AD-A040 684 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
A DESIGN LANGUAGE FOR MODULAR ASYNCHRONOUS CONTROL STRUCTURES.(U)
FEB 77 T MUDGE, G METZE DAAB07-72-C-0259
UNCLASSIFIED R-759 NL

1 OF 1
AD
A040684

END
DATE
FILMED
7-77

REPORT R-759 FEBRUARY, 1977 UILU-ENG 77-2206

CSL COORDINATED SCIENCE LABORATORY

A DESIGN LANGUAGE FOR MODULAR ASYNCHRONOUS CONTROL STRUCTURES

TREVOR MUDGE
GERNOT METZE

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) A DESIGN LANGUAGE FOR MODULAR ASYNCHRONOUS CONTROL STRUCTURES.	5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report,
	6. PERFORMING ORG. REPORT NUMBER (14) R-759, UILU-ENG-77-2206
7. AUTHOR(s) (10) Trevor Mudge and Gernot Metze	8. CONTRACT OR GRANT NUMBER(s) (15) DAAB07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 66p.
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program	12. REPORT DATE (11) Feb[redacted]977 13. NUMBER OF PAGES 60
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Asynchronous Digital Systems
Design Language
Asynchronous Logic Modules

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A design language for asynchronous digital systems is presented. The language transaltes onto a set of asynchronous logic modules, to form the control structure of a digital system. The behavior of these modules is discussed. The process of translating the design language into networks of these modules is outlined. An example design is presented. The design language is shown to be able to describe machines having all the capabilities normally required of a microcontroller, as well as the capability to describe the control of parallel processes without necessarily binding processes to

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

097 700

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

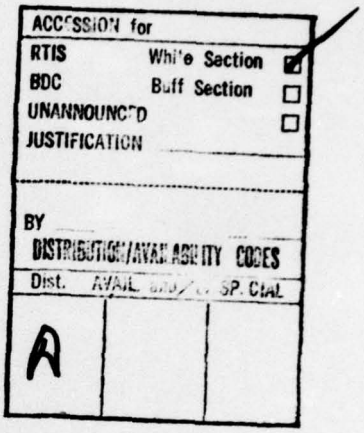
20. ABSTRACT (continued)

one another. Parallel processes imply the possibility of designing a system having the potential for deadlock. It is shown, in a quantitative way, that it is easier to check a design described by the design language for correct syntax and absence of deadlock, than to check an arbitrary network of modules for absence of deadlock. It is inferred from this, that fault-free design is easier using the design language.

A DESIGN LANGUAGE FOR MODULAR ASYNCHRONOUS CONTROL STRUCTURES

by

Trevor Mudge and Gernot Metze



This work was supported by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

A DESIGN LANGUAGE FOR MODULAR ASYNCHRONOUS CONTROL STRUCTURES

by

Trevor Mudge and Gernot Metze
Coordinated Science Laboratory
University of Illinios at Urbana, September 1976

ABSTRACT

A design language for asynchronous digital systems is presented. The language translates onto a set of asynchronous logic modules, to form the control structure of a digital system. The behavior of these modules is discussed. The process of translating the design language into networks of these modules is outlined. An example design is presented. The design language is shown to be able to describe machines having all the capabilities normally required of a microcontroller, as well as the capability to describe the control of parallel processes without unnecessarily binding processes to one another. Parallel processes imply the possibility of designing a system having the potential for deadlock. It is shown, in a quantitative way, that it is easier to check a design described by the design language for correct syntax and absence of deadlock, than to check an arbitary network of modules for absence of deadlock. It is inferred from this, that fault-free design is easier using the design language.

TABLE OF CONTENTS

1. INTRODUCTION

In an attempt to formalize the design process for large digital systems, many researchers have suggested the use of design languages. [Bar, Chu, Com, Dul, Fri, Gla, Met]. However, using a design language (DL) does not necessarily facilitate the design process. An ill conceived language can encumber the design process and fail to guide it away from design faults.

The purpose of this report is twofold: first, to develop a DL for large asynchronous digital systems - this allows large systems to be designed which utilize their hardware more efficiently than the conventional synchronous systems; second, to specify the DL so that design faults can be easily avoided - this is especially important when designing asynchronous as opposed to synchronous machines, because of the greater number of pitfalls associated with the asynchronous design process.

The DL is capable of being used to design asynchronous digital systems which conform to the following model: the system partitions into a control structure (CS) and a data structure (DS). Actions in the DS are assumed to be representable as register transfers. The coordination of these actions is accomplished by the CS. The register transfers themselves are initiated by request (R) signals, which issue from the CS and travel over bidirectional signal paths called links to the DS. Upon their completion acknowledge (A) signals are transmitted back along the links to the CS. (See figure 1.1.)

The DL translates onto a set of eight asynchronous modules to produce a network of modules which forms the CS of the target system. Inter-module communication also occurs over links.

By constructing the CSs out of a small set of modules that communicate over links many design pitfalls are avoided. Since the modules are simple and their design is a one-time affair, classical asychronous design methods may be used in their design [Ung]. Once they have been designed large systems can be constructed from them, without concern for races, hazards, fan-out constraints, etc., by virtue of the rigid inter-module communication protocol. Furthermore, since networks of these modules are speed-independent [1] and delay-insensitive [2], variations in the response time of the modules and in the delays

---

1. A network of modules is speed-independent if its external behavior is independent of the delay of the constituent modules.
2. A network of modules is delay-insensitive if its external

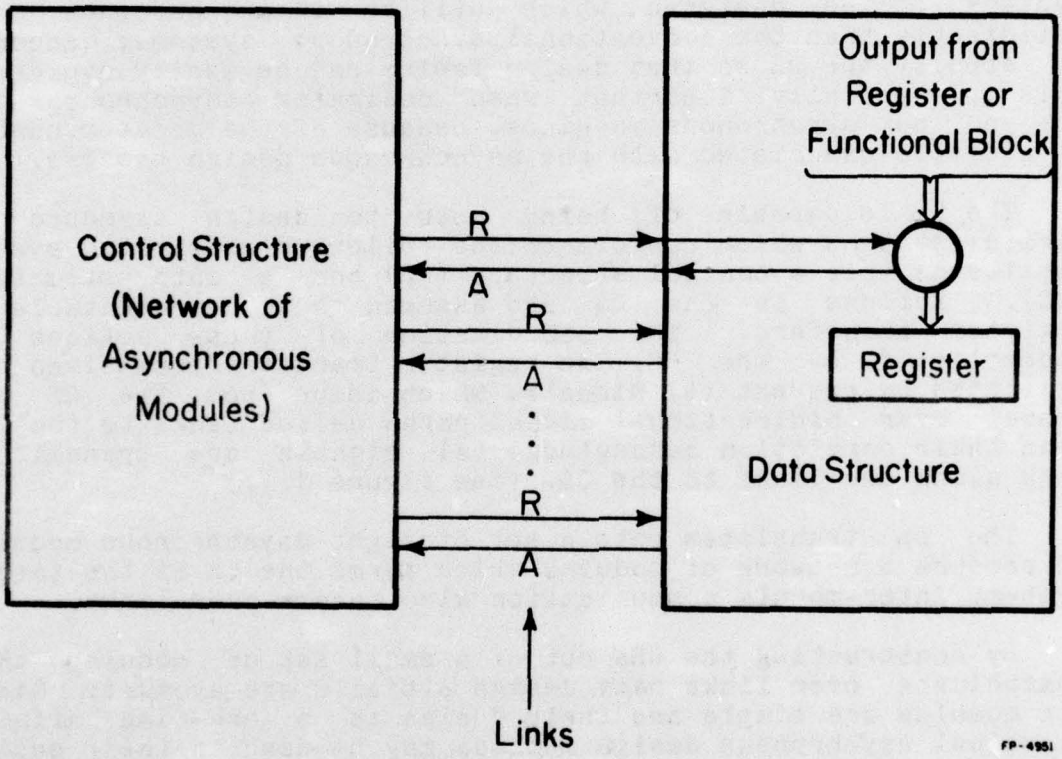


Figure 1.1. The System Model.

between modules can be ignored . (They need only be finite, but can be unbounded.) Hence the change in the response time of particular modules due to aging, and the delays between modules due to their spatial distribution, need not be considered. Finally the common critism of modular CSs , that they imply a hardwired CS, does not apply. There exists procedures for translating networks of modules of the type used in this report into programmable diode arrays without losing any of the desirable features outlined above. [Pat 75, Jum]

Although modularization reduces some of the design problems there are still several associated with the interconnection patterns of the modules in CS networks. Certain interconnection patterns result in networks which eventually deadlock or hang-up [1]. (This phenomenon is analagous to deadlock in operating systems. [Hab]) Recognition of this type of design problem is a complex procedure. The DL is specified so that within the class of networks that it describes recognizing potential deadlock is very much simplified. This advantage would be lost if recognizing the class of networks produced by the DL were as complex a procedure as recognizing deadlock in any arbitary network of modules. This is not the case. The advantage would also be lost if the class of networks produced by the DL were not general enough to include types of control structures commonly found in digital systems. Neither is this the case.

The plan of the report is as follows. Section 2 defines the DL's syntax and gives an interpretation of that syntax by using the notion of a process. The syntax is an improvement of that found in [Pet]. Recognition of potential deadlock has been made easier, and so has recognition of the class of networks produced by the DL. This has been achieved without reducing the class of CSs that the DL can describe. Section 3 defines the modules that the DL translates onto. These are the So (source) module, the Si (sink) module, the W (wye) module, the S (sequence) module, the J (junction) module, the SR (shared resource) module, the D (decode) module and the I (iterate) module. They are discussed at length in the literature [Alt, Bru] and have been used in

---

behavior remains unchanged, when any number of delay elements is inserted into, or removed from, any of the two lines in the links.

1. A network of modules which communicate over bidirectional links, as above, deadlocks when at least one of the links in the network reaches a state of having a permanent request signal on it, which is not the result of a circuit failure in a module or link.

paper designs to illustrate their suitability as building blocks in the design of CSs of complex digital systems [Den]. Section 4 describes the translation procedure, which relates the modules and the syntax of the DL. Section 5 shows the design of a simple digital system using the DL. Section 6 examines the scope of the DL and shows it to be capable of describing machines having all of the capabilities normally required of a microcontroller, as well as the capability to describe the control of parallel processes without the unnecessary binding of processes to one another that is demanded by a synchronous environment. Section 7 shows, in a quantitative way, that the computational complexity of an algorithm required to check a design in the DL for correct syntax and absence of deadlock is of a lower degree than that required to check an arbitary network of modules for absence of deadlock. (From sections 6 and 7 the advantages of using a good DL may be inferred.) Finally section 8 gives a brief summary of the design approach developed in this report.

2. A DESIGN LANGUAGE

The notion of a process is useful in discussing the interpretation of the DL, before explicit reference is made to the modules. In this report the term "process" is used in the following way: a process in a digital system is some activity that is initiated by placing a request signal on some link in the CS of the system and that signals its completion by placing an aknowledge signal on the link.

The action of a CS can therefore be viewed as a complex process which decomposes into successively less complex processes until the atomic (indivisible) process, the register transfer, is reached.

This hierachical way of looking at CSs conforms to the block- structured nature of the DL. By using the DL in the design procedure, the CS produced for the target system is a network of modules organized as a hierachy of sub-networks (represented by blocks in the DL) which communicate with one another over single links. Thus, for any two subnetworks joined by a link the "higher" one (the one that puts a request on the link) can view the "lower" one (the one that puts an acknowledge on the link) as a process. Its detailed specification can then be deferred to a convenient point in the design procedure.

The DL makes use of three types of blocks to describe complex processes. These blocks are composed of two types of statements (see the syntax diagram, figure 2.1) .

2.1 The Two Statement Types.

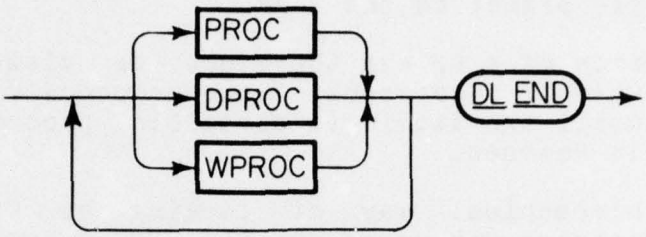
The two types of statements, process call (PROC-CALL) and register transfer (REG-TRF), are illustrated in the block of statements shown in figure 2.2. Their syntax is shown in production 6 of the syntax diagram.

Statements 1 and 3 are REG-TRF type. They represent links out of the CS to register transfer points in the DS, over which requests for and acknowledgments of register transfers pass. In statement 1 of this example the control of the register transfer "move the contents of the register named SOURCE to that named DEST" is indicated. Similarly for statement 3. Both statements are atomic processes of the CS being described.

Statements 2 and 4 are PROC-CALL type. They represent internal links of the CS over which one subnetwork controls

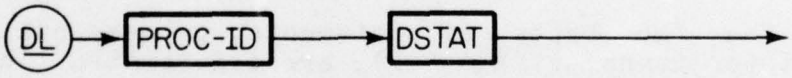
Production No.

1. Program



2. PROC

3. DPROC

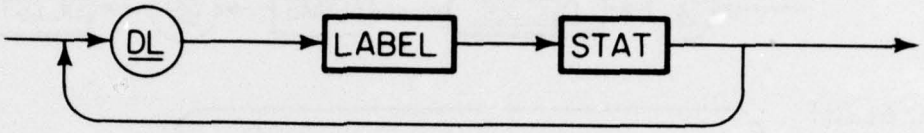


4. WPROC

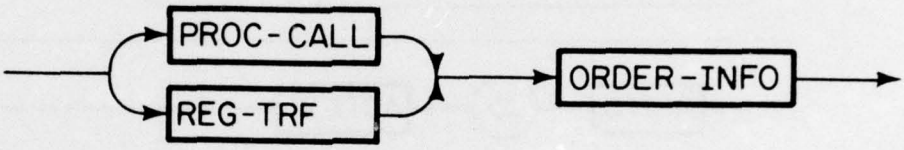
FP-4958

Figure 2.1. The Syntax Diagram (First of six).

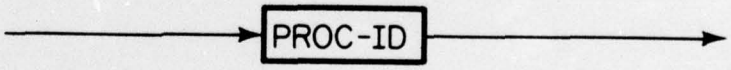
5.STAT-LIST



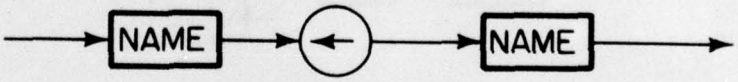
6.STAT



7.PROC-CALL



8.REG-TRF



FP-4959

Figure 2.1. The Syntax Diagram (Second of six).

