignal graph the predicate p_4 leads the transition to node 2.

If the product terms have more than two literals, then the current literals will be included in the predecessor literal set, the third literals of the product terms will be set as the current literals and the above process will be repeated. The creation of the predicate nodes will continue until the last literals in all the product terms are processed.

During the modification, some special predicate nodes, which specify only one predicate, are created. They can be thought of as the special *case* statements which have only one predicate. This predicate will always be true when the control flow reaches the predicate node specifying it. The modified graph no longer holds the property mentioned in CHAPTER II that the predicates specified by a single predicate node are disjoint. The overlapped predicates, however, always lead the transitions to a common immediate suc-

cessor. The modified graph is only an internally used graph which does not change the overall system performance.

The algorithm for the above described modification is as follows.

Algorithm 3.3

1) Find a predicate which is of a Boolean expression form.

2) Expand this predicate to the sum of products form.

3) Find a predicate node in the cf-graph which specifies this predicate.

4) If the complement of this predicate is also a predicate specified by this predicate node, generate the complement of this predicate and expand it to the sum of products form.

5) Set the first literals in the product terms included in the sum of products forms generated by step 3) and step 4) to be the *current literals*. Replace the predicate node in the cf-graph by a predicate node which specifies the first literals of the product terms in the generated sum of products forms as the predicates.

6) Include each current literal in the predecessor literal set of the product term of the sum of products forms, and set the next literal in the product terms to be current literals.

7) Group those product terms of the sum of products forms that have the same predecessor literal set.

8) For each group of product terms create a predicate node which specifies the predicates corresponding to the current literals in the product terms from this group. The transition from the previous predicate node to the new node is determined by the predicate which is specified by the previous predicate node and corresponds to the last predecessor literal from the product terms of the sum of products forms.

9) For the predicates which are specified by the node created in step 8) and whose corresponding literals are the last literals in the product terms of the sum of products forms, find the corresponding original next nodes to be transferred from new predicate node. The original next nodes are determined by the original predicates that include the specified product terms of the sum of products forms.

10) Repeat steps 6) to 9) until the last literal in all product terms of the sum of products forms is processed.

11) Repeat steps 1) to 10) until no predicate of the Boolean expression form is left.

In the above algorithm the expansion of the Boolean expression to a sum of products form is performed by the algorithm developed by Dr.Perkowski [Perk 89b].

§ 3.6 STATE MINIMIZATION WITH NONDISJOINT STATE TABLES

The FSM Synthesizer generated FSM descriptions in two formats: the *Kiss* format and the *Stab* format. Each row of the table in the *Stab* corresponds to a present state, and each column corresponds to an input expression, usually a binary cube. The entry at the intersection of a given row and column specifies a pair of the next state and the output expression determined by the given present state and the given input expression. The *Stab* format generated by the FSM Synthesizer is different from the conventional state table (such tables are explained in [Koha 78]). In the conventional state table, the input expression (usually a product of literals or a minterm) implied by each column is disjoint with the expression implied by any other column. Each input expression implies that the Boolean *AND* of the whole set of corresponding predicates is satisfied (each expression is a product or a sum of products).

The nondisjoint input expressions are in one to one correspondence with the control flow paths of the high level behavioral description. It is defined that two input expressions are disjoint when the two sets of predicate products implied by them can not

happen at the same time. For example, if one predicate implied by a certain input expression is the logic value of the relation ($a := 4$) and another predicate corresponding to a different input expression is the logic value of the relation ($a := 3$), then these two predicates can not happen at the same time, so their corresponding input expressions are disjoint. If the second predicate is ($b := 3$), then the two input expressions are nondisjoint, since without additional restriction there is no reason to assume that these two predicates are in contradiction.

The conventional state table, with the restriction of disjoint input patterns, provides a great deal of redundant information. In consequence, a great deal of computer memory would be wasted during the state minimization computation using such a table.

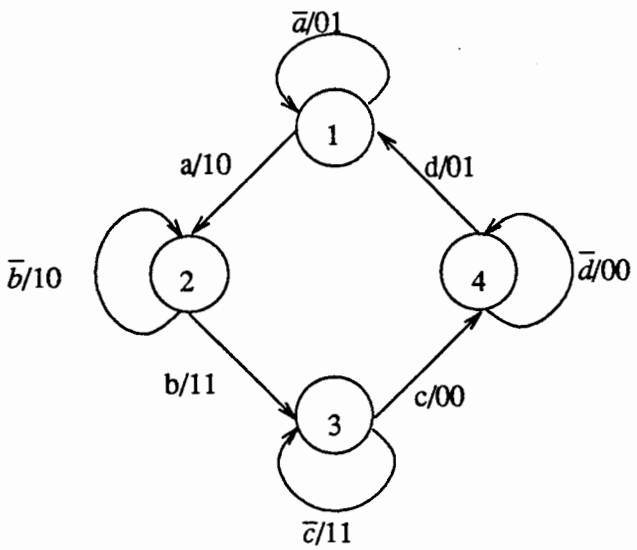


Figure 22. State graph example.

The high level behavioral descriptions are always implicit. For example, if the behavior can be described as *"if a then do ..."* it will never be tediously described as *"if (a* and *b or a* and (*not b*)) *then do ..."*, since the first description includes all the information provided by the second one. Based on the implicit description, the FSM description with nondisjoint input expressions includes all the information of state transitions. When this description is expressed in *Stab* format, it needs, however, some new symbols to

communicate the information which can not be represented by the conventional state table.

TABLE I

EXAMPLE OF DISJOINT STATE TABLE

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1111	1110	1101	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001	0000
1	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
2	3/3	3/3	3/3	3/3	2/2	2/2	2/2	2/2	3/3	3/3	3/3	3/3	2/2	2/2	2/2	2/2
3	4/0	4/0	3/3	3/3	4/0	4/0	3/3	3/3	4/0	4/0	3/3	3/3	4/0	4/0	3/3	3/3
4	1/1	4/0	1/1	4/0	1/1	4/0	1/1	4/0	1/1	4/0	1/1	4/0	1/1	4/0	1/1	4/0

* For output pattern: 0 - 00, 1 - 01, 2 - 10, 3 - 11.

TABLE II

EXAMPLE OF NONDISJOINT STATE TABLE

	1	2	3	4	5	6	7	8
	1---	0---	-1--	-0--	--1-	--0-	---1	---0
1	2/2	1/1	-1/*	-1/*	-1/*	-1/*	-1/*	-1/*
2	-1/*	-1/*	3/3	2/2	-1/*	-1/*	-1/*	-1/*
3	-1/*	-1/*	-1/*	-1/*	4/0	3/3	-1/*	-1/*
4	-1/*	-1/*	-1/*	-1/*	-1/*	-1/*	1/1	4/0

*For output pattern : 0-00, 1-01, 2-10, 3-11.

In TABLE I and TABLE II, two equivalent state tables are presented which describe the state graph shown in Figure 22. TABLE I is a conventional state table. TABLE II contains columns corresponding to nondisjoint input expression. The four bits input expression reprsents the logic value of the four variables *a*, *b*, *c* and *d* respectively. They are defined as the *disjoint format* and the *nondisjoint format,* respectively. Each state transition specified in the state graph is described by the specifications determined by a number of input expressions in TABLE I. For example, in Figure 22 the state transition from the present state 1 under the predicates *a*, which is to a next state 2 and gives

the output signal 10, is described by the specifications in row 1 and columns from 1 to 8 which is determined by the input expressions 1111, 1110, 1101, 1100, 1011, 1010, 1001, 1000, respectively. The state transitions specified in state graph are one-to-one map to the specifications defined in TABLE II. The same state transition in above example is described by the specification in row 1 and column 1 which is determined by the input expression 1--- in TABLE II.

In TABLE II, some next states and output patterns are represented by number -1 and symbol *, respectively, which indicates *unspecified transitions*. The unspecified transition of this kind implies the multiple state transition descriptions defined by the corresponding input expression. The *actual transitions* will be determined by the other input expressions which are nondisjoint to this input expression. This is the case different to a *don't–care* case (as it is known, *don't–care* means that the current input expression will never happen under the present state). Therefore, a new term *multi–care* is introduced here for this special unspecified case. For example, in TABLE II at the intersection of row 1 and column 4 a multi-care is appeared. It means that the state transition from the present state 1 and under the corresponding input expression -1-- is undetermined. It depends on the first bit of the input expression. If the first bit is 1, then the input expression will be 11--. The state transition defined by this input expression is implied by the specified state transition determined by the input expression 1---, which is 2/10. The state transition defined by the input expression 01-- is implied by the specified state transition determined by the input expression 0---, which is 1/01. Both 1--- and 0--- are nondisjoint to -1--.

Besides the nondisjoint state table a matrix, called *disjoint relation matrix,* to indicate the relations of the columns is needed. Instead of comparing two input expressions the relation of two corresponding columns can be found by checking the matrix. The disjoint relation matrix for TABLE II is

11000000
11000000
00110000
00110000
00001100
00001100
00000011
00000011

The value at the position of row i and column j the matrix indicates the disjoint relation of columns i and j in TABLE II. The value 1 means disjoint and 0 means nondisjoint.

It can be seen that the *nondisjoint format* state table is more compact than the conventional table. For the input expression of n bits, the total number of disjoint expressions would be 2^n - the corresponding conventional state table would have up to 2^n columns. In practical designs, huge conventional state table would be always involved, which is exactly what the *nondisjoint* state table tries to avoid.

The reader interested in the theorems and programs for state minimization as well as their applications can be referred to [Zhao 89] for details. The program of state minimization for a conventional state table can not be directly applied to a *nondisjoint format* state table.

Formally a FSM can be defined as 5-tuple (X, Y, Z, δ, λ) where,

$X = x_1, x_2, ..., x_{\|X\|}$	set of input patterns,
$Y = y_1, y_2, ..., y_{\|Y\|}$	set of internal states,
$Z = z_1, z_2, ..., z_{\|Z\|}$	set of output patterns,
$\delta: X \times Y \rightarrow Y$	next state function,
$\lambda: X \times Y \rightarrow Z$ ($\lambda: Y \rightarrow Z$)	output function.

With the above definition, the following property should be fulfilled for all FSMs.

If $x_i \cap x_j \neq \varnothing$, then

$$\forall(y_i \in Y)[\delta(x_j, y_i) = \delta(x_k, y_i), \lambda(x_j, y_i) = \lambda(x_k, y_i)]$$

This property means that the state transitions defined by the same present state and the nondisjoint input expressions should be always the same. This property holds because for two nondisjoint input expressions there exists certain input patterns which are implied by both of these expressions. If the state transitions determined by these two input expressions were different, the *conflict state transition specifications* would exist for this implied input pattern.

For example, let us assume that the state transition is defined from the present state m to the state n under input 1- and to the state k under input -1. Under input 11 which is implied by both 1- and -1, the contradict transition specifications happen.

In the discussed Synthesizer, the generated FSM descriptions always fulfill this property. Since for any two nondisjoint input expressions, only the state transitions determined by one of them are specified by the FSM description generated by the Synthesizer. The state transitions determined by its nondisjoint input expressions are implied by the specified state transition descriptions. The state minimization program, therefore, does not need to check the consistency of the state table.

Let the set of rows of the state table is denoted by $R = \{r_i \mid i=1,2,..., \mid Y \mid\}$ and the set of columns is denoted by $C = \{c_j \mid j=1,2,..., \mid X \mid\}$, The corresponding relations between the rows and states as well as the columns and input expressions are represented as

$r_i \rightarrow y_i$

and

$c_j \rightarrow x_j$

where $\rightarrow$ means *corresponds*.

Definition 3.1 The two *nondisjoint columns,* denoted by NDis(c_i, c_k) = 1, on the state table are the columns whose corresponding input expressions are nondisjoint. The definition can be formulated as

$$NDis(c_i, c_k)=1 \Leftrightarrow x_i \cap x_j \neq \varnothing.$$

In TABLE II, the columns 1, 3, 5 and 7 as well as columns 2, 4, 6 and 8 are nondisjoint.

Definition 3.2 A column is said to be *covered* by a set of columns if its corresponding input expression is covered by the union of the input expressions of the columns in the covering set and it is nondisjoint with all the columns in the covering set. This definition can be expressed as follows.

$$c_i \subseteq \cup \{c_k \mid k \neq i, NDis(c_i, c_k)=1\} \Leftrightarrow x_i \subseteq \cup \{x_k \mid k \neq i\}$$

In TABLE II, column 3 is covered by the union of the columns 1 and 2.

Definition 3.3 An *entry(i,j)* on a state table, represented by the symbol e_{ij}, is defined as the position on the intersection of row i and column j.

Definition 3.4 For a given row i, an entry e_{ij} is nondisjoint to another one e_{ik} only if column j is nondisjoint to column k. Consequently an entry e_{ij} is covered by a set of entries $\{e_{ik_1}, e_{ik_2}, \ldots, e_{ik_m}\}$, if the column j is covered by the union of the columns k_1, k_2, ... and k_m. The entries in the covering set are called the *covering entries*.

If the next state and the output pair in an entry are multicares, then this entry is called a *multi-care entry;* otherwise it is a *non-multi-care entry*.

The nondisjoint state tables should have the following property:

For a multi-care entry, there always exists a set of non-multi-care entries to cover it.

For example, in TABLE II, the entry e_{13} is covered by $\{e_{11}, e_{12}\}$, and both e_{11} and e_{12} specify the definite transition descriptions.

In the case of an multi-care entry, if a set of non-multi-care entries can not be found to cover it, then it would mean there exist certain input expressions under which the actual state transitions are neither *don't–cares* nor specified transitions. This would

mean the loss of information in state transition description. For example, if for a present state m, under the input 1- the specified state transition is a multi-care, and non-multi-care transition is only specified under the input 10 which is to a next state n. Then the transition information about the input 11 is unknown.

The problem of state minimization is to find a minimal cardinality group of subsets of the internal states Y so that these subsets would include Y and inside each subset all states were compatible. Two states are compatible if for any sequence of input patterns X_s,

$$\lambda(y_i, x_k) = \lambda(y_j, x_k)$$

where $x_k \in X_s$. The above equation requires that for a *disjoint format* state table, if the states i and j are compatible, then the output expressions specified by the entries e_{ik} and e_{jk} should be the same or one of them is a don't-care, where k is any one of the column numbers.

