CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science Department

Technical Report 4298

From Geometry to Logic

by

Tzu-Mu Lin

Submitted in Partial Fulfillment of the Requirements for Degree of

Master of Science

March 29, 1981

Abstract

Transformation between five different intermediate forms used in VLSI design are discussed. The intermediate forms are: the D language, Akers' Diagrams, transistor listings, the sticks standard, and CIF language. They represent architecture, logic, transistor, topology and geometric levels, respectively. To understand more about the relationships between these levels, a series of transformations from the CIF to the sticks standard, from the sticks standard to the transistor listing, and from the transistor listing to the Akers' Diagram are presented. By doing this, the description gap between the logical world and the physical world is bridged.

CAD developers often complain about the lack of a model that can be applied uniformly throughout the entire design process. Akers' Diagrams seem to meet this demand. This work highlights this point.

As an example, a shift register implemented in NMOS technology will appear many times in this thesis.

Index terms: Akers' Diagrams, CIF, Composition cell, D language, D Simulator, DBJ, Leaf cell, MOS Simulator, Sticks Standard, Transistor Listing.

Acknowledgements

Special thanks are due Dr. Carver Mead. His constant encouragement and stimulating instruction have made my graduate study at Caltech challenging and rewarding.

I wish to express my gratitude to Dr. Charle' Rupp, in whose office this thesis was born. His endless patience with my slow progress and his boundless enthusiasm for new ideas encouraged and guided me.

I would like to thank Erik Debenedictis and Catherine Liou for their effective help in transforming my ideas into written English.

I must also thank Peggy Li, with whom I shared many interesting ideas. The D simulator described in this thesis was a joint effort with her.

Many people in our department have helped me through all kinds of problems. I would like to extend my thanks to all of them, especially Marina Chen, Erik Debenedictis, Dick Lang and Dan Whelan.

The research described in this thesis was sponsored by Silicon Structures Project.

Table of Contents

List of Figures

1. Introduction

People are talking about design automation. The physical layout is the final goal, but they do not want to design at that level. This situation is quite similar to programming languages. The binary code is the only executable form, but people prefer to write programs at higher levels. As time goes on, all kinds of chip assemblers [1,2] and chip compilers [3,4] have been developed to help design in various ways. It is anticipated that integrated circuit design can be fully automated in the near future. The designer needs only to give a logic specification and the computer will take care of everything.

This goal is not very easy to achieve. The semantic gap between logic specification and physical layout is so great that the direct transformation is almost impossible. Stepwise approaches are taken and various intermediate forms have been developed. To the designer, CIF [5,6] can be considered at the lowest level since it maps directly into the geometry. Right above this level is the sticks standard [7] that describes the topology of the circuits. At the present time, circuits are mostly designed in one of the following two ways: 1) write programs, probably using embedded languages [8-11], and compile them into CIF files ; 2) use a sticks editor [12] to create sticks files and apply a compactor program [12] and sticks-to-CIF translator [13] to obtain the final CIF files. Once described in CIF formats, the circuits are ready for fabrication.

Simulation and testing are as important as the physical design itself. Functional simulation at the register transfer level is purely logical and has only a slight relationship to the physical layout. The MOS model [14,15] for simulating the bidirectional behavior of MOS elements is more accurate and has been used frequently. The input format to the MOS simulator is the transistor connection information, that is at a slightly higher level than the sticks representation. It is desirable to transform the CIF and the sticks standard to this format (refered to as <u>transistor listing</u>) so that simulation can be done [16].

At this point, the geometric level (CIF), the topological level (sticks standard), and the transistor level (transistor listing) are discussed. Although these are at three different levels, the gaps between them are relatively small. The pass transistors in the CIF files are bidirectional, the source and drain are symmetric, and the direction of signal flow cannot be determined beforehand. This property, that also holds at the topological and the transistor levels, is the reason for the very large gap between the logical and the physical descriptions of digital systems. Logic designers are more familiar with the logical operations that are strictly unidirectional (where a node can unambiguously be determined to be input or output) To bridge this description gap, the next transformation is from the transistor level to the logic level. The logic level descriptions may be in terms of boolean operators or the like. Here, the Akers' Diagrams [17] are selected because of their expressive power and close relationship to the physical behaviors of MOS elements [18,19,20] (detailed in section 4).

One may wonder why these geometry-to-logic transformations are discussed. It is well known that one needs to transform digital circuits from logic to geometry. What are the reverse transformations for? There are several reasons for this:

1) The complete transformation from logic to efficient geometry is too difficult to be realized in the near future. Most people still design cells at the geometric or stick level. In order to simulate these cells, geometry-to-logic transformations are necessary. The system described in [21] was designed for this same reason.

2) Even if the logic-to-geometry transformation is feasible in the future, it will be necessary to verify that the cells generated by the machine actually operate as desired. This reverse transformation provides a means for double-checking.

3) As stated above, the transformation from the logic level to the physical level is very difficult. It is anticipated that, by doing the reverse transformation, the relationships between these levels can be better understood. The experiences gained here may become very helpful in dealing with the logic-to-geometry transformation.

D language is a register-transfer-level hardware design language [22]. It has the same syntax as the programming language C [23], but with different interpretations. Digital systems can be effectively described in D which, in turn, may be compiled into the Akers' Diagrams. The Akers' Diagram is the meeting point between the logical world and the physical world. Hopefully in the future, the design task will be nothing more than writing D-like programs. The computer will transform them all the way to the geometric layouts.

description level	intermediate forms selected
architecture	D Language
logic	Akers' Diagram
transistor	Transistor Listing
topology	Sticks Standard
geometry	Caltech Intermediate Form

Fig.1.1.1 The Five Intermediate Forms Involved in This Thesis

The five intermediate forms involved in this thesis are shown in Fig.1.1.1. In the next two sections, the algorithms and examples of CIF-to-sticks and sticks-to-transistor transformations are presented. Section 4 is the central part of this thesis. It starts with a brief introduction to the D language and the Akers' Diagrams. The transformation from the transistor listing to the Akers' Diagram are then discussed in some detail. Described in section 5 is a simulator with an operating mechanism based upon the Akers' Diagrams. Conclusions and future work are briefly discussed in section 6. A SIMULA [24] program, called DAN (D analyzer) , is now in operation. Examples and Users' guides for the program are contained in the Appendix.

2. CIF to Sticks Standard Transformation

2.1 Assumptions

This section deals with the CIF to sticks transformation. A CIF file is a raw listing of all the geometric objects that appear in the physical layout. Boxes, wires and polygons interact with one another and produce different kinds of physical components. These components are abstracted in the sticks standard as transistors, contacts, and so on.

The components produced by these geometric objects can be investigated through a checkplot of the CIF files [25]. When a poly wire crosses a diffusion wire, one realizes that there is a transistor. When a cut is covered by a metal box, one knows that there is a contact. This transformation is simply to automate the component recognition process.

At this point, some decisions must be made regarding how well the performance of the transformation is intended to be. Should it be designed to deal with general geometry or only regular cells? Should it include the capability to check the design rules at the same time or only accept correct cells? It is well known that CIF itself does not impose any restriction upon the designer. One can design in whatever way one likes. For example, one can put a poly wire at one cell and another diffusion wire at another cell. These two cells overlap in such a way that these two wires intersect. From each individual cell, nothing can be detected; however, when this chip is fabricated, a transistor is created. This kind of design is acceptable to CIF, but is a bad design style. For such complicated systems as VLSI circuits, the most important thing is the management of complexity [6]. Structural design based upon the concept of separated hierarchy [26] is one of the better alternatives. For this reason, the first assumption is made that all the geometric objects contained in one circuit but defined in different cells overlap only at the edges. This assumption makes it possible to deal with everything on a local basis. The hierarchical structure of the original design remains unchanged after various transformations.

The next assumption is that the cells contain no design rule errors. Cells submitted to the system must pass through the design rule checker first. However, several of the design rule checking and the component recognition procedures are quite similar so that they could possibly be integrated into one system at a later date.

To further simplify the task, some additional assumptions are made. All wires must be described by Wire commands. Box, Polygon and Round-flash (circle) commands [7] are only allowed for creating components, not for wires. For instance, if two transistors are connected by using a box instead of a wire, there is no guarantee that this connection will be detected properly. On the other hand, wire commands are supposed to be used for wires and not for components. It is acceptable if one applies poly wires and diffusion wires to make transistors or use metal wires to cover a contact cut. However, if the cut box is replaced by a cut wire, then the system may fail to detect this contact. The above assumptions do restrict the applicability of the system; however, it really saves a lot of programming effort and processing time. This heuristic approach does work for most of the cells.

2.2 The Algorithms

At the bottom level, there are several routines that deal with the interaction of different geometric objects. These geometric objects include points, wires, boxes and polygons. Boxes only include objects with edges parallel to the x or y axes; otherwise they are considered to be polygons. Circles are approximated by boxes. Polygons are relatively hard to deal with so they are approximated by a series of boxes. This approximation is very rough, but it is not significant to the problem. We are only concerned about the existence and position of components, not their detailed physics.

The interaction between wires and boxes can be easily detected by comparison of the coordinates of the edge points [27]. This is done in several procedures with different parameters (wire-wire, wire-box or box-box). Whether a point is contained in a wire or a box can also be detected easily. All these geometric routines are technology-independent. The whole system can be easily adapted to different sets of design rules and technologies. At present, only NMOS and CMOS circuits based upon the Mead & Conway design rules [6] are considered. In such circuits, five kinds of components are intended to be detected:

1. Transistors: these include the enhancement and the depletion mode transistors in the NMOS circuits, and P-type and N-type transistors in the CMOS circuits.

2. Contacts: these include the metal-to-poly (RB), the poly-to-diffusion (BUTT) and the metal-to-diffusion (GB) contacts.

3. Connectors: these interface cells with the outside world. Each connector is associated with the edge information (LEFT, RIGHT, TOP or BOTTOM) and the layer information (METAL, DIFFUSION or POLY).

4. Subcells and their connection points.

5. Joints: these are the joint points between different wires in the same layer.

The front end of the system is the CIF parser which reads in the CIF files and creates the data structure shown in Fig.2.2.1.

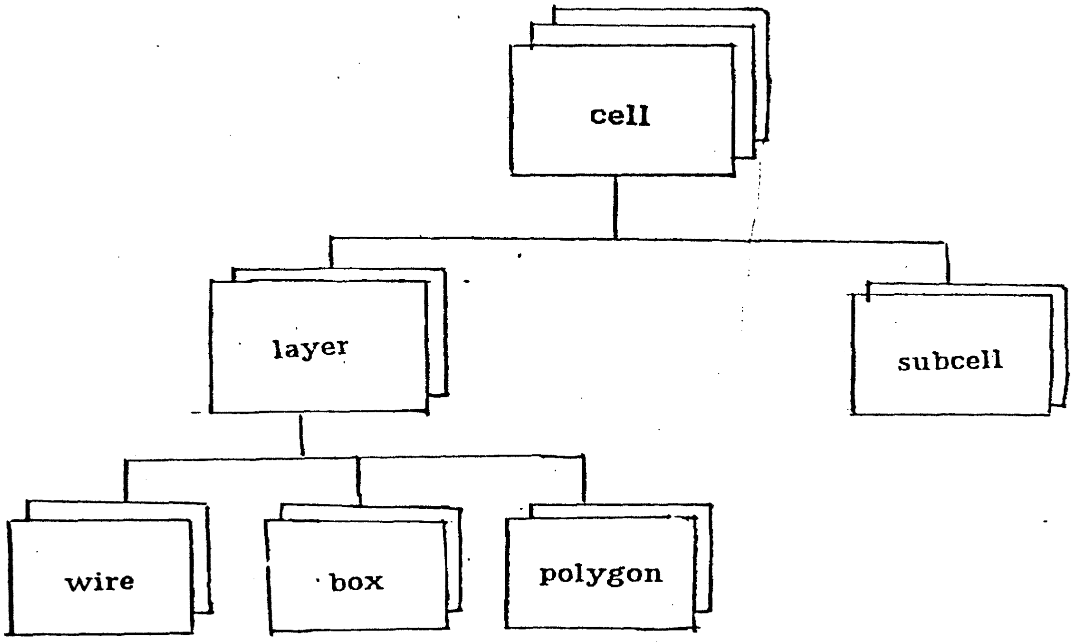


Fig.2.2.1 The Data Structure Created by the CIF Parser

Contained in the top level are a series of cell definitions. Each cell, in turn, is composed of layers of wires and boxes and instances of other cells. With this structure, all the components can be detected. This process is summarized in the following pidgin SIMULA code:

```
BEGIN

  PROCEDURE solve(c); REF(cell)c;
  INSPECT c DO IF NOT solved THEN BEGIN

      ! solve the subcells first ;
      INSPECT subcell-list DO FOR i:=1 STEP 1 UNTIL length DO
      INSPECT a.a[i] QUA subcell DO BEGIN
          solve(c);
          pass-connection-points;                              (P1)
      END of inspect;

      detect-contacts;                                         (P2)
      detect-transistors;                                      (P3)
      solve-wires;                                             (P4)
      detect-connectors;                                       (P5)
      solved:=TRUE;

   END of PROCEDURE solve-cell;

   INSPECT cell-list DO FOR i:=1 STEP 1 UNTIL length DO
   solve(a.a[i] QUA cell);

 END of CIF-to-sticks-transform;
```

(P1) : All the subcells must be solved first. All the connectors of these subcells are then passed to the calling cell. Such points will become either the connectors of the calling cell, if they are outside its shrinked bounding box (see (P5)), or the internal connection points, if inside. Internal connection points from different subcells will merge together if they overlap and belong to the same layer.