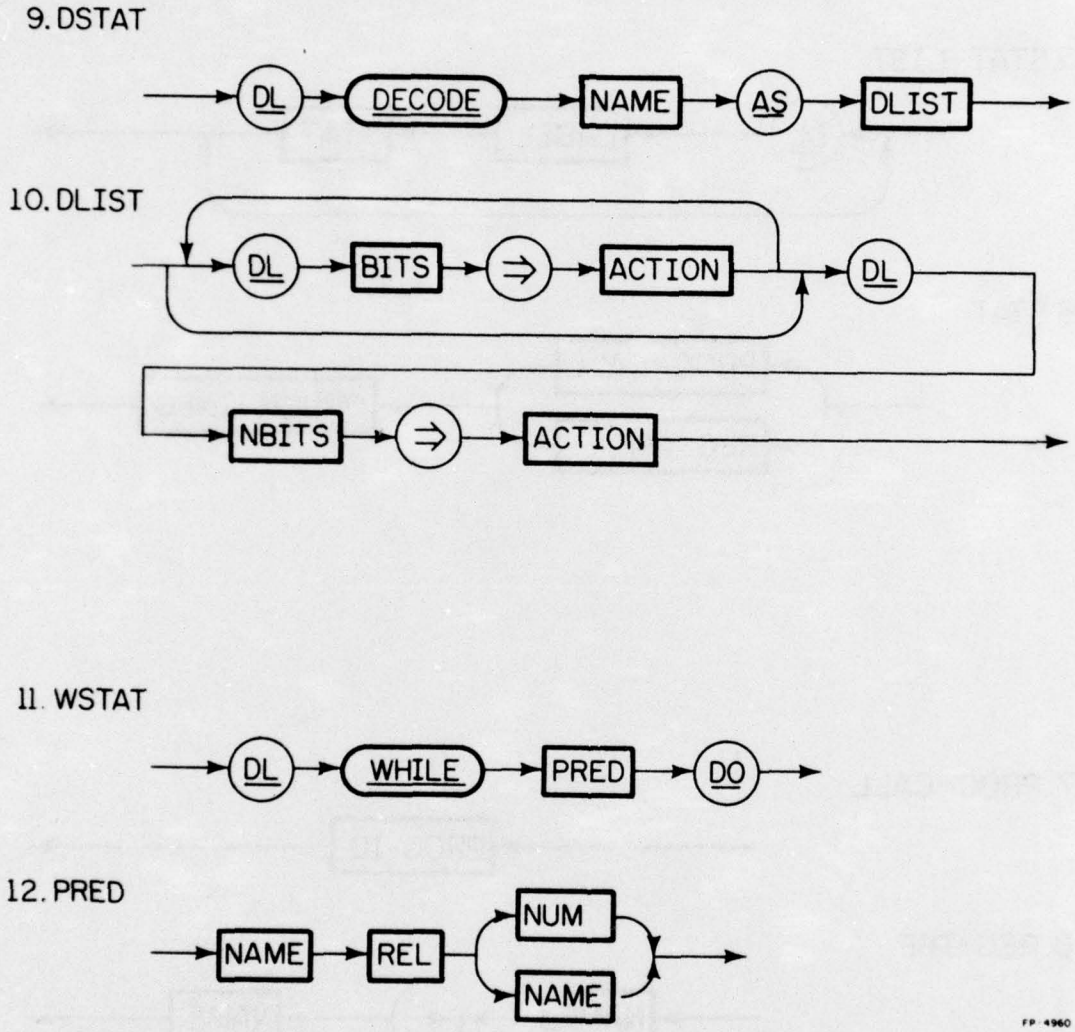
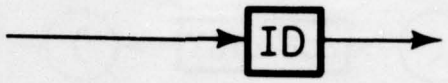


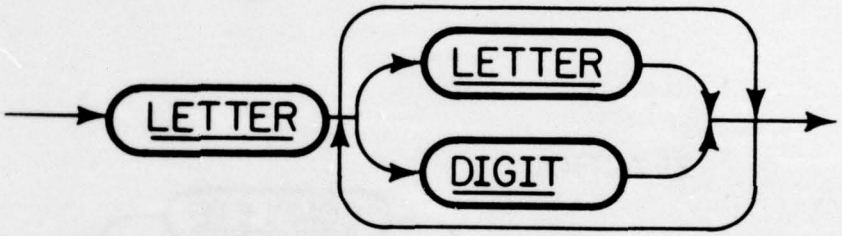
Figure 2.1. The Syntax Diagram (Third of six).

13. PROC-ID

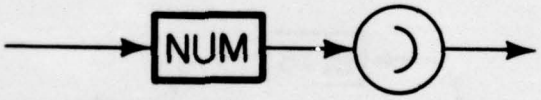


14. NAME

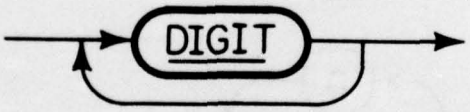
15. ID



16. LABEL



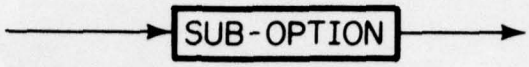
17. NUM



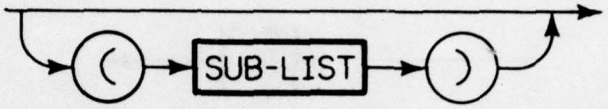
FP-4961

Figure 2.1. The Syntax Diagram (Fourth of six).

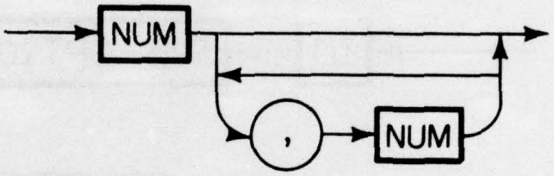
18. ORDER - INFO



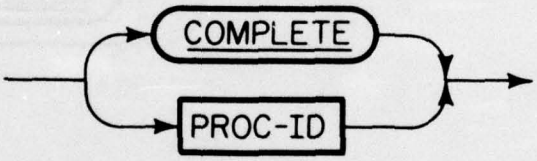
19. SUB-OPTION



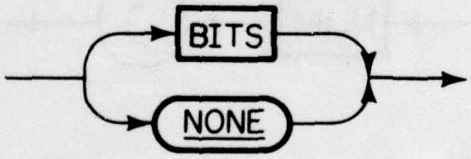
20. SUB-LIST



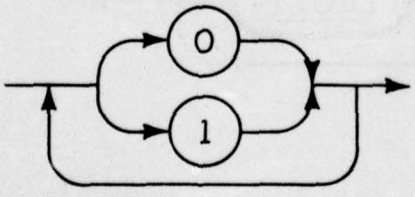
21. ACTION



22. NBITS

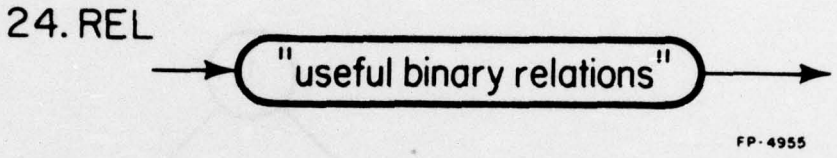


23. BITS



FP-4957

Figure 2.1. The Syntax Diagram (Fifth of six).



Terminal symbols are enclosed in circles or boxes with round ends. Non-terminal symbols are enclosed in rectangles. Those terminal symbols that are spelt differently in the text are listed below together with their spelling.

DL	carriage return, line feed.
LETTER	A,...,Z.
DIGIT	0,...,9.

Blanks may be inserted between terminals for ease of reading; they are ignored by the syntax analyser.

Figure 2.1. The Syntax Diagram (Sixth of six).

Block 1
1) Dest ← Source
2) Block 2 (1)
3) D ← S (1)
4) Block 3 (2,3)

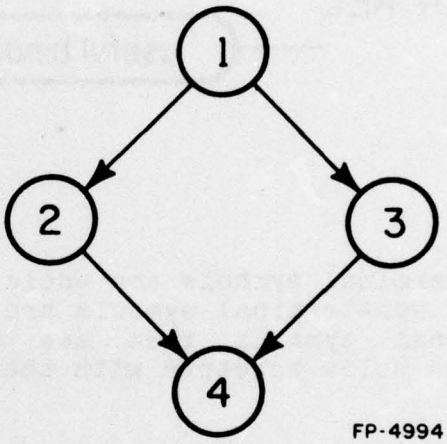


Figure 2.2. An Example Block Of Statements.

another. In statement 2 the control of a subnetwork identified as BLOCK2 by BLOCK1 is indicated.

BLOCK1 is an example of a complex process, which decomposes into two atomic processes (statements 1 and 3) and two other less complex processes BLOCK2 and BLOCK3 (statements 2 and 4). The order in which these four processes are to occur is indicated in the DL by the parenthesized integers to the right of the statements (termed ORDER-INFO in the syntax diagram). The directed graph farther right illustrates pictorially how the order information is to be interpreted.

2.2 The Three Block Types.

The three types of blocks, a process block (PROC), a decision making process block (DPROC), and an iterative process block (WPROC) are illustrated in figure 2.3.Their syntax is shown in productions 2 thru 4 of the syntax diagram. P is a PROC type, D is a DPROC type and I is a WPROC type block.

The PROC type block is used to represent a complex process which decomposes into other prossess denoted by the block's statements. Both types of statements may be used, so the constituent processes of a PROC block may be atomic or complex.

The DPROC type block is used to represent a branch point in the CS. Depending on the value of some variables external to the CS, control is switched to one of a set of blocks listed in the DPROC. These blocks may be any of the three types.

The WPROC type block represents a complex process that is only initiated if some predicate external to the CS is true. Upon its completion the process is reinitiated if the predicate is still true. Reinitiation continues as long as the predicate remains true.

A clearer idea of the processes that can be represented by the three block types may be got by examining figure 2.3 more closely.

Figure 2.3(a) shows a PROC type block named P which represents the following complex process: when the process P is requested, A is initiated together with the register transfer action D← S (move the contents of register S to register D). When A is completed process B is to be initiated. When both D← S and B are completed process A is to be reinitiated . Finally with the completion of A, process P is considered completed and

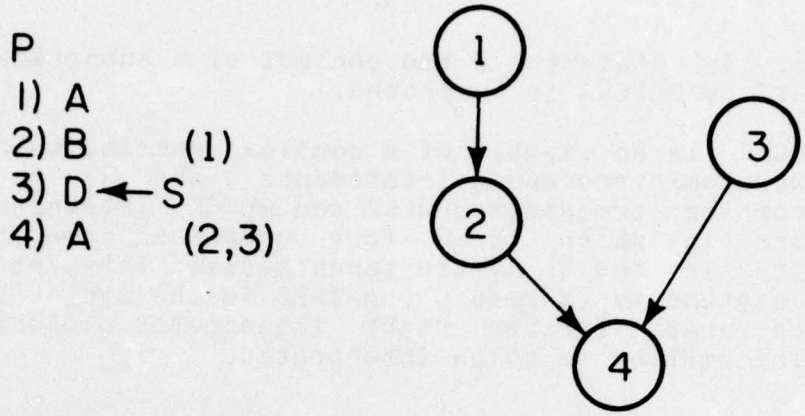


Figure 2.3(a).

D
DECODE × AS
00 ⇒ A
NONE ⇒ COMPLETE

Figure 2.3(b).

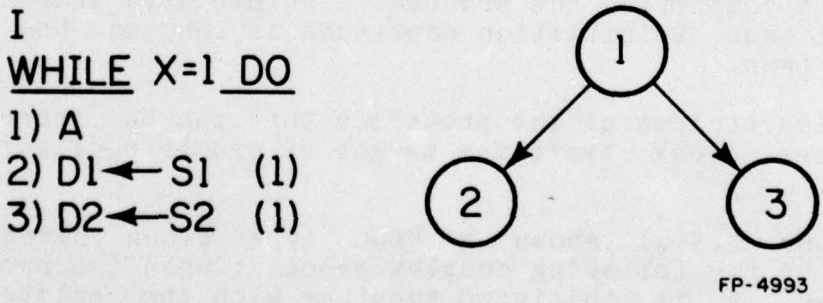


Figure 2.3(c). Further Example Blocks.

an aknowledge signal is sent along the link controlling P. The directed graph to the right of the block illustrates the ordering of the constituent processes of P. (Notice block BLOCK1 of figure 2.2 was an example of a PROC type block.) The processes identified by A and B may themselves be complex and will be represented by one of the three block types. They stand in the same relation to P as P does to the complex process that "calls" it.

Figure 2.3(b) shows an example of a DPROC type block. This indicates how the control is to branch according to the state of some external variables (X). If x0 V x1 = 0 then the process represented by the block A is to be initiated. Upon its completion the process represented by D is considered completed and control passes back to the calling process. The reserved word NONE stands for the union of all those conditions of the external variables not explicitly listed (i.e. x0 v x1 = 1). The reserved word COMPLETE indicates that if NONE is true the empty process is to be completed, in other words, that control is returned directly back to the calling process.

Figure 2.3(c) shows an example of a WPROC type block. If the predicate x = 1 is false, then the empty process is to be completed. If the predicate x = 1 is true, then process A is to be initiated and when it is completed the register transfer processes D1← S1 and D2← S2 are both to be initiated. When they are both completed the whole block, starting with A, is to be reinitiated as long as the predicate is true. As soon as the predicate is no longer true, and both D1← S1 and D2← S2 have been completed, the process represented by I is then considered completed. Control passes back to the calling process.

2.3 Comments On The Syntax.

The similarities between the DL and a procedure oriented language such as Algol are obvious. Both may be thought of as representing computational processes, and both translate into machines that realize those processes. Nevertheless there are several important differences. First, there is no parameter passing by the PROC-CALL type statements (this would be analagous to subroutine parameter passing). This was done for simplicity and there is no conceptual reason why the DL could not be extended to incorporate this facility. Parameters could be registers, then a block could be shared by several similar processes which operate on different registers. (The translator would have to add switches to the DS, so that different sections of the DS could be switched to the same section of the CS for

processing.) Second, the DL explicitly indicates the order in which processes are to be initiated, whereas a procedure oriented programming language assumes a serial operation of the processes represented by its statements, modified only by branch statements. Finally, the DL does not have a defined data set nor are any data operators defined. These should be suggested by the particular requirements of the target system.

3. THE MODULES

Before going on to show how the DL translates onto a set of modules, it is necessary to define the modules more precisely. Many useful sets of asynchronous modules have been proposed [Cla, Kel]. The set defined below was chosen because its members have a natural correspondence with the DL.

Each module is formulated with the aid of a state diagram, after the style of Keller [Kel]. In this formulation the modules are defined at the level of signal behavior. If the modules were to be built some decision on the signalling convention for the request and the acknowledge signals would be necessary, then flow tables could be constructed and the detailed design of the modules could be completed. (Figure 3.1 shows three of the most common signalling conventions.)

The design of the modules has been completed for some signalling conventions. Patil has presented designs for the module set [Pat 72] based on a set of simpler micro-modules (the NOT gate, the EXCLUSIVE-OR gate and the MULLER-C gate), which use simple (also called symmetric) signalling. These are fault-tolerant to any combination of stuck-at faults on their links. (This is a consequence of using simple signalling on the interconnecting links, and is independent of the way in which the module set is realized.) Peterson has presented designs for the module set [Pet] using reset (also called asymmetric) signalling. These are fault-tolerant to single stuck-at faults on their links. (This is a consequence of their design rather than the signalling convention used on the interconnecting links.) No designs for the module set are known to the author which use other signalling conventions. Were such sets desired the design approach presented in [Kel] deserves consideration.

3.1 The Source Module.

The source (So) module is shown in figure 3.2. On the left is a diagrammatic representation. It has one output link. (The arrows on links are always shown in the direction that the request signal travels.) Operation of the module is shown by the state diagram to the right. The arcs of the state diagram are labelled as follows: input signal / set of resulting outputsignals. The occurrence of a request signal is denoted by R, an acknowledge signal by A. (In later sub-sections subscripts are used to indicate the link to which the signal belongs.) From the state diagram it can be seen that the module transmits a request signal whenever an acknowledge signal is received. It thus acts as a source of requests.

