DIGITEST II: An Intergrated Structural and Behavioral Language

Franz J. Rammig
Universität Dortmund
West Germany

1. Abstract

This paper presents the experiment of an intergrated structural and behavioral language. The language is based on three "ancestors": DIGITEST to describe the structure, PL/1 to describe the behavior and Petri nets to describe complex control structures. Instead of directly using PL/1 as "ancestor" we decided to use XPL, for it is relatively easy to modify the XPL-compiler. Similary we decided to use the "Control Graph" as described by Rose /8/ and used by project LOGOS of the CWRU instead of pure Petri nets. The language is so defined that a program can be placed arbitrarily anywhere between the two extremes "pure DIGITEST-program" (pure structural description) and "pure PL/1-program" (pure behavioral description).

After a very short introduction to the language-philosophy, the following three aspects will be discussed:"Data-types and data-structures","Petri net implementation" and "Constructs for the behavioral description". A description of the structural-language-aspect of DIGITEST II is given in /7/.

2. Philosophy of the language

Complaints by potential users (industry) about existing CHDL's as well as our own ideas motivated the decision to design an additional CHDL. Breuer and Hayes /2/ also give two important reasons for this decision:

First, there are deficiencies in the description of the structure by exisitng CHDL's.

The reason for this is that existing CHDL's are aimed at accurate behavioral rather than structural description (the structure perhaps just to be derived from the behavioral description). This not only limits the range of description unnecessarily but also makes it impossible for the user to force a definite structure upon some parts of a described hardware.

To avoid these deficiencies, DIGITEST II consists of language-constructs for structural as well as for behavioral description. A user can place his description arbitrarily anywhere between pure behavioral description and structural description. The language-constructs for the structural description are identical to those of DIGITEST (any valid DIGITEST-program is a valid DIGITEST II program !), while the language-constructs for the behavioral language from a PL/1 (XPL) - dialect.

Second there are deficiencies in the description of asynchronism and parallelism by existing CHDL's.

To my knowlage all CHDL's allow the description of parallel processes, some with the restriction on clocked systems. In some cases this is done by intermixing the control-part and the data-part, making it very difficult to check algorithmicly for attributes like "deadlock-free","safe","conflict-free" and "deterministic". For the same reason the division between data-part and control-part may be ambiguous.

To avoid these deficiencies in DIGITEST II a modified Petri net model based on the model as described by Rose /8/ is implemented. The implemented model uses statement-labels as places. From this results a certain similarity with known CHDL's. Obviously, at the level of structural description of hardware only pure parallelism exist. On the other hand, at this level we are faced with the problem of describing the absolute time-behavior of digital circuits. For this purpose DIGITEST has a lot of language-constructs which are all valid in DIGITEST II as well. DIGITEST II therefore allows a much more precise description of the time-behavior such as propagation delay and intertial delay than other CHDL's.

In addition to the two reasons given by Breuer and Hayes for the necessity of a new CHDL, I want to mention two additional reasons:

Third, existing CHDL's operate on overly simplified data structures.

(The term "register" is out of place in a behavioral language in any case.) Therefore in DIGITEST II some of the PL/1-data-structures, in particular "structures" in the PL/1 sense are implemented. Besides the data-type "bit-string" there are various representations of numerical data to give a concrete sense to arithmetic operators which may be used by a programmer.

Forthly, existing CHDL's have been defined independent of other languages or widely differing from their "ancestors".

However, as nobody likes to learn an additional language, this is an important impediment to a more common use of CHDL's. As a consequence of this, the DIGITEST II-constructs for the behavioral description are oriented to PL/1 as nearly as possible. We had various reasons for choosing PL/1:

- PL/1 offers sufficient constructs to describe complex data-structures.
- PL/1 offers sufficient constructs for bit-string-manipulation.
- PL/1 offers constructs for the description of asynchronism and concurrency.
- PL/1 is a block-oriented language in the ALGOL-sense.

Besides restrictions, the main modifications of the language are:

- The basic point of view is that all statements are processed in parallel.
- Every procedure is a co-routine if not otherwise specified.
- All kinds of sequential processing are expressed with the help of "On-conditions" on labels. (The PL/1 "wait"-"event"-concept is not used in DIGITEST II!)

Below some aspects of DIGITEST II will be described:

a) The data-types and the data-strucutres valid in DIGITEST II and their declaration.
b) The description of control-structures, in particular asynchronous and concurrent ones.
c) The language-constructs for the behavioral description.
d) The communication between the structural description and the behavioral description.

As the language-concepts of DIGITEST II for the structural description are identical to those of DIGITEST /7/ they are not described within this paper.

3) Data-types valid in DIGITEST II and their declaration.

