AN INTRODUCTION TO THE DDL-P LANGUAGE

W.E. Cory, J.R. Duley, W.M. vanCleemput

Technical Report No. 163

March 1979

Computer Systems Laboratory
Stanford University
Stanford, California 94305

ABSTRACT

This report describes the Pascal-based implementation of DDL (Digital Design Language) and its simulator.

INDEX TERMS: Design automation, computer-aided design, hardware description languages, DDL, digital design language.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

This report is one of two manuals describing a compiler and simulator for DDL-P, a subset of DDL (Digital Design Language). DDL is a language for describing the behavior of digital systems at the Boolean equation, register transfer, and algorithmic levels. It uses a finite state machine notation and it may be used to describe systems over a wide range of levels.

DDL was originally formulated by Duley at the University of Wisconsin in 1967 [5, 6, 3]. A translator and simulator for a subset of DDL were implemented in FORTRAN [1, 7, 4, 8, 9, 10]. In 1971-73, J. Duley, B. Clark, and J. Welsch implemented an interactive simulation system for a subset of DDL (with modified syntax) on the HP 2100 system in HP-Algol at Hewlett-Packard Laboratories. The DDL-P language, compiler, and simulator are based on this HP implementation. In order to enhance portability the system was rewritten in PASCAL on the DEC-20 system under the TOPS-20 Operating System at Stanford University. Small changes were made to the

syntax, mainly to enhance the readability. The system will still accept the input format of the original HP-Algol version.

This report describes the DDL-P subset of the language as it was implemented at Stanford. Several examples are given together with instructions for using the compiler on the LOTS DEC-20 system at Stanford. The appendices contain a list of the error messages and a formal BNF definition of the language accepted by the compiler. A companion manual describes the use of the simulator and its command language [2].

Chapter 2

CHARACTER SET, IDENTIFIERS, AND CONSTANTS

2.1 CHARACTER SET

```
letter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
           N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
           a|b|c|d|e|f|g|h|i|j|k|l|m|
           n|o|p|q|r|s|t|u|v|w|x|y|z

digit  ::= 0111213141516171819
```

DDL-P uses a subset of the 7-bit ASCII character set, including the letters A-Z,a-z and the digits 0-9. Upper and lower case letters are considered to be equivalent; in the listing generated by DDL-P, all letters are printed in upper case. The following non-alphanumeric characters are also used ("SSR" stands for "state sequencing register"):

Octal	Char.	Use
042	"	Comment delimiter
043	#	NOT EQUAL relational operator or SSR marker
044	$	(Optional) end of file marker
050	(	Expression, parameter list, or SSR value delimiter
051	)	Expression, parameter list, or SSR value delimiter
052	*	AND operator
053	+	OR operator

054	,	List separator
055	~	NOT logical operator
056		Declaration and conditional terminator or indicator for left-justification in constants
057	/	State terminator
072		Label delimiter or value range indicator
073	;	Separator in conditionals
074	<	LESS THAN relational operator
075	=	Immediate transfer action
076	>	GREATER THAN relational operator
100	@	Set-terminal action, octal constant indicator
133	[	Dimension or subscript delimiter
135	]	Dimension or subscript delimiter
136	¢	Case-selecting expression delimiter
137	_	Delayed transfer action

In addition to the above symbols, DDL-P uses the following multiple-character special symbols:

Symbol	Use
[+]	EXCLUSIVE-OR logical operator
(+)	Arithmetic addition operator
(-)	Arithmetic subtraction or negation operator
>=	GREATER-THAN-OR-EQUAL relational operator
<=	LESS-THAN-OR-EQUAL relational operator
(=)	EQUALS relational operator
->	Goto or set-next-state action
=>	Set-next-state action (save state for RETURN)
<-	Delayed transfer action
CASE	Conditional operator
DO	Conditional separator
ENDCASE	Conditional terminator
IF	Conditional operator
THEN	Conditional separator
ELSE	Conditional separator
ENDIF	Conditional terminator
END	Declaration terminator
CON	Concatenation operator
RED	Reduction operator
EXT	Replication operator
HEAD	Head substring operator
TAIL	Tail substring operator
TINE	Time specification action
INPUT	Input action

OUTPUT	Output action
LEVEL	Return to higher level state machine
RETURN	Set-next-state action (by RETURN)
REGISTER	Register declaration header
MEMORY	Memory declaration header
TERMINAL	Terminal declaration header
OPERATION	Operation section header
CONTROL	State machine declaration header

All printing characters not mentioned above are illegal characters and should appear only in comments.

Note that on input, the compiler maps the characters "`", "{", "|", "}", "¬", and DEL (ASCII code 177 octal) into "@", "[", "\", "]", "¢" and "_", respectively. For this reason, avoid the use of "`", "{", etc.

2.2 IDENTIFIERS

```
letter-or-digit  ::=  letter | digit

identifier  ::= letter { || letter-or-digit }***
```

Identifiers in DDL-P are made up of letters and digits. Upper and lower case letters are equivalent.

Examples

```
RegisterName
rEgIsTeRnAmE                    (equivalent to RegisterName)
Identifiersl anBeVeryLong
mln2
```

Note that in the last example, "2" would be interpreted as a subscript if "m1n" were previously declared. This is discussed in Section 5.1.1.

Identifiers must start with a letter and may be of any length up to 132 characters. The alphabetic special symbols listed in Section 2.1 are keywords and may not be used as identifiers.

Unless otherwise noted in this manual, all identifiers *declared in DDL-P are global; that is, an identifier may only be declared once in a DDL-P description.

2.3 CONSTANTS

```
hex-digit    :   := digit | A | B | C | D | E | F

octal-digit  :   := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

quartal_digit ::= 0 | 1 | 2 | 3

bit              := 0 | 1

decimal-constant : := digit { | | digit }***

constant ::= decimal-constant
        | decimal-constant || B { || . } || bit
                                  { || bit }***
        | decimal-constant || Q { || . }
                                  | | quartal_digit
                                { || quartal_digit }***
        | decimal-constant | | 8 { | | . }
                                  || octal-digit
                                { || octal-digit }***
        | decimal-constant || D || decimal-constant
        | decimal-constant || H { || . }
                                  || hex-digit
                                { || hex-digit }***
```

Every constant in DDL-P has two attributes, its value and its length in bits. Generally the length of a constant is given explicitly. The general format for a constant follows:

Length (in decimal) of constant in bits,
followed by base designator B base 2
Q base 4
@ base 8
D base 10
H base 16 ,
optionally followed by '.' denoting left-justification,
followed by value (before left-justification)
in appropriate base.

The length of a constant must be in the range 0 < length <= 256.

If a constant is left-justified, then it is truncated on the right or padded on the right with zeroes to make it the proper length. Decimal constants may not be left-justified.

Note that leading zeroes in the left-most digit of a left-justified constant ARE NOT truncated; hence left-justification does not imply that the most significant bit is set to one.

If a constant is not left-justified, then it is truncated on the left or extended on the left with zeroes to make it the proper length.

Examples

Constant	Binary representation
6D22	010110
1B1	1
8B101	00000101
8B.101	10100000

2B101	01
2B.101	10
1682321	0000000010111001
1181367	01011110111
6H3C	111100
10H.74	0111010000

A decimal constant may be written without the length specification and base designator 'D', in which case a default length of 16 is assumed. The value specified must then be in the range 0 <= value <= 65535.

Examples

Constant	Binary representation
1	0000000000000001
10	0000000000001010
100	0000000001100100
4095	0000111111111111

2.4 COMMENTS

A comment (any text not containing a '"') is contained on one line and enclosed in double quotes '"':

" This is a valid comment . "

The second double quote may be omitted, in which case the entire line following the first '"' is treated as a comment:

" This is also a valid comment .

Comments are ignored by DDL-P and may be inserted freely for documentation anywhere blanks are permitted (see next section>.

2.5 DDL-P DESCRIPTION FORMAT

A DDL-P description is free-format. Blanks may be inserted at will to improve readability, except that blanks must not be embedded in constants, identifiers, or the multiple-character special symbols listed in Section 2.1. Comments may also appear anywhere blanks are allowed.

DDL-P expands the non-printing character HT (tab, ASCII code 011 octal) to blanks; tabs may appear anywhere blanks are permitted, All other non-printing characters (except DEL, noted in Section 2.1) are ignored.

All lines should be no longer than 132 characters (AFTER tabs are expanded). DDL-P truncates longer lines. The end of a line may occur anywhere blanks are allowed.

Chapter 3

DDL-P DESCRIPTION STRUCTURE

```
ddl_description ::= declaration {operation_decl}
                                  control-decl {$}

declaration  ::= register-decl {memory_decl}
                               {terminal_decl}
               | memory-decl {terminal_decl}
               | terminal-decl
```

A DDL-P description in general contains the following sections in the order given:

Chapter

4 REGISTER declarations \
4 MEMORY declarations) at least one of these
6 TERMINAL declarations / must be present
7 OPERATION section - optional
8 CONTROL section

The REGISTER and MEMORY sections contain declarations of synchronous and asynchronous storage elements, respectively. The terminal section contains declarations of combinational networks. The OPERATION section defines data transfers which may occur, along with optional timing infor-

mation. The CONTROL section contains the finite state machine, which controls the use of the physical facilities previously defined. The dollar sign at the end of the description is optional.

Each of these sections is presented in turn in the following chapters, with a discussion of Boolean expressions in chapter 5.

Chapter 4

REGISTER AND MEMORY DECLARATIONS

DDL-P allows the declaration of synchronous and asynchronous storage elements in REGISTER and MEMORY declarations, respectively.

4.1 REGISTER DECLARATIONS

```
register_decl ::= REGISTER register-spec
                  { , register-spec }*** .

register-spec ::= {#} identifier
                  { [ {constant:} constant] }
              | identifier [ {constant:} constant ,
                             {constant:} constant ]
```

In DDL-P, registers are storage elements which may be written either synchronously or asynchronously. The timing of synchronous vs. asynchronous stores is covered in chapters 7 and 8.

All registers are declared in a list following the keyword 'REGISTER' and terminated by period '.'. A register may be a Single flip-flop? or it may be a one- or two-dimensional array of flip-flops.

Example

```
REGISTER V,C,N,Z,R[0:7,15:0],
         X[10:1,5:10],Y[200:100].
```

In the above example? VF CF NF and Z are each declared to be unsubscripted single-bit registers. Register R is a two-dimensional array of bits logically organized as eight 16-bit words. The bits in each word are labeled 15,14,13,...,1,0. Bit 15 is the most significant bit (MSB). The 16-bit words of R are labeled 0 through 7.

The declarations of X and Y in the example illustrate two points:

1. The subscript range need not begin or end with 0 or 1.

2. The subscript range may be either ascending (5:10) or descending (200: 100). Note that in register XF the MSB of each word is bit 5, while in register Y, the MSB is bit 200.

The first number and colon ':' may be omitted from a subscript range? in which case a bound of one is assumed.

Example

```
"same as 'ARRAY[1:10,1:20]'"
REGISTER ARRAY[ 10,201.
```

4.1.1 State Sesuencins Resisters

A state sequencing register may be used to encode the states of the finite state machine defined in the CONTROL section. A state sequencing register is declared by immediately preceding the identifier by '#' in the REGISTER declarations. It may not have two dimensions.