(P2) and (P3): In [28], a set of rules for detecting transistors and contacts in the NMOS circuits are described. They are repeated as follows:

Combination of Layers   Resultant Device

Poly & Diffusion ----> Transistor (N-type)
Transistor & Implant -----> Pull-up (NMOS)
(Transistor & P-type diffusion -----> P-type Transistor in CMOS)

Cut & Metal -----> To-Metal-Contact
To-Metal-Contact & Poly -----> RB-Contact
To-Metal-Contact & Diffusion ----> GB-Contact
RB-Contact & GB-Contact -----> Butt-Contact

Checking contacts starts with the cut boxes and checks if they are covered by metal (only metal boxes and metal wires with width no less than 4 lambda are checked). The investigation of these cuts with poly or diffusion determines the type of the contact. A check is then made for butting contacts, if the original CIF file contains two cut boxes, they are merged into one. This merging operation is treated uniformly throughout the whole component recognition process since, with the assumption that no design rule error may occur, distinct components must be separated by at least 4 lambda. Different components with a distance less than 4 lambda will be merged into one. In procedure (P3), wire-wire and box-wire transistors are detected.

(P4): Components, detected in procedures (P2) and (P3) or passed from subcells in procedure (P1), are put to wires. The wire-wire transistors and the joint points between different wires of one layer are also detected at this stage. As described earlier, all components must be separated by at least 4 lambda; otherwise they will be merged into one. If a transistor is very near a joint point, then the joint point will be merged into the transistor (not the transistor merged into the joint point).

(P5) Connectors are detected by finding the minimum bounding box first. Since most CIF files already contain this information, it is obtained without recomputing. The bounding box is then shrinked by 4 lambda. The points that are outside this shrinked box are considered to be connectors.

Each component is associated with a name given by the system and a coordinate pair indicating its position. Each name is composed of a capital letter and an integer. The letter identifies the type of components: 'B' for connectors (bristles), 'N' for N-type (or enhancement mode) transistors, 'P' for P-type (or depletion mode) transistors, 'C' for contacts, 'J' for joint points and 'I' for internal connection points. The integer is sequentially assigned to each type of component. For instance, if there are three N-type transistors and two contacts, they are named N1, N2, N3, C1 and C2, respectively.

The sticks diagrams may be displayed and plotted using any graphic device defined in the super class DISPLA [29]. DISPLA is a SIMULA program that contains various control and plotting routines for several devices such as Charles terminal [30], GIGI terminal [31] and HP 7221 plotter [32]. Contents of subcells may be drawn at any level desired. Shown in Fig.2.3.5 is an NMOS shift register plotted at level 1 (the bounding boxes of subcells). The same cell plotted at level 99 (all the details) is shown in Fig.2.3.6.

2.3 Examples

Shown in Fig.2.3.3 is a 2-bit shift register implemented in NMOS technology. It is constructed by repeating the leaf cell, shown in Fig.2.3.1, four times along the horizontal direction. Its cell structure (composition cell) is revealed in Fig.2.3.2. Note that the subcells only overlap at the edges. The sticks diagrams obtained from the CIF-to-sticks transformation are shown in Fig.2.3.4-6. Fig.2.3.4 indicates the topology of the leaf cell. 8 connectors, 2 enhancement transistors, 1 pull-up resister, 2 joints and 3 contacts are detected. Fig.2.3.5 and Fig.2.3.6 show the composition part of the whole circuit. The connectors of the cell and the interconnection points between the subcells are shown and distinguished. The original CIF and the sticks standard obtained from the transformation are presented in Fig.2.3.7 and Fig.2.3.8.

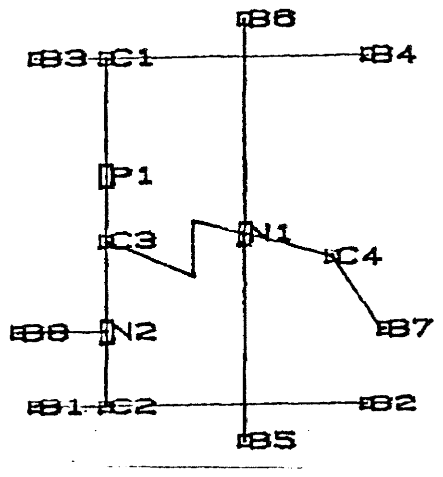


Fig.2.3.4 The Sticks Leaf Cell of the Half Shift Register

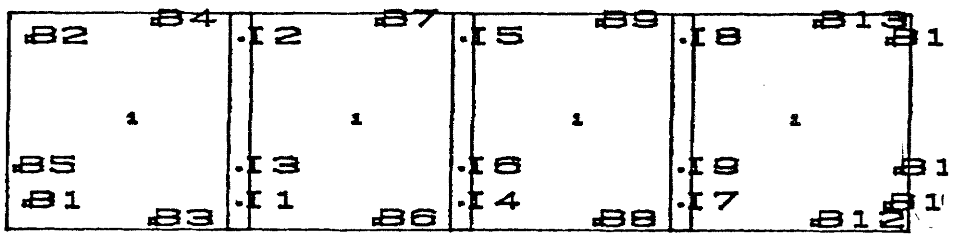


Fig.2.3.5 The Composition Cell under the Sticks Representation

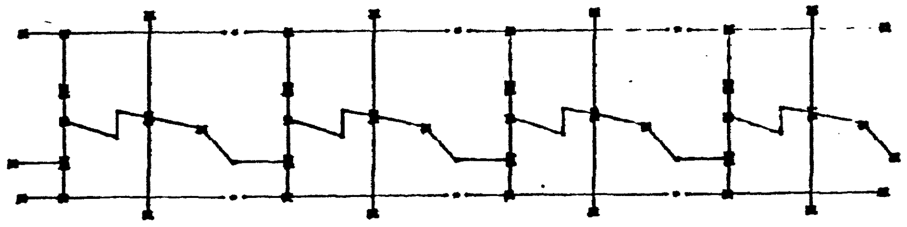


Fig.2.3.6 The Detailed Topology of the NMOS Shift Register

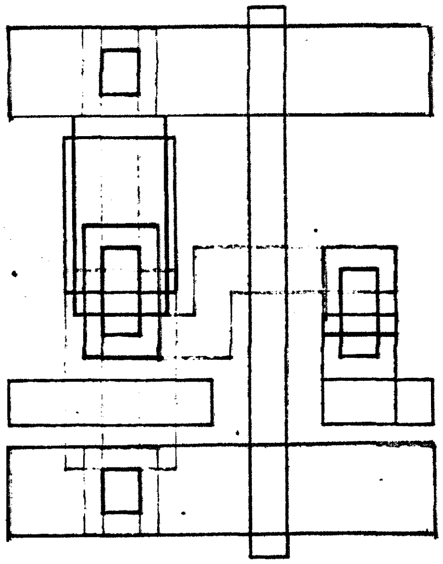


Fig.2.3.1 A Geometric Leaf Cell (Half Shift Register in NMOS)

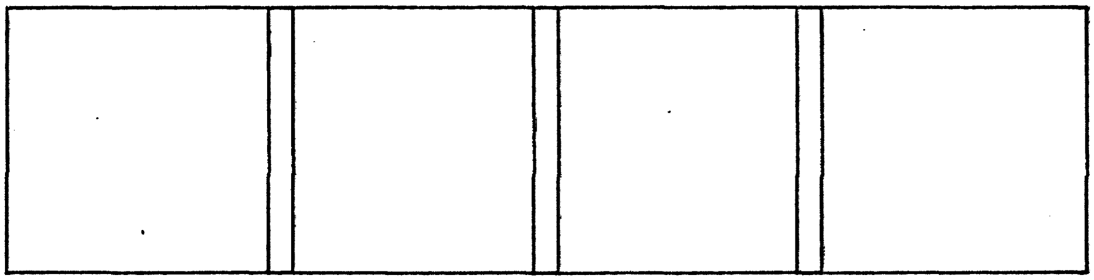


Fig.2.3.2 The Composition Cell of a NMOS 2-bit Shift Register

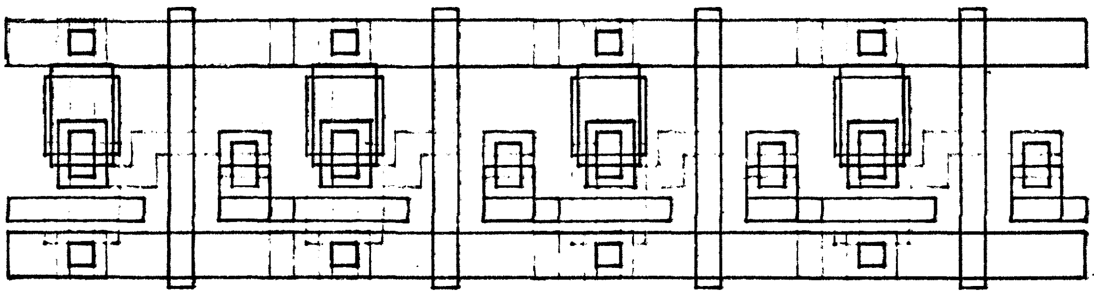


Fig.2.3.3 The Detailed Geometry of the NMOS Shift Register

```
(symbol SHIFT);
DS 1 250 10;
L NP;
W 20 130,0 130,230;
W 20 170,60 210,60;
W 20 0,60 90,60;
B 60 70 50,145;
B 40 50 180,75;
L ND;
W 20 50,220 50 10;
W 20 50,90 100,90 100,120 190,120;
B 40 40 50,20;
B 40 40 50,210;
B 40 40 180,110;
B 60 90 50,75;
L NM;
W 40 10,20 200,20;
W 40 10,210 200,210;
B 60 40 50,110 0,1;
B 60 40 180,100 0,1;
L NI;
B 50 90 50,145;
L NC;
B 20 20 50,20;
B 20 20 50,210;
B 40 20 50,110 0,1;
B 40 20 180,100 0,1;
DF;
(mbb is  -1,-1  22,24 in lambda);

(symbol CHIP);
DS 2 250 10;
C 1 T 0,0;
C 1 T 210,0;
C 1 T 420,0;
C 1 T 630,0;
DF;
(mbb is  -1,-1  85,24 in lambda);

E
```

Fig.2.3.7 The CIF Listing of the NMOS Shift Register.

```
CELL SHIFT 250 10
[ MBB is -1,-1 22,24  in lambda  ]

COMPONENTS
CONNECTOR :   B1 10 20  B2 200 20  B3 10 210  B4 200 210  B5 130 0
      B6 130 230  B7 210 60  B8 0 60;
NTRN :   N1 130 113  N2 50 60;
NRES :   P1 50 145;
NCON :   C1 50 210  C2 50 20;
NBUT :   C3 50 110  C4 180 100;

TWIGS
  Metal     : T1 =  B1 C2 B2;
  Metal     : T2 =  B3 C1 B4;
  Poly      : T3 =  B5 N1 B6;
  Poly      : T4 =  C4 B7;
  Poly      : T5 =  B8 N2;
  Diffusion : T6 =  C1 P1 C3 N2 C2;
  Diffusion : T7 =  C3 100,90 100,120 N1 C4;

CONSTRAINTS
END

CELL CHIP 250 10
[ MBB is -1,-1 85,24  in lambda  ]

COMPONENTS
CONNECTOR :   B1 10 20  B2 10 210  B3 130 0  B4 130 230  B5 0 60
      B6 340 0  B7 340 230  B8 550 0  B9 550 230  B10 830 20
      B11 830 210  B12 760 0  B13 760 230  B14 840 60;
POINT : I1 210 20  I2 210 210  I3 210 60  I4 420 20  I5 420 210
      I6 420 60  I7 630 20  I8 630 210  I9 630 60;
SHIFT 0 0 [ Connectors :B1 I1 B2 I2 B3 B4 I3 B5  ]
SHIFT 210 0 [ Connectors :I1 I4 I2 I5 B6 B7 I6 I3  ]
SHIFT 420 0 [ Connectors :I4 I7 I5 I8 B8 B9 I9 I6  ]
SHIFT 630 0 [ Connectors :I7 B10 I8 B11 B12 B13 B14 I9  ]

TWIGS
CONSTRAINTS
END
```

Fig.2.3.8 The Sticks Standard Created from the CIF Listing Shown in Fig.2.3.7

2.4 Remarks

As one may perceive, a heuristic approach has been taken for transforming the CIF to the sticks standard. Instead of the geometric objects themselves, only their centers (the paths of wires and the centers of boxes) are considered. All transistors act as ideal switches, and all wires are of zero delay. The results are far from general but are sufficient for logic analysis. However, if one also wants to analyze the electrical behavior, one may feel quite disappointed. One possible solution for this is to apply the polygon package [33]. That package deals with general geometry.

3. Sticks Standard to Transistor Listing Transformation

3.1 Another Intermediate Form

Although described in a more logical manner than the CIF, the sticks standard is still in the physical domain. This is of the following reasons: 1) Many points in the sticks standard are of no logical meaning at all. Bending points of wires are of this kind (refer to Fig.2.3.8). In the sticks standard, such points, without names, are just described by their original coordinate pairs. They are also refered to as numbered-points, while real components are refered to as named-points. Numbered-points cannot be omitted since they affect the path of wires. The sticks standard serves as an intermediate form for creating CIF files. At this level of description, the path of wires is left unchanged. 2) Many named-points effectively refer to only one logical point. Components that are in the same poly or metal wire or in the same diffusion wire without transistors on it are of this kind. These points are electrically equivalent and always have the same logical value. For the same reason, they must all appear in the sticks standard. These two facts make the sticks standard a physical description, not a logical one. In fact, the only information necessary to determine the logical behavior of a circuit are the following information of each transistor : 1) the type of the transistor, 2) the signal that connects to the gate, and 3) the two signals that connect to the drain-source pair of the transistor. To do any kind of logic analysis, one needs to filter out all the numbered-points, merge named-points into a non-redundant set of logic points (such points will be refered to as nodes later), and express the connection of transistors in terms of these nodes.

Shown in Fig.3.1.1 is the sticks diagram of a 2-input NOR gate implemented in CMOS technology. The corresponding sticks standard is listed in Fig.3.1.2. Originally, there are 13 named-points (the transistors are not included) in the sticks description. However, only 5 of them remain after filtering (refer to Fig.3.1.3).