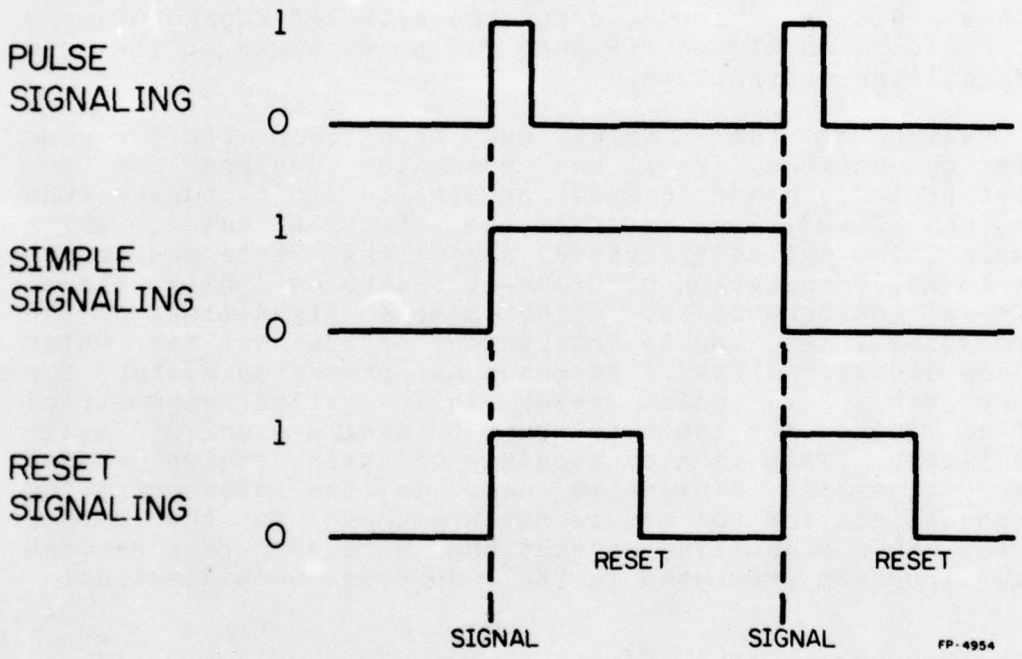


Figure 3.1. Some Signalling Conventions.

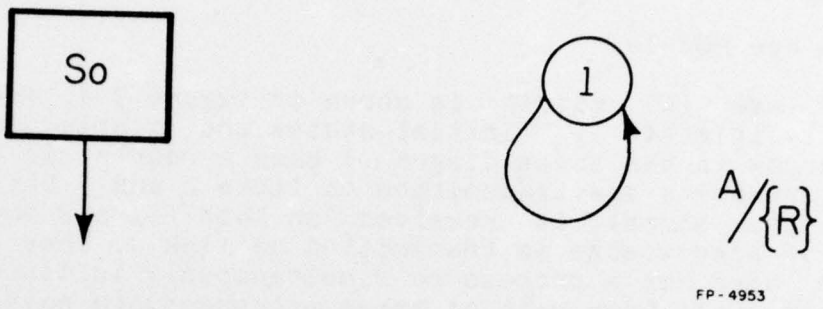


Figure 3.2. The Source Module.

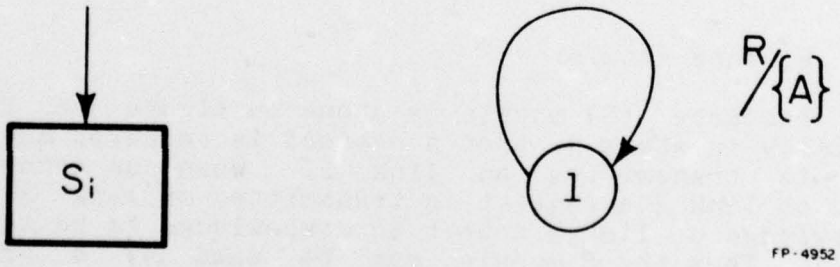


Figure 3.3. The Sink Module.

3.2 The Sink Module.

The sink (Si) module is shown in figure 3.3. In operation it complements the So module. Whenever it receives a request signal it transmits an acknowledge signal. It thus acts as a sink for requests.

3.3 The Wye Module.

The wye (W) module is shown in figure 3.4. The module is initially in state 1. (Initial states are distinguished by a free arrow in the state diagram.) When a request is received on link 1, requests are transmitted on links 2 and 3 both. When an acknowledge signal is received on both links 2 and 3 (in any order) an acknowledge is transmitted on link 1. Thus a W module may be used by a process to simultaneously initiate two other processes. Only when both of these processes are completed (i.e. when the module has received acknowledge signals on links 2 and 3) is the calling process notified by the transmittal of an acknowledge along link 1.

The function A, shown to the right of the state diagram, places constraints on the way the module's environment acts on it. For any state q of the module $A(q) \subseteq P[I]$ represents the allowable input sets (P[I] is the power set of all possible inputs, I). If $S \in A(q)$ then any subset of the lines corresponding to the elements of S is allowed to be signalled concurrently. Hence in state 2 inputs A2 and A3 can be signalled concurrently. It is assumed that this and all other modules that can receive concurrent signals choose to assimilate the signals one at a time in an arbitrary order. The state diagram then exhibits the required behavior.

3.4 The Sequence Module.

The sequence (S) module is shown in figure 3.5. The module is initially in state 1. When a request is received on link 1 a request is transmitted on link 2. When an acknowledge is received on link 2 a request is transmitted on link 3. Finally an acknowledge on link 3 causes an acknowledge to be transmitted on link 1. Thus the S module may be used by a process to initiate two processes one after the other. The calling process requests on link 1 whereupon the process controlled by link 2 is performed. On its completion the process controlled by link 3 is performed, and an acknowledge is returned to the calling process.

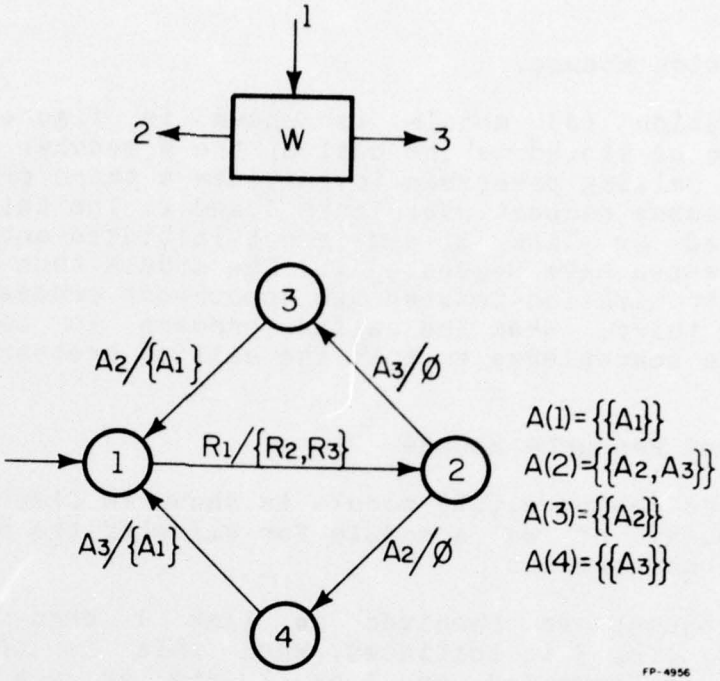


Figure 3.4. The Wye Module.

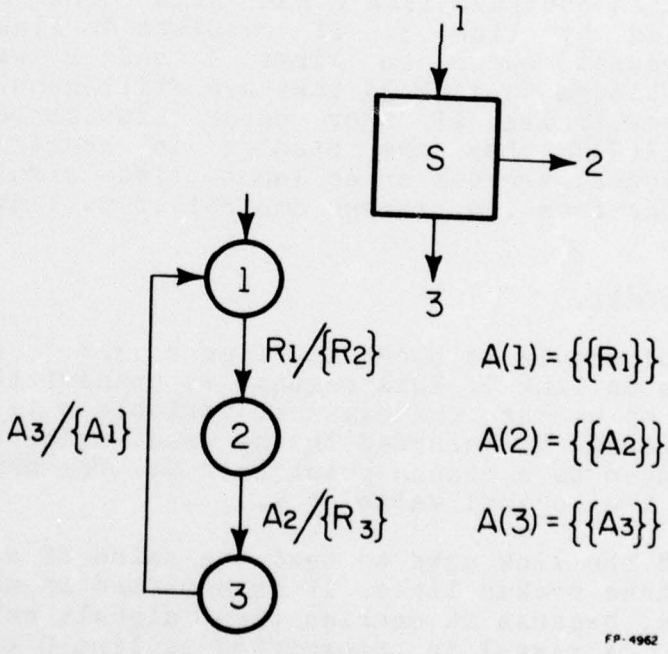


Figure 3.5. The Sequence Module.

3.5 The Junction Module.

The junction (J) module is shown in figure 3.6. Its operation can be viewed as the dual of the W module. It may be used by two calling processes to initiate a third process. The calling processes request over links 1 and 2. The third process is controlled by link 3, and is not initiated until both the calling processes have requested it. The module thus performs an act of synchronization between two concurrent processes, before initiating a third. When the called process is completed it broadcasts an acknowledge to both the calling processes.

3.6 The Shared Resource Module.

The shared resource (SR) module is shown in figure 3.7. It can be thought of as a module for allowing two processes to share some other process.

If a request is received on link 1 then the process controlled by link 3 is initiated. When this is completed an acknowledge is received on link 3 and an acknowledge is transmitted along link 1. Similarly if the request is received on link 2. Thus either the calling process which controls link 1 or the process which controls link 2 can gain control of the process controlled by link 3. If requests on links 1 and 2 overlap (i.e. requests occur on links 1 and 2 without an intervening acknowledge on link 3) they are still handled in the order in which they arrive. If they occur simultaneously (as they can; see A(1)) they are handled in arbitrary order. Similarly if a request arrives on an input link simultaneously with an acknowledge from the process controlled by link 3.

3.7 The Decode Module.

The decode (D) module is shown in figure 3.8. A calling process requests on link 1. This request is transmitted on link 3 or 2 depending on whether the external variable x is true or false. The acknowledge is returned in the usual manner. Thus the D module may be used as a branch point in a CS. The branch taken is controlled by the logical value of x.

In figure 3.8 the link used to test the value of x is shown as a bundle of three broken lines. It is depicted in more detail than normal links, because it carries three signals rather than the normal two. A signal is transmitted on line 0 to test the external variable x and is returned on either I_T or I_F

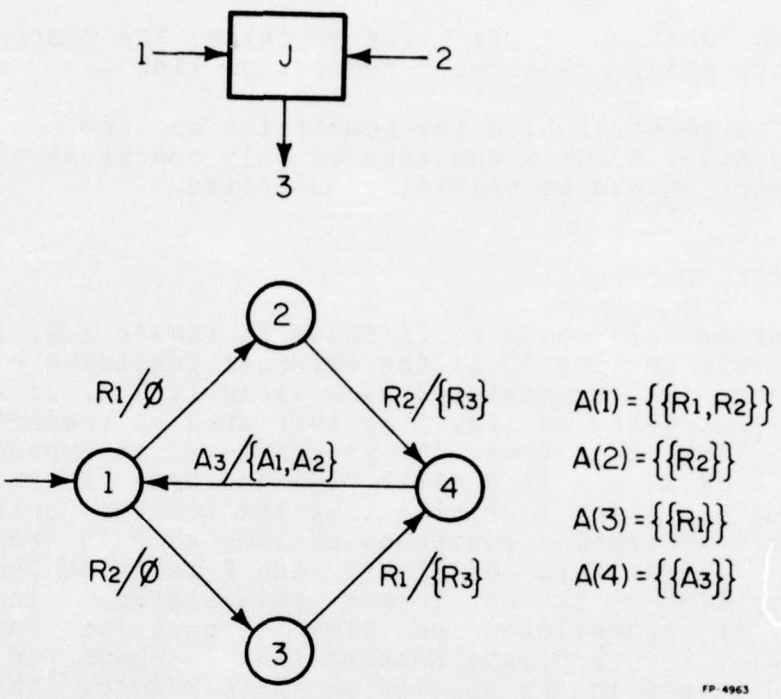


Figure 3.6. The Junction Module.

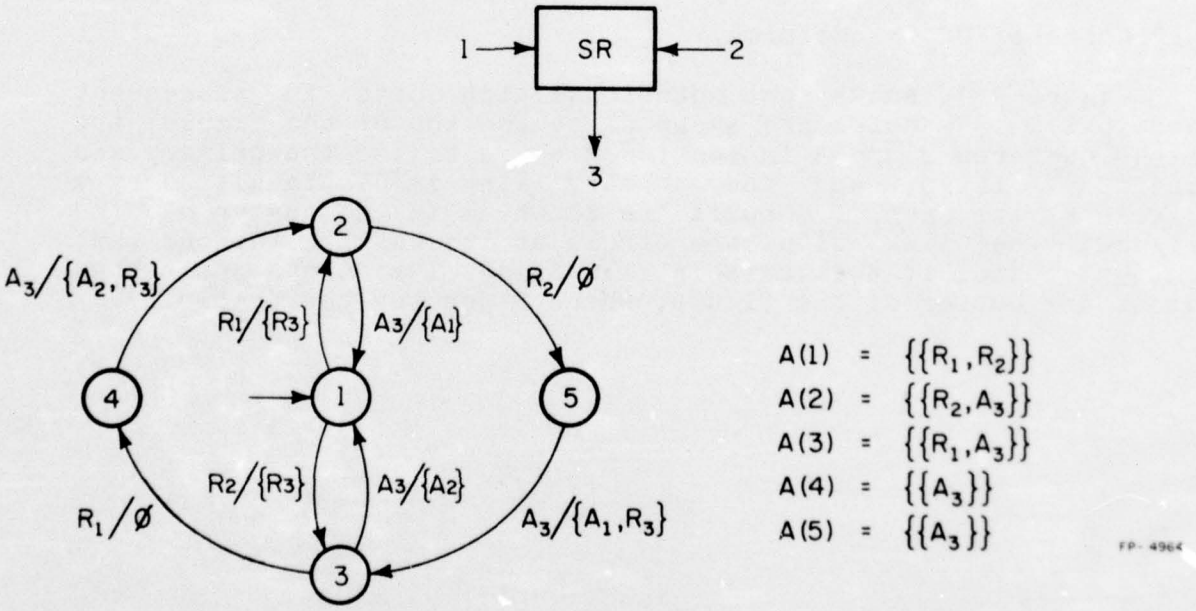


Figure 3.7. The Shared Resource Module.

depending on whether x is true or false. The testing occurs every time the module receives a request on link 1.

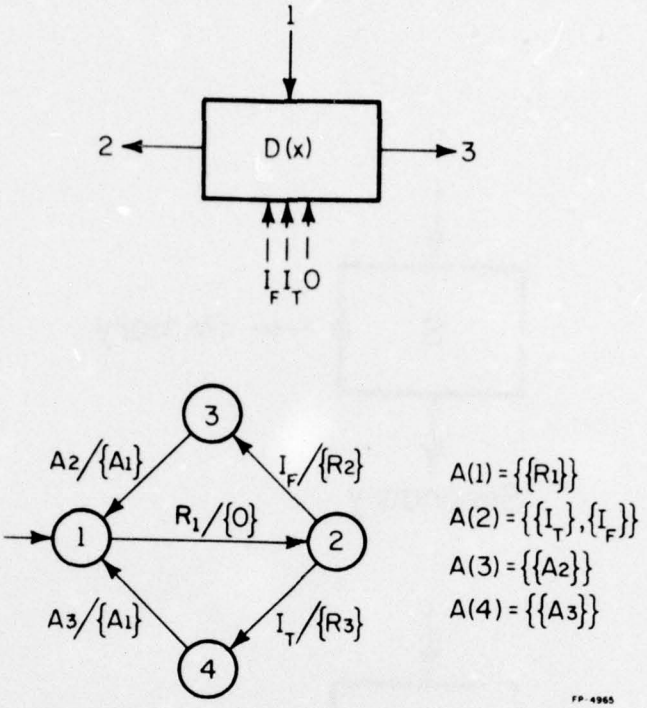
As a final point, notice the constraint on the environment described by A(2). Since x can take on only one value at a time, this constraint should be trivially satisfied.