Example

```
REGISTER A,B,#SSR[3:0],C.
```

Up to seven state sequencing registers may be declared. The use of state sequencing registers is discussed in chapter 8.

4.2 MEMORY DECLARATIONS

```
memory-decl  ::= MEMORY  memory-spec
                                { , memory-spec }*** .

memory-spec  ::= identifier  {[  (constant:}  constant  ]}
                 | identifier  [  {constant:}  constant  ,
                                  {constant:}  constant  ]
```

Memories in DDL-P are storage elements which may be written asynchronously only. All memories are declared in a list following the keyword 'MEMORY' and terminated by period '.'.

The syntax of the list is identical to that for the register list? with the following exception: State sequencing registers may not be declared as memories; hence, the symbol '#' must not appear in the MEMORY declarations.

Example

```
MEMORY  PM[0:1023,15:0], C[0:511,7:0],
        X, Y, S[2048].
```

4.3 MEMORY SIZE LIMITATIONS

The current implementation of DDL-P will support rather complex simulations? but only with modest memory sizes. The

designer should restrict the total register? memory? and terminal space declared to, say, 50000 bits. (Terminals are discussed in Chapter 6.)

Chapter 5

BOOLEAN EXPRESSIONS AND OPERATORS

In DDL-P, a Boolean expression is a string of one or more bits formed by zero or more operations on registers? memories? terminals? and/or constants. (Terminals are discussed in the next chapter.)

First, the syntax for specifying the operands is presented. Then the operators will be discussed.

5.1 OPERANDS IN BOOLEAN EXPRESSIONS

```
term ::= reference
     | INPUT(constant,identifier_ref
                {,identifier_ref}***)
     | CASE boolean-exp DO boolean-exp
                        DO boolean-exp
                      { DO boolean-exp }*** ENDCASE
     | ¢boolean_exp¢ boolean-exp ; boolean-exp
                                 {; boolean_exp}*** .
     | IF boolean-exp THEN boolean-exp
                      ELSE boolean-exp ENDIF
     | constant
     | ( boolean-exp )

reference ::= identifier-ref | terminal-ref
```

An operand to which an operator is applied in a Boolean expression may be an identifier reference? a terminal reference? an INPUT function? a conditional expression? a Boolean expression enclosed in parentheses, or a constant.

5.1.1 Identifier References

```
field ::= boolean-exp : boolean-exp

identifier-ref ::= identifier
                 | identifier || decimal-constant
                 | identifier [ boolean-exp ]
                 | identifier || decimal-constant
                                              [ boolean-exp ]
                 | identifier [ boolean-exp I [ boolean-exp ]
                 | identifier [ boolean-exp F boolean-exp ]
                 | identifier [ field ]
                 | identifier || I decimal-constant [ field ]
                 | identifier [ boolean-exp ] [ field ]
                 | identifier [ boolean-exp F field ]
```

An identifier reference is a reference to one or more contiguous bits in a facility? where a 'facility' is a register? memory? or terminal. The bits referenced may be specified by subscripts (Boolean expressions enclosed in brackets) following the facility identifier. The allowed kinds of subscripting operations are illustrated by example

below. The examples assume the following declaration:

```
MEMORY ZERO, ONE[ 16:1 ], TWO[ 16,0:31 ].
```

Examples of valid references:

ZERO

Reference to the bit named 'ZERO'. This is the only allowable type of reference to ZERO; ZERO may not be subscripted.

ONE[7]

Reference to the bit labeled 7 in the facility ONE. Note that 7 is in the allowed range 16:1; ONE[20] would be an invalid reference? by contrast.

ONE[10:6]

Reference to the string of five bits ONE[10] through ONE[6] inclusive. The order of the expresssions in the field (increasing or decreasing) must be the same as in the declaration; e.g., ONE[6:10] is an invalid reference.

ONE

Reference to the entire facility ONE; equivalent to ONE[16:1].

TWO[8,17]

Reference to the bit labeled 17 in the word labeled 8 in the facility TWO.

TWO[8][171

Equivalent to TWO[8,17].

TWO[4,16:23]

Reference to the string of eight bits TWO[4,16] through TWO[4,23] inclusive.

TWO[4][16:23]

Equivalent to TWO[4,16:23].

TWO[15]

Reference to the entire word labeled 15 in facility TWO; equivalent to TWO[15,0:31].

The above examples show all valid kinds of subscripting operations. In particular, note that in a Boolean expression, the identifer for a two-dimensional facility must be followed by at least one subscript, and this first subscript must not be a field.

In the above examples, all subscripts were constants. In general, however, a subscript may be any valid Boolean expression.

When the first subscript following an identifier is a constant, then in some cases the subscript, expressed in decimal, may be concatenated to the identifier without being enclosed in brackets. For example, 'ID[n]' may be written as 'IDn', where 'n' denotes a decimal constant. This feature is designed to reduce the need for brackets in complicated Boolean expressions. Its use is subject to two restrictions:

1. The identifier name 'ID' must not end with a decimal digit. Hence, 'EX3[2]' cannot be written as 'EX32'.

2. The identifier 'IDn' must not have been declared. That is, 'EXAMPLE5' does NOT mean 'EXAMPLE[5]' if 'EXAMPLE5' is itself a declared identifier.

Examples (assuming restrictions satisfied)

Reference	Equivalent to
ONE8	ONE[8]
TWO11[31	TWO[11,31]
TWO16[A:B]	TWO[16,A:B]

DDL-P may apply subscript concatenation in unexpected places. For example, an identifier 'L3' cannot be declared if the identifier 'L' was previously declared. A good rule is to not declare identifiers of the form 'IDn' ('n'=decimal constant, 'ID' ends in letter) if 'ID' will also be declared.

5.1.2 Terminal References

```
terminal-ref ::= identifier ( boolean-exp
                              {, boolean_exp}*** )
```

Most facility references are of the forms described in the above section. The exception is the special case of a terminal reference with an actual-parameter list. The actual parameters supplied in such a terminal reference are inputs to the combinational network represented by the terminal. These parameters may be arbitrary Boolean expressions. (Recall that an unsubscripted two-dimensional facility name is NOT a valid Boolean expression? however.>

Examples

```
SUM(X[11:16], 16D2)
PRODUCT(7,X)
```

Note that a terminal reference with a parameter list may not be subscripted. Terminals are discussed in Chapter 6.

5.1.3 INPUT Function

The INPUT function is an action discussed in Chapter 7. In short, it allows for the setting of facilities by the user from the teletype at simulation-time. When an INPUT action appears as a function in a Boolean expression? then the function receives as its value the last number entered by the user.

For example, if the user inputs the value 16D2 for INCREMENT in the expression R(+)INPUT(1,INCREMENT), then the expression is equivalent to R(+)2.

5.1.4 Conditional Expressions

In a conditional expression, a selector expression appears along with at least two alternative expressions. According to the value of the selector expression, one of the alternative expressions is evaluated and used as the value of the conditional expression. A conditional expression

```
CASE select DO expr 1
            DO expr 2
               . . .
            DO expr n-l
            DO expr n ENDCASE
```

is evaluated by

```
     if select=1 then expression value = expr 1
else if select=2 then expression value = expr 2
else    . . .
else if select=n-1 then expression value = expr n-l
else expression value = expr n.
```

Note that expression n above is chosen for select=0 and for select>=n. Conditional expressions may be nested.

Two other forms of the conditional expression are allowed. The above conditional expression may alternately be written

```
¢ select ¢ expr 1 ;
           expr 2 ;
            . . .
           expr n-l ;
           expr n .
```

(with period terminating the expression). This notation has the advantage of being compact. A conditional expression with just two cases may be written

```
IF select THEN expr 1
          ELSE expr 2 ENDIF .
```

5.1.5 Other Boolean Expression Operands

An arbitrarily complex Boolean expression enclosed in parentheses may itself be an operand in another Boolean expression. The enclosing parentheses may be omitted, in which case the order in which operators are applied is determined by operator precedence.

Constants may also appear as operands in Boolean expressions.

5.2 OPERATOR PRECEDENCE

```
boolean-exp  ::= minterm { + minterm }***

minterm  ::= product { [+] product }***

product  ::= complement { * complement }***

complement  ::= {-} reduction { CON reduction }***

reduction  ::=  adjustment
              | + RED adjustment
              | * RED adjustment
              |[+] RED adjustment
              | (+) RED adjustment

adjustment  ::=  relation
               | adjustment EXT  arithmetic-exp
               | adjustment TAIL arithmetic-exp
               | adjustment HEAD arithmetic-exp

relation  ::= arithmetic-exp
            | arithmetic-exp (=) arithmetic-exp
            | arithmetic-exp #  arithmetic-exp
            | arithmetic-exp <  arithmetic-exp
            | arithmetic-exp >  arithmetic-exp
            | arithmetic-exp >= arithmetic-exp
            | arithmetic-exp <= arithmetic-exp

arithmetic-exp  ::= { (-) } term
                  I arithmetic-exp (+) term
                  | arithmetic-exp (-) term
```

The syntax of Boolean expressions defines a precedence of operators. Operators with higher precedence are applied before operators with lower precedence unless a different order is specified with parentheses. Operators of the same precedence in an expression are applied left to right; e.g.,

```
A EXT B TAIL C HEAD D
```

is equivalent to

```
((A EXT B) TAIL C) HEAD D
```

The operators are listed below in order of precedence, highest to lowest. All operators on the same line have the same precedence. The reduction (RED) operators and '-' are unary operators. The '(-)' operator may be either unary or binary. The remaining operators are binary.

```
(+)       (-)
(=)       #          <          >          <=         >=
EXT       TAIL       HEAD
+ RED     * RED      [+] RED    (+) RED
CON

*
[+]
+
```

5.3 OPERATORS

The operators are discussed in order of decreasing precedence, except for the reduction operators, which are discussed last. Recall that a Boolean expression has two attributes, its length in bits and its value. Hence, for each operator, the length of the result, as well as its value, must be defined. A warning on the detection of negative results from subtraction appears at the end of this section.

In the examples below, the operators are used with constant operands. However, operands can be quite general, as discussed above in OPERANDS IN BOOLEAN EXPRESSIONS.

5.3.1 Arithmetic Operators

The arithmetic addition operator is "(+)". The two operands in an addition are considered to be non-negative binary numbers generating a non-negative sum. If one operand is shorter than the other, the shorter operand is extended on the left with zeroes before the addition. The length of the result is the length of the longer operand plus one, where the extra bit on the left is the carry out.

Examples

```
1B1 (+) 4B1011          = 5B01100
4B1111 (+) 4B1111       = 5B11110
1B1 (+) 1B0             = 2B01
```

The arithmetic subtraction operator is "(-)". As with addition, the two operands in a subtraction are considered to be non-negative binary numbers, the operands need not be the same length, and the length of the result is the length of the longer operand plus one. However, the result of a subtraction is a two's complement signed binary number, with a one in the carry out denoting a negative result.

The "(-)" operator may also be used for unary two's complement negation, in which case the result length is the same as that of the operand.