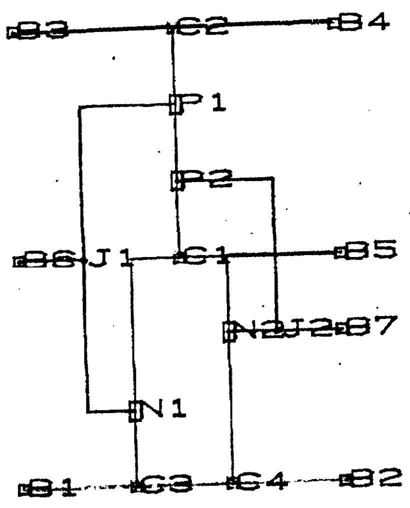


Fig.3.1.1 The Sticks Diagram of a CMOS 2-input NOR Gate

```
CELL NOR 250 10
[ MBB is -1,-2 22,32  in lambda  ]

COMPONENTS
CONNECTOR :   B1 0 0  B2 200 0  B3 0 300  B4 200 300  B5 200 15A0
      B6 0 150  B7 200 100;
NTRN :   N1 70 50  N2 130 100;
PTRN :   P1 100 250  P2 100 200;
CONTACT :   C1 100 150  C2 100 300  C3 70 0  C4 130 0;
POINT :   J1 40 150  J2 160 100;

TWIGS
  Metal     : T1 =  B1 C3 C4 B2;
  Metal     : T2 =  B3 C2 B4;
  Metal     : T3 =  C1 B5;
  Poly      : T4 =  B6 J1;
  Poly      : T5 =  N1 40,50 J1 40,250 P1;
  Poly      : T6 =  N2 J2 160,200 P2;
  Poly      : T7 =  B7 J2;
  Diffusion : T8 =  C3 N1 70,150 C1 P2 P1 C2;
  Diffusion : T9 =  C1 130,150 N2 C4;

CONSTRAINTS
END
```

Fig.3.1.2 The Sticks Standard of the CMOS NOR Gate.

node number	rep.	members
node 1	B1	B1, C3, C4, B2
node 2	B3	B3, C2, B4
node 3	B5	B5, C1
node 4	B6	B6, J1
node 5	B7	B7, J2

Fig.3.1.3 The Node Listing of the CMOS NOR Gate.

Every node is named by one of the components connected to a physical node. When there are several possibilities, the most important is chosen. The reason B5 is selected to represent node 3 (Fig. 3.1.1) is because that B5 is more significant than the other point C1. Likewise, B6, not J1, is selected to represent node 2. On the other hand, both B1 and B2 may be selected for node 1. Here B1 is arbitrarily selected because it is detected first. In general, connectors are considered to be more significant than contacts and joints.

There are four transistors in the circuit and their connections are expressed in the following listing:

name	type	gate	source-drain
P1	p	B6	P2 - B3
P2	p	B7	B5 - P1
N1	n	B6	B1 - B5
N2	n	B7	B5 - B1

Fig.3.1.4 The Transistor Listing of the CMOS NOR Gate.

Only logic nodes and transistors may appear in the transistor listings.

The transistor listing shown above represents a nonredundant set of logic information of the physical layout. It is completely general since no assumptions are made. Different analysis, simulation for instance, may be carried out from this level. With further assumptions about the circuit, one can transform this transistor listing into other forms that may be more convenient for certain kinds of analysis (In section 4, one such transformation will be discussed). For this reason, this format is considered to be a very good intermediate form for describing digital systems.

3.2 Algorithms and Example -- The Leaf Cell

As stated in section 3.1, two kinds of logical redundancies that exist in the sticks standards need to be removed. Numbered-points are relatively easy to deal with. One simply discards them when scanning through the wires. At the same time, one must be able to find all the components (named-points) that belong to the same logic node, choose the most appropriate component to represent this node (this special component is called <u>node representative</u>), and refer all the components to this node representative. Finally, all the connections of the transistors must be expressed in terms of the node representatives. All these are done in the following three procedures :

```
BEGIN

    find-representatives;
    solve-transistors;
    link;

END of sticks-to-transistor transform;
```

Each component is associated with a pointer, called iso , that points to its node representative. Only connectors, contacts and joints are candidates for node representatives. Connectors are, of course, the most desired. In the procedure find-representatives, all the poly and metal wires are scanned. If any component in a wire, say wire A, already points to a connector (this component must also belong to another wire, say wire B), then this connector, the node representative of wire B, also becomes the node representative of wire A. All components in wire A now point to this connector. If none of the components in wire A points to a connector, then node representative is selected according to the following preference ordering: 1) if any component in the wire is a connector, then this component is selected; else 2) if any component already points to some contact or joint, then the contact or the joint is selected; else 3) if there is a contact or a joint in the wire, then the contact or the joint is selected. Note that after the procedure find-representatives is completed, each transistor points to its gate node.

In the procedure solve-transistors, diffusion wires are processed. Diffusion wires without transistors are treated in the same way as poly and metal wires. Components on both sides of a transistor become the source-drain pair of that transistor.

As one may notice, the node representatives selected for certain wires may be changed later if the node representatives are not connectors. This change, however does not propagate through all the components in the wire. The procedure link is included to remove this inconsistancy.

The listing in Fig.3.2.1 indicates, wire by wire, how the sticks standard in Fig.3.1.2 can be transformed into the transistor listing shown in Fig. 3.1.4.

3.3 Algorithms and Examples -- The Composition Cell

As one may notice, some of the connectors in the leaf cells will be deleted after sticks-to-transistor transformation. To transform the composition cell, those parameters corresponding to the deleted connectors in the leaf cells are removed. Some modifications are necessary to link logic nodes through different subcells. This is illustrated in the following example:

The NMOS shift register described in section 2.3 will be applied again here. The sticks diagram is shown in Fig.2.3.4-6. The transistor listing for the leaf cell (Fig.2.3.4) is

name	type	gate	source-drain
P1	d		B3 - C3
N1	n	B5	C3 - B7
N2	n	B8	C3 - B1

(3.3.1)

Procedure	find-rep.							solve-tran.		link	
wires	T1	T2	T3	T4	T5	T6	T7	T8	T9		
(layer)	(metal)			(poly)				(diffusion)			
	-------------------------------------->										
component	(logic representative for each component)										result
B1	B1										B1
B2	B1										B1
B3		B3									B3
B4		B3									B3
B5			B5								B5
B6				B6							B6
B7							B7				B7
N1					B6			(B1,B5)			B6
N2						J2			(B5,B1)	B7	B7*
P1					B6			(P2,B3)			B6
P2						J2			(B5,P1)	B7	B7*
C1			B5								B5
C2		B3									B3
C3	B1										B1
C4	B1										B1
J1				B6							B6
J2						J2	B7				B7*

* pointer <u>iso</u> is changed during the search process.

Fig.3.2.1 The Steps for Transforming the Sticks Standard in Fig.3.1.2 into the Transistor Listing in Fig.3.1.4

There are 6 nodes in the circuit. Besides connectors, joint J1 is also a node. In fact, it is a state variable inside the shift register. Originally, there are 8 connectors. Only 5 remain after the transformation. They are B1, B3, B5, B7 and B8, respectively. B2, B4 and B6 have been deleted. There are four instances of this cell in the whole circuit. The first thing that needs to be done is to delete, for each subcell, the parameters corresponding to the connectors B2, B4 and B6 in the leaf cell. After this, the parameter lists of these four subcells become

subcell	parameter list			
	(Vdd)	(Gnd)		(Shift Sig.)
shift-1	B1	B2	B3	I3 B5
shift-2	I1	I2	B6	I6 I3
shift-3	I4	I5	B8	I9 I6
shift-4	I7	I8	B12	B14 I9

There are three signals travelling through these four subcells: Vdd, Ground and the shifted signal. In the above listing, the path of the shifted signal is already linked together. However, the Vdd and ground are not. For subcells shift-1 and shift-2, the Vdd is connected through the inter-connection point I1. However, I1 corresponds to different parameters for shift-1 and shift-2. Thus the link message is lost after the transformation. However, this information may be recovered through <u>iso</u> pointer. The resulting listing becomes

subcell	parameter list				
	(Vdd)	(Gnd)		(Shift Sig.)	
shift-1	B1	B2	B3	I3 B5	
shift-2	B1	B2	B6	I6 I3	
shift-3	B1	B2	B8	I9 I6	(3.3.2)
shift-4	B1	B2	B12	B14 I9	

3.4 Remarks

The transistor listings obtained from the sticks standards are compatible with the input format of MOS simulator [14,15]. With the CIF-to-sticks and sticks-to-transistor transformations, circuits designed at geometric or topological levels may be simulated. The MOS simulator is a pure logic simulator, however it may be modified so that certain physical behaviors can also be analyzed. Besides transistor connections, one also needs to know the size of the transistors and the width and length of the wires. All this information is optional in the sticks standard. At the present time, no physical property can be dealt with but the system may easily be expanded to include this capability.

4. Transistor Listing to Akers' Diagram Transformation

4.1 D language, DBJ notation and Akers' Diagrams

This section deals with the transistor to DBJ transformation. The DBJ notation is the text equivalents of an Akers' Diagram. The DBJ notation can be compiled from a D program. Before discussing the transformation, it is appropriate to say something about the D language, the DBJ notation and Akers' diagrams first.

D [22] is a hardware design language that describes the register transfer behavior of the digital systems. It has the same syntax as the programming language C, but with different interpretations. A simple D program is shown as follows:

```
/* 1-bit adder */
adder1(a,b,cin,y,cout)
Boolean a,b,cin,y,cout;
{  cout = ( a & b ) | ( a & cin ) | ( b & cin ) ;
   y = a ↑ b ↑ cin ;
}


/* 4-bit adder */
adder4(a0,a1,a2,a3,b0,b1,b2,b3,cin,y0,y1,y2,y3,cout)
Boolean a0,a1,a2,a3,b0,b1,b2,b3,cin,y0,y1,y2,y3,cout;
{ Boolean int0,int1,int2;

   adder1(a0,b0,cin,y0,int0);
   adder1(a1.b1.int0,y1,int1);
   adder1(a2,b2,int1,y2,int2);                    (4.1.1)
   adder1(a3,b3,int2,y3,cout);
```

D , as well as C, is full of logical operators : & for AND, | for OR , ~ for NOT, ↑ for XOR .... The 4-bit adder shown above is constructed from four 1-bit adders (cell adder1). Contained in the main cell are nothing but four instances of the cell adder1 with corresponding parameters. The cell adder1 is defined separately. Cout (carry out) is the majority of a, b and cin (carry in), whereas y (sum bit) is their XOR result.

Besides these logical operators, there are branch operations. Here is a simple example:

```
/* two to one multiplexer */
mux2to1(a,d0,d1,y)
Boolean a,d0,d1,y;
{ y=a0?d1:d0 ;                                  (4.1.2)
 }
```

y=a0?d1:d0 is the standard statement of C for describing conditional computation. y=a?d1:d0 is interpreted as " if a (is TRUE), then d1 is assigned to y else d0 is assigned to y ". In D, it is interpreted similarly as: if a equals 1, then d1 is assigned to y else (a equals 0) d0 is assigned to y. This is exactly the description of the 2-to-1 multiplexer. From the standpoint of programming language C, such operators as AND, OR, NOT, XOR are regular, while y=a?d1:d0-like statements are included mainly for dealing with branch conditions. In D, however, this conditional statement serves as the basic construct of the whole system. Every other statement is intended to be transformed into this format. The format is desirable since it models the behavior of MOS elements very nicely. In MOS circuits, the direction of signal flow is governed by the gate value of each transistor. If it is one, then the signal goes one way; if it is zero, then the signal goes another way. Every signal in the MOS circuits is something that either controls or be controlled.

D programs can be compiled into DBJ notation which is nothing but a series of (y=a?d1:d0)-like statements. As the simplest case , the D program for the 2-to-1 multiplexer shown above is compiled into the following code:

```
DEFI MUX2TO1
BOOA A
BOOA D0
BOOA D1
BOOA Y
DEFB
! Y=A0?D1:D0        (the original D statement)
NODE Y=A0?D1:D0
DEFE MUX2TO1
```

DEFI and DEFE statements start and end a block. BOOA statements declare variables. DEFB starts the definition of the block. Each NODE statement creates one conditional branch . Since the original D statement y=a0?d1:d0 is of the desired format it is not changed. The next example is the 4-to-1 multiplexer. The D program is like

```
/* 4-to-1 multiplexer */
mux4to1(a0,a1,d0,d1,d2,d3,y)
Boolean a0,a1,d0,d1,d2,d3,y;                    (4.1.3)
{ y=a0?(a1?d3:d2):(a1?d1:d0);
 }
```

After compilation, the DBJ notation (only NODE statements are shown) is:

```
! Y=A0?(A1?D3:D2):(A1?D1:D0)
NODE Y=A0?*2:*1
NODE *1=A1?D1:D0
NODE *2=A1?D3:D2
```

By using pointers, the compound statement Y=A0?(A1?D3:D2):(A1?D1:D0) is compiled into three basic statements. All branch structures can be dealt with similarly. In fact, this node structure is equivalent to a special notation for describing logic systems: the Akers' Diagrams [17]. The Akers' Diagram for example (4.1.3) is shown in Fig.4.1.1. Such diagrams have been widely used for analyzing large digital networks, especially for generating test vectors [17]. Several algorithms and advantages of applying Akers' Diagrams to analyze and synthesize VLSI circuits are discussed in [18,19]. Both D language and DBJ notation are based upon this idea. In later sections, the DBJ notation and the Akers' Diagrams will be used interchangeably.

