ISE-TR-83-33

SYSTEMATIC DESIGN OF A PIPELINED LEXICAL SCANNER

by

Kozo Itano

April 15, 1983

Systematic Design of a Pipelined Lexical Scanner

Kozo Itano

Institute of Information Sciences
& Electronics
University of Tsukuba
Sakura, Niihari, Ibaraki 305
Japan

This work was partly performed while the author was visiting at the Department of Computer Science, University of Maryland.

Contents

Abstract

A simple lexical scanner for ALGOL 60 subset was designed for the demonstration of the hardware design methodology. The lexical scanner was originally described in Software Design Language (SDL) [1]. A top-down approach was employed; first the equivalent hardware lexical scanner was described in the high-level hardware design language, and then it was translated into the register transfer level hardware design in Computer Design Language CDL [3-9]. Five stages of the pipeline architecture was described in both languages, and its simulation was also performed on the CDL3 simulator of the UNIVAC 1100 to test the algorithm.

Systematic Design of a Pipelined Lexical Scanner

Kozo Itano

University of Tsukuba

1. Introduction

As a result of the recent improvements in fabrication technology, achievable circuit density has rapidly increased and LSI chips now could consist of multiple complex systems easily. However, the design of such complex hardware is still difficult, since it is usually done in ad-hoc manner. In order to cope with this problem, a systematic approach has been introduced for the design of large and complex hardware systems.

For the demonstration of the methodology, a simple subset of ALGOL 60 was used; the hardware was described in high-level design language, and then it was translated into a lower level design gradually. As the lexical scanner has a pipeline architecture, an algorithm for parallel processing is described both in high-level design language and low-level language. The high-level language is an extention of SDL (Software Design Language) [1,2], and the low level language is the register transfer level Computer Design Language CDL [3-9]. In this paper, the hardware design methodology is demonstrated basing on the experimental design of the pipelined lexical scanner [17].

2. Hardware Design Methodology

The hardware is designed systematically in a top-down manner. First the algorithm to be executed in the hardware is described in software-like manner, then it is translated into the high-level hardware design , and finally the register transfer level design is developed.

(1) Algorithmic Level Design

The first and probably most improtant step of the design is to describe precisely what to be implemented; a complete and understandable description can be a good start point. From the view point of the algorithm description, software-like "sequential" description would be easiest, because a designer doesn't need to pay extra attention about detailed hardware concepts and sophisticated parallel processing possibility. Therefore, Software Design Language (SDL) [1] was chosen to design the algorithm at the first level.

Once an algorithm is described in SDL, it is translated into a high-level hardware description using high-level hardware design language. At this level, a physical structure of the hardware is reflected to the declaration and definitions of the operations. Each data unit is defined as physical storage such as a register and a memory. Hence, the operations are defined at the register transfer level in some sence. As the parallel data processing is to be described, parallel processing constructs are introduced into the design language.

Translation of the sequential algorithm defined in SDL into the parallel processing algorithm is carefully performed by analyzing it. There are two cases: (1) If the algorithm consists of explicit parallelism, it can be executed in parallel; (2) If the algorithm is purely sequential, we consider about sequential parallelism, or reconstruction of the algorithm. One important guide line which is commonly applied to these cases is to analyze the flow of data instead of control flow of the algorithm.

2.2 Register-Transfer-Level Design

The next step of the design is to develop the precise hardware design at the register transfer level using the Computer Design Language CDL from the high-level algorithmic description metioned above. The translation at this level is not so difficult, because it is made rather straightforward manner. The described algorithm in CDL can be simulated and the design can be checked. When some errors are found at this level, they are corrected at the high-level design and translated into CDL design again. These steps are repeated until the design is tested completely as possible.

2.3 Physical Design and Implementation

Once a verified hardware design is developed, it could be implemented easily. Several approaches are possible for the physical design and its implementation. Y. Chu at the Univ. of Maryland suggested the algorithm for the translation of the CDL design into VLSI pattern. Microprogram implementation would also be feasible as well as wired logic implementation.

3. Case Study - Design of a Lexical Scanner

As an example of the systematic hardware design, a simple lexical scanner was designed. A very simplified ALGOL 60 subset is scanned by the lexical scanner. It has 26 kind of tokens: 13 single character tokens ('=', '≠', '+', '-', '*', '/', '(', ')', ',', ';', ':', '$', and blank), 11 multiple character tokens ('**', ':=', 'GOTO', 'IF', 'THEN', 'ELSE', 'BEGIN', 'END', 'INTEGER', 'READ', and 'WRITE'), unsigned integer, and identifier. A source program of this language is scanned by the lexical scanner, separated into tokens, and each token is translated into a binary token code.

3.1 Design Blueprint

The lexical scanner was first described in SDL [1], The blueprint has seven procedures: MAIN, SCAN, NEXTCHAR, LOOKUPLEGAL, ERRORHANDLER, LOOKUPOPERATOR, and WRITETOKEN. These procedures were defined considering the control structure shown in Fig.1(a). The blueprint shows precisely and completely what to do in the lexical scanner and it behaves a good specification of the system to be designed.

In this design, a source program is assumed to be stored in the area IN and the generated tokens are to be stored in the area TOKEN. Two tables: LEGAL and OPERATOR, define legal characters to be permitted in the language and operators and reserved words of the language respectively.

3.2 High-level Description of the Pipeline Structure

At the next step, the blueprint was translated into the pipelined lexical scanner. The original procedures are reorganized as the pipelined stages considering the data flow. Fig.1(b) shows the structure of the pipelined stages. The blueprint of the pipelined lexical scanner is given in Appendix A. The operation of each stage is described below.

(1) First Stage (NEXTCHAR)

The first stage of the pipeline is used to fetch a character from the IN. Register I points to the current character location. This location is incremented one by one. The current character is fetched from the IN and sent to register CHV. At the same time, the contents of register I are sent to register II before the incrementation is performed. Thus, register II points to the location of the character in register CHV. In the blueprint shown in Appendix A, these operations are described as three assignment statements:

```
CHV:=IN(I); II:=I; I:=I+1;
```

These statements are performed in parallel.

(2) Second Stage (LOOKUPLEGAL)

The second stage of the pipeline is used to get the type of the character from the character-type check unit. An associative table is used in this unit; the character code is used as a key to the table LEGAL, and it outputs the type of the character. This operation is simply described as 'LCR:=LEGAL(CHV)', where LEGAL is declared as an associative table. In this stage, the contents of register CHV are sent to register CH; the contents of register II are sent to register III, and the output of the associative table LEGAL is sent to register LCR.