Examples

```
1B1 (-) 4B1011        = 5B10110
4B1111 (-) 4B1111     = 5B00000
1B1 (-) 1B0           = 2B01
1B0 (-) 3B110         = 4B1010
(-) 3B110             = 3B010
(-) 3B001             = 3B111
```

5.3.2 Relational Operators

In relational operations, the two operands are considered to be non-negative binary numbers. The result of the operations is 1B1 if the indicated relation is true and 1B0 otherwise. The two operands need not be the same length.

The relations denoted by the operators are as follow:

```
(=)   equal to
#     not equal to
<     less than
>     greater than
<=    less than or equal to
>=    greater than or equal to
```

Examples

```
2B10 > 16D1        = 1B1
10D1 # 1B1         = 1B0
8D2 (=) 8D3        = 1B0
1B1 >= 2@3         = 1B0
```

5.3.3 Substrins Operators

The operators "EXT", "TAIL", and "HEAD" access parts of the first operand or replicate the first operand as indicated by the count given as the second operand. The operation "ARG EXT n" concatenates ARG with itself n-l times. The operations "ARG HEAD n" and "ARG TAIL n" yield the most significant (leftmost) n bits of ARG and the least significant n bits of ARG, respectively. A run-time error occurs if the length of an EXT operation result exceeds 256 bits, or if the number of bits specified in a HEAD or TAIL operation is greater than the length of the first operand.

Examples

```
3B101 EXT 3          = 9B101101101
8B11010110 HEAD 4    = 4B1101
8811010110 TAIL 2    = 2B10
```

5.3.4 Concatenation

The "CON" operator concatenates its two operands. The left operand becomes the most significant (leftmost) portion of the result. A run-time error occurs if the result is longer than 256 bits.

Examples

```
4B1101 CON 6B1       = 10B1101000001
4B1 CON 6B.1         = 10B0001100000
```

5.3.5 One's Complement

The one's complement operator "-" complements each bit of the operand. The length of the result is the same as that of the operand.

Examples

```
- 1B1          = 1B0
- 6B110101     = 6B001010
- 10@1473      = 10B0011000100
```

5.3.6 Binary Logical operators

A binary logical operator performs the indicated bit-wise logical function on its two operands. If the two operands are of differing lengths, then a run-time warning is issued and the shorter operand is extended with zeroes before the operation. The length of the result is the same as that of the longer operand.

The functions denoted by the operators are as follow:

```
*      Logical AND (highest precedence)
[+]    Exclusive OR
+      Inclusive OR (lowest precedence)
```

Examples

```
5B10110  *  5B00101    = 5B00100
5B10110 [+] 5B00101    = 5B10011
5B10110  +  5B00101    = 5B10111
5B10110 * 7B1111111    = 7B0010110 with warning
```

5.3.7 Reduction Operators

Using the bits of its single operand as arguments, a reduction operator performs the indicated operation n-l times, where n is the length of the operand. For example,

```
[+] RED ARG[1:4]
```

is equivalent to

```
ARG1 [+] ARG2 [+] ARG3 [+] ARG4
```

where ARG is singly dimensioned. The result is a single bit, except in the case of addition, where the result has length 16.

Examples

```
  + RED 5B00010    = 1B1
  * RED 5B11111    = 1B1
[+] RED 5B00101    = 1B0
(+) RED 5B11101    = 16B0000000000000100
```

5.3.8 Warning on Use of Subtraction

As noted earlier, the result of a subtraction is a two's complement signed binary number. However, the other arithmetic and relational operators always consider their operands to be unsigned non-negative numbers. Hence the expression

```
A (-) B < 0
```

does not perform the desired function of detecting a negative result. In fact, the expression always evaluates as 1B0, since no unsigned number is less than zero. A simple expression performing the desired function in this case is

```
A (-) B HEAD 1 ,
```

or even simpler,

```
A < B .
```

In general, care is required when using arithmetic or relational operators with negative numbers.

Chapter 6

TERMINAL DECLARATIONS

```
terminal-decl  ::= TERMINAL  terminal-spec
                             { , terminal-spec }*** .

terminal-spec  ::=  identifier
                             { [ {constant:} constant  1 }
                 | identifier [ {constant:} constant  ,
                                (constant:} constant  ]
                 | identifier
                        { (identifier {,identifier}*** ) }
                        { [{constant:} constant] }
                                          = boolean-exp
```

Terminal identifiers are names for the outputs of combinational networks, called 'terminals' in DDL-P. All terminals are declared in a list following the keyword 'TERMINAL' and terminated by period '.'.

It is convenient to consider a combinational network in three parts:

1. OUTPUTS

2. INPUTS - Constants, registers, memories, and other terminals

3. FUNCTION - Logical function (combinational circuitry) mapping inputs to the outputs

When a terminal is declared, the function mapping the inputs to the outputs may or may not be given. If the function is given, then the inputs themselves may be completely specified, or some inputs may be left unspecified; the unspecified inputs will be given later via a parameter list.

In general, terminals may be one-dimensional. Some terminals may also be two-dimensional. The syntax for specifying terminal dimensions is identical to that for giving the dimensions of registers or memories.

6.1 TERMINALS WITH UNSPECIFIED FUNCTIONS

In the simplest case, terminals are declared without giving the associated functions. Such terminals may be singly or doubly subscripted.

Example

```
TERMINAL  A,B,C,              "one bit each"
          D[ 101,             "equivalent to D[1:10]"
          T[5:16],S[7:0,15:4].
```

The values or functions to be associated with these terminals may be specified in the OPERATION or CONTROL section. This will be discussed in chapters 7 and 8.

6.2 TERMINALS WITH SPECIFIED FUNCTIONS AND INPUTS

The function associated with a terminal may be specified with a Boolean expression in the declaration. Terminals so defined may be singly-subscripted.

Example

```
TERMINAL SUMXY[1:16]= X(+)Y TAIL 16,
         YL17= SCORE<17,
         NEXTQ= CASE J CON K DO 1B0
                             DO 1B1
                             DO -Q
                             DO Q ENDCASE ,
. .      ONE[1:8]= 8B00000001.
```

Any terminals referenced in Boolean expressions in TERMINAL declarations must be declared prior to their appearance in the Boolean expressions.

6.3 TERMINALS WITH UNSPECIFIED INPUTS

When a terminal function is specified in the terminal declaration, some or all of the inputs may not be identified. These inputs are represented by formal parameters in

the declaration, where the formal parameters appear in a list following the terminal identifier. Whenever such a terminal is referenced, the unspecified inputs must then be supplied in an actual-parameter list. A terminal with a formal parameter list may be singly subscripted.

Example

```
TERMINAL SUM(X,Y)[1:12] = (10D0 CON X(+)Y) TAIL 12.
```

In this example, inputs X and Y are unspecified. A valid reference to SUM might then be

```
SUM(IR[21:32],R[IR[17:20]])  ,
```

where IR and R are registers. The formal parameters are assumed to be Boolean expressions, but no assumption is made regarding the lengths of the parameters.

Parameter passing is by value. The following restrictions apply:

1. Formal parameters may not be subscripted. A subscripting operation may be simulated with HEAD and TAIL, although this is less efficient than normal subscripting.

2. A formal parameter may not appear in the INPUT function.

Formal parameter names are declared locally to the terminal definition; the names may be re-declared outside the terminal definition.

Chapter 7

OPERATION SECTION

As used in this chapter, the term "operation" refers to a named sequence of actions, where actions may specify data transfers, sequencing or timing information, or other (previously defined) operations to be invoked. These named sequences of actions are defined in the operations section.

The several possible actions are presented first, followed by a discussion of operation definitions.

7.1 OPERATION ACTIONS

```
action ::= identifier-ref {CON identifier-ref)
                                        <- boolean-exp
         | identifier-ref {CON identifier-ref}
                                        = boolean-exp
         | identifier-ref @
         | INPUT ( constant , identifier-ref
                        { , identifier-ref }*** )
         | OUTPUT ( constant , reference
                        { , reference }*** )
         | identifier { ( boolean-exp
                        { , boolean-exp }*** ) }
         | CASE boolean-exp
              DO action { , action }***
             { DO action { , action }*** }*** ENDCASE
         | ¢boolean_exp¢
              action { , action }***
             { ; action { , action }*** }*** .
         | IF boolean-exp
              THEN action { , action }***
             { ELSE action { , action }*** } ENDIF
         | TIME boolean-exp
         | -> identifier
         | identifier : action
```

7.1.1 Immediate and Delayed Stores

The store actions "=" and "<-" indicate that the Boolean expression on the right hand side is to be evaluated immediately and its value assigned to or stored in the facility given on the left hand side. Two facility references may be concatenated on the left hand side, in which case the

right hand facility in the concatenation receives the rightmost bits in the Boolean expression, and the left hand facility gets the bits to the left. For example,

```
A[1:3] CON B[1:5] = 8B10101100
```

is equivalent to

```
A[1:3] = 3B101 ,    B[1:5] = 5B01100 .
```

The length in bits of the facility reference(s) on the left hand side should be the same as the length of the Boolean expression on the right hand side. If these two lengths differ, then a run-time warning is issued. If the Boolean expression is too long, then high order (leftmost) bits will be truncated as necessary. If the Boolean expression is too short, then high order bits of the destination facility will be LEFT UNCHANGED.

The evaluation of the Boolean expression on the right occurs exactly once (each time the store action is encountered); any subsequent changes to operands in the right side do not cause a re-evaluation and assignment. Consider, for example, the following sequence where A, B, and R are single-bit flip-flops and T is a terminal:

```
[ R = 1B1,
  T = R,
  A = T,
  R = 1B0,
  B = T ]
```

The assignment of 1B0 to R in the fourth line does not cause a re-evaluation of T. The value assigned to B is the same as that assigned to A, namely 1B1. Even though terminals were described in Chapter 6 as representing combinational networks, their use (as above) may suggest the existence of additional sequential circuitry, if indeed the DDL-P description corresponds closely to the hardware design.

The above example should be contrasted with the following, in which A, B, R, and T are as before, but the function associated with T is defined in the TERMINAL section:

```
MEMORY A, B, R.
TERMINAL T = R.
OPERATION EXAMPLE =
  [ R = 1B1,
    A = T,
    R = 1B0,
    B = T   ].
```

In this case, T is re-evaluated each time it is referenced in the operation. Hence, A is set to 1B1, but B is set to 1B0.

This illustrates an important difference between the specification of a terminal function in the TERMINAL section and the assignment of a function to a terminal as an action in an operation. Another important difference is that an assignment to a terminal made in an operation is not permanent. This will be discussed further in the next chapter.

7.1.1.1 Immediate Store

In an immediate store, denoted by "=", the value of the right hand side is assigned IMMEDIATELY to the facility on the left hand side. This facility may be any facility (register, memory, or terminal) EXCEPT a state sequencing register or a terminal for which a function was already assigned in the TERMINAL section.

Examples

```
T[7:4] = 4D9,
MEM[R[IR[6:8]]] = R[IR[9:11]]
```

7.1.1.2 Delayed Store

In a delayed store, denoted by "<-", the right hand side is evaluated immediately, but the resulting value is not assigned to the facility until the end of the current "state". Subsequent references to the left hand facility made before the end of the current state will access the old, not the new, value of the facility.

This timing will be discussed further in the next chapter; for now it will suffice to understand that the sequence of actions

```
A <- B,  B <- A
```

will effect a swap of facilities A and B. By contrast, the sequence

```
A = B, B = A
```

will not effect a swap, but is equivalent to "A=B" alone.