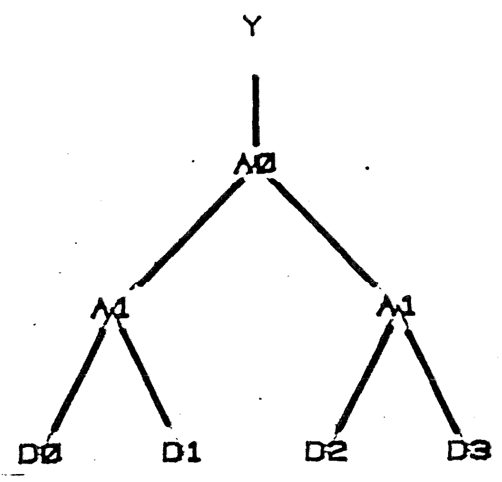


Fig.4.1.1 The Akers' Diagram of a 4-to-1 Multiplexer

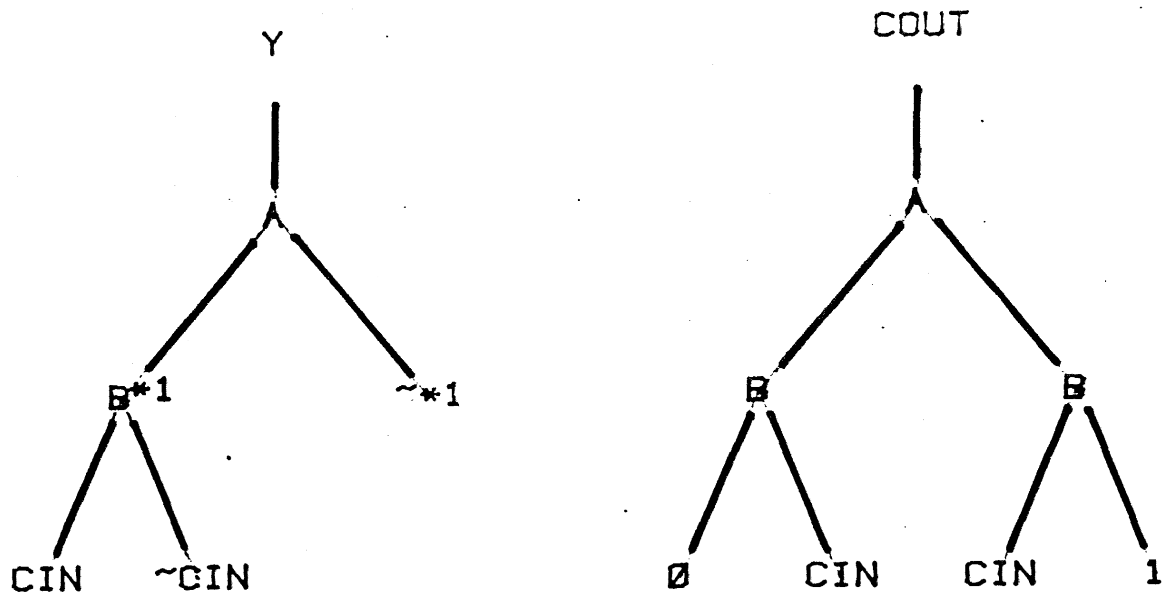


Fig.4.1.2 The Akers' Diagrams of the Two Output Variables of the 1-bit Adder Shown in (4.1.1)

The DBJ notation is also very effective for describing ordinary logical operators. For instance, a&b is transformed into a?b:0 and a|b is transformed into a?1:b. Other operators can be described similarly. Afer compilation, the DBJ notation for the 1-bit adder in (4.1.1) is as follows:

```
DEFI ADDER1
BOOA A
BOOA B
BOOA CIN
BOOA Y
BOOA COUT
DEFB
! COUT=(A&B)|(A&CIN)|B&CIN
NODE COUT=A?*2:*1
NODE *1=B?CIN:0
NODE *2=B?1:CIN
! Y=A↑B↑CIN                              (4.1.4)
NODE Y=A?~*1:*1
NODE *1=B?~CIN:CIN
DEFE ADDER1
```

The corresponding Akers' Diagrams are shown in Fig.4.1.2

The DBJ notation also allows block structures. For instance, the D program for the 4-bit adder, shown in (4.1.1), may be compiled into the following code:

```
DEFI ADDER4
BOOA A0
BOOA A1
BOOA A2
BOOA A3
BOOA B0
BOOA B1
BOOA B2
BOOA B3
BOOA CIN
BOOA Y0
BOOA Y1
BOOA Y2
BOOA Y3
BOOA COUT
DEFB
BOOA INT0
BOOA INT1
BOOA INT2
BLDB ADDER1
PARM A0
PARM B0
PARM CIN
PARM Y0
PARM INT0
BLDE ADDER1
BLDB ADDER1
PARM A1
PARM B1
PARM INT0
PARM Y1
PARM INT1
BLDE ADDER1
BLDB ADDER1
PARM A2
PARM B2
PARM INT1
PARM Y2
PARM INT2
BLDE ADDER1
BLDB ADDER1
PARM A3
PARM B3
PARM INT2                                    (4.1.5)
PARM Y3
PARM COUT
BLDE ADDER1
DEFE ADDER4
```

The DEFB statement starts the definition of cell adder4. After that, the three BOOA statements declare the three internal variables int0, int1 and int2. There are four BLDB statements, each of which corresponds to one instance of the cell adder1. In each instance, the five PARM statements map the parameters from cell adder4 to cell adder1. The BLDE statement ends each cell instance. Note that the description hierarchy used in the D language and the DBJ notation is compatible with that used in CIF, sticks standards and transistor listings.

4.2 MOS vs Logic

The transistor listings obtained in section 3 represent logical relationships between different transistors in the circuits. From these listings, logic simulation can be done under the MOS model. In the model, every transistor is bidirectional, and the direction of signal flows is determined at run time. This model is good in the sense that it models the physical behavior of MOS elements. It does not, however bridge the description gap between the physical world and the logic world. The logic itself is unidirectional. There may be different methods, unidirectional or bidirectional, to implement this function. However, at the logic level of description, everything must be expressed unidirectionally. In fact, even in the MOS circuits, most of the signals flow only in one direction, although both directions are physically possible. In most cases, the bidirectional specification represents some kind of redundancy.

For the above reason, it is intended to transform the transistor listings into some logical representations. The Akers' Diagram is selected as the target, since it is as fundamental and general as other logic model [34-37], and also resembles the behavior of MOS elements. The Akers' Diagram is strictly unidirectional.

Besides the bidirectionality, another redundancy is intended to be removed under this transformation. All nodes in the transistor listings are treated as variables. This is also true for those nodes that are supposed to connect to Vdd or ground. The transistor listings are so general that they also allow such nodes to be changeable. For all practical purpose, however, such nodes will always remain at fixed values. For this reason, they are replaced by constants in the Akers' Diagrams. The Akers' Diagram is something that really belongs to the logical world.

4.3 Examples

The transformation must start from some output or state variables since only output and state variables are associated with Akers' Diagrams. MOS elements are bidirectional, however. It cannot be determined which nodes are inputs and which are outputs. At first glance, it is hopeless. However, some observations of usual MOS circuits suggest that restoring logic may be a good point from which to start, since it always acts as an output or a state variable. In a CMOS circuit, such points are the joint points between the N network and the P network. In the NMOS circuit, they are the joint points between the N network and the pull-up resistors.

The CMOS NOR gate presented in section 3.1 will be applied again. The sticks diagram is shown in Fig.4.3.1. and the transistor listing is repeated as follows:

name	type	gate	source-drain	
P1	p	B6	P2 - B3	
P2	p	B7	B5 - P1	
N1	n	B6	B1 - B5	(4.3.1)
N2	n	B7	B5 - B1	

Either from the diagram or from the listing, one immediatedly finds that B5 is a restoring point from which the Akers' Diagrams can be constructed. Connected to B5 are two networks, one NMOS and one PMOS. Both networks contribute to the value of B5, and hence B5 is called a bus variable. They had rather be treated separately, and B5 actually acts as a bus variable. Later on, B5 will be associated with two Akers' Diagrams. When compiled from D programs, the DBJ notation may also contain bus variables. The bus concept is very common in the digital systems, so it is included here.

P network is dealt with first. After the two serial transistors p1 and p2, comes the connector B3. Beyond B3 is the outside world that is undefined. One has to stop here and makes some decisions. The sticks diagram suggests that B3 is a Vdd node. However, it may not be. It must not be an output or a ground node, but it may be an input variable. In any case, insufficient information is available from the transistor listing and is necessary to ask the user for help. The question is "Is this connector an input, an output, Vdd or ground?" The user enters this information, and the Akers' Diagrams are constructed accordingly. Suppose that the user enters Vdd for B3, then the DBJ notation for the PMOS part becomes

```
NODE B5=B7?X:*1
NODE *1=B6?X:1
```

If B7 is logic one, then transistor P1 is open. The P network makes no contribution to the value of B5 and is therefore assigned the value X (standing for undefined). If B7 is zero, then pointer *1 is investigated. At this point, B5 will be one if B6 is zero, or it will be undefined if B6 is one. The DBJ notation describe this behavior beautifully. On the other hand, there are two parallel transistors in the NMOS network. Likewise, the user has to supply the type information of connector B1, probably ground signal. Suppose this is exactly what the user enters, then the DBJ notation for the NMOS network is as follows:

```
NODE B5=B6?0:*1
NODE *1=B7?0:X
```

If B6 is one then transistor N1 is closed and B5 will become zero. If B6 is zero, then pointer *1 is investigated. At this point, B5 will be zero if B7 is one, or it will be undefined if B7 is zero. The overall DBJ notation for the CMOS NOR gate is

```
DEFI NOR
BOOA B5
BOOA B6
BOOA B7
DEFB
! P-NET
NODE B5=B7?X:*1
NODE *1=B6?X:1
! N-NET
NODE B5=B6?0:*1                              (4.3.2)
NODE *1=B7?0:X
DEFE NOR
```

B5 is a bus variable and associated with two Akers' Diagrams as shown in Fig.4.3.2. In this particular example, one and only one of the two diagrams will return a non-undefined value. Either one or zero will overwrite the undefined value. If one and zero are written to the same place, a conflict happens and this error is represented by a value Z. In this example, such error will never occur. For others that will, the system provides a means for detecting them.

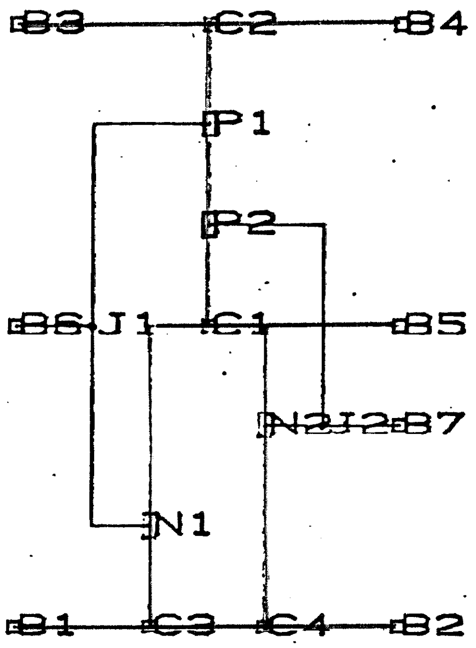


Fig.4.3.1 The Sticks Diagram of a CMOS 2-input NOR Gate

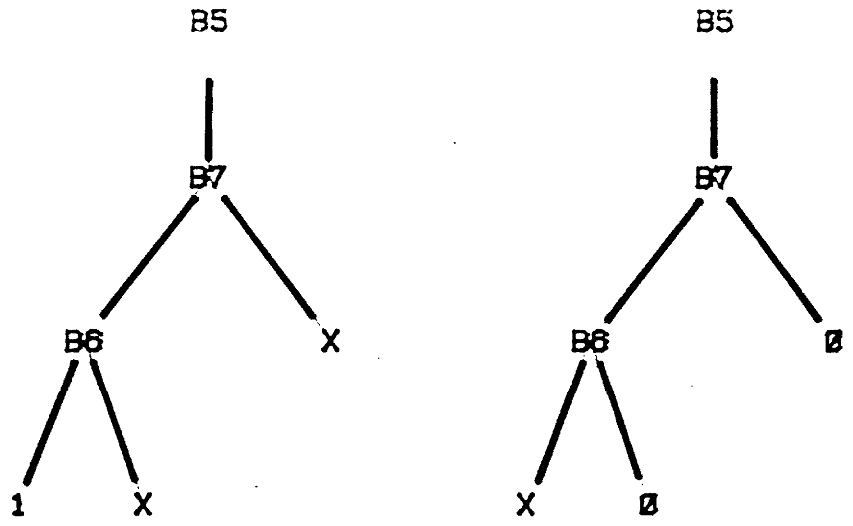


Fig.4.3.2 The Akers' Diagrams of the Bus Variable B5

It is instructive to compare the transistor listing (4.3.1) and the DBJ notation (4.3.2) for this particular example. In the transistor listing, there are five variables. Only three remain in the DBJ notation. B1 and B3 become constants. In the transistor listing, all data are unstructured. It is difficult to determine the logical behavior of the circuit. In the DBJ notation, however B6 and B5 are input variables (this information is not provided by the user, there is no ambiguity as to whether they are inputs or outputs) and B7 is " some logic " function of B5 and B6. From this simple example, the roles of of the DBJ notation and the transistor listing are clear. Roughly speaking, the transistor listing is general and unstructural, while the DBJ notation is strict and structural. The DBJ notation is in the logical domain, while the transistor listing is between the logical and the physical domain. Both are very important in VLSI designs, and this transformation bridges their gap.

Suppose, in the same example, the user enters a type "input" for both connectors B1 and B3, then the DBJ notation becomes

```
DEFI NOR
BOOA B1
BOOA B3
BOOA B5
BOOA B6
BOOA B7
DEFB
NODE B5=B7?X:*1
NODE *1=B6?X:B3
NODE B5=B6?B1:*1
NODE *1=B7?B1:X
DEFE NOR
```

B3 and B5 remain variables now.

Provided that the NOR gate in (4.3.2) is replaced by the NAND gate, i.e., the P-type transistors are in parallel and the N-type transistors are in serial, then the DBJ notation becomes

```
!P-NET
NODE B5=B7?*1:1
NODE *1=B6?X:1
!N-NET
NODE B5=B6?*1:X                                         (4.3.3)
NODE *1=B7?0:X
DEFE NOR
```

Comparing (4.3.2) with (4.3.3), one may easily find what in the DBJ notation is affected by P-type transistors, what is affected by the N-type transistors, what is affected by the serial connection, and what is affected by the parallel connection. These, together with a new technique, called <u>backtrack</u>, are general enough to deal with almost all kinds of circuit structures. The detailed algorithms will be given in section 4.4. Following is another example which the user assigns some connectors as output variables.

