A COMPARISON OF HARDWARE DESCRIPTION LANGUAGES

Prepared by

SAJJAN G. SHIVA

Department of Computer and Information Sciences

Final Technical Report

October 1978

Grant NSG-8057

"Evaluation of Digital System Design Languages"

George C. Marshall Space Flight Center

National Aeronautics and Space Administration

Huntsville, Alabama

Alabama Agricultural and Mechanical University

SCHOOL OF TECHNOLOGY

HUNTSVILLE, ALABAMA

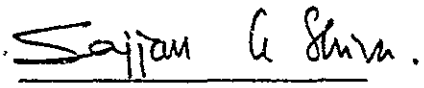
A COMPARISON OF HARDWARE DESCRIPTION LANGUAGES

Prepared by
SAJJAN G. SHIVA
Department of Computer and Information Sciences
Alabama A & M University
Normal, AL 35762

Final Technical Report
October 1978

Grant NSG-8057
"Evaluation of Digital System Design Languages"
George C. Marshall Space Flight Center
National Aeronautical and Space Administration
Huntsville, Alabama

Submitted by;



Sajjan G. Shiva
Principal Investigator

FOREWORD

This is a technical summary of the research work conducted since October 1, 1977 by the Alabama A & M University towards the fulfillment of Grant NSG-8057, from the George C. Marshall Space Flight Center, Huntsville, Alabama. The NASA Technical officer for this grant is Mr. John M. Gould, Electronics and Control Laboratory.

A COMPARISON OF HARDWARE DESCRIPTION LANGUAGES

Sajjan G. Shiva

ABSTRACT

Several high level languages have evolved over the past few years to describe and simulate the structure and behavior of digital systems, on digital computers. The characteristics of the four prominent languages (CDL, DDL, AHPL, ISP) are summarized. A criterion for selecting a suitable HDL for use in an automatic Integrated Circuit design environment is provided.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Any digital system can be described in the following six levels of complexity:

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; and
6) Circuit level where the gates and flip-flops are replaced by the circuit elements such as transistors, diodes, resistors etc.

Logic diagrams, Boolean equations and programming languages have been used as the media of description. The complexity of logic diagrams and Boolean equations increases as the system complexity increases and are not suitable for describing the hardware to a computer in a design automation environment, although the recent advances in computer graphics might make the input of logic diagrams to a computer easier. The common programming languages do not have all the features required to describe the hardware. Hardware description languages (HDL) evolved as a solution. An HDL is similar to any other high level programming language and makes the hardware designer's task easier by providing a means of:

1) Precise yet concise description of the system;
2) Convenient documentation to generate users manuals, service manuals, etc;
3) Inputting 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) Incorporating design changes and corresponding changes in documentation, efficiently;
6) Designer/user (teacher/student) communication interface at the desired level of complexity.

Several HDL's have been reported [1-30]. The translators and simulators are also written for some of these HDL's. The tendency has been to invent a new HDL to suit a perticular design automation environment, basically due to the difficulty in transporting the translators and simulators on to the new computing systems and extending these to accomodate the requirements of the new design environment. Attempts to standardize HDL's are underway.

Hardware Description Languages are designed to describe both the structural and behavioral characteristics of a digital system, to a computer. 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 design 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 so that a user interested in the

behavior of the system need not concern himself with the structure of the system. This also 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 an interactive environment.
5) The language should be easy to learn and remember, to accommodate the software shy, hardware designer, although the hardware engineer can not neglect the software aspects anymore, due to the impact of microprocessors. The design system should be portable, thus necessitating the the translators and simulators of HDL be written in higher level languages.
6) Two approaches to systems 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 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 LSI and MSI modules as system components should be straight forward, so is 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.

All the above requirements are not met by any one HDL now available. The solution has been to design a new HDL to suit

an individual design environment. AHPL, CDL, DDL and ISP have been the most popular languages, partly due to their early introduction as general purposes 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. A bibliography of the literature on HDL's is provided in [8].

The characteristics of CDL, ISP, AHPL and DDL are summarized in Chapters 2,3,4 and 5. A comparison of these characteristics to select a language suitable for use in an automatic integrated circuit design environment is provided in Chapter 6.

2. COMPUTER DESIGN LANGUAGE (CDL)

Computer Design Language was proposed originally by Chu [1-3]. A translator and simulator were written for a subset of this language [4]. Several modifications are made to this translator and simulator [5,6]. The present version [7] is implemented on IBM 370/115.

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 are 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 opeations 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. Gate delays and asynchronous operations can not be adequately described. It is not possible to include special hardware components like integrated circuits in a description. However, its simplicity of structure and its portability resulting from the FORTRAN implementation, have made CDL a popular language. The descripton of CDL Syn- tax and Semantics as accepted by the present version of tran- slator and simulator is presented.

2.1 SYNTAX RULES

Variables - variable names may contain 1 - 4 characters, the first of which must be alphabetic. The remaining characters may be alphabetic or numeric. Embedded blanks or special characters other than "+","-",",","*","/",".","'","$", and "=" are ignored. Variable names longer than 4 characters may be used and will appear on the listing, however their significance to the Translator is limited to the leftmost 4 characters. Some examples of variable naming follow:

Valid:

Name	CDL Name	
A1BC	A1BC	
ABbCD	ABCD	b is blank
A1#CD	A1CD	
START1	STAR	
START 2	STAR	

Invalid:

	Reason
1B3D	Nonalphabetic 1st character
AB.CD	contains special character

Reserved words - The following may not be used as variable names:
IF, THEN, ELSE, DO, CALL, RETURN, and END.

Constants - Constants may be entered in three forms subject to the following conditions:

Form	Max. Digits
Binary	32
Hexadecimal	8
Decimal	9

Hexadecimal constants are denoted by a colon (:) preceding the significant digits. Binary constants are preceeded by a semicolon (;).

In general, for numeric constants, blanks and special characters other than those listed in the discussion of variable names are ignored. Characters outside the permissible set for the form used are also ignored. Some examples of numeric constants follow:

Valid	Value
1234	1234_{10}
:100F	$100F_{16}$
;110011	110011_{2}
;12b101	1101_{2}

Invalid	Reason
A345	1st character not numeric (will be treated as variable name)

2.2 DECLARATION STATEMENTS

The following devices are permitted in CDL:

Registers	Terminals
Sub registers	Lights
Memories	Bus
Decoders	Clocks
Switches	

The syntax for a declaration statement is:

DEVICE, list

where the type of device begins in column two.

Mixed notation for devices and other keywords is shown in the following paragraphs. The first four characters are significant and must not contain embedded blanks or special characters. The following table relates keywords and acceptable abbreviations:

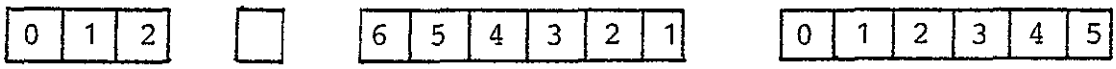
Keyword	Abbreviation
REGISTER	REGI
SUBREGISTER	SUBR
MEMORY	MEMO
DECODER	DECO
SWITCH	SWIT
TERMINAL	TERM
LIGHT	LIGH
BUS	BUS
BLOCK	BLOC
CLOCK	CLOC

Note that the comma trailing the device name is mandatory.

REGISTER DECLARATION:

REGI,A(0-2),R,F(6-1),G(0-5)

This declares the following:



SUBREGISTER DECLARATION:

SUBREGISTER, G(OP)=G(0-2),F(OR)=F(6-4)

The subregister is always used with a register name, and it refers to a part of that register. All referenced registers must have been previously declared.

A general tendency is to give two subregisters the same name.

e.g. SUBR,R(OP)=R(0-3),A(OP)=A(0-3)

This is incorrect!

A correct statement of the above would be:

SUB,R(OPR)=R(0-3), A(OP)=A(0-3)

MEMORY DECLARATION:

MEMO, M(R)=M(0-77,0-10), N(J)=N(0-6,3-1)

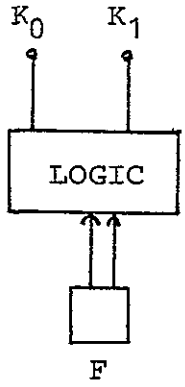
M and N are the names of the memories; R and J are the corresponding address registers which should have previously been declared; 0-77 and 0-6 represent the limits of the addresses of the words in the memories; 0-10 and 3-1 represent the order of the bits of each word.

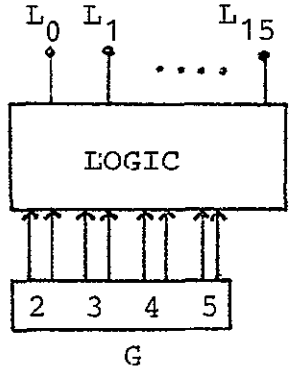
DECODER DECLARATION:

A decoder is a logic network which translates each value of the contants of a register to one and only one of the outputs;

DECO, K(0,1)=F,L(0,15)=G(2-5)

This declares the following:





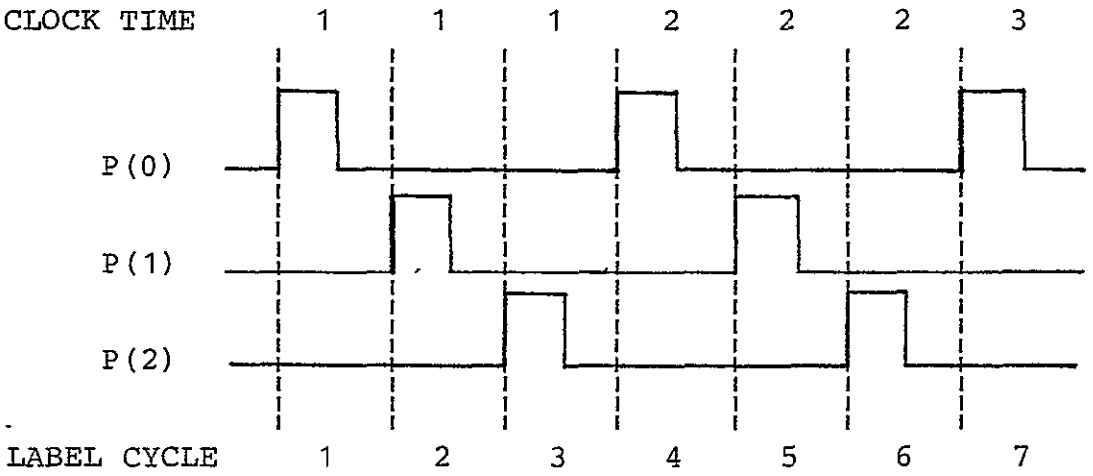
Where F and G have been previously declared.

CLOCK DECLARATION:

A clock may be specified for event synchronization.

CLOCK, P(2)

This declaration defines three clocks, P(0),P(1),P(2). The impulse diagrams are assumed to be the following:



The time intervals between the impulses given by the clocks are the same. A clock may be referenced only in the expression of the label.

During execution, a clock cycle is designated on the simulation printed results as clock time.

SWITCH DECLARATION:

SWITCH,STRT(OFF,ON),SENS(P1,P2,P3)

The switch names are STRT and SENS the positions for STRT are OFF and ON and for SENS they are P1, P2, and P3 where OFF and P1 are the respective initial positions.

In later references a switch is either checked for one of its positions, or set to one of its positions. When a switch is checked for a position, it has the form:

NAME(POS) e.g. SENS(P2)

when setting a switch to a position:

NAME=pos e.g. SENS=P2

NOTE: A maximum of 10 switch positions is permitted.

TERMINAL DECLARATION:

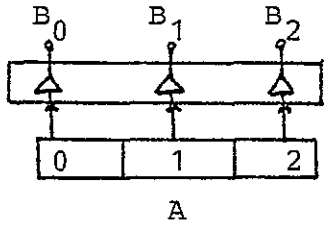
A terminal statement can rename a terminal or describe a logic network.

```
REGISTER, A(0-2)
TERMINAL, BO=A(0)',B1=A(1)',
          B2=A(2)'
```

or using subscripted terminals

```
REGISTER, A(0-2)
TERMINAL, B(0-2)=A(0-2)'
```

Both of these describe the following:



Referring to the decoder example:

```
TERMINAL, ADD=L(0),
          SUB=L(1),
          JUM=L(2),
          etc.
```

LIGHT DECLARATION

LIGHT, RUN(ON,OFF), FINI(OFF,ON)

The lights RUN and FINI, each have two positions, with the initial position being ON for RUN and OFF for FINI.

A reference where a light is checked for its position,

has the form:

NAME (POSITION) ex: RUN (ON)

When setting the position of a light, the following format is used:

NAME=POSITION ex: RUN=OFF

BUS DECLARATION

BUS, INTERNAL (15-0), DATA (7-0)

This declares INTERNAL bus with 16 lines and DATA bus with 8 lines.

BLOCK DECLARATION:

In order to avoid the repeated writing of a group of microstatements, the block statement and do statement are created. The block statement declares the name for a group of microstatements. Whenever these microstatements are declared in an execution statement, a do statement is used to call them.

Example: Serial Complement

```
REGISTER,   T(1-5),A(5-1)
SWITCH,     START(ON)
CLOCK,      P
BLOCK,      SERCOM(A=A(1)'-A(5-2))
/START(ON)/T=20
/T(1)*P/    DO/SERCOM,T(1,2)=01
/T(2)*P/    DO/SERCOM,T(2,3)=01
/T(3)*P/    DO/SERCOM,T(3,4)=01
/T(4)*P/    DO/SERCOM,T(4,5)=01
/T(5)*P/    DO/SERCOM,T(5)=0
```

NOTE: DO is followed by a slash'/' and then the BLOCK name.

Another example of Block Declaration:

```
BLOCK, PAR(A=B,R=0),CYC(A=A.COUNT.,
      IF (A.EQ.1)THEN(R=1)ELSE(R=0))
```

All of the microstatements following the block names (PAR,CYC) will be executed when DO/PAR or DO/CYC are called.

2.3 CONTINUATIONS

ONLY declaration statements may be continued onto other cards by placing a '1' in column one of the subsequent cards. Label statements and switch statements may be continued on subsequent cards by leaving column one blank.

Declaration and Label statements are limited to 250 terms, where a term is considered to be:

a device name, a variable name, a constant, or any of the following special characters:

"+","-","*","!","$","/","(",")", or "."

Example:

```
REGI,A(0-2),
1    B(0-6),
1    C(6-1)
/K*A(2)/ B=B.COUNT.
         C=C.COUNT.
```

2.4 COMMENT CARDS

A comment may be made by placing a 'C' in column one. The comment will be ignored by the translator. Comments are not continued in the conventional manner, rather a 'C' in column

one of every subsequent card will continue the comment.

Example:

```
C  SIMULATION OF A SECOND
C  GENERATION COMPUTER
```

2.5 OPERATORS

STANDARD OPERATORS

The following standard operators are available in CDL:

SYMBOL	FUNCTION	EXAMPLE	EXPLANATION
' (Apostrophe)	Complement	A'	
= (Equal Sign)	Replace	A=B	Contents of A are replaced by contents of B.
- (Dash)	Concatenate	A-B	Contents of A & B are placed side by side.
+ (Plus Sign)	Logical OR	A+B	Bit-wise OR (A and B must be conformal).
* (Asterisk)	Logical AND	A*B	
.EQ.		A.EQ.B	Gives '1' if A and B are equal.
.NE.		A.NE.B	Gives '1' if A and B are unequal.

SPECIAL OPERATORS

Special operators can be established by the user through a separate subprogram. It is referenced by a symbolic name delimited by periods.

Example:

```
*OPERATOR, X(1-4).COUNT.
// IF (X(4).EQ.0)THEN(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
```

The first line is a heading line of the subprogram: *OPERATOR (or *OPER) specifies the type of the subprogram, it is followed by a comma and by the name of the first argument, the name of the operator enclosed in a pair of dots, and by the name of the second argument if the operator is binary. If the arguments represent more than one bit, the bit addresses must follow the argument's name in parenthesis, e.g., S(1-4).

The subsequent lines are headed by a blank label, i.e., two slashes. This indicates an immediate execution of the operations when the operator is called. Following the blank label, there must be an expression, which may be a conditional expression, giving the result of the output terminal. The subprogram must be ended with a RETURN and an END statement.

RESTRICTIONS:

The following special operators are built in for Simulation and they may not be defined by separate subprograms.

A.ERA.B	performs the exclusive OR of A and B
A.ADD.B	performs the algebraic sum of A and B, an overflow bit is discarded
A.SUB.B	performs the algebraic sum of A and the complement of B, an overflow bit is discarded
A.COUNT.	Adds one to A, an overflow bit is discarded
A.LT.B	gives one bit of output: 1 if the conditions
A.LE.B	algebraically less than, less or equal

A.GE.B greater or equal, greater than, are satis-
A.GT.B fied: 0 if the conditions are not met

2.6 MICRO STATEMENTS

An UNCONDITIONAL MICROSTATEMENT consists of a variable representing a storage element, the REPLACE OPERATOR and an expression.

Example:

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

NOTE: A given device or portion of a device must not appear on the left of a 'replace by' operator more than once in any set of microstatements to be executed during a given label cycle.

