https://ntrs.nasa.gov/search.jsp?R=19810024343 2020-03-21T11:20:56+00:00Z

General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)

DIGITAL SYSTEMS DESIGN LANGUAGE

(NASA-CR-161332) DIGITAL SYSTEMS DESIGN
LANGUAGE Annual Report, 1 Oct. 1978 - 30
Sep. 1979 [illegible]
[illegible] (Alabama Univ. in Huntsville.) 131 p Unclas
G4/61 [illegible]

Prepared by

SAJJAN G. SHIVA
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama 35807

First Annual Technical Report
October 1979

(NASA-CR-161332) DIGITAL SYSTEMS DESIGN N81-32880
LANGUAGE Annual Technical Report (Alabama
Univ. in Huntsville.) 131 p HC A07/MF A01
CSCL 09B Unclas
G3/61 37452

RECEIVED
NASA STI FACILITY
ACCESS DEPT.

The University
Of Alabama
In Huntsville

DIGITAL SYSTEMS DESIGN LANGUAGE

Prepared by

SAJJAN G. SHIVA
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama 35807

First Annual Technical Report
October 1979

for

NAS 8 - 33096
DESIGN SYNTHESES OF DIGITAL SYSTEMS
George C. Marshall Space Flight Center
Alabama, 35812

The University
Of Alabama
In Huntsville

FOREWORD

This is a technical summary of the research work conducted during October 1, 1978 to September 30, 1979 by The University of Alabama in Huntsville towards the fulfillment of the Contract NAS8-33096 from the George C. Marshall Space Flight Center, Alabama. The NASA technical officer for this contract is Mr. Robert E. Jones.

The author greatfully acknowledges the numerous discussions with and helpful comments of Mr. John M. Gould during this research work, and thanks Professor Donald Dietmeyer of the University of Wisconsin-Madison for providing the DDL Software.

DIGITAL SYSTEMS DESIGN LANGUAGE

Sajjan G. Shiva

ABSTRACT

Digital Systems Design Language (DDL) has been implemented on the SEL-32 Computer Systems of the Electronics and Controls Laboratory. This document provides the details of the language; the translator and the simulator programs. Several example descriptions and a tutorial on hardware description languages are provided, to guide the user.

TABLE OF CONTENTS

LIST OF TABLES------VI

LIST OF FIGURES------VI

1. INTRODUCTION------1

2. THE LANGUAGE (DDL)------2

2.1 Syntax Rules------5

2.2 Declaration Statements------5

2.3 Operations------8

2.4 If-Value Clause------10

2.5 Identifier------11

2.6 Operator Declarations------11

2.7 State Declaration------13

2.8 Automaton and System Declarations------14

3. THE TRANSLATOR (DDLTRN)------19

3.1 The Translation Process------20

3.2 An Example------23

4. THE SIMULATOR (DDLSIM)------31

4.1 Simulation Models------32

4.2 Simul.tor Command Language------37

4.3 Simuiation Algorithm------61

4.4 Errors------64

5. EXAMPLES

Example 1: A Serial Twos Complementer------66

Example 2: The Serial Twos Complementer (variation 1)------77

Example 3: Twos Complementer (variation 2)------84

Example 4: Multiplier------94

Example 5: Minicomputer------99

6. CONCLUSIONS--102

APPENDIX

LIST OF TABLES

2.1 OPERATORS--18

3.1 FLAG INTECERS--20

LIST OF FIGURES

2.1 Local and Global Facilities----------------------------------4

5.1 A Serial Twos Complementer----------------------------------70

5.2 The Serial Twos Complementer (variation 1)------------------78

5.3 The Serial Twos Complementer (variation 2)------------------85

5.4 Multiplier--96

5.5 Minicomputer--101

1. INTRODUCTION

Hardware Description Languages (HDL) provide a convenient medium of inputting the design details into a design automation system. This report gives the details of one such language, Digital Systems Design Language (DDL), selected for integration into the Computer Aided Design and Test System (CADAT) of the Electronics and Controls Laboratory.

Chapter 2 provides the language details, Chapter 3 discusses the translator program and Chapter 4 discusses the Simulator Program. Some example descriptions are provided in Chapter 5. A tutorial on Hardware Description Languages is provided in the Appendix. An exhaustive bibliography for some of the literature in this area is also provided in the Appendix. Readers not familiar with any HDL are referred to the Appendix before reading the rest of the report.

The Simulator and Translator Programs are currently being tested on SEL-32 Computer System and hence, the complete deck set up details for the use of these programs is not included in this manual.

2. THE LANGUAGE [31]*

DDL was introduced in 1967 by Duley and Dietmeyer [33]. A translator and a simulator are written for a subset of this language in IFTRAN, an extended version of FORTRAN [35,36]. These programs are implemented in FORTRAN on SEL 32 Computer System. The translator (DDLTRN) translates a DDL description into a set of Boolean equations and register-transfer statements. The simulator (DDLSIM) enables the system designer to verify his design. The output of the translator is an input to the simulator. Simulation parameters are to be input by the designer. In DDL the structural elements are explicitly declared. At the lower level of description, functional and structural elements correspond directly to the actual elements of the system. DDL is highly suitable for describing the system at the gate, register transfer and major combinational block level.

The logical statements can be formed using the available primitive operators. The functional specification of the system consists of these logical statements, in blocks. The statements describe the state transitions of a finite state machine controlling the processes of the intended algorithm. The block then appears as an automaton.

Parallel operations are permitted. Synchronous behavior is described by either identifying the pulses or by including delay elements described in terms of multiples of clock pulses. Asynchronous behavior is modelled by using conditional statements. Data paths can be explicitly declared by using terminal declarations.

---

*The numbers in brackets point to the references listed in the Appendix.

DDL is a "block-oriented" language; the blocks of a DDL description usually correspond to natural divisions (blocks) of the hardware being described. Thus a computer may have a major block called an "ALU," which contains a block called "adder," which consists of interconnected logic blocks called "full-adders." This nested view of the hardware can be directly reflected in the DDL description of the computer.

Both facility declarations and operations can appear within the body of the more complex declarations that have a heading part. Identifiers declared within such complex declarations are said to be local facilities of that declaration, and are global facilities of complex declarations that appear in the body of the encompassing declaration. Other complex declarations that parallel the encompassing declaration cannot control or sense such facilities. Operations can reference only facilities that are local or global to the block in which they appear. Thus the same identifier may be declared in more than one parallel block without ambiguity.

Figure 2-1 illustrates some of the possibilities. Facilities A, B, and C are declared facilities of the overall block named SYSTEM. These facilities are global to all blocks within SYSTEM; any or all of these blocks may control or sense the states of facilities A, B, and C. Hence A, B, and C are said to be public facilities. Facilities D and E are local to SUBSYSTEM 1, global to PART 1 and PART 2. SUBSYSTEM 2 and its inner blocks are not aware that facilities D and E exist; no reference to D and E may appear in the description of SUBSYSTEM 2.

Facilities H and I are local to PART 1; no other block of Fig. 2-1 may control or sense these facilities. PART 2 has its own facility I which may be of a very different hardware nature than facility I of

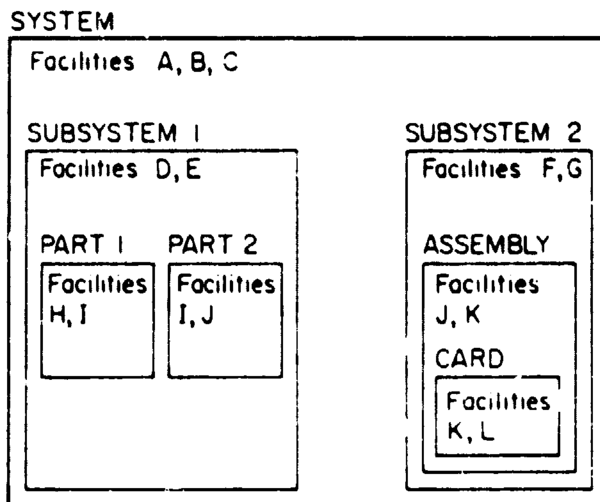


Fig. 2-1. Local and global facilities.

PART 1. PART 1 and PART 2 each control and sense their own facility I.

Similarly, PART 2 controls and senses its local facility J as does ASSEMBLY for its local facility J, which is global to CARD and hence can be controlled and sensed by CARD. References to K within CARD pertain to the most locally declared facility K, e.g., the one declared within CARD.

Permitting the same identifier to be used in parallel blocks allows designers working in parallel on the blocks to select without restriction names that appeal to them. If parallel blocks must communicate, facilities global to them must be established and assigned unique names. The designers of the parallel blocks must know and use these global names. Thus in Fig. 2-1 SUBSYSTEM 1 and SUBSYSTEM 2 may communicate via A, B, or C. PART 1 and PART 2 may communicate via D or E, or via A, B, or C.

2.1 SYNTAX RULES

VARIABLES:

Variable name may contain 1 to 8 characters, the first of which must be alphabetic. The remaining characters must be letters or digits.

Examples: MULT

SYS1

COMPLMNT

CONSTANTS:

Constants take the general form nRk. n is the number in base R (R=D for decimal, O for octal). k is the number of bits required for the representation, k ≤ 32. k is decimal.

Examples:

Representation	Binary equivalant
1D2	01
5D4	0101
20D5	10100
2O3	010
20O6	010000
0	0
1	1

2.2 DECLARATION STATEMENTS

The general format of a declaration statement is <DT> body.

The declaration type (device) is enclosed in angle brackets and the period terminates the declaration. Body consists of a list of items separated by commas. Following devices are allowed:

TErminal	Sets of wires
REgisters	Sets of synchronized flip-flops
MEmory	Sets of synchronized flip-flops
LAtches	Sets of asynchronous latches
TIme	Clock
DElay	Delay elements
BOolean	Combinational logic
ELement	Off-the-shelf components

The device type can be abbreviated to the first two characters.

Examples:

<TE> X, Y(4), Z(0:2), W(3,4:1), A(12) = B ¢ C(0:10) identifies a single wire X, four wires Y_1, Y_2, Y_3, Y_4 with Y_1 on the left, 3 wires Z_0, Z_1, Z_2 and 12 wires corresponding to W, placed in 3 rows, ith row of wires numbered W_{i4}, W_{i3}, W_{i2}, W_{i1}. The subscripts always have a left to right interpretation. A single subscript n indicates the range 1 to n while a range n:m indicates n to m left to right. In the above declaration, A_1 is also named B, A(2:12) are named C(0:10). ¢ is the concatenation operator. The concatenation of B and C is a 12 bit terminal A with the most significant bit same as that of B and the least significant 11 bits same as those of C.

<u>REgister and LAtch DECLARATIONS</u>

<RE> IR(16) = OP(0:3) ¢ IX(1:3) ¢ ADRS(9), X(12). declares a 16 bit register IR and a 12 bit register X.

IR is identified with 3 subregisters OP, IX and ADRS.

<LA> BUF(4).

declares a set of 4 latches BUF.

<RE> A(8).

declares an 8 bit register, bits numbered from 1 to 8, left to right.

MEmory DECLARATION

<ME> M(X:Y).

declares X words (numbered from 0 to X-1) of Y bits each (numbered 1 through Y).

<ME> MP(256:8).

declares a 256 word memory, 8 bits/word.

References to the memory must be of the form M(MAR) where MAR is the same register in all references to M. MAR is declared in a RE declaration. Only full words may be accessed from memories.

TIme DECLARATION

<TI> A(1E-6), Q(20E-9) 2.

declares a single phase clock A with a 1 microsecond period and a two-phase clock Q with 20 nanosecond period.

<TI> P.

declares a single phase clock with an arbitrary time period (unit).

DElay DECLARATION

<DE> P(10E-9), Q(5E-7).

declares two delays P with 10 nanoseconds and Q with .5 microsecond. The context in which the DElay element is referenced determines whether its input or output terminal is used.

BOolean DECLARATION

<BO> Identifier = Boolean expression.

Examples:

<TE> A, B(5), C(0:4), D(6, 5:1).

<BO> D(4) = B+C, D(5) = A*B.

declares that the fourth row of D is formed by ORing terminals B and C i.e. ($D_{45} = B_1 + C_0$ etc.) bit by bit; the fifth row of D is a bit by bit AND of A and B. Since A is 1 wire and B is a set of 5 wires, A is fanned out to combine with each bit of B.

ELement DECLARATION

Enables the description of an element in the system whose logical specifications are unknown or impertinent.

For example,

<EL> JKFF (Q1,NQ1: C, J1, K1), COUNT (K(5:1), ZERO:
UPDWN, CLK).

declares an element JKFF with 3 inputs C,J1,K1 and two output Q1 and NQ1; and an element COUNT with two inputs and 6 outputs. The only information available on these black boxes is the input/output terminals.

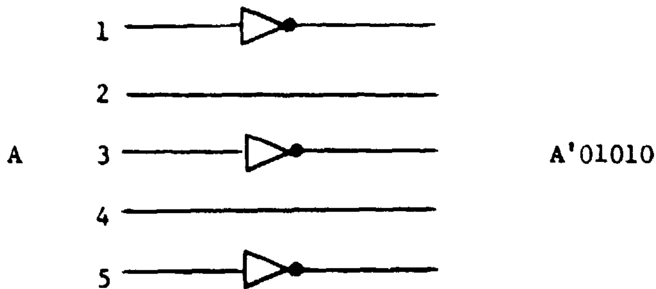
2.3 OPERATIONS

Table 2.1(a) shows the operations allowed and their hierarchy; Table 2.1(b) shows three special operators. "=" is used to show the connections while <- indicates a data transfer from one facility to the other -> is equivalent to a "GOTO', used to show a state transition.

The extension operator "$" creates k copies of the terminal or terminal set offered as its left operand.

The selection operator ', selectively complements, or not complements the bits of the facility (left hand operand) depending on the value of the corresponding bit in kDn is a 0 or 1.

For example A' 10D5 is equivalent to

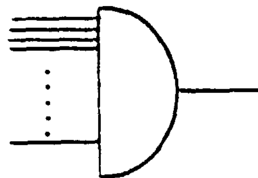


A'01010

The operator preceding the reduction operator (/) determines the nature of the reduction on the right hand operand of /. Six types of reductions are possible. For example, given a signal A,

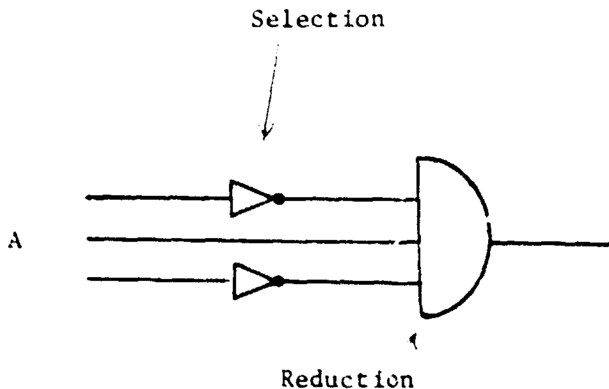
*/A implies

A



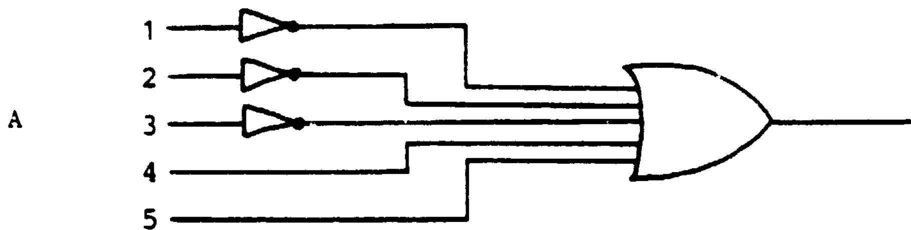
If A is a 3 bit signal,

*/A' 2D3 implies



+/A'3D5 implies

A



Boolean expressions (Be) can be formed by using the operators and variables in the usual manner. Paranthesis could be used where there is an ambiguity. The expressions are evaluated from left to right following the operator hierarchy.

Conditional operations have th_ format

!BE! OP_1. or

!BE! OP_1 ;OP_2.

The first form implies: If the value of BE is 1, perform OP_1; the second form implies: If BE is 1, perform OP_1 else perform OP_2. "If ... then" operations can be nested:

! A ! ! B ! OP_1.; ! C ! OP_2..

2.4 IF - VALUE CLAUSE

"¦" is used for "IF" and "#va" is used for the value in an IF-value clause. For example;

B = ¦ C #0 D0 #1 D1 #2 D2.

implies that D0 is connected to B if the value of C is 0, D1 is connected to B if the value of C is 1, etc.

As another example,

¦X #0D2 A<-B #1D2 A<-C #2D2 A<-AB #3D2 A<-AC.

describes a 4 way conditional transfer operation into A depending on the

value of X.

2.5 IDENTIFIER

IDentifier declaration enables the naming of a group of operations so that they do not have to be written repeatedly (equivalent to MACROs). The general format of IDentifier declaration is,

<ID> list.

where list takes the form

id = compound facility

id = (CSOP)

For example, <ID> X = C(2:10) ¢1. names the compound facility ¢(2:10)¢1 to be X. Then, any reference to X is expanded into C(2:10)¢.
For example, S = R ⊕ X. is equivalent to S = R ⊕ C (2:10) ¢1.

(A compatible set of operations (CSOP) is a set of operations separated by commas. It must be possible for the hardware to perform all these operations simultaneously.)

The order in which the operations are listed is of no consequence. For example,

<ID> A = (Y <- X, Z <- Z(2:5) ¢ΛZ(1)),

B = (Y <- X, Z<- Y).

names two CSOPS. Note that the operations Y <- X and Z <- Y in B are simultaneous and are compatible.

2.6 OPERATOR DECLARATION

Blocks of combinational circuitry can be defined with the OPerator declaration. The body of the OPerator declaration consists of a BOolean declaration and perhaps a TErminal declaration. Boolean equations in the body of the BOolean declaration include Boolean expressions which

may involve conditions and be relatively complex. References in these Boolean equations may be made to (1) facilities global to the OPerator declaration. (2) local terminals declared within the OPerator declaration by a TErminal delcaration, and (3) terminals delcared and dimensioned in the head of the OPerator declaration. The TErminal declaration may be used to define local terminals of the operator, and must be used to dimension "dummy" identifiers listed in the heading, if any.

The head of the Operator declaration consists of one or a list (separated by commas) of identifiers with or without an argument list enclosed in $s, with or without parenthetic subscript ranges. Permitted syntactic forms for heads are:

$$id_1,\ id_2(i_2),\ id_3\ \$\ X_1,\ X_2,\ldots X_k\$,\ id_4\ (i_4)\$$$

$$X_1,\ X_2\ldots\ X_k\$$$

where subscript ranges can also be placed within the parenthesis. The identifiers name the combinational logic blocks and their output terminals. Parenthetic integers dimension the output terminal sets with the same syntax and semantics as in TErminal declarations. The arguments are local dummy identifiers of input terminals of the combinational blocks. Such dummy identifiers must be dimensioned via a local terminal declaration within the OPerator body.

As an example of a time-shared operator block, ALU is declared below. This combinational block is able to add two 16-bit binary sequences presented to it on lines X and Y or form their bit-by-bit EXCLUSIVE-OR. Input signal F determines which task is performed. The carry into rightmost full-adder must also be presented to the unit.

```
<OP> ALU(16) $ X,Y, CIN, F$
<TE> X(16), Y(16), CIN, F, C(16) = CX¢CC(15).
```

```
<BO> C=X*Y + CC¢ CIN* (X+Y),

ALU = (!F! X@Y@ CC¢CIN; X@Y•)..(end of BO, end of OP)
```

Note the inline comment capability of DDL (end of BO, end of OP).

Suppose the following declaration is global to ALU,

```
<RE> ACC(16), MBR(16), COUNT (12).
```

we can define several operations using ALU as following:

```
!LDA!  ACC <-  ALU$O,MBR,O,O$

!ADD!  ACC <-  ALU$ACC,MBR,O,1$

!SUB!  ACC <-  ALU$ACC,ΛMBR,1,1$

!KNT!  COUNT<-ALU(5:16) $OD4¢COUNT,O,1,1$

!XOR!  ACC <-ALU$ACC,MBR,O,O$
```

2.7 STATE DECLARATION

DDL views the operation sequencing (control) circuitry as a finite state machine. Each state (step) of the control circuitry is described by a STate declaration:

```
<ST>State List.
```

State list consists of a list of state statements (without separating commas). Each state statement has one of the following forms:

```
Sid (n): csop.

Sid (n): Be: csop.
```

Sid is a simple unsubscripted identifier. n is the decimal state assignment.csops include the state change operations using the state transition operator ->.

In the first form, csop is performed whenever the automaton is in the state Sid.

In the second form, csop is performed when the automaton is in Sid and also Be is satisfied. The automaton waits in the state till Be is

satisfied.

A 15 bit multiplier control can be described as following:

```
<ST>   S0(0):MPY:ACC<-0, CNT<-15D4,->S1.
       S1(1):->S2, DECR$ CNT$ !Q(15) ! ACC<-ACC+R..
       S2(2):SHR$ACC¢Q$, !+/CNT! ->S1;S0 ...
       (end of conditional, end of S2, end of ST)
```

SHR is shift right (zero fill) operator and DECR is a decrement operator assumed to be defined using <OP> declaration.

2.8. AUTOMATON and SYSTEM DECLARATIONS

Relatively independent disjoint portions of a digital system are identified as automata in DDL with syntax.

<AU> head body.

The AUtomaton declaration is the most complex type of declaration of DDL. Its head may take any of four forms, for example;

auid:

auid:csop

auid:Be:

auid:Be:csop

First, an automaton identifier, auid, may be subscripted, but may not include parenthetical arguments; it names the block only. A compatible set of operations may be included in the head of an automaton. These operations are to be performed whenever the Be of the heading, if any, is satisfied. Conditional as well as unconditional operations may be included in this heading csop, so whether a specific operation is performed or not may depend on conditions throughout the automaton or system.

Be in the heading of the AUtomaton declaration is a condition on

all operations declared throughout the body of the declaration except connection operations. Usually Be is the clock signal that synchronizes the automaton. It is generally unnecessary and undesirable to include such global conditions as clock signals in combinational circuits; in fact, signal propagation in combinational networks usually precedes clock pulses. If a clock with n phases is used to synchronize an automaton, then a dimensional Be or a concatenation of n Bes appears in place of the single Be in the AUtomaton declaration head.

The body of an AUtomaton declaration consists of other declarations. Each of these declarations is terminated with its own period; punctuation is not placed between them. The following declaration types may appear:

<ME>, <RE>,<LA>, <TE>

<TI>, <DE>, <OP>,<EL>,<ID>,<BO>,<ST>

ME, RE, LA, TE, TI, DE, AND EL declarations are used to declare the existence of local facilities of the automaton. The OPerator and BOolean declarations specify combinational blocks and interconnections of facilities. The IDentifier declaration may be used to simplify or clarify the overall AUtomaton declaration. The STate declaration is usually used to specify the operations of the automaton. If the STate declaration is not used, then all operations appear in the csop of the AUtomaton declaration head.

The <u>SYstem declaration</u> has syntax identical to the AUtomaton declaration. The system is identified in the head. Global conditions and csop may be specified also. The body of a SYstem declaration may contain AUtomaton declarations as well as all other types of declarations, but STate declarations must appear within AUtomaton declarations. Public facilities are declared with ME, RE, TE, etc., declarations outside of all

AUtomaton or OPerator declarations.

Example:

A multiplier controller is described below to illustrate the SYstem and AUtomaton facilities. The counter is treated as a separate automaton. Perhaps other unspecified automaton of SYSTEM 1 can use the counter when automaton MC is not.

```
<SY> SYSTEM 1:
     <RE> ACC(15), Q(15), R(15).
     <TE> SET, DEC, DONE, MPY.
     <TI> P(1E-7).
     <AU> CPU: P:
       <ST> .
            .
            .
          Q17: DONE: Q <- Multiplier,
            .         R <- Multiplicand, MPY = 1.
            .
           .. (end CPU)
     <AU> MC: P:
       <ST> S0: MPY: ACC <- 0, SET = 1,-> S1.
            S1: -> S2, DEC = 1,!Q (15)! ACC <- ACC+R..
            S2: ACC¢Q <- SHR$ACC¢Q$ !DONE! -> S1 ...
     <AU> K: P:
       <ST> [i=1:15]  T(i): DEC: ->T(i-1)..
         T(0): DONE = 1, !SET! -> T(15); -> T(0)...
         (end SY)
```

Automaton CPU is shown only as placing the multiplier and multiplicand in public registers and issuing command MPY to multiplier control MC. If the counter automaton K is idle, it will be issuing DONE = 1. CPU waits in its state Q17 until this condition is satisfied (perhaps K is still doing a job for some other automaton). MC clears ACC, but the counter is initialized by SET = 1. Specifically SET = 1 will cause K to go from its state T(0) to T(15) where it will remain until it is told to decrement via public terminal DEC. MC tests the multiplier, adds or not and shifts repeatedly until it is informed by K via public terminal DONE that all multiplier bits have been examined. In the example above interacting automata MC and K operate in parallel.

NOTE: The "For clause" shown in the Automaton K for the decrement operation [i=1;15] T(i):DEC: -> T(i-1) is not allowed in the present version of the DDL software. This statement has to be broken up into;

```
T(1): DEC: ->T(0)
T(2): DEC: ->T(1)
      .
      .
T(15):DEC:->T(14)
```

SHR is a single argument operator (assumed to be declared earlier) that shifts the argument one bit right, and fills zero on the left.

TABLE 2.1(a) : OPERATORS

OPERATOR	SYMBOL	TYPICAL SYNTAX	COMMENTS
Extension	$	A$k	k copies of A
Concatenation	¢	A¢B	Bit by bit
Complementation	Λ	ΛA	Bit by bit complement
Selection	'	A'kDn	Selective complementation
Reduction	/	p/A	$A_1pA_2p\ldots pA_n$, where p is one of these:*, Λ*, Λ+,Λ@,@,+.
AND	*	A*B	Bit by bit
NAND	Λ*	AΛ*B	Operations
NOR	Λ+	AΛ+B	
XNOR	Λ@	AΛ@B	
XOR	@	A@B	
OR	+	A+B	

TABLE 2.1(b) : SPECIAL OPERATORS

CONNECTION	=
TRANSFER	<-
GO TO	->

NOTE: <u>Refer to Chapter 3 (The Translator) for variations of these Operators</u>.