3.1) Introduction

The basic data-type of a CHDL is of course the bit-string. To give a concrete meaning to arithmetic operators which may be used by the programmer in DIGITEST II we also have the data-type "FIXED" and the data-type "FLOAT", each in various representations. Besides the data-structure "array", DIGITEST II also has the data-structure "structure" as known from PL/1. To allow a partial identification of different variables (overlay) the "DEFINED"-attribute as known from PL/1 can be used very freely.

3.2) Data-types

There are three basic data-types: FIXED-data, FLOAT-data and bit-strings. We have four representation-types of FIXED-data:SIGN_ VALUE (or SV), UNSIGNED (or US), TWO'S_COMPLEMENT (or TC) and ONE'S_COMPLEMENT (or OC). In addition the length of the data word in bits (the Variable is also a bit-string of this length) and the position of the sign or hig-order bit can be specified. Default is 32 for the word-length, TWO'S_COMPLEMENT as representation type and the high-order bit to be the left-most bit of the word. (Default-XPL convention!)

FLOAT-data can be specified in a similar way. Bit-strings are specified as in PL/1 with the difference that normally the bits are counted from right to left. If the length of a bit-string is specified by a negative integer the bit count is from left to right. Examples of types are:

FIXED (32,TC,31)

This denotes a 32-bit two's-complement representation with the high-order bit at the leftmost position.

BIT (927)

denotes a bit-string of length 927, the leftmost bit is bit 926, the rightmost is bit 0.

BIT(-17)

denotes a bit-string of length 17, the leftmost bit is bit 0, the rightmost is bit 16.

3.3) Declaration

Every undeclared variable that is not used as a label is interpreted as a bit-string of length 1. (DIGITEST II-default = DIGITEST-convention!) The DIGITEST II declare statement is very similar to the declare statement as known from PL/1 and XPL. The formal definition of the syntax (see appendix 0) has been derived by slightly modifying the syntax definition of the XPL declare statement. The modifications have been made to add the "structure"-declaration and the attributes "DEFINED","STATIC", and "EXTERNAL" as known from PL/1, and to implement the declaration of the data-types as presented above. The semantics of the declarations being very similar to the semantics of declarations in PL/1, we will say only very few words about it.

a) The scope of variables is defined as it is defined in ALGOL or PL/1. (Variables declared within a block are local to this block and global to all blocks within this block.) By the use of the attributes INTERNAL and EXTERNAL, which have the same meaning as in PL/1 these scope-conventions can be overwritten. INTERNAL is default.
b) Variables with the attribute STATIC can alter their value only if an assignment-statement with this variable standing on the left side is explicitly processed, while this is not true for variables with the attribute DYNAMIC. Compared with conventional CHDL's the meaning of the attribute STATIC is similar to a register-declaration, while DYNAMIC is related to terminals. DYNAMIC is default.
c) There is made a distinction between "type declaration" (delaration of unstructured data),"structure declaration" (declaration of structured data) and "literal declaration" (as in XPL, similar to PL/1-preprocessor).
d) Every declared variable can be initialized with the aid of the INITIAL-attribute and an "initial-list" as known from PL/1.
e) With the aid of the DEFINED-attribute, variables or parts of variables can be identified with other variables or parts of other variables without limitations implied by data-type or physical location.
   Constants are written according to XPL-conventions instead of PL/1-notation. In DIGITEST II notations for the PL/1 bit-string '111111111111 1111'B are- for example:"1111111111111111" (bit notation) or
   "(4) FFFF" (hexadecimal notation),
   "Inverse"notation is used for multivalued algebra. E.g."(-2)01" means "up" A short example will illustrate the use of DIGITEST II declarations:

```
DCL(A,B,C) BIT(-16)"1","(4)FFFF",("(1)1(4)0")"1");
DECLARE NUMER FIXED (4,UN) DEFINED (B Position(2));
DCL 1 INPUT (5),
      2 (START1,START2) BIT(1) INIT("1"),
      2 ADR(3),
        3 LEFTADR FIXED(6,TC) DEFINED (A POSI-
          TION(4)),
        3 RIGHTADR FIXED(6,TC) DEFINED (POSI-
          TION (0));
DCL SUBADR BIT(5) DEFINED(INPUT(1).ADR.LEFTADR
                                          (0));
 DECLARE ACCUMULATOR BIT (32) STATIC EXTERNAL;
```

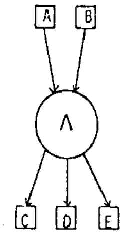
Everybody who is familiar with PL/1, and I think that nearly every computer scientist today is familiar with PL/1, can understand these declarations without any difficulty. A reader familiar with XPL will see immediately that the three variables declared in the first statement are all initialized with 16 ones. The second declaration and especially the third one illustrates that an overlay of variables is allowed without any limitations. This overlaying capability and the possibility of declaring structured data are very valuable for documentation as every variable can be given a name that is related to its use instead of its location. It is even more valuable in supporting a top-down design as the designer who wants to specify the problems to be processed by the hardware under construction can specify the data-structure which he needs instead of being forced to accept a given data-structure.