A CONDITIONAL MICROSTATEMENT has the following forms:

(a) IF(expression) THEN (microstatements)
IF (expression) has the value '1' then the operations indicated by the (microstatements) will be executed.

Example:

```
IF(A.EQ.B)THEN(R=0)
```

(b) IF (expression) THEN (microstatements) ELSE (microstatements) if the (expression) is true then the operations indicated by the (microstatements) immediately following THEN will be executed, otherwise the operations indicated by the (microstatements) immediately following ELSE will be executed.

Example:

```
IF(C.NE.D)THEN(R=0)ELSE(R=1)
```

Conditional statements may be nested in order to form a very powerful decision-making capability. An example of a nested appears in section 5.2. Note that each of the nested IF's must be enclosed in parentheses as shown in the following generalized example:

```
IF (exp 1) THEN (microstatements)
   ELSE (IF(exp 2) THEN (microstatements)
   ELSE (IF(exp 3) THEN (microstatements)
   ELSE (IF . . . . . .
   ELSE (microstatements)))))..)
```

2.7 SWITCH STATEMENTS

The Switch Statement has the following form:

```
/NAME(POSITION)/microstatements
```

where Name corresponds to a declared Switch Name.

Example:

```
SWITCH, STRT(OFF,ON), SENS(S1,S2,S3)
/STRT(ON)/A=0,F=1,SENS=S2
```

The indicated microstatements here would not be executed since STRT(ON) is FALSE.

2.8 LABELED STATEMENTS

The Labeled Statement has the following form:

```
/LABEL/microstatements
```

where;

LABEL = expression * clock

RESTRICTION;
The expression must not include any clock reference.

Example:

```
/K(O)*P/ A=B, B=A
```

2.9 END STATEMENT

The physical end of the description of a design is indicated by the word END.

2.10 CARD FORMAT

HEADING CARD

The main design and the user's defined operators must be preceded by heading cards as follows:

```
Col. 1-5 *MAIN
```

or

```
Col. 1-...*OPERATOR...
```

where no embedded blanks are allowed. The Operator card should contain the arguments and the name of the operator.

OTHER CARDS

Declaration cards, Labeled Statement cards, and the End Card may be punched anywhere in columns 2-72. Blanks may be used freely.

2.11 TRANSLATOR

The translator accepts the logical design written in CDL from punched cards. It translates the design into a form

suitable for simulation. This consists of various tables and a pseudo program called Polish String. If the $TRANSLATE card is punched PRIN in columns 14-17, the various tables and Polish String will be printed.

The Translator is called by a special control card having $TRANSLATE (or $TRANS) punched in col. 1-10. This card is followed by the deck of cards describing the logical design using CDL. The Translator remains in control until a new control card with $ in column 1 is read in. A typical deck set-up should appear as follows:

```
$TRANSLATE        ---Translator is called
*MAIN
  .
  .
  .
END
*OPERATOR,...     ---Translator is in control
   .
   .
   .
END
   .
   .
   .
$SIMULATE         ---Simulator is called
   .                  Simulator is in control
   .
   .
```

2.12 SIMULATOR

The Simulator consists of 5 parts: Loader, Output, Switch, Simulate, and Reset routines. The Loader accepts test

data from punched cards and stores them into memory or into specified registers of the designed computer. The Output routine handles the printout of the contents of the chosen registers, memory words and position of switches during the simulation. The Switch routine simulates the operation of the manual switches. The Simulate routine actually executes the test program. Reset routine reinitializes the Simulator.

The execution of the test program is controlled by a loop which is called the Label Cycle. During each label cycle, the following steps are taken: (a) If a manual switch opera- tion occurs, the corresponding executable statement for the switch operation is carried out. (b) All label values are evaluated. The activated label, i.e., the label expressions having the value TRUE, are accounted for. (c) The microstatements of the activated labels are carried out in two steps. First, all values to be stored in various registers and memory words are evaluated and collected. Then, the collected values are stored one after the other. (d) It is checked whether the simulation should be terminated.

If the Simulation is terminated, the Reset routine can be called and another set of data can be inserted as a test program.

Example: The following is a demonstration of a simulation with 2 test programs.

```
$TRANSLATE
*MAIN
.....
END
                CDL Design cards
*OPERATOR, ...
.....
END
$SIMULATE
*OUTPUT
```

```
*SWITCH
*LOAD
.....
*SIM...
*RESET
*LOAD
.....
*SIM...
```

LOAD ROUTINE

The Loader provides the storing of test programs. The data cards should use Col. 2-72, blanks may be inserted anywhere. Only declared full registers and memory words with constant addresses can be loaded with data. The format of the data cards is as follows:

Data loaded into a register: REG = d

Data loaded into memory words M(L)=d

or $M(L_1\text{-}L_n)=d_1,d_2,\ldots,d_n$

or $M(L_1\text{-})=d_1,d_2,\ldots,d_n$

Where REG is the name of the register whose contents must be set to the value d. M is the name of the memory and L_1 denotes word addresses. In the first case d is loaded into memory word M(L). In the second case, the words d_1-d_n are loaded into memory words with addresses L_1 to L_n consecutively. In the last case, the last address is defined by the number of numbers punched.

A data card may contain any number of lists separated by commas. There is no provision for a continuation card, thus each data card must start with the name of a register or a memory in column 2.

Example:

```
*LOAD
 R1=0,AC=20,SEP=72,M(0-3)=1,2,3,4,M(77)=345
 M(10-)=70,71,72,73,74,75,76,77,100
```

OUTPUT ROUTINE

The output routine provides the printed output of the contents of specified registers, memory words and positions of switches during the simulation. The following format is required:

Col. 1-7	*OUTPUT
	LABEL
Col. 11-15	or
	CLOCK
Col. 16-21	(n,m)=
Col. 22-72	list

Where the label or clock punch specifies the type of output required, n specifies the start of output and m specifies the frequency of the output. The list consists of the names of the registers, switches and memory words whose contents should be printed.

During the simulation, the output may occur at every mth label cycle or at every mth clocktime depending on the type, Label or Clock in Col. 11-15. If the design happens to have one clock, then the two types coincide.

Example:

```
*OUTPUT  CLOCK (1,10)=RR,START,M(0),M(777),
   AC,MQ,M(10),OVER
```

The list may be continued on the next card(s) provided that column 1 is left blank on the continuation cards. The output of all listed devices is given in hexadecimal, regardless of input format.

SWITCH ROUTINE

Manual switch settings are initiated by the Switch routine. The necessary information is given on *SWITCH cards. For each switch setting, a separate *SWITCH card is necessary. It has the following format:

Col. 1-7 *SWITCH
Col. 11-12 L,
Col. 13- NAME=POSITION

Where L specifies the Lth label cycle before which the switch operation occurs. The NAME corresponds to the name of the switch with POSITION as one of its positions declared. During the simulation, an output will occur after every switch setting with a heading stating the interrupt.

SIMULATE ROUTINE

The actual simulation starts by calling the Simulate routine using the control card with the following format:

Col. 1-4 *SIM
Col. 11- n,r

Where n and r are the terminating conditions, n is the maximum number of label cycles allowed, r is the allowed maximum number of consecutive label cycles such that the same group of labels is activated in the CDL program.

Example:

```
*SIM    400,3
```

RESET ROUTINE

The Reset routine reinitializes the Simulator to its initial conditions. It is called by a control card with the following format:

Col. 1-6 *RESET
Col. 11- (options)

The options may be one or more of the following terms separated by commas:

OUTPUT, resets the output requested previously, it is assumed that another *OUTPUT card will be given

SWITCH, resets the manual switch operations re-

quested it is assumed that another *SWITCH card will be given

CLOCK, resets the counter of the clock cycle

CYCLE, resets the label cycle counter and the clock cycle counter

Example:

```
*RESET          CYCLE,OUTPUT
```

The reset card then followed by other OUTPUT card, possibly other LOAD card with data, and by a SIM card.

CDL translator and simulator also have an extensive error diagnostics capability.

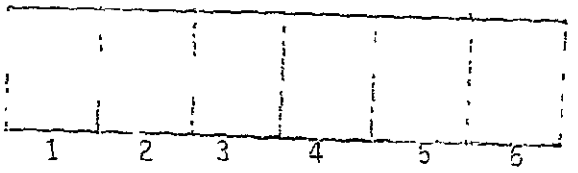
2.13 DESIGN EXAMPLES

The design process using CDL for a serial two's complementer circuit is illustrated [8] in Figure 2-1. A similar design example is provided in [9]. Several CDL descriptions can be found in [10].

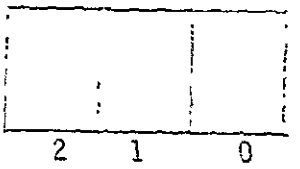
Figure 2-2 shows a sequential circuit along with the CDL description and simulation results.

A CDL description of a variable timer circuit shown in Figure 2-3 is provided in Figure 2-4. CDL is highly suitable for this level of description.

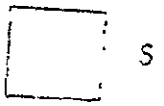
Basically, the circuit consists of a devide by 3600 circuit R along with a counter CT that counts the number of times R goes to zero. CT is compared with the input setting IN. IF IN equals CT an output pulse is given and the START flip-flop is reset to disable the clock. ABORT input clears



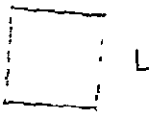
REGISTER, R (1-6)



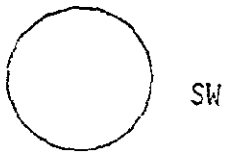
REGISTER, C (2-0)



REGISTER, S



LIGHT, L (on, off)



SWITCH, SW (on,off)

(A) (B)

(A) Storage structure

(B) CDL description

Figure 2-1 Serial Twos Complementer

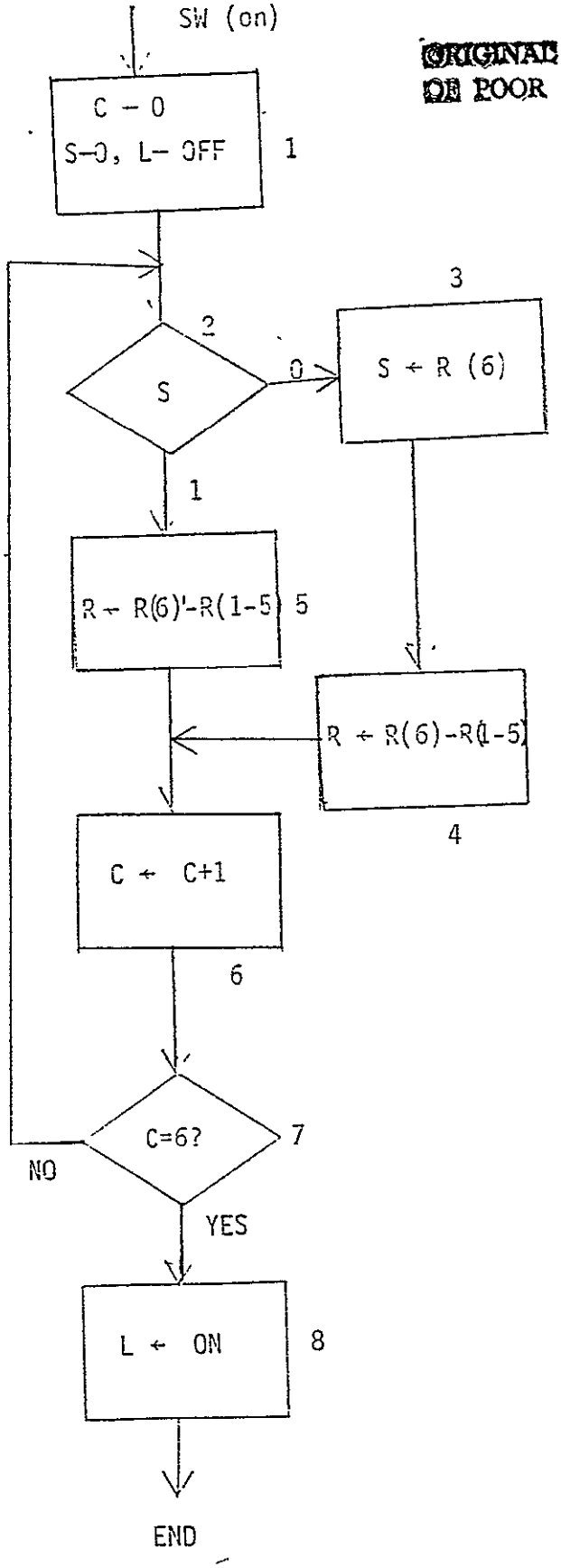
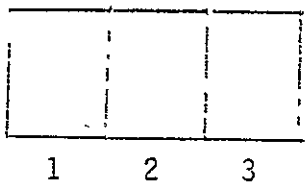


Figure 2-1 (C): Twos Complementer Sequence Chart



REGISTER, T (1-3)

clock  P

CLOCK, P

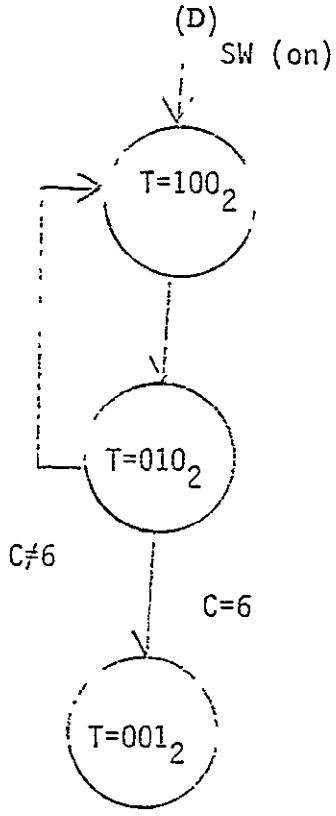


Figure 2-1 (D) Control elements
(E) State diagram
(F) CDL description of control sequence

END

(E)

/SW (ON) / T ← 100_2

/T(1)*P / T ← 010_2

/T(2)*P / IF (C=6) THEN (T ← 001_2) ELSE (T ← 100_2)

/T(3)*P / T ← 000_2

(F)

```
$TRANSLATE
*MAIN
C    STORAGE
 REGISTER,R(1-6),C(2-0),S
 LIGHT,L(ON,OFF)
 SWITCH,SW(ON)
C    CONTROL
 REGISTER,T(1-3)
 CLOCK,P
C    PROCESSOR
 /SW(ON)/T=;100,C=0,L=OFF,S=0
 /T(1)*P/T=;010,C=C.COUNT.,IF(S.EQ.0)THENS=R(6),R=(R(6)-R(1-5))
       ELSE(R=(R(6)'-R(1-5)))
 /T(2)*P/IF(C.EQ.;110)THEN(T=;001)ELSE(T=;100)
 /T(3)*P/T=;000,L=ON
 END

$SIMULATE
*OUTPUT    CLOCK(1,1)=R,C,S,T,L
*SWITCH    1,SW=ON
*LOAD
 R=5
*SIM       20,6
```

Figure 2-1 (G): CDL Complete Deck Set-Up

OUTPUT OF SIMULATION

```
**** SWITCH INTERRUPT ****
    SW   = ON
     R = ..05      C = ...0      S = ...0      T = ...4      L = OFF
*****************************************************************************
LABEL CYCLE    1                TRUE LABELS                CLOCK CYCLE    1
                               /T(1)*P/
     R = ..22      C = ...1      S = ...1      T = ...2      L = OFF
*****************************************************************************
LABEL CYCLE    2                TRUE LABELS                CLOCK CYCLE    2
                               /T(2)*P/
     R = ..22      C = ...1      S = ...1      T = ...4      L = OFF
*****************************************************************************
LABEL CYCLE    3                TRUE LABELS                CLOCK CYCLE    3
                               /T(1)*P/
     R = ..31      C = ...2      S = ...1      T = ...2      L = OFF
*****************************************************************************
LABEL CYCLE    4                TRUE LABELS                CLOCK CYCLE    4
                               /T(2)*P/
     R = ..31      C = ...2      S = ...1      T = ...4      L = OFF
*****************************************************************************
LABEL CYCLE    5                TRUE LABELS                CLOCK CYCLE    5
                               /T(1)*P/
     R = ..18      C = ...3      S = ...1      T = ...2      L = OFF
*****************************************************************************
LABEL CYCLE    6                TRUE LABELS                CLOCK CYCLE    6
                               /T(2)*P/
     R = ..18      C = ...3      S = ...1      T = ...4      L = OFF
*****************************************************************************
LABEL CYCLE    7                TRUE LABELS                CLOCK CYCLE    7
                               /T(1)*P/
     R = ..2C      C = ...4      S = ...1      T = ...2      L = OFF
*****************************************************************************
LABEL CYCLE    8                TRUE LABELS                CLOCK CYCLE    8
                               /T(2)*P/
     R = ..2C      C = ...4      S = ...1      T = ...4      L = OFF
*****************************************************************************
LABEL CYCLE    9                TRUE LABELS                CLOCK CYCLE    9
                               /T(1)*P/
     R = ..36      C = ...5      S = ...1      T = ...2      L = OFF
*****************************************************************************
LABEL CYCLE   10                TRUE LABELS                CLOCK CYCLE   10
                               /T(2)*P/
     R = ..36      C = ...5      S = ...1      T = ...4      L = OFF
*****************************************************************************
LABEL CYCLE   11                TRUE LABELS                CLOCK CYCLE   11
                               /T(1)*P/
     R = ..38      C = ...6      S = ...1      T = ...2      L = OFF
*****************************************************************************
LABEL CYCLE   12                TRUE LABELS                CLOCK CYCLE   12
                               /T(2)*P/
     R = ..38      C = ...6      S = ...1      T = ...1      L = OFF
*****************************************************************************
LABEL CYCLE   13                TRUE LABELS                CLOCK CYCLE   13
                               /T(3)*P/
     R = ..38      C = ...6      S = ...1      T = ...0      L = ON
*****************************************************************************
**** SIMULATION TERMINATED - NO TRUE LABELS
        *RESET    CYCLE,CLOCK
        *LOAD
         R=21
```