3. THE TRANSLATOR (DDLTRN) [36]

DDLTRN is a program that translates a DDL description of a digital system to 1) a DDL description that consists of Boolean equations and register transfer statements in the heading of a system declaration only, and 2) a tablation of facilities and subfacilities declared in the DDL description and/or defined in the translation process. Some modifications of DDL recognized by DDLTRN are listed below. The translation process is briefly discussed and illustrated in Section 3.1.

1) The following operators are changed to accomodate the SEL-32 peripherals:

DDL Operator		Key Punch	CRT Terminal	Printer
Concatenation	¢	¢	[	[
Complement	∧	⌐	∧	↑
IF - THEN	!	!	]	]
IF - VALUE	¦	¦	!	!

The other operators of DDL are compatible with the peripherals of SEL-32 and remain the same.

2) COmment declarations end with a left angle bracket<.

3) Values in "If-value" clauses are limited to a single integer values. Ranges, lists and else (;) values are not permitted.

4) Concatenation operands must be simple facilities with or without subscripts, or binary strings.

5) State assignments are specified in decimal following the state identifier of each state statement, e.g., "S1(2):..."

6) Automata names are used as state sequencing register names and thus should be dimensioned in the <AU> declaration head, e.g., "<AU> CPU (5): P:..."

7) DDLTRN accepts FLag declarations with syntax: <FLag> list. List consists of integers, and/or integers preceded by the complement symbol (Λ), separated by commas. Each integer specifies the setting of a flag. Each complemented integer specifies that the corresponding flag is to be reset. Table 3.1 summarizes the significance of set flags and the default states of the flags.

8) Identifiers defined in IDentifier declarations must not be subscripted.

TABLE 3.1 FLAG INTEGERS

Flag	Significance	Default
1	Print Source Card Images	Set
2	Print Declared Facilities and Operations	"
3	Print DDL string after Pass 2	Reset
4	" " " " " 3	"
5	" " " " " 4	"
6	" " " " " 5	"
7	" " " " " 6	Set
8	Print F Table after Last Pass	Reset
9	Print Encoded string after Last Pass	"
10	Execute through Pass 2 only	"
11	" " " 3 "	"
12	" " " 4 "	"
13	" " " 5 "	"
14	" " " 6 "	Set

3.1. THE TRANSLATION PROCESS

DDLTRN is the result of a research effort to develop efficient language translation algorithms. As a result it emphasizes translation

efficiency rather than error detection and control. Neither the syntax of supplied DDL descriptions nor the translation process itself are checked in detail.

A DDL description is stored as a single string in a singly linked list in memory. Operator and punctuation symbols are represented by codes. As processing proceeds facility names and subscript ranges are also encoded to shorten the string and hence the time required to pass over it.

Facts about declared facilities such as name, subscript range, type, etc. are recorded in a facility table F. Translation consists of passing over the DDL string a number of times. With each pass the DDL string and F table are modified according to unique rules. Six main passes may be identified by the user: The DDL string and F table may be printed after any of these main passes.

Pass 1 -- Facilities Identified

Data cards bearing a DDL description are read and echo printed. All blank columns are ignored; all card columns 1 - 80 are examined. Declared facilities are entered in the F table. TIme, REgister, MEmory, LAtch, TErminal and DElay declarations are removed from the DDL description, as are all COmment declarations and parenthesized comments. Identical primary names declared in nested or parallel blocks are made unique by appending a double quote (") and integer. Identical names declared in the same block are rejected, of course.

Pass 2 -- Syntax Reduced

Names and binary strings in connection and register transfer operations are encoded. Secondary names (names appearing on the right of an equal sign in a TErminal, REgister, etc. declaration) are replaced with

their subscripted primary name equivalents. Identifiers from IDentifier declarations are replaced in operations and expressions serving as conditions on operations with the symbol string they represent. The syntax of OPerator, BOolean and STate declarations is removed, the connection operations being transferred to the head of the enclosing AUtomaton or SYstem declaration. STate statement syntax is replaced with "if-then" conditions on operations. OPerator call arguments are transformed to connection statements. Compound Boolean expressions serving as conditions on operations are replaced with terminals of unit dimension. These new terminals are connected to the Boolean expressions via connection operations inserted in the head of the enclosing AUtomaton or SYstem declaration.

Pass 3 -- Conditions Distributed

"If-then" and "if-value" conditions on sets of operations are combined and distributed over the members of the set so that each operation appears as the body of a simple "if-then" clause. "Go-to" operations are converted to conditional transfers of a constant (the state assignment) to the state sequencing register (the enclosing automaton). Automaton syntax is eliminated by recognizing the global condition, if any, and distributing it as a clocking condition on all register transfer and memory operations within the AUtomaton declaration.

Pass 4 -- Concatenation Removed

All concatenation operations except those that form operands for reduction operators are eliminated by breaking operations into operations on subfacilities formed by partitioning operand facilities according to the dimensions of the concatenation operands.

Pass 5 -- Operations Gathered

All connection and transfer operations with the same data sink (left operand) are gathered into one compound operation.

Pass 6 -- Subfacilities Disjoined

Facilities with subfacilities serving as data sinks of connection and transfer operations are broken into disjoint subfacilities and a right-hand side of a connection or transfer operation is formed for each new subfacility.

3.2 AN EXAMPLE

System EX1 illustrates the use of secondary names and IDentifier declarations. Registers A and D of automaton A1 are each broken into subregisters via secondary names in the REgister declaration. Ascending and descending subscripts are illustrated. Identifiers X, Y and Z name a new concatenation of the subregisters of D, a portion of one of these subregisters, and a NOR reduction, respectively. A register A is declared in automaton A2 also. The operations of A2 all appear in the head portion of its AUtomaton declaration.

The listing obtained after Pass 1 summarizes the declared facilities and their relations. Since two A registers are declared in parallel blocks, the name of one is changed to A"1 so that the two may be distinguished. The declared operations are listed with indentation used to indicate the nested relations of blocks. Block structure errors would be easily identified.

Pass 2 replaces secondary names and identifiers with their primary equivalents. A careful examination of the results after Pass 2 indicates that operation A←X in state S becomes A←F∨E when X is replaced. Then

secondary names are removed giving A←D(4:1)¢D(8:5). The operations of state T require that secondary names F, B, C and E be replaced with their primary equivalents. Then Z within "if-then" punctuation is replaced with ¬+/Y is replaced with ¬+/F(3:2) is replaced with ¬+/D(2:1). Note that state statement syntax is also converted to "if-then" syntax in Pass 2. A state decoder network on automaton register A1 is prescribed by equations in the head of the SYstem declaration at this point.

Pass 3 distributes conditions over sets of operations and removes AUtomaton declaration syntax. The results listed indicate that five internal signals named "double-quote-integer" have been formed in order to simplify the expression of conditions on operations. Each of the conditioned operations can be traced back to the source DDL description. "Go to" operations are converted to conditioned transfers to the automaton register.

Pass 4 eliminates the concatenation operations. As a first example observe that

```
!P*S! A<-S*D(4:1)¢D(8:5).
```

is broken to

```
!P*S! A(1:4)<-S*D(4:1).,
!P*S! A(5:8)<-S*D(8:5).
```

Pass 5 gathers operations with the same left operand. The operations

```
!P*S! A(1:4)<-S*D(4:1).,
!P*"5! A(1:4)<-"5*D(4:1).
```

are gathered to

```
!P*S+P*"5! A(1:4)<-S*D(4:1) + "5*D(4:1).
```

No logic minimization or even simplification is performed as part of the gathering process.

In Pass 6 the A and D registers of automaton A1 are partitioned and transfer statements are developed for each subfacility. Pass 5 provides the following transfers to the A register or some part of it.

!P*S + P*"5! A(1:4)<-S*D(4:1) + "5*D(4:1).,

!P*S + P*"5! A(5:8)<-S*D(8:5) + "5*D(8:5).,

!P*"3! A<-"3*D.

The last of these operations involves the entire A register; the others involve a part of it. Pass 6 partitions the A register to A(1:4) and A(5:8), and forms the correct transfers to each of these subfacilities.

The F table as it appears after Pass 6 is listed as the final result of this example. Facility names are followed by left and right subscripts and facility dimensions. The next column indicates the type of the facility with negative entries (-1 for SYstem, -6 for REgister, -9 for TErminal, etc.). Positive entries point to the row of the parent facility. The final columns point to the beginning and ending points of facility operation statements in the DDL string.

```
 1:   <CO>AN EXAMPLE SYSTEM THAT EMPHASIZES SECONDARY NAMES
 2:        AND IDENTIFIERS<
 3:   <SY>EX1: <IT>M.
 4:            <AU>A1(2):M:
 5:                <RE>A(8)=M(2:6)IC(3:7),B(8:1)=C(4)IF(5:2).
 6:                <IE>X=MIL, Y = M(3:2),Z = T+/Y.
 7:                 <SI> S(0):A <=X, =>I.
 8:                      I(1):F <=MIL(7), E<=1004, IZI =>5;=>0..
 9:                      I(2):=>3, IF NOLZ A<=0 ALLZ U<=A
10:                                    NZLZ A<=A....
11:             <AU>A2:M: A<=M,C<=A
12:                    <RE>A(24),B(24)...(END OF SYSTEM)
13:   <FL>3,4,5,6,8.
```

PASS1--FACILITIES IDENTIFIED

DECLARED FACILITIES

```
        <SY> EX1
        <TI> P(1:1)
             <AU> A1
             <RE> A(1:5)  =  B(2:6)|C(3:7)
                  D(8:1)  =  E(1:4)|F(5:2)
             <ID> X(1:1)
                  Y(1:1)
                  Z(1:1)
             <ST> S
                  T
                  U
             <AU> A2
                                                     A => A"1
             <RE> A"1(1:24)
                  B(1:24)
```

DECLARED OPERATIONS

```
 0)
<SY> EX1:
     <AU> A1: P:
          <ST>
               S: A<=X, =>T.

               T: F<=B|C(7), E<=1DE4,
                   |Z| =>S; =>U..

               U: =>S,
                   !Y
                        #0B2A<=C
                        #1D2D<=A
                        #2D2A<=X....

     <AU> A2: P: A<=B, B<=A ..
```

PASS2--SYNTAX REDUCED

```
<SY> EX1: S=*/A1'002  , T=*/A1'102  , U=*/A1'202  ,
     <AU> A1: P:
          ]S] A<-D(4:1)[D(8:5), ->T.,
          ]T] D(4:1)<-A(1:3)[A(8), D(8:5)<-1004  ,
              ]T+/D(2:1)] ->S; ->U..,
          ]U] ->S,
              ]D(2:1)
                   #002   A<-D
                   #102   D<-A
                   #202   A<-D(4:1)[D(8:5)..., .

     <AU> A2: P: A"1<-B, B<-A"1.., .
```

PASS3--CONDITIONS DISTRIBUTED

```
<SY> EX1: S=*/A1'002  ,
     T=*/A1'102  ,
     U=*/A1'202  ,
     "1=T*T+/D(2:1),
     "2=T*T(T+/D(2:1)),
     "3=U*(*/D(2:1)'002  ),
     "4=U*(*/D(2:1)'102  ),
     "5=U*(*/D(2:1)'202  ),

     ]P*S]  A<-S*D(4:1)[D(8:5).,
     ]P*S]  A1<-S*102  ..
     ]P*T]  D(4:1)<-T*A(1:3)[A(8).,
     ]P*T]  D(8:5)<-T*1004  ..
     ]P*"1]  A1<-"1*002  ..
     ]P*"2]  A1<-"2*202  ..
     ]P*U]  A1<-U*002  ..
     ]P*"3]  A<-"3*D.,
     ]P*"4]  D<-"4*A.,
     ]P*"5]  A<-"5*D(4:1)[D(8:5).,
     ]P]  A"1<-B.,
     ]P]  B<-A"1.,
     , .
```

PASS4--CONCATENATION REMOVED

```
<SY> EX1: S=*/A1'002  ,
     T=*/A1'102  ,
     U=*/A1'202  ,
     "1=T*T+/D(2:1),
     "2=T*T(T+/D(2:1)),
     "3=U*(*/D(2:1)'002  ),
     "4=U*(*/D(2:1)'102  ),
     "5=U*(*/D(2:1)'202  ),
     ]P*S]  A(1:4)<-S*D(4:1).,
     ]P*S]  A(5:8)<-S*D(8:5).,
     ]P*S]  A1<-S*102  ..
     ]P*T]  D(4:2)<-T*A(1:3).,
     ]P*T]  D(1)<-T*A(8).,
     ]P*T]  D(8:5)<-T*1004  ..
     ]P*"1]  A1<-"1*002  ..
     ]P*"2]  A1<-"2*202  ..
     ]P*U]  A1<-U*002  ..
     ]P*"3]  A<-"3*D.,
     ]P*"4]  D<-"4*A.,
     ]P*"5]  A(1:4)<-"5*D(4:1).,
     ]P*"5]  A(5:8)<-"5*D(8:5).,
     ]P]  A"1<-B.,
     ]P]  B<-A"1.,
     .
```

ORIGINAL PAGE
OF POOR QUALITY

PASS5--OPERATIONS GATHERED

```
<SY> EX1: S=*/A1'002  ,
     T=*/A1'102  ,
     U=*/A1'202  ,
     "1=T*↑+/D(2:1),
     "2=T*↑(↑+/D(2:1)),
     "3=U*(*/D(2:1)'002  ),
     "4=U*(*/D(2:1)'102  ),
     "5=U*(*/D(2:1)'202  ),
     ]P*S + P*"5] A(1:4)<-S*D(4:1) + "5*D(4:1).,
     ]P*S + P*"5] A(5:8)<-S*D(8:5) + "5*D(8:5).,
     ]P*S + P*"1 + P*"2 + P*U] A1<-S*102   + "1*002   + "2*202   + U*002  .,
     ]P*T] D(4:2)<-T*A(1:3).,
     ]P*T] D(1)<-T*A(8).,
     ]P*T] D(8:5)<-T*1004  .,
     ]P*"3] A<-"3*D.,
     ]P*"4] D<-"4*A.,
     ]P] A"1<-B.,
     ]P] B<-A"1.,
     .
```

PASS6--SUBFACILITIES DISJOINED

```
<SY> EX1: S=*/A1'002  ,
     T=*/A1'102  ,
     U=*/A1'202  ,
     "1=T*↑+/D(2:1),
     "2=T*↑(↑+/D(2:1)),
     "3=U*(*/D(2:1)'002  ),
     "4=U*(*/D(2:1)'102  ),
     "5=U*(*/D(2:1)'202  ),

     ]P*"3 + P*S + P*"5] A(1:4)<-"3*D(8:5) + S*D(4:1) + "5*D(4:1).,
     ]P*"3 + P*S + P*"5] A(5:8)<-"3*D(4:1) + S*D(8:5) + "5*D(8:5).,
     ]P*S + P*"1 + P*"2 + P*U] A1<-S*102   + "1*002   + "2*202   + U*002  .,

     ]P*"4 + P*T] D(8:5)<-"4*A(1:4) + T*1004  .,
     ]P*"4 + P*T] D(4:2)<-"4*A(5:7) + T*A(1:3).,
     ]P*"4 + P*T] D(1)<-"4*A(8) + T*A(8).,

     ]P] A"1<-B.,
     ]P] B<-A"1.,
     .
```

[illegible] PAGE IS
OF POOR [illegible]

PASS5--OPERATIONS GATHERED

```
<SY> EX1: S=*/A1'002  ,
     T=*/A1'102  ,
     U=*/A1'202  ,
     "1=T*T+/D(2:1),
     "2=T*T(T+/D(2:1)),
     "3=U*(*/D(2:1)'002  ),
     "4=U*(*/D(2:1)'102  ),
     "5=U*(*/D(2:1)'202  ),
     ]P*S + P*"5] A(1:4)<=S*D(4:1) + "5*D(4:1).,
     ]P*S + P*"5] A(5:8)<=S*D(8:5) + "5*D(8:5).,
     ]P*S + P*"1 + P*"2 + P*U] A1<=S*102   + "1*002   + "2*202   + U*002  .,
     ]P*T] D(4:2)<=T*A(1:3).,
     ]P*T] D(1)<=T*A(8).,
     ]P*T] D(8:5)<=T*1004  .,
     ]P*"3] A<="3*D.,
     ]P*"4] D<="4*A.,
     ]P] A"1<=B.,
     ]P] B<=A"1.,
     .
```

PASS6--SUBFACILITIES DISJOINED

```
<SY> EX1: S=*/A1'002  ,
     T=*/A1'102  ,
     U=*/A1'202  ,
     "1=T*T+/D(2:1),
     "2=T*T(T+/D(2:1)),
     "3=U*(*/D(2:1)'002  ),
     "4=U*(*/D(2:1)'102  ),
     "5=U*(*/D(2:1)'202  ),

     ]P*"3 + P*S + P*"5] A(1:4)<="3*D(8:5) + S*D(4:1) + "5*D(4:1).,
     ]P*"3 + P*S + P*"5] A(5:8)<="3*D(4:1) + S*D(8:5) + "5*D(8:5).,
     ]P*S + P*"1 + P*"2 + P*U] A1<=S*102   + "1*002   + "2*202   + U*002  .,

     ]P*"4 + P*T] D(8:5)<="4*A(1:4) + T*1004  .,
     ]P*"4 + P*T] D(4:2)<="4*A(5:7) + T*A(1:3).,
     ]P*"4 + P*T] D(1)<="4*A(8) + T*A(8).,

     ]P] A"1<=B.,
     ]P] B<=A"1.,
     .
```

ORIGINAL PAGE IS
OF POOR QUALITY

FACILITY TABLE

#	Name	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9
1	EX1	1	1	1	-1	0	304	0	0
2	P	1	1	1	-5	0	0	0	0
3	A1	1	2	2	-6	0	154	254	263
4	A	1	8	8	-6	0	0	0	0
5	*1	1	1	1	-9	0	213	0	222
6	*2	1	1	1	-9	0	233	0	243
7	0	8	1	-8	-6	0	0	0	0
8	*3	1	1	1	-9	0	260	0	274
9	*4	1	1	1	-9	0	256	0	294
10	*5	1	1	1	-4	0	307	0	320
11	C	1	1	1	7	0	516	525	516
12	C	4	2	-3	7	0	544	551	542
13	S	21	0	1	-13	0	158	0	163
14	T	22	1	1	-13	0	165	0	170
15	U	23	2	1	-13	0	172	0	177
16	*2	1	1	1	-6	0	0	0	0
17		0	0	0	0	0	0	0	0
18	A*1	1	24	24	-6	0	324	330	332
19	B	1	24	24	-6	0	333	334	336
20	1004		10	4	-17	0	0	0	0
21	002		0	2	-17	0	0	0	0
22	102		1	2	-17	0	0	0	0
23	202		2	2	-17	0	0	0	0
24		0	0	0	0	0	0	0	0
25		0	0	0	0	0	0	0	0
26	A	5	8	4	4	0	455	456	453
27	A	1	4	4	4	0	441	502	474
28	J	8	5	-4	7	0	560	575	566

4. THE SIMULATOR (DDLSIM) [35]

DDLSIM is a program for simulating digital systems described using DDL. The simulator has a simple, powerful and completely free-format command language that provides the user with complete control over the simulation process without requiring that the DDL system description be modified. DDLSIM does very extensive error-checking of described systems, simulation control cards, and the simulation process itself. Self-explanatory messages that pin-point errors are issued.

Digital systems to be simulated are first described in DDL. This description is translated by DDLTRN into a set of Boolean equations and Register Transfer expressions. These can be used for implementation or simulation of the digital system. They, together with other data structures and tables established by DDLTRN constitute the system description required by DDLSIM. This description is pre-processed by the simulator to establish data structures and tables that permit more efficient and controlled simulation.

The original and translated DDL descriptions of a system neither contain any information for controlling simulation nor do they supply any input data for simulation. These items are supplied by a second source to DDLSIM, a <u>simulation</u> deck. This deck consists of simulator control <u>declarations</u> described using a simulator command language that is not unlike DDL. Twelve different declaration types are available for selecting options and supplying control information, parameters, and data for simulation. Every simulation job consists of:

1. processing the system description,
2. processing the simulation deck, and
3. simulation of the system.

The following notational conventions are used in subsequent sections to describe the syntax of translated DDL and to define control language syntax.

Script characters - an item of the language. Item α will be defined in terms of items β and γ with notation

$\alpha : \beta, \gamma$

which is read "an α is a β or a γ."

[ ] - appearance of the enclosed syntatic structure is optional

$[\ ]^n$ - the enclosed syntatic structure may be repeated an arbitrary number of times or not at all.

Blanks have no significance in syntax descriptions just as they have no significance in DDL or the DDLSIM control language.

4.1 SIMULATION MODELS

As mentioned earlier, Boolean equations (*BE*) and Register Transfer Expressions (*RTE*) generated by DDLTRN constitute the system description required by DDLSIM. The models of combinational networks and registers used by DDLSIM is the subject of this section.

4.1.1 Terminals, Element Inputs, and State Terminals

The terminals, element inputs, and state terminals declared in a system are described using *BE*s. In addition, DDLTRN generates *BE*s for a number of intermediate signals. All such *BE*s constitute the combinational portion of a system. They are first sorted into an ordered list according to the level of their operands, i.e., if a terminal A is used in the *BE* for another terminal B, A will appear before B in the sorted list. However, if the system contains loop(s) in it's combinational portion, it is not possible to sort the equations in this

manner. In such cases the BEs constituting the loop(s) (or loop equations) are separated from other BEs. The remainder of the BEs are then sorted into an ordered list as described Loop equations are then added to the sorted list at an appropriate point.

During simulation the combinational portion of a system is simulated at the BE level. BEs can vary from a simple sum-of-products form to the most complex and compound of forms. The BEs are evaluated in the order established by sorting with the loop equations being simulated repeatedly until their output values stabilize. Failure of a loop to stabilize after a fixed number (determined by the characteristics of the loop equations) simulations, indicates instability in the loop. In such a case a warning is issued to the user and the simulation is continued with the last computed values for the loop equations taken as their final values. Thus DDLSIM also permits the simulation of systems having loops in their combinational portion.

4.1.2 Delays

The delays declared in a system (using <DE> declarations of DDL or DDLSIM) are also described using BEs. These delays are assigned their delay time periods (Δs) using <DElay> declaration of DDLSIM (see Sec. 4.2.4). All the delay facilities are assumed to be inertial delays, i.e., an output signal(s) will assume a new value(s) Δ time units after it's input prescribes that change, if and only if the input signal prescribes that value for at least Δ consecutive time units. Unlike the BEs discussed above, the BEs for delays are not sorted in any particular order.

During simulation each delay is simulated at the BE level with specified inertial delay assigned to it's output. The new computed

value(s) for each delay is compared with its present output value(s). If they are different, a future event at Δ time units from present simulation time T is scheduled to carry out the change(s) in the output value(s). However, if the BE does not continue to prescribe the change for at least the next Δ time units, the scheduled event is cancelled and the output(s) of the delay remains unchanged.

It is possible to assign the <u>same delay time</u> $\lfloor t_d/2 \rfloor$ (see Sec. 4.2.2, 4.2.5) to all the BEs for the combinational portion (see Sec. 4.2.1) of the system by setting flag number 13 (see Sec. 4.2.14) In such a case all these facilities become equivalent to delays. It is important to note that the delay time assigned to these BEs is the same for all of them, irrespective of their complexity.

4.1.3 <u>Registers</u>

The registers declared in a system are described using RTEs.

RTE consists of a Condition Expression (CE) followed by a Transfer Expression (TE). RTEs generated by DDLTRN have the following general syntax:

RTE : | CE | TE.

CE : C $[+C]^n$

Condition term C : C_c * C_ℓ , C_ℓ

Clock condition C_c : global condition in the heading of an <AU> declaration of DDL, a clock declared in a <TI> declaration of DDL.

Load condition C_ℓ : δ with δ_w = 1. (see Sec. 4.2.1)

TE : $\delta \leftarrow E$

Load expression E : e $[+e]^n$

Expression term e: $C_{\ell} * V_{e}$

Load value V_{e}: an expression

Example: | P*LDX + P*ORXY + P*LDY | ACC ← LDX*X + ORXY* (X+Y) + LDY*Y.

In the example P is a clock; ACC, X, and Y are all registers having dimensions of 24; LDX, ORXY, and LDY are terminals declared using appropriate declarations. The *CE* in this example has three condition terms specifying the conditions for performing different register transfers on ACC. All the register transfers in this case are carried out under the control of the same clock P. In the *RTE* for registers declared as global facilities and used in different automata, each having a separate clock or global condition, the *CE*s may have different clock conditions C_c. For each condition term C in the *CE*, there is a corresponding expression term e in the *TE*. When a load condition C_{ℓ} becomes true (logic 1) and the corresponding clock condition C performs a 0-to-1 transition, the next-output value for the register is computed using the load value V_e from corresponding expression term e. On the next 0-to-1 transition of the C_c, this next-output value becomes the present-output value.

During simulation *CE*s for all the registers are evaluated only at certain event-times (see Sec. 4.3). On a 0-to-1 transition in the value for a *CE*, the corresponding *E* is evaluated and the computed value is stored as the next-output value for the register. On a 1-to-0 transition of the same *CE* at some future evaluation, the next-output value for the register becomes it's present-output value. In order to make simulation fast and efficient, *CE*s are evaluated *only* at event-times at which 0-to-1 or 1-to-0 transitions of clock conditions take place. It is not possible to have a 1-to-0 and 0-to-1 transitions for the same *CE* at the same

simulation time T. It is possible to simulate asynchronous sequential systems using DDLSIM.