(3) Third Stage (SCAN)

The third stage of the pipeline is used to control the most essential operations of the lexical scanner. Operations to be performed here depend upon the type of the next character LCR and the type of the token processed in the string register SA. The type of the token M indicates that the string register holds the state of (i) empty, (ii) numeric, (iii) identifier, (iv) operator, etc.

Initially, the string register SA is cleared. Then, the character in register CH is sent to the string register, and it is appended to the character string in SA. When the first character of the current token is sent to the string register, the corresponding location is also sent to register IV to hold the first character location of the token in the IN. This operation is repeated until the current token string is terminated by the next token. When the next token is detected in this way, the state of the string register is changed and the next stage of the pipeline is invoked. Then the string register is cleared again, and the processing of the next token begins. State transition of this stage is shown in Fig.2.

In this stage, '++' means bitwise concatenation of the both sides of the operator, and ':=+' means that a new item is appended to the left-hand register. The construct 'Exec' is used to invoke the corresponding stage.

(4) <u>Fourth Stage (LOOKUPOPERATOR)</u>

The fourth stage of the pipeline is used to access the operator and reserved word look-up unit. An associative table is used in this unit; the string register SA is directly connected to the associative table OPERATOR in parallel, and it outputs the type of the token. When this stage is invoked by the third stage, the type of the token generated by the reserved word look-up unit is sent to register TOKENR, and the location in register IV is also sent to register V.

(5) <u>Fifth Stage (WRITETOKEN)</u>

The fifth stage of the pipeline is used to output the token into the memory TOKEN. After the token and its location are written into the memory, register J is incremented.

(6) <u>Error Code Handler (ERRORHANDLER)</u>

An error code handler was installed in the lexical scanner. When an error is detected on the third stage of the pipeline, the error code handler receives the error code indicating the type of the error and its location. The handler stores the error code into the memory ERROR and increments register Q.

3.3 CDL Design

The design blueprint of the pipeleined lexical scanner was translated nto the CDL design by hand. Each stage of the pipeline was retained in the CDL design. As shown in Fig.3, the design defines 19 data registers, 5 control registers, two associative tables, and three memories: IN, TOKEN and ERROR. As the hardware is to be described concretely at the detailed level, the three high-level notations: SA, LEGAL and OPERATOR were defined in detail. Synchronization mechanism was implemented also.

(1) String Register SA

The string register SA is used to hold a token string which is concatenated with a character fetched from the memory IN one by one. To implement this register, a shift register C0-C6 and OV was introduced as shown in Fig.3. OV is an overflow indicator of the register C0-C6. Every concatenate operation is replaced by a parallel shift operations of the shift rehister. By the use of this shift register, the string register SA was realized very compact, and the related operations can be performed very fast.

(2) Associative Table LEGAL

The associative table LEGAL is used to generate the type of the input character. The type specifies blank, digit, letter, single character operator, two character operator, and illegal characters. For the speed up of the table look-up operation, a sequential table look-up mechanism was avoided, and a ROM was used for the table LEGAL. As only 50 kind of characters are to be processed, the character code itself is used directly as an address to the ROM LEGAL. The addressed word of the ROM contains the type of the corresponding character. This table look up is assumed to be performed within a single clock cycle.

(3) Associative Table OPERATOR

The associative table OPERATOR is used to generate the type of a token from the token string. The token type specifies each token with a binary number. Every single character operators, multiple character operators and reserved words are transformed into a unique binary number.

As the token string is too long for ROM implementation, a PLA is used to realize the table OPERATOR. The PLA logic is defined with TERMINAL statements in CDL3 [4-7]. The table consists of whole predefined multiple and single letter operators and reserved words. Characters of these operators and reserved words are arranged in a reversed order as shown in Fig.4, so that they can be compared with input token string in the shift register C0-C6 and OV. If the input token string matches with one of the contents in the PLA, the corresponding signal is sent to the encoder logic and a proper token type is generated from the encoder as an output.

(4) Synchronization of the Pipeline Stages

For the synchronization of each stage of the pipeline, four control registers are used: NEXTCH, NEXCH2, G, and WT. NEXTCH is used to control the frist and second stages of the pipeline, and NEXCH2 is used to control the NEXTCH. G is used to control the third stage of the pipeline. WT is used to control the fifth stage of the pipeline.

4. Simulation in CDL3

Fig.5 shows the surmarized traced chart of the simulation and the output of the simulation of the pipelined lexical scanner. Simulation was made by the translator/simulator CDL3 on the UNIVAC 1100 at the University of Maryland. The necessary simulation cycles for the test data (79 characters) shown in Fig.5 are 85 cycles and it takes about 35 seconds in execution on the UNIVAC 1100. The simulation result shows that each character of the source program is processed in one clock period.

5. Concluding Remarks

A systematic hardware design methodology was developed to design a complex hardware system. The experimental design of a pipelined lexical scanner was successfully done in both high-level and low-level languages. Actual simulation was performed using CDL3 simulator on the UNIVAC 1100 to test the design. The methodology showed that the high-level design language gave us good prospect what to be made and also a precise specification of the hardware system. However, the pipeline processing is a special subclass of parallel processing which we should cope with. Therefore, the capability of the description of the high-level design language should be evaluated for the more general cases.

6. Acknowledgements

The author would like to express his sincere thanks to Professor Yaohan Chu at the University of Maryland for his helpful suggestions and discussions.

9. References

[1] Y. Chu, "Software Blueprint and Examples", Lexington Books, 1982.

[2] Y. Chu, "Computer System Design Description", Proc. of Design Automation Conference, Las Vegas, June 1982.

[3] Y. Chu, "An Algol-like Computer Design Language", CACM, Vol.8, No.10, 1965, pp.607-615.

[4] C.K. Mesztenyi, "Translator and Simulator for the Computer Design and Simulation Program (CDSP) Version I ", Technical Report TR-67-48, Computer Science Center, University of Maryland, May 1967.

[5] C.K. Mesztenyi, "Computer Design Language - Simulation and Boolean Translation", Technical Report TR-68-72, Computer Science Center, University of Maryland, June 1968.

[6] Y. Chu, "User's Manual for the CDL3 Simulator", Department of Electrical Engineering, University of Maryland, May 1978.

[7] Y. Chu, "How to Use the CDL3 Simulator on the Univac 1100", Department of Electrical Engineering, University of Maryland, May 1978.

[8] J. R. Heath, B. D. Carroll, and T. T. Cwik, "Capabilities and Limitations of CDL as a System Hardware and Software Design Aid", Journal of Design Automation & Fault-Tolerant Computing, Vol.2, No.2, May 1978, pp.93-116.

[9] F. J. Mowle and L. R. Stine, "Purdue Extended CDL - A Digital Design Language for the Specification and Simulation of Computer Hardware", Technical Report TR-EE 75-14, School of Electrical Engineering, Purdue University, May 1975.