Figure 2-1 (H): Simulation Results for $R=5_{10}$

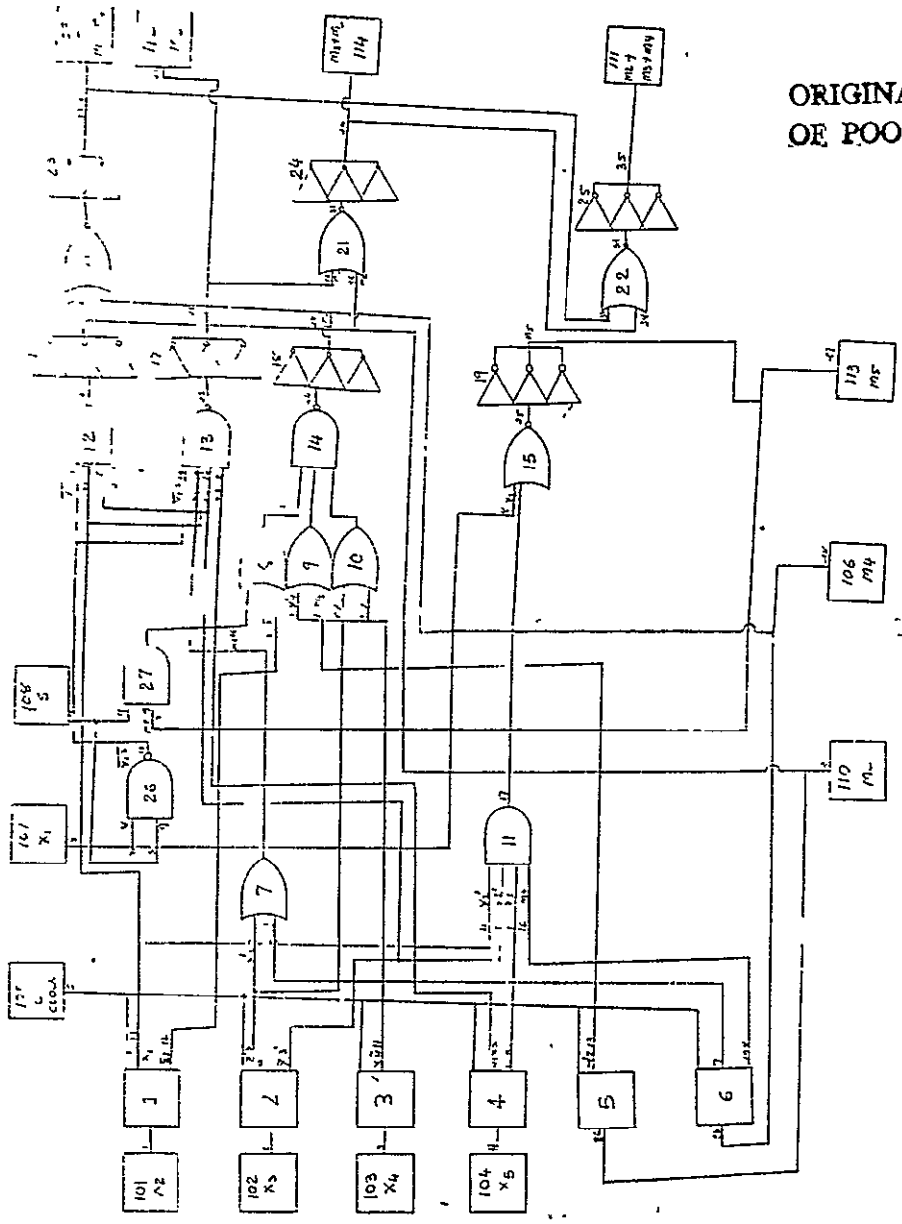


Figure 2-2 (A): A Sequential Circuit

$$M_2 = (X_3' + M_4') \cdot X_2' \cdot \overline{(X_1 \cdot S_1)}$$

$$M_3 = \overline{(X_1 \cdot S)} \cdot X_2' \cdot \overline{X_3'} \cdot X_5'$$

$$M4 = (M_5 \cdot S + \overline{X_2'}) \cdot (X_4' + M_2') \cdot (X_3' + X_4')$$

$$M_5 = X_1 + (\overline{X_2'} \cdot \overline{X_3'} \cdot \overline{X_5'} \cdot \overline{M_4})$$

Figure 2-2 (B): Boolean Equations for the Circuit

```
$TRANSLATE
*MAIN
 SWITCH, START(ON)
 REGI, FF(1-6), M2,M4,M5, X(1-6),ST(0-1)
 CLOC, P
 TERM, T1=M2+M4,M3=(X(1)*X(6))'*FF(1)*FF(2)'*FF(4),
1      T3=M2+M3, T4= M2+M3+M4
 /START(ON)/ST=1
 /P*ST(1)/FF=X,ST=2
 /P*ST(0)/M2=(FF(2)+FF(6))*FF(1)*(X(1)*X(6))',
         M4= (M5*X(6)+(FF(1))')*(FF(3)+FF(5))*
      (FF(2)+FF(4)),
       M5=X(1)+(FF(1)*(FF(2))'*(FF(4))'*(FF(6))'),
       ST=1
 END
$SIMULATE
*OUTPUT    CLOCK(1,1)=FF,M2,M4,M5,X,ST,START
*SWITCH    1,START=ON
*LOAD
 M2=1,M4=1,M5=1
*SIM       25,2
END OF DATA ON INPUT
```

NOTES: S is renamed as X(6) in the above description.
Modules 1-6 of the circuit, are named FF (1-6).

Figure 2-2 (6): CDL description of Sequential Circuit

OUTPUT OF SIMULATION

```
**** SWITCH INTERRUPT ****
    STAR = ON
    FF = ..00    M2 = ...1    M4 = ...1    M5 = ...1    X = ..00    ST = ...1    STAR =
*************************************************************************************
LABEL CYCLE   1              TRUE LABELS                 CLOCK CYCLE   1
                            /P*ST(1)/
    FF = ..00    M2 = ...1    M4 = ...1    M5 = ...1    X = ..00    ST = ...2    STAR =
*************************************************************************************
LABEL CYCLE   2              TRUE LABELS                 CLOCK CYCLE   2
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR =
*************************************************************************************
LABEL CYCLE   3              TRUE LABELS                 CLOCK CYCLE   3
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   4              TRUE LABELS                 CLOCK CYCLE   4
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
LABEL CYCLE   5              TRUE LABELS                 CLOCK CYCLE   5
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   6              TRUE LABELS                 CLOCK CYCLE   6
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
LABEL CYCLE   7              TRUE LABELS                 CLOCK CYCLE   7
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   8              TRUE LABELS                 CLOCK CYCLE   8
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
LABEL CYCLE   9              TRUE LABELS                 CLOCK CYCLE   9
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   10             TRUE LABELS                 CLOCK CYCLE   10
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
LABEL CYCLE   11             TRUE LABELS                 CLOCK CYCLE   11
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   12             TRUE LABELS                 CLOCK CYCLE   12
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
LABEL CYCLE   13             TRUE LABELS                 CLOCK CYCLE   13
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   14             TRUE LABELS                 CLOCK CYCLE   14
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
LABEL CYCLE   15             TRUE LABELS                 CLOCK CYCLE   15
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   16             TRUE LABELS                 CLOCK CYCLE   16
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
LABEL CYCLE   17             TRUE LABELS                 CLOCK CYCLE   17
                            /P*ST(1)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...2    STAR = ON
*************************************************************************************
LABEL CYCLE   18             TRUE LABELS                 CLOCK CYCLE   18
                            /P*ST(0)/
    FF = ..00    M2 = ...0    M4 = ...0    M5 = ...0    X = ..00    ST = ...1    STAR = ON
*************************************************************************************
```

Figure 2-2 (D): Simulation results for the Sequential Circuit

```
**********************************************************************************************
LABEL CYCLE   20                TRUE LABELS                CLOCK CYCLE   20
                              /P*ST(0)/
    FF = ..00     M2 = ...0     M4 = ...0     M5 = ...0      X = ..00      ST = ...1    STAR = ON
**********************************************************************************************
LABEL CYCLE   21                TRUE LABELS                CLOCK CYCLE   21
                              /P*ST(1)/
    FF = ..00     M2 = ...0     M4 = ...0     M5 = ...0      X = ..00      ST = ...2    STAR = ON
**********************************************************************************************
LABEL CYCLE   22                TRUE LABELS                CLOCK CYCLE   22
                              /P*ST(0)/
    FF = ..00     M2 = ...0     M4 = ...0     M5 = ...0      X = ..00      ST = ...1    STAR = ON
**********************************************************************************************
LABEL CYCLE   23                TRUE LABELS                CLOCK CYCLE   23
                              /P*ST(1)/
    FF = ..00     M2 = ...0     M4 = ...0     M5 = ...0      X = ..00      ST = ...2    STAR = ON
**********************************************************************************************
LABEL CYCLE   24                TRUE LABELS                CLOCK CYCLE   24
                              /P*ST(0)/
    FF = ..00     M2 = ...0     M4 = ...0     M5 = ...0      X = ..00      ST = ...1    STAR = ON
**********************************************************************************************
LABEL CYCLE   25                TRUE LABELS                CLOCK CYCLE   25
                              /P*ST(1)/
    FF = ..00     M2 = ...0     M4 = ...0     M5 = ...0      X = ..00      ST = ...2    STAR = ON
**********************************************************************************************
**** SIMULATION TERMINATED AFTER REQUESTED LABEL CYCLES
```

Figure 2-2 (D): (continued) Simulation results for the Sequential Circuit

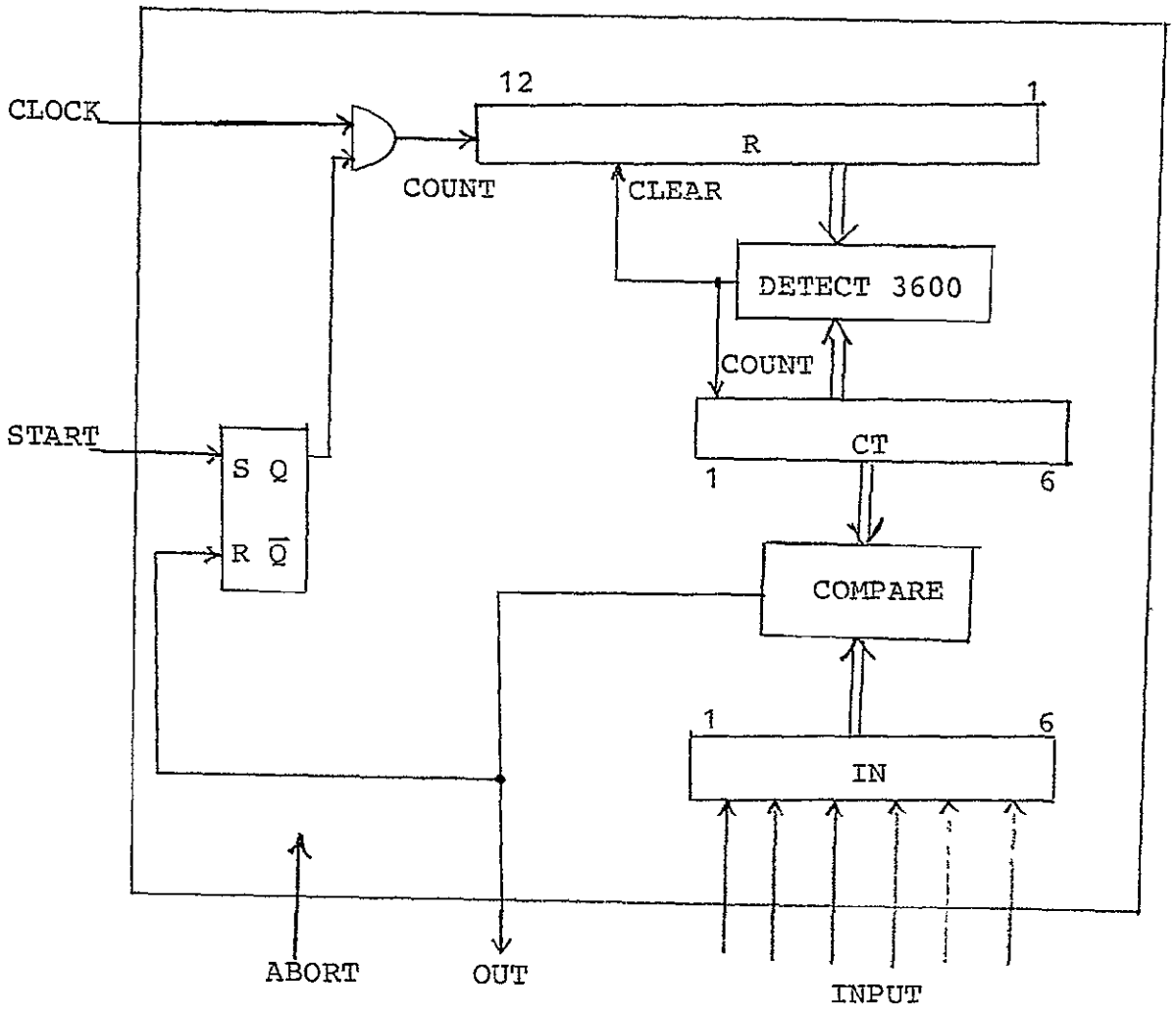


Figure 2-3: Variable Timer

```
$TRANSLATE
*MAIN
 REGI, R(12-1), CT(1-6), IN(1-6), START
 CLOCK, P
 SWITCH, ABORT (ON,OFF)
 LIGHT, OUT(ON,OFF)
 /ABORT (ON)*P/ R=3575, CT=0, START=1, OUT=OFF,
      ABORT=OFF
 /START*P/ R=R.COUNT., IF ((R(5)*R(10)*R(11)*R(12)).
      EQ.1) THEN (R=3575, CT =CT.COUNT.),
      IF (CT.EQ.IN) THEN (OUT=ON, START=0)
 END
$SIMULATE
*OUTPUT  CLOCK (1,1)=R, IN, ABORT, OUT, START
*SWITCH  1, ABORT=ON
*SWITCH  2, START=ON
*LOAD
      IN=5
*SIM  3600,600
```

Figure 2-4: CDL description of the Variable Timer

R, CT and START. START input sets START. In the CDL description, R is counted up from 3575 rather than zero, to save some simulation steps.

Each cell of R (or CT) consists of an 1-0 edge-triggered flip- flop F, a multiplexor M and a Nand-gate and the ith cell (i=1,12) is shown in Figure 2-5.

There is no direct way of describing the internal operation of this ripple counter in CDL. A description for a three stage counter is shown in Figure 2-6.

This description implies a parallel operation, according to CDL convention, rather than the ripple action of the counter.