The simulation model used for a register is very similar to a GL (gate and latch) flip-flop. A logical OR of load conditions C_ℓ from CE constitutes the Boolean equation for the GATE of the flip-flop, E from RE constitutes the LATCH equation for the flip-flop, and a logical OR of the clock conditions C_c from CE constitute the CLOCK of the flip-flop. (See the figure below)

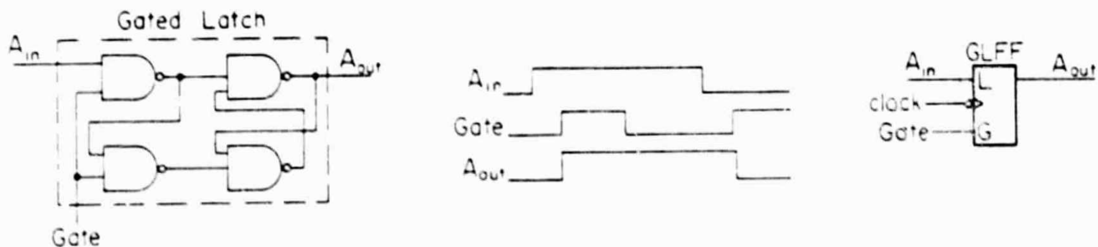
4.1.4 Memories

The memories declared in a system are also described using RTEs. A RTE for a memory is similar to that for a register with an address specified for the facility δ, i.e.,

memory δ : $\delta(a)$

address expression a : an expression

The simulation model used for memories is also similar to that used for registers. For memory-write operations the address expression a is evaluated on a 0-to-1 transition of the associated CE and the computed value is stored as the address of the memory location. On the next 1-to-0 transition of the condition expression CE, the contents of the addressed location are changed to the supplied value. Memory-read operations are instantaneous, i.e., contents of the referenced memory location are fetched immediately.



4.2 SIMULATOR COMMAND LANGUAGE AND DECLARATIONS

The DDLSIM command language consists of twelve different types of declarations for supplying parameters, input data, options and other control information necessary for simulation. The language is largely free of format restrictions. Card images are scanned in turn from left to right. Any declaration may start at any point and end at any later point in the card deck. A declaration can be continued on as many cards as necessary; more than one declaration can be supplied on the same card. The start of a declaration automatically ends the previous declaration. The last declaration in a simulation deck is ended by an End-Of-File (normally a card having '$' in the first column). In general, the order in which the declarations are specified is not important. It is possible to have more than one declaration of the same type. Everything following the vertical line character (¦) on a card is treated as a comment, and is not processed as a part of a declaration. Scanning continues on the next card. This provides the capability of having in-line comments in a simulation deck.

Each card from a simulation deck is processed sequentially by the simulator. First it is printed together with it's sequence number. It is possible to suppress echo printing of the simulation deck by turning the list option off, i.e., resetting Flag 1.

Each simulator declaration has the general syntax

<Declaration-id> *Body*

Each declaration begins with a left angle (<) followed by a <u>Declaration-id</u> that identifies the type of the declaration. Only the first two characters of the Declaration-id are examined by the simulator. The Declaration-id is terminated by a right angle (>). All declarations except the

<SImulate> declaration have a *Body* following the heading.

4.2.1 Facilities

Facilities are defined here as in DDL to be any piece of hardware declared in a digital system including terminals, registers, memories, and assemblage of hardware, clocks, delays, etc. If a facility name f_n exceeds 8 characters, only the last 8 characters are retained. If a facility has dimension greater than one, individual elements are identified by appending a non-negative integer subscript S_1 enclosed in parentheses to f_n. A range of elements of a facility is identified by using a DDL subscript range i.e., $f_n(S_1 : S_2)$. A script letter f will be used to represent a facility or a part of it.

$$f : f_n(S_1 : S_2),\ f_n(S_1),\ f_n \qquad \text{where}$$

$$f_n(S_1) = f_n(S_1 : S_1)$$

$$f = f_n(S_\ell : S_r)$$

S_ℓ = subscript for leftmost element of f_n.

= subscript for rightmost element of f_n.

Facility width f_w of a facility f is defined as the total number of elements in it, i.e.,

$$h_w = \max(S_1 , S_2) - \min(S_1 , S_2) + 1$$

During simulation one machine word is used to store the values of facility f. The SEL 32 machine has 32 bits per word. Hence it is necessary that the facility width f_w for any facility f_w in the system not exceed 32, i.e., $f_w \leq 32$. However, S_ℓ and S_r may have larger values; only their difference is restricted.

A facility list ℓ_f is defined as a list of facilities f separated by commas, i.e.,

$$\ell_f : f[,f]^n$$

Whether a specific facility can be used in a facility list for a specific type of declaration is determined by both the type f_t of the facility and the type of the declaration. The following facility types exist for DDLSIM.

f_t : System clock, Register, Memory, Terminal, System delay, Element input, Element output, State terminals, Trigger, Simulation delay, Simulation clock, List name.

Every facility f used in a DDLSIM declaration must satisfy exactly one of the following conditions:

1. f is declared in the DDL description of the system.
2. f is declared during the present simulation run using a <CLock>, <DElay>, <TRigger>, or <LIst> declaration. The type of declaration in which f appears determines its type f_t which cannot be changed for the remainder of the simulation job.
3. f is declared during any previous simulation run as discussed in 2 above.

4.2.2 <u>Numbers and Data Lists</u>

T_{MAX} - a decimal integer having the value $(2^{31} - 1)$.

P_{MAX} - a decimal integer having the value $(2^{16} - 1)$.

$n_{i,j}$ - a decimal integer n in the range $i \leq n \leq j$ where i and j are each non-negative decimal integers. Whenever j is not specified $j = T_{MAX}$ is assumed; whenever i is not specified $i = 0$ is assumed.

$n^P_{i,j} = n_{i,j}$ enclosed in parentheses

$n^p_{i,j}$: $(n_{i,j})$

R - Repeat factor, a positive decimal integer

R : n_1

A repeat factor R can be used before a data value or parameter value, i.e.,

R*value,

to indicate that the same value is to be repeated R times in the list.

T - Simulation time

T : n

t_d - Default time period

t_d : n_1, p_{MAX}

Data is described with the following syntatic structures.

d_B - a binary digit

d_B : 0,1

d_O - an octal digit

d_O : 0,1,2,3,4,5,6,7

d_D - a decimal digit

d_D : 0,1,2,3,4,5,6,7,8,9

d_H - a hexadecimal digit

d_H : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

d_G - a general digit excluding the hexadecimal digits B and D.

d_G : 0,1,2,3,4,5,6,7,8,9,A,C,E,F

B - a binary number

B [+,-]Bd_B [d_B]^n, ([+,-]Bd_B [Bd_B [d_B]^n)

O - an octal number

O : $[+,-]\mathrm{O}d_O\ [d_O]^n$, $([+,-]\mathrm{O}d_O\ [d_O]^n)$

D - a decimal number

D : $[+,-]\mathrm{D}d_D[d_D]^n$, $([+,-]\mathrm{D}d_D\ [d_D]^n)$

H - a hexadecimal number

H : $[+,-]\mathrm{H}d_H[d_H]^n$, $([+,-]\mathrm{H}d_H\ [d_H]^n)$

N - a binary, octal, decimal or hexadecimal number.

N : $[+,-]d_G\ [d_H]^n$, $([+,-]\ d_G\ [d_H]^n)$

Optional leading minus signs (-) before any of above five types of numbers specifies the 2's complement of the number. 1's complement encoded negative numbers are obtained by setting Flag 10 (see Sec. 4.2.13).

N_2 - Data value

N_2 : B, O, D, H, N,

N_1 - Data spec

N_1 : $[R*]\ N_2$

ℓ_d - a data list consisting of data specs separated by commas.

$\ell_d : N_1\ [,N_1]^n$

Whenever a data value is specified as a number N without leading radix specification, the default radix is used for computing the value of number. The default radix of 8 (octal) can be changed to 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal) by setting flag numbers 2 thru 5 (see Sec. 4.2.14 respectively. Resetting these flags returns the radix to the default value of 8 (octal).

4.2.3 <CLock> Declarations

This declaration provides a means for specifying or changing the time period, pulse width and phase of clock facilities. It also permits users to declare new clocks to be used to control simulation input and

output activities. Syntax for this declaration is as follows:

<CLock> *Body*

Body		:	$[\ell/,]^n\ \ell[/]$
List	ℓ	:	ℓ_c/ℓ_t
Clock list	ℓ_c	:	ℓ_f where
Facility type	f_t	:	system clock, simulation clock
Time list	ℓ_t	:	$t[,t]^n$
Time spec	t	:	$[R*]\ P\ [W[\theta]]$
Time period	P	:	$n_{2,P_{MAX}}$
Pulse width	W	:	$n^P_{1,P-1}$
Phase	θ	:	$n^P_{0,P-W}$

Example: <CL> CLOCK1(1:5), CLOCK2/2*100(30) (50)/,

CLOCK1(6:10), CLOCK3/100,100(30)/

Time period P - the P field specifies the time period of a clock. In the above example each clock has a time period of 100 in some arbitrary units.

Pulse width W - This is an optional field specifying the time W for which a clock remains at logic 1 during any clock period P. For the remaining time ($P-W$) the clock remains at logic 0. When the pulse width is not specified along with the time period, the following default value W is used.

$$W = \lfloor P/2 \rfloor$$

In the example a pulse width of 30 units is supplied for both CLOCK(1:5) and CLOCK2. CLOCK3 is assigned a pulse width of 30 units. No pulse width is explicitly specified for CLOCK1(6:10), hence a default value of $\lfloor 100/2 \rfloor = 50$ units is used as the pulse width.

Phase θ - At the start of a simulation run, i.e., $T = 0$ a clock with a

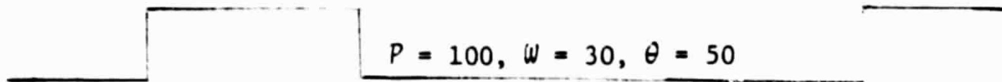
period P and the pulse width W is set to start at logic 0. It remains at logic 0 for the next $(P-W)$ time units; then a 0-to-1 transition takes place. For the next W time units, it stays at logic 1; then a 1-to-0 transition takes place and the cycle is repeated. The occurrence of the first and every subsequent 0-to-1 transition can be advanced relative to the starting of simulation by specifying the phase θ.

1. For phase $\theta < P - W$ a clock starts at logic 0 and has it's first 0-to-1 transition at $(P-W-\theta)$ time units after the start of simulation.
2. For phase $\theta = P - W$, a 0-to-1 transition takes place at $T = 0$.

The default value for θ is zero. In the example a phase of 50 units is specified for CLOCK1(1:5) and CLOCK2. Since no phase specification is given for CLOCK1(6:10) and CLOCK3, $\theta = 0$ is assumed for them. Waveforms for these clocks are shown below. Note that it is necessary to specify pulse width W, if it is desired to specify phase θ.

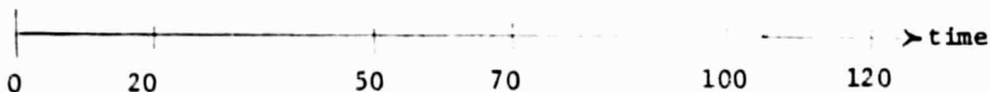
During a simulation run, none of the parameters, P, W, and θ can be respecified for a clock facility. These parameters remain effective in all subsequent runs until respecified.

CLOCK1 (1:5)
CLOCK2



CLOCK1 (6:10) $P = 100$, $W = 50$, $\theta = 0$

CLOCK $P - 100$, $W = 30$, $\theta = 0$



As mentioned earlier this declaration allows new facilities to be declared as simulation clocks. Since these clocks cannot affect the activity within the system itself, they are a source of periodic signals which can be used to control input, reinitialization, output, etc., during simulation. They can be used in realizing signals with complex waveforms that are needed to control various activities during simulation. Simulation clocks may also be used as sources of input signals to the networks being simulated.

Each facility f from clock list ℓ_c is assigned parameters t from associated time list ℓ_t. Insufficient or excess data in time list ℓ_t will result in a non-fatal error (see Sec. 4.4 for errors). In the case of insufficient data, default parameters are assigned to facilities remaining in ℓ_c.

4.2.4 <DElay> Declarations

This declaration provides a means for specifying delay time Δ for delay facilities. Syntax for this declaration is very similar to that of the <CLock> declaration.

<DElay> *Body*

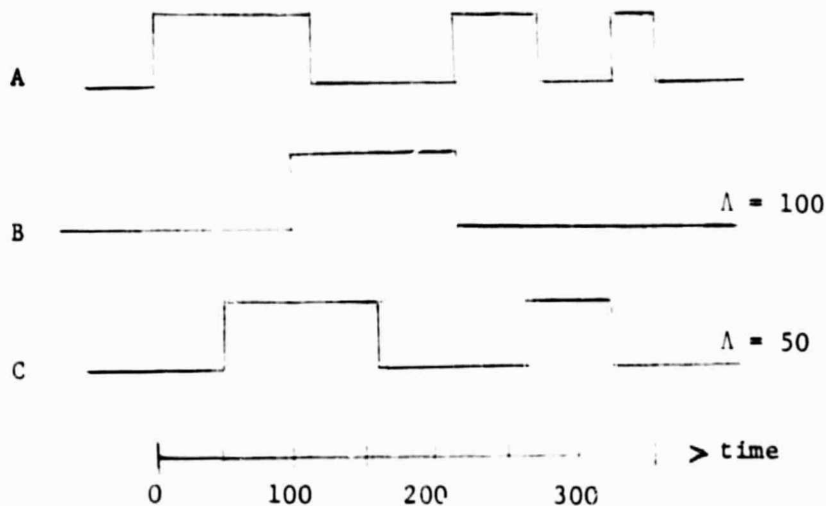
Body $[\ell/,]^n \ \ell \ [/]$

list	ℓ	:	ℓ_d/ℓ_t
Delay list	ℓ_d	:	ℓ_f where
Facility type	f_t	:	system delay, simulation delay
Time list	ℓ_t	:	$t[,t]^n$
Time spec	t	:	$[R*]\Delta$
Delay time	Δ	:	n_1

Example: <DElay> DELAY1(1:2), DELAY2, DELAY1(3:5)/2*100,50/

DELAY1(1:2) and DELAY2 are each assigned a delay time of 100 units. DELAY1(3:5) is assigned a delay time of 50 units.

All the delay facilities are assumed to be <u>inertial</u> <u>delays</u>, i.e., an output signal(s) will assume a new value(s) Δ time units after its input signal prescribes that change, if and only if the input signal prescribes that new value for at least Δ consecutive time units. As an example of inertial delay assume that waveform A below serves as the input signal to both DELAY1(1) and DELAY1(3). Waveforms B and C represent the actual output of DELAY1(1) and DELAY(3) respectively.



Delay time period Δ can not be respecified within a simulation run. Once specified, Δ remains effective in all subsequent simulation runs until respecified.

Like the <CLock> declaration, this declaration also allows a user to declare new delay facilities that may also be used for controlling various activities during simulation.

Every delay facility from ℓ_d is assigned, in turn, delay times from the associated time list ℓ_t. Insufficient or excess data in ℓ_t will result in a non-fatal error. In the case of insufficient data, the default delay time (4.2.5) is used for remaining facilities in ℓ_d.

4.2.5 Default Values for Clock Parameters and Delay Times

Before any simulation can be performed, it is necessary to assign clock parameters to every clock facility and delay time to every delay facility. Values specified through <CLock> and <DElay> declarations are used for specified facilities. For the remaining clock and delay facilities, default values are used. A default time period t_d is used in determining the default values.

1. Default clock parameters

   Default time period $P = t_d$

   Default pulse width $W = \lfloor t_d/2 \rfloor$

   Default Phase $\theta = 0$

2. Default delay time period $= \lfloor t_d/2 \rfloor$

At the start of a simulation job t_d is set to a value of 2. If any <CLock> or <DElay> declaration is encountered in the simulation deck, the value t_d is changed to

$t_d = \min(P, 2\Delta)$ where

P is any clock period specified, if none $P = 2$, and Δ is any delay time specified, if none $\Delta = 1$.

4.2.6 <INitialize> Declaration

This declaration provides a means for initializing the output value(s) of delays, registers, memories, element outputs, primary input signals, terminals and triggers with delays. Syntax for this declaration is as follows:

<INitialize> *Body*

Body : $[\ell/,]^n\ \ell[/]$

List ℓ : ℓ_i/ℓ_d

Initialize List ℓ_i : ℓ_f where

Facility type f_t : system delays, simulation delays, registers, memories, element outputs, primary inputs, terminals, and triggers with delays.

ℓ_d : Data list (see Sec. 4.2.2.)

Every facility f from ℓ_f is initialized to a specified value obtained from the associated ℓ_d. Insufficient or excess data in ℓ_d will result in a non-fatal error. If data in ℓ_d is insufficient, remaining facilities from ℓ_i are initialized to default values.

EXAMPLE: <IN> INPUT, MEM(0:1023)/B1011,1024*0/

INPUT (declared as register having width 4) is initialized to the binary value 1011 and the first 1024 locations of MEM are all initialized to 0.

Before any simulation can be performed during a run, it is necessary to define output values or initialize all the facilities. For all the facilities initialized through an <INitialize> declaration(s), specified values are used. For remaining facilities initial values are determined as follows:

1. Delays, Registers, Element outputs, Primary inputs, Terminals, or Triggers with delays are all initialized to zero.
2. Memory locations are not initialized at all. They will have the same contents as at the termination of previous simulation run. For the first simulation run their contents are unpredictable.
3. Initial values for Terminal, Triggers, and Element inputs without delays are determined by using intitial values for other facilities

and simulating the system at $T = 0$.

4.2.7 <REad> Declarations

This declaration provides a means for establishing input data values for various facilities. Syntax for this declaration is as follows:

<REad> *Body*

Body : $[\ell/,]^n \ell[/]$

List	ℓ :	$m/\ell_r/\ell_d$
Mode	m :	X, Y, Z
Triggered or Mode,	X :	f where $f_w = 1$
Periodic or Mode,	Y :	$P[\theta]$
		Period $P : n_{1,P_{MAX}}^{P}$
		Phase $\theta : n_{0,P}$
Specific Time or Mode	Z :	n
Read List	ℓ_r:	ℓ_f where
Facility type	f_t:	registers, system delays, simulation delays, memory locations, element outputs, terminal or triggers or element inputs with delays
Data List	ℓ_d:	same as in <INitialize>

Example: <TR> TR/EXINP+EXBIN1/ (see Sec. 4.2.15)

<CL> P/100(30)/

<RE> TR/INPUT/1,2,3,4,-5/

As shown in the syntax, the READ operation may be carried out in three different modes:

1. Mode X -- Triggered Mode

In this mode a 0-to-1 transition of the triggering signal establishes a new set of input values, obtained sequentially from the associated data

list ℓ_d, for the facilities specified in the associated read list ℓ_r. At any simulation time input values are established before any other simulation activity except for updating of clocks and delay outputs. Hence, if the triggering signal itself is not a clock or delay facility, input values will be established at a time later than the actual 0-to-1 transition time of the triggering signal. In fact they are established at the next event time.

2. Mode Y -- Periodic Mode

This mode provides an easy means for establishing input values periodically. P specifies the time period for performing the READ operation. The first READ operation is performed at $T = P$, the next at $T = 2P$, and so on. However, the first and all subsequent READ operations may be advanced relative to the beginning of simulation i.e., $T = 0$, by optionally specifying the phase θ. Thus, in the case of $P = 100$, and $\theta = 30$, the first READ operation will be performed at $T = 70$ (advanced by 30), the the next at $T = 170$, and so on. When $\theta = P$, the first READ operation is performed at $T = 0$. This is equivalent to initializing the associated facilities using an <INitialize> declaration.

In all cases except for $P = 1$, an identical periodic READ operation can be obtained using a clock with period P, any valid pulse-width W and appropriate phase θ as a triggering signal in mode X.

3. Mode Z -- Specific Time Mode:

In this mode the READ operation is performed only once at the specified time.

In Mode X and Mode Y READ operations, data values are supplied in sets. The first set of values are used for the first READ operation,

and the next set is used for the second READ operation. These sets are not separated by any special delimiter. Instead they are grouped in the form of a single data list ℓ_d. In Mode Z only one set of data values are necessary.

4.2.8 <u><LOad> Declarations</u>

This declaration provides a means for establishing the <u>same</u> input values repeatedly on specified facilities. Syntax for this declaration is as follows:

<LOad> *Body*

Body : same as in the <REad> except that the Load list ℓ_ℓ is used in place of the Read list ℓ_n.

Three modes of LOAD operation function in the same way as the three modes of READ operation. The only difference between LOAD and READ operations is the input data values used during successive operations. In the case of READ operations, a new set of data values is used for each successive operation. The LOAD operation uses the same data set repeatedly, requiring only one set of data values. This peculiar property of the LOAD operation provides an easy means for establishing <u>identical conditions</u> in the system at desired times. If the READ operation were used to achieve the same results, the same data set would have to be repeated for every occurrence of READ operation. The Mode Z or specific time LOAD operation is identical in all respects to the Mode Z READ operation.

The three modes available for READ and LOAD operations give a high degree of freedom in setting up data sets in an efficient manner. Each of these modes may be used more than once. More than one mode may be used in a simulation. All the data lists ℓ_d specified in <REad> and

<LOad> declarations are transferred to an incore buffer (if necessary to a disk data file) and retrieved from there whenever needed. This facility of having muptiple input streams for simulation is very helpful to the user.

More than one READ and/or LOAD operation may take place at the same simulation time. Simultaneous operations may attempt to establish input values on the same facilities. As long as they do not attempt to establish conflicting values, simulation will proceed, otherwise a fatal error condition results in an immediate termination of the current simulation run. A similar situation may arise with <INitialize> declarations. In this case remaining declarations for the simulation run are processed before terminating that simulation run. This fatal error condition may be avoided by setting Flag 12 (see Sec. 4.2.14).

The following order is used in performing various input operations during simulation:

1. Periodic REad
2. Specific Time READ
3. Periodic LOAD
4. Specific Time LOAD
5. Triggered READ
6. Triggered LOAD

If more than one operations of the same type and same mode take place at the same time, they are performed according to their order in the simulation deck. This is one case in which the order of declarations may be important.

Insufficient data to complete a READ or LOAD operation during simulation will result in an immediate termination of the run. This

provides a means to terminate a simulate run without using the <STop> (see Sec. 4.2.11) declaration.

4.2.9 <OUtput> Declarations

This declaration provides a means for printing the values of various facilities at various instants during simulation. Syntax for this declaration is as follows:

<OUtput> $Body$

	$Body$:	$[\ell/,]^{n}\ \ell[/]$
List	ℓ :	m/ℓ_{o}
Mode	m :	X, Y, Z
Triggered or Mode X	:	s where $s_{w} = 1$
Periodic or Mode Y	:	$P[\theta]$
Period	P :	$n_{1,P_{MAX}}$
Phase	θ :	$n^{P}_{0,P}$
Specific Time or Mode Z	:	n
Output List	ℓ_{o}:	ℓ_{s} where $s_{t} \neq$ memory

Like <REad> and <LOad> declarations, this declaration has three modes of operation. Values are printed when a specific output operation takes place. It is important to note that in the case of triggered or Mode X output, 0-to-1 transition of the triggering signal causes the output values to be printed at the <u>same time</u> rather than the next event time as in case of READ or LOAD operations. This is due to the fact that output operations are performed after all other operations in a simulation step. More than one <OUtput> declarations may be specified. Any combination of the three <OUtput> modes may be used.

Values are normally printed in an octal format. They may be printed

in binary, octal, decimal or hexadecimal by setting the appropriate flag number from 5 to 9 (see Sec. 4.2.14). All values are printed in the same format.

Output formatting is done by the simulator with the objective of maximizing the total no. of facilities than can be printed. If one or more output operations occur at a simulation time only a single line of output is printed. The first entry in each line printed is the simulation time in decimal. Values for each facility specified in any output lists ℓ are printed in fixed columns. Facilities are allocated columns from left to right in the following way:

1. Triggered or Mode X OUTPUT lists
2. Periodic or Mode Y OUTPUT lists
3. Specific time or Mode Z OUTPUT lists

If more than one lists of a mode are specified, they are allocated columns in the order of their specification in the simulation deck. If output values for all specified facilities cannot be printed due to lack of room, excess facilities are ignored and a message listing them is printed. Output values for two neighboring facilities are always printed. Output values for two neighboring facilities are always separated by at least one blank column. A heading of the names of the facilities along with the subscript(s), if necessary, whose values appear below is printed on alternate pages of the simulation report. If a complete facility is included in ℓ_0, no subscripts are printed in the heading. When the value of a partial facility is to be printed, a subscript range is included in the heading. The name of a facility is normally printed in a horizontal format in the heading. A vertical format (in a column) is used if doing

so saves room on output line. A subscript range is indicated by two suc-scripts separated by a colon (:).

Whenever an output operation occurs, only the output values for facilities from the associated output list ℓ_0 are printed. Other columns in the line are left blank. This tends to increase the readability of results. This feature of multiple output lists with each list having it's own output control may be used to make simulation reports look more informative. If the output values for one group of facilities change less frequently than those of another group, both groups can be printed with different periods. Such an output will clearly illustrate the actual activity within the system.

4.2.10 <DUmp> Declarations

This declaration provides a means for dumping the contents of specified memory locations at various instants during simulation. Syntax for this declaration is given below:

<DUmp> *Body*

Body : same as for <OUtput> except δ_t : memory

A DUMP operation functions in a manner identical to the OUTPUT operation. The print format is different, however. First, values for each specified memory facility are printed separately. For each facility, the first line printed indicates the facility name, locations dumped and simulation time. Following this line a heading that separates addresses and contents of locations is printed. One or more lines of DUMP output follows. In each line the first entry represents the octal address of the first location in the line. The rest of the line contains the octal contents of the next eight locations. Various DUMP operations are carried out in the following order:

1. Triggered or Mode X DUMPS

2. Periodic or Mode Y DUMPS

3. Specific time or Mode Z DUMPS

DUMP operations are performed before any OUTPUT operations within a simulation step.

4.2.11 <STop> Declarations

This declaration provides a means for stopping or terminating a simulation run at a specified simulation time or on a 0-to-1 transition of a triggering signal. Syntax for this declaration is as follows:

<STop> *Body*

Body : $m[,m]^n$

Mode m : X, Z

Triggered or Mode X : δ where $h_w = 1$

Specific Time or Mode Z : n

It is clear from the syntax that more than one triggering signal or specific time may be specified to stop the simulation. More than one <STop> declarations may be specified. In any case the occurrence of a first stop event will cause the simulation for that run to be terminated. At a given simulation time stop events are simulated after all other events have been simulated. If no <STop> declaration is supplied for the current simulation run stop events, if any, from a previous simulation run are used for the current run.

Insufficient data to complete a READ or LOAD operation will result in an immediate termination of the simulation run. This condition is described as "END-OF-FILE ON INPUT." If one is sure of EOF terminations, <STop> declarations may be omitted altogether. Whenever simulation is

stopped or terminated a message describing the reason for termination (a stop event or EOF on input) is printed and the simulator moves to the next simulation run. At the end of all simulation runs an "END OF SIMULATION" message is printed.