The destination in a delayed store must be a register.

Examples

```
R[NUM] <- MBR,
ACC <- ALU(ACC, R[I], 2)
```

7.1.2 Set-Terminal Action

DDL-P provides a shorthand notation for an immediate store where the left hand side is a terminal of length one bit and the right hand side is 1B1. The action

```
T = 1B1
```

may be written as

```
T @ .
```

The advantage of this notation, aside from its brevity, is that it may also appear in the CONTROL section (Chapter 8), whereas the store actions "=" and "<-" may not.

7.1.3 INPUT Action

The INPUT action allows the user to enter facility values from the teletype at simulation time. The first item in the list enclosed in parentheses is a device number, which is ignored for the INPUT action in the current implementation of DDL-P. The identifier references following the device number indicate the facilities to be input. Any valid identifier reference may appear, and the input values are stored in the facilities immediately.

Examples

```
INPUT(1,ACC),
INPUT(1,R[1],T[L:R])
```

The values entered by the user should have the same lengths in bits as the corresponding facilities being set. If the lengths do not agree, then the actions taken are the same as those for a store in which the lengths do not agree (Section 7.1.1).

The user will be prompted for new values each time the INPUT action is encountered, even if new values were previously supplied during the current operation. Each value input will be shown in the listing file.

7.1.4 OUTPUT Action

The OUTPUT action immediately lists the values of the indicated facilities either at the teletype (if device number is zero or one) or in the listing file (if device number is greater than one). The device number is the first item in the OUTPUT list; the remaining references indicate the facilities to be displayed. Any valid facility reference may appear in this list.

Examples

```
OUTPUT(1, NUM, R[NUM]),              "To teletype"
OUTPUT(2, MAR, MEM[MAR], MBR)        "To listing "
```

7.1.5 Operation Reference

An action may be the name of a previously defined operation, in which case the sequence of actions appearing in that definition is executed. If the operation was defined with formal parameters (Section 7.21, then a corresponding list of actual parameters (arbitrary Boolean expressions) must be supplied enclosed in parentheses following the operation name.

An operation may invoke itself (recursion), but forward references are not allowed.

Examples

```
INITIALIZE,
GETREG(NUM, MEM[ADDR])
```

7.1.6 Conditional Action

The interpretation of conditional actions is similar to that of conditional expressions. A selector expression appears along with one or more lists of actions. According to the value of the selector expression, one of the lists will be executed. The action

```
CASE selector DO list of actions 1
              DO list of actions 2
                    . . .
              DO list of actions n-l
              DO list of actions n ENDCASE
```

. .

is executed as

```
     if selector=1 then list of actions 1
else if selector=2 then list of actions 2
else      . . .
else if selector=n-1 then list of actions n-l
else list of actions n .
```

The alternative forms for the conditional syntax presented in Section 5.1.4 may also be used.

A special case occurs when only one list is specified. In this case, the list is executed only if the selector has value one; otherwise, the entire action is skipped. A conditional action with just one list may be written

IF selector THEN list of actions ENDIF

Conditional actions may be nested.

Examples

```
IF I>8 THEN ACC<-16D0
       ELSE R[I]<-16D0 ENDIF ,
IF OVFL THEN PC <- EMA ENDIF
CASE NA DO K=B
        DO IF FF THEN X=B ELSE Y=-B ENDIF ENDCASE
```

The last example may also be written using the alternate form in the following way:

```
¢ NA ¢ K=B ; ¢ FF ¢ X=B ; Y=-B . .
```

7.1.7 Labels and Gotos

Any action may be preceded by one or more labels (identifiers), each followed by a colon. The course of execution may then be altered with a goto "->". A goto must point to a label in the same operation definition. Forward referencing of labels is permitted.

Labels and gotos are useful for specifying loops within operations.

Example

```
"Initialize M[1:100,15:0] to all ones"

   ADDR = 8D1,
L: M[ADDR] = 16HFFFF,
   ADDR = ADDR(+)1 TAIL 8,
   IF ADDR <= 100 THEN ->L ENDIF
```

7.1.8 TIME Specification

The designer may specify that a certain action or operation requires T units of time, where T is an arbitrary Boolean expression, and a "unit of time" is a convenient unit selected by the designer. This timing is specified by the action

TIME T .

The default length of time assumed is one. All actions in an operation are considered as occurring in parallel, so the total length of time required for an operation will be

maximum { times required for each action
executed in the operation }

A TIME of zero should not be given; such an action will be ignored.

7.2 OPERATION DEFINITIONS

```
operation-decl ::= OPERATION operation-def
                          {, operation_def}*** .

operation-def ::= identifier
                    { (identifier {,identifier}*** ) }
                    = [ action {, action}*** ]
```

The OPERATION section consists of a list of operation definitions. The list is preceded by "OPERATION" and followed by period. A simple operation definition gives the operation name followed by a list of actions enclosed in brackets. The actions in an operation must form a "compatible set of actions." This term is defined in Chapter 8; basically, it denotes a group of actions in which illegal simultaneous stores to a register do not occur.

The operation name may optionally be followed by a formal parameter list enclosed in parentheses. When such an operation is referenced later, a corresponding list of actual parameters must be supplied. Parameter passing is by value. The following restrictions apply to actual and formal parameters:

1. Actual parameters must be valid Boolean expressions. In particular, an unsubscripted two-dimensional facility name may not be an actual parameter.

2. Formal parameters may not be subscripted. A subscripting operation may be simulated with HEAD and TAIL, although this is less efficient than normal subscripting.

3. No assignment of a value may be made to a formal parameter.

4. Formal parameters may not appear in INPUT or OUTPUT lists.

Formal parameter names are declared locally'to the operation definition; the names may be re-declared outside the operation definition.

Example

```
OPERATION
   READ(ADDR) = [MBR=M[ADDR],
                 M[ADDR]=8D0,
                 TIME 41,
   "ALU is assumed to be a terminal"
   ARITH(FCT) = [ACC<-ALU(FCT)],
   CLEAR(N) = [ IF N>7 THEN ACC<-8D0
                       ELSE REG[N]<-8D0 ENDIF ],
   TTYINP = [INPUT(1,ACC)],
```

```
INIT = [    N=8D0,
        L:  REG[N]=8D0,
            N=N(+)1 TAIL 8,
             IF N<=7 THEN ->L ENDIF ,
            ACC=8D0],
DOFET = [ADDR=PC, PC<-PC(+)1 TAIL 16,
         READ(ADDR)].
```

Chapter 8

CONTROL SECTION

In the preceding chapters, we have seen how to define the facilities in a system (REGISTER, MEMORY, and TERMINAL declarations) and how to specify actions which may occur within the system (OPERATION section). Using this base, we must now specify the overall behavior of the system. In DDL-P this is done by describing the system as a finite state machine in the CONTROL sections.

8.1 THE FINITE STATE MACHINE

```
level-decl ::= CONTROL state-def { state-def }***

state-def  ::= {identifier { (constant) } :}
                 { state-action {, state-action}***  } ;
```

In general, a DDL-P description may have more than one CONTROL section. These sections are called "interpretively linked machines? For now, however, consider the case where there is just one CONTROL section, or level.

A finite state machine consists of one or more states. A state is an abstract expression of the condition of the machine, where the condition may be as simple as the contents of a register, or may be far more complex. The finite state machine is in one state at any given time, and has rules for determining the outputs and the next state, given the current state and the inputs.

Hence, the CONTROL section is a list of state definitions, where each state has up to three parts:

1. Optional label (identifier> naming the state.

2. Optional constant, enclosed in parentheses, denoting the value which should be in the state sequencing register when the system is in this state. The same value may not be assigned to two different states.

3. Optional state actions for determining the next state and the values of facilities when the system is in this state.

The state actions are separated by commas. Each state definition is terminated by "/".

8.2 THE STATE ACTIONS

```
state-action ::=
        identifier { (boolean-exp {,boolean_exp}***) }
      | identifier-ref @
      | -> identifier
      | => identifier
      | RETURN
      | CASE boolean-exp
              DO  state-action {, state-action}***
            { DO state-action {, state_action}*** }***
              ENDCASE
      | ¢boolean_exp¢ state-action {,state_action}***
                  {; state-action
                  {, state-action}*** }*** .
      | IF boolean-exp
              THEN  state-action {, state-action}***
            { ELSE  state-action {, state-action}*** }
              ENDIF
      | LEVEL
```

8.2.1 Determination of Facility Values

Two state actions exist for modifying facility contents or functions. One action names an operation, defined in the OPERATION section, to be executed. The syntax and semantics of this action are identical to those for the operation reference discussed in Section 7.1.5. The number of actual parameters in the reference must match the number of formal parameters in the definition.

The second action is the set-terminal action, which was discussed in Section 7.1.2. Its syntax is

identifier-ref @ ;

it denotes that the named one-bit terminal is to be set immediately to 1B1.

Examples

```
"Operation references"
READ(ADDR[1:16]), ARITH(4B1011),
CLEAR(4), TTYINP,

"Set terminals to 1B1"
T@, BITS[100,12] @, BUS[8] @
```

8.2.2 Determination of Next State

The next state may be specified explicitly, implicitly, or by default. Only one next state may be given explicitly or implicitly, with the exception that two next states may be specified if exactly one of them was given with a subroutine call "=>".

8.2.2.1 Explicit Next State

The next state may be set explicitly by any of three actions. The action

-> STATE

specifies that the name of the next state is "STATE". A subroutine capability is also provided; the action

```
=> STATE
```

will cause the next state to be "STATE", and the system will remember where this call was made. When the third action

```
RETURN
```

is encountered, then the next state will be the return state; the return state is the one which would have been the next state if the original "=> STATE" had been missing. Subroutine calls may be nested.

These actions will be illustrated by example in Section 8.2.2.4.

8.2.2.2 Implicit Next State

The next state is specified implicitly when the state sequencing register for this CONTROL declaration is written by a delayed store "<-". This may occur in an operation invoked by a state action, Such a store is equivalent to a got0 "->" to the corresponding state.

8.2.2.3 Default Next State

If the next state is not specified explicitly or implicitly, then by default the next state is the state following the current state in the CONTROL declaration. Note that no default exists for the last state in a CONTROL declaration; in the last state, a next state must be specified implicitly or by goto "->" or "RETURN".

8.2.2.4 Example

The following example illustrates the next-state actions:

```
REGISTER #SSR[2:0].
OPERATION SETSSR(N) = [SSR<-N TAIL 3].
CONTROL P(1): ->S/
        Q(2): =>T/
        R(3): ->P, =>U/
        S(4): SETSSR(2), =>V/
        T(5): /
        U(6): SETSSR(0)/
        V   : =>X/
        W(0): RETURN/
        X(7): RETURN, =>W/.
```

State sequence for this example:

System starts in first state, P.

```
*****STATE=P, SSR=3D1, RETURN STATE STACK IS EMPTY.
```

Next state is S by goto "->".

```
*****STATE=S, SSR=3D4, RETURN STATE STACK IS EMPTY.
```

If "=>V" were missing, then "SETSSR(2)" would imply next state is Q. State Q goes on return state stack, and next state is V. State V has no SSR value specified, so SSR is left equal to the value set in state S. If state V did have an SSR value given, then it would override the value supplied in state S.