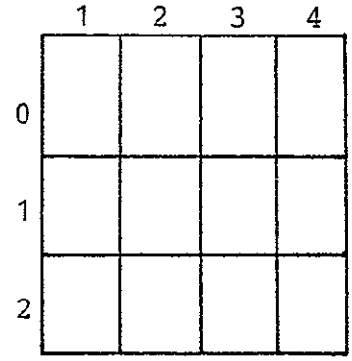
2.14 EXTENSIONS

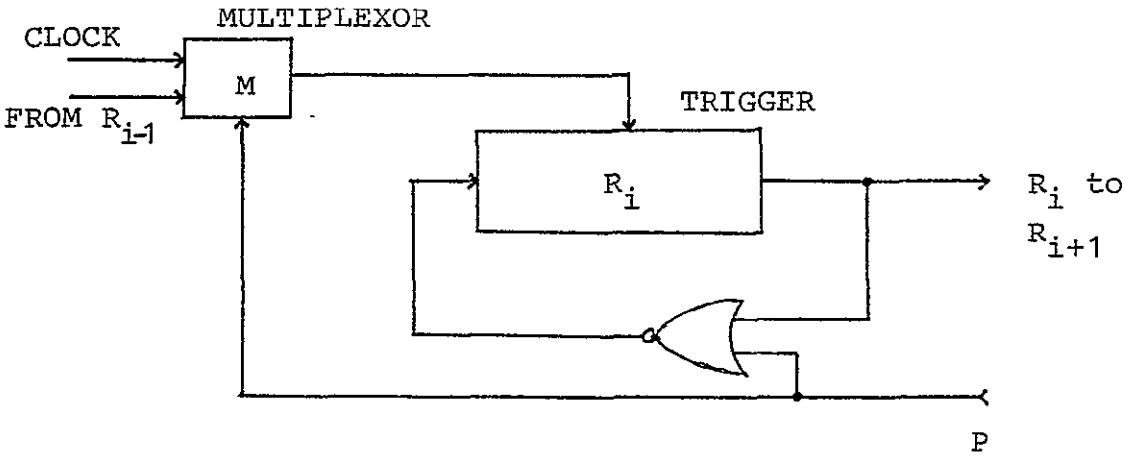
Some extensions to the language are reported. Bara and Born [6] report a version of CDL with the following additions:

ARRAY REGISTER

ARRAY REGISTER, AR(0-2,1-4)

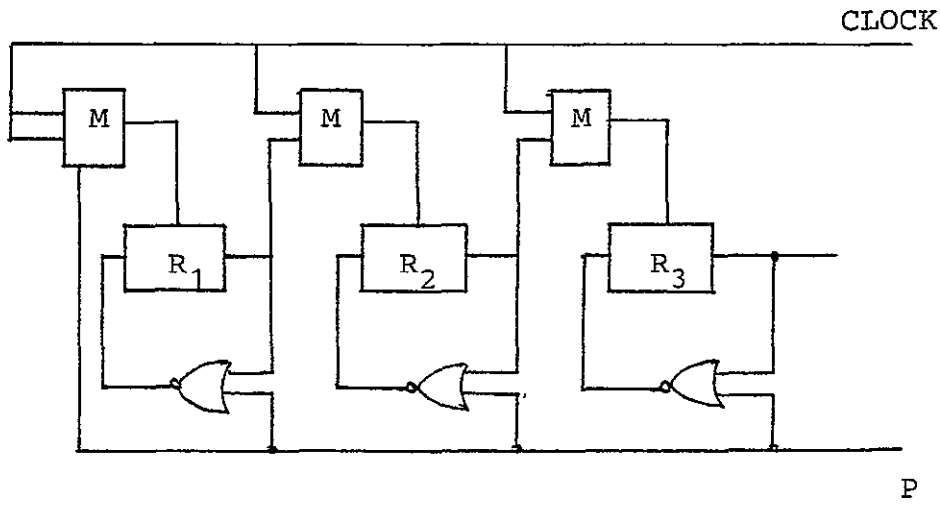
implies





Trigger = $(R_{i-1})\ \bar{P}$ + (Clock) P

FIGURE 2-5: A TYPICAL CELL OF THE RIPPLE COUNTER




```
/CLOCK/IF (P.EQ.1) THEN (R=0) ELSE
     (R(1) = R(1)', IF (R(1).EQ.1) THEN (R(2) = R(2)'),
          IF (R(2).EQ.1) THEN (R(3) = R(3)'))
```

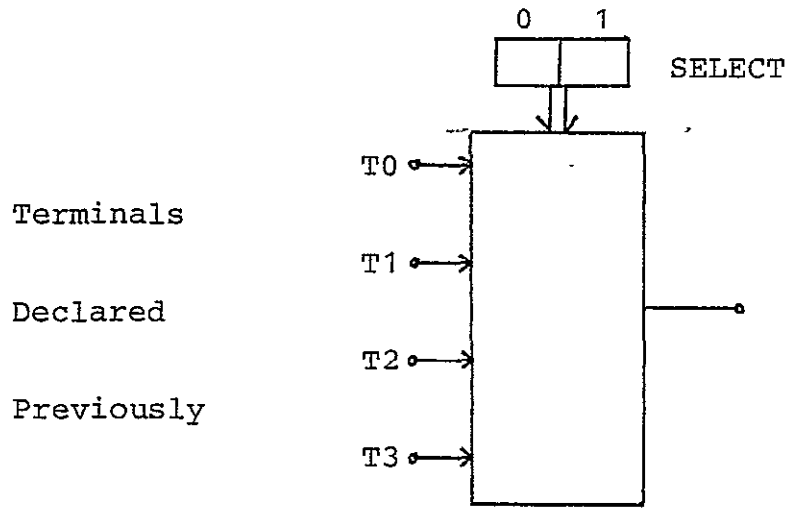
FIGURE 2-6: A 3 STAGE RIPPLE COUNTER

MULTIPLEXOR (DATA SELECTOR)

EX: REGISTER, SELECT (0-1)

DATA SELECTOR, DTA(SELECT)= DTA (T0-T3)

implies



PARTITION

Devides a bus into partitions.

BUS,DATA(0-7)

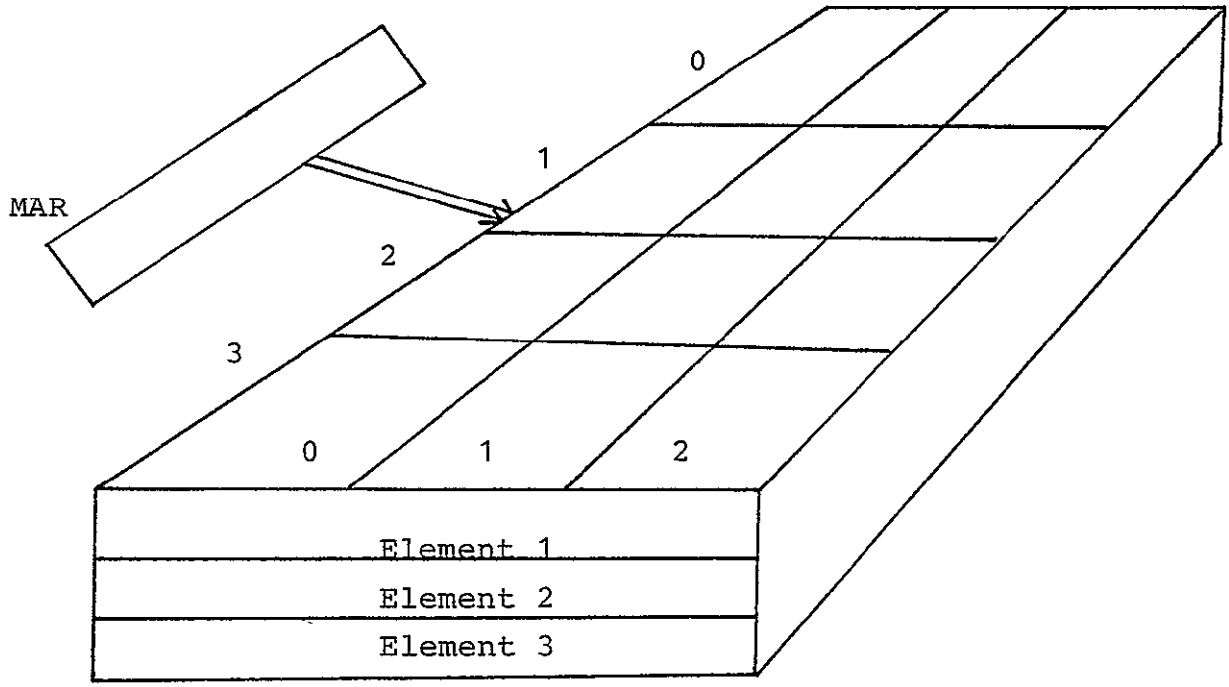
PARTITION, DATA(OP) = DATA (0-2)

STACK MEMORY

STACK MEMORY, Name (level, Mar) = Name (size, word length)

STACK MEMROY, STK (3, MAR) = STK (0-3, 0-2)

Implies



IOFLAG

IOFLAG,R declares a 1 bit flag R. It can be set (=1) or reset (=0). When set, the OUTPUT is enabled for the label cycle: when reset, the OUTPUT is disabled.

Following are some features of purdue extended CDL [5] simulator:

SET (M,7)	places a 1 in a single bit register M at the current time plus 7 cycles.
CLEAR (M,7)	opposite to SET.
COMP (M,7)	complement M after 7 cycles.
EXIT	is used to terminate simulation from design specifications. /ERROR*P/RUN<0,EXIT

Both the extensions have a complete set of operators to include decrement, circular shifts, shifts, ADD, SUB, MPY and DIV.

3. INSTRUCTION SET PROCESSOR (ISP)

The ISP notation was first introduced by Bell and Newell [11] as a formalism to describe digital systems at the programming level. The original notation was used mainly for publication purposes. A subset of the notation, ISPL [12] was implemented and was used in the design automation and architecture evaluation applications [13]. The present version ISPS is an extension of ISPL and is implemented on PDP-10 computers. A translator [14] and a simulator [15] are available.

ISP was designed to provide a precise description of computer systems as seen by the programmer. The description consists of four main parts: a declarative section, an interpreter, an instruction set description and an effective address calculation. The structural components are described in the declarative section. The effect of each instruction on the processor, registers, control flip-flops and memory forms the instruction set description. The interpreter section describes the fetch and execute cycles. The effective address part describes the processing of the address part of an instruction.

The language is suitable for the behavioral description at high levels, where timing information is absent. It allows parallel and sequential operations. Modular descriptions are possible at higher levels. The lower level description of the system is difficult because of the inability to describe details. The Syntax and Semantics of ISPS are described below.

3.1 SYNTAX RULES

VARIABLES (IDENTIFIERS)

Identifiers must start with a letter, can contain A-Z, 0-9 and "." and can be upto 80 characters long.

CONSTANTS

The following examples illustrate the formats allowed:

NOTATION	BASE	LENGTH (bits)	
4095	10	13	one bit longer than needed
"1000	16	16	
#1000	8	12	
'1000	2	4	
'110?	2	4	? is don't care
12?	10	UNKNOWN	

COMMENTS

A comment is indicated by a '!'. Everything from '!' to the end of the line in which it appears is treated as a comment.

ALIAS

Alias is an alternate name given to an identifier or a constant. It follows a "\" in an identifier declaration. It is a comment, not a usable name in the description.

ex: IR\INSTRUCTION REGISTER
'0110\MASK
#204\AND INSTRUCTION

NAME-PAIRS (RANGES)

A name-pair is an abbreviated notation for a list of consecutive constants.

ex: 3:5 is equivalent to 3,4,5
7:2 is equivalent to 7,6,5,4,3,2
4,5:7,8 is same as 4,5,6,7,8

3.2 OPERATORS

The following is the list of operators in increasing order of precedence:

transfer-op	::=	← | = | <=
or-op	::=	OR| XOR
and-op	::=	AND | EQV
relational-op	::=	EQL | NEQ | LSS | LEQ | GTR | GEQ | TST
add-op	::=	+ | =
mult-op	::=	* | / | MOD
shift-op	::=	SLO | SL1 | SLR | SLD | SRO | SR1 | SRR | SRD
concat-op	::=	@
unary-op	::=	NOT | + | -

← and = are equivalent and perform a logical transfer, while <= performs an arithmetic transfer. Other operators are self-explanatory.

3.3 ARITHMETIC REPRESENTATIONS

Four standard arithmetic representations are available: Twos - complement, Ones - complement, Signed - Magnitude, Unsigned - Magnitude. The selection of the representation in the context of an operation is indicated by,

{TC}, {OC}, {SM} or {US}.

ex: Y ← Y+{TC} 2 Twos complement addition.

3.4 DESCRIPTIONS

ISPS defines the structure and behavior of the components that make up a digital system. The structure of the components is defined in terms of the carriers used to transfer and store information. The behavior of the components is defined in terms of the sequence of operations that transform values contaned in the carriers and produce new values.

```
ISPS-declaration   ::= e-declaration
e-declaration      ::= e-head |
                       e-head := e-body |
                       ISPS-definition
```

An ISPS-declaration is the minimal parsing unit. An entity declaration (e-declaration) defines a hardware component which might have a structure and exhibit some behavior. The entity head defines the structural properties. The entity body, if present, defines the behavioral properties.

ISPS-definitions are described in 3.7.

ISPS ENTITIES -- STRUCTURE

The structural part of an ISPS entity is defined by the entity head (e-head):

```
e-head     ::= identifier fc-set fs-set
fc-set     ::= nil |
               () |
               (e-head-LIST')
fs-set     ::= nil |
               bit-fs-set |
               word-fs-set bit-fs-set
```

```
word-fs-set  ::= [name-pair]
bit-fs-set   ::= <> |
                 <name-pair>
```

The identifier distinguishes the entity from other entities defined at the same level or scope.

FORMAL STRUCTURE SETS

The formal structure set (fs-set) defines a carrier. The carrier may consist of a single register or an array of registers (a memory). Syntactically, there is no difference between the structure of a storage carrier (e.g. a register) and the structure of a non-storage carrier (e.g. a bus). A multi-register carrier specifies the dimensions of the array inside '[' and ']' (the '[' and ']' brackets indicate the presence of an addressing mechanism whose implemtnation is not specified). The dimensions of each register are specified inside '<' and '>'. The elements of the name-pairs (the dimensions) specify a naming convention for the 'words' and 'bits' of a carrier. An empty bit-fs-set (<>) stands for a single, unnamed bit.

Examples

```
Ir\Instruction Register<0:31>
Mp[0:255]<7:0>
```

The first example above, defines a 'register' (IR) whose structure consists of 32 bits (0,1,...30,31). The elements of the name-pair 0:31 specify the name of the bits. The second example shows the declaration of a 'memory' (Mp) whose structure consists of 256 words, each 8 bits long. The words are named 0,1,2,..255 while the bits inside each word are named 7,6,...,1,0.

The examples show that the bit and word names can be specified in ascending or descending order. In fact, the name-pairs do not even have to begin or end on 0, as the following example shows:

vma\Virtual. Memory. Address <13:35>

The VMA register is declared to be 23 bits long, the bits named as 13,14,...,34,35.

FORMAL CONNECTION SETS

The formal connection set (fc-set) defines an interface for connecting entities. In ISPS, the default implementation of a fc-set is by means of storage units which are loaded when the entity is activated. This default can be overruled and the interface implemented as a non-storage unit:

Examples:

```
ALU(Areg<0:15>,Breg<0:15>)<0:16>
F(REG[0:7]<0:7> {REF})
```

The first example defines the structure of a 'functional unit' (ALU) which consists of two interface registers (AREG<0:15> and BREG<0:15>). By default, any activation of ALU implies the storing of some values into the interface registers. After this initialization takes place, the interface registers can be read or written inside the body of ALU without affecting the registers from which the initial values came. This is the default mechanism for "parameter" passing in ISPS.

The second example presents a different type of "parameter". Its interface (REG[0:7]<0:7>) has been tagged with the string "[REF]" to indicate that the interface is not a storage unit, local to F, but that it is a REFerence to some external entity to be specified at the activation site. When F is activated, no transfer of data takes place. REG is simply "connected" to whatever entity was specified at the call site. This connection remains in effect throughout the length of the activation.

ISPS ENTITIES -- BEHAVIOR

```
e-body              ::= BEGIN section-LIST END |
                        BEGIN b-expression END |
                        e-head-LIST@
section             ::= section-header e-declaration-LIST
SECTION=HEADER      ::= ** identifier **
```

An entity body (e-body) defines the behavior of an entity. The most general case of an e-body consists of a list of sections (section-LIST), each consisting of a section-header followed by a list of declarations (e-declaration-LIST) local to the body. This type of body is bracketed by BEGIN/END pairs which can be substituted by '('/')' pairs. However, BEGIN can not be matched by ')' and '(' can not be matched by END.

Example:

```
Mini :=
BEGIN
**Mp\State**
     MP\Primary.Memory[255:0]<11:0>,
**Pc.State**
     PC\Program.Counter<7:0>,
     L\Link<7:0>,
     ACC\Accumulator<11:0>,
**External.State**
     IO.Reg<7:0>,
     Run<0>,
**Instruction.Format**
     IR\Instruction.Register<11:0>,
**Address.Calculation**
     Z\Address.Register<7:0>:=BEGIN.....END,
**Instruction.Execution**
     Fetch := BEGIN.....END,
     Execute := BEGIN.....END,
**Instruction.Cycle**
     Icycle := BEGIN.....END,
END
```

The example depicts the body of the declaration of an entity, in this case a minicomputer. The declarations inside the sections can be as large or complicated as one wishes to make them. They can, in fact, have bodies with local sections to any level of nesting.

> Declarations are grouped in sections as an abstraction mechanism. Application programs which manipulate ISPS parse trees will require specific sections to be present while possibly ignoring others.

Simpler bodies are defined by a b-expression (a behavioral expression) which can be thought of as a sequential or combinational network depending on the nature of the operations used and the implementation thereof. As in the previous case, the BEGIN/END bracketing the body can be substituted by '('/')'.

Example:

```
Z\Address.Register<7:0> :=
   BEGIN
   DECODE I.Bit =>       !test the indirect bit
      BEGIN
      Z←Adr,                    ! I.Bit = 0
      Z←Mp[Adr]<7:0>            ! I.Bit = 1
      END
   END
```

Notice the use of the carrier associated with Z (Z<7:0>) in the computation of the effective address. Algol-like scope rules are used in ISPS and non-local carriers can be accessed from inside a body (e.g. I.Bit, Addr, and Mp).