For a *nondisjoint format* state table, the comparison of two output expression should consider the case in which both of them are multi-cares as well as the case that only one of them is a multi-care.

Since the *multi-care entry* would be covered by a set of *non-multi-care entries,* the result of the comparison of two multi-cares would be determined by the comparisons of their covering entries. For example, in TABLE II, the comparison of the entries e_{17} and e_{27} is determined by the comparisons of all the entries from column 1 to column 6 in rows 1 and 2, because these columns are all nondisjoint to column 7 and the entries in these columns will be compared in any case. Therefore the comparison of two multi-cares can be ignored, in other words they are assumed to be equal.

In the case of a single multi-care involved in the comparison, if the output expression specified by the non-multi-care entry is a don't-care, then the multi-care and the nonmulti-care entries are considered to be equal; otherwise all the entries in the covering

set of the given multi-care entry have to be compared to the given non-multi-care entry. For example, in TABLE II, to compare the entry e_{14}, which is -1/*, to the entry e_{24}, which is 2/10, it has to compare e_{24} to e_{11} and e_{24} to e_{12}, respectively. Only if all the output expressions specified by the entries in the covering set are equal to the output expression of the *non-multi-care entry,* is the output in the multi-care entry considered to be equal to the output of the non-multi-care entry.

In state minimization with the state table description, the algorithm of finding compatible groups is actually based on the process of checking out incompatible states [Zhao 89]. It first identifies those incompatible states because of the different output expressions appeared in their next states under the same input. Then the incompatible states of the incompatible next states under the same input are found. For the conventional state table the algorithm for the first stage of finding the incompatible states is shown below.

```
for (i = 1, i = nr, i++)
        for (j = i+1, j = nr, j++)
                for (k = 1, k = nc, k++)
                        compare (out(i, k), out(j, k));
```

where *nr* and *nc* represent the number of the rows and the number of the columns of the state table, respectively.

The modified algorithm for the same purpose with the state table containing multi-care is shown as follows.

Algorithm 3.4

```
for (i = 1, i = nr, i++)
        for (j = i+1, j = nr, j++)
                for (k = 1, k = nc, k++)
                        if (out(i, k) = * and out(j, k) ≠ *)
```

{for *all* (*entry* $(i, k) \cap$ *entry* $(i, x) \neq \phi$

and *out* $(i, x) \neq *$)

compare (*out* (i, x), *out* (j, k))};

else if (*out* $(i, k) \neq *$ and *out* $(j, k) = *$)

{for *all* (*entry* $(j, k) \cap$ *entry* $(j, x) \neq \phi$

and *out* $(j, x) \neq *$)

compare (*out* (j, x), *out* (i, k))};

else

compare (*out* (i, k), *out* (j, k));

The above algorithm modifies the algorithm to find the incompatible states which have different outputs in next states under the same input. The rest of the algorithm for the disjoint state table minimization can be applied to the nondisjoint state table without any modification.

For the nondisjoint state table TABLE II, the state minimization procedure first tries to find the incompatible states of different outputs by comparing each pair of rows from column to column. It first compares the entries e_{11}, which is 2/10, and e_{21}, which is a multi-care. By checking the disjoint relation matrix, it is found that entries e_{23} and e_{24} are nondisjoint to entry e_{21} and both of them specify the nonmulti-care transitions 3/11 and 2/10, respectively. Therefore, the entry e_{11} has to compare with these two entries. Since the output of e_{11}, which is 10, is not compatible with the output of e_{23}, which is 11, the states 1 and 2 are incompatible. The state 1 is incompatible with the states 3 and 4 can be found when comparing the entry pairs e_{11} and e_{35}, e_{11} and e_{47}, respectively. By the same procedure, it can be found that all the four states are incompatible because of the incompatible outputs. The TABLE II, therefore, can not be minimized.

§ 3.7 STATE ASSIGNMENT FOR NONDISJOINT FORMAT FSM

Most of the state assignment programs deal with disjoint state tables. The influence of the selection of the nondisjoint state table versus the disjoint state table for the state assignment task varies according to the different state assignment algorithms. In this thesis the analysis of the influences to two well known state assignment algorithms is discussed. The first algorithm was developed by Armstrong [Arm 62]. Since it has not been assigned a name, we call it the *Arm's* algorithm. The second algorithm was developed by Micheli [Mich 85] and applied to the program *Kiss* so that we call it *Kiss* algorithm.

Arm's algorithm is based on the generation of the so called *adjacency graph* which consists of a set of state nodes as well as the weighted edges between those nodes. The state encoding is carried out according to the principle that the states linked by the heavier weighted edges are likely to be assigned the codes of shorter Hamming distance. The Hamming distance is defined as the number of different values (0 and 1) in the same positions of the codes. If for the nondisjoint state table, there exists an algorithm to generate the same *adjacency graph* as the one generated with the *Arm's* algorithm, then it can be assumed that the same state assignments are found by using this algorithm.

The edge weights can be of two kinds, which are called the *first edge weight* and the *second edge weight.* The first edge weight between two states, A and B, is defined as the number of the state transitions from a common present state C to states A and B as the next states under the two adjacent inputs. The adjacent inputs are the inputs of distance one. For instance, the distance of inputs 110 and 100 is one since they have only one different bit. The second edge weight between two states is the number of the state transitions from two present states A and B to a common next state C under the same inputs. The generation of an adjacency graph can be illustrated by the next example.

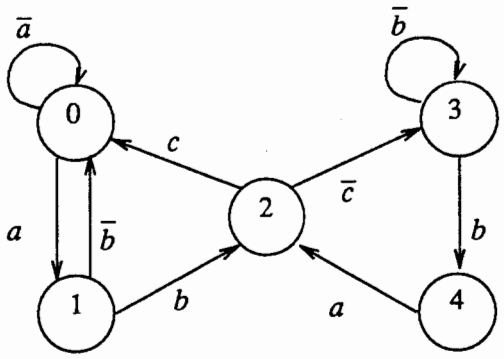


Figure 23. Example of state graph.

TABLE III

DISJOINT STATE TABLE FOR THE EXAMPLE IN FIGURE 21

	abc	$ab\bar{c}$	$a\bar{b}c$	$a\bar{b}\bar{c}$	$\bar{a}bc$	$\bar{a}b\bar{c}$	$\bar{a}\bar{b}c$	$\bar{a}\bar{b}\bar{c}$
0	1	1	1	1	0	0	0	0
1	2	2	0	0	2	2	0	0
2	0	3	0	3	0	3	0	3
3	4	4	3	3	4	4	3	3
4	2	2	2	2	d	d	d	d

* d - don't-care

Example 3.3:

Figure 23 is the state diagram of an FSM. Its corresponding disjoint and nondisjoint state tables are presented in Table III and Table IV respectively. By checking from the disjoint state table, the adjacency graph which contains only the first edge weights is shown in Figure 24 (a). The edge between state nodes 0 and 1 is weighted 4 since there are four adjacent input pairs (abc and $\bar{a}bc$, $ab\bar{c}$ and $\bar{a}b\bar{c}$, $a\bar{b}c$ and $\bar{a}\bar{b}c$, $a\bar{b}\bar{c}$ and $\bar{a}\bar{b}\bar{c}$) which lead the state transitions from the present state 0 to the next state pair 1 and 0. The adjacency graph which contains only the second edge weights is presented in Figure 24 (b). The weight between states 0

TABLE IV

NONDISJOINT STATE TABLE FOR THE EXAMPLE IN FIGURE 21

	a—	$\bar{a}$—	—b—	—$\bar{b}$—	—c	—$\bar{c}$
0	1	0	-1	-1	-1	-1
1	-1	-1	2	0	-1	-1
2	-1	-1	-1	-1	0	3
3	-1	-1	4	3	-1	-1
4	2	d	-1	-1	-1	-1

* d - don't-care
* -1 - multi-care

and 1 is 2 since the state transitions from the pair of present states 0 and 1 are both to the next state 0 under the input $\bar{a}\bar{b}c$ or $\bar{a}\bar{b}\bar{c}$. The final adjacency graph is the combination of the two graphs, which is shown in Figure 24 (c). The weight of the corresponding edges are added to create an edge of the new combined graph. The encoding procedure will assign the codes of the shortest distances to state pair 0 and 1 as well as the pair 0 and 2.

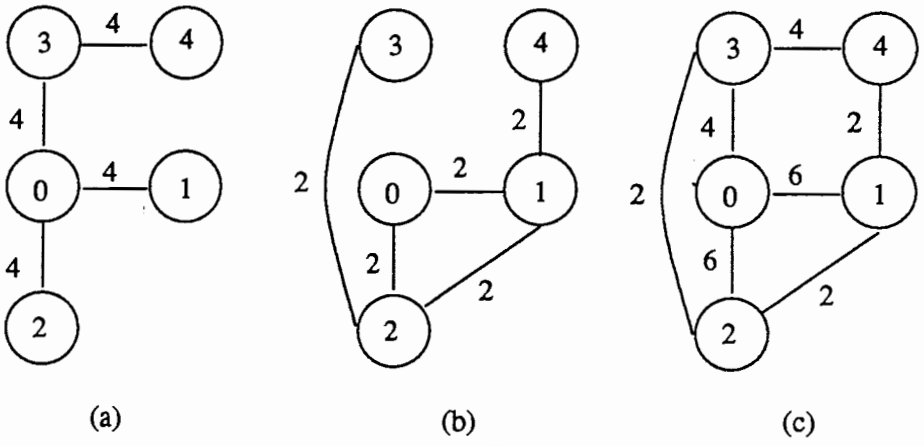


Figure 24. Adjacency graph for the example in Figure 23.

For the nondisjoint state table, the adjacent inputs are overlapped. Therefore, the adjacency graph generation routine has to be modified. The first edge weight for two states can be also checked out by the state transitions from a common present state to these two states as the next states under the pair of adjacent inputs. The value of the weight, however, should be the number of all the state transitions determined by those inputs which are nondisjoint to the pair of adjacent inputs. For example, the weight between nodes 0 and 1 in Figure 24(a), which is 4, can also be found by checking the nondisjoint state table in Table IV. From the first row of the state table a pair of adjacent inputs a— and $\bar{a}$— is found which specifies one case of the first weight edge. Since there are four inputs ($-b-$, $-\bar{b}-$, —c and —$\bar{c}$) that are nondisjoint to both the adjacent inputs, the weight is 4.

The second edge weight for two states can be checked out by the state transitions from these two states as the present states to a common next state under a pair of inputs. The value of the weight is the total number of the state transitions determined by all the inputs which are nondisjoint to both inputs from the input pair. For example, the weight between nodes 0 and 1 in Figure 24(b) is 2, since two inputs —c and —$\bar{c}$ are nondisjoint to both inputs $\bar{a}$— and $-\bar{b}-$, both of which lead the transition to the next state 0 from the present states 0 and 1. Obviously the final Adjacency Graph generated from the nondisjoint state table is the same as the one generated from the disjoint state table. The results of the state encoding would be the same for the same Adjacency State Table. It can be concluded that the influence of the nondisjoint FSM description on the *Arm's* algorithm can be overcome by the modification of the Adjacency Graph generation procedure.

The *Kiss* algorithm performs the optimal state assignment to minimize PLA which implements combinational components of the FSM. The algorithm is divided into two stages. At first, the logic minimization is performed on a symbolic (code independent) representation of the combinational components of the FSM. Then the state encod-

ing is carried out on the symbolically minimized FSM description. Symbolic representation can be represented by a multiple-valued logic specification. Different symbols correspond to different logic values.

0 *state* –1 *state* –6 00
0 *state* –2 *state* –5 00
0 *state* –3 *state* –5 00
0 *state* –4 *state* –6 00
0 *state* –7 *state* –5 00
1 *state* –4 *state* –6 10
1 *state* –7 *state* –6 10

(*a*)

0 1000000 0000010 00
0 0100000 0000100 00
0 0010000 0000100 00
0 0001000 0000010 00
0 0000001 0000100 00
1 0001000 0000010 10
1 0000001 0000010 10

(*b*)

Figure 25. FSM example in *Kiss* and its multiple-valued specification.

For example, the *Kiss* format description shown in Figure 25 (a) can be rewritten to the multiple-valued specification shown in Figure 25 (b), in which each state value is represented by a positional cube. The symbolic minimization problem is to find the minimal symbolic cover of the function. This is carried out by the consideration that any positional cube representing a present state is of a multiple value and any positional cube corresponding to a next state is part of a multiple-output expression of the logic function. Under this assumption, the input codes of the function are composed of two parts: the input expression for FSM and the state code. The first part is the binary-valued cube. The minimal symbolic cover of the example in Figure 25 (a) is shown in Figure 26. After

```
0 0110001 0000100 00
0 1001000 0000010 00
1 0001001 0000010 10
```

Figure 26. Minimal symbolic cover of the example in Figure 25 (a).

symbolic minimization, a set of new positional cubes is generated which are combinations of the positional cubes representing the present states. The new positional cubes form a so called *constraint matrix* which is denoted by

$$A = \begin{bmatrix} a_{1.} \\ a_{2.} \\ \cdots \\ a_{n_p.} \end{bmatrix} = [a_{.1} \mid a_{.2} \mid \ldots \mid a_{.n_s}] = \{a_{ij}\}.$$

The constraint matrix A for the symbolic cover in Figure 26 is

$$A = \begin{bmatrix} 0110001 \\ 1001000 \\ 0001001 \end{bmatrix}$$

The state encoding is to find such a code matrix

$$S = \begin{bmatrix} s_{1.} \\ s_{2.} \\ \cdots \\ s_{n_s.} \end{bmatrix} = [s_{.1} \mid s_{.2} \mid \ldots \mid s_{.n_b}] = \{s_{ij}\}.$$

that

$$(\overline{a_{.j}} \cdot s_{j.}) \Lambda (A \cdot S) = \overline{F}^{j} \Lambda F = \begin{bmatrix} \overline{f}_{1.}^{j} \Lambda f_{1.} \\ \overline{f}_{2.}^{j} \Lambda f_{2.} \\ \cdots \\ \overline{f}_{n_p.}^{j} \Lambda f_{n_p.} \end{bmatrix} = \Phi.$$

The definitions of operations Λ and $\cdot$ are given in [Mich 85]. The difference between the disjoint and nondisjoint FSM descriptions in *Kiss* format is the existence of some state transition descriptions in disjoint format that are different only in input expressions. For example, the *Kiss* format descriptions corresponding to state transitions from the present state 0 in the state table descriptions in Table III and Table IV are shown in Figure 27(a)

000 *state* –0 *state* –0
001 *state* –0 *state* –0
010 *state* –0 *state* –0
011 *state* –0 *state* –0
100 *state* –0 *state* –1
101 *state* –0 *state* –1
110 *state* –0 *state* –1
111 *state* –0 *state* –1
(*a*)

0— *state* –0 *state* –0
1— *state* –0 *state* –1
(*b*)

Figure 27. *Kiss* descriptions for the example in Figure 23.

and (b) respectively. The description of the nondisjoint FSM combines some rows in the description of the disjoint format which are only different in input expressions, for instance the rows 1, 2, 3, 4 in Figure 27.(a). In the procedure of finding minimal symbolic cover, these rows are expected to be eventually combined. So the nondisjoint description saves some work for the state assignment by *Kiss* algorithm.