4.2.12 <LIst> Declarations

When a list of same facilities ℓ_δ is used in a number of different declarations, it is convenient to identify the entire list with a single name. This name can then replace the list of facilities in all of the declarations. This is achieved by using a <LIst> declaration. This declaration provides a means for assigning a unique name to a list of facilities. Syntax for this declaration is as follows:

<LIst> *Body*

Body : $[\ell/,]^n\ \ell[/]$

List $\ell : L/\ell_\delta$

List Name $L : \delta$ where $\delta_w = 1$

A list-name can be included in the facility list ℓ_δ for a declaration only if the list of facilities identified by it can be directly used there. It is also possible to use list-names in defining new list-names. Nesting of list-names can be done up to any desired level. List recursion, i.e., using a list-name in defining itself, is not permitted. Once declared for a simulation run, list-name cannot be redefined in the same run.

For large systems, use of list-names will result in reduction of data structure storage space. List-names are most commonly used in <REad> and <LOad> declarations since it is necessary to respecify these declarations for each simulation run, if a <REad> or <LOad> declaration requires a

long facility list, it is very worthwhile to assign a list-name to these facilities and use the list-name in their place.

4.2.13 <SImulate> Declarations

As discussed earlier this declaration is used to separate different simulation runs in a simulation job. Syntax for this declaration is very simple:

<SImulate>

On encountering a <SImulate> declaration, simulation is performed for current run. If this simulation is terminated normally i.e., through a <STop> command or EOF on input condition, processing for the next run is initiated. If the termination is abnormal, the entire simulation job is terminated. Declarations and parameters effective during one run are carried over to the next run as described below. Modifications and additions are easily made with appropriate declarations.

1. Parameters for clock and delay facilities remain effective from one simulation run to the next; any parameter may be changed by using the appropriate declaration. New clocks and delays may also be declared.
2. Trigger expressions remain unchanged from run to run unless they are respecified. New trigger facilities may be declared for a simulation run.
3. <REad> or <LOad> declarations do not carry from one run to the next. <REad> and <LOad> declarations must be respecified for each run or replaced with new declarations.
4. <OUTPUT>, <DUmp>, and <STop> declarations are carried from run to run. However, supplying one of these declaration with a specified mode (X, Y, or Z) will nullify all declarations from previous run

of that type and mode.

5. All flags are carried from run to run. Flags can be changed in any way by including a new <FLag> declaration.

6. Lists are carried from run to run. If it is necessary to redefine a list the new definition must be declared before the list is used directly or indirectly in any declaration for the current run.

4.2.14 <FLag> Declarations

This declaration provides a means for selecting various options for simulation runs by setting or resetting associated flags. A flag number is associated with each option. Syntax for this declaration is as follows:

<FLag> *Body*

Body : V_0 $[,V_0]^n$

Option Value V_0 : [⌐] F

Flag Number F : $n_{1,14}$

If the flag number F is not preceded by a complement sign (⌐), the associated option is set, otherwise it is reset. An option may be set or reset as many times as desired. The Flag table provides a description of the option controlled by each flag number and the default value for the option. As shown in the table flag numbers 2 thru 5 are used to select the radix for input data. This option applies only to data values not having any explicit radix specification (see Sec. 4.2.2). Data values having explicit radix specifications are interpreted accordingly. If more than one of these options is set, only the last set option is used, i.e., <FL> 2, 4 is equivalent to <FL> 4. Moreover, resetting any of these options brings the default radix specification to it's default value of 8 (octal). Similar action is taken for output format selection flags 6 thru 9.

FLAG TABLE		
Flag	Significance	Default
1	Print source card images	Set
2	Binary data input	Reset
3	Octal data input	Set
4	Decimal data input	Reset
5	Hexadecimal data input	Reset
6	Binary output format	Reset
7	Octal output format	Set
8	Decimal output format	Reset
9	Hexadecimal output format	Reset
10	Use 1's complement notation	Reset
11	Write processed system to file	Reset
12	Do not abort on "conflicting inputs" error	Reset
13	Simulate comb. portion of the system with delay	Reset
14	Not used	

4.2.15 <TRigger> Declarations

As discussed earlier, a triggering signal is used to control triggered or mode X READ, LOAD, OUTPUT, DUMP, and STOP operations. Any element of a declared facility, except a list-name, may be used as the triggering signal for these operations. Appropriate triggering signals to control the simulation may not be available in the DDL description of a system. The <TRigger> declaration provides a means for declaring new facilities that can be used as triggering signals to control simulation

without requiring that the DDL system description be modified. The syntax for this declaration is as follows:

<TRigger> *Body*

Body : $[t_E/,]^n \; t_E \; [/]$

Trigger expression t_E : t_δ/E

Trigger facility t_δ : δ where $\delta_\omega = 1$

Expression E : an expression

Example: <TR> TR/EXINP+EXBIN1

The expression E in the above syntax is a logical expression which can vary from simple sum-of-products form to the most complex and compound of forms. It defines the associated trigger facility t_δ. A trigger facility may be used in defining other trigger facilities. Looping or trigger facilities, i.e., using a trigger facility directly or indirectly to define itself, is not permitted, however. In the example trigger TR is defined as the logical OR of EXINP and EXBIN1. Both EXINP and EXBIN1 have been declared to be states of an automaton. The automaton waits in each of these states for data from an input device. The input device can be simulated using a triggered <REad> operation with TR as the triggering signal as shown in the example in Sec. 4.2.7.

A trigger facility cannot be redefined during a simulation run. The definition of a trigger facility remains effective in all subsequent runs until respecified. A trigger facility may be assigned a delay time Δ using a <DElay> declaration. Similarly a delay declared during simulation may also be defined using a <TRigger> declaration. For such facilities, both the delay time Δ and the expression E remain effective in subsequent runs until respecified.

4.3 SIMULATION ALGORITHM

DDLSIM is a table-driven, event-oriented simulator. Time is treated as a discrete quantity and advanced from event time to event time where the following actions are considered by the simulator to be <u>events</u>:

1. Zero-to-one transitions of clocks. During these events data input signals to registers and memories are sampled, and next values of register and memory contents are computed and saved.
2. One-to-zero transitions of clocks. During these events register and memory contents are updated to new values.
3. Delay lines taking new values.
4. Simulator input, output and control events.

The simulator maintains a list of events to be executed in the future. Simulation is performed only at event-times. The simulation clock is always stepped from one event-time to the next event-time, no simulation being performed for the intermediate time interval. The absence of any event during these intermediate time intervals implies that no change is taking place in the system. For each event-time tests are performed to establish the need for simulation. Simulation is performed at event-time only if needed. A periodic event causes a future event to be scheduled.

Event time simulation makes the units used for time specification unimportant. Any arbitrary units can be used. The number of events simulated and not the number of time units elapsed determine the computer time consumed by a simulation run. Since the largest integer handled by the SEL 32 machine is $(2^{32}-1)$, it is necessary to keep the simulation termination time within this limit. It is suggested that smaller

time periods be used for long simulation runs to avoid overflow of the simulation clock.

At a given event-time various events, if present, are processed in the following order:

1. Zero-to-one transitions of clocks
2. One-to-zero transitions of clocks
3. Change of output values for delays
4. Periodic or Mode Y READ operations
5. Specific time or Mode Z READ operations
6. Periodic or Mode Y LOAD operations
7. Specific time or Mode Z LOAD operations
8. Periodic or Mode Y DUMP operations
9. Specific time or Mode Z DUMP operations
10. Periodic or Mode Y OUTPUT operations
11. Specific time or MODE Z OUTPUT operations
12. Specific time STOP operation

After processing these events different simulations steps, if necessary, are performed in the following order:

1. Triggered or Mode A <REad> operations are completed
2. Triggered or Mode A <LOad> operations are completed
3. If there were any new one-to-zero transitions of clocks declared in the DDL description or combinational clocks, i.e., signals generated using combinational logic and used as clocks for registers, output values for affected Registers or memory locations are changed to their new values.

4. If necessary, the combinational portion of the system is simulated. If any new one-to-zero transitions of combinational clocks are detected, Step 3 and 4 are repeated until no more one-to-zero transitions occur.
5. If any one-to-zero transitions of system clocks or combinational clocks were registered, new output values are computed and saved for affected registers and memory locations.
6. If necessary, delays are simulated to compute new future output values.
7. Triggered or Mode X DUMP operations are completed
8. Triggered or Mode X OUTPUT operations are completed
9. Triggered or Mode X STOP operations are completed

This procedure is repeated until the simulation is terminated.

4.4 ERRORS

DDLSIM performs very extensive error checking. On detection of an error, an error message is printed. Whenever possible an attempt is made to pin-point the error. Error messages are printed in one of the two formats discussed below.

1. Error messages which can be associated with a card in the simulation deck resulting from syntax errors are printed in the following format. The card containing the error is printed (if not already printed) with a vertical bar (¦) placed under the column containing the error or the column next to the item containing error. A dotted line starting from the column next to vertical bar (¦) and terminating with the error message on right end of the page is printed.

   Example: <CL> CLOCK1(185) CLOCK (6:10)/2*100/

   ```
                                        ¦....................INVALID DELIMITER
   ```

   Processing of the remainder of the declaration and the simulation deck is continued by skipping to an appropriate position in the declaration.

2. Errors which cannot be easily associated with a particular card in the simulation deck are printed in this format. The error message preceded by three asterisks, i.e., '***' is printed on the left end of the line. Error messages printed in this format normally contain an error description with associated parameters, i.e., facility name with appropriate subscripts, simulation time, etc., to help in locating the error. Some of the error messages require more than one line.

   Example: ***RESPECIFICATION OF DATA FOR INPUT(1:5)

   ***AT TIME = 200

Errors are generally classified as fatal or non-fatal depending upon the nature, position and stage of simulation during which they occur. Fatal errors normally result in an immediate termination or abort of the simulation job. However, up to 10 fatal errors are allowed during the processing of the simulation deck for a simulation run. If any fatal errors were detected during the processing of the simulation deck, the entire simulation job is aborted. Whenever a simulation job is terminated due to fatal error(s) a message identifying the action is printed, i.e.,

***TO MANY FATAL ERRORS - SIMULATION TERMINATED.

Non-fatal errors do not cause the termination of the simulation job. In this regard they are warnings rather than errors.

DDLSIM performs complete syntax checking on the BEs and RTEs describing a digital system. Any errors detected during the processing of system description are treated as fatal errors. However, the simulation job is not terminated immediately. Since the errors detected during this stage cannot be easily associated with the DDL deck, they are printed using the second format described above. During the simulation stage complete error-checking is performed on the simulation process itself looking for errors such as:

1. invalid memory addressing,
2. instability in networks containing loops, and
3. attempts to input conflicting data on a facility.

5. EXAMPLES

This chapter provides some example DDL descriptions. The examples range from small synchronous circuits to a simple, but complete computer. These examples do not illustrate all the capabilities of DDL, but provide a good introduction to the user unfamiliar with DDL.

Example 1: A Serial Twos Complementer

The serial twos complementer uses the familiar copy/complement algorithm: starting from the least significant end of the number, copy the bits as they are till and including the first non-zero bit; complement the remaining bits till the most significant end. As an example,

```
0 0 1 0 1 0:1 0 0          Number
           :
1 1 0 1 0 1:1 0 0          Twos complement
           :
complement : copy
```

This algorithm is implemented using a shift register and right circulating its contents while copying or complementing as required. The number of shifts is equivalent to the number of bits in the register. A flip-flop can be used to store the copy or complement state.

Figure 5-1(a) shows the description of the serial twos complementer in DDL. The content of the six bit register R is to be replaced by its twos complement. Register C (3 bits) counts the number of shifts. S is a state flip-flop to indicate the copy or complement state. T is a control flip-flop to indicate RUN/STOP state for the complementer. The complementer waits for SW to be ON, to start complementing. There is a clock P. An OPerator ADD is described in lines 5-8. This is a 3 bit adder to increment the contents of the argument register by 1. The AUtomaton COMP has two states: a waiting state I, and a processing

state S1. Setting of SW is required for the transition to S1 state. In S1, the register R is circulated right 1 bit with the least significant bit copied or complemented, depending on the state of S being 0 or 1. If register C has reached a value of 5, the complementation is stopped by setting T to 0 and returning to state I. If C $\neq$ 5, COMP stays in S1 state and increments C. The FLag statement (line 13) sets the flags of the translator to provide the outputs at each of its six phases. Figure 5.1(b) shows these outputs. A detailed description of Figure 5.1(a) follows:

Line 1: The name of the system is COMPLEMENTER. Only the last 8 characters of this name are retained by DDLTRN. There is no period at the end of this line, since the system description is not complete yet.

Line 2: REgister R has 6 bits numbered 1 through 6, left to right; C has 3 bits numbered 2 through 0; S and T are single bit registers. C counts the shifts; S is the copy (0)/COMPLEMENT (1) state flip-flop. T is a flip-flop indicating that the complementing process is underway. It is not really required, but included to illustrate some DDL features. Period terminates the REgister description.

Line 3: A LAtch by name SW

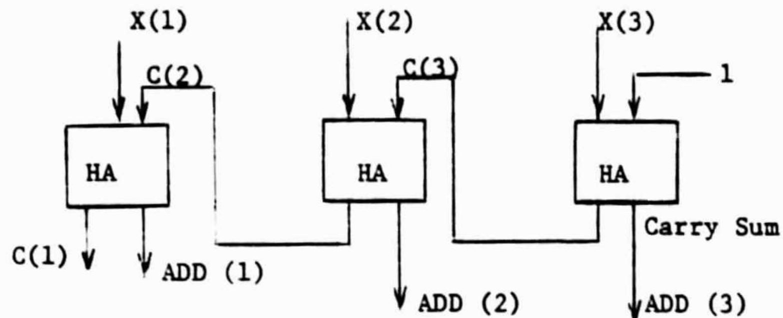
Line 4: A single phase CLock (Time) P.

Line 5: A special OPerator by name ADD. The output of the operator is a 3 bit number. The input is through the argument X (X is a formal parameter). No period to terminate, since the operator description is not complete yet.

Line 6: Declares the TErminal X to be of 3 bits and a new 3 bit register C. DDLTRN changes this name to C"1.

Line 7: Declares a new IDentifier for the concatenation of the last two bits of C and a 1.

Line 8: Declares the CARRY and SUM bits of an adder consisting of 3 half adders. C has the carry bits from each half adder, CC consists of carry bits from previous stages along with a 1 for the least significant bit. ADD consists of the SUM bits output. Note that ADD is the name of the operator, which is simply an ADD 1 circuit. The circuit implied (modelled) by lines 5-8 is:



Note the periods at the end of line 8. The first terminates <BO> and the second terminates <OP>.

Line 9: AUtomaton COMP is controlled by the clock P. Since COMP is not subscripted (by parenthesis) it is assumed to be having only two states (1 bit). (If there are more than 2 states, then the number of bits required for state identification must be shown)

Line 10: STate (Step) I with identification 0. AUtomaton COMP waits in I till SW is 1. When SW is 1, T is set to 1, C and S are set to 0, and a transition is made to state S 1 (all in parallel). The period terminates I.

<u>Line 11</u>: State S1 with the designation 1. waits for T to be 1. If S is 1, R is circulated right one bit with the bit R(6) complemented; otherwise R is simply circulated. S receives R(6) if S = 0. Also in this state, the value of C is checked be equivalent to $5(=101_2)$. If C=5, T is set to 0 and a transition . is made; if not, C is incremented and S 1 state continues. T iods at the end of line 12 terminate the If .... THEN on C, S 1, S. .AU and SY declarations respectively.

<u>Line 13</u>: Sets the FLags of DDLTRN to output results of each of the six passes.

Figure 5.1(c) shows the input commands for DDLSIM. FLags for DDLSIM are set for decimal data input (4) and binary output (6) in Line 1. SW is initialized to 1 in line 2. Two values are read into R one each time state I is reached (line 3). An output trigger OUTTR is declared to be ON at the falling edge of clock P (line 4). The values of COMP, R, S, C, T are to be OUtput when OUTTR is ON and that of R when in I state (line 5). The simulation is started with <SI> in line 6. Figure 5.1(d) shows the simulation output. The TIME starts with the raising edge of clock. Each edge is a time event. At time 0, all registers are zeroed and the circuit is in state I. At 1 SW is set to 1. At 2, R receives 5. 12 more time slots (6 clock pulses) are required for R to have its twos complement (time 14). At time 16, the new value for R (20) is received and its twos complement is ready at time 28. Since all the inputs are exhausted, the simulator stops at time 29.

```
DIGITAL DESIGN LANGUAGE TRANSLATOR

 1:    <SY>COMPLEMENTER:
 2:    <RE>R(1:6),C(2:0),S,I.
 3:    <LA>SZ.
 4:    <TI>P.
 5:    <OP> ADD(3)*A*
 6:         <TE> X(3),C(3).
 7:            <TO> CL=(C(2:3)(1C1).
 8:          <RO> C=X*(C,ADD=X¬CC..
 9:    <AU>COMP:P:
10:          <ST>I(0):SP:I<-1,C<-0,S<-0,->S1.
11:           S1(1):I:)S1 R(1)<-TR(6),R(2:6)<-R(1:5) ;S<-R(6),R<-R(6)(R(1:5).,
12:            IF(2)*T(C(1)*C(6))I<-0,->I;C<-ADD*C*.....
13:    <FL>3,4,5,6,6.
```

FIGURE 5.1(a): DDLTRN INPUT

```
DIGITAL DESIGN LANGUAGE TRANSLATOR

     PASS1--FACILITIES IDENTIFIED



     DECLARED FACILITIES

          <SY> GENERATOR
          <RE> R(1:6)
               C(2:6)
               S(1:1)
               T(1:1)
          <LA> S*(1:1)
          <TI> P(1:1)
               <OP> ADD(1:3)
               <TE> X(1:3)                                    C -> C*1

                    C*1(1:3)
               <TE> CC(1:1)
               <AU> COMP
               <ST> I
                    S1


     DECLARED OPERATIONS

<SY> GENERATOR:
     <OP> ADD=X*
          <TE> C=X*CC, ADD=X*CC..

     <AU> COMP: P:
          <ST>
               I: SA: T<-1, C<-0, S<-0, ->S1.

               S1: T:
                    |S| R(1)<-TR(R), R(2:6)<-R(1:5); S<-R(6), R<-R(6)(R(1: 5).,
                    |T(2)*T(1)*C(6)| T<-0, ->T; C<-ADD(C+.....
```

Figure 5.1(b) DDLTRN Output

```
DIGITAL DESIGN LANGUAGE TRANSLATOR

     PASS2--SYNTAX REDUCED


<SY> LEVENTER: I=*/COMP'0, S1=*/COMP'1D1   , C"1=X*C"1(2.3)[1D1   , ADD=X@C"1(2:3)[1D1   ,
     <AU> COMP: P:
          ]1]
               ]SW] I<-1D1   , C<-0, S<-0, ->S1..,
          ]S1]
               ]T]
                    ]S] R(1)<-TR(6), R(2:6)<-R(1:5); S<-R(6), R<-R(6)[R(1:5).,
                    ]C(2)*TC(1)*C(0)] I<-0, ->1; C<-ADD  , X=C...., ., .
```

Figure 5.1(b)(Continued)

DIGITAL DESIGN LANGUAGE TRANSLATOR

```
    PASS3--COMPLEXITIES DISTRIBUTED

<SY> LEMENTER:
     I=*/COMP'0,
     S1=*/COMP'101   ,
     "1=I*SY,
     "2=S1*I,
     "3="2*S,
     "4="2*TS,
     "5="2*C(2)*TC(1)*C(0),
     "6="2*T(C(2)*TC(1)*C(0)),
      C"1=X*C"1(2:3)(101   ,
     ADD=(X*C"1(2:3)(101   ),
     ]P*"1]  I<-"1*101   ..
     ]P*"1]  C<-"1*0..
     ]P*"1]  S<-"1*0..
     ]P*"1]  COMP<-"1*101   ..
     ]P*"3]  R(1)<-"3*TR(6)..
     ]P*"3]  R(2:6)<-"3*R(1:5)..
     ]P*"4]  S<-"4*R(6)..
     ]P*"4]  R<-"4*R(6)(R(1:5)..
     ]P*"5]  I<-"5*0..
     ]P*"5]  COMP<-"5*0..
     ]P*"6]  C<-"6*ADD..
     X="6*C, .
     .
```

```
    PASS4--CONCATENATION REMOVED

<SY> LEMENTER:
     I=*/COMP'0,
     S1=*/COMP'101   ,
     "1=I*SY,
     "2=S1*I,
     "3="2*S,
     "4="2*TS,
     "5="2*C(2)*TC(1)*C(0),
     "6="2*T(C(2)*TC(1)*C(0)),
     C"1(1:2)=X(1:2)*C"1(2:3),
     C"1(3)=X(3)*101   ,
     ADD(1:2)=(X(1:2)*C"1(2:3)),
     ADD(3)=(X(3)*101   ),
     ]P*"1]  I<-"1*101   ..
     ]P*"1]  C<-"1*0..
     ]P*"1]  S<-"1*0..
     ]P*"1]  COMP<-"1*101   ..
     ]P*"3]  R(1)<-"3*TR(6)..
     ]P*"3]  R(2:6)<-"3*R(1:5)..
     ]P*"4]  S<-"4*R(6)..
     ]P*"4]  R(1)<-"4*R(6)..
     ]P*"4]  R(2:6)<-"4*R(1:5)..
     ]P*"5]  I<-"5*0..
     ]P*"5]  COMP<-"5*0..
     ]P*"6]  C<-"6*ADD..
     X="6*C, .
```

Figure 5.1(b): (Continued)

DIGITAL DESIGN LANGUAGE TRANSLATOR

PASS5--OPERATIONS GATHERED

```
<SY> LEMENTER:
     I=*/COMP'0,
     S1=*/COMP'101   ,
     "1=I*S.,
     "2=S1*I,
     "3="2*S,
     "4="2*TS,
     "5="2*C(2)*TC(1)*C(0),
     "6="2*T(C(2)*TC(1)*C(0)),
     C"1(1:2)=X(1:2)*C"1(2:3),
     C"1(3)=X(3)*101   ,
     ADD(1:2)=(X(1:2)aC"1(2:3)),
     ADD(3)=(X(3)a101   ),
     ]P*"1 + P*"5] I<-"1*101    + "5*0.,
     ]P*"1 + P*"6] C<-"1*0 + "6*ADD.,
     ]P*"1 + P*"4] S<-"1*0 + "4*R(6).,
     ]P*"1 + P*"5] COMP<-"1*101    + "5*0.,
     ]P*"3 + P*"4] R(1)<-"3*TR(6) + "4*R(6).,
     ]P*"3 + P*"4] R(2:6)<-"3*R(1:5) + "4*R(1:5).,
     X="6*C, .
```

PASS6--SUBFACILITIES DISJOINED

```
<SY> LEMENTER:
     I=*/COMP'0,
     S1=*/COMP'101   ,
     "1=I*S.,
     "2=S1*I,
     "3="2*S,
     "4="2*TS,
     "5="2*C(2)*TC(1)*C(0),
     "6="2*T(C(2)*TC(1)*C(0)),
       C"1(1:2)=X(1:2)*C"1(2:3),
     C"1(3)=X(3)*101   ,
       ADD(1:2)=(X(1:2)aC"1(2:3)),
     ADD(3)=(X(3)a101   ),
     ]P*"1 + P*"5] I<-"1*101    + "5*0.,
     ]P*"1 + P*"6] C<-"1*0 + "6*ADD.,
     ]P*"1 + P*"4] S<-"1*0 + "4*R(6).,
     ]P*"1 + P*"5] COMP<-"1*101    + "5*0.,

     ]P*"3 + P*"4] R(1)<-"3*TR(6) + "4*R(6).,
     ]P*"3 + P*"4] R(2:6)<-"3*R(1:5) + "4*R(1:5).,
     X="6*C, .
```

Figure 5.1(b): Continued

DIGITAL DESIGN LANGUAGE TRANSLATOR

FACILITY TABLE

1	LEMENTER	1	1	1	-1	0	339	0	0
2	R	1	6	6	-6	0	0	0	0
3	C	2	0	-3	-6	0	227	358	362
4	S	1	1	1	-6	0	235	291	295
5	I	1	1	1	-6	0	216	321	325
6	SV	1	1	1	-8	0	0	0	0
7	P	1	1	1	-5	0	0	0	0
8	ACC	1	3	3	-9	0	0	0	0
9	*	1	3	3	-9	0	191	0	194
10	"1	1	1	1	-9	0	213	0	219
11	C"1	1	3	3	-9	0	0	0	0
12	"2	1	1	1	-9	0	252	0	258
13	COMP	1	1	1	-6	0	244	330	340
14	1	17	0	1	-13	0	196	0	201
15	S1	16	1	1	-13	0	203	0	208
16	101		1	1	-17	0	0	0	0
17	0		0	1	-17	0	0	0	0
18	"3	1	1	1	-9	0	255	0	264
19	"4	1	1	1	-9	0	280	0	287
20	"5	1	1	1	-9	0	304	0	318
21	"6	1	1	1	-9	0	337	0	354
22		0	0	0	0	0	0	0	0
23		0	0	0	0	0	0	0	0
24	C"1	3	3	1	11	0	513	0	519
25	C"1	1	2	2	11	0	521	0	531
26	ALL	3	3	1	8	0	489	0	497
27	ALL	1	2	2	9	0	499	0	511
28	K	2	6	5	2	0	456	463	454
29	K	1	1	1	2	0	480	487	478

Figure 5.1(b) (Continued)

...NAL PAGE IS
...OR QUALITY

```
DIGITAL DESIGN LANGUAGE SIMULATOR      VERSION WSEC 1979        SIMULATION RUN  1

        1:      <EL>4,6
        2:      <IO>S/Z/1
        3:      <RE>I/R/5,20
        4:      <TR>OUTTR/T0/
        5:      <OU>OUTTR/COVF,R,S,C,T/,I/R/
        6:      <SI>
```

Figure 5.1(c) DDLSIM Input

```
        C
        O
        V
   TIME P      R     S   C   T     R

     0  0 000000 0 000 0 000000
     2  1 000101 0 000 1
     4  1 100010 1 001 1
     6  1 110001 1 010 1
     8  1 011000 1 011 1
    10  1 101100 1 100 1
    12  1 110110 1 101 1
    14  0 111011 1 101 0 111011
    16  1 010100 0 000 1
    18  1 001010 0 001 1
    20  1 000101 0 010 1
    22  1 100010 1 011 1
    24  1 110001 1 100 1
    26  1 011000 1 101 1
    28  0 101100 1 101 0 101100

 NO    OF FILE REACHED ON INPUT
 INDICATION TERMINATED AT TIME =           29
```