The third type of e-body is defined as the concatenation of one or more carriers using the @ operator. This is useful when defining alternative structures and naming conventions over previously declared carriers.

```
Examples
   IR\Instruction.Register<15:0>,
      OP\OP.Code<2:0>   :=IR<15:13>,
      I.Bit\Indirect<>  :=IR<12>,
      Adr<11:0>         :=IR<11:0>,
      IO.Bits<4:0>      :=IR<12:8>,
      Special <7:0>     :=IR<7:0>,

   MQ\Multiplier.Quotient<0:11>,
   ACC\Accumulator<0:11>,
   DACC\Double.ACC<0:23> :=Acc<0:11>@MQ<0:11>,
   CCodes[0:3]<>   :=PSW<15:18>,
```

In the above examples, several fields of IR have been defined as if they were independent registers (i.e. each field has its own name, with an optional alias, and a structure or dimension specification). The bit (or word) names used on the left hand side of a field specification are independent from the bit or word names used on the right had side. Both sides of a field definition must, however, specify structures of the same size (# word * # bits/word). The equivalence between the bits of the right hand side and the bits in the left hand side is obtained by aligning the leftmost bit of the leftmost word of the left hand side with the leftmost bit of the leftmost word of the right hand side. Thus, bit 4 of IO.BITS corresponds to bit 12 of IR, bit 3 to bit 11, bit 2 to bit 10, etc., etc..

In the second example, DACC<0:23> is defined as the concatenation of two registers, MQ and ACC. The registers appearing on the right hand side of a register definition might in turn be defined as subfields or concatenations of registers. Definition chains of this sort can be of arbitrary length.

The last example shows how different structures can be

mapped on top of a previously declared register. CCODES is defined as an ARRY of 4 1-bit registers. Thus, one can access the bits in the field PSW<15:18> using two alternative structures (i.e. an array of 1-bit registers or a 4-bit field). The equivalence of bits is as follows: The leftmost bit of word 0 of CCODES corresponds to bit 15 of PSW. Since this is the only bit of word 0, we continue on word 1, whose bit corresponds to bit 16 of PSW, etc., etc..

3.5 BEHAVIORAL EXPRESSIONS

```
b-expression        ::= s-action
s-action            ::= p-action-LIST NEXT
p-action            ::= action-LIST;
```

A b-expression defines the behavior of an entity. b-expressions are built by specifying the sequence of transformations and transfers of values stored in carriers. Simple b-expressions (actions) can be combined to build larger b-expressions by activating them in sequence (s-actions separated by NEXTs) or concurrently (p-actions separated by ';').

```
A←1; B←2 NEXT C←3
```

In the above example the first two transfers are executed in parallel and then, after their completion, the third one is performed. No synchronization must be assumed between parallel actions. Actions separated by ';' are considered to be 'order independent' and can be executed in any fashion, even sequentially. In particular, this means that one can not assume rules like: "all right-hand sides are evaluated first an then all transfers take place". The only requirement is that parallel actions are completed before proceeding beyond the following NEXT separator.

ACTIONS

```
action      ::= c-expression |
                identifier := action |
                control-action |
                conditional-action |
                BEGIN b-expression END
```

Actions are used to build complex behavioral expressions ranging from a primitive c-expression, to conditional or unconditional control flow operations, to a complex b-expression inside BEGIN/END pairs. The latter type can be used to build arbitrarily nested b-expressions. As in the case of an e-body, BEGIN and END can be replaced by '(' and ')'.

```
...NEXTa←1; B←2 NEXT (C←3 NEXT D←4); E←5 NEXT...
```

A←1 and B←2 are executed in parallel. Then, the sequence C←3 followed by D←4 is executed in parallel with E←5.

Actions may be labelled to allow the description of complex activities, including selection and premature termination or reinitialization of actions.

```
x := BEGIN.....END
```

The BEGIN/END brackets used to build compound actions can be optionally followed by a quoted-text or block name to provide the reader with some degree of visual identification of the levels of nesting:

```
x :=
   BEGIN | this is the outer block |
         .....
         BEGIN | this is the inner block |
               .....
         END | this is the inner block |
         .....
```

```
END / this is the outer block /
```

The quoted-texts attached to matching BEGIN/END pairs must be identical.

CONDITIONAL ACTIONS: IF and DECODE

```
conditional-action  ::= IF c-expression => action |
                        DECODE c-expression =>BEGIN
                        numbered-action-LIST' END
numbered-action     ::= action |
                        name-pair := action |
                        [name-pair-LIST'] := action |
                        OTHERWISE := action
```

Two operators, IF and DECODE, are used to specify conditional actions. If the value of the c-expression associated with an IF operation is non-zero the action following the => operator is executed, otherwise it is skipped.

```
IF Acc EQL X => PC←PC+2
IF Z => BEGIN ..... END
```

In the first example, the operator EQL defines a 1-bit result (0 stands for FAULSE, 1 for TRUE). Depending on the value of this bit, PC is incremented (1) or not (0). The second example shows that in general, the c-expression does not have to be 1 bit long. The action following the '=>' will be executed if ANY bit in the Z carrier is 1 (i.e. Z≠0).

The c-expression associated with a DECODE operator is evaluated and its value used to select one of the actions specified in the numbered-action-LIST' following the => operator. The c-expression is treated as an unsigned value. As in previous cases, the BEGIN/END brackets for the list of alternatives can be enclosed in '(' and ')'.

```
DECODE OP<1:0> =>
   BEGIN
   ACC←0,                  !OP<0:1> is 0
   ACC←ACC+M[Z],           !OP<1:0> is 1
   M[Z]←ACC,               !OP<1:0> is 2
   PC←M[Z]                 !OP<1:0> is 3
   END
```

When the DECODE operation specifies a large number of numbered-actions, it is sometimes difficult for a reader to associate the numbered-actions with the values of the c-expression which select them. In ISPS one can explicitly write the value of the c-expression associated with the action as a label-like action selector:

```
DECODE OP<1:0> =>
   BEGIN
   0 := ACC←0,             ! IF OP<1:0> is 0
   2 := M[Z]←ACC,          ! IF OP<1:0> is 2
   1 := ACC←ACC+M[Z]       ! IF OP<1:0> is 1
   3 := PC←M[Z]            ! IF OP<1:0> is 3
```

Notice that in the example we have altered the order of the actions. If explicit action selectors are used as in the example, one is free to write the actions in any order. For instance, when describing the instruction decoding in a computer, one might wish to group all the ADD instructions (half word, full word, double word, floating point, etc), followed by all the SUBTRACT instructions, etc. even though the operation codes are not consecutive.

A constant used to select a numbered-action identifies the value of the c-expression associated with the action. A name-pair used to select an action identifies a set of values of the c-expression associated with the action. The operator OTHERWISE is used to define a default action if the outcome of the c-expression is not covered by the other action-selectors.

If a constant appears in more than one action selector (either alone or as part of a @ [name-pair]) only the first action associated with the constant is executed (i.e. exactly one action can be executed as a result of a DECODE operation). Another use of the explicit selectors is given below:

```
Decode F =>
        BEGIN
0 :=    CR←M[S],
1 :=    CR←CR+M[S],
2 :=    ACC← -M[S],
3 :=    M[S]←Acc,
4;5 :=  ACC←ACC-M[S],
6 :=    IF ACC LSS 0 =>CR←CR+1,
7 :=    STOP(),
        END NEXT
```

Notice that there are two operation codes (4 and 5) associated with the Subtract operation.

It is a bad practice to mix actions with implicit and explicit action-selectors. The syntax allows it to handle the situation in which a designer is not yet sure of the proper constant action-selectors to use and wants to go ahead developing the ISPS description.

The basic rule to remember is that ALL outcomes of the c-expression must be acconted for. OTHERWISE must be used in some action if the number of actions is less than the number of possible values of the c-expression.

CONTROL ACTIONS: REPEAT, LEAVE, RESTART, and RESUME

```
control-action  ::= REPEAT action |
                    LEAVE identifier |
                    RESTART identifier |
                    RESUME identifier ;
```

An action that must be executed repeatedly (a loop) can be described by the use of the REPEAT operator preceding the action:

```
ICycle :=        !PDP-10 Instruction Cycle
   BEGIN
        REPEAT
              BEGIN
              IR←Memory[Pc] NEXT
              Pc←Pc + 1;VAM←IR<13:35> NEXT
              EA←VMA()<18:35> NEXT
              IExecute()
              END
   END
```

A looping action can be terminated by the use of the LEAVE operator as the following example shows:

```
I\Indirect <>:= VMA<13>,
X\IndeX<0:3>:=VMA<14:17>,
Y\Offset<0:17>:=VMA<18:35>,

VMA\Virtual Memory Address<13:35> :=     !PDP-10
BEGIN
REPEAT
   BEGIN
   IF X=>Y← Reg [X]+Y NEXT ! add the index register
   DECODE I =>
      BEGIN
      0:=  BEGIN VMA<13:17>←0 NEXT LEAVE VMA END,
      1:=  VMA←Memory[Y]<13:35> !indirect address
                                   loop
      END
   END
END
```

The LEAVE operator is not limited to loop termination. It

can be used to terminate the execution of any labelled action. The LEAVE operation must occur inside the action to which the label refers. It causes control to terminate that action, and continue normally, as if the action had been completed (any actons initiated during the execution of the action to be terminated and not yet completed are also terminated by the LEAVE operator).

The following 'procedure' searches the first 512 words of Mp for KEY:

```
S(Key<0:3>)<> :=
   BEGIN
   INDEX←0 NEXT
   REPEAT
        BEGIN
        IF MP[Index] EQL Key =>(S←1 NEXT LEAVE S)
                                        NEXT
        INDEX ← INDEX+1 NEXT
        IF INDEX EQL 512 => (S←0 NEXT LEAVE S)
        END
   END  ! end of S
```

The reactivation of an executing action can be forced by using the RESTART operation to indicate a termination of the action (as in the LEAVE operation) followed by a re-execution of the action. The RESTART operator must occur inside the action to be restarted (the pseudo LEAVE operation does cause termination of all actions initiated by the action to be restarted and not yet completed).

```
S(Key<0:3> <> :=
   BEGIN
   INDEX←0 NEXT
```

```
S1:=  BEGIN
      IF INDEX EQL 512 => (S<-1 NEXT LEAVE S) NEXT
      IF MP[INDEX]NEQ Key => (INDEX<-INDEX+ 1 NEXT
                              RESTART S1) NEXT
      S<-1 NEXT
      END
END   ! end of S
```

The RESUME operator provides another mechanism to terminate the execution of an action. As shown above, LEAVE is followed by the label of the action to be terminated. RESUME is followed by the label of the action whose execution is to be continued. As with LEAVE, the RESUME operation must occur inside the action to which the label refers. Any actions initiated during the execution of the action to be resumed and not completed are terminated. The following example shows the use of RESUME.

```
Interpreter :=
   BEGIN
   .....NEXT
   Icycle() NEXT
   IF Error EQL 1 => BEGIN.....END NEXT
   .....
   END,
Icycle :=
   BEGIN
   PC<-PC + 2 NEXT
   IR<-Rword(PC) NEXT
   DECODE IR<0:3> =>
      BEGIN
      .....
      ACC<-ACC + Rword(IR<4:15>)
      .....
      END,
Rword(Addr<0:11>)<0:15> :=
   BEGIN
```

```
IF Addr GTR Upper.Bound =>
   (Error← 1 NEXT RESUME Interpreter) NEXT
Rword← MP[Addr]
END,
```

In the example, procedure Interpreter activates procedure ICYCLE which fetches, decodes, and executes the instructions. In doing so, ICYCLE activates procedure RWORD which is used to access the memory (MP) of the machine. RWORD checks that the memory address is in bounds before performing the access operation. If a boundary error is detected, a flag (ERROR) is set and the rest of the operation of ICYCLE is aborted (by returning to procedure Interpreter, at the point where it activated ICYCLE). It is up to the 'resumed' procedure (INTERPRETER) to take the proper corrective action, if any. Notice that we could have let ICYCLE handle the error by terminating RWORD with 'LEAVE RWORD'. However, this would have meant that the ICYCLE procedure had to check the error flag (ERROR) after every call to RWORD. Depending on the size or complexity of the description, this might be undesirable.

Beware that these operators affect the sequence of operations and might be meaningless or unimplementable when used in parallel actions, e.g.:

```
X := (...NEXT...B←C;LEAVE X NEXT ...)
```

is ambiguous since no order of evaluation can be imposed on

```
B←C: LEAVE X
```

When 'LEAVE X' is executed, the transfer 'B ← C' may or may not have been executed.

3.6 QUALIFIERS

The qualifier set is used to specify lists of attri-

bute/value pairs which are used to define, amplify or modify the semantics of an ISPS description.

Example:

```
ALU (F<0:3>, A<0:15>, B<0:15>) <0:15> {SPEED:
     250, MODULE: SN74181} :=
```

3.7 ISPS DEFINITIONS

```
ISPS-definition    ::= DEFINE identifier := q-set |
                       DEFINE identifier := quoted-text |
                       DEFINE identifier := constant |
                       MACRO identifier m-parameter-set
                           := quoted-text |
                       REQUIRE ISP quoted-text
m-parameter-set    ::= nil |
                       () |
                       (identifier-LIST)
```

The reserved keyword DEFINE is used to name a q-set, a constant, or a quoted-text.

```
Define ROM := {MODULE: SN74187;SPEED: 40},
Define MSIZE := 255,
M1[0:MSIZE]<0:3> {ROM},
```

The reserved keyword MACRO provides a simple mechanism to declare test strings that are to be substituted for instances of the identifier in the ISPS description. Optional parameters can be specified by enclosing a list of identifiers inside parenthesis. These "formal parameters" are matched by corresponding 'actual parameters' at the expansion site.

The reserved keyword REQUIRE.ISP is used to signal the expansion of a an external file inside the ISPS description. The quoted-text describes the file name. The expansion takes

place at the point the REQUIRE.ISP construct appears:

```
REQUIRE ISP |MINLISP[L410MB25]|,
```

3.8 PREDECLARED ENTITIES

The following entities are predeclared in the language:

UNDEFINED

UNDEFINED is a predeclared entity which has some structure and exhibits some behavior, both unknown to the user. UNDEFINED<0:7> defines a carrier, 8 bits long, containing an undetermined value. Any number of "undefined" bits can be obtained by specifying a program bit range.

UNDEFINED() activates an entity with undetermined side effects. No assumptions about the values contained in ANY carriers can be made after an activation of UNDEFINED. Activations of UNDEFINED are guaranteed to terminate after some undetermined amount of time.

UNPREDICTABLE

UNPREDICTABLE is a predeclared entity which does not have a structure but which exhibits a totally unpredictable behavior. It is different from UNDEFINED() in that the latter preserves the flow of control. An activation like UNPREDICTABLE() is not guaranteed to terminate or that upon termination, control will return to the activation site.

NO.OP

NO.OP is a predeclared entity which does not have a structure and whose behavior has no side effects. NO.OP() can be used as a null action.

STOP

STOP is a predeclared entity which does not have a structure and whose invocation, STOP(), terminates the activation of all entities, including the invoking action.

DELAY

DELAY is a predeclared entity which does not have a structure and whose invocation, DELAY(c-expression), does not have side effects. DELAY terminates its activation after a number of application-defined time units given by the value of the c-expression.

WAIT

WAIT is a predeclared entity which does not have a structure and whose invocation, WAIT(c-expression), continuously evaluates the c-expression. WAIT terminates its activation when the value of the c-expression is not equal to 0.