4. Description of control structures in DIGITEST II

Description of concurrent asynchronous processes by programming languages is a well-knwon problem. Therefore there are numerous implemented solutions as in PL/1, Burroughs Extended ALGOL, SIMULA 67 and various CHDL's. It should be noted that the potential parallelism of normaly sequential processes in common algorithmic languages should be replaced by potential sequential flow of normally parallel processes in HDL's. The question whether a programmer can actually think in parallel and if not, whether there should be algorithms which would generate parallel processes out of a sequential description, will not be discussed within this paper. Undoubtedly, the programmer must be able to describe arbitary control structures and an easy processing of this desciption by algorithm to prove the "correctness" and to do optimizations must be possible. A description method for the control structure of asynchonous concurrent processes is given by the Petri net model /1,5,6/. For problems of relevant size pure Petri nets tend to be unwieldy. Therefore it has been proposed that more complex modules be constructed out of Petri nets. In this case the description of control structures is based on these more complex modules. For our pupose the method described by Rose /8,9/ (LOGOS Control Graph) especially seems to be useful. The LOGOS Control Graph uses very few modules, at the same time offering the description of data-dependent decisions and block-structures. In this paper only the implementation of the LOGOS Control Graph in DIGITEST II and not the LOGOS philosophy will be described. Any kind of Petri net consists of a set of places which are capable of containing tokens, a set of transitions which in accordance with a certain firing-rule are able to withdraw tokens from some places and to put tokens into some places, and a set of directed edges connecting certain places with certain transitions and vice versa. A transition may be related to one or more activities which are processed if and only if the transition fires in accordance with its firing rule.

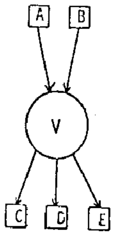
In DIGITEST II, labels are used as places. One can imagine that a labelled statement puts a token into its label(s) after it has been processed. On the other hand, the processability of a statement can be controlled by "On-conditions" on labels. Every statement is processed if and only if the last "On-conditions" previous to this statement has become true. At this time, in accordance with the firing rule (="On-condition"), "tokens" are withdrawn from certain labels used within the "On-condition" and placed in the labels of the statement.

We have just briefly summarized the principles of the DIGITEST II control mechanism. The basic assertion of DIGITEST II that all statements are processed in parallel is modified to the assertion that all statements between two "On-conditions" are processed in parallel. Consequently, one can visualize all labels within this program-part as beign placed in front of the related "On-condition". A short introduction to the LOGOS operators with their equivalent DIGITEST II notation follows:



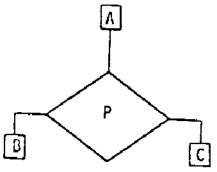
If there is a token in A and B and there are no tokens in C,D,E, the transition fires withdrawing the tokens from A, B and placing a token in C,D, E. (The number of places is arbitrary !)

Notation in DIGITEST II: C:D:E:ON (&(A,B)):<statementlist>



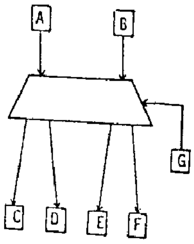
If there is at least one token in A or in B and there are no tokens in C,D,E the transition fires withdrawing the token from A or B (or from B only if there is a token in A and B and placing a token in C,D,E. (The number of places is arbitrary !)

Notation in DIGITEST II:C:D:E:ON(|(A,B)):<statementlist>



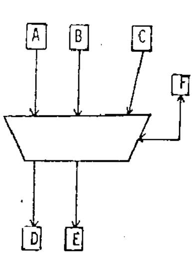
If there is a token in A and there are no tokens in B,C, the transition fires withdrawing the token from A and, depending on the value of the predicate P, placing a token into B or C.

Notation in DIGITEST II: ON(A):IF(P) THEN B: <statementlist> ELSE C: <statenentlist>



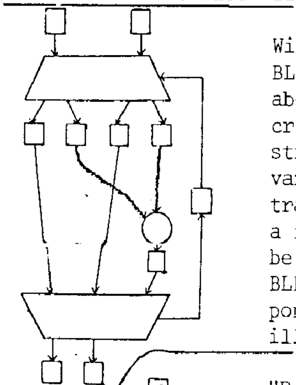
If there is a token in A or B and in G and there are no tokens in C,D,E,F, the transition fires. If there is a token in A, it is withdrawn and a token is placed in C and D, while if there is a token in B, it is withdrawn and a token is placed in E and F. In every case a token is withdrawn from G. If there are tokens in A and B, A has priority.