Shown in Fig.4.3.3 is a 6-transistor NMOS circuit. B3 is connected to Vdd. For NMOS circuits, It is always assumed that pull-up resistors are connected to Vdd since they are of no value otherwise. Starting from J3, there are three transistor paths. Both paths N3-N1 and N4-N5 lead to connector B1 that is probably connected to ground. Path N5-N6 leads to connector B11, that may either be an input or an output. Suppose it is an input, then this situation is basically the same as the last example. J3 is a function of B5, B7, B6, B8, B9, B10 and B11. In this case, however, the value of J3 is not available from the outside world so probably B11 is an output, not an input. If it is indeed an output, then B11 will be a function of J3, not the other way around. Paths N3-N1 and N4-N2 are left unaffected, whereas the direction of path N5-N6 must be reversed. J3 serves as an state variable and the actual output variable is B11. Both J3 and B11 are associated with their own Akers' Diagrams as shown in Fig.4.3.4. The process of reversing the direction of signal flows, as what was done for path N5-N6, will be refered to as <u>backtrack</u> later. When an output connector is encountered, one needs to backtrack. When the gate of a transistor is reached, one needs to backtrack. When a restoring point is encountered, one needs to backtrack. All these situations are dealt with similarly.

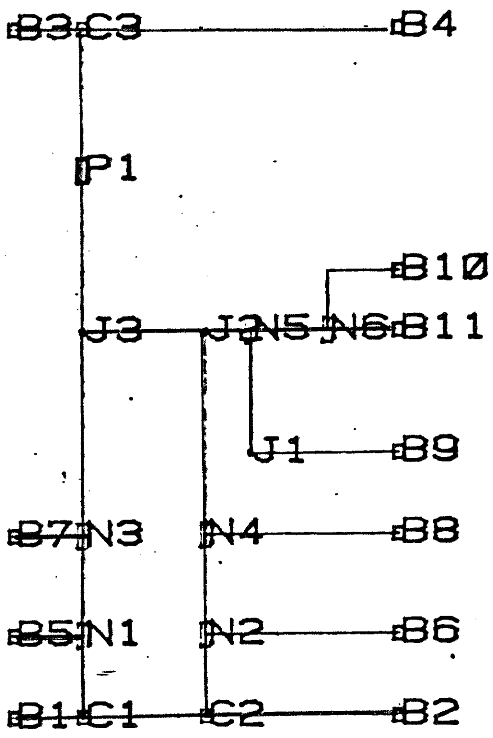
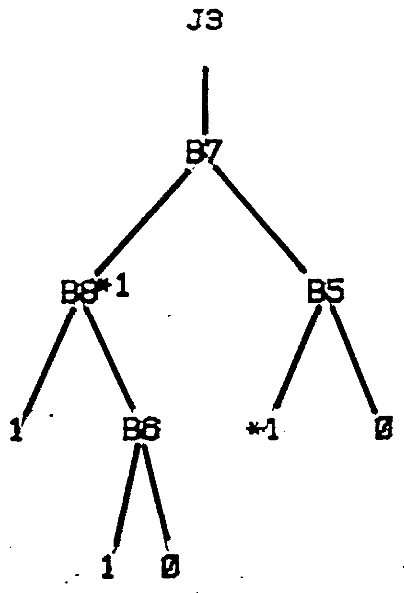


Fig.4.3.3 The Sticks Diagrams of an NMOS Circuit with State Variables



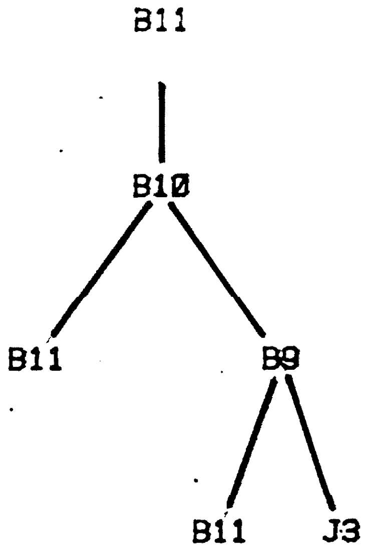


Fig.4.3.4 The Akers' Diagrams of J3 and B11

The way the system deals with the bidirectionality of pass transistors is best illustrated by the following examples. The sticks diagram of the circuit is shown in Fig.4.3.5. B1 is connected to ground and B5 is connected to Vdd. From B4, there are four paths that lead to ground. They are N1-N2, N1-N3-N5, N4-N5, and N4-N3-N2, respectively. Here, transistor N3 does work as a bidirectional element, however it causes no problem. It is still possible to transform this circuit into unidirectional descriptions and construct Akers' Diagrams accordingly. This is by way of backtrack. Immediatedly following B4, there are two transistor paths: path N1 leads to J2 and path N4 leads to J1. J2 is examined first. Besides N1, two other paths also connect to J2. Path N2 leads to ground and can be dealt with very easily. Path N3 leads to J1 which, in turn, is connected to N4 and N5. Path N5 leads to ground and causes no problem. However, path N4 leads to the starting point B4. A loop is detected, indicating a wrong turn somewhere in the search. This error can be found by backtrack. Through the transistor N4 return to J1 that is followed by paths N3 and N5. At this point, path N3 is in the inward direction and is of no immediate help. Path N5, however, is in the outward direction and is exactly what is attempted. Here is the very point that the wrong decision was made. Path N5 should be selected instead of N4. It is never too late to mend, path N4 is simply ignored and everything is set. The N3-N5 path is already connected. So far, path N1 has been solved, and the corresponding Akers' Diagram is shown in Fig.4.3.6. Path N4 can be dealt with almost equally, and the corresponding Akers' Diagram is shown in Fig.4.3.7. The final Akers' Diagram is simply their combination, however the intermediate nodes within each subdiagram may be shared (Fig.4.3.8).

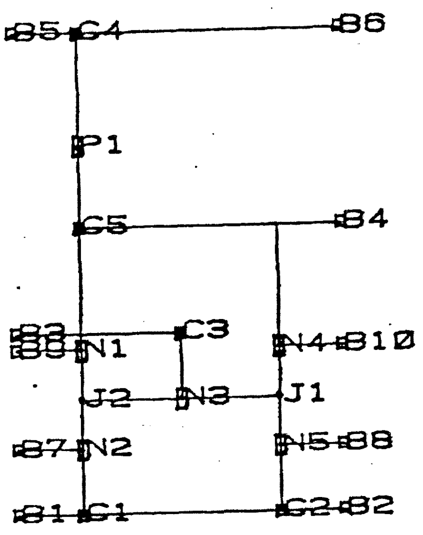


Fig.4.3.5 The Sticks Diagram of an NMOS Circuit Illustrating the the bidirectionality of MOS Transistors

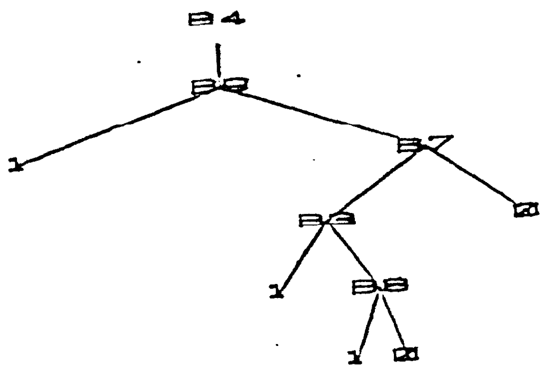


Fig.4.3.6 The Akers' Diagram Corresponding to Path N1

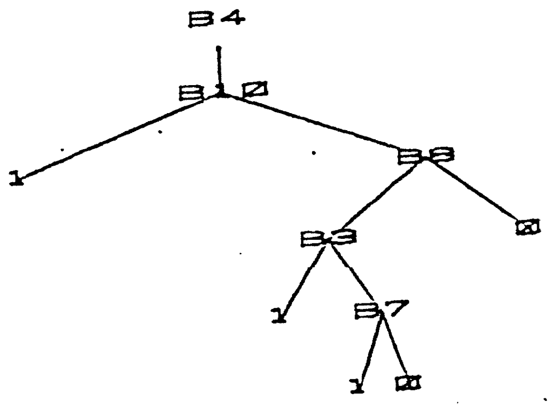


Fig.4.3.7 The Akers' Diagram Corresponding to Path N4

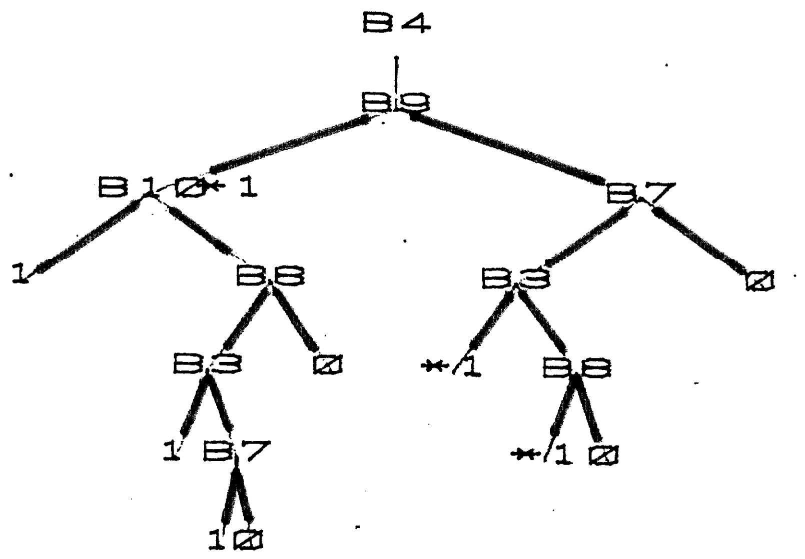


Fig.4.3.8 The Akers' Diagram for the Whole Circuit in Fig.4.3.5

4.4 The Algorithms

With the examples presented in section 4.3, the transformation will be discussed in a more formal manner. The whole process is summarized in the following pidgin SIMULA code:

```
BEGIN

    create-data-structures;                    (P1)
    FOR each restoring node DO BEGIN
        search-forward;                        (P2)
        search-backward;                       (P3)
        solve-backward;                        (P4)
        solve-forward;                         (P5)
    END;

END of transistor-to-DBJ transform;
```

There are two other procedures: 1) othernode : Given a transistor and a component (either a transistor or a node) that belongs to the source-drain pair of that transistor, this procedure will return the other node (or transistor) of that pair. 2) iotype : This procedure accepts a connector, and determines if its type is already known. If not, it will ask the user for help. Once available, this information is returned to the calling procedure.

(P1): In the transistor listing, each transistor is associated with three nodes: the gate and the source-drain pair. To obtain the Akers' Diagrams, the transistors connected to each node must be known (Fig.4.4.1). In fact, both of these structures can be created during the sticks-to-transistor transformation. All the restoring nodes can be detected from the connection information. Each node is associated with a boolean flag restoring, which will be set for restoring nodes.

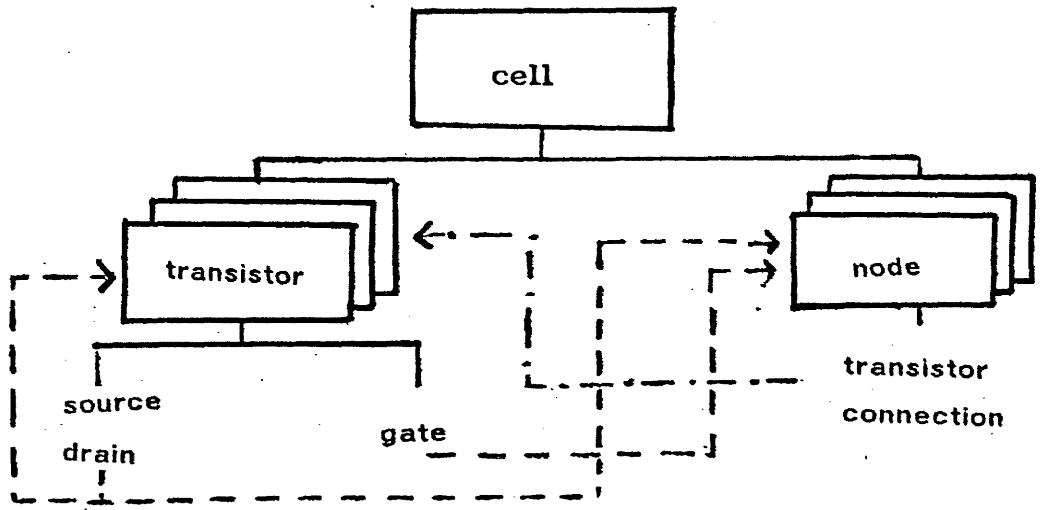


Fig.4.4.1 The Relationship between Transistors and Nodes

(P2) and (P3): Serial transistors can be treated as a whole since signal always flows through them in the same direction. A new data structure path is introduced for this purpose. Each path corresponds to one such series of transistors. In (P2), all such paths are searched from a certain restoring node. Every transistor that is immediatedly connected to this restoring node will create a path. All such paths are in parallel. Procedure othernode is applied here to find the other component of each transistor. If a transistor is returned, then these two transistors are in series and belong to the same path. This procedure is called repeatedly until it returns a non-transistor component (a node). This node and the original restoring node may be considered as the source-drain pair of the transistor path. In the ordinary sense, the source and drain are symmetric and the direction of the signal flow is unknown. In this case, however, it is believed that the signal is more likely to flow out from the restoring node (It means that current flows out from the restoring node in the N network and flows into the restoring node in the P network). Through these paths, the restoring nodes are able to communicate with other nodes that, in turn, may create their own family of paths. This process is repeated recursively until each of the terminating paths reaches a terminating node. Terminating nodes are those nodes that contain no path of their own. Terminating nodes are of two kinds: 1) If they are connectors, the procedure iotype is called, and the returned type message is recorded. 2) If they are internal terminating nodes : such nodes are possibly connected to the gate of transistors. Paths are created in a hierarchical manner. Every path is directive and associated with two nodes. The node that near the restoring logic is called the source of the path, whereas the other node is called the drain. The normal direction of the signal flow is from source to drain. During backtrack condition, however, the direction of the signal flow will be reversed.