3.9 RESERVED KEYWORDS and IDENTIFIERS

AND	
DECODE	
DELAY	
EQL	
EQV	
GEQ	
GTR	
IF	
K	when attached to a constant
LEAVE	
LEQ	
LSS	
M	when attached to a constant
MOD	

```
NEQ
NEXT
NOT
NO.OP
OC                when used as qualifier
OR
REF               when used as qualifier
REPEAT
RESTART
RESUME
SL0
SL1
SLD
SLR
SM                when used as qualifier
SR0
SR1
SRD
SRR
STOP
TC                when used as qualifier
TST
UNDEFINED
UNPREDICTABLE
US                when used as qualifier
XOR
WAIT
```

3.10 THE COMPLETE MINICOMPUTER

An example description.

```
Mini :=
   BEGIN
   ** Memory State **
```

```
MP\Primary Memory[0:255]<0:11>,
** Processor State **
PC\Program Counter<0:11>,
ACC\Accumulator<0:11>,
IR\Instruction Register<0:11>,
   OP\Operation<0:2> := IR<0:2>,
   IBIT<->Indirect Bit<>:=IR<3>,
   ADR<->Address<0:7>:=IR<4:11>,
** Effective Address Calculation **
Z\Effective Address<0:7> :=
   BEGIN
   DECODE IBIT =>
   0 := Z<- ADR,
   1 := BEGIN
        IF ADR EQL 0 => Z<- MP[0] + 1;
        IF ADR NEQ 0 => Z<- MP[ADR]
        END
   END
END,
** Instruction Cycle **
IEXEC\Instruction Execution :=
   BEGIN
   DECODE OP =>
   0\AND:=ACC<- ACC and MP [Z()],
   1\TAD:=ACC<-ACC+MP[Z()],  !2's Complement Add
   2\ISZ:=      !Increment and Skip if Zero
        BEGIN
        MP[Z]<- MP [Z()] +1 NEXT
        IF MP[Z] EQL 0 => PC<- PC +1
        END,
   3\DCA:= MP[Z()]@ACC <- ACC@#0000, !Deposit and
                                      Clear ACC
   4\JSR:= BEGIN         !Jump to SubRoutine
           Mp[0]<-MP[0] + 1 NEXT
           MP[MP[0]]<-PC NEXT
           PC<-Z()
```

```
            END,
      5\JMP*= PC - Z()                ! JUMP
      6\RET:= BEGIN          !RETurn from subroutine
            PC ← MP[MP[0]] NEXT
            MP[0] ← MP[0] - 1
            END,
      7\CTL:=
            BEGIN
                  IF IR <3> => PC← PC +1 NEXT
                  IF IR <4> => ACC← NOT ACC NEXT
                  IF IR <5> => ACC← ACC +1 NEXT
                  DECODE IR<6:7> =>
                     BEGIN
                     '10 :=ACC←ACC SRO 1,
                     '01 :=ACC←ACC SLO 1,
                     OTHERWISE := NOOP()
                     END NEXT
                  IF IR<8> => IF ACC LSS 0=> PC←PC+1;
                  IF IR<9> => IF ACC EQL 0=> PC←PC+1;
                  IF IR<10>=> IF ACC GTR 0=>PC←PC+1
                              NEXT
                  IF IR<11> => STOP()
                  END
            END
        END,
ICYCLE\Interpretation Cycle :=
   BEGIN
   REPEAT
      BEGIN
      IR← MP[PC] NEXT
      PC← PC + 1 NEXT
      IEXEC()
      END
   END
END
```

3.11 ISPS SIMULATOR

The command set of the simulator is summarized below:

START <label list> begins the simulation of procedures in label list.

EXIT terminates the simulation.

READ <dev: filename> enables the simulator to read and execute commands from the specified device.

DUMP is used to save the status of a simulation run.

DEFINE name = command-string $ defines a user command <name>. After this definition, the user can simply use <name> as a command to execute the corresponding command-string.

DDEFINE name deletes the user defined command.

TELLDEFINE prints the list of user defined commands.

DO command-string $ label, label,.... defines a command-string to be invoked when any of the procedures listed after the $ is entered.

ADO is similar to DO, but invokes the command-string when the procedures are terminated.

ECHO (DECHO) command sets (resets) an internal flag controlling the ECHOing the commands being read from a command file to a user terminal.

RADIX <base> is used to set the numeric base.

CONTEXT <varname> defines <varname> as a prefix for all names that are typed in future commands.

CTR <name> displays the value of the counter (s) associated with <name>.

OPAQUE <label list> and DOPAQUE <label list> are used to inhibit or enable the variable and label activity counters. If a procedure is OPAQUE then no activity counts are incremented during its execution.

VALUE and SETVALUE commands are used to set and interrogate the contents of ISP variables.

TRACE (DTRACE) <varlist> enables (disables) the tracing of variables during simulation.

BRAKE (ABRAKE) <label list> is used to enable the setting of Break points before (after) a procedure is excuted.

DBRAKE (DABRAKE) disables the break-point setting.

ICONNECT (OCONNECT) <identifier>, <channel>, <variable> is used to connect ISP variables to the system files which will act as sources (sinks) for variable values.

EVERY (AEVERY) count label, label,... forces a breakpoint every <count>th time one of the named procedure is entered (completed).

ONCE (AONCE) are similar to above except the breakpoint is forced only once after the <count>th time.

HELP tells the user about command names and their format.

WAIT makes the simulator to continuosly test the register used as parameter to wait until it is non-zero and then continue the execution.

DELAY procedure takes as parameter the number of simulated time increments that should go by before operations on this procedure continues.

SERIAL and PARALLEL accept procedure names as parameters and cause the register transfer code belonging to the named procedures to be SERIALized or unserialized.

PROCESS label-list (dentifies all procedures in the label-list as processes: DPROCESS undoes PROCESS. Any time a routine flagged PROCESS is called from ISP, an autonomous operating environment for that process is initiated. The caller continues without waiting for a signal from the called process, and may even terminate without further affecting the new process.

CRITICAL <label list> tags the procedures in label list to be non-interruptable. DCRITICAL clears this tag.

INITIATE (KILL) <label list> initiates (terminates) the list of processes in the label list.

TIME (DTIME) begins (terminates) the simulator's timing facility.

SETCLOCK <procedure> <value> sets the clock for that procedure at the new value. Each procedure in ISP has its own clock which increments as the register transfers proceed.

OPTIME <op-label> = <value> is used to establish the times associated with the individual register transfer operation. The default value for each is one.

4. A HARDWARE PROGRAMMINNG LANGUAGE (AHPL)

AHPL is based on the notational conventions of APL. Some special conventions are added to APL to take care of the hardware features like parallelism, asynchronous transfers and conditional transfers [16,17]. AHPL is a clock mode register transfer level language with the register as the primitive circuit description element. A hardware compiler capable of generating a wire list specifying the interconnection of available integrated circuits and a functional simulator which interprets the AHPL description and executes the connections and register transfers [18] are available.

AHPL is based on the philosophy that a digital system can be divided into two parts: a control section and a processing section. Specification of the processors is done at one level. Hierarchical descriptions of both structural and functional elements are possible through the subroutine feature of the language. Both parallel and sequential operations can be described, either by suppressing timing information completly or including it to a sufficiently high degree. Synchronous description facilities include tests for pulses and counting of pulses and delays. Asynchronous operations can be represented either by conditional statements or by implementing completion signals and using WAIT to indicate delay. The language as accepted by the compiler and the simulator [19] is described below. The simulator (HPSIM) is written in FORTRAN. The compiler (HPCOM) is written in SNOBOL. Both are implemented on CDC-6400 and DEC-10 systems.

4.1 SYNTAX RULES

VARIABLES:

Variable names may contain up to 20 characters, the first of which must be alphabetic. The remaining may be numeric or alphabetic. Only the first 10 characters are retained in the translator and simulator.

CONSTANTS:

Constants may be entered in decimal. A vector of binary constants should be separated by commas and placed in back slashes.

Examples:

```
\1,0,1,0,1\
  25
```

OPERATORS:

The following operators are allowed:

AND	&	
OR	+	
Exclusive-OR	@	
ALL BITS OR	+/	
ALL BITS AND	&/	
ENCODE	$	Ex: 5$13=/0,1,1,0,1/
TRANSFER	<=	(2 characters)
BRANCH	=>	(2 characters)
COMPLEMENT	¬	
CONCATENATE	,	
CONNECTION	=	

4.2 DECLARATIONS

Each AHPL module description begins with the declarations, the first statement being,

AHPL MODULE: module name.

The rest of the declarations have the format

TYPE: symbol <n> [m]; symbol <n>,[m];....;.... .

where the TYPE can be MEMORY, INPUTS, OUTPUTS, BUSSES, EXINPUTS, EXBUSSES, or CLUNITS. The integers n and m indicate the number of rows and columns of the facility. Either one or both n and m can be eliminated if their value is 1. They can also indicate a range n1:n2, m1:m2.

MEMORY, BUSSES, ONESHOTS and CLUNITS are local symbols. These are declared in each module they are used. When redeclared in other modules, these refer to a new value location.

INPUTS and OUTPUTS are semilocal symbols. When these symbols are redeclared in other modules, they refer to the same value location.

EXINPUTS and EXBUSSES declare Global symbols which can be valued externally and are common between all modules.

4.3 CONTROL SEQUENCE

The control sequence consists of a list of steps; each step starting with a step number followed by valid operations, separated by semicolon and ending with a period. For example,

```
3 AC <=IR&AC; OUT = MD;
=> (AC[0:3])/(2,3,5,6).
```

indicates a transfer, connection and a branch. The branch is to the statement 2,3,5 or 6 according to the value of AC [0:3]

is 8,4,2 or 1 respectively. Three formats are possible for branch operation:

```
=> (destination)                    unconditional
=> (expression)/(destinations)      multiple branch
=> operator (expression1, expression 2) (destination)
```

Where operator is the comparison operator between expressions 1 and 2 and can be NE, EQ, GT, GE, LT and LE.

4.4 COMBINATIONAL LOGIC UNITS (CLUNIT)

User can define combinational logic units at the beginning of the description and can use them later. A partial CLUNIT description of a 4 bit adder is shown below:

```
UNIT:  CLADD (A,B,CIN)
       INPUTS: A [4]; B[4];CIN [1].
       OUTPUTS:  CLADD [5].
       CLUNITS:  PG [3], SUM [1], CLA [6].
1.  I <= 4
2.  C [I] = CIN [0]
3.  I <= I-1
4.  NOTP[I], P[I], G[I]= PG[0:2] (A[I], B[I]).
5.  S[I] = SUM[0] (NOTP[I], P[I], C[I+1]).
6.  => NE (I,0)/(3)
7.  I <= 12
.
.
.
END.
```

The CLUNIT description starts with a UNIT:, followed by the module's name and the argument list. Input and outputs for the module are identified. Any other CLUNIT used in the module is then identified. The description ends with an END.

Each CLUNIT is a module that can be used in the description of the system. Facilities exist to imply several copies of the module (rather than sharing the module) in the description of the system.

4.5 COMMENTS

Comments can be placed anywhere in the AHPL sequence. They should be enclosed in double quotes.

4.6 SIMULATOR (HPSIM)

The communications with HPSIM follow the hardware description and include the following commands:

CLOCK LIMIT:

Tells HPSIM, the number of clock periods for which execution is to continue, if it does not reach a DEAD END.

BUSEFFECT -- nn:

Is used to assign values to the external inputs or busses. nn is the number of clock periods for which the corresponding line receives the values from the cards following the BUSEFFECT command.

OUTPUTS:

Enables printing the values of selected variables during the simulation.

SUPPRESS:

Command is used to suppress the printing of results during a specified step in the control sequence.

4.7 DESIGN EXAMPLE [19]

To illustrate the preparation of a circuit description for HPSIM and HPCOM consider a simple multiplier circuit with nine input lines and nine output lines as shown in Figure 4-1. In the reset state it waits for a 1 on line DATAREADY, which indicates that data is on the INPUTBUS lines. The four most significant bits of INPUTBUS are the first operand and the other four consititute the second operand. When the operands are accepted, the BUSY flip-flop is set to one and the multiplier starts the multiplication process. When done, BUSY is set back to zero, and the eight bit result is placed on the eight RESULT lines and a one on the DONE line. Then the multiplier goes to the reset state waiting for another set of data.

Figure 4-2 shows the input file to HPSIM. the first line in this figure assigns a name to the module description. This is followed by declaration of all lines and registers. The circuit requires three four-bit registers for the two operands and the intermediate results, a single flip-flop for the DONE indicator, and a two bit counter for the number of bits shifted out of the first operand register. The lines to be assigned values by the user are DATAREADY and INPUTBUS. These are, therefore, declared as EXINPUTS and EXBUSES respectively. The last of the declarations, CLUNITS, indicates the presence of combinational logic networks implementing a 2-bit incrementer and a 4-bit adder.

The circuit AHPL sequence follows the declarations. Step 1 receives the operands, resets the intermediate register (EXTRA). If there is a 1 on the DATAREADY line control proceeds to Step 2; otherwise, Step 1 remains active. Step 2,

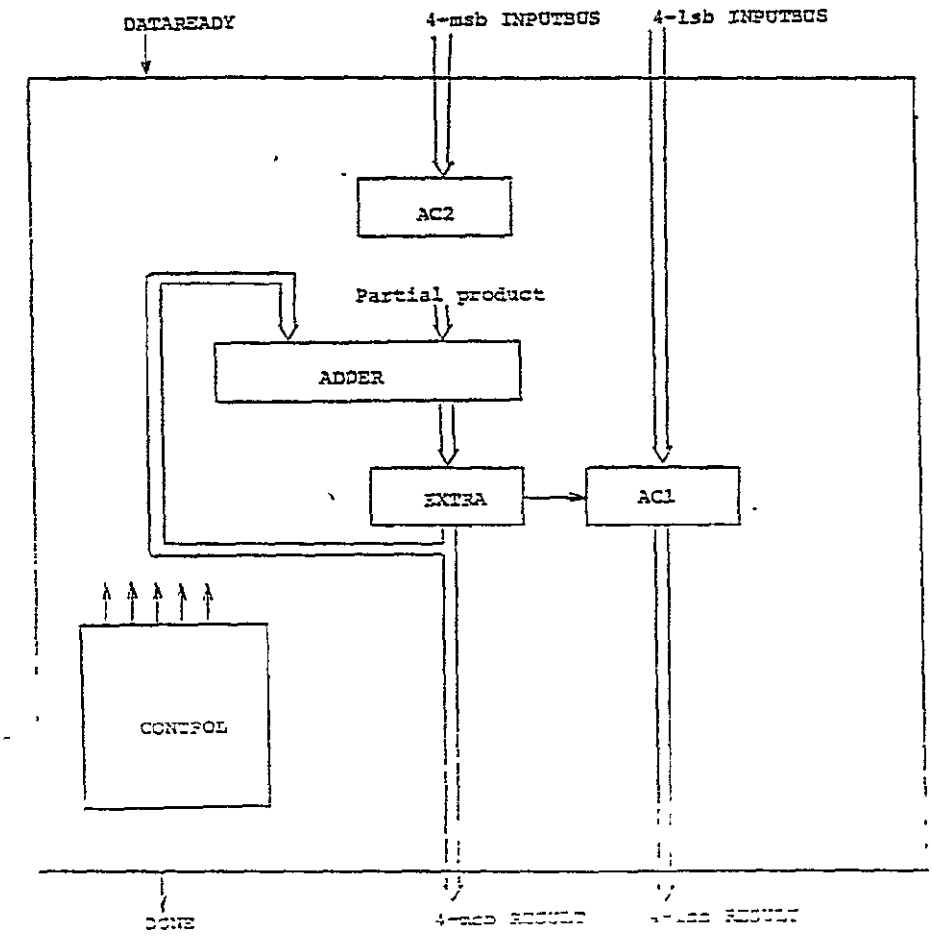


Figure 4-1: Multiplier Block Diagram

```
AHPLMODULE:MULTIPLIER.
    MEMORY:AC1[4];AC2[4];COUNT[2],EXTRA[4];BUSY.
    EXINPUTS:DATAREADY.
    EXBUSES:INPUTBUS[8].
    OUTPUTS:RESULT[8];DONE.
    CLUNITS:INC[2](COUNT);ADD[5](EXTRA;AC2).

1       AC1;AC2  <= INPUTBUS[0:3];INPUTBUS[4:7]; EXTRA <= 4$0;
        => (~ DATAREADY)/(1).
2       BUSY <= \1\;
        => (~AC1[3])/(4).
3       EXTRA <= ADD[1:4](EXTRA;AC2).
4       EXTRA;AC1 <= \0\;EXTRA;AC1[0:2]; COUNT <= INC(COUNT);
        => (~(&/COUNT))/(3).
5       RESULT = EXTRA;AC1; DONE = \1\; BUSY <= \0\;
        => (1).
    ENDSEQUENCE
    CONTROLRESET(1).
END.

CLOCKLIMIT 0030
EXLINES:
BUSEFFECT  0021
0400010
BUSEFFECT  0021
0410132
BUSEFFECT  0021
0000012
BUSEFFECT  0021
7777700
BUSEFFECT  0021
7777777
BUSEFFECT  0021
0000132
BUSEFFECT  0021
4000100
BUSEFFECT  0021
7777746
BUSEFFECT  0021
 >>>154
OUTPUTS:
DATAREADY
INPUTBUS
BUSY
DONE
EXTRA
AC1
AC2
COUNT
RESULT
```

Figure 4-2: HPSIM Program input File for the Multiplier

```
AHPL FUNCTION LEVEL SIMULATOR OUTPUT IS LISTED BELOW.

                DATAREADY
                  INPUTBUS
                  |  |          BUSY
                  |  |          | DONE
                  |  |          | | EXTRA
                  |  |          | | |     AC1
                  |  |          | | |     |     AC2
                  |  |          | | |     |     |     COUNT
                  |  |          | | |     |     |     |  RESULT
CLOCK #           |  |          | | |     |     |     |  |
00000001    0 00110111 0 0 0000 0000 0000 00 00000000
00000002    0 00110010 0 0 0000 0011 0111 00 00000000
00000003    0 00110010 0 0 0000 0011 0010 00 00000000
00000004    1 10110010 0 0 0000 0011 0010 00 00000000
00000005    0 00110010 0 0 0000 1011 0010 00 00000000
00000006    0 00110010 1 0 0000 1011 0010 00 00000000
00000007    0 00110010 1 0 0010 1011 0010 00 00000000
00000008    0 00110010 1 0 0001 0101 0010 01 00000000
00000009    0 10110010 1 0 0001 0101 0010 01 00000000
00000010    0 00110010 1 0 0011 0101 0010 01 00000000
00000011    0 00110010 1 0 0001 1010 0010 10 00000000
00000012    0 00110010 1 0 0001 1010 0010 10 00000000
00000013    0 00110010 1 0 0000 1101 0010 11 00000000
00000014    0 00110010 1 0 0000 1101 0010 11 00000000
00000015    0 10111111 1 0 0010 1101 0010 11 00000000
00000016    0 00010011 1 1 0001 0110 0010 00 00010110
00000017    0 10011000 0 0 0001 0110 0010 00 00000000
00000018    1 11011001 0 0 0000 1001 1000 00 00000000
00000019    0 00010011 0 0 0000 1101 1001 00 00000000
00000020    0 11011010 1 0 0000 1101 1001 00 00000000
00000021    0 00010000 1 0 1001 1101 1001 00 00000000
00000022    0 00000000 1 0 0100 1110 1001 01 00000000
00000023    0 00000000 1 0 0100 1110 1001 01 00000000
00000024    0 00000000 1 0 0010 0111 1001 10 00000000
00000025    0 00000000 1 0 0010 0111 1001 10 00000000
00000026    0 00000000 1 0 1011 0111 1001 10 00000000
00000027    0 00000000 1 0 0101 1011 1001 11 00000000
00000028    0 00000000 1 0 0101 1011 1001 11 00000000
00000029    0 00000000 1 0 1110 1011 1001 11 00000000
00000030    0 00000000 1 1 0111 0101 1001 00 01110101

EEEEE PROGRAM REACHED THE TIME LIMIT SPECIFIED BY USER.
```

Figure 4-3: HPSIM Output Listing for the Multiplier

sets the BUSY flip-flop to 1 and causes Step 3 to be skipped if AC1[3], which is the LSB of AC1 register, is zero. In Step 3 the addition of the partial products is accomplished. Step 4 right shifts the catenation of the EXTRA and AC1 registers, increments the counter and activates Step 2 if count has not reached (1,1). If the COUNT register contains (1,1), control will proceed to Step 5 where the catenation of the EXTRA and AC1 registers is placed on the 8 RESULT Lines, a one is placed on line DONE and the BUSY flip-flop is reset to zero. Step 5 also returns control to Step 1 waiting for another set of operands.