Notation in DIGITEST II: ON CASE (A:(C,D),B:(E,F))

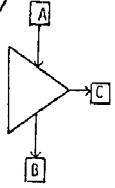


If there are tokens in A and C or B and C and there are no tokens in D and E, the transition fires. If there are tokens in A and C they are withdrawn and tokens are placed in D and F, while if there are tokens in B and C they are withdrawn and tokens are placed in E and F.

Notation in DIGITEST II: ON CASE (D:(A,C),E:(B,C):

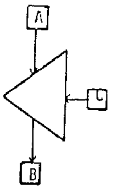


With the aid of the operators BLKHD and BLKEND as described above it is possible to describe the control of block-structures. Blocks may have various entries and their reentrance may be controlled by a feedback-loop. There must be a proper combination of a BLKHD operator and its corresponding BLKEND operator as illustrated in the margin.



"Procedure call": If there is a token in A and there are no tokens in C and B, the transition fires, withdrawing the token from A and placing a token in B and C.

Notation in DIGITEST II: B:ON(A):CALL C;



"Return from procedure": If there is a token in A and C and no token in B, the transition fires, withdrawing a token from A and C and placing a token in B.

Notation in DIGITEST II: B:ON(←(C),A):<statement-list>

It should be noted that in DIGITEST II we always have co-routines instead of sub-routines. As we have seen, by a simple expansion of the usability of labels, it is possible in DIGITEST II to describe easily control structures of arbitrary complexity. Furthermore, for a PL/1-programmer this notation is easy to understand. The usual language-constructs like DO;...END;,DO CASE...END;,DO WHILE...END;, DO I=...END; are well suited to this method.(Note that GO TO A; is equivalent to A:;.)

In addition there is a "DO SEQUENTIAL" to allow short-hand notation.

5) DIGITEST II language constructs for the behavioral description

With the exception of the assignment statement the language constructs for the behavioral description have been discussed above. The assignment statement is nearly the same as the assignment statement in XPL (and in PL/1), with the following differncies (relative to XPL):

a) Concatenation is allowed on the left side of an assignment.
b) SUBSTR is allowed on the left side of an assignment (as in PL/1).
c) The logic operators¬&(NAND),¬|(NOR) (EXOR) and ¬ (EXNOR) have been added.
d) Logic operators may be used not only as dyadic operators, but also as monadic operators in the sense of reduction.

From the above it follows, that the following example is a valid DIGITEST II-assignment statement:

```
A||B,C||SUBSTR(F(I,J,K),2*I MOD (J+K),K)=((A&(B¬|C))
¬&D)¬|(E@F);
```

Every operator is processed on bit-strings of the length of the largest operand or goal. Alignment is always to the right. Logic operators are processed bit by bit, arithmetic ones over the whole data word (bit-string). Constants are writtten in the XPL - notation.

6) The communication between structural and behavioral description in DIGITEST II

DIGITEST-Description-Statements (DDS) may be intermixed with DIGITEST II statements for the behavioral description in any way desired. Communication takes place through the usage of identical names of variables. Since DIGITEST works only on one-bit data, bit strings are interpreted by a DDS as bit-arrays. One must remember that a DDS has to be looked at as being processed constantly with no interference by any control structure. On the other hand, a DDS can influence the control structure:
If a DDS is labelled, with a delay as described in this delay description, it places a token in its label after the "On-condition" related to the program-part containing the DDS has become true.
It is up to the programmer to decide where between pure behavioral description and pure structural description he wants to place his program. In particular, this method enables us to influence arbitrarily the translation to hardware within an interactive hardware-generating-system.
The soft transition from pure behavioral description to pure structural description is illustrated in appendix 1. In this example we have a sequential decimal adder for 16 digits. It is described three times: First, by a pure behavioral description, second by a mixed behavioral and structural description and finally by a pure structural one.

Referencies

/1/ Agerwala, Tilik:
Comments on Capabilities, Limitations and "Correctness" of Petri nets
Proc. of the 1st Annual Symposium on Computer Architecture 1973 Gainesville/Florida

/2/ Chu, Yaohan et al:
Why do we need Computer Hardware Description Languages
IEEE Computer 12/1974

/3/ Crocket, E. David et al:
Computer aided system design
FJCC 197o pp 287

/4/ McKeemann, W.M. et al:
A Compiler Generator Prentice - Hall 197o

/5/ Patil, Suhas S.:
Macro - modular Circuit Design
MIT, MAC Computation Strucutres Group Memo Nr. 4o/1969

/6/ Petri, C.A.:
Concepts of Net Theory
Proc. of Symp.& Summer School about Mathem. found.of Camp.Science. Hig Tatras 1973

/7/ Rammig Franz J.:
DIGITEST: A Structural Language Based on Algebraic Modelsof the Logic Topology and the Time Behavior of Digital Circuits
Proceedings of the 2^{nd} Annual Workshop on CHDL's
Darmstadt 1974