[10] Y. Chu, "High-Level Language Computer Architecture", Academic Press, 1975.

[11] Y. Chu, "JOVIAL Direct-Execution Computer", Proc. of High-Level Language Computer Architecture Conference, 1980, pp.17-32.

[12] Y. Chu and M. Abrams, "Programming Languages and Direct-Execution Computer Architecture", Computer, Vol.14, No.4, July 1981, pp.22-32.

[13] Y. Chu, K. Itano, Y. Fukunaga, and M. Abrams, "Interactive Direct-Execution Programming and Testing", Proc. of COMPSAC'82, Chicago, Nov. 1982.

[14] K. Itano and Y. Chu, "A Pascal Interactive Direct-Execution Computer: PASDEC, Part I: High Level Design", TR-1198, Department of Computer Science, University of Maryland, August 1982.

[15] K. Itano, "A Pascal Interactive Direct-Execution Computer: PASDEC, Part II: CDL Design and Simulation", TR-1202, Department of Computer Science, University of Maryland, August 1982.

[16] K. Itano, "PASDEC: A Pascal Interactive Direct-Execution Computer", Proc. of High-Level Language Computer Architecture Conference, Dec. 1982.

[17] K. Itano, "CDL Design of a Pipelined Lexical Scanner", TR-1093, Department of Computer Science, University of Maryland, Sept. 1981.

[18] D. P. Sengphiel, "Design of a Hardware Text Editor", TR-283, Computer Science Center, University of Maryland, Dec. 1973.

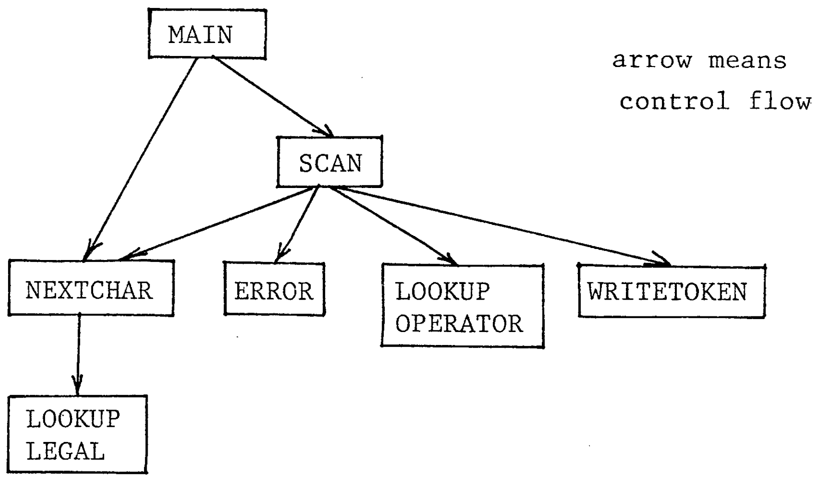


Fig.1(a) Relation between procedures in SDL Design

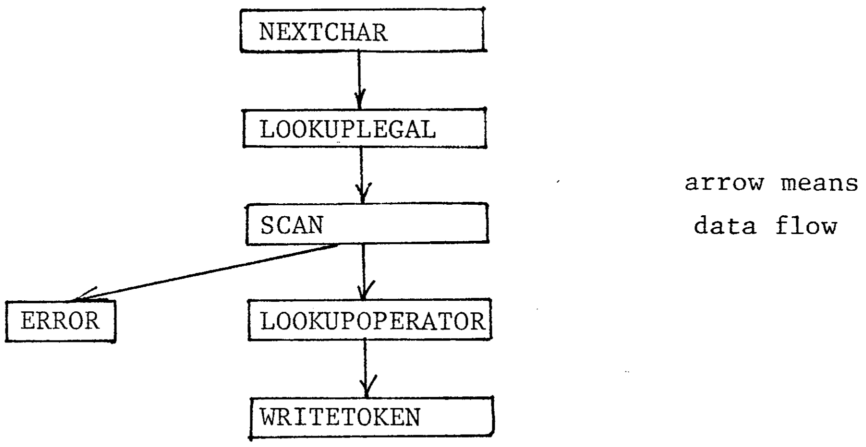
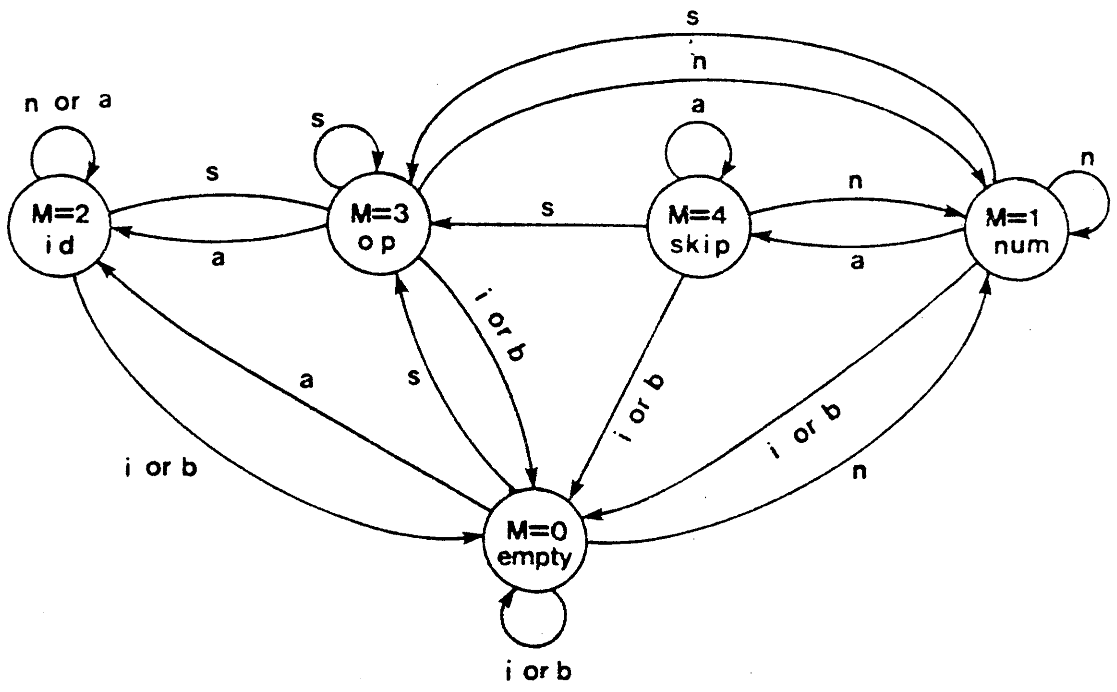