Figure 5.1(d) DDLSIM Output

Example 2: The Serial Twos Complementer (variation 1)

Figure 5.2(a) illustrates another version of the twos complementer. Two operators are used. The six bit COM operator circulates register X. The bit fed into X(1) during circulation is either X(6) or $\overline{X(6)}$ depending on the value of Y is 0 or 1. respectively. The CNTUP operator is the same as the ADD operator in example 1. This version just illustrates the use of operators. Figures 5.2(b) shows the DDLTRN output, 5.2(C) shows DDLSIM input and 5.2(d) shows the DDLSIM output.

```
[illegible]

 1:  <SY>COMPLEMENTER:
 2:  <RE>R(1:6),C(2:0),S,I.
 3:  <IS>S.
 4:  <II>R.
 5:  <RP>(CR(6)SX,YX
 6:     <IE>X(6),Y.
 7:  <RO>COM(1)=(1Y)TX(6);X(6).),COM(2:6)=X(1:5)..
 8:  <OP>CNTOP(5)?X*
 9:      <IE> Y(3),C(3).
10:         <IO> CC=(C(2:3)(1)).
11:     <AO>C=X*CC,CNTOP=X*CC..
12:   <AD> COMP:P:
13:    <SI>I(0):S:I<-1,C<-0,S<-0,->S1.
14:   S1(1):T:R<-CO R,S*, I+S) S<-R(6).,)C(2)*C(1)*C(0)) I<-0,->I;
15:          C<-CNTOPSC*.....
16:   <EL>3,4,5,6,7.
```

FIGURE 5.2(a) DDLTRN INPUT

```
DIGITAL DESIGN LANGUAGE TRANSLATOR

    PASS1--FACILITIES IDENTIFIED



   DECLARED FACILITIES

         <SY> LEVENTER
         <RE> R(1:n)
              C(2:0)
              S(1:1)
              T(1:1)
         <LA> SA(1:1)
         <TI> P(1:1)
              <TR> CNX(1:n)
              <TR> X(1:n)
                   Y(1:1)
              <OP> COMP(1:3)
                                                      X -> X"1
              <TE> X"1(1:3)
                                                      C -> C"1
                   C"1(1:3)
              <IO> CC(1:1)
              <AU> COMP
              <ST> I
                   S1


   DECLARED OPERATIONS

<SY> LEVENTER:
     <TR> CNX, Y:
          <RO> CNX(1)=(
               Y1 *X(N): X(N).), CNX(2: N)=X(1: S)..

     <TR> COMP:X:
          <RO> C=X*CC, COMP=X*CC..

     <AU> COMP: P:
          <ST>
               I: S*: I<=1, C<=0, S<=0, =>S1.

               S1: I: R<=CNX, SA,
                     1TS1 S<=R(N).,
                     ((2)*T(1)*R(1)) I<=0, =>I: C<=C [illegible].....
```

FIGURE 5.2(b): DDLTRN OUTPUT

ORIGINAL [illegible]
OF POOR Q[illegible]

```
PASS2--SELECT REQUIRED

<SY> LEMENTER: T=*/COMP'0, S1=*/COMP'101  , COM(1)=C
  ENT TX(6): X(6),1, COM(2:6)=X(1:5), C"1=X"1*C"1(2:3)[101  , CNTUP=X"1*C"1(2:3)[101  ,
  <AD> CODE: F:
        1]]
              ]SA] T<-101  , C<-0, S<-0, ->S1..,
        ]S1]
              ]1] R<-COM  , X=R, Y=S,
                   ]1S] S<-R(6).,
                   ]C(2)*TC(1)*C(0)] T<-0, ->T: C<-CNTUP , X"1=C...., .. .
```

```
PASS3--CONDITIONS DISTRIBUTED

<SY> LEMENTER:
     T=*/COMP'0,
     S1=*/COMP'101  ,
     "1=TY, "2=T*S',
     "3=S1*T,
     "4="3*TS,
     "5="3*C(2)*TC(1)*C(0),
     "6="3*T(C(2)*TC(1)*C(0)),
     COM(1)=(Y*TX(6) + "1*X(6)),
     COM(2:6)=X(1:5),
     C"1=X"1*C"1(2:3)[101  ,
     CNTUP=(X"1*C"1(2:3)[101   ),
     ]P*"2] T<-"2*101   ..
     ]P*"2] C<-"2*0..
     ]P*"2] S<-"2*0..
     ]P*"2] COMP<-"2*101   ..
     ]P*"3] R<-"3*COM..
     X="3*R, Y="3*S,
     ]P*"4] S<-"4*R(6)..
     ]P*"5] T<-"5*0..
     ]P*"5] COMP<-"5*0..
     ]P*"6] C<-"6*CNTUP..
     X"1="6*C,
     . .
```

```
PASS4--CONCATENATION RECOVER

<SY> LEMENTER:
     T=*/COMP'0,
     S1=*/COMP'101  ,
     "1=TY, "2=T*S',
     "3=S1*T,
     "4="3*TS,
     "5="3*C(2)*TC(1)*C(0),
     "6="3*T(C(2)*TC(1)*C(0)),
     COM(1)=(Y*TX(6) + "1*X(6)),
     COM(2:6)=X(1:5),
     C"1(1:2)=X"1(1:2)*C"1(2:3),
     C"1(3)=X"1(3)*101  ,
     CNTUP(1:2)=(X"1(1:2)*C"1(2:3)),
     CNTUP(3)=(X"1(3)*101   ),
     ]P*"2] T<-"2*101   ..
     ]P*"2] C<-"2*0..
     ]P*"2] S<-"2*0..
     ]P*"2] COMP<-"2*101   ..
     ]P*"3] R<-"3*COM..
     X="3*R, Y="3*S,
     ]P*"4] S<-"4*R(6)..
     ]P*"5] T<-"5*0..
     ]P*"5] COMP<-"5*0..
     ]P*"6] C<-"6*CNTUP..
     X"1="6*C,
     .
```

FIGURE 5.2(b) (Continued)

```
PASS5--[illegible]

<SY> [illegible]:
     I=*/COMP*0,
     S1=*/COMP*101   ,
     "1=TY, "2=I*SP,
     "3=S1*T,
     "4="3*TS,
     "5="3*C(2)*TC(1)*C(0),
     "6="3*T(C(2)*TC(1)*C(0)),
     COMP(1)=(Y*TX(6) + "1*X(6)),
     COMP(2:6)=X(1:5),
     C"1(1:2)=X"1(1:2)*C"1(2:3),
     C"1(3)=X"1(3)*101   ,
     [illegible](1:2)=(X"1(1:2)*C"1(2:3)),
     [illegible](3)=(X"1(3)*101   ),
     ]P*"2 + P*"5] T<-"2*101    + "5*0.,
     ]P*"2 + P*"6] C<-"2*0 + "6*CTOP.,
     ]P*"2 + P*"4] S<-"2*0 + "4*F(6).,
     ]P*"2 + P*"5] COMP<-"2*101    + "5*0.,
     ]P*"3] [illegible]<-"3*COMP.,
     X="3*F, Y="3*S,
     X"1="[illegible],
     .
```

```
PASS6--[illegible]

<SY> [illegible]:
     I=*/COMP*0,
     S1=*/COMP*101   ,
     "1=TY, "2=I*SP,
     "3=S1*T,
     "4="3*TS,
     "5="3*C(2)*TC(1)*C(0),
     "6="3*T(C(2)*TC(1)*C(0)),
       COMP(1)=(Y*TX(6) + "1*X(6)),
     COMP(2:6)=X(1:5),
       C"1(1:2)=X"1(1:2)*C"1(2:3),
     C"1(3)=X"1(3)*101   ,
       [illegible](1:2)=(X"1(1:2)*C"1(2:3)),
     CTOP(3)=(X"1(3)*101   ),
     ]P*"2 + P*"5] T<-"2*101    + "5*0.,
     ]P*"2 + P*"6] C<-"2*0 + "6*CTOP.,
     ]P*"2 + P*"4] S<-"2*0 + "4*F(6).,
     ]P*"2 + P*"5] COMP<-"2*101    + "5*0.,
     ]P*"3] F<-"3*COMP.,
     X="3*F, Y="3*S,
     X"1="[illegible],
     .
```

FIGURE 5.2(b) (Continued)

ORIGINAL PAGE IS
OF POOR QUALITY

FACILITY TABLE

1	[illegible]	1	1	1	-1	0	3?1	0	0
2	B	1	6	6	-6	0	296	302	306
3	C	2	0	-3	-6	0	268	380	384
4	S	1	1	1	-6	0	276	321	325
5	T	1	1	1	-6	0	257	343	347
6	S	1	1	1	-8	0	0	0	0
7	F	1	1	1	-5	0	0	0	0
8	COS	1	6	6	-9	0	0	0	0
9	X	1	6	6	-9	0	218	0	137
10	Y	1	1	1	-9	0	220	0	140
11	CSTEP	1	3	3	-9	0	0	0	0
12	"1	1	1	1	-9	0	245	0	250
13	X"1	1	3	3	-9	0	222	0	225
14	"2	1	1	1	-9	0	254	0	260
15	C"1	1	3	3	-9	0	0	0	0
16	"3	1	1	1	-9	0	293	0	299
17	COMP	1	1	1	-6	0	285	352	362
18	I	21	0	1	-13	0	227	0	232
19	SI	20	1	1	-13	0	234	0	239
20	11 1		1	1	-17	0	0	0	0
21	0		0	1	-17	0	0	0	0
22	"4	1	1	1	-9	0	311	0	318
23	"5	1	1	1	-9	0	326	0	340
24	"6	1	1	1	-9	0	359	0	376
25	CSTEP	3	3	1	11	0	455	0	463
26	CSTEF	1	2	2	11	0	465	0	477
27		0	0	0	0	0	0	0	0
28		0	0	0	0	0	0	0	0
29	C"1	3	3	1	15	0	479	0	485
30	C"1	1	2	2	15	0	487	0	497
31	[illegible]	2	6	5	8	0	431	0	437
32	[illegible]	1	1	1	8	0	439	0	453

FIGURE 5.2(b) (Continued)

ORIGINAL PAGE IS OF POOR QUALITY

```
DIGITAL DESIGN LANGUAGE SIMULATOR      VERSION DEC 1979      SIMULATION RUN  1      PAGE  1

    1:      <FL>4,6
    2:      <TP>SW/1
    3:      <RF>I/R/5,20
    4:      <TR>OUTTR/TR/
    5:      <OG>OUTTR/(COWR,R,S,C,TZ,I/R/
    6:      <ST>
```

FIGURE 5.2(c) DDLSIM INPUT

```
         C
         O
         W
   TIME  R     R     S   C   T     R

      0  0 000000 0 000 0 000000
      2  1 000101 0 000 1
      4  1 100010 1 001 1
      6  1 110001 1 010 1
      8  1 011000 1 011 1
     10  1 101100 1 100 1
     12  1 110110 1 101 1
     14  0 111011 1 101 0 111011
     16  1 010100 0 000 1
     18  1 001010 0 001 1
     20  1 000101 0 010 1
     22  1 100010 1 011 1
     24  1 110001 1 100 1
     26  1 011000 1 101 1
     28  0 101100 1 101 0 101100

 END     OF FILE REACHED ON INPUT
 SIMULATION TERMINATED AT TIME =            29
```

FIGURE 5.2(d): DDLSIM OUTPUT

Example 3: Twos Complementer (variation 2)

Figure 5.3(a) shows a version of twos complementer description with the use of several AUtomata. Automaton CNT adds 1 to C, checks if it is 5 and sets DONE to 1 if C = 5. It is activated by COUNT. AUtomaton CMP is activated by CPT; performs the one bit circulation of R; sets COUNT to 1 to activate CNT. COMP is the controlling AUtomaton, activated by SW and in turn activates CMP in state S1 and waits for CCT to be 1 (for CNT completed) in S2. If DONE is 1, goes into wait state.

Figure 5.3(b) shows the DDLTRN output. Figure 5.3(c) and (d) show the DDLSIM input and output respectively. Note the effect of this version of description (AUtomata interaction) on simulation time.

```
 1:   <SY>COMPLEMENTER:
 2:   <VR>R(1:6),C(2:0),S,T.
 3:   <IF> COUNT,DONE,CPT.
 4:   <IF>CCT.
 5:   <II>P.
 6:   <LA>SA.
 7:   <OP> ADD(3):X-
 8:       <IF> X(3),C(3).
 9:           <IO> CC=LL(2:3)(101).
10:         <RO> C=X*CC,ADD=YACC..
11:   <AL>CNT:M:
12:   <ST>C1(0):COUNT:C<-ADD*CS,->C2.
13:           C2(1):CCT=1,1C(2)*C(1)*TC(0)] DONE=1;DONE=0.,->C1..
14:   .
15:   <AL>CMP:M:
16:     <ST>S0(0):CPT:
17:     1S1 R(1)<-TR(6),R(2:6)<-R(1:5);S<-R(6),R<-R(6)[R(1:5).,->S1.
18:         S1(1):COUNT=1,->S1..
19:   .
20:   <AL> COMP(2):M:
21:     <ST>I(0):S0:T<-1,C<-0,S<-0,->S1.
22:         S1(1):T:CPT=1,->S2.
23:          S2(2): CLT: 1DONE1->T;->S1....
24:   .
25:   <RL>3,4,5,6,4.
```

ORIGINA
OF POOR QUALITY

Figure 5.3(a):DDLTRN Input

```
PASS1--FACILITIES IDENTIFIED



[illegible] FACILITIES

      <SY> LENGTH
      <RE> R(1:6)
           C(2:0)
           S(1:1)
           T(1:1)
      <TE> COUNT(1:1)
           CODE(1:1)
           CFT(1:1)
           CCT(1:1)
      <II> P(1:1)
      <LA> SA(1:1)
           <OP> ACC(1:3)
           <TE> X(1:3)
                                                    C  => C"1
                C"1(1:3)
           <IC> CC(1:1)
           <AL> CNT
           <ST> C1
                C2
           <AL> CMP
           <ST> S0
                S1
           <AL> CONV
           <ST> T
                                                    S1  => S1"1
                S1"1
                S2
```

Figure 5.3(b) : DDLTRN Output

```
[illegible]

<[illegible]> [illegible]:
    <[illegible]> [illegible]
        <[illegible]> C=*CC, ADD=[illegible]..

    <AU> [illegible]: P:
        <S1>
            C1: COUNT: C<-ALPHCS, ->C2.

            C2: CC1=1,
                1C(2)*C(1)*1C(0)1 DONE=1; DONE=0.. ->C1...

    <AU> [illegible]: P:
        <S1>
            S0: CH1:
                1S1 R(1)<-1R(6), R(2: 6)<-R(1: 5); S<-R(6), R<-R(6)1R(1: 5).. ->S1.

            S1: COUNT=1, ->S0...

    <AU> LONP: P:
        <S1>

            I: S0: I<-1, C<-0, S<-[illegible], ->S1.

            S1*1: I: CF1=1, ->S2.

            S2: CLT:
                1DONE1 ->I; ->S1.....
```

Figure 5.3(b) : Continued

```
[illegible]

<SY> [illegible]: C1=*/CNT'0, C2=*/CNT'101  , S0=*/COMP'0, S1=*/COMP'101  , 1=*/COMP'0D2, S1"1=*/COMP'1D2,
    S2=*/COMP'2D2, C"1=X*C"1(2:3)[101  , ADD=X*C"1(2:3)[101  ,
    <CU> CNT: F:
        [C1]
            [(ADD)] C<-ADD , X=1, ->C2...
        [C2] C(1=101  ,
            ](C(2)*C(1)*TC(6)] DONE=101  ; DONE=0.. ->C1.. .

    <AU> UPF: F:
        [S0]
            ](+1]
                [S] R(1)<-TR(6), R(2:6)<-R(1:5); S<-R(6), P<-P(6)[R(1:5).. ->S1...
        [S1] COUNT=101  , ->S0.. .

    <AU> COMP: F:
        [1]
            [S0] I<-101  , C<-0, S<-0, ->S1"1...
        [S1"1]
            [1] CPT=101  , ->S2...
        [S2]
            ](T]
                [DONE] ->1; ->S1"1.... .. .
```

Figure 5.3(b) : Continued

```
PASS3==CONDITIONS DISTRIBUTED

<SY> LENGTHS:
     C1=*/CNT'0,
     C2=*/CNT'101    ,
     S0=*/CMP'0,
     S1=*/CMP'101    ,
     T=*/COMP'002,
     S1"1=*/COMP'102,
     S2=*/COMP'202,
     "1=C1*COUNT,
     "2=C2*C(2)*C(1)*TC(0),
     "3=C2*T(C(2)*C(1)*TC(0)),
     "4=S0*CPT,
     "5="4*S,
     "6="4*TS,
     "7=T*S  ,
     "8=S1"1*T,
     "9=S2*CCT,
     "10="9*CONF,
     "11="9*TCONF,
      C"1=X*C"1(2:3)[101    ,
     ADD=(X*C"1(2:3)[101    ),
     [H*"1] C<="1*ADD..
     X="1*C,
     [H*"1] CNT<="1*101    ..
     CC1=C2*101    ,
     [illegible]="2*101    ,
     [illegible]="3*0,
     [H*C2] CNT<=C2*0..
     [H*"5] R(1)<="5*T-(6)..
     [H*"5]  (2:6)<="5*R(1:5)..
     [H*"6] S<="6*R(6)..
     [H*"6] R<="6*-(6)[R(1:5)..
     [H*"4] C[illegible]<="4*101    ..
     C[illegible] T=S1*101    ,
     [H*S1] C[illegible]<=S1*0..
     [H*"7] T<="7*101    ..
     [H*"7] C<="7*0..
     [H*"7] S<="7*0..
     [H*"7] CONF<="7*102..
     CHT="8*101    ,
     [H*"8] CONF<="8*202..
     [H*"10] CONF<="10*002..
     [H*"11] CONF<="11*102..
     . .
```

```
PASS4==CON[illegible]

<SY> LENGTHS:
     C1=*/CNT'0,
     C2=*/CNT'101    ,
     S0=*/CMP'0,
     S1=*/CMP'101    ,
     T=*/COMP'002,
     S1"1=*/COMP'102,
     S2=*/COMP'202,
     "1=C1*COUNT,
     "2=C2*C(2)*C(1)*TC(0),
     "3=C2*T(C(2)*C(1)*TC(0)),
     "4=S0*CPT,
     "5="4*S,
     "6="4*TS,
     "7=T*S  ,
     "8=S1"1*T,
     "9=S2*CCT,
     "10="9*CONF,
     "11="9*TCONF,
     C"1(1:2)=X(1:2)*C"1(2:3),
     C"1(3)=X(3)*101    ,
     ADD(1:2)=(X(1:2)*C"1(2:3)),
     ADD(3)=(X(3)*101    ),
     [H*"1] C<="1*ADD..
     X="1*C,
     [H*"1] CNT<="1*101    ..
     CC1=C2*101    ,
     [illegible]="2*101    ,
     [illegible]="3*0,
     [H*C2] CNT<=C2*0..
     [H*"5] R(1)<="5*T-(6)..
     [H*"5] R(2:6)<="5*R(1:5)..
     [H*"6] S<="6*R(6)..
     [H*"6] R(1)<="6*-(6)..
     [H*"6] R(2:6)<="6*R(1:5)..
     [H*"4] C[illegible]<="4*101    ..
     C[illegible] T=S1*101    ,
     [H*S1] C[illegible]<=S1*0..
     [H*"7] T<="7*101    ..
     [H*"7] C<="7*0..
     [H*"7] S<="7*0..
     [H*"7] CONF<="7*102..
     CHT="8*101    ,
     [H*"8] CONF<="8*202..
     [H*"10] CONF<="10*002..
     [H*"11] CONF<="11*102..
     .
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 5.3(b) : Continued

```
PASS5--OPERATIONS GATHERER

<ST> [illegible]:
     C1=*/CNT'0,
     C2=*/CNT'101    ,
     S0=*/CNP'0,
     S1=*/CNP'101    ,
     T=*/COMP'012,
     S1"1=*/COMP'102,
     S2=*/COMP'202,
     "1=C1*[illegible],
     "2=C2*C(2)*C(1)*TC(0),
     "3=C2*T(C(2)*C(1)*TC(0)),
     "4=S0*CP1,
     "5="4*S,
     "6="4*TS,
     "7=T*S[illegible],
     "8=S1"1*T,
     "9=S2*CCT,
     "10="9*[illegible],
     "11="9*T[illegible],
     C"1(1:2)=X(1:2)*C"1(2:3),
     C"1(3)=X(3)*101    ,
     ADD(1:2)=(X(1:2)+C"1(2:3)),
     ADD(3)=(X(3)+101   ),
     IP*"1 + P*"7] C<-"1*ADD + "7*C.,
     F="1*C,
     IP*"1 + F*C2] CNT<-"1*101    + C2*0.,
     CCT=C2*101    ,
     DONE="2*101    + "3*0,
     IP*"5 + P*"6] R(1)<-"5*R(6) + "6*R(6).,
     IP*"5 + P*"6] R(2:5)<-"5*R(1:5) + "6*R(1:5).,
     IP*"6 + P*"7] S<-"6*R(6) + "7*C.,
     IP*"4 + P*S1] CNP<-"4*101    + S1*0.,
     COMP1=S1*101    .
     IP*"7] T<-"7*101    ..
     IP*"7 + P*"8 + P*"10 + P*"11] COMP<-"7*102 + "8*202 + "10*002 + "11*102..
     C11="8*101    ,
     .
```

Figure 5.3(b) : Continued

ORIGINAL PAGE IS
OF POOR QUALITY

```
PASS6--SUBFACILITIES GENERATED

<5Y> GENERATOR:
     C1=*/CNT'0,
     C2=*/CNT'101   ,
     S0=*/CMP'0,
     S1=*/CMP'101   ,
     I=*/COMP'0D2,
     S1"1=*/COMP'1D2,
     S2=*/COMP'2D2,
     "1=C1*COUNT,
     "2=C2*C(2)*C(1)*TC(0),
     "3=C2*T(C(2)*C(1)*TC(0)),
     "4=S0*CPT,
     "5="4*S,
     "6="4*TS,
     "7=I*S*,
     "8=S1"1*I,
     "9=S2*CCT,
     "10="9*COMF,
     "11="9*TCOMF,
       C"1(1:2)=X(1:2)*C"1(2:3),
     C"1(3)=X(3)*101   ,
       COD(1:2)=(X(1:2)*C"1(2:3)),
     COD(3)=(X(3)*101   ),
     IF*"1 + F*"7] C<-"1*ADD + "7*0..
     X="1*C,
     IF*"1 + F*C2] CCT<-"1*101    + C2*0..
     CCT=C2*101   ,
     CONF="2*101    + "4*0,

     IF*"5 + F*"6] R(1)<-"5*TR(6) + "6*R(6)..
     IF*"5 + F*"6] R(2:6)<-"5*R(1:5) + "6*R(1:5)..
     IF*"6 + F*"7] S<-"6*R(6) + "7*0..
     IF*"4 + F*S1] COP<-"4*101    + S1*C..
     COUNT=S1*101   ,
     IF*"7] I<-"7*101   ..
     IF*"7 + F*"8 + F*"10 + F*"11] COMF<-"7*1D2 + "8*2D2 + "10*0D2 + "11*1D2..
     CPT="8*101   ,
     .
```

Figure 5.3(b) : Continued

FACILITY TABLE

No.	Name	C1	C2	C3	C4	C5	C6	C7	C8
1	LEVENTER	1	1	1	-1	0	592	0	0
2	M	1	6	6	-6	1	0	0	0
3	C	2	0	-3	-6	0	370	530	534
4	S	1	1	1	-6	0	477	538	542
5	1	1	1	1	-6	0	516	522	526
6	COUNT	1	1	1	-9	0	192	0	199
7	DONE	1	1	1	-9	1	101	0	115
8	CPT	1	1	1	-9	1	239	0	244
9	LCT	1	1	1	-9	0	78	0	83
10	F	1	1	1	-5	0	0	0	0
11	SA	1	1	1	-8	0	0	0	0
12	ADC	1	3	3	-9	0	0	0	0
13	*	1	3	3	-9	0	246	0	70
14	002		0	2	-17	0	0	0	0
15	C"1	1	3	3	-9	0	0	0	0
16	102		1	2	-17	0	0	0	0
17	CPT	1	1	1	-6	0	384	436	440
18	C1	29	0	1	-13	0	315	0	320
19	C2	28	1	1	-13	0	322	0	327
20	CSF	1	1	1	-6	0	494	508	512
21	S0	29	0	1	-13	0	329	0	334
22	S1	28	1	1	-13	0	336	0	341
23	CCSF	1	2	2	-6	0	544	602	606
24	T	14	0	1	-13	0	343	0	348
25	202		2	2	-17	0	0	0	0
26	S1"1	16	1	1	-13	0	350	0	355
27	S2	25	2	1	-13	0	357	0	362
28	101		1	1	-17	0	0	0	0
29	0		0	1	-17	0	0	0	0
30	"1	1	1	1	-9	0	367	0	373
31	"2	1	1	1	-9	0	394	0	[illegible]
32	"3	1	1	1	-9	1	413	0	[illegible]
33	"4	1	1	1	-9	1	441	0	447
34	"5	1	1	1	-9	0	444	0	453
35	"6	1	1	1	-9	0	[illegible]	0	[illegible]
36	"7	1	1	1	-9	0	[illegible]	0	[illegible]
37	"8	1	1	1	-9	1	552	0	[illegible]
38	"9	1	1	1	-9	1	569	0	575
39	"10	1	1	1	-9	1	572	0	581
40	"11	1	1	1	-9	0	590	0	597
41		0	0	0	0	0	0	0	0
42		0	0	0	0	1	1	0	0
43	C"1	3	3	1	15	1	755	0	761
44	C"1	1	2	2	15	1	763	0	773
45	ADC	3	3	1	12	1	731	0	739
46	ADC	1	2	2	12	1	741	0	753
47	M	2	6	5	2	1	[illegible]	705	[illegible]
48	M	1	1	1	2	1	722	724	[illegible]

Figure 5.3(b) · Continued

DIGITAL DESIGN LANGUAGE SIMULATOR VERSION NSEC 1979