After (P2), the data structure of the whole circuit is in terms of paths (Fig.4.4.2). All paths that connected to the same source are linked together with a two-way list. These paths are in parallel. The source node, at the same time, holds a pointer, called chain, that refers to (the head of) the list of paths. Besides a series of transistors, each path also records the following information: 1) drain: this refers to the drain of the path; 2) pre, suc: these refer to the preceding and the succeeding paths in the same list.

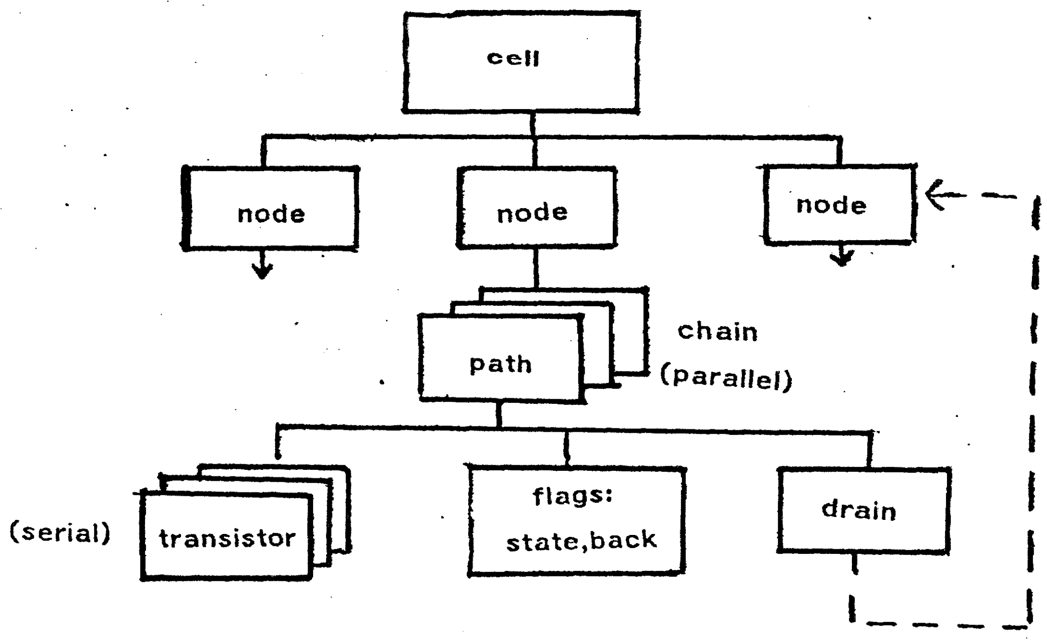


Fig.4.4.2 The Cell Structure Described under the Concept of Path

3) back, state : these two boolean flags indicate the type of the path. The setting of back flag implies that the direction of the signal flow along this path is reversed. The drain of such paths will be refered to as back nodes later. The setting of state flag implies that the drain of this path is a state (or an output) variable. For the terminating path, both the state and back flags will be set if the corresponding terminating node is an internal terminating node or an output connector. For the intermediate path, state flag will be set if some of the back flags of the paths in the chain of its drain are set. Back flag will be set if all the back flags of the paths in that chain are set. It is obvious that all back nodes are also state nodes (variables). In procedure (P2), the setting of both flags are decided for all the terminating paths. The intermediate nodes are dealt with in procedure (P3).

(P4) and (P5): The Akers' Diagrams for the back nodes are constructed in (P4), and those for the non-back state nodes, including the restoring nodes, are done in (P5). Both procedures are basically the same, except that the direction of the construction is different. In (P4), all paths are searched from drain to source while in (P5), source to drain. All the transistors in the same path are serial so the corresponding DBJ notation is of the following form :

```
A=G1?*1:X                              A=G1?X:*1
*1=G2?*2:X                             *1=G2?X:*2
. . . .              (N-net)           . . . .            (P-net)
*(n-1)=Gn?B:X **                       *(n-1)=Gn?X:B
```

(4.4.1)

** In the NMOS circuit, the last X will be replaced by '1'. When all the pull-down transistors are turned off, the Vdd signal will activate through the pull-up resistors.

In the backward case (P4), A is the drain of the path and B is the source of the path. In the forward case (P5), A is the source and B is the drain. G1, G2 ... Gn are the gate nodes of the transistors along this path. G1 is the one nearest to node A, and Gn is the one nearest to node B. If node B is a state node, whether a back node or not, it will associate with its own Akers' Diagrams, and the procedure stops. However, if node B is not a state node, it should not appear in any place of the Akers' Diagrams. In this case, B is replaced by another pointer from which the construction of the Akers' Diagram continues.

For parallel paths, the DBJ notation is of the following form:

```
A=G1?B1:*1                             A=G1?*1:B1
*1=G2?B2:*2                            *1=G2?*2:B2
. . . .              (N-net)           . . . .            (P-net)
*(n-1)=Gn?Bn:X                         *(n-1)=Gn?X:Bn
```

(4.4.2)

It is assumed, for simplicity, that each path contains only one transistor. There is a total of n paths in the chain. The gate nodes are G1, G2 ...

Gn, and the corresponding drain nodes are B1, B2 ... Bn. This is for the forward case. In the backward case, the only modification is to interchange the source and drain. All DBJ notation can be constructed by various combinations of (4.4.1) and (4.4.2).

Here is an example illustrating how this algorithm works. The sticks diagram for the circuit is shown in Fig.4.4.1. B5 is connected to Vdd, and B1 is connected to ground. There are three restoring points: J3, J2 and B4. From J3, there are two paths. Path N1-N2 leads to B1 (ground) and is in the normal (forward) direction. Path N6 leads to node C8 which, in turn, creates another two paths. If B18 is of type input, then path N9 is in the forward direction. Path N7 leads to node C9 that is followed by path N4 and path N8. Path N4 leads to gate node C6, and path N8 leads to restoring point J2. Both paths need backtrack, so their back flags are set to one. Now procedure (P2) is finished. Among the four terminating paths, N1-N2 and N9 are in the forward direction, while N4 and N8 are in the backward direction (refer to Fig.4.4.2). The status of intermediate paths (or nodes) are determined in procedure (P3). Since the back flags are set for both path N8 and path N4, C9 is a back node, and the back flag of path N7 is set to one. The next node is C8 whose status is determined by both paths N7 and N9. The back flag is set for N7 but not for N9. Thus C8 is a state node but not a back node. Path N6 is still in the forward direction. The <u>path-tree</u> constructed during procedure (P2) and (P3) is shown in Fig.4.4.2. In procedure (P4), the Akers' Diagrams for the back nodes C6 and C9 are constructed. J2 is a restoring node and will create its own path-tree later. In procedure (P5), the Akers' Diagrams for the non-back state nodes C8 and J3 are constructed. Finally, the DBJ notation for this family of nodes is

```
NODE C6=B13?C9:C6
NODE C9=B9?C8:C9
NODE C8=B15?B19:C8
NODE J3=B18?*1:*2
NODE *1=B17?0:*2
NODE *2=B7?C8:1
```

(only NODE statements are shown)

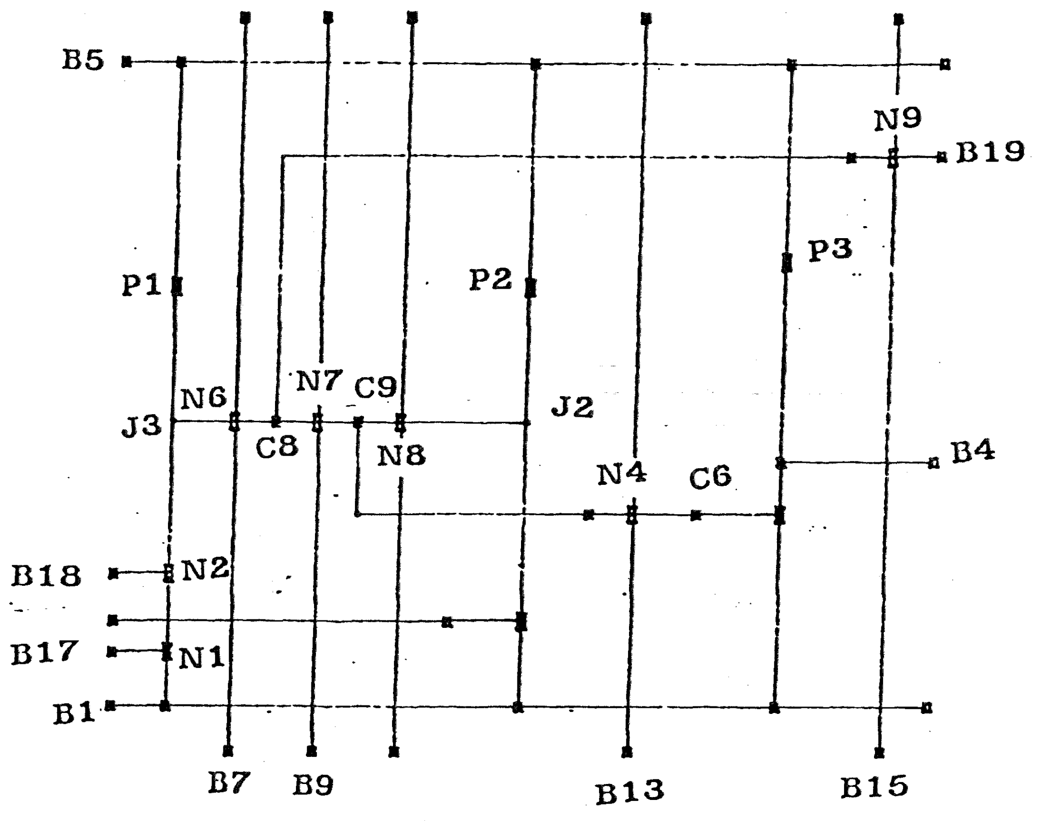


Fig.4.4.3 An NMOS Circuit Illustrating the Algorithm of Transforming Transistor Listings to Akers' Diagrams

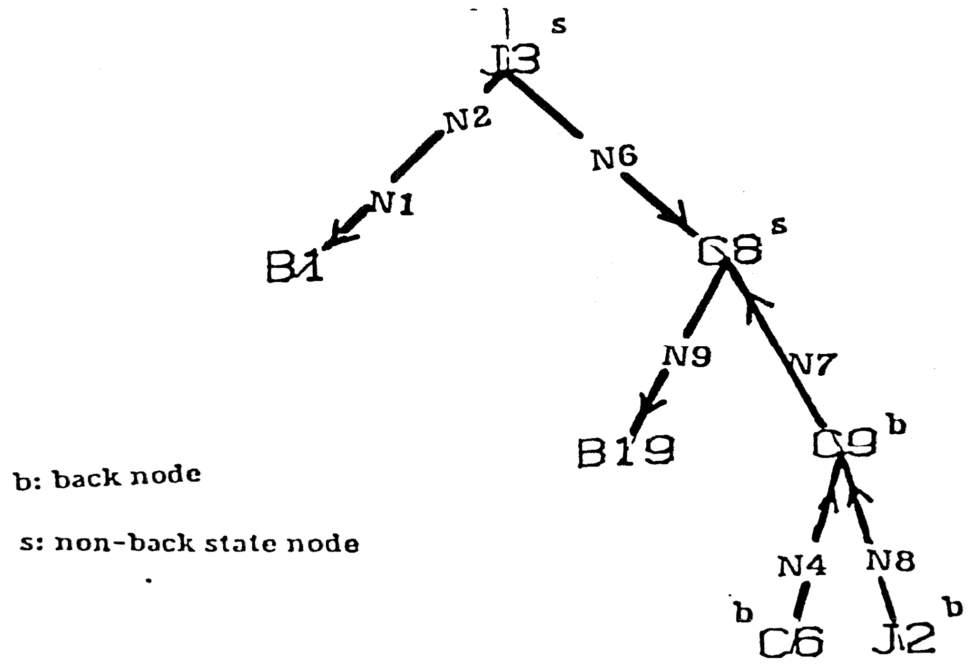


Fig.4.4.4 The Path-tree Created for a Restoring Node J3

The restoring nodes J2 and B4 can also be dealt with in the same way. Later on, one will find that node C6 is a bus variable.

The NMOS shift register described in section 2.3 and 3.3 is applied again. The DBJ notation, after transformation, is

```
DEFI SHIFT
BOOA B5
BOOA B7
BOOA B8
DEFB
BOOA C3
NODE B7=B5?C3:B7
NODE C3=B8?0:1
DEFE SHIFT
DEFI CHIP
BOOA B3
BOOA B5
BOOA B6
BOOA B8
BOOA B12
BOOA B14
DEFB
BOOA I3
BOOA I6
BOOA I9
BLDB SHIFT
PARM B3
PARM I3
PARM B5
BLDE SHIFT
BLDB SHIFT
PARM B6
PARM I6
PARM I3
BLDE SHIFT
BLDB SHIFT
PARM B8
PARM I9
PARM I6
BLDE SHIFT
BLDB SHIFT                                   (4.4.1)
PARM B12
PARM B14
PARM I9
BLDE SHIFT
```

The cell structure is still preserved. Comparing (4.4.1) with (4.1.4-5), it can be seen that the physical world and the logical world have already met each other.

4.5 Remarks

There are quite a few approaches to VLSI design. Some of them are simply based upon engineering discipline; others do have theoretical background. However, none of them seems able to provide a conceptual model that can be applied uniformly throughout the entire design process. As the system grows larger and larger, the need for such a model will become more and more urgent.

As shown in the previous sections, Akers' Diagrams may either be compiled from the hardware design languages, or constructed from the physical layouts. These levels span almost the entire spectrum. Conceptually, the Akers' Diagram is just a binary decision scheme, as fundamental and general as other computing models. However, when applied to all practical design tasks, it always fits the specific problem very nicely. There is a deep feeling that this approach will lead to a homogeneous set of design tools.