/8/ Rose, Charles William:
Ph.D. Thesis CWRU 197o

/9/ Rose, Charles William:
LOGOS and the software engineer
FJCC 1972

Appendix O): DIGITEST II GRAMMAR

```
<PROGRAM>::=<PROCEDURE DEFINITION><STATEMENT LIST>
            <PROCEDURE ENDING> EOF
<STATEMENT LIST>:: = <STATEMENT>
                    |<STATEMENT LIST><STATEMENT>
<STATEMENT> ::= <BASIC STATEMENT>
               |<IF STATEMENT>
               |<DDS>
<BASIC STATEMENT>::= <ASSIGNMENT>;
                    |<GROUP>;
                    |<RETURN STATEMENT>;
                    |<CALL STATEMENT>;
                    |<GO TO STATEMENT>;
                    |;
                    |<LABEL DEFINITION><BASIC STATE-
                                             MENT>
                    |<PROCEDURE DEFINITION>
<IF STATEMENT> :: = <IF CLAUSE><STATEMENT>
                   |<IF CLAUSE><TRUE PART><STATEMENT>
                   |<LABEL DEFINITION><IF STATEMENT>
<IF CLAUSE >::= IF <EXPRESSION> THEN
<TRUE PART> ::= <BASIC STATEMENT> ELSE
<GROUP >::= <GROUP HEAD><ENDING>
<GROUP HEAD>::= <DO>;
               |<DO><STEP DEFINITION> ;
               |<DO><WHILE CLAUSE>;
               |<DO><CASE SELECTOR>;
               |<GROUP HEAD><STATEMENT>
<DO> ::=  DO
        | DO SEQUENTIAL
<STEP DEFINITION> :: = <VARIABLE><REPLACE><EXPRESS-
                       ION><ITERATION CONTROL>
<ITERATION CONTROL> ::= TO <EXPRESSION>
                      | TO <EXPRESSION> BY <EXPRESSION>
<WHILE CLAUSE> ::= WHILE <EXPRESSION>
<CASE SELECTOR> ::= CASE <EXPRESSION>
<ENDING >::  END
<PROCEDURE DEFINITION> ::= <FULL PROCEDURE HEAD>
                    <STATEMENT LIST><PROCEDURE
                                       ENDING>
<FULL PROCEDURE HEAD> ::= <PROCEDURE HEAD>
                         |<PROCEDURE HEAD><FEEDBACK
                                         CONTROL>
<PROCEDURE HEAD> ::= <PROCEDURE NAME> ;
                   | <PROCEDURE NAME><TYPE>;
                   | <PROCEDURE NAME><PARAMETER
                                        LIST>;
                   | <PROCEDURE NAME><PARAMETER
                                LIST><TYPE> ;
<PROCEDURE NAME> ::= <PROCEDURE ON CONDITION>
                                 PROCEDURE
<PROCEDURE ON CONDITION>::=<CASE START><IDENTIFIER>:
                          <IDENTIFIER SPECIFICATION>):
<CASE START>::=  ON CASE (
              | <CASE HEAD>,
<PARAMETER LIST> ::= <PARAMETER HEAD><IDENTIFIER>)
<PARAMETER HEAD> ::= (
                    |<PARAMETER HEAD><IDENTIFIER>,
<PROCEDURE ENDING> ::= <PROCEDURE ON CONDITION>
                          <P-END TAIL>
<P-END TAIL>::= <ENDING>
              | <ENDING><FEEDBACK CONTROL>
<FEEDBACK CONTROL>::=, ON (<IDENTIFIER>)
<LABEL DEFINITION>::= <IDENTIFIER>:
                    | <ON CONDITION>
                    | <LABEL DEFINITION><IDENTIFIER>:
                    | <LABEL DEFINITION><ON CONDI
                                            TION>
<ON CONDITION>::= <N(><IDENTIFIER SPECIFICATION>):
                 |<P-RETURN ON CONDITION>
<P-RETURN ON CONDITION>::= ON (←<IDENTIFIER>,
                                <IDENTIFIER>)
<N(> ::= ON (|<N(> &|<N(>|
<RETURN STATEMENT>::= RETURN
                    | RETURN <EXPRESSION>
<CALL STATEMENT> ::=  CALL <VARIABLE>
<GO TO STATEMENT> ::= <GO TO><IDENTIFIER>
<GO TO> ::= GO TO | GOTO
<DECLARATION STATEMENT> ::= DECLARE <DECLARATION
                                     ELEMENT>
                  | DCL <DECLARATION ELEMENT>
                  |<DECLARATION STATEMENT><DECLARATION
                                           ELEMENT>
<DECLARATION ELEMENT> ::= <FULL TYPE DECLARATION>
                  |<IDENTIFIER> LITERALLY <STRING>
                  |<LEVLDEC>
<FULL TYPE DECLARATION>::= <TYPE DECLARATION>
                  |<TYPE DECLARATION><MODIFICATION>
<TYPE DECLARATION>::= <IDENTIFIER SPECIFICATION>
                                   <TYPE>
                  |<IDENTIFIER SPECIFICATION><TYPE>
                   <STORAGE CLASS>
                  |<IDENTIFIER SPECIFICATION><STORAGE
                    CLASS><TYPE>
                  |<IDENTIFIER SPECIFICATION><STORAGE
                    CLASS>
                  |<IDENTIFIER SPECIFICATION>
                  |<BOUND HEAD><NUMBER> ) <TYPE>
                  |<BOUND HEAD><NUMBER> ) <TYPE>
                               <STORAGE CLASS>
                  |<BOUND HEAD><NUMBER> ) <STORAGE
                                CLASS><TYPE>
                  |<BOUND HEAD><NUMBER><) <STORAGE
                                CLASS>
                  |<BOUND HEAD><NUMBER> )
<LEVLDEC> ::= <NUMBER><FULL TYPE DECLARATION>
            | <LEVLDEC>,<NUMBER><FULL TYPE DECLARA
                                          TION>
<TYPE> ::=  FIXED
          | <FIXED HEAD><FI-REPRESENTATION> )
          | FLOAT
          | <FLOAT HEAD><FL-REPRESENTATION> )
          | LABEL
          | <BIT HEAD><NUMBER> )
          | <BIT HEAD><NUMBER> )
<FIXED HEAD> ::=  FIXED  (
<FLOAT HEAD> ::= FLOAT (
<FI-REPRESENTATION> ::= <NUMBER>
                      | <REPRESENTATION TYPE>
              |<NUMBER><REPRESENTATION TYPE>
              |<NUMBER><REPRESENTATION TYPE><NUMBER>
<REPRESENTATION TYPE> ::= SV
              | SIGN_VALUE | UN | UNSIGNED | TC |
              TWO_COMPLEMENT | OC | ONE_COMPLEMENT
```