CHAPTER IV

THE FSM SYNTHESIZER OVERVIEW

The FSM Synthesizer is a part of DIADES system. It takes cf-graph description from a specified file as the input and generates the *FSM description* in both *Kiss* format and *state table (Stab)* format as well as logic description in *truth table* format or *Eqn* (a set of Boolean equation) format. The input file *inn* within DIADES system is either: generated by the behavioral description ADL compiler *TAG* [Perk 89], originates from the p-graph high-level transformations [Perk 82], [Yang 89], or is specified by the user in a low level cf-graph and structure description in language GRAPH88 (see CHAPTER II). The FSM Synthesizer is composed of five parts, as shown in Figure 28. They communicate through user-readable text files. Also each part can be accessed by the user separately, according to the various design level requirements.

The first part, *FGEN,* takes the p-graph description as the input data and generates the *FSM description.* The p-graph contains both the data flow and the control flow information. The *FGEN* assigns states to the operations specified in the p-graph and generates the state transition description. It also encodes the input and output signals that interface the data path and the control unit. It supports, as options, several *CU/DP interface design styles*. To accommodate such CU/DP interface design styles it is sometimes necessary to perform transformations on p-graph.

The FSM description generation function is carried out by the following procedures.

- *GRAM:* This program checks the predicate list to find out those predicates which are Boolean expressions of a set of other predicates. Whenever such a predicate is

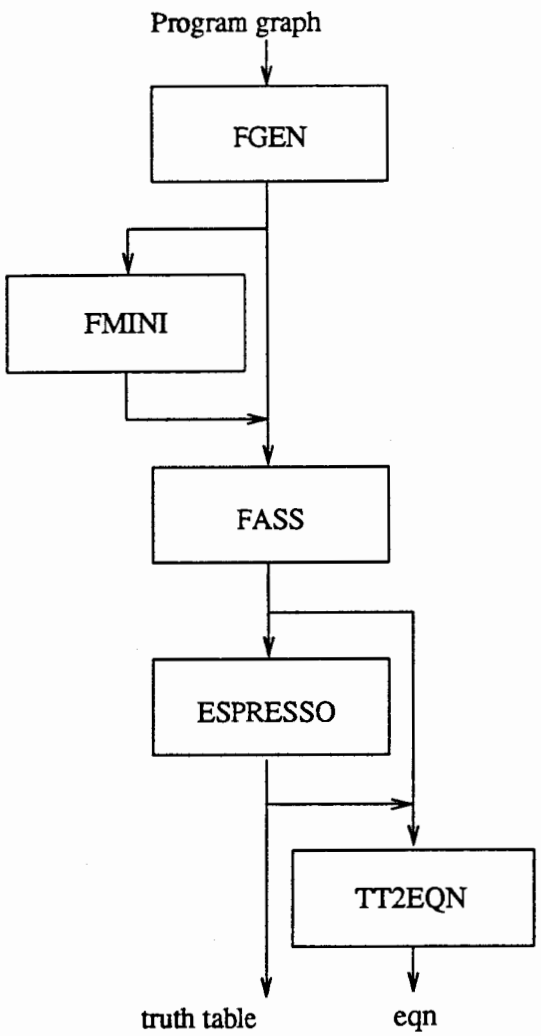


Figure 28. Diagram of the FSM Synthesizer.

found, the graph description lists contained in **coplisset**, **plisset**, **nolisset** and **anlisset** are modified to make each predicate specified by a predicate node being a relation or variable only. The modified graph is described by the new internal lists *coplis*, *plis*, *nolis*, *anlis* corresponding to one of the graph descriptions read out from the **coplisset**, **plisset**, **nolisset** and **anlisset** lists, respectively. This program can be optionally called by the user to get a different variant of FSM.

TRANSLATION: This program takes one of the cf-graph descriptions, either read

out from internal list **coplisset**, **nalisset**, **plisset**, **nolisset** and **anlisset**, or generated by *GRAM,* as the input, and creates a state transition description. This state transition description consists of a transition list (*relgrliststar*) and a list of relationships between each transition (*relalis*). The format of these two lists will be described in CHAPTER V. The relationship specifies the predicates under which the transition occurs and the operations that need to be executed during the next state. Both predicates and operations at this stage are represented by node numbers. For a sequential graph, the generated description is actually a *state transition graph.* The state transitions are described by the *transition list* and the input-output relations are described by the *relation list.* For the parallel graph, the transition and relation lists describe a *compact parallel graph.* In this compact graph the predicates between two operational nodes of the original parallel graph are combined as a single predicate, being the logic *AND* of these predicates. Besides these two lists, some extra information is stored in a number of lists needed by the Parallel to Serial Conversion procedure *CONVERSION*.

CONVERSION: This procedure will convert the transition and relation lists generated by program *TRANSLATION* for a parallel control flow to the sequential state transition description. The converted description has the same format as the description generated by program *TRANSLATION* for a sequential graph.

RELATION: The relationship of state transition description will be converted to a readable description by this program. Instead of the node number, the actual meanings of the predicates and operations are presented in the description generated by this procedure. It will also give out an error message whenever a *conflicting register transfer* occurs during the transition. The conflicting register transfers are detected when different values are assigned to the same variable during the same machine cycle. This program is useful only for user interface,

debugging and as the input to the verification program.

TRANS: This procedure reads the state transition list as well as the relationship list and generates the *Kiss* format and *Stab* format FSM description. These two formats will be stored in a *KISSOUT* file and a *FSMtable* file respectively. Another file called *IMPLY* specifies the implication of each bit of input and output in *Kiss* format as well as the implication of input code for each column on *Stab* format. For the requirement of the state minimization program[Zhao 89] in Fortran, a *fort.1* file is generated which contains the read-write format description of the *FSMtable.*

The second part, the state minimization program *FMINI,* performs state minimization of FSMs. It reads either the *Stab* format or the *Kiss* format description, and minimizes the FSM : either for both states and inputs or only for states. If the designer's goal is to minimize only the number of the internal states then the corresponding variant is called which gives the guarantee that the solution has the minimum number of internal states. If the designer attempts to minimize both the input states and the internal states, the solution can cause smaller total FSM circuit area (for both FSM's main logic and its *input encoder),* however the algorithm does not warranty the minimum number of inputs or the minimum number of internal states [Zhao 89]. During its execution the *Kiss* format will be first converted to the state table format. The output of this program is a minimized *Kiss* format description which is stored back to *KISSOUT* file. Another *decodett* file is generated which contains the truth-table format description for the input decoding function. More information about this part can be found in [Zhao 89].

The third part of the synthesizer, *FASS,* performs state assignment of FSM's internal states [Per 89]. The input for this part is the *Kiss* format description which is the data format created by either *FMINI* or *FGEN,* and a standard truth table format description will be generated at this stage and put into "ttfile" file. This program is based on the

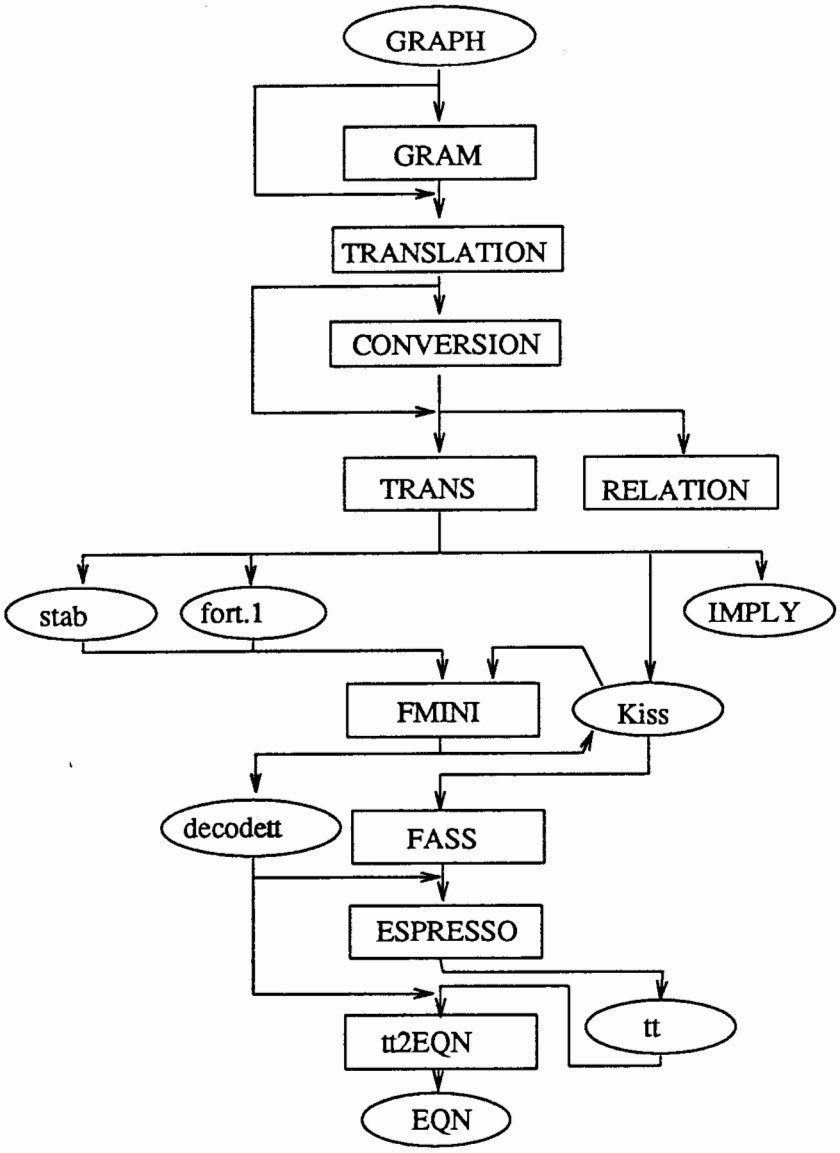


Figure 29. Diagram of program connection.

idea, which is the extension of the Armstrong's algorithm [Arm 62], and uses very efficient heuristic algorithm to solve the *quadratic assignment problem.* It permits to assign machines with more than 100 states and optimizes the assignment for multi-level logic, while keeping the number of flip-flops under partial user's control.

The fourth part of FSM Synthesizer is the Logic Synthesizer. Currently the U.C. Berkeley tool *ESPRESSO* [Bray 84] is adopted to serve for this task, but we plan to con-

nect other programs that use truth tables (*tt* format) as the input. They are: PALMINI for PALs minimization [Nguy 87], EXORCISM for mixed polarity generalized Reed Muller Forms [Hell 87], and DECOMP for decomposition [Perk 88]. The minimized logic description is in a "MINItt" file.

The last part *TT2EQN* is the format conversion program, which converts the truth table logic description to *Eqn* (set of equations) description. The purpose of the conversion is to provide versatile data formats for lower level synthesis procedures. Currently we work on other netlist formats converters, including language M from Silicon Compiler Systems Company (SCS).

The diagram of the program connections as well as the data files generated is presented in Figure 29.

CHAPTER V

OUTPUT DESCRIPTIONS OF THE SYNTHESIZER

The results of FSM based control unit generation are the state transition description and the Boolean function description. The state transition description is in *Kiss* format and *state table* format. These two data formats are defined by University of California [Walt 85]. The Boolean function description is in *truth table (tt)* format and *Eqn* format. With these formats the results can be versatilely interfaced with other CAD software. Since both *Kiss* format and *Stab* descriptions have been discussed in CHAPTER II, in this chapter only *truth table* format and *Eqn* format, as well as an internal *compact* format are presented.

1) *Truth table* format description

The *truth table* format is a PLA oriented description of two level Boolean switching function. This is described as a character matrix with keywords embedded at the head and the end of the file. The keywords consist of the following items.

- .i [d] Specifies the number of input variables.
- .o [d] Specifies the number of output functions.
- .p [d] Specifies the number of product terms.
- .ibl [s] Specifies the labels of input variables.
- .obl [s] Specifies the labels of output functions.
- .e Specifies the end of the description.

Both ".ibl" and ".obl" can be default and only ".e" is specified at the end of the file.

The logic description is specified by the matrix. Each row of the matrix is a term

of the function. A term is represented by a "cube" which can be considered a compact representation of an algebraic product term. A cube has an input part and an output part. The matrix has the structure like PLA. The rows in the matrix correspond to the rows in PLA, the input part corresponds to the row in *AND* plane and the output part corresponds to the row in *OR* plane.

Each position in the input part corresponds to an input variable where an 0 implies that the corresponding input literal appears complemented in the product term, a 1 implies that the input literal appears uncomplemented in the product term, and - implies that the input literal does not appear in the product term.