5. D Simulator

5.1 General Descriptions

With Akers' Diagrams, it is possible to simulate the logical behavior of digital systems. A simulator, called D simulator, has been constructed. It accepts DBJ notations, that are either compiled from D programs or constructed from CIF, sticks standards or transistor listings. As described earlier, the DBJ notation is a two-level description. Described in the leaf cell are a series of variable definitions and a collection of Akers' Diagrams. Variables may be classified into three classes. _Primary variables_ are those without any Akers' Diagram, _function variables_ with only one, and _bus variables_ with more than one. Variables may be further distinguished into external connectors and internal states. Primary variables must also be external connectors, and their values can only be changed from the outside. Bus or function variables may either be external or internal. Their values are determined by the Akers' Diagrams. The composition cell describes the subcells and the mapping of parameters. The D simulator is based upon these two-level data structures.

The D simulator is a logic simulator, and no circuit information could be investigated. There are four different logic values: _1_, _0_, _X_, _Z_. 1 and 0 simply stand for logic one and logic zero, while X stands for undefined and Z stands for a conflict condition. Originally, all variables are set to X. When the simulation begins, the user is allowed to change the values of the primary variables, and the values of the function and bus variables may be determined accordingly. The simulation can be done either at the leaf cell or at the composition cell level. Only those variables declared at the selected level are accessible to the user. All operations proceed interactively.

For DBJ notation constructed from CIF or sticks standard, the sticks diagrams are available. In this case, the simulation can also be done graphically. By moving the cursor to the position of a primary variable, one can change its value by pressing appropriate buttons on the "mouse". Different buttons represent different input values. Pressing one special button starts the simulation, and the calculated values will be displayed at the corresponding position in the sticks diagram. This kind of simulation is very appealing.

5.2 The Algorithms

Simulation is done on the cell basis. Each cell contains its local variables and subcells. the variables may be primaries, functions or buses. The subcells, defined elsewhere, maintain the mapping of parameters. Those variables whose values may be affected by certain subcells will hold pointers pointing to the affecting subcells. It is assumed, for the purpose of explanation, that cell A contains a subcell B, that is an instance of cell C. V1, a variable declared in cell A, is in the parameter list of subcell B and corresponds to the variable v2 declared and defined in cell C. Suppose that v1 is a primary variable in cell A (it contains no Akers' Diagram), then its value can only be changed from the outside. This change may come from the user (off-chip) or from the subcells. In this example, the value of v1 may be affected by subcell B if v2 is a function or bus variable in cell C. In fact, variable v1 is a function variable, only that its Akers' Diagram is implied in subcell B. To maintain this relationship, v1 is associated with a pointer that points to subcell B. In order to obtain the value of v1, subcell B must be simulated first. Each pointer is associated with a logic value. After subcell B is simulated, the value of v1 is passed back and recorded here. This pointer is also necessary even if variable v1 is a function or bus variable in cell A. Variables may have some of their Akers' Diagrams defined internally while others defined externally. The data structure of the whole system is shown in Fig.5.2.1

The system is composed of a series of cells. Each cell possesses a number of variables and subcells. Each variable contains a value and various number of Akers' Diagrams and pointers. The Akers' Diagrams refer to the variables of the cell, while pointers refer to the subcells. Those variables with neither pointers nor Akers' Diagrams are the real primary variables. Other variables are function variables. Functions and buses are no longer distinguished since their simulation methods are basically the same. The variables may either be external or internal. They are ordered in the same sequence as appeared in the DBJ notation. External variables come first and then internal ones. Each subcell holds a pointer and a list of parameters. The pointer refers to the cell where the subcell is defined. The length of the parameter list must be the same as the number of variables in the pointed cell. They must be in the correct order, also.

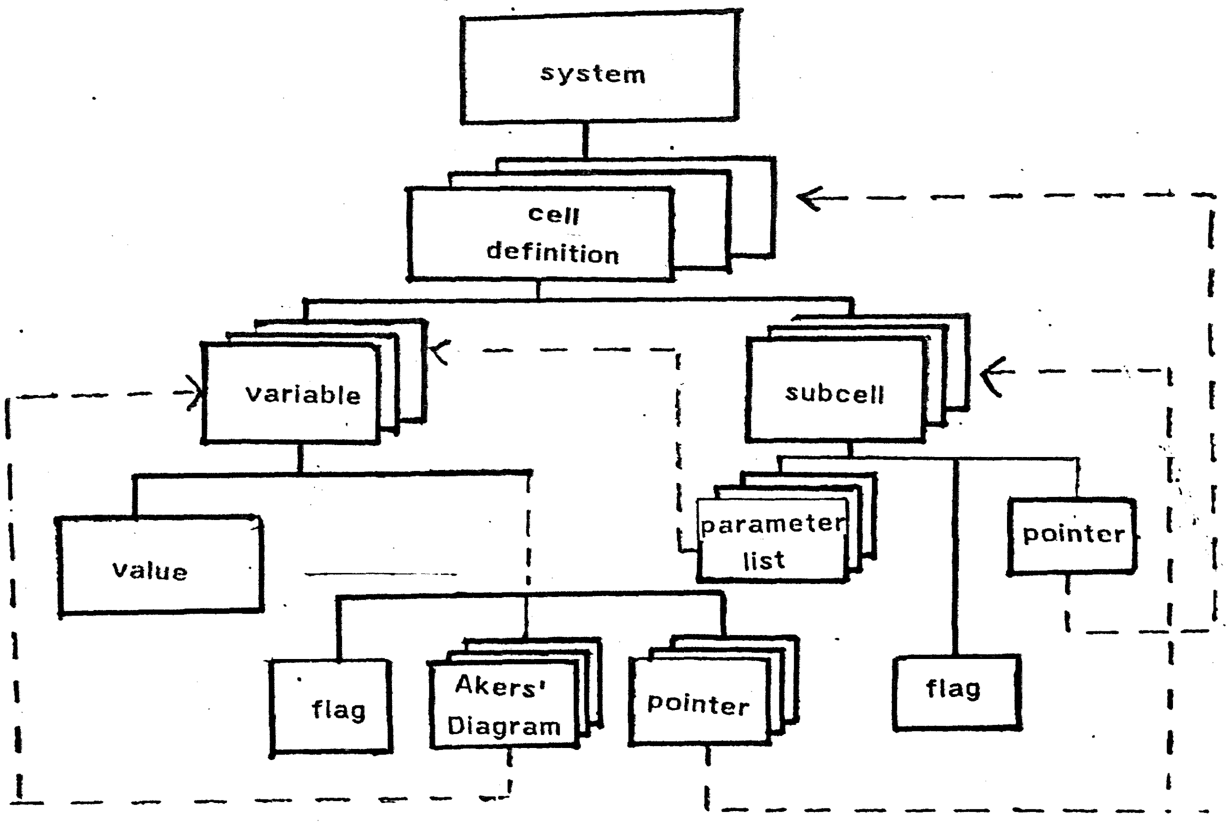


Fig.5.2.1 The Data Structure for the D Simulator

Each function variable or subcell is associated with a flag. Before each simulation step, all flags are reset to zero. The value of each variable is then calculated one by one. For primary variables, no action is taken. For function variables, the values of their Akers' Diagrams and pointers are calculated first. To calculate the value of an Akers' Diagram, one starts with the variable in the top node. If its value has already been calculated (it is a primary variable or its flag has been set to one), then this value is read in. Otherwise, it has to be calculated first. Once this value is available, proper action is taken accordingly: 1) If it is logic 1, then the calculation is passed to the one-subdiagram (the right branch of the Akers' Diagram). This process is repeated until the bottom node is reached. The value is calculated and returned. 2) If it is logic 0, then the calculation is passed to the zero-subdiagram (the left branch). This process is again repeated recursively. 3) If it is logic X, then both subdiagrams need to be evaluated. If the returned values are the same, then this value is returned; otherwise value X is returned instead. 4) Finally, if it is logic Z, then logic X is returned. To calculate the value of each pointer, the first thing is to determine the pointed subcell has already been simulated. If it has, then no action is taken, otherwise that subcell is simulated first. The simulation of subcells is quite similar to the procedure call in programming languages. Values and flags are saved for the called cell, the values of the parameter list are passed from the main cell, the simulation is done in the called cell, the calculated values are passed back, and the original values and flags are restored. Upon returning, the flag of the subcell is set to one.

After all the values of the Akers' Diagrams and the pointers are available, the value of the function variable can be decided accordingly. Finally, the value is recorded, and the flag is set to one. Under this mechanism, the calculation may be activated either by the system or by another variable. Although the original sequence of the variables may not be correct, their values are always calculated in the correct order. No explicit sorting is necessary. Loops may be detected sometimes if there are state variables contained in the circuit. In such cases, this variable is not recalculated (otherwise it will enter an infinite loop), and the original value is returned instead. The SIMULA code for the data structure and the simulation steps of the system are listed in Appendix 2.

5.3 Examples

The 4-bit adder (4.1.3-4) and the 2-bit NMOS shift register (4.4.1) are illustrated again. Although the DBJ notations are from different sources (one from the D program and the other from the CIF), the D simulator treats them in the same way. There are 17 variables in the 4-bit adder. From A0 to Cin are 9 primary variables. Suppose that their present values are : A = [A0, A1, A2, A3] = [0 0 1 0], B = [B0, B1, B2, B3] = [0 1 1 0], and Cin = 1. When the simulation starts, the value of the first function variable Y0 is calculated. It contains no Akers' Diagram so no local calculation is needed. However, it contains a pointer referring to subcell 1. In order to obtain the value of Y0, this subcell has to be simulated first. The values of A0, B0, Cin, Y0 and Int0 are passed to cell adder1. Upon returning, Y0 = A0↑B0↑ Cin = 1 and Int0 = majority(A0, B0, Cin) = 0. The flag of subcell 1 is set to 1. When variable Int0 which also points to this subcell is encountered later, no simulation is needed any more. There are 8 function variables in the cell adder4. Simulation follows the sequence as shown below:

	Y0	Y1	Y2	Y3	Cout	Int0	Int1	Int2	activated
step	value returned								subcell
1	1					0			1
2		1					1		2
3			0					1	3
4				1	0				4
5-8									None

This example may be oversimplified since neither loop condition nor cell nesting happens during simulation. To illustrate this capability, the shift register is discussed. For convenience, let vector IN be [B3, B6, B8, B12, B5]. There are four function variables: B14, I3, I6 and I9. Suppose the initial value of IN is [1 1 1 1 0], then the simulation will follow the sequence as shown below:

	B14	I3	I6	I9	activated	
step	value returned				subcell	
1					4	(value of I9 not available)
2					3	" I6 "
3					2	" I3 "
4		1			1	(The input value is shifted in and inverted. Subcell 1 is completed)
5			0			(subcell 2 completed)
6				1		" 3 "
7	0					" 4 "

All pass transistors are turned on, and the value of B14 can not be determined before all the subcells are simulated. In this case, the cell SHIFT will be nested four steps deep as shown above. At the next step, IN is changed to [1 0 1 0 1]. The simulation proceeds as follows :

step	B14	I3	I6	I9	activated subcell	
	value returned					
1	0				4 *	
2					3	(I6 not available)
3			0		2 *	
4				1		(subcell 3 completed)
5		0			1	

* Loop detected

Two pass transistors are turned off, and the corresponding state variables will remain their original values. This is revealed by the statement B7=B3?C3:B7 in the DBJ notation (4.4.1). Loops are detected in this case. Also note that function C3 no longer needs to be evaluated at all.

5.4 Remarks

In the transistor-to-DBJ transformation, the Akers' Diagrams constructed from the N network are the same as those constructed from the P network. Both P-type and N-type transistors are treated as ideal switches. However, they are not the same even under first order approximation. P-type transistors are only effective in transmitting 1's, while N-type transistors are only effective in transmitting 0's. The two branches in the Akers' Diagrams are not totally symmetric. At the present time, all the Akers' Diagrams are technology-independent. However, if one desires, they may be associated with technology flags, and the simulation scheme may be modified to highlight the different behaviors between different technologies.

6. Conclusions

In this thesis, the five intermediate forms shown in Fig.1.1. are briefly discussed. Programs to transform from the CIF to the sticks standard, the sticks standard to the transistor listing, and the transistor listing to the Akers' Diagram are discussed. One important feature of all these transformations is that only the description of leaf cells are changed, whereas the composition cell (cell structure) always remains unaffected. Finally, the general operation of a D simulator is presented. The model itself is conceptually simple, however its application spans the entire design spectrum.

As pointed out previously, the system only deals with functional behavior. However, the capability for analyzing physical properties can be included easily. Currently, the system is under development and still needs a lot of testing. With all the experience gained so far, it is about time to start the logic-to-geometry transformations.

REFERENCES

[1] Tarolli, Gary " Towards a working VLSI CAD Tool : A Chip Assembler ", SSP (Silicon Structure Project) File #3131, Caltech, October 1979.

[2] Rowson, Jim & Trimberger, Stephen " Riot -- A Stupid Graphical Composition Tool ", SSP File #4142, Caltech, January 1981.

[3] Johannsen, Dave, " Bristle Blocks : A Silicon Compiler ", Proc. 16th Design Automation Conference, 1979.

[4] Johannsen, Dave, " Building a Chip with Bristle Blocks ", SSP File #3519, Caltech, Febeuary 1980.

[5] Lang, Richard " The Caltech Intermidiate Form for LSI Layout ", SSP File #1120, Caltech, Febrary 1978.

[6] Mead, Carver M. & Conway, Lynn " Introduction to VLSI Systems ", Addison Wesley, 1980.

[7] Trimberger, Stephen " The Proposed Sticks Standard ", Technical Roport #3880, Caltech, August 1980.

[8] Locanthi, Bart " LAP : A SIMULA Package for IC Layout ", Display File #1862, Caltech, July 1978.

[9] Lang, Richard " LAP User's Mannual ", Technical Report #3356 , Caltech, December 1979.

[10] Roberto Suaya & Yagil Hertzberg " ICLIC Mannual - Version 0 ", Caltech, January 1981.

[11] Trimberger, Stephen " Nick - FORTRAN Layout Language Package ", SSP File #3486, Caltech, July 1980.

[12] Mosteller, Ricky C. " Rest User's Guide ", SSP File #4030, Caltech, October 1980.

[13] Trimberger, Stephen " Sticks Standard to CIF Translator ", SSP File #4042, Caltech, December 1980.

[14] Bryant, Randal E. " An Algorithm for MOS Logic Simulation ", Lambda Magzine, Fourth Quarter 1980.