3.8 The Iterate Module.

The iterate (I) module is shown in figure 3.9. A calling process requests on link 1. If the external predicate P is false an acknowledge is transmitted back along link 1. If P is true the process controlled by link 2 is initiated by transmitting a request on link 2. When it is completed an acknowledge is received on link 2, and if P still remains true a request is retransmitted on link 2 reinitiating the process controlled by link 2. This reinitiation continues as long as P is true. If an acknowledge is received on link 2 when P is false the process controlled by link 2 is no longer reinitiated. Instead an acknowledge is transmitted on link 1 back to the calling process. Notice the predicate testing link is shown in greater detail, as it was in the section on the D module. This module may be used in a CS when a process is required to be reinitiated as long as some external predicate holds.

3.9 Comments On The Modules.

Figure 3.10 shows some notational aids used in subsequent sections. S modules are shown as at the top of the figure; the links numbered 2 and 3 in section 3.4 are called the primary and secondary links, and the primary link is distinguished by a circle at its base. D modules are shown as in the center of the figure; the link with the circle at its base is the one that relays control if the variable x is false. I modules are shown as at the bottom of the figure; where P denotes the predicate.



igure 3.8. The Decode Module.

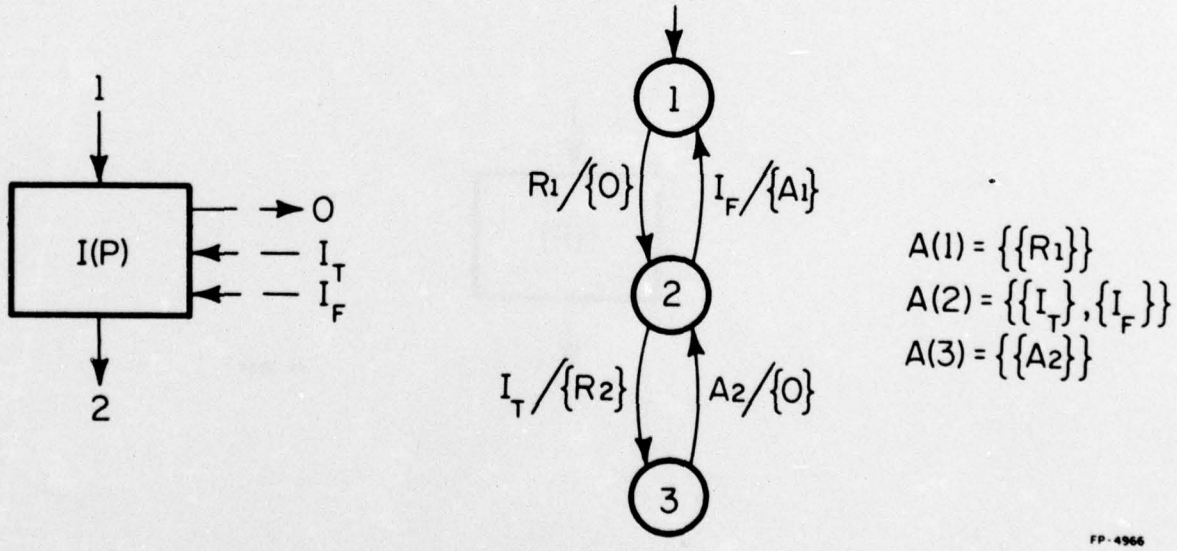


Figure 3.9. The Iterate Module.

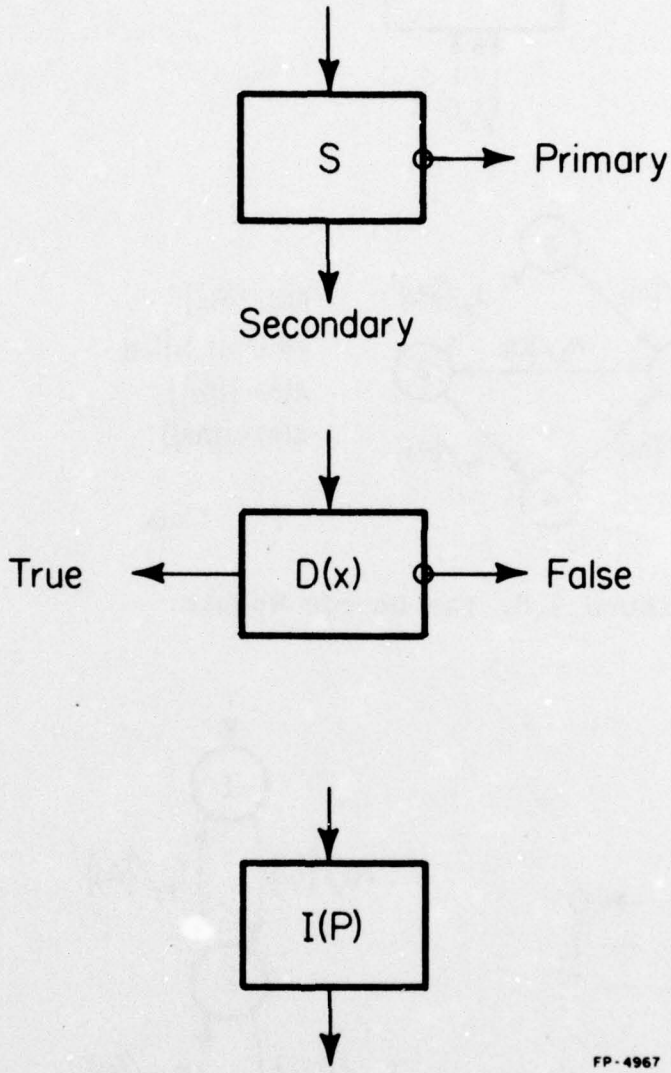


Figure 3.10. The Diagrammatic Representation Of Some Modules.

4. THE TRANSLATION PROCEDURE

A procedure is presented below that translates descriptions in the DL into networks of modules. Thus for each description in the DL there exists an asynchronous machine that is uniquely determined by the syntax (defined in the syntax diagram of section 2), the translation procedure below and the state diagrams that define the modules in section 3.

The behavior of these asynchronous machines is consistent with the informal interpretation of the DL accompanying the syntax diagram in section 2, and the informal interpretation of the behavior of the modules accompanying the state diagrams in section 3.

The translation procedure is quite simple, as can be deduced from figure 4.1. Each of the three blocks in figure 2.3 is shown again, this time alongside of its modular realization. PROC blocks translate to networks of W, S, and J modules, DPROC blocks translate to trees of D modules, and WPROC blocks translate to networks of W, S and J modules headed by an I module.

The translation procedure partitions naturally into two phases: determining the intra-block connections of the modules, and determining the inter-block connections of the modules.

4.1 The Intra-Block Connections.

In the case of PROC and WPROC blocks the intra-block connections of W, S, and J modules are deduced from their intra-block matrices. Examples of these are also shown in figure 4.1. These matrices are derived directly from the order information. If m denotes the intra-block matrix of a block, then m(i,j) = 1 iff the statement labelled i in that block has the integer j in its order information. The matrix can be interpreted to represent the binary relation "is the successor of ". Then m(i,j) = 1 indicates that the statement labelled i "is the successor of" the statement labelled j. In every PROC and WPROC block there should be at least one statement which is not the successor of any other in that block; it is distinguished by having no order information. To complete the matrix a dummy statement is assumed which precedes such statements. (In the two examples of the figure, the dummy statements are labelled 0, but they need only be labelled distinctly.)

Once the intra-block matrix of a block has been found an S

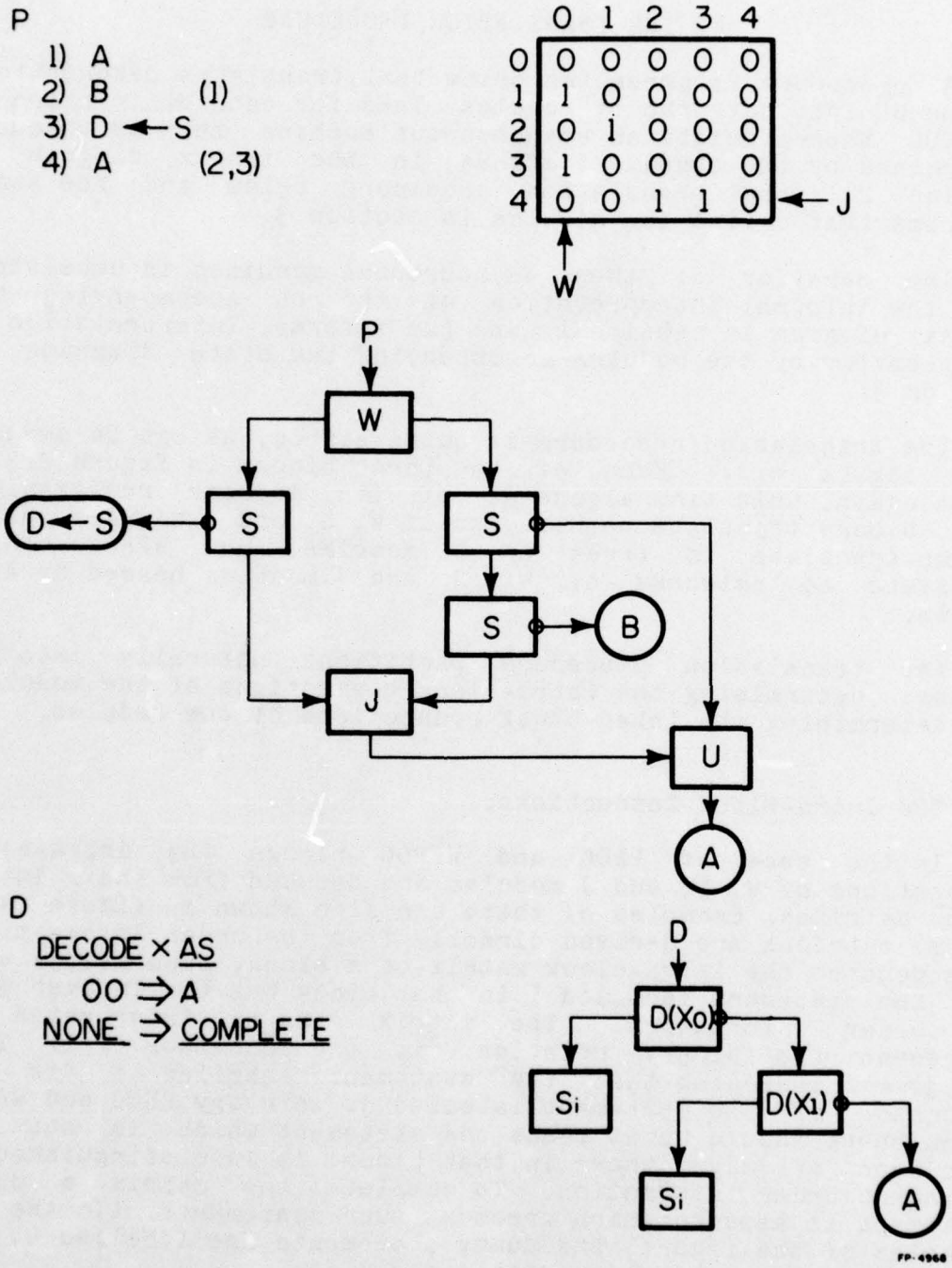


Figure 4.1. Examples Of Translation (First of two).

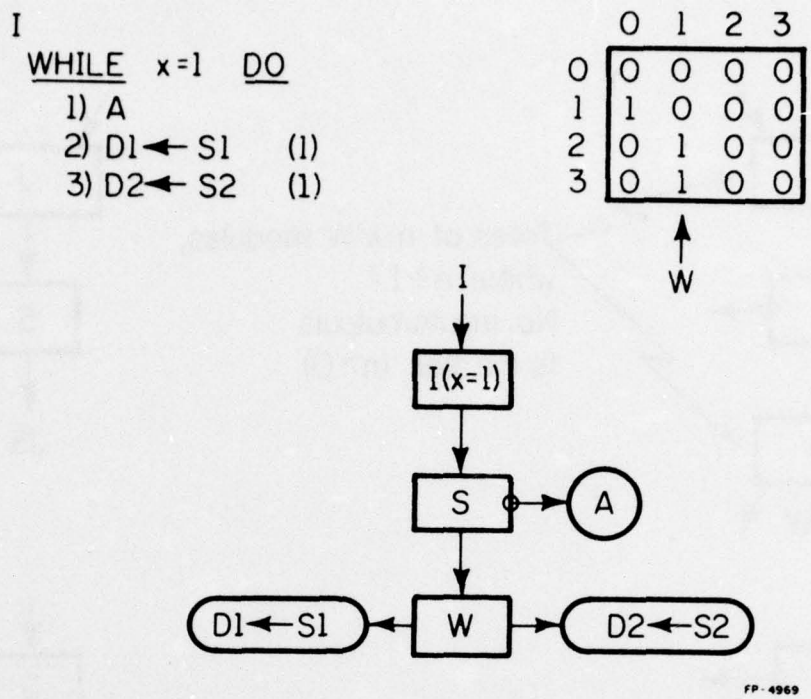


Figure 4.1. Examples Of Translation (Second of two).

module is allocated to each statement in the block, except to dummy statements and those having no successors. (A statement j has no successors iff $m(i,j) = 0 \forall i$.) For each statement i, a tree of (n - 1) J modules is allocated to i iff $\sum_j m(i,j) = n$. If i has already been allocated an S module, its input link is connected to the output link of the J tree. Similarly for each statement j, a tree of (n - 1) W modules is allocated to j iff $\sum_i m(i,j) = n$. If j has already been allocated an S module, its secondary output link is connected to the input link of the W tree. At this point each statement has been allocated a star shaped network of modules. This star comprises a J tree connected to an S module, which is connected to a W tree. In general these stars may be degenerate, and have any of the forms shown in figure 4.2, or in the most degenerate case no modules at all. The final step in determing the intra-block connections is made directly from the matrix. If $m(i,j) = 1$ then an output link of the star allocated to statement j is connected to an input link of the star allocated to statement i. In this way each statement becomes associated with a unique output link. In the case of those statements which are allocated an S module this link is the primary output link of the S module. In the case of statements with no successors this link is either one output of the star allocated to its predecessor (whence it would be the secondary output of an S module or the output of a W

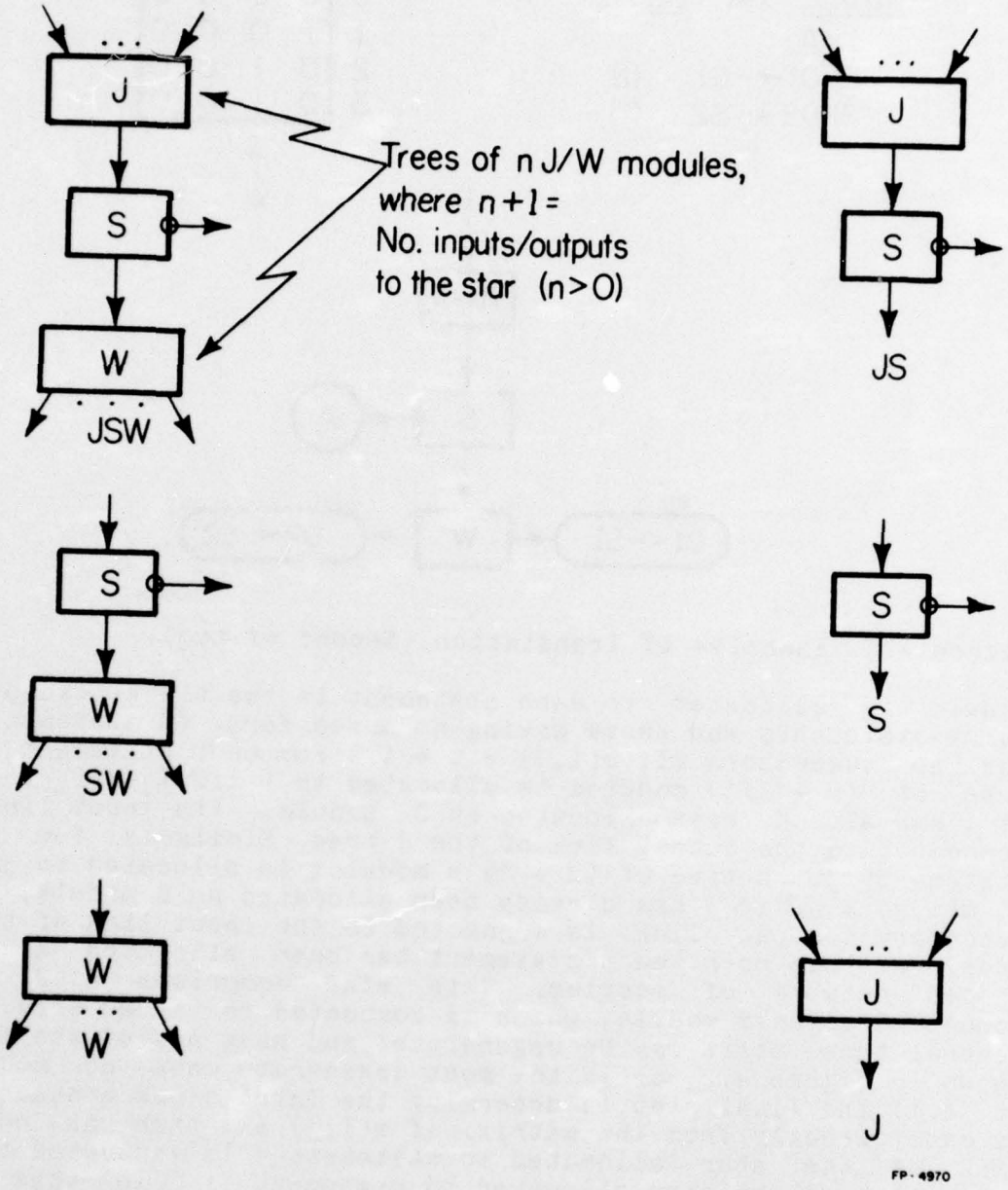


Figure 4.2. Examples Of Stars.

module), or the output link of a J tree.