```
1:  <FL>4,6
2:  <I >9N/1
4:  <RE>I/R/5,20
6:  <IW>OUTTR/TR/
5:  <CU>OUTTR/CONT,R,S,C,T,CNT,COUNT,[illegible],CRT,CNT,CYR/,I/R/
6:  <SI>
```

Figure 5.3(c) : DDLSIM Input

TIME	W	R	S	C	T	I	T	F	T	T	F	R
0	00	000000	0	000	0	0	0	0	0	0	0	000000
2	01	001101	0	000	1	0	0	0	1	0	0	
4	10	100010	1	000	1	0	1	0	0	0	1	
6	10	100010	1	001	1	1	0	0	0	1	0	
8	01	100010	1	001	1	0	0	0	1	0	0	
10	10	110001	1	001	1	0	1	0	0	0	1	
12	10	110001	1	010	1	1	0	0	0	1	0	
14	01	110001	1	010	1	0	0	0	1	0	0	
16	10	011000	1	010	1	0	1	0	0	0	1	
18	10	011000	1	011	1	1	0	0	0	1	0	
20	01	011000	1	011	1	0	0	0	1	0	0	
22	10	101100	1	011	1	0	1	0	0	0	1	
24	10	101100	1	100	1	1	0	0	0	1	0	
26	01	101100	1	100	1	0	0	0	1	0	0	
28	10	110110	1	100	1	0	1	0	0	0	1	
30	10	110110	1	101	1	1	0	0	0	1	0	
32	01	110110	1	101	1	0	0	0	1	0	0	
34	10	111011	1	101	1	0	1	0	0	0	1	
36	10	111011	1	110	1	1	0	1	0	1	0	
38	00	111011	1	110	1	0	0	0	0	0	0	111011
40	01	010100	0	000	1	0	0	0	1	0	0	
42	10	001010	0	000	1	0	1	0	0	0	1	
44	10	001010	0	001	1	1	0	0	0	1	0	
46	01	001010	0	001	1	0	0	0	1	0	0	
48	10	000101	0	001	1	0	1	0	0	0	1	
50	10	000101	0	010	1	1	0	0	0	1	0	
52	01	000101	0	010	1	0	0	0	1	0	0	
54	10	100010	1	010	1	0	1	0	0	0	1	
56	10	100010	1	011	1	1	0	0	0	1	0	
58	01	100010	1	011	1	0	0	0	1	0	0	
60	10	110001	1	011	1	0	1	0	0	0	1	
62	10	110001	1	100	1	1	0	0	0	1	0	
64	01	110001	1	100	1	0	0	0	1	0	0	
66	10	011000	1	100	1	0	1	0	0	0	1	
68	10	011000	1	101	1	1	0	0	0	1	0	
70	01	011000	1	101	1	0	0	0	1	0	0	
72	10	101100	1	101	1	0	1	0	0	0	1	
74	10	101100	1	110	1	1	0	1	0	1	0	
76	00	101100	1	110	1	0	0	0	0	0	0	101100

[illegible] FILE READ [illegible]
SIMULATION TERMINATED AT TIME = 77

Figure 5.3(d) : DDLSIM Output

ORIGINAL PAGE IS OF POOR QUALITY

Example 4: MULTIPLIER [35]

A MULTIPLIER unit that calculates the product of two 8-bit numbers is described in Figure 5.4(a). A listing of the deck used for simulating the MULTIPLIER system along with the simulation report is given in Figure 5.4(b). The <FLag> declaration in the simulation deck specifies that all data-values specified without radix specification be interpreted in decimal (Flag 4), and that output values be printed in binary (Flag 6). The control unit MPY of the system waits idly in state S1 until it receives a START command. A <INitialize> declaration is used to initialize the START signal to 1 and start the MULTIPLIER unit. On receiving the START command in state S1, the control unit proceeds to load the R register with the multiplicand obtained from the BUS and proceeds to state S2. In state S2 the B register is loaded with the multiplier obtained from the BUS. A triggered READ operation with state terminal S1 as the triggering signal is used to supply the BUS with the multiplicand. During simulation, whenever the control unit reaches state S1, the BUS is supplied with a new value of the multiplicand. The multiplier is supplied to the BUS in a similar manner with another triggered READ operation using state terminal S2 as the triggering signal. After loading the multiplicand and the multiplier, the control unit proceeds to state S3. In state S3 the multiplicand is added to the partial product, if the multiplier bit is a logic 1. The control proceeds to state S4 in any case. The A and B registers are shifted right together and the multiplication cycle counter MCOUNT is incremented. If the count has been completed, status line DONE is set to logic 1 and the control unit returns to its idle state S1. If not all bits of the multiplier have been tested, the control unit returns to state S3.

A triggering signal OUTTR defined using a <TRigger> declaration is used in a triggered OUTPUT operation to control the printing of the values for MPY, MCOUNT, A, and B. These values are printed in binary on every trailing edge of the clock P signal. Another triggered OUTPUT operation using state terminal S1 as the triggering signal controls the printing of the values for the multiplicand, multiplier and the final product. Note that these values are printed only once, i.e., when the final product is available, during a given multiplication operation. The two output lists printed with different frequency make the simulation report more informative and readable. Since no <CLock> declaration is included in the simulation deck, default values are used for P, W, and θ. Note that for a single simulation run a <SImulate> declaration is not required. Since an EOF condition is expected no explicit <STop> declaration is included in the simulation deck to terminate the simulation.

```
 1:   <CO>MULTIPLIER<
 2:   <SY>MULTIPLIER:<IN>P.<RE>A(0:8),B(8),R(8),MCOUNT(3).
 3:   <RE>ZERO,ONE.
 4:   <TE>START,BUS(8),DONE.
 5:   <TE>SUM(8),COUT(8),CSUM(3),CCOUT(3).
 6:   <ID>CIN=COUT(2:8)(ZERO.
 7:   <ID>CCIN=CCOUT(2:3)(ONE.
 8:   <BO>COUT=R*A(1:8)+R*CIN+A(1:8)*CIN,SUM=R@A(1:8)@CIN,
 9:     CCOUT=MCOUNT*CCIN,CSUM=MCOUNT@CCIN.
10:   <AU>MPY(2):P:
11:   <ST>S1(0):START:R<-BUS,MCOUNT<-0,ZERO<-0,ONE<-1,->S2.
12:     S2(1):B<-BUS,A<-0,->S3.
13:    S3(2):]B(8)]A<-COUT(1)(SUM.,->S4.
14:    S4(3):A(1:8)(B<-A(R(1:7),A(0)<-0,
15:   MCOUNT<-CSUM,]*/MCOUNT]DONE=1,->S1;->S3.....
16:   <FL>3,4,5,6,8.
```

Figure 5.4(a) : DDLTRN Input

```
DIGITAL DESIGN LANGUAGE SIMULATOR     VERSION MSFC 1979

        1:      <FL>4,6
        2:      <IN>START/1
        3:      <RE>S1/BUS/6.10
        4:      <RE>S2/BUS/5.13
        5:      <TR>OUTTR/TP/
        6:      <OU>OUTTR/MPY,MCOUNT,A,B/,
        7:      S1/R,BUS,A(1:8),6
        8:      <SI>
```

Figure 5.4(b) : DDLSIM Input

DIGITAL DESIGN LANGUAGE SIMULATOR VERSION MSFC 1979

TIME	MPY	MCOUNT	A	B	R	BUS	M1:8	D
0	00	000	000000000	00000000	00000000	00000000	00000000	00000000
2	01	000	000000000	00000000				
4	10	000	000000000	00000101				
6	11	000	000000110	00000101				
8	10	001	000000011	00000010				
10	11	001	000000011	00000010				
12	10	010	000000001	10000001				
14	11	010	000000111	10000001				
16	10	011	000000011	11000000				
18	11	011	000000011	11000000				
20	10	100	000000001	11100000				
22	11	100	000000001	11100000				
24	10	101	000000000	11110000				
26	11	101	000000000	11110000				
28	10	110	000000000	01111000				
30	11	110	000000000	01111000				
32	10	111	000000000	00111100				
34	11	111	000000000	00111100				
36	00	000	000000000	00011110	00000110	00000101	00000000	00011110
38	01	000	000000000	00011110				
40	10	000	000000000	00001101				
42	11	000	000001010	00001101				
44	10	001	000000101	00000110				
46	11	001	000000101	00000110				
48	10	010	000000010	10000011				
50	11	010	000001100	10000011				
52	10	011	000000110	01000001				
54	11	011	000010000	01000001				
56	10	100	000001000	00100000				
58	11	100	000001000	00100000				
60	10	101	000000100	00010000				
62	11	101	000000100	00010000				
64	10	110	000000010	00001000				
66	11	110	000000010	00001000				
68	10	111	000000001	00000100				
70	11	111	000000001	00000100				
72	00	000	000000000	10000010	00001010	00001101	00000000	10000010

END OF FILE REACHED ON INPUT
SIMULATION TERMINATED AT TIME = 73

ORIGINAL PAGE IS OF POOR QUALITY

Figure 5.4(b) : DDLSIM Output

Example 5: MINICOMPUTER [52]

A description of a simple minicomputer is given in Figure 5.5. The details of the minicomputer are given in the Appendix. Lines 2-4 in Figure 5.5 describe the registers. Lines 5 declares a memory bus. Line 6 declares a START latch. Line 7 declares a four phase clock. Lines 8-11 declare a Increment (by 1) circuit. Lines 12-16 declare a 12 bit adder. Lines 18-19 are CPU initialization. Lines 20-23 show the FETCH cycle. Lines 24-25 show the DEFER state for Indirect Address calculation. Lines 26-27 show the OPCODE decoding. Lines 28-43 show the microoperations for each instruction.

```
 1:   <SY>MINI:
 2:   <RE>MAR(0:7),MBR(0:11),PC(0:7),ACC(0:11), X(0:11).
 3:   <RE>IR(0:11)=OP(3)|IBIT|ADR(8),RUN.
 4:   <RE>M(256:12).
 5:   <TE>MBUS(12).
 6:   <LA>START.
 7:   <TI>P(4).
 8:   <OP>CNTUP(8)$X$
 9:        <TE>X(8),C(8).
10:        <ID>CC=(C(2:8)|1B1).
11:        <BO>C=X*CC,CNTUP=X@CC..
12:   <OP>ADD(12)$X,Y$
13:        <TE>X(12),C(12),Y(12),COUT(12).
14:        <ID>CIN=COUT(2:12)!0B1.
15:        <BO>COUT=X*Y+X*CIN+Y*CIN,
16:             ADD=X@Y@CIN..
17:   <AU>CPU(4):P(1)+P(2)+P(3)+P(4):
18:         <ST>IN(0):START:ACC<=0,MAR<=PC,MBR<=0,X<=0,
19:                       RUN<= 1,->FE.
20:              FE(1):RUN:|P(1)|MAR<=PC.,|P(2)|PC<=CNTUP:PC.,
21:                         MBUS=M(MAR),MBR<=MBUS.,|P(3)|IR<=MBR.,
22:                         |P(4)||OP(1)*OP(2)*OP(3)|RUN<=0,->IN.;
23:                          !IB[1]->DEF;->EX....
24:              DEF(2):|P(1)|MAR<=ADR.,|P(2)|MBUS=M(MAR),MBR<=MBUS.,
25:                     |P(3)|ADR<=MBR(4:11).,|P(4)|->EX..
26:                EX(3)::OP#0B3->AND#1B3->ADD#2B3->ISZ#3B3->DCA
27:                       #4B3->JSR#5B3->JMP#6B3->RET..
28:               AND(4):|P(1)|X<=ACC.,|P(2)|MAR<=ADR.,|P(3)|MBUS=M(MAR),
29:                       MBR<=MBUS.,
```

Figure 5.5(a): Minicomputer Description

```
30:              ]P(4)]]OP(3)]ACC<=MBR*A;ACC<=ACC+MBR,X.,=>PC..
31:          ISZ(5):]P(1)]MAR<=ADR.,
32:              ]P(2)]MBUS=M(MAR),MBR<=MBUS.,
33:              ]P(3)]MBR<=ADD*MBR,1,12*.,
34:              ]P(4)]MBUS=MBR,M(MAR)<=MBUS,]T(+/MBR)]
35:                   PC<=CNTUP*PC*.,=>PC..
36:          DCA(6):]P(1)]MBR<=ACC.,]P(2)]MAR<=ADR.,
37:              ]P(3)]ACC<=0,MBUS=MBR,M(MAR)<=MBUS.,]P(4)]=>PC..
38:          JSR(7):]P(1)]MBR<=CDA*PC.,]P(2)]MAR<=0.,
39:              ]P(3)]MBUS=MBR,M(MAR)<=MBUS.,
40:              ]P(4)]PC<=ADR.,=>PC.
41:           RET(8):]P(1)]MAR<=0.,]P(2)]MBUS=M(MAR),MBR<=MBUS.,
42:               ]P(4)]PC<=MBR(4:11),=>PC..
43:            JMP(9):]P(1)]PC<=ADR.,]P(4)]=>PC.....
44:   <FL>3,4,5,6,8.
```

Figure 5.5(a) : (Continued)

ORIGINAL PAGE IS
OF POOR QUALITY

6. CONCLUSIONS

DDLTRN and DDLSIM programs are currently being tested on SEL-32 Computer System. The output of the DDLTRN is suitable for logic generation. The output at PASS 6 and the Facility table are now being analyzed to derive the algorithms for logic synthesis. With the logic synthesis programs complete, CADAT will be a true automatic design system.

APPENDIX

This is a preprint of the article to be published in the December 1979 issue of the "Proceedings of IEEE."

Computer Hardware Description Languages—A Tutorial

SAJJAN G. SHIVA, MEMBER, IEEE

Abstract—Just as software designers use high level languages (HLL) to express the algorithms in terms of language statements, digital hardware designers use hardware description languages (HDL) to describe the system they are designing. Although HDL's were originated as a medium of precise yet concise description of digital hardware, they have found a variety of applications such as generating user manuals, teaching logic design, acting as an input medium for an automatic design system, etc. This tutorial paper introduces HDL's as useful tools for hardware design and documentation. The capabilities and limitations of HDL's are discussed, along with the guidelines to select an HDL. The directions for future work and an extensive bibliography are provided.

I. Introduction

ANY digital system can be described in the following six levels of complexity [1]-[4]:

1) algorithmic level which specifies only the algorithm used by the hardware for the problem solution;
2) Processor memory switch (PMS) level which describes the system in terms of processing units, memory components, peripherals, and switching networks;
3) instructional level (programming level) where the instructions and their interpretation rules are specified;
4) register transfer level where the registers are system elements and the data transfer between these registers are specified according to some rule;
5) switching circuit level where the system structure consists of an interconnection of gates and flip-flops and the behavior is given by a set of Boolean equations;
6) circuit level where the gates and flip-flops are replaced by the circuit elements such as transistors, diodes, resistors, etc.

Logic diagrams and Boolean equations have been used as media of hardware description. The complexity of these media increases rapidly as the system complexity increases and they are not convenient to suppress the details and still provide accurate descriptions as we move into the higher levels from the switching circuit level. Hardware description languages (HDL's) evolved as a solution. Although the use of computer oriented languages to describe digital system design can be traced back to Shannon's work on switching circuits in 1939, Aiken's work on switching theory in the 1940's, the logic diagrams at M.I.T. and NBS in the late 1940's and the flip-flop equations in the 1950's [5]. Iverson's work [6] on a formal HDL probably initiated the contemporary interest in this area of research. An HDL is similar to any other high-level programming language (HLL) and provides a means of

Manuscript received May 23, 1979; revised August 29, 1979. This work was supported by the National Aeronautics and Space Administration under Grants NSG-8057 and NAS8-33096. The submission of this paper was encouraged after review of an advance proposal.

The author is with the Department of Computer Science, University of Alabama in Huntsville, Huntsville, AL 35807.

ORIGINAL PAGE IS OF POOR QUALITY

1) precise yet concise description of the system;
2) convenient documentation to generate users manuals, service manuals, etc.;
3) input of the system description into a computer for simulation and design verification at various levels of detail;
4) software generation at the preprototype level, thus bridging the hardware/software development time gap;
5) incorporation of design changes and corresponding changes in documentation, efficiently;
6) designer/user (teacher/student) communication interface at the desired level of complexity.

HDL's are capable of describing the parallelism, nonrecursive nature, and timing issues in the hardware more naturally, and thus differ from the pure sequential nature of a general HLL. (Some existing HLL's provide concurrency or simulated concurrency constructs in their language elements, for example, PFOR on PEPE [7].) An HDL can be classified as a *procedural* or a *nonprocedural* language [4]. Each statement in a nonprocedural HDL description would contain a *label* which describes the condition under which the activities described by the statement are to be performed. Thus the sequential ordering of the statements does not impose the ordering of the activities. In a procedural HDL description, the activities are performed following the sequential ordering of the statements.

HDL's are designed to describe both the structural and behavioral characteristics of a digital system. The fundamental properties of hardware systems and the art of hardware design process dictate the essential features of an HDL. For an HDL to be a useful tool, it has to possess the following properties:

1) It has to have a natural way of describing the parallelism, nonrecursive nature, and timing issues in digital hardware.

2) The structure and control parts of the hardware should be easily described and preferably the description of the two parts be separated (if such a division enhances the description) so that a user interested in the behavior of the system need not concern himself with the structure of the system. This division provides the flexibility to use hardware, software, or firmware for the control part, whichever is economical.

3) The language should serve as a medium at all levels of system description.

4) The design changes should easily be incorporated into the description and corresponding translation should be done preferably without a complete retranslation. This feature will be useful for the interactive environment. (A translator translates the HDL description into an intermediate code from which the simulator and other programs can be driven (see Fig. 1). The intermediate code could be a set of Boolean and register transfer equations [31] or a computer executable code like polish strings [23].)

5) The language should be easy to learn and remember, to accommodate the software-shy hardware designer, although the hardware engineer cannot neglect the software aspects anymore, due to the impact of microprocessors. The design system should be portable, thus necessitating the translators and simulators of HDL be written in higher level languages.

6) Two approaches to system design are often proposed: the bottom-up approach where the elementary components are combined to form more complex ones and the top-down approach, where the system is decomposed into a collection of subsystems until the elementary components are reached. In practice, the designer may choose a combination of the two approaches. The structural detail at any design level varies from designer to designer. The HDL should allow the designer to control the amount of detail during each design phase.

7) The description of the large and medium scale integrated circuit (LSI and MSI) modules as system components should be straightforward, so should be the inclusion of newer modules. If the system is partitioned by the designer to accommodate standard modules, this partitioning should be retained by the HDL translators and simulators.

Several HDL's have been reported [9]–[76] since Iverson's proposal of an HDL. Translators to convert the description into an intermediate executable code and simulators to execute this code have been written for some of these languages. No single HDL has met all the above characteristics. The tendency has been to invent a new HDL to suit a particular design environment, basically due to the difficulty in transporting the translators and simulators on to the new computing systems and extending them to accommodate the requirements of the new design environment. Table I [8] lists the implementation details of several HDL's reported. This list is by no means exhaustive.

Section II discusses the utility of HDL's in system design. A brief discussion of one popular HDL, the computer design language, is given in Section III along with two example descriptions. Two case studies are presented: one to select an HDL for an integrated circuit design environment (Section IV) and the other to show the utility of HDL's in concurrent hardware–software development (Section V). Future work required and current research topics are discussed in Section VI.

II. HDL's in System Design

Fig. 1 shows the utility of an HDL in a digital system design environment. The designer uses the HDL to describe his design. This description is translated into a computer executable data base, which serves as the source for various other operations. The design can be refined by simulating at the description level (Loop 1), before proceeding to a more detailed simulation (Loop 2) at the logic level. The data-base also serves as a source for logic diagram generation, microcode and test set generation. The physical construction of the system follows the simulation and refinement at the logic level.

Translation and simulation of HDL's have been well defined [9]–[76]. Physical construction aspects have also been automated and are widely used in industry [77]. Test generation [78] and hardware compilation [12], [39] need further investigation. The variety of design methodologies, the artistic nature of the design process, and the ambiguity posed by the variety of components available make the hardware compilation a tedious task.

III. Computer Design Language

A hardware programming language (AHPL), computer design language (CDL), digital systems design language (DDL) and the instruction set processor (ISP) have been the most popular languages, partly due to their early introduction as general purpose HDL's. These languages were developed in university environments and are used in teaching digital logic design. New features are being added to these languages to make them more versatile. Well-tested translators and simulators are available for these languages (see Table I for references). Although several HDL's have been designed for an industrial use [59], [64], the design process being proprietary in nature, the use of HDL's is not widely reported [79].

This section provides a brief introduction to CDL. Example descriptions in CDL are provided. CDL was chosen over the others due to its simple structure and the author's familiarity with the language.

CDL was proposed originally by Chu [20]–[22]. A translator and simulator were written for a subset of this language [23]. Several modifications were made to this translator and simulator [26]–[29].

CDL describes the structural and functional parts of a digital system. The structural components like memory, registers, clocks, switches, etc. are declared explicitly at the beginning of the description. The functional behavior of the element is described by the commonly used operators and user defined operators. Valid data paths are declared implicitly whenever there is a data transfer. Both parallel and sequential operations are allowed. Synchronous operations require a conditional test for an appropriate signal. The language is easy to understand and is highly readable.

All the variables in a CDL description are global. The system description can be only at one level, and there is no subroutine facility in CDL, thus making it unsuitable for describing hardware in a modular fashion. It is not possible to include special hardware components like integrated circuits (IC's) in a description. However, its simplicity of structure and its portability resulting from the FORTRAN implementation, have made CDL a popular language. The description of CDL syntax and semantics as accepted by the present version of translator and simulator [29] is given below. Table II shows the standard operators in CDL. Facilities are declared at the beginning of the system description with *declaration statements* of the format

DEVICE, list

where DEVICE can be a REGISTER, SUBREGISTER, MEMORY, DECODER, SWITCH, TERMINAL, BUS, BLOCK, and CLOCK. Some example declarations are shown below.

REGISTER, *A*(0-2), *R*, *F*(6-1)

SUBREGISTER, *F*(OP)=*F*(3-1), *F*(OR)=*F*(6-4)

MEMORY, *M*(*R*)=*M*(0-77,0-10) — Memory with 78, 11 bit words. Address register *R*.

DECODER, *L*(0-15)=*G*(2-5) — 4 bits of *G* are decoded into $L_0, \ldots, L_{15}$.

CLOCK, *P*(2) — A clock with 3 phases *P*(0), *P*(1), *P*(2) •

SWITCH, STRT (OFF, ON) — A switch with 2 positions. A maximum of 10 positions allowed.

TERMINAL, *B*=*A*′, *C*=*A*+*B*1, *D*1=*A*•*B*

BUS, *Z*(0-7) — A 8 line BUS *Z*

BLOCK, SERCOM (*A*=*A*(1)′-*A*(5-2)) — SERCOM is an alternate name for the operations within the parentheses.

A DO/SERCOM statement is used to invoke the set of statements declared by BLOCK, SERCOM.

An *unconditional microstatement* has the form

Variable = Expression

Example: *A*=1, *B*(1,3-5)=*C*•*D* + *E*(2,0-2)

A conditional microstatement has the forms

IF (expression) THEN (microstatements)

IF (expression) THEN (microstatements)
ELSE (microstatements)

Examples: IF (*A*·*EQ*·*B*) THEN (*R*=0)
IF (*C*·*NE*·*D*) THEN (*R*=0, *Z*=1) ELSE (*R*=1)

Conditional statements may be nested to any number.

A *labeled statement* has the format

/label/microstatements

where

label = expression•clock

Example: /*K*(0)•*P*/*A*=*B*,*B*=*A*

Special operators can be established by the user through a separate subprogram. The format is

*OPERATOR: Parameters/Name/
/microstatements, RETURN
END

ORIGINAL PAGE IS
OF POOR QUALITY

A *count operator* is defined below