[15] Bryant, Rendal E. " MOSSIM : A Logic - Level Simulator for MOS LSI ", User's Manual, VLSI Memo No. 80-21, M.I.T. , July 1980.

[16] Rowson, Jim & Trimberger, Stephen " STKPLT and STKSIM " SSP File #4025 ,Caltech, October 1980.

[17] Akers, Sheldon B. " On the Specification and Analysis of Large Digital Functions ", Proc. 7th International Symposium on Fault Tolerent Computing, pp.88-93, June 1977.

[18] Rupp, Charle' R. " Akers' Diagrams ", Caltech, Fall 1980.

[19] Rupp, Charle' R. " Application of Akers' Diagrams in VLSI Design", SSP Symposium, Caltech, Fall 1980.

[20] William, John " Akers' Diagrams to Layouts ", SSP Symposium, Caltech, Fall 1980.

[21] Baker, Clark M. & Terman, Chris " Tools for Verifying Integrated Circuit Designs", Lambda Magzine, Fourth Quarter, 1980.

[22] Rupp, Charle' R. " D Reference Mannual ", Caltech, Fall 1980.

[23] Kernighan, Brian W. & Ritchie, Dennis, M. " The C Programming Language ", Prentice-Hall, 1978.

[24] Birtwistle, G. M. , Dahl, O-J, Myhrhaug, B. & Nygaard, K. " SIMULA Begin ", Petrocelli, New York, 1973.

[25] Lang, Richard & Torolli, Gary " CIF20P Instructon Mannual ", Display File #3357, Caltech, December 1979.

[26] Rowson, Jim A. " Understanding Hierarchical Design ", Ph.D thesis, Technical Report #3710, Caltech, April 1980.

[27] Newman , William M. & Sproull, Robert F. " Principles of Interactive Computer Graphics ", McGraw-Hill CSS , 1979.

[28] Trimberger, Stephen " Recognizing Circuits and Parsing ", Display File #3471, Caltech, Febuary 1980.

[29] Wipfli, John " A SIMULA Graphic Package ", SSP File #1929 Caltech, October 1978.

[30] Burke, W. T. " Graphic Workstation Protocol, Version 4.0 ", SSP File #3489, Caltech, April 1980.

[31] " GIGI/ReGIS Handbook ", Digital Equipment Corp. , February, 1981.

[32] " HP 7221A Graphic Plotter : Operating and Programming Mannual ", Hewlett-Packard Company, November 1977.

[33] Barton, Eric E. " The Polygon Package ", SSP File #3129, Caltech, November 1979.

[34] Hennie, Fred " Introduction to Computability ", Addison-Wesley Publishing , 1977.

[35] Savage, J. E. " The Complexity of Computing ", Wiley-Interscience, 1978

[36] Engeler, E. " Introduction to the Theory of Computation ", Academic Press, 1973.

[37] Burge, W. H. " Recursive Programming Techniques ", Addison-Wesley Publishing, 1975.

Appendix 1: Users' Guide for D Analyzer

All the ideas in this thesis are implemented in a SIMULA program called DAN, which stands for D analyzer. It is now running at DEC-20 under the Tops-20 operating system. All operations proceed interactively. The prompt for the user command is ">". At this point, the user can enter any DAN command. Here is the summary of all the commands avaiable. Some of them are valid only when one is using a graphic terminal.

1.File Commands:

CIF : reads in a CIF file , detects all the components, and put the sticks standard in the file with extension .STK.

STK : reads in sticks standard. The sticks standard may either be constructed by this program or come from other sources.

LOAD : reads in a DBJ file, and create the data structure shown in Fig.5.2.1. This DBJ file may either be compiled from D programs or be constructed from CIF, sticks standard, or transistor listing.

TYPE : prints out the contents of a file.

2.Transformation Commands:

TRN: converts the sticks standard to the transistor listing. The data structure shown in Fig.3.1.3-4 is created and the listing is put to the file with extension .TRN.

DBJ: converts the transistor listing to DBJ and saves the code in the file with extension .DBJ. If the cell is still at the stick level, command TRN is automatically taken before command DBJ is executed.

* CIF to sticks covertion is done by the CIF command.

3.Simulation Commands:

SIMULATE: starts the D simulator. This command only works when running on a graphic device. The default device is the GIGI terminal, however, you can use the command DEVICE to specify another. The sticks diagram of the selected cell is displayed. All variables start with value 'X'. The primary variables are indicated by small yellow boxes and are the only places one can set their values. Setting values is by way of mouse buttons, button 1 enters '0', button 2 enters '1'. Button 3 starts the simulation, and the final values are displayed at the corresponding positions in the sticks diagram. To leave the simulator, press buttons 1 and 2 at the same time. This simulation mode is valid only for those DBJ structures that are created from the CIF and sticks files. For those cells without any topological information, SETV and RUN commands are used instead.

SETV: changes the values of the primary variables in the current cell. Adjacent variables in the variable list may be changed at the same time.

RUN: starts simulation. The result is displayed in text form. Only one step simulation is allowed so far. One uses the SETV command to set the values of primary variables, and uses RUN command to start calculation.

SELECT: select the cell intended for simulation.

4.Inquiry Commands:

PLOT: plots the sticks diagram of the desired cell.

CELL: displays all the topological information of each cell. This includes the number of each type of component and the number of subcells.

DISPLAY: displays the DBJ information of the selected cell. This includes the names of all the variables and subcells. For each variable, the number of Akers' Diagrams and the names of the pointed subcells are also indicated.

SHOW: shows the current value of each variable in the selected cell.

AKERS: plots the Akers' Diagrams of the desired variable. If the number of Akers' Diagrams are more than one, they will be diaplayed one by one. This commands is valid only when one works with a graphic device.

5.Graphic Device Control Commands :

DEVICE: changes the device. Only GIGI terminal, Charles terminal and HP plotter are allowed.

LEVEL: changes the plot level of sticks diagrams.

TEXT: turns on the text mode or turns it off.

WINDOW: zooms in.

CURSOR: returns the user coordinate of the selected point.

QUIT: leaves the system.

All the commands may either be upcased or be lowercased. Abbreviated input is also allowed.

Here is a simple example :

```
@dan
>cif shift

Read File SHIFT.CIF
Warn: NMOS Technology
2 Cells Entered
Make Sticks Standard
Saved in SHIFT.STK
```

```
>type shift.cif

   ( Fig.2.3.7 )

>type shift.stk

   ( Fig.2.3.8 )

>cell

    CELL    NAME     NTRN  PRES  CONT  CONN  INTER   SUBCELL
       1    SHIFT       2     1     4     8
       2    CHIP                           14      9         4

>plot shift

   ( Fig.2.3.4 )

>text off
>plot 2

   ( Fig.2.3.6 )

>text on
>level 1
>plot chip

   ( Fig.2.3.5 )

>trn

Make Transistor Listing
Saved in SHIFT.TRN

>type shift.trn

   ( Fig.3.1.3 )
   ( Fig.3.1.4 )
```

```
>dbj

Make DBJ File
Saved in SHIFT.DBJ

Cell SHIFT
B1? /H/L/I/O/ or type 'Y' if need graphic aide>y

   ( Fig.3.1.1 )

B1? /H/L/I/O>O
Warn:B2 is assumed to connect to Vdd.
B7? /H/L/I/O/ or type 'Y' if need graphic aide>o

Cell CHIP

2 Cells Entered

  1. SHIFT
* 2. CHIP           <--- current cell

>type shift.dbj

   ( 4.4.3 )

>select 1

Current Cell : SHIFT

   Variable       Type   Akers  Pointer
 1.    B5         PRIM
 2.    B7         FUNC      1
 3.    B8         PRIM
 4.    C3         FUNC      1

>akers b7
```

```
   ( Akers' Diagram of b7 )

>akers 2

   ( the same diagram )

>simulate

   ( Fig.2.3.4 with X at B5,B7,B8 and C3, 1 at B2, and 0 at B1 )

             .
             .
             .
             .
             .

>load adder

Read File ADDER.DBJ
2 Cells Entered

  1. ADDER1
* 2. ADDER4

>display

Current cell : ADDER4

    Variable       Type    Akers  Pointer          Subcell
  1.    A0         PRIM                         1.  ADDER1
  2.    A1         PRIM                         2.  ADDER1
  3.    A2         PRIM                         3.  ADDER1
  4.    A3         PRIM                         4.  ADDER1
  5.    B0         PRIM
  6.    B1         PRIM
  7.    B2         PRIM
  8.    B3         PRIM
  9.    CIN        PRIM
 10.    Y0         FUNC            1
 11.    Y1         FUNC            2
```

```
12.    Y2          FUNC          3
13.    Y3          FUNC          4
14.    COUT        FUNC          4
15.    INT0        FUNC          1
16.    INT1        FUNC          2
17.    INT2        FUNC          3

>simulate
Sorry, No Sticks Diagram.  Try SETV and RUN.
>setv 1=001001101
>show

   Variable       Type     Value
  1.   A0         PRIM       0
  2.   A1         PRIM       0
  3.   A2         PRIM       1
  4.   A3         PRIM       0
  5.   B0         PRIM       0
  6.   B1         PRIM       1
  7.   B2         PRIM       1
  8.   B3         PRIM       0
  9.   CIN        PRIM       1
 10.   Y0         FUNC       X
 11.   Y1         FUNC       X
 12.   Y0         FUNC       X
 13.   Y0         FUNC       X
 14.   COUT       FUNC       X
 15.   INT0       FUNC       X
 16.   INT1       FUNC       X
 17.   INT2       FUNC       X

>run

   Variable       Type     Value
  1.   A0         PRIM       0
  2.   A1         PRIM       0
  3.   A2         PRIM       1
  4.   A3         PRIM       0
  5.   B0         PRIM       0
  6.   B1         PRIM       1
 7.    B2         PRIM      1
 8.    B3         PRIM      0
 9.    CIN        PRIM      1
10.    Y0         FUNC      1
11.    Y1         FUNC      0
12.    Y0         FUNC      0
13.    Y0         FUNC      1
14.    COUT       FUNC      0
15.    INT0       FUNC      0
16.    INT1       FUNC      1
17.    INT2       FUNC      1


>quit

@
```

Appendix 2: SIMULA Code for the D Simulator

```
! part of the program are expressed in pidgen codes ;

    thing CLASS cell;
    BEGIN
        REF(vector)variables,subcells;

        PROCEDURE calculate;
        BEGIN
            reset the flags of function variables and subcells to 0;
            INSPECT variables DO FOR i:=1 STEP 1 UNTIL length DO
            a.a[i] QUA variable.calculate;
        END;

    END of CLASS cell;

    thing CLASS variable;
    VIRTUAL: CHARACTER PROCEDURE calculate;
    BEGIN
        CHARACTER logic-value;
    END of class variable;

    variable CLASS primary;
    BEGIN

        CHARACTER PROCEDURE calculate;
        calculate:=logic-value;

    END of CLASS primary variable;

    variable CLASS function;
    BEGIN
        REF(vector)dependents;  INTEGER flag;

        ! intermediate variable ;
        CHARACTER old-value,temp;
    CHARACTER PROCEDURE calculate;
    IF flag = 1 THEN calculate := logic-value
    ELSE IF flag = -1 THEN BEGIN
        report.warn("Loop Detected");
        calculate := old-value;
    END ELSE BEGIN

        ! initialize ;
        old-value :=
        IF NOT logic-value = 'Z' THEN logic-value ELSE 'X';
        logic-value := 'X';
        flag:=-1;

        ! start calculation;
        INSPECT dependents DO
        FOR i := 1 STEP 1 UNTIL length DO BEGIN
            IF logic-value \= 'Z' THEN BEGIN
               INSPECT a.a[i]
               WHEN Akers'-Diagram DO temp := calculate
               WHEN pointer DO temp := calculate;
               logic-value := IF logic-value = 'X' THEN temp
               ELSE IF temp = 'X' THEN logic-value
               ELSE IF temp = logic-value THEN logic-value
               ELSE 'Z';
            END;
        END;
        flag := 1;                                          ( 5.2.1 )
        calculate := logic-value;
    END of calculate;

END of CLASS function variable;

thing CLASS Akers'-Diagram;
BEGIN
    REF(node)condition,left,right;
    INTEGER pointer;  BOOLEAN negative;

    CHARACTER PROCEDURE calculate;
    BEGIN
```

```
            IF pointer = -1 THEN calculate := '1'
            ELSE IF pointer = -2 THEN calculate := '0'
            ELSE IF pointer = -3 THEN calculate := 'X'
            ELSE BEGIN
               CHARACTER ch1;
               IF pointer > 0 THEN
               ch1 :- current-cell.variables.a.a[pointer]
                 QUA variable.calculate;
               ELSE BEGIN
                   CHARACTER ch2;
                   ch2 := condition.calculate;
                   IF ch2 = '1' THEN ch1 := left.calculate
                   ELSE IF ch2 = '0' THEN ch1 := right.calculate
                   ELSE IF ch2 = 'X' THEN BEGIN
                      CHARACTER ch3;
                      ch3 := left.calculate;
                      ch1 := IF ch3 = right.calculate THEN ch3 ELSE 'X';
                   END ELSE calculate := 'X';
               END;
               calculate := IF NOT negative THEN ch1
               ELSE IF ch1 = '1' THEN '0'
               ELSE IF ch1 = '0' THEN '1'
               ELSE ch1;
            END;
        END of calculate;

    END of CLASS Akers'-Diagram;

    thing CLASS pointer;
    BEGIN
        REF(subcell)pointed-subcell;
        CHARACTER value;

        CHARACTER PROCEDURE calculate;
        BEGIN
           pointed-subcell.calculate;
           calculate := value;
        END of calculate;
```

```
END of CLASS pointer;

thing CLASS subcell;
BEGIN
    REF(cell)pointed-cell;  REF(vector)parameter-list;
    BOOLEAN flag;

    CHARACTER PROCEDURE calculate;
    IF flag = 0 THEN BEGIN
        store the flag and value in the pointed-cell;
        pass parameter value to the pointed-cell;
        pointed-cell.calculate;
        pass parameter value back;
        flag := 1;
    END of calculate;

END of class subcell;
```