Each position in an output part corresponds to an output function where a 0 implies that the corresponding input expression is a maxterm of the function, a 1 implies that the corresponding input expression is a minterm of the function, and a - implies a don't-care term. An example of this format is shown in Figure 30.

```
--100101  0001100
1-0-1-00  0110000
1--0-110  1000010
0-00-101  1000100
-0-10101  1001000
-1--0101  1001000
-1--1101  1000100
0-11--01  1000001
```

Figure 30. Example of the *truth table* format.

2) *Eqn* format description

The *Eqn* format is a standard cell oriented description of a Boolean switching function. This is described as a set of logic equations. An example of the *Eqn* format is shown in Figure 31. In this format, symbol | represents logic *OR*, symbol & represents logic *AND* while the symbol ! represents the complement. The commonly included product forms in the sum of products forms are separately listed as the internal product forms. The sum of products forms are actually the sum of the internal products, the

products of the inputs as well as the products of internal products and inputs. The results of the internal product forms are labeled by Q[d], where [d] is the order number as an index.

```
Q1 = IN1 & ! IN2 & ! IN4
Q2 = IN1 & IN2 & ! IN3
Q3 = ! IN1 & IN2 & IN3 & IN4
OUT1 = Q1 | Q2 | IN5 & IN2
OUT2 = Q1 | Q3
OUT3 = Q2 & IN5 | Q3
```

Figure 31. The example of *Eqn* format.

In Figure 31, the Q1, Q2 and Q3 are internal products which are commonly included in part of the sum of product forms OUT1, OUT2 and OUT3. The (Q2 & IN5) in the sum of products form OUT3 is the product of the internal product and input.

In DIADES system, the *Eqn* format is converted from the *truth table* format. The variables in the internal product forms, as well as the outputs from the sum of products forms, are the input labels and the output labels specified in the *truth table* format, respectively. When the labels of input and output signals in the *truth table* format are not declared, the default labels, IN[d] and OUT[d], are used, respectively.

3) *Compact* format description

In the generation of the FSM description an internal transition description is created. For the sequential cf-graph, this description describes the state transition graph in terms of node numbers. When parallel graph is processed, this description provides a compact parallel flow graph description for the Parallel to Sequential Conversion procedure. This internal transition description is generated in a so called **compact** format. The compact format is composed of a number of following lists.

a) The list of transitions (*relgrlistar)

Each element of this list describes a transition. The element consists of three items, each of the form *(current point (success point, transition number))* For the FSM generation starting from a sequential graph, each point is a state. The list describes, therefore, the state transition graph. In the case of a generation starting from a parallel graph, each point is an operational node which is represented by the node number.

b) The list of relations (*relalis)

This list contains the elements composed of three items. Each element describes a relation *(tn (opn) (stn))*, where *tn* denotes the transition number, (*opn*) is a list of operations performed after the transition and (*stn*) is a list of predicates determining the transition. The operations and the predicates are represented by the operation numbers and the predicate numbers (or complements of predicate numbers), respectively. Both of the numbers are specified in the cf-graph. For two complementary predicates, only one number to represent one of them is specified in the cf-graph. When referring another predicate, the complement of that number (not number) is used. In the case of the generation process starting from the sequential graph, this list specifies the input and output information of the state transition.

c) The list of pass *FORK* transition

This list describes the transitions between the operational nodes before and after the *FORK* node. Each element of it is composed of the before *FORK* node number and a subset of its immediate successors determined by a *leading-set* (see § 3.4). It is of the form (<node before *FORK* > <list of immediate successors>).

d) The list of nodes between *FORK* and *DAND* (*dandlisstar)

All the operational nodes on the path from *FORK* to *DAND* are stored in this list.

e) The list of nodes between *FORK* and *DEXOR* (*dexorlisstar)

All the operational nodes on the path from *FORK* to *DEXOR* are put in this list.

f) The list of passing *DAND* transitions (*dandpair)

This is the set of all the transitions passing *DAND*. Each element of the set is a pair of operational nodes which is of the form

(<node before *DAND* > <node after *DAND* >).

The operational nodes are also represented by node numbers.

g) The list of passing *DEXOR* transitions (*dexorpair)

This is the set of all the transitions passing *DEXOR*. Each element of the set is a pair of operational nodes which is of the form

(<node before *DEXOR* > <node after *DEXOR* >).

The operational nodes are also represented by node numbers.

h) The list of *DROP* nodes (*droplisstar)

All the *DROP* nodes are stored in this list.

The lists from c) to g) are useful only in the FSM generation process which starts from a parallel cf-graph.

An example to explain the above internal description will be illustrated in Figure 32. Figure 32 (a) presents a part of a parallel graph. Its compact graph is presented in Figure 32 (b). In the parallel graph, the node 7 is a *FORK* node, node 15 is a *DROP* node and node 16 is a *DAND* node. In the compact graph, the predicates leading the transitions from the operational nodes to their immediate successors are described in the relation list *relalis. The parallel relations described by the control nodes in the parallel graph are described by the passing *FORK*, passing *DAND* transition lists in the compact graph. In the compact graph, therefore, only the operational nodes are left. The numbers of the operational nodes are reordered since the predicate nodes are deleted. The passing *FORK* list, *forkpair, contains the elements representing the parallel transition node

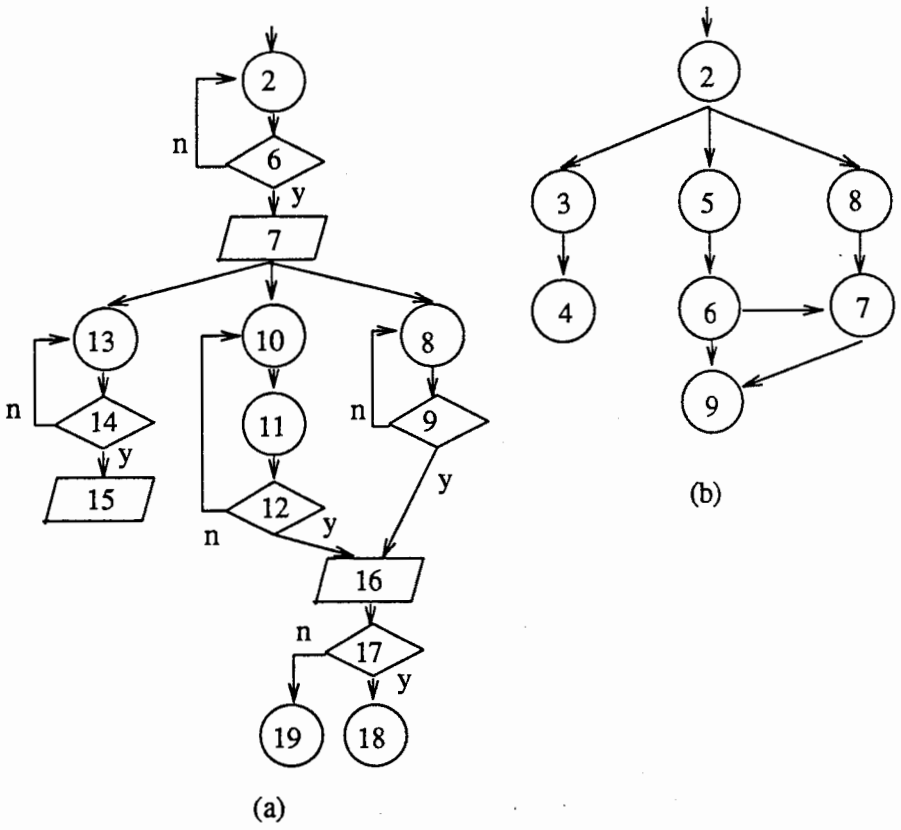


Figure 32. Example of parallel graph and its compact graph.

pairs (2 3), (2 5) and (2 8). The nodes 3, 5, 8 execute operations specified by the nodes 13, 10, 8 in the parallel graph respectively. The passing *DAND* list, *dandpair, describes the transitions from nodes 11 and 8 in the parallel graph to the nodes 18 or 19 depending on the predicates specified by the nodes 9, 12, 17. Since the nodes 11, 8, 18, 19 are reordered as nodes 6, 8, 7, 9, the passing *DAND* list contains the node pairs (6 7), (6 9), (8 7) and (8 9). The predicates determining these transitions are also provided in the relation list. The *DROP* list has only one element which is node 4 in compact graph. The details about this example can be referred by the signal delay processor design example in CHAPTER VI as well as the data in APPENDIX B.

CHAPTER VI

TWO DESIGN EXAMPLES

§ 6.1 THE CONTROL UNIT OF EIGHT-INSTRUCTION CPU

This example is adapted from [Hayes 88]. Although the eight-instruction CPU is too simple to have a practical use, the design of its control unit demonstrates the procedures which are common to universal microprocessor CPU's.

The flowchart describing the instruction fetch cycle common to all instructions, as well as the distinct executions, is shown in Figure 32. The instruction is first fetched to the data register (DR). Then the control unit will interpret the instruction in DR and give out the sequence of the corresponding control output signals. Since the total number of instructions is 8, they can be encoded using three bit codes as follows:

000 - load

001 - store

010 - add

011 - and

100 - jump

101 - jumpz

110 - comp

111 - rshift

The ADL description for this CPU is presented in APPENDIX A.1. The cf-graph generated by the ADL compiler *ntag* is given in APPENDIX A.2.

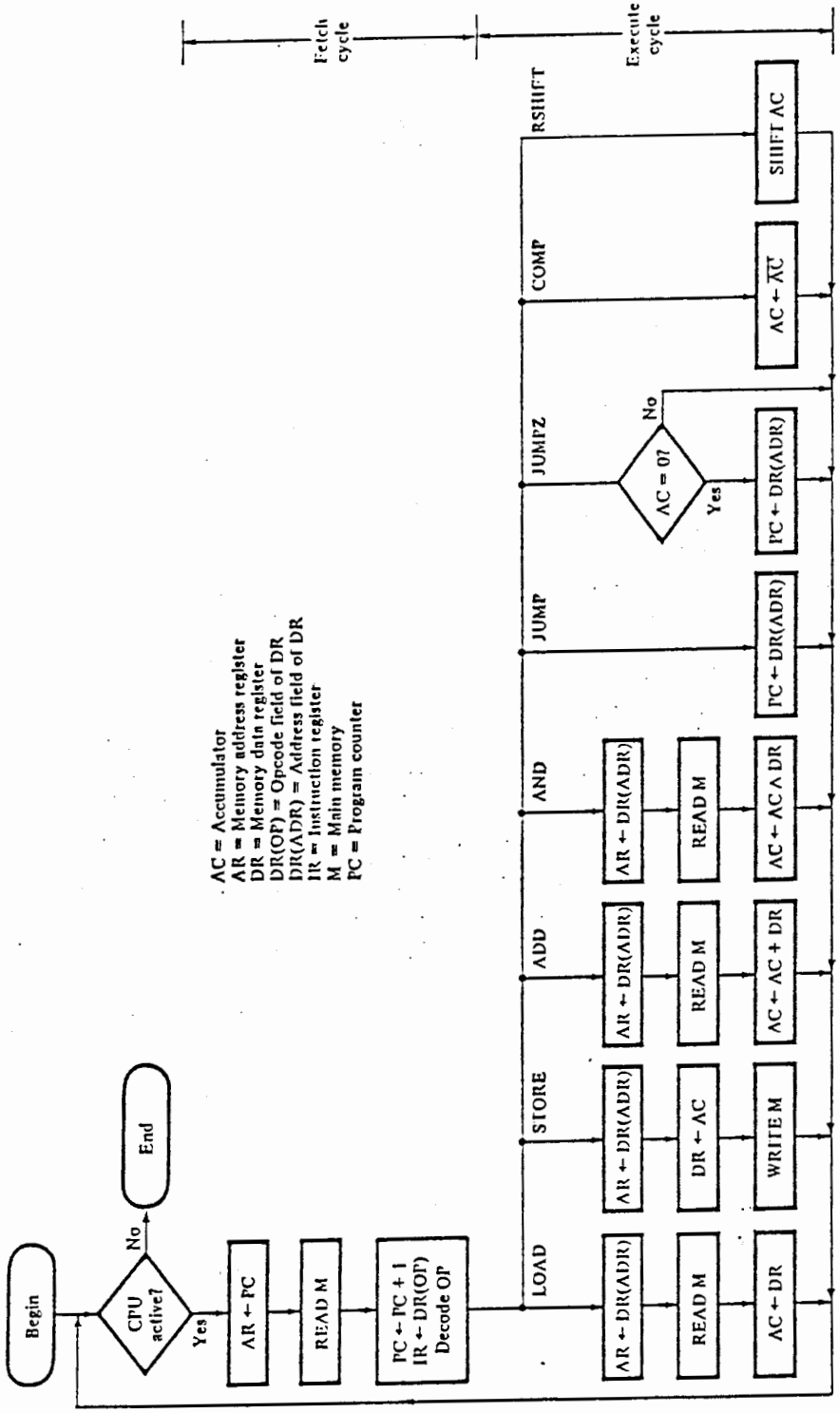


Figure 32. Flowchart of eight-instruction CPU.

The graph contains 20 operational nodes as well as 3 predicate nodes which specify 10 predicates. Program *FGEN* takes this cf-graph description as the input and generates the FSM description in both *Kiss* and *Stab* formats presented in APPENDIX A.3.

The FSM contains 13 states and 22 transitions. In the *Stab* format description, the input expressions corresponding to the columns of the state table, as well as the disjoint relations of those columns, are also provided after keywords .ip and .dr, respectively. Because the table size is too large to print on this page, instead of the next state output expression, the output number is printed. The corresponding output number for each output expression is appended to each binary cube in output specification. The disjoint relations are presented using a two dimensional matrix. The value at the intersection of the i-th row and the j-th column specifies the disjoint relation (by value 1) or the non-disjoint relation (by value 0) of the i-th and the j-th columns of the state table.

In a separate file, *IMPLY*, the actual implications of each input bit and control output bit are specified in two lists, as shown in APPENDIX A.4. The input implication list contains the predicates. The logic value of the predicates are specified by the signals of their corresponding bit positions. In the output implication list, there are two forms of specification. For each group of encoded control output signals, the set of triple items is provided. The first item specifies the number of bits for the encoded group. The second item is a set of code lists, while the third item specifies the operations controlled by the codes presented in the second item. For instance, the list :

(3 ((1 0 0)(0 1 1)(0 1 0)(0 0 1)) ((:= AC (not AC))(:= AC (and AC DR))(:= AC (plus AC DR)) (:= AC DR)))

specifies the control codes for the group of register transfer operations with register AC as the destination register. The order of the codes included in the second item is in a one-to-one correspondence with the operations listed in the third item. The MR and MW

correspond to the memory read and write operations, respectively. They are always the last two bits whenever memory reference operations are involved.