*****STATE=V, SSR=3D2, RETURN STATE STACK = Q.

A nested call. If "=>X" were missing, then next state would be W by default.

*****STATE=X, SSR=3D7, RETURN STATE STACK = Q,W.

If "=>W" were missing, then next state would be W by RETURN. State W is popped off the stack by RETURN, but then pushed back on the stack by "=>W".

*****STATE=W, SSR=3D0, RETURN STATE STACK = Q,W.

Next state is W by RETURN.

*****STATE=W, SSR=3D0, RETURN STATE STACK = Q.

Next state is Q by RETURN.

*****STATE=Q, SSR=3D2, RETURN STATE STACK IS EMPTY.

If "=>T" were missing, then next state would be R by default.

*****STATE=T, SSR=3D5, RETURN STATE STACK = R.

Next state is U by default.

*****STATE=U, SSR=3D6, RETURN STATE STACK = R.

"SETSSR(0)" implies next state is W.

*****STATE=W, SSR=3D0, RETURN STATE STACK = R.

Next state is R by RETURN.

*****STATE=R, SSR=3D3, RETURN STATE STACK IS EMPTY.

If "=>U" were missing, then next state would be P by got0 "->".

*****STATE=U, SSR=3D6, RETURN STATE STACK = P.

"SETSSR(0)" implies next state is W.

*****STATE=W, SSR=3D0, RETURN STATE STACK = P.

Next state is P by RETURN. This completes the cycle.

*****STATE=P, SSR=3D1, RETURN STATE STACK IS EMPTY.

8.2.3 Other State Actions

The conditional state action looks and behaves like the conditional action discussed in Section 7.1.6. According to the value of a selector expression, one of several lists of actions is executed. The action

```
CASE selector DO list of state actions 1
              DO list of state actions 2
                        . . .
              DO list of state actions n-l
              DO list of state actions n
              ENDCASE
```

is executed as

```
     if selector=1 then list of state actions 1
else if selector=2 then list of state actions 2
else      . . .
else if selector=n-1 then list of state actions n-l
else list of state actions n
```

In the special case where only one list of actions is specified, the list is executed only if the selector has value one; otherwise, the entire state action is skipped. Conditional state actions may be nested.

Example

```
IF SCORE<17
   THEN ->B
   ELSE IF SCORE<22 THEN ->JK
                    ELSE IF FF THEN ->D ENDIF,
                         KFF, TMT
            ENDIF
   ENDIF
```

The last state action is LEVEL. This is mentioned here for completeness only. LEVEL will be discussed in Section 8.4.

8.3 TIMING CONSIDERATIONS

8.3.1 Timing of Immediate and Delayed Stores

As the simulation of a state is carried out, store or input actions may be encountered. The timing for such actions is as follows:

1. If a store action ("=", "<-", or "@") is encountered, the right hand side is evaluated immediately. This evaluation occurs just once, as discussed in Section 7.1.1. If an INPUT action is encountered, the input value is requested immediately.

2. In the case of an immediate store ("=" or "@") or INPUT action, the new value is stored in or assigned to the specified facility immediately. In the case of a delayed store to a register ("<-"), the new value is not stored immediately, but is saved in a temporary buffer.

3. At the end of the state, the simulator checks to see if it should halt the simulation or do any I/O. This occurs BEFORE the new register values specified in delayed stores are stored. Hence, any register values accessed or output by the simulator at this time are still old values.

4. When the simulation continues, the new values given in delayed stores are stored in the registers. At the same time, any terminals which were set are cleared to zero. Hence, any assignment to

a terminal is temporary and lasts (at most) until the end of the current state.

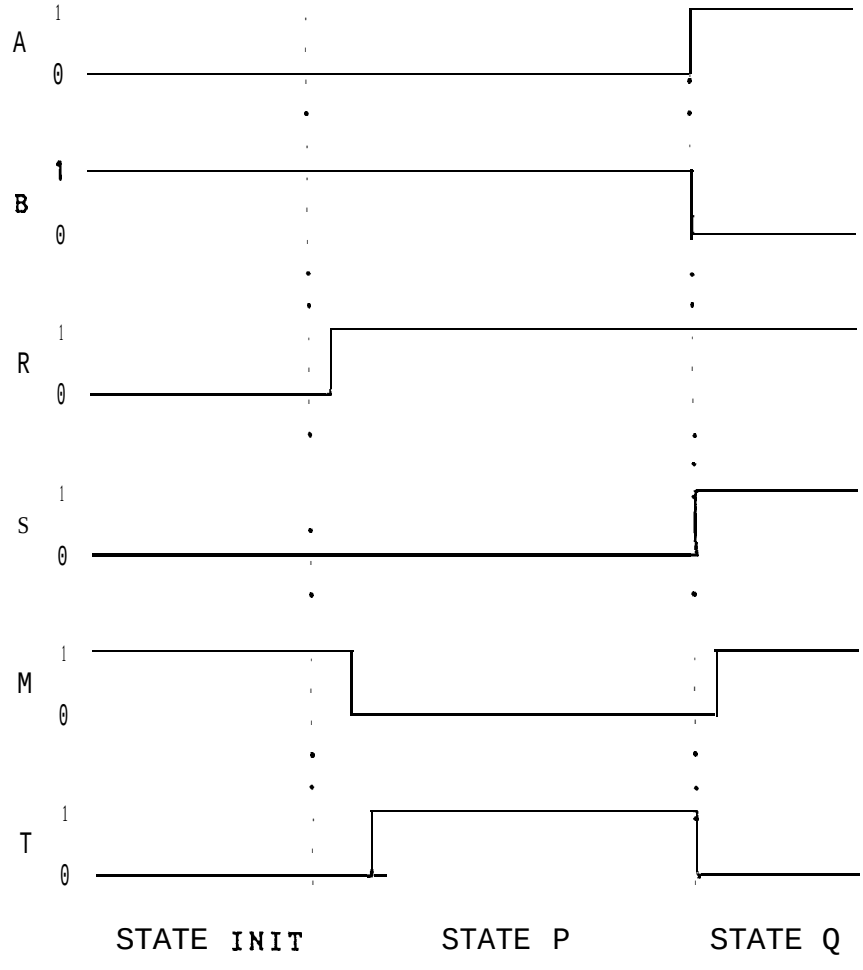
5. After the registers have been updated and the terminals cleared, the simulator begins simulation of the next state.

This timing is illustrated by the following example with waveforms:

Example

```
REGISTER A,B,R,S.
MEMORY M.
TERMINAL T.
OPERATION
  SWAP = [A<-B, B<-A],
  AIMM(X) = [A=X], BIMM(X) = [B=X],
  RIMM(X) = [R=X],
  SIMM(X) = [S=X], SDEL(X) = [S<-X],
  MIMM(X) = [M=X],
  TIMM(X) = [T=X].
CONTROL
  INIT: AIMM(1B0), BIMM(1B1), RIMM(1B0),
        SIMM(1B0), MIMM(1B1), TIMM(1B0), ->P/
  P   : SWAP, RIMM(1B1), SDEL(1B1),
        MIMM(S),T ii), ->Q/
  Q   : MIMM(1B1), ->Q/.
```

W A V E F O R M S



Note that the value assigned to M in state P is 1B0, the "old" value of S, even though the delayed store to S has already been encountered. The value assigned to S is stored at the beginning of state Q. Also, terminal T is cleared before the start of state Q. Finally, note how the swapping of values of A and B occurs.

During the simulation, once a delayed store to a register is specified, any further stores to the same bits during

the same state will cause a "SIMULTANEOUS STORE" warning message to be issued and the previously specified delayed store canceled. Such an error indicates an illegal attempt to simultaneously store multiple values into the same register. A set of actions in which such errors do not occur is called a "compatible set of actions."

8.3.2 Timing of Operations and States

All the operations in a state are considered as occurring in parallel. Furthermore, all the actions in an operation occur in parallel. Hence, the time required for a state is just the time required for the longest action invoked by the state. This time is one by default and may be increased by the TIME action. Note that the times required for the actions in a state may vary widely, but ALL the actions will be completed before the system moves to the next state, even if this in some sense implies that a large portion of the system is "idle" much of the time.

The designer may specify that the actions in a state occur sequentially, rather than in parallel, by splitting the state into a sequence of states. This idea is illustrated in the example below.

Example

```
REGISTER    A[ 101, B[ 10], OVERFLOW.
MEMORY      MEM[ 1024,101.
OPERATION   ADD = [OVERFLOW CON A <- A(+)B],
            STORE = [MEM[A] <- A, TIME 21.
```

In the above specification, one unit of time is implied for operation ADD while two units of time are specified for STORE. If the two operations are executed in parallel in one state,

ADD, STORE/ ,

then the time required for the operations is two units. The designer may alternately specify that ADD and STORE are executed sequentially by breaking the state into a sequence of two states, as in

ADD/ STORE/ .

This sequence will take three units of time.

Note that the choice of parallel vs. sequential execution will have a marked effect on system operation. In the example above, suppose that A and B originally contained 10D5 and 10D10, respectively. Then in the first case, operation STORE accesses the old value of A and stores 10D5 in MEM[5], while in the second case, STORE accesses the new value of A and stores 10D15 in MEM[15].

The technique of splitting states can be used in the description of a system with a multi-phase clock. Each clock cycle may be specified as a sequence of states, where each state represents one phase, as suggested below:

```
    CONTROL
    "     PHASE 1    PHASE 2    ...      PHASE I"
STATE1:   ACTS11/    ACTS12/ . . . /     ACTS1I/
STATE2:   ACTS21/    ACTS22/ . . . /     ACTS2I/
                       . . .
STATEN:   ACTSN1/    ACTSN2/ . . . /     ACTSNI/.
```

Alternately, a designer may describe a system with a multi-phase clock as two interpretively linked machines (next section); a state in the higher level machine would correspond to one clock cycle, while a state in the lower level machine would correspond to one phase. This illustrates the freedom the designer has in choosing the level of detail and the significance of a "state" in a DDL-P description.

8.4 INTERPRETIVELY LINKED MACHINES

```
control-decl ::= level-decl { level-decl }*** .
```

In general, a DDL-P description may have several (up to seven) CONTROL sections. Each CONTROL section defines a

complete finite state machine. The several machines have a hierarchical relationship; the first machine defined is at the highest or top level, level 1. The succeeding machines are at level 2, level 3, etc.

The higher level machines are typically control machines which set terminals serving as control lines. The lower level machines test these terminals and "interpret" them, executing the indicated functions. Hence, the machines are said to be "interpretively linked machines?

During the simulation, control must pass from state machine to state machine in some orderly fashion. The LEVEL action aids in this process. When the state action

. . LEVEL

is encountered in a machine, then control will pass to the next higher level machine at the end of the current state, AFTER the new values of registers changed at this level have been stored, AFTER terminals set at this level have been cleared, and BEFORE simulation of the next state at this level begins. A LEVEL in the top level machine is ignored.

Conversely, control will pass unconditionally to the next lower level machine, if such a level exists, at the end of the current state, AFTER all actions in the current state

have been executed, BEFORE the new values of registers changed at this level have been stored, and BEFORE terminals set at this level have been cleared.