```
*OPERATOR, X(1-4)-COUNT
//IF (X(4)-EQ-0THEN(X(1-3)-1)
    ELSE (IF (X(3)-EQ-0 THEN (X(1-2)-1-0)
        ELSE (IF(X(2)-EQ-0)THEN (X(1)-1-0-0)
            ELSE (X(1)'-0-0-0))).RETURN
END
```

Several commonly used operations (Table II) are included in the current CDL software:

Examples: *A*=*A*·CNTUP·, *C*=*A*·ADD·*B*

The *CDL TRANSLATOR* performs a syntax check of the description and translates it into a set of tables and a polish string program.

The *CDL SIMULATOR* executes the output of the translator and can accept simulation parameters through the following command set:

LOAD	Used to initialize registers and memory.
OUTPUT	Provides a hexadecimal printout of the specified register and memory contents and switch positions at the desired clock or label.
SWITCH	Enables setting switch positions.
RESET	Resets the earlier settings of the simulation parameters.
SIMULATE	Provides the start and stop conditions for simulation.

CDL can be used to describe simple to very complex digital systems. Two example descriptions are provided below to illustrate this feature.

Example 1: A Serial Two's Complementer

A circuit to replace the contents of a 6-bit register *R* by its two's complement will be described. The complementation is done by the well-known copy/complement algorithm (starting from the least significant bit of *R*, copy the bits as they are till the first nonzero bit; complement the bits after the first nonzero bit, till the most significant end of the register). Fig. 2 shows the circuit and its CDL description. A 3-bit register *C* is used to count the number of shifts. Flip-flop *S* indicates the COPY (S=0) and COMPLEMENT (S=1) states. A switch SW is used to start the complementation process. Statements 2,3, and 5 describe these facilities. The control circuitry includes a single phase clock *P* and a 1-bit state register *T* (Statements 6 and 5). Fig. 3 shows the state diagram for the control circuitry. The controller waits in T=0 state as long as the SW is off. When SW is on, the *C* and *S* are cleared, and a state change occurs (Statement 8). As long as $C < 5$, the shift signal is on. Statement 9 describes the process of copying or complementing according to S=0 or 1. Note that the circulation of the register *R* is described using the concatenation operator. When the count reaches 5 the controller goes to T=0 state, thus completing the complementation.

CDL, being a nonprocedural language, evaluates labels and performs the activities corresponding to the active label. Each such evaluation is a label cycle. During simulation, the values of *R*, *C*, *S*, and *T* are requested to be OUTPUT at each label cycle (Statement 11). The switch is turned on in cycle 1 (Statement 12). *R* is loaded with $(5)_8$ (subscripts indicate the base of the number; the number is decimal if not subscripted) initially (Statements 13,14) and simulation is requested for 20 label cycles with 6 label cycle evaluation repetitions to seek an active label before terminating. Fig. 4 shows the simulation results. The contents of *R* $(73)_8$ at the end of the label cycle 6 are the two's complement of the original contents $(05)_8$, thus indicating the validity of the design.

The clock and label cycles are RESET and *R* was loaded with $(21)_8$. Fig. 4(b) shows the corresponding simulation results.

The CDL description in Fig. 2 serves as a compact and precise description of the structure and behavior of the hardware.

Example 2: A Minicomputer

Fig. 5 shows the structural details, instruction set, and the CDL description of a minicomputer [52]. The minicomputer has a 256 word 12-bit memory, with an 8-bit memory address register (MAR) and a 12-bit memory buffer register (MBR). There is an 8-bit program counter (PC) and an accumulator (ACC) of 12 bits. The arithmetic/logic unit (ALU) receives the operands from MBR and a 12-bit *X* register, and puts the results on to the 12-bit BUS. The instructions consist of a 3-bit operation code, an indirect address flag bit, and 8 address bits. The register-set description is provided by the Statements 1-3 of Fig. 5(b). The BUS is not explicitly described to retain the high level description nature. Fig. 5(c) shows the details of the instruction set. Statement 4 in Fig. 5(b) describes a START switch, a RUN switch to indicate the RUN/STOP state, and a three state switch for indicating instruction fetch (*F*), indirect address computation (Defer, *D*) and Execution (*E*) phases. Statements 5 and 6 provide the instruction decoding details. There is a 4-phase clock *P* (Statement 7) which activates the synchronous control unit. Each major cycle consists of 4 minor cycles. The comments in the CDL description identify the Fetch cycle, Defer cycle, and the Execution cycle for each instruction. Fig. 5(d) shows a program to add the four numbers in memory locations 0-3 and place the sum in location 7. The program will be located in memory locations 10-16. Location 4 is initialized to -3 and incremented by 1 each time through the loop, and tested for zero to terminate the summing operation. The data values are accessed by an indirect reference (TAD* 6) to location 6 which is incremented from 0 by 1 each time through the loop. Fig. 5(d) shows the program in assembly, binary, and decimal forms. Fig. 5(c) shows the memory map just before the execution of the program. This memory map is simulated by the LOAD command for the CDL simulator (Statements 43-45) in Fig. 5(b). The program counter is set to 10 (Statement 46), the switch is turned ON (Statement 42) and the simulator is requested for 200 label cycles (Statement 47), outputting several register contents (Statement 41) at each label cycle. The simulator results are similar to the two's complementer example and are not shown for the sake of brevity. It is evident that the CDL description of the minicomputer is concise and more precise than any natural language description could be.

IV. Selection of HDL

Due to the large number of HDL's proposed, the selection of an HDL for a particular design environment becomes a nontrivial task. Although the structure of the language, the operators available, the capabilities of the language to describe the design in a logical manner are important considerations, the implementation issues seem to override them. One such selection process is described here along with the system description.

Fig. 6 shows the details of the computer aided design and test (CADAT) system of the NASA Marshall Space Flight Center [80]. The designer inputs the details of the IC to CADAT as a set of standard cells and their interconnections. The standard cell selection is done manually from a standard cell library. This description is at the logic diagram level. Detailed logic simulation and refinements are carried out on the design. The final design is input to the automatic test-vector generation and placement and routing programs. The IC mask pattern generation is done interactively and a mask analysis and performance simulation are done before fabricating the mask. The last two steps in the IC fabrication are the wafer processing and the final testing.

At present, the generation of logic diagrams and choosing the standard cells from the cell library for the design are done manually. Integration of a high-level design language would help the designer to simulate his design and refine it at a high level before entering his design into the current system. This requires an HDL with a simulator and logic synthesizer (hardware compiler) that generates the logic net input required by

ORIGINAL PAGE IS
OF POOR QUALITY

the CADAT System. The breadboard implementation and testing of a complex large scale integrated circuit (LSIC) design is not feasible since it cannot be properly breadboarded with anything but the LSIC itself. With an HDL, this breadboarding process can be substituted with a computer simulation of the LSIC design, thus minimizing the design changes and, hence, the cost of mask fabrication and wafer processing.

The following five criteria were used in selecting a suitable language [81]:

1) activity
2) level of description
3) software availability and portability
4) ease of logic generation
5) modularity.

1) Activity: It is essential to choose a language which is being used elsewhere to receive the benefits of the extensions to the language. Most of the HDL's proposed do not have a translator and a simulator that is up-to-date and fairly versatile, though the language itself is versatile. The process of improving the HDL software and capabilities would be aided by the active interest of the other groups in the language.

2) Level of Description: The selected HDL should accommodate a description at the register transfer level and/or below to facilitate the logic generation. A higher than register transfer level description may not be needed for the IC design environment of CADAT.

3) Software Availability and Portability: The development of a HDL is incomplete without a simulator and a translator written for it, since this software development process refines the language structure. The software should be portable to accommodate the general portability of the CADAT software.

4) Ease of Logic Generation: Any HDL translator oriented towards providing information for a simulator collects and rearranges the combinational logic and register transfers. This intermediate translated output should be amenable to logic generation.

Modularity: The HDL description should be modular enough to reflect the modularity of the hardware, to enable easier understanding and modular design verification.

A comparison of the four prominent HDL's with respect to the above criteria is shown in Table III. ISP, although versatile, does not lend itself to the logic generation level very well. CDL is suitable for microprogram generation. The nonmodular description feature of CDL and the difficulty in using the polish string output of the translator to generate logic diagram level description make it unsuitable for the CADAT system environment. AHPL and DDL were the strong contenders. Both have a fairly portable software package and are suitable for the level of description needed for CADAT. The modularity is brought about by the subroutining feature in AHPL, whereas the block structure of DDL is closer to the hardware modularity. From a traditional hardware designer's point of view, programming in either language is equally difficult. Although a hardware compiler is available for AHPL [12], its SNOBOL implementation raises newer implementation issues for CADAT, which is predominantly in FORTRAN. The DDL translator provides a set of Boolean equations and register transfer expressions which can be used for hardware compilation [39], [79] though not very easily. The block structure and the software of DDL made it a better choice over AHPL for the CADAT system.

Note that the selection of the HDL is oriented more towards the implementation issues, rather than a rigorous analysis of the capabilities and the characteristics of the HDL, such as the structure of the language, operators available, ease of understanding, etc. Such a rigorous analysis, although valuable, will not aid in the selection of the language since the implementation issues override the other characteristics. Also, the selection creteria ignored the possibility of developing a new language to exactly fit the CADAT environment. The *activity* criteria also eliminated several other HDL's like LCD [59] and SDL [72] from consideration.

V. Concurrent Hardware and Software Development

The use of HDL's in hardware development is obvious. The recent advances in IC technology have tremendously increased the speed of new system announcements. But the software development for the new system has not caught this pace. With the ability of the HDL to describe and simulate the hardware accurately, it is possible to develop the software for the digital system concurrently to bridge the software-hardware development gap. This section describes an experiment to measure the performance of CDL in software development [26].

A multiprocessing system, consisting of a Digital Equipment Corporation PDP-8 Minicomputer and an INTEL 8080 Microprocessor was used. The two processors were simulated individually, followed by the simulation of the shared memory and the input device for the system. The input device is an on-line inspection station which interrupts the 8080 after each part is examined to enter the measurements of the part into a 64-word 8-bit memory. Intel 8080 handles the bookkeeping of these measurements for use by PDP-8. Several programs were written both for 8080 and PDP-8. The programs on PDP-8 accept the measurement from 8080, determine if they are within specifications, and transmit the condition of the part to 8080. The 8080 programs handle this interrupt and keep a record of the number of parts inspected and their condition. The programs were written in assembly language of the particular processor and were stored in the shared memory in the machine language form. The details of the simulations can be found in [27].

An important consideration in developing programs is the assembly time required by the host processor running the CDL simulation of PDP-8 and Intel-8080. Table IV shows the CPU times required for typical programs on an IBM 370/155. Clearly, the cost of such simulations is prohibitive. However, assuming that the cross assemblers are available on the host machine, developing an application program using CDL simulation would not be very expensive, since these programs will usually be shorter than an assembler or a compiler. A related issue would be the performance comparison of such simulations using high level languages for the description of the hardware, rather than an HDL. Much of the overhead of the HDL translator/simulator software could be reduced by using an HLL for describing and simulating the *particular* hardware. A comparison of such HLL versus HDL descriptions and their run times is needed.

VI. Future Work

Although the suitability of an HDL for hardware system description is well recognized, the HDL's are not used extensively, partly due to the variety of structures and notations used in these HDL's, making them harder to understand. Many structures found in HDL's are simple for a software professional to understand and use. But a hardware designer not familiar with programming finds them hard to use. This problem will be partially solved by the popularity of the microprocessors as design elements, requiring the hardware designer to understand software.

The differences in notations and structures used by HDL's make it difficult to borrow a language developed elsewhere. This difficulty is augmented by the nonstandard design meth-

ORIGINAL PAGE IS
OF POOR QUALITY

odologies and nonportable HDL software. The problem of nonuniform notations and structures will be reduced by the introduction of a consensus language (CONLAN) [3], [82]. The following guidelines are used in arriving at CONLAN.

1) CONLAN must support design, description and simulation of at least the following classes of systems: gate network, register networks, processors, memories, processor systems.

2) Any system may be displayed via either
   a) a network structure description or
   b) a behavior description.

3) CONLAN is to service
   a) computer architects and logic designers for purpose of trade-off exploration and optimization, design verification, and design documentation
   b) systems, micro-, and application programmers
   c) electronics production engineers
   d) maintenance engineers.

4) CONLAN syntax and semantics must support
   a) well-defined descriptions
   b) machine parsing, interpretation, and simulation with error detection
   c) comprehension of complex system structure and function
   d) division of design efforts
   e) control over the level of abstraction at which subsystems are described
   f) simulation control.

5) CONLAN is to be evaluated in terms of benchmarks such as standard function declarations, time operator declarations, IC descriptions (including microprocessors), and design descriptions (including a multiprocessor system).

The basic aim of CONLAN is to provide a versatile uniform base language with the capabilities of augmenting the basic syntax with the specific constructs with their own semantic interpretation, as required by the environment.

The efficiency of the HDL software depends on its efficient use of the host computer on which it was developed. Hence, the software tends to be machine dependent, making it fairly nonportable. Although the efficiency suffers [28] if the software is made portable, a well-documented software package (along with a good discussion of the algorithms used) is a necessity. Several other areas of investigations could be identified.

Procedures to analyze the HDL descriptions of digital systems [4] are to be developed in order to avoid simulation or at least minimize simulation costs. HDL's are to be designed improved to accommodate easier description of LSI circuits and microprocessors [83], [84]. Suitability of HDL's as languages for microprocessor software development [85] and architecture comparison needs investigation.

Comparison studies of HLL and HDL with respect to ease of programming, ease of understanding, description length, simulation cost and efficiency are required.

Logic synthesis from the HDL description is not well developed [12], [39], [79], [86]. Decomposition of the digital system to accommodate the LSI and MSI components and retaining this decomposition till the final stage in the design are of paramount importance. The ability of the procedures to search through a library of available IC's and the capabilities to accommodate newer modules is necessary.

The availability of inexpensive processors has increased the popularity of distributed processing systems. The HDL's have traditionally been designed for a single processor environment and lack the facilities to describe the interprocessor communication. Such ability will make the HDL more attractive [87].

The acceptability of an HDL for a particular environment depends on its capabilities to accommodate the operations in the environment. Since the majority of HDL's are designed for a particular environment, they tend to be less suitable for other environments. For example, a HDL developed with a goal of efficient high-level description and simulation would hardly suit a logic synthesis environment. A classification of available HDL's according to their underlying models for behavior is needed.

VII. Conclusions

The capabilities of HDL's were discussed. A brief introduction to one such language (CDL) along with example descriptions were given. Case studies for selection of an HDL and the use of HDL in hardware/software development were given. The areas for further investigations were identified.

Acknowledgment

The author wishes to thank J. M. Gould and P. Hsia for helpful discussions during the preparation of this paper: Michigan Technological University Computer Center for providing the CDL Software; and C. Chandler for the assistance in preparation of the manuscript.

References

References [1]-[3] are collections of papers. References [31], [77], and [88] also provide extensive bibliographies.

[1] *Proc. Int. Symp. CHDL's Applications*, 1975.
[2] *Computer*, vol. 7, no. 12 (Special Issue on CHDL's), Dec. 1974.
[3] *Computer*, vol. 10, no. 6 (HDL Applications), June 1977.
[4] M. R. Barbacci, "A comparison of register transfer languages for describing computers and digital systems," *IEEE Trans. Comput.*, vol. C-24, pp. 137-150, Feb. 1975.
[5] I. S. Reed, "Symbolic syntheses of digital computers," in *Proc. Ass. Comput. Mach.*, pp. 90-94, 1952.
[6] K. E. Iverson, "A common language for hardware, software, and applications," in *Proc. Fall Joint Comput. Conf.*, pp. 121-129, 1962.
[7] K. J. Thurber, *Large Scale Computer Architecture*. Rochelle Park, NJ: Hayden, 1976, ch. 5, pp. 231-251.
[8] S. G. Shiva, "Hardware design languages—a bibliography," Alabama A & M Univ., Status Rep. NSG-8057, Normal, AL, Mar. 1978.
[9] G. M. Bodner and J. H. Tracey, "An asynchronous circuit design language (ACDL)," *IEEE Trans. Comput.*, vol. C-23, pp. 971-976, Sept. 1974.
[10] F. J. Hill and G. R. Peterson, *Digital Systems: Hardware Organization and Design*. New York: Wiley, 1978.
[11] F. J. Hill, "Updating AHPL," in *Proc. Int. Symp. CHDL's Applications*, pp. 22-29, 1975.
[12] R. E. Swanson, Z. Navabi, and F. J. Hill, "An AHPL compiler/simulator system," in *Proc. Sixth Texas Conf. Comput. Syst.*, pp. 1-11, 1977.
[13] T. D. Friedman, "ALERT: A program to compile designs from new computers," in *Dig. First Annu. IEEE Comput. Conf.*, pp. 128-139, 19[illegible].
[14] T. D. Friedman and S. C. Yang, "Methods used in an automatic logic design generator (ALERT)," *IEEE Trans. Comput.*, vol. C-18, pp. 595-613, July 1968.
[15] J. A. Darringer, "The description, simulation, and automatic implementation of digital computer processor," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, AD700 144, May 1969.
[16] —, "A language for the description of digital computer processors," in *Proc. Design Auto. Workshop*, pp. 15-1-15-8, 1968.
[17] W. R. Franta and W. K. Giloi, "APL*DS: A hardware description language for design and simulation," in *Proc. Int. Conf. CHDL's Applications*, pp. 45-52, 1975.
[18] E. D. Crockett *et al.*, "Computer aided systems design," in *Proc. Fall Joint Comput. Conf.*, pp. 287-296, 1970.
[19] Bogo, Guyot, Lux, Mermet, and Payan, "CASSANDRE and the computer aided logical systems design," TA 6, 26, *Proc. IFIP Congr.*, 1971.
[20] Y. Chu, "An ALGOL-like computer design language," in *Commun. Ass. Comput. Mach.*, vol. 8, pp. 607-615, Oct. 1965.
[21] —, *Computer Organization and Microprogramming*. Englewood Cliffs, NJ: Prentice-Hall, 1972.
[22] —, "Design automation by the computer design language," Maryland Computer Science Center, Tech. Rep. 69-86, Mar. 1968.
[23] C. K. Mesztenyi, "Translator and simulator for the computer design and simulation program, version 1," Computer Science Center, Univ. Maryland, Tech. Rep. 67-48, June 1968.
[24] Y. Chu, "A higher-order language for describing microprogrammed computers," Computer Science Center, Univ. Maryland, Tech. Rep. 68-78, Sept. 1968.
[25] —, "Structure of CDL programs," Dep. Computer Science, Univ. Maryland, Tech. Note 74-58, May 1974.
[26] J. R. Heath, B. D. Carroll, and T. C. Cwik, "CDL-A tool for hardware and software development," in *Proc. Design Auto. Conf.*, pp. 445-449, June 1977.
[27] T. T. Cwik, "Multiprocessing simulation of the intel 8080 and the PDP-8 using CDL," Master's thesis, Auburn Univ., Auburn, AL, Mar. 1976.
[28] L. R. Stine and F. J. Mowle, "A position paper on extensions to the CDL," in *Proc. Int. Conf. CHDL's Applications*, pp. 103-114, 1975.

ORIGINAL PAGE IS
OF POOR QUALITY

[29] J. Bara and R. Born, "A CDL compiler for designing and simulating digital systems at the register transfer level," in *Proc. Int. Conf. CHD's Applications*, pp. 96–102, 1975.

[30] D. R. Smith, "Computer structure language," in *Proc. Int. Conf. CHDL's Applications*, pp. 153–160, 1975.

[31] D. L. Dietmeyer and J. R. Duley, "Register transfer languages and their translation," in *Digital System Design Automation: Languages Simulation and Data Base*, M. S. Breuer, Ed. Woodland Hills, CA: Computer Science Press, 1975, ch. 2, pp. 117–218.

[32] J. R. Duley, "DDL-A digital system design language," Ph D. dissertation, Univ. of Wisconsin, Madison, 1967.

[33] J. R. Duley and D. L. Dietmeyer, "A digital system design language (DDL)," *IEEE Trans. Comput.*, vol. C-17, pp. 850–861, Sept. 1968.

[34] —, "Translation of a DDL digital system specification to Boolean equations," *IEEE Trans. Comput.*, vol. C-18, pp. 305–313.

[35] R. L. Arndt and D. L. Dietmeyer, "DDLSIM—A digital design language simulator," *Proc. Nat. Electronics Conf.*, vol. 26, pp. 116–118, 1970.

[36] D. L. Dietmeyer, "DDLTRN-users manual," Dep. Elec. Comput. Eng., Univ. Wisconsin-Madison.

[37] —, "Translation of DDL descriptions of digital systems," Univ. of Wisconsin-Madison, Rep No. ECE-77-13, Sept. 1977.

[38] — "DDLSIM-users manual," Dep. Elec. Comput. Eng., Univ. of Wisconsin-Madison.

[39] M. S. Dushi and D. L. Dietmeyer, "Automated PLA Synthesis of the combinational logic of DDL descriptions," Univ. Wisconsin-Madison, Rep. ECE 78-17, Nov. 1978.

[40] F. J. Rammig, "Digitest II: An integrated structural and behavioral language," in *Proc. Int. Symp. CHDL's Applications*, pp. 36–44, 1975.

[41] A. M. Despain, "The use of two CHDL's, PMS and DIDL in the design of a Fourier transform processor," in *Proc. Int. Symp. CHDL's Applications*, pp. 76–84, 1975.

[42] J. L. Houle, "A formal language for the description modeling and realization of digital systems," Ph.D. dissertation, Univ. Waterloo, Waterloo, Ont., Canada, 1974.

[43] S. S. Chang and J. H. Tracey, "An interactive computer graphics language for the design and simulation of digital systems," *Computer*, vol. 10, pp. 35–41, June 1977.

[44] E. A. Franke, "Automated functional design of digital systems," Ph.D dissertation, Case Western Reserve Univ., Nov. 1967.

[45] A. C. Parker and J. W. Gault, "A language for the specification of digital interfacing problems," in *Proc. Int. Symp. CHDL's Applications*, pp. 85–90, 1975.

[46] A. Giese, "HARGOL—A hardware oriented algol language," A/S Regnecentralen, Internal Rep. VAS, Copenhagen, Denmark, Aug 1966.

[47] P. L. Flake, G. Musgrave, and M. Shorland, "The HILO logic simulation package," in *Proc. Int. Symp. CHDL's Applications* pp. 161–171, 1975.

[48] C. G. Bell, J. Grason, and A. Newell, *Designing Computer and Digital Systems*. Maynard, MA: Digital Press, 1973.

[49] G. Bell and A. Newell, *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1971.

[50] —, "The PMS and ISP description systems for computer structures," in *Proc. Spring Joint Computer Conf.*, pp. 351–374, 1970.

[51] M. R. Barbacci and D. P. Siewiorek, "Applications of an ISP compiler in a design automation laboratory," in *Proc. Int. Symp. CHDL's Applications*, pp. 69–75, 1975.

[52] M. R. Barbacci *et al.*, "The ISPS computer description language," Dep. of CS and EE Rep., Carnegie-Mellon University, Aug. 1977.

[53] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek, "The ISPS computer description language," Dep. of CS and EE Rep., Carnegie-Mellon University, Aug. 1977.

[54] M. R. Barbacci and A. W. Nagel, "An ISPS simulator," Dep. of CS and EE Rep., Carnegie-Mellon University, Nov. 1977.

[55] M. B. Baray and S. Y. H. Su, "A digital system modeling and design language," in *Proc. Design Auto. Workshop*, pp. 1–22, 1971.

[56] S. Y. H. Su, "A language for automated logic and system design," *presented at the Workshop on Computer Descriptive Languages*, Rutgers University, New Brunswick, NJ (Sept. 6–7, 1973).

[57] S. Y. H. Su, M. B. Baray, and R. L. Carberry, "A system modeling language translator," in *Proc. Design Auto. Workshop*, pp. 35–49, 1971.

[58] D. Borrione, "LASCAR: A language for simulation of computer architecture, in *Proc. Int. Symp. CHDL's Applications*, pp. 143–152, 1975.

[58] C. J. Evangelisti, G. Goertzel, and H. Ofek, "Designing with LCD: Language for computer design," in *Proc. Design Auto. Conf.*, pp. 369–376, 1977.

[60] D. F. Gorman and J. P. Anderson, "A logic design translator," in *Proc. Fall Joint Comput. Conf.*, pp. 251–261, 1962.

[61] J. Lund, "LOGAL-logic algorithmic language," Univac Tech. Memo A00317, Roseville, MN, Mar. 5, 1973.

[62] H. P. Schlaeppi, "A formal language for describing machine logic, timing and sequencing (LOTIS), *IEEE Trans. Comput.*, vol. EC-13, Aug. 1964.

[63] G. J. Lipovski, "Naming convention for modular design languages," *presented at the First Workshop on Computer Hardware Description Languages*, Rutgers Univ., New Brunswick, NJ (Sept. 6–7, 1973).

[64] I. Lewin and A. M. Peskin, "The MODEL/LINDA design automation system, in *Proc. Int. Symp. CHDL's Applications*, pp. 53–61, 1975.

[65] R. W. Marczynski, W. T. Pulczyn, and J. M. Suchacki, "OSM-microprogrammed hardware structure description language," in *Proc. Int. Symp. CHDL's Applications*, pp. 161–171, 1975.

[66] R. Piloty, "RTSI (Registertransfersprache)," 3. Aufl., Institut fur Nachrichtenverarbeitung, TH Darmstadt, 1969.

[67] U. Blumenschein and W. Langner, "Entwicklung and definition der registertransfersprache RTS II " NVSt 83, Institut fur Nachrichtenverarbeitung, TH Darmstadt, 1970.

[68] R. Piloty, "Rechnerorganisation I und. II (lecture notes)," Institut für Nachrichtenverarbeitung, TH Darmstadt, 1974.

[69] R. Piloty, "Segmentation constructs for RTL III," in *Proc. Int. Symp. CHDL's Applications*, pp. 115–124, 1975.

[70] E. P. Stabler, "System description languages," *IEEE Trans. Comput.*, vol. C-19, pp. 1160–1173, Dec. 1970.

[71] D. R. Gorman, "A system descriptive language and its uses," Ph.D. dissertation, Univ. of Pennsylvania, 1968.

[72] W. M. VanCleemput, "An hierarchical language for the structural description of digital systems," in *Proc. Design Auto. Conf.*, pp. 377–385, 1977.

[73] SMITE Training Manual, TRW Inc., Aug. 1977.

[74] W. Goerke and H. J. Hoffman, "Simulation of switching circuits by SBM—a new hardware simulation language," in *Proc. Int. Symp. CHDL's Applications*, pp. 125–133, 1975.

[75] J. A. N. Lee, "VDL—a definitional system for all levels," in *Proc. First Annu. Symp. Comput. Arch.*, pp. 41–48, Dec. 1973.

[76] D. L. Parnas, "A language for describing the functions of synchronous systems," *Commun. Ass. Comput. Mach.*, vol. 9, no. 2, Feb. 1966.

[77] W. M. VanCleemput, *Computer Aided Design of Digital Systems. A Bibliography*. Woodland Hills, CA, Computer Science Press, 1976.

[78] B. M. Huey and F. J. Hill, "Fault test generation using a design language," in *Proc. Int. Symp. CHDL's Applications*, pp. 91–95, 1975.

[79] N. Kawato, T. Saito, F. Maruyama, and T. Uehara, "Design and verification of large scale computers by using DDL," in *Proc. Design Auto. Conf.*, pp. 360–366, 1979.

[80] J. M. Gould, "The large scale microelectronics computer-aided design and test system," NASA TM-78202, Marshall Space Flight Center, AL, Oct. 1978.

[81] S. G. Shiva, "A comparison of hardware description languages," NASA CR-157762, Marshall Space Flight Center, AL, Oct. 1978.

[82] Progress Report of the Working Group of the Conference on CHLD's, Oct. 1977.

[83] G. J. Lipovski, "On gray box descriptions of microprocessors," in *Proc. Int. Symp. CHDL's Applications*, pp. 184–186, 1975.

[84] Y. Chu, "Concepts of a microcomputer design language," in *Proc. Design Auto. Conf.*, pp. 45–50, 1979.

[85] R. A. Mueller and G. R. Johnson, "A generator for microprocessor assemblers and simulators," *Proc. IEEE*, vol. 64, pp. 921–931, June 1976.

[86] L. J. Hafer and A. C. Parker, "Register-transfer level digital design automation: The allocation process," *Proc. Design Auto. Conf.*, pp. 213–219, 1978.

[87] W. H. Huen and D. P. Siewiorek, "Intermodule protocol for register transfer level modules: Representation and analytical tools," *Proc. Symp. Comput. Arch.*, pp. 56–62, 1975.

[88] M. A. Breuer, "General survey of design automation of digital computers," *Proc. IEEE*, vol. 54, pp. 1708–1721, Dec. 1966.

TABLE I
IMPLEMENTATION DETAILS OF HDL's

Language	Reference	Adapted from	Implemented Machine	Implementation Language
ACDL	9	-	-	-
AHPL	10-12	APL	CDC-6400	SNOBOL
			DEC-10	FORTRAN
ALERT	13-14	APL	IBM 7094	-
APDL	15-16	ALGOL	CDC-G20	ALGOL-60
APL	7	-	many	assembly
APL•DS	17	APL	-	-
CASSANDRE	18	PL/1	IBM 360	P1/1
CASSANDRE	19	ALGOL	IBM 360,370	assembly
CDL	20-29	ALGOL	IBM 370	FORTRAN ASSEMBLY
CSL	30	ALGOL	IBM 370/155	BCPL
DDL	31-39	-	Harris 6024/3	-
DIGITEST II	40	P1/1: petry Nets	-	-
DIDL	41	-	-	-
DSDL	42	DDL	IBM 360	XPL
FLOWWARE	43	flowcharts	IBM 360/50	P1/1
		CDL	NOVA-80L	NOVA ASSEMBLY
FST	44	-	IBM 360	FORTRAN IV
GLIDE	45	-	-	-
HARGOL	46	ALGOL	-	-
HILO	47	-	ICL 1900	-
ISP				
ISPL	48-54	ALGOL	PDP-10	BLISS
ISPS				
LALSD	55-57	-	IBM 360/91	PL/1
			CDC 6400	SNOBOL
LASCAR	58	CASSANDRE	-	-
LCD	59	P1/1	-	-
LDT	60	RTL	Burroughs	ALGOL-58
LOGAL	61	RTL	UNIVAC 1108	FORTRAN IV
LOTIS	62	ALGOL	-	-
MDL	63	APL	-	-
MODEL/LINDA	64	-	-	-
OSM	65		ODRA-1305	PLAN
PMS	49	ALGOL	PDP-10	SNOBOL
RTL	6	-	CDC1604	ALGOL
RTS I	66	ALGOL	Siemens 4004/151	FORTRAN
RTS II	67-68	RTS-I	-	-
RTS III	69	CDL,RTS II	-	-
SDL	70	RTL	-	-
SDL II	71	ALGOL	-	-
SDL	72	-	-	-
SMITE	73	-	CYBER 174	-
SSM	74	-	-	-
VDL	75	-	-	-
V-PMS	76	PMS	-	-

NOTES: 1) "-" Indicates that the detail is either not available or not known.
2) No accuracy is claimed for the contents of this table.

TABLE II
CDL MICROOPERATIONS

erator	Example	Explanation
.EQ.	A.EQ.B	Expression is 1 iff (A) = (B)
.NE.	A.NE.B	Expression is 1 iff (A) ≠ (B)
.GT.	A.GT.B	Expression is 1 iff (A) > (B)
.LT.	A.LT.B	Expression is 1 iff (A) < (B)
.GE.	A.GE.B	Expression is 1 iff (A) ≥ (B)
.LE.	A.LE.B	Expression is 1 iff (A) ≤ (B)
.AND.,·	A.AND.B,A·B	Performs logical AND bit by bit
.OR.,+	A.OR.B,A+B	Performs logical inclusive OR bit by bit
.ERA.	A.ERA.B	Performs logical exclusive OR bit by bit
'	A'	Performs 1's complement of A
.ADD.	A.ADD.B	Binary sum of (A) and (B) or (A) + (B)
.SUB.	A.SUB.B	Binary difference of (A) and (B) or (A) − (B)
.CNTUP.	A.CNTUP.	Increments A by 1 or (A) ← (A) + 1
.CNTDN.	A.CNTDN.	Decrements A by 1 or (A) ← (A) − 1
−	A−B	Cascades registers A and B
.SHR.	A.SHR	Shifts A right one bit position, enters 0 at left
.SHL.	A.SHL.	Shifts A left one bit position, enters 0 at right
.CIR	A.CIR.	Circular (closed) right shift of A 1 bit
.CIL.	A.CIL.	Circular (closed) left shift of A 1 bit
.SHRA.	A.SHRA.	Arithmetic right shift of A 1 bit, no change in left most bit (sign bit)
=	A = B	Contents of A are replaced by contents of B

TABLE III
HDL COMPARISON

	ISP	CDL	AHPL	DDL
1) Software				
translator	PDP-10 BLISS	Many – Fortran/Assembly	CDC 6400 Fortran	Harris 6024 Iftran
simulator	PDP-10 BLISS	Many – Fortran/Assembly		Harris 6024 Iftran
hardware compiler	no	no	CDC 6400 Snobol	(partial)
portability	no	fairly	fairly	fairly
2) Level of Description	instruction set level	register transfer level	register transfer and below	register transfer and below
3) Modularity	yes	no	yes	yes
4) Logic Generation	no	not very well	yes	yes
5) Programming Ease	difficult	easy	difficult	difficult

TABLE IV
IBM 370 155 CPU TIMES FOR CDL SIMULATION

	PDP-8	INTEL 8080
Average time to simulate an instruction	0.25 s	1.5 s
Number of instructions/pass to assemble an instruction	0.8K	1.6K
CPU time for assembling a 20 instruction program	8[illegible] s	96 000 s

Note: $K = 1024$.

ORIGINAL PAGE IS
OF POOR QUALITY

TEST SET

HIGH LEVEL DESCRIPTION
USING HDL

TEST GENERATION

TRANSLATION

REFINE
DESCRIPTION

1

MICRO
CODE

MICROCODE
GENERATION

DATA BASE

SIMULATE

SYSTEM
BEHAVIOR

SIMULATION
PARAMETERS

DESIGN
CONSTRAINT,
STANDARD
CELL LIBRARY

HARDWARE
COMPILATION

CONVERSION

LOGIC DIAGRAMS

REFINE
LOGIC

PHYSICAL
CONSTRUCTION
(PARTITIONING,
ROUTING,
PLACING)

LOGIC SIMULATION
FAULT DIAGNOSIS
TIMING

2

LOGIC
BEHAVIOR

TESTING

PHYSICAL
SYSTEM

SIMULATION
PARAMETERS

FINAL
SYSTEM

Figure 1 : Digital System Design Automation Process

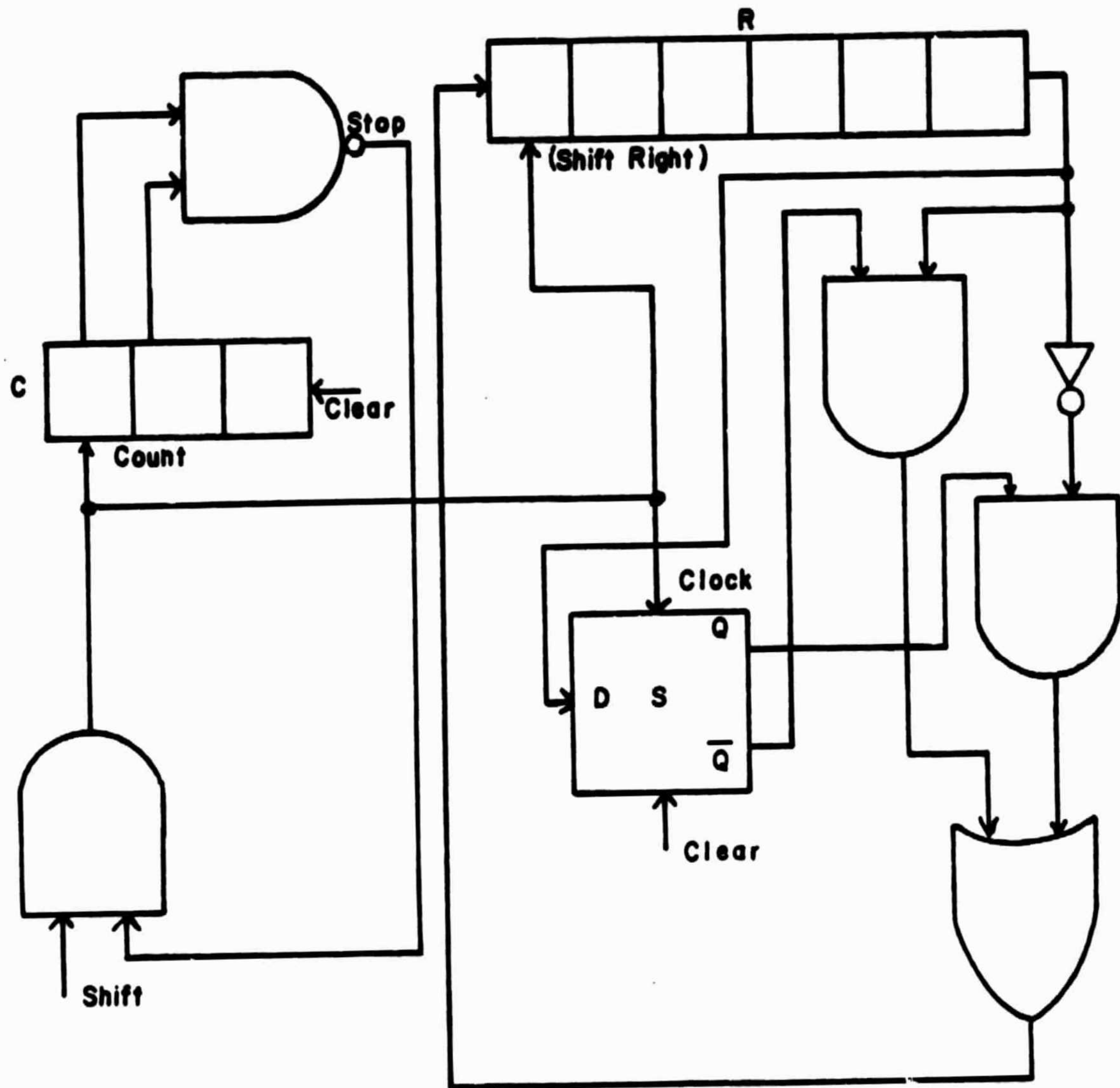


Figure 2(a) : Serial Twos Complementer Hardware Structure