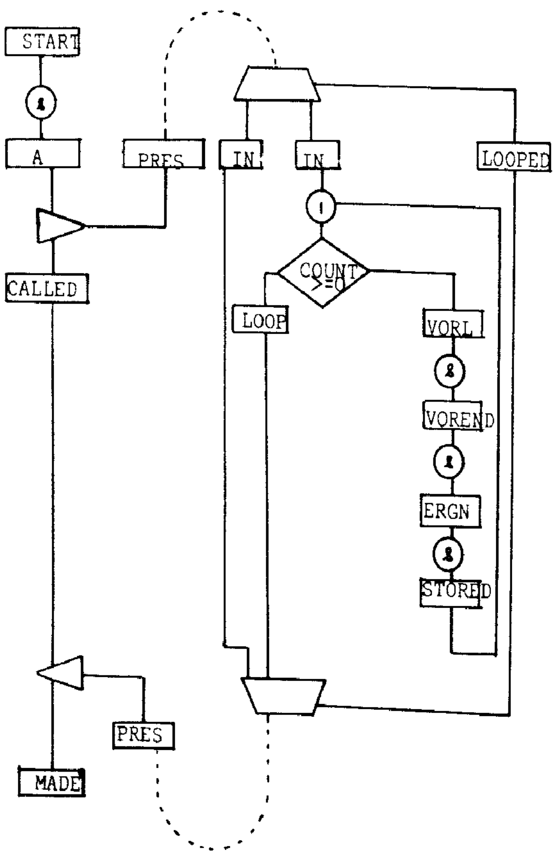
```
<FL-REPRESENTATION> ::= <MANT FI-REPRESENTATION>
                        <EXP FI-REPRESENTATION >
<MANT FI-REPRESENTATION> ::=  MANT <FI-REPRESENTA
                                              TION>
< EXP FI-REPRESENTATION> ::= EXP <FI-REPRESENTATION>
<BIT HEAD> ::= BIT  (
<BOUND HEAD> ::= <IDENTIFIER SPECIFICATION> (
               | <BOUND HEAD><NUMBER> ,
<IDENTIFIER SPECIFICATION> ::= <IDENTIFIER>
               | <IDENTIFIER LIST><IDENTIFIER> )
<IDENTIFIER LIST> ::=  (
               | <IDENTIFIER LIST><IDENTIFIER> ,
<MODIFICATION> ::= <INITIAL LIST>
               | <DEFINED LIST>
<INITIAL LIST> ::= <INITIAL HEAD><ITERATED CONSTANT>)
<INITIAL HEAD> ::=  INITIAL (
               | <INITIAL HEAD><ITERATED CONSTANT>
<ITERATED CONSTANT> ::= <CONSTANT>
                (<CONSTANT>) <CONSTANT>
                (<CONSTANT)(<CONSTANT>)<CONSTANT>
<DEFINED LIST> ::= <DEFINED HEAD><DEFINED ASSIGN-
                                           MENT>)
<DEFINED HEAD> :: = DEFINED (
               | <DEFINED HEAD><DEFINED ASSIGNMENT>,
<DEFINED ASSIGNMENT> ::= <VARIABLE>
                     | <VARIABLE> POSITION (<NUMBER>)
<STORAGE CLASS> ::= <SCOPE>
                  | <DURATION>
                  | <SCOPE><DURATION>
                  | <DURATION><SCOPE>
<SCOPE> ::= INTERNAL | EXTERNAL
<DURATION> ::= STATIC | DYNAMIC
<ASSIGNMENT> ::= <FREE VARIABLE><REPLACE><EXPRESSION>
               | <LEFT PART><ASSIGNMENT>
<REPLACE> ::= :=
<LEFT PART> ::= <FREE VARIABLE>,
              | <FREE VARIABLE> ||
<FREE VARIABLE> ::= <VARIABLE>
                  | <SUBSTRHEAD>)
<SUBSTRHEAD> ::=  SUBSTR (<VARIABLE>
           SUBSTR (<VARIABLE>,<EXPRESSION>
           SUBSTR (<VARIABLE>,<EXPRESSION>,<EXPRESSION>
<EXPRESSION> ::= <LOGICAL FACTOR>
  | <EXPRESSION> <OR EQUIVALENT><LOGICAL FACTOR>
<OR EQUIVALENT> ::= |  | ¬|  |  | ¬
<LOGICAL FACTOR> ::= <LOGICAL SECCNDARY>
  |<LOGICAL FACTOR><AND EQUIVALENT><LOGICAL
                                   SECONDARY>
<AND EQUIVALENT> ::=  & |¬&
<LOGICAL SECONDARY> ::= <LOGICAL PRIMARY>
                      |¬ <LOGICAL PRIMARY>
<LOGICAL PRIMARY> ::= <STRING EYPRESSION>
       |<STRING EXPRESSION><RELATION><STRING EX-
                                     PRESSION>
<RELATION> ::= = | < | > | ¬ = | ¬ < | ¬ > | < =
                      | > =
<STRING EXPRESSION> ::= <ARITHMETIC EXPRESSION>
  | <STRING EXPRESSION> <ARITHMETIC EXPRESSION>
<ARITHMETIC EXPRESSION> ::= <TERM>
  |  <ARITHMETIC EXPRESSION> + <TERM>
  |  <ARITHMETIC EXPRESSION> - <TERM>
  |  + <TERM>
  |  - <TERM>
<LOGIC OPERATOR> ::= <OR EQUIVALENT>
                   | <AND EQUIVALENT>
<TERM> ::= <PRIMARY>
         | <TERM>*<PRIMARY>
         | <TERM>/<PRIMARY>
         | <TERM> MOD <PRIMARY>
<PRIMARY> ::= <CONSTANT>
          | <VARIABLE>
          | <PRIMARY HEAD><EXPRESSION>)
<PRIMARY HEAD> ::= (
          | <LOGICAL OPERATOR> (
<VARIABLE> ::= <IDENTIFIER>
          | <SUBSCRIPT HEAD><EXPRESSION> )
          | <VARIABLE>.<IDENTIFIER>
          | <VARIABLE>.<SUBSCRIPT HEAD><EXPRESSION>)
<SUBSCRIPT HEAD> ::= <IDENTIFIER> (
          | <SUBSCRIPT HEAD><EXPRESSION>,
<CONSTANT> ::= <STRING>
          | <NUMBER>
```