The FSM control unit generated for this example contains no redundant states, so the *Kiss* format description generated by the state minimization process remains unchanged. The state assignment program reads the *Kiss* format description, either minimized or non-minimized, encodes the states, and performs logic minimization procedure as well. The encoded truth-table format description and its corresponding *Eqn* format description for the given example are presented in APPENDIX A.5. and APPENDIX A.6, respectively. It can be seen that with a proper state encoding, 3 rows are reduced from the original truth-table description.

This example also demonstrates the control statements for the encoded predicates in ADL description. The encoded instruction is specified by an *AND* statement of all encoded bits. With the *graph modification* (see § 3.5) each bit will be treated as an unique input signal to FSM, which is the often applied approach for microprocessor design.

§ 6.2 THE CONTROL UNIT OF THE SIGNAL DELAY DEVICE

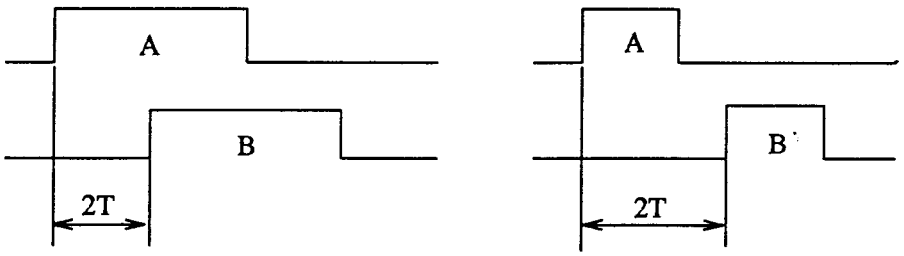


Figure 33. Different cases of signal delay.

This is an example demonstrating the design with a parallel program description. The design is to delay an input signal A by the time 2T. The device is supposed to have

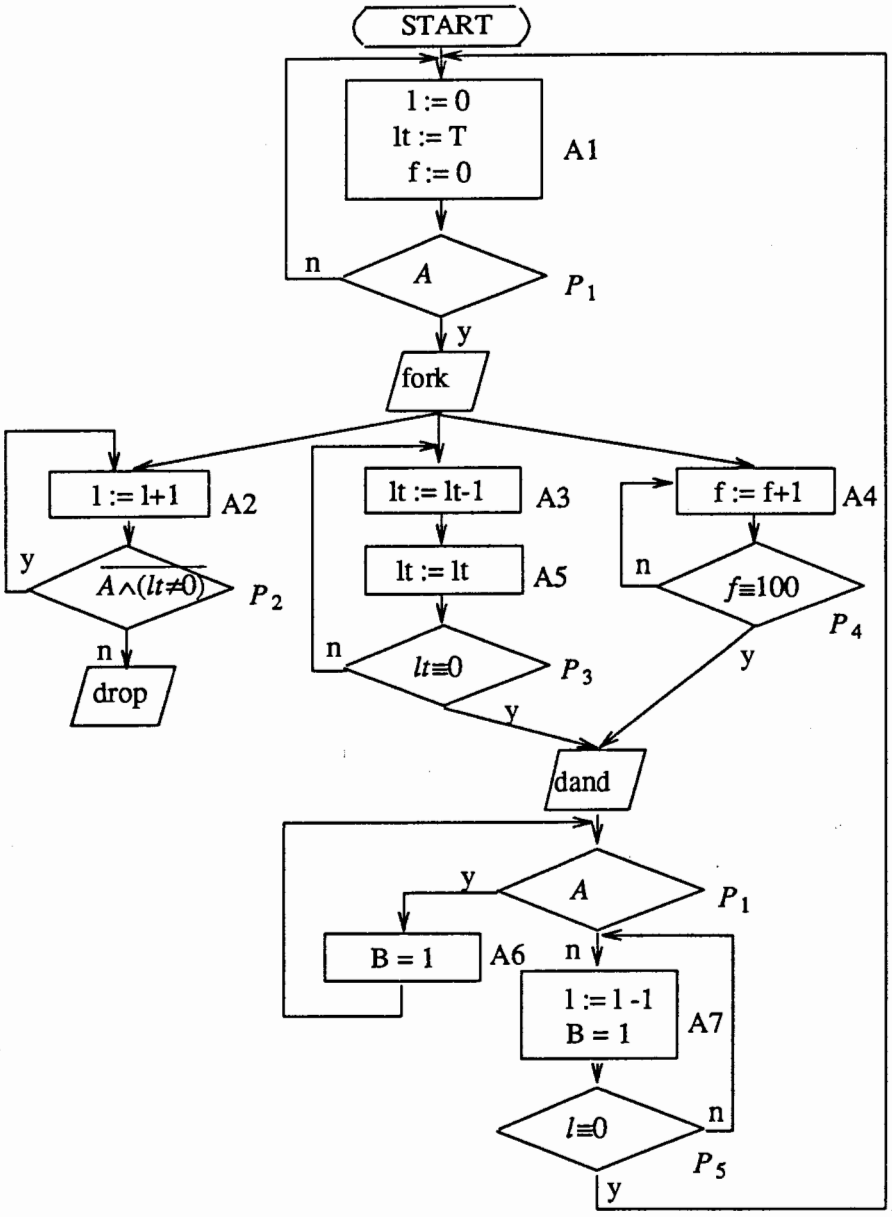


Figure 34. Control flow diagram for signal delay device.

an information input signal A, an information output signal B, and a control input T. A and B are logic signals (variables) and T is an integer number of the specified clock cycles.

Two cases of different signal width and delay time should be considered in the design. In the first case, the delay time (2T) is shorter than the signal width, as shown in Figure 33 (a). The second case is an inverse case which is shown in Figure 33 (b). With

the restriction that the delay time should not be less than a specified value, say 100, the ADL behavioral description for this device is shown in APPENDIX B.1, and its corresponding parallel control flow diagram is presented in Figure 34.

Since the input delay specification is T and actual delay time is 2T, an additional delay cycle is put before the operation $lt := lt-1$. Thus *lt* decreases 1 in every two machine cycles and total 2T cycles delay is achieved whenever *lt* becomes zero. With the *DAND* node, the output B is available only when both the required delay time (2T) is approached and the minimum delay time (100) is exceeded. If 2T < 100, the minimum delay time will be considered as the actual delay time.

In Figure 34, each rectangular block of the diagram corresponds to an operational node of the cf-graph. The functions in the block are the set of operations simultaneously performed in one machine cycle. Each set is labeled by the literals beside the block. All of the predicates are labeled by the literals beside the diamond blocks which correspond to the predicate nodes in the cf-graph. The parallel cf-graph is described by the program graph description in APPENDIX B.2. This cf-graph is first converted to a compact parallel flow graph shown in Figure 35. In actual compact parallel graph, there are no *FORK* and *DAND* nodes. They are described by the passing *FORK (DAND)* transition. They were drawn here because of easy understanding.

In the compact parallel flow graph, the labeled arrows, instead of the predicate nodes, determine the sequence of the function flow. The compact flow graph is described by a number of internal lists introduced in CHAPTER V . For the current example, the internal lists are composed of a transition list (*relgrliststar), a relation list (*relalis), a list of the passing *FORK* transitions (*forkpair), and a list of passing *DAND* transitions (*dandpair). All these lists are presented in APPENDIX B.3.

It can be concluded from the compact flow graph that the transition from the state, which performs operation A1, is to the next state which enables operations A2, A3

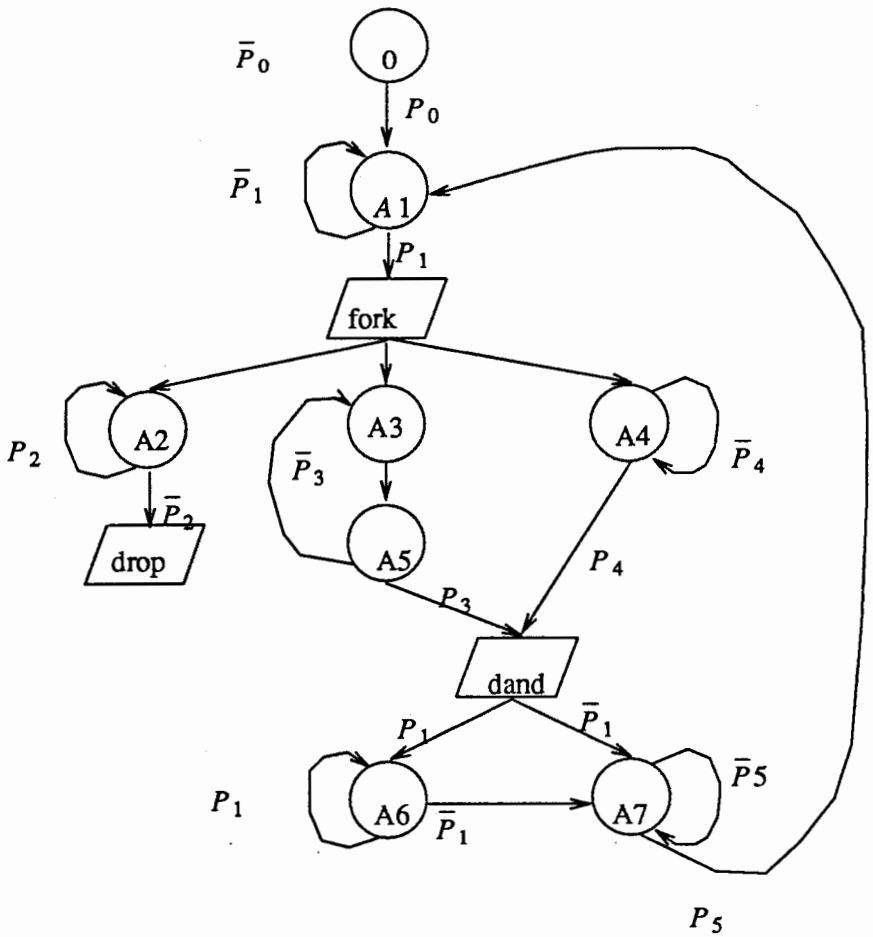


Figure 35. Compact control flow graph of signal delay device.

and A4, since these operations are executed in parallel in the graph from Figure 35. There are four possible transitions starting from this next state, as shown in Figure 36(a). All the transitions are based on the possible combinations of the predicates which are judged in parallel. At this stage, the predicate

$$\bar{P}_2 \cdot \bar{P}_4 = \overline{A \wedge (lt \neq 0)} \wedge (f \neq 100) = (\bar{A} \vee (lt = 0)) \wedge (f \neq 100)$$

corresponds to either case 1 or case 2 from Figure 34. The consequent state transition from the state enabling operations A4 and A5 is shown in Figure 36(b). In the present state the signal A turns to low or the specified delay time has been approached but the actual delay has not yet exceeded the minimum delay time. Five transitions are available. The transitions to A6 and to A7 correspond to the cases that the specified delay time

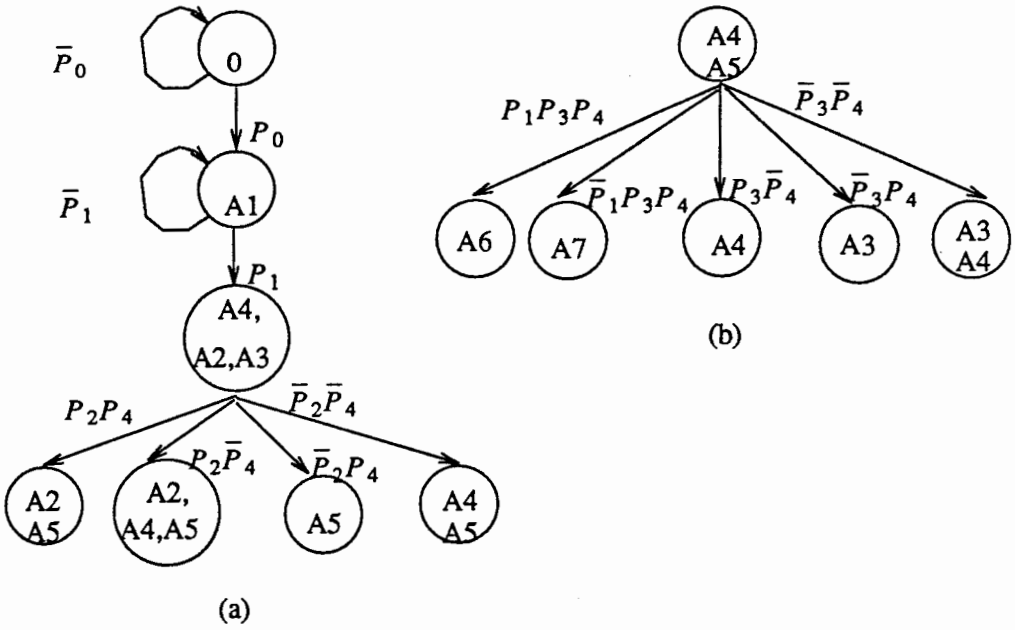


Figure 36. Conversion example for part of parallel graph in Figure 35.

and the minimum delay time are both satisfied. For the first transition, however, the input signal A keeps high, and for the second transition it turns to low. A complete FSM description in *Kiss* format is presented in APPENDIX B.4. Since the state table is too wide to be printed out, it is omitted in the APPENDIX. After state minimization, one state is reduced and the new state transition description in *Kiss* format is shown in APPENDIX B.5. The states are encoded by *FASS*, as shown in APPENDIX B.6. The combinational part of the FSM description in truth-table format is presented in APPENDIX B.7, while its minimized result is shown in APPENDIX B.9. If this combinational part is implemented by a PLA, this PLA has been reduced by 16 rows in its *OR* plane, comparing to the PLA that corresponds to the initial description.