Fig.1(b) Relation between stages in SDL-H2 Design



Class of the input character	
i(LCR=1):	illegal char
b(LCR=2):	blank
n(LCR=3):	numeric
a(LCR=4):	alphabetic
s(LCR=5):	symbolic

State of the shift-register	
M=0:	empty
M=1:	number
M=2:	identifier or multi-letter op.
M=3:	single or double letter op.
M=4:	error skip

Figure 2. State diagram of the SCAN

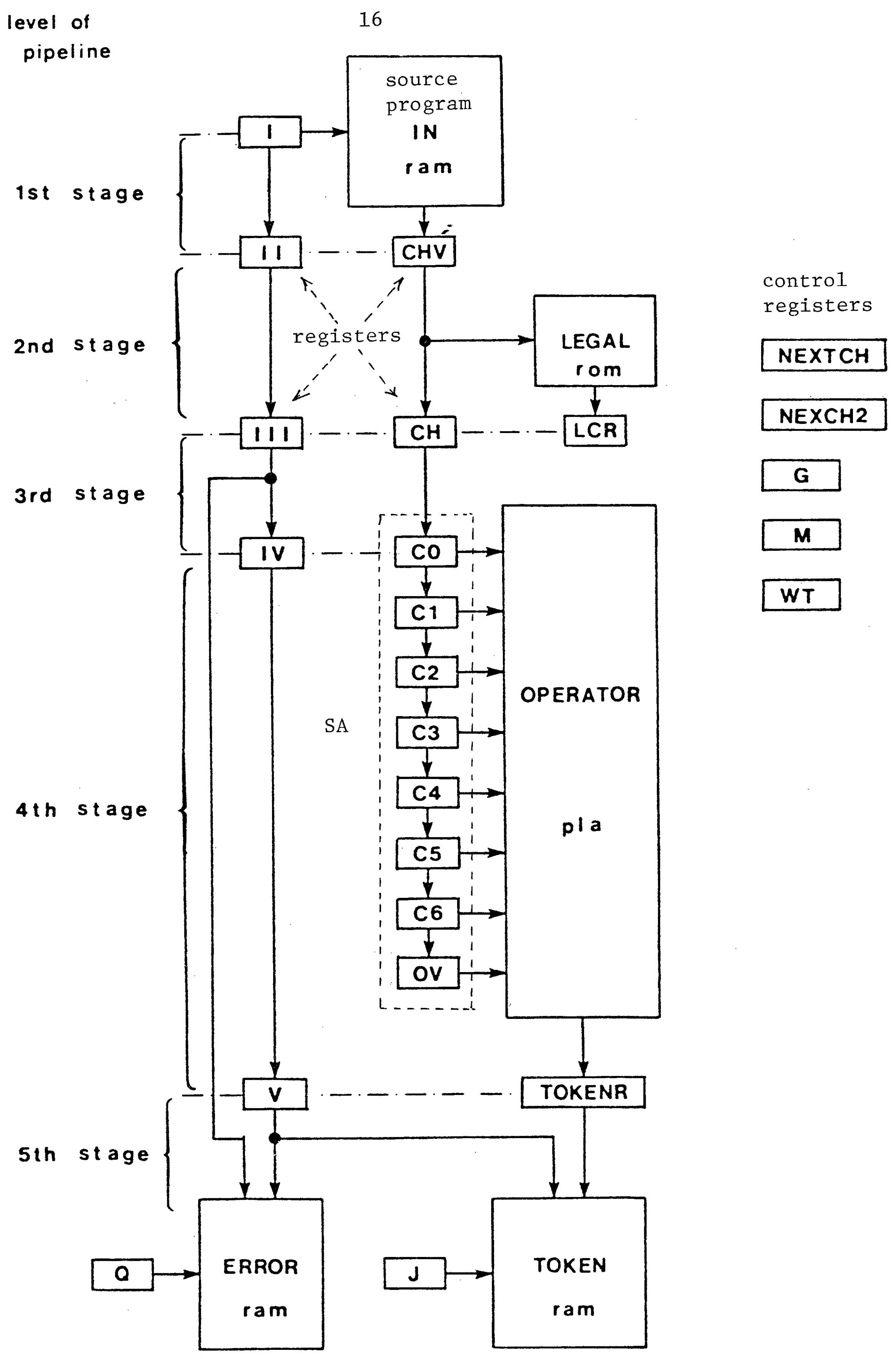


Figure 3. Hardware organization of the pipelined lexical scanner

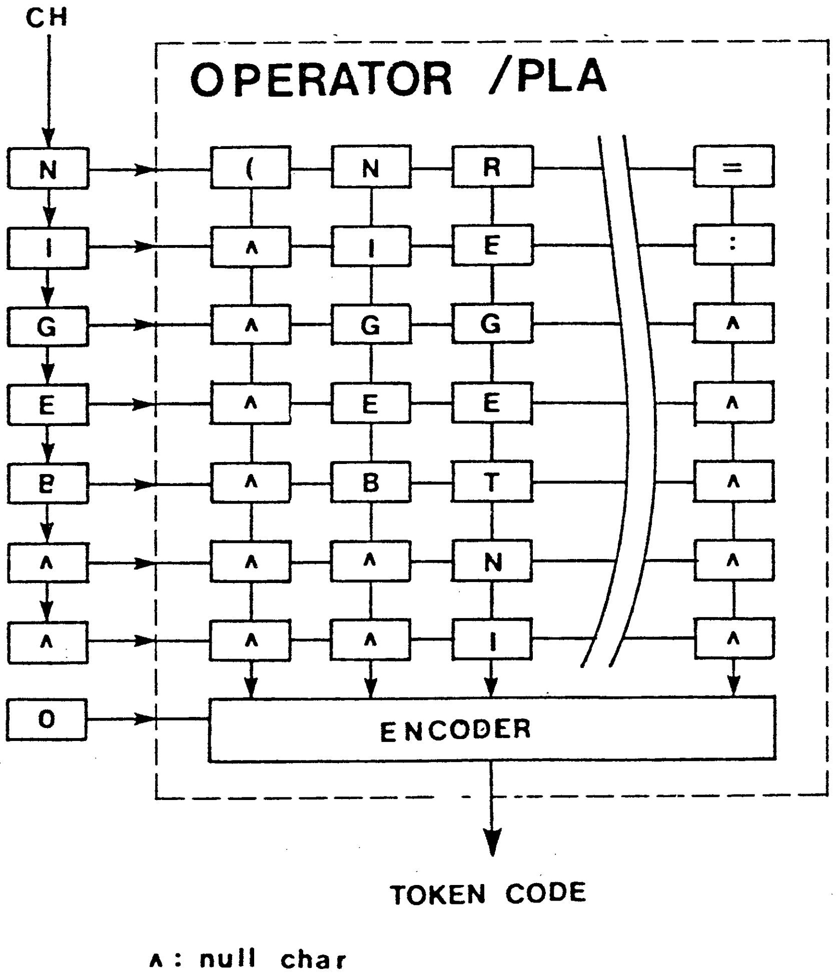


Figure 4. Organization of the "OPERATOR" PLA

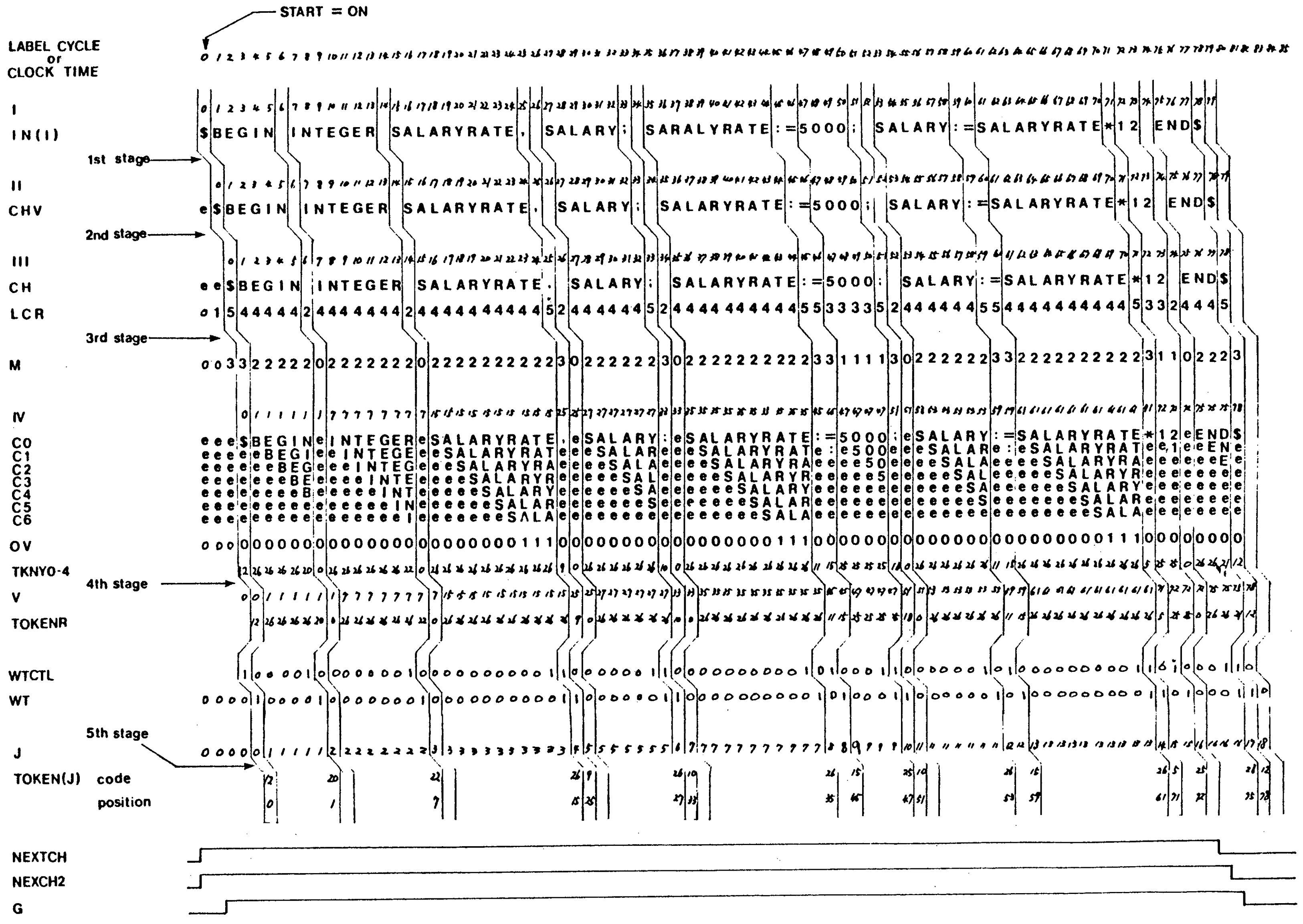


Figure 5. Surmmarized trace chart of the simulation.