Appendix 1: DIGITEST II description of a decimal adder.

a) Pure behavioral description

```
# DECADD:CIRCUIT (START) RETURNS (MADE);

DCL (REGA,REGB) (16) FIXED (4,UN) STATIC EXTER-
                                          NAL;
DCL (REGAS,REGBS)(4)BIT(16)DEFINED (REGA,REGB);
DCL COUNT FIXED, START LABEL;
DCL LOOPED LABEL INIT (1);
ON (START):A:COUNT=15;UEBIN="0";
CALLED:ON(A): CALL PRES;
ON CASE (PRES:IN,IN) PROCEDURE, CNT (LOOPED);
DCL I FIXED;
DCL (VORLERG, ERG) BIT(4), EUBERTRAG BIT(1)
                                  STATIC;
ON (IN): LOOP:DO WHILE (COUNT >=0);
VORL:VORLERG = REGA(0)+REGB(0)+UEBIN;
ON (VORL): VOREND:IF VORLERG >9 THEN ERG=VORLERG
                               + 6;
                             ELSE ERG=VORLERG;
ON (VOREND): ERGN:IF ERG >15  THEN UEBERTRAG="1";
                             ELSE UEBERTRAG="0";
ON (ERGN):STORED:REGAS = ERG ||SUBSTR (REGAS,1,
                                          15);
REGBS=SUBSTR (REGBS,0,1)||SUBSTR (REGBS,1,15);
COUNT = COUNT -1;
UEBIN = UEBERTPAG;
END;
ON CASE (PRES:(IN,LOOP))
END PRES, CNT (LOOPED);
MADE: ON (←PRES, CALLED):;
# CIRCUITEND DECADD;
```