CHAPTER VII

CONCLUSION AND FUTURE WORK

In digital circuit design, most of the FSMs considered by the industry consist of less than twenty internal states. The large FSMs (the FSM's with many states) are usually decomposed into small FSMs. By this approach, the design period can sometimes be speeded up since the time needed for state generation, state minimization and state assignment are exponentially proportional to the number of the FSM states. Sometimes, the silicon area can also be saved for a multi-FSM approach. The disadvantage of this approach is the human interference involved to specify the separate components, as well as the protocols between them. For example, if there are two operations to be controlled by an FSM and they need m and n machine cycles to perform, respectively, then (m - 1) and (n - 1) waiting states have to be included to keep the control of these two operations during their executions. If the control unit is composed of a single FSM, the total number of the waiting states would be (m + n - 2). If the control unit is composed of one FSM and a counter of module (m - 1) (suppose m > n and only one counter is used to signalize the ends of both operations), then the FSM needs only 3 states to initiate the operations and the counter, to keep waiting for the terminations of the executions as well as to acknowledge the termination of the operations and to reset the counter. Taking into account the number of states needed for the counter, the total number of (m - 1 + 3) states is needed by the whole control unit. In the second approach, the organization of the control unit has to be decided by the designer and the counter has to be separately designed. Whenever possible, the multi-FSM approach is always preferred and the FSMs of less than twenty states, therefore, are the main design objects in the practical design.

TABLE V

STATISTIC DATA OF DESIGN EXAMPLES

Machine Name	Number of									
	Op. Nodes	Status Nodes	Opera -tions	Predi -cates	States init./final	Inputs	Outputs	State bits	Rows in KISS	Rows in PLA
Gcd	2	4	3	3	3/2	3	2	1	7	3
Class	10	5	6	6	4/4	6	5	2	10	8
Squen	6	3	8	4	4/4	4	8	2	10	6
Regis	9	4	5	8	4/4	8	5	2	18	15
Pulse	6	5	6	4	6/6	4	5	3	15	11
Ohm	14	4	11	4	12/12	4	11	4	18	17
Trian	14	4	15	5	12/12	5	15	4	20	17
Micro	23	2	22	13	12/7	13	21	3	24	18
Telep	16	8	10	12	13/13	12	9	4	35	34
CPU	21	2	14	10	13/13	5	14	4	22	19
Delay	9	6	9	6	14/13	6	7	4	50	36
Volt	14	4	11	4	18/18	8	11	4	12	11

* Pulse, Delay are generated from parallel control flow graphs.
* Op. nodes - Operational nodes.
* inti./final - before state minimization and after minimization.

Several FSM control units of less than 15 states have been experimentally designed by the Synthesizer introduced above and the resultant data are presented in Table V. Since the state transition descriptions are generated by Algorithm 3.1, which tries to eliminate the redundant states in generation, and no don't-cares are involved in the current version of the Synthesizer, for most of the examples shown in Figure 7-1 the state minimization seems to be necessary. The state minimization stage, however, will benefit the future developments. With proper state assignment, the truth table descriptions for the combinational parts of the FSMs can be minimized %3 - %57 rows of the initial descriptions, which will in consequence save the silicon area for hardware implementation.

From the design experiments, the following improvements are needed for the current synthesizer.

1) The output minimization process should be included either before or after the state assignment stage. The urgent task of the minimization is to combine the same output columns or columns which complement each other in output part of *Kiss* description. For large digital systems, the optimal output encoding should be considered.

2) The significance of the state minimization depends on the don't-care generation. The don't-care can be generated with the information provided by the designer. In the cf-graph description there has to be an extra list to specify the don't-care information. The list should contains those predicate nodes, as well as the predicates, which would be impossible to happen at the time when the corresponding predicate node is approached. For example, after the assignment of value 5 to a variable x, the relation (x = 0) would be impossible to occur. From this point of view, the ADL compiler has to support the unhappening predicate specifications in future ADL descriptions. Another approach for the don't-care generation is by tracing the cf-graph during the state transition generation. The tracing can be implemented by keeping a list of the values of those variables involved in all predicates. Each time when there is a change of for any variable in the list, the list is modified.

3) Some improvement of the data structure should be made to speed up the performance of the programs. In the state transition description generation process, three lists of the input expression codes (inlis), the output expression codes (outlis) and the state transition pairs (stalis) are first generated. Each input or output expression code is represented by a list of binary numbers. The elements at the same position in the three lists describe one state transition. Later on, the state table format and *Kiss* format descriptions are generated from these three lists. The disjoint relations for the input expressions are also checked out by the bitwise

binary operations among the binary lists. The process can be speeded up by combining these three lists to a property list and changing the input and output expressions to a binary matrix instead of the list of binary lists. By changing the input expressions to a binary matrix, those disjoint checking processes can be performed by the LISP binary functions instead of the user defined binary function (*include* 1), which is frequently called in the state table generation, for the binary lists. Also, the same change can be done to the disjoint record list and the state table list (*tablis) which include the binary list as their elements.

In the control output generation for memory reference assignment, four lists are first generated. Then the read-write control code and memory address register selection code are generated with the information provided by these four lists. The four lists are the List of Read Assignments, the List of Write Assignments, the List of Different Memory Address Register References, and the List of Multiplexer Addressing Codes for the References. The example of these four lists is as follows:

read-lis : ((:= a (mem [AR1]));

write-lis : ((:= (mem [AR1]) b)(:= (mem [AR2]) c));

bind-control : ((mem [AR1])(mem [AR2]));

mem-control : (1 ((1) (0))((mem [AR1])(mem [AR2]))).

In the above lists, AR1 and AR2 are two memory address registers and "mem" indicates the memory reference assignment. The first number in the list mem_control is the number of bits for multiplexer addressing, the second element is a list of addressing codes and the third element is a list of different memory address register references. Obviously, these four lists can be combined to a property list so that the time to include the memory reference control codes into the general output control codes would be reduced.

4) The current version of DIADES takes each predicate as an input signal for the control unit. The hardware to evaluate the predicates is implemented in data path except those predicates of single logic variables. The behavioral description of the logic variable predicate *a* is of the form "if *a* then ...". To evaluate this *a*, it can be directly connected to the control unit. If the predicates are the set of instructions for a microprocessor, the instructions are first encoded so that each predicate is a sequence of binary codes. The description of this kind of predicate can be of the form as "if (and (IR [0 % 0])(not (IR [1 % 1])) ...) then ...". Here IR is assumed to be the register to store those instruction codes. when specified in this way, the predicate is of a Boolean expression form. If each bit of the IR is supposed to be directly connected to the control unit, the approach introduced in 3.5 can be used, which is the cf-graph modification to eliminate the hardware implementation of the Boolean expression form predicates. The future work is to make the specification of the predicates more compact. For example, the specification for the above predicate is expected to be of the form as "if (equal IR (...01)) then ...". The algorithm and the programs for optimal encoding instructions should be also developed.

5) The interface to the high level simulation tools should be implemented to verify the design results at high level.

6) More examples are needed to prove that the entire methodology presented in this thesis is practical, and to modify and improve this methodology, if necessary.

REFERENCES

[1] [Bray 88] Brayton, R., Camposano, R., De Micheli, G., Otten, R., and J. van Eijndhoven, "The Yorktown Silicon Compiler System", *in Silicon Compilation,* D. Gajski (ed.), Addison Wesley, 1988.

[2] [DeMi 83] De Micheli, G., and A. Sangiovanni-Vincentelli, "Computer-Aided Synthesis of PLA-Based Finite State Machine", *Proc. IEEE ICCAD'83,* pp. 154-156. Santa Clara, California, Sept. 1983.

[3] [DeMi 83a] De Micheli, G., Brayton, R., and A. Sangiovanni-Vincentelli, "Kiss A Program for Optimal State Assignment of Finite State Machine", *Proc. IEEE ICCAD'83,* pp. 209-211, Santa Clara, California, Sept. 1983.

[4] [Hell 88] Helliwell, M., and M. Perkowski, "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms", *Proc. DAC'88.*

[5] [Koha 82] Kohavi, Z., "Switching and Finite Automata Theory", McGraw-Hill, New York, 1978.

[6] [Kowa 85] Kowalski, T. J., Geiger, D. J., Wolf, W. H., and W. Fichtner, "The VLSI Design Automation Assistant From Algorithms to Silicon", *IEEE Design and Test,* Aug. 1985.

[7] [Lee 84] Lee, E.B., and M. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines", *Proc. of the 1984 Intern. Conf. on Systems, Man and Cybernetics, IEEE,* Halifax, Nova Scotia, Canada, 1984.

[8] [Meye 84] Meyer, M. J., Agrawal, P., and R. G. Pfister, "A VLSI FSM Design System", *Proc. of 21st DAC,* pp. 434-440, Albuquerque, New Maxico, 1984.

[9] [Newt 86] Newton, A. R., and L. Sangiovanni-Vincentelli, "Computer-Aidded Design for VLSI Circuits", *IEEE Computer,* Apr. 1986.

[10] [Nguy 87] Nguyen, L.B., Perkowski, M.A., and N.B. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers", *Proc. of the 24th Design Automation Conference,* Miami, 1987.

[11] [Wu 89] Wu, Pan, M.A. Perkowski, "KUAI-EXACT: A new approach for multi-valued logic minimization in VLSI synthesis", *accepted by ISCAS '89.*

[12] [Pang 87] Pangrle, B. M., and D. Gajski, "Design tools for Intelligent Silicon Compilation", *IEEE Trans. Computer-Aided Design,* Vol. CAD-6, No. 6, Nov. 1987.

[13] [Perk 76] Perkowski, M, "ADL - source language of the system for automatic design", in R. Marczyn'ski (ed.) *Organization of digital computers and microprogramming,* Polish Scientific Publishers (PWN), Łódź 1976. Vol. 1, pp. 167-180.

[14] [Perk 77] Perkowski, M, "A Method of Validation of Parallel Programs in the System for Automatic Design of Block-Oriented Digital Systems", *Proc. 2nd IFAC Symposium on Discrete Systems,* Dresden, GDR, 14-19 March, 1977, Vol. 2, pp.71-88.

[15] [Perk 79] Perkowski, M, "Automatischer Entwurf von MOS-LSI-digitalen Schaltungen in System DIADES", *Messen, Steuern, Regeln,* Vol. 6., 1979, pp. 346-350 (in German).

[16] [Perk 82] Perkowski, M., "Digital Devices Design by Problem-Solving Transformations", *Journal on Computers and Artificial Intelligence,* Vol. **1,** No. 4., 1982, pp. 343-365.

[17] [Perk 85] Perkowski, M., and N. Nguyen, "Minimization of Finite State Machines in SuperPeg", *Proc. of the Midwest Symposium on Circuits and Systems,* Luisville, Kentucky, 22 - 24 August 1985.

[18] [Perk 88] M. Perkowski, J.E. Brown: "An Unified Approach to Designs Implemented with Multiplexers and to the Decomposition of Boolean Functions", *Proceedings of 1988 ASEE National Conference,* Portland, Oregon, June 19-23, 1988.

[19] [Perk 89a] Perkowski, M., "Parallel Programs in ADL and Their Semantics", *Diades Research Group Report,* PSU. 1989.

[20] [Perk 89b] Perkowski, M., "FASS - An Optimal State Assignment Program", *Diades Research Group Report,* PSU, 1989.

[21] [Rude 85] Rudell, R. L., and A. Sangiovanni-Vincetelli, "ESPRESSO-MV Algorithms for Multiple-level Logic minimization," *Proc. of 13th DAC,* pp. 230-233, Portland, Oregon, May 1985.

[22] [Sauc 87] Saucier, G., Crastes de Paulet, M., and P. Socard, "ASYL: A Rule-Based System for Controller Synthesis", *IEEE Trans. Computer-Aided Design,* Vol. CAD-6, No. 6, Nov. 1987.

[23] [Sout 83] Southard, J. R., "MacPitts: An Approach to Silicon Compilation", *IEEE Computer,* 1983.

[24] [Tsen 86] Tseng, C. J., Prbhu, A. M., Li, C., Memood, Z., and M. M. Tong, "A Versatile Finite State Machine Synthesizer", *Proc. IEEE ICCAD'86,* pp. 206-209, 1986.

[25] [Walt 85] Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, editors, "1986 VLSI Tools: Still More Works by the Original Artists", *Report No. UCB/CSD/86/272,* EECS, University of California, December 1985.

[26] [Yang 89] Yang, L., Perkowski, M., and D. Smith, "Design and Optimization of Microprogrammed Control Units in Diades", *Diades Research Group Report,* PSU, 1989.

[27] [Zhao 89] Zhao, W., "Two Dimensional Minimization of Finite State Machines", *Master Thesis,* Department of Electrical Engineering, Portland State University, 1989.