The communication section of the HPSIM program follows the AHPL description. Recall, that this section is line oriented. The first card, "CLOCKLIMIT", indicates how long the execution is to be carried on. This parameter is set at 30, so that the multiplication process will be carried out at least twice. The "EXLINES:" card is a heading for the subsection in which binary values are assigned to the external input lines. The next 18 cards are 9 sets of data for all 9 external lines. Each data set is headed by a "BUSEFFECT" card which indicates the number of clock periods for which the corresponding external line is to receive data. The nth data set corresponds to the nth declared external line. For example, since the first declared external line is DATAREADY, the first data set, which is 0400010 (octal), is assigned to this line. This line will receive binary data from this data set for the first 21 clock periods. The data for the 7th most significant line of INPUT-BUS is 7777746. This line also receives data from this set only for the first 21 clock periods. The next card in the communication section "OUTPUTS", is a heading for the print request subsection. The next five cards are the lines or registers for which print is requested. If a row in a register matrix is to be printed, the row number should follow the name of the register matrix on the same card. Row number zero is the default value which is assumed for all one dimen-

sional lines or registers. All print requests are unformatted and the printout format is set by the HPSIM system. The complete output sequence is shown in Figure 4-3. Notice that DATAREADY is 1 during the 4th clock period causing control to go to Step #2 for period 5, the two operand registers (AC1, AC2) containing 1011 and 0010. The multiplication starts at period 5 and ends at clock #16 where a 1 appears on the line DONE indicating that the 8 RESULT lines have the result of multiplication. Notice that a new partial product appears at clock periods 7, 10, and 15 to be shifted right by the next clock pulse. Only a right shift takes place after period 12. A second multiplica tion begins at 19 and is completed at Step 30 where DONE = 1 and the product appears on the 8 RESULT lines.

All but the communication section of Figure 4-2 is the input file to the HPCOM program.

For this example, the compiler printout in Figure 4-4 lists first the clock and data inputs for the data registers. Next are the gates realizing each module output, and last are the D inputs to each control D flip-flop. The remaining compiler output detailing inputs to each gate is given in Figure 4-5. They are in numerical order showing type on the left, and the inputs either as control signals or source gates on the right.

Using the information of Figure 4-4 complete schematic diagram such as the one shown in Figure 4-6 may be obtained. Figure 4-4 shows the registers listed first with the segment under consideration given at the left edge followed on the same line by the clock enable. This signal is gated with the clock should the register type used, not have an enable input provided. (The full schematic of Figure 4-6 shows these gates with a dot.) Further right are the individual data inputs for each bit of the register segment. They are listed by bit, on the left, and gate or signal implementation on the right.

COMPILER FOR AHPL HARDWARE ROUTINES. MAY 18, 1977.

REGISTER SEGMENTS	CLOCK ENABLE	REGISTER BIT	INPUT SOURC
[illegible]1[0:3]	G1		
		AC1[0]	G4
		AC1[1]	G7
		AC1[2]	G10
		AC1[3]	G13
AC2[0:3]	CS[1]		
		AC2[0]	G14
		AC2[1]	G15
		AC2[2]	G16
		AC2[3]	G17
BUSY[0]	G18		
		BUSY[0]	CS[2]
COUNT[0:1]	CS[4]		
		COUNT[0]	G19
		COUNT[1]	G20
EXTRA[0:3]	G21		
		EXTRA[0]	G22
		EXTRA[1]	G25
		EXTRA[2]	G28
		EXTRA[3]	G31

CONNECTION SEGMENTS	OUTPUT BIT	TERMINAL
RESULT[0:7]		
	RESULT[0]	G32
	RESULT[1]	G33
	RESULT[2]	G34
	RESULT[3]	G35
	RESULT[4]	G36
	RESULT[5]	G37
	RESULT[6]	G38
	RESULT[7]	G39
DONE[0]		
	DONE[0]	CS[5]

CONTROL BIT	D INPUT
[illegible]	G44
[illegible]	G45
[illegible]	G46
[illegible]	[illegible]
[illegible]	[illegible]

[illegible] INITIALIZATION ALL CS FLIP-FLOP RESETS ARE [illegible]NECTED TO MASTER RESET EXCEPT CS[illegible] WHOSE [illegible] TERMINAL IS CONNECTED TO MASTER RESET

Figure 4-4: HPCOM Output

GATE TYPE	GATE NO.	DATA INPUTS		
OR	G1	CSC1]	CSC4]	
AND	G2	CSC1]	INPUTBUSC0]	
AND	G3	CSC4]	EXTRAC3]	
OR	G4	G2	G3	
AND	G5	AC1C0]	CSC4]	
AND	G6	CSC1]	INPUTBUSC1]	
OR	G7	G5	G6	
AND	G8	AC1C1]	CSC4]	
AND	G9	CSC1]	INPUTBUSC2]	
OR	G10	G8	G9	
AND	G11	AC1C2]	CSC4]	
AND	G12	CSC1]	INPUTBUSC3]	
OR	G13	G11	G12	
AND	G14	CSC1]	INPUTBUSC4]	
AND	G15	CSC1]	INPUTBUSC5]	
AND	G16	CSC1]	INPUTBUSC6]	
AND	G17	CSC1]	INPUTBUSC7]	
OR	G18	CSC2]	CSC5]	
AND	G19	CSC4]	INCC0]	
AND	G20	CSC4]	INCC1]	
OR	G21	CSC1]	CSC3]	CSC4]
AND	G22	ADDC1]	CSC3]	
AND	G23	ADDC2]	CSC3]	
AND	G24	CSC4]	EXTRAC0]	
OR	G25	G23	G24	
AND	G26	ADDC3]	CSC3]	
AND	G27	CSC4]	EXTRAC1]	
OR	G28	G26	G27	
AND	G29	ADDC4]	CSC3]	
AND	G30	CSC4]	EXTRAC2]	
OR	G31	G29	G30	
AND	G32	CSC5]	EXTRAC0]	
AND	G33	CSC5]	EXTRAC1]	
AND	G34	CSC5]	EXTRAC2]	
AND	G35	CSC5]	EXTRAC3]	
AND	G36	AC1C0]	CSC5]	
AND	G37	AC1C1]	CSC5]	
AND	G38	AC1C2]	CSC5]	
AND	G39	AC1C3]	CSC5]	
AND	G40	COUNTC0]	COUNTC1]	
OR	G41	G40		
[illegible]	G42	CSC4]	G41	
[illegible]	G43	CSC1]	[illegible]	
[illegible]	G44	G42	G43	
[illegible]	G45	CSC4]	G44	
[illegible]	G46	AC1C0]	CSC0]	
[illegible]	G47	DATAREADYC0]		
[illegible]	G48	CSC1]	[illegible]	
[illegible]	[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	AC1C0]		
[illegible]	[illegible]	CSC0]	[illegible]	
[illegible]	[illegible]	CSC0]	[illegible]	

Figure 4-5: HPCOM Output

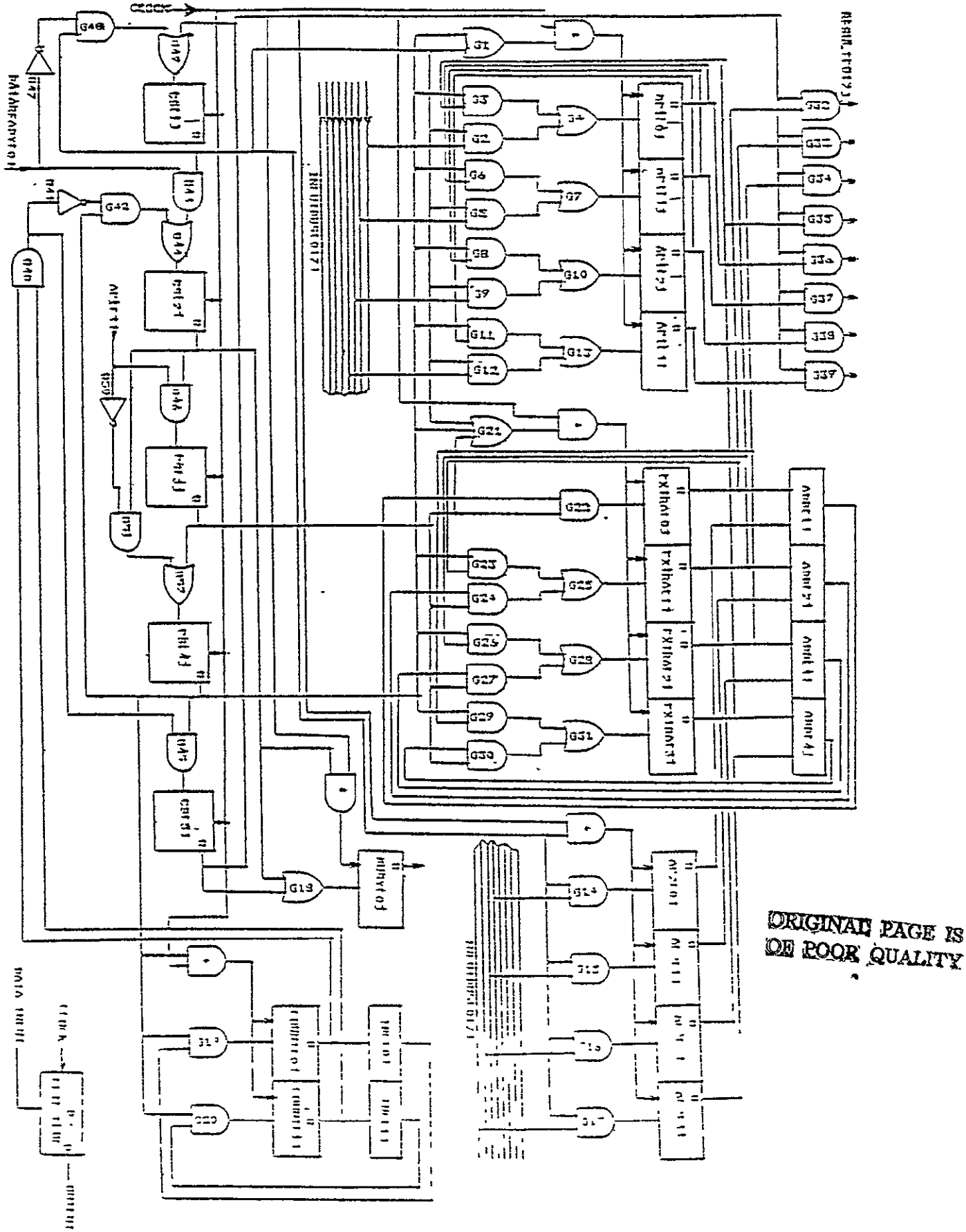


Figure 4-6: Multiplier

5. DIGITAL SYSTEM DESIGN LANGUAGE (DDL) [20]

DDL was introduced in 1968 by Duley and Dietmeyer [21, 22]. A translator and a simulator are written for a subset of this language in IFTRAN an extended version of FORTRAN [23, 24]. These programs are being implemented in FORTRAN on SEL 32 Computer System. DDL is a "block oriented" language. Each subsystem of a system appears as a block in the description of the system. The following sections introduce the language as required by the translator and simulator. DDL is suitable for the intermediate level of description between the extremely abstract level and the fabrication level.

All 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.

5.1 SYNTAX RULES

VARIABLES:
Variable name may contain 1 to 6 characters, the first of which must be alphabetic. The remaining characters must be letters or digits.

CONSTANTS:
Constants take the general form nRk. n is the number in base R (R=D for decimal, 0 for octal). k is the number of bits required for the representation k< or = 32. k is decimal.

5.2 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

<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.

REgister and LAtch DECLARATIONS

<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.

MEmory DECLARATION

<ME> M(X:Y).

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

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.

DElay DECLARATION

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

declares two delays P with 10 nanoseconds and Q with .5 micro-

second. The context in which the DElay element is referenced determines whether its input or output terminal is used.

BOolean DECLARATION

<BO> identifier = Boolean expression.

For example,

<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.

5.3 OPERATIONS

Figure 5-1 (a) shows the operations allowed and their hierarchy; Figure5-1 (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

FIGURE 5-1(a): OPERATORS

OPERATOR	SYMBOL	TYPICAL SYNTAX	COMMENTS
Extension	$	A$k	k copies of A
Concatenation	"	A"B	
Complementation	∧	∧A	Bit by bit complement
Selection	'	A'kDn	Selective complementation
Reduction	/	p/A	$A_1pA_2p...pA_n$ where pE[*,∧*, ∧+,∧@,@,+]
AND	*	A*B	Bit by Bit
NAND	∧*	A∧*B	Opertions
NOR	∧+	A∧+B	
XNOR	∧@	A∧@B	
XOR	@	A@B	
OR	+	A+B	

FIGURE 5-1(b): SPECIAL OPERATORS

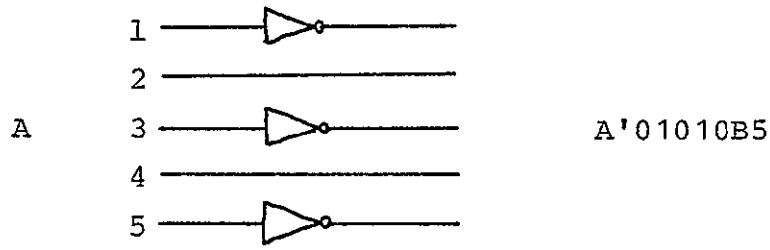
CONNECTION	=
TRANSFER	<-
GO TO	->

a "GOTO", usually used to show the next state.

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 0,1.

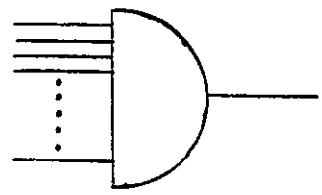
For example A' 01010B5 is equivalent to



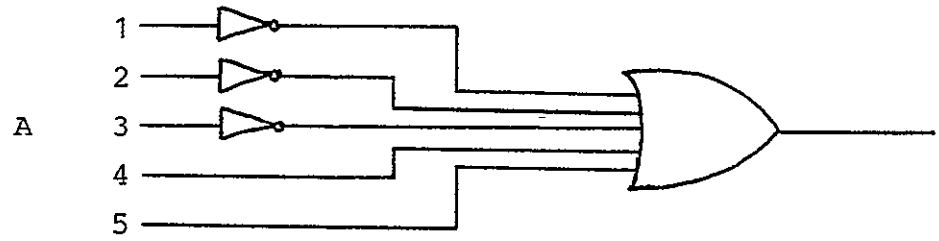
The operator preceding the reduction operator (/) determines the nature of the reduction on the right hand operator of /. Six types of reductions are possible.

*/A implies

A



+/A'3D5 implies



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 the 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..