The important point to notice about the structure of blocks formed in this way is that they only have a single input link and that no intra-block connections involve the primary output links of S modules.

In the case of WPROC blocks the output link of an I module is connected to the input link of the block, to complete the intra-block translation.

In the case of DPROC blocks a tree of D modules is allocated as follows. The argument of the DECODE statement is a name (see production 9 of the syntax diagram), which represents a string of external bit variables. If there are n bits in the string the tree needs to be n levels deep. The first bit is tested by the top D module, the second by the two D modules on the second level of the tree, and so on to the n-th level. Each directed path thru the D trees links corresponds to a unique value for the bit string. Those paths which correspond to a COMPLETE action (see productions 10 and 21 in the syntax diagram) are terminated with an Si module. Any D module with both output links connected to Si modules as a result of this can be replaced by an Si module.

4.2 Inter-Block Connections.

Inter-block connections are determined in a manner similar to the intra-block connections. They are deduced from the inter-block matrix M. A fragment of such a matrix is given in figure 4.3, showing the inter-block connections of a series of blocks, one of which is the PROC block in figure 4.1. This matrix is derived directly from the block identifiers, the PROC-CALL type statements, and the DLIST actions which are PROC-IDs. M(i,j) = k iff block j has k PROC-CALL type statements whose PROC-ID is i, or (in the case of j being a DPROC block) k DLIST actions whose PROC-ID is i. (Note that i and j are no longer necessarily integers; they may be alphameric strings.) As before the matrix can be interpreted to represent the binary relation "is the successor of".

If M(i,j) = 1 then the output link of block j, which is associated with the statement or DLIST action whose PROC-ID is i, is connected to the input link of the block with identifier i. If $\sum_j M(i,j) = n$ $(n>1)$ more than one block output link connects to the input of block i. This is done with a tree of (n - 1) SR modules. If $\sum_j M(i,j) = 0$ then block i has no

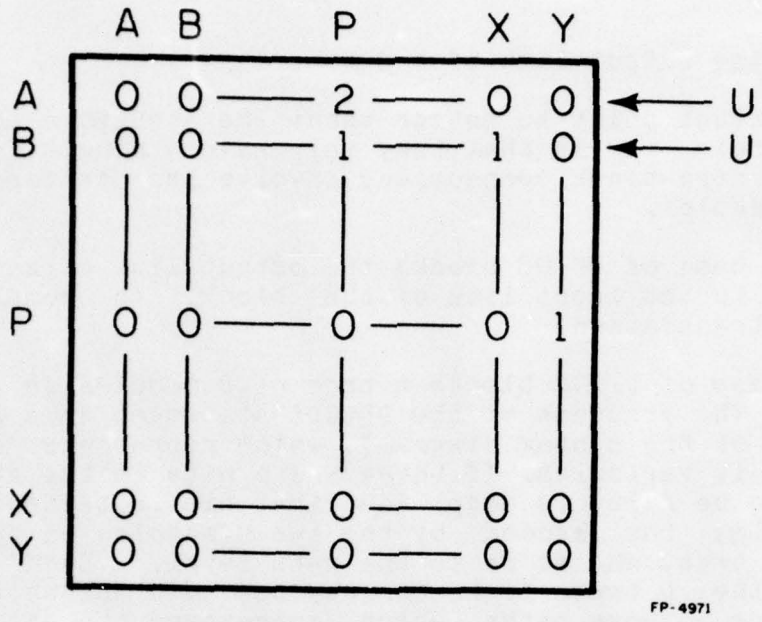


Figure 4.3. An Example Of An Inter-Block Matrix.

predecessors, and its input link must be capped with a So module.

Finally, register transfers may in general be shared, in other words the same REG-TRF type statement occurs more than once. This can be deduced from an auxiliary inter-block matrix M'. $M'(i,j) = k$ iff block j contains k REG-TRF statements i. If $\sum_j M'(i,j) = n \quad (n>1)$ then the register transfer i is shared using a tree of (n - 1) SR modules as before.

4.3 Comments On The Translation Procedure.

The intra-block matrices together with the inter-block matrix and auxiliary inter-block matrix may also be used to check that the network of modules does not contain potential deadlock [Mud].

N.B. Read SR for U in Figures 4.1, 4.3 and 5.3.

5. AN EXAMPLE USING THE DL

The design example presented here is a single instruction computer called SIM (Single Instruction Machine) which performs the single three address instruction SUBTST A, B, P. The operation of this instruction is equivalent to the two instructions;

1. A ← C(A) - C(B)
2. IF C(A) = 0 THEN PC ← C(P)

(PC is the program counter.) Its control structure is described in the DL below.

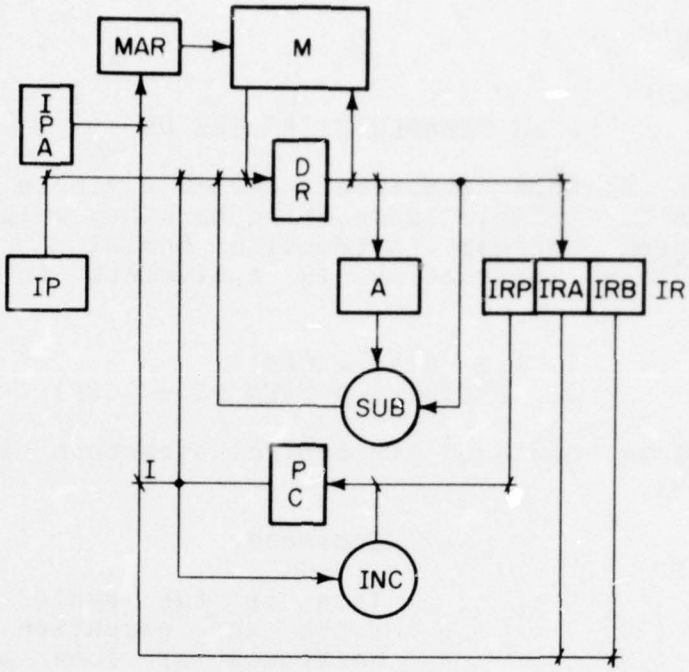
```
SIM                                    Comments.
WHILE RUN = 1 DO
   1. DECI                             This is the basic instruction
   2. FETCH     (1)                    fetch and execution cycle. It
   3. EXEC      (2)                    continues as long as the RUN
                                       button is on. Furthermore the
                                       block DECI tests the interrupt
                                       flag before each cycle.

DECI                                   INT is the interrupt flag. If it
DECODE INT AS                          is on (=1) the interrupt is
       1 => INTR                       handled by block INTR.
       0 => COMPLETE

INTR                                   INTR handles the interrupt. IPA
   1. MAR ← IPA                        holds the address at which the
   2. DR ← IP                          interrupting data is to be
   3. M ← DR    (1,2)                  stored. The interrupt is only for
                                       inputting data.

FETCH                                  This is the FETCH routine. MAR is
   1. MAR ← PC                         the memory address register. PC
   2. DR ← M     (1)                   is the program counter. DR is the
   3. PC ← INC   (1)                   data register; it holds the data
                                       input to and output from M. INC
                                       is the incrementing unit. Its
                                       output is its input plus one.

EXEC                                   This executes the three address
   1. IR ← DR                          instruction. IR is the
   2. MAR ← IRA   (1)                  instruction register. It is split
   3. DR ← M      (2)                  into three subfields. IRA holds
   4. DR1 ← DR    (3)                  the address A, and IBA holds the
   5. A ← DR      (3)                  address B. The REG-TRF statements
```



DP-1560

Figure 5.1. The Data Structure.

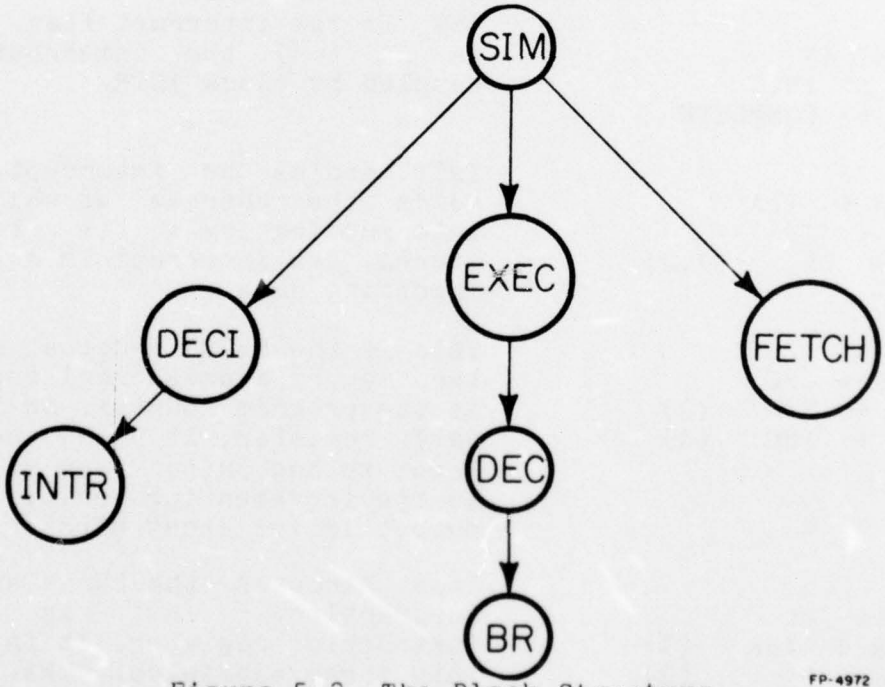


Figure 5.2. The Block Structure.

FP-4972

```
   6. MAR ← IRB     (3)
   7. DR ← M        (4,5,6)
   8. DR ← SUB      (7)
   9. MAR ← IRA     (7)
  10. DR2 ← DR      (8)
  11. M ← DR        (8,9)
  12. DEC           (10,11)
```

4 and 10 are links out of the CS (DR1 and DR2 are dummy registers) which indicate to the external environment that DR is loaded with data which could be accessed by some external device.

```
DEC
DECODE DR AS
       0...0 => BR
       NONE  => COMPLETE
```

DEC checks to see if the result of the subtraction is zero. If it is, block BR is initiated.

```
BR
   1. DR ← PC
   2. PC ← IRP      (1)
   3. MAR ← PC      (2)
   4. M ← DR        (3)
   5. PC ← INC      (3)
END
```

IRP holds the branch address. C(PC) is loaded at the branch address and PC is loaded with the branch address.

The resulting DS is shown in figure 5.1. The lines connecting the registers represent buses, and the slashes at the bus intersections represent identity operators. The one shown in figure 5.1 next to the letter I moves data from the program counter to the memory address register when requested to do so by the S module corresponding to the DL REG-TRF statement "MAR ← PC" (see FETCH block statement 1). The block structure of the design is shown in figure 5.2, and the modular realization of the CS is shown in figure 5.3. (In this figure FETCH(2) indicates a link from statement 2 of block FETCH.)

When a CS is hooked up to a particular DS the determinism of the total system must be verified. The presentation of a systematic method to check for determinism is outside the scope of this report. In the above example the determinism was checked by visual examination of the REG-TRF statements in the DL description. (This could clearly be inadequate in a much larger design.)

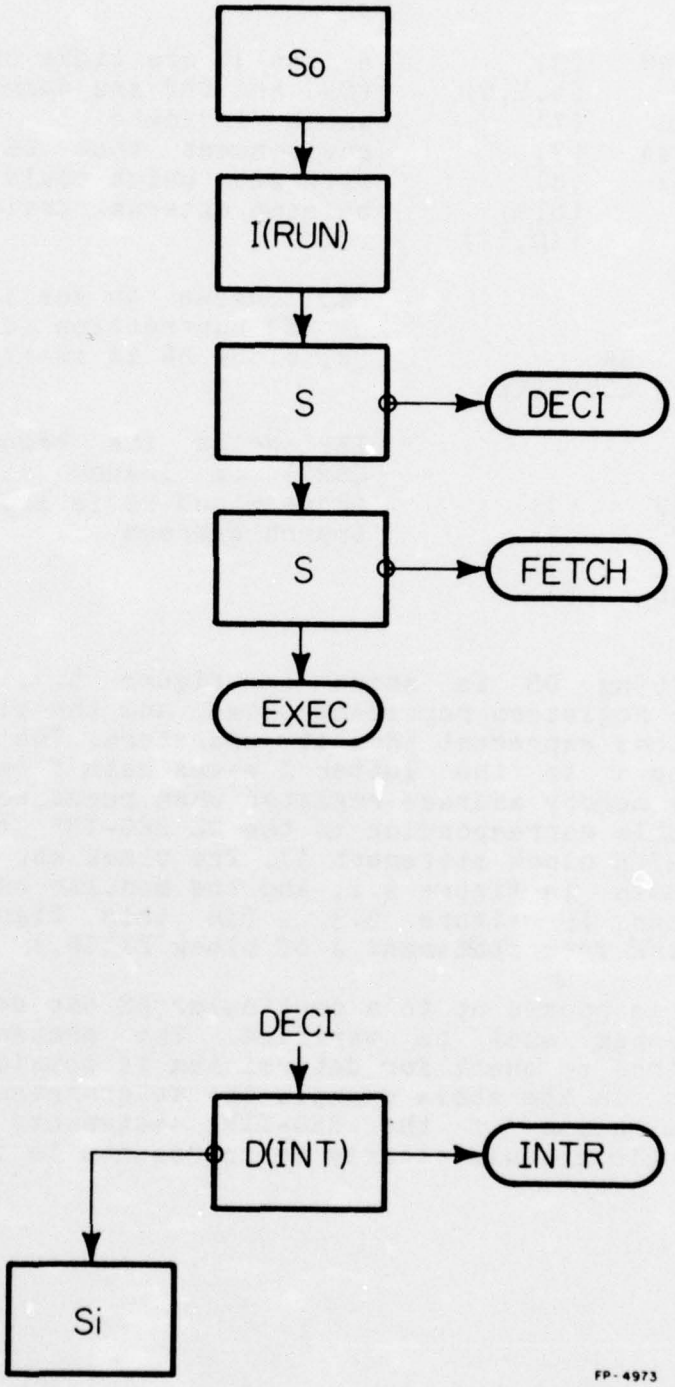


Figure 5.3. The Modular Realization Of The CS (First of four).