Appendix A. Hardware Blueprint of the Pipelined Lexical Scanner

```
01 Hardware Blueprint

   02 Stage

      03 Declaration

         #1  MAIN              /* main control */
         #2  NEXTCHAR          /* fetch next character from IN */
         #3  LOOKUPLEGAL       /* look up legal character table */
         #4  SCAN              /* recognize a token */
         #5  LOOKUPOPERATOR    /* look up operator table */
         #6  WRITETOKEN        /* output token into TOKEN */
         #7  ERRORHANDLER      /* error code handler */

   02 Data

      03 Declaration

         Ram
             IN(I)=IN(0-127,0-5); /* stores source program */
             TOKEN(J)=TOKEN(0-127,0-11); /* stores token codes */
             ERROR(Q)=ERROR(0-127,0-14); /* stores error code & location */

         Associative Table
             LEGAL(CHV)=  ('0',3),('1',3),('2',3),('3',3),
                          ('4',3),('5',3),('6',3),('7',3),
                          ('8',3),('9',3),('A',4),('B',4),
                          ('C',4),('D',4),('E',4),('F',4),
                          ('G',4),('H',4),('I',4),('J',4),
                          ('K',4),('L',4),('M',4),('N',4),
                          ('O',4),('P',4),('Q',4),('R',4),
                          ('S',4),('T',4),('U',4),('V',4),
                          ('W',4),('X',4),('Y',4),('Z',4),
                          ('=',5),('≠',5),('+',5),('-',5),
                          ('*',5),('/',5),('(',5),(')',5),
                          (',',5),(';',5),(':',5),('$',5),
                          (' ',2),('@',1);
            OPERATOR(SA)=  ('=',1   ),('≠',2   ),('+',3   ),('-',4   ),
                           ('*',5   ),('/',6   ),('(',7   ),(')',8   ),
                           (',',9   ),(';',10  ),(':',11  ),
                           ('$',12  ),(' ',13  ),('**',14  ),
                           (':=',15  ),('GOTO',16  ),('IF',17  ),
                           ('THEN',18  ),('ELSE',19  ),
                           ('BEGIN',20  ),('END',21  ),
                           ('INTEGER',22  ),('READ',23  ),
                           ('WRITE',24  ),('UN',25)
                           ('ID',26  ),('@',27  );
```

```
    Register
        I(0-6);          /* source program pointer */
        II(0-6);
        III(0-6);
        IV(0-6);
        V(0-6);
        Q(0-6);          /* ERROR Memory pointer */
        J(0-4);          /* TOKEN memory pointer */
        CHV(0-5);        /* character register */
        CH(0-5);
        SA(0-2,0-5);     /* token string register */
        TOKENR(0-4);     /* token code register */
        ERR(0-7);        /* error code & location register */

02 Control

   03 Declaration

      Switch LCR Of Status(1,2,3,4,5);
                    /* LCR=1: illegal character,
                       LCR=2: blank,
                       LCR=3: digit,
                       LCR=4: letter,
                       LCR=5: special symbolic character */
             M   Of Status(0,1,2,3,4);
                    /* M=0: SA is empty,
                       M=1: SA holds unsigned integer,
                       M=2: SA holds identifier,
                       M=3: SA holds operator,
                       M=4: error skip */

02 Definition

   #1 Stage MAIN;

      /* Initialization & synchronization */

      End MAIN;

   #2 Stage NEXTCHAR;

         Block
            CHV:=IN(I); II:=I; I:=I+1;
         Endblock;
      End NEXTCHAR;

   #3 Stage LOOKUPLEGAL;

         Block
            LCR:=LEGAL(CHV); III:=II; CH:=CHV;
         Endblock;
      End LOOKUPLEGAL;
```

```
#4 Stage SCAN;

    Case M Of
    M=0: /* string register SA is empty */
        Case LCR Of
        LCR=1: Block ERR:=0++III; Exec ERRORHANDLER; Endblock;
        LCR=2: Do Nothing;
        LCR=3: Block M:=1; SA:=CH; IV:=III; Endblock;
        LCR=4: Block M:=2; SA:=CH; IV:=III; Endblock;
        LCR=5: Block M:=3; SA:=CH; IV:=III; Endblock;
        Endcase;
    M=1: /* string register SA has unsigned integer */
        Case LCR Of
        LCR=1: Block
                   ERR:=0++III; Exec ERRORHANDLER;
                   Exec WRITETOKEN; SA:=empty; M:=0;
               Endblock;
        LCR=2: Block Exec WRITETOKEN; SA:=empty; M:=0; Endblock;
        LCR=3: Block SA:=+ CH; Endblock;
        LCR=4: Block
                   ERR:=1++III; Exec ERRORHANDLER;
                   Exec WRITETOKEN; SA:=empty; M:=4
               Endblock;
        LCR=5: Block
                   Exec WRITETOKEN; SA:=CH; M:=3; IV:=III;
               Endblock;
        Endcase;
    M=2: /* identifier */
        Case LCR Of
        LCR=1: Block
                   ERR:=0++III; Exec ERRORHANDLER;
                   Exec WRITETOKEN; SA:=empty; M:=0;
               Endblock;
        LCR=2: Block Exec WRITETOKEN; SA:=empty; M:=0; Endblock;
        LCR=3: Block SA:=+CH; Endblock;
        LCR=4: Block SA:=+CH; Endblock;
        LCR=5: Block
                   Exec WRITETOKEN; SA:=CH; M:=3; IV:=iii;
               Endblock;
        Endcase;
    M=3: /* operator */
        Case LCR Of
        LCR=1: Block
                   ERR:=0++III; Exec ERRORHANDLER;
                   Exec WRITETOKEN; SA:=empty; M:=0;
               Endblock;
        LCR=2: Block Exec WRITETOKEN; SA:=empty; M:=0; Endblock;
        LCR=3: Block
                   Exec WRITETOKEN; SA:=CH; M:=1; IV:=III;
               Endblock;
        LCR=4: Block
                   Exec WRITETOKEN; SA:=CH; M:=2; IV:=III;
               Endblock;
```

```
          LCR=5: If SA='*' And CH='*' Or SA=':' And CH='='
                 Then SA:=+CH;
                 Else Block
                         Exec WRITETOKEN; SA:=CH; IV:=III;
                      Endblock;
                 Endif;
          Endcase;
      M=4: /* error skip */
          Case LCR Of
          LCR=1: Block
                    ERR:=0++III; Exec ERRORHANDLER; SA:=empty; M:=0;
                 Endblock;
          LCR=2: Block SA:=empty; M:=0; Endblock;
          LCR=3: Block SA:=CH; M:=1; IV:=III; Endblock;
          LCR=4: Do Nothing;
          LCR=5: Block SA:=CH; M:=3; IV:=III; Endblock;
          Endcase;
      Endcase;

#5 Stage LOOKUPOPERATOR;

      Block TOKENR:=OPERATOR(SA); V:=IV; Endblock;

   End LOOKUPOPERATOR;

#6 Stage WRITETOKEN;

      Block
        TOKEN(J):=TOKENR++V; J:=J+1;
      Endblock;
   End WRITETOKEN;

#7 Stage  ERRORHANDLER;

      Block
         ERROR(Q):=ERR; Q:=Q+1;
      Endblock;
   End ERROR;
```