APPENDIX A

DESIGN DATA OF EIGHT-INSTRUCTION CPU

1) ADL program for eight-instruction CPU :

```
adl
graph
subgraph
(((adl c 8-inst-cpu
  ((clock (1000)))
  (intern (read (d))(write (d))(pc (p k1 8))(AR (p k1 8))(IR (p k1 8))
        (DR (p k1 8))(AC (p k1 8))((mem [ k ])(p k1 8))(k (p k1 8))))
((start) c
      (AR := pc)
      (DR := (mem [ AR ]))
      (sim (pc := (pc + 1))
           (IR := DR))
           ;load
      (cond ((and  (not (DR [ 2 % 2 ]))
                   (not (DR [ 1 % 1 ]))
                   (not (DR [ 0 % 0 ])))
            (AR := DR)
            (DR := (mem [ AR ]))
            (AC := DR))
           ;store
           ((and  (not (DR [ 2 % 2 ]))
                  (not (DR [ 1 % 1 ]))
                  (DR [ 0 % 0 ]))
            (AR := DR)
            (DR := AC)
            ((mem [ AR ]) := DR))
           ;add
           ((and  (not (DR [ 2 % 2 ]))
                  (DR [ 1 % 1 ])
                  (not (DR [ 0 % 0 ])))
            (AR := DR)
            (DR := (mem [ AR ]))
            (AC := (AC + DR)))
           ;and
           ((and  (not (DR [ 2 % 2 ]))
                  (DR [ 1 % 1 ])
                  (DR [ 0 % 0 ]))
            (AR := DR)
            (DR := (mem [ AR ]))
            (AC := (and AC DR)))
```

```
        ;jump
        ((and  (DR [ 2 % 2 ])
               (not (DR [ 1 % 1 ]))
               (not (DR [ 0 % 0 ])))
         (pc := DR))
        ;jumpz
        ((and  (DR [ 2 % 2 ])
               (not (DR [ 1 % 1 ]))
               (DR [ 0 % 0 ]))
         (if (AC = 0) then (pc := DR)))
        ;comp
        ((and  (DR [ 2 % 2 ])
               (DR [ 1 % 1 ])
               (not (DR [ 0 % 0 ])))
         (AC := (not AC)))
        ;rshift
        ((and  (DR [ 2 % 2 ])
               (DR [ 1 % 1 ])
               (DR [ 0 % 0 ]))
         (sim
         ((AC [ 0 % 6 ]) := (AC [ 1 % 7 ]))
         ((AC [ 7 % 7 ]) := (AC [ 0 % 0 ])))))
    (stopadl)
    )))
end
```

2) The cf-graph description of eight-instruction CPU :

a)List of arrows :

```
(setq *coplisset* '
((1
 ((x 31 34)
  (x 29 34)
  ((not 26) 26 34)
  (x 27 34)
  (x 24 34)
  (x 22 34)
  (x 18 34)
  (x 14 34)
  (x 10 34)
  (30 7 31)
  (28 7 29)
  (26 26 27)
  (25 7 26)
  (23 7 24)
  (x 21 22)
  (x 20 21)
  (19 7 20)
  (x 17 18)
  (x 16 17)
  (15 7 16)
  (x 13 14)
  (x 12 13)
```

```
(11 7 12)
(x 9 10)
(x 8 9)
(7 7 8)
(x 4 7)
(x 3 4)
(x 2 3)
(x 1 2)))) )
```

b) List of node properties :

```
(setq *nolisset* '
((1
((stopadl 34 nil)
(31 31 nil)
(29 29 nil)
(24 27 nil)
(cond 26 nil)
(24 24 nil)
(22 22 nil)
(3 21 nil)
(8 20 nil)
(18 18 nil)
(3 17 nil)
(8 16 nil)
(14 14 nil)
(13 13 nil)
(8 12 nil)
(10 10 nil)
(3 9 nil)
(8 8 nil)
(cond 7 nil)
(4 4 nil)
(3 3 nil)
(2 2 nil)
(start 1 nil)))) )
```

c) List of assignments :

```
(setq *nalisset* '
((1
((31 (32 33))
(33 (:= (AC |[| 7 % 7 |]|) (AC |[| 0 % 0 |]|)))
(32 (:= (AC |[| 0 % 6 |]|) (AC |[| 1 % 7 |]|)))
(29 (:= AC (not AC)))
(24 (:= pc DR))
(22 (:= AC (and AC DR)))
(18 (:= AC (plus AC DR)))
(14 (:= (mem |[| AR |]|) DR))
(13 (:= DR AC))
(10 (:= AC DR))
(8 (:= AR DR))
(4 (5 6))
(6 (:= IR DR))
(5 (:= pc (plus pc 1)))
```

```
        (3 (:= DR (mem |[| AR |]|)))
        (2 (:= AR pc))))) )
```

d) List of predicates :

```
        (setq *plisset* '
       ((1
        ((30 (and (DR |[| 2 % 2 |]|) (DR |[| 1 % 1 |]|) (DR |[| 0 % 0 |]|)))
         (28 (and (DR |[| 2 % 2 |]|) (DR |[| 1 % 1 |]|) (not (DR |[| 0 % 0
       |]|))))
         (26 (equal AC 0))
         (25 (and (DR |[| 2 % 2 |]|) (not (DR |[| 1 % 1 |]|)) (DR |[| 0 % 0
       |]|)))
         (23 (and (DR |[| 2 % 2 |]|) (not (DR |[| 1 % 1 |]|)) (not (DR |[| 0 %
       0 |]|))))
         (19 (and (not (DR |[| 2 % 2 |]|)) (DR |[| 1 % 1 |]|) (DR |[| 0 % 0
       |]|)))
         (15 (and (not (DR |[| 2 % 2 |]|)) (DR |[| 1 % 1 |]|) (not (DR |[| 0 %
       0 |]|))))
         (11 (and (not (DR |[| 2 % 2 |]|)) (not (DR |[| 1 % 1 |]|)) (DR |[| 0 %
       0 |]|)))
         (7 (and (not (DR |[| 2 % 2 |]|)) (not (DR |[| 1 % 1 |]|))
              (not (DR |[| 0 % 0 |]|)))))))) )
```

e) List of node groups :

```
        (setq *anlisset* '
       ((1
        ((stopadl (34))
         (31 (31))
         (29 (29))
         (24 (24 27))
         (cond (7 26))
         (22 (22))
         (3 (3 9 17 21))
         (8 (8 12 16 20))
         (18 (18))
         (14 (14))
         (13 (13))
         (10 (10))
         (4 (4))
         (2 (2))
         (start (1))))) )
```

f) List of memory variables :

```
        (setq *lzmset* '
       ((1 ((AC |[| 7 % 7 |]|) (AC |[| 0 % 6 |]|) (mem |[| AR |]|) AC IR pc
       DR AR))) )
```

g) List of the systems under design :

```
        (setq *symlis* '
       ((c 1)) )
```

3) *Kiss* format and state table format description of FSM :

```
----0 st1 st1 00000000000000
----- st13 st5 00000000100000
----- st12 st13 00010000000010
-000- st4 st12 10000000000000
1101- st4 st5 00000000010000
----- st11 st5 00000001000000
----- st10 st11 00010000000010
-010- st4 st10 10000000000000
0101- st4 st1 00000000000000
-111- st4 st5 00000000001100
----- st9 st5 00000000000001
----- st8 st9 00100000000000
-001- st4 st8 10000000000000
----- st7 st5 00000001100000
----- st6 st7 00010000000010
-011- st4 st6 10000000000000
-100- st4 st5 00000000010000
----- st5 st1 00000000000000
-110- st4 st5 00000010000000
----- st3 st4 00001100000000
----- st2 st3 00010000000010
----1 st1 st2 01000000000000

1./9 -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 2./13
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 3./12 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 4./11 -1./*
-1./* 12./7 5./8 10./7 1./9 5./3 8./7 6./7 5./8 5./10 -1./* -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 1./9 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 7./12 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 5./6 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 9./5 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 5./4 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 11./12 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 5./2 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 13./12 -1./*
-1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* -1./* 5./8 -1./*

.ip
----0
-000-
1101-
-010-
0101-
-111-
-001-
-011-
-100-
-110-
-----
----1
.op
00000000100000 (1)
00000001000000 (2)
```

```
00000000001100 (3)
00000000000001 (4)
00100000000000 (5)
00000001100000 (6)
10000000000000 (7)
00000000010000 (8)
00000000000000 (9)
00000010000000 (10)
00001100000000 (11)
00010000000010 (12)
01000000000000 (13)
.dr
000000000001
001111111100
010111111100
011011111100
011101111100
011110111100
011111011100
011111101100
011111110100
011111111000
000000000000
100000000000
end
```

4) Input and output implications of FSM

```
input-implication:
        ((equal AC 0) (DR |[| 2 % 2 |]|) (DR |[| 1 % 1 |]|) (DR |[| 0 % 0 |]|)
       (start))

output-implication:
        ((2 ((1 0) (0 1)) ((:= AR DR) (:= AR pc)))
         (2 ((1 0) (0 1)) ((:= DR AC) (:= DR (mem |[| AR |]|))))
         (:= pc (plus pc 1))
         (:= IR DR)
         (3 ((1 0 0) (0 1 1) (0 1 0) (0 0 1))
           ((:= AC (not AC)) (:= AC (and AC DR)) (:= AC (plus AC DR))
       (:= AC DR)))
         (:= pc DR)
         (:= (AC |[| 0 % 6 |]|) (AC |[| 1 % 7 |]|))
         (:= (AC |[| 7 % 7 |]|) (AC |[| 0 % 0 |]|))
         MR
         MW)
```

5) Truth-table description for the combinational part of FSM

a) Encoded states :

st1	0 0 0 0
st2	0 0 0 1
st3	1 0 0 1
st4	0 1 0 0

st5 0 0 1 0
st6 1 1 1 0
st7 1 0 1 0
st8 1 1 0 0
st9 1 0 0 0
st10 0 1 0 1
st11 0 1 1 1
st12 0 1 1 0
st13 0 0 1 1

b) Unminimized truth-table :

```
.i 9
.o 18

----1 0000 0001 010000000000000
----0 0000 0000 000000000000000
----- 0001 1001 000100000000010
----- 1001 0100 000011000000000
-110- 0100 0010 000000100000000
-100- 0100 0010 000000000010000
-011- 0100 1110 100000000000000
-001- 0100 1100 100000000000000
-111- 0100 0010 000000000001100
0101- 0100 0000 000000000000000
-010- 0100 0101 100000000000000
1101- 0100 0010 000000000010000
-000- 0100 0110 100000000000000
----- 0010 0000 000000000000000
----- 1110 1010 000100000000010
----- 1010 0010 000000011000000
----- 1100 1000 001000000000000
----- 1000 0010 000000000000001
----- 0101 0111 000100000000010
----- 0111 0010 000000010000000
----- 0110 0011 000100000000010
----- 0011 0010 000000001000000
.e
```

c) Minimized truth-table :

```
.i 9
.o 18
.p 19
-111-0100 000000000000001100
110--0100 001000000000010000
-010-0100 010110000000000000
-110-0100 001000000010000000
-100-0100 001000000000010000
-000-0100 011010000000000000
-0-1-0100 110010000000000000
----10000 000101000000000000
-----1001 010000001100000000
-----0011 001000000000100000
-----1010 001000000001100000
```

```
--11-01-- 001000000000000000
-----1100 100000100000000000
-----1000 001000000000000001
-----0111 001000000001000000
-----0001 100100010000000010
-----1110 101000010000000010
-----0110 001100010000000010
-----0101 011100010000000010
.e
```

6) The result of the conversion from truth-table description to EQN description

```
Q1 = IN2 & IN3 & IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q2 = IN1 & IN2 & ! IN3 & ! IN6 & IN7 & ! IN8 & ! IN9
Q3 = ! IN2 & IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q4 = IN2 & IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q5 = IN2 & ! IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q6 = ! IN2 & ! IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q7 = ! IN2 & IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q8 = IN5 & ! IN6 & ! IN7 & ! IN8 & ! IN9
Q9 = IN6 & ! IN7 & ! IN8 & IN9
Q10 = ! IN6 & ! IN7 & IN8 & IN9
Q11 = IN6 & ! IN7 & IN8 & ! IN9
Q12 = IN6 & IN7 & ! IN8 & ! IN9
Q13 = IN6 & ! IN7 & ! IN8 & ! IN9
Q14 = ! IN6 & IN7 & IN8 & IN9
Q15 = ! IN6 & ! IN7 & ! IN8 & IN9
Q16 = IN6 & IN7 & IN8 & ! IN9
Q17 = ! IN6 & IN7 & IN8 & ! IN9
Q18 = ! IN6 & IN7 & ! IN8 & IN9
OUT1 = Q7 | Q12 | Q15 | Q16
OUT2 = Q3 | Q6 | Q7 | Q9 | Q18
OUT3 = Q2 | Q4 | Q5 | Q6 | Q10 | Q11
     | & IN3 & IN4 & ! IN6 & IN7
     | Q13 | Q14 | Q16 | Q17 | Q18
OUT4 = Q3 | Q8 | Q15 | Q17 | Q18
OUT5 = Q3 | Q6 | Q7
OUT6 = Q8
OUT7 = Q12
OUT8 = Q15 | Q16 | Q17 | Q18
OUT9 = Q9
OUT10 = Q9
OUT11 = Q4
OUT12 = Q11 | Q14
OUT13 = Q10 | Q11
OUT14 = Q2 | Q5
OUT15 = Q1
OUT16 = Q1
OUT17 = Q15 | Q16 | Q17 | Q18
OUT18 = Q13
```

APPENDIX B

DESIGN DATA OF SIGNAL DELAY PROCESSOR

1) The description of signal delay device in ADL :

```
graph
subgraph
  (((adl c opimp
     ((clock(1000)))
     (input(a(d))(t(p k1 8)))
     (intern(l(p k1 8))(lt(p k1 8))(f(p k1 8)))
     (output(b(d))))
    ((start) c
 10  (sim (l := 0) (lt := t)(f := 0))
     (if(! a)then(go 10))
     (fork
      (15 (f := (f + 1))
       (if (!(f = 100)) then (go 15)))
      (12(lt := lt)
        (lt := (lt - 1))
       (if(!(lt = 0))then(go 12)))
      (11 (l := (l + 1))
       (if(and a (!(lt = 0))) then(go 11))
       (drop))
       dand)
      13(if a then (b == 1)(go 13) )
      14(sim
        (l := (l - 1))
        (b == 1))
     (if(l = 0)then(go 10)else(go 14))
     )))
end
```