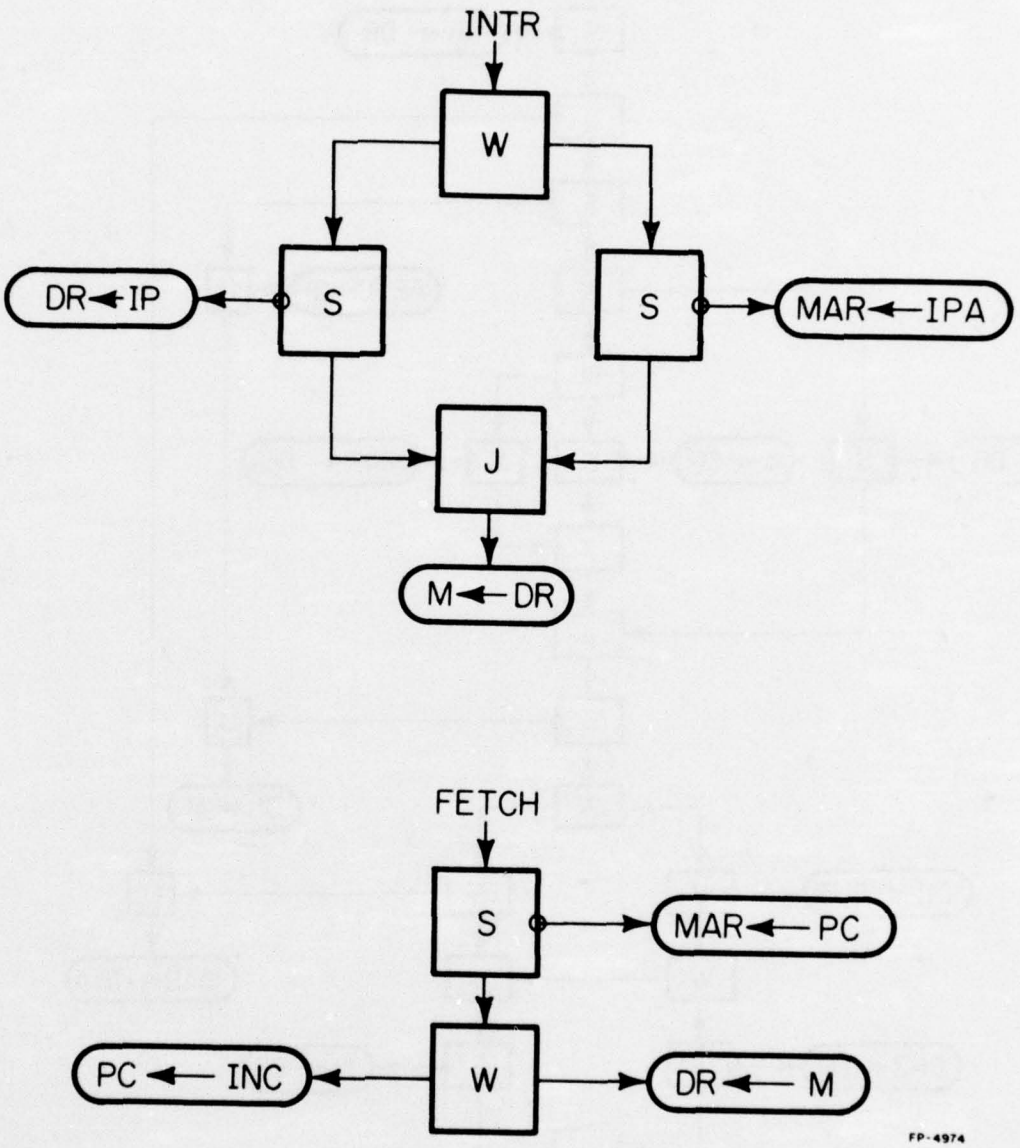


Figure 5.3. The Modular Realization Of The CS (Second of four).

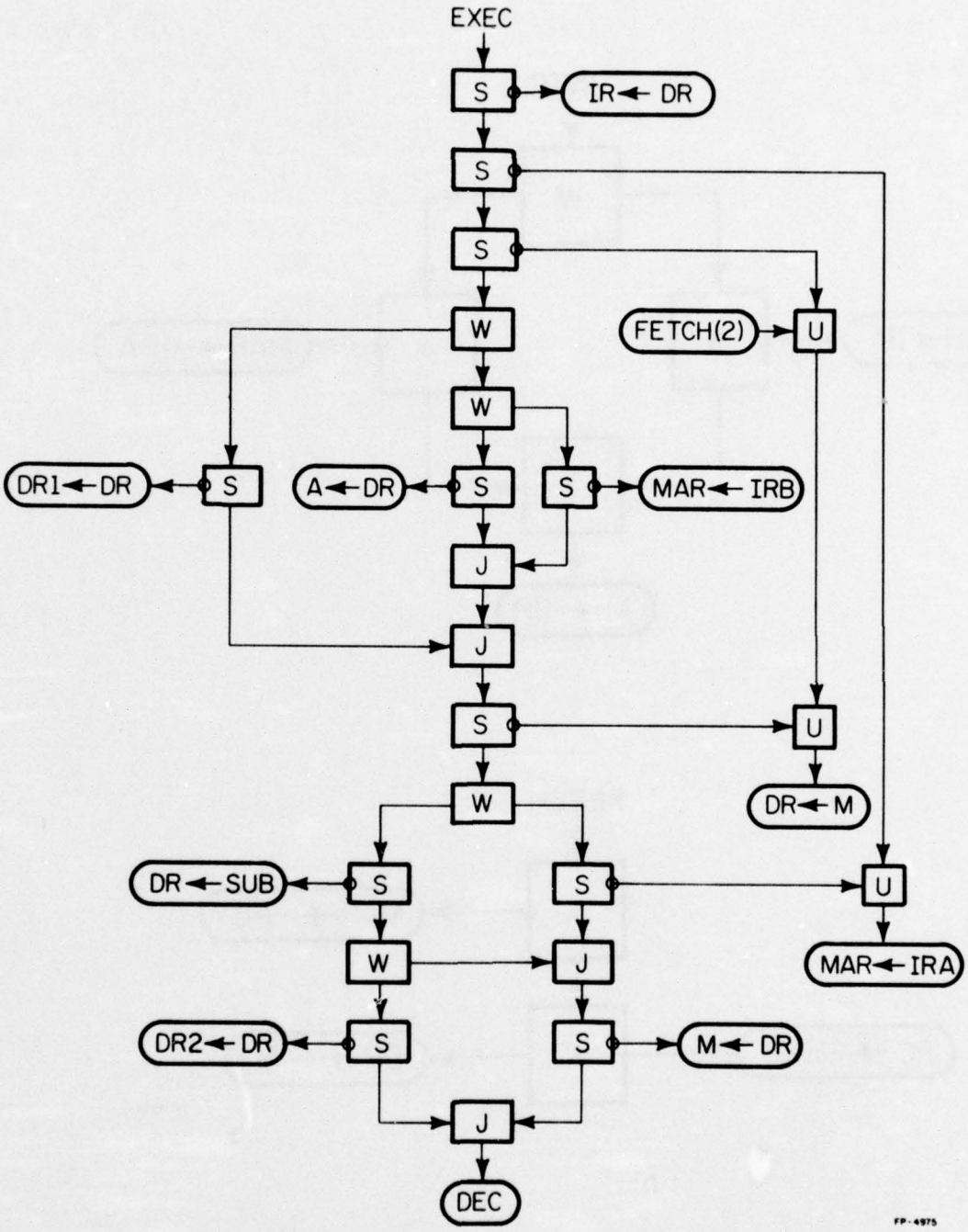


Figure 5.3. The Modular Realization Of The CS (Third of four).

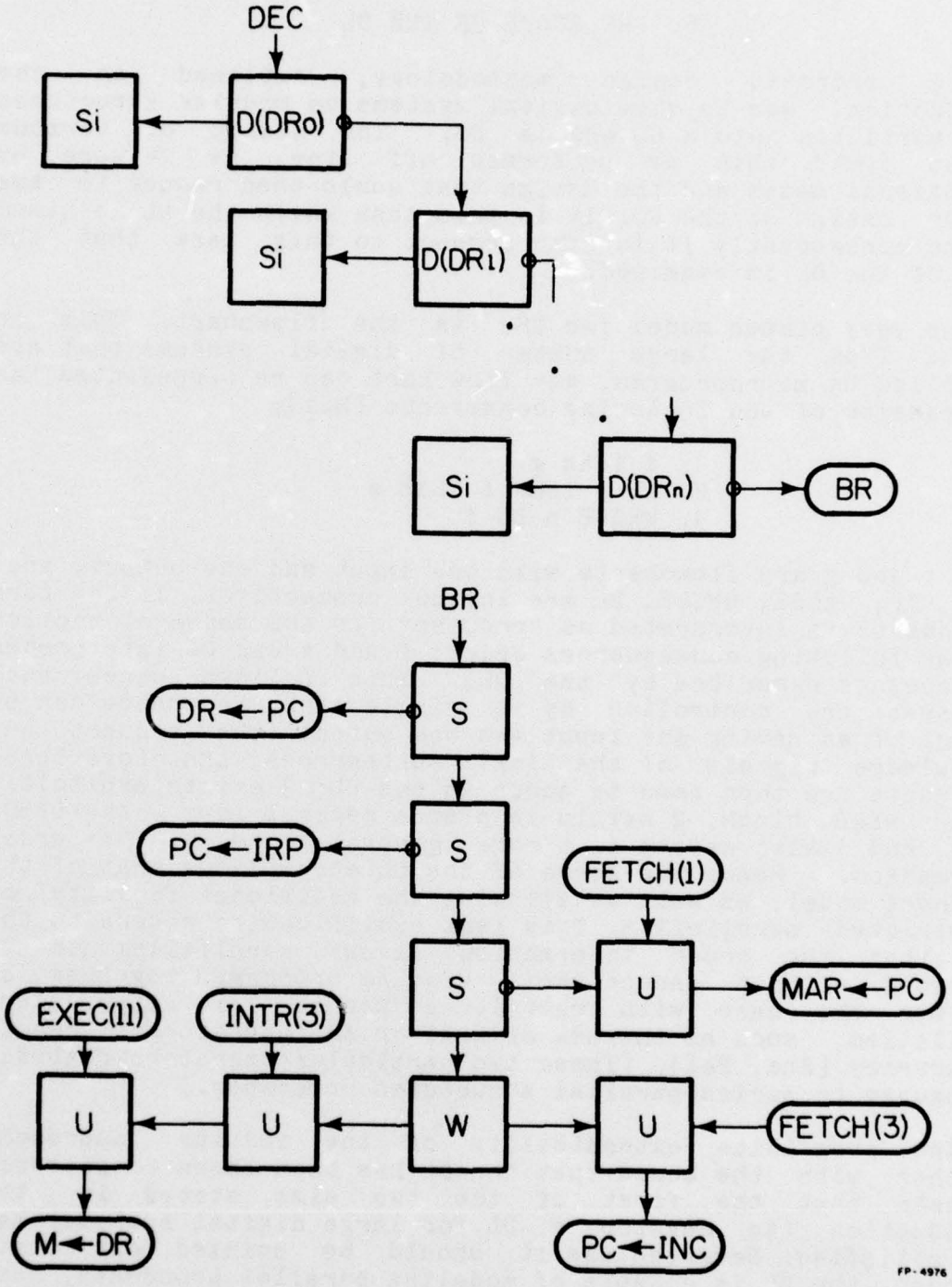


Figure 5.3. The Modular Realization Of The CS (Fourth of four).

6. THE SCOPE OF THE DL

The proposed design methodology, outlined in the Introduction, was to view digital systems as modular structures which partition into a CS and a DS. The design of various modules could then be performed off line, as it were, by conventional means and the design task would then reduce to the modular design of the CS. It is this task which the DL is aimed at, and consequently it is with respect to this task that the scope of the DL is examined.

One very common model for CSs is the flowchart. This is evident from the large number of digital systems that are controlled by microprogram. Any flowchart can be represented as an expansion of the following constructs [Mil];

1. f THEN g
2. IF p THEN f ELSE g
3. WHILE p DO f

where f and g are flowcharts with one input and one output, and, THEN, IF, ELSE, WHILE, DO are logical connectives. If the term "flowchart" is interpreted as "process" (in the sense of section 2), the following consequences arise: f and g can be interpreted as processes described by the DL. This follows since such processes are controlled by a single link, and hence can be thought of as having one input and one output (the request and acknowledge signals of the link). Furthermore, the above three constructs are then seen to occur in the DL; 3 exists explicitly - the WPROC block, 2 exists in a more general form - the DPROC block, and 1 also exists in a more general form - the order information. Hence the scope of the DL encompasses that of the flowchart model, as well as allowing the additional facility of unrestricted parallelism. This last qualification refers to the fact that the order information allows parallelism to be described without unnecessarily binding processes together, as is often the case with restricted methods of representing parallelism, such as the use of NEXT or AND operators to denote concurrency [Sha, Bel]. (These two particular operators restrict a language to series/parallel structured processes.)

The indefinite extensibility of the modular approach, together with the scope that the DL has been shown to possess, suggests that the first of the two aims stated in the introduction (to develop a DL for large digital systems) has been satisfied. Nevertheless it should be pointed out that, although the DL is capable of modeling parallel processes, some processes such as pipelined processes, overlapped processes,

etc., (which are closely related to parallel processes) are outside the scope of the DL. Such capabilities are often required when a high performance system is to be designed [Den].

Finally one other model should be mentioned in passing. This is the Petri net [Hol], and it is a more general model than the flowchart. It can describe CSs that are outside the scope of the DL,but much of the additional scope of the Petri net is of questionable use, and the considerable complexity of any Petri net that models a large CS can confuse rather than aid the design process.

7. A QUANTITATIVE LOOK AT THE DL

In this section it is shown, in a quantitative way, that the computational complexity of an algorithm required to check a design in the language for correct syntax and absence of deadlock is of a lower degree, than that required to check an arbitary network of modules for absence of deadlock. Hence it may be concluded that fault free design using the DL is easier (from a computational complexity view point) than constructing a CS by the ad hoc interconnection of modules.

Computational complexity analyses are concerned with the "amount of work" done by algorithms. For the purpose of this discussion this is measured in terms of the number of operations which must be performed.

7.1 The Complexity Of Checking The Syntax Of A Design In The DL.

The first step in checking the syntax of a design in the DL is to take the string of characters representing the design and to partition it into a sequence of tokens, where a token is a string of characters that forms a single logical unit.

The syntax given in the syntax diagram of figure 2.1 can be simplified if the following logical units are tokenized: identifiers, decode statements, while statements, labels, register transfers and bit strings. The resulting simplified grammar is shown in figure 7.1. Notice that the tokenized entities are now represented by a single generic terminal symbol.

Strings of symbols produced by this grammar can be checked for syntactical correctness by the finite automaton whose control state diagram is shown in figure 7.2. (This implies the tokenized DL is a regular language, although it is not characterized by a regular grammar, as can be seen from the simplified grammar of figure 7.1.) The control states are shown as circles, with the start and finish states labelled S and F respectively. A string is accepted as syntactically correct if starting in state S there exists a path to F such that the arc labels taken in the order in which they occur in the path agree with the string. Otherwise the string is considered to have a syntax error.