Simulation always begins in the level 1 machine. By default, the very first state to be executed at each level will be the first state in the corresponding level declaration, although the designer may specify different starting states on each level at simulation time. The simulation of the finite state machine at level i (call it machine(i)) will then proceed as follows:

1. If a higher level machine exists (if i>1), then machine(i-1) passes control to machine(i) at the end of a level i-l state.

2. The simulator executes the actions for the next state in machine(i) (where the next state was previously determined). It completes immediate stores while saving delayed-store values as discussed in Section 8.3.1. The simulator also determines the next state for machine(i).

3. At the end of this state (but BEFORE register values are stored), machine(i) passes control to ma-

chine(i+1). If there is no such machine, then the simulator checks to see if it should halt or do any I/O. (These actions are requested by the designer at simulation time; see DDL-P Command Language Manual.) Note that register values accessed by lower level machines will not reflect changes by delayed stores at level-i in the previous state, the new values not yet having been stored.

4. When control is returned to machine(i) from machine(i+1) or from simulator I/O activity, then machine(i) stores the new register values given in the previous level-i state, also clearing terminals that were set in that state.

5. If a LEVEL command was not encountered in the previous level-i state, then machine(i) proceeds with the execution of the next level-i state. Otherwise, machine(i) passes control to machine(i-1). In the latter case, machine(i) remembers what the next level-i state will be when control is returned from machine(i-1).

The system is considered as being in one state in each machine at any given time. The "total state" of the system

is then a list of states, with one state from each machine. An example of a three-level control declaration is given below, followed by a description of the simulation of state A. The simulation begins at level 1 in "total state" A:C:E.

```
CONTROL  A:  OPA,  ->B/
          B:  OPB,  ->A/
CONTROL  C:  OPC,  ->D/
          D:  OPD,  ->C, LEVEL/
CONTROL  E:  OPE,  ->F/
          F:  OPF,  LEVEL, ->E/.
```

Execution of State A

```
TOTAL STATE
A:C:E:   Start at level 1 in total state A:C:E.
A:C:E:   Execute OPA, next machine(l) state will be B.
A:C:E:   Drop to level 2.

A:C:E:   Execute OPC, next machine(2) state will be D.
A:C:E:   Drop to level 3.

A:C:E:   Execute OPE, next machine(3) state will be F.
A-:C:E:  Check for simulator I/O or halt.
A:C:E:   Store registers, clear terminals from OPE.
A:C:F:   Execute OPF, next machine(3) state will be E.
A:C:F:   Check for simulator I/O or halt.
A:C:F:   Store registers, clear terminals from OPF.
A:C:E:   Rise to level 2.

A:C:E:   Store registers, clear terminals from OPC.
A:D:E:   Execute OPD, next machine(2) state will be C.
A:D:E:   Drop to level 3.

A:D:E:   Execute OPE, next machine(3) state will be F.
A:D:E:   Check for simulator I/O or halt.
A:D:E:   Store registers, clear terminals from OPE.
A:D:F:   Execute OPF, next machine(3) state will be E.
A:D:F:   Check for simulator I/O or halt.
A:D:F:   Store registers, clear terminals from OPF.
A:D:E:   Rise to level 2.

A:D:E:   Store registers, clear terminals from OPD.
A:C:E:   Rise to level 1.
```

A:C:E: Store registers, clear terminals from OPA, completing simulation of state A. Simulation of state B will proceed in similar fashion.

State Sesuencins Registers. Each finite state machine may have its own state sequencing register (S.S.R.). The first S.S.R. declared is associated with the highest level machine, the second S.S.R. declared is associated with the level-2 machine, etc.

Restrictions. The following restrictions apply in the specification of multiple-level control declarations:

1. A finite state machine should not modify the state sequencing register for a lower level machine. For example, the machine at level 2 should not modify the S.S.R. for level 3.

2. All state gotos "->" and subroutine calls "=>" must point to states within the same finite state machine. It is not possible, for example, to use one group of states as a subroutine in two different finite state machines. The following is not permitted:

```
CONTROL A: ->C/  "<-- ERROR"
        B: ->A/
CONTROL C: ->B/. "<-- ERROR"
```

3. The LEVEL action should not appear in a state which is in a subroutine or which contains a subroutine call "=>". If this rule is violated, unpredictable simulator errors may occur.

Chapter 9

EXAMPLES

Five complete DDL-P descriptions are now presented with brief discussions.

The first machine plays Blackjack by dealer's rules, accepting cards until its score exceeds 16, and counting the first Ace as eleven unless that causes its score to exceed 21. The machine shows its status in terminals HIT, STAND, and BROKE. Card values are entered through terminal VALUE, and YCRD is a "card ready" strobe.

Example 1

```
" B L A C K J A C K    M A C H   I   N E ."
REGISTER  SCORE[5], CARDBUF[5], FF.
TERMINAL  HIT, BROKE, STAND,
   VALUE[1:5] = INPUT(1,VALUE),
   YCRD = INPUT(1,YCRD),
   YL17 = SCORE<17, YL22 = SCORE<22,
   NACE = CARDBUF#1.
OPERATION
   TPT = [CARDBUF <- 5D10],
   TMT = [CARDBUF <- 5D22],
   TVC = [CARDBUF <- VALUE],
   IHIT = [HIT=1B1],
   ISTD = [STAND=1B1], IBRK = [BROKE=1B1],
   CLS = [SCORE <- 5D0],
   ADD=[SCORE <-(SCORE(+)CARDBUF)TAIL 51,
   KFF = [FF<-1D0], JFF = [FF <- 1D1 1
CONTROL
   A:  CLS, KFF, ->B/
   B:  IHIT, TVC,
       IF YCRD THEN ->C ELSE ->B ENDIF/
   C:  IF YCRD THEN ->C ELSE ->D ENDIF/
   D:  ADD,  IF NACE+FF THEN -3F
                        ELSE ->E ENDIF/
   E:  JFF, TPT, ->D/
   F:  IF YL17 THEN ->B ELSE ->G ENDIF/
   G:  IF YL22 THEN ->K ELSE ->H ENDIF/
   H:  KFF, TMT,
       IF FF THEN ->D ELSE ->J ENDIF/
   J:  IBRK,
       IF YCRD THEN ->A ELSE ->J ENDIF/
   K:  ISTD,
       IF YCRD THEN ->A ELSE ->K ENDIF/.$
```

The second machine is identical to the first except that states F, G, H, J, and K have been combined into two states, F and JK. This example shows nested conditional state actions, and also suggests the flexibility possible in describing a system as a finite state machine.

Example 2

```
" B L A C K J A C K     M A C H I N E . "
REGISTER  SCORE[5], CARDBUF[5], FF.
TERMINAL  HIT, BROKE, STAND,
   VALUE[1:5] = INPUT(1,VALUE),
   YCRD = INPUT(1,YCRD),
   YL17 = SCORE<17, YL22 = SCORE<22,
   NACE = CARDBUF#1.
OPERATION
   TPT = [CARDBUF <- 5D10],
   TMT = [CARDBUF <- 5D22],
   TVC = [CARDBUF <- VALUE],
   IHIT = [HIT=1B1 I,
   ISTD = [STAND=1B1], IBRK = [BROKE=1B1],
   CLS = [SCORE <- 5D0],
   ADD=[SCORE <-(SCORE(+)CARDBUF)TAIL 51,
   KFF = [FF<-1D0], JFF = [FF <- 1D1 ]
CONTROL
   A:  CLS, KFF, ->B/
   B:  IHIT, TVC,  IF YCRD THEN ->C ELSE ->B ENDIF/
   C:  IF YCRD THEN ->C ELSE ->D ENDIF/
   D:  ADD,  IF NACE+FF THEN ->F ELSE ->E ENDIF/
   E:  JFF, TPT, ->D/
   F:  IF YL17 THEN ->B
               ELSE IF YL22 THEN ->JK
                            ELSE IF FF THEN ->D ENDIF,
                                 KFF, TMT
               ENDIF        ENDIF/
   JK: IF YL22 THEN ISTD ELSE IBRK ENDIF,
       IF YCRD THEN ->A ELSE ->JK ENDIF/.$
```

The third machine is functionally equivalent to the first two, but its state is encoded in a state sequencing register Q consisting of three J-K flip-flops. The characteristic functions and input equations for register Q are given in operation NS. The next state in this machine is then implied by the results of operation NS.

Note the heavy use of subscript concatenation (e.g., "Q1") in operation NS.

Example 3

```
" B L A C K J A C K    M A C H  I  N E."
REGISTER  SCORE[5], CARDBUF[5], FF, # Q[3:1].
TERMINAL  HIT, BROKE, STAND, TMP,
   J[3:1], K[3:1],
   VALUE[1:5] = INPUT(1,VALUE),
   YCRD = INPUT(1,YCRD),
   YL17 = SCORE<17, YL22 = SCORE<22,
   NACE = CARDBUF#1.
OPERATION
   NS = [TMP = YCRD,
         J1= -Q3*(FF+NACE)+-Q2*-Q3,
         K1= Q2*-Q3*-TMP + -Q2*Q3 + Q3*-YL17*YL22,
         Q1<- CASE J1 CON K1 DO 1D0  DO 1D1
                             DO -Q1  DO Q1 ENDCASE,
         J2= Q1*Q3*FF + Q1*-Q3*TMP,
         K2= Q1*Q3,
         Q2<- CASE J2 CON K2 DO 1D0  DO 1D1
                             DO -Q2  DO Q2 ENDCASE,
         J3= -Q1*Q2,
         K3= 33 + -Ql*TMP + Q2*YL17 + Q1*-Q2*FF,
         Q3<- CASE J3 CON K3 DO 1D0  DO 1D1
                             DO -Q3  DO Q3 ENDCASE],
   TPT = [CARDBUF <- 5D10],
   TMT = [CARDBUF <- 5D22],
   TVC = [CARDBUF <- VALUE],
   IHIT = [HIT=1B1],
   ISTD = [STAND=1B1], IBRK = [BROKE=1B1],
   CLS = [SCORE <- 5D0],
   ADD=[SCORE <-(SCORE(+)CARDBUF)TAIL 51,
   KFF = [FF<-1D0], JFF = [FF <- 1D1 1    .
CONTROL
   A(0):  CLS, KFF, NS/
   B(1):  IHIT, TVC, NS/
   C(3):  NS/
   D(2):  ADD, NS/
   E(6):  TPT, JFF, NS/
   FG(7): NS/
   H(5):  TMT, KFF, NS/
   JK(4): IF YL22 THEN ISTD ELSE IBRK ENDIF, NS/.$
```

The fourth example has two finite state machines. The top level machine simply sets terminals which are interpreted by the lower level machine. On receipt of a signal START, the system computes the sum of the 256 words of MEM. Note the destructive-read/restore sequence simulated by the lower level machine.