5.4 IF - VALUE CLAUSE

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

B = !C #0 DO #1 D1 #2 D2.

implies that DO 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<-∧B #3D2 A<-∧C

describes a 4 way conditional transfer operation into A depending on the value of X.

5.5 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 C(2:10)"1 to be X. Then, any reference to X is expanded into C(2:10)"1.

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)"∧Z(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.

5.6 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 declaration, and (3) terminals declared 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,...X_k\$, id_4 (i_4)\$
X_1, X_2... 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 decalred 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$0,MBR,0,0$
?ADD?   ACC <-  ALU$ACC,MBR,0,1$
?SUB?   ACC <-  ALU$ACC,^MBR,1,1$
?KNT?   COUNT<- ALU(5:16) $0$4"COUNT,0,1,1$
?XOR?   ACC  <- ALU$ACC,MBR,0,0$
```

5.7 STATE DECLARATION

DDL views the operation sequencing (control) circuitry as a finite state machine. Each state 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.

5.8 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 declaratons. 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 SYstem declaration has syntax identical to the AUtomaton declaration. The system is identified in the head. Global coditions 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> SYSTEM1:
     <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: SHR$ACC"Q$,?DONE?-> S0; -> 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 decremnt 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.

5.9 ADVANCED FEATURES

The following features of DDL are not accepted by the present version of DDL software:

(a) Shift and count operations.
(b) SEGMENT declarations, which allow the Automaton to be broken up into several partitions.

5.10 TRANSLATOR (DDLTRN) [25,26]

DDLTRN translates a restricted DDL description into a set of tables suitable for simulation of the system. It is a six pass translator performing a syntax check, facility identification, syntax reduction, condition distribution, concatenation removal, operation gathering and disjoining the subfacilities. The FLAG statement can be used to control the printed output of the intermediate steps.

5.11 SIMULATOR (DDLSIM) [27]

DDLSIM uses the tables produced by DDLTRN to simulate the system. Multiple simulations are possible with the DDLSIM control statements. The following commands are available:

<CLock> declarations provide a means of specifying or changing the time period, pulse width and phase of the clock facilities. New clocks can be declared to control simulation input and output activities.

<DElay> declaration provides a means for specifying delay

time for delay facilities (old and new).

<INitialize> provides a means for initializing the output values of delays, registers, memories, element outputs, primary input signals, terminals and triggers with delays.

<REad> enables input data values for various facilities in three modes: triggered, periodic and specific time.

<LOad> provides a means for establishing the same input values repeatedly on specified facilities. The above three modes are possible.

<OUtput> specifies the printing of the values of various facilities at various instants during simulation. The values are printed in octal (default), binary, decimal or hexadecimal mode by setting the appropriate flag.

<DUmp> dumps the contents of specified memory locations at various instants during simulation.

<STop> stops the simulation at a specified simulation time.

<LIst> is used to assign a unique name to a list of facilities and can be used when the same set of facilities are used in various declarations of the simulation deck.

<SImulate> is used to separate different simulation runs in a simulation job.

<FLag> enables the selection of various options for simulation runs by setting or resetting the associated flags.

<TRigger> provides a means of declaring new facilities that can be used as triggering signals to control the simulation, without altering the DDL description.

5.12 DESIGN EXAMPLE [27]

A MULTIPLIER unit that calculates the product of two 8-bit numbers is described in DDL. A listing of the deck used for simulating the MULTIPLIER system along with the simulation report is given on the following pages. 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 receivng 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 multipl- ier bit is 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 period, pulse width and phase. 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.

```
$DDLTRN
<CO> DESIGN OF A 8-BIT MULTIPLIER.
<SY> MULTIPLIER:<TI>P.<RE> A(0:8), B(8), R(8),MCOUNT(3).
                <TE> START, BUS(8), DONE.
                <TE> SUM(8), COUT(8), CSUM(3), CCOUT(3).
                <ID> CIN = COUT(2:8)"OD1.
                <ID> CCIN = CCOUT(2:8)"1D1.
                <BO> COUT = R*A(1:8) + R*CIN+A(1:8)*CIN,
                SUM = R@A(1:8)@CIN,
                CCOUNT=MCOUNT*CCIN,CSUM=MCOUNT @ CCIN.
           <AU> MPY(2): P:
           <ST> S1(0): START: R<-BUS, NCOUNT <-0,->S2.
                S2(1): B<-BUS, A<-C, ->S3.
                S3(2): ?B(8)? A<-COUT(1)" SUM.,->S4.
                S4(3): A(1:8)" B<-A"B(1:7), A(0)<-0,
                       MCOUNT <-CSUM, ?*/MCOUNT?DONE=1,
                       ->S1;->S3.....
```

```
$DDLSIM                    |SIMULATION.
    <FL> 4,6.              |USE DECIMAL DATA AND PRINT OUTPUT
                           |IN BINARY FORMAT.
    <IN> START/1.          |GIVE A START COMMAND.
    <RE> S1/BUS/6,10.      |DATA VALUES FOR THE MULTIPLICAND.
    <RE> S2/BUS/5,13.      |DATA VALUES FOR THE MULTIPLIER.
    <TR> OUTTR/AP.         |DEFINE A TRIGGER TO CONTROL PRINT-
                           |ING ON
                           |TRAILING EDGE OF THE CLOCK.
    <OU> OUTTR/MPY,MCOUNT,A,B/,        |PRINT MULTIPLICAND,
                                       |MULTIPLIER
         S1/8,BUS,A(1:8), B.           |& FINAL PRODUCT.
    $EOJ
```

SIMULATION RESULTS

```
.DIGITAL DESIGN LANGUAGE SIMULATOR     VERSION - 07.030176    15:33:07    03/29/76      SIMULATION RUN   1

                M
                C
                O                                              A
          M     U                                              1
          P     N                                              1
TIME      Y     T      A          B          R        BUS      B        B

   0      00    000    00000000   00000000   00000000 0000000C 00C00000 00000000
   2      01    000    00000000   00000000
   4      10    000    00000000   00000101
   6      11    000    00000110   00000101
   8      10    001    00000011   00000010
  10      11    001    00000011   00000010
  12      10    010    00000001   10000001
  14      11    010    00000111   10000001
  16      10    011    00000011   11000000
  18      11    011    00000011   11000000
  20      10    100    00000001   11100000
  22      11    100    00000001   11100000
  24      10    101    00000000   11110000
  26      11    101    00000000   11110000
  28      10    110    00000000   01111000
  30      11    110    00000000   01111000
  32      10    111    00000000   00111100
  34      11    111    00000000   00111100
  36      00    000    00000000   00011110 00000110 00000101 00C00000 00011110
  38      01    000    00000000   00011110
  40      10    000    00000000   00001101
  42      11    000    00000110   00001101
  44      10    001    00000101   00000110
  46      11    001    00000101   00000110
  48      10    010    00000010   10000011
  50      11    010    00000110   10000011
  52      10    011    00000011   01100001
  54      11    011    00000011   01100001
  56      10    100    00000001   00100000
  58      11    100    00000001   00100000
  60      10    101    00000000   00010000
  62      11    101    00000000   00010000
  64      10    110    00000000   00001000
  66      11    110    00000000   00001000
  68      11    111    00000001   00000100
  70      11    111    00000001   00000100
  72      00    000    00000000   10000010 00001010 00001101 00C00000 10000010

.O OF FILE REACHED ON INPUT -
IMULATION TERMINATED AT TIME =        73

.O OF SIMULATIO
```

6. COMPARISON

All the available Hardware Description Languages satisfy the basic requirement of describing the hardware in a concise unambiguous and readable way. But, one language would be better than the other depending on the design environment. When designing systems of very low complexity, an HDL can be used only as a description media; this is because the HDL description usually is at a high-level and the designer can usually verify his design without resorting to the simulation. As the system complexity grows, it will be essential to verify the design at high levels using the HDL simulator, before proceeding to the detailed design.

After verifying the design through high-level simulation, the HDL description of the design can be used as an input to the programs, that generate the logic diagrams. This logic diagram data base serves as the starting point for the programs that decide the physical aspects like, placement, routing, partition etc. of the integrated circuit design.

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

1) Activity
2) Level of Description
3) Software availability and portability
4) Ease of logic generation, and
5) Modularity.

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 HDLs proposed do not have a translator and a simulator that is up-to-date and fairly versatile, though the language itself is versatile. All the four HDLs described, have been implemented at several locations and

there is a considerable amount of interest in making these HDLs more versatile.

LEVEL OF DESCRIPTION:

ISP is suitable for comparing systems at the instruction level. CDL is suitable at the register transfer level and does not have adequate time and delay facilities. AHPL and DDL could be used very well in the circuit design work, since they are capable of description at and slightly below the register transfer level.

SOFTWARE AVAILABILITY and PORTABILITY:

The ISP software is developed in BLISS on PDP-10 Computer system and is not portable. CDL software has many implementations. Most of the software is in FORTRAN, with some essential routines in assembly language of the computer system it is implemented (IBM 360/370, UNIVAC 1110[6], CDC 6000 [5]). AHPL hardware compiler is written in SNOBOL and the simulator in FORTRAN on DEC-10 and CDC 6400 systems. A few changes related to machine word length are required to make these operative on the other machines. DDL software is written in IFTRAN (Structural FORTRAN). An IFTRAN preprocessor is available. These programs also need some changes related to machine word length, to be operative on other machines.

EASE OF LOGIC GENERATION:

ISP is not suited to generating logic diagrams. CDL being a pure register transfer level language, does not tend itself very well to the logic generation. The AHPL hardware compiler provides a wiring list of the system consisting of gates and memory elements. DDL translator provides the Boolean functions for the system as an intermediate step in the translation process. These could be used to generate the logic diagrams.

MODULARITY:

High level modular description is possible with ISP. CDL is a one-level description language. The subroutine features of AHPL could be used to describe separate modules. DDL block structure is more closer to the hardware modularity.

From the above discussion, it is seen that AHPL and DDL are suitable for an integrated circuit design environment. However, the block structure of DDL, the right-to-left conventions of AHPL due to its origination from APL and the portability of DDL's FORTRAN software, makes DDL more suitable.

7. CONCLUSIONS

The characteristics of the four prominent HDLs are summarized. DDL was found to be most suitable among these four languages for an integrated circuit design environment.

Since there are so many languages proposed [8], it is very hard to perform a critical evaluation of their capabilities. Such a critical evaluation of the language capabilities might not be of much use, since the implementation issues more or less influence the selection of the language. The evaluation reported here caters more to the implementation aspects of the selected language.

DDL translator and simulator are currently being implemented on the SEL-32 computer system of the electronics and controls laboratory of the Marshall Space Flight Center. The future work includes the development of procedures to generate logic diagrams from a DDL description and integration of these procedures into the current automatic design system.

Three other languages that are heirarchic in nature, use a multi-level design philosophy allowing the designer to specify his design at any level of detail. They are: Language for Computer Design [28], an Hierarchical Language for the structural description of Digital Systems [29] and a language for Automated Logic and System Design [30-32]. The RT-CAD research group at Carnegie-Mellon University is using ISP in their register-transfer level design automation [33].

The languages described in the literature seem to be used mostly in the academic environment. Industrial design groups usually make use of internal, proprietary languages. There seems to be a growing interest in HDLs. Recognizing the need for common notations and a standard language, a working group consisting of professionals in this area [34] has been set-up, which is trying to develop a consensus language.

REFERENCES

[1] Chu, Y.,"An ALGOL-Like Computer Design Language," Communications of ACM, Oct. 1965, pp. 607-615.

[2] Chu, Y., "Structure of CDL Programs," Technical Note 74-58, Department of Computer Science, University of Maryland, May 1974.

[3] Chu, Y., "A Higher-Order Language for Describing Microprogrammed Computers," Technical Report 68-78, Computer Science Center, University of Maryland, September 1968.

[4] Mesztenyi, C.K., "Translator and Simulator for the Computer Design and Simulation Program, Version 1, "Technical Report 67-48, Computer Science Center, University of Maryland, June 1968.

[5] Stine L.R. and Mowle F.J., "A Position Paper on Extensions to the CDL," pp. 103-114, Proc. International Symposium on CHDLs and Their Applications, New York, September 1975.

[6] Bara J. and Born R. "A CDL Compiler for Designing and Simulating Digital Systems at the Register Transfer Level", pp. 96-102, Proc. of 1975. International Symposium on CHDLs and Their Applications, New York, September 1975.

[7] Cwik T.T., "Multiprocessing Simulation of the Intel 8080 and the PDP-8 using CDL," Masters Thesis, Auburn University, March 76.

[8] Shiva, S.G., "Hardware Design Languages - A Bibliography", Semi annual Status Report, Alabama A & M Universty, March 78.

[9] Chu, Y.,"Introducing CDL", Computer, pp. 31-33, December 1974.

[10] Chu, Y., Computer Organization and Microprogramming, Prentice-Hall, Englewood-Cliffs, New York, 1972.

[11] Bell, G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971.

[12] Barbacci, M.R., "The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator". Technical Report, Department of Computer Science, Carnegie-Mellow University, August 1976.

[13] Barbacci, M.R., Siewiorek, D.P., Gordon, R., Howbrigg, R., and Zuckerman, S.: "An Architecture Research Facil-

ity: ISP Descriptions, Simulation, Data Collection". Proceedings of the AFIPS, Vol. 46, NCC-77, pp. 161-173.

[14] Barbacci, M.R., Barnes, G.E., Cattell, R.G., and Siewiorck, D.P., "The ISPS Computer Description Language", Department of Computer Science and Electrical Engineering Report, Carnegie-Mellon University, August 1977.

[15] Barbacci, M.R. and Nagel, A.W., "An ISPS Simulator", Department of Computer Science and Electrical Engineering Report, Carnegie-Mellon University, November 1977.

[16] Hill, F.J. and Peterson, G.R., "Digital Systems: Hardware Organization and Design, Wiley, New York, 1978, Second Edition.

[17] Hill, F.J. and Peterson, G.R., Introduction to Switching Theory and Logic Design, Wiley, New York, 1974, Secod Edition.

[18] Swanson, R.E., Navabi, Z., Hill, F.J., "An AHPL Compiler/Simulator System," Sixth Texas Conference on Computing Systems, pp. 1-10, November 1977.

[19] Swanson, R.E., Navabi, Z., Hill, F.J., "User Manual for AHPL Simulator/Compiler", Engineering Experiment Station, The University of Arizona.

[20] Breuer, M.A., Digital System Design Automation: Languages, Simulation and Data Base, Woodland Hills, CA., Computer Sciences Press, 1975.

[21] Duley, J.R. and Dietmeyer, D.L., "A Digital System Design Language (DDL)," IEEE Transactions on Computers, Vol. C-17, September 1968. pp. 850-861.

[22] Duley, J.R., "DDL-A Digital System Design Language," PhD dessertation, University of Wisconsin, Madison, 1967.

[23] Arndt, R.L. and Dietmeyer, D.L., "DDLSIM - A Digital Design Language Simulator, "Proceedings of NEC, Vol. 26, December 1970, pp. 116-118.

[24] Soares, L.E.R., "An Implementation of Digital Design Language," MS Thesis, University of Wisconsin, Madison, 1970.

[25] Dietmeyer, D.L., "DDLTRN-Users Manual", Department of Electrical and Computer Engineering, University of Wisconsin- Madison.

[26] Dietmeyer, D.L., "Translation of DDL descriptions of Digital Systems," Report No. ECE-77-13, University of

Wisconsin-Madison, September 1977.

[27] Dietmeyer, D.L., "DDLSIM - Users Manual," Department of Electrical and Computer Engineering, University of Wisconsin-Madison.

[28] Evangelistic, C.J., Goertzel, G., Ofek, H., "Designing with LCD: Language for Computer Design," Proc. 14th Design Automation Conference, June 1977, pp. 369-376, New Orleans.

[29] Vancleemput, W.M., "An Hierarchical Language for the Structural Description of Digital Systems," Proc. 14th design auto. conf; June 1977, pp. 377-385, New Orleans.

[30] Baray, M.B. and Su, S.Y.H., "A Digital System Modeling and Design Language," Proc. of the 8th Annual Design Automation Workshop, 1971. pp. 1-22.

[31] Su, S.Y.H., "A language for Automated Logic & System Design," presented at the Workshop on Computer Descriptive Languages, Rutgers University, New Brunswick, New Jersey, September 6-7, 1973.

[32] Su, S.Y.H., Baray, M.B., and Carberry, R.L., "A System Modeling Language Translator," Proc. of the 8th Annual Design Automation Workshop, 1971, pp. 35-49.

[33] Hafer, L.J. and Parker, A.C., "Register-Transfer level Digital Design Automation: The Allocation Process", 15th Design Automation Conference Proceedings, pp. 213-219, Las Vegas, Nevada, June 1978.

[34] Su, S.Y.H., "HDL Applications: An Introduction and Prognosis," Computer, June 1977, pp. 10-13.

GENERAL REFERENCES

[35] Proceedings of the International Symposium on CHDLs and their Applications, New York, September 1975.

[36] Computer, Special Issue on CHDLs: December 1974.

[37] Computer, HDL Applications, June 1977.