APPENDIX B. CDL Coding of the Pipelined Lexical Scanner

```
$TRANSLATE
*MAIN

COMMENT ** A Pipelined Lexical Scanner

 REGISTER, I(0-6),        $points to buffer IN
1          II(0-6),       $position of character in CHV
1          III(0-6),      $position of character in CH
1          IV(0-6),       $position of the first char of the token
1          V(0-6),        $head position of token in TOKENR
1          J(0-4),        $points to the memory TOKEN
1          Q(0-6),        $points to the memory ERROR
1          M(0-2),        $state of the shift register C0-C6, OV
1          G,             $control reg for 3rd and 4th stage
1          NEXTCH,        $control reg for 1st and 2nd stage
1          NEXCH2,        $control NEXTCH
1          WT,            $control reg for 5th stage
1          LCR(0-2),      $holds class of character in CH
1          CHV(0-5),      $holds character (between 1st and 2nd stage)
1          CH(0-5),       $holds character (between 2nd and 3rd stage)
1          C0(0-5),       $1st register of the shift register
1          C1(0-5),       $2nd register of the shift register
1          C2(0-5),       $3rd register of the shift register
1          C3(0-5),       $4th register of the shift register
1          C4(0-5),       $5th register of the shift register
1          C5(0-5),       $6th register of the shift register
1          C6(0-5),       $7th register of the shift register
1          OV,            $overflow indicator of the shift register
1          TOKENR(0-4)    $holds token code

 MEMORY,   IN(I)=IN(0-177,0-5),           $source program memory
1          TOKEN(J)=TOKEN(0-31,0-13),     $token memory
1          ERROR(Q)=ERROR(0-177,0-16),    $error information
1          LEGAL(CHV)=LEGAL(0-77,0-2)     $character class table

 DECODER,  LC(1-5)=LCR,             $class of character
1          MX(0-4)=M                $state of the shift register

 TERMINAL, MDE=MX(0),               $shift register is empty (C0-C6,OV)
           MDN=MX(1),               $shift register holds numeric
1          MDI=MX(2),               $shift register holds identifier
C                                    or multi-letter operator
1          MDO=MX(3),               $shift register holds symbolic operator
1          MDK=MX(4),               $skip mode (error handling)
1          MDNI=MX(1)+MX(2),
1          MDNIO=MX(1)+MX(2)+MX(3)

 TERMINAL, LCILL=LC(1),             $CH is an illegal character
1          LCBLK=LC(2),             $CH is a blank character
1          LCNUM=LC(3),             $CH is a numeric character
```

```
1          LCALP=LC(4),          $CH is an alphabetic character
1          LCSYB=LC(5)           $CH is a symbolic character

 TERMINAL, OPTWO=CO.EQ.':'*CH.EQ.'='+CO.EQ.'*'*CH.EQ.'*'    $':=' OR '**

 TERMINAL, WTCTL=LCILL*MDNIO+    $error/illegal character
1                LCBLK*MDNIO+    $separator/blank
1                LCNUM*MDO+      $numeric preceding by operator
1                LCALP*MDO+      $alphabetic preceding by operaor
1                LCSYB*MDNI+     $num or id followed by symbolic. char
1                LCSYB*MDO*OPTWO $two char operator (:= or **)


COMMENT PLA Definition

 TERMINAL, ZC1=C1.EQ.77,       $C1 is empty
1          ZC2=C2.EQ.77,       $C2 is empty
1          ZC3=C3.EQ.77,       $C3 is empty
1          ZC4=C4.EQ.77,       $C4 is empty
1          ZC5=C5.EQ.77,       $C5 is empty
1          ZC6=C6.EQ.77,       $C6 is empty
1          ZC56=ZC5*ZC6*OV',   $C5,C6,OV are empty
1          ZC46=ZC4*ZC56,      $C4,C5,C6,OV are empty
1          ZC36=ZC3*ZC46,      $C3,C4,C5,C6,OV are empty
1          ZC26=ZC2*ZC36,      $C2,C3,C4,C5,C6,OV are empty
1          ZC16=ZC1*ZC26       $C1,C2,C3,C4,C5,C6,OV are empty

C      Definition of AND terms

 TERMINAL, Y1=CO.EQ.'='*ZC16,    $   '='
1          Y2=CO.EQ.'≠'*ZC16,    $   '≠'
1          Y3=CO.EQ.'+'*ZC16,    $   '+'
1          Y4=CO.EQ.'-'*ZC16,    $   '-'
1          Y5=CO.EQ.'*'*ZC16,    $   '*'
1          Y6=CO.EQ.'/'*ZC16,    $   '/'
1          Y7=CO.EQ.'('*ZC16,    $   '('
1          Y10=CO.EQ.')'*ZC16,   $   ')'
1          Y11=CO.EQ.','*ZC16,   $   ','
1          Y12=CO.EQ.';'*ZC16,   $   ';'
1          Y13=CO.EQ.':'*ZC16,   $   ':'
1          Y14=CO.EQ.'$'*ZC16,   $   '$'
1          Y15=CO.EQ.' '*ZC16,   $   ' '
1          Y16=CO.EQ.'*'*C1.EQ.'*'*ZC26,    $   '**'
1          Y17=CO.EQ.'='*C1.EQ.':'*ZC26,    $   ':='
1          Y20=CO.EQ.'O'*C1.EQ.'T'*C2.EQ.'O'*C3.EQ.'G'*ZC46,   $'GOTO'
1          Y21=CO.EQ.'F'*C1.EQ.'I'*ZC26,                       $'IF'
1          Y22=CO.EQ.'N'*C1.EQ.'E'*C2.EQ.'H'*C3.EQ.'T'*ZC46,   $'THEN'
1          Y23=CO.EQ.'E'*C1.EQ.'S'*C2.EQ.'L'*C3.EQ.'E'*ZC46,   $'ELSE'
1          Y24=CO.EQ.'N'*C1.EQ.'I'*C2.EQ.'G'*C3.EQ.'E'*
1              C4.EQ.'B'*ZC56,                                 $'BEGIN'
1          Y25=CO.EQ.'D'*C1.EQ.'N'*C2.EQ.'E'*ZC36,             $'END'
1          Y26=CO.EQ.'R'*C1.EQ.'E,*C2;EQ.'G'*C3.EQ.'E'*
1              C4.EQ.'T'*C5.EQ.'N'*C6.EQ.'I'*OV'               $'INTEGE
1          Y27=CO.EQ.'D'*C1.EQ.'A'*C2.EQ.'E'*C3.EQ.'R'*ZC46,   $'READ'
```

```
1           Y30=C0.EQ.'E'*C1.EQ.'T'*C2.EQ.'I'*C3.EQ.'R'*
1               C4.EQ.'W'*ZC56,                                      $'WRITE'

C           Definition of OR terms

 TERMINAL,  TK0=Y20+Y21+Y22+Y23+Y24+Y25+Y26+Y27+Y30,                  $BIT 0
1           TK1=Y10+Y11+Y12+Y13+Y14+Y15+Y16+Y17+Y30,                  $BIT 1
1           TK2=Y4+Y5+Y6+Y7+Y14+Y15+Y16+Y17+Y24+Y25+Y26+Y27,          $BIT 2
1           TK3=Y2+Y3+Y6+Y7+Y12+Y13+Y16+Y17+Y22+Y23+Y26+Y27,          $BIT 3
1           TK4=Y1+Y3+Y5+Y7+Y11+Y13+Y15+Y17+Y21+Y23+Y25+Y27           $BIT 4

 TERMINAL,  ZTK=(TK0+TK1+TK2+TK3+TK4)'+OV   $SHIFT REG HOLDS IDENTIFIER

C                31:NUMERIC, 32:IDENTIFIER