```
$TRANSLATE

                                        T R A N S L A T I O N

     *MAIN
  1  C     **STORAGE***
  2   REGISTER,R(1-6),S
  3   SWITCH,SW(ON,OFF)
  4  C     **CONTROL**
  5   REGISTER,C(2-0),T
  6   CLOCK,P
  7  C     **PROCESSOR**
  8   /SW(ON)/T=1,C=0,S=0
  9   /T*P/IF(S.EQ.0)THEN(S=R(6),R=R(6)-R(1-5))ELSE
          (R=R(6)'-R(1-5)),IF(C.EQ.5)THEN(T=0)
          ELSE(C=C.CNTUP.)
 10   END
     $SIMULATE

                                        S I M U L A T I O N

 11  *OUTPUT    LABEL(1)=R,C,S,T
 12  *SWITCH    1,SW=ON
 13  *LOAD
 14   R=5
 15  *SIM       20,6
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 2(b) : CDL Description

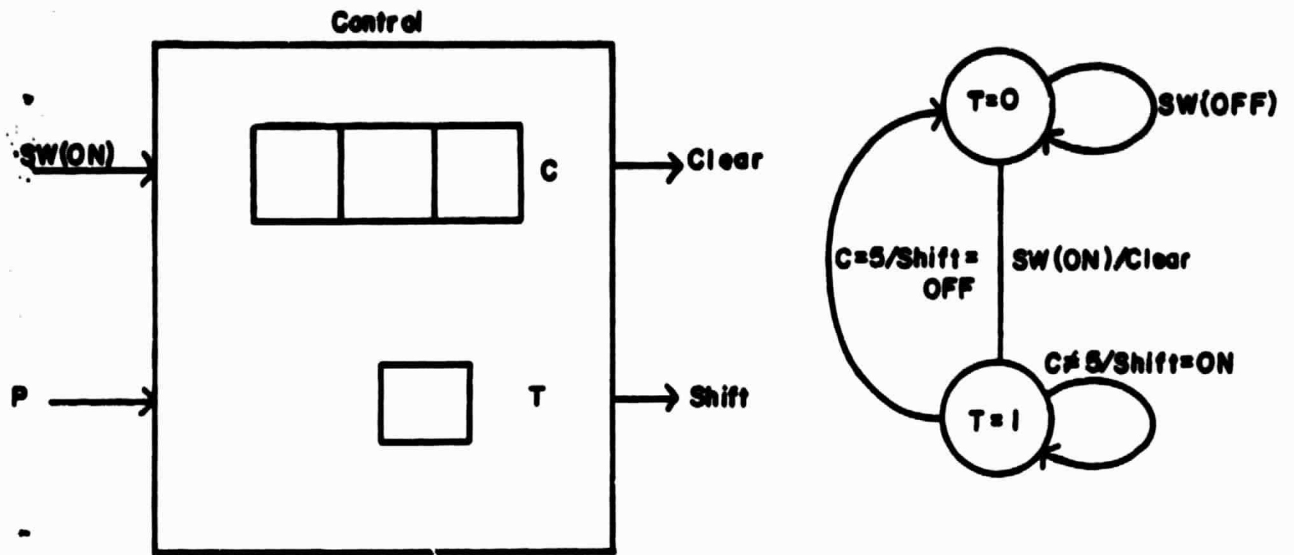


Figure 3 : Controller for the Twos Complementer

OUTPUT OF SIMULATION - OCTAL

```
SWITCH TRANSITION AT LABEL CYCLE  1

    SW      -> ON

       R = 05              C = 0              S = 0              T = 1
****************************************************************************
LABEL CYCLE 1                    TRUE LABELS                CLOCK TIME 1
                                 /T*P/
       R = 42              C = 1              S = 1              T = 1
****************************************************************************
LABEL CYCLE 2                    TRUE LABELS                CLOCK TIME 2
                                 /T*P/
       R = 61              C = 2              S = 1              T = 1
****************************************************************************
LABEL CYCLE 3                    TRUE LABELS                CLOCK TIME 3
                                 /T*P/
       R = 30              C = 3              S = 1              T = 1
****************************************************************************
LABEL CYCLE 4                    TRUE LABELS                CLOCK TIME 4
                                 /T*P/
       R = 54              C = 4              S = 1              T = 1
****************************************************************************
LABEL CYCLE 5                    TRUE LABELS                CLOCK TIME 5
                                 /T*P/
       R = 66              C = 5              S = 1              T = 1
****************************************************************************
LABEL CYCLE 6                    TRUE LABELS                CLOCK TIME 6
                                 /T*P/
       R = 73              C = 5              S = 1              T = 0
****************************************************************************

                  SIMULATION ENDS AFTER    6 REPETITIONS
                      FINAL LABEL CYCLE IS:
                                  6

*RESET     CYCLE,CLOCK


*LOAD
 R=21
*SIM       30,6
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 4(a) : Twos Complementer Simulation Results for R = $(05)_8$

```
OUTPUT OF SIMULATION - OCTAL


SWITCH TRANSITION AT LABEL CYCLE  1

   SW      -> ON

      R = 21              C = 0               S = 0              T = 1
*****,**********,**********,**********,**********,**********,**********,*****
LABEL CYCLE 1                  TRUE LABELS              CLOCK TIME 1
                               /T*F/
      R = 50              C = 1               S = 1              T = 1
*****,**********,**********,**********,**********,**********,**********,*****
LABEL CYCLE 2                  TRUE LABELS              CLOCK TIME 2
                               /T*F/
      R = 64              C = 2               S = 1              T = 1
*****,**********,**********,**********,**********,**********,**********,*****
LABEL CYCLE 3                  TRUE LABELS              CLOCK TIME 3
                               /T*F/
      R = 72              C = 3               S = 1              T = 1
*****,**********,**********,**********,**********,**********,**********,*****
LABEL CYCLE 4                  TRUE LABELS              CLOCK TIME 4
                               /T*F/
      R = 75              C = 4               S = 1              T = 1
*****,**********,**********,**********,**********,**********,**********,*****
LABEL CYCLE 5                  TRUE LABELS              CLOCK TIME 5
                               /T*F/
      R = 36              C = 5               S = 1              T = 1
*****,**********,**********,**********,**********,**********,**********,*****
LABEL CYCLE 6                  TRUE LABELS              CLOCK TIME 6
                               /T*F/
      R = 57              C = 5               S = 1              T = 0
*****,**********,**********,**********,**********,**********,**********,*****

                    SIMULATION ENDS AFTER    6 REPETITIONS
                        FINAL LABEL CYCLE IS:
                                  6
```

Figure 4(b) : Twos Complementer Simulation Results for R = $(21)_8$

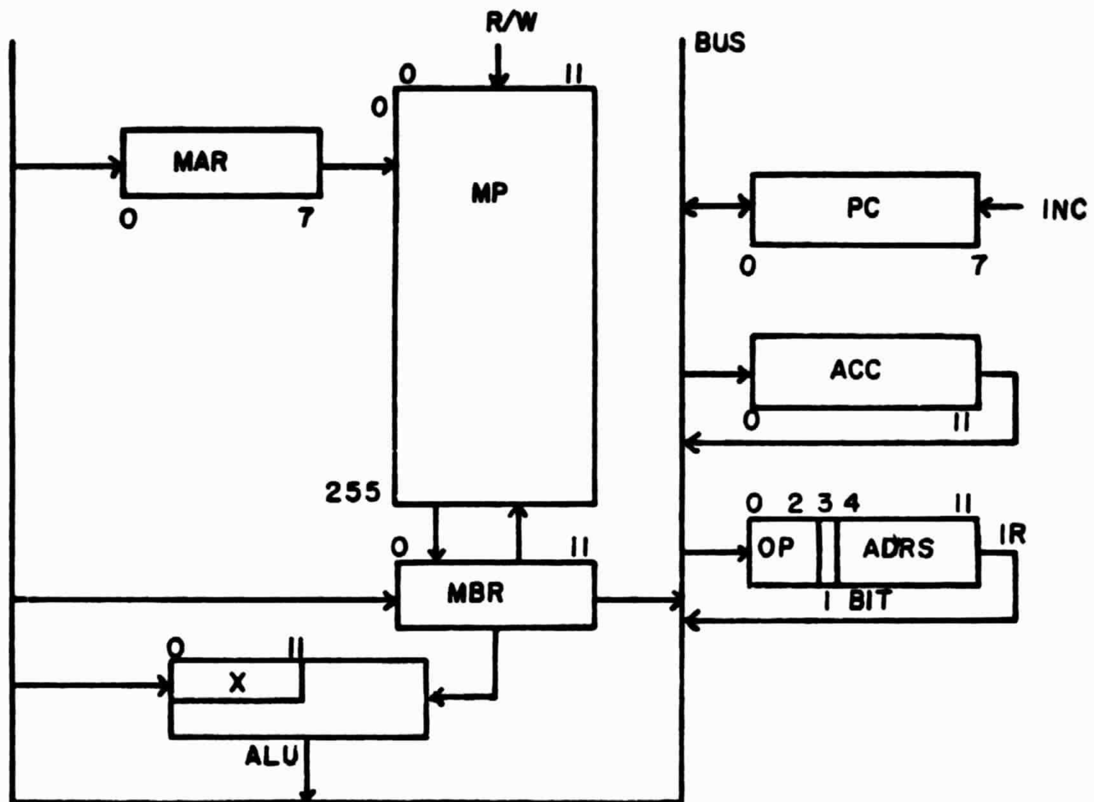


Figure 5(a) : Minicomputer Bus Structure

T R A N S L A T I O N

```
*MAIN
  1  REGISTER,MAR(0-7),MBR(0-11),PC(0-7),ACC(0-11),IR(0-11),X(0-11)
  2  SUBREGISTER,IR(OP)=IR(0-2),IR(IBIT)=IR(3),IR(ADR)=IR(4-11)
  3  MEMORY,M(MAR)=M(0-255,0-11)
  4  SWITCH,START(OFF,ON),RUN(OFF,ON),STATE(F,D,E)
  5  DECODER,K(0-7)=IR(0-2)
  6  TERMINAL,AND=K(0),TAD=K(1),ISZ=K(2),DCA=K(3),JSR=K(4),JMP=K(5),
     1       RET=K(6),HLT=K(7)
  7  CLOCK,P(3)
     C
     C ****    INITIALIZATION
     C
  8  /START(ON)/ACC=0,MAR=PC,IR=0,MBR=0,X=0,START=OFF,RUN=ON,STATE=F
     C
     C ****    FIRST THREE MINOR CYCLES OF FETCH
     C
  9  /RUN(ON)*STATE(F)*P(0)/MAR=PC
 10  /RUN(ON)*STATE(F)*P(1)/PC=PC.CNTUP,MBR=M(MAR)
 11  /RUN(ON)*STATE(F)*P(2)/IR=MBR
     C
     C ****    FOURTH FETCH MINOR CYCLE FOR NOT 'HALT' INSTRUCTION
     C ****    DEFER STATE IF INDIRECT, EXECUTE STATE IF NOT
     C
 12  /RUN(ON)*STATE(F)*P(3)*(HLT)'/IF(IR(IBIT).EQ.1)THEN(STATE=D)
          ELSE(STATE=E)
     C
     C ****    'HALT' (SINGLE CYCLE)
     C
 13  /RUN(ON)*STATE(F)*P(3)*HLT/RUN=OFF
     C
     C ****    DEFER STATE                    INDIRECT ADDRESS COMPUTATIONS
     C
 14  /RUN(ON)*STATE(D)*P(0)/MAR=IR(ADR)
 15  /RUN(ON)*STATE(D)*P(1)/MBR=M(MAR)
 16  /RUN(ON)*STATE(D)*P(2)/IR(ADR)=MBR(4-11)
 17  /RUN(ON)*STATE(D)*P(3)/STATE=E
     C
     C ****    EXECUTION OF 'AND' AND 'TAD'
     C
 18  /RUN(ON)*STATE(E)*P(0)*(AND+TAD)/X=ACC
 19  /RUN(ON)*STATE(E)*P(1)*(AND+TAD)/MAR=IR(ADR)
 20  /RUN(ON)*STATE(E)*P(2)*(AND+TAD)/MBR=M(MAR)
 21  /RUN(ON)*STATE(E)*P(3)*AND/ACC=MBR*X,STATE=F
 22  /RUN(ON)*STATE(E)*P(3)*TAD/ACC=MBR.ADD.X,STATE=F
```

ORIGINAL PAGE IS OF POOR QUALITY

Figure 5(b) : CDL Description

```
      C****      'ISZ' EXECUTION
      C
 23    /RUN(ON)*STATE(E)*P(0)*ISZ/MAR=IR(ADR)
 24    /RUN(ON)*STATE(E)*P(1)*ISZ/MBR=M(MAR)
 25    /RUN(ON)*STATE(E)*P(2)*ISZ/MBR=MBR.CNTUP.
 26    /RUN(ON)*STATE(E)*P(3)*ISZ/M(MAR)=MBR,IF(MBR.EQ.0)THEN
             (PC=PC.CNTUP.),STATE=F
      C
      C ****     'DCA' EXECUTION
      C
 27    /RUN(ON)*STATE(E)*P(0)*DCA/MBR=ACC
 28    /RUN(ON)*STATE(E)*P(1)*DCA/MAR=IR(ADR)
 29    /RUN(ON)*STATE(E)*P(2)*DCA/ACC=0,M(MAR)=MBR
 30    /RUN(ON)*STATE(E)*P(3)*DCA/STATE=F
      C
      C ****     'JSR' EXECUTION
      C
 31    /RUN(ON)*STATE(E)*P(0)*JSR/MBR=10000+PC
 32    /RUN(ON)*STATE(E)*P(1)*JSR/MAR=0
 33    /RUN(ON)*STATE(E)*P(2)*JSR/M(MAR)=MBR
 34    /RUN(ON)*STATE(E)*P(3)*JSR/PC=IR(ADR),STATE=F
      C
      C ****     RETURN EXECUTION
      C
 35    /RUN(ON)*STATE(E)*P(0)*RET/MAR=0
 36    /RUN(ON)*STATE(E)*P(1)*RET/MBR=M(MAR)
 37    /RUN(ON)*STATE(E)*P(3)*RET/PC=MBR(4-11),STATE=F
      C
      C ****     'JMP' EXECUTION
      C
 38    /RUN(ON)*STATE(E)*P(0)*JMP/PC=IR(ADR)
 39    /RUN(ON)*STATE(E)*P(3)*JMP/STATE=F
 40    END

      $SIMULATE


                                        S I M U L A T I O N

 41 *OUTPUT     LABEL(1)=MAR,IR,PC,ACC,MBR,X,STATE,RUN,START,M(7),M(8)
 42 *SWITCH     1,START=ON
 43 *LOAD
 44  M(0-6)=5,6,7,4,4092,0,0
 45  M(10-16)=5,774,1030,1026,2571,1543,3564
 46  PC=10
 47 *SIM        200,5
```

Figure 5(b) (Continued)

Operation Code	Mnemonic	Comments	
0	AND	(ACC) * (Mem) → ACC	AND Memory
1	TAD	(ACC) + (Mem) → ACC	ADD
2	ISZ	Increment memory and skip next instruction, if zero.	
3	DCA	Deposit and clear ACC.	
4	JSR	Jump to Subroutine, (PC) → MP(0)	
5	JMP	Jump	
6	RET	Return	
7	HLT	Halt	

NOTE: ( ) indicates "Contents of"

Figure 5(c) : Instruction Set

PROGRAM

Memory Location		Assembly	Binary	Decimal
10		AND 5	000 0 00000101	5
11	L1	TAD* 6	000 1 00000110	774
12		ISZ 6	010 0 00000110	1030
13		ISZ 4	010 0 00000100	1028
14		JMP L1	101 0 00001011	2571
15		DCA 7	011 0 00000111	1543
16		HLT	111 0 00000000	3584

Figure 5(d) : Program to Add Four Integers

Address	Contents	
0	5	DATA
1	6	
2	7	
3	8	
4	-3	COUNT (-4092 in ones complement 12 bits)
5	0	
6	0	
7	-	RESULT
8	-	NOT USED
9	-	
10	5	PROGRAM
11	774	
12	1030	
13	1028	
14	2571	
15	1543	
16	3584	

Figure 5(e) : Memory Map

Standard Cell Library

Logic Diagram

Standard Cell Selection (Manual)

Automatic Placement and Routing

Interactive Graphics

Mask Pattern Generation

Logic Simulation

Performance Simulation

Mask Analysis

Automatic Test Vector Generation

Mask Fabrication

Wafer Processing

Testing

Completed Device

Hardware Compilation (Logic Synthesis)

High Level Simulation

Refine

HDL Description

Figure 6 : CADAT System