For any input string no input symbol is examined by the above parsing procedure more than once, hence any input string is parsed in a number of operations linearly proportional to the

```
M    ->  PM' | DM' | WM'
M'   ->  b$ | PM' | DM' | WM'
P    ->  baL
D    ->  badA
W    ->  bawL
L    ->  blSL'
L'   ->  # | blSL'
S    ->  aI | rI
I    ->  # | (X)
X    ->  n | n,X
A    ->  bkaA'
A'   ->  # | bkaA'
```

Non-Terminals	Equivalent In Syntax Diagram
M, M'	PROGRAM
P	PROC
D	DPROC
W	WPROC
L, L'	STAT-LIST
S	STAT
I	ORDER-INFO
A, A'	DLIST
X	SUB-LIST
Terminals	
b	DL
$	END
a	tokenized identifiers
d	tokenized DSTATs
w	tokenized WSTATs
l	tokenized LABELs
#	null string
r	tokenized REG-TRFs
n	tokenized NUMs
k	tokenized BITS => s

Figure 7.1. The Simplified Grammar.

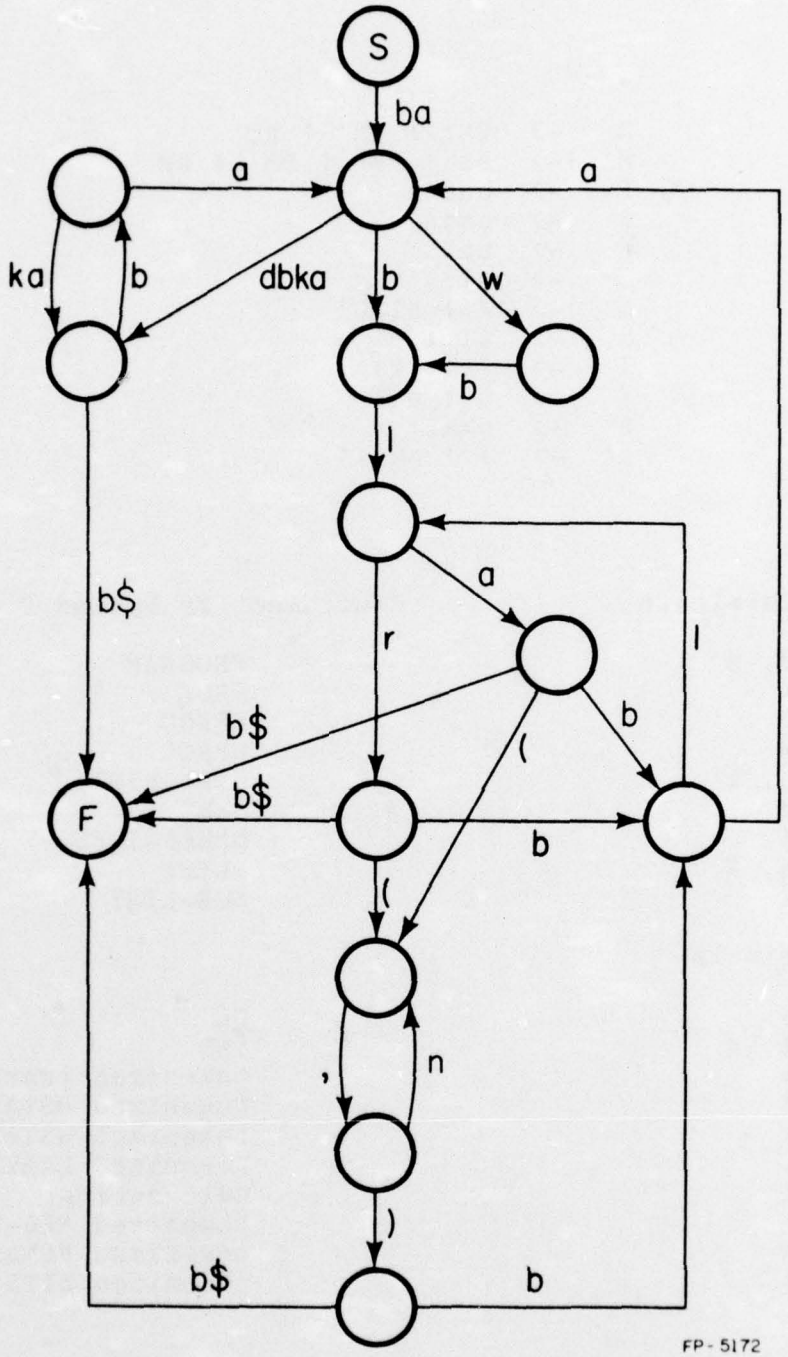


Figure 7.2. The State Diagram Of The Finite Control Of The Finite Automaton That Accepts The DL.

length of the input string. Thus an algorithm for checking any design in the DL for syntactical correctness need not have a complexity of greater than O(n), where n is the number of statements in the design.

It might be argued that any such algorithm also has to tokenize the logical units mentioned earlier, and that this could increase the degree of complexity of the algorithm. As long as there is a fixed upper bound on the number of characters in a token, this is not the case. This is because tokenizing a particular string of characters would not be a function of n, but of the fixed bound.

The conclusions of the above discussion can be embodied in the following theorem.

Theorem 7.1.1: The complexity of an algorithm to check any design in the DL for syntactical correctness need be no greater than O(n), where n is the number of statements in the design.

7.2 The Complexity Of Checking DL Designs For Absence Of Deadlock.

To develop a measure of the complexity of an algorithm that checks designs in the DL for absence of deadlock, some results about networks of modules must first be established.

A network of modules, N, can be viewed as a directed graph G(N). The nodes of the graph are the various modules and the arcs are the interconnecting links. Let DN be the class of networks described by the DL. The following transformation can then be described.

Transformation 7.2.1: Given a network N such that

A. $N \in DN$.
B. G(N) is circuit-free.

Then T is a transformation which when applied to N produces a new network T(N). An example of the transformation is shown in figure 7.3. It is performed at a block level and can therefore be thought of as a transformation on the inter-block matrix. The transformation discards those SR modules associated with the auxiliary inter-block matrix, and replicates those blocks that are used more than once, thus eliminating all the SR modules. T is deadlock preserving: if network N has a potential deadlock, then so will T(N).

It is now possible to take a network N, which is in general composed of So, Si, W, S, J, SR, D and I modules, and if $N \in DN$, convert it to a new network T(N) composed of So, Si, W, S, J, D and I modules which has the same deadlock characteristics. For networks without SR modules a set of necessary and sufficient conditions to ensure deadlock-free operation has been presented in [Bru]. These are as follows:

If N is a network without SR modules it is deadlock-free iff

1. The I modules are two-way articulation points in G(N).
2. The D modules are three-way articulation points in G(N).

When the I, D and Si modules are removed the remaining components are composed of So, W, S and J modules.

3. These components must be circuit-free.
4. Their precedence graphs (see definition 7.2.1 below) must be circuit-free.

Definition 7.2.1: If N is a circuit-free network of So, W, S and J modules, the precedence graph, P(N), associated with N is formed as follows.

Label each S module in N distinctly, 1 thru s. Associate an s component vector P, called the precedence vector, with each output link of N, such that:

$$P = \langle p_1, \ldots, p_s \rangle$$

Where $p = \begin{cases} 0 & \text{if there exists a directed path in G(N) from the primary output of S module i to the output link.} \\ 1 & \text{if there exists a directed path in G(N) from the secondary output of i to the output link.} \\ x & \text{otherwise.} \end{cases}$

Each output link of N, and hence each precedence vector, corresponds to a node in P(N). If a and b are two nodes in P(N) with corresponding precedence vectors P and Q, then there is an arc from a to b, if for

$$P = \langle p_1, \ldots, p_s \rangle$$
$$Q = \langle q_1, \ldots, q_s \rangle$$

there exists an i such that $p_i = 0$ and $q_i = 1$.

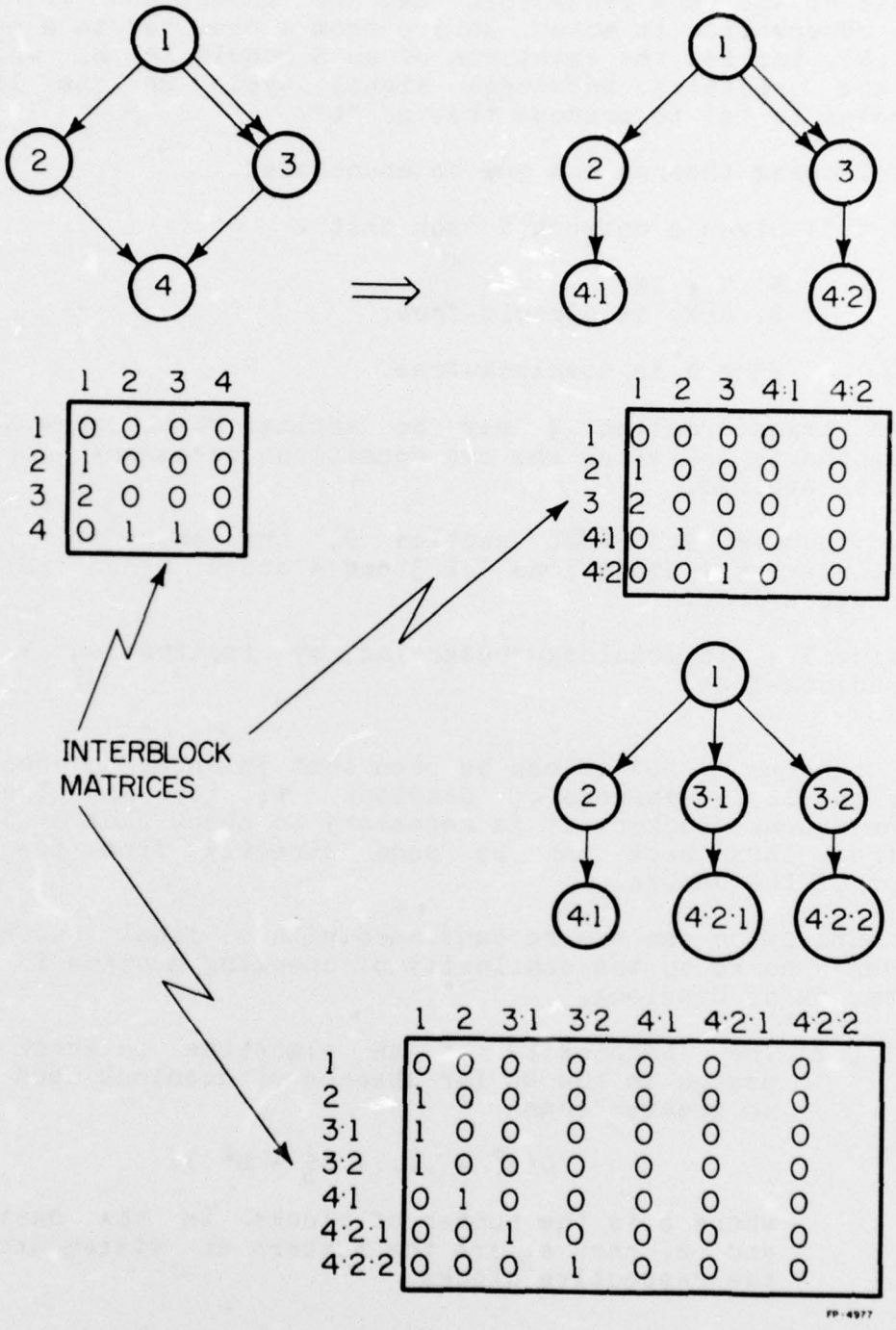


Figure 7.3. An Example Of The Transformation.

The use of the term precedence can be understood if the following observation is noted. An arc from a node "a" to a node "b", in P(N), implies the existence of an S module in N, which causes the request/acknowledge signal cycle on the link corresponding to "a" to precede that on "b".

The following theorem can now be enunciated.

Theorem 7.2.1: Given a network N such that

A. $N \in DN$.
B. G(N) is circuit-free.

Then N is deadlock-free.

Proof: The transformation T may be applied to N, since N by hypothesis satisfies the two conditions necessary for T to be applied.

By theorem 9.1 (see section 9, the appendix) T(N) satisfies the conditions 1,2,3 and 4 above. Hence T(N) is deadlock-free.

Since T(N) is deadlock-preserving, by implication N is deadlock-free.

Q.E.D.

From theorem 7.2.1 it can be seen that in order to check a design in the DL for absence of deadlock, if it has already passed the syntax checker, it is necessary to check that G(N) is circuit-free. This check can be done directly from the DL desciption of the design.

This subsection can now be concluded with a final theorem, which gives a bound on the complexity of checking designs in the DL for absence of deadlock.

Theorem 7.2.2: The complexity of an algorithm to check any design in the DL for absence of deadlock need be no greater than

$$O(s_1^2 + \ldots + s_b^2 + b^2)$$

where b is the number of blocks in the design, and s_1 thru s_b are the numbers of statements in the respective blocks.

Proof: From theorem 7.2.1 it is clear that the check consists of

A. Checking to see if $N \in DN$.
B. Checking to see if G(N) is circuit-free.

From theorem 7.1.1 it is seen that task A need have a complexity no greater than O(n), where n is the number of statements in the design.

From theorem 9.2 (see section 9, the appendix) it is seen that task B need have a complexity no greater than $O(s_1^2 + \ldots + s_b^2 + b^2)$.

Since $n = s_1 + \ldots + s_b$,

then $n \leq s_1^2 + \ldots + s_b^2$

and $O(n + s_1^2 + \ldots + s_b^2) = O(s_1^2 + \ldots + s_b^2)$

Hence the complexity of an algorithm to check any design in the DL for absence of deadlock need be no greater than

$$O(s_1^2 + \ldots + s_b^2 + b^2)$$

Q.E.D.

7.3 The Complexity Of Checking An Arbitrary Network Of Modules For Absence Of Deadlock.

No necessary and sufficient conditions to guarantee absence of potential deadlock exist (as of January 1976) for arbitrary networks of So, Si, W, S, J, SR, D, and I modules. Nevertheless, they do exist (as was seen previously) for subsets of these modules. Theorem 7.3.1 , below, shows that the complexity of an algorithm associated with checking arbitrary networks of the subset of modules, So, W, S, and J, is $O(s.k^2 + m)$, where s is the number of S modules, k is the number of output links, and m is the total number of modules. Hence the complexity of an algorithm to check for absence of deadlock in such networks is at least $O(s.k^2 + m)$ and hence the complexity of an algorithm which performs the check for the complete set of modules is at least $O(s.k^2 + m)$, because it must include networks of So, W, S and J modules as a special case.

Theorem 7.3.1: The complexity of an algorithm to form the

precedence graph for any network, N, composed only of So, W, S, and J modules is $O(s.k^2 + m)$, where s is the number of S modules, k is the number of output links and m is the total number of modules.

Proof: To form P(N)

1. Generate (k) s element precedence vectors. These correspond to nodes in P(N).

2. Compare pair-wise the precedence vectors. This gives the arcs of P(N).

Step 1 involves visiting each arc in G(N) once. None of the modules So, W, S and J has more than two output links, hence the number of arcs in G(N) is O(m). Hence the complexity of step 1 is O(m).

Step 2 involves $s.k.(k - 1)/2$ comparisons. Hence the complexity of step 2 is $O(s.k^2)$.

Hence the combined complexity is $O(s.k^2 + m)$.
Q.E.D.

7.4 Comparing Computational Complexities.

In subsection 7.2 a measure of the complexity of checking the design of a CS for absence of deadlock, if the design is described in the DL, was developed. In subsection 7.3 a similar measure was developed if the design was the result of an unstructured interconnecting of modules from the module set. These are compared in figure 7.4. The values substituted for the parameters in the case with the DL are shown explicitly. The values used in the case without the DL need some explanation.