Example 4

```
" EXAMPLE OF INTERPRETIVELY LINKED MACHINE "
REGISTER   At 161, B[16], ADDR[8].
MEMORY     MEM[0:255, 161.
TERMINAL   YCLR, YINC, YADD, YREAD,
           NDONE = ADDR # 255,
           START = INPUT (1, START).
OPERATION  CLEAR = [A <- 16D0, ADDR <- 8D0],
           INC = [ADDR <- ADDR(+)8D1 TAIL 81,
           ADD = [A <- A(+)B TAIL 161,
           READ = [B <- MEM[ADDR],
                    MEM[ADDR] = 16D0],
           RESTORE = [MEM[ADDR] = Bl.
CONTROL
   P1: IF START THEN YCLR @, YREAD @, ->P2
                ELSE ->P1 ENDIF/
   P2: YADD @,
       IF NDONE THEN YINC ii), YREAD @, ->P2
                ELSE ->P1 ENDIF/
CONTROL
   Q1: IF YINC THEN INC ENDIF,
       IF YCLR THEN CLEAR ENDIF,
       IF YADD THEN ADD ENDIF,
       IF YREAD THEN ->Q2
                ELSE ->Q1, LEVEL ENDIF/
   Q2: READ, ->Q3/
   Q3: RESTORE, LEVEL, ->Q1/.$
```

The last example shows terminals and operations with parameters. The concatenation in terminals SUM and AND ensures that the results are at least 16 bits long, even for short operands. Note that the actual parameters may be of varying lengths, and the formal parameters of EXM may be used as the actual parameters of SUM and AND.

Example 5

```
MEMORY R[1:16].
TERMINAL S1[1:2]=INPUT(1,S1),
         S2[1:2]=INPUT(1,S2),
         L1[1:12]=INPUT(1,L1),
         L2[1:12]=INPUT(1,L2),
         SUM(A1,A2)[1:16]=(16D0 CON (A1(+)A2)) TAIL 16,
         AND(A1,B2)[1:16]=(16D0 CON (A1 * B2)) TAIL 16.
OPERATION EXM(X,Y)=[
        R=SUM(X,Y), OUTPUT(1,R),
        R=AND(X,Y), OUTPUT(1,R)1.
CONTROL Q:EXM(S1,S2),EXM(L1,L2),->Q/.$
```

Chapter 10

HOW TO RUN DDL-P

The procedure for using DDL-P should be roughly the same on any TOPS-20 installation; only the file names should differ. The procedure described below for Stanford LOTS is typical.

1. Prepare the DDL-P description. The file containing the description may have any valid file name with the following restrictions:

   i) The file name (excluding extension) must have no more than six characters.

   ii) The extension must have no more than three characters.

2. Type the RUN command for DDL-P. At Stanford LOTS, the command is

   ```
   @<sources.ddl>ddl
   ```

3. DDL-P will prompt for INPUT and OUTPUT file names. If a directory specification is to be supplied with a file name, then it must be in the form of a PPN in brackets following the file name, e.g.,

   ```
   DESC.DDL[4,524]
   ```

   The INPUT file is the DDL-P description. A listing is written to the OUTPUT file; also, any simulation-time disk output goes to the same file.

4. DDL-P will now ask for a third file name, "DDLINI = ". At Stanford LOTS, respond with the file name

   ```
   DDL.INI[4,1550]
   ```

5. After prompting for the above file names, DDL-P will type

   TO CONTINUE, HIT THE RETURN KEY *

   and wait for a line of input. Just press <RETURN> to get started.

6. If there are no fatal errors, DDL-P will request the radix to be used for all output. Choose base 2, 4, 8, 10, or 16.

7. The ">" prompt will be printed indicating readiness to accept simulation commands. Use of the DDL-P Simulator is described in DDL-P Command Language Manual [2].

Chapter 11

SAMPLE COMPILER OUTPUT

The listings generated by the DDL-P compiler for two examples from Chapter 9 are shown on the next two pages. SIMADDR is an internal location counter; the values of SIMADDR included in the listing are useful for pinpointing the places where simulation errors occur.

SIMADDR

```
      0    00100 " B L A C K J A C K     M A C H 1   N E . "
           00200  REGISTER  SCORE[5], CARDBUF[5], FF.
           00300  TERMINAL  HIT, BROKE, STAND,
           00400     VALUE[1:5] = INPUT(1,VALUE),
      9    00500     YCRD = INPUT(1,YCRD),
     18    00600     YL17 = SCORE<17, YL22 = SCORE<22, ,
     30    00700     NACE = CARDBUF#1.
     37    00800  OPERATION
           00900     TPT = [CARDBUF <- 5D10],
     41    01000     TMT = [CARDBUF <- 5D22],
     45    01100     TVC = [CARDBUF <- VALUE],
     50    01200     IHIT = [HIT=1B1],
     54    01300     ISTD = [STAND=1B1], IBRK = [BROKE=1B1],
     62    01400     CLS = [SCORE <- 5D0],
     66    01500     ADD=[SCORE <-(SCORE(+)CARDBUF)TAIL 51,
     76    01600     KFF = [FF<-1D0], JFF = [FF <- 1D1 ]
     84    01700  CONTROL
     86    01800     A:   CLS, KFF, ->B/
     96    01900     B:   IHIT, TVC,
    100    02000          IF YCRD THEN ->C ELSE ->B ENDIF/
    120    02100     C:   IF YCRD THEN ->C ELSE ->D ENDIF/
    140    02200     D:   ADD, IF NACE+FF THEN ->F
    151    02300                          ELSE ->E ENDIF/
    165    02400     E:   JFF, TPT, ->D/
    175    02500     F:   IF YL17 THEN ->B ELSE ->G ENDIF/
    195    02600     G:   IF YL22 THEN ->K ELSE ->H ENDIF/
    215    02700     H:   KFF, TMT,
    219    02800          IF FF THEN ->D ELSE ->J ENDIF/
    239    02900     J:   IBRK,
    241    03000          IF YCRD THEN ->A ELSE ->J ENDIF/
    261    03100     K:   ISTD,
    263    03200          IF YCRD THEN ->A ELSE ->K ENDIF/.$

END OF TRANSLATION,    0 FATAL ERROR(S).

MEMORY USE:
ZSY:       31 OUT OF     1001 SYMBOL TABLE ENTRIES
ZST:       26 OUT OF     5000 STRING POINTERS
ZI:       413 OUT OF    22001 WORDS
ZC:        91 OUT OF     4650 CHARACTERS
ZB:       107 OUT OF    71229 BITS
```

```
SIMADDR

     0    00100 " EXAMPLE OF INTERPRETIVELY LINKED MACHINE "
          00200 REGISTER  A[16], B[16], ADDR[8].
          00300 MEMORY    MEM[0:255, 161.
          00400 TERMINAL  YCLR, YINC, YADD, YREAD,
          00500           NDONE = ADDR # 255,
     6    00600           START = INPUT (1, START).
    16    00700 OPERATION CLEAR = [A <- 16D0, ADDR <- 8D0],
    23    00800           INC = [ADDR <- ADDR(+)8D1 TAIL 81,
    32    00900           ADD = [A <- A(+)B TAIL 161,
    42    01000           READ = [B <- MEM[ADDR],
    48    01100                   MEM[ADDR] = 16D0],
    54    01200           RESTORE = [MEM[ADDR] = B].
    61    01300 CONTROL
    63    01400    P1: IF START THEN YCLR @, YREAD @, ->P2
    75    01500                ELSE ->P1 ENDIF/
    89    01600     P2: YADD @,
    92    01700         IF NDONE THEN YINC @, YREAD @, ->P2
   104    01800                ELSE ->P1 ENDIF/
   118    01900 CONTROL
   121    02000    Q1: IF YINC THEN INC ENDIF,
   132    02100         IF YCLR THEN CLEAR ENDIF,
   143    02200         IF YADD THEN ADD ENDIF,
   154    02300         IF YREAD THEN ->Q2
   160    02400                ELSE ->Q1, LEVEL ENDIF/
   175    02500    Q2: READ, ->Q3/
   183    02600     Q3: RESTORE, LEVEL, ->Q1/.$

END OF TRANSLATION,    0 FATAL ERROR(S).

MEMORY USE:
ZSY:        20 OUT OF     1001 SYMBOL TABLE ENTRIES
ZST:       277 OUT OF     5000 STRING POINTERS
ZI:        322 OUT OF    22001 WORDS
ZC:         68 OUT OF     4650 CHARACTERS
ZB:       4238 OUT OF   '71229 BITS
```

Appendix A

ERROR MESSAGES

The error messages issued by the DDL-P compiler are listed below, along with their severity codes. When the compiler detects errors in the DDL-P description, DDL-P lists the appropriate error message both at the teletype and in the listing file.

DDL-P will halt compilation immediately if an error of severity ABORT appears. Such an error occurs when DDL-P runs out of memory. If FATAL errors are detected, the compilation will proceed to completion, but simulation will not be allowed. If the most severe errors are WARNINGS, then simulation will be allowed.

A brief discussion of a few of the errors follows the list.

fatal	Syntax error
warning	Illegal character
warning	Input line longer than 132 characters
fatal	Constant too large
fatal	Illegal number length spec. (zero or >256)
fatal	Decimal number may not be left-justified
fatal	Illegal char. or digit of wrong radix in no.
fatal	Digit is of improper radix

warning	"END" not expected here
fatal	"THEN" not expected here
fatal	"ELSE" not expected here
fatal	"ENDIF" not expected here
fatal	"DO" not expected here
fatal	'ENDCASE" not expected here
fatal	";" not expected here
warning	"," not expected here
fatal	Undeclared identifier
fatal	Multiply-defined identifier
fatal	Too many dimensions (just 2 allowed)
fatal	This identifier may not be subscripted
fatal	Two-dimensional array requires subscript
fatal	This identifier may only have 1 subscript
fatal	Field can't be used to denote range of words
fatal	Subscripting nested too deeply (>10 levels)
fatal	Improper field or access to non-existent bits
fatal	Too many dimensions (>2) or invalid field
fatal	Formal parameter subscripted
fatal	Predefined terminal subscripted
fatal	More than 63 arguments
fatal	Missing argument list
fatal	Wrong number of arguments
fatal	This identifier may not have arguments
fatal	This identifier not allowed in expression
fatal	Operation identifier not allowed in expr.
fatal	Output operation not allowed in expression
fatal	Need >1 case in conditional expression
fatal	Constants required in field in declaration
fatal	State sequencing register too big'
fatal	State sequencing reg. can't have 2 dimensions
fatal	Predefined terminal may not have 2 dimensions
warning	Delayed store will be changed to immediate
warning	Immediate store will be changed to delayed
fatal	More than two-part concatenation
fatal	Formal parameter may not appear in I/O list
fatal	Operation identifier not allowed in I/O list
fatal	Predefined terminal not allowed in input list
fatal	Improper label (wrong type)
fatal	Illegal use of label defined in other section
fatal	Undefined state label referenced
fatal	Undefined statement label referenced
fatal	Assignment to identifier of wrong type
fatal	Operand must be terminal (and not predefined)
fatal	No SSR specified for this I.L.M. level
fatal	Value too big to fit into SSR
fatal	Same SSR value assigned to different states
fatal	More than 7 I.L.M. levels are not allowed

```
warning    "LEVEL" in top level I.L.M. ignored
fatal      Identifier must be an operation
fatal      Identifier must be a state
fatal      Too many conditional cases (try nesting)
fatal      Conditionals nested too deeply (>10 levels)
fatal      Unexpected end of input
fatal      Unexpected end of file or program
abort      Internal error:  parse stack overflow
abort      Internal error:  symbol table overflow
abort      Internal error:  memory overflow
```

The term "predefined terminal" refers to a terminal for which a function was specified in the TERMINAL declarations (Sections 6.2-6.3). An **argument** is the same as an "actual parameter.**

The abbreviation "I.L.M." stands for "interpretively linked machine," Section 8.4, while "SSR" stands for "state sequencing register."