 TERMINAL,  TKNY0=TK0+ZTK*(MDN+MDI),        $BIT 0 OF TOKEN CODE
1           TKNY1=TK1+ZTK*(MDN+MDI),        $BIT 1 OF TOKEN CODE
1           TKNY2=TK2,                      $BIT 2 OF TOKEN CODE
1           TKNY3=TK3+ZTK*MDI,              $BIT 3 OF TOKEN CODE
1           TKNY4=TK4+ZTK*MDN               $BIT 4 OF TOKEN CODE

COMMENT Block Definition

 BLOCK,     CLEAR(C0=77,C1=77,C2=77,        $Clear whole shift
1                C3=77,C4=77,C5=77,         $register: C0-C6 and OV
1                C6=77,OV=0),
1           INI(  C0=CH,C1=77,C2=77,        $Set the 1st char in C0,
1                C3=77,C4=77,C5=77,         $and keep the location
1                C6=77,OV=0,IV=III),        $of it
1           SHIFT(C0=CH,C1=C0,C2=C1,C3=C2,  $Shift data and
1                C4=C3,C5=C4,C6=C5,         $check overflow
1                IF(C6.NE.77) THEN(OV=1))

 SWITCH,    START(ON)
 CLOCK,     P

C         Initialization

 /START(ON)/   I=0,J=0,Q=0,M=0,G=0,WT=0,LCR=0,
               NEXTCH=1,           $invoke FETCH seq.
               NEXCH2=1,
               CHV=77,             $Set empty character
               CH=77,              $Set empty character
               DO-CLEAR

C           Fetch character from memory IN

 /NEXTCH*P/    CHV=IN(I),          $1ST STAGE
               I=I.COUNT.,
               II=I,
               III=II,             $2ND STAGE
               CH=CHV,
               LCR=LEGAL(CHV),
               IF(CHV.EQ.'$') THEN(G=1,M=3)   $invoke SCAN seq.
 /G'*P/         CO=CH,                       $3RD STAGE
                IV=III

 /G*P/          IF(CHV.EQ.'$') THEN(NEXTCH=0),  $terminate FETCH seq.
                IF(NEXTCH.EQ.0) THEN(NEXCH2=0), $turn off G
                IF(NEXCH2.EQ.0) THEN(G=0),      $terminate SCAN seq.
                TOKENR=TKNY0-TKNY1-TKNY2-TKNY3-TKNY4,  $4TH STAGE
                V=IV                                   $4TH STAGE

C           State transition of SCAN

 /G*LCILL*MDE*P/    M=0,          $NEXT STATE=MDE
                    ERROR(Q)=0-V-III,    $err code 0 means illegal char
                    Q=Q.COUNT.,
                    DO-CLEAR
C/G*LCBLK*MDE*P/    DO NOTHING
 /G*LCNUM*MDE*P/    M=1,          $NEXT STATE=MDN
                    DO-INI
 /G*LCALP*MDE*P/    M=2,          $NEXT STATE=MDI
                    DO-INI
 /G*LCSYMB*MDE*P/   M=3,          $NEXT STATE=MDO
                    DO-INI
 /G*LCILL*MDNIO*P/  M=0,          $NEXT STATE=MDE     WTCTL=1
                    ERROR(Q)=0-V-III,    $err code 0 means illegal char
                    Q=Q.COUNT.,
                    DO-CLEAR
 /G*LCBLK*MDNIO*P/  M=0,          $NEXT STATE=MDE     WTCTL=1
                    DO-CLEAR
 /G*LCNUM*MDNI*P/   DO-SHIFT
 /G*LCNUM*MDO*P/    M=1,          $NEXT STATE=MDN     WTCTL=1
                    DO-INI
 /G*LCALP*MDN*P/    M=4,          $NEXT STATE=MDK
                    ERROR(Q)=1-V-III,    $err code 1 means sequence er
                    DO-CLEAR
 /G*LCALP*MDI*P/    DO-SHIFT
 /G*LCALP*MDO*P/    M=2,          $NEXT STATE=MDI     WTCTL=1
                    DO-INI
 /G*LCSYB*MDNI*P/   M=3,          $NEXT STATE=MDO     WTCTL=1
                    DO-INI
 /G*LCSYB*MDO*P/    IF(OPTWO) THEN(DO-SHIFT)
                              ELSE(DO-INI)        $WTCTL=1
 /G*LCILL*MDK*P/    M=0,          $NEXT STATE=MDE
                    ERROR(Q)=0-V-III,    $err code 0 means illegal cha
                    Q=Q.COUNT.,
                    DO-CLEAR
 /G*LCBLK*MDK*P/    M=0,          $NEXT STATE=MDE
                    DO-CLEAR
 /G*LCNUM*MDK*P/    M=1,          $NEXT STATE=MDN
                    DO-INI
C/G*LCALP*MDK*P/    DO NOTHING (CONTINUE SKIP)
 /G*LCSYB*MDK*P/    M=3,          $NEXT STATE=MDO
                    DO-INI

C           Write token into memory TOKEN
```

```
 /WT*P/                   TOKEN(J)=TOKENR-V,    $5TH STAGE
                          J=J.COUNT.

C          Synchronization for WT

 /MDE'*WTCTL*G*P/           WT=1
 /(MDE'*WTCTL*G)'*WT*P/  WT=0

                          END


$SIMULATE

*OUTPUT    CLOCK(1)=I,II,III,IV,NEXTCH,V,WT
*OUTPUT    CLOCK(1)=G,CHV,CH,LCR,NEXCH2,J,TOKENR
*OUTPUT    CLOCK(1)=C0,C1,C2,C3,C4,C5,C6
*OUTPUT    CLOCK(1)=OV,Q,M

*LOAD
   LEGAL(00-)=3,3,3,3,3,3,3,3,3,3,4,4,4,4,4,4,    $'0123456789ABCDEF'
   LEGAL(20-)=4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,    $'GHIJKLMNOPQRSTUV'
   LEGAL(40-)=4,4,4,4,5,5,5,5,5,5,5,5,5,5,5,5,    $'WXYZ=≠+-*/(),;:$'
   LEGAL(60-)=2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,    $' @'

   IN(000-)="$BEGIN INTEGER SALARYRATE, SALARY;"
   IN(035-)=" SALARYRATE:=5000; SALARY:="
   IN(061-)="SALARYRATE*12 END$ "

*SWITCH    1,START=ON
*SIM       125,20
```

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA.
SAKURA-MURA, NIIHARI-GUN, IBARAKI 305 JAPAN

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-83-33
TITLE Systematic Design of a Pipelined Lexical Scanner	
AUTHOR(S) Kozo Itano	
REPORT DATE April 15, 1983	NUMBER OF PAGES 27
MAIN CATEGORY Hardware, Register-transfer-level implementation	CR CATEGORIES B1, B5, B6
KEY WORDS hardware design methodology, systematic design, hardware logic simulation, lexical scanner, pipelined architecture	

ABSTRACT

A simple lexical scanner for Algol 60 subset was designed for the demonstration of the hardware design methodology. The lexical scanner was originally described in Software Design Language (SDL). A top-down approach was employed; first the equivalent hardware lexical scanner was described in the high-level hardware design language, and then it was transtlated into the register transfer level hardware design in Computer Design Language CDL. Five stages of the pipeline architecture was described in both languages, and its simulation was also performed on the CDL3 simulator of the UNIVAC 1100 to test the algorithm.

SUPPLEMENTARY NOTES