2) The cf-graph description of signal delay device :

a) List of arrows :

```
(setq *coplisset* ' ((1
(((not 22) 22 19)
 (22 22 2)
 (x 19 22)
 ((not 6) 17 19)
 (x 18 17)
 (6 17 18)
 (e 16 17)
 (12 12 16)
```

```
(9 9 16)
((not 14) 14 15)
(14 14 13)
(x 13 14)
(e 7 13)
((not 12) 12 10)
(x 11 12)
(x 10 11)
(e 7 10)
((not 9) 9 8)
(x 8 9)
(e 7 8)
(6 6 7)
((not 6) 6 2)
(x 2 6)
(x 1 2)))) )
```

b) List of node properties :

```
(setq *nolisset* ' ((1
((cond 22 nil)
(19 19 nil)
(18 18 nil)
(cond 17 nil)
(dand 16 nil)
(drop 15 nil)
(cond 14 nil)
(13 13 nil)
(cond 12 nil)
(11 11 nil)
(10 10 nil)
(cond 9 nil)
(8 8 nil)
(fork 7 nil)
(cond 6 nil)
(2 2 nil)
(start 1 nil)))) )
```

c) List of Assignments :

```
(setq *nalisset* ' ((1
((19 (18 20))
(20 (:= 1 (plus 1 (minus 1))))
(18 (== b 1))
(13 (:= 1 (plus 1 1)))
(11 (:= lt (plus lt (minus 1))))
(10 (:= lt lt))
(8 (:= f (plus f 1)))
(2 (3 4 5))
(5 (:= f 0))
(4 (:= lt t))
(3 (:= 1 0))))) )
```

d)List of predicates :

```
(setq *plisset* ' ((1
```

```
((22 (equal l 0))
 (14 (and a (not (equal lt 0))))
 (12 (equal lt 0))
 (9 (equal f 100))
 (6 a)))) )
```

e) List of node groups :

```
(setq *anlisset* ' ((1
((cond (6 9 12 14 17 22))
(19 (19))
(18 (18))
(dand (16))
(drop (15))
(13 (13))
(11 (11))
(10 (10))
(8 (8))
(fork (7))
(2 (2))
(start (1))))) )
```

f) List of memory variables :

```
(setq *lzmset* ' ((1 (b f lt l))) )
```

g) List of systems under design :

```
(setq *symlis* ' ((c 1)) )
```

3) Compact parallel cf-graph description :

a) Transition description list :

```
((1 (1 1))
(9 (7 2))
(9 (9 3))
(7 (7 4))
(7 (2 5))
(8 (9 6))
(8 (8 7))
(6 (9 8))
(6 (5 9))
(3 (3 10))
(8 (7 11))
(2 (8 12))
(6 (7 13))
(5 (6 14))
(2 (5 15))
(3 (4 16))
(2 (3 17))
(2 (2 18))
(1 (2 19)))
```

b) Relation description list :

```
((1 (1) ((not (start))))
```

(2 (18 20) ((not 6)))
(3 (18) (6))
(4 (18 20) ((not 22)))
(5 (3 4 5) (22))
(6 (18) (6 9))
(7 (8) ((not 9)))
(8 (18) (6 12))
(9 (10) ((not 12)))
(10 (13) (14))
(11 (18 20) ((not 6) 9))
(12 (8) (6))
(13 (18 20) ((not 6) 12))
(14 (11) nil)
(15 (10) (6))
(16 (drop) ((not 14)))
(17 (13) (6))
(18 (3 4 5) ((not 6)))
(19 (3 4 5) ((start))))

c) Passing *FORK* transitions list :
((2 8) (2 5) (2 3))

d) Passing *DAND* transitions list :
((8 9) (6 9) (8 7) (6 7))

4) *Kiss* format and state table descriptions of FSM control unit :

-----0 st1 st1 0000000
-----1 st1 st2 1110000
----1- st2 st3 0001010
----0- st2 st2 1110000
-1-11- st3 st4 0000110
-0-11- st3 st5 0000100
-1-0-- st3 st6 0001110
-0-0-- st3 st7 0001100
-0-10- st3 st5 0000100
--111- st7 st8 0000000
--011- st7 st9 0000000
--101- st7 st10 0001000
--00-- st7 st11 0001000
--100- st7 st10 0001000
--010- st7 st9 0000000
--110- st7 st12 0000001
---11- st11 st5 0000100
---0-- st11 st7 0001100
---10- st11 st5 0000100
---11- st10 st8 0000000
---0-- st10 st10 0001000
---10- st10 st12 0000001
------ st9 st5 0000100
-0111- st6 st8 0000000
-1011- st6 st13 0000010
-0011- st6 st9 0000000

-0101- st6 st10 0001000
-100-- st6 st14 0001010
-000-- st6 st11 0001000
-0100- st6 st10 0001000
-0010- st6 st9 0000000
-0110- st6 st12 0000001
-1-11- st14 st4 0000110
-0-11- st14 st5 0000100
-1-0-- st14 st6 0001110
-0-0-- st14 st7 0001100
-0-10- st14 st5 0000100
-1---- st13 st4 0000110
-0---- st13 st5 0000100
--1-1- st5 st8 0000000
--0--- st5 st9 0000000
--1-0- st5 st12 0000001
-01-1- st4 st8 0000000
-10--- st4 st13 0000010
-00--- st4 st9 0000000
-01-0- st4 st12 0000001
0----- st12 st12 0000001
1----- st12 st2 1110000
----0- st8 st12 0000001
----1- st8 st8 0000000

.ip
-----0
-----1
--111-
--011-
--101-
--00--
--100-
--010-
--110-
---11-
---0--
---10-

-0111-
-1011-
-0011-
-0101-
-100--
-000--
-0100-
-0010-
-0110-
-1-11-
-0-11-
-1-0--
-0-0--
-0-10-

```
-1----
-0----
--1-1-
--0---
--1-0-
-01-1-
-10---
-00---
-01-0-
0-----
1-----
----0-
----1-
0001010
0001000
0001110
0001100
0000110
0000100
0000010
1110000
0000001
0000000
.dr
010000000000000000000000000000000000000000
100000000000000000000000000000000000000000
000111111011001111111100111000110111001 0
001011111011010011111100111001011001001 0
001101111101011101111111001000110111001 0
001110111101011110011111001001011001000 0
001111011101011111101111001001110110000 1
001111101110011111111011110001011001000 1
001111111011001111111101111000110111000 1
000011111011000011111100111000010001001 0
001100011101011100001111001000000000000 0
001111100110011111110011110001001000000 1
000000000000000000000000000000000000000000
000111111011001111111110111100110111001 0
001011111011010111111101111011011001001 0
001011111011011011111110111101011101001 0
001101111101011101111111101100110111001 0
001110111101011101111111011010111001000 0
001111011101011111101111101101101110000 1
001111101110011111111011110101011101000 1
001111111011001111111101111010111001000 1
000011111011010111111101111010011001001 0
000011111011001011111110111000010101001 0
001100011101011101111111011010001011000 0
001100011101011101001111101100000100000 0
001111100110011111110011110101001100000 1
000000000000010110111101011010001011000 0
000000000000001001000010100100000100000 0
```

```
000101111001001101111100001000110111001
001010101000010010010100000001011001000
001111010100011111101011000001101110000
000101111001001101111110101100110111001
001010101000010110111101011011011011000
001010101000011011010110100101011101000
001111010100011111101011100101101110000
000000000000000000000000000000000000100
000000000000000000000000000000000001000
001110000100011110000011000001001000000
000000111001000000011100001000010001001
end
```

5) Minimized *Kiss* format format description :

```
-----0 st 1 st 1 0000000
-----1 st 1 st 2 1110000
----0- st 2 st 2 1110000
----1- st 2 st13 0001010
-01-1- st 3 st 7 0000000
-10--- st 3 st12 0000010
-00--- st 3 st 8 0000000
-01-0- st 3 st11 0000001
--1-1- st 4 st 7 0000000
--0--- st 4 st 8 0000000
--1-0- st 4 st11 0000001
-0111- st 5 st 7 0000000
-1011- st 5 st12 0000010
-0011- st 5 st 8 0000000
-0101- st 5 st 9 0001000
-100-- st 5 st13 0001010
-000-- st 5 st10 0001000
-0100- st 5 st 9 0001000
-0010- st 5 st 8 0000000
-0110- st 5 st11 0000001
--111- st 6 st 7 0000000
--011- st 6 st 8 0000000
--101- st 6 st 9 0001000
--00-- st 6 st10 0001000
--100- st 6 st 9 0001000
--010- st 6 st 8 0000000
--110- st 6 st11 0000001
----0- st 7 st11 0000001
----1- st 7 st 7 0000000
------ st 8 st 4 0000100
---11- st 9 st 7 0000000
---0-- st 9 st 9 0001000
---10- st 9 st11 0000001
---11- st10 st 4 0000100
---0-- st10 st 6 0001100
---10- st10 st 4 0000100
0----- st11 st11 0000001
1----- st11 st 2 1110000
```

```
-1---- st12 st 3 0000110
-0---- st12 st 4 0000100
-1-11- st13 st 3 0000110
-0-11- st13 st 4 0000100
-1-0-- st13 st 5 0001110
-0-0-- st13 st 6 0001100
-0-10- st13 st 4 0000100
```

6) Input and output implications

```
input-implication:
        ((equal l 0) (equal f 10) (equal lt 0) (and a (not (equal lt 0))) a
        (start))

output-implication:
        ((:= l 0)
         (:= lt t)
         (:= e 0)
         (:= f 0)
         (:= l (plus l 1))
         (:= lt (plus lt (minus 1)))
         (:= e (plus e 1))
         (:= f (plus f 1))
         (:= l (plus l (minus 1))))
```

7) Encoded states :

```
st1 .     0 0 0 0
st2       0 0 0 1
st3       1 1 0 1
st4       0 1 0 0
st5       1 0 0 0
st6       1 1 1 0
st7       1 1 0 0
st8       0 1 1 0
st9       1 0 1 0
st10      0 0 1 0
st11      1 0 0 1
st12      1 1 1 1
st13      0 1 0 1
```

8) Truth-table description for the combinational part of FSM :

```
.i 10
.o 11

-----1 0000 0001 1110000
-----0 0000 0000 0000000
----1- 0001 0101 0001010
----0- 0001 0001 1110000
-01-0- 1101 1001 0000001
-00--- 1101 0110 0000000
-10--- 1101 1111 0000010
```

```
-01-1- 1101 1100 0000000
--1-0- 0100 1001 0000001
--0--- 0100 0110 0000000
--1-1- 0100 1100 0000000
-0110- 1000 1001 0000001
-0010- 1000 0110 0000000
-0100- 1000 1010 0001000
-000-- 1000 0010 0001000
-100-- 1000 0101 0001010
-0101- 1000 1010 0001000
-0011- 1000 0110 0000000
-1011- 1000 1111 0000010
-0111- 1000 1100 0000000
--110- 1110 1001 0000001
--010- 1110 0110 0000000
--100- 1110 1010 0001000
--00-- 1110 0010 0001000
--101- 1110 1010 0001000
--011- 1110 0110 0000000
--111- 1110 1100 0000000
----1- 1100 1100 0000000
----0- 1100 1001 0000001
------ 0110 0100 0000100
---10- 1010 1001 0000001
---0-- 1010 1010 0001000
---11- 1010 1100 0000000
---10- 0010 0100 0000100
---0-- 0010 1110 0001100
---11- 0010 0100 0000100
1----- 1001 0001 1110000
0----- 1001 1001 0000001
-0---- 1111 0100 0000100
-1---- 1111 1101 0000110
-0-10- 0101 0100 0000100
-0-0-- 0101 1110 0001100
-1-0-- 0101 1000 0001110
-0-11- 0101 0100 0000100
-1-11- 0101 1101 0000110
.e
```

9) Optimized truth-table description for the combinational part of FSM :

```
.i 10
.o 11
.p 36
-001--1000 01100000000
-0-11-1--0 01000000000
-1011-1000 11110000010
-0110-1--0 00010000001
-0-0--0101 10100001000
-100--1000 01010001010
-1-0--0101 10000001110
---11--1-0 01000000000
```

```
-1-11-0101 11010000110
--110-11-0 00010000001
--01--1110 01100000000
---11-1010 11000000000
----0-0001 00011110000
-01-1-110- 11000000000
-----10000 00011110000
-0-0--10-0 00100001000
---10-1010 10010000001
-0----0101 01000000100
-01-0-110- 10010000001
1-----1001 00011110000
--0---0100 01100000000
--0---1101 01100000000
-10---11-1 10010000010
----1-0001 01010001010
-1----1111 10010000010
0-----1001 10010000001
---0--1-10 00100001000
--1-1--100 11000000000
--1-0--100 10010000001
--1---1-10 10000000000
-01---1--0 10000000000
---0---010 10100001000
------1111 01000000100
----1-1100 11000000000
----0-1100 10010000001
------0-10 01000000100
.e
```

10) The result of the conversion from truth-table format to EQN format :

```
Q1 = IN2 & IN3 & IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q2 = IN1 & IN2 & ! IN3 & ! IN6 & IN7 & ! IN8 & ! IN9
Q3 = ! IN2 & IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q4 = IN2 & IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q5 = IN2 & ! IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q6 = ! IN2 & ! IN3 & ! IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q7 = ! IN2 & IN4 & ! IN6 & IN7 & ! IN8 & ! IN9
Q8 = IN5 & ! IN6 & ! IN7 & ! IN8 & ! IN9
Q9 = IN6 & ! IN7 & ! IN8 & IN9
Q10 = ! IN6 & ! IN7 & IN8 & IN9
Q11 = IN6 & ! IN7 & IN8 & ! IN9
Q12 = IN6 & IN7 & ! IN8 & ! IN9
Q13 = IN6 & ! IN7 & ! IN8 & ! IN9
Q14 = ! IN6 & IN7 & IN8 & IN9
Q15 = ! IN6 & ! IN7 & ! IN8 & IN9
Q16 = IN6 & IN7 & IN8 & ! IN9
Q17 = ! IN6 & IN7 & IN8 & ! IN9
Q18 = ! IN6 & IN7 & ! IN8 & IN9
OUT1 = Q7 | Q12 | Q15 | Q16
OUT2 = Q3 | Q6 | Q7 | Q9 | Q18
OUT3 = Q2 | Q4 | Q5 | Q6 | Q10 | Q11
```

```
      | & IN3 & IN4 & ! IN6 & IN7
      | Q13 | Q14 | Q16 | Q17 | Q18
OUT4 = Q3 | Q8 | Q15 | Q17 | Q18
OUT5 = Q3 | Q6 | Q7
OUT6 = Q8
OUT7 = Q12
OUT8 = Q15 | Q16 | Q17 | Q18
OUT9 = Q9
OUT10 = Q9
OUT11 = Q4
OUT12 = Q11 | Q14
OUT13 = Q10 | Q11
OUT14 = Q2 | Q5
OUT15 = Q1
OUT16 = Q1
OUT17 = Q15 | Q16 | Q17 | Q18
OUT18 = Q13
```