b)Mixed structural and behavioral description.

```
# DECADD:CIRCUIT (START) RETURNS (MADE);
DCL (REGA,REGB) (16) FIXED (4,UN), COUNT FIXED,
                                   START LABEL;
DCL (REGAS,REGBS) (4) BIT (16) DEFINED (REGA,
                                       REGB);
DCL LOOPED LABEL INIT (1);
ON (START):A: COUNT = 15; UEBIN ="0";
ON CASE (PRES:IN,IN)):PROCEDURE,CNT (LOOPED);
DCL (VORLERG,ERG) BIT (4);
ON (IN):LOOP:SEQUENTIAL DO WHILE (COUNT >=0);
# VORLERG,UEB1 = ADD4 (REGA(0),REGB(0),UEBIN);
CORRECT = VORLERG >9;
# ERGN: ERG, UEBERTRAG = ADD4 (VORLERG,"0",
                    CORRECT,CORRECT,"0",
UEB1),DELAY (→ERG:UP 24-28, DOWN 22-26/→ UEBERTRAG
                                       '15');
REGAS = ERG ||SUBSTR (REGAS,1,15);
REGBS = SUBSTR (REGBS,0,1) ||SUBSTR (REGBS,1,15);
COUNT=COUNT -1;
UEBIN = UEBERTRAG;
END
ON CASE (PRES:(IN,LOOP)):END, CNT (LOOPED);
CALLED ; ON (A) : CALL PRES;
MADE: ON (← PRES,CALLED):;
# CIRCUITEND DECADD;
```

d) Pure structural description.

```
# DECADD:CIRCUIT (START) RETURNS (MADE);
# PRES:CIRCUIT (REGAA1,REGAA2,REGAA3,REGAA4,
               REGBA1,REGBA2,REGBA3,REGBA4, START)
              RETURNS (ERGN,SHR,MADE);
  DCL (ERGN,VORLERG) BIT (4);
# CARRY:UEBIN,NQ = DFF (UEBERTRAG,SHR,LOAD,VCC),
  DELAY (SHR →'UEBIN':18-2o/LOAD →'UEBIN':2o-24),
  Mw (1o), FAN(IN: 1/OUT: 1o);
# BUSY: Q,NQ = CARRY. DFF (VCC,GND,LOAD,BORROW);
# CNT:CNT3,CNT2,CNT1,CNTo, BORROW,CARRY = HEXCNT
               (VCC,VCC,VCC,VCC,VCC,LOAD,VCC,
                CNTDN),DELAY (→ 12-18);
# LOAD = NOT (START);
# LOOP1 = AND (CNTDN,LOAD);
# LOOP2 = NOT (LOOP1),DELAY (→ '100';
# CNTDN = NAND (LOOP2,BUSY.NQ);
# SHR = NOT (CNTDN);
# VORLERG,UEB1 = ADD4 (REGAA1,REGAA2,REGAA3,REGAA4,
              REGBA1,REGBA2,REGBA3,REGBA4, UEBIN);
# CORRECT = NOT (CORR1);
# CORR1 = EXOR (CORR2,UEB1);
# CORR2 = EXOR (CORR3,CORR4);
# CORR3 = NOT (SUBSTR (VORLERG,3,1);
# CORR4 = EXOR (SUBSTR(VORLERG,2,1),SUBSTR(VORLERG,
                                             1,1);
# ERGN:ERG,UEBERTRAG = ADD4 (VORLERG,"0",CORRECT,
                                  CORRECT, "0",
  UEB1),DELAY(→ ERG:UP 24-28, DOWN 22-26/  UEBER-
                                  TRAG: '15');
# CIRCUITEND PRES;
# REGAA1,REGAA2,REGAA3,REGAA4=DECREG (SHR, REGAE1,
                            REGAE2,REGAE3,REGAE
# REGBA1,REGBA2,REGBA3,REGBA4=DECREG (SHR, REGBA1,
                            REGBA2,REGBA3,REGBA
# REGAE1,REGAE2,REGAE3,REGAE4,SHR,MADE = PRES
  (REGAA1,REGAA2,REGAA3,REGAA4,REGBA1,REGBA2,
   REGBA3,REGBA4,START);
# CIRCUITEND DECADD;
```