The message

SYNTAX ERROR

flags many kinds of errors in DDL-P descriptions. In case of such errors, the designer should refer to the DDL-P BNF to determine the proper syntax.

DDL-P allocates table space of 2**n words for a state sequencing register of width n bits. The message

STATE SEQUENCING REGISTER TOO BIG

will appear only if the register is wider than 35 bits. However, a MEMORY OVERFLOW problem is likely for state sequencing registers wider than, say, 10 to 12 bits.

The error message

MORE THAN TWO-PART CONCATENATION

refers only to concatenation in the left hand side of an immediate or delayed store ("=" or "<-"). In general, an arbitrary number of operands may be concatenated in Boolean expressions. (However, the width of the result must not exceed 256 bits.1

If the error

TOO MANY CONDITIONAL CASES (TRY NESTING)

occurs (it won't unless the number of conditional cases is well into the hundreds), then it may be corrected by nesting, as illustrated below. The two examples are equivalent, but the one on the right uses nesting.

"NO NESTING"

```
CASE C[1:4]

     DO R( 1)
     DO R( 2)
     DO R( 3)
     DO R( 4)


     DO R( 5)
     DO R( 6)
     DO R( 7)
     DO R( 8)


     DO R( 9)
     DO R(10)
     DO R(11)
     DO R(12)


     DO R(13)
     DO R(14)
     DO R(15)
     DO R( 0)
. .
ENDCASE
```

"NESTING"

```
CASE C[1:2]
     DO CASE C[3:4]
          DO R( 5)
          DO R( 6)
          DO R( 7)
          DO R( 4)
        ENDCASE
     DO CASE C[3:4]
          DO R( 9)
          DO R(10)
          DO R(11)
          DO R( 8)
        ENDCASE
     DO CASE C[3:4]
          DO R(13)
          DO R(14)
          DO R(15)
          DO R(12)
        ENDCASE
     DO CASE C[3:4]
          DO R( 1)
          DO R( 2)
          DO R( 3)
          DO R( 0)
        ENDCASE
ENDCASE
```

Appendix B

DDL-P BNF

The complete Backus-Naur Form for DDL-P is listed below. Non-terminals are written in lower-case letters and underscore; "ddl_description" is a non-terminal, e.g. All other symbols are terminals except for the following special symbols ("meta-symbols"):

: =	REPLACEMENT SYMBOL - Left-hand side may be replaced by right-hand side.
{ }	OPTIONAL STRING SYMBOL - String of symbols enclosed in braces is optional.
{ }***	REPETITION SYMBOL - String of symbols enclosed may appear zero or more times in succession.
| |	CONCATENATION SYMBOL - Symbol on left must be concatenated with symbol on right (i.e., with no intervening blanks or end-of-line).
|	OR SYMBOL - This separates several right-hand sides of productions.

In a DDL description, a comment is any string of symbols except double-quote (") enclosed in double-quotes and contained on one line, e.g., "THIS IS A COMMENT". A comment may appear anywhere a blank is permitted.

The REGISTER, MEMORY, TERMINAL, OPERATION, and CONTROL declaration sections may be terminated by "END" instead of period ".". The underscore "_" may be used as the delayed store action in place of "<-".

```
letter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
           N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
           a|b|c|d|e|f|g|h|i|j|k|l|m|
           n|o|p|q|r|s|t|u|v|w|x|y|z

digit ::= 0 1 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9

hex-digit ::= digit | A | B | C | D | E | F

octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

quartal_digit ::= 0 | 1 | 2 | 3

bit ::= 0 | 1

decimal-constant ::= digit { || digit }***

constant ::= decimal-constant
       | decimal-constant || B { || . } || bit
                                  { || bit }***
       | decimal-constant || Q { || . }
                             | | quartal_digit
                          { || quartal_digit }***
       | decimal-constant || 8 { || . }
                             || octal-digit
                          { || octal-digit }***
       | decimal-constant || D || decimal-constant
       | decimal-constant || H { || . }
                             || hex-digit
                          { || hex-digit }***
```

```
letter-or-digit ::= letter | digit

identifier ::= letter { || letter-or-digit }***

field ::= boolean-exp : boolean-exp

identifier-ref ::= identifier
         |  identifier || decimal-constant
         |  identifier [ boolean-exp ]
         |  identifier || decimal-constant
                                        [ boolean-exp ]
         |  identifier [ boolean-exp ][ boolean-exp ]
         |  identifier [ boolean-exp , boolean-exp ]
         |  identifier [ field ]
         |  identifier || decimal-constant [ field ]
         |  identifier [ boolean-exp ][ field ]
         |  identifier [ boolean-exp , field ]

terminal-ref ::= identifier ( boolean-exp
                           {, boolean_exp}*** )

reference ::= identifier-ref | terminal-ref

boolean-exp ::= minterm { + minterm }***

minterm ::= product { [+] product }***

product ::= complement { * complement }***

complement ::= {-} reduction { CON reduction }***

reduction ::= adjustment
            | + RED adjustment
            | * RED adjustment
            |[+ ] RED adjustment
            |(+) RED adjustment

adjustment ::= relation
             | adjustment EXT  arithmetic-exp
             | adjustment TAIL arithmetic-exp
             | adjustment HEAD arithmetic-exp
```

```
relation ::= arithmetic-exp
           | arithmetic-exp (=) arithmetic-exp
           | arithmetic-exp  #  arithmetic-exp
           | arithmetic-exp  <  arithmetic-exp
           | arithmetic-exp  >  arithmetic-exp
           | arithmetic-exp  >= arithmetic-exp
           | arithmetic-exp  <= arithmetic-exp

arithmetic-exp ::= { (-) } term
                 | arithmetic-exp (+) term
                 | arithmetic-exp (-) term

term ::= reference
     | INPUT(constant,identifier_ref
                  {,identifier_ref}***)
     | CASE boolean-exp DO boolean-exp
                        DO boolean-exp
                      { DO boolean-exp }*** ENDCASE
     | ¢boolean_exp¢ boolean-exp ; boolean-exp
                                 {; boolean_exp}*** .
     | IF boolean-exp THEN boolean-exp
                      ELSE boolean-exp ENDIF
     | constant
     | ( boolean-exp )

ddl_description ::= declaration {operation_decl}
                                 control-decl {$}

declaration ::= register-decl {memory_decl}
                              {terminal_decl}
              | memory-decl {terminal_decl}
              | terminal-decl

register-decl ::= REGISTER register-spec
                        { , register-spec }*** .

register-spec ::= {#} identifier
                        { [ {constant:) constant 1 }
            | identifier [ {constant:1 constant ,
                           {constant:} constant I

memory-decl ::= MEMORY memory-spec
                           { , memory-spec }*** .

memory-spec ::= identifier {[ {constant:} constant ]}
              | identifier  [ {constant:} constant ,
                            {constant:} constant 1
```

```
terminal-decl  ::= TERMINAL  terminal-spec
                        { ,  terminal-spec }*** .

terminal-spec  ::=  identifier
                        { [ {constant:} constant  1 }
              I identifier [ {constant:} constant  ,
                             {constant:} constant  1
               I identifier
                       { (identifier {,identifier}*** ) }
                       { [{constant:} constant] }
                                          =  boolean-exp

operation-decl  ::= OPERATION  operation-def
                           {, operation_def}*** .

operation-def  ::=  identifier
                       { (identifier {,identifier}*** ) }
                       = [ action {, action}***  ]

action  ::= identifier-ref  {CON  identifier_ref}
                                        <- boolean-exp
          I  identifier-ref  {CON  identifier_ref}
                                         =  boolean-exp
          I  identifier-ref  @
          I  INPUT ( constant  ,  identifier-ref
                           { , identifier-ref  }***  )
          I  OUTPUT ( constant  ,  reference
                            { , reference  }***  )
          I  identifier  { (  boolean-exp
                          { ,  boolean-exp  }***  ) }
          I  CASE  boolean-exp
                DO  action { ,  action }***
              { DO  action { ,  action }*** }*** ENDCASE
          I  ¢boolean_exp¢
                    action { ,  action }***
              { ; action { ,  action }*** I***  .
          I  IF  boolean-exp
                THEN  action { ,  action }***
              { ELSE  action { ,  action }*** } ENDIF
          I  TIME  boolean-exp
          I  -> identifier
          I  identifier  :  action

control-decl  ::= level-decl { level-decl }***  .

level-decl  ::= CONTROL  state-def  {  state-def  }***
```

```
state-def ::= {identifier { (constant) } :}
                { state-action {, state-action}*** } /

state-action ::=
       identifier { (boolean-exp {,boolean_exp}***) }
     | identif ier_ref @
     | -> identifier
     | => identifier
     | RETURN
     | CASE boolean-exp
              DO state-action {, state-action}***
            { DO state-action {, state_action}*** }***
              ENDCASE
     | ¢boolean_exp¢ state-action {,state_action}***
                  {; state-action
                  {, state-action}*** }*** .
     | IF boolean-exp
              THEN state-action {, state-action}***
            { ELSE state-action {, state_action}*** }
              ENDIF
     | LEVEL
```

REFERENCES

1. Arndt, R.L. and Dietmeyer, D.L. "DDLSIM--A Digital Design Language Simulator," Proc. National Electronics Conf ., Vol. 26, pp. 116-118, December 1970.

2. Cory, W.E., Duley, J.R., and vanCleemput, W.M. DDL-P Command Language Manual. Palo Alto: Stanford University, Computer Systems Laboratory, March 1979, 39 pp.

3. Dietmeyer, D.L. and Duley, J.R. "Register Transfer Languages and Their Translation" in Disital System Design Automation:. Lanquages, Simulation and Data Base. Woodland Hills, CA.: Computer Science Press, Inc., 1975, pp. 117-218.

4. Dietmeyer, D.L. Translation of DDL Descriptions of Digital Systems, ECE-77-13. Madison: U. of Wisconsin, Dept. of Electrical and Computer Engineering, September 1977, 46 pp.

5. Duley, J.R. DDL--A Disital System Desisn Lanquage. Madison: U. of Wisconsin, Ph.D. Thesis, 1967.

6. Duley, J.R. and Dietmeyer, D.L. "A Digital System Design Language (DDL)," IEEE Trans. Comp. Vol. c-17 (September 1968), pp. 850-861.

7. Duley, J.R. and Dietmeyer, D.L. "Translation of a DDL Digital System Specification to Boolean Equations," IEEE Trans. Comp. Vol. C-18 (April 1969), pp. 305-313.

8. N, Digital Design Language Translator--DDLTRN. Madison: U. of Wisconsin, Dept. of Electrical and Computer Engineering, 13 pp.

9. N, Disital Desisn Lanquage Simulator--DDLSIM. Madison: U. of Wisconsin, Dept. of Electrical and Computer Engineering, 36 pp.

10. Soares, L.E.R. An Implementation of Digital Desisn Lanquage. Madison: U. of Wisconsin, Dept. of Electrical and Computer Engineering, M.S. thesis, 1970.