It is assumed that the design procedures being compared are being used to design the same target system. Then

$$\left.\begin{array}{l}\text{no. output links in}\\ \text{the case without}\\ \text{the DL.}\end{array}\right\} = \left\{\begin{array}{l}\text{no. REG-TRF statements}\\ \text{in the DL case.}\end{array}\right.$$

If it is also assumed that the number of REG-TRFs is a constant fraction of n, the total number of statements. Then

$$k = O(n)$$

With DL: $O(s_1^2 + \ldots + s_b^2 + b^2)$			Without DL: $O(s.k^2 + m)$
$b = 1$	$b = n$	$b = \sqrt{n}$	$k = O(n)$ $s = O(n)$ $m = O(n)$
$O(n^2)$	$O(n^2)$	$O(n^{3/2})$	$O(n^3)$
$O(n)$*	$O(n)$*	$O(n)$*	

$b = 1$ corresponds to a design in the DL, described in one block.
$b = n$ corresponds to a design in the DL, with one statement per block (the complement of the $b = 1$ case).
$b = \sqrt{n}$ corresponds to a design in the DL, with an equal number of statements per block ($s_1 = \ldots = s_b = \sqrt{n}$).

Figure 7.4. A Comparison Of Complexities.

Furthermore, since each statement corresponds to one S module (see the translation procedure in section 4), and both procedures aim at producing behaviorally equivalent CSs, it is assumed that

$$s = O(n)$$

Finally, it is assumed that neither approach is significantly more economical, in terms of the number of modules used. Hence;

$$m = O(n)$$

With these assumptions, it is clear from figure 7.4 that fault free design is easier from a computational complexity viewpoint if the DL approach is used. If the block structure facility is properly used (idealized in figure 7.4 as $\sqrt{n}$ equal blocks of $\sqrt{n}$ statements) the advantage is more pronounced. Furthermore, as can be seen from the assumptions made in proving theorem 9.2 in the appendix, the complexity of an algorithm to

check for absence of circuits in the network of modules associated with the design in the DL is, in most cases, more likely to be $O(s_1 + \ldots + s_b + b)$. Such cases are represented in figure 7.4 by the entries with asterisks. These indicate a further advantage for the DL approach.

In concluding this subsection, a weak point in the preceding arguments should be pointed out. The complexity of the algorithm to form the precedence graph developed in theorem 7.3.1 is not shown to be a lower bound, as it ideally snould be to reinforce the conclusions arrived at when comparing the complexities of the two design procedures. Nevertheless, it should be born in mind that this is only one of many tasks involved in the larger task of checking any network for absence of deadlock, as was noted in section 7.3.

7.5 Comments On The Quantitative Look At The DL.

If reducing the computational complexity of the design task makes it easier for a human to perform the task fault-free, then it can be seen from the preceding sections that the second of the two aims stated in the introduction (to specify a DL so that design faults can be easily avoided) has been achieved.

Deciding whether reducing the computational complexity of the design task makes the task easier for a human cannot be done quantitatively. Nevertheless, there is a consensus that would equate reduced complexity with ease of human performance, if account is taken of how the complexity is reduced. In section 6 it was seen that the reduction of complexity is accomplished by using a DL whose syntax has no facility for unrestricted branching, and which is block structured, so that any design must be formulated in a top down fashion. This maintains a correspondence between the textual description of the CS of the digital system and its proposed operation. Advocates of structured programming consider that this correspondence greatly enhances the fault-free properties of the design procedure viewed as a human task [Mil]. From this point of view the second aim has been achieved.

8. CONCLUSION

The approach to designing digital systems that is promoted in this report can be summarized as follows. Logic gates are organized into asynchronous modules, which in turn are organized into constructs in the DL. At each stage a certain flexibility is given up in order to simplify the design process. Too much flexibility can confuse the design process, too little can limit the class of systems that it is possible to design.

From a more general viewpoint a hierachy of models can be seen:

1. Quantum mechanical theory
2. Circuit theory
3. Switching theory
4. The DL

Each level uses some of the products of the previous lower level as primitives. Selecting which products to use as primitives needs to be done intelligently and with a specific model in view. (E.g. op-amps are a product of level 2, but are useless primitives for switching theory.) If the selection is done judiciously it can result in a simple model, which avoids pitfalls possible in badly specified models. Thus the underlying precept is prevention rather than cure.

In conclusion it can be said that this report presents a model - a DL for digital systems, whose primitives - the asynchronous modules, are products of switching theory (see the list above). Furthermore, and this is the novel point to the report, this model is shown by quantitative arguments to be a better model for design than the ad hoc interconnection of the modules to suit each design problem. The ad hoc interconnection of modules to one another can be thought of as a crude DL in which the step from level 3 to level 4 has been taken thoughtlessly. (This highlights the need for careful specification when proposing a DL.)

9. APPENDIX

Theorem 9.1: Given a network, N, such that

A. $N \in DN$,
B. G(N) is circuit-free.

Then T(N) satisfies the conditions 1,2,3 and 4 of section 7.2 that are necessary and sufficient for absence of deadlock.

Proof (part 1): Assume T(N) violates condition 1 for a particular I module. Then G(T(N)) must be of the form shown in figure 9.1.

Since T(N) has no SR modules, node "a" must represent a J module. This implies, that either there exists a block in T(N) which has a J module at its head, or else there exists a block in T(N) containing an I module which does not head the block.

Transformation T does not alter the intra-block connections. But J modules cannot occur at the head of blocks, and I modules can only occur at the head of blocks, therefore (see section 4.1) the initial assumption must be false.

Hence T(N) satisfies condition 1.

Proof (part 2): Similarly it can be shown that, as long as N satisfies the hypotheses A and B, T(N) satisfies condition 2.

Proof (part 3): Condition 3 is satisfied by hypothesis B.

Proof (part 4): Removing the I, D and Si modules from T(N) leaves neworks of So, W, S and J modules.

Consider one of these networks. Its directed graph representation can be viewed as a tree whose nodes are blocks of W, S and J modules. The top of the tree may be capped by an So module. Figure 9.2 depicts such a tree. The arcs out of each node represent output links of the network (designated leaves in the figure) or links to other blocks of W, S and J modules.

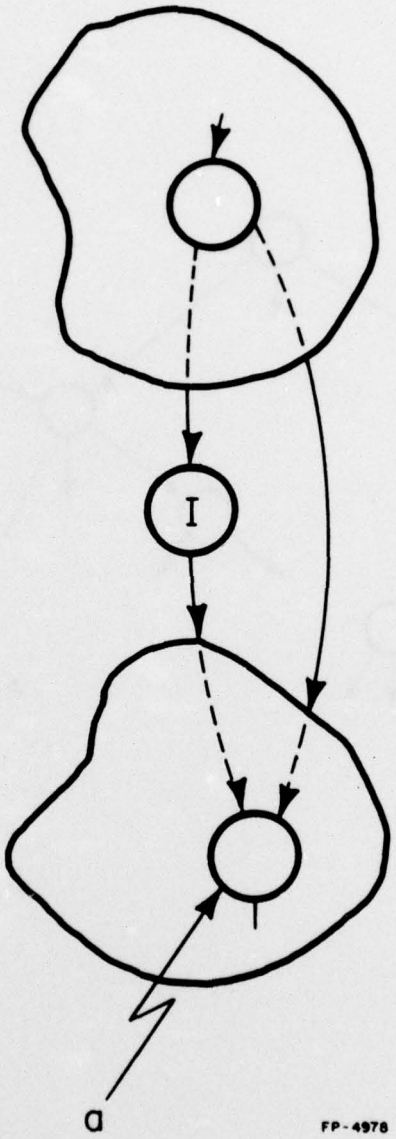


Figure 9.1. A G(T(N)) In Which Condition 1 Is Violated.

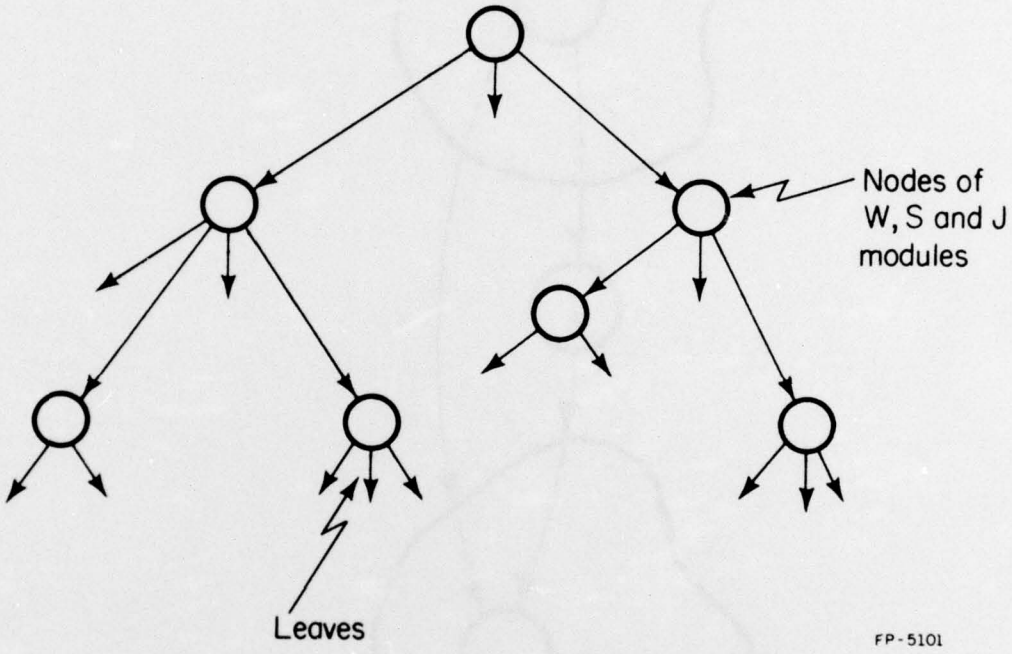


Figure 9.2. A Tree Of Blocks Of W, S And J Modules.

As was seen in section 4.1, these blocks have only a single input link and no intra-block connections involve the primary output links of S modules. This implies that the precedence of the arcs out of a node representing such a block, form a partial ordering.

Starting at the root of the tree a partial ordering can be formed among the links out of the root. This can then be expanded to incoperate the partial ordering among the links of each one of the successor nodes. Continuing down the tree in this fashion yields a partial ordering representing the precedence relationship between the output links or leaves of the tree.

Since a partial ordering is circuit-free, condition 4 is satisfied by T(N).

Q.E.D.

Theorem 9.2: The complexity of an algorithm to check whether the modular realization of a design in the DL is circuit-free need be no greater than

$$O(s_1^2 + \ldots + s_b^2 + b^2)$$

where b is the number of blocks in the design, and s_1 thru s_b are the number of statements in the respective blocks.

Proof: The modular realization of a design is circuit-free if the realization of each block is circuit-free and if the inter-block connections are also circuit-free.

For each block the order information can be viewed as a directed graph represented as an adjacency structure (recall the interpretation "successor of" used in section 4). The modular realization of each block is circuit-free if this graph is circuit-free. Johnson [Joh] presents an algorithm which finds all the elementary circuits of such a graph. The complexity of the algorithm is $O((v + e)(c + 1))$, where v is the number of vertices, e is the number of edges and c is the number of circuits.

For a block having s statements the worst case is given by

$v = s$

$e = s^2/4$ (The unlikely case of the order information describing a bipartite graph.)

$c = 0$ (The algorithm is being used to confirm that the design is circuit free.)

This yields a complexity for the single block case of no greater than $O(s^2)$.

The inter-block connections are defined by the way blocks "call" each other and are explicitly denoted by the block statements. This can also be viewed as a directed graph represented as an adjacency structure. If the design has b blocks, the above reasoning yields a complexity for the inter-block case of no greater than $O(b^2)$.

Hence, for a design having b blocks with s_1 thru s_b statements, the complexity of an algorithm to check whether the modular realization is circuit-free need be no greater than

$$O(s_1^2 + \ldots + s_b^2 + b^2)$$

Q.E.D.

Note that in most cases, for a block having s statements e is more likely to be closer to $O(s)$ than $O(s^2)$. Similarly for a design having b blocks, e is more likely to be closer to $O(b)$ than $O(b^2)$.

10. REFERENCES

Alt Altman, S. M., and P. J. Denning, "Decomposition of Control Networks", Rec. of the Proj. MAC Conf. on Concurrent Systems and Parallel Computation, pp. 81-92, 1970.

Bar Barbacci, M. R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE TC, Vol. c-24, No. 2, pp. 137-150, Feb. 75.

Bel Bell, C. G., and A. Newell, Computer Structures: Readings and Examples. , McGraw-Hill, New York, 71.

Bru Bruno, J., and S. Altman, "A Theory of Asynchronous Control Networks", IEEE TC, Vol. c-20, No. 6, pp. 629-638, Jun. 71.

Chu Chu, Y., "Introducing the Computer Design Language", Proc. IEEE Comp. Conf., COMPCON 72, pp. 215-218, Calif., Sept. 72.

Cla Clark, M. A.,"Macromodular Computer Systems", Compiler", Proc. Spring Joint Comp. Conf., pp. 335-401, 67.

Com Computer Magazine, IEEE Comp. Soc., Dec. 74.

Den Dennis, J. B., "Modular, Asynchronous Control Structures for a High Performance Processor", Rec. of the Proj. MAC Conf. on Concurrent Systems and Parallel Computation, pp. 55-80, 1970.

Dul Duley, J. R., and D. L. Dietmeyer, "A Digital System Design Language (DDL)", IEEE TC, Vol. c-17, No. 9, pp. 850-861, Sept. 68.

Fri Friedman, T. D., "ALERT: A Program to Compile Logic Designs of New Computers", Digest 1st Annual IEEE Comp. Conf., pp. 128-130, Sept. 67.

Gla Glaser, E. L., "Introduction and Overview of the LOGOS Project", Proc. IEEE Comp. Conf., COMPCON 72, pp. 175-177, Calif., Sept. 72.

Hab Habermann, A. N., "Prevention of System Deadlock", Comm. ACM, Vol. 12, No. 7, pp. 373-377, Jul. 69.

.ol Holt, A. W., and F. Commoner, "Events and Conditions", Rec. of the Proj. MAC Conf. on Concurrent Systems and Parallel Computation, pp. 3-52, 1970.

Jon Johnson, D. B.,"Finding all the Elementary Circuits of a Directed Graph", SIAM J. Comput. , Vol. 4, No. 1, pp. 77-84, Mar. 75.

Jum Jump, J., "Asynchronous Control Arrays", IEEE TC, Vol. c-23, No. 10, pp. 1020-1029, Oct. 74.

Kel Keller, R. M.,"Towards a Theory of Universal Speed-Independent Modules", IEEE TC, Vol. c-23, No. 1, pp. 21-33, Jan. 74.

Met Metze, G., and Seshu, "A Proposal for a Computer Compiler", Proc. Spring Joint Comp. Conf., pp. 121-129, 62.

Mil Mills, H. , "Mathematical Foundation For Structured Programming", FSC72-6012, Federal Systems Division, IBM Corp., Gaithersburg, Md. , Feb. 72.

Mud Mudge, T., "Specifying a Design Language for Digital Systems", Proc. 13-th Annual Allerton Conf. on Circuit and System Theory, pp. 905-915, Oct. 75.

Pat 72 Patil, S. S., and J. B. Dennis, "The Description and Realization of Digital Systems", Proc. IEEE Comp. Conf., COMPCON 72, pp. 313-316, Calif., Sept. 72.

Pat 75 Patil, S. S., "An Asynchronous Logic Array", Comp. Structures Group Memo 111-1, Proj. MAC, MIT, Feb. 75.

Pet Peterson, J. B.,"On High Level Digital System Design", Rept. R-653, CSL, Jul. 74.

Sha Shaw, A. C., The Logical Design of Operating Systems. , Prentice-Hall, New Jersey, 74.

Ung Unger, S. H.,Asynchronous Sequential Switching Circuits. , Wiley